更多硬核内容就在公众号「yes 的练级攻略」

鲈整理的算法笔记，授权公众号「yes 的练级攻略」

前言：该算法笔记是本人学习之余整理出来的，特此分享出来供大家学习，恐涉侵权，禁止商用和其他以本整理材料盈利的行为。本人发此材料没有盈利的目的，只做知识分享和资源整合，帮助更多热爱 java 的朋友。以下所有的题解均加上了原作者思路的链接，如有疑问可以点击原链接与原作者进行讨论。资源整合的标准是尽量将同类型的题目归纳在一起，该算法笔记适用于准备系统性刷题的群体，建议初学者先挑每个类中的部分题目练习，后面再做其他题目。以下所有代码均是当前的可运行版本，不能保证之后是否一定可以运行，希望大家主要学习思路。题目主要来源于 Leetcode、Lintcode 和 acwing。个人觉得只看整理的材料远远不够，建议大家找到原题自己运行和 debug 一遍。

特别鸣谢各位发高质量题解的大佬们，经过某些大佬的同意，推广一下 Sweetiee、liweiwei1419 和 zerotrac 三位大佬的主页。

Sweetiee 主页：https://leetcode-cn.com/u/sweetiee/

Sweetiee 知乎专栏：甜姨的力扣题解

https://zhuanlan.zhihu.com/c_1224355183452614656

禁止商用！欢迎关注公众号「yes 的练级攻略」

更多硬核内容就在公众号「yes 的练级攻略」

Sweetiee 公众号链接：

https://sweeetiee-1256505723.cos.ap-beijing.myqcloud.com/sweetiee_wechat.jpeg

liweiwei1419 主页：https://leetcode-cn.com/u/liweiwei1419/

zerotrac 主页：https://leetcode-cn.com/u/zerotrac2/

禁止商用
公众号：
yes的练级攻略

禁止商用！欢迎关注公众号「yes 的练级攻略」

更多硬核内容就在公众号「yes 的练级攻略」

目录

OJ 输入输出 30
- 多行同类数据（int），不知长度 30
- 第一行知道数据组数，接下去每行都是两个整数（int） 30
- 如果输入遇到 0 0 就结束 31
- 计算一系列数的和 32
- t 行，计算一系列数据的和 32
- 计算一系列数的和，没有停止符号 33
- 计算一系列数的和，没有长度，n 行无限 35
- 字符串排序（1）【第一行是数目】 35
- 字符串排序（2）【多行，未知行数，空格隔开】 36
- 字符串排序（3）【多行，未知行数，逗号隔开】 37
- 输入是已知大小的二维数组 38
- 输入的每组测试数据有多行 39
- 每行测试数据的数量在该行开头给出 40
- 测试数据组数未知且结果需要取模 40
- 判断子串是否与主串的某段相等 40
- 输入三行数据，第一行是整数，第二行和第三行要去除中间的空格 41
- 输出保留 2 位的小数 42
- Arrays.sort 42
- 二值问题的 43

一、 BFS（Breadth First Search，广度优先搜索） 44

1、 地图分析（1162） 44

方法 1：比较偏门和省事一些 45

方法 2：比较正统的方法 47

2、 克隆图（133） 48

方法 1：DFS 53

方法 2：BFS 54

3、 岛屿数量（200） 55

方法 1：BFS，广度优先遍历 56

方法 2：DFS 57

4、 岛屿的最大面积（695） 58

方法 1：BFS 58

方法 2：DFS 60

5、 水域大小（面试题 16.19） 61

方法 1：DFS（12ms） 61

方法 2：BFS（20ms） 62

方法 3：并查集（20ms） 63

6、 被围绕的区域（130） 66

方法 1：DFS 66

方法 2：BFS 67

7、 腐烂的橘子（994） 69

8、 棋子旋转（linkcode1146） 71

9、 丢手绢（lintcode1269） 72

禁止商用 公众号：yes的练级攻略

10、 接雨水 2（407）.....73
具体代码（用优先队列+bfs）.....75
11、 完全平方数（279）.....77
方法 1：贪心+广搜.....78
方法 2：数学运算（最优）.....80
12、 节点间通路（面试题 04.01）【邻接表】.....81
13、 最低成本联通所有城市（1135）【并查集】.....84
14、 冗余连接（684）【并查集】.....86
15、 冗余连接 2（685）【并查集】.....88
16、 课程表（207）.....91
方法 1：BFS（6ms）【邻接表】.....92
方法 2：DFS（4ms）.....93
17、 课程表 2（210）.....94
方法 1：BFS（7ms）【邻接表】.....95
方法 2：DFS（5ms）.....96
18、 不邻接植花（1042）【邻接表】.....98
19、 钥匙和房间（841）【图】.....99
方法 1：DFS.....100
方法 2：BFS.....101
20、 图像渲染（733）【颜色填充（面试题 08.10）】.....102
方法 1：BFS（2ms）.....103
方法 2：DFS（1ms）.....103
21、 扫雷游戏.....104
方法 1：DFS（0ms）.....106
方法 2：BFS（4ms）.....108
22、 单词接龙（127）.....109
23、 单词接龙 2（126）.....113
24、 火星词典（269）【拓扑排序】.....120
25、 平行课程（1136）【拓扑排序】.....122
二、 DFS （Depth first search 深度优先搜索）& 回溯算法.....125
1、 旅行计划（lintcode1891）.....125
2、 旅行计划 2（lintcode1306）.....126
3、 电话号码的字母组合（17）.....128
4、 组合总和（39）.....129
方法 1：含有排序的解法，时间复杂度 o(nlogn).....132
方法 2：不排序的解法.....133
5、 组合总和二（40）.....134
40 具体代码.....135
6、 组合总数三（216）.....136
7、 组合（77）.....140
77 具体代码.....142
8、 全排序（46）.....143
46 具体代码.....145
9、 无重复字符串的排列组合（面试题 08.07）.....146

禁止商用 公众号：yes的练级攻略

10、 全排序二（47）.....147
47 具体代码.....149
11、 有重复字符串的排列组合（面试题 08.08）【字符串排序（剑指 38）】.....150
12、 字符串逆字典序排序（字节 1 面）.....152
13、 第 k 个排序（60）.....158
14、 N 皇后（51）.....161
15、 N 皇后 2（52）.....166
16、 八皇后（面试题 08.12）.....168
17、 单词矩阵（面试题 17.25）.....170
18、 字母大小写全排序（784）.....174
784 具体代码.....176
19、 子集（78）.....177
20、 子集二（90）【幂集（面试题 08.04）】.....178
方法 1：dfs.....178
方法 2：bfs.....179
21、 复原 IP 地址（93）.....179
22、 判断二分图（785）.....182
方法 1：DFS.....183
方法 2：BFS.....184
23、 可能的二分法（886）【并查集】.....185
24、 数独（lintcode802）.....186
25、 解数独（37）.....189
26、 矩阵中的最长递增路径（329）【dfs+memo】【拓扑结构 BFS】.....194
27、 单词搜索（79）【DFS+回溯】【矩阵中的路径（剑指 12）】.....201
28、 单词搜索 2（212）【DFS+回溯+剪枝+字典树】.....203
29、 除法求值（399）.....207
并查集代码 1.....208
并查集代码 2（带路径压缩，更短）.....210
30、 跳跃游戏 3（1306）.....212
方法 1：DFS.....213
方法 2：BFS.....213
31、 跳跃游戏 4（1345）.....214
32、 大容量背包（lintcode 1382）.....216
33、 机器人的运动范围（剑指 13）.....217
方法 1：DFS（1ms）.....218
方法 2：BFS（7ms）.....219
34、 打印从 1 到最大的 n 位数（剑指 17）.....219
方法 1：本题方法.....220
方法 2：大数打印（DFS）.....222
大数打印出字符串的代码.....223
35、 24 点游戏（679）.....223
36、 安卓系统手势解锁（351）.....225
37、 查找集群内的[关键连接]（1192）.....228
38、 重新安排行程（332）.....231

禁止商用
公众号
yes的练级攻略

39、 破解保险箱（753）.....234
40、 汉诺塔问题（面试题 08.06）.....236
三、 双指针.....237
1、 盛最多水的容器（11）.....237
2、 接雨水（42）【动规】【双指针】【单调栈】【直方图的水量（面试题 17.21）】.....238
方法 1：动态规划.....239
方法 2：双指针.....241
方法 3：单调栈.....245
3、 两数之和（1）【两数之和（lintcode 56）】.....246
方法 1：hashmap.....247
方法 2：双指针.....247
4、 两数之和 2-输入有序数组（167）.....248
方法 1：与上一题类似，用 HashMap（2ms）.....249
方法 2：双指针（1ms）.....249
5、 和为 s 的两个数字（剑指 57）【和上一题只有返回值的区别】.....250
6、 两数之和 2（lintcode 443）【多少对整数的和大于目标值】.....250
7、 两数之和 3-数据结构设计（lintcode 607）【add 和 find】【哈希表】.....252
8、 两数之和-不同组成（lintcode 587）.....253
9、 两数之和 7（lintcode 1879）【按绝对值升序排列】【二分查找】.....254
10、 两数之和-BST 版本（lintcode 689）.....255
方法 1：BST_Iterator+双指针.....256
方法 2：暴力中序+双指针【最短】.....258
11、 两数之和 4-输入 BST（653）.....259
12、 两数和-差等于目标值（lintcode 610）.....261
13、 两数和的最接近值（lintcode 533）.....262
14、 三数之和（15）.....264
15、 最接近的三数之和（16）.....267
16、 四数之和（18）.....268
17、 四数相加 2（454）【哈希表】.....271
18、 下一个排序（31）.....272
19、 部分排序（面试题 16.16）.....273
20、 删除排序数组中的重复项（26）【删除排序数组中的重复数字（lintcode100）】274
方法 1：未优化版本双指针.....275
方法 2：优化版本双指针.....276
21、 寻找峰值（162）（lintcode 75）.....276
22、 绝对差不超过限制的三元子数组（Lintcode1529）.....278
23、 移动零（283）.....280
24、 调整数组顺序使奇数位于偶数前面（剑指 21）.....280
25、 合并两个有序数组（88）.....282
方法 1：双指针从前往后【空间复杂度为 o(n)】.....282
方法 2：双指针从后往前【空间复杂度为 o(1)】.....283
26、 最小差（面试题 16.06）.....284
27、 测试函数.....285

东北酱甲 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

四、 动态规划，偶尔有贪心.....289
1、 斐波那契数列（剑指 10-1）.....289
2、 青蛙跳台阶问题（剑指 10-2）【爬楼梯（70）】.....290
3、 三步问题（面试题 08.01）.....292
4、 变态跳台阶.....293
5、 通配符匹配（44）.....293
方法 1：二维动规（36ms）.....295
方法 2：贪心（3ms）.....297
6、 一次编辑（面试题 01.05）.....299
7、 编辑距离（72）（lintcode119）.....300
72 具体代码.....304
8、 最大子矩阵（面试题 17.24）.....305
9、 判断子序列（392）.....307
方法 1：双指针.....308
方法 2：动规【适用于需要依次检查它们是否为 T 的子序列】.....309
方法 3：内置函数.....311
10、 最长公共子序列（1143）.....311
方法 1：二维.....313
方法 2：一维.....314
11、 俄罗斯套娃信封问题（354）（lintcode 602）【贪心+二分】.....314
12、 摇动序列（376）【返回可以变成摇摆序列的最长子序列的长度】.....317
方法 1：一维动规数组.....318
方法 2：常数级动规【面试推荐】.....318
方法 3：贪心.....319
13、 买卖股票的最佳时机（121）【股票的最大利润（剑指 63）】【最多 1 笔交易】320
14、 买票股票的最佳时机 2（122）【尽可能更多次的交易】.....321
方法 1：动态规划代码.....321
方法 2：贪心算法【最优解】.....322
15、 买卖股票的最佳时机 3（123）【2 笔交易】.....322
方法 1：动态规划代码.....323
方法 2：状态机代码.....324
16、 买卖股票的最佳时机 4（188）【最多 k 笔交易】.....324
17、 最佳买卖股票时机含冷冻期（309）.....325
18、 买卖股票的最佳时机含手续费（714）.....326
19、 假期（腾讯 2020 校园招聘-后台第 4 题）.....327
20、 鸡蛋掉落（887）【lintcode 584 丢鸡蛋二】.....329
方法 1：数学法（13ms）将 N 层楼扔多少次的问题转化为扔多少次可以覆盖 N 层楼.....331
方法 2：上述方法的优化：因为次数都是 i-1，将层数省略【K 为鸡蛋，N 为层数】.....332
21、 丢鸡蛋（lintcode 254）【只有两个鸡蛋】.....332
22、 戳气球（312）【区间 DP】.....333
312 具体代码.....335
23、 移除盒子（546）.....336

禁止商用 公众号：yes的练级攻略

24、 多边形三角剖分的最低得分（1039）【区间 DP】 338
25、 奇怪的打印机（664）【刷串问题，区间 DP】 339
26、 整数拆分（343）【与下一题剪绳子类似】 341
方法 1：朴素的动规 342
方法 2：数学 345
27、 剪绳子（剑指 14 - 1）【贪心】 346
方法 1：数学推导 348
方法 2: 贪心 349
28、 剪绳子 2（剑指 14 - 2） 349
方法 1：数学推导 351
方法 2：贪心 352
29、 分组选项（linkcode1834） 352
30、 工作安排（lintcode1147） 353
31、 方案数量（Lintcode1528） 354
32、 最小调整代价（lintcode91） 356
方法 1：常规 dp 357
方法 2：利用滚动数组 358
33、 按摩师（面试题 17.16） 359
34、 打家劫舍（198） 360
方法 1：o(N)空间复杂度 360
方法 2：o(1)空间复杂度 361
35、 打家劫舍 2（213） 361
36、 打家劫舍 3（337）【二叉树】 363
37、 找零（lintcode1503）【贪心】 366
38、 不同路径（62）求路径条数 367
方法 1：二维数组，用甜姨的思路，与 64 题的类似 368
方法 2：甜姨的代码压缩空间 369
方法 3：另外一种解法，压缩空间，省略压栈的过程 371
39、 不同路径 2（63） 372
63 具体代码 373
40、 迷路的机器人（面试题 08.02）【是不同路径 2 的求路径题目（用回溯）】 374
41、 最小路径和（64） 376
方法 1：二维 dp 数组 377
方法 2：一维的 377
42、 礼物的最大价值（剑指 47） 378
43、 不相交的线（1035） 379
44、 地下城游戏（174） 381
方法 1：二维反向动态规划 383
方法 2：一维 383
方法 3：DFS+memo 384
45、 三角形最小路径和（120） 385
方法 1：dfs+memo 386
方法 2：一维 dp 387
46、 交错字符串（97） 387

禁止商用 公众号：yes的练级攻略

方法 1：二维.....388
方法 2：一维.....389
47、 单词拆分（139）.....390
方法 1：hashset+动规.....390
方法 2：动规+单词表遍历.....391
方法 3：字典树+BFS.....392
48、 单词拆分 2（140）.....394
49、 运动会（lintcode1291）【贪心】.....397
50、 课程表 3（630）【贪心】.....398
51、 翻倍（lintcode1328）【贪心】.....400
52、 跳柱子（lintcode1125）.....401
53、 跳跃游戏（55）【贪心】.....402
54、 跳跃游戏 2（45）【贪心】.....404
55、 跳跃游戏 5（1340）.....406
方法 1：动规.....407
方法 2：动规+memo.....408
56、 最小跳跃次数（LCP 09）.....409
方法 1：动规（13ms）.....410
方法 2：BFS（134ms）.....410
57、 老鼠跳跃（lintcode 1861）.....411
58、 预测赢家（486）.....413
方法 1:.....414
方法 2:.....415
59、 石头游戏（877）.....415
方法 1：数学（0ms）.....417
方法 2：动规（10ms）.....418
60、 石头游戏 2（1140）.....418
61、 石头游戏 3（1406）.....420
方法 1：一维动规.....420
方法 2：常数级动规.....421
62、 石子游戏 4（1510）.....422
63、 加油站（134）（lintcode187）【贪心】.....424
64、 最低票价（983）.....426
65、 区间合并（54）【贪心】.....429
66、 视野争夺（腾讯 2020 校园招聘-后台第 5 题）【贪心】.....433
67、 种花问题（605）【贪心】.....435
方法 1：贪心.....436
方法 2：贪心+常数优化.....436
68、 最大平均值和的分组（813）【与二分题中 410 的动规思路类似】.....437
69、 可被三整除的最大和（1262）.....438
70、 寻宝（LCP 13）.....440
LCP13 具体代码.....443
71、 比特位计数（338）.....446
方法 1：动规+最高有效位.....447

禁止商用 公众号：yes的练级攻略

方法 2：动规+最低有效位【推荐】.....448
方法 3：动规+最后设置位.....449
72、 根据身高重建队列（406）【贪心】.....449
406 具体代码.....451
73、 我能赢吗（464）【状态压缩 DP】.....451
74、 优美的排列（526）【状态压缩 DP】.....454
75、 参加考试的最大学生数（1349）【状态压缩 DP】.....455
76、 骑士拨号器（935）.....458
方法 1：动规.....459
方法 2：常规.....460
77、 最大为 N 的数字组合（902）.....461
方法 1：数位 DP（1ms）.....462
方法 2：数学（0ms）.....463
78、 2 出现的次数（面试题 17.06）【数位 dp】.....464
79、 n 个骰子的点数（剑指 60）.....466
80、 粉刷房子（256）.....468
81、 粉刷房子 2（265）.....469
82、 布尔运算（面试题 08.14）.....470
83、 堆箱子（面试题 08.13）.....472
84、 最大黑方阵（面试题 17.23）.....474
85、 476
五、 前缀.....476
1、 区域和检索-数组不可变（303）.....479
2、 二维区域和检索 - 矩阵不可变（304）.....480
3、 和为 K 的子数组（560）【前缀和+哈希表优化】.....484
4、 统计优美子数组（1248）.....485
方法 1：滑动窗口（6ms）.....486
方法 2：前缀和（3ms）.....487
5、 区间和的个数（327）.....488
方法 1：TreeMap（49ms）.....488
方法 2：归并（8ms）.....489
6、 和等于 k 的最长子数组长度（325）【前缀和+哈希表】.....491
7、 连续数组（525）【前缀和+哈希表】.....492
8、 每个元音包含偶数次的最长子字符串（1371）.....494
9、 最长公共前缀（14）【lintcode78】.....496
14 具体代码.....497
10、 最长公共前缀二（lintcode784）.....497
11、 最长公共前缀三（lintcode1159）.....498
12、 数飞机（lintcode391）.....499
算法一：前缀和.....501
算法二：扫描线.....502
13、 会议室（lintcode920）.....503
14、 会议室 2（253）（lintcode919）.....504
Lintcode 做法.....504

禁止商用
公众号：yes的练级攻略

Leetcode 做法.....505
15、 会议室 3（lintcode1897）.....506
16、 最多可以参加的会议数目（1353）【贪心+排序】.....508
17、 生存人数（面试题 16.10）.....511
18、 全零子矩阵（lintcode1330）.....513
19、 实现 Tire（前缀树）（208）.....514
208 具体代码.....519
20、 添加与搜索单词（211）【只能用前缀树，其他方法超时】.....520
21、 字典序的第 k 小数字（440）.....523
22、 字典序排数（386）.....524
方法 1：用上一题的解法进行递归.....524
23、 字母与数字（面试题 17.05）.....526
六、 背包问题.....528
1. 0-1 背包.....528
- 01 背包问题（awing 2）【与背包问题 2 一样】.....528
- 背包问题（lintcode92）.....530
- 背包问题 2（lintcode125）.....530
- 背包问题 9（lintcode 800）.....531
- 分割等和子集（416）【0-1 背包】.....533
- 最小划分（lintcode 724）【两个子集中的和的差的绝对值最小】.....537
- 目标和（494）【0-1 背包，与上一题类似】.....538
- 货物转移（lintcode1161）【背包】.....540
- 最后一块石头的重量 2（1049）.....541

2. 完全背包.....543
- 完全背包问题（acwing 3）.....543
- 背包问题 3（lintcode 440）.....545
- 背包问题 10（lintcode801）.....546
- 剩余价值背包（lintcode 971）.....546
- 零钱兑换（322）.....549
- 零钱兑换 2（518）.....551
- 组合总和四（377）（lintcode 564）【和上一题的内外循环不能相反】.....552
- 硬币（面试题 08.11）.....557

3. 多重背包.....558
- 多重背包问题 1 2 3（acwing 4 5 6）.....559
- 背包问题 7（lintcode 798）.....564
- 背包问题 8（lintcode 799）.....565
- 有限纸币找零.....567

4. 混合三种背包.....567
- 混合背包问题（acwing 7）.....567

5. 二维费用的背包问题.....570
- 二维费用的背包问题（acwing 8）.....570
- 一和零（474）.....571

6. 分组的背包问题.....573
- 分组背包问题（acwing 9）.....573

禁止商用 公众号：yes的练级攻略

7. 有依赖的背包问题.....574
● 有依赖的背包问题（acwing 10）.....574
8. 泛化物品.....577
9. 背包问题问法的变化.....577
（1） 输出方案.....577
（2） 输出字典序最小的最优方案.....579
（3） 求方案总数.....582
（4） 最优方案的总数.....584
（5） 求次优解、第 K 优解.....584
七、 HashMap.....585
HahMap 常见方法.....585
1、 有效的数独（36）.....588
方法 1：HashMap（4ms）.....589
方法 2：位运算（2ms）.....592
2、 猜数字游戏（299）.....593
方法 1：map 数组.....593
方法 2：只有一个 numbers 数组.....594
3、 珠玑妙算（面试题 16.15）.....594
方法 1：数组.....595
方法 2：hashmap.....596
4、 存在重复元素（217）.....596
方法 1：排序.....598
方法 2：哈希表.....598
5、 两个数组的交集（349）.....599
方法 1：两个 Set.....600
方法 2：内置函数.....600
方法 3：双指针.....601
6、 两个数组的交集二（350）.....601
方法 1：hashmap.....602
方法 2：双指针.....603
7、 设计推特（355）.....603
355 具体代码.....605
8、 O（1）时间插入、删除和获取随机元素（380）.....608
9、 O（1）时间插入、删除和获取随机元素 - 允许重复（381）.....610
10、 元音拼写检查器（966）.....612
11、 面试题 16.24 数对和.....614
方法 1：hashmap.....615
方法 2：双指针.....616
12、 特征提取（lintcode1135）.....616
13、 LRU 缓存机制（146）（面试题 16.25）（最近最少使用 LRU）.....617
方法 1：hashmap+双向链表.....619
方法 2：源于 LinkedHashMap 源码.....624
14、 LFU 缓存（460）【双向链表】（最不经常使用 LFU）.....624
方法 1：甜姨.....625

禁止商用 公众号：yes的练级攻略

方法 2：weiwei【包含输入输出】 630
15、 英语软件（lintcode1331） 635
16、 人数统计（lintcode1302） 636
17、 婴儿名字（面试题 17.07）【并查集】 637
18、 最短超串（面试题 17.18） 638
19、 T9 键盘（面试题 16.20） 639
八、 HashSet 641
1、 分糖果（575） 641
2、 最长连续序列（128） 641
3、 判断数独是否合法（lintcode389） 643
4、 存在重复元素 2（219） 646
5、 存在重复元素 3（220） 647
6、 扑克牌中的顺子（剑指 61） 649
方法 1：hashset 650
方法 2：排序+遍历 651
九、 各种排序算法 651
(1) 插入排序 652
对链表进行插入排序（147） 652
(2) 希尔排序 654
(3) 冒泡排序 654
(4) 快速排序 655
● 字符串排序（lintcode1153）【由一些逗号隔开的字符串，返回字典序排序】 658
● 把数组排成最小的数（剑指 45） 659
方法 1：使用内置函数 660
方法 2：快排 661
● 最小 k 个数（面试题 17.14）【剑指 40 题】（包含前 k 小和第 k 小） 662
方法 1：快排变形 o(N)【顺序按大小顺序】【随机版本的快排】 663
方法 1 完整代码如下： 664
方法 2：大根堆（前 K 小）/大根堆（前 K 大） o（NlogK）【顺序与原数组相同】 666
方法 3：二叉搜索树 667
方法 4：桶排序（2ms） 669
● 数组中第 k 个最大元素（215）【快排，堆排，桶排】 670
方法 1：快排 671
前 K 大的主函数 673
方法 2：堆排序 675
方法 3：桶排序 678
● 第 k 个数（acwing786）【快排求第 k 小的数】 679
● 颜色分类（75） 680
方法 1：快排 partition 681
方法 2：三指针【荷兰三色旗】 683
(5) 双路快排 684
(6) 三路快排 685
(7) 选择排序 686

禁止商用
公众号：yes的练级攻略

(8) 堆排序 688
优先队列实现大根堆和小根堆 691
● 前 K 个高频元素（347） 691
方法 1：小根堆 692
方法 2：二叉搜索树 693
方法 3：桶排序 694
方法 4：快排 694
● 前 K 个高频单词（692） 696
● 单词频率（面试题 16.02）【hashmap 或者字典序】 697
● 最接近原点的 K 个点（973） 698
● 连接棒材的最低费用（1167） 701
(9) 归并排序： 703
● 计算右侧小于当前元素的个数（315） 704
● 数组中的逆序对（剑指 51） 706
● 逆序对（腾讯 2020 校园招聘-后台第 3 题） 710
● 快速排序和归并排序的区别与联系 714
(10) 桶排序（基数排序）： 715
● 任务调度器（621） 717
(11) 计数排序 718
● 排序测试代码 719
十、 数组 724
1、 滑动窗口的中位数（linkcode360） 724
2、 滑动窗口的最大值（239）（lintcode362）（剑指 59-1）【单调队列】 726
Lintcode 版本 727
Leecode 版本 728
3、 队列的最大值（剑指 59-2） 729
4、 滑动窗口内数的和（lintcode 604） 731
方法 1：前缀和 731
方法 2：滑动窗口 732
5、 数据流的中位数（lintcode81）【优先队列维护最大堆和最小堆】 732
方法 1：复杂版本 733
方法 2：简单版本 734
6、 数据流的中位数（295）【剑指 41】【连续中值（面试题 17.20）】 735
7、 在排序数组中查找数组 1（剑指 53） 737
8、 合并排序的数组（面试题 10.01） 739
9、 多数元素（169）【数组中出现次数超过一半的数字（剑指 39）】【主要元素（面试题 17.10）】 740
10、 求众数 2（229） 744
11、 数组的相对排序（1122）【桶排序】 746
12、 雀魂启动（lintcode1527） 747
13、 最大点的集合（lintcode1356） 749
14、 顺时针打印矩阵（剑指 29）【与下一题只有返回值不同】 751
思考题：顺时针打印矩阵的最后一个点的坐标（假设坐标从 0 开始） 753
15、 螺旋矩阵（54）【与上一题只有返回值不同】 755

禁止商用
公众号：yes的练级攻略

16、 螺旋矩阵 2（59）【类比上面两题】.....756
17、 旋转图像（48）【旋转矩阵（面试题 01.07）】.....757
18、 跳水板（面试题 16.11）.....758
19、 序列维护（lintcode1329）.....759
20、 乘积为 B（lintcode1322）.....762
21、 除自身以外数组的乘积（238）.....764
22、 构建乘积数组（剑指 66）.....765
86、 乘积最大子数组（152）【找到一个乘积最大的子数组】.....767
方法 1：二维滚动数组.....769
方法 2：表格复用（常量级别）.....770
23、 将数组重新排序以构造最小值（lintcode379）.....771
24、 最大数（179）.....772
25、 拼接最大数（321）.....773
26、 最小区间（632）.....777
方法 1：贪心+多指针+优先队列.....778
方法 2：滑动窗口.....780
方法 3：堆.....782
27、 判断能否成为等差数列（1502）.....783
28、 柠檬水找零（860）.....785
29、 分发糖果（135）.....786
30、 分糖果 2（1103）.....791
31、 分发饼干（455）.....792
32、 和为 s 的连续正数序列（剑指 57）.....793
33、 寻找 k 值（字节面试题）.....795
34、 打乱数组（384）【洗牌算法】.....796
35、 生成数组（1420）.....797
36、 递增子序列（491）.....800
如果要输出长度最长的子序列.....801
37、 最长上升子序列（300）.....802
（1）普通的动态规划代码（12ms）.....804
（2）用到了贪心算法和二分查找.....805
38、 最长递增子数组.....806
39、 最大子序和（53）【最大子数组（lintcode 41）】【最大和的子数组】【连续数列（面试题 16.17）】.....808
输出具体的子数组.....809
40、 第 k 大的子数组（lintcode1139）.....809
41、 和至少为 K 的最短子数组（862）【滑动窗口】.....811
42、 长度最小的子数组（209）.....814
方法 1：前缀和+二分.....815
方法 2：双指针.....815
43、 最短无序连续子数组（581）【内置函数】【单调栈】【双指针】.....816
方法 1：用内置排序（8ms）.....816
方法 2：单调栈（10s）.....817
方法 3：双指针，同时遍历.....818

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

44、 分割数组的最大值（410）【二分+贪心】【动规】.....818
方法 1：二分+贪心.....820
方法 2：动态规划.....821
87、 最长重复子数组（718）.....822
方法 1：用二维数组（82ms）.....823
方法 2：缩减 dp 成一位数组（48ms）.....823
方法 3：不是逆序的，而是存放临时值，与下一题类似（63ms）.....824
45、 电话目录管理系统（379）.....824
46、 递增的三元子序列（334）.....826
47、 摆动排序（280）.....827
48、 摆动排序 2（376）【峰与谷（面试题 10.11）】.....828
49、 矩阵置零（73）【零矩阵（面试题 01.08）】.....831
十一、 二分.....832
1. 二分的总结【找数字，找左边界，找右边界】.....832
2. 寻找两个正序数组的中位数（4）.....835
方法 1：weiwei 的参考代码 4.....840
方法 2.....844
3. 搜索旋转排序数组（33）.....845
方法 1.....846
方法 2：另外一种解法（也是 0ms）.....847
4. 搜索旋转排序数组二（81）.....848
5. 搜索旋转数组（面试题 10.03）.....850
6. 寻找旋转排序数组中的最小值（153）.....852
7. 寻找旋转排序数组中的最小值 2（154）（剑指 offer11 旋转数组的最小数字）【二分】.....854
8. 在排序数组中查找元素的第一个和最后一个位置（34）.....856
方法 1.....857
方法 2.....857
9. 搜索插入位置（35）.....859
方法 1：零神写的用语言自带 API.....859
方法 3：Weiwei 写的.....860
10. 二分查找（704）.....860
11. 魔术索引（面试题 08.03）.....862
12. 山脉数组的峰顶索引（852）【山形数组的顶峰坐标（lintcode1476）】.....864
13. 山脉序列中的最大值（lintcode585）.....865
14. 山脉数组中查找目标值（1095）.....865
1095 具体代码.....867
15. 在 D 天内送达包裹的能力（1011）【只能用二分+贪心，动规会超出时间限制】870
16. Shopee 的零食柜（Shopee2019 年第 2 题）.....872
17. 小张刷题计划（LCP 12）【二分+贪心】.....873
18. 爱吃香蕉的珂珂（875）.....875
19. 制作 m 束花所需的最少天数（1482）.....877
20. x 的平方根（69）.....879
方法 1：二分.....880

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法 2：牛顿法 881
21. 有序矩阵中第 k 小的元素（378）【可以转为有序矩阵中第 k 大的元素】 881
378 具体代码 883
可运行版本的第 k 大元素 884
22. 有序矩阵中第 k 个最小数组和（1439） 885
23. 搜索二维矩阵（74） 887
24. 搜索二维矩阵 2（240）【二维数组中的查找（剑指 offer 4）】【排序矩阵查找（面试题 10.09）】 890
25. 最后一块石头的重量（1046） 892
26. 马戏团人塔（面试题 17.08） 893
27. 交换和（面试题 16.21） 895
28. 稀疏数组搜索（面试题 10.05） 896
29. 稀疏相似度（面试题 17.26） 897
30. 数字流的秩（面试题 10.10） 899
十二、 数字问题 900
1、 寻找重复数（287） 900
287 具体代码 901
2、 数组中重复的数字（剑指 3） 902
方法 1：HashSet 902
方法 2：原地 903
3、 缺失的第一个正数（41） 903
4、 缺失数字（268）【异或】【0~n-1 中缺失的数字（剑指 53-2）】【消失的数字（面试题 17.04）】 904
方法 1：置换（时间复杂度 o(n)） 905
方法 2：二分法（时间复杂度 o(logn)） 906
方法 3：异或 907
方法 2：数学（高斯求和公式） 908
5、 找到所有数组中消失的数字（448） 908
6、 消失的两个数字（面试 17.19） 909
方法 1：数学 910
方法 2：位运算 911
7、 缺失区间（163） 912
8、 只出现一次的数字（136）【异或】 913
方法 1：时间复杂度 o(n)，空间复杂度 o(1)，要 4ms 913
方法 2：更快的做法，用异或，只要 1ms 914
9、 只出现一次的数字 2（137）【异或】【数组中数字出现的次数 2（剑指 56-2）】914
137 具体代码 917
10、 只出现一次的数字 3（260）【异或】【数组中数字出现的次数（剑指 56-1）】918
11、 错误的集合（645） 920
方法 1：异或 921
方法 2：使用额外空间 922
12、 情侣牵手（765） 922
方法 1：找规律 923
方法 2：贪心+异或 924

禁止商用 公众号：yes的练级攻略

13、 中心对称数（246） 925
14、 中心对称数 2（247） 926
15、 中心对称数 3（248） 928
16、 下一个数（面试题 05.04）【二进制表达式中 1 个数相同且大小最接近的两个数（一个略大，一个略小）】 930
17、 翻转数位（面试题 05.03） 931
十三、 字符串 932
● 常见数据类型的转换 932
字符串转 int、long、double、float 的方法 932
int、long、double、float 转字符串的方法 933
● String 类中常见的函数 934
● StringBuffer 类和 StringBuilder 类中常见的函数 935
1、 自动校对程序（lintcode1531） 936
2、 字符串的排序（567） 937
3、 自定义字符串排序（791）【桶排序】 939
4、 验证外星语词典（953） 941
5、 根据字符出现频率排序（451） 943
6、 单词规律（290） 946
方法 1：hashmap 946
方法 2：双指针 947
7、 单词规律 2（291） 947
8、 多次搜索（面试题 17.17） 949
9、 模式匹配（面试题 16.18）【pattern 只有 a 和 b】 951
10、 判断是否互为字符重排（面试题 01.02）【有效的字母异位词（242）】 955
方法 1：排序 956
方法 2：错位的字符串中每个字符的和相等，乘积相等 957
方法 3：哈希表（适用于整个 Unicode 字符范围） 958
11、 字母异位词分组（49）【变位词组（面试题 10.02）】 959
方法 1：通用方法，用到上面的判断两两相等（765ms） 959
方法 2：按字母顺序排序（9ms） 961
12、 找到字符串中所有字母异位词（438） 962
438 具体代码 964
13、 字符串压缩（面试题 01.06） 965
14、 压缩字符串（443） 966
15、 编码最短长度的字符串（471） 968
16、 字符串解码（394） 971
方法 1：辅助栈（2ms） 973
方法 2：递归（0ms） 973
17、 压缩算法（腾讯 2020 招聘-后台第一题）【解压缩】 974
【String 中常见的函数】 975
具体代码： 975
18、 反转数组 976
19、 旋转数组（189） 976
方法 1：反转 3 次 977

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法 2：使用环状替换.....978
20、 反转字符串（344）.....979
21、 反转字符串 2（541）【反转从字符串开头算起的每隔 2k 个字符的前 k 个字符】979
22、 旋转字符串（796）【字符串轮转（面试题 01.09）】.....980
方法 1：判断子串.....981
方法 2：KMP.....984
23、 左旋转字符串（剑指 58-2）.....985
方法 1：字符串切片.....985
方法 2：三次翻转字符串.....986
方法 3：列表遍历拼接.....987
24、 整数反转（7）【不需要双指针】.....987
25、 翻转字符串中的单词（151）（lintcode53）（剑指 58-1）【快手真题】.....989
【快手真题】需要结尾依旧是"."，且结尾的"."与前一个单词之间无空格。.....990
26、 翻转字符串 3（lintcode 927）.....991
27、 反转字符串中的单词 3（557）【反转字符串 3（lintcode 1173）】.....992
方法 1：调用自身函数.....992
方法 2：自己写反转函数.....992
28、 恢复空格（面试题 17.13）.....993
方法 1：DP（885ms）.....994
方法 2：字典树（16ms）.....995
29、 最长单词（面试题 17.15）.....996
30、 词典中最长的单词（720）.....997
31、 最常见的单词（819）.....999
32、 单词转换（面试题 17.22）.....1001
33、 最短单词距离（243）.....1003
34、 最短单词距离 2（244）.....1004
35、 最短单词距离 3（245）【单词距离（面试题 17.11）】.....1009
36、 重新排列日志文件（937）.....1011
37、 字符串切分（lintcode1587）.....1012
38、 整数转罗马数字（12）【贪心】.....1014
39、 罗马数字转整数（13）.....1017
方法 1：根据罗马数字规律求解.....1018
方法 2：普通的 hashMap.....1021
方法 3：自定义 hashMap.....1022
40、 整数转换英文表示（273）【整数的英语表示（面试题 16.08）】.....1024
41、 最小覆盖子串（76）.....1026
76 具体代码.....1027
42、 替换空格（剑指 5）.....1028
方法 1：StringBuffer 1028
方法 2：char 数组 1029
43、 URL 化（面试题 01.03）【上一题代码改一改】.....1029
44、 无重复字符的最长子串（3）【最长无重复字符的子串（lintcode384）】.....1030
方法 1：双指针加哈希表（9ms）.....1032
方法 2：直接用数组（3ms）.....1033

禁止商用
公众号：
yes的练级攻略

输出具体子串的可运行版本.....1033
45、 最长重复子串（1044）.....1035
46、 重复叠加字符串匹配（686）.....1039
47、 串联所有单词的子串（30）.....1040
30 具体代码.....1042
48、 计数二进制子串（696）.....1043
49、 至少有 K 个重复字符的最长子串（395）.....1045
50、 至少 K 个不同字符的子串（lintcode 1375）.....1047
方法 1：滑动窗口.....1047
方法 2：HashSet.....1048
51、 按字典序排在最后的子串（1163）.....1049
52、 按字典序排列最小的等效字符串（1061）【并查集】.....1050
53、 替换后的最长重复字符（424）.....1059
滑动窗口法具体代码.....1061
54、 重复的子字符串（459）.....1062
方法 1：indexof.....1064
方法 2：优化的 KMP.....1065
55、 字符串中的第一个唯一字符（387）.....1065
方法 1：哈希表（42ms）.....1066
方法 2：用数组替换哈希表（6ms）.....1066
方法 3：根据给定字符串长度分情况优化（2ms）.....1067
56、 判定字符串是否唯一（面试题 01.01）【位运算】.....1067
57、 实现 strStr()函数（28）【方法 2 有 KMP 详解】.....1068
方法 1：子串逐一比较（1ms）.....1069
方法 2：双指针（2ms）.....1071
方法 3：KMP【含有详解】.....1072
KMP 算法完整代码.....1082
这题具体的 KMP 代码（24ms）.....1084
58、 找到所有好字符串（1397）.....1085
59、 外观数列（38）.....1087
方法 1：递归（1ms）.....1088
方法 2：迭代（4ms）.....1089
60、 机器人能否返回原点（657）.....1089
61、 字符串相加（415）.....1090
62、 字符串相乘（43）.....1090
方法 1：普通竖式.....1091
方法 2：优化的竖式（去除相加的函数）.....1093
63、 井字游戏（面试题 16.04）.....1094
64、 Z 字形变换（6）.....1096
65、 比较版本号（165）.....1097
方法 1：分割+解析.....1098
方法 2：双指针.....1099
66、 把数字翻译成字符串（剑指 46）.....1100
方法 1：动规.....1102

禁止商用 公众号：yes的练级攻略

方法 2：数字求余.....1103
67、 第一个只出现一次的字符（剑指 50）.....1103
68、 统计重复个数（466）.....1105
69、 括号生成（22）【括号（面试题 08.09）】.....1109
70、 有效的括号（20）.....1110
71、 有效的括号字符串（678）.....1111
72、 最长有效括号（32）.....1113
73、 有效括号的嵌套深度（1111）.....1115
74、 括号的分数（856）.....1119
方法 1：数学.....1120
方法 2：栈.....1120
75、 删除无效的括号（301）.....1121
方法 1：DFS（快，讲解在后面）.....1121
方法 2：BFS（较慢）.....1124
76、 移除无效的括号（1249）.....1125
77、 强密码检验器（420）.....1127
78、 基本计算器（224）.....1129
79、 基本计算器 2（227）【计算器（面试题 16.26）】.....1131
80、 基本计算器 3（772）.....1132
81、 基本计算器 4（770）.....1134
82、 逆波兰表达式求值（150）.....1139
方法 1：纯数组模拟栈.....1140
方法 2：用栈.....1140
83、 Excel 表列序号（171）.....1141
84、 Excel 表列名称（168）.....1143
85、 二进制数转字符串（面试题 05.02）.....1143
86、 兰顿蚂蚁（面试题 16.22）.....1144
十四、回文.....1148
1、 最长回文子串（5）.....1148
方法 1：动态规划（时间复杂度 o(n^2），空间复杂度 o(n^2））.....1153
方法 2：中心扩散法（时间复杂度 o(n^2），空间复杂度 o(1)）.....1155
方法 3：马拉车算法（Manacher 算法）【时间复杂度 o(n)，是最优解，但是比较难】.....1156
2、 回文子串（647）【统计回文子串的数目（不删除字符）】.....1161
方法 1：中心扩散法（时间复杂度 o(n^2），空间复杂度 o(1)）.....1161
方法 2：马拉车（时间复杂度和空间复杂度都是 o(n)）.....1162
3、 最长回文子序列（516）.....1163
本题代码.....1167
打印一个最长回文子序列.....1167
4、 统计不同回文子序列（730）【可以删除零至多个字符来获得子序列】.....1169
具体动规代码.....1170
5、 最长回文串（409）.....1171
ASCII 编码.....1171
方法 1.....1172

禁止商用 公众号：yes的练级攻略

方法 2：Java8 的流式写法【更慢】.....1173
6、 最短回文串（214）.....1173
方法 1：马拉车算法（Manacher 算法）.....1174
方法 2：KMP 算法（8ms）.....1176
7、 回文数（9）【验证数】.....1177
方法 1：Java 中将 int 转成字符串的方法 o(n).....1177
方法 2：反转一半的数字 o(logn).....1178
8、 验证回文串（125）【验证字符串】.....1180
9、 验证回文字符串 2（680）.....1180
10、 验证回文字符串 3（1216）.....1182
11、 回文对（336）.....1184
方法 1：HashMap.....1185
方法 2：字典树.....1187
12、 删除回文子数组（1246）【区间 DP】.....1189
13、 分割回文串（131）.....1190
14、 分割回文串 2（132）.....1192
15、 分割回文串 3（1278）.....1195
16、 回文排列（266）（面试题 01.04）.....1197
17、 回文排列 2（267）.....1198
十五、 链表.....1200
ArrayList 和 LinkedList 中常用的方法.....1200
常见的 List 和数组之间的相互转换.....1201
【定义链表】.....1201
【生成链表】.....1202
【输出链表】.....1202
【计算链表的长度】.....1203
1、 两数相加（2）.....1203
2、 链表求和（面试题 02.05）【与上一题相比，有很多边界条件】.....1204
本题代码.....1204
可运行代码（含反向存放，反向输出和正向存放和正向输出）.....1206
3、 两数相加 2（445）【正向存放，正向输出】.....1208
4、 给单链表加一（369）.....1210
5、 反转链表（206）（剑指 24）.....1212
方法 1：迭代法（0ms）：.....1212
方法 2：递归法：.....1213
可运行版本.....1213
6、 反转链表 2（92）.....1214
7、 从尾到头打印链表（剑指 6）.....1216
方法 1：辅助队列（7ms）.....1217
方法 2：反转链表以后输出（0ms）.....1217
8、 重排链表（143）.....1218
方法 1：直接反转.....1219
方法 2：单独写一个反转函数（推荐）.....1219
9、 两两交换链表中的节点（24）.....1221

禁止商用 公众号：yes的练级攻略

方法 1：递归.....1221
方法 2：迭代.....1222
10、 K 个一组翻转链表（25）.....1222
11、 奇偶链表（328）.....1225
12、 旋转链表（61）.....1228
13、 相交链表（160）【两个链表的第一个公共节点（剑指 52）】【链表相交（面试题 02.07）】.....1230
160 具体代码.....1233
14、 回文链表（234）（面试题 02.06）【验证是否为回文链表】.....1234
234 具体代码.....1234
15、 环形链表（141）【判断有没有环】.....1235
16、 环形链表 2（142）【返回链表开始入环的第一个节点】【环路检测（面试题 02.08）】.....1237
17、 删除链表的节点（剑指 18）.....1240
面试题 18 具体代码.....1241
18、 删除链表中的节点（237）【删除中间节点（面试题 02.03）】.....1242
237 具体代码.....1244
19、 删除链表的倒数第 N 个节点（19）.....1244
20、 链表中倒数第 k 个节点（剑指 22）.....1245
21、 返回倒数第 k 个节点（面试题 02.02）【返回节点的值】.....1247
22、 寻找链表的中点.....1247
23、 删除排序链表中的重复元素（83）.....1248
24、 删除排序链表中的重复元素 2（82）【只保留原始链表中没有重复出现的数字】1249
25、 移除重复节点（面试题 02.01）.....1251
方法 1：hashSet（5ms）.....1252
方法 2：两重循环（395ms）.....1253
方法 3：位运算（3ms）.....1253
26、 移除链表元素（203）.....1254
删除链表的测试函数.....1256
27、 合并两个有序链表（21）.....1258
28、 合并 K 个排序链表（23）.....1259
29、 分隔链表（86）【分割链表（面试题 02.04）】【所有小于 x 的节点都在大于或等于 x 的节点之前】.....1262
30、 分隔链表（725）【分隔为 k 个连续部分】.....1263
31、 排序链表（148）【要求 O(n log n) 时间复杂度和常数级空间复杂度】.....1265
方法 1：常数级空间复杂度一定要用非递归.....1267
方法 2：如果不要求常数级的空间复杂度，可以使用递归.....1269
32、 复制带随机指针的链表（138）【复杂链表的复制（剑指 35）】.....1270
方法 1：分别拷贝链表、指定随机指针、重连链表.....1272
方法 2：利用 HashMap.....1274
33、 设计跳表（1206）.....1275
34、 特定深度节点链表（面试题 04.03）.....1280
方法 1：DFS.....1280
方法 2：BFS.....1282

禁止商用 公众号：yes的练级攻略

十六、 二叉树 1283
【定义 TreeNode】 1283
【生成 TreeNode】将包含前后[]的 String 转成 TreeNode。 1283
【输出成 String】 1285
1、 二叉树的前序遍历（144） 1285
方法 1：递归 1285
方法 2：迭代 1286
2、 二叉树的中序遍历（94） 1287
方法 1：递归算法（最快最简单）： 1287
方法 2：迭代算法： 1287
3、 二叉树后序遍历（145） 1288
方法 1：递归 1288
方法 2：迭代 1289
//前序遍历-方法 1：递归 1291
//前序遍历-方法 2：迭代 1291
//中序遍历-方法 1：递归 1292
//中序遍历-方法 2：迭代 1292
//后序遍历-方法 1：递归 1292
//后序遍历-方法 2：迭代 1292
4、 从上到下打印二叉树（剑指 32-1） 1294
5、 从上到下打印二叉树 2（剑指 32-2）【二叉树的层序遍历（102）】 1296
方法 1：BFS（1ms） 1298
方法 2：DFS（0ms） 1298
6、 从上到下打印二叉树 3（剑指 32-3） 1299
方法 1：层序遍历加双端队列 1300
方法 2：层序遍历加倒序 1301
7、 二叉树的层序遍历 2（107） 1302
方法 1：BFS 1302
方法 2：DFS 1303
8、 二叉树的锯齿形层次遍历（103） 1304
方法 1：BFS 1305
方法 2：DFS 1307
9、 二叉树的层平均值（637） 1308
10、 输出二叉树（655） 1309
方法 1：递归 1311
方法 2：迭代 1312
11、 二叉树的堂兄弟节点（993） 1314
12、 二叉树的完全性检验（958） 1316
13、 二叉树的序列化和反序列化（297）【序列化二叉树（剑指 37）】【序列化和反序列化二叉搜索树（449）】（lintcode 1235） 1318
方法 1：DFS（163ms） 1319
方法 2：BFS（15ms） 1320
方法 3：括号表示编码+递归下降解码（23ms） 1323
14、 序列化和反序列化 N 叉树（428） 1324

非商用 公众号：yes的练级攻略

lintcode 版本 1532 1327
15、 N 叉树的前序遍历（589） 1329
方法 1：递归 1329
方法 2：迭代 1330
16、 N 叉树的后序遍历（590） 1331
方法 1：递归 1331
方法 2：迭代 1332
17、 N 叉树的层序遍历（429） 1333
方法 1：递归 1334
方法 2：迭代 1336
18、 不同的二叉搜索树（96） 1337
方法 1：动态规划 1338
方法 2：数学定理 1339
19、 不同的二叉搜索树 2（95） 1339
方法 1：卡特兰数 1341
方法 2：递归 1343
本地中可以运行的代码 1344
20、 后继者（面试题 04.06）【找到二叉搜索树中指定节点的下一个节点】 1347
21、 验证二叉树（1361） 1349
22、 验证二叉搜索树（98）【合法二叉搜索树（面试题 04.05）】 1352
23、 验证前序遍历序列二叉搜索树（255） 1353
24、 恢复二叉搜索树（99） 1355
25、 修建二叉搜索树（669） 1358
26、 二叉搜索树序列（面试题 04.09） 1359
27、 二叉搜索树迭代器（173） 1360
28、 将二叉搜索树变平衡（1382） 1363
29、 把二叉搜索树转换为累加树（538）【1038 从二叉搜索树到更大和树】 1365
方法 1：回溯 双 o(n) 1366
方法 2：反序中序 Morris 遍历 空间复杂度降为 o(1) 1368
30、 查找两棵二叉搜索树之和（1214） 1369
31、 二叉搜素子树的最大键值和（1373） 1371
32、 二叉搜索树中的众数（501） 1374
33、 二叉搜索树的最小绝对差（530）【二叉搜索树节点最小距离（783）】 1376
34、 二叉搜索树中第 K 小的元素（230） 1378
方法 1：递归 1379
方法 2：迭代（更短） 1380
35、 二叉搜索树的第 k 大节点（剑指 54）【跟上一题的区别就是遍历时候右节点先遍历】 1381
方法 1：先反转链表，再找第 k 小的元素 1381
方法 2：递归 1382
方法 3：迭代 1382
36、 删除二叉搜索树中的节点（450） 1383
方法 1：递归 1387
方法 2：迭代 1388

公众号：yes的练级攻略

37、 二叉搜索树的后续遍历序列（剑指 33）.....1391
38、 将二叉搜索树转化为排序的双向链表（426）【二叉搜索树和双向链表（剑指 36）】.....1392
426 具体代码.....1396
39、 对称二叉树（101）（剑指 28）.....1397
方法 1：递归算法（1ms）.....1398
方法 2：递归算法（0ms）简写版本.....1399
方法 3：迭代算法（1ms）.....1399
40、 二叉树的镜像（剑指 27）【与下一题翻转二叉树一样】.....1400
41、 翻转二叉树（226）【与牛客网 18 题一致】.....1402
方法 1：递归.....1402
方法 2：迭代.....1403
42、 上下翻转二叉树（156）.....1403
43、 二叉树的最大深度（104）（剑指 55-1）.....1406
方法 1：递归（0ms）.....1407
方法 2：BFS 迭代.....1407
44、 N 叉树的最大深度（559）.....1408
方法 1：递归.....1409
方法 2：迭代.....1410
45、 二叉树的最小深度（111）.....1410
1、 递归（0ms）.....1411
2、 迭代 BFS.....1412
46、 平衡二叉树（110）（剑指 55-2）【检查平衡性（面试题 04.04）】.....1413
47、 二叉树的直径（543）.....1414
543 具体代码.....1415
48、 树的直径（1245）.....1416
49、 二叉树的所有路径（257）.....1418
方法 1：递归.....1419
方法 2：迭代.....1420
50、 二叉树中所有路径中最大路径和的路径（从根到叶子）.....1421
51、 二叉树中的最大路径和（124）.....1423
方法 1：DFS.....1424
方法 2：DFS 更简短的.....1425
52、 路径总和（112）.....1425
方法 1：递归代码.....1425
方法 2：BFS 代码.....1426
53、 路径总和 2（113）【二叉树中和为某一值的路径（剑指 34）】.....1426
113 代码：深度优先加状态重置.....1427
54、 路径总和 3（437）【求和路径（面试题 04.12）】.....1429
55、 路径和 4（666）.....1431
56、 同长同值路径（687）.....1433
57、 求根到叶子节点数字总和（129）.....1436
方法 1：BFS.....1437
方法 2：DFS.....1438

禁止商用 公众号：yes的练级攻略

58、 左叶子之和（404）.....1439
59、 二叉树中所有距离为 K 的结点（863）.....1440
60、 从前序与中序遍历序列构造二叉树（105）【重建二叉树（剑指 7）】.....1444
方法 1：递归做法（3ms）.....1445
方法 2：最优解：.....1446
如果有重复元素，可以用方法 2，完整版代码如下：.....1447
61、 从中序与后序遍历序列构造二叉树（106）.....1450
62、 根据前序和后序遍历构造二叉树（889）.....1453
63、 二叉树展开为链表（114）.....1455
解法 1：利用前序遍历的递归（1ms，不是最佳）.....1457
解法 2：采用后序遍历.....1458
64、 BiNode（面试题 17.12）.....1459
65、 将有序数组转换为二叉搜索树（108）【最小高度树（面试题 04.02）】.....1460
66、 有序链表转换二叉搜索树（109）.....1462
方法 1：快慢指针找中点（时间复杂度 o(nlogn)）.....1463
方法 2：分治加中序（时间复杂度 o(n)）.....1465
67、 相同的树（100）.....1466
方法 1：递归.....1467
方法 2：迭代.....1468
68、 合并二叉树（617）.....1469
方法 1：递归.....1470
方法 2：迭代.....1471
69、 完全二叉树的节点个数（222）.....1472
70、 计算右侧小于当前元素的个数（315）.....1473
71、 最大树（654）（lintcode126）【单调栈（数据量大）】【递归（数据量小）】.....1475
方法 1：单调栈（不是很快，但是适用于数据比较大的时候）.....1477
方法 2：递归（适用于数字比较小的时候）.....1478
72、 二叉搜索树的最近公共祖先（235）（剑指 68-1）.....1479
方法 1：直接用下一题优化后的代码（7ms）.....1480
方法 2：（递归）利用二叉搜索树的特性（6ms）.....1481
方法 3：（迭代）利用二叉搜索树的特性（6ms）.....1481
73、 二叉树的最近公共祖先（236）（剑指 68-2）【首个共同祖先（面试题 04.08）】.....1482
方法 1：后序遍历 DFS.....1484
方法 2：优化以后【因为 p、q 为不同节点且均存在于给定的二叉树中，删除都为空的判断】.....1485
74、 树节点的第 K 个祖先（1483）.....1485
75、 节点与其祖先之间的最大差值（1026）.....1487
76、 树的子结构（剑指 26）【子结构不是子树】.....1489
77、 另一个树的子树（572）【子树是节点和所有子孙】【检查子树（面试题 04.10）】.....1492
78、 最大 BST 子树（333）.....1494
79、 二叉树的右视图（199）.....1496

禁止商用 公众号：yes的练级攻略

方法 1：BFS.....1497
方法 2：DFS.....1498
80、 二叉树中的最长交错路径（1372）.....1498
方法 1：递归.....1500
81、 二叉树的边界（545）.....1502
十七、 线段树.....1504
1、 线段树的构造（lintcode 201）.....1505
2、 线段树的构造 2（lintcode 439）.....1506
3、 线段树的查询（lintcode 202）.....1507
4、 线段树的查询 2（lintcode 247）.....1509
5、 线段树的修改（lintcode 203）.....1510
6、 天际线问题（218）.....1512
十八、 栈.....1517
1、 用栈实现队列（232）【用两个栈实现队列（剑指 9）】【化栈为队（面试题 03.04）】.....1517
232 可运行版本.....1519
剑指 9 版本.....1520
2、 用队列实现栈（225）.....1520
3、 动物收容所（面试题 03.06）.....1522
4、 最小栈（155）【包含 min 函数的栈（剑指 30）】【栈的最小值（面试题 03.02）】.....1524
思路 1，用辅助栈（7ms）.....1526
思路 2：不利用辅助栈，自建 Node（5ms）.....1527
5、 最大栈（716）.....1528
6、 扁平化嵌套列表迭代器（341）.....1531
7、 验证栈序列（946）【栈的压入、弹出序列（剑指 31）】.....1533
8、 栈排序（面试题 03.05）.....1535
9、 堆盘子（面试题 03.03）.....1537
10、 三合一（面试题 03.01）.....1539
11、 最终优惠价（leedcode1852 题）【单调栈】.....1541
12、 最大区间（lintcode1339）【单调栈】【区间中最小值*区间所有数的和】.....1543
13、 逛街（腾讯 2020 校园招聘-后台第 2 题）【单调栈】.....1544
14、 柱状图中最大矩形（84）；直方图最大矩形覆盖（lintcode122）【单调栈】1545
方法 1：九章.....1546
方法 2：甜姨.....1547
15、 最大矩形（85）（lintcode510）【单调栈+DP】.....1547
Lintcode510 代码.....1548
与在 leetcode 中的不同用加粗表示。以下是 leetcode85 代码.....1549
16、 最大正方形（221）（lintcode436）【单调的+DP，上一题改造了两行】【只 DP】.....1550
方法 1：单调栈+DP.....1550
方法 2：只用动规.....1553
17、 每日温度（739）（lintcode1060）【方法 2 是单调栈】.....1553
方法 1：从后面开始找，效率比较高（4ms）.....1553

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法 2：单调栈（17ms）.....1554
18、 股票价格跨度（901）（lintcode1740）【单调栈】.....1555
19、 下一个更大元素 1（496）【单调栈】.....1556
20、 去除重复字母（316）【单调栈】.....1557
21、 不同字符的最小子序列（1081）【单调栈，与上一题的意思一样】.....1558
22、 移掉 k 位数字（402）【单调栈】【双指针】.....1559
方法 1：单调栈.....1560
方法 2：双指针.....1561
23、 子数组的最小值之和（907）（lintcode1734）【单调栈】.....1561
24、 K 个空花盆（683）.....1563
方法 1：单调栈（111ms）.....1564
方法 2：双指针（5ms）.....1565
方法 3：红黑树（81ms）.....1566
十九、 有限自动机.....1566
1、 字符串转换整数 atoi（8）.....1569
方法 1：有限自动机.....1571
方法 2：简单模拟.....1572
2、 正则表达式匹配（10）（剑指 19）.....1573
方法 1：动态规划.....1575
方法 2：有限自动机（时间复杂度 o(MN)）.....1575
3、 有效数字（65）【表示数值的字符串（剑指 20）】.....1577
方法 1：正则表达式（22ms）.....1577
方法 2：逐个匹配（3ms）.....1578
方法 3：有限自动机（3ms）.....1581
二十、 数学推导题.....1581
1、 机器人跳跃（lintcode1426）.....1581
2、 质因数统计（lintcode1295）【枚举】.....1582
3、 丑数（263）【判断一个数是不是丑数】.....1583
4、 丑数 2（264）【丑数（剑指 49）】【2,3,5 用动规】.....1585
5、 超级丑数（313）.....1585
6、 第 k 个数（面试题 17.09）【3,5,7 用动规】.....1587
7、 连续整数求和（829）（lintcode1439）.....1588
方法 1：进阶数学.....1590
方法 2：数学.....1590
8、 除数博弈（1025）.....1591
方法 1：找规律.....1592
方法 2：动规.....1592
9、 Nim 游戏（292）.....1592
10、 Pow(x,n)（50）【数值的整数次方（剑指 16）】.....1593
方法 1：快速幂+递归.....1595
方法 2：快速幂+迭代.....1596
11、 1~n 整数中 1 出现的次数（面试题 43.1）.....1596
12、 数字 1 的个数（233）.....1600
13、 可被 K 整除的最小整数（1015）.....1600

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

14、 灯泡开关（319）.....1603
15、 月份天数（lintcode1141）.....1603
16、 圆圈中最后剩下的数字（剑指 62）【约瑟夫环问题】.....1604
17、 第 N 个数字（400）【数字序列中某一位的数字（剑指 44）】.....1608
18、 求 1+2+...+n（剑指 64）.....1610
19、 交点（面试题 16.03）【两个浮点型大小的判断】.....1612
20、 平分正方形（面试题 16.13）.....1619
21、 最小公倍数和最大公因数.....1621
代码 1：迭代式辗转相除法（时间复杂度 o(logn)，在《离散数学》上给出了一个证明（P77），可以证明，gcd 的运行次数不会超过 2*log2（n+1）n 为较小数。）1624
代码 2：递归式辗转相除法.....1625
22、 丑数 3（1201）.....1626
23、 水壶问题（365）.....1628
方法 1：BFS.....1629
方法 2：数学【贝祖定理】.....1632
24、 卡牌分组（914）.....1633
25、 分数加减运算（592）.....1635
26、 直线上最多的点数（149）.....1637
27、 最佳直线（面试题 16.14）.....1639
方法 1：数学（直线的一般方程）.....1640
方法 2：LinkedHashMap.....1641
28、 字符串的最大公因子（1071）.....1642
29、 第 N 个神奇数字（878）.....1644
30、 Rand7 求 rand10（470）.....1645
方法 1：拒绝采样.....1646
31、 快乐数（202）.....1647
32、 服务中心的最佳位置（1515）.....1648
方法 1：梯度下降法（97ms）.....1652
方法 2：爬山法（10ms）.....1654
方法 3：三分查找（246ms）.....1656
33、 分数到小数（166）.....1657
34、 阶乘后的零（172）【阶乘尾数（面试题 16.05）】.....1660
35、 阶乘函数后 k 个零（793）.....1663
36、 计算质数（204）.....1664
37、 质数排列（1175）.....1665
38、 递归乘法（面试题 08.05）.....1666
方法 1：递归.....1667
方法 2：位运算.....1667
39、 运算（面试题 16.09）.....1668
40、 最大数值（面试题 16.07）.....1671
41、 范围内的数字技术（1067）.....1675
二十一、 大数据.....1678
1、 推荐结果打散（lintcode1166）.....1678
2、 关联查询（lintcode1145）.....1680

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

二十二、 多线程.....1681
1、 哲学家进程（1226）.....1681
2、 按序打印（1114）.....1683
3、 交替打印 FooBar（1115）.....1686
方法 1：synchronized.....1687
方法 2：两个 Semaphore 交替运行.....1689
4、 打印零与奇偶数（1116）.....1690
方法 1：synchronized.....1690
方法 2：三个 Semaphore 交替运行.....1692
两个都能用的测试类.....1692
5、 Fizz Buzz（412）.....1693
6、 交替打印字符串（1195）.....1695
方法 1：synchronized.....1695
方法 2：Semaphore.....1697
两个都能用的测试类.....1699
二十三、 位运算.....1700
【位异或(^)的性质】.....1700
【位与(&) 的性质】.....1703
【位非(~) 的性质】.....1703
【位或(|) 的性质】.....1703
【移位】.....1703
1. 交换数字（面试题 16.01）.....1703
方法 1：位运算.....1703
方法 2：加法（+--）.....1703
方法 3：减法（-+-）.....1704
2. 汉明距离（461）.....1704
方法 1：内置函数.....1705
方法 2：移位.....1706
方法 3：布赖恩·克尼根（Brian Kernighan）算法.....1707
3. 整数转换（面试题 05.06）.....1707
4. 数字范围按位与（201）.....1708
方法 1：布赖恩·克尼根（Brian Kernighan）算法.....1710
方法 2：位移.....1711
5. 颠倒二进制位（190）.....1712
方法 1：o(n)时间复杂度.....1712
方法 2：o(1)时间复杂度.....1713
6. 位 1 的个数（191）【二进制中 1 的个数（剑指 15）】.....1713
7. 两数相除（29）.....1715
8. 两整数之和（371）【不用加减乘除做加法（剑指 65）】【不用加号的加法（面试题 17.01）】.....1716
9. 格雷编码（89）【移位和异或】.....1717
10. 生命游戏（289）.....1719
11. 二进制求和（67）.....1722
方法 1：模拟，类似链表求和.....1722

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法 2：位运算.....1724
12. 配对交换（面试题 05.07）.....1725
13. 插入（面试题 05.01）.....1726
14. 绘制直线（面试题 05.08）.....1726

OJ 输入输出

- 多行同类数据（int），不知长度

题目描述

计算a+b

输入描述:

输入包括两个正整数a,b(1 <= a, b <= 10^9),输入数据包括多组。

输出描述:

输出a+b的结果

示例1

输入

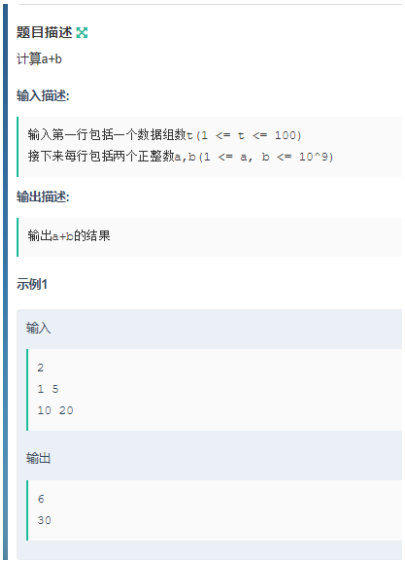
```
1 5
10 20
```

输出

```
6
30
```

```
import java.util.Scanner;
public class Main{
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()){
            int a = sc.nextInt();
            int b = sc.nextInt();
            System.out.println(a+b);
        }
    }
}
```

- 第一行知道数据组数，接下去每行都是两个整数（int）



题目描述

计算a+b

输入描述:

输入第一行包括一个数据组数t(1 <= t <= 100)
接下来每行包括两个正整数a,b(1 <= a, b <= 10^9)

输出描述:

输出a+b的结果

示例1

输入

```
2
1 5
10 20
```

输出

```
6
30
```

禁止商用 公众号：yes的练级攻略

```
import java.util.*;
public class Main{
    public int function (int a, int b) {
        return a + b;
    }
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        Main text = new Main();
        int t = sc.nextInt();
        int[] a = new int [t];
        int[] b = new int [t];
        for (int i = 0; i < t; i++) {
            a[i] = sc.nextInt();
            b[i] = sc.nextInt();
            System.out.println(text.function(a[i], b[i]));
        }
    }
}
```

- 如果输入遇到 0 0 就结束

题目描述

计算a+b

输入描述:

输入包括两个正整数a,b(1 <= a, b <= 10^9),输入数据有多组, 如果输入为0 0则结束输入

输出描述:

输出a+b的结果

示例1

输入

```
1 5
10 20
0 0
```

输出

```
6
30
```

```
import java.util.*;
public class Main{
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()){
            int a = sc.nextInt();
            int b = sc.nextInt();
            if (a != 0 || b != 0) {
                System.out.println(a+b);
            }
        }
    }
```

禁止商用
号：
级攻略

}

- 计算一系列数的和

题目描述

计算一系列数的和

输入描述:

输入数据包括多组。
每组数据一行,每行的第一个整数为整数的个数n(1 <= n <= 100), n为0的时候结束输入。
接下来n个正整数,即需要求和的每个正整数。

输出描述:

每组数据输出求和的结果

示例1

输入

```
4 1 2 3 4
5 1 2 3 4 5
0
```

输出

```
10
15
```

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            int n = sc.nextInt();
            if (n == 0) return;
            int sum = 0;
            for (int i = 0; i < n; i++) {
                sum += sc.nextInt();
            }
            System.out.println(sum);
        }
    }
}
```

- t 行，计算一系列数据的和

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

时间限制：C/C++ 1秒，其他语言2秒
空间限制：C/C++ 32768K，其他语言65536K
64bit IO Format: %lld

题目描述

计算一系列数的和

输入描述:

输入的第一行包括一个正整数t(1 <= t <= 100)，表示数据组数。
接下来t行，每行一组数据。
每行的第一个整数为整数的个数n(1 <= n <= 100)。
接下来n个正整数，即需要求和的每个正整数。

输出描述:

每组数据输出求和的结果

示例1

输入

```
2
4 1 2 3 4
5 1 2 3 4 5
```

输出

```
10
15
```

商用
公众号：
yes的练级攻略

```
import java.util.Scanner;
public class Main{
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        for (int i = 0; i < t; i++) {
            int n = sc.nextInt();
            int[] nums = new int[n];
            int res = 0;
            for (int j = 0; j < n; j++){
                nums[j] = sc.nextInt();
                res += nums[j];
            }
            System.out.println(res);
        }
    }
}
```

- 计算一系列数的和，没有停止符号

题目描述

计算一系列数的和

输入描述:

> 输入数据有多组，每行表示一组输入数据。
> 每行的第一个整数为整数的个数n(1 <= n <= 100)。
> 接下来n个正整数，即需要求和的每个正整数。

输出描述:

> 每组数据输出求和的结果

示例1

输入

```
4 1 2 3 4
5 1 2 3 4 5
```

输出

```
10
15
```

用
:
yes的练级攻略

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextInt()) {
            int n = sc.nextInt();
            int sum = 0;
            int[] nums = new int[n];
            for (int i = 0; i < n; i++) {
                nums[i] = sc.nextInt();
                sum += nums[i];
            }
            System.out.println(sum);
        }
    }
}
```

- 计算一系列数的和，没有长度，n 行无限

题目描述

计算一系列数的和

输入描述:

输入数据有多组，每行表示一组输入数据。

每行不定有n个整数，空格隔开。(1 <= n <= 100)。

输出描述:

每组数据输出求和的结果

示例1

输入

```
1 2 3
4 5
0 0 0 0 0
```

输出

```
6
9
0
```

```
import java.util.*;
public class Main{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()){
            String[] s = sc.nextLine().split(" ");
            int sum = 0;
            for (int i = 0; i < s.length; i++) {
                sum += Integer.parseInt((s[i]));
            }
            System.out.println(sum);
        }
    }
}
```

- 字符串排序（1）【第一行是数目】

非商用
公众号：
yes的练级攻略

题目描述

对输入的字符串进行排序后输出

输入描述:

输入有两行，第一行n

第二行是n个空格隔开的字符串

输出描述:

输出一行排序后的字符串，空格隔开，无结尾空格

示例1

输入

```
5
c d a bb e
```

输出

```
a bb c d e
```

```
import java.util.*;
public class Main{

    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        String[] s = new String[n];
        for (int i = 0; i < n; i++){
            s[i] = sc.next();
        }
        Arrays.sort(s);
        System.out.print(s[0]);
        for (int i = 1; i < n; i++){
            System.out.print(" " + s[i]);
        }
    }
}
```

- 字符串排序（2）【多行，未知行数，空格隔开】

禁止商用 公众号：yes的练级攻略

时间限制：C/C++ 1秒，其他语言2秒
空间限制：C/C++ 262144K，其他语言524288K
64bit IO Format: %lld

题目描述

对输入的字符串进行排序后输出

输入描述:

多个测试用例，每个测试用例一行。

每行通过空格隔开，有n个字符，n＜100

输出描述:

对于每组测试用例，输出一行排序过的字符串，每个字符串通过空格隔开

示例1

输入

```
a c bb
f dddd
nowcoder
```

输出

```
a bb c
dddd f
nowcoder
```

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] s = sc.nextLine().split(" ");
            Arrays.sort(s);
            StringBuffer res = new StringBuffer();
            for (int i = 0; i < s.length; i++) {
                if (i == s.length - 1) {
                    res.append(s[i]);
                } else {
                    res.append(s[i]);
                    res.append(" ");
                }
            }
            System.out.println (res);
        }
    }
}
```

- 字符串排序（3）【多行，未知行数，逗号隔开】

商用 公众号：yes的练级攻略

字符串排序(3)

时间限制：C/C++ 1秒，其他语言2秒
空间限制：C/C++ 262144K，其他语言524288K
64bit IO Format: %lld

题目描述

对输入的字符串进行排序后输出

输入描述:

多个测试用例，每个测试用例一行。
每行通过,隔开，有n个字符，n＜100

输出描述:

对于每组用例输出一行排序后的字符串，用','隔开，无结尾空格

示例1

输入

```
a,c,bb
f,dddd
nowcoder
```

输出

```
a,bb,c
dddd,f
nowcoder
```

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String[] s = sc.nextLine().split(",");
            Arrays.sort(s);
            StringBuffer res = new StringBuffer();
            for (int i = 0; i < s.length; i++) {
                if (i == s.length - 1) {
                    res.append(s[i]);
                } else {
                    res.append(s[i]);
                    res.append(",");
                }
            }
            System.out.println (res);
        }
    }
}
```

- 输入是已知大小的二维数组

输入描述第一行是一个正整数 n，表示二维数组有 n 行 n 列。

禁止商用 公众号：yes的练级攻略

```
public static void main(String[] args) {
     Scanner scan = new Scanner(System.in);
     int n=scan.nextInt();//定义需要的阶层数n
     int[][]array=new int[n][n];//定义一个n*n的数组array
     System.out.println("输入数组的各个元素：");
     for(int i=0;i<n;i++) {
         for (int j = 0; j < n; j++)
             array[i][j] = scan.nextInt();
     }
}
```

- 输入的每组测试数据有多行

输入描述：第一行是一个正整数T，表示有T组测试数据。
之后每组数据有三行，第一行为n（1<=n<=10000），第二行有n个正整数，第三行也有n个正整数；都在整数范围内。

```
输入:
3
3
1 2 3
1 2 3
4
4 3 2 1
1 1 1 1
2
1 2
10 20
输出:
7
4
30
```

```
public static void main(String[] args) {
     Scanner in = new Scanner(System.in);
     int T = in.nextInt();
     while(T>0){
         T--;//为了保证输出的第一行会换行
         //a和b分别放每组测试用例的第二行和第三行
         int[] a = new int[10000];
         int[] b = new int[10000];
         int n = in.nextInt();
         for(int i=1; i<=n; i++){
             a[i] = in.nextInt();
         }
         for(int i=1 ;i<=n; i++){
             b[i] = in.nextInt();
         }
     }//将输入带到function()方法里去运算
     System.out.println(function(a,b))
}
```

- 每行测试数据的数量在该行开头给出

输入描述：第一行是一个正整数T，表示有T组测试数据。
之后每组数据第一个数为N（1<=n<=10000），紧接着有N个正整数；（注意在一行）

```
public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        int T = input.nextInt();
        for(int i=0; i<T; i++) {
            int N = input.nextInt();
            String s = input.next();//注意用的是next()，不换行
            char[] ch = new char[10000];
            for (int j=0; j<N; j++) {
                ch[j] = s.charAt(j);
            }
            System.out.println(function(ch));
        }
    }
```

- 测试数据组数未知且结果需要取模

输入描述：每行输入两个数分别为 N 和 L ,直到N 和 L为止;结果对1000000007取模

```
public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        Long mod = 1000000007l;
        while (input.hasNext()) {
            Long n = input.nextLong();
            Long l = input.nextLong();
            if (n == 0 && l == 0) {
                break;
            }//将输入带到function()方法里去运算
            System.out.println(function(n,l)%mod);
        }
}
```

- 判断子串是否与主串的某段相等

```java
public class StringExist {
    public static void main(String [] args){
        //输入主串和子串
        Scanner input = new Scanner(System.in);
        String mainString = input.nextLine();
        String subString = input.nextLine();
        //判断方法调用
        boolean result = jugdeExist(mainString,subString);
        System.out.println(result);

    }
    //判断子串是否与主串的某段相等
    public static boolean jugdeExist(String mainString,String subString){
        if (null== mainString)return false;
            String[] forArray = mainString.split(",");
            for (String Id : forArray) {
                if(Id.equals(subString))
                    return true;
            }
        return false;
    }
}
```

- 输入三行数据，第一行是整数，第二行和第三行要去除中间的空格

nextInt()之后的 nextLine 无效

Java中Scanner类在nextInt()后无法输入nextLine()的问题

首先，Scanner是一个扫描器，它扫描数据都是去内存中一块缓冲区中进行扫描并读入数据的，而我们在控制台中输入的数据也都是被先存入缓冲区中等待扫描器的扫描读取。这个扫描器在扫描过程中判断停止的依据就是"空白符"，空格啊，回车啊什么的都算做是空白符。

nextInt()方法在扫描到空白符的时候会将前面的数据读取走，但会丢下空白符"\r"在缓冲区中，但是，nextLine()方法在扫描的时候会将扫描到的空白符一同清理掉。

了解了这两个方法特性和区别，就知道了上边的代码究竟是怎么回事，以及知道了解决的方法。像是上边的代码nextInt()方法之后在缓冲区中留下了"\r"，然后nextLine()方法再去缓冲区找数据的时候首先看到了"\r"，然后就把这个"\r"扫描接收进来，并在缓冲区内清除掉。其实，nextLine()方法是执行过的，并没有不执行。

解决方法：

在nextInt()之后，nextLine（）之前插入一个空白的nextLine（）把残留的\r读取掉。

方法 1：

```
Scanner sc = new Scanner(System.in);
int N = sc.nextInt();
String s1 = "";
for (int i =0; i < N; i++){
        s1 += sc.next().charAt(0);
}
String s2 = "";
for (int i =0; i < N; i++){
        s2 += sc.next().charAt(0);
}
```

方法 2：

```
Scanner sc = new Scanner(System.in);
int N = sc.nextInt();
sc.nextLine(); //换行
String s1 = sc.nextLine();
String s2 = sc.nextLine();
s1 = s1.replaceAll(" ", "");
s2 = s2.replaceAll(" ", "");
```

方法 3：

yes的练级攻略

```
Scanner sc = new Scanner(System.in);
int N = Integer.valueOf(sc.nextLine()); //不能用 sc.nextInt()，否则后面的 nextLine 无效
String s1 = sc.nextLine();
String s2 = sc.nextLine();
sc.close();
s1 = s1.replaceAll(" ", "");
s2 = s2.replaceAll(" ", "");
```

- 输出保留 2 位的小数

```
System.out.print(String.format("%.2f", res));
```

- Arrays.sort

```
import java.util.*;
public class Main {
    public static void main(String[] args){
        int[] nums = new int[] {1,5,3,2,4,6};
        int[][] arrs = new int[][] {{1,1},{4,2},{2,1},{3,1},{1,5},{5,2},{2,6},{1,1},{3,2}};
        //一维数组升序
          System.out.println("一维数组，升序排序");
        Arrays.sort(nums);
        for (int num : nums){
            System.out.print(num+" ");
        }
        System.out.println();
        //一维数组降序 Arrays.sort(newNums,Collections.reverseOrder()); 不能用于基本数据类型
          System.out.println("一维数组，降序排序");
        Integer newNums[] = new Integer[nums.length];
        for (int i = 0; i < nums.length; i++){
            newNums[i] = new Integer(nums[i]);
        }
        Arrays.sort(newNums,Collections.reverseOrder());
        for (int num : newNums){
            System.out.print(num+" ");
        }
        System.out.println();
        //二维数组按第一列升序，第二列升序的方式排序
          System.out.println("二维数组，按第一列升序，第二列升序的方式排序");
        Arrays.sort(arrs, (o1, o2) -> o1[0] == o2[0] ? o1[1] - o2[1] : o1[0] - o2[0]);
        for (int i = 0; i < arrs.length; i++) {
            for (int j = 0; j < arrs[0].length; j++) {
                System.out.print(arrs[i][j] + " ");
            }
            System.out.println();
        }
        //二维数组按第一列升序，第二列降序的方式排序
          System.out.println("二维数组，按第一列升序，第二列降序的方式排序");
```

```
        Arrays.sort(arrs, (o1, o2) -> o1[0] == o2[0] ? o2[1] - o1[1] : o1[0] - o2[0]);
        for (int i = 0; i < arrs.length; i++) {
            for (int j = 0; j < arrs[0].length; j++) {
                System.out.print(arrs[i][j] + " ");
            }
            System.out.println();
        }
        //二维数组按第一列降序，第二列升序的方式排序
        System.out.println("二维数组，按第一列降序，第二列升序的方式排序");
        Arrays.sort(arrs, (o1, o2) -> o1[0] == o2[0] ? o1[1] - o2[1] : o2[0] - o1[0]);
        for (int i = 0; i < arrs.length; i++) {
            for (int j = 0; j < arrs[0].length; j++) {
                System.out.print(arrs[i][j] + " ");
            }
            System.out.println();
        }
        //二维数组按第一列降序，第二列降序的方式排序
        System.out.println("二维数组，按第一列降序，第二列降序的方式排序");
        Arrays.sort(arrs, (o1, o2) -> o1[0] == o2[0] ? o2[1] - o1[1] : o2[0] - o1[0]);
        for (int i = 0; i < arrs.length; i++) {
            for (int j = 0; j < arrs[0].length; j++) {
                System.out.print(arrs[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

- 二值问题的

比如说很多题目都类似这样，巴拉巴拉给你说一堆条件，然后问你输入的数据能不能达成这些条件，如果能的话请输出 YES，不能的话输出 NO。

如果你会做当然好，如果不会做怎么办？

首先这样提交一下：

```
public class Main {
    public static void main(String[] args) {
        System.out.println("YES");
    }
}
```

看下 case 通过率，假设是 60%，那么说明结果为 YES 有 60% 的概率，所以可以这样写代码：

```
public class Main {
    public static void main(String[] args) {
        // 60% 的概率输出 YES，40% 的概率输出 NO
        System.out.println((new Random().nextInt() % 100) < 60 ? "YES" : "NO");
    }
}
```

多提交几次，整出个 80% 以上的 case 通过不是问题。

一、BFS（Breadth First Search，广度优先搜索）

1、地图分析（1162）

1162. 地图分析

难度 中等 124 收藏 分享 切换为英文 关注 反馈

你现在手里有一份大小为 N x N 的「地图」（网格） grid，上面的每个「区域」（单元格）都用 0 和 1 标记好了。其中 0 代表海洋，1 代表陆地，请你找出一个海洋区域，这个海洋区域到离它最近的陆地区域的距离是最大的。

我们这里说的距离是「曼哈顿距离」（Manhattan Distance）：(x0, y0) 和 (x1, y1) 这两个区域之间的距离是 |x0 - x1| + |y0 - y1| 。

如果我们的地图上只有陆地或者海洋，请返回 -1。

示例 1：

1	0	1
0	0	0
1	0	1

输入： [[1,0,1],[0,0,0],[1,0,1]]
输出： 2
解释：
海洋区域 (1, 1) 和所有陆地区域之间的距离都达到最大，最大距离为 2。

示例 2：

1	0	0
0	0	0
0	0	0

输入： [[1,0,0],[0,0,0],[0,0,0]]
输出： 4
解释：
海洋区域 (2, 2) 和所有陆地区域之间的距离都达到最大，最大距离为 4。

提示：

1. 1 <= grid.length == grid[0].length <= 100
2. grid[i][j] 不是 0 就是 1

攻略

题目分析

相信对于 Tree 的 BFS 大家都已经轻车熟路了：要把 root 节点先入队，然后再一层一层的无脑遍历就行了。

对于图的 BFS 也是一样滴～ 与 Tree 的 BFS 区别如下：

1、tree 只有 1 个 root，而图可以有多个源点，所以首先需要把多个源点都入队。

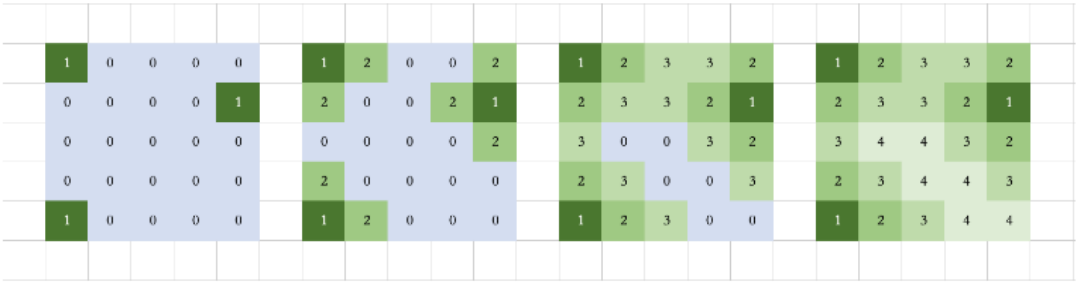
2、tree 是有向的因此不需要标志是否访问过，而对于无向图来说，必须得标志是否访问过！并且为了防止某个节点多次入队，需要在入队之前就将其设置成已访问！

这是一道典型的 BFS 基础应用，为什么这么说呢？

因为我们只要先把所有的陆地都入队，然后从各个陆地同时开始一层一层的向海洋扩散，那么最后扩散到的海洋就是最远的海洋！

并且这个海洋肯定是被离他最近的陆地给扩散到的！

下面是扩散的图示，1 表示陆地，0 表示海洋。每次扩散的时候会标记相邻的 4 个位置的海洋：



你可以想象成你从每个陆地上派了很多支船去踏上伟大航道，踏遍所有的海洋。每当船到了新的海洋，就会分裂成 4 条新的船，向新的未知海洋前进（访问过的海洋就不去了）。如果船到达了某个未访问过的海洋，那他们是第一个到这片海洋的。很明显，这么多船最后访问到的海洋，肯定是离陆地最远的海洋。

链接：

https://leetcode-cn.com/problems/as-far-from-land-as-possible/solution/jian-dan-java-miao-dong-tu-de-bfs-by-sweetiee/

方法 1：比较偏门和省事一些

```
import java.util.*;
class Solution {
    public int maxDistance(int[][] grid) {
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};

        Queue<int[]> queue = new LinkedList<>();
        int m = grid.length, n = grid[0].length;
        // 先把所有的陆地都入队。
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    queue.offer(new int[] {i, j});
                }
            }
        }

        // 从各个陆地开始，一圈一圈的遍历海洋，最后遍历到的海洋就是离陆地最远的海洋。
        boolean hasOcean = false;
        int[] point = null;
        while (!queue.isEmpty()) {
            point = queue.poll();
            int x = point[0], y = point[1];
            // 取出队列的元素，将其四周的海洋入队。
            for (int i = 0; i < 4; i++) {
                int newX = x + dx[i];
                int newY = y + dy[i];
```

```
                if (newX < 0 || newX >= m || newY < 0 || newY >= n || g
rid[newX][newY] != 0) {
                    continue;
                }
                grid[newX][newY] = grid[x][y] + 1; // 这里我直接修改了原
数组，因此就不需要额外的数组来标志是否访问
                hasOcean = true;
                queue.offer(new int[] {newX, newY});
            }
        }

        // 没有陆地或者没有海洋，返回-1。
        if (point == null || !hasOcean) {
            return -1;
        }

        // 返回最后一次遍历到的海洋的距离。
        return grid[point[0]][point[1]] - 1; //因为从陆地开始遍历海洋,陆地
的初始值是 1,而每次将不为 0 的块的上下左右的海洋赋值为该块值+1,所以该最后找到的
块的值减去陆地的值 1 就是距离,可以看上面绿色那个图理解一下.

    }
}
```

另外一种解释：

题意分析：

解读题目中的「最远」和「最近」

题目问的是「距离陆地区域最远的海洋区域」，其实就是从陆地（1）开始，要扩散多少次，才能把所有的海洋给覆盖掉。「最远」应该从这个角度来理解。

而「该海洋区域到离它最近的陆地区域的距离」，「最近」是因为一定是距离这个最后才扩散到的海洋的最近的陆地才能扩散到它。

分析：

二维表格上的问题，常用的算法是深度优先遍历、广度优先遍历和并查集，由于这里计算的结果和距离相关，显然使用广度优先遍历；

但是题目问的是「距离陆地区域最远的海洋区域」，这和我们的经验稍微有点出入。一般而言，「广度优先遍历」求的是最短路径，但仔细一想，发现其实广度优先遍历也是适用的：

求最短路径的时候，只要找到目标值，返回即可；

求最长路径的时候，所有目标值都看完以后，才返回。

这道题「广度优先遍历」的起点有多个，但完全不影响算法的正确性，可以假想一个虚拟的起点，初始的起点就是由虚拟起点扩散开来的，示意图可以参考 题解 ，特别形象。

这里题目中介绍的「曼哈顿距离」，其实就是对广度优先遍历（BFS）逐层向外扩散的精准数学解释，每扩散一次，曼哈顿距离就加 1。

编写广度优先遍历算法的注意事项：

如果题目要求返回的结果和距离相关，需要在 while 循环内部一下子把当前列表的所有元素都依次取出来，这种「一下子」操作的次数就是我们需要的距离；

如果一个单元格被添加到队列以后，需要马上将它标记为已经访问（根据情况，可以直接在原始输入数组上修改，也可以使用一个布尔数组 visited 进行标记），如果不这么做，很可能会出现死循环，这一点如果一开始没有注意到，也可以通过调试代码观察出。

链接：

https://leetcode-cn.com/problems/as-far-from-land-as-possible/solution/yan-du-you-xian-bian-li-java-by-liweiwei1419/

方法 2：比较正统的方法

```
import java.util.*;
class Solution {
    public int maxDistance(int[][] grid) {
        final int [][] directions = {{1, 0}, {-1, 0}, {0,1}, {0,-1}};
        final int N = grid.length;
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (grid[i][j] == 1) {
                    queue.add(getIndex(i,j,N));
                }
            }
        }
        int size = queue.size();
        if (size == 0 || size == N*N) {
            return -1;
        }

        int step = 0;
        while (!queue.isEmpty()){
            int curSize = queue.size();
            for (int i = 0; i < curSize; i++){
                Integer cur = queue.poll();
                int curX = cur / N;
                int curY = cur % N;
                for (int[] d : directions) {
                    int newX = curX + d[0];
                    int newY = curY + d[1];
                    // 只关心有效范围内的海洋（0）
                    if (newX >=0 && newX < N && newY >= 0 && newY < N &
& grid[newX][newY] == 0) {
                        grid[newX][newY] = 1;
                        queue.add(getIndex(newX,newY,N));
                    }
                }
            }
            if(queue.size() > 0) {
```

```
                step++;
            }
        }
         return step;
    }
    // x: 横坐标； y: 纵坐标； cols：二维表格列数
    private int getIndex(int x, int y, int cols) {
        return x * cols + y;
    }
    //自测代码
    public static void main(String[] args) {
        //int[][] grid = {{1, 0, 0}, {0, 0, 0}, {0, 0, 0}};
        int N = 3;
        Scanner sc = new Scanner(System.in);
        int[][] grid = new int[N][N];
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                grid[i][j] = sc.nextInt();
            }
        }
         Solution s = new Solution();
         int res = s.maxDistance(grid);
         System.out.println(res);
    }
}
```

2、克隆图（133）

133. 克隆图

难度 中等 207 收藏 分享 切换为英文 关注 反馈

给你无向 连通 图中一个节点的引用，请你返回该图的 深拷贝（克隆）。

图中的每个节点都包含它的值 `val`（`int`）和其邻居的列表（`list[Node]`）。

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

测试用例格式：

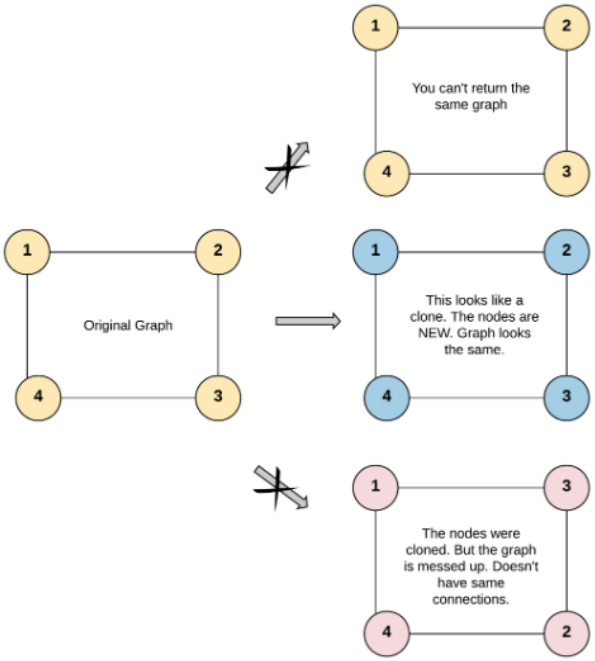
简单起见，每个节点的值都和它的索引相同。例如，第一个节点值为 1（`val = 1`），第二个节点值为 2（`val = 2`），以此类推。该图在测试用例中使用邻接列表表示。

邻接列表 是用于表示有限图的无序列表的集合。每个列表都描述了图中节点的邻居集。

给定节点将始终是图中的第一个节点（值为 1）。你必须将 **给定节点的拷贝** 作为对克隆图的引用返回。

更多硬核内容就在公众号「yes 的练级攻略」

示例 1：



输入： `adjList = [[2,4],[1,3],[2,4],[1,3]]`
输出： `[[2,4],[1,3],[2,4],[1,3]]`
解释：
图中有 4 个节点。
节点 1 的值是 1，它有两个邻居：节点 2 和 4 。
节点 2 的值是 2，它有两个邻居：节点 1 和 3 。
节点 3 的值是 3，它有两个邻居：节点 2 和 4 。
节点 4 的值是 4，它有两个邻居：节点 1 和 3 。

yes的练级攻略

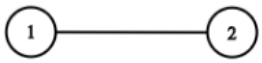
示例 2：

```
输入：adjList = [[]]
输出：[[]]
解释：输入包含一个空列表。该图仅仅只有一个值为 1 的节点，它没有任何邻居。
```

示例 3：

```
输入：adjList = []
输出：[]
解释：这个图是空的，它不含任何节点。
```

示例 4：




```
输入：adjList = [[2],[1]]
输出：[[2],[1]]
```

提示：

1. 节点数不超过 100 。
2. 每个节点值 `Node.val` 都是唯一的，`1 <= Node.val <= 100`。
3. 无向图是一个简单图，这意味着图中没有重复的边，也没有自环。
4. 由于图是无向的，如果节点 *p* 是节点 *q* 的邻居，那么节点 *q* 也必须是节点 *p* 的邻居。
5. 图是连通图，你可以从给定节点访问到所有节点。

yes的练级攻略

https://leetcode-cn.com/problems/clone-graph/solution/ke-long-tu-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：深度优先搜索

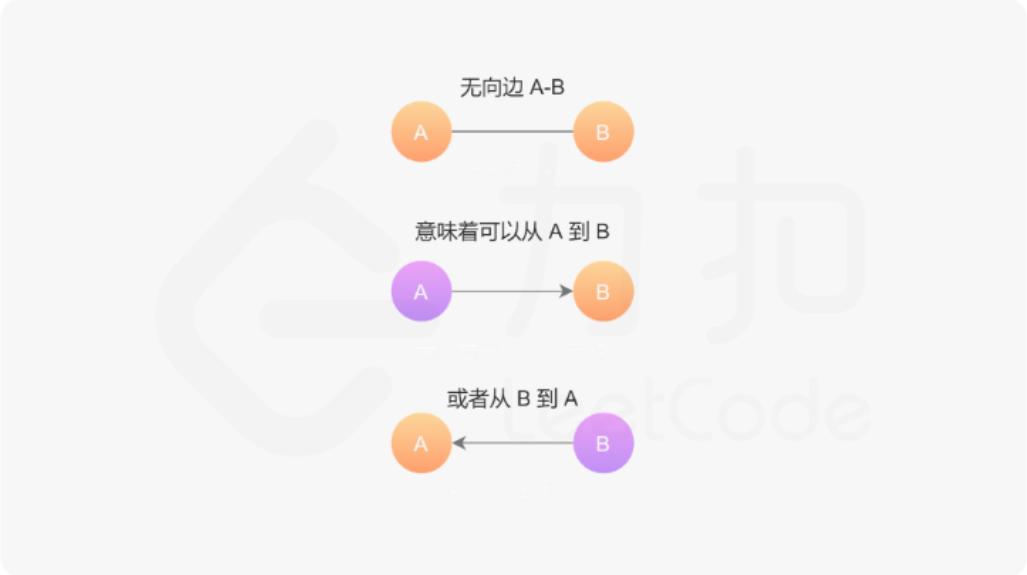
思路

对于本题而言，我们需要明确**图的深拷贝**是在做什么，对于一张图而言，它的**深拷贝**即构建一张与原图结构，值均一样的图，但是其中的节点**不再是原来图节点的引用。**因此，为了深拷贝出整张图，我们需要知道整张图的结构以及对应节点的值。

由于题目只给了我们一个节点的引用，因此为了知道整张图的结构以及对应节点的值，我们需要从给定的节点出发，进行「图的遍历」，并在遍历的过程中完成**图的深拷贝。**

为了避免在深拷贝时陷入死循环，我们需要理解图的结构。对于一张无向图，任何给定的无向边都可以表示为**两个有向边**，即如果节点 `A` 和节点 `B` 之间存在无向边，则表示该图具有从节点 `A` 到节点 `B` 的有向边和从节点 `B` 到节点 `A` 的有向边。

☆ 收藏



为了防止多次遍历同一个节点，陷入死循环，我们需要用一种数据结构记录已经被克隆过的节点。

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

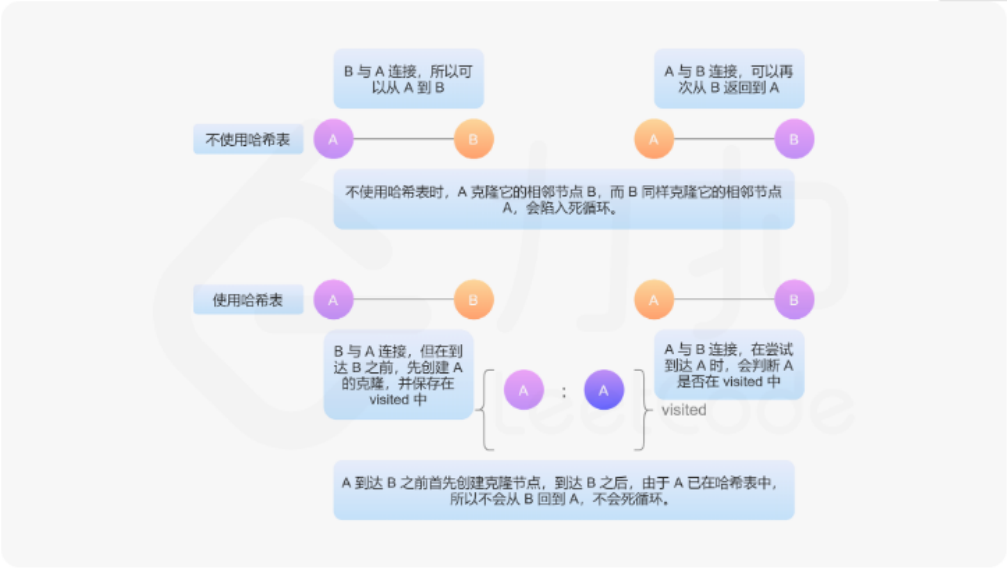
算法

1. 使用一个哈希表存储所有已被访问和克隆的节点。哈希表中的 `key` 是原始图中的节点，`value` 是克隆图中的对应节点。

2. 从给定节点开始遍历图。如果某个节点已经被访问过，则返回其克隆图中的对应节点。

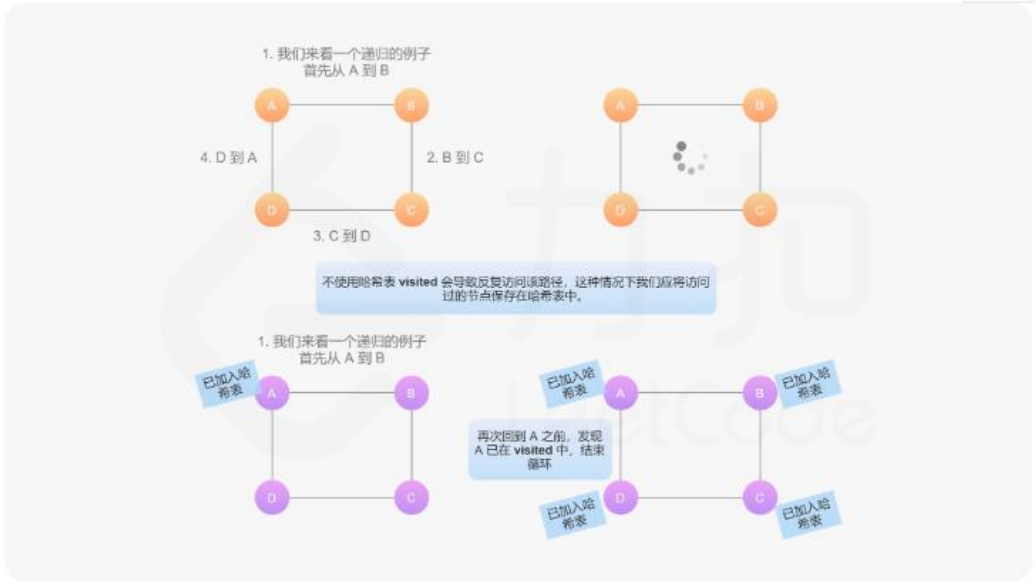
   如下图，我们给定无向边边 `A - B`，表示 `A` 能连接到 `B`，且 `B` 能连接到 `A`。如果不对访问过的节点做标记，则会陷入死循环中。

☆ 收藏



3. 如果当前访问的节点不在哈希表中，则创建它的克隆节点并存储在哈希表中。注意：在进入递归之前，必须先创建克隆节点并保存在哈希表中。如果不保证这种顺序，可能会在递归中再次遇到同一个节点，再次遍历该节点时，陷入死循环。

☆ 收藏



4. 递归调用每个节点的邻接点。每个节点递归调用的次数等于邻接点的数量，每一次调用返回其对应邻接点的克隆节点，最终返回这些克隆邻接点的列表，将其放入对应克隆节点的邻接表中。这样就可以克隆给定的节点和其邻接点。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 表示节点数量。深度优先搜索遍历图的过程中每个节点只会被访问一次。
- 空间复杂度：$O(N)$。存储克隆节点和原节点的哈希表需要 $O(N)$ 的空间，递归调用栈需要 $O(H)$ 的空间，其中 H 是图的深度，经过放缩可以得到 $O(H) = O(N)$，因此总体空间复杂度为 $O(N)$。

方法 1：DFS

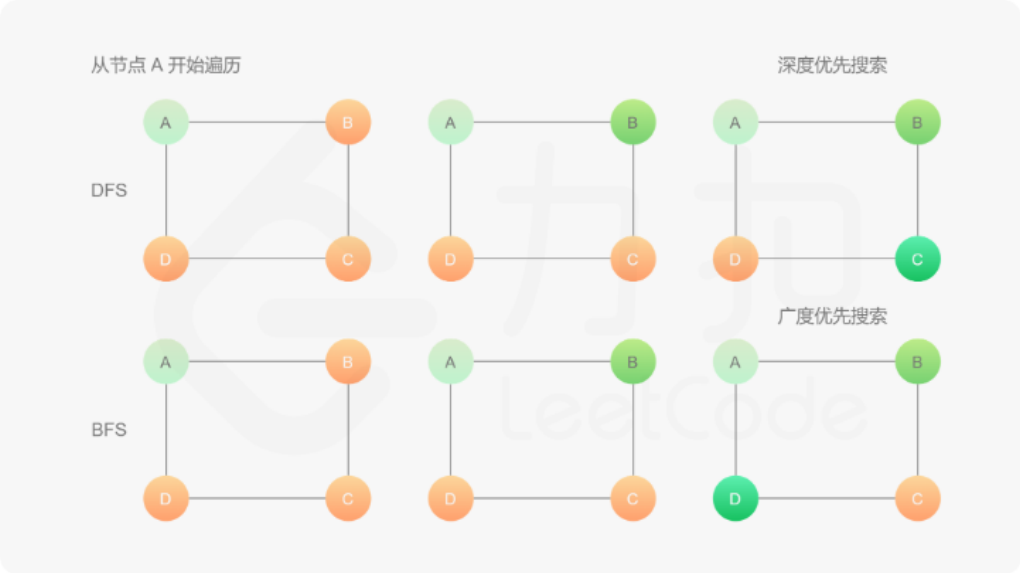
```
class Solution {
    private HashMap <Node, Node> visited = new HashMap <> ();
    public Node cloneGraph(Node node) {
        if (node == null) {
            return node;
        }

        // 如果该节点已经被访问过了，则直接从哈希表中取出对应的克隆节点返回
        if (visited.containsKey(node)) {
            return visited.get(node);
        }

        // 克隆节点，注意到为了深拷贝我们不会克隆它的邻居的列表
        Node cloneNode = new Node(node.val, new ArrayList());
        // 哈希表存储
        visited.put(node, cloneNode);

        // 遍历该节点的邻居并更新克隆节点的邻居列表
        for (Node neighbor: node.neighbors) {
            cloneNode.neighbors.add(cloneGraph(neighbor));
        }
        return cloneNode;
    }
}
```

禁止商用
公众号：
yes的练级攻略

方法二：广度优先遍历

思路

同样，我们也可以用广度优先搜索来进行「图的遍历」。



方法一与方法二的区别仅在于搜索的方式。深度优先搜索以深度优先，广度优先搜索以广度优先。这两种方法都需要借助哈希表记录被克隆过的节点来避免陷入死循环。

算法

1. 使用一个哈希表 `visited` 存储所有已被访问和克隆的节点。哈希表中的 `key` 是原始图中的节点，`value` 是克隆图中的对应节点。
2. 将题目给定的节点添加到队列。克隆该节点并存储到哈希表中。
3. 每次从队列首部取出一个节点，遍历该节点的所有邻接点。如果某个邻接点已被访问，则该邻接点一定在 `visited` 中，那么从 `visited` 获得该邻接点，否则创建一个新的节点存储在 `visited` 中，并将邻接点添加到队列。将克隆的邻接点添加到克隆图对应节点的邻接表中。重复上述操作直到队列为空，则整个图遍历结束。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 表示节点数量。广度优先搜索遍历图的过程中每个节点只会被访问一次。
- 空间复杂度：$O(N)$。哈希表使用 $O(N)$ 的空间。广度优先搜索中的队列在最坏情况下会达到 $O(N)$ 的空间复杂度，因此总体空间复杂度为 $O(N)$。

方法 2：BFS

```
class Solution {
    public Node cloneGraph(Node node) {
        if (node == null) {
            return node;
        }

        HashMap<Node, Node> visited = new HashMap();
```

```
        // 将题目给定的节点添加到队列
        LinkedList<Node> queue = new LinkedList<Node> ();
        queue.add(node);
        // 克隆第一个节点并存储到哈希表中
        visited.put(node, new Node(node.val, new ArrayList()));

        // 广度优先搜索
        while (!queue.isEmpty()) {
            // 取出队列的头节点
            Node n = queue.remove();
            // 遍历该节点的邻居
            for (Node neighbor: n.neighbors) {
                if (!visited.containsKey(neighbor)) {
                    // 如果没有被访问过，就克隆并存储在哈希表中
                    visited.put(neighbor, new Node(neighbor.val, new
ArrayList()));
                    // 将邻居节点加入队列中
                    queue.add(neighbor);
                }
                // 更新当前节点的邻居列表
                visited.get(n).neighbors.add(visited.get(neighbor));
            }
        }

        return visited.get(node);
    }
}
```

3、岛屿数量（200）

yes的练级攻略

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1:

输入:
```
[
['1','1','1','1','0'],
['1','1','0','1','0'],
['1','1','0','0','0'],
['0','0','0','0','0']
]
```
输出: 1

示例 2:

输入:
```
[
['1','1','0','0','0'],
['1','1','0','0','0'],
['0','0','1','0','0'],
['0','0','0','1','1']
]
```
输出: 3
解释: 每座岛屿只能由水平和/或竖直方向上相邻的陆地连接而成。

https://leetcode-cn.com/problems/number-of-islands/solution/number-of-islands-shen-du-you-xian-bian-li-dfs-or-/

思路二：广度优先遍历 BFS

- 主循环和思路一类似，不同点是在于搜索某岛屿边界的方法不同。
- **bfs 方法:**
  - 借用一个队列 `queue`，判断队列首部节点 `(i, j)` 是否未越界且为 `1`：
    - 若是则置零（删除岛屿节点），并将此节点上下左右节点 `(i+1, j)`, `(i-1, j)`, `(i, j+1)`, `(i, j-1)` 加入队列；
    - 若不是则跳过此节点；
  - 循环 `pop` 队列首节点，直到整个队列为空，此时已经遍历完此岛屿。

方法 1：BFS，广度优先遍历

```
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
```

```
                if(grid[i][j] == '1'){
                    bfs(grid, i, j);
                    count++;
                }
            }
        }
        return count;
    }
    private void bfs(char[][] grid, int i, int j){
        Queue<int[]> list = new LinkedList<>();
        list.add(new int[] { i, j });
        while(!list.isEmpty()){
            int[] cur = list.remove();
            i = cur[0]; j = cur[1];
            if(0 <= i && i < grid.length && 0 <= j && j < grid[0].length && gri
d[i][j] == '1') {
                grid[i][j] = '0';
                list.add(new int[] { i + 1, j });
                list.add(new int[] { i - 1, j });
                list.add(new int[] { i, j + 1 });
                list.add(new int[] { i, j - 1 });
            }
        }
    }
}
```

思路一：深度优先遍历DFS

- 目标是找到矩阵中“岛屿的数量”，上下左右相连的 `1` 都被认为是连续岛屿。
- **dfs方法：** 设目前指针指向一个岛屿中的某一点 `(i, j)`，寻找包括此点的岛屿边界。
  - 从 `(i, j)` 向此点的上下左右 `(i+1, j)`，`(i-1, j)`，`(i, j+1)`，`(i, j-1)` 做深度搜索。
  - 终止条件：
    - `(i, j)` 越过矩阵边界；
    - `grid[i][j] == 0`，代表此分支已越过岛屿边界。
  - 搜索岛屿的同时，执行 `grid[i][j] = '0'`，即将岛屿所有节点删除，以免之后重复搜索相同岛屿。
- **主循环：**
  - 遍历整个矩阵，当遇到 `grid[i][j] == '1'` 时，从此点开始做深度优先搜索 `dfs`，岛屿数 `count + 1` 且在深度优先搜索中删除此岛屿。
- 最终返回岛屿数 `count` 即可。

方法 2：DFS

```
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(grid[i][j] == '1'){
```

```
                    dfs(grid, i, j);
                    count++;
                }
            }
        }
        return count;
    }
    private void dfs(char[][] grid, int i, int j){
        if(i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[i]
[j] == '0') return;
        grid[i][j] = '0';
        dfs(grid, i + 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i - 1, j);
        dfs(grid, i, j - 1);
    }
}
```

4、岛屿的最大面积（695）

695. 岛屿的最大面积

难度 中等 341 收藏 分享 切换为英文 关注 反馈

给定一个包含了一些 0 和 1 的非空二维数组 grid 。

一个 岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在水平或者竖直方向上相邻。你可以假设 grid 的四个边缘都被 0 （代表水）包围着。

找到给定的二维数组中最大的岛屿面积。(如果没有岛屿，则返回面积为 0 。)

示例 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,0,0,0,0,0,0,1,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

对于上面这个给定矩阵应返回 6 。注意答案不应该是 11 ，因为岛屿只能包含水平或垂直的四个方向的 1 。

示例 2:

```
[[0,0,0,0,0,0,0,0]]
```

对于上面这个给定的矩阵, 返回 0 。

注意: 给定的矩阵 grid 的长度和宽度都不超过 50。

https://leetcode-cn.com/problems/max-area-of-island/solution/java-bfsyan-du-you-xian-bian-li-xiang-xi-zhu-shi-b/

遍历整个矩阵，当遇到grid[i][j]==1时，从(i,j)向四周扩散,寻找相邻的1，扩散的过程可以用bfs模拟或者dfs模拟，这里采用的是bfs，

扩散过程中使用boolean[][]数组记录已访问过的坐标

方法 1：BFS

```java
class Solution {
    public int maxAreaOfIsland(int[][] grid) {
        if(grid==null || grid.length==0 || grid[0]==null || grid[0].length==0){
            return 0;
        }
        int n = grid.length, m = grid[0].length;
        // 记录已访问过的坐标
        boolean[][] flag = new boolean[n][m];
        // 最终结果，初始值为0
        int max=0;
        //遍历矩阵
        for(int i=0; i<n; i++){
            for(int j=0; j<m; j++){
                //如果当前位置是0或者当前位置访问过，则进入下一轮循环
                if(grid[i][j]==0 || flag[i][j]==true){
                    continue;
                }
                //当前位置是1，并且没有访问过，从这里开始向四周扩散，返回扩散的最大面积
                int cur = bfs(grid, flag, i, j);
                max = Math.max(max, cur);
            }
        }
        return max;
    }
    // 用来表示上下左右四个移动方向
    private int[][] moves = {{-1,0}, {1,0}, {0,-1}, {0,1}};
    // 使用bfs进行扩散，并返回扩散结束后占据的面积
    private int bfs(int[][] grid, boolean[][] flag, int i, int j){
        int n = grid.length, m = grid[0].length;
        // 扩散面积初始值是1，因为grid[i][j]==1
        int res = 1;
        // 辅助bfs的队列
        LinkedList<int[]> queue = new LinkedList<>();
        queue.add(new int[]{i,j});
        flag[i][j] = true;
        // bfs主体循环
        while(!queue.isEmpty()){
            int[] cur = queue.poll();
            // 当前位置进行上下左右移动
            for(int[] move : moves){
                int x = cur[0]+move[0];
                int y = cur[1]+move[1];
```

```
                // 如果移动后的坐标不越界并且移动后的坐标对应的值为1，同时还没有访问过，说明该位置有效
                if(x>=0 && x<n && y>=0 && y<m && grid[x][y]==1 && flag[x][y]==false){
                    queue.add(new int[]{x, y});
                    flag[x][y] = true;
                    //扩散面积加1
                    res++;
                }
            }
        }
        return res;
    }
}
```

https://leetcode-cn.com/problems/max-area-of-island/solution/biao-zhun-javadong-tai-gui-hua-jie-fa-100-by-mark-/

方法 2：DFS

```
class Solution {
    public int maxAreaOfIsland(int[][] grid) {
        int res = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == 1) {
                    res = Math.max(res, dfs(i, j, grid));
                }
            }
        }
        return res;
    }
    // 每次调用的时候默认 num 为1，进入后判断如果不是岛屿，则直接返回0，就可以避免预防错误的情况。
    // 每次找到岛屿，则直接把找到的岛屿改成0，这是传说中的沉岛思想，就是遇到岛屿就把他和周围的全部沉默。
    // ps：如果能用沉岛思想，那么自然可以用朋友圈思想。有兴趣的朋友可以去尝试。
    private int dfs(int i, int j, int[][] grid) {
        if (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j] == 0) {
            return 0;
        }
        grid[i][j] = 0;
        int num = 1;
        num += dfs(i + 1, j, grid);
        num += dfs(i - 1, j, grid);
        num += dfs(i, j + 1, grid);
```

```
            num += dfs(i, j - 1, grid);
            return num;

        }
    }
```

5、水域大小（面试题 16.19）

面试题 16.19. 水域大小

难度 中等 25 收藏 分享 切换为英文 关注 反馈

你有一个用于表示一片土地的整数矩阵 land，该矩阵中每个点的值代表对应地点的海拔高度。若值为0则表示水域。由垂直、水平或对角连接的水域为池塘。池塘的大小是指相连接的水域的个数。编写一个方法来计算矩阵中所有池塘的大小，返回值需要从小到大排序。

示例：

输入：
```
[
  [0,2,1,0],
  [0,1,0,1],
  [1,1,0,1],
  [0,1,0,1]
]
```
输出： [1,2,4]

提示：

- 0 < len(land) <= 1000
- 0 < len(land[i]) <= 1000

https://leetcode-cn.com/problems/pond-sizes-lcci/solution/ba-fang-shui-yu-nang-zhong-zhi-wu-bfsdfsbing-cha-j/

方法 1：DFS（12ms）

```
class Solution {
    int [][] dirs = new int[][] {
            {1,0},{-1,0},{0,1},{0,-1},{1,1},{1,-1},{-1,1},{-1,-1}
    };
    int area = 0;
    public int[] pondSizes(int[][] land) {
        List<Integer> res = new ArrayList<>();
        int m = land.length;
        int n = land[0].length;
        for(int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                area = 0;
                dfs(land, i, j);
                if(area > 0) {
                    res.add(area);
                }
            }
        }
```

```
        int [] ans = new int[res.size()];
        for(int i = 0; i < ans.length; i++) {
            ans[i] = res.get(i);
        }
        Arrays.sort(ans);
        return ans;
    }

    void dfs(int[][] land, int i, int j) {
        if(land[i][j] != 0) {
            return ;
        }
        // 标记已经访问过的水域为陆地(-1 特殊化)
        land[i][j] = -1;
        area ++;
        for(int [] dir : dirs) {
            int nX = i + dir[0];
            int nY = j + dir[1];
            if(inArea(land, nX, nY) && land[nX][nY] == 0) {
                dfs(land, nX, nY);
            }
        }
    }
    boolean inArea(int[][] land, int x, int y) {
        return x >= 0 && y >= 0 && x < land.length && y < land[0].lengt
h;
    }
}
```

方法 2：BFS（20ms）

```
import java.util.*;
class Solution {
    int[][] dirs = new int[][]{
            {1, 0}, {-1, 0}, {0, 1}, {0, -1}, {1, 1}, {1, -1}, {-1, 1},
 {-1, -1}
    };
    public int[] pondSizes(int[][] land) {
        List<Integer> res = new ArrayList<>();
        int m = land.length;
        int n = land[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (land[i][j] == 0) {
```

```
                    Queue<Map.Entry<Integer, Integer>> queue = new Link
edList<>();
                    queue.offer(new AbstractMap.SimpleEntry<>(i, j));
                    // 已经访问过的水域 将其标记为陆地(-1 是为了特殊化)
                    land[i][j] = -1;
                    int count = 1;
                    // BFS 计算水域大小
                    while (!queue.isEmpty()) {
                        Map.Entry<Integer, Integer> curr = queue.poll()
;
                        for (int[] dir : dirs) {
                            int newX = curr.getKey() + dir[0];
                            int newY = curr.getValue() + dir[1];
                            if (inArea(land, newX, newY) && land[newX][
newY] == 0) {
                                // 已经访问过的水域 将其标记为陆地(-1 是为
了特殊化)
                                land[newX][newY] = -1;
                                count ++;
                                queue.offer(new AbstractMap.SimpleEntry
<>(newX, newY));
                            }
                        }
                    }
                    res.add(count);
                }
            }
        }
        int[] ans = new int[res.size()];
        for (int i = 0; i < ans.length; i++) {
            ans[i] = res.get(i);
        }
        Arrays.sort(ans);
        return ans;
    }
    boolean inArea(int[][] land, int x, int y) {
        return x >= 0 && y >= 0 && x < land.length && y < land[0].lengt
h;
    }
}
```

方法 3：并查集（20ms）

```
import java.util.*;
class Solution {
    // 四个方向就够了
```

```
    int[][] dirs = new int[][]{
            {0, 1}, {1, 0}, {1, -1}, {1, 1}
             // {1, 0}, {-1, 0}, {0, 1}, {0, -1}, {1, 1}, {1, -1}, {-1,
1}, {-1, -1}
    };
    int area = 0;

    public int[] pondSizes(int[][] land) {
        List<Integer> res = new ArrayList<>();
        int m = land.length;
        int n = land[0].length;
        int nodeCount = m * n;
        UnionFind uf = new UnionFind(nodeCount);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // 从单点水域向八面扩散搜索 同为水域者收入麾下
                if (land[i][j] == 0) {
                    for (int[] dir : dirs) {
                        int x2 = i + dir[0];
                        int y2 = j + dir[1];
                        // 发现新水域 收入麾下 野蛮。。。
                        if(inArea(land, x2, y2) && land[x2][y2] == 0) {
                            uf.union(i*n+j, x2*n+y2);
                        }
                    }
                }
            }
        }

        for(int i = 0; i < uf.parent.length; i++) {
            // 统计各首领的领"水"面积
            // 满足两个条件: 1. 各个分支的首领 2. 该首领是水域
            if(uf.parent[i] == i && land[i/n][i%n] == 0) {
                res.add(uf.size[i]);
            }
        }

        int[] ans = new int[res.size()];
        for (int i = 0; i < ans.length; i++) {
            ans[i] = res.get(i);
        }

        Arrays.sort(ans);
        return ans;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    }

    boolean inArea(int[][] land, int x, int y) {
        return x >= 0 && y >= 0 && x < land.length && y < land[0].lengt
h;
    }

    class UnionFind {
        public int count;
        public int[] parent;
        public int[] size;

        public UnionFind(int amount) {
            this.count = amount;
            parent = new int[count];
            size = new int[count];
            for (int i = 0; i < count; i++) {
                parent[i] = i;
            }
            Arrays.fill(size, 1);
            // 让所有水域都以此为首
            // size[count-1] = count;
        }

        public int find(int x) {
            while (parent[x] != x) {
                parent[x] = parent[parent[x]];
                x = parent[x];
            }
            return x;
        }

        public void union(int p, int q) {
            int pRoot = find(p);
            int qRoot = find(q);
            if (pRoot == qRoot) {
                return;
            }
            // 大树好乘凉
            if (size[pRoot] > size[qRoot]) {
                parent[qRoot] = pRoot;
                size[pRoot] += size[qRoot];
            } else {
                parent[pRoot] = qRoot;
```

```
                size[qRoot] += size[pRoot];
            }
            count--;
        }
    }
}
```

6、被围绕的区域（130）

130. 被围绕的区域

难度 中等 268 收藏 分享 切换为英文 关注 反馈

给定一个二维的矩阵，包含 'X' 和 'O'（字母 O）。

找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例:

```
X X X X
X O O X
X X O X
X O X X
```

运行你的函数后，矩阵变为：

```
X X X X
X X X X
X X X X
X O X X
```

解释:

被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

方法 1：DFS

https://leetcode-cn.com/problems/surrounded-regions/solution/dfs-bfs-bing-cha-ji-by-powcai/

```
class Solution {
    private static final int[][] dire = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    private static int row, col;
    private static char[][] board;
    public void solve(char[][] board) {
        if (board == null || board.length == 0 || board[0] == null || board[0].length == 0) return;
        row = board.length;
        col = board[0].length;
        this.board = board;
        for (int j = 0; j < col; j++) {
            // 第一行
            if (board[0][j] == 'O') dfs(0, j);
            // 最后一行
            if (board[row - 1][j] == 'O') dfs(row - 1, j);
        }
```

```
        for (int i = 0; i < row; i++) {
            // 第一列
            if (board[i][0] == 'O') dfs(i, 0);
            // 最后一列
            if (board[i][col - 1] == 'O') dfs(i, col - 1);
        }

        // 转变
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (board[i][j] == 'O') board[i][j] = 'X';
                if (board[i][j] == 'B') board[i][j] = 'O';
            }
        }

    }

    private void dfs(int i, int j) {
        board[i][j] = 'B';
        for (int[] d : dire) {
            int newX = d[0] + i;
            int newY = d[1] + j;
            if (newX < 0 || newX >= row || newY < 0 || newY >= col || b
oard[newX][newY] != 'O') continue;
            dfs(newX, newY);
        }
    }
}
```

方法 2：BFS

https://leetcode-cn.com/problems/surrounded-regions/solution/zhu-jie-xie-de-hen-xiang-xi-fang-bian-xiao-bai-tui/

```
class Solution {
    public void solve(char[][] board) {
        int row = board.length;
        if (row == 0 ) {//行是 0 再拿列会报错
            return;
        }
        int col = board[0].length;
        Queue<int []> q = new LinkedList<>();//存取之后拿到的 .
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                //下标为第 0 列、行和下标最后一列、行之外的都不做操作 跳过
                if ((i > 0 && i < row - 1) && (j > 0 && j <col - 1))
                    continue;
```

```
                //边界为O 的加入队列，并且把边界设置为.
                if (board[i][j] == 'O'){
                    q.add(new int[]{i, j});
                    board[i][j] = '.';
                }
            }
        }

        while (!q.isEmpty()){
            //size 每次进来都不一样 先在外面定义 size
            int sz = q.size();
            for (int i = 0; i < sz; i++) {
                int [] cur = q.poll();
                int r = cur[0];//取出行的值
                int c = cur[1];//取出列的值
               if (r > 0 && board[r - 1][c] == 'O') {//查找'.'的上边
                    board[r - 1][c] = '.';
                    q.add(new int[]{r - 1,c});//找到了，把当前位置也要加入
队列
                }

                if (r + 1 < row && board[r + 1][c] == 'O') {//找下边
                    board[r + 1][c] = '.';
                    q.add(new int[]{r + 1,c});
                }

                if (c > 0 && board[r][c - 1] == 'O') {//左边
                    board[r][c - 1] = '.';
                    q.add(new int[]{r,c - 1});
            }

                if (c + 1 < col && board[r][c + 1] == 'O') {//右边
                    board[r][c + 1] = '.';
                    q.add(new int[]{r,c + 1});
                }
            }
        }
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
               board[i][j] = board[i][j]=='.'?'O':'X';//是.就把他设置为
O 不是就是设置为X
            }
        }
```

```
    }
}
```

7、腐烂的橘子（994）

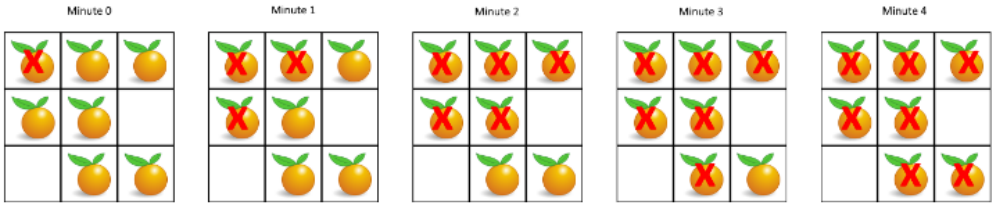
在给定的网格中，每个单元格可以有以下三个值之一：

- 值 0 代表空单元格；
- 值 1 代表新鲜橘子；
- 值 2 代表腐烂的橘子。

每分钟，任何与腐烂的橘子（在 4 个正方向上）相邻的新鲜橘子都会腐烂。

返回直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 -1 。

示例 1：



输入： [[2,1,1],[1,1,0],[0,1,1]]
输出： 4

示例 2：

输入： [[2,1,1],[0,1,1],[1,0,1]]
输出： -1
解释： 左下角的橘子（第 2 行， 第 0 列）永远不会腐烂，因为腐烂只会发生在 4 个正向上。

示例 3：

输入： [[0,2]]
输出： 0
解释： 因为 0 分钟时已经没有新鲜橘子了，所以答案就是 0 。

思路分析：

首先，腐烂橘子会感染上下左右四个方向的橘子，如果我们将腐烂橘子看成一个结点，那么其上下左右四个方向的橘子腐烂后也成为一个结点，他们都是相连接的。所以腐烂橘子其实构成了一个图，这个图中有一个或者多个连通子图。

由于橘子一分钟可以将上下左右的橘子腐烂，这个过程就像无向图中的搜索过程，要求橘子的腐烂完全的最短分钟数，就等价于在无向图中找到达到各个结点的最短路径中的最大值。所以明确之后就很简单，就是一个 BFS，只不过有这里有多个连通子图。

就像回溯算法题解中很常见的二维平面的搜索。实例变量 int[][] ds 表示上下左右移动的坐标变化量，并且用 private int n, m;分别表示行列数。且写一个防止数组索引越界的判断函数 private boolean isIn(int x, int y)。

接下来，完成我们 BFS 函数的设计。

需要一个保留结点的队列 private Queue<int[]> queue;

由于连通子图的存在，9-13 行将所有初始时候腐烂的橘子 grid[i][j] == 2 的坐标放入队列中。对于某个点的搜索，使用函数 newRot(int[][] grid, int[] pos)，第二个参数表示该点的坐标，将该点上下左右四个点的坐标算出，保证不越界的情况下 isIn(newX, newY)，如果相邻四点有新鲜橘子 grid[newX][newY] == 1。将该橘子感染时间设置为上一个橘子感染时间加 1。然后放入队列，准备用它再去感染别的橘子（用它接着进行广度优先遍历）

感染时间直接使用 grid[][]来记录，规定初始时候感染的橘子为 2 这个时刻。

14-17 行，腐烂橘子队列非空时取出一个腐烂的橘子，将其周围感染，通过 newRot 函数只会感染相邻的。

搜索完成后，再一次遍历，如果有发现未感染的橘子 grid[i][j] == 1，依照题意返回-1。否则记录最晚感染的橘子在第几分钟被感染。

当然最后还需要注意，有一种情况，初始时压根就没有新鲜橘子，这个时候要返回的是 0，所以最后返回值还需要一个判断 return res >= 2 ? res - 2 : 0;

链接：

https://leetcode-cn.com/problems/rotting-oranges/solution/994-java-bfsxiang-jie-by-ustcyyw/

```
class Solution {
    public int orangesRotting(int[][] grid) {
        int [][] directions = {{1, 0}, {-1, 0}, {0,1}, {0,-1}};
        int row = grid.length;
        if (row == 0) {
            return 0;
        }
        int col = grid[0].length;
        Queue<int[]> queue = new LinkedList<>();
        int count = 0;
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (grid[i][j] == 2) {
                    queue.add(new int[] {i,j});
                }
            }
        }
        while (!queue.isEmpty()) {
            int[] cur = queue.poll();
            int curX = cur[0], curY = cur[1];
            for (int[] d : directions) {
                int newX = curX + d[0];
                int newY = curY + d[1];
                if (newX >= 0 && newX < row && newY >= 0 && newY < col
&& grid[newX][newY] == 1) {
                    grid[newX][newY] = grid[curX][curY] + 1;
                    queue.add(new int[] {newX,newY});
                }
            }
        }
        int res = 0;
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (grid[i][j] == 1) {
                    return -1;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                }
                res = Math.max(res,grid[i][j]);
            }
        }
        return res >= 2 ? res-2 : 0;
    }

}
    public static void main(String[] args) {
//      int[][] grid1 = {{2,1,1}, {1,1,0}, {0,1,1}};
        Scanner sc = new Scanner (System.in);
        int[][] grid1 = new int[3][3];
        for (int i = 0; i < grid1.length; i++) {
            for (int j = 0; j < grid1[0].length; j++) {
                grid1[i][j] = sc.nextInt();
            }
        }
        OrangesRotting or = new OrangesRotting();
        int time = or.orangesRotting(grid1);
        System.out.println(time);

    }
```

8、棋子旋转（linkcode1146）

1146. 棋子旋转 中文 English

在4x4的棋盘上摆满了黑白棋子，黑白两色的位置和数目随机其中左上角坐标为(1,1),右下角坐标为(4,4),现在依次有一些翻转操作，
要对一些给定支点坐标为中心的上下左右四个棋子的颜色进行翻转，请计算出翻转后的棋盘颜色。
给定两个数组A和f,分别为初始棋盘和翻转位置。其中翻转位置共有3个。

样例
例1:

输入:
A:[[0,0,1,1],[1,0,1,0],[0,1,1,0],[0,0,1,0]]
F: [[2,2],[3,3],[4,4]]
输出: [[0,1,1,1],[0,0,1,0],[0,1,1,0],[0,0,1,0]]
注意事项
请返回翻转后的棋盘。

```
public class Solution {
    /**
     * @param A: Initial chessboard
     * @param F: Flipped coordinates
     * @return:    return to the flipped board.
     */
    public int[][] ChessPieceRotation(int[][] A, int[][] F) {
        // write your code here
        int[] dx = new int[] {0, 1, 0 , -1};
        int[] dy = new int[] {1, 0, -1 , 0};
        for (int[] item : F) {
```

禁止商用 公众号

```
                int x = item[0] - 1, y = item[1] - 1;
                for (int i = 0; i < 4; i++) {
                    if (x + dx[i] > -1 && x + dx[i] < 4 && y + dy[i] > -1 && y + dy[i] < 4) {
                        A[x + dx[i]][y + dy[i]] = 1 - A[x + dx[i]][y + dy[i]];
                    }
                }
            }
            return A;
        }
}
```

9、丢手绢（lintcode1269）

1269. 丢手绢

有 n 个人正在玩丢手绢。编号为 $0, 1 \ldots n-1$。
一开始，每个人手上都拿着一块自己手绢。接下来的每一轮，第 i 个人会把当前手里的手绢丢给一个固定的人 to_i。
当有人重新拿到自己的手绢时，游戏就会结束。
请求出游戏会在第几轮结束。

样例
样例输入：

```
to = [1, 2, 0, 4, 3, 0]
```

样例输出：

```
2
```

说明
0, 1, 2 号都要经过三轮拿到自己的手绢，但 3, 4 号只要两轮就能拿到自己的手绢，所以 2 轮后游戏结束。

注意事项
$1 <= n <= 10^3$
$0 <= to_i <= n-1$, to_i 可能和 i 相等，即丢给自己。
保证游戏一定会结束。

解题思路

我们可以把 `N` 个人抽象成图的节点，`N` 个丢手绢的方向抽象成边，所以就形成了一个 `N` 个节点 `N` 条边的有向图，这道题就成为了有向图上的最小环问题。我们可以用宽度优先搜索来解决。

但是这个图有一个特殊的性质，就是所有节点的出度为 `1`。也就是说在宽度优先搜索时，队列里只有 `1` 个节点。我们直接模拟每一块手绢在游戏中的轨迹即可，当它的位置回到出发点的话，记录下来最少的轮数。

代码思路

计算某个手绢 `i` 回到出发点的轮数：

1. 令 `position` 等于 `i`，`turnCount` 等于 `0`。
2. `position` 跳到下一个位置，`turnCount` 加上 `1`。
3. 如果 `position` 等于 `i`，即回到出发点的话，返回 `turnCount` 并统计。
4. 如果 `position` 不等于 `i`：
    - 如果已经访问过当前位置，代表进入死循环，回不到起点。退出。
    - 标记 `position` 已访问过，返回第 `2` 步。

复杂度分析

设游戏人数为 `N`。

时间复杂度

- 每个节点最多跳 `N` 步，时间复杂度为 `O(N^2)`。

空间复杂度

- 需要 `O(N)` 的额外空间记录已访问过的节点，空间复杂度为 `O(N)`。

```
public class Solution {
    /**
```

```
     * @param to: The target of everyone will throw the handkerchief to.
     * @return: Return the number of turns the game will stop.
     */
    public int gameTurns(List<Integer> to) {
        int minTurn = to.size();
        for (int i = 0; i < to.size(); i++){
            int position = i;
            int turnCount = 0;
            Set<Integer> visited = new HashSet<>();
            position = to.get(position);
            turnCount++;
            while (position != i) {
                position = to.get(position);
                turnCount++;
                // 如果到了已访问过的点，代表这个手绢进入死循环，到不了出发点
                if (visited.contains(position)) {
                    break;
                }
                visited.add(position);
            }
            if (position == i) {
                minTurn = Math.min(minTurn, turnCount);
            }
        }
        return minTurn;
    }
}
```

10、接雨水 2（407）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

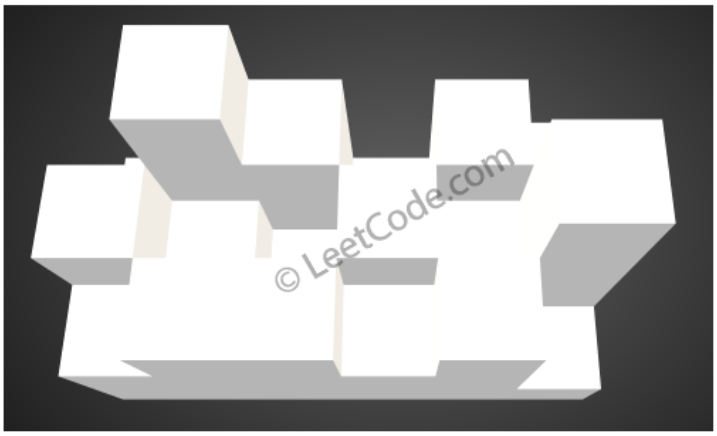
407. 接雨水 II

难度 困难 213 收藏 分享 切换为英文 关注 反馈

给你一个 `m x n` 的矩阵，其中的值均为非负整数，代表二维高度图每个单元的高度，请计算图中形状最多能接多少体积的雨水。

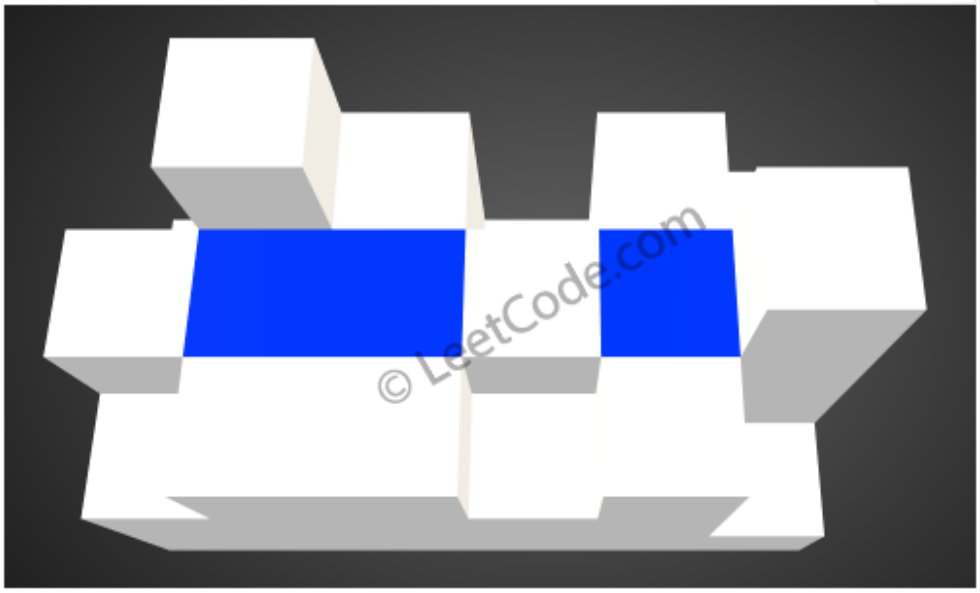
示例：

```
给出如下 3x6 的高度图：
[
  [1,4,3,1,3,2],
  [3,2,1,3,2,4],
  [2,3,3,2,3,1]
]

返回 4 。
```



如上图所示，这是下雨前的高度图 `[[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]]` 的状态。

yes的练级攻略



下雨后，雨水将会被存储在这些方块中。总的接雨水量是4。

提示：

- `1 <= m, n <= 110`
- `0 <= heightMap[i][j] <= 20000`

https://leetcode-cn.com/problems/trapping-rain-water-ii/solution/you-xian-dui-lie-de-si-lu-jie-jue-jie-yu-shui-ii-b/

解题思路

前言：可能不是最好的思路，如果有好的思路麻烦大佬们艾特我学习下！谢谢了！

接雨水I中，我们维护了左右两个最高的墙，那么在这里，就是维护周围一个圈，用堆来维护周围这一圈中的最小元素。为什么是维护最小的元素不是最大的元素呢，因为木桶原理呀。这个最小的元素从堆里弹出来，和它四个方向的元素去比较大小，看能不能往里灌水，怎么灌水呢，如果用方向就比较复杂了，我们可以用visited数组来表示哪些遍历过，哪些没遍历过。如果当前弹出来的高度比它周围的大，他就能往矮的里面灌水了，灌水后要把下一个柱子放进去的时候，放的高度要取两者较大的，也就是灌水后的高度，不是它原来矮的时候的高度了，如果不能灌水，继续走。

```
Given the following 3x6 height map:
[
  [1, 4, 3, 1, 3, 2],
  [3, 2, 1, 3, 2, 4],
  [2, 3, 3, 2, 3, 1]
]
```

就拿这个例子来说，我们先把第一圈都放进去，然后开始从堆中弹出，第一圈，最小值是1（遍历时候标记为访问过），1从堆里弹出来，比如弹出来1(坐标[0,3])，它下方的3没有被访问过，尝试灌水，发现不能灌水，3入堆，然后继续弹。比如说，我此时弹出来一个3（坐标[1,0]），它能向右边2(坐标[1,1])灌水，那这边就可以统计了，然后我们要插入2(坐标[1,1])这个位置，但是插入的时候，要记得你得是插入的高度得是灌水后的高度，而不是原来的高度了

具体代码（用优先队列+bfs）

```
class Solution {
    public int trapRainWater(int[][] heights) {
```

```
        if (heights == null || heights.length == 0 || heights[0].length
== 0) return 0;
        int m = heights.length;
        int n = heights[0].length;
        // 用一个 vis 数组来标记这个位置有没有被访问过
        boolean[][] vis = new boolean[m][n];
        // 优先队列中存放三元组 [x,y,h] 坐标和高度
     PriorityQueue<int[]> pq = new PriorityQueue<>((o1, o2) -> o1[2] -
o2[2]);

        // 先把最外一圈放进去
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i == 0 || i == m - 1 || j == 0 || j == n - 1) {
                    pq.offer(new int[]{i, j, heights[i][j]});
                    vis[i][j] = true;
                }
            }
        }
        int res = 0;
        // 方向数组，把 dx 和 dy 压缩成一维来做
        int[][] dir = {{1, 0}, {-1,0}, {0, 1}, {0, -1}};
        while (!pq.isEmpty()) {
            int[] poll = pq.poll();
            // 看一下周围四个方向，没访问过的话能不能往里灌水
            for (int[] d : dir) {
                int newX = poll[0] + d[0];
                int newY = poll[1] + d[1];
                // 如果位置合法且没访问过
                if (newX >= 0 && newX < m && newY >= 0 && newY < n && !
vis[newX][newY]) {
                    // 如果外围这一圈中最小的比当前这个还高，那就说明能往里面
灌水啊
                    if (poll[2] > heights[newX][newY]) {
                        res += poll[2] - heights[newX][newY];
                    }
                    // 如果灌水高度得是你灌水后的高度了，如果没灌水也要取高的
                    pq.offer(new int[]{newX, newY, Math.max(heights[new
X][newY], poll[2])});
                    vis[newX][newY] = true;
                }
            }
        }
        return res;
```

```
    }
}
```

11、完全平方数（279）

279. 完全平方数

难度 中等 513 收藏 分享 切换为英文 关注 反馈

给定正整数 *n*，找到若干个完全平方数（比如 `1, 4, 9, 16, ...`）使得它们的和等于 *n*。你需要让组成和的完全平方数的个数最少。

示例 1:

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
```

示例 2:

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9.
```

https://leetcode-cn.com/problems/perfect-squares/solution/wan-quan-ping-fang-shu-by-leetcode/

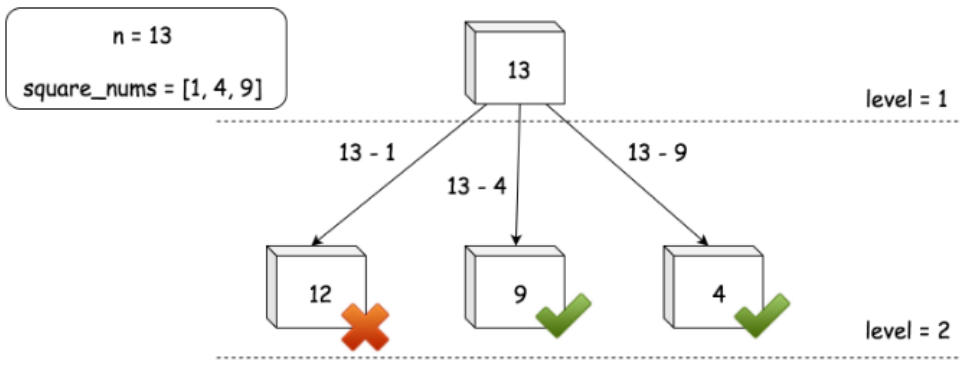
方法四：贪心 + BFS（广度优先搜索）

正如上述贪心算法的复杂性分析种提到的，调用堆栈的轨迹形成一颗 N 元树，其中每个结点代表 `is_divided_by(n, count)` 函数的调用。基于上述想法，我们可以把原来的问题重新表述如下：

给定一个 N 元树，其中每个节点表示数字 n 的余数减去一个完全平方数的组合，我们的任务是在树中找到一个节点，该节点满足两个条件：

(1) 节点的值（即余数）也是一个完全平方数。
(2) 在满足条件（1）的所有节点中，节点和根之间的距离应该最小。

下面是这棵树的样子。



在前面的方法3中，由于我们执行调用的贪心策略，我们实际上是从上到下逐层构造 N 元树。我们以 BFS（广度优先搜索）的方式遍历它。在 N 元树的每一级，我们都在枚举相同大小的组合。

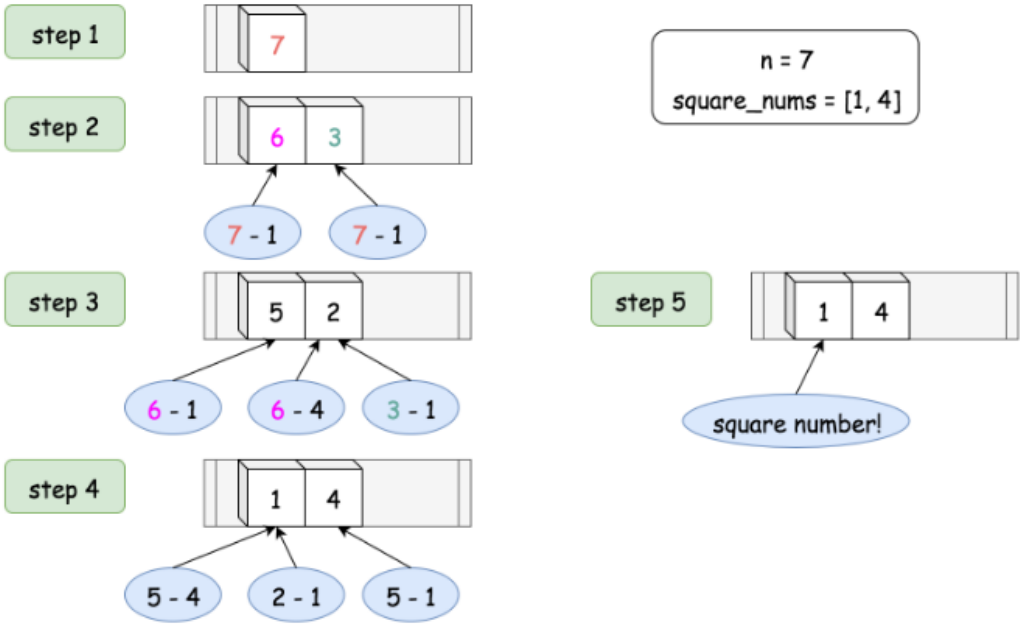
遍历的顺序是 BFS，而不是 DFS（深度优先搜索），这是因为在用尽固定数量的完全平方数分解数字 n 的所有可能性之前，我们不会探索任何需要更多元素的潜在组合。

算法：

- 首先，我们准备小于给定数字 `n` 的完全平方数列表（即 `square_nums`）。
- 然后创建 `queue` 遍历，该变量将保存所有剩余项在每个级别的枚举。
- 在主循环中，我们迭代 `queue` 变量。在每次迭代中，我们检查余数是否是一个完全平方数。如果余数不是一个完全平方数，就用其中一个完全平方数减去它，得到一个新余数，然后将新余数添加到 `next_queue` 中，以进行下一级的迭代。一旦遇到一个完全平方数的余数，我们就会跳出循环，这也意味着我们找到了解。

注意：在典型的 BFS 算法中，`queue` 变量通常是数组或列表类型。但是，这里我们使用 `set` 类型，以消除同一级别中的剩余项的冗余。事实证明，这个小技巧甚至可以增加 5 倍的运行加速。

在下图中，我们以 `numSquares(7)` 为例说明队列的布局。



方法 1：贪心+广搜

```
class Solution {
    public int numSquares(int n) {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        for (int i = 1; i * i <= n; ++i) {
        nums.add(i * i);
        }
        Set<Integer> que = new HashSet<Integer>();
        que.add(n);

        int level = 0;
        while (que.size() > 0) {
        level += 1;
        Set<Integer> next = new HashSet<Integer>();
```

```
            for (Integer re : que) {
                for (Integer sq : nums) {
                if (re.equals(sq)) {
                    return level;
                } else if (re < sq) {
                    break;
                } else {
                    next.add(re - sq);
                }
                }
            }
            que = next;
            }
            return level;
        }
}
```

随着时间的推移，已经提出并证明的数学定理可以解决这个问题。在这一节中，我们将把这个问题分成几个例子。

1770 年，Joseph Louis Lagrange证明了一个定理，称为四平方和定理，也称为 Bachet 猜想，它指出每个自然数都可以表示为四个整数平方和：

$$p = a_0^2 + a_1^2 + a_2^2 + a_3^2$$

其中 a_0, a_1, a_2, a_3 表示整数。

例如，3，31 可以被表示为四平方和如下：

$$3 = 1^2 + 1^2 + 1^2 + 0^2 \qquad 31 = 5^2 + 2^2 + 1^2 + 1^2$$

情况 1：拉格朗日四平方定理设置了问题结果的上界，即如果数 `n` 不能分解为较少的完全平方数，则至少可以分解为 4个完全平方数之和，即 $\text{numSquares}(n) \leq 4$。

正如我们在上面的例子中可能注意到的，数字 0 也被认为是一个完全平方数，因此我们可以认为数字 3 可以分解为 3 个或 4 个完全平方数。

然而，拉格朗日四平方定理并没有直接告诉我们用最小平方数来分解自然数。

后来，在 1797 年，Adrien Marie Legendre用他的三平方定理完成了四平方定理，证明了正整数可以表示为三个平方和的一个特殊条件：

$$n \neq 4^k(8m + 7) \iff n = a_0^2 + a_1^2 + a_2^2$$

其中 k 和 m 是整数。

情况 2：与四平方定理不同，Adrien-Marie-Legendre 的三平方定理给了我们一个充分必要的条件来检验这个数是否只能分解成 4 个平方。

更多硬核内容就在公众号「yes 的练级攻略」

从三平方定理看我们在第 2 种情况下得出的结论可能很难。让我们详细说明一下推论过程。

首先，三平方定理告诉我们，如果 `n` 的形式是 $n = 4^k(8m + 7)$，那么 `n` 不能分解为 3 个平方的和。此外，我们还可以断言 `n` 不能分解为两个平方和，数本身也不是完全平方数。因为假设数 `n` 可以分解为 $n = a_0^2 + a_1^2$，然后通过在表达式中添加平方数 0，即 $n = a_0^2 + a_1^2 + 0^2$，我们得到了数 `n` 可以分解为 3 个平方的结论，这与三平方定理相矛盾。因此，结合四平方定理，我们可以断言，如果这个数不满足三平方定理的条件，它只能分解成四个平方和。

如果这个数满足三平方定理的条件，则可以分解成三个完全平方数。但我们不知道的是，如果这个数可以分解成更少的完全平方数，即一个或两个完全平方数。

所以在我们把这个数视为底部情况（三平方定理）之前，还有两种情况需要检查，即：

情况 3.1：如果数字本身是一个完全平方数，这很容易检查，例如 `n == int(sqrt(n)) ^ 2`。

情况 3.2：如果这个数可以分解成两个完全平方数和。不幸的是，没有任何数学定理可以帮助我们检查这个情况。我们需要使用枚举方法。

算法：

可以按照上面的例子来实现解决方案。

- 首先，我们检查数字 `n` 的形式是否为 $n = 4^k(8m + 7)$，如果是，则直接返回 4。
- 否则，我们进一步检查这个数本身是否是一个完全平方数，或者这个数是否可以分解为两个完全平方数和。
- 在底部的情况下，这个数可以分解为 3 个平方和，但我们也可以根据四平方定理，通过加零，把它分解为 4 个平方。但是我们被要求找出最小的平方数。

复杂度分析

- 时间复杂度：$\mathcal{O}(\sqrt{n})$，在主循环中，我们检查数字是否可以分解为两个平方和，这需要 $\mathcal{O}(\sqrt{n})$ 个迭代。在其他情况下，我们会在常数时间内进行检查。
- 空间复杂度：$\mathcal{O}(1)$，该算法消耗一个常量空间。

方法 2：数学运算（最优）

```
class Solution {

  protected boolean isSquare(int n) {
    int sq = (int) Math.sqrt(n);
    return n == sq * sq;
  }

  public int numSquares(int n) {
    // four-square and three-square theorems.
    //(n & 3)与n % 4 等效
    while ((n & 3) == 0)
        //n /= 4 与 n >>= 2 等效
        n >>= 2;
    //(n & 7)与n % 8 等效
    if ((n & 7) == 7)
      return 4;

    if (this.isSquare(n))
      return 1;
    // enumeration to check if the number can be decomposed into sum of
 two squares.
```

```
    for (int i = 1; i * i <= n; ++i) {
      if (this.isSquare(n - i * i))
        return 2;
    }
    // bottom case of three-square theorem.
    return 3;
  }
}
```

12、节点间通路（面试题 04.01）【邻接表】

面试题 04.01. 节点间通路

难度 中等 14 收藏 分享 切换为英文 关注 反馈

节点间通路。给定有向图，设计一个算法，找出两个节点之间是否存在一条路径。

示例1:

```
 输入: n = 3, graph = [[0, 1], [0, 2], [1, 2], [1, 2]], start = 0, target = 2
 输出: true
```

示例2:

```
 输入: n = 5, graph = [[0, 1], [0, 2], [0, 4], [0, 4], [0, 1], [1, 3], [1, 4], [1, 3], [2, 3], [3, 4]],
start = 0, target = 4
 输出 true
```

提示:

1. 节点数量n在[0, 1e5]范围内。
2. 节点编号大于等于 0 小于 n。
3. 图中可能存在自环和平行边。

https://leetcode-cn.com/problems/route-between-nodes-lcci/solution/javalin-jie-biao-yan-du-bian-li-by-wonderzlf/

https://www.cnblogs.com/xuqiang/archive/2011/03/28/1997680.html

yes的练级攻略

主要思路是先转邻接表，然后遍历方法用深度和广度应该都可以。

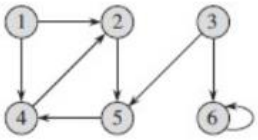
1、数据结构

理论上，首先要理解图，其次理解有向图。

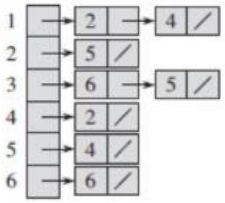
题中graph, [0,1]这种数组表示一条从0到1的有向边，多个有向边连成整个有向图。

数据结构上，有向图使用邻接表存储的。所以这里把有向图转为链表处理。

有向图的邻接表和邻接矩阵表示如下所示：



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

参考：
https://www.cnblogs.com/xuqiang/archive/2011/03/28/1997680.html

2、算法设计

先将graph转为邻接表存储，依次遍历每个链表，对start能到达的节点依次遍历，如果节点是target直接返回true.

2.1 例子：

int[][] graph=new int [][] {{0,1},{1,2},{1,2}};

2.2 转为邻接表：

0->1

1->2->2

2.3 广度遍历：

1）维护一个队列：从0的链表访问，访问过程中0能到达的节点继续入栈

2）维护一个数组：对已经入列的节点，不再重复处理

3）不断从队列中取节点得到新的可访问链表，对链表上的可访问节点中判断是否有target

4）遍历过程中有target则直接返回true, 否则返回false

```
class Solution {
    public boolean findWhetherExistsPath(int n, int[][] graph, int start, int t
arget) {
        // 将矩阵转为邻接表
        List<Integer>[] adj= new ArrayList [n];
        for (int[] edge:graph){
            int from=edge[0];
```

```
            int to=edge [1];
            if (adj[from]==null)
                adj[from]=new ArrayList<>();
            adj[from].add(to);
        }
        // 建一个函数进行广度遍历
        return hasPath(n,adj,start,target);
    }
    private boolean hasPath(int n, List<Integer>[] adj, int start, int target){
        // 维护一个队列：从 0 的链表访问，访问过程中 0 能到达的节点继续入列
        LinkedList<Integer> queue = new LinkedList<> ();
        queue.offer(start);
        // 维护一个数组：对已经入列的节点，不再重复处理
        boolean[] visited = new boolean [n];
        visited[start]=true;
        while(!queue.isEmpty()){
            int size=queue.size();
            // 不断从队列中取节点得到新的可访问链表，对链表上的可访问节点中判断是否有
target
            for(int i = 0; i < size; i++){
                int node = queue.poll();
                List<Integer> nextList = adj [node];
                if (nextList==null){
                    continue;
                }
                for (Integer next : nextList){
                    // 遍历过程中有target 则直接返回true, 否则最终返回false
                    if (next==target){
                        return true;
                    }
                    // 已经入列的节点，不再重复处理
                    if (visited[next]){
                        continue;
                    }
                    // 用数组标记新的节点
                    visited [next] =true;
                    // 访问过程中 0 能到达的节点继续入列
                    queue.add(next);
                }
            }
        }
        return false;
    }
}
```

13、最低成本联通所有城市（1135）【并查集】

1135. 最低成本联通所有城市

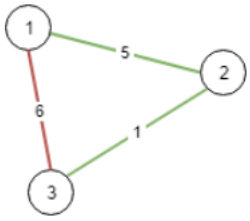
难度 中等 27 收藏 分享 切换为英文 关注 反馈

想象一下你是个城市基建规划者，地图上有 `N` 座城市，它们按以 `1` 到 `N` 的次序编号。

给你一些可连接的选项 `conections`，其中每个选项 `conections[i] = [city1, city2, cost]` 表示将城市 `city1` 和城市 `city2` 连接所要的成本。（**连接是双向的**，也就是说城市 `city1` 和城市 `city2` 相连也同样意味着城市 `city2` 和城市 `city1` 相连）。

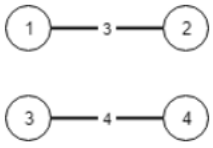
返回使得每对城市间都存在将它们连接在一起的连通路径（可能长度为 1 的）最小成本。该最小成本应该是所用全部连接代价的综合。如果根据已知条件无法完成该项任务，则请你返回 -1。

示例 1：



输入： N = 3, conections = [[1,2,5],[1,3,6],[2,3,1]]
输出： 6
解释：
选出任意 2 条边都可以连接所有城市，我们从中选取成本最小的 2 条。

示例 2：



输入： N = 4, conections = [[1,2,3],[3,4,4]]
输出： -1
解释：
即使连通所有的边，也无法连接所有城市。

提示：

1. `1 <= N <= 10000`
2. `1 <= conections.length <= 10000`
3. `1 <= conections[i][0], conections[i][1] <= N`
4. `0 <= conections[i][2] <= 10^5`
5. `conections[i][0] != conections[i][1]`

https://leetcode-cn.com/problems/connecting-cities-with-minimum-cost/solution/java-kruskal-suan-fa-27ms-by-jzj1993/

更多硬核内容就在公众号「yes 的练级攻略」

Kruskal算法思路

Kruskal算法 = 贪心 + 并查集

流程：将所有边按cost从小到大排序，然后使用并查集依次尝试合并每个边：

- 如果合并成功，则加入这条边。
- 如果合并失败（边的两个节点已经是合并过的），说明产生了环，则丢弃这条边。

通过并查集合并后，每个连通分量节点都会有相同的root，因此检查所有节点的root：

- 如果检查到只有一个root，说明这个图只有一个连通分量，是连通图，返回cost。
- 如果检查到超过一个root，说明这个图有多个连通分量，不是一个连通图，返回-1。

```
class Solution {
    public int minimumCost(int N, int[][] connections) {
        Arrays.sort(connections, (a,b) -> a[2]-b[2]);
        int[] parent = new int[N+1];
        for (int i = 1; i <= N; ++i) {
            parent[i] = i;
        }
        int cost = 0;
        for (int[] edge : connections) {
            if (union(edge[0], edge[1], parent)) {
                cost += edge[2];
            }
        }
        int p = -1;
        for (int i = 1; i <= N; ++i) {
            int root = findRoot(i, parent);
            if (p == -1) {
                p = root;
            } else if (p != root) {
                return -1;
            }
        }
        return cost;
    }
    public int findRoot(int x, int[] parent) {
        while (x != parent[x]) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }
    public boolean union(int a, int b, int[] parent) {
        a = findRoot(a, parent);
        b = findRoot(b, parent);
```

```
        if (a == b) return false;
        parent[a] = b;
        return true;
    }
}
```

14、冗余连接（684）【并查集】

684. 冗余连接

难度 中等 164 收藏 分享 切换为英文 关注 反馈

在本问题中, 树指的是一个连通且无环的**无向**图。

输入一个图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以 边 组成的二维数组。每一个 边 的元素是一对 `[u, v]` ，满足 `u < v`，表示连接顶点 `u` 和 `v` 的**无向**图的边。

返回一条可以删去的边，使得结果图是一个有着N个节点的树。如果有多个答案，则返回二维数组中最后出现的边。答案边 `[u, v]` 应满足相同的格式 `u < v`。

示例 1：

```
输入: [[1,2], [1,3], [2,3]]
输出: [2,3]
解释: 给定的无向图为:
  1
 / \
2 - 3
```

示例 2：

```
输入: [[1,2], [2,3], [3,4], [1,4], [1,5]]
输出: [1,4]
解释: 给定的无向图为:
5 - 1 - 2
    |   |
    4 - 3
```

注意:

- 输入的二维数组大小在 3 到 1000。
- 二维数组中的整数在1到N之间，其中N是输入数组的大小。

更新(2017-09-26):
我们已经重新检查了问题描述及测试用例，明确图是*无向*图。对于有向图详见冗余连接II。对于造成任何不便，我们深感歉意。

https://leetcode-cn.com/problems/redundant-connection/solution/rong-yu-lian-jie-by-leetcode/

方法二：并查集

算法：
如果我们熟悉并查集（DSU）数据结构，我们可以直接使用它来解决这个问题：我们只需找到已经连接的图中出现的第一条边。本解释的其余部分将重点介绍实现 DSU 的细节。

一个 DSU 数据结构可以用来维护图形连接组件的数据，并快速查询它们。有两种操作：

- `dsu.find(node x)`，找到元素 `x` 所在的集合的代表，该操作也可以用于判断两个元素是否位于同一个集合。
- `dsu.union(node x, node y)`，把元素 `x` 和元素 `y` 所在的集合合并，要求 `x` 和 `y` 所在的集合不相交，如果相交则不合并。

为了实现这一点，我们跟踪父结点，它会记录同一连接节点中较小结点的所在的集合。如果结点是它自己的父结点，我们将其称为连接结点的领导者。

DSU 结构的简单实现如下所示：
伪代码：
我们使用两种技术来提高运行时的复杂性：路径压缩和按秩合并。

- 路径压缩涉及将 `find` 函数中的 `x=parent[x]` 更改为 `parent[x]=find(parent[x])`。
- 按秩合并涉及到将发现的工作量平均分配给领导者。每当 `dsu.union(x, y)` 时，我们都有两个领导者 `xr，yr`，并且我们要选择是要 `parent[x]=yr` 还是 `parent[y]=xr`。我们选择有更多子节点的领导者作为领导者。
- 具体地说，`rank` 的含义是 `x` 的跟随者少于 `2 ^ rank[x]`，这个策略可以作为 `dsu.find` 中的递归循环可中的界限。

复杂度分析

- 时间复杂度：$O(N\alpha(N)) \approx O(N)$。其中，$N$ 是图中顶点的数目（以及边的数目），α 是 `Inverse-Ackermann` 函数。我们对 `dsu.union` 进行了多达 N 次查询，这需要花费 $O(\alpha(N))$ 时间。可以去查阅资料为什么 `dsu.union` 具有 $O(\alpha(N))$ 复杂性、`Inverse-Ackermann` 函数是什么以及为什么 $O(\alpha(N))$ 大约是 $O(1)$ 的原因。
- 空间复杂度：$O(N)$，图的当前构造（嵌入在我们的 DSU 结构中）最多有 N 个结点。

https://leetcode-cn.com/problems/redundant-connection/solution/java-bing-cha-ji-by-luma730/

```
class Solution {
    //缓存结果集
    int[] result = new int[2];
    public int[] findRedundantConnection(int[][] edges) {
        int[] father = new int[edges.length + 1];
        for (int i = 0; i < father.length; i++) {
            father[i] = i;
        }
        for (int[] edge : edges) {
            union(father, edge[0], edge[1]);
        }
        return result;
    }
    //路径压缩
    public int findXFather(int[] father, int x) {
        while (father[x]!=x){
            father[x] = father[father[x]];
            x = father[x];
        }
        return x;
    }
```

```
    //合并两个能连接上的点，father 合为最后确定的father
    public void union(int[] father, int x, int y) {
        int xFather = findXFather(father, x);
        int yFather = findXFather(father, y);
        if (xFather != yFather) {
            father[xFather]=yFather;
        } else {
            //在发现两个点的连接已经存在时，就更新缓存，题目要最后一个，遍历到最后一个就是结果
            result[0] = x;
            result[1] = y;
        }
    }
}
```

15、冗余连接 2（685）【并查集】

685. 冗余连接 II

难度 困难 114 收藏 分享 切换为英文 关注 反馈

在本问题中，有根树指满足以下条件的**有向**图。该树只有一个根节点，所有其他节点都是该根节点的后继。每一个节点只有一个父节点，除了根节点没有父节点。

输入一个有向图，该图由一个有着N个节点 (节点值不重复1, 2, ..., N) 的树及一条附加的边构成。附加的边的两个顶点包含在1到N中间，这条附加的边不属于树中已存在的边。

结果图是一个以 边 组成的二维数组。 每一个 边 的元素是一对 [u, v] ，用以表示**有向**图中连接顶点 u 和顶点 v 的边，其中 u 是 v 的一个父节点。

返回一条能删除的边，使得剩下的图是有N个节点的有根树。若有多个答案，返回最后出现在给定二维数组的答案。

示例 1:

```
输入: [[1,2], [1,3], [2,3]]
输出: [2,3]
解释: 给定的有向图如下:
  1
 / \
v   v
2-->3
```

示例 2:

```
输入: [[1,2], [2,3], [3,4], [4,1], [1,5]]
输出: [4,1]
解释: 给定的有向图如下:
5 <- 1 -> 2
     ^    |
     |    v
     4 <- 3
```

注意:

- 二维数组大小的在3到1000范围内。
- 二维数组中的每个整数在1到N之间，其中 N 是二维数组的大小。

https://leetcode-cn.com/problems/redundant-connection-ii/solution/rong-yu-lian-jie-ii-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：并查集

思路与算法

在一棵树中，边的数量比节点的数量少 1。如果一棵树有 N 个节点，则这棵树有 $N-1$ 条边。这道题中的图在树的基础上多了一条附加的边，因此边的数量也是 N。

树中的每个节点都有一个父节点，除了根节点没有父节点。在多了一条附加的边之后，可能有以下两种情况：

- 附加的边指向根节点，则包括根节点在内的每个节点都有一个父节点，此时图中一定有环路；
- 附加的边指向非根节点，则恰好有一个节点（即被附加的边指向的节点）有两个父节点，此时图中可能有环路也可能没有环路。

要找到附加的边，需要遍历图中的所有的边构建出一棵树，在构建树的过程中寻找导致冲突（即导致一个节点有两个父节点）的边以及导致环路出现的边。

具体做法是，使用数组 $parent$ 记录每个节点的父节点，初始时对于任何 $1 \le i \le N$ 都有 $parent[i] = i$，另外创建并查集，初始时并查集中的每个节点都是一个连通分支，该连通分支的根节点就是该节点本身。遍历每条边的过程中，维护导致冲突的边和导致环路出现的边，由于只有一条附加的边，因此最多有一条导致冲突的边和一条导致环路出现的边。

当访问到边 $[u, v]$ 时，进行如下操作：

- 如果此时已经有 $parent[v] \neq v$，说明 v 有两个父节点，将当前的边 $[u, v]$ 记为导致冲突的边；
- 否则，令 $parent[v] = u$，然后在并查集中分别找到 u 和 v 的祖先（即各自的连通分支中的根节点），如果祖先相同，说明这条边导致环路出现，将当前的边 $[u, v]$ 记为导致环路出现的边，如果祖先不同，则在并查集中将 u 和 v 进行合并。

根据上述操作，同一条边不可能同时被记为导致冲突的边和导致环路出现的边。如果访问到的边确实同时导致冲突和环路出现，则这条边被记为导致冲突的边。

在遍历图中的所有边之后，根据是否存在导致冲突的边和导致环路出现的边，得到附加的边。

如果没有导致冲突的边，说明附加的边一定导致环路出现，而且是在环路中的最后一条被访问到的边，因此附加的边即为导致环路出现的边。

如果有导致冲突的边，记这条边为 $[u, v]$，则有两条边指向 v，另一条边为 $[parent[v], v]$，需要通过判断是否有导致环路的边决定哪条边是附加的边。

- 如果有导致环路的边，则附加的边不可能是 $[u, v]$（因为 $[u, v]$ 已经被记为导致冲突的边，不可能被记为导致环路出现的边），因此附加的边是 $[parent[v], v]$。
- 如果没有导致环路的边，则附加的边是后被访问到的指向 v 的边，因此附加的边是 $[u, v]$。

复杂度分析

- 时间复杂度：$O(N \log N)$，其中 N 是图中的节点个数。需要遍历图中的 N 条边，对于每条边，需要对两个节点查找祖先，如果两个节点的祖先不同则需要进行合并，需要进行 2 次查找和最多 1 次合并。一共需要进行 $2N$ 次查找和最多 N 次合并，因此总时间复杂度是 $O(2N \log N) = O(N \log N)$。这里的并查集使用了路径压缩，但是没有使用按秩合并，最坏情况下的时间复杂度是 $O(N \log N)$，平均情况下的时间复杂度依然是 $O(N\alpha(N))$，其中 α 为阿克曼函数的反函数，$\alpha(N)$ 可以认为是一个很小的常数。
- 空间复杂度：$O(N)$，其中 N 是图中的节点个数。使用数组 $parent$ 记录每个节点的父节点，并查集使用数组记录每个节点的祖先。

```
class Solution {
    public int[] findRedundantDirectedConnection(int[][] edges) {
        int nodesCount = edges.length;
        UnionFind uf = new UnionFind(nodesCount + 1);
        int[] parent = new int[nodesCount + 1];
        for (int i = 1; i <= nodesCount; ++i) {
            parent[i] = i;
```

```
        }
        int conflict = -1;
        int cycle = -1;
        for (int i = 0; i < nodesCount; ++i) {
            int[] edge = edges[i];
            int node1 = edge[0], node2 = edge[1];
            if (parent[node2] != node2) {
                conflict = i;
            } else {
                parent[node2] = node1;
                if (uf.find(node1) == uf.find(node2)) {
                    cycle = i;
                } else {
                    uf.union(node1, node2);
                }
            }
        }
        if (conflict < 0) {
            int[] redundant = {edges[cycle][0], edges[cycle][1]};
            return redundant;
        } else {
            int[] conflictEdge = edges[conflict];
            if (cycle >= 0) {
                int[] redundant = {parent[conflictEdge[1]], conflictEdge[1]};
                return redundant;
            } else {
                int[] redundant = {conflictEdge[0], conflictEdge[1]};
                return redundant;
            }
        }
    }
}
class UnionFind {
    int[] ancestor;
    public UnionFind(int n) {
        ancestor = new int[n];
        for (int i = 0; i < n; ++i) {
            ancestor[i] = i;
        }
    }
    public void union(int index1, int index2) {
        ancestor[find(index1)] = find(index2);
    }
    public int find(int index) {
```

```
        if (ancestor[index] != index) {
            ancestor[index] = find(ancestor[index]);
        }
        return ancestor[index];
    }
}
```

16、课程表（207）

207. 课程表

难度 中等 416 收藏 分享 切换为英文 关注 反馈

你这个学期必须选修 `numCourse` 门课程，记为 `0` 到 `numCourse-1` 。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：`[0,1]`

给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习?

示例 1:

输入: 2, [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。所以这是可能的。

示例 2:

输入: 2, [[1,0],[0,1]]
输出: false
解释: 总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0；并且学习课程 0 之前，你还应先完成课程 1。这是不可能的。

提示:

1. 输入的先决条件是由 **边缘列表** 表示的图形，而不是 邻接矩阵 。详情请参见图的表示法。
2. 你可以假定输入的先决条件中没有重复的边。
3. `1 <= numCourses <= 10^5`

https://leetcode-cn.com/problems/course-schedule/solution/course-schedule-tuo-bu-pai-xu-bfs dfsliang-chong-fa/

解题思路：

- **本题可约化为：** 课程安排图是否是 **有向无环图(DAG)**。即课程间规定了前置条件，但不能构成任何环路，否则课程前置条件将不成立。
- 思路是通过 **拓扑排序** 判断此课程安排图是否是 **有向无环图(DAG)** 。 **拓扑排序原理：** 对 DAG 的顶点进行排序，使得对每一条有向边 (u,v)，均有 u（在排序记录中）比 v 先出现。亦可理解为对某点 v 而言，只有当 v 的所有源点均出现了，v 才能出现。
- 通过课程前置条件列表 `prerequisites` 可以得到课程安排图的 **邻接表** `adjacency` ，以降低算法时间复杂度，以下两种方法都会用到邻接表。

方法一：入度表（广度优先遍历）

算法流程：

1. 统计课程安排图中每个节点的入度，生成 **入度表** `indegrees`。
2. 借助一个队列 `queue`，将所有入度为 0 的节点入队。
3. 当 `queue` 非空时，依次将队首节点出队，在课程安排图中删除此节点 `pre`：
   - 并不是真正从邻接表中删除此节点 `pre`，而是将此节点对应所有邻接节点 `cur` 的入度 -1，即 `indegrees[cur] -= 1`。
   - 当入度 -1后邻接节点 `cur` 的入度为 0，说明 `cur` 所有的前驱节点已经被“删除”，此时将 `cur` 入队。
4. 在每次 `pre` 出队时，执行 `numCourses--`；
   - 若整个课程安排图是有向无环图（即可以安排），则所有节点一定都入队并出队过，即完成拓扑排序。换个角度说，若课程安排图中存在环，一定有节点的入度始终不为 0。
   - 因此，拓扑排序出队次数等于课程个数，返回 `numCourses == 0` 判断课程是否可以成功安排。

复杂度分析：

- **时间复杂度** $O(N+M)$：遍历一个图需要访问所有节点和所有临边，N 和 M 分别为节点数量和临边数量；
- **空间复杂度** $O(N+M)$：为建立邻接表所需额外空间，`adjacency` 长度为 N，并存储 M 条临边的数据。

方法 1：BFS（6ms）【邻接表】

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int[] indegrees = new int[numCourses];
        List<List<Integer>> adjacency = new ArrayList<>();
        Queue<Integer> queue = new LinkedList<>();
        for(int i = 0; i < numCourses; i++)
            adjacency.add(new ArrayList<>());
        // Get the indegree and adjacency of every course.
        for(int[] cp : prerequisites) {
            indegrees[cp[0]]++;
            adjacency.get(cp[1]).add(cp[0]);
        }
        // Get all the courses with the indegree of 0.
        for(int i = 0; i < numCourses; i++)
            if(indegrees[i] == 0) queue.add(i);
        // BFS TopSort.
        while(!queue.isEmpty()) {
            int pre = queue.poll();
            numCourses--;
            for(int cur : adjacency.get(pre))
                if(--indegrees[cur] == 0) queue.add(cur);
        }
        return numCourses == 0;
    }
}
```

方法二：深度优先遍历

原理是通过 DFS 判断图中是否有环。

算法流程：

1. 借助一个标志列表 `flags`，用于判断每个节点 `i` （课程）的状态：
   1. 未被 DFS 访问：`i == 0`；
   2. 已被**其他节点启动**的 DFS 访问：`i == -1`；
   3. 已被**当前节点启动**的 DFS 访问：`i == 1`。
2. 对 `numCourses` 个节点依次执行 DFS，判断每个节点起步 DFS 是否存在环，若存在环直接返回 $False$。DFS 流程；
   1. 终止条件：
      - 当 `flag[i] == -1`，说明当前访问节点已被其他节点启动的 DFS 访问，无需再重复搜索，直接返回 $True$。
      - 当 `flag[i] == 1`，说明在本轮 DFS 搜索中节点 `i` 被第 2 次访问，即 **课程安排图有环**，直接返回 $False$。
   2. 将当前访问节点 `i` 对应 `flag[i]` 置 1，即标记其被本轮 DFS 访问过；
   3. 递归访问当前节点 `i` 的所有邻接节点 `j`，当发现环直接返回 $False$；
   4. 当前节点所有邻接节点已被遍历，并没有发现环，则将当前节点 `flag` 置为 -1 并返回 $True$。
3. 若整个图 DFS 结束并未发现环，返回 $True$。

复杂度分析：

- **时间复杂度** $O(N+M)$：遍历一个图需要访问所有节点和所有临边，N 和 M 分别为节点数量和临边数量；
- **空间复杂度** $O(N+M)$：为建立邻接表所需额外空间，`adjacency` 长度为 N，并存储 M 条临边的数据。

方法 2：DFS（4ms）

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> adjacency = new ArrayList<>();
        for(int i = 0; i < numCourses; i++)
            adjacency.add(new ArrayList<>());
        int[] flags = new int[numCourses];
        for(int[] cp : prerequisites)
            adjacency.get(cp[1]).add(cp[0]);
        for(int i = 0; i < numCourses; i++)
            if(!dfs(adjacency, flags, i)) return false;
        return true;
    }
    private boolean dfs(List<List<Integer>> adjacency, int[] flags, int
 i) {
        if(flags[i] == 1) return false;
        if(flags[i] == -1) return true;
        flags[i] = 1;
        for(Integer j : adjacency.get(i))
            if(!dfs(adjacency, flags, j)) return false;
        flags[i] = -1;
        return true;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

17、课程表 2（210）

210. 课程表 II

难度 中等 234 收藏 分享 切换为英文 关注 反馈

现在你总共有 *n* 门课需要选，记为 `0` 到 `n-1`。

在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们：`[0,1]`

给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。

可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

示例 1:

```
输入: 2, [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1，你需要先完成课程 0。因此，正确的课程顺序为 [0,1] 。
```

示例 2:

```
输入: 4, [[1,0],[2,0],[3,1],[3,2]]
输出: [0,1,2,3] or [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3，你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
     因此，一个正确的课程顺序是 [0,1,2,3] 。另一个正确的排序是 [0,2,1,3] 。
```

说明:

1. 输入的先决条件是由**边缘列表**表示的图形，而不是邻接矩阵。详情请参见图的表示法。
2. 你可以假定输入的先决条件中没有重复的边。

提示:

1. 这个问题相当于查找一个循环是否存在于有向图中。如果存在循环，则不存在拓扑排序，因此不可能选取所有课程进行学习。
2. 通过 DFS 进行拓扑排序 - 一个关于Coursera的精彩视频教程（21分钟），介绍拓扑排序的基本概念。
3. 拓扑排序也可以通过 BFS 完成。

https://leetcode-cn.com/problems/course-schedule-ii/solution/tuo-bu-pai-xu-shen-du-you-xian-bian-li-python-dai-/

先说最重要的部分：

- 「拓扑排序」是专门应用于有向图的算法；
- 这道题用 BFS 和 DFS 都可以完成，只需要掌握 BFS 的写法就可以了，BFS 的写法很经典；
- BFS 的写法就叫「拓扑排序」，这里还用到了贪心算法的思想，贪的点是：当前让入度为 0 的那些结点入队；
- 「拓扑排序」的结果不唯一；
- 删除结点的操作，通过「入度数组」体现，这个技巧要掌握；
- 「拓扑排序」的一个附加效果是：**能够顺带检测有向图中是否存在环**，这个知识点非常重要，如果在面试的过程中遇到这个问题，要把这一点说出来。
    - 具有类似附加功能的算法还有：Bellman-Ford 算法附加的作用是可以用于检测是否有负权环（在这里不展开了，我也不太熟）。

更多硬核内容就在公众号「yes 的练级攻略」

拓扑排序实际上应用的是贪心算法，贪心算法简而言之：每一步最优，则全局最优。

具体到拓扑排序，每一次都从图中删除没有前驱的顶点，这里并不需要真正的做删除操作，我们可以设置一个入度数组，每一轮都输出入度为 0 的结点，并移除它、修改它指向的结点的入度（-1即可），依次得到的结点序列就是拓扑排序的结点序列。如果图中还有结点没有被移除，则说明“不能完成所有课程的学习”。

拓扑排序保证了每个活动（在这题中是“课程”）的所有前驱活动都排在该活动的前面，并且可以完成所有活动。拓扑排序的结果不唯一。拓扑排序还可以用于检测一个有向图是否有环。相关的概念还有 AOV 网，这里就不展开了。

算法流程：

1、在开始排序前，扫描对应的存储空间（使用邻接表），将入度为 0 的结点放入队列。

2、只要队列非空，就从队首取出入度为 0 的结点，将这个结点输出到结果集中，并且将这个结点的所有邻接结点（它指向的结点）的入度减 1，在减 1 以后，如果这个被减 1 的结点的入度为 0，就继续入队。

3、当队列为空的时候，检查结果集中的顶点个数是否和课程数相等即可。

（思考这里为什么要使用队列？如果不用队列，还可以怎么做，会比用队列的效果差还是更好？）

在代码具体实现的时候，除了保存入度为 0 的队列，我们还需要两个辅助的数据结构：

1、邻接表：通过结点的索引，我们能够得到这个结点的后继结点；

2、入度数组：通过结点的索引，我们能够得到指向这个结点的结点个数。

这个两个数据结构在遍历题目给出的邻边以后就可以很方便地得到。

复杂度分析：

- 时间复杂度：$O(E+V)$。这里 E 表示邻边的条数，V 表示结点的个数。初始化入度为 0 的集合需要遍历整张图，具体做法是检查每个结点和每条边，因此复杂度为 $O(E+V)$，然后对该集合进行操作，又需要遍历整张图中的每个结点和每条边，复杂度也为 $O(E+V)$；
- 空间复杂度：$O(V)$：入度数组、邻接表的长度都是结点的个数 V，即使使用队列，队列最长的时候也不会超过 V，因此空间复杂度是 $O(V)$。

方法 1：BFS（7ms）【邻接表】

```
class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        if (numCourses <= 0) return new int[0];
        HashSet<Integer>[] adjacency = new HashSet[numCourses];
        for (int i = 0; i < numCourses; i++) {
            adjacency[i] = new HashSet<>();
        }
        int[] indegrees = new int[numCourses];
        for(int[] cp : prerequisites){
            indegrees[cp[0]]++;
            adjacency[cp[1]].add(cp[0]);
        }
        Queue<Integer> que = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (indegrees[i] == 0) que.offer(i);
        }
        int[] res = new int[numCourses];
        int count = 0;
        while (!que.isEmpty()){
```

```
                Integer head = que.poll();
                res[count++] = head;
                Set<Integer> successors = adjacency[head];
                for (Integer nextCourse : successors){
                    indegrees[nextCourse]--;
                    if (indegrees[nextCourse] == 0) {
                        que.offer(nextCourse);
                    }
                }
            }
            return count == numCourses ? res : new int[0];
        }
    }
```

方法 2：DFS（5ms）

方法二：深度优先遍历（不用掌握，因为要检测环，代码中会做一些处理，代码没有参考价值）

这里要使用逆邻接表。其实就是检测这个有向图中有没有环，只要存在环，这些课程就不能按要求学完。

算法流程：

第 1 步：构建逆邻接表；

第 2 步：递归处理每一个还没有被访问的结点，具体做法很简单：对于一个结点来说，**先输出指向它的所有顶点，再输出自己。**

第 3 步：如果这个顶点还没有被遍历过，就递归遍历它，把所有指向它的结点都输出了，再输出自己。注意：**当访问一个结点的时候，应当先递归访问它的前驱结点，直至前驱结点没有前驱结点为止。**

复杂度分析：

- 时间复杂度：$O(E+V)$；
- 空间复杂度：$O(V)$。

```
class Solution {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        if (numCourses <= 0) return new int[0];
        int plen = prerequisites.length;
        if (plen == 0) {
            int[] res = new int[numCourses];
            for  (int i = 0; i < numCourses; i++) {
                res[i] = i;
            }
            return res;
        }
        int[] marked = new int[numCourses];
        // 初始化有向图 begin
        HashSet<Integer>[] graph = new HashSet[numCourses];
        for (int i = 0; i < numCourses; i++) {
            graph[i] = new HashSet<>();
```

```
        }
        // 初始化有向图 end
        // 有向图的 key 是前驱结点，value 是后继结点的集合
        for (int[] p : prerequisites) {
            graph[p[1]].add(p[0]);
        }
        // 使用 Stack 或者 List 记录递归的顺序，这里使用 Stack
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < numCourses; i++) {
            if (dfs(i, graph, marked, stack)) {
                // 注意方法的语义，如果图中存在环，表示课程任务不能完成，应该返回空数组
                return new int[0];
            }
        }
        // 在遍历的过程中，一直 dfs 都没有遇到已经重复访问的结点，就表示有向图中没有环
        // 所有课程任务可以完成，应该返回 true
        // 下面这个断言一定成立，这是拓扑排序告诉我们的结论
        assert stack.size() == numCourses;
        int[] res = new int[numCourses];
        for  (int i = 0; i < numCourses; i++) {
            res[i] = stack.pop();
        }
        return res;
    }
    /**
     * 注意这个 dfs 方法的语义
     *
     * @param i       当前访问的课程结点
     * @param graph
     * @param marked 如果 == 1 表示正在访问中，如果 == 2 表示已经访问完了
     * @return true 表示图中存在环，false 表示访问过了，不用再访问了
     */
    private boolean dfs(int i, HashSet<Integer>[] graph, int[] marked,
Stack<Integer> stack) {
        // 如果访问过了，就不用再访问了
        if (marked[i] == 1) {
            // 从正在访问中，到正在访问中，表示遇到了环
            return true;
        }
        if (marked[i] == 2) {
            // 表示在访问的过程中没有遇到环，这个节点访问过了
            return false;
```

```
            }
            // 走到这里，是因为初始化呢，此时 marked[i] == 0
            // 表示正在访问中
            marked[i] = 1;
            // 后继结点的集合
            HashSet<Integer> successorNodes = graph[i];
            for (Integer successor : successorNodes) {
                if (dfs(successor, graph, marked, stack)) {
                    // 层层递归返回 true ，表示图中存在环
                    return true;
                }
            }
            // i 的所有后继结点都访问完了，都没有存在环，则这个结点就可以被标记为已经访问结束
            // 状态设置为 2
            marked[i] = 2;
            stack.add(i);
            // false 表示图中不存在环
            return false;
        }
    }
```

18、不邻接植花（1042）【邻接表】

1042. 不邻接植花

难度 简单 62 收藏 分享 切换为英文 关注 反馈

有 `N` 个花园，按从 `1` 到 `N` 标记。在每个花园中，你打算种下四种花之一。

`paths[i] = [x, y]` 描述了花园 `x` 到花园 `y` 的双向路径。

另外，没有花园有 3 条以上的路径可以进入或者离开。

你需要为每个花园选择一种花，使得通过路径相连的任何两个花园中的花的种类互不相同。

以数组形式返回选择的方案作为答案 `answer`，其中 `answer[i]` 为在第 `(i+1)` 个花园中种植的花的种类。花的种类用 1, 2, 3, 4 表示。保证存在答案。

示例 1:

```
输入：N = 3, paths = [[1,2],[2,3],[3,1]]
输出：[1,2,3]
```

示例 2:

```
输入：N = 4, paths = [[1,2],[3,4]]
输出：[1,2,1,2]
```

示例 3:

```
输入：N = 4, paths = [[1,2],[2,3],[3,4],[4,1],[1,3],[2,4]]
输出：[1,2,3,4]
```

提示:

- `1 <= N <= 10000`
- `0 <= paths.size <= 20000`
- 不存在花园有 4 条或者更多路径可以进入或离开。
- 保证存在答案。

https://leetcode-cn.com/problems/flower-planting-with-no-adjacent/solution/marveljian-dan-de-xue-xi-bi-ji-1042-by-tyanyonecan/

用邻接表数组构造无向图，假设初始时每个花园都是 0 号花，然后按 1 到 N 的顺序给每个花园种花。对每个花园，都有 1、2、3、4 四种花可选，用 boolean 数组标记每种花是否被与当前花园相通的邻接花园使用过，标记完后，从剩余的没被使用过的花中选取号数最小的花种在当前花园。直至对所有花园完成种植。

```
class Solution {
    public int[] gardenNoAdj(int N, int[][] paths) {
        int[] res = new int[N];
       LinkedList<Integer>[] adj =(LinkedList<Integer>[]) new LinkedLis
t[N + 1];//邻接表数组
        for(int i = 0; i < adj.length; i++)
            adj[i] = new LinkedList<Integer>();
        for(int[] temp : paths) {//建立邻接表
            adj[temp[0]].add(temp[1]);
            adj[temp[1]].add(temp[0]);
        }
        for(int i = 1; i <= N; i++) {//从 1 号花园开始种花
            boolean[] color = new boolean[5];//每个花园初始都有 4 种可选，
false 意为没被用过
            for(int id : adj[i])
                color[res[id - 1]] = true;//邻接顶点花园使用过的种类，不能
再用
            for(int j = 1; j <= 4; j++) {//找剩余的可用种类
                if(!color[j]) {
                    res[i - 1] = j;
                    break;
                }
            }
        }
        return res;
    }
}
```

19、钥匙和房间（841）【图】

更多硬核内容就在公众号「yes 的练级攻略」

841. 钥匙和房间

难度 中等 105 收藏 分享 切换为英文 关注 反馈

有 N 个房间，开始时你位于 0 号房间。每个房间有不同的号码：0，1，2，...，N-1，并且房间里可能有一些钥匙能使你进入下一个房间。

在形式上，对于每个房间 i 都有一个钥匙列表 rooms[i]，每个钥匙 rooms[i][j] 由 [0,1，...，N-1] 中的一个整数表示，其中 N = rooms.length。 钥匙 rooms[i][j] = v 可以打开编号为 v 的房间。

最初，除 0 号房间外的其余所有房间都被锁住。

你可以自由地在房间之间来回走动。

如果能进入每个房间返回 true，否则返回 false。

示例 1：

```
输入：[[1],[2],[3],[]]
输出：true
解释：
我们从 0 号房间开始，拿到钥匙 1。
之后我们去 1 号房间，拿到钥匙 2。
然后我们去 2 号房间，拿到钥匙 3。
最后我们去了 3 号房间。
由于我们能够进入每个房间，我们返回 true。
```

示例 2：

```
输入：[[1,3],[3,0,1],[2],[0]]
输出：false
解释：我们不能进入 2 号房间。
```

提示：

1. 1 <= rooms.length <= 1000
2. 0 <= rooms[i].length <= 1000
3. 所有房间中的钥匙数量总计不超过 3000。

https://leetcode-cn.com/problems/keys-and-rooms/solution/yao-chi-he-fang-jian-by-leetcode-solution/

前言

当 x 号房间中有 y 号房间的钥匙时，我们就可以从 x 号房间去往 y 号房间。如果我们将这 n 个房间看成有向图中的 n 个节点，那么上述关系就可以看作是图中的 x 号点到 y 号点的一条有向边。

这样一来，问题就变成了给定一张有向图，询问从 0 号节点出发是否能够到达所有的节点。

方法一：深度优先搜索

思路及解法

我们可以使用深度优先搜索的方式遍历整张图，统计可以到达的节点个数，并利用数组 vis 标记当前节点是否访问过，以防止重复访问。

复杂度分析

- 时间复杂度：$O(n+m)$，其中 n 是房间的数量，m 是所有房间中的钥匙数量的总数。
- 空间复杂度：$O(n)$，其中 n 是房间的数量。主要为栈空间的开销。

方法 1：DFS

```
class Solution {
    boolean[] vis;
    int num;
    public boolean canVisitAllRooms(List<List<Integer>> rooms) {
        int n = rooms.size();
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        num = 0;
        vis = new boolean[n];
        dfs(rooms, 0);
        return num == n;
    }
    public void dfs(List<List<Integer>> rooms, int x) {
        vis[x] = true;
        num++;
        for (int i : rooms.get(x)) {
            if (!vis[i]) dfs(rooms, i);
        }
    }
}
```

方法二：广度优先搜索

思路及解法

我们也可以使用广度优先搜索的方式遍历整张图，统计可以到达的节点个数，并利用数组 vis 标记当前节点是否访问过，以防止重复访问。

复杂度分析

- 时间复杂度：$O(n+m)$，其中 n 是房间的数量，m 是所有房间中的钥匙数量的总数。
- 空间复杂度：$O(n)$，其中 n 是房间的数量。主要为队列的开销。

方法 2：BFS

```
class Solution {
    public boolean canVisitAllRooms(List<List<Integer>> rooms) {
        int n = rooms.size(), num = 0;
        boolean[] vis = new boolean[n];
        Queue<Integer> que = new LinkedList<>();
        vis[0] = true;
        que.offer(0);
        while (!que.isEmpty()) {
            int x = que.poll();
            num++;
            for (int i : rooms.get(x)) {
                if (!vis[i]) {
                    vis[i] = true;
                    que.offer(i);
                }
            }
        }
        return num == n;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

20、图像渲染（733）【颜色填充（面试题 08.10）】

733. 图像渲染

难度 简单 116 收藏 分享 切换为英文 关注 反馈

有一幅以二维整数数组表示的图画，每一个整数表示该图画的像素值大小，数值在 0 到 65535 之间。

给你一个坐标 `(sr, sc)` 表示图像渲染开始的像素值（行，列）和一个新的颜色值 `newColor`，让你重新上色这幅图像。

为了完成上色工作，从初始坐标开始，记录初始坐标的上下左右四个方向上像素值与初始坐标相同的相连像素点，接着再记录这四个方向上符合条件的像素点与他们对应四个方向上像素值与初始坐标相同的相连像素点，……，重复该过程。将所有有记录的像素点的颜色值改为新的颜色值。

最后返回经过上色渲染后的图像。

示例 1:

输入:
`image = [[1,1,1],[1,1,0],[1,0,1]]`
`sr = 1, sc = 1, newColor = 2`
输出: `[[2,2,2],[2,2,0],[2,0,1]]`
解析:
在图像的正中间，(坐标(sr,sc)=(1,1))，
在路径上所有符合条件的像素点的颜色都被更改成2。
注意，右下角的像素没有更改为2，
因为它不是在上下左右四个方向上与初始点相连的像素点。

注意:

- `image` 和 `image[0]` 的长度在范围 `[1, 50]` 内。
- 给出的初始点将满足 `0 <= sr < image.length` 和 `0 <= sc < image[0].length`。
- `image[i][j]` 和 `newColor` 表示的颜色值在范围 `[0, 65535]` 内。

https://leetcode-cn.com/problems/flood-fill/solution/tu-xiang-xuan-ran-by-leetcode-solution/

本题要求将给定的二维数组中指定的「色块」染成另一种颜色。「色块」的定义是：直接或间接相邻的同色方格构成的整体。

可以发现，「色块」就是被不同颜色的方格包围的一个同色岛屿。我们从色块中任意一个地方开始，利用广度优先搜索或深度优先搜索即可遍历整个岛屿。

注意：当目标颜色和初始颜色相同时，我们无需对原数组进行修改。

方法一：广度优先搜索

思路及算法

我们从给定的起点开始，进行广度优先搜索。每次搜索到一个方格时，如果其与初始位置的方格颜色相同，就将该方格加入队列，并将该方格的颜色更新，以防止重复入队。

注意：因为初始位置的颜色会被修改，所以我们需要保存初始位置的颜色，以便于之后的更新操作。

复杂度分析

- 时间复杂度：$O(n \times m)$，其中 n 和 m 分别是二维数组的行数和列数。最坏情况下需要遍历所有的方格一次。
- 空间复杂度：$O(n \times m)$，其中 n 和 m 分别是二维数组的行数和列数。主要为队列的开销。

更多硬核内容就在公众号「yes 的练级攻略」

方法 1：BFS（2ms）

```
class Solution {
    public int[][] floodFill(int[][] image, int sr, int sc, int newColo
r) {
        int[][] dire = {{0,1},{0,-1},{1,0},{-1,0}};
        int cur = image[sr][sc];
        if (cur == newColor) return image;
        int row = image.length, col = image[0].length;
        Queue<int[]> que = new LinkedList<>();
        que.offer(new int[] {sr, sc});
        image[sr][sc] = newColor;
        while (!que.isEmpty()){
            int[] cell = que.poll();
            int x = cell[0], y = cell[1];
            for (int i = 0; i < 4; i++){
                int nx = x + dire[i][0];
                int ny = y + dire[i][1];
                if (nx >= 0 && nx < row && ny >= 0 && ny < col && image
[nx][ny] == cur) {
                    que.offer(new int[] {nx, ny});
                    image[nx][ny] = newColor;
                }
            }
        }
        return image;
    }
}
```

方法二：深度优先搜索

思路及算法

我们从给定的起点开始，进行深度优先搜索。每次搜索到一个方格时，如果其与初始位置的方格颜色相同，就将该方格的颜色更新，以防止重复搜索；如果不相同，则进行回溯。

注意：因为初始位置的颜色会被修改，所以我们需要保存初始位置的颜色，以便于之后的更新操作。

复杂度分析

- 时间复杂度：$O(n \times m)$，其中 n 和 m 分别是二维数组的行数和列数。最坏情况下需要遍历所有的方格一次。
- 空间复杂度：$O(n \times m)$，其中 n 和 m 分别是二维数组的行数和列数。主要为栈空间的开销。

方法 2：DFS（1ms）

```
class Solution {
    int[][] dire = {{0,1},{0,-1},{1,0},{-1,0}};
    public int[][] floodFill(int[][] image, int sr, int sc, int newColo
r) {
        int cur = image[sr][sc];
```

```
        if (cur != newColor) dfs(image, sr, sc, cur, newColor);
        return image;
    }
    public void dfs(int[][] image, int x, int y, int cur, int newColor)
{
        if (image[x][y] == cur){
            image[x][y] = newColor;
            for (int i = 0; i < 4; i++){
                int nx = x + dire[i][0];
                int ny = y + dire[i][1];
                if (nx >= 0 && nx < image.length && ny >= 0 && ny < ima
ge[0].length){
                    dfs(image, nx, ny, cur, newColor);
                }
            }
        }
    }
}
```

21、扫雷游戏

系统简单略

529. 扫雷游戏

难度 中等 118 收藏 分享 切换为英文 关注 反馈

让我们一起来玩扫雷游戏!

给定一个代表游戏板的二维字符矩阵。 'M' 代表一个**未挖出的**地雷，'E' 代表一个**未挖出的**空方块，'B' 代表没有相邻（上，下，左，右，和所有4个对角线）地雷的**已挖出的**空白方块，**数字**（'1' 到 '8'）表示有多少地雷与这块**已挖出的**方块相邻，'X' 则表示一个**已挖出的**地雷。

现在给出在所有**未挖出的**方块中（'M'或者'E'）的下一个点击位置（行和列索引），根据以下规则，返回相应位置被点击后对应的面板：

1. 如果一个地雷（'M'）被挖出，游戏就结束了- 把它改为 'X'。
2. 如果一个**没有相邻地雷**的空方块（'E'）被挖出，修改它为（'B'），并且所有和其相邻的**未挖出**方块都应该被递归地揭露。
3. 如果一个**至少与一个地雷相邻**的空方块（'E'）被挖出，修改它为数字（'1'到'8'），表示相邻地雷的数量。
4. 如果在此次点击中，若无更多方块可被揭露，则返回面板。

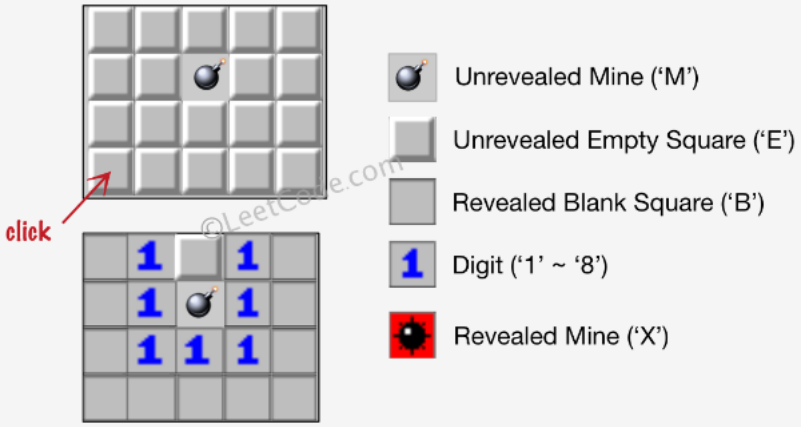
输入:

```
[['E', 'E', 'E', 'E', 'E'],
 ['E', 'E', 'M', 'E', 'E'],
 ['E', 'E', 'E', 'E', 'E'],
 ['E', 'E', 'E', 'E', 'E']]

Click : [3,0]
```

输出:

```
[['B', '1', 'E', '1', 'B'],
 ['B', '1', 'M', '1', 'B'],
 ['B', '1', '1', '1', 'B'],
 ['B', 'B', 'B', 'B', 'B']]
```

解释:



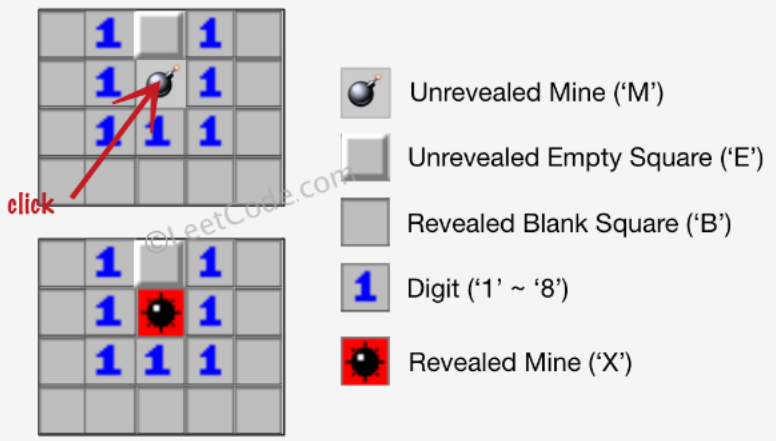
输入:

```
[['B', '1', 'E', '1', 'B'],
 ['B', '1', 'M', '1', 'B'],
 ['B', '1', '1', '1', 'B'],
 ['B', 'B', 'B', 'B', 'B']]

Click : [1,2]
```

输出:

```
[['B', '1', 'E', '1', 'B'],
 ['B', '1', 'X', '1', 'B'],
 ['B', '1', '1', '1', 'B'],
 ['B', 'B', 'B', 'B', 'B']]
```

解释:



更多硬核内容就在公众号「yes 的练级攻略」

注意：

1. 输入矩阵的宽和高的范围为 [1,50]。
2. 点击的位置只能是未被挖出的方块 ('M' 或者 'E')，这也意味着面板至少包含一个可点击的方块。
3. 输入面板不会是游戏结束的状态（即有地雷已被挖出）。
4. 简单起见，未提及的规则在这个问题中可被忽略。例如，当游戏结束时你不需要挖出所有地雷，考虑所有你可能赢得游戏或标记方块的情况。

https://leetcode-cn.com/problems/minesweeper/solution/sao-lei-you-xi-by-leetcode-solution/

方法一：深度优先搜索 + 模拟

思路与算法

由于题目要求你根据规则来展示执行一次点击操作后游戏面板的变化，所以我们只要明确该扫雷游戏的规则，并用代码模拟出来即可。

那我们着眼于题目的规则，会发现总共分两种情况：

1. 当前点击的是「未挖出的地雷」，我们将其值改为 X 即可。
2. 当前点击的是「未挖出的空方块」，我们需要统计它周围相邻的方块里地雷的数量 cnt（即 M 的数量）。如果 cnt 为零，即执行规则 2，此时需要将其改为 B，且**递归地处理**周围的八个**未挖出**的方块，递归终止条件即为规则 4，没有更多方块可被揭露的时候。否则执行规则 3，将其修改为数字即可。

整体看来，一次点击过程会从一个位置出发，逐渐向外圈扩散，所以这引导我们利用「搜索」的方式来实现。这里以深度优先搜索为例：我们定义递归函数 `dfs(x, y)` 表示当前在 (x, y) 点，执行扫雷规则的情况，我们只要按照上面理出来的情况来进行模拟即可，在 cnt 为零的时候，对当前点相邻的未挖出的方块调用递归函数，否则将其改为数字，结束递归。

复杂度分析

- 时间复杂度：$O(nm)$，其中 n 和 m 分别代表面板的宽和高。最坏情况下会遍历整个面板。
- 空间复杂度：$O(nm)$。空间复杂度取决于递归的栈深度，而递归栈深度在最坏情况下有可能遍历整个面板而达到 $O(nm)$。

一、题目分析

给定初始二维数组和起点，返回修改后的二维数组。

- 若起点处是雷，即 'M'，直接将其修改为 'X'，游戏结束；
- 若起点处是空，即 'E'，则从起点开始向 8 邻域中的空地搜索，直到到达邻接***的空地停止。
  - 和二叉树从根结点开始搜索，直到达到叶子节点停止，是几乎一样的，所以会写二叉树的 BFS/DFS，那么这题也就写出来了。

方法 1：DFS（0ms）

```
class Solution {
    // 定义 8 个方向
    int[] dx = {-1, 1, 0, 0, -1, 1, -1, 1};
    int[] dy = {0, 0, -1, 1, -1, 1, 1, -1};

    public char[][] updateBoard(char[][] board, int[] click) {
        int x = click[0], y = click[1];
        // 1. 若起点是雷，游戏结束，直接修改 board 并返回。
        if (board[x][y] == 'M') {
            board[x][y] = 'X';
```

```
        } else { // 2. 若起点是空地，则从起点开始向 8 邻域的空地进行深度优先搜索。
            dfs(board, x, y);
        }
        return board;
    }
    private void dfs(char[][] board, int i, int j) {
        // 递归终止条件：判断空地 (i, j) 周围是否有雷，若有，则将该位置修改为雷数，终止该路径的搜索。
        int cnt = 0;
        for (int k = 0; k < 8; k++) {
            int x = i + dx[k];
            int y = j + dy[k];
            if (x < 0 || x >= board.length || y < 0 || y >= board[0].length) {
                continue;
            }
            if (board[x][y] == 'M') {
                cnt++;
            }
        }
        if (cnt > 0) {
            board[i][j] =  (char)(cnt + '0');
            return;
        }
        // 若空地 (i, j) 周围没有雷，则将该位置修改为 ‘B’，向 8 邻域的空地继续搜索。
        board[i][j] = 'B';
        for (int k = 0; k < 8; k++) {
            int x = i + dx[k];
            int y = j + dy[k];
            if (x < 0 || x >= board.length || y < 0 || y >= board[0].length || board[x][y] != 'E') {
                continue;
            }
            dfs(board, x, y);
        }
    }
}
```

方法二：广度优先搜索 + 模拟

思路与算法

同样地，我们也可以将深度优先搜索改为广度优先搜索来模拟，我们只要在cnt为零的时候，将当前点相邻的未挖出的方块加入广度优先搜索的队列里即可，其他情况不加入队列，这里不再赘述。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(nm)$，其中 n 和 m 分别代表面板的宽和高。最坏情况下会遍历整个面板。
- 空间复杂度：$O(nm)$。我们需要 $O(nm)$ 的标记数组来标记当前位置是否已经被加入队列防止重复计算。

BFS 代码

这里要注意的是，与树的 BFS 不一样（每个节点只有一个父亲节点），本题图中的点会由多个点达到，因此需要加上 boolean[][] visited 数组记录访问标志，防止每个点重复入队而超时。

方法 2：BFS（4ms）

```
class Solution {
    // 定义 8 个方向
    int[] dx = {-1, 1, 0, 0, -1, 1, -1, 1};
    int[] dy = {0, 0, -1, 1, -1, 1, 1, -1};
    public char[][] updateBoard(char[][] board, int[] click) {
        // 1. 若起点是雷，游戏结束，直接修改 board 并返回。
        int x = click[0], y = click[1];
        if (board[x][y] == 'M') {
            board[x][y] = 'X';
            return board;
        }
        // 2. 若起点是空地，则将起点入队，从起点开始向 8 邻域的空地进行宽度优先搜索。
        int m = board.length, n = board[0].length;
        boolean[][] visited = new boolean[m][n];
        visited[x][y] = true;
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[] {x, y});
        while (!queue.isEmpty()) {
            int[] point = queue.poll();
            int i = point[0], j = point[1];
            // 判断空地 (i, j) 周围是否有雷
            int cnt = 0;
            for (int k = 0; k < 8; k++) {
                int newX = i + dx[k];
                int newY = j + dy[k];
                if (newX < 0 || newX >= board.length || newY < 0 || newY >= board[0].length) {
                    continue;
                }
                if (board[newX][newY] == 'M') {
                    cnt++;
                }
            }
```

```
            // 若空地 (i, j) 周围有雷，则将该位置修改为雷数；否则将该位置更新为 'B'，并将其 8 邻域中的空地入队，继续进行 bfs 搜索。
            if (cnt > 0) {
                board[i][j] = (char)(cnt + '0');
            } else {
                board[i][j] = 'B';
                for (int k = 0; k < 8; k++) {
                    int newX = i + dx[k];
                    int newY = j + dy[k];
                    if (newX < 0 || newX >= board.length || newY < 0 || newY >= board[0].length
                            || board[newX][newY] != 'E' || visited[newX][newY]) {
                        continue;
                    }
                    visited[newX][newY] = true;
                    queue.offer(new int[] {newX, newY});
                }
            }
        }
        return board;
    }
}
```

22、单词接龙（127）

127. 单词接龙

难度 中等 408 收藏 分享 切换为英文 关注 反馈

给定两个单词（*beginWord* 和 *endWord*）和一个字典，找到从 *beginWord* 到 *endWord* 的最短转换序列的长度。转换需遵循如下规则：

1. 每次转换只能改变一个字母。
2. 转换过程中的中间单词必须是字典中的单词。

说明:

- 如果不存在这样的转换序列，返回 0。
- 所有单词具有相同的长度。
- 所有单词只由小写字母组成。
- 字典中不存在重复的单词。
- 你可以假设 *beginWord* 和 *endWord* 是非空的，且二者不相同。

示例 1:

输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",
返回它的长度 5。

示例 2:

```
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: 0

解释: endWord "cog" 不在字典中，所以无法进行转换。
```

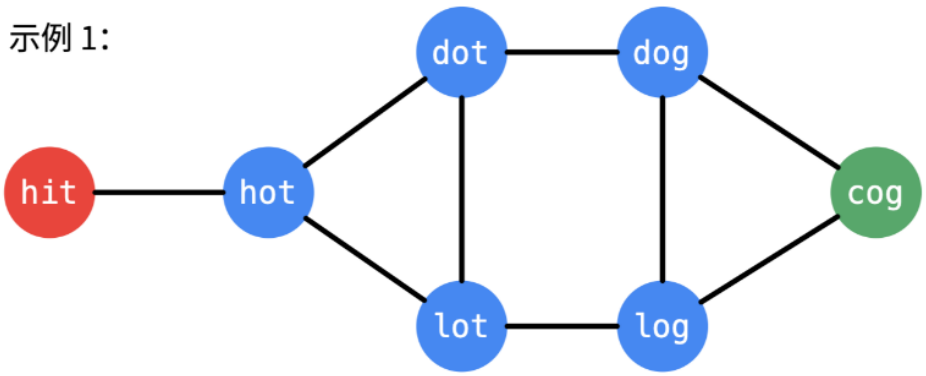
https://leetcode-cn.com/problems/word-ladder/solution/yan-du-you-xian-bian-li-shuang-xiang-yan-du-you-2/

一句话题解

- 无向图中两个顶点之间的最短路径的长度，可以通过广度优先遍历得到；
- 为什么 BFS 得到的路径最短？**可以把起点和终点所在的路径拉直来看，两点之间线段最短**；
- 已知目标顶点的情况下，可以分别从起点和目标顶点（终点）执行广度优先遍历，直到遍历的部分有交集，这是双向广度优先遍历的思想。

分析题意：

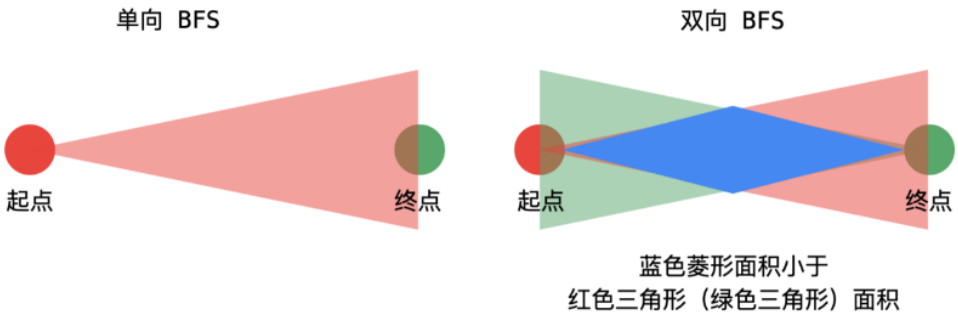
- 「转换」意即：两个单词对应位置只有一个字符不同，例如 "hit" 与 "hot"，这种转换是可以逆向的，因此，根据题目给出的单词列表，可以构建出一个无向（无权）图；



- 如果一开始就构建图，每一个单词都需要和除它以外的另外的单词进行比较，复杂度是 $O(N\text{wordLen})$，这里 N 是单词列表的长度；
- 为此，我们在遍历一开始，把所有的单词列表放进一个哈希表中，然后在遍历的时候构建图，每一次得到在单词列表里可以转换的单词，复杂度是 $O(26 \times \text{wordLen})$，借助哈希表，找到邻居与 N 无关；
- 使用 BFS 进行遍历，需要的辅助数据结构是：
  - 队列；
  - `visited` 集合。说明：可以直接在 `wordSet`（由 `wordList` 放进集合中得到）里做删除。但更好的做法是新开一个哈希表，遍历过的字符串放进哈希表里。这种做法具有普遍意义。绝大多数在线测评系统和应用场景都不会在意空间开销。

方法二：双向广度优先遍历

- 已知目标顶点的情况下，可以分别从起点和目标顶点（终点）执行广度优先遍历，直到遍历的部分有交集。这种方式搜索的单词数量会更小一些；
- 更合理的做法是，**每次从单词数量小的集合开始扩散**；
- 这里 `beginVisited` 和 `endVisited` 交替使用，等价于单向 BFS 里使用队列，每次扩散都要加到总的 `visited` 里。




```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
public class Solution {
    public int ladderLength(String beginWord, String endWord, List<Stri
ng> wordList) {
        // 先将 wordList 放到哈希表里，便于判断某个单词是否在 wordList 里
        Set<String> wordSet = new HashSet<>(wordList);
        if (wordSet.size() == 0 || !wordSet.contains(endWord)) return 0
;
        // 标准写法，总的 visited 数组
        Set<String> visited = new HashSet<>();
        // 分别用左边和右边扩散的哈希表代替单向 BFS 里的队列
        Set<String> beginVisited = new HashSet<>();
        beginVisited.add(beginWord);
        Set<String> endVisited = new HashSet<>();
        endVisited.add(endWord);
        int len = beginWord.length();
        int step = 1;
        while (!beginVisited.isEmpty() && !endVisited.isEmpty()) {
            // 打开以方便调试
            // System.out.println("beginVisited => " + beginVisited);
            // System.out.println("  endVisited => " + endVisited + "\n
");
            // 优先选择小的哈希表进行扩散，考虑到的情况更少
```

```
            if (beginVisited.size() > endVisited.size()) {
                Set<String> temp = beginVisited;
                beginVisited = endVisited;
                endVisited = temp;
            }
            // 逻辑到这里，保证 beginVisited 是相对较小的集合
            // nextLevelVisited 在扩散完成以后，会成为新的 beginVisited
            Set<String> nextLevelVisited = new HashSet<>();
            for (String word : beginVisited) {
                char[] charArray = word.toCharArray();
                for (int i = 0; i < len; i++) {
                    char originChar = charArray[i];
                    for (char c = 'a'; c <= 'z'; c++) {
                        if (originChar == c) {
                            continue;
                        }
                        charArray[i] = c;
                        String nextWord = String.valueOf(charArray);
                        if (wordSet.contains(nextWord)) {
                            if (endVisited.contains(nextWord)) {
                                return step + 1;
                            }
                            if (!visited.contains(nextWord)) {
                                nextLevelVisited.add(nextWord);
                                visited.add(nextWord);
                            }
                        }
                    }
                    // 恢复，下次再用
                    charArray[i] = originChar;
                }
            }
            // 这一行代表表示从 begin 这一侧向外扩散了一层
            beginVisited = nextLevelVisited;
            step++;
        }
        return 0;
    }
    public static void main(String[] args) {
        List<String> wordList = new ArrayList<>();
        String[] words = {"hot", "dot", "dog", "lot", "log", "cog"};
        Collections.addAll(wordList, words);

        Solution solution = new Solution();
```

```
        String beginWord = "hit";
        String endWord = "cog";
        int ladderLength = solution.ladderLength(beginWord, endWord, wo
rdList);
        System.out.println(String.format("从 %s 到 %s 的最短转换序列的长
度：%d。", beginWord, endWord, ladderLength));
    }
}
```

23、单词接龙 2（126）

126. 单词接龙 II

难度 困难 311 收藏 分享 切换为英文 关注 反馈

给定两个单词（*beginWord* 和 *endWord*）和一个字典 *wordList*，找出所有从 *beginWord* 到 *endWord* 的最短转换序列。转换需遵循如下规则：

1. 每次转换只能改变一个字母。
2. 转换后得到的单词必须是字典中的单词。

说明:

- 如果不存在这样的转换序列，返回一个空列表。
- 所有单词具有相同的长度。
- 所有单词只由小写字母组成。
- 字典中不存在重复的单词。
- 你可以假设 *beginWord* 和 *endWord* 是非空的，且二者不相同。

示例 1:

```
输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出:
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
```

示例 2:

```
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: []

解释: endWord "cog" 不在字典中，所以不存在符合要求的转换序列。
```

https://leetcode-cn.com/problems/word-ladder-ii/solution/yan-du-you-xian-bian-li-shuang-xiang-yan-du-you--2/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

这道题是一道常见的考题，没有特别难理解的技巧，完全可以由已有的知识迁移过来。**主要考察的是编码和耐心调试的能力。**以下代码量较多，仅供参考，建议读者理解清楚解题思想以后，自行编码、调试完成。

思路分析：

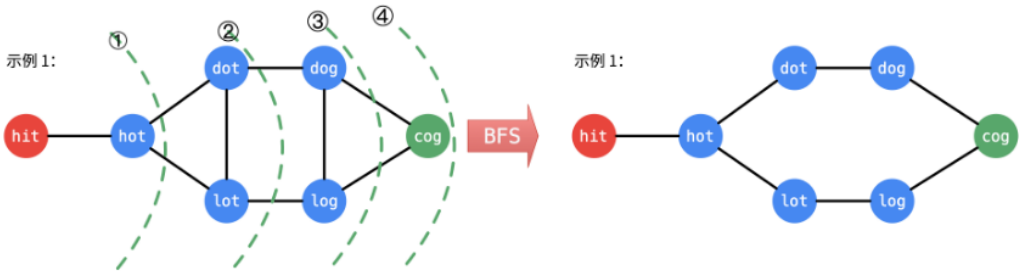
- 单词列表、起点、终点构成**无向图**，容易想到使用**广度优先遍历**找到最短路径：图中广度优先遍历要使用到；
  - 队列；
  - 标记是否访问过的「布尔数组」或者「哈希表」`visited`，这里 key 是字符串，故使用哈希表；

说明：本题可以在 `wordList` （添加到哈希表以后）上做减法，不过标准的、更可靠的做法是，另外使用一个 `visited` 哈希表，将访问以后的单词添加到 `visited` 哈希表中，下面给出的参考代码都采用这种不节约空间的做法。

- 重点：由于要记录**所有的**路径，广度优先遍历「当前层」到「下一层」的所有路径都得记录下来。因此找到下一层的结点 `wordA` 以后，**不能马上添加到 `visited` 哈希表里，还需要检查当前队列中未出队的单词是否还能与 `wordA` 建立联系**；
- 广度优先遍历位于同一层的单词，即使有联系，也是不可以被记录下来的，这是因为**同一层的连接肯定不是起点到终点的最短路径的边**；
- 使用 BFS 的同时记录遍历的路径，形式：哈希表。哈希表的 key 记录了「顶点字符串」，哈希表的值 value 记录了 key 对应的字符串在广度优先遍历的过程中得到的所有后继结点列表 `successors`；
- 最后根据 `successors`，使用**回溯算法**（全程使用一份路径变量搜索所有可能结果的深度优先遍历算法）得到所有的最短路径。

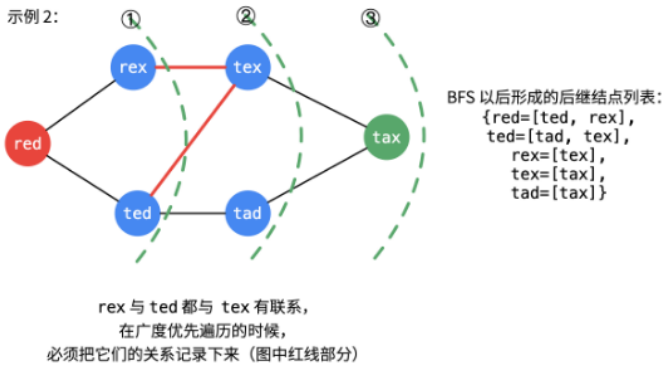
方法一：广度优先遍历

- 如果开始就构建图，结果可能会超时，这里采用的方法是：先把 `wordList` 存入哈希表，然后是一边遍历，一边找邻居，进而构建图，这是解决这个问题的特点；
- 每一层使用一个新的 `nextLevelVisited` 哈希表，记录当前层的下一层可以访问到哪些结点。**直到上一层队列里的值都出队以后，`nextLevelVisited` 哈希表才添加到总的 `visited` 哈希表，这样记录当前结点和广度优先遍历到的子结点列表才不会遗漏。**



广度优先遍历以后，dot <-> lot 与 dog <-> log 的关系不能被记录下来，
由于它们在同一层，
包含它们的路径一定不是起点到终点的最短路径。

更多硬核内容就在公众号「yes 的练级攻略」



为此，设计算法流程如下：

- 第 1 步：使用广度优先遍历找到终点单词，并且记录下沿途经过的所有结点，以邻接表形式存储；
- 第 2 步：通过邻接表，使用回溯算法得到所有从起点单词到终点单词的路径。

复杂度分析：（如有错误还请朋友们请指正）

- 时间复杂度：$O(N \times 26 \times wordLen + N2^N)$
  - 广度优先遍历：最差情况下每个结点遍历一次，时间复杂度为 $O(N)$，这里 N 是单词列表的单词总数；
  - 建图：一开始将所有单词放进哈希表 $O(N)$，每一个单词查找邻居 $O(26 \times wordLen)$，`wordLen` 表示单词的长度，一般而言不会很大，比起 N 会小很多，甚至可以视为常数；
  - 回溯算法：指数级别 $O(2^N)$，记录每一个结果 $O(N)$，这里认为最坏情况下，最短路径为单词列表的单词总数。
- 空间复杂度：$O(ways \times N \times wordLen)$，这里 ways 表示最短路径的方案数。

说明：

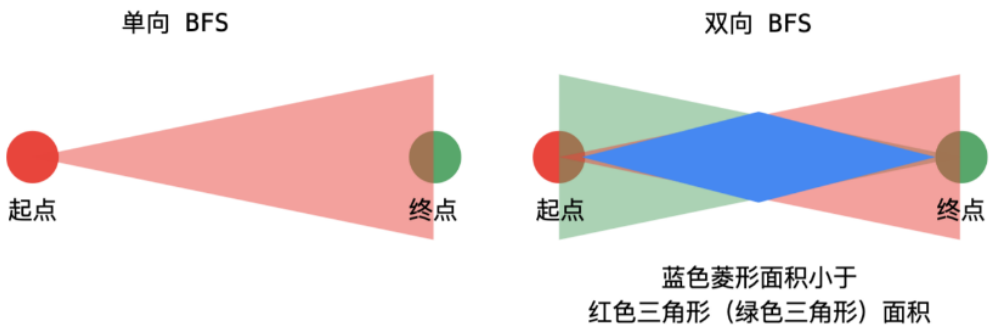
```
successors.computeIfAbsent(currentWord, a -> new HashSet<>());
successors.get(currentWord).add(nextWord);
```

以上代码从 Java 1.8 以后支持，等价的代码片段是：

```
// 维护 successors 的定义
if (successors.containsKey(currentWord)) {
    successors.get(currentWord).add(nextWord);
} else {
    Set<String> newSet = new HashSet<>();
    newSet.add(nextWord);
    successors.put(currentWord, newSet);
}
```

方法二：双向广度优先遍历

- 由于我们这个问题知道目标值，因此可以让起点和终点「同时」做 BFS，直到它们又交集，两边一起扩散形成的路径也是最短路径；
- 双向 BFS 有点像山洞挖隧道，两边一起挖，直到打通为止。双向 BFS 搜索到的集合更小，相当于做了一些剪枝操作，因此会更快一些。



编码说明：

- 参考代码 2 与参考代码 1 仅仅把「广度优先遍历」方法的实现改成了「双向广度优先遍历」，回溯的代码是一样的；
- `beginVisited` 和 `endVisited` 交替使用，实现类似队列（符合顺序性，先进先出）的效果；
- 注意：不一定非要严格按照「一左一右」的顺序交替向中间扩散，**每次都从较小的集合扩散，统计意义上搜索的范围更小**；
- 理解下面设计 `forward` 变量的作用，虽然是双向扩散，但是记录「后继结点列表」的时候方向固定。

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
public class Solution {
    public List<List<String>> findLadders(String beginWord, String endW
ord, List<String> wordList) {
        // 先将 wordList 放到哈希表里，便于判断某个单词是否在 wordList 里
        List<List<String>> res = new ArrayList<>();
        Set<String> wordSet = new HashSet<>(wordList);
        if (wordSet.size() == 0 || !wordSet.contains(endWord)) {
            return res;
        }
        // 第 1 步：使用双向广度优先遍历得到后继结点列表 successors
        // key：字符串，value：广度优先遍历过程中 key 的后继结点列表
        Map<String, Set<String>> successors = new HashMap<>();
```

```
        boolean found = bidirectionalBfs(beginWord, endWord, wordSet, successors);
        if (!found) {
            return res;
        }
        // 第 2 步：基于后继结点列表 successors ，使用回溯算法得到所有最短路径列表
        Deque<String> path = new ArrayDeque<>();
        path.addLast(beginWord);
        dfs(beginWord, endWord, successors, path, res);
        return res;
    }
    private boolean bidirectionalBfs(String beginWord, String endWord, Set<String> wordSet, Map<String, Set<String>> successors) {
        // 记录访问过的单词
        Set<String> visited = new HashSet<>();
        visited.add(beginWord);
        visited.add(endWord);

        Set<String> beginVisited = new HashSet<>();
        beginVisited.add(beginWord);
        Set<String> endVisited = new HashSet<>();
        endVisited.add(endWord);

        int wordLen = beginWord.length();
        boolean forward = true;
        boolean found = false;
        // 在保证了 beginVisited 总是较小（可以等于）大小的集合前提下，&& !endVisited.isEmpty() 可以省略
        while (!beginVisited.isEmpty() && !endVisited.isEmpty()) {
            // 一直保证 beginVisited 是相对较小的集合，方便后续编码
            if (beginVisited.size() > endVisited.size()) {
                Set<String> temp = beginVisited;
                beginVisited = endVisited;
                endVisited = temp;

                // 只要交换，就更改方向，以便维护 successors 的定义
                forward = !forward;
            }
            Set<String> nextLevelVisited = new HashSet<>();
            // 默认 beginVisited 是小集合，因此从 beginVisited 出发
            for (String currentWord : beginVisited) {
                char[] charArray = currentWord.toCharArray();
                for (int i = 0; i < wordLen; i++) {
```

```
                    char originChar = charArray[i];
                    for (char j = 'a'; j <= 'z'; j++) {
                        if (charArray[i] == j) {
                            continue;
                        }
                        charArray[i] = j;
                        String nextWord = new String(charArray);
                        if (wordSet.contains(nextWord)) {
                            if (endVisited.contains(nextWord)) {
                                found = true;
                                // 在另一侧找到单词以后，还需把这一层关系添加到「后继结点列表」
                                addToSuccessors(successors, forward, currentWord, nextWord);
                            }

                            if (!visited.contains(nextWord)) {
                                nextLevelVisited.add(nextWord);
                                addToSuccessors(successors, forward, currentWord, nextWord);
                            }
                        }
                    }
                    charArray[i] = originChar;
                }
            }
            beginVisited = nextLevelVisited;
            visited.addAll(nextLevelVisited);
            if (found) {
                break;
            }
        }
        return found;
    }

    private void dfs(String beginWord,
                     String endWord,
                     Map<String, Set<String>> successors,
                     Deque<String> path,
                     List<List<String>> res) {

        if (beginWord.equals(endWord)) {
            res.add(new ArrayList<>(path));
            return;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }

        if (!successors.containsKey(beginWord)) {
            return;
        }

        Set<String> successorWords = successors.get(beginWord);
        for (String successor : successorWords) {
            path.addLast(successor);
            dfs(successor, endWord, successors, path, res);
            path.removeLast();
        }
    }

    private void addToSuccessors(Map<String, Set<String>> successors, b
oolean forward,
                                 String currentWord, String nextWord) {
        if (!forward) {
            String temp = currentWord;
            currentWord = nextWord;
            nextWord = temp;
        }

        // Java 1.8 以后支持
        successors.computeIfAbsent(currentWord, a -> new HashSet<>());
        successors.get(currentWord).add(nextWord);
    }

    public static void main(String[] args) {
        List<String> wordList = new ArrayList<>();
        wordList.add("hot");
        wordList.add("dot");
        wordList.add("dog");
        wordList.add("lot");
        wordList.add("log");
        wordList.add("cog");
        Solution solution = new Solution();
        String beginWord = "hit";
        String endWord = "cog";
        List<List<String>> res = solution.findLadders(beginWord, endWor
d, wordList);
        System.out.println(res);
    }
}
```

yes的练级攻略

24、火星词典（269）【拓扑排序】

269. 火星词典

难度 困难 73 收藏 分享 切换为英文 关注 反馈

现有一种使用字母的全新语言，这门语言的字母顺序与英语顺序不同。

假设，您并不知道其中字母之间的先后顺序。但是，会收到词典中获得一个 **不为空的** 单词列表。因为是从词典中获得的，所以该单词列表内的单词已经 **按这门新语言的字母顺序进行了排序。**

您需要根据这个输入的列表，还原出此语言中已知的字母顺序。

示例 1：

```
输入：
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
输出："wertf"
```

示例 2：

```
输入：
[
  "z",
  "x"
]
输出："zx"
```

示例 3：

```
输入：
[
  "z",
  "x",
  "z"
]
输出：""
解释：此顺序是非法的，因此返回 ""。
```

提示：

- 你可以默认输入的全部都是小写字母
- 若给定的顺序是不合法的，则返回空字符串即可
- 若存在多种可能的合法字母顺序，请返回其中任意一种顺序即可

https://leetcode-cn.com/problems/alien-dictionary/solution/java-de-tuo-bu-pai-xu-keng-shi-li-jie-ti-mu-by-use/

解题思路

坑爹的是这个题目，看了题解才了解
举个栗子:
"abf" -> "abc" 这个意思是 abf 本身之间没有顺序关系， 但是 abf 和 abc 之间有顺序 所以 f 在 c之前;
"abc" -> "ab" 这个就是错误的，因为 c -> "没有" ，有问题。

```
class Solution {
    public String alienOrder(String[] words) {
        HashMap<Character, Set<Character>> graph = new HashMap();
        StringBuilder res = new StringBuilder();
        for (int i = 0; i < words.length - 1; i++) {
            // 这里改成这样就是为了防止 abc
          int len = Math.max(words[i].length(), words[i + 1].length());
            for (int j = 0; j < len; j++) {
                // 这里是个坑 要防止 abc -> ab 这种情况
                if (j >= words[i].length()) break;
                if (j >= words[i+1].length()) return "";

                if (words[i].charAt(j) == words[i + 1].charAt(j)) {
                    continue;
                }
                Set<Character> set = graph.computeIfAbsent(words[i].cha
rAt(j), k -> new HashSet<>());
                set.add(words[i + 1].charAt(j));
                graph.put(words[i].charAt(j), set);
                break;
            }
        }
        // 拓扑排序
        LinkedList<Character> queue = new LinkedList();
        // 入度初始化
        int[] inDegree = new int[26];

        int numChar = 0;
        Arrays.fill(inDegree, -1);
        for (String word : words) {
            for (char c : word.toCharArray()) {
                inDegree[c - 'a'] = 0;
            }
        }
        // 入度为 0 的先学习 想成一个课程表
        //
        for (Character key : graph.keySet()) {
            for (Character value: graph.get(key)) {
```

```
                inDegree[value - 'a']++;
            }
        }
        for (int i = 0; i < 26; i++) {
            if (inDegree[i] == 0) {
                queue.add((char) (i + 'a'));
            }
            if (inDegree[i] != -1) {
                numChar++;
            }
        }
        while (!queue.isEmpty()) {
            Character firstChar = queue.poll();
            res.append(firstChar);
            if (graph.containsKey(firstChar)) {
                for (Character nextChar : graph.get(firstChar)) {
                    inDegree[nextChar - 'a']--;
                    if (inDegree[nextChar - 'a'] == 0) {
                        queue.add(nextChar);
                    }
                }
            }
        }
        if (res.length() != numChar) {
            return "";
        }
        return res.toString();
    }
}
```

25、平行课程（1136）【拓扑排序】

1136. 平行课程

难度 困难 14 收藏 分享 切换为英文 关注 反馈

已知有 `N` 门课程，它们以 `1` 到 `N` 进行编号。

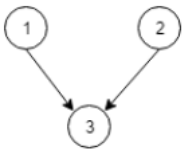
给你一份课程关系表 `relations[i] = [X, Y]`，用以表示课程 `X` 和课程 `Y` 之间的先修关系：课程 `X` 必须在课程 `Y` 之前修完。

假设在一个学期里，你可以学习任何数量的课程，但前提是你已经学习了将要学习的这些课程的所有先修课程。

请你返回学完全部课程所需的最少学期数。

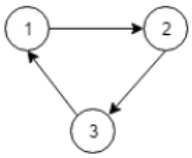
如果没有办法做到学完全部这些课程的话，就返回 `-1`。

示例 1：



输入： `N = 3, relations = [[1,3],[2,3]]`
输出： `2`
解释：
在第一个学期学习课程 `1` 和 `2`，在第二个学期学习课程 `3`。

示例 2：



输入： `N = 3, relations = [[1,2],[2,3],[3,1]]`
输出： `-1`
解释：
没有课程可以学习，因为它们相互依赖。

提示：

1. `1 <= N <= 5000`
2. `1 <= relations.length <= 5000`
3. `relations[i][0] != relations[i][1]`
4. 输入中没有重复的关系

https://leetcode-cn.com/problems/parallel-courses/solution/java-tuo-bu-pai-xu-by-purerain-2/

思路

算是比较经典的拓扑排序模板题。具体步骤如下

1. 首先，建立一个邻接表，用来存在每门课程的后继课程列表；
2. 然后，求得每个节点的入度（即每门课程先行课程的个数）；
3. 接着，将入度为0（即没有先行课程的课程）的节点加入到队列中；
4. 最后，将队列中（只存放当前入度为0的节点）的所有节点出队，然后更新他们的所有后继节点的入度（减1），将更新后的入度为0的节点再加入到队列中，直到队列为空。同时，这里定义了两个变量：ans和leftNodeCount，ans就是学期数，leftNodeCount就是剩下的节点数，如代码所示。如果最终leftNodeCount没有减到0，就说明还有一些课程没发修，因为找不到入度为0的节点，如示例2中的两两相互依赖的情况。具体代码如下：

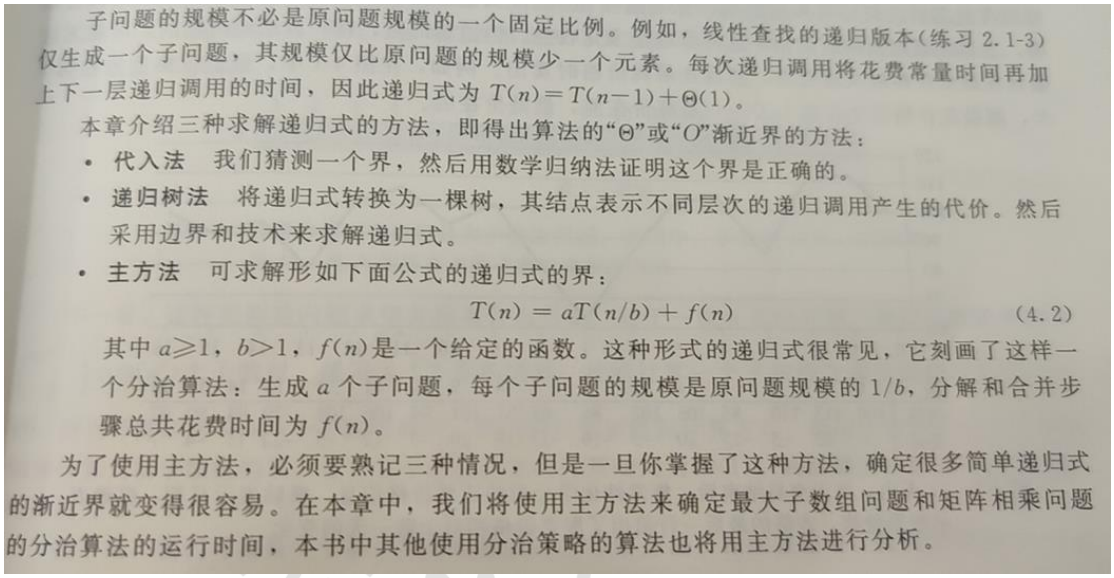
```
class Solution {
    public int minimumSemesters(int n, int[][] relations) {
```

```
ArrayList<Integer>[] adj = new ArrayList[n+1];
for (int i = 1; i <= n; i++) {
    adj[i] = new ArrayList<>();
}
for (int i = 0; i < relations.length; i++) {
    int v1 = relations[i][0];
    int v2 = relations[i][1];
    adj[v1].add(v2);
}
int[] inDegree = new int[n + 1];
for (int i = 1; i <= n; i++) {
    ArrayList<Integer> neighbors = adj[i];
    for (int j = 0; j < neighbors.size(); j++) {
        int neighbor = neighbors.get(j);
        inDegree[neighbor]++;
    }
}
LinkedList<Integer> queue = new LinkedList<>();
for (int i = 1; i <= n; i++) {
    if (inDegree[i] == 0) {
        queue.addLast(i);
    }
}
if (queue.isEmpty()) {
    // 没有入度为0的
    return -1;
}
int ans = 0;
int leftNodeCount = n;
while (!queue.isEmpty()) {
    List<Integer> list = new ArrayList<>();
    while (!queue.isEmpty()) {
        list.add(queue.removeFirst());
    }
    leftNodeCount -= list.size();
    ans++;
    for (int i = 0; i < list.size(); i++) {
        int v = list.get(i);
        ArrayList<Integer> neighbors = adj[v];
        for (int j = 0; j < neighbors.size(); j++) {
            int neighbor = neighbors.get(j);
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                queue.addLast(neighbor);
```

```
                }
            }
        }
    }
    return leftNodeCount > 0 ? -1 : ans;
  }
}
```

二、DFS （Depth first search 深度优先搜索）& 回溯算法

【时间复杂度计算】



子问题的规模不必是原问题规模的一个固定比例。例如，线性查找的递归版本(练习 2.1-3)仅生成一个子问题，其规模仅比原问题的规模少一个元素。每次递归调用将花费常量时间再加上下一层递归调用的时间，因此递归式为 $T(n)=T(n-1)+\Theta(1)$。

本章介绍三种求解递归式的方法，即得出算法的“Θ”或“O”渐近界的方法：

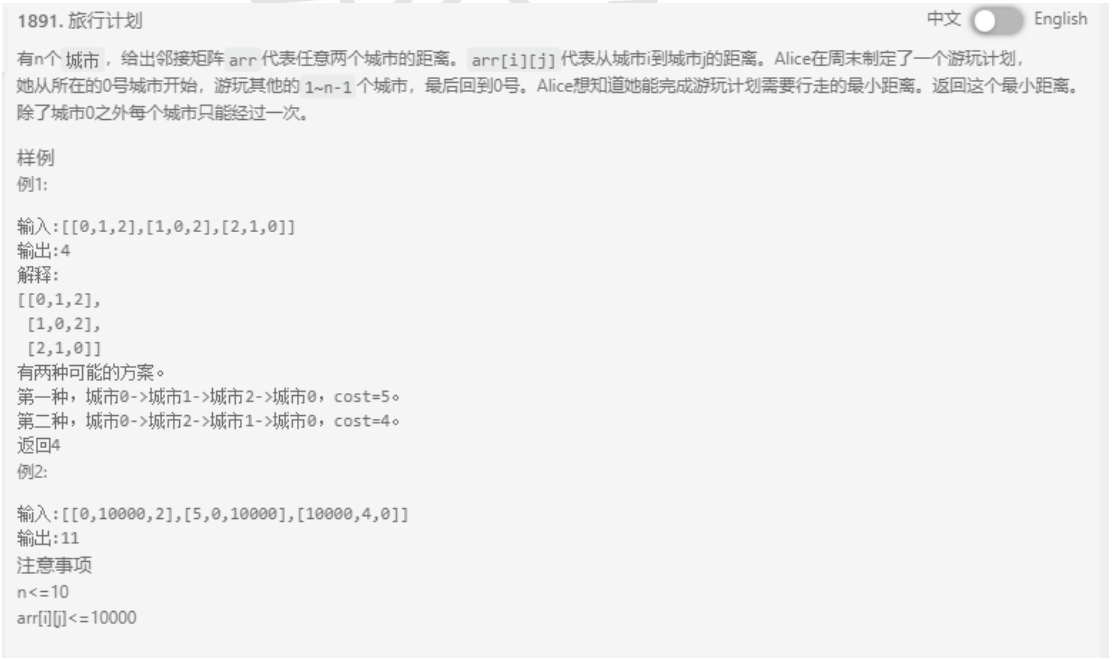
- **代入法** 我们猜测一个界，然后用数学归纳法证明这个界是正确的。
- **递归树法** 将递归式转换为一棵树，其结点表示不同层次的递归调用产生的代价。然后采用边界和技术来求解递归式。
- **主方法** 可求解形如下面公式的递归式的界：

$$T(n) = aT(n/b) + f(n) \qquad (4.2)$$

其中 $a \geqslant 1$，$b > 1$，$f(n)$是一个给定的函数。这种形式的递归式很常见，它刻画了这样一个分治算法：生成 a 个子问题，每个子问题的规模是原问题规模的 $1/b$，分解和合并步骤总共花费时间为 $f(n)$。

为了使用主方法，必须要熟记三种情况，但是一旦你掌握了这种方法，确定很多简单递归式的渐近界就变得很容易。在本章中，我们将使用主方法来确定最大子数组问题和矩阵相乘问题的分治算法的运行时间，本书中其他使用分治策略的算法也将用主方法进行分析。

1、旅行计划（lintcode1891）



1891. 旅行计划

中文 English

有n个城市，给出邻接矩阵 arr 代表任意两个城市的距离。 arr[i][j] 代表从城市i到城市j的距离。Alice在周末制定了一个游玩计划，她从所在的0号城市开始，游玩其他的 1~n-1 个城市，最后回到0号。Alice想知道她能完成游玩计划需要行走的最小距离。返回这个最小距离。除了城市0之外每个城市只能经过一次。

样例
例1:

输入:[[0,1,2],[1,0,2],[2,1,0]]
输出:4
解释:
[[0,1,2],
 [1,0,2],
 [2,1,0]]
有两种可能的方案。
第一种，城市0->城市1->城市2->城市0，cost=5。
第二种，城市0->城市2->城市1->城市0，cost=4。
返回4
例2:

输入:[[0,10000,2],[5,0,10000],[10000,4,0]]
输出:11
注意事项
n<=10
arr[i][j]<=10000

https://www.jiuzhang.com/solution/travel-plan/

```
public class Solution {
    /**
     * @param arr: the distance between any two cities
```

```
     * @return: the minimum distance Alice needs to walk to complete the travel plan
     */
    public int res;
    public int[] travel = new int[20]; //当前访问节点的花费
    public int[] vis = new int[20]; //是否已经访问过的标识
    public int travelPlan(int[][] arr) {
        int n = arr.length;
        travel[0] = 0;
        res = Integer.MAX_VALUE;
        for (int i = 0; i < 20; i++) {
            vis[i] = 0;
        }
        dfs(arr, 1, n);
        return res;
    }
    // k 是准备游玩的城市，n 是最后一个城市
    public void dfs(int[][] arr, int k, int n) {
        int i, tmp;
        if (k == n) {
            travel[k] = 0;
            tmp = 0;
            for (i = 1; i <= k; i++) {
                tmp += arr[travel[i-1]][travel[i]];
            }
            res = Math.min(tmp,res);
        }
        for (i = 1; i < n; i++) {
            if (vis[i] == 0) {
                vis[i] = 1;
                travel[k] = i;
                dfs(arr, k+1, n);
                vis[i] = 0;
            }
        }
    }
}
```

2、旅行计划 2（lintcode1306）

禁止商用
公众号：
yes的练级攻略

描述

中文 English

有n个 城市 ，给出邻接矩阵 arr 代表任意两个城市的距离。 arr[i][j] 代表从城市i到城市j的距离。Alice在周末制定了一个游玩计划，她从所在的0号城市开始，游玩其他的 1~n-1 个城市，最后回到0号。Alice想知道她能完成游玩计划需要行走的最小距离。返回这个最小距离。除了城市0之外每个城市只能经过一次。

n<=15
arr[i][j]<=10000

您在真实的面试中是否遇到过这个题? 是 题目纠错

样例

Example 1:

```
Input:[[0,1,2],[1,0,2],[2,1,0]]
Output:4
Explanation:
There are two possible plans.
The first, city 0-> city 1-> city 2-> city 0, cost = 5.
The second, city 0-> city 2-> city 1-> city 0, cost = 4.
Return 4
```

Example 2:

```
Input:[[0,10000,2],[5,0,10000],[10000,4,0]]
Output:11
```

与上一题的差别在于 n 变大了，所以必须用动态规划

```
public class Solution {
    /**
     * @param arr: the distance between any two cities
     * @return: the minimum distance Alice needs to walk to complete the travel plan
     */
    public int travelPlan(int[][] arr) {
        if (arr.length == 0) return 0;
        int cityNum = arr.length;
        // v 表示的是集合元素的数量，如 3 个城市则 v = 4，也就是{0,0} {0,1} {1,0}{1,1}这 4 种状态，注意，这里是默认从 0 号城市开始，所以只有两个城市的状态表示
        int v = 1 << (cityNum - 1);
        int[][] dp = new int[cityNum][v];
        for (int i = 0; i < cityNum; i++) dp[i][0] = arr[i][0];
        for (int j = 1; j < v; j++) {
            for (int i = 0; i < cityNum; i++) {
                dp[i][j] = Integer.MAX_VALUE;
                //相与为 0,表示从 i 号城市出发，经过城市子集 v[j],回到起点 0 号城市
                int tmp = (j >> (i-1)) & 1;
                if (tmp == 0) {
                    for (int k = 1; k < cityNum; k++) {
                        //curTmp 为 1，表示城市 k 在子集中
                        int curTmp = (j >> (k-1)) & 1;
                        if (curTmp == 1) {
                            // j - (1 << (k-1)) 表示将城市 k 从子集中去掉的城市集
```

```
                                int removeTmp = j-(1 << (k-1));
                                dp[i][j] = Math.min(dp[i][j], arr[i][k] + dp[k][removeTmp]);
                            }
                        }
                    }
                }
            }
            return dp[0][v-1];
        }
}
```

3、电话号码的字母组合（17）

17. 电话号码的字母组合

难度 中等 755 收藏 分享 切换为英文 关注 反馈

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例:

输入： "23"
输出： ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

说明:
尽管上面的答案是按字典序排列的，但是你可以任意选择答案输出的顺序。

https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number/solution/17-dian-hua-hao-ma-de-zi-mu-zu-he-java-stringbuild/

解题思路

本题虽然需要有一个 数组->字符串->字符 的过程，但由于每个数字对应的字符无重复，因此只需要在每个位置针对每个数字对应的所有字符进行遍历即可，实质上就是一个排列组合。

```
class Solution {
    // 数字与字符串进行对应
```

```
    String[] digits2String = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs"
, "tuv", "wxyz"};
    List<String> ans = new ArrayList<>();
    public List<String> letterCombinations(String digits) {
        if (digits.length() == 0) return ans;
        StringBuilder sb = new StringBuilder();
        char[] digitStr = digits.toCharArray();
        combineDigits(sb, digitStr, 0);
        return ans;
    }

    public void combineDigits(StringBuilder sb, char[] digitStr, int k) {
        // 当当前位置超出界限，则将 sb 加入 ans 中，并返回
        if (k == digitStr.length) {
            ans.add(sb.toString());
            return;
        }
        int digit = digitStr[k] - '0';
        char[] str = digits2String[digit].toCharArray();
        // 遍历当前位置数字对应的所有字符，并递归。
        // 此处即为回溯。
        // 回溯后需要删除当前位置添加的字符，并加入此位置对应的下一个字符继续递归
        for (char c : str) {
            sb.append(c);
            combineDigits(sb, digitStr, k + 1);
            sb.deleteCharAt(k);
        }
    }
}
```

4、组合总和（39）

39. 组合总和

难度 中等 736 收藏 分享 切换为英文 关注 反馈

给定一个**无重复元素**的数组 `candidates` 和一个目标数 `target` ，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的数字可以无限制重复被选取。

说明：

- 所有数字（包括 `target`）都是正整数。
- 解集不能包含重复的组合。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7,
所求解集为：
[
  [7],
  [2,2,3]
]
```

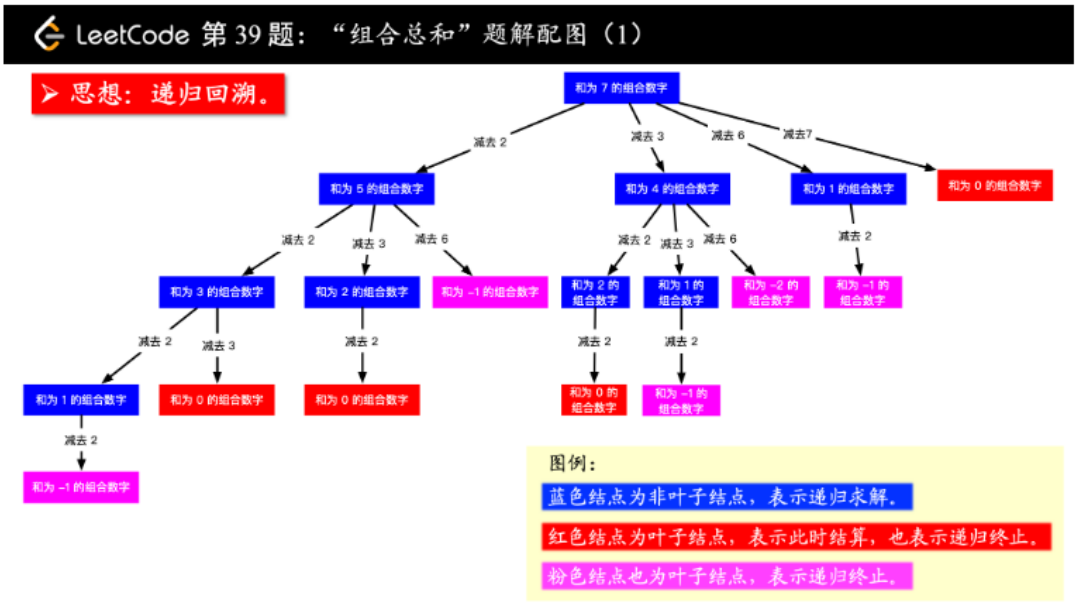
示例 2：

```
输入：candidates = [2,3,5], target = 8,
所求解集为：
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

链接：
https://leetcode-cn.com/problems/combination-sum/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-m-2/

思路：根据示例 1：输入: candidates = [2,3,6,7]，target = 7。
候选数组里有 2 ，如果找到了 7 - 2 = 5 的所有组合，再在之前加上 2 ，就是 7 的所有组合；
同理考虑 3，如果找到了 7 - 3 = 4 的所有组合，再在之前加上 3 ，就是 7 的所有组合，依次这样找下去；
上面的思路就可以画成下面的树形图。

公众号·yes的练级攻略

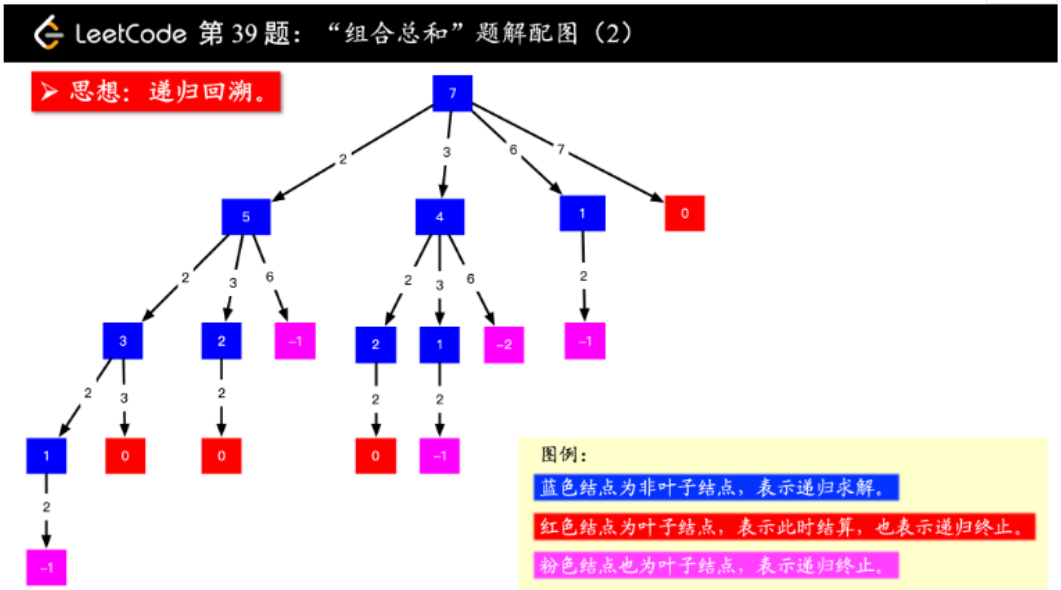


说明：

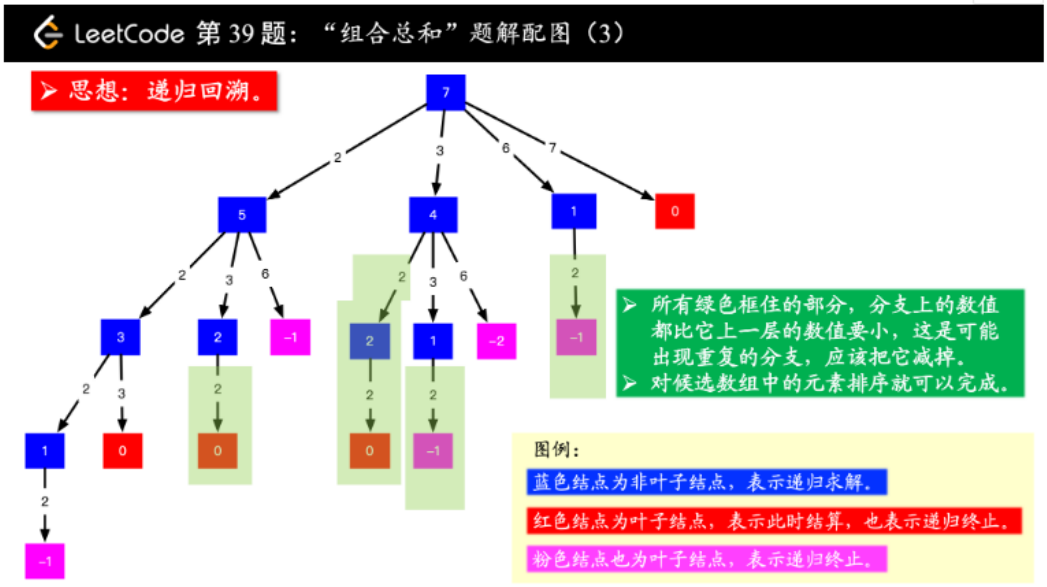
- 蓝色结点表示：尝试找到组合之和为该数的所有组合，怎么找呢？逐个减掉候选数组中的元素即可；
- 以 `target = 7` 为根结点，每一个分支做减法；
- 减到 0 或者负数的时候，到了叶子结点；
- 减到 0 的时候结算，这里 “结算” 的意思是添加到结果集；
- 从根结点到叶子结点（必须为 `0`）的路径，就是题目要我们找的一个组合。

把文字的部分去掉。

☆收藏



如果这样编码的话，会发现提交不能通过，这是因**为递归树画的有问题**，下面看一下是什么原因。



画出图以后，我看了一下，我这张图画出的结果有 4 个 0，对应的路径是 `[[2, 2, 3], [2, 3, 2], [3, 2, 2], [7]]`，而示例中的解集只有 `[[7], [2, 2, 3]]`，很显然，**重复的原因是在较深层的结点值考虑了之前考虑过的元素，因此我们需要设置“下一轮搜索的起点”即可（这里可能没有说清楚，已经尽力了）。**

去重复

- 在搜索的时候，需要设置搜索起点的下标 `begin`，由于一个数可以使用多次，下一层的结点从这个搜索起点开始搜索；
- 在搜索起点 `begin` 之前的数因为以前的分支搜索过了，所以一定会产生重复。

剪枝提速

- 如果一个数位搜索起点都不能搜索到结果，那么比它还大的数肯定搜索不到结果，基于这个想法，我们可以对输入数组进行排序，以减少搜索的分支；
- 排序是为了提高搜索速度，非必要；
- 搜索问题一般复杂度较高，能剪枝就尽量需要剪枝。把候选数组排个序，遇到一个较大的数，如果以这个数为起点都搜索不到结果，后面的数就更搜索不到结果了。

方法 1：含有排序的解法，时间复杂度 o(nlogn)

```
public class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> combinationSum(int[] candidates, int tar
get) {
        if (candidates.length == 0) return res;
        // 排序是为了提前终止搜索
        Arrays.sort(candidates);
        dfs(candidates, target, 0, new ArrayList<>());
        return res;
    }
    /**
     * @param candidates 数组输入
     * @param residue    剩余数值
```

```
     * @param begin      本轮搜索的起点下标
     * @param path       从根结点到任意结点的路径
     */
    private void dfs(int[] candidates,  int residue, int begin, List<In
teger> path) {
        if (residue == 0) {
            // 由于 path 全局只使用一份，到叶子结点的时候需要做一个拷贝
            res.add(new ArrayList<>(path));
            return;
        }
        for (int i = begin; i < candidates.length; i++) {
            // 在数组有序的前提下，剪枝
            if (residue - candidates[i] < 0) {
                break;
            }
            path.add(candidates[i]);
            dfs(candidates, residue - candidates[i], i, path);
            path.remove(path.size()-1);
        }
    }
}
```

方法 2：不排序的解法

```
public class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> combinationSum(int[] candidates, int tar
get) {
        if (candidates.length == 0) return res;
        dfs(candidates, target, 0, new ArrayList<>());
        return res;
    }
    /**
     * @param candidates 数组输入
     * @param residue    剩余数值
     * @param begin      本轮搜索的起点下标
     * @param path       从根结点到任意结点的路径
     */
    private void dfs(int[] candidates,  int residue, int begin, List<In
teger> path) {
        if (residue == 0) {
            // 由于 path 全局只使用一份，到叶子结点的时候需要做一个拷贝
            res.add(new ArrayList<>(path));
            return;
        }
        for (int i = begin; i < candidates.length; i++) {
```

```
                //如果减数大于目标值，则差为负数，不符合结果
                if (candidates[i] <= residue) {
                    path.add(candidates[i]);
                    dfs(candidates, residue - candidates[i], i, path);
                    //每次回溯将最后一次加入的元素删除
                    path.remove(path.size()-1);
                }
            }
        }
}
```

5、组合总和二（40）

40. 组合总和 II

难度 中等　292　收藏　分享　切换为英文　关注　反馈

给定一个数组 `candidates` 和一个目标数 `target` ，找出 `candidates` 中所有可以使数字和为 `target` 的组合。

`candidates` 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 解集不能包含重复的组合。

示例 1:

```
输入: candidates = [10,1,2,7,6,1,5], target = 8,
所求解集为:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

示例 2:

```
输入: candidates = [2,5,2,1,2], target = 5,
所求解集为:
[
  [1,2,2],
  [5]
]
```

参考：

https://leetcode-cn.com/problems/combination-sum-ii/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-m-3/

一定要用排序，dfs 的索引要加一，要小剪枝

这道题与上一问的区别在于：

- 第 39 题：candidates 中的数字可以无限制重复被选取。
- 第 40 题：candidates 中的每个数字在每个组合中只能使用一次。

编码的不同在于下一层递归的起始索引不一样。

- 第 39 题：还从候选数组的当前索引值开始。
- 第 40 题：从候选数组的当前索引值的下一位开始。

相同之处：解集不能包含重复的组合。

为了使得解集不包含重复的组合。我们想一想，**如何去掉一个数组中重复的元素**，除了使用哈希表以外，我们还可以**先对数组升序排序，重复的元素一定不是排好序以后的第 1 个元素和相同元素的第 1 个元素**。根据这个思想，我们先对数组升序排序是有必要的。候选数组有序，对于在递归树中发现重复分支，进而“剪枝”也是有效的。

思路分析：

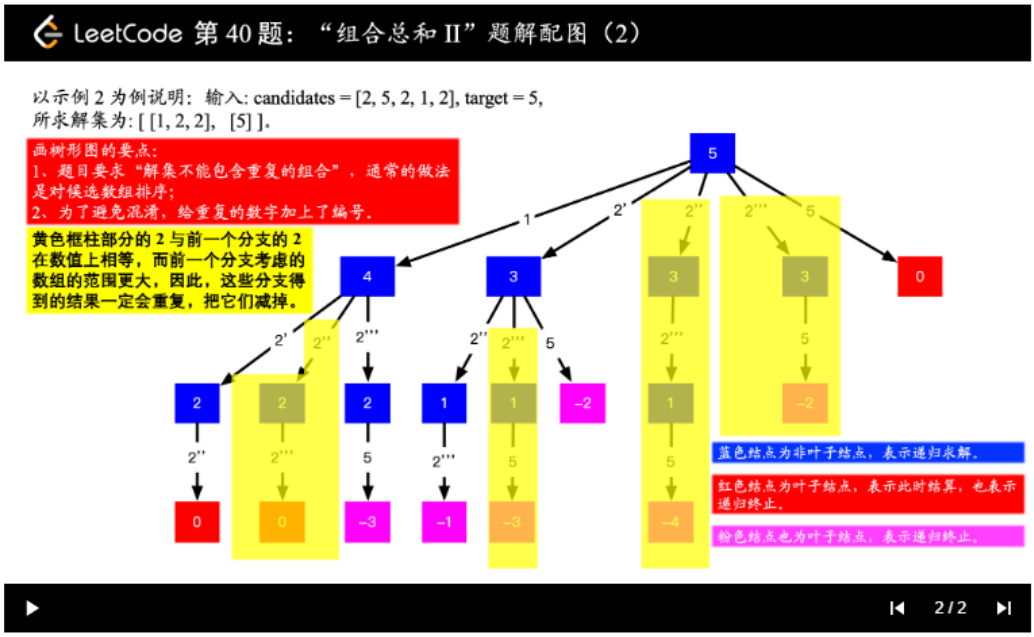
这道题其实比上一问更简单，思路是：

> 以 target 为根结点，依次减去数组中的数字，直到小于 0 或者等于 0，把等于 0 的结果记录到结果集中。

当然你也可以以 0 为根结点，依次加上数组中的数字，直到大于 target 或者等于 target，把等于 target 的结果记录到结果集中。

- “解集不能包含重复的组合”，就提示我们得对数组先排个序（**“升序”或者“降序”均可**，下面示例中均使用“升序”）。
- “candidates 中的每个数字在每个组合中只能使用一次”，那就按照顺序依次减去数组中的元素，递归求解即可：遇到 0 就结算且回溯，遇到负数也回溯。
- `candidates` 中的数字可以重复，可以借助「力扣」第 47 题：“全排列 II” 的思想，在搜索的过程中，找到可能发生重复结果的分支，把它剪去。

（温馨提示：下面的幻灯片中，有几页上有较多的文字，可能需要您停留一下，可以点击右下角的后退 “|◀” 或者前进 “▶|” 按钮控制幻灯片的播放。）



40 具体代码

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> combinationSum2(int[] candidates, int ta
rget) {
        if (candidates.length == 0) return res;
```

```
        Arrays.sort(candidates);
        List<Integer> path = new ArrayList<>(candidates.length);
        dfs (candidates, target, 0, path);
        return res;
    }
    public void dfs (int[] candidates, int residue, int begin, List<Int
eger> path) {
        if (residue == 0) {
            res.add(new ArrayList<>(path));
            return;
        }
        for (int i = begin ; i <candidates.length; i++) {
            // 大剪枝
            if (residue - candidates[i] < 0) {
                break;
            }
            // 小剪枝
            if (i > begin && candidates[i] == candidates[i-1]){
                continue;
            }
            path.add(candidates[i]);
            // 因为元素不可以重复使用，这里递归传递下去的是 i + 1 而不是 i
            dfs(candidates, residue - candidates[i],i+1, path);
            path.remove(path.size()-1);
        }
    }
}
```

6、组合总数三（216）

YES的练级攻略

216. 组合总和 III

难度 中等 133 收藏 分享 切换为英文 关注 反馈

找出所有相加之和为 *n* 的 *k* 个数的组合。组合中只允许含有 1 - 9 的正整数，并且每种组合中不存在重复的数字。

说明：

- 所有数字都是正整数。
- 解集不能包含重复的组合。

示例 1:

```
输入: k = 3, n = 7
输出: [[1,2,4]]
```

示例 2:

```
输入: k = 3, n = 9
输出: [[1,2,6], [1,3,5], [2,3,4]]
```

与上题相比，多了个 k 和 nums 数组

https://leetcode-cn.com/problems/combination-sum-iii/solution/hui-su-jian-zhi-by-liweiwei1419/

解题思路：

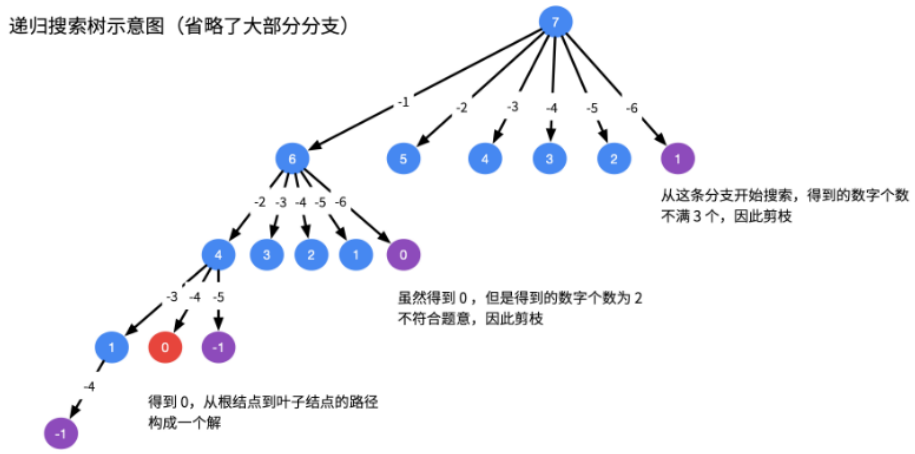
回溯问题，细节很多，但是也有一点套路：

- 先画出递归树，帮助分析；
- 代码实现，即使用 **深度优先遍历**，搜索 **所有** 可能的解（因为是遍历，所以可以得到所有的解）；
- 注意 **状态重置**（恢复现场，以免出错）和 **剪枝**（避免不必要的搜索消耗时间），这一步没有技巧，根据一些特殊用例， **画图帮助分析，看清剪枝和结算的条件** 。

说明：

- 以下给出的解答仅供参考，建议大家看懂了一点点思路和剪枝的方法以后，自己尝试实现。因为这个问题涉及到的只是一些计算，没有太多技巧；
- 剪枝在这道问题里是可以做得比较多的，但是本身这道问题数据量小，因此剪枝不是很有必要。

以示例为例：



我只画了一部分，全部画出来会占用很多版面。不过大致能够帮助我们分析出代码需要怎样写。

- 尝试做减法，减到 0 就说明可能找到了一个符合题意的组合，但是题目对组合里元素的个数有限制，因此还需要对元素个数做判断；
- 如果减到负数，没有必要继续搜索下去；
- 由于结果集里的元素互不相同，因此**下一层搜索的起点应该是上一层搜索的起点值 + 1**；
- **根据画出的递归树设计递归方法的参数。**

更多硬核内容就在公众号「yes 的练级攻略」

编写代码就是要在这棵递归树上搜索所有符合条件的解，使用的是深度优先遍历。需要设计递归方法，参数有：

- 剩下的数的和是多少，这里命名为 `residue` ，一维剩余元素之和，初始的时候传入 `n` ；
- 因为题目规定，需要 `k` 个数，每递归一层，需要找的数就减去 1，因此需要一个变量表示还需要搜索多少个数，我们仍然用 `k` 表示，到 `0` 的时候搜索结束；
- 搜索的起点，最小是 1，最大是 9，**事实上，这个最大值有一个上界**。我们暂时忽略对它的讨论；
- 需要一个变量 `path` 表示从根结点到叶子结点的路径，因为我们的操作总是在末尾进行，**栈** 是最符合当前语义的数据结构；

> 注意：以下 3 点针对 Java 语言，其它类型的语言请大家区别对待。

- 对象类型的状态变量（这里是 `path` ），在方法传递中是引用传递，在结算的时候要做一个拷贝；
- 对象类型的状态变量（这里是 `path` ），在方法传递中是引用传递，在回溯的时候，需要重置现场，这就是在 `dfs` 前后，代码是对称的原因；
- 基本类型的状态变量值（这里是 `k` 、 `start` 、 `residue` ），在方法传递中是值传递，每一次都用一个新的，因此无需状态重置（恢复现场）。

这里感谢 @wardseptember 这位朋友提供的建议。由于历史问题，Java 官方的 `Stack` 类已经不建议使用，建议使用 `Deque<Integer> stack = new ArrayDeque<Integer>();` ，注意只使用这个类里关于栈的相关接口。

- 栈使用： `addLast()` ，不建议使用 `add()` ，因为 `add()` 依然是调用 `addLast()` ，使用 `addLast()` 语义更清晰；

```
484             /**
485              * Inserts the specified element at the end of this deque.
486              *
487              * <p>This method is equivalent to {@link #addLast}.
488              *
489              * @param e the element to add
490              * @return {@code true} (as specified by {@link Collection#add})
491              * @throws NullPointerException if the specified element is null
492              */
493 @           public boolean add( @NotNull() E e) {
494                 addLast(e);
495                 return true;
496             }
```

- 出栈使用： `removeLast()` 或者 `pollLast()` ， `removeLast()` 里面依然是调用 `pollLast()` ，注意：不要使用 `poll()` ，因为 `poll()` 调用 `pollFirst()` ，不符合栈的语义。

保存结果集的变量。这些参数可以在编写代码的过程中去思考，缺啥补啥。

yes的练级攻略

这一版代码提交以后，成绩还可以，我个人感觉是因为题目限制了数字的范围，所以搜索起来即使不剪枝，很快也会结束。

下面考虑剪枝，在这题中不是必须，仅供参考。

- 首先是对输入的特殊判断：

```
if (k <= 0 || n <= 0 || k >= n) {
    return res;
}
```

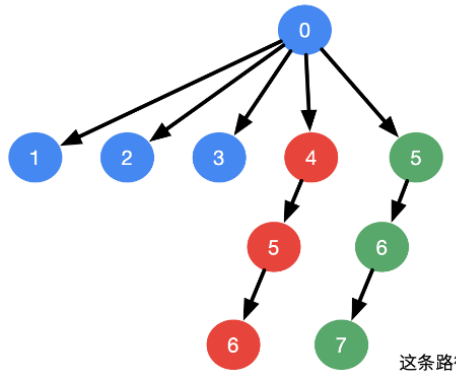
- `n` 其实是有上限的，考虑最大的 `k` 个数字：`[9, 8, ... , (9 - k + 1)]`，他们的和为 `(19 - k) * k / 2`，如果比这个数还大，就不用搜索下去了。
- 每一轮搜索的起点值也有一个上限，起点值最多到 `9`，我们找一下规律：

起点值列表：`[..., 7, 8, 9]`

- 找 3 个数，起点最多到 7；
- 找 2 个数，起点最多到 8。

因此一般规律是：`起点上界 + k = 10`，故 `起点上界 = 10 - k`。事实上，还可以计算得到一个更小的上界，但是在这题里没有必要了，每次去计算也要消耗时间。

k = 3，n = 17，下面这张图按照加法来看
第 1 层起点最多到 7，但其实还可以更小



这条路径为搜索下去的最小路径，
其和为 15，有必要搜索下去

这条路径为搜索下去的最小路径，
其和为 18 > 17，没有必要搜索下去

```java
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.List;
public class Solution {
    public List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> res = new ArrayList<>();
        // 一开始做一些特殊判断
        if (k <= 0 || n <= 0 || k >= n) return res;
        // 寻找 n 的上限：[9, 8, ... , (9 - k + 1)]，它们的和为 (19 - k) * k / 2
        // 比上限还大，就不用搜索了：
        if (n > (19 - k) * k / 2) return res;
        // 根据官方对 Stack 的使用建议，这里将 Deque 对象当做 stack 使用
        // 注意只使用关于栈的接口
```

```
        Deque<Integer> path = new ArrayDeque<>();
        dfs(k, n, 1, path, res);
        return res;
    }

    /**
     * @param k       剩下要找 k 个数
     * @param residue 剩余多少
     * @param start   下一轮搜索的起始元素是多少
     * @param path    深度优先遍历的路径参数（状态变量）
     * @param res     保存结果集的列表
     */
    private void dfs(int k, int residue, int start, Deque<Integer> path, List<L
ist<Integer>> res) {
        // 剪枝：[start, 9] 这个区间里的数都不够 k 个，不用继续往下搜索
        if (10 - start < k) return;
        if (k == 0) {
            if (residue == 0) {
                res.add(new ArrayList<>(path));
                return;
            }
        }
        // 枚举起点值 [..., 7, 8, 9]
        // 找 3 个数，起点最多到 7
        // 找 2 个数，起点最多到 8
        // 规律是，起点上界 + k = 10，故起点上界 = 10 - k
        for (int i = start; i <= 10 - k; i++) {
            // 剪枝
            if (residue - i < 0) break;
            path.addLast(i);
            dfs(k - 1, residue - i, i + 1, path, res);
            path.removeLast();
        }
    }
}
```

7、组合（77）

77. 组合

难度 中等 292 收藏 分享 切换为英文

给定两个整数 *n* 和 *k*，返回 1 ... *n* 中所有可能的 *k* 个数的组合。

示例:

```
输入: n = 4, k = 2
输出:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

参考：

https://leetcode-cn.com/problems/combinations/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-ma-/

方法：回溯

和全排列问题一样，这是一道使用回溯算法解决的经典问题。**而分析回溯问题，我们常常需要画图来帮助我们理清思路和寻找边界条件。**

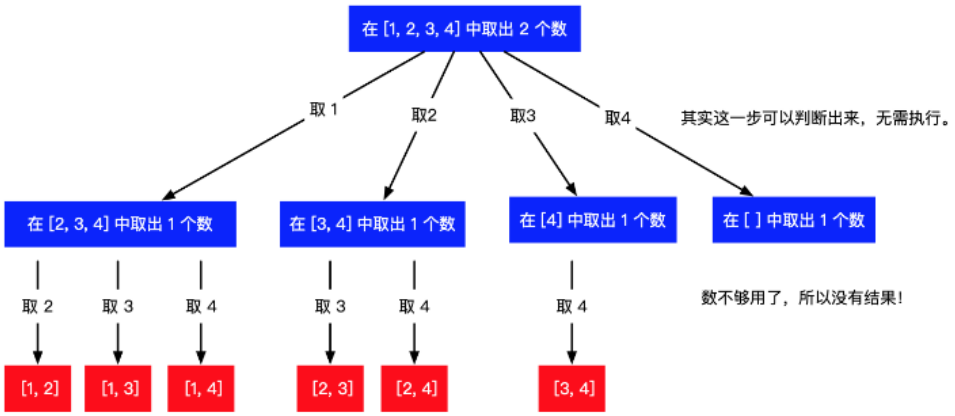
☆ 收藏

关键：

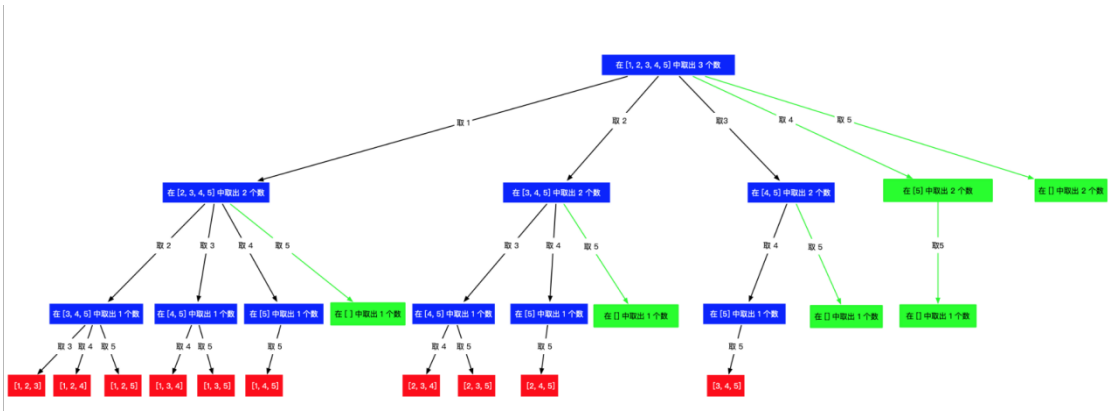
1、画图，清晰地展现递归结构，按顺序取就无需使用 used 数组判断是否取过；

2、在树的第 k 层结算，因此 pre 的长度是需要关注的；

3、可能有一些分支是没有必要执行的，我们放在优化中介绍。



优化：剪枝



其中绿色的部分，是不能产生结果的分支，但是我们的代码确实又执行到了这部分。

上面的代码中，我们发现：其实如果 `pre` 已经选择到 `[1,4,5]` 或者 `[2,4,5]` 或者 `[3,4,5]` ，后序的代码就没有必要执行，继续走也不能发现新的满足题意的组合。干了类似于下面事情，其实有很多步骤是多余的：选择了 `[1,4,5]` 以后，`5` 弹出 `[1,4,5]` 成为 `[1,4]` ，`4` 弹出 `[1,4]` 成为 `4` ，然后 `5` 进来，成为 `[1,5]` ，在进来发现 `for` 循环都进不了（因为没有可选的元素），然后 `5` 又弹出，接着 `1` 弹出。

发现多余操作：那么我们如何发现多余的步骤呢，其实也是有规律可寻的，就在 for 循环中：

发现多余操作：那么我们如何发现多余的步骤呢，其实也是有规律可寻的，就在 for 循环中：

```
for (int i = start; i <= n; i++) {
    pre.add(i);
    generateCombinations(n, k, i + 1, pre);
    pre.remove(pre.size() - 1);
}
```

这个函数干的事情，是从 `[i, n]` 这个区间里（注意，左右都是闭区间），找到 `k - pre.zize()` 个元素。 `i <= n` 不是每一次都要走完的， `i` 有一个上限。

寻找规律：我们再看图，可以发现一些边界情况，帮助我们发现规律：

当选定了一个元素，即 `pre.size() == 1` 的时候，接下来要选择 2 个元素， i 最大的值是 4 ，因为从 5 开始选择，就无解了；
当选定了两个元素，即 `pre.size() == 2` 的时候，接下来要选择 1 个元素， i 最大的值是 5 ，因为从 6 开始选择，就无解了。

再如：如果 `n = 6` , `k = 4` ，
`pre.size() == 1` 的时候，接下来要选择 3 个元素， `i` 最大的值是 4，最后一个被选的是 `[4,5,6]` ；
`pre.size() == 2` 的时候，接下来要选择 2 个元素， `i` 最大的值是 5，最后一个被选的是 `[5,6]` ；
`pre.size() == 3` 的时候，接下来要选择 1 个元素， `i` 最大的值是 6，最后一个被选的是 `[6]` ；

再如：如果 `n = 15` ， `k = 4` ，
`pre.size() == 1` 的时候，接下来要选择 3 个元素， `i` 最大的值是 13，最后一个被选的是 `[13,14,15]` ；
`pre.size() == 2` 的时候，接下来要选择 2 个元素， `i` 最大的值是 14，最后一个被选的是 `[14,15]` ；
`pre.size() == 3` 的时候，接下来要选择 1 个元素， `i` 最大的值是 15，最后一个被选的是 `[15]` ；

多写几遍（发现 `max(i)` 是我们倒着写出来），我么就可以发现 `max(i)` 与 接下来要选择的元素貌似有一点关系，很容易知道：
`max(i) + 接下来要选择的元素个数 - 1 = n` ，其中， 接下来要选择的元素个数 `= k - pre.size()` ，整理得到：

```
max(i) = n - (k - pre.size()) + 1
```

所以，我们的剪枝过程就是：把 `i <= n` 改成 `i <= n - (k - pre.size()) + 1` ：

77 具体代码

更多硬核内容就在公众号「yes 的练级攻略」

```java
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> combine(int n, int k) {
        if (n <= 0 || k <= 0 || n < k)  return res;
        List<Integer> path = new ArrayList<>();
        dfs(n, k, 1, path); //从 1 开始
        return res;
    }
    public void dfs(int n, int k, int index, List<Integer> path) {
        if (path.size() == k) {
            res.add(new ArrayList<>(path));
            return;
        }
        // i 的极限值满足： n - i + 1 = (k - pre.size())
        // 【关键】n - i + 1 是闭区间 [i,n] 的长度。
        // k - pre.size() 是剩下还要寻找的数的个数。
        for (int i = index; i <= n - (k - path.size()) + 1; i++) {
            path.add(i);
            dfs(n, k, i+1, path);
            path.remove(path.size()-1);
        }
    }
}
```

8、全排序（46）

46. 全排列

难度 中等　756　收藏　分享　切换为英

给定一个 **没有重复** 数字的序列，返回其所有可能的全排列。

示例:

```
输入: [1,2,3]
输出:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

参考：

https://leetcode-cn.com/problems/combination-sum-ii/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-m-3/

请读者带着以下问题理解回溯搜索算法的思想。

> 1、什么是“树形问题”？为什么为什么是在树形问题上使用“深度优先遍历”？不用深度优先遍历我们还可以用什么？
> 2、什么是“回溯”？为什么需要回溯？
> 3、不回溯可以吗？

首先介绍“回溯”算法的应用。“回溯”算法也叫“回溯搜索”算法，主要用于在一个庞大的空间里搜索我们所需要的问题的解。我们每天使用的“搜索引擎”就是帮助我们在庞大的互联网上搜索我们需要的信息。“搜索”引擎的“搜索”和“回溯搜索”算法的“搜索”意思是一样的。

“回溯”指的是“状态重置”，可以理解为“回到过去”、“恢复现场”，是在编码的过程中，是为了节约空间而使用的一种技巧。而回溯其实是“深度优先遍历”特有的一种现象。之所以是“深度优先遍历”，是因为我们要解决的问题通常是在一棵树上完成的，在这棵树上搜索需要的答案，一般使用深度优先遍历。

“全排列”就是一个非常经典的“回溯”算法的应用。我们知道，`N` 个数字的全排列一共有 $N!$ 这么多个。

大家可以尝试一下在纸上写 3 个数字、4 个数字、5 个数字的全排列，相信不难找到这样的方法。

以数组 `[1, 2, 3]` 的全排列为例。

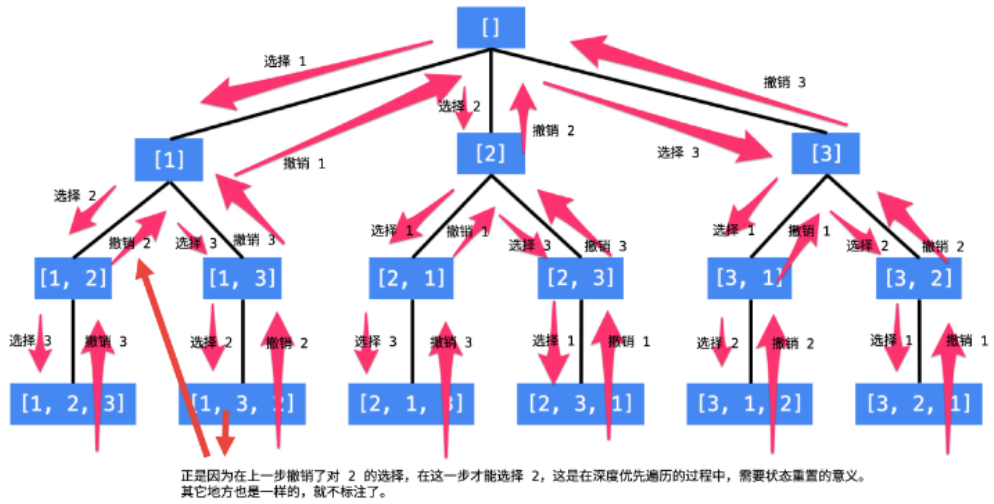
- 我们先写以 1 开头的全排列，它们是：`[1, 2, 3]`，`[1, 3, 2]`；
- 再写以 2 开头的全排列，它们是：`[2, 1, 3]`，`[2, 3, 1]`；
- 最后写以 3 开头的全排列，它们是：`[3, 1, 2]`，`[3, 2, 1]`。

我们只需要按顺序枚举每一位可能出现的情况，已经选择的数字在接下来要确定的数字中不能出现。按照这种策略选取就能够做到不重不漏，把可能的全排列都枚举出来。

- 在枚举第一位的时候，有 3 种情况。
- 在枚举第二位的时候，前面已经出现过的数字就不能再被选取了；
- 在枚举第三位的时候，前面 2 个已经选择过的数字就不能再被选取了。

这样的思路，我们可以用一个树形结构表示。看到这里的朋友，建议自己先尝试画一下“全排列”问题的树形结构。

使用编程的方法得到全排列，就是在这样的一个树形结构中进行编程，具体来说，就是**执行一次深度优先遍历，从树的根结点到叶子结点形成的路径就是一个全排列。**



说明：

1、每一个结点表示了“全排列”问题求解的不同阶段，这些阶段通过变量的“不同的值”体现；
2、这些变量的不同的值，也称之为“状态”；
3、使用深度优先遍历有“回头”的过程，在“回头”以后，状态变量需要设置成为和先前一样；
4、因此在回到上一层结点的过程中，需要撤销上一次选择，这个操作也称之为“状态重置”；
5、深度优先遍历，可以直接借助系统栈空间，为我们保存所需要的状态变量，在编码中只需要注意遍历到相应的结点的时候，状态变量的值是正确的，具体的做法是：往下走一层的时候，`path` 变量在尾部追加，而往回走的时候，需要撤销上一次的选择，也是在尾部操作，因此 `path` 变量是一个栈。
6、深度优先遍历通过“回溯”操作，实现了全局使用一份状态变量的效果。

下面我们解释如何编码：

1、首先这棵树除了根结点和叶子结点以外，每一个结点做的事情其实是一样的，即在已经选了一些数的前提，我们需要在剩下还没有选择的数中按照顺序依次选择一个数，这显然是一个递归结构；

2、递归的终止条件是，数已经选够了，因此我们需要一个变量来表示当前递归到第几层，我们把这个变量叫做 `depth`；

3、这些结点实际上表示了搜索（查找）全排列问题的不同阶段，为了区分这些不同阶段，我们就需要一些变量来记录为了得到一个全排列，程序进行到哪一步了，在这里我们需要两个变量：

（1）已经选了哪些数，到叶子结点时候，这些已经选择的数就构成了一个全排列；
（2）一个布尔数组 `used`，初始化的时候都为 `false` 表示这些数还没有被选择，当我们选定一个数的时候，就将这个数组的相应位置设置为 `true`，这样在考虑下一个位置的时候，就能够以 $O(1)$ 的时间复杂度判断这个数是否被选择过，这是一种“以空间换时间”的思想。

我们把这两个变量称之为“状态变量”，它们表示了我们在求解一个问题的时候所处的阶段。

4、在非叶子结点处，产生不同的分支，这一操作的语义是：在还未选择的数中依次选择一个元素作为下一个位置的元素，这显然得通过一个循环实现。

5、另外，因为是执行深度优先遍历，从较深层的结点返回到较浅层结点的时候，需要做“状态重置”，即“回到过去”、“恢复现场”，我们举一个例子。

从 `[1, 2, 3]` 到 `[1, 3, 2]`，深度优先遍历是这样做的，从 `[1, 2, 3]` 回到 `[1, 2]` 的时候，需要撤销刚刚已经选择的数 `3`，因为在这一层只有一个数 `3` 我们已经尝试过了，因此程序回到上一层，需要撤销对 `2` 的选择，好让后面的程序知道，选择 `3` 了以后还能够选择 `2`。

这种在遍历的过程中，从深层结点回到浅层结点的过程中所做的操作就叫“回溯”。

yes的练级攻略

46 具体代码

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> permute(int[] nums) {
        if (nums.length == 0) return res;
        List<Integer> path = new ArrayList<>(nums.length);
        boolean[] used = new boolean[nums.length];
        dfs(nums, 0, path, used);
        return res;
    }
    public void dfs (int[] nums, int depth, List<Integer> path, boolean
[] used){
        if (depth == nums.length) {
            res.add(new ArrayList<>(path));
            return;
        }
```

```
            for (int i = 0; i < nums.length; i++) {
                if (!used[i]) {
                    path.add(nums[i]);
                    used[i] = true;
                    dfs(nums, depth+1, path, used);
                    // 注意：这里是状态重置，是从深层结点回到浅层结点的过程，代码在形式上和递归之前是对称的
                    used[i] = false;
                    path.remove(path.size() - 1);
                }
            }
        }
}
```

9、无重复字符串的排列组合（面试题 08.07）

面试题 08.07. 无重复字符串的排列组合

难度 中等 20 收藏 分享 切换为英文 关注 反馈

无重复字符串的排列组合。编写一种方法，计算某字符串的所有排列组合，字符串每个字符均不相同。

示例1:

```
输入: S = "qwe"
输出: ["qwe", "qew", "wqe", "weq", "ewq", "eqw"]
```

示例2:

```
输入: S = "ab"
输出: ["ab", "ba"]
```

提示:

1. 字符都是英文字母。
2. 字符串长度在[1, 9]之间。

https://leetcode-cn.com/problems/permutation-i-lcci/solution/su-du-shao-man-hui-su-fa-18ms-75-100-by-wei-yu-13/ 的评论区

```
class Solution {
    List<String> result = new ArrayList();

    public String[] permutation(String S) {
        if (S == null || S.length() == 0) {
            return new String[0];
        }
        boolean[] isUsed = new boolean[S.length()];
        StringBuilder stb = new StringBuilder();
        dfs(S, isUsed, stb);
        return result.toArray(new String[result.size()]);
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    }

    /**
     * @param S       被处理的字符串,即permutation(.)传入参数S
     * @param isUsed  该数组标记字符串S对应位是否被使用
     * @param stb     中间量,临时保存元素
     */
    private void dfs(String S, boolean[] isUsed, StringBuilder stb) {
        if (stb.length() == S.length()) {
            result.add(stb.toString());
            return;
        }

        for (int i = 0; i < S.length(); i++) {
            if (!isUsed[i]) {
                //使用当前元素
                stb.append(S.charAt(i));
                isUsed[i] = true;

                //递归调用
                dfs(S, isUsed, stb);

                //撤销当前操作
                stb.deleteCharAt(stb.length() - 1);
                isUsed[i] = false;
            }
        }
    }
}
```

10、全排序二（47）

更多硬核内容就在公众号「yes 的练级攻略」

47. 全排列 II

难度 中等 321 收藏 分享 切换为英文

给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例:

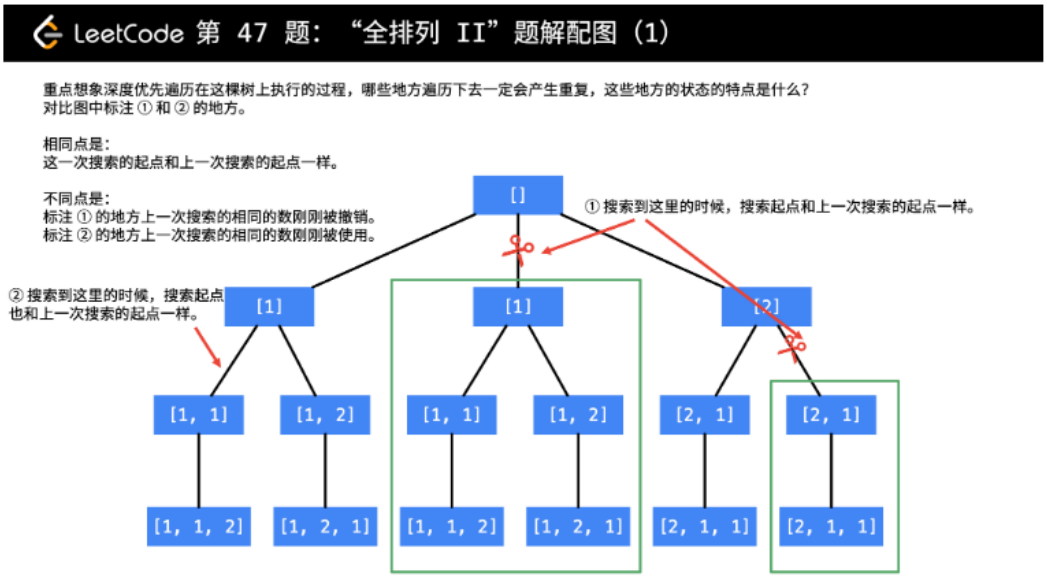
```
输入: [1,1,2]
输出:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

参考：
https://leetcode-cn.com/problems/permutations-ii/solution/hui-su-suan-fa-python-dai-ma-java-dai-ma-by-liwe-2/

思路：**在一定会产生重复结果集的地方剪枝。**

一个比较容易想到的办法是在结果集中去重。但是问题又来了，这些结果集的元素是一个又一个列表，对列表去重不像用哈希表对基本元素去重那样容易。

如果要比较两个列表是否一样，一个很显然的办法是分别排序，然后逐个比对。既然要排序，我们可以**在搜索之前就对候选数组排序**，一旦发现这一支搜索下去可能搜索到重复的元素就停止搜索，这样结果集中不会包含重复元素。



产生重复结点的地方，正是图中标注了"剪刀"，且被绿色框框住的地方。

大家也可以把第 2 个 `1` 加上 `'` ，即 `[1, 1', 2]` 去想象这个搜索的过程。只要遇到起点一样，就有可能产生重复。这里还有一个很细节的地方：

1、在图中 ② 处，搜索的数也和上一次一样，但是上一次的 `1` 还在使用中；
2、**在图中 ① 处，搜索的数也和上一次一样，但是上一次的 `1` 刚刚被撤销，正是因为刚被撤销，下面的搜索中还会使用到，因此会产生重复，剪掉的就应该是这样的分支。**

代码实现方面，在第 46 题的基础上，要加上这样一段代码：

Java

```
if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
    continue;
}
```

这段代码就能检测到标注为 ① 的两个结点，跳过它们。注意：这里 `used[i - 1]` 不加 `!` ，测评也能通过。有兴趣的朋友可以想一想这是为什么。建议大家做这样几个对比实验：

1、干脆就不写 `!used[i - 1]` 结果是什么样？
2、写 `used[i - 1]` 结果是什么，代码又是怎样执行的。这里给的结论是：`!used[i - 1]` 这样的剪枝更彻底。附录会分析原因。

复杂度分析：（理由同第 46 题，重复元素越多，剪枝越多。但是计算复杂度的时候需要考虑最差情况。）

- 时间复杂度：$O(N \times N!)$，这里 N 为数组的长度。
- 空间复杂度：$O(N \times N!)$。

47 具体代码

```
class Solution {
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> permuteUnique(int[] nums) {
        if (nums.length == 0) return res;
        Arrays.sort(nums); //一定要排序
        List<Integer> path = new ArrayList<>(nums.length);
        boolean[] used = new boolean[nums.length];
        dfs(nums, 0, path, used);
        return res;
    }
    public void dfs (int[] nums, int depth, List<Integer> path, boolean
[] used){
        if (depth == nums.length) {
            res.add(new ArrayList<>(path));
            return;
        }
        for (int i = 0; i < nums.length; ++i) {
            if (used[i]) continue;
            // 剪枝条件：i > 0 是为了保证 nums[i - 1] 有意义
            // 写 !used[i - 1] 是因为 nums[i - 1] 在深度优先遍历的过程中刚
刚被撤销选择
            if (i > 0 && nums[i] == nums[i-1] && !used[i-1]){
```

```
                continue;
            }
            path.add(nums[i]);
            used[i] = true;
            dfs(nums, depth+1, path, used);
            // 注意：这里是状态重置，是从深层结点回到浅层结点的过程，代码在形式上和递归之前是对称的
            used[i] = false;
            path.remove(path.size() - 1);
        }
    }
}
```

11、有重复字符串的排列组合（面试题 08.08）【字符串排序（剑指 38）】

面试题 08.08. 有重复字符串的排列组合

难度 中等 19 收藏 分享 切换为英文 关注 反馈

有重复字符串的排列组合。编写一种方法，计算某字符串的所有排列组合。

示例1:

```
输入: S = "qqe"
输出: ["eqq","qeq","qqe"]
```

示例2:

```
输入: S = "ab"
输出: ["ab", "ba"]
```

提示:

1. 字符都是英文字母。
2. 字符串长度在[1, 9]之间。

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 38. 字符串的排列

难度 中等 79 收藏 分享 切换为英文 关注 反馈

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

```
输入：s = "abc"
输出：["abc","acb","bac","bca","cab","cba"]
```

限制：

`1 <= s 的长度 <= 8`

https://leetcode-cn.com/problems/permutation-ii-lcci/solution/hui-su-suan-fa-jian-zhi-9980-10000-by-miracle-131/

```java
import java.util.List;
import java.util.Arrays;
class Solution {
    public String[] permutation(String S) {
        List<String> res = new ArrayList<String>();
        int len = S.length();
        if (len == 0) return new String[0];
        boolean[] used = new boolean[len];
        char[] sChar = S.toCharArray();

        StringBuilder path = new StringBuilder(len);
```

```
        // 排序是为了后面的剪枝
        Arrays.sort(sChar);

        dfs(res, sChar, len, path, 0, used);
        return res.toArray(new String[0]);
    }
    /**
    * @param res 结果集
    * @param sChar 输入字符数组
    * @param len 字符数组长度
    * @param path 根结点到任意结点的路径
    * @param depth 当前树的深度
    * @param used 使用标记数组
    */
    private void dfs(List<String> res
                    , char[] sChar
                    , int len
                    , StringBuilder path
                    , int depth
                    , boolean[] used) {
        // 到达叶子结点
        if (depth == len) {
            res.add(path.toString());
            return;
        }
        for (int i = 0; i < len; i++) {
            if (!used[i]) {
                // 根据已排序字符数组，剪枝
                if (i > 0 && sChar[i] == sChar[i-1] && !used[i-1]) {
                    continue;
                }

                path.append(sChar[i]);
                used[i] = true; // 标记选择
                dfs(res, sChar, len, path, depth+1, used);
                path.deleteCharAt(depth);
                used[i] = false; // 撤销选择
            }
        }
    }
}
```

12、字符串逆字典序排序（字节 1 面）

（1）有重复数据

```
import java.util.*;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
public class SamePermutation {
    public String[] noSamePermutation(String A, int n) {
        if (A == null || n == 0) return new String[0];
        boolean[] isUsed = new boolean[n];
        A = sortArray(A);
        System.out.println(A);
        StringBuilder stb = new StringBuilder();
        List<String> res = new ArrayList<>();
        dfs(A, isUsed, stb, res);
        return res.toArray(new String[res.size()]);
    }

    public void dfs (String A, boolean[] isUsed, StringBuilder stb,
List<String> res) {
        if (stb.length() == A.length()) {
            res.add(stb.toString());
            return;
        }
//      for (int i = A.length() - 1; i >= 0; i--) {
//          if (isUsed[i]) continue;
//          if (i < A.length() - 1 && A.charAt(i) == A.charAt(i + 1)
&& !isUsed[i+1]) {
//              continue;
//          }
//          stb.append(A.charAt(i));
//          isUsed[i] = true;
//          dfs(A, isUsed, stb, res);
//          isUsed[i] = false;
//          stb.deleteCharAt(stb.length() - 1);
//      }

        for (int i = 0; i < A.length(); i++) {
            if (isUsed[i]) continue;
            if (i > 0 && A.charAt(i) == A.charAt(i - 1) && !isUsed[i-1]) {
                continue;
            }
            stb.append(A.charAt(i));
            isUsed[i] = true;
            dfs(A, isUsed, stb, res);
            isUsed[i] = false;
            stb.deleteCharAt(stb.length() - 1);
        }
    }
```

禁止商用 公众号：yes的练级攻略

```
    public String sortArray(String A) {
        char[] tmp = A.toCharArray();
        Arrays.sort(tmp);
        //正确的方法 1
//      for (int i = 0; i < tmp.length / 2; i++) {
//          char tmp_one = tmp[i];
//          tmp[i] = tmp[tmp.length - 1 - i];
//          tmp[tmp.length - 1 - i] = tmp_one;
//      }
//      return String.valueOf(tmp);
        //正确的方法 2
//      String reverse = "";
//      for (int i = tmp.length - 1; i >= 0; i--) {
//          reverse += tmp[i];
//      }
//      return reverse;
        //正确的方法 3
        StringBuilder sb = new StringBuilder(String.valueOf(tmp));
        StringBuilder reverse = sb.reverse();
        return reverse.toString();
    }

    public static void main(String[] args) {
        while(true) {
            System.out.println("请输入数据:");
            Scanner sc = new Scanner(System.in);
            String A = sc.next();
            int n = A.length();
            SamePermutation test = new SamePermutation();
            String[] res = test.noSamePermutation(A, n);
            for (int i = 0; i < res.length; i++) {
                if (i < res.length - 1) {
                    System.out.print(res[i] + ",");
                } else {
                    System.out.print(res[i]);
                }
            }
            System.out.println("");
        }

    }
}
```

另外一种方法是直接抄面试题 08.08 代码，然后加一个双指针方法的反转数组

```
import java.util.*;
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
public class SamePermutation2 {
    public static String[] permutation(String S) {
            if (S == null || S.length() == 0) {
                return new String[0];
            }
            List<String> res = new ArrayList<>();
            //排序
            S = sortString(S);

            boolean[] isUsed = new boolean[S.length()];
            StringBuilder stb = new StringBuilder();
            dfs(S, isUsed, stb, res);
            return res.toArray(new String[res.size()]);
        }

        /**
         * @param S      被处理的字符串,即 permutation(.)传入参数 S
         * @param isUsed 该数组标记字符串 S 对应位是否被使用
         * @param stb    中间量,临时保存元素
         */
        private static void dfs(String S, boolean[] isUsed, StringBuilder
stb, List<String> res) {
            if (stb.length() == S.length()) {
                res.add(stb.toString());
                return;
            }

            for (int i = 0; i < S.length(); i++) {
                if (isUsed[i]) continue;
                if (i > 0 && S.charAt(i) == S.charAt(i - 1) && !isUsed[i -
1]) continue;
                //使用当前元素
                stb.append(S.charAt(i));
                isUsed[i] = true;

                //递归调用
                dfs(S, isUsed, stb,res);

                //撤销当前操作
                isUsed[i] = false;
                stb.deleteCharAt(stb.length() - 1);
            }
        }
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
        //对字符串进行重新排序
        private static String sortString (String before){
            char[] tmp = before.toCharArray();
            Arrays.sort(tmp);
            return String.valueOf(tmp);
        }

        //对字符串数组进行逆字典序排序
        private static void reverse(String[] strs) {
            int left = 0, right = strs.length - 1;
            while (left < right) {
                String tmp = strs[left];
                strs[left] = strs[right];
                strs[right] = tmp;
                left++;
                right--;
            }
        }

        public static void main(String[] args) {
            while(true) {
                System.out.println("请输入数据:");
                Scanner sc = new Scanner(System.in);
                String A = sc.nextLine();
                String[] res = permutation(A);
                reverse(res);
                for (int i = 0; i < res.length; i++) {
                    if (i < res.length - 1) {
                        System.out.print(res[i] + ",");
                    } else {
                        System.out.print(res[i]);
                    }
                }
                System.out.println("");
            }
        }

}
```

（2）无重复数据

```
import java.util.*;
public class NoSamePermutation {
    public String[] noSamePermutation(String A, int n) {
        if (A == null || n == 0) return new String[0];
```

禁止商用 公众号：yes的练级攻略

```
        boolean[] isUsed = new boolean[n];
        A = sortArray(A);
        StringBuilder stb = new StringBuilder();
        List<String> res = new ArrayList<>();
        dfs(A, isUsed, stb, res);
        return res.toArray(new String[res.size()]);
    }

    public void dfs (String A, boolean[] isUsed, StringBuilder stb,
List<String> res) {
        if (stb.length() == A.length()) {
            res.add(stb.toString());
            return;
        }

        for (int i = A.length() - 1; i >= 0; i--) {
            if (!isUsed[i]) {
                stb.append(A.charAt(i));
                isUsed[i] = true;
                dfs(A, isUsed, stb, res);
                stb.deleteCharAt(stb.length() - 1);
                isUsed[i] = false;
            }
        }
    }

    public String sortArray(String A) {
        char[] tmp = A.toCharArray();
        Arrays.sort(tmp);
        return String.valueOf(tmp);
    }
//  public static void main(String[] args) {
//      while(true) {
//          System.out.println("请输入数据:");
//          Scanner sc = new Scanner(System.in);
//          String A = sc.next();
//          int n = A.length();
//          NoSamePermutation test = new NoSamePermutation();
//          String[] res = test.noSamePermutation(A, n);
//          for (int i = 0; i < res.length; i++) {
//              if (i < res.length - 1) {
//                  System.out.print(res[i] + ",");
//              } else {
//                  System.out.print(res[i]);
```

禁止商用 公众号：yes的练级攻略

```
//             }
//         }
//     System.out.println("");
//     }
// }
}
```

13、第 k 个排序（60）

60. 第k个排列

难度 中等 262 收藏 分享 切换为英文 关注 反馈

给出集合 $[1,2,3,\cdots,n]$，其所有元素共有 $n!$ 种排列。

按大小顺序列出所有排列情况，并一一标记，当 $n = 3$ 时，所有排列如下：

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

给定 n 和 k，返回第 k 个排列。

说明：

- 给定 n 的范围是 [1, 9]。
- 给定 k 的范围是[1, $n!$]。

示例 1:

```
输入: n = 3, k = 3
输出: "213"
```

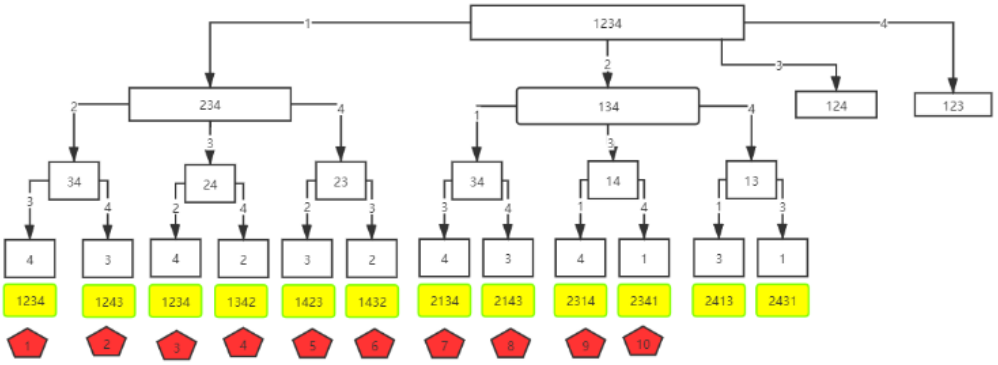
示例 2:

```
输入: n = 4, k = 9
输出: "2314"
```

https://leetcode-cn.com/problems/permutation-sequence/solution/pai-lie-zu-he-zhi-di-kge-pai-lie-golden-monkey-by-/

更多硬核内容就在公众号「yes 的练级攻略」

- 举例$n=4, k=9$的场景，下面的代码用到阶乘加速剪枝的过程
  - 当首先选1时，剩下的可选的数有234这三个，阶乘$f(4-1)=f(3)=3*2*1=6$，也就是说1后面与234组合最多有6种组合，但是题目要求的是$k=9$，这一层选完了也不可能到到k，需要跳过，并将$k-6=9-6=3$，下一层需要找到第3个排列
  - 当再次选2时，这时候2与后面的134也有6种组合，但是这次要选的是上面留下来的3，需要再进一层，选1，也就是前面是21，留下了34，阶乘$f(4-2)=f(2)=2*1=2$, 这里-2是因为用掉了两层21这两个，也就是两层，现在很明显了，21与剩下的34组合只有两种，2134和2143，而2又小于上一步留下来的$k=3$，需要跳过，$k-2=3-2=1$，下一层需要找到第1个排列
  - 退回上一层选3,也就是当前是23，剩下了14，这时候有$f(4-2)=f(2)=2*1=2$种选择，目标出现了，因为$2>k=1$,选1得到231,这时$f(4-3)=f(1)=1$, 也就是说2314即所求，到底此时，因为长度已经到达n，开始收集结果



- 其中的$depth$可以理解成层级，当到达n层，即达到出口条件，每一层dfs需要在当前的$depth+1$,进入下一层

需要 8ms

```
class Solution {
    public String getPermutation(int n, int k) {
        int[] nums = new int[n];
        for (int i = 0; i < n; i++) {
            nums[i] = i+1;
        }
        boolean[] used = new boolean[n];
        List<String> levelList = new ArrayList<String>();
        return dfs(nums, levelList, used, 0, n, k);
    }
    /**
     * @param nums      源数组
     * @param levelList 每一层的选择
     * @param used      数组的元素是否被使用过
     * @param depth     深度，也就是当前使用的元素的索引
     * @param n         上限值
     * @param k         第 k 个
     * @return 第 k 个排列
     */
    private String dfs(int[] nums, List<String> levelList, boolean[] us
ed, int depth, int n, int k){
        if (depth == n) {
            StringBuilder res = new StringBuilder();
```

```
            for (String s : levelList) {
                res.append(s);
            }
            return res.toString();
        }
        int cur = factorial(n - depth - 1);//当前的 depth 也就是索引，有多少排列数
        for (int i = 0; i < n; i++) {
            //当前元素被使用过，做剪枝
            if (used[i]) continue;
            //当前的排列组合数小于 k，说明就算这一层排完了，也到不了第 k 个，剪枝
            if (cur < k) {
                k -= cur;
                continue;
            }
            levelList.add(nums[i] + ""); //list 收的是 string 类型
            used[i] = true;//当前元素被使用过标记
            return dfs(nums, levelList, used, depth + 1, n, k);
        }
        return null;
    }
    //返回 n 的阶乘，如 3!=3*2*1=6
    private int factorial(int n) {
        int res = 1;
        while (n > 0) {
            res *= n--;
        }
        return res;
    }
}
```

另外一种思路：

https://leetcode-cn.com/problems/permutation-sequence/solution/zan-neng-bu-hui-su-zan-jiu-jin-liang-bu-hui-su-by-/

大致思路：
k不断减小，减小到1的时候 数组的顺序就是要的结果
减小的过程是这样的：比如123456789，
当我们k=123时，k>5的阶乘120，我们把倒数第6个数字4 和后面第一个比他大的数字5交换位置得到123546789，这样我们就直接从最小的跳到第121个小的了，此时k=3；k>2的阶乘也就是2，我们把我们把倒数第3个数字7 和后面第一个比他大的数字8交换位置得到123546879，这样我们就直接从第121个的跳到第123个小的了，此时k=1；返回结果

```
class Solution {
    public String getPermutation(int n, int k) {
        int []factorial=new int[n+1];//存放阶乘
        int []nums=new int[n];//存放 1-n 个数字
        for (int i = 0; i <n; i++) {
```

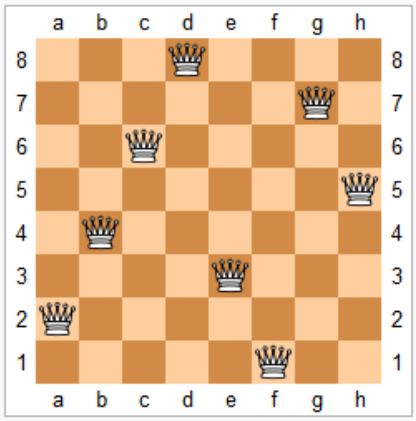
```
            nums[i]=i+1;
        }
        factorial[0]=1;
        for (int i = 1; i < n+1; i++) {//把 1 到 9 的阶乘先全都算出来
            factorial[i]=i*factorial[i-1];
        }
        StringBuilder sb=new StringBuilder();//用来保存结果的
        if (k==factorial[n]){//这是一个特殊情况 后面的 while 循环没法处
理  所以单独拎出来了
            for (int i = n-1; i >=0 ; i--) {
                sb.append(nums[i]);
            }
            return sb.toString();
        }
        int n2=n-1;
        while (k>1){
            if (k>factorial[n2]){
                k-=factorial[n2];
                int tmp=nums[n-1-n2];//要进行交换的位置
                for (int i = n-n2; i <n ; i++) {//找到第一个比当前位置的的
                    if (nums[i]>tmp){//交换
                        nums[n-1-n2]=nums[i];
                        nums[i]=tmp;
                        break;
                    }
                }
            }
            if (k<=factorial[n2])n2--;
        }
        for (int i = 0; i <n ; i++) {
            sb.append(nums[i]);
        }
        return sb.toString();
    }
}
```

14、N 皇后（51）

51. N皇后

难度 困难 447 收藏 分享 切换为英文 关注 反馈

n 皇后问题研究的是如何将 *n* 个皇后放置在 *n*×*n* 的棋盘上，并且使皇后彼此之间不能相互攻击。



One solution to the eight queens puzzle

上图为 8 皇后问题的一种解法。

给定一个整数 *n*，返回所有不同的 *n* 皇后问题的解决方案。

每一种解法包含一个明确的 *n* 皇后问题的棋子放置方案，该方案中 `'Q'` 和 `'.'` 分别代表了皇后和空位。

示例:

```
输入: 4
输出: [
 [".Q..",  // 解法 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // 解法 2
  "Q...",
  "...Q",
  ".Q.."]
]
解释: 4 皇后问题存在两个不同的解法。
```

https://leetcode-cn.com/problems/n-queens/solution/gen-ju-di-46-ti-quan-pai-lie-de-hui-su-suan-fa-si-/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

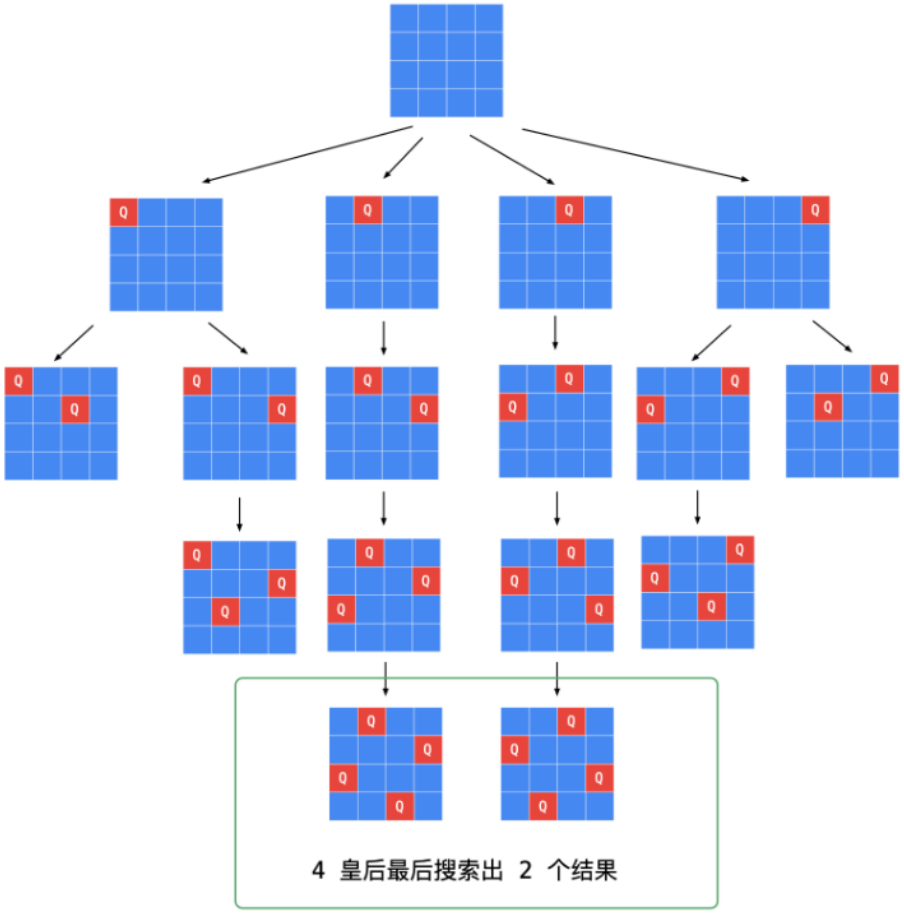
一句话题解：回溯算法是一种遍历算法，以 **深度优先遍历** 的方式尝试所有的可能性。有些教程上也叫「暴力搜索」。回溯算法是 **有方向地** 搜索，区别于多层循环实现的暴力法。

> 许多复杂的、规模较大的问题都可以使用回溯法，有「通用解题方法」的美称。 (来自 百度百科)

说明：

- N 皇后问题很多时候作为例题出现在教科书中，可以当做理解回溯算法的例题进行学习；
- 对于回溯算法还比较陌生的朋友，可以参考我的题解《回溯算法入门级详解 + 练习（持续更新）》。

先尝试画出递归树，以 4 皇后问题为例，画出的递归树如下：



更多硬核内容就在公众号「yes 的练级攻略」

搜索的过程蕴含了 **剪枝** 的思想。「剪枝」的依据是：题目中给出的 「N 皇后」 的摆放规则：1、不在同一行；2、不在同一列；3、不在同一主对角线方向上；4、不在同一副对角线方向上。

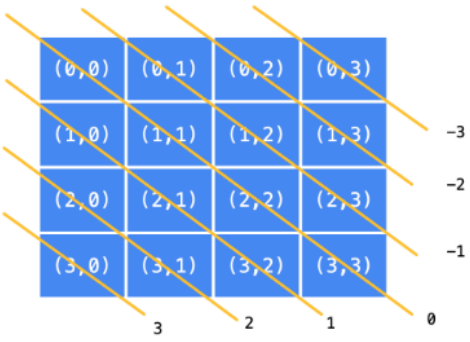
小技巧：记住已经摆放的皇后的位置

这里记住已经摆放的位置不能像 Flood Fill 一样，简单地使用 `visited` 布尔数组。放置的规则是：一行一行考虑皇后可以放置在哪一个位置上，某一行在考虑某一列是否可以放置皇后的时候，需要根据前面已经放置的皇后的位置。

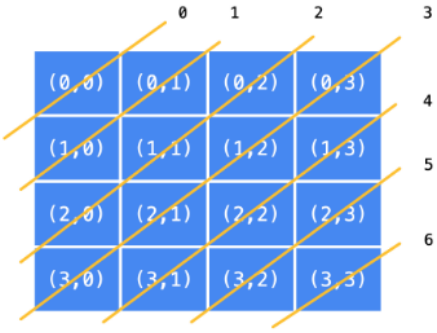
由于是一行一行考虑放置皇后，摆放的这些皇后肯定不在同一行，为了避免它们在同一列，需要一个长度为 N 的布尔数组 `cols`，已经放置的皇后占据的列，就需要在对应的列的位置标注为 `True`。

考虑对角线（找规律）

下面我们研究一下主对角线或者副对角线上的元素有什么特性。在每一个单元格里写下行和列的 **下标**。



主对角线上的元素的特点：（横坐标 − 纵坐标）的值固定。



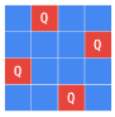
副对角线上的元素的特点：横纵坐标之和固定。

为了保证至少两个皇后不同时出现在 **同一主对角线方向** 或者 **同一副对角线方向**。检查策略是，只要「检测」到新摆放的「皇后」与已经摆放好的「皇后」冲突，就尝试摆放同一行的下一个位置，到行尾还不能放置皇后，**就退回到上一行**。

可以像全排列 `used` 数组那样，再为 「主对角线（Main diagonal）」 和 「副对角线（Sub diagonal）」 设置相应的 **布尔数组变量**，只要排定一个 「皇后」 的位置，就需要占住对应的位置。

编码

我们使用一个 1 到 4 的排列表示一个 4×4 的棋盘，例如：



[2, 4, 1, 3]

[3, 1, 4, 2]

第 1 行，第 2 列摆放皇后
第 2 行，第 4 列摆放皇后
第 3 行，第 1 列摆放皇后
第 4 行，第 3 列摆放皇后

得到一个符合要求的全排列以后，生成棋盘的代码就很简单了。

说明：

- 将「行状态」、「主对角线状态」、「副对角线状态」 设置为成员变量，以避免递归方法参数冗长（参考代码 2、参考代码 3 类似看待）。至于是否有必要这样做，以后在项目开发中需要遵守项目文档的规定；
- Java 中 `Stack` 已经废弃，推荐使用 `ArrayDeque`，可以查阅文档。

方法：回溯搜索算法（深度优先遍历）

下面虽然给出了 3 版代码，但都只是在如何记住已经摆放的皇后的位置上做文章，代码结构都一样，大家只看不同的部分就可以了。

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.List;
public class Solution {
    private int n;
    // 记录某一列是否放置了皇后
    private boolean[] col;
    // 记录主对角线上的单元格是否放置了皇后
    private boolean[] sub;
    // 记录了副对角线上的单元格是否放置了皇后
    private boolean[] main;
    private List<List<String>> res;
    public List<List<String>> solveNQueens(int n) {
        res = new ArrayList<>();
        if (n == 0) {
            return res;
        }
        // 设置成员变量，减少参数传递，具体作为方法参数还是作为成员变量，请参考团队开发规范
        this.n = n;
        this.col = new boolean[n];
        this.sub = new boolean[2 * n - 1];
        this.main = new boolean[2 * n - 1];
        Deque<Integer> path = new ArrayDeque<>();
        dfs(0, path);
        return res;
    }
    private void dfs(int row, Deque<Integer> path) {
        if (row == n) {
            // 深度优先遍历到下标为 n，表示 [0.. n - 1] 已经填完，得到了一个结果
            List<String> board = convert2board(path);
            res.add(board);
            return;
        }
        // 针对下标为 row 的每一列，尝试是否可以放置
        for (int j = 0; j < n; j++) {
            if (!col[j] && !sub[row + j] && !main[row - j + n - 1]) {
                path.addLast(j);
                col[j] = true;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                sub[row + j] = true;
                main[row - j + n - 1] = true;

                dfs(row + 1, path);

                main[row - j + n - 1] = false;
                sub[row + j] = false;
                col[j] = false;
                path.removeLast();
            }
        }
    }
    private List<String> convert2board(Deque<Integer> path) {
        List<String> board = new ArrayList<>();
        for (Integer num : path) {
            StringBuilder row = new StringBuilder();
            row.append(".".repeat(Math.max(0, n)));
            row.replace(num, num + 1, "Q");
            board.add(row.toString());
        }
        return board;
    }
}
```

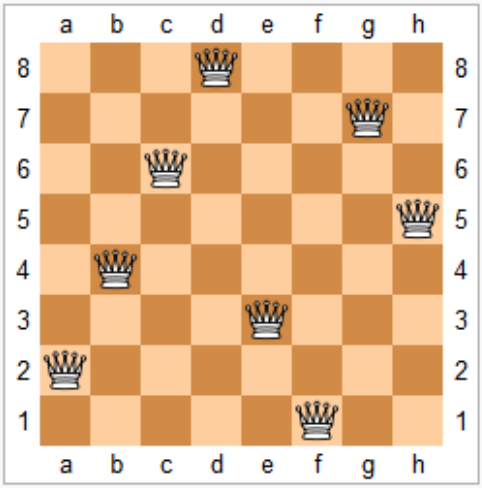
15、N 皇后 2（52）

公众号：yes的练级攻略

52. N皇后 II

难度 困难 126 收藏 分享 切换为英文 关注 反馈

n 皇后问题研究的是如何将 *n* 个皇后放置在 *n*×*n* 的棋盘上，并且使皇后彼此之间不能相互攻击。



One solution to the eight queens puzzle

上图为 8 皇后问题的一种解法。

给定一个整数 *n*，返回 *n* 皇后不同的解决方案的数量。

将上题的思路改成计数就行了

```
class Solution {
    private boolean[] col;
    private boolean[] master;
    private boolean[] slave;
    private int count = 0;
    public int totalNQueens(int n) {
        if (n == 0) return count;
        col = new boolean[n];
        master = new boolean[2*n - 1];
        slave = new boolean[2*n - 1];
        dfs(0, n);
        return count;
    }
    private void dfs(int row, int n){
        if (row == n) {
            count++;
            return;
        }
        for (int i = 0; i < n; i++) {
            if (!col[i] && !master[row+i] && !slave[row-i+n-1]) {
                col[i] = true;
```

```
                master[row+i] = true;
                slave[row-i+n-1] = true;
                dfs(row+1, n);
                slave[row-i+n-1] = false;
                master[row+i] = false;
                col[i] = false;
            }
        }
    }
}
```

16、八皇后（面试题 08.12）

面试题 08.12. 八皇后

难度 困难 24 收藏 分享 切换为英文 关注 反馈

设计一种算法，打印 N 皇后在 N × N 棋盘上的各种摆法，其中每个皇后都不同行、不同列，也不在对角线上。这里的“对角线”指的是所有的对角线，不只是平分整个棋盘的那两条对角线。

注意： 本题相对原题做了扩展

示例:

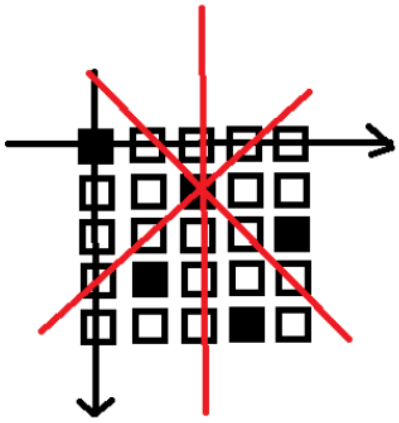
```
 输入：4
 输出：[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
 解释：4 皇后问题存在如下两个不同的解法。
[
 [".Q..",  // 解法 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // 解法 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

YES的练级攻略

https://leetcode-cn.com/problems/eight-queens-lcci/solution/java-hui-su-fa-shuang-100-san-shu-zu-ji-lu-dui-jia/

解题思路

每一个皇后的出现，导致对角线和反对角线，以及十字线内不能出现皇后，由于使用了回溯，逻辑上避免了同一行出现皇后，
所以只需要记录出现过的对角线和反对角线以及，出现的Column就行了



```java
class Solution {
    private char[][] pos;
    private int n;
    private int[] col;
    private int[] rcSum; //行和列的差   //对角线为y = x + n  (n 为一个有关上下文的数)
    private int[] rcDis; //行和列的和   //反对角线为y = -x + m （m 为一个有关上下文的数）
    private List<List<String>> res = new ArrayList<>();
    public List<List<String>> solveNQueens(int n) {
        this.n = n;
        if ( n == 0) return res;
        pos = new char[n][n];
        col = new int[n];
        rcSum = new int[2*n-1];
        rcDis = new int[2*n-1];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                pos[i][j] = '.';
            }
        }
        dfs(0);
        return res;
    }
    private void dfs (int row) {
        if (row == n) return;
        for (int column = 0; column < n; column++) {
            boolean canInsert = checkSet(row, column);
            if (canInsert) {
                putOrRemoveSetElement(row, column, true);
                pos[row][column] = 'Q';
                if (row + 1 == n) {
                    List<String> path = new ArrayList<>();
                    for (int i = 0; i < n; i++) {
                        StringBuilder sb = new StringBuilder();
                        for (int j = 0; j < n; j++) {
                            sb.append(pos[i][j]);
                        }
                        path.add(sb.toString());
                    }
                    res.add(path);
                } else {
                    dfs(row+1);
                }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                pos[row][column] = '.';
                putOrRemoveSetElement(row, column, false);
            }
        }
    }
    private boolean checkSet(int row, int column) {
        if (col[column] == 1) return false;
        else if (rcSum[row + column] == 1) return false;
        else if (rcDis[n - 1 - row + column] == 1) return false;
        else return true;
    }
    private void putOrRemoveSetElement(int row, int column, boolean isP
ut) {
        if (isPut) {
            col[column] = 1;
            rcSum[row + column] = 1;
            rcDis[n - 1 - row + column] = 1;
        } else {
            col[column] = 0;
            rcSum[row + column] = 0;
            rcDis[n - 1 - row + column] = 0;
        }
    }
}
```

17、单词矩阵（面试题 17.25）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

面试题 17.25. 单词矩阵

难度 困难 6 收藏 分享 切换为英文 关注 反馈

给定一份单词的清单，设计一个算法，创建由字母组成的面积最大的矩形，其中每一行组成一个单词(自左向右)，每一列也组成一个单词(自上而下)。不要求这些单词在清单里连续出现，但要求所有行等长，所有列等高。

如果有多个面积最大的矩形，输出任意一个均可。一个单词可以重复使用。

示例 1:

```
输入: ["this", "real", "hard", "trh", "hea", "iar", "sld"]
输出:
[
   "this",
   "real",
   "hard"
]
```

示例 2:

```
输入: ["aa"]
输出: ["aa","aa"]
```

说明:

- `words.length <= 1000`
- `words[i].length <= 100`
- 数据保证单词足够随机

https://leetcode-cn.com/problems/word-rectangle-lcci/solution/jian-dan-dai-ma-shi-xian-zi-dian-shu-hui-su-jian-z/

简单代码实现字典树+回溯，剪枝提速

宁宁 发布于 2020-05-02 767 Java 字典树 回溯算法

回溯算法通常有一个"套路"，经典问题有全排列和八皇后。用伪代码表示出来：

```
function backTrack(供选择的列表，路径){
    if(终止条件){
        退出;
    }
    做选择;  //从列表选择一项加入路径
    backTrack(列表，路径);
    撤销选择;
}
```

这题的思路就是通过回溯来构成单词矩阵，构造字典树用来快速判断形成的单词是否在清单里。

现在这么做已经没问题了，但提交上去会有超时，需要给回溯做一个剪枝来优化一下。代码：

```
class Solution {
    class Trie{
        Trie[] childs;
        boolean isLeaf;
        public Trie(){
            childs = new Trie[26];
        }
    }
```

```
Trie root;
Map<Integer, Set<String>> map;  //把清单根据单词长度集合起来
int maxArea;
int maxLength;
List<String> ans;   //别忘最后转换成数组输出
public String[] maxRectangle(String[] words) {
    root = new Trie();
    //构造字典树
    for(String str: words){
        Trie node = root;
        for(char c: str.toCharArray()){
            if(node.childs[c-'a'] == null){
                node.childs[c-'a'] = new Trie();
            }
            node = node.childs[c-'a'];
        }
        node.isLeaf = true;
    }

    map = new HashMap<>();
    ans = new ArrayList<>();
    maxArea = 0;
    maxLength = 0;
    for(String w: words){
        int len = w.length();
        maxLength = Math.max(maxLength, len);
        Set<String> set = map.getOrDefault(len, new HashSet<>());
        set.add(w);
        map.put(len, set);
    }

    List<String> path = new ArrayList<>();
    for(int key: map.keySet()){
        path.clear();
        //回溯需要的参数是：相同长度单词的集合，存放路径的列表，当前单词的长度
        dfs(map.get(key), path, key);
    }

    String[] ultimate = new String[ans.size()];
    return ans.toArray(ultimate);
}

//回溯的“套路”
public void dfs(Set<String> dic, List<String> path, int wordLen){
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        //剪枝，dic 里的情况不可能得到最优解，提前过滤掉不考虑
        if(wordLen*maxLength <= maxArea)   return;

        //终止条件：如果 path 矩阵的高度已经超过清单中最长单词长度，结束
        if(path.size() > maxLength) return;

        for(String str: dic){
            //做选择
            path.add(str);

            boolean[] res = isValid(path);
            if(res[0]){ //下面还可以再加
                int area = path.size()*path.get(0).length();
                if(res[1] && (area>maxArea)){   //每列都是单词的矩阵
                    maxArea = area;
                    //ans = path;    一定注意这里不能直接把 path 引用交给答案
                    ans = new ArrayList<>(path);
                }
                //回溯
                dfs(dic, path, wordLen);
            }
            //撤销选择
            path.remove(path.size()-1);
        }
    }

    /** 判断一个矩阵是否每一列形成的单词都在清单里
     *   存在两种情况：1.有的列中的字母不在字典树中，即这一列不可能构成单词，整个矩阵不
合要求
     *   2.每列的所有字母都在字典树中但有的结尾不是 leaf，也就是有的列目前还不是个单词
     *   所以需要一个 boolean 数组 res[]来存放结果：
     *   res[0]表示是否有字母不在字典树中，true:都在，false:有不在的
     *   res[1]表示是否所有的列都构成了清单里的单词
     */
    public boolean[] isValid(List<String> input){
        boolean[] res = new boolean[2];
        boolean allLeaf = true;
        int m = input.size();
        int n = input.get(0).length();
        for(int j=0; j<n; j++){
            //按列来看单词是否在字典树
            Trie node = root;
            for(int i=0; i<m; i++){
                int c = input.get(i).charAt(j) - 'a';
```

```
                if(node.childs[c] == null)  return new boolean[]{false, false};
                node = node.childs[c];
            }
            if(!node.isLeaf)    allLeaf = false;
        }
        return new boolean[]{true, allLeaf};
    }
}
```

18、字母大小写全排序（784）

784. 字母大小写全排列

难度 简单 177 收藏 分享 切换为英文 关注 反馈

给定一个字符串 `S`，通过将字符串 `S` 中的每个字母转变大小写，我们可以获得一个新的字符串。返回所有可能得到的字符串集合。

示例：
输入： `S = "a1b2"`
输出： `["a1b2", "a1B2", "A1b2", "A1B2"]`

输入： `S = "3z4"`
输出： `["3z4", "3Z4"]`

输入： `S = "12345"`
输出： `["12345"]`

注意：

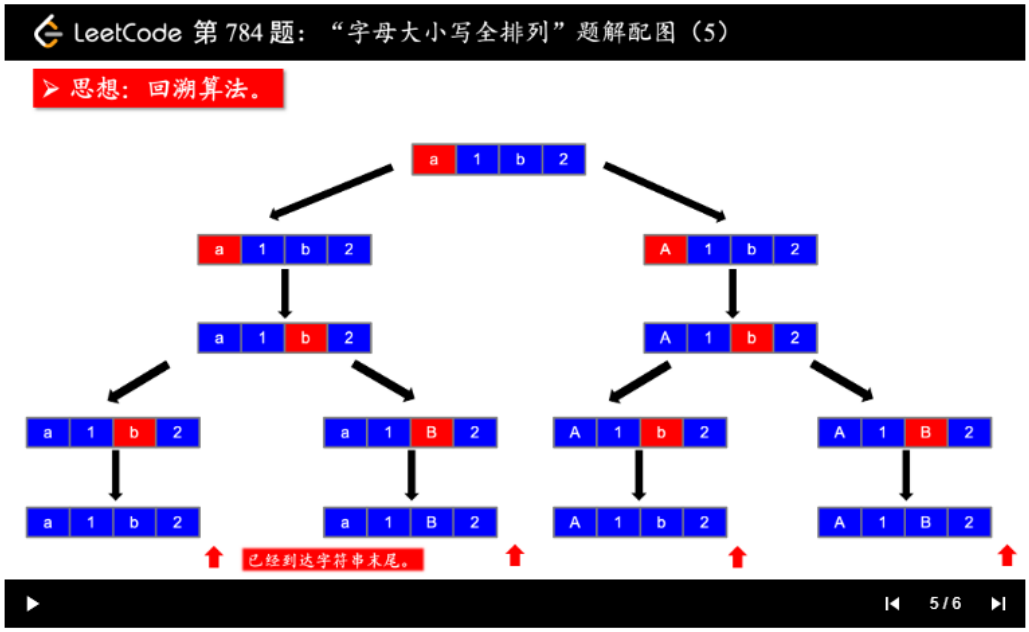
- `S` 的长度不超过 `12`。
- `S` 仅由数字和字母组成。

参考：
https://leetcode-cn.com/problems/letter-case-permutation/solution/shen-du-you-xian-bian-li-hui-su-suan-fa-python-dai/

思路分析：

- 这一类搜索问题是在一个隐式的树上进行的搜索问题，即“树形问题”。因此，**先画出递归树是十分重要的**，然后看着图形把代码写出来；
- 问题所求的解，就是这棵树的叶子结点上的值，因此，可以使用深度优先遍历，得到叶子结点的值，这样的算法也叫回溯算法；
- 回溯算法的显著特征是“状态重置”，而由于字符串在拼接的过程中，每次都产生新的字符串，因此，在叶子结点处，直接保存结果即可，并且从深层结点回到浅层结点的过程中，无需状态重置，因为每一步的装填都是新的（“回溯”算法的入门问题是「力扣」第 46 题：“全排列”）。

更多硬核内容就在公众号「yes 的练级攻略」



另外说一下，回溯的问题，这道题居然被标为“简单”，也是非常神奇的一件事情，不管你信不信，反正我觉得没那么简单。

思路搞定了，下面要介绍一个技巧，即解决字母大小写转换的问题。

公众号：
yes的练级攻略

2、技巧：使用异或运算转换字母大小写

这一步比较有技巧，我也是参考了别人的思路，用自己话写了出来。

我们先看一看 ASCII 表，A 到 Z，Z 完了以后没有直接到 a，中间隔了 6 个字符。

0100 0001	0101	65	0x41	A	大写字母A
0100 0010	0102	66	0x42	B	大写字母B
0100 0011	0103	67	0x43	C	大写字母C
0100 0100	0104	68	0x44	D	大写字母D
0100 0101	0105	69	0x45	E	大写字母E
0100 0110	0106	70	0x46	F	大写字母F
0100 0111	0107	71	0x47	G	大写字母G

（中间省略）

☆ 收藏

0101 1010	0132	90	0x5A	Z	大写字母Z
0101 1011	0133	91	0x5B	[	开方括号
0101 1100	0134	92	0x5C	\	反斜杠
0101 1101	0135	93	0x5D	]	闭方括号
0101 1110	0136	94	0x5E	^	脱字符
0101 1111	0137	95	0x5F	_	下划线
0110 0000	0140	96	0x60	`	开单引号
0110 0001	0141	97	0x61	a	小写字母a
0110 0010	0142	98	0x62	b	小写字母b
0110 0011	0143	99	0x63	c	小写字母c
0110 0100	0144	100	0x64	d	小写字母d
0110 0101	0145	101	0x65	e	小写字母e
0110 0110	0146	102	0x66	f	小写字母f
0110 0111	0147	103	0x67	g	小写字母g

我们发现大写字符与其对应的小写字符的 ASCII 的差为 32，32 这个值如果敏感的话，它是 2^5，在编程语言中，可以表示为 `1 << 5`。而

变换大小写这件事等价于：

1、如果字符是小写字符，减去 32 得到大写字符；
2、如果字符是大写字符，加上 32 得到小写字符。

而这两者合并起来，就是给这个字符做一次不进位的加法，即异或上 `1 << 5`。

784 具体代码

```
class Solution {
    List<String> res = new ArrayList<>();
    public List<String> letterCasePermutation(String S) {
        if (S == null && S.length() == 0) return res;
        char [] arr = new char[S.length()];
        dfs(S,0,arr);
        return res;
    }
    private void dfs(String S, int start, char[] arr){
```

```
            if (start == S.length()) {
                res.add(new String(arr));
                return;
            }
            arr[start] = S.charAt(start);
            dfs (S, start + 1, arr);
            if (Character.isLetter(S.charAt(start))){
                //(1<<5)是32， 如果字符是小写字符，减去 32 得到大写字符；如果字符是大写字符，加上 32 得到小写字符。通过异或可以实现大小写转换，举个例子：'A' 0100 0001，‘a’ 0110 0001
                arr[start] = (char) (S.charAt(start) ^ (1<<5));
                dfs (S, start + 1, arr);
            }
        }
    }
```

19、子集（78）

78. 子集

难度 中等 657 收藏 分享 切换为英文 关注 反馈

给定一组**不含重复元素**的整数数组 *nums*，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

```
输入: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

https://leetcode-cn.com/problems/subsets/solution/zhi-xing-yong-shi-1-ms-zai-suo-you-java-ti-jia-227/

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new LinkedList<>();
        backtracks(nums, 0, new LinkedList<>(), result);
        return result;
    }

    //递归遍历数组，每到一个元素就有两种情况：1.选该元素 2.不选该元素
```

```
    public void backtracks(int[] nums, int current, LinkedList<Integer>
 track, List<List<Integer>> result) {
        //已经遍历完所有元素
        if (current == nums.length) {
            result.add(new LinkedList<>(track));
            return;
        }
        //不选当前元素
        backtracks(nums, current + 1, track, result);
        //选当前元素
        track.add(nums[current]);
        backtracks(nums, current + 1, track, result);
        //撤销选取当前元素（回溯）
        track.removeLast();
    }
}
```

20、子集二（90）【幂集（面试题 08.04）】

给定一个可能包含重复元素的整数数组 ***nums***，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例：

```
输入: [1,2,2]
输出:
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

https://leetcode-cn.com/problems/subsets-ii/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-19/

方法 1：dfs

```
class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(nums); //排序
        getAns(nums, 0, new ArrayList<>(), ans);
        return ans;
    }
```

```
    private void getAns(int[] nums, int start, ArrayList<Integer> temp, List<Li
st<Integer>> ans) {
        ans.add(new ArrayList<>(temp));
        for (int i = start; i < nums.length; i++) {
            //和上个数字相等就跳过
            if (i > start && nums[i] == nums[i - 1]) {
                continue;
            }
            temp.add(nums[i]);
            getAns(nums, i + 1, temp, ans);
            temp.remove(temp.size() - 1);
        }
    }
}
```

方法 2：bfs

```
class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        List<List<Integer>> ans = new ArrayList<>();
        ans.add(new ArrayList<>());// 初始化空数组
        Arrays.sort(nums);
        int start = 1; //保存新解的开始位置
        for (int i = 0; i < nums.length; i++) {
            List<List<Integer>> ans_tmp = new ArrayList<>();
            // 遍历之前的所有结果
            for (int j = 0; j < ans.size(); j++) {
                List<Integer> list = ans.get(j);
                //如果出现重复数字，就跳过所有旧解
                if (i > 0 && nums[i] == nums[i - 1] && j < start) {
                    continue;
                }
                List<Integer> tmp = new ArrayList<>(list);
                tmp.add(nums[i]); // 加入新增数字
                ans_tmp.add(tmp);
            }

            start = ans.size(); //更新新解的开始位置
            ans.addAll(ans_tmp);
        }
        return ans;
    }
}
```

21、复原 IP 地址（93）

更多硬核内容就在公众号「yes 的练级攻略」

93. 复原IP地址

难度 中等 301 收藏 分享 切换为英文 关注 反馈

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

有效的 IP 地址正好由四个整数（每个整数位于 0 到 255 之间组成），整数之间用 '.' 分隔。

示例:

```
输入："25525511135"
输出：["255.255.11.135", "255.255.111.35"]
```

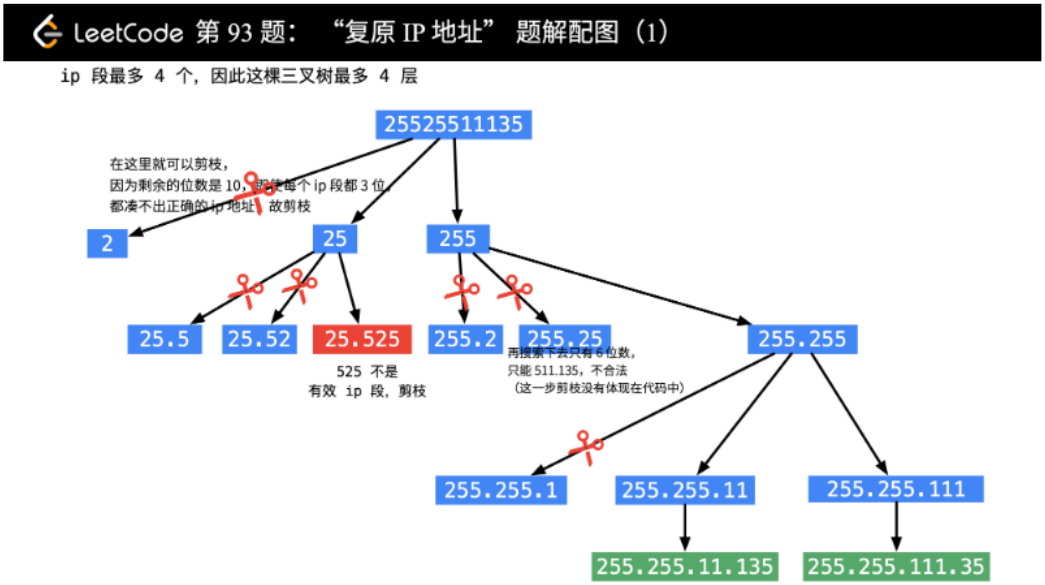
https://leetcode-cn.com/problems/restore-ip-addresses/solution/hui-su-suan-fa-hua-tu-fen-xi-jian-zhi-tiao-jian-by/

禁止商用
公众号：
yes的练级攻略

回溯算法事实上就是在一个树形问题上做深度优先遍历，因此 **首先需要把问题转换为树形问题**。这里请大家一定要拿起纸和笔，模拟一下如何通过指定的字符串 `s` 生成 IP 地址的过程，把树形图画出来（这一点很重要）。

下面这张图我没有画完（如果画完，枝叶太多），请读者尽量不看我画的这张图，自己动手尝试一下这个问题的树形图应该怎么画。

在画树形图的过程中，你一定会发现有些枝叶是没有必要的，把没有必要的枝叶剪去的操作就是剪枝，在代码中一般通过 `break` 或者 `contine` 和 `return` （表示递归终止）实现。



分析剪枝条件（下面只写出一些我想到的要点，有些点能想到，但是编码很复杂，我就没有写了）：

1、一开始，字符串的长度小于 4 或者大于 12 ，一定不能拼凑出合法的 ip 地址（这一点可以一般化到中间结点的判断中，以产生剪枝行为）；

2、每一个结点可以选择截取的方法只有 3 种：截 1 位、截 2 位、截 3 位，因此每一个结点可以生长出的分支最多只有 3 条分支；

根据截取出来的字符串判断是否是合理的 ip 段，这里写法比较多，可以先截取，再转换成 int ，再判断。我采用的做法是先转成 int，是合法的 ip 段数值以后，再截取。

3、由于 ip 段最多就 4 个段，因此这棵三叉树最多 4 层，这个条件作为递归终止条件之一；

4、每一个结点表示了求解这个问题的不同阶段，需要的状态变量有：

- `splitTimes` ：已经分割出多少个 ip 段；
- `begin` ：截取 ip 段的起始位置；
- `path` ：记录从根结点到叶子结点的一个路径（回溯算法常规变量，是一个栈）；
- `res` ：记录结果集的变量，常规变量。

> 总结：这个问题思想不难，但是细节比较繁琐，什么时候递归终止，如何手动截取字符串，再转换成 int 类型，还有如何在中间结点发现可以剪枝，这些细节需要在编码的时候考虑清楚。

有一些编码细节写在代码注释中，供大家参考，可能还有漏掉的地方，欢迎大家给出意见。我给出的代码执行时间也不是很好。

参考代码 2：（与参考代码 1 不同之处只在于剪枝少判断，而且也是先判断截取的 ip 段是否合法，然后用截取函数截取字符串，执行结果上会快一些）

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
```

```java
import java.util.List;
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        int len = s.length();
        List<String> res = new ArrayList<>();
        if (len > 12 || len < 4) return res;
        Deque<String> path = new ArrayDeque<>(4);
        dfs(s, len, 0, 4, path, res);
        return res;
    }
    // 需要一个变量记录剩余多少段还没被分割
    private void dfs(String s, int len, int begin, int residue, Deque<String> p
ath, List<String> res) {
        if (begin == len) {
            if (residue == 0) res.add(String.join(".", path));
            return;
        }
        for (int i = begin; i < begin + 3; i++) {
            if (i >= len) break;
            if (residue * 3 < len - i) continue;
            if (judgeIpSegment(s, begin, i)) {
                String currentIpSegment = s.substring(begin, i + 1);
                path.addLast(currentIpSegment);
                dfs(s, len, i + 1, residue - 1, path, res);
                path.removeLast();
            }
        }
    }
    private boolean judgeIpSegment(String s, int left, int right) {
        int len = right - left + 1;
        if (len > 1 && s.charAt(left) == '0') return false;
        int res = 0;
        while (left <= right) {
            res = res * 10 + s.charAt(left) - '0';
            left++;
        }
        return res >= 0 && res <= 255;
    }
}
```

22、判断二分图（785）

785. 判断二分图

难度 中等 135 收藏 分享 切换为英文 关注 反馈

给定一个无向图 graph，当这个图为二分图时返回 true。

如果我们能将一个图的节点集合分割成两个独立的子集A和B，并使图中的每一条边的两个节点一个来自A集合，一个来自B集合，我们就将这个图称为二分图。

graph 将会以邻接表方式给出，graph[i] 表示图中与节点 i 相连的所有节点。每个节点都是一个在 0 到 graph.length-1 之间的整数。这图中没有自环和平行边：graph[i] 中不存在 i，并且 graph[i] 中没有重复的值。

示例 1:
输入: [[1,3], [0,2], [1,3], [0,2]]
输出: true
解释:
无向图如下:
0----1
| | |
| | |
3----2
我们可以将节点分成两组: {0, 2} 和 {1, 3}。

示例 2:
输入: [[1,2,3], [0,2], [0,1,3], [0,2]]
输出: false
解释:
无向图如下:
0----1
| \ |
| \ |
3----2
我们不能将节点分割成两个独立的子集。

注意:

- graph 的长度范围为 [1, 100]。
- graph[i] 中的元素的范围为 [0, graph.length - 1]。
- graph[i] 不会包含 i 或者有重复的值。
- 图是无向的: 如果 j 在 graph[i] 里边, 那么 i 也会在 graph[j] 里边。

https://leetcode-cn.com/problems/is-graph-bipartite/solution/bfs-dfs-bing-cha-ji-san-chong-fang-fa-pan-duan-er-/

方法 1：DFS

```
class Solution {
    public boolean isBipartite(int[][] graph) {
        // 定义 visited 数组，初始值为 0 表示未被访问，赋值为 1 或者 -1 表示两种不同的颜色。
        int[] visited = new int[graph.length];
        // 因为图中可能含有多个连通域，所以我们需要判断是否存在顶点未被访问，若存在则从它开始再进行一轮 dfs 染色。
        for (int i = 0; i < graph.length; i++) {
            if (visited[i] == 0 && !dfs(graph, i, 1, visited)) {
                return false;
            }
        }
        return true;
    }

    private boolean dfs(int[][] graph, int v, int color, int[] visited)
{
```

```
        // 如果要对某顶点染色时，发现它已经被染色了，则判断它的颜色是否与本次要染的颜色相同，如果矛盾，说明此无向图无法被正确染色，返回 false。
        if (visited[v] != 0) {
            return visited[v] == color;
        }

        // 对当前顶点进行染色，并将当前顶点的所有邻接点染成相反的颜色。
        visited[v] = color;
        for (int w: graph[v]) {
            if (!dfs(graph, w, -color, visited)) {
                return false;
            }
        }
        return true;
    }
}
```

方法 2：BFS

```
class Solution {
    public boolean isBipartite(int[][] graph) {
        // 定义 visited 数组，初始值为 0 表示未被访问，赋值为 1 或者 -1 表示两种不同的颜色。
        int[] visited = new int[graph.length];
        Queue<Integer> queue = new LinkedList<>();
        // 因为图中可能含有多个连通域，所以我们需要判断是否存在顶点未被访问，若存在则从它开始再进行一轮 bfs 染色。
        for (int i = 0; i < graph.length; i++) {
            if (visited[i] != 0) {
                continue;
            }
            // 每出队一个顶点，将其所有邻接点染成相反的颜色并入队。
            queue.offer(i);
            visited[i] = 1;
            while (!queue.isEmpty()) {
                int v = queue.poll();
                for (int w: graph[v]) {
                    // 如果当前顶点的某个邻接点已经被染过色了，且颜色和当前顶点相同，说明此无向图无法被正确染色，返回 false。
                    if (visited[w] == visited[v]) {
                        return false;
                    }
                    if (visited[w] == 0) {
                        visited[w] = -visited[v];
                        queue.offer(w);
                    }
```

```
                }
            }
        }

        return true;
    }
}
```

23、可能的二分法（886）【并查集】

886. 可能的二分法

难度 中等 46 收藏 分享 切换为英文 关注 反馈

给定一组 N 人（编号为 1, 2, ..., N），我们想把每个人分进任意大小的两组。

每个人都可能不喜欢其他人，那么他们不应该属于同一组。

形式上，如果 dislikes[i] = [a, b]，表示不允许将编号为 a 和 b 的人归入同一组。

当可以用这种方法将每个人分进两组时，返回 true；否则返回 false。

示例 1：

```
输入：N = 4, dislikes = [[1,2],[1,3],[2,4]]
输出：true
解释：group1 [1,4], group2 [2,3]
```

示例 2：

```
输入：N = 3, dislikes = [[1,2],[1,3],[2,3]]
输出：false
```

示例 3：

```
输入：N = 5, dislikes = [[1,2],[2,3],[3,4],[4,5],[1,5]]
输出：false
```

提示：

- 1 <= N <= 2000
- 0 <= dislikes.length <= 10000
- dislikes[i].length == 2
- 1 <= dislikes[i][j] <= N
- dislikes[i][0] < dislikes[i][1]
- 对于 dislikes[i] == dislikes[j] 不存在 i != j

https://leetcode-cn.com/problems/possible-bipartition/solution/ke-neng-de-er-fen-fa-chong-lei-bing-cha-ji-by-gyb-/

思路：运用种类并查集的思想，开两倍的数组，前半部分存放可以分在一组的人，后半部分存放自己不喜欢的人。

```
class Solution {
    int N;  //共多少人
    int[] father;   //并查集
    public boolean possibleBipartition(int _N, int[][] dl) {
        N = _N+1;
        father = new int[N*2+5];    //开两倍的数组
        for(int i = 1; i < N*2+5; i++) //初始化并查集
            father[i] = i;
        for(int i = 0; i < dl.length; i++){
            int x = find(dl[i][0]); //查找自己的根节点
            int y = find(dl[i][1]);
            int a = find(dl[i][0] + N); //查找自己不喜欢的人的根节点
            int b = find(dl[i][1] + N);
            if(x == y) return false; //发现这俩人已经在一组
            else{
                father[a] = y;  //将不喜欢的人合并
                father[b] = x;
            }
        }
        return true;
    }
    private int find(int x){ //寻找根节点
        if(x != father[x])
            father[x] = find(father[x]); //路径压缩
        return father[x];
    }
}
```

24、数独（lintcode802）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

802. 数独

编写一个程序，通过填充空单元来解决数独难题。
空单元由数字`0`表示。
你可以认为只有一个唯一的解决方案。

样例
样例 1:

给定的数独谜题:

```
[
[0,0,9,7,4,8,0,0,0],
[7,0,0,0,0,0,0,0,0],
[0,2,0,1,0,9,0,0,0],
[0,0,7,0,0,0,2,4,0],
[0,6,4,0,1,0,5,9,0],
[0,9,8,0,0,0,3,0,0],
[0,0,0,8,0,3,0,2,0],
[0,0,0,0,0,0,0,0,6],
[0,0,0,2,7,5,9,0,0]
]
```

返回结果:

```
[
[5,1,9,7,4,8,6,3,2],
[7,8,3,6,5,2,4,1,9],
[4,2,6,1,3,9,8,7,5],
[3,5,7,9,8,6,2,4,1],
[2,6,4,3,1,7,5,9,8],
[1,9,8,5,2,4,3,6,7],
[9,7,5,8,6,3,1,2,4],
[8,3,2,4,9,1,7,5,6],
[6,4,1,2,7,5,9,8,3]
]
```

适用
号：
级攻略

更多硬核内容就在公众号「yes 的练级攻略」

- 对于这道题目来说，我们要明确几点
  - 目标(base case)：填满每个格子。
  - 选择列表(choice list)：每个空的格子可以填入 `1-9` 这九个数字。
  - 约束条件(constrain)：`1-9` 在每一行、每一列和每个3x3的大格中都只能出现一次。
  - 选择路径(path)：由于我们是inplace操作，`board` 就存储了已经做过的选择。
- 我们在这里设置 `backtrack` 函数的返回类型为 `bool`，这样一来，当程序找到可行解后就能终止递归。

算法流程

- 从最左上角的方格开始，即 `i = 0, j = 0`，开始一行一行填充。
- corner case：
  - 如果 `j = 9`，说明越界，从下一行的第0列开始填。
  - 如果 `i = 9`，说明找到可行解，返回 `True`。
  - 如果`boardi != 0`，这个格子不需要我们填，于是去填下一个格子。
- 从 `1` 到 `9` 迭代循环数组，尝试放置数字 `val` 进入 `(i, j)` 的格子。
  - 如果符合约束条件：
  - 将 `val` 放置在 `(i, j)` 的格子中。
  - 进行dfs，对后面的格子进行相同的操作。
  - 判断是否能找出数独的解。如果可以，返回 `True`。
  - 进行回溯，即将 `(i, j)` 恢复为0。
- 如果遍历完 `1-9` 都没有找到数独的解，返回 `False`。

复杂度分析

- 时间复杂度：时间复杂度是常数。由于数独的大小是固定的，因此没有 N 变量来衡量。上限是$(9!)^9$。对第一行来说，第一个格子不会超过9种情况，第二个格子不会超过8种...所以第一行不会超过$9!$个情况。那么所有行不会超过$(9!)^2$
- 空间复杂度：数独大小固定，空间用来存储数独，行，列和子方块的结构，每个有 81 个元素。

禁止商用
公众号：
yes的练级攻略

```
public class Solution {
    /**
     * @param board: the sudoku puzzle
     * @return: nothing
     */
    public void solveSudoku(int[][] board) {
        // write your code here
        backtrack(board, 0, 0);
    }
    public boolean backtrack(int[][] board, int row, int col) {
        int m = 9, n = 9;
        // 到达第 n 列，越界，换到下一行第 0 列重新开始
        if (col == n) {
            return backtrack(board, row + 1, 0);
        }
        // 到达第 m 行，说明找到可行解，触发 base case
        if (row == m) {
            return true;
        }
        // 如果有预设数字，不用我们穷举
        if (board[row][col] != 0){
            return backtrack(board, row, col + 1);
        }

        for (int num = 1; num <= 9; num++) {
            //遇到已经存在重复的数字就跳过
            if (! isValid(board, row, col, num)) {
```

```
                continue;
            }
            //添加选择
            board[row][col] = num;
            // 如果找到一个可行解，立即结束
            if (backtrack(board, row, col + 1)) {
                return true;
            }
            // 撤回选择
            board[row][col] = 0;
        }
        return false;
    }

    public boolean isValid(int[][] board, int row, int col, int val) {
        for (int i = 0; i < 9; i++) {
            // 判断行是否存在重复
            if (board[row][i] == val) {
                return false;
            }
            // 判断列是否存在重复
            if (board[i][col] == val) {
                return false;
            }
            // 判断 3 x 3 方框是否存在重复
            if (board[(row / 3) * 3 + i / 3][(col / 3) * 3 + i % 3] == val)
                return false;
            }
        return true;
    }
}
```

25、解数独（37）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

37. 解数独

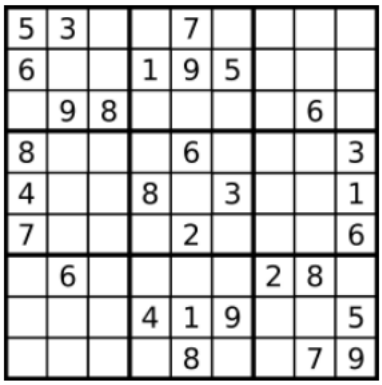
难度 困难 554 收藏 分享 切换为英文 关注 反

编写一个程序，通过已填充的空格来解决数独问题。

一个数独的解法需**遵循如下规则**：

1. 数字 `1-9` 在每一行只能出现一次。
2. 数字 `1-9` 在每一列只能出现一次。
3. 数字 `1-9` 在每一个以粗实线分隔的 `3x3` 宫内只能出现一次。

空白格用 `'.'` 表示。



一个数独。

答案被标成红色。

Note:

- 给定的数独序列只包含数字 `1-9` 和字符 `'.'` 。
- 你可以假设给定的数独只有唯一解。
- 给定数独永远是 `9x9` 形式的。

https://leetcode-cn.com/problems/sudoku-solver/solution/jie-shu-du-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

前言

我们可以考虑按照「行优先」的顺序依次枚举每一个空白格中填的数字，通过递归 + 回溯的方法枚举所有可能的填法。当递归到最后一个空白格后，如果仍然没有冲突，说明我们找到了答案；在递归的过程中，如果当前的空白格不能填下任何一个数字，那么就进行回溯。

由于每个数字在同一行、同一列、同一个九宫格中只会出现一次，因此我们可以使用 $line[i]$，$column[j]$，$block[x][y]$ 分别表示第 i 行，第 j 列，第 (x, y) 个九宫格中填写数字的情况。在下面给出的三种方法中，我们将会介绍两种不同的表示填写数字情况的方法。

> 九宫格的范围为 $0 \leq x \leq 2$ 以及 $0 \leq y \leq 2$。
> 具体地，第 i 行第 j 列的格子位于第 $(\lfloor i/3 \rfloor, \lfloor j/3 \rfloor)$ 个九宫格中，其中 $\lfloor u \rfloor$ 表示对 u 向下取整。

由于这些方法均以递归 + 回溯为基础，算法运行的时间（以及时间复杂度）很大程度取决于给定的输入数据，而我们很难找到一个非常精确的渐进紧界。因此这里只给出一个较为宽松的渐进复杂度上界 $O(9^{9\times 9})$，即最多有 9×9 个空白格，每个格子可以填 $[1, 9]$ 中的任意整数。

方法一：递归

思路

最容易想到的方法是用一个数组记录每个数字是否出现。由于我们可以填写的数字范围为 $[1, 9]$，而数组的下标从 0 开始，因此在存储时，我们使用一个长度为 9 的布尔类型的数组，其中 i 个元素的值为 True，当且仅当数字 $i + 1$ 出现过。例如我们用 $line[2][3] = \text{True}$ 表示数字 4 在第 2 行已经出现过，那么当我们在遍历到第 2 行的空白格时，就不能填入数字 4。

算法

我们首先对整个数独数组进行遍历，当我们遍历到第 i 行第 j 列的位置：

- 如果该位置是一个空白格，那么我们将其加入一个用来存储空白格位置的列表中，方便后续的递归操作；
- 如果该位置是一个数字 x，那么我们需要将 $line[i][x-1]$，$column[j][x-1]$ 以及 $block[\lfloor i/3 \rfloor][\lfloor j/3 \rfloor][x-1]$ 均置为 True。

当我们结束了遍历过程之后，就可以开始递归枚举。当递归到第 i 行第 j 列的位置时，我们枚举填入的数字 x。根据题目的要求，数字 x 不能和当前行、列、九宫格中已经填入的数字相同，因此 $line[i][x-1]$，$column[j][x-1]$ 以及 $block[\lfloor i/3 \rfloor][\lfloor j/3 \rfloor][x-1]$ 必须均为 False。

当我们填入了数字 x 之后，我们要将上述的三个值都置为 True，并且继续对下一个空白格位置进行递归。在回溯到当前递归层时，我们还要将上述的三个值重新置为 False。

方法 1：dfs

```
class Solution {
    private boolean[][] line = new boolean[9][9];
    private boolean[][] column = new boolean[9][9];
    private boolean[][][] block = new boolean[3][3][9];
    private boolean valid = false;
    private List<int[]> spaces = new ArrayList<int[]>();

    public void solveSudoku(char[][] board) {
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    spaces.add(new int[]{i, j});
                } else {
                    int digit = board[i][j] - '0' - 1;
```

```
                    line[i][digit] = column[j][digit] = block[i / 3][j / 3][digit] = true;
                }
            }
        }

        dfs(board, 0);
    }

    public void dfs(char[][] board, int pos) {
        if (pos == spaces.size()) {
            valid = true;
            return;
        }

        int[] space = spaces.get(pos);
        int i = space[0], j = space[1];
        for (int digit = 0; digit < 9 && !valid; ++digit) {
            if (!line[i][digit] && !column[j][digit] && !block[i / 3][j / 3][digit]) {
                line[i][digit] = column[j][digit] = block[i / 3][j / 3][digit] = true;
                board[i][j] = (char) (digit + '0' + 1);
                dfs(board, pos + 1);
                line[i][digit] = column[j][digit] = block[i / 3][j / 3][digit] = false;
            }
        }
    }
}
```

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

思路与算法

在方法一中，我们使用了长度为 9 的数组表示每个数字是否出现过。我们同样也可以借助位运算，仅使用一个整数表示每个数字是否出现过。

具体地，数 b 的二进制表示的第 i 位（从低到高，最低位为第 0 位）为 1，当且仅当数字 $i+1$ 已经出现过。例如当 b 的二进制表示为 $(011000100)_2$ 时，就表示数字 3，7，8 已经出现过。

位运算有一些基础的使用技巧。下面列举了所有在代码中使用到的技巧：

- 对于第 i 行第 j 列的位置，$line[i] \mid column[j] \mid block[\lfloor i/3 \rfloor][\lfloor j/3 \rfloor]$ 中第 k 位为 1，表示该位置不能填入数字 $k+1$（因为已经出现过），其中 $\mid$ 表示按位或运算。如果我们对这个值进行 $\sim$ 按位取反运算，那么第 k 位为 1 就表示该位置可以填入数字 $k+1$，我们就可以通过寻找 1 来进行枚举。由于在进行按位取反运算后，这个数的高位也全部变成了 1，而这是我们不应当枚举到的，因此我们需要将这个数和 $(111111111)_2 = (1\text{FF})_{16}$ 进行按位与运算 $\&$，将所有无关的位置为 0；
- 我们可以使用按位异或运算 $\wedge$，将第 i 位从 0 变为 1，或从 1 变为 0。具体地，与数 $1 << i$ 进行按位异或运算即可，其中 $<<$ 表示左移运算；
- 我们可以用 $b \ \& \ (-b)$ 得到 b 二进制表示中最低位的 1，这是因为 $(-b)$ 在计算机中以补码的形式存储，它等于 $\sim b+1$。b 如果和 $\sim b$ 进行按位与运算，那么会得到 0，但是当 $\sim b$ 增加 1 之后，最低位的连续的 1 都变为 0，而最低位的 0 变为 1，对应到 b 中即为最低位的 1，因此当 b 和 $\sim b+1$ 进行按位与运算时，只有最低位的 1 会被保留；
- 当我们得到这个最低位的 1 时，我们可以通过一些语言自带的函数得到这个最低位的 1 究竟是第几位（即 i 值），具体可以参考下面的代码；
- 我们可以用 b 和最低位的 1 进行按位异或运算，就可以将其从 b 中去除，这样就可以枚举下一个 1。同样地，我们也可以用 b 和 $b-1$ 进行按位与运算达到相同的效果，读者可以自行尝试推导。

实际上，方法二中整体的递归 + 回溯的框架与方法一是一致的。不同的仅仅是我们将一个数组「压缩」成了一个数而已。

方法 2：dfs+位运算

```
class Solution {
    private int[] line = new int[9];
    private int[] column = new int[9];
    private int[][] block = new int[3][3];
    private boolean valid = false;
    private List<int[]> spaces = new ArrayList<int[]>();

    public void solveSudoku(char[][] board) {
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 9; ++j) {
                if (board[i][j] == '.') {
                    spaces.add(new int[]{i, j});
                } else {
                    int digit = board[i][j] - '0' - 1;
                    flip(i, j, digit);
                }
            }
        }

        dfs(board, 0);
    }
```

公众号：yes的练级攻略

```
    public void dfs(char[][] board, int pos) {
        if (pos == spaces.size()) {
            valid = true;
            return;
        }

        int[] space = spaces.get(pos);
        int i = space[0], j = space[1];
        int mask = ~(line[i] | column[j] | block[i / 3][j / 3]) & 0x1ff;
        for (; mask != 0 && !valid; mask &= (mask - 1)) {
            int digitMask = mask & (-mask);
            int digit = Integer.bitCount(digitMask - 1);
            flip(i, j, digit);
            board[i][j] = (char) (digit + '0' + 1);
            dfs(board, pos + 1);
            flip(i, j, digit);
        }
    }

    public void flip(int i, int j, int digit) {
        line[i] ^= (1 << digit);
        column[j] ^= (1 << digit);
        block[i / 3][j / 3] ^= (1 << digit);
    }
}
```

26、矩阵中的最长递增路径（329）【dfs+memo】【拓扑结构 BFS】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

329. 矩阵中的最长递增路径

难度 困难 260 收藏 分享 切换为英文 关注 反馈

给定一个整数矩阵，找出最长递增路径的长度。

对于每个单元格，你可以往上，下，左，右四个方向移动。 你不能在对角线方向上移动或移动到边界外（即不允许环绕）。

示例 1:

```
输入: nums =
[
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
输出: 4
解释: 最长递增路径为 [1, 2, 6, 9]。
```

示例 2:

```
输入: nums =
[
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
输出: 4
解释: 最长递增路径是 [3, 4, 5, 6]。注意不允许在对角线方向上移动。
```

https://leetcode-cn.com/problems/longest-increasing-path-in-a-matrix/solution/ju-zhen-zhong-de-zui-chang-di-zeng-lu-jing-by-le-2/

方法一：记忆化深度优先搜索

将矩阵看成一个有向图，每个单元格对应图中的一个节点，如果相邻的两个单元格的值不相等，则在相邻的两个单元格之间存在一条从较小值指向较大值的有向边。问题转化成在有向图中寻找最长路径。

深度优先搜索是非常直观的方法。从一个单元格开始进行深度优先搜索，即可找到从该单元格开始的最长递增路径。对每个单元格分别进行深度优先搜索之后，即可得到矩阵中的最长递增路径的长度。

但是如果使用朴素深度优先搜索，时间复杂度是指数级，会超出时间限制，因此必须加以优化。

朴素深度优先搜索的时间复杂度过高的原因是进行了大量的重复计算，同一个单元格会被访问多次，每次访问都要重新计算。由于同一个单元格对应的最长递增路径的长度是固定不变的，因此可以使用记忆化的方法进行优化。用矩阵 memo 作为缓存矩阵，已经计算过的单元格的结果存储到缓存矩阵中。

使用记忆化深度优先搜索，当访问到一个单元格 (i,j) 时，如果 $\text{memo}[i][j] \neq 0$，说明该单元格的结果已经计算过，则直接从缓存中读取结果，如果 $\text{memo}[i][j] = 0$，说明该单元格的结果尚未被计算过，则进行搜索，并将计算得到的结果存入缓存中。

遍历完矩阵中的所有单元格之后，即可得到矩阵中的最长递增路径的长度。

复杂度分析

- 时间复杂度：$O(mn)$，其中 m 和 n 分别是矩阵的行数和列数。深度优先搜索的时间复杂度是 $O(V+E)$，其中 V 是节点数，E 是边数。在矩阵中，$O(V)=O(mn)$，$O(E) \approx O(4mn)=O(mn)$。
- 空间复杂度：$O(mn)$，其中 m 和 n 分别是矩阵的行数和列数。空间复杂度主要取决于缓存和递归调用深度，缓存的空间复杂度是 $O(mn)$，递归调用深度不会超过 mn。

```
class Solution {
```

```
    public int[][] dire = {{-1,0}, {1,0}, {0, -1}, {0, 1}};
    public int row, col;
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length ==
 0) return 0;
        row = matrix.length;
        col = matrix[0].length;
        int[][] memo = new int[row][col];
        int res = 0;
        for (int i = 0; i < row; i++){
            for (int j = 0; j < col; j++) {
                res = Math.max(res, dfs(matrix, i, j, memo));
            }
        }
        return res;
    }
    public int dfs(int[][] matrix, int r, int c, int[][] memo){
        if (memo[r][c] != 0) return memo[r][c];
        ++memo[r][c];
        for (int[] dir : dire){
            int newR = r + dir[0], newC = c + dir[1];
//以下的 matrix[newR][newC] > matrix[r][c]改成
matrix[newR][newC] < matrix[r][c]也可以，前者代码 memo 以 r，c 开头，后者代表
memo 以 r，c 结尾
            if (newR >= 0 && newR < row && newC >= 0 && newC < col && m
atrix[newR][newC] > matrix[r][c]){
                memo[r][c] = Math.max(memo[r][c], dfs(matrix, newR, new
C, memo) + 1);
            }
        }
        return memo[r][c];
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

方法二：拓扑排序

从方法一可以看到，每个单元格对应的最长递增路径的结果只和相邻单元格的结果有关，那么是否可以使用动态规划求解？

根据方法一的分析，动态规划的状态定义和状态转移方程都很容易得到。方法一中使用的缓存矩阵 memo 即为状态值，状态转移方程如下：

$$\text{memo}[i][j] = \max\{\text{memo}[x][y]\} + 1$$
其中 (x, y) 与 (i, j) 在矩阵中相邻，并且 $\text{matrix}[x][y] > \text{matrix}[i][j]$

动态规划除了状态定义和状态转移方程，还需要考虑边界情况。这里的边界情况是什么呢？

如果一个单元格的值比它的所有相邻单元格的值都要大，那么这个单元格对应的最长递增路径是 1，这就是边界条件。这个边界条件并不直观，而是需要根据矩阵中的每个单元格的值找到作为边界条件的单元格。

仍然使用方法一的思想，将矩阵看成一个有向图，计算每个单元格对应的出度，即有多少条边从该单元格出发。对于作为边界条件的单元格，该单元格的值比所有的相邻单元格的值都要大，因此作为边界条件的单元格的出度都是 0。

基于出度的概念，可以使用拓扑排序求解。从所有出度为 0 的单元格开始广度优先搜索，每一轮搜索都会遍历当前层的所有单元格，更新其余单元格的出度，并将出度变为 0 的单元格加入下一层搜索。当搜索结束时，搜索的总层数即为矩阵中的最长递增路径的长度。

复杂度分析

- 时间复杂度：$O(mn)$，其中 m 和 n 分别是矩阵的行数和列数。拓扑排序的时间复杂度是 $O(V + E)$，其中 V 是节点数，E 是边数。在矩阵中，$O(V) = O(mn)$，$O(E) \approx O(4mn) = O(mn)$。
- 空间复杂度：$O(mn)$，其中 m 和 n 分别是矩阵的行数和列数。空间复杂度主要取决于队列，队列中的元素个数不会超过 mn。

思考题

为了让大家更好地理解这道题，小编出了四道思考题，欢迎感兴趣的同学在评论区互动哦。

- 「方法一」中使用了记忆化存储和深度优先搜索，这里的深度优先搜索可以替换成广度优先搜索吗？
- 「方法二」中基于拓扑排序对排序后的有向无环图做了层次遍历，如果没有拓扑排序直接进行广度优先搜索会发生什么？
- 「方法二」中如果不使用拓扑排序，而是直接按照矩阵中元素的值**从大到小**进行排序，并依此顺序进行状态转移，那么可以得到正确的答案吗？如果是**从小到大**进行排序呢？
- 「变式」给定一个整数矩阵，找出符合以下条件的路径的数量：这个路径是严格递增的，且它的长度至少是 3。矩阵的边长最大为 10^3，答案对 $10^9 + 7$ 取模。其他条件和题目相同。**思考：是否可以借鉴这道题的方法？**

问题 1：

Peony
7 小时前
广度优先搜索应该是不行吧，备忘录存的是节点能表示的最长路径，bfs需要得到最后的层数，如果同时出现访问过得和没访问过的节点，bfs还会按照没访问过的遍历出最大深度，这样备忘录就没用了，个人理解
赞 踩 收起回复 回复 分享 举报
最早 | 最新
码行
（编辑过）4 小时前
@Peony 用备忘录记录广度优先搜索的确实不行。因为没法确定该点处在正确的链上。用python，无备忘录直接广搜（单点多次入队），3944 ms，用拓扑排序566ms。当然如果能确定找到正确的起点，那么广搜就能达到单点单次入队的效果。一种就是用拓扑排序，能确定唯一的顺序，另一种直接根据值排序后遍历（大小排序都行），能保证得到正确的最长路径。
1 踩 回复 分享 举报
扭秧歌的一只泱
3 小时前
@Peony +1，我一开始就是广搜的思路，但是发现不引入入度的概念的话会出问题（于是又变回dfs了）。另外，小破站秋招开始了哈哈哈
赞 踩 回复
收起回复
Dexter
8 小时前
思考题：（我先抛个砖，等大佬来引个玉） 如果使用广度优先搜索没办法根据层数确认最长的路径，因为如果没有根据出度为0的点开始遍历，每个点的深度还是跟该点的下一层的点有关，就又变成深度优先搜索了；将矩阵元素从大到小排序，然后更新从最大元素位置开始转移， 然后每更新一个点就检查其周围四个点是否有更新并且比该点值大，如果有则选择周围四个点最大的值+1更新该点值，在更新过程不断维护最长的路径，从小到大排序反过来和从大到小一样； 变式那个应该可以吧，大佬们说一下
赞 踩 收起回复 回复
最早 | 最新
lauiappin
8 小时前
@Dexter 同意
赞 踩 回复
DPSDL
7 小时前
@Dexter 广搜把每一层累加直至最大不就是最大路径么，当然要优化，否则超时
赞 踩 回复

问题 2：

更多硬核内容就在公众号「yes 的练级攻略」

Cameroceras (编辑过) 16 小时前

> 「方法二」中基于拓扑排序对排序后的有向无环图做了层次遍历，如果没有拓扑排序直接进行广度优先搜索会发生什么?

当然是以更长的时间跑完啦[捂脸]

```
typedef pair<int,int>PII;

class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        const int dx[] = {1,-1,0,0};
        const int dy[] = {0,0,1,-1};
        const int n = matrix.size();
        if(n == 0) return 0;
        const int m = matrix[0].size();
        vector<vector<bool>>vis(n,vector<bool>(m,false));
        vector<vector<int>>ans(n,vector<int>(m,1));
        queue<PII>que;
        for(int i = 0;i < n;++i) for(int j = 0;j < m;++j){
            bool flag = true;
            for(int k = 0;k < 4;++k){
                if(i+dx[k] >= 0 && i+dx[k] < n&& j + dy[k] >=0 && j+dy[k] < m)
                    flag &= matrix[i][j] <= matrix[i+dx[k]][j+dy[k]];
            }
            if(flag) {que.push({i,j});vis[i][j] = true;}
        }
        int ret = 0;
        while(!que.empty()){
            PII u = que.front();
            que.pop();
            vis[u.first][u.second] = false;
            ret = max(ret,ans[u.first][u.second]);
            for(int i = 0,nx,ny;i < 4;++i){
                nx = u.first + dx[i];
                ny = u.second + dy[i];
                if(nx >= 0 && nx < n && ny >= 0 && ny<m){
                    if(matrix[u.first][u.second] <  matrix[nx][ny] && ans[u.first][u.second]+1 > ans[nx][ny]){
```

问题 3：

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

没有出口的迷宫 7 小时前

先从大到小排序是可以的，因为任一特定位置为起点的最长递增路径只依赖于它的递增方向的最长递增路径。此法甚至不需要一般记忆化搜索中的递归与回溯。附代码

```python
def longest_ascending_path_in_matrix(_matrix):
    num_rows = len(_matrix)
    if num_rows == 0:
        return 0
    num_columns = len(_matrix[0])
    if num_columns == 0:
        return 0
    # 将矩阵元素从大到小排序
    inverted_index = [(i, j, _matrix[i][j]) for i in range(num_rows) for j in range(num_columns)]
    inverted_index.sort(key=lambda x: x[2], reverse=True)
    global_max = 0
    _cache = {}
    # 然后就是搜索了，注意一定要按上面排好的顺序
    for current_row, current_column, _value in inverted_index:
        local_max = 0
        # 四个方向
        if current_row > 0 and _matrix[current_row - 1][current_column] > _value:
            local_max = max(local_max, _cache[(current_row - 1, current_column)])
        if current_row < num_rows - 1 and _matrix[current_row + 1][current_column] > _value:
            local_max = max(local_max, _cache[(current_row + 1, current_column)])
        if current_column > 0 and _matrix[current_row][current_column - 1] > _value:
            local_max = max(local_max, _cache[(current_row, current_column - 1)])
        if current_column < num_columns - 1 and _matrix[current_row][current_column + 1] > _value:
            local_max = max(local_max, _cache[(current_row, current_column + 1)])
        _cache[(current_row, current_column)] = local_max + 1
        global_max = max(global_max, local_max + 1)

    return global_max
```

1 踩 查看 1 条回复 回复 分享 举报

最早 | 最新

zerotrac 6 小时前

@没有出口的迷宫 对的，但是排序就多了一个 log 的时间复杂度呀

赞 踩 回复

tzyf2000 7 小时前

回答一下第三个问题，答案应该是可以的。理由比较简单。当你决定从大到小dp的时候，小数的值完全依赖于比它大的值，那么当我先将大数的值（最大值为1）确定了，那么下一个比它小的值只受所有确定的在它四周的值影响，那么这个值也就确定了。反之，反之。

2 踩 回复 分享 举报

```java
class Solution {
    public int[][] dire = {{-1,0}, {1,0}, {0, -1}, {0, 1}};
    public int row, col;
    public int longestIncreasingPath(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
        row = matrix.length;
        col = matrix[0].length;
        int res = 0;
        int[][] dp = new int[row][col];
```

```
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++){
                for (int[] dir : dire){
                    int newR = i + dir[0], newC = j + dir[1];
                    if (newR >= 0 && newR < row && newC >= 0 && newC <
col && matrix[newR][newC] > matrix[i][j]){
                        ++dp[i][j];
                    }
                }
            }
        }

        Queue<int[]> que = new LinkedList<>();
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++){
                if (dp[i][j] == 0) {
                    que.offer(new int[] {i, j});
                }
            }
        }

        while (!que.isEmpty()){
            ++res;
            int size = que.size();
            for (int i = 0; i < size; ++i){
                int[] cell = que.poll();
                int r = cell[0], c = cell[1];
                for (int[] dir : dire){
                    int newR = r + dir[0], newC = c + dir[1];
                    if (newR >= 0 && newR < row && newC >= 0 && newC <
col && matrix[newR][newC] < matrix[r][c]){
                        --dp[newR][newC];
                        if (dp[newR][newC] == 0) que.offer(new int[] {n
ewR, newC});
                    }
                }
            }
        }
        return res;
    }
}
```

27、单词搜索（79）【DFS+回溯】【矩阵中的路径（剑指 12）】

更多硬核内容就在公众号「yes 的练级攻略」

79. 单词搜索

难度 中等 491 收藏 分享 切换为英文 关注 反馈

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例:

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

给定 word = "ABCCED", 返回 true
给定 word = "SEE", 返回 true
给定 word = "ABCB", 返回 false
```

提示:

- board 和 word 中只包含大写和小写英文字母。
- 1 <= board.length <= 200
- 1 <= board[i].length <= 200
- 1 <= word.length <= 10^3

禁止商用

https://leetcode-cn.com/problems/ju-zhen-zhong-de-lu-jing-lcof/solution/mian-shi-ti-12-ju-zhen-zhong-de-lu-jing-shen-du-yo/

解题思路:

> 本问题是典型的矩阵搜索问题，可使用 **深度优先搜索（DFS）+ 剪枝** 解决。

算法原理:

- **深度优先搜索：** 可以理解为暴力法遍历矩阵中所有字符串可能性。DFS 通过递归，先朝一个方向搜到底，再回溯至上个节点，沿另一个方向搜索，以此类推。
- **剪枝：** 在搜索中，遇到 这条路不可能和目标字符串匹配成功 的情况（*例如：此矩阵元素和目标字符不同、此元素已被访问*），则应立即返回，称之为 可行性剪枝 。

算法剖析:

- **递归参数：** 当前元素在矩阵 board 中的行列索引 i 和 j ，当前目标字符在 word 中的索引 k 。
- **终止条件:**
  1. **返回** $false$ ： ① 行或列索引越界 **或** ② 当前矩阵元素与目标字符不同 **或** ③ 当前矩阵元素已访问过 *（③ 可合并至 ②）* 。
  2. **返回** $true$ ： 字符串 word 已全部匹配，即 k = len(word) - 1 。
- **递推工作:**
  1. **标记当前矩阵元素：** 将 board[i][j] 值暂存于变量 tmp ，并修改为字符 '/' ，代表此元素已访问过，防止之后搜索时重复访问。
  2. **搜索下一单元格：** 朝当前元素的 **上、下、左、右** 四个方向开启下层递归，使用 或 连接 *（代表只需一条可行路径）* ，并记录结果至 res 。
  3. **还原当前矩阵元素：** 将 tmp 暂存值还原至 board[i][j] 元素。
- **回溯返回值：** 返回 res ，代表是否搜索到目标字符串。

复杂度分析：

> M, N 分别为矩阵行列大小， K 为字符串 `word` 长度。

- **时间复杂度** $O(3^K MN)$ ： 最差情况下，需要遍历矩阵中长度为 K 字符串的所有方案，时间复杂度为 $O(3^K)$；矩阵中共有 MN 个起点，时间复杂度为 $O(MN)$ 。
  - **方案数计算：** 设字符串长度为 K ，搜索中每个字符有上、下、左、右四个方向可以选择，舍弃回头（上个字符）的方向，剩下 3 种选择，因此方案数的复杂度为 $O(3^K)$ 。
- **空间复杂度** $O(K)$ ： 搜索过程中的递归深度不超过 K ，因此系统因函数调用累计使用的栈空间占用 $O(K)$ （因为函数返回后，系统调用的栈空间会释放）。最坏情况下 $K = MN$ ，递归深度为 MN ，此时系统栈使用 $O(MN)$ 的额外空间。

```
class Solution {
    public boolean exist(char[][] board, String word) {
        char[] words = word.toCharArray();
        for(int i = 0; i < board.length; i++) {
            for(int j = 0; j < board[0].length; j++) {
                if(dfs(board, words, i, j, 0)) return true;
            }
        }
        return false;
    }
    boolean dfs(char[][] board, char[] word, int i, int j, int k) {
        if(i >= board.length || i < 0 || j >= board[0].length || j < 0 || board
[i][j] != word[k]) return false;
        if(k == word.length - 1) return true;
        char tmp = board[i][j];
        board[i][j] = '/';
        boolean res = dfs(board, word, i + 1, j, k + 1) || dfs(board, word, i -
 1, j, k + 1) || dfs(board, word, i, j + 1, k + 1) || dfs(board, word, i , j -
1, k + 1);
        board[i][j] = tmp;
        return res;
    }
}
```

28、单词搜索 2（212）【DFS+回溯+剪枝+字典树】

更多硬核内容就在公众号「yes 的练级攻略」

212. 单词搜索 II

难度 困难 201 收藏 分享 切换为英文 关注 反馈

给定一个二维网格 **board** 和一个字典中的单词列表 **words**，找出所有同时在二维网格和字典中出现的单词。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

示例:

```
输入:
words = ["oath","pea","eat","rain"] and board =
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]

输出: ["eat","oath"]
```

说明:
你可以假设所有输入都由小写字母 `a-z` 组成。

提示:

- 你需要优化回溯算法以通过更大数据量的测试。你能否早点停止回溯？
- 如果当前单词不存在于所有单词的前缀中，则可以立即停止回溯。什么样的数据结构可以有效地执行这样的操作？散列表是否可行？为什么？ 前缀树如何？如果你想学习如何实现一个基本的前缀树，请先查看这个问题： 实现Trie（前缀树）。

禁止商用

https://leetcode-cn.com/problems/word-search-ii/solution/qian-zhui-shu-ying-yong-zhi-dan-ci-sou-suo-cape-gi/

```java
class Solution {
    class TrieNode {
        TrieNode[] next = new TrieNode[26];
        String word = null;
        int count = 0;
    }


    List<String> result = new ArrayList<>();
    int m;//行
    int n;//列
    int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
    TrieNode root = new TrieNode();

    public List<String> findWords(char[][] board, String[] words) {
        //1.构造前缀树，遍历words，在里层遍历每一个word
        addWords(words);
        //2.对单元格进行回溯
        m = board.length;
        n = board[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (root.next[board[i][j] - 'a'] != null) {
                    backtracking(board, i, j, root);
                }
            }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }


        return result;
    }

    private void backtracking(char[][] board, int i, int j, TrieNode ro
ot) {
        char c = board[i][j];
        TrieNode curr = root.next[c - 'a'];
        if (curr != null && curr.word != null) {
            result.add(curr.word);
            curr.word = null;
        }
        board[i][j] = '#';
        for (int k = 0; k < 4; k++) {
            int nextI = i + directions[k][0];
            int nextJ = j + directions[k][1];
            if ((nextI < 0 || nextI >= m || nextJ < 0 || nextJ >= n) ||
 board[nextI][nextJ] == '#') continue;
            if (curr != null) backtracking(board, nextI, nextJ, curr);
        }
        board[i][j] = c;
        if (curr != null && curr.count == 0) {
            root.next[c - 'a'] = null;
            root.count--;
        }
    }

    private void addWords(String[] words) {

        for (String word : words) {
            TrieNode curr = root;
            for (char c : word.toCharArray()) {
                if (curr.next[c - 'a'] == null) {
                    curr.count++;
                    curr.next[c - 'a'] = new TrieNode();
                }
                curr = curr.next[c - 'a'];
            }
            curr.word = word;
        }
    }
}
```

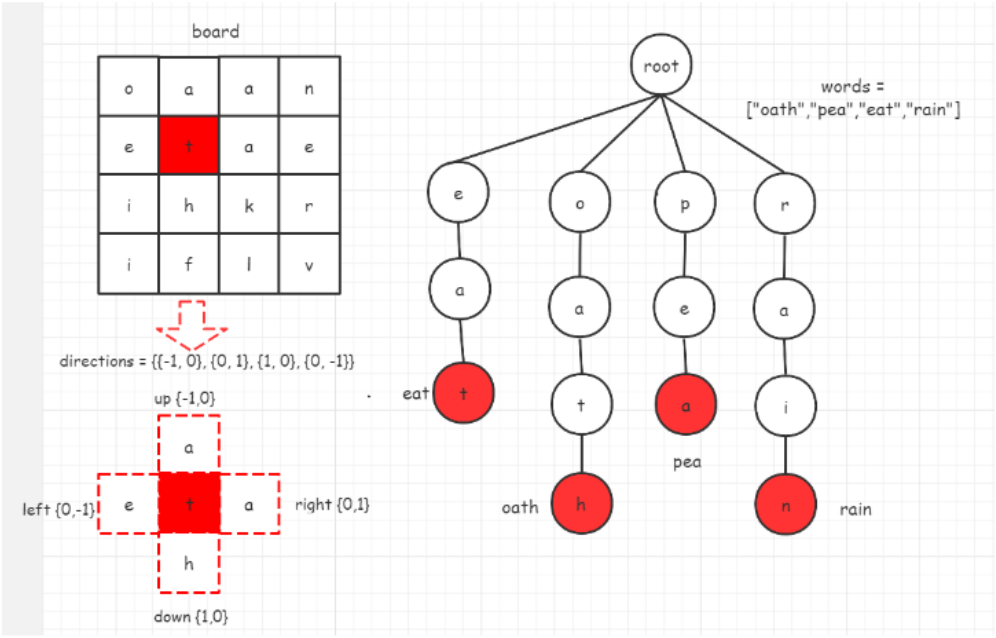
更多硬核内容就在公众号「yes 的练级攻略」

1.构建$words$的字典前缀树

构建前缀树，可以参考208题,区别在于当到达一个单词的末尾时，将单词赋值给$word$

```
TrieNode root = new TrieNode();
for (String word : words) {
    TrieNode curr = root;
    for (char c : word.toCharArray()) {
        if (curr.next[c - 'a'] == null) {
            curr.next[c - 'a'] = new TrieNode();
        }
        curr = curr.next[c - 'a'];
    }
    curr.word = word;
}
```

2.dfs搜索



- 推荐一种写法：

```
int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
 for (int k = 0; k < 4; k++) {
            int nextI = i + directions[k][0];
            int nextJ = j + directions[k][1];
 }
//详细的解释见上图，二维数组，拿行时，取[k][0],拿列时，取[k][1]
```

- 出口条件：(当前节点是一个单词的末尾字符形成的节点)

```
        if (curr != null && curr.word != null) {
            result.add(curr.word);
            curr.word = null;
        }
```

- 标记与边界

```
 board[i][j] = '#';
 board[i][j] = c;
//在整个board区域内，且没有走过
if ((nextI < 0 || nextI >= m || nextJ < 0 || nextJ >= n) || board[nextI][nextJ] == '#') continue;
```

> 当然，也可以使用与board同样大小的访问数组来标记，缺点是浪费空间

```
boolean[][] visited = new boolean[m][n];
```

- 剪枝，准备一个变量count,当叶子节点的时候，也就是count=0时，开始剪枝，叶子节点已无用处，在构造前缀树的过程中，维护好count

29、除法求值（399）

399. 除法求值

难度 中等 184 收藏 分享 切换为英文 关注 反馈

给出方程式 `A / B = k`, 其中 `A` 和 `B` 均为用字符串表示的变量， `k` 是一个浮点型数字。根据已知方程式求解问题，并返回计算结果。如果结果不存在，则返回 `-1.0`。

示例：
给定 `a / b = 2.0, b / c = 3.0`
问题: `a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?`
返回 `[6.0, 0.5, -1.0, 1.0, -1.0 ]`

输入为: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries`(方程式，方程式结果，问题方程式)，其中 `equations.size() == values.size()`，即方程式的长度与方程式结果长度相等（程式与结果一一对应），并且结果值均为正数。以上为方程式的描述。 返回 `vector<double>`类型。

基于上述例子，输入如下：

```
equations(方程式) = [ ["a", "b"], ["b", "c"] ],
values(方程式结果) = [2.0, 3.0],
queries(问题方程式) = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
```

输入总是有效的。你可以假设除法运算中不会出现除数为0的情况，且不存在任何矛盾的结果。

https://leetcode-cn.com/problems/evaluate-division/solution/zhen-zheng-de-xiao-bai-du-neng-kan-dong-de-bing-ch/

首先如果有a/b = 3，我们令parent[a] = b，即让b成为a的父亲，同时让val[a] = 3。
所以此处的定义非常重要：**即val[]数组存放的是节点与他直接父节点的关系**
因此我们在遍历equations时，根据from/to = val 的关系，要初始化一些指向和val数组的值。
但是在from已经存在（即parent里存在这个key），说明from已经在某个集合中（作为根或非根），我们为了方便起见（不需要合并集合），让不在集合的to指向from，val也取倒数。

最关键的一步在于，如果from和to在两个不同的集合内怎么办?
这时候注意看我们的find函数，find函数除了返回根以外，还进行了**路径压缩**以及val值更新，我们利用回溯对val值进行了更新，使得**val值实际上变成了节点与根的关系**，而且每个非根节点还会直接指向根节点。（路径压缩也保证了多次find操作不会使得val值被多次错误更新，因为根节点的val值是1）
也就是说如果一个集合有a/b = 3, b/c = 3，其中c作为根节点。那么在find返回之后，我们不仅拿到了c，还使得集合的形状变为了：

- a -> c,val[a] = 9
- b -> c, val[b] = 3
- c -> c, val[c] = 1

那么回到我们from和to在两个不同集合的问题，我们发现在判断from和to是否位于一个集合中的时候，我们分别对from和to各调用了一次**find**操作，这说明什么?
说明此时from和to已经直接指向他们各自的根节点，并且val[from]和val[to]直接存放他们和各自根节点的比值
现在我们要对两个集合进行合并，假设from的根为root1， to的根为root2，我们让parent[root1] = root2
注意，这时候val[root1]不能再为1了，那么我们要更新为多少呢?
其实我们发现val[root1] 就是 root1 / root2的值
而且我们已有from/root1, to/root2, from/to的值
简单的数学：root1/root2 = to/root2 * from/to ÷ from/root1也就是val[root1] = val[to] * from/to ÷ val[from]

好了此时我们的val[root1]已经是正确的值了，它确实反映了root1和root2的正确关系。至于原本在root2为根的集合的节点，它们是不用发生任何变化的，这点很显然。
那么原本在root1集合中的点呢?
我们惊奇的发现，由于val[root1]已经不再是1了，因此在下一次find()操作中，其他节点会同样经过回溯更新val值，使得val值最终成为该节点与新根节点root2的比值。

非常完美！而且由于我们在query阶段，绝对会对from和to各自调用一次find操作，确保了这之后val[from]和val[to]都是必定正确的，也就是他们都真实反映了他们和各自根节点的比值。

因此如果from和to位于同一集合（假设根为root），也就是我们根据val能得到绝对正确的 from / root 和 to / root，那么 from / to等于多少呢?
这下问题就变成了小学数学了，拿下!

并查集代码 1

```
class Solution {
    static Map<String, String> parents;
    static Map<String, Double> val;


    public String find(String x) {

        if (!x.equals(parents.get(x))) {
            String tmpParent = parents.get(x);
            String root = find(tmpParent);
            double oldVal = val.get(x);
            val.put(x, oldVal * val.get(tmpParent));
            parents.put(x, root);
        }
        return parents.get(x);
    }
```

```
    public double[] calcEquation(List<List<String>> equations, double[]
values, List<List<String>> queries) {
        parents = new HashMap<>();
        val = new HashMap<>();
        int i = 0;
        for (List<String> equation : equations) {
            String from = equation.get(0);
            String to = equation.get(1);
            double cur = values[i];
            if (!parents.containsKey(from) && !parents.containsKey(to))
{
                parents.put(from, to);
                val.put(from, cur);
                parents.put(to, to);
                val.put(to, 1.0);
            } else if (!parents.containsKey(from)) {
                parents.put(from, to);
                val.put(from, cur);
            } else if (!parents.containsKey(to)) {
                parents.put(to, from);
                val.put(to, 1 / cur);
            } else {
                String pa = find(from);
                String pb = find(to);
                if (!pa.equals(pb)) {
                    parents.put(pa, pb);
                    val.put(pa, cur * val.get(to) / val.get(from));
                }
            }
            i++;
        }
        i = 0;
        double[] res = new double[queries.size()];
        for (List<String> query : queries) {
            String from = query.get(0);
            String to = query.get(1);
            if (!parents.containsKey(from) || !parents.containsKey(to))
{
                res[i++] = -1;
                continue;
            }
            String pa = find(from);
            String pb = find(to);
```

```
                if (!pa.equals(pb)) res[i] = -1;
                else {
                    res[i] = val.get(from) / val.get(to);
                }
                i++;
            }
            return res;
        }
}
```

https://leetcode-cn.com/problems/evaluate-division/solution/javabing-cha-ji-lu-jing-ya-suo-by-im1gw0/

解法一

其实这题我开始想到就是建图然后BFS，并查集我是真没想到，看来还是不够敏锐，不过有一说一并查集的方法确实比较麻烦。

这里我的并查集的方向是

```
a/b=2 , b/c=3

        c  1
        ^
        |
        b  3
        ^
        |
        a  2
```

`val` 代表的是**当前节点是直接父节点**的多少倍，也就是 `A/fatherA` ，重点就是合并两个集合的时候需要注意：

```
已知
a / fa = val[a]
b / fb = val[b]
现在我们要合并a，b且 a / b=value
所以我们需要设置 parent[fa]=fb
由于fa父节点发生了变化所以它的值也需要变化，也就是要求 fa/fb的值
val[fa] = fa/fb = a/b * b/fb * fa/a = value * (val[b] / val[a])
```

这里可以看到已经省略了 `cal` 函数计算从当前节点到根节点的总权值积，因为这里路径压缩已经将树压缩到只有两层了，所以并不需要了

并查集代码 2（带路径压缩，更短）

```
class Solution {
    private HashMap<String,String> parent=new HashMap<>();
    private HashMap<String,Double> quotient=new HashMap<>();
    //带路径压缩的
    public String find(String p){
        if (parent.get(p)!=p) {
            //需要先保存父亲的值,因为后面压缩后树只有两层,后面*的就是根节点的权值1,是不对的
            //这里可以看看上面的并茶几的方向和值来判断
            String f=parent.get(p);
```

```
            parent.put(p,find(f));
            //这样压缩后的子节点才是正确的
            quotient.put(p,quotient.get(p)*quotient.get(f));
        }
        return parent.get(p);
    }

    public void init(String s){
        if (!parent.containsKey(s)) {
            parent.put(s,s);
            quotient.put(s,1.0);
        }
    }

    public void merge(String a,String b,Double value){
        init(a);init(b);
        String fa=find(a); // a/fa=val[a], b/fb=val[b]
        String fb=find(b);
        if (fa.equals(fb)) {
            return;
        }
        parent.put(fa,fb);
        quotient.put(fa,value*(quotient.get(b)/quotient.get(a)));
    }

    public double[] calcEquation(List<List<String>> equations, double[]
 values, List<List<String>> queries) {
        for (int i=0;i<equations.size();i++) {
            List<String> equation=equations.get(i);
            merge(equation.get(0),equation.get(1),values[i]);
        }
        double[] res=new double[queries.size()];
        int index=0;
        for (List<String> query:queries) {
            String a=query.get(0);
            String b=query.get(1);
            if (!parent.containsKey(a) || !parent.containsKey(b)) {
                res[index++]=-1;
            }else{
                //先做路径压缩
                res[index++]=find(a).equals(find(b))?quotient.get(a)/qu
otient.get(b):-1;
            }
        }
```

```
            return res;
        }
    }
```

30、跳跃游戏 3（1306）

1306. 跳跃游戏 III

难度 中等 38 收藏 分享 切换为英文 关注 反馈

这里有一个非负整数数组 arr，你最开始位于该数组的起始下标 start 处。当你位于下标 i 处时，你可以跳到 i + arr[i] 或者 i - arr[i]。

请你判断自己是否能够跳到对应元素值为 0 的 任一 下标处。

注意，不管是什么情况下，你都无法跳到数组之外。

示例 1：

输入： arr = [4,2,3,0,3,1,2], start = 5
输出： true
解释：
到达值为 0 的下标 3 有以下可能方案：
下标 5 -> 下标 4 -> 下标 1 -> 下标 3
下标 5 -> 下标 6 -> 下标 4 -> 下标 1 -> 下标 3

示例 2：

输入： arr = [4,2,3,0,3,1,2], start = 0
输出： true
解释：
到达值为 0 的下标 3 有以下可能方案：
下标 0 -> 下标 4 -> 下标 1 -> 下标 3

示例 3：

输入： arr = [3,0,2,1,2], start = 2
输出： false
解释： 无法到达值为 0 的下标 1 处。

提示：

- 1 <= arr.length <= 5 * 10^4
- 0 <= arr[i] < arr.length
- 0 <= start < arr.length

https://leetcode-cn.com/problems/jump-game-iii/solution/a-fei-xue-suan-fa-zhi-si-ke-yi-dao-ti-1306-tiao-yu/

> 最近刷跳跃游戏数题，做个记录

方法1： DFS

- 准备一个函数：dfs(int[] arr, int curPos, boolean[] visited)
  - 其中 curPos 表示当前访问的位置
  - visited 表示当前的 curPos 位置有无被访问过
- 出口条件：
  - 当前 curPos 越界了，也就是不在 [0,len-1] 范围内时，返回 false
  - 当前 curPos 的访问过，返回 false
  - 当 arr[curPos]==0 时，表示找到了，返回 true
- 探索左边和右边位置

更多硬核内容就在公众号「yes 的练级攻略」

方法 1：DFS

```
class Solution {
    public boolean canReach(int[] arr, int start) {
        boolean[] visited = new boolean[arr.length];
        return dfs (arr, start, visited);
    }
    private boolean dfs (int[] arr, int curPos, boolean[] visited){
        if (curPos < 0 || curPos >= arr.length || visited[curPos]) retu
rn false;
        if (arr[curPos] == 0) return true;
        visited[curPos] = true;
        return dfs(arr, curPos - arr[curPos], visited) || dfs(arr, curP
os + arr[curPos], visited);
    }
}
```

方法2： `BFS`

- 准备一个 `bool` 类型的数组 `visited` 表示当前的下标有无被访问过
- 准备一个 `queue` ，转这个 `queue`
  - 取到这一轮的总的 `size` 大小，进行 `for loop`
  - 弹出当前的 `curPos` ,如果 `arr[curPos]== 0` 说明找到了，返回 `true`
  - 分别渠道左右两边去找， `curPos` 的位置不越界并且 `leftPos` 和 `rightPos` 未被访问过
  - 访问后要设置下 `visited` 的属性，并且将位置放置于 `queue` 中

方法 2：BFS

```
class Solution {
    public boolean canReach(int[] arr, int start) {
        boolean[] visited = new boolean[arr.length];
        LinkedList<Integer> que = new LinkedList<>();
        que.add(start);
        while (!que.isEmpty()){
            int size = que.size();
            for (int i = 0; i < size; i++){
                int curPos = que.removeFirst();
                int curVal = arr[curPos];
                if (curVal == 0) return true;
                int leftPos = curPos - curVal;
                if (leftPos >= 0 && !visited[leftPos]){
                    visited[leftPos] = true;
                    que.addFirst(leftPos);
                }
                int rightPos = curPos + curVal;
                if (rightPos < arr.length && !visited[rightPos]) {
                    visited[rightPos] = true;
                    que.addFirst(rightPos);
```

```
                }
            }
        }
        return false;
    }
}
```

31、跳跃游戏 4（1345）

1345. 跳跃游戏 IV

难度 困难 31 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `arr` ，你一开始在数组的第一个元素处（下标为 0）。

每一步，你可以从下标 `i` 跳到下标：

- `i + 1` 满足：`i + 1 < arr.length`
- `i - 1` 满足：`i - 1 >= 0`
- `j` 满足：`arr[i] == arr[j]` 且 `i != j`

请你返回到达数组最后一个元素的下标处所需的 **最少操作次数** 。

注意：任何时候你都不能跳到数组外面。

示例 1：

输入： `arr = [100,-23,-23,404,100,23,23,23,3,404]`
输出： `3`
解释： 那你需要跳跃 `3` 次，下标依次为 `0 --> 4 --> 3 --> 9` 。下标 `9` 为数组的最后一个元素的下标。

示例 2：

输入： `arr = [7]`
输出： `0`
解释： 一开始就在最后一个元素处，所以你不需要跳跃。

示例 3：

输入： `arr = [7,6,9,6,9,6,9,7]`
输出： `1`
解释： 你可以直接从下标 `0` 处跳到下标 `7` 处，也就是数组的最后一个元素处。

示例 4：

```
输入：arr = [6,1,9]
输出：2
```

示例 5：

```
输入：arr = [11,22,7,7,7,7,7,7,7,22,13]
输出：3
```

提示：

- `1 <= arr.length <= 5 * 10^4`
- `-10^8 <= arr[i] <= 10^8`

https://leetcode-cn.com/problems/jump-game-iv/solution/wei-shi-yao-bfsyi-ji-zen-yang-on-by-tian-ye/

题目要求计算从第一个元素跳到最后一个元素最少的操作次数。首先容易想到的是，坐标位于最后一个元素时，结果应为0，因为不用任何操作；对于这个元素紧挨着的左右元素（此时只存在左边的元素），它的操作数是1，因为只需要跳一次就跳到了最后；同理，对于数组里值和最后一个元素相等的那些数，他们的操作数都是1。
然后，对于倒数第二个元素，它的操作数前一步已经求得为1，它右边数的结果也已经知道了，是0；它左边的数如果不等于最后一个数，那么它跟最后一个数的 距离 应为2。这里距离的意思是指操作次数，即跳两次才能到达最后一个数；同样，和倒数第二个数相等的那些元素最小的操作数也都是1+1=2.

这样，可以理解为最后一个数所在层数为 0 ，第 1 层就是挨着它的元素以及值和它相等的那些数，第 2 层就是第一层挨着的元素以及值和第一层元素相等的那些数...按层次依次往上加。因为题目要求最少操作数，对于"挨着它的元素"如果前面已经求出了距离（层数），这时就不用再去算了，因为+1之后结果只会更大。例如，用一个数组 dis[] 表示每一个元素对应的最小操作数，初始化一个大数，如果对于元素 i 它左边有 dis[i-1] 等于这个大数，就表明 i-1 位置还没有求，于是 dis[i-1]=dis[i]+1 ;如果 dis[i-1] 不等于这个大数就说明左边的元素前面已经算过了，就不要再去给它+1。

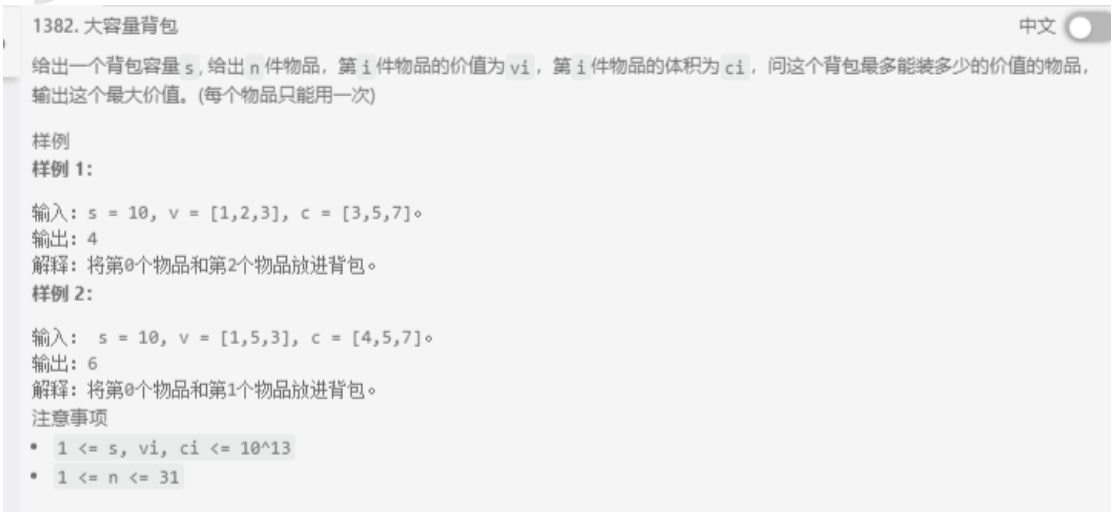
为了去找数组中和某一个数相等的所有元素位置，使用 HashMap 来存下所有值和相应位置（列表）的映射，可以避免一遍遍遍历这个数组去寻找。
（这时候你就可以自己动手去写了，小总结：一个数组存结果，一个Map映射和一个队列来广度优先搜索）

现在写出的代码运行那几个试例是没问题的，不过提交的时候会遇到一个满页是7，最后一个数是13的测例，可能会超时。这里还可以继续优化：对于那堆7，从倒数第二个元素开始其实就已经算出了他们的距离是[2,2,...,2,1,0],每次他都会把新出现的7放入队列，然后又遍历一遍7所在的所有位置。于是可以用一个数组 visited[] 来记录这个数是否被访问（计算）过，如果已经访问过了就没必要再去map里遍历它对应的所有位置了。

```
class Solution {
    final int MAX = Integer.MAX_VALUE;
    public int minJumps(int[] arr) {
        int n = arr.length;
        Queue<Integer> queue = new LinkedList<>();
        Map<Integer, List<Integer>> map = new HashMap<>();
        for(int i = 0; i < n; i++){
            int a = arr[i];
            List<Integer> list = map.getOrDefault(a, new ArrayList<>())
;
            list.add(i);
            map.put(a, list);
        }
        int[] dis = new int[n]; //每个元素到最后一个元素的距离，用来存结果
        Arrays.fill(dis, MAX);
```

```
            dis[n-1] = 0;
            queue.offer(n-1);
            boolean[] visited = new boolean[n]; //记录元素是否被访问过，初始都是false
            while(!queue.isEmpty()){
                int x = queue.poll();
                if(x-1 >= 0 && dis[x-1] == MAX){ //等于MAX 说明x 左边的元素还没有计算过
                    dis[x-1] = dis[x]+1;
                    queue.offer(x-1);
                }
                if(x+1 < n && dis[x+1] == MAX){
                    dis[x+1] = dis[x]+1;
                    queue.offer(x+1);
                }
                if(!visited[x]){
                    for(int i: map.get(arr[x])){
                        if(dis[i] == MAX){
                            dis[i] = dis[x]+1;
                            queue.offer(i);
                            visited[i] = true;
                        }
                    }
                }
            }
            return dis[0];
        }
}
```

32、大容量背包（lintcode 1382）



1382. 大容量背包 中文

给出一个背包容量 s，给出 n 件物品，第 i 件物品的价值为 vi，第 i 件物品的体积为 ci，问这个背包最多能装多少的价值的物品，输出这个最大价值。(每个物品只能用一次)

样例
样例 1:

输入: s = 10, v = [1,2,3], c = [3,5,7]。
输出: 4
解释: 将第0个物品和第2个物品放进背包。
样例 2:

输入: s = 10, v = [1,5,3], c = [4,5,7]。
输出: 6
解释: 将第0个物品和第1个物品放进背包。
注意事项

- 1 <= s, vi, ci <= 10^13
- 1 <= n <= 31

```
public class Solution {
    /**
```

```
     * @param s: The capacity of backpack
     * @param v: The value of goods
     * @param c: The capacity of goods
     * @return: The answer
     */
     //DFS 版本,相当于组合问题，求所有满足和小于 s 的组合，再根据符合条件组合看看 value 是否大于 maxValue
    long maxValue = Long.MIN_VALUE;
    public long getMaxValue(int s, int[] v, int[] c) {
        // Write your code here

        for (int i = 0; i < c.length; i++) {
            dfs(s, v, c, i, c[i], v[i]);
        }
        return maxValue;
    }

    public void dfs(int s, int[] v, int[] c, int index, int sum, long value) {
        if (sum > s) {
            return;
        }

        if (value > maxValue) {
            maxValue = value;
        }
        for (int i = index + 1; i < c.length; i++) {
            dfs(s, v, c, i, sum + c[i], value + (long)v[i]);
        }
    }
}
```

33、机器人的运动范围（剑指 13）

剑指 Offer 13. 机器人的运动范围

难度 中等 139 收藏 分享 切换为英文 关注 反馈

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于k的格子。例如，当k为18时，机器人能够进入方格 [35, 37] ，因为3+5+3+7=18。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

示例 1：

```
输入：m = 2, n = 3, k = 1
输出：3
```

示例 2：

```
输入：m = 3, n = 1, k = 0
输出：1
```

提示：

- 1 <= n,m <= 100
- 0 <= k <= 20

禁止商用 公众号：yes的练级攻略

方法 1：DFS（1ms）

```
class Solution {
    public int movingCount(int m, int n, int k) {
        if (k == 0) return 1;
        boolean[][] isVisited = new boolean[m][n];
        int cnt = dfs(m, n, k, 0, 0, isVisited);
        return cnt;
    }
    public int dfs(int m , int n, int k, int r, int c, boolean[][] isVisited){
        if (r < 0 || r >= m || c < 0 || c >= n || isVisited[r][c] || cal(r) + c
al(c) > k) return 0;
        isVisited[r][c] = true;
        return dfs(m, n, k, r-1, c, isVisited) + dfs(m, n, k, r+1, c, isVisited
)+ dfs(m, n, k, r, c-1, isVisited) + dfs(m, n, k, r, c+1, isVisited) + 1;
    }
    public int cal(int num){
        int sum = 0;
        while (num > 0) {
            sum += num % 10;  //个位
            num /= 10; //移位到下一高位
        }
        return sum;
    }
}
```

https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/solution/mian-shi-ti-13-ji-qi-ren-de-yun-dong-fan-wei-dfs-b/

- BFS/DFS： 两者目标都是遍历整个矩阵，不同点在于搜索顺序不同。DFS 是朝一个方向走到底，再回退，以此类推；BFS 则是按照"平推"的方式向前搜索。
- BFS 实现： 通常利用队列实现广度优先遍历。

算法解析：

- **初始化：** 将机器人初始点 $(0, 0)$ 加入队列 `queue` ；
- **迭代终止条件：** `queue` 为空。代表已遍历完所有可达解。
- **迭代工作：**
  1. **单元格出队：** 将队首单元格的 索引、数位和 弹出，作为当前搜索单元格。
  2. **判断是否跳过：** 若 ① 行列索引越界 **或** ② 数位和超出目标值 `k` **或** ③ 当前元素已访问过 时，执行 `continue` 。
  3. **标记当前单元格：** 将单元格索引 `(i, j)` 存入 Set `visited` 中，代表此单元格 **已被访问过** 。
  4. **单元格入队：** 将当前元素的 **下方、右方** 单元格的 **索引、数位和** 加入 `queue` 。
- **返回值：** Set `visited` 的长度 `len(visited)` ，即可达解的数量。

复杂度分析：

> M, N 分别为矩阵行列大小。

- **时间复杂度 $O(MN)$：** 最差情况下，机器人遍历矩阵所有单元格，此时时间复杂度为 $O(MN)$ 。
- **空间复杂度 $O(MN)$：** 最差情况下，Set `visited` 内存储矩阵所有单元格的索引，使用 $O(MN)$ 的额外空间。

方法 2：BFS（7ms）

```
class Solution {
    public int movingCount(int m, int n, int k) {
        if (k == 0) return 1;
        boolean[][] isVisited = new boolean[m][n];
        int cnt = 0;
        Queue<int[]> que = new LinkedList<>();
        que.add(new int[] {0, 0, 0, 0});
        while (que.size() > 0) {
            int[] x = que.poll();
            int i = x[0], j = x[1], si = x[2], sj = x[3];
            if (i >= m || j >= n || k < si + sj || isVisited[i][j]) continue;
            isVisited[i][j] = true;
            cnt++;
            que.add(new int[] {i + 1, j, (i + 1) % 10 != 0 ? si + 1: si - 8, sj
});
            que.add(new int[] {i, j + 1, si, (j + 1) % 10 != 0 ? sj + 1 : sj -
8});
        }
        return cnt;
    }
}
```

34、打印从 1 到最大的 n 位数（剑指 17）

剑指 Offer 17. 打印从1到最大的n位数

难度 简单　41　收藏　分享　切换为英文　关注　反馈

输入数字 `n`，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

示例 1:

```
输入: n = 1
输出: [1,2,3,4,5,6,7,8,9]
```

说明：

- 用返回一个整数列表来代替打印
- n 为正整数

https://leetcode-cn.com/problems/da-yin-cong-1dao-zui-da-de-nwei-shu-lcof/solution/mian-shi-ti-17-da-yin-cong-1-dao-zui-da-de-n-wei-2/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

题目要求打印 “从 1 至最大的 n 位数的列表”，因此需考虑以下两个问题：

1. **最大的 n 位数（记为 end）和位数 n 的关系：** 例如最大的 1 位数是 9，最大的 2 位数是 99，最大的 3 位数是 999。则可推出公式：

$$end = 10^n - 1$$

2. **大数越界问题：** 当 n 较大时，end 会超出 $int32$ 整型的取值范围，超出取值范围的数字无法正常存储。但由于本题要求返回 int 类型数组，相当于默认所有数字都在 int32 整型取值范围内，因此不考虑大数越界问题。

因此，只需定义区间 $[1, 10^n - 1]$ 和步长 1，通过 for 循环生成结果列表 res 并返回即可。

复杂度分析：

- **时间复杂度 $O(10^n)$：** 生成长度为 10^n 的列表需使用 $O(10^n)$ 时间。
- **空间复杂度 $O(1)$：** 建立列表需使用 $O(1)$ 大小的额外空间（列表作为返回结果，不计入额外空间）。

方法 1：本题方法

```
class Solution {
    public int[] printNumbers(int n) {
        int end = (int)Math.pow(10, n) - 1;
        int[] res = new int[end];
        for(int i = 0; i < end; i++)
            res[i] = i + 1;
        return res;
    }
}
```

公众号：
yes的练级攻略

大数打印解法：

实际上，本题的主要考点是大数越界情况下的打印。需要解决以下三个问题：

1. 表示大数的变量类型：

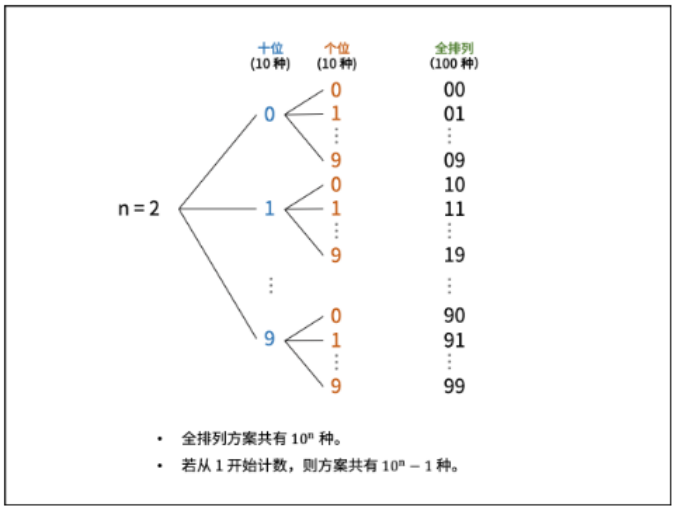
- 无论是 short / int / long ... 任意变量类型，数字的取值范围都是有限的。因此，大数的表示应用字符串 String 类型。

2. 生成数字的字符串集：

- 使用 int 类型时，每轮可通过 $+1$ 生成下个数字，而此方法无法应用至 String 类型。并且， String 类型的数字的进位操作效率较低，例如 `"9999"` 至 `"10000"` 需要从个位到千位循环判断，进位 4 次。
- 观察可知，生成的列表实际上是 n 位 0 - 9 的 **全排列** ，因此可避开进位操作，通过递归生成数字的 String 列表。

3. 递归生成全排列：

- 基于分治算法的思想，先固定高位，向低位递归，当个位已被固定时，添加数字的字符串。例如当 $n = 2$ 时（数字范围 $1 - 99$ ），固定十位为 0 - 9 ，按顺序依次开启递归，固定个位 0 - 9 ，终止递归并添加数字字符串。



观察可知，当前的生成方法仍有以下问题：

1. 诸如 $00, 01, 02, \cdots$ 应显示为 $0, 1, 2, \cdots$ ，即应 **删除高位多余的** 0 ；
2. 此方法从 0 开始生成，而题目要求 **列表从** 1 **开始** ；

以上两个问题的解决方法如下：

1. 删除高位多余的 0 **：**

- **字符串左边界定义：** 声明变量 $start$ 规定字符串的左边界，以保证添加的数字字符串 `num[start:]` 中无高位多余的 0 。例如当 $n = 2$ 时， $1 - 9$ 时 $start = 1$ ， $10 - 99$ 时 $start = 0$ 。
- **左边界** $start$ **变化规律：** 观察可知，当输出数字的所有位都是 9 时，则下个数字需要向更高位进 1 ，此时左边界 $start$ 需要减 1 （即高位多余的 0 减少一个）。例如当 $n = 3$ （数字范围 $1 - 999$ ）时，左边界 $start$ 需要减 1 的情况有： "009" 进位至 "010" ， "099" 进位至 "100" 。设数字各位中 9 的数量为 $nine$ ，所有位都为 9 的判断条件可用以下公式表示：

$$n - start = nine$$

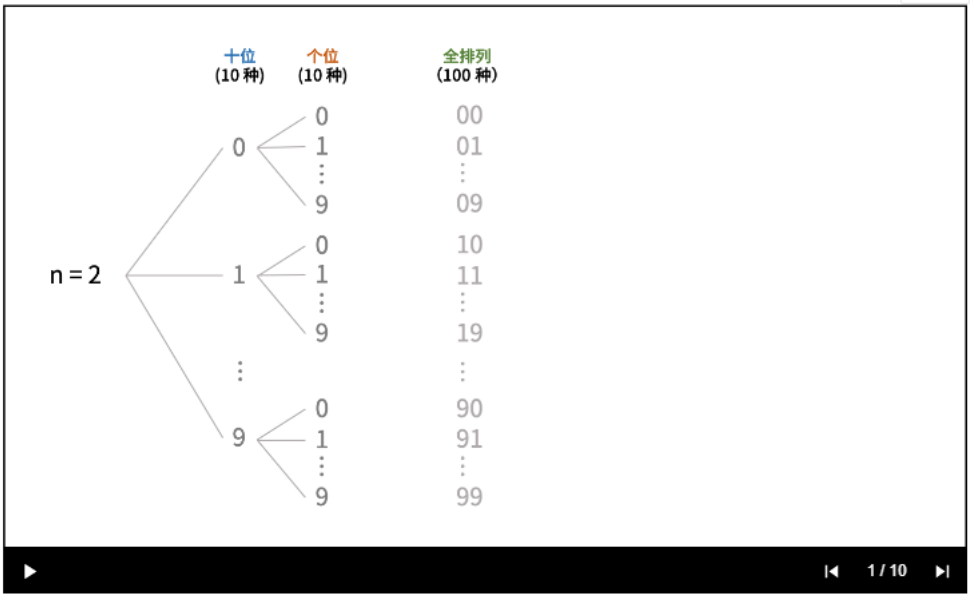
- **统计** $nine$ **的方法：** 固定第 x 位时，当 $i = 9$ 则执行 $nine = nine + 1$ ，并在回溯前恢复 $nine = nine - 1$ 。

更多硬核内容就在公众号「yes 的练级攻略」

2. 列表从 1 开始：

- 在以上方法的基础上，添加数字字符串前判断其是否为 `"0"` ，若为 `"0"` 则直接跳过。

☆ 收藏



复杂度分析：

- **时间复杂度** $O(10^n)$ ： 递归的生成的排列的数量为 10^n 。
- **空间复杂度** $O(n)$ ： 字符列表 num 使用线性大小的额外空间。

代码：

为 **正确表示大数** ，以下代码的返回值为数字字符串集拼接而成的长字符串。

方法 2：大数打印（DFS）

```
class Solution {
    int[] res;
    int nine = 0, count = 0, start, n;
    char[] num, loop = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
    public int[] printNumbers(int n) {
        this.n = n;
        res = new int[(int)Math.pow(10, n) - 1];
        num = new char[n];
        start = n - 1;
        dfs(0);
        return res;
    }
    void dfs(int x) {
        if(x == n) {
            String s = String.valueOf(num).substring(start);
            if(!s.equals("0")) res[count++] = Integer.parseInt(s);
            if(n - start == nine) start--;
            return;
        }
        for(char i : loop) {
```

```
            if(i == '9') nine++;
            num[x] = i;
            dfs(x + 1);
        }
        nine--;
    }
}
```

大数打印出字符串的代码

```
class Solution {
    StringBuilder res;
    int nine = 0, count = 0, start, n;
    char[] num, loop = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
    public String printNumbers(int n) {
        this.n = n;
        res = new StringBuilder();
        num = new char[n];
        start = n - 1;
        dfs(0);
        res.deleteCharAt(res.length() - 1);
        return res.toString();
    }
    void dfs(int x) {
        if(x == n) {
            String s = String.valueOf(num).substring(start);
            if(!s.equals("0")) res.append(s + ",");
            if(n - start == nine) start--;
            return;
        }
        for(char i : loop) {
            if(i == '9') nine++;
            num[x] = i;
            dfs(x + 1);
        }
        nine--;
    }
}
```

35、24 点游戏（679）

更多硬核内容就在公众号「yes 的练级攻略」

679. 24 点游戏

难度 困难 116 收藏 分享 切换为英文 关注 反馈

你有 4 张写有 1 到 9 数字的牌。你需要判断是否能通过 *，/，+，-，(，) 的运算得到 24。

示例 1:

```
输入: [4, 1, 8, 7]
输出: True
解释: (8-4) * (7-1) = 24
```

示例 2:

```
输入: [1, 2, 1, 2]
输出: False
```

注意:

1. 除法运算符 / 表示实数除法，而不是整数除法。例如 4 / (1 - 2/3) = 12 。
2. 每个运算符对两个数进行运算。特别是我们不能用 - 作为一元运算符。例如，[1, 1, 1, 1] 作为输入时，表达式 -1 - 1 - 1 - 1 是不允许的。
3. 你不能将数字连接在一起。例如，输入为 [1, 2, 1, 2] 时，不能写成 12 + 12 。

https://leetcode-cn.com/problems/24-game/solution/24dian-you-xi-by-leetcode/

方法：回溯

思路和算法

只有 4 张牌，且只能执行 4 种操作。即使所有运算符都不进行交换，最多也只有 $12 * 6 * 2 * 4 * 4 * 4 = 9216$ 种可能性，这使得我们可以尝试所有这些可能。

具体来说，我们有 12 种方式先选出两个数字（有序），并执行 4 种操作之一（12 * 4）。然后，剩下 3 个数字，我们从中选择 2 个并执行 4 种操作之一（6 * 4）。

最后我们剩下两个数字，并在 2 * 4 种可能之中作出最终选择。

我们将对我们的数字或结果数字执行 3 次二元运算（+，-，*，/ 是运算）。因为 - 和 / 不满足交换律，我们必须仔细考虑 a / b 和 b / a。

对于在我们的列表中移除 a, b 这两个数字的每一种方法，以及它们可能产生的每种结果，如 a + b、a / b 等，我们将采用递归的方法解决这个较小的数字列表上的问题。

复杂度分析

- 时间复杂度：$O(1)$，总计 9216 种可能的硬性限制，对于每种可能，我们执行操作的复杂度为 $O(1)$。
- 空间复杂度：$O(1)$，我们的中间数组最多有 4 个元素，所生成的数字由复杂度 $O(1)$ 的因子限定。

```
class Solution {
    public boolean judgePoint24(int[] nums) {
        ArrayList A = new ArrayList<Double>();
        for (int v: nums) A.add((double) v);
        return solve(A);
    }
    private boolean solve(ArrayList<Double> nums) {
        if (nums.size() == 0) return false;
        if (nums.size() == 1) return Math.abs(nums.get(0) - 24) < 1e-6;

        for (int i = 0; i < nums.size(); i++) {
            for (int j = 0; j < nums.size(); j++) {
                if (i != j) {
```

```
                    ArrayList<Double> nums2 = new ArrayList<Double>();
                    for (int k = 0; k < nums.size(); k++) if (k != i && k != j)
{
                        nums2.add(nums.get(k));
                    }
                    for (int k = 0; k < 4; k++) {
                        if (k < 2 && j > i) continue;
                        if (k == 0) nums2.add(nums.get(i) + nums.get(j));
                        if (k == 1) nums2.add(nums.get(i) * nums.get(j));
                        if (k == 2) nums2.add(nums.get(i) - nums.get(j));
                        if (k == 3) {
                            if (nums.get(j) != 0) {
                                nums2.add(nums.get(i) / nums.get(j));
                            } else {
                                continue;
                            }
                        }
                        if (solve(nums2)) return true;
                        nums2.remove(nums2.size() - 1);
                    }
                }
            }
        }
        return false;
    }
}
```

36、安卓系统手势解锁（351）

yes的练级攻略

351. 安卓系统手势解锁

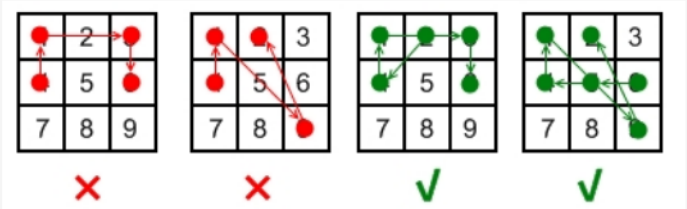
难度 中等 49 收藏 分享 切换为英文 关注 反馈

我们都知道安卓有个手势解锁的界面，是一个 **3 x 3** 的点所绘制出来的网格。

给你两个整数，分别为 **m** 和 **n**，其中 1 ≤ m ≤ n ≤ 9，那么请你统计一下有多少种解锁手势，是至少需要经过 **m** 个点，但是最多经过不超过 **n** 个点的。

先来了解下什么是一个有效的安卓解锁手势:

1. 每一个解锁手势必须至少经过 **m** 个点、最多经过 **n** 个点。
2. 解锁手势里不能设置经过重复的点。
3. 假如手势中有两个点是顺序经过的，那么这两个点的手势轨迹之间是绝对不能跨过任何未被经过的点。
4. 经过点的顺序不同则表示为不同的解锁手势。



解释:

```
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
```

无效手势: `4 - 1 - 3 - 6`
连接点 1 和点 3 时经过了未被连接过的 2 号点。

无效手势: `4 - 1 - 9 - 2`
连接点 1 和点 9 时经过了未被连接过的 5 号点。

有效手势: `2 - 4 - 1 - 3 - 6`
连接点 1 和点 3 是有效的，因为虽然它经过了点 2 ，但是点 2 在该手势中之前已经被连过了。

有效手势: `6 - 5 - 4 - 1 - 9 - 2`
连接点 1 和点 9 是有效的，因为虽然它经过了按键 5 ，但是点 5 在该手势中之前已经被连过了。

示例:

```
输入: m = 1, n = 1
输出: 9
```

https://leetcode-cn.com/problems/android-unlock-patterns/solution/an-zhuo-xi-tong-shou-shi-jie-suo-by-klb/

解题思路

题目要求统计至少经过 m 个点，但是不超过 n 个点的所有解锁手势种类数量，因此，我们分别统计走过 m、走过 m+1、...、走过 n 个点的解锁手势种类，然后全部加起来就是至少经过 m 个点又不超过 n 个点的手势种类数。

以下为统计走过 i 个点的手势种类数计算：

我们从一个点出发，不断滑来滑去，直到总共划过 i 个点，就是一种解锁手势，现在要统计总共有多少种，很明显要进行 DFS，即深度优先搜索遍历。

而且再仔细观察本题，会发现一个规律：从 1、3、7、9 出发的种类数是一样的，从 2、4、6、8 出发的种类数也是一样的。

因此，我们只要计算从 1 出发的解锁种类数；然后乘以4，计算 从 2 出发的解锁种类数，然后乘以 4；最后加上从 5 出发的解锁种类数；三者加起来，就是经过 i 个点的手势种类数了。

本题还有一个要求，即，如果两个点中间存在未走过的点，是不能走过去的。

因为总共才 1-9 的点，我们用二维整型数组 skip[][] 来记录两点中间的点的情况，比如 skip[1][3] = 2，表示 1 和 3 中间有一个 2。

再用一个一维布尔型数组 visited[] 来表示某个点是否走过，比如 visited[3] true 表示 3 已经走过。

明白了上面的准备工作，我们就可以设计 DFS 函数了，DFS 函数的功能是：统计从某个点出发的所有不同的解锁手势的种类数。

public int DFS(int current, boolean []visited, int [][]skip,int remainKeyCount);

其中，current 表示出发点，remainKeyCount 表示走了 current 后的剩余可走步数。

注意：在 DFS 函数的开始，要把 current 设置为 true，到 DFS 函数结束之前，要重新设置为 false。毕竟每一个点都是可以重复利用的。

```
class Solution {
    public int numberOfPatterns(int m, int n) {
        int[][] skip = new int[10][10];
        //这个 skip 数组是为了记录跳跃的点数，比如说从 1 到 3，就跳跃 2
        //而且因为是对称的操作，所以 3 到 1 也是如此
        skip[1][3] = skip[3][1] = 2;
        skip[1][7] = skip[7][1] = 4;
        skip[3][9] = skip[9][3] = 6;
        skip[4][6] = skip[6][4] = skip[2][8] = skip[8][2] = 5;
        skip[1][9] = skip[9][1] = skip[3][7] = skip[7][3] = 5;
        skip[7][9] = skip[9][7] = 8;
        int result = 0;
        boolean[] visited = new boolean[10];
        //深度遍历所有情况，即走过的路程分别为 m,m+1,...,n-1,n 时的方法数
        for(int i = m; i<=n; i++){
            // 为什么 i 要减一呢，因为我们是从 i 出发，即出发也算走了一步
            //因为从 1,3,7,9 出发都是对称的
            result += DFS(1,visited,skip,i-1)*4;
            //2,4,6,8 对称
            result += DFS(2,visited,skip,i-1)*4;
            //唯独 5 独立
            result += DFS(5,visited,skip,i-1);
        }
```

```
        return result;
    }

    /**
     *
     * @param current    起点
     * @param visited    保存所有点是否走过的布尔数组
     * @param skip       记录两点之间是否存在其他点的二维数组
     * @param remainKeyCount     剩余可走步数
     * @return
     */
    public int DFS(int current, boolean []visited, int [][]skip,int remainKeyCo
unt){
        if(remainKeyCount == 0){
            return 1;
        }
        int result = 0; // 从这个点开始，往后走完 remainKeyCount 的所有可能情况数
        visited[current] = true;    // 当前点已经走过了
        // 遍历 current 的下一步所有的可能目标
        for(int i = 1; i <= 9; i++){
            // 看当前的节点到i 节点的路径中有没有其他节点在中间
            int crossThroughNumber = skip[current][i];
            // 如果这一次我们的i 节点没有被读过，如果读过，就忽略这个点
            // 如果没用过，看能不能走过去
            // 即判断有没有路过中间节点(visited[crossThroughNumber])或者这两个节点
相邻没有中间节点（currentThrough=0）
            if(!visited[i] && (crossThroughNumber == 0 ||visited[crossThroughNu
mber])){
                // current 点已经处理好，剩下的路就交给 i 点了，已走了一步，因
此 remainKeyCount-1
                result += DFS(i,visited,skip,remainKeyCount-1);
            }
        }
        visited[current] = false;   // 已经统计好 current 往下走的所有可能情况数，
重置为 false
        return result;
    }
}
```

37、查找集群内的[关键连接]（1192）

1192. 查找集群内的「关键连接」

难度 困难 64 收藏 分享 切换为英文 关注 反馈

力扣数据中心有 n 台服务器，分别按从 0 到 n-1 的方式进行了编号。

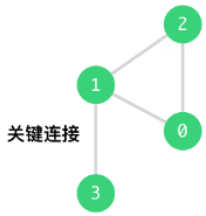
它们之间以「服务器到服务器」点对点的形式相互连接组成了一个内部集群，其中连接 connections 是无向的。

从形式上讲，connections[i] = [a, b] 表示服务器 a 和 b 之间形成连接。任何服务器都可以直接或者间接地通过网络到达任何其他服务器。

「关键连接」是在该集群中的重要连接，也就是说，假如我们将它移除，便会导致某些服务器无法访问其他服务器。

请你以任意顺序返回该集群内的所有「关键连接」。

示例 1：



输入： n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]
输出： [[1,3]]
解释： [[3,1]] 也是正确的。

提示：

- 1 <= n <= 10^5
- n-1 <= connections.length <= 10^5
- connections[i][0] != connections[i][1]
- 不存在重复的连接

2.[Tarjan算法]

Tarjan算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量。

> DFN[] 数组存储访问顺序，也就是先遍历的点，会先分配一个的**序号**(小到大)，然后序号存在这个数组当中。
> 如：DFN(u)为节点u搜索的次序编号(时间戳)

> Low[] 表示该点所能**直接**或**间接**达到时间**最小的顶点**。如： Low(u)为u或u的子树能够追溯到的最早的栈中节点的次序号

> stack 存储该连通子图的所有点。

当DFN(u)=Low(u)时，以u为根的搜索子树上所有节点是一个强连通分量。

https://leetcode-cn.com/problems/critical-connections-in-a-network/solution/1192-cha-zhao-ji-qun-nei-de-guan-jian-lian-jie-jav/

```
class Solution {
    private List<Integer>[] edges;
    private int[] DFN;
```

```
    private int[] LOW;
    private boolean[] visited;
    private List<List<Integer>> ans;
    private int t;
    public List<List<Integer>> criticalConnections(int n, List<List<Int
eger>> connections) {
        this.edges = new ArrayList[n];
        this.DFN = new int[n];
        this.LOW = new int[n];
        this.visited = new boolean[n];
        this.ans = new ArrayList<>();
        this.t = 0;
        for(int i = 0; i < n; i ++) {
            edges[i] = new ArrayList<>();
        }
        for (List<Integer> conn : connections) {
            int n1 = conn.get(0), n2 = conn.get(1);
            edges[n1].add(n2);
            edges[n2].add(n1);
        }
        tarjan(0, -1);
        return ans;
    }
    public void tarjan(int cur, int pre) {
        t ++;
        DFN[cur] = t;
        LOW[cur] = t;
        visited[cur] = true;
        for (int node : edges[cur]) {
            if (node == pre) continue;
            if (!visited[node]) {
                tarjan(node, cur);
                LOW[cur] = Math.min(LOW[cur], LOW[node]);
                if (LOW[node] > DFN[cur]) {
                    List<Integer> list = new ArrayList<>();
                    list.add(cur);
                    list.add(node);
                    ans.add(list);
                }
            } else {
                LOW[cur] = Math.min(LOW[cur], DFN[node]);
            }
        }
    }
```

}

38、重新安排行程（332）

332. 重新安排行程

难度 中等 174 收藏 分享 切换为英文 关注 反馈

给定一个机票的字符串二维数组 [from, to]，子数组中的两个成员分别表示飞机出发和降落的机场地点，对该行程进行重新规划排序。所有这些机票都属于一个从 JFK（肯尼迪国际机场）出发的先生，所以该行程必须从 JFK 开始。

说明：

1. 如果存在多种有效的行程，你可以按字符自然排序返回最小的行程组合。例如，行程 ["JFK", "LGA"] 与 ["JFK", "LGB"] 相比就更小，排序更靠前
2. 所有的机场都用三个大写字母表示（机场代码）。
3. 假定所有机票至少存在一种合理的行程。

示例 1:

```
输入: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
输出: ["JFK", "MUC", "LHR", "SFO", "SJC"]
```

示例 2:

```
输入: [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
输出: ["JFK","ATL","JFK","SFO","ATL","SFO"]
解释: 另一种有效的行程是 ["JFK","SFO","ATL","JFK","ATL","SFO"]。但是它自然排序更大更靠后。
```

https://leetcode-cn.com/problems/reconstruct-itinerary/solution/zhong-xin-an-pai-xing-cheng-by-leetcode-solution/

本题和 753. 破解保险箱 类似，是力扣平台上为数不多的求解欧拉回路 / 欧拉通路的题目。读者可以配合着进行练习。

我们化简本题题意：给定一个 n 个点 m 条边的图，要求从指定的顶点出发，经过所有的边恰好一次（可以理解为给定起点的「一笔画」问题），使得路径的字典序最小。

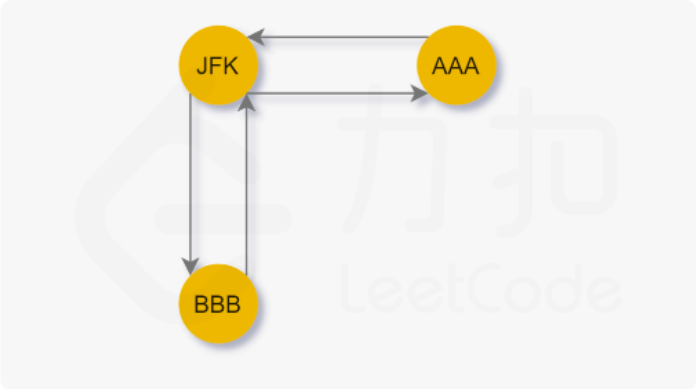
这种「一笔画」问题与欧拉图或者半欧拉图有着紧密的联系，下面给出定义：

- 通过图中所有边恰好一次且行遍所有顶点的通路称为欧拉通路。
- 通过图中所有边恰好一次且行遍所有顶点的回路称为欧拉回路。
- 具有欧拉回路的无向图称为欧拉图。
- 具有欧拉通路但不具有欧拉回路的无向图称为半欧拉图。

因为本题保证至少存在一种合理的路径，也就告诉了我们，这张图是一个欧拉图或者半欧拉图。我们只需要输出这条欧拉通路的路径即可。

> 如果没有保证至少存在一种合理的路径，我们需要判别这张图是否是欧拉图或者半欧拉图，具体地：
> - 对于无向图 G，G 是欧拉图当且仅当 G 是连通的且没有奇度顶点。
> - 对于无向图 G，G 是半欧拉图当且仅当 G 是连通的且 G 中恰有 2 个奇度顶点。
> - 对于有向图 G，G 是欧拉图当且仅当 G 的所有顶点属于同一个强连通分量且每个顶点的入度和出度相同。
> - 对于有向图 G，G 是半欧拉图当且仅当 G 的所有顶点属于同一个强连通分量且
>   - 恰有一个顶点的出度与入度差为 1；
>   - 恰有一个顶点的入度与出度差为 1；
>   - 所有其他顶点的入度和出度相同。

让我们考虑下面的这张图：



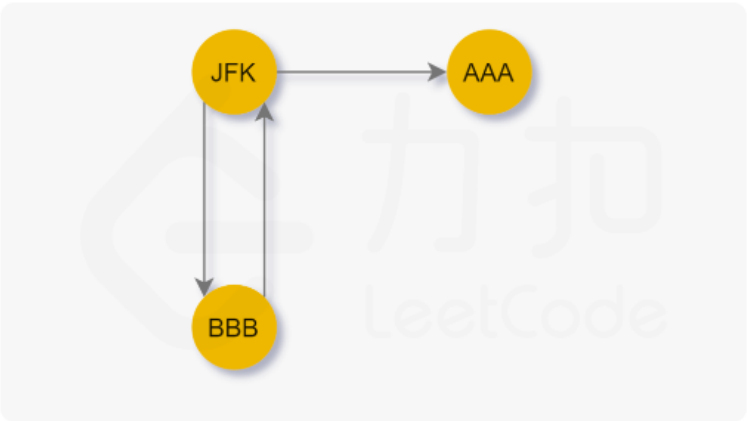
我们从起点 JFK 出发，合法路径有两条：

- JFK → AAA → JFK → BBB → JFK
- JFK → BBB → JFK → AAA → JFK

既然要求字典序最小，那么我们每次应该贪心地选择当前节点所连的节点中字典序最小的那一个，并将其入栈。最后栈中就保存了我们遍历的顺序。

为了保证我们能够快速找到当前节点所连的节点中字典序最小的那一个，我们可以使用优先队列存储当前节点所连到的点，每次我们 $O(1)$ 地找到最小字典序的节点，并 $O(\log m)$ 地删除它。

然后我们考虑一种特殊情况：



当我们先访问 AAA 时，我们无法回到 JFK，这样我们就无法访问剩余的边了。

也就是说，当我们贪心地选择字典序最小的节点前进时，我们可能先走入「死胡同」，从而导致无法遍历到其他还未访问的边。于是我们希望能够遍历完当前节点所连接的其他节点后再进入「死胡同」。

> 注意对于每一个节点，它只有最多一个「死胡同」分支。依据前言中对于半欧拉图的描述，只有那个入度与出度差为 1 的节点会导致死胡同。

更多硬核内容就在公众号「yes 的练级攻略」

方法一：Hierholzer 算法

思路及算法

Hierholzer 算法用于在连通图中寻找欧拉路径，其流程如下：

1. 从起点出发，进行深度优先搜索。
2. 每次沿着某条边从某个顶点移动到另外一个顶点的时候，都需要删除这条边。
3. 如果没有可移动的路径，则将所在节点加入到栈中，并返回。

当我们顺序地考虑该问题时，我们也许很难解决该问题，因为我们无法判断当前节点的哪一个分支是「死胡同」分支。

不妨倒过来思考。我们注意到只有那个入度与出度差为 1 的节点会导致死胡同。而该节点必然是最后一个遍历到的节点。我们可以改变入栈的规则，当我们遍历完一个节点所连的所有节点后，我们才将该节点入栈（即逆序入栈）。

对于当前节点而言，从它的每一个非「死胡同」分支出发进行深度优先搜索，都将会搜回到当前节点。而从它的「死胡同」分支出发进行深度优先搜索将不会搜回到当前节点。也就是说当前节点的死胡同分支将会优先于其他非「死胡同」分支入栈。

这样就能保证我们可以「一笔画」地走完所有边，最终的栈中逆序地保存了「一笔画」的结果。我们只要将栈中的内容反转，即可得到答案。

复杂度分析

- 时间复杂度：$O(m \log m)$，其中 m 是边的数量。对于每一条边我们需要 $O(\log m)$ 地删除它，最终的答案序列长度为 $m+1$，而与 n 无关。
- 空间复杂度：$O(m)$，其中 m 是边的数量。我们需要存储每一条边。

1. 由于题目中说必然存在一条有效路径(至少是半欧拉图)，所以算法不需要回溯（既加入到结果集里的元素不需要删除）
2. 整个图最多存在一个死胡同(出度和入度相差1），且这个死胡同一定是最后一个访问到的，否则无法完成一笔画。
3. DFS的调用其实是一个拆边的过程（既每次调用删除一条边），一定是递归到这个死胡同（无边可拆）后递归函数开始返回。所以死胡同是第一个加入栈中的元素。
4. 最后逆序的输出即可。

解题思路

我目前的思路是采用回溯法，即为每一条边设为未访问节点，从初始点开始，每次找到所有的未访问可行点，若不可行，则回退
回退的同时将之前访问过的边设为未访问节点，但是会导致死循环啊，莫法，看解析
首先这是一个一笔画问题，其中涉及到了欧拉回路，即不重复访问一个边来把一个图中所有边访问一次。
（无向图）欧拉路径存在充要条件（本题无关）：图中每个非起点（终点）的点入度和出度相差0，或者起始点的入度和出度差1，同时终点入度和出度差1，
（无向图）欧拉路径存在充要条件（本题无需证明）：起点入度比出度大1，终点入度比出度小1，其他点入度出度相同。
Hierholzier算法是通过给每条边建标注，不同于给普通的dfs给点加标注然后一个边一个边删去即可。

```
class Solution {
    public List<String> findItinerary(List<List<String>> tickets) {
        Map<String,PriorityQueue<String>> map=new HashMap<>();
        for(List<String> ticket:tickets){
            PriorityQueue<String> neighbor=map.computeIfAbsent(ticket.get(0),k->new PriorityQueue<>());
            neighbor.offer(ticket.get(1));
        }
        List<String> result=new LinkedList<>();
```

```
        dfs(map,"JFK",result);
        return result;
    }

    public void dfs(Map<String,PriorityQueue<String>> map,String src,List<Strin
g> result){
        PriorityQueue<String> queue=map.get(src);
        while(queue!=null&&queue.size()>0){
            String poll=queue.poll();
            dfs(map,poll,result);
        }
        result.add(0,src);
    }
}
```

39、破解保险箱（753）

753. 破解保险箱

难度 困难 47 收藏 分享 切换为英文 关注 反馈

有一个需要密码才能打开的保险箱。密码是 `n` 位数, 密码的每一位是 `k` 位序列 `0, 1, ..., k-1` 中的一个 。

你可以随意输入密码，保险箱会自动记住最后 `n` 位输入，如果匹配，则能够打开保险箱。

举个例子，假设密码是 `"345"`，你可以输入 `"012345"` 来打开它，只是你输入了 6 个字符.

请返回一个能打开保险箱的最短字符串。

示例1:

输入: `n = 1, k = 2`
输出: `"01"`
说明: `"10"`也可以打开保险箱。

示例2:

输入: `n = 2, k = 2`
输出: `"00110"`
说明: `"01100", "10011", "11001"` 也能打开保险箱。

提示:

1. `n` 的范围是 `[1, 4]`。
2. `k` 的范围是 `[1, 10]`。
3. `k^n` 最大可能为 `4096`。

https://leetcode-cn.com/problems/cracking-the-safe/solution/po-jie-bao-xian-xiang-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

前言

本题和 332. 重新安排行程 类似，是力扣平台上为数不多的求解欧拉回路 / 欧拉通路的题目。读者可以配合着进行练习。

方法一：Hierholzer 算法

Hierholzer 算法可以在一个欧拉图中找出欧拉回路。

具体地，我们将所有的 $n-1$ 位数作为节点，共有 k^{n-1} 个节点，每个节点有 k 条入边和出边。如果当前节点对应的数为 $a_1 a_2 \cdots a_{n-1}$，那么它的第 x 条出边就连向数 $a_2 \cdots a_{n-1} x$ 对应的节点。这样我们从一个节点顺着第 x 条边走到另一个节点，就相当于输入了数字 x。

在某个节点对应的数的末尾放上它的某条出边的编号，就形成了一个 n 位数，并且每个节点都能用这样的方式形成 k 个 n 位数。

> 例如 $k=4$，$n=3$ 时，节点分别为 $00, 01, 02, \cdots, 32, 33$，每个节点的出边的编号分别为 $0, 1, 2, 3$，那么 00 和它的出边形成了 $000, 001, 002, 003$ 这 4 个 3 位数，32 和它的出边形成了 $320, 321, 322, 323$ 这 4 个 3 位数。

这样共计有 $k^{n-1} \times k = k^n$ 个 n 位数，恰好就是所有可能的密码。

由于这个图的每个节点都有 k 条入边和出边，因此它一定存在一个欧拉回路，即可以从任意一个节点开始，一次性不重复地走完所有的边且回到该节点。因此，我们可以用 Hierholzer 算法找出这条欧拉回路：

设起始节点对应的数为 u，欧拉回路中每条边的编号为 $x_1, x_2, x_3, \cdots$，那么最终的字符串即为

$$u\ x_1\ x_2\ x_3 \cdots$$

Hierholzer 算法如下：

- 我们从节点 u 开始，**任意地**经过还未经过的边，直到我们「无路可走」。此时我们一定回到了节点 u，这是因为所有节点的入度和出度都相等。
- 回到节点 u 之后，我们得到了一条从 u 开始到 u 结束的回路，这条回路上仍然有些节点有未经过的出边。我么从某个这样的节点 v 开始，继续得到一条从 v 开始到 v 结束的回路，再嵌入之前的回路中，即

$$u \to \cdots \to v \to \cdots \to u$$

变为

$$u \to \cdots \to v \to \cdots \to v \to \cdots \to u$$

以此类推，直到没有节点有未经过的出边，此时我们就找到了一条欧拉回路。

实际的代码编写具有一定的技巧性。

复杂度分析

- 时间复杂度：$O(n \times k^n)$。
- 空间复杂度：$O(n \times k^n)$。

```
class Solution {
    Set<Integer> seen = new HashSet<Integer>();
    StringBuffer ans = new StringBuffer();
    int highest;
    int k;

    public String crackSafe(int n, int k) {
        highest = (int) Math.pow(10, n - 1);
        this.k = k;
        dfs(0);
        for (int i = 1; i < n; i++) {
```

```
            ans.append('0');
        }
        return ans.toString();
    }

    public void dfs(int node) {
        for (int x = 0; x < k; ++x) {
            int nei = node * 10 + x;
            if (!seen.contains(nei)) {
                seen.add(nei);
                dfs(nei % highest);
                ans.append(x);
            }
        }
    }
}
```

40、汉诺塔问题（面试题 08.06）

面试题 08.06. 汉诺塔问题

难度 简单 44 收藏 分享 切换为英文 关注 反馈

在经典汉诺塔问题中，有 3 根柱子及 N 个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。一开始，所有盘子自上而下按升序依次套在第一根柱子上(即每一个盘子只能放在更大的盘子上面)。移动圆盘时受到以下限制:
(1) 每次只能移动一个盘子;
(2) 盘子只能从柱子顶端滑出移到下一根柱子;
(3) 盘子只能叠在比它大的盘子上。

请编写程序，用栈将所有盘子从第一根柱子移到最后一根柱子。

你需要原地修改栈。

示例1:

```
输入: A = [2, 1, 0], B = [], C = []
输出: C = [2, 1, 0]
```

示例2:

```
输入: A = [1, 0], B = [], C = []
输出: C = [1, 0]
```

提示:

1. A中盘子的数目不大于14个。

https://leetcode-cn.com/problems/hanota-lcci/solution/chao-jian-dan-de-yi-nuo-ta-jiang-jie-you-liang-xi-/

【核心思想】

- 递归

【思路】

- 假设有n个盘子需要移动
- 首先将最上面的n-1个盘子从A移到B柱子
- 然后将最下面的一个盘子从A移到C柱子
- 最后将n-1个盘子从B移到C柱子
- 以上是汉诺塔的整体操作，其中移动n-1个盘子的操作是递归操作

```
class Solution {
    public void hanota(List<Integer> A, List<Integer> B, List<Integer> C) {
        move(A.size(),A, B, C);
    }
    public void move(int n, List<Integer> A, List<Integer> B, List<Integer> C){
        if(n == 1){
            C.add(A.remove(A.size() - 1));
            //注意：题目给的盘子是从大到小给的，所以这里 remove 的是【A.size() - 1】
            return;
        }
        move(n-1,A,C,B);
        C.add(A.remove(A.size() - 1));
        move(n - 1, B, A, C);
    }
}
```

三、双指针

刷题总结：https://www.cnblogs.com/codingstory/p/11334827.html

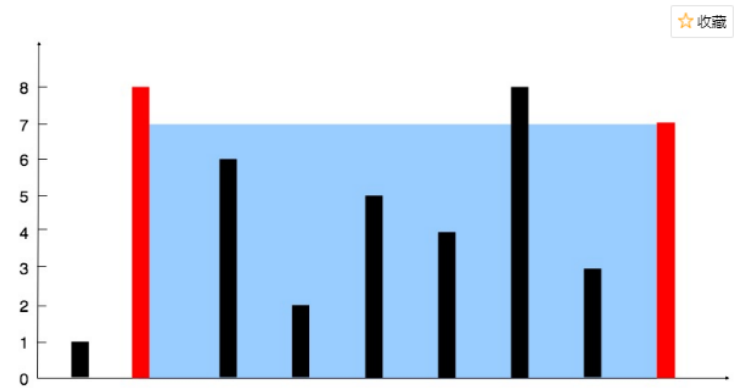
1、盛最多水的容器（11）

11. 盛最多水的容器

难度 中等 1515 收藏 分享 切换为英文 关注 反馈

给你 n 个非负整数 a_1，a_2，...，a_n，每个数代表坐标中的一个点 (i, a_i)。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, a_i) 和 $(i, 0)$。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

https://leetcode-cn.com/problems/container-with-most-water/solution/container-with-most-water-shuang-zhi-zhen-fa-yi-do/

更多硬核内容就在公众号「yes 的练级攻略」

思路：

- **算法流程：** 设置双指针 i,j 分别位于容器壁两端，根据规则移动指针（后续说明），并且更新面积最大值 `res` ，直到 `i == j` 时返回 `res` 。
- **指针移动规则与证明：** 每次选定围成水槽两板高度 $h[i],h[j]$ 中的短板，向中间收窄 1 格。以下证明：
  - 设每一状态下水槽面积为 $S(i,j),(0 <= i < j < n)$，由于水槽的实际高度由两板中的短板决定，则可得面积公式 $S(i,j) = min(h[i],h[j]) \times (j-i)$。
  - 在每一个状态下，无论长板或短板收窄 1 格，都会导致水槽 **底边宽度** -1：
    - 若向内移动短板，水槽的短板 $min(h[i],h[j])$ 可能变大，因此水槽面积 $S(i,j)$ 可能增大。
    - 若向内移动长板，水槽的短板 $min(h[i],h[j])$ 不变或变小，下个水槽的面积一定小于当前水槽面积。
  - 因此，向内收窄短板可以获取面积最大值。换个角度理解：
    - 若不指定移动规则，所有移动出现的 $S(i,j)$ 的状态数为 $C(n,2)$，即暴力枚举出所有状态。
    - 在状态 $S(i,j)$ 下向内移动短板至 $S(i+1,j)$ （假设 $h[i] < h[j]$ ），则相当于消去了 $S(i,j-1),S(i,j-2),...,S(i,i+1)$ 状态集合。而所有消去状态的面积一定 $<= S(i,j)$：
      - 短板高度：相比 $S(i,j)$ 相同或更短（$<= h[i]$）；
      - 底边宽度：相比 $S(i,j)$ 更短。
    - 因此**所有消去的状态的面积都** $< S(i,j)$。通俗的讲，我们每次向内移动短板，所有的消去状态都**不会导致丢失面积最大值**。
- **复杂度分析：**
  - 时间复杂度 $O(N)$，双指针遍历一次底边宽度 N 。
  - 空间复杂度 $O(1)$，指针使用常数额外空间。

注意：与直方图最大矩形覆盖不同，那题不能用双指针

```
class Solution {
    public int maxArea(int[] height) {
        int i = 0, j = height.length - 1, res = 0;
        while(i < j){
            res = height[i] < height[j] ?
                Math.max(res, (j - i) * height[i++]):
                Math.max(res, (j - i) * height[j--]);
        }
        return res;
    }
}
```

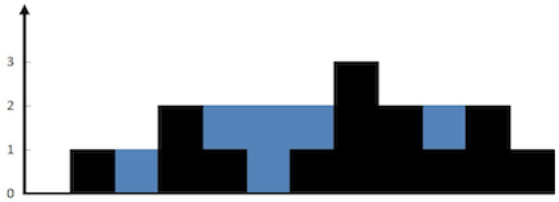
2、接雨水（42）【动规】【双指针】【单调栈】【直方图的水量（面试题 17.21）】

更多硬核内容就在公众号「yes 的练级攻略」

42. 接雨水

难度 困难 1370 收藏 分享 切换为英文 关注 反馈

给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。 **感谢 Marcos** 贡献此图。

示例:

```
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

https://leetcode-cn.com/problems/trapping-rain-water/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-w-8/

解法三: 动态规划

我们注意到，解法二中。对于每一列，我们求它左边最高的墙和右边最高的墙，都是重新遍历一遍所有高度，这里我们可以优化一下。

首先用两个数组， `max_left [i]` 代表第 `i` 列左边最高的墙的高度， `max_right[i]` 代表第 `i` 列右边最高的墙的高度。（一定要注意下，第 i 列左（右）边最高的墙，是不包括自身的，和 leetcode 上边的讲的有些不同）

对于 `max_left` 我们其实可以这样求。

`max_left [i] = Max(max_left [i-1], height[i-1])` 。它前边的墙的左边的最高高度和它前边的墙的高度选一个较大的，就是当前列左边最高的墙了。

对于 max_right我们可以这样求。

`max_right[i] = Max(max_right[i+1], height[i+1])` 。它后边的墙的右边的最高高度和它后边的墙的高度选一个较大的，就是当前列右边最高的墙了。

这样，我们再利用解法二的算法，就不用在 `for` 循环里每次重新遍历一次求 `max_left` 和 `max_right` 了。

时间复杂度：$O(n)$。

空间复杂度：$O(n)$，用来保存每一列左边最高的墙和右边最高的墙。

方法 1：动态规划

```
class Solution {
    public int trap(int[] height) {
        int sum = 0;
        int[] max_left = new int[height.length];
        int[] max_right = new int[height.length];

        for (int i = 1; i < height.length - 1; i++) {
            max_left[i] = Math.max(max_left[i - 1], height[i - 1]);
        }
        for (int i = height.length - 2; i >= 0; i--) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            max_right[i] = Math.max(max_right[i + 1], height[i + 1]);
        }
        for (int i = 1; i < height.length - 1; i++) {
            int min = Math.min(max_left[i], max_right[i]);
            if (min > height[i]) {
                sum = sum + (min - height[i]);
            }
        }
        return sum;
    }
}
```

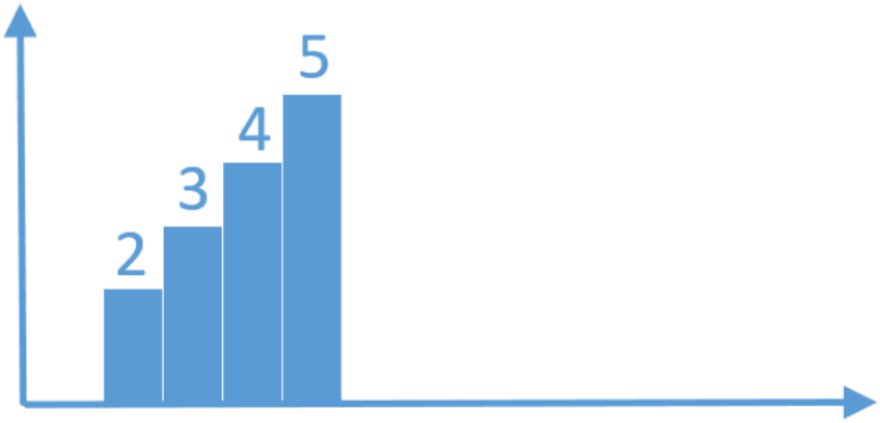
https://leetcode-cn.com/problems/trapping-rain-water/solution/shuang-zhi-zhen-qiu-jie-yu-shui-wen-ti-tu-wen-jie-/

2，双指针求解

这里我们还可以使用双指针，一个指向最左边，一个指向最右边，如下图所示。

这里要明白一点，最开始的时候如果左边柱子从左往右是递增的，那么这些柱子是不能盛水的，比如像下面这样

yes的练级攻略



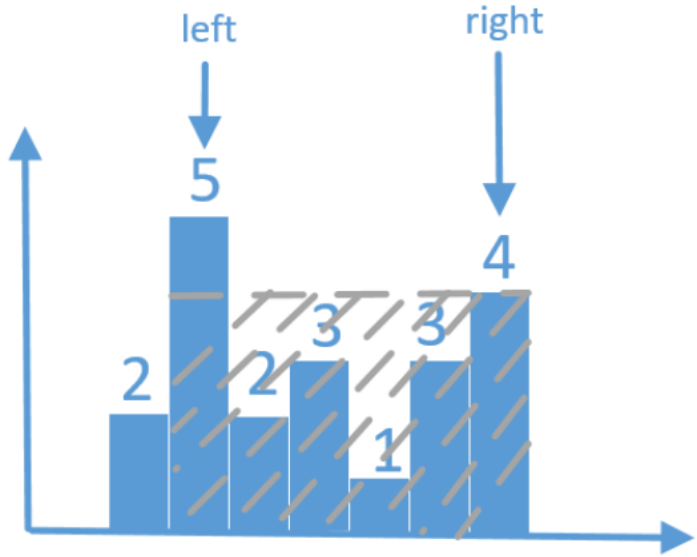
同理最开始的时候如果右边的柱子从右往左是递增的，也是不能盛水的。所以上面图中right指向的是右边第2根柱子。确定左右两边柱子的的代码如下

```
int left = 0, right = height.length - 1;
while (left < right && height[left] <= height[left + 1])
    left++;
while (left < right && height[right] <= height[right - 1])
    right--;
```

通过上面的计算，确定left和right的值之后，在left和right之间相当于构成了一个桶，桶的高度是最矮的那根柱子。然后我们从两边往中间逐个查找，如果查找的柱子高度小于桶的高度，那么盛水量就是桶的高度减去我们查找的柱子高度，如果查找的柱子大于桶的高度，我们要更新桶的高度。我们来看下最终代码

3，双指针代码简化

实际上我们还可以再进一步简化，我们看下下面这个图。此时left和right围成的桶的高度是4，这个时候如果right往左移，那么移动之后这个值是小于4的，也就是小于桶的高度，所以这个时候桶的高度是不变的。假如right往左移之后的值是大于4，比如5，那么桶的高度是要更新的。



我们只要确定桶的高度之后，那么盛水量就好求了。

方法 2：双指针

```
class Solution {
    public int trap(int[] height) {
```

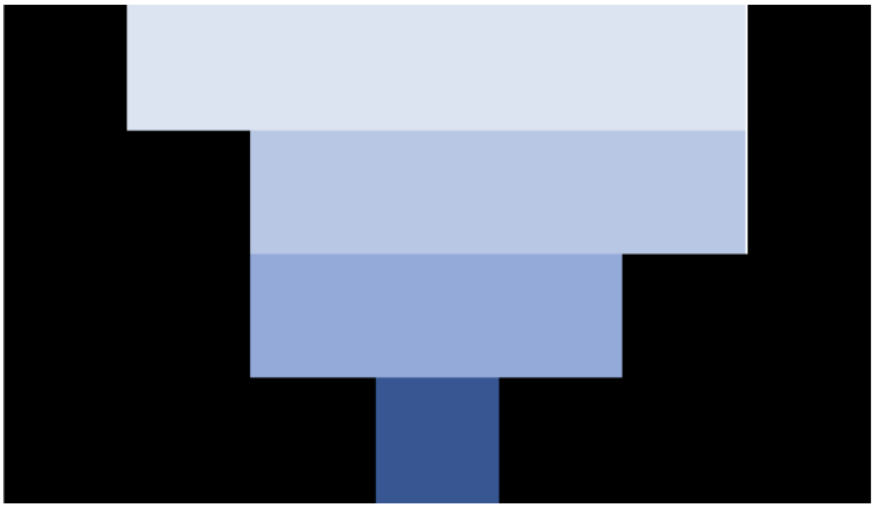
```
        int left = 0, right = height.length - 1, water = 0, bucketHeight = 0;
        while (left < right) {
            //取 height[left]和 height[right]的最小值
            int minHeight = Math.min(height[left], height[right]);
            //如果最小值 minHeight 大于桶的高度 bucketHeight，要更新桶的高度到
minHeight
            bucketHeight = bucketHeight < minHeight ? minHeight : bucketHeight;
            water += height[left] >= height[right] ? (bucketHeight - height[rig
ht--]) : (bucketHeight - height[left++]);
        }
        return water;
    }
}
```

禁止商用
公众号：
yes的练级攻略

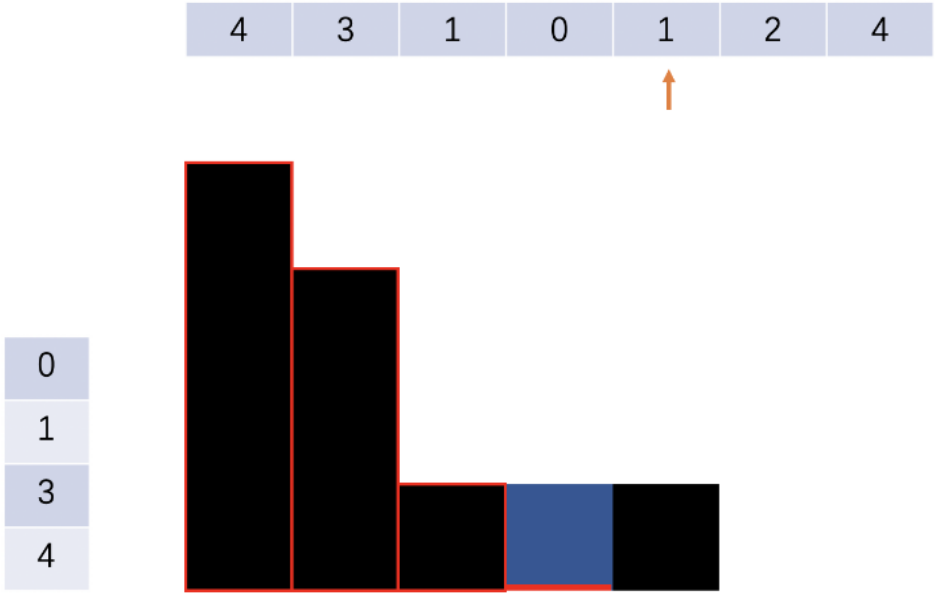
更多硬核内容就在公众号「yes 的练级攻略」

这道题目可以用单调栈来做。**单调栈就是比普通的栈多一个性质，即维护一个栈内元素单调。**
比如当前某个单调递减的栈的元素从栈底到栈顶分别是：`[10, 9, 8, 3, 2]`，如果要入栈元素`5`，需要把栈顶元素pop出去，直到满足单调递减为止，即先变成`[10, 9, 8]`，再入栈`5`，就是`[10, 9, 8, 5]`。

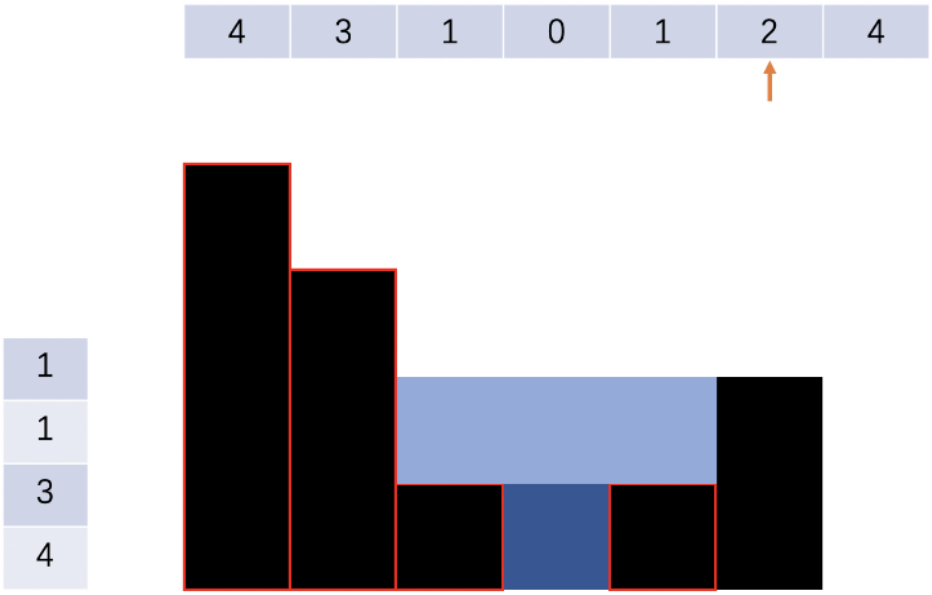
我们为这道题演示一下`[4, 3, 1, 0, 1, 2, 4]`是怎么接雨水的，下图是最终的接雨水效果，蓝色部分是雨水。



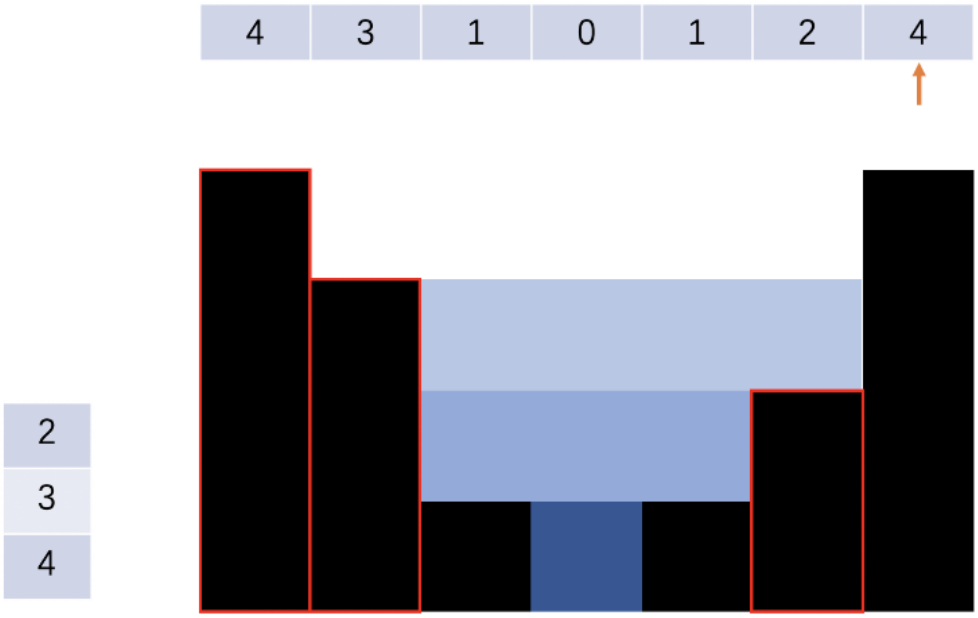
可以看下下面的图示。图示最上方是每个柱子的高度。左侧是单调栈的元素，`0`是栈顶。图中有红色边框的柱子是存在单调栈里的元素。
遍历到图示中箭头所指向的位置时，栈内元素是`[4, 3, 1, 0]`。由于当前的柱体的`1`大于栈顶元素`0`，那就可以接住雨水。接住雨水的量的高度是栈顶元素和左右两边形成的高度差的min。宽度是`1`。



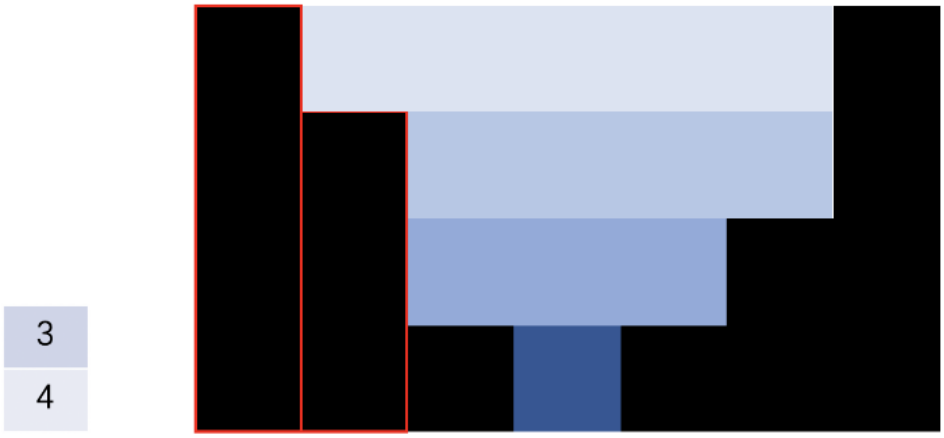
到下一个柱体高度为`2`时，栈内元素是`[4, 3, 1, 1]`。由于当前的柱体的`2`大于栈顶元素`1`，那就可以接住雨水。由于栈顶元素有相等的情况，所以可以把`1`全都pop出去，变成`[4, 3]`。此时最后一个pop出去的是`1`，高度是此时的栈顶元素和当前`2`柱体的高度的min再减去当前的`1`，宽度是`1`那个数字的位置和当前`2`柱体所在位置的差值。 可以算出来此次接住的雨水是1 * 3。



在下一个柱体高度为 `4` 时，栈内元素是 `[4, 3, 2]`，先把 `2` pop出来，栈顶元素 `3` 所在位置和当前的 `4` 可以接住雨水，雨水量是 1 * 4。



但是由于栈顶元素 `3` 仍然小于当前的 `4`，再pop出 `3`。栈顶元素 `4` 所在位置和当前的 `4` 可以接住雨水，雨水量是 1 * 5：



这样每个部分的雨水量都可以算出来，加在一起就可以了。由于每个柱体最多入栈出栈一次，所以时间复杂度是 $O(n)$

方法 3：单调栈

https://leetcode-cn.com/problems/trapping-rain-water/solution/dan-diao-zhan-jie-jue-jie-yu-shui-wen-ti-by-sweeti/

```
class Solution {
    public int trap(int[] height) {
        if (height.length == 0) return 0;
        Deque<Integer> stack = new ArrayDeque<>();
        int res = 0;
        for (int i = 0; i < height.length; i++){
            while (!stack.isEmpty() && height[stack.peek()] < height[i]
) {
                int cur = stack.pop();
                // 如果栈顶元素一直相等，那么全都pop出去，只留第一个。
                while (!stack.isEmpty() && height[stack.peek()] == heig
ht[cur]) {
                    stack.pop();
                }
                if (!stack.isEmpty()){
                    int top = stack.peek();
                    // top 此时指向的是此次接住的雨水的左边界的位置。右边界是
当前的柱体，即i。
                    // Math.min(height[top], height[i]) 是左右柱子高度的
min，减去height[curIdx]就是雨水的高度。
                    // i - top - 1 是雨水的宽度。
                    res += (Math.min(height[top], height[i]) - height[c
ur]) * (i - top - 1);
                }
            }
```

```
            stack.push(i);
        }
        return res;
    }
}
```

3、两数之和（1）【两数之和（lintcode 56）】

1. 两数之和

难度 简单 8390 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 `nums` 和一个目标值 `target`，请你在该数组中找出和为目标值的那 **两个** 整数，并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

示例:

```
给定 nums = [2, 7, 11, 15], target = 9

因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]
```

56. 两数之和

给一个整数数组，找到两个数使得他们的和等于一个给定的数 *target*。

你需要实现的函数 `twoSum` 需要返回这两个数的下标, 并且第一个下标小于第二个下标。注意这里下标的范围是 0 到 *n-1*。

样例
Example1:
给出 numbers = [2, 7, 11, 15], target = 9, 返回 [0, 1].
Example2:
给出 numbers = [15, 2, 7, 11], target = 9, 返回 [1, 2].
挑战
给自己加点挑战

- $O(n)$ 空间复杂度，$O(nlogn)$ 时间复杂度，
- $O(n)$ 空间复杂度，$O(n)$ 时间复杂度，

注意事项
你可以假设只有一组答案。

更多硬核内容就在公众号「yes 的练级攻略」

事实证明，我们可以一次完成。在进行迭代并将元素插入到表中的同时，我们还会回过头来检查表中是否已经存在当前元素所对应的目标元素。如果它存在，那我们已经找到了对应解，并立即将其返回。

Java

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }
        throw new IllegalArgumentException("No two sum solution");
    }
}
```

复杂度分析：

- 时间复杂度：$O(n)$，
  我们只遍历了包含有 n 个元素的列表一次。在表中进行的每次查找只花费 $O(1)$ 的时间。
- 空间复杂度：$O(n)$，
  所需的额外空间取决于哈希表中存储的元素数量，该表最多需要存储 n 个元素。

方法 1：hashmap

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++){
            if (map.containsKey(target-nums[i])) {
                return new int[] {map.get(target-nums[i]), i};
            }
            map.put(nums[i],i);
        }
        return null;
    }
}
```

https://www.jiuzhang.com/solution/two-sum/#tag-lang-java

使用双指针算法。 时间复杂度 $O(nlogn)$ 额外空间复杂度 $O(n)$

方法 2：双指针

```
public class Solution {
    /**
     * @param numbers: An array of Integer
     * @param target: target = numbers[index1] + numbers[index2]
     * @return: [index1 + 1, index2 + 1] (index1 < index2)
     */
    public int[] twoSum(int[] numbers, int target) {
        int[] result = {-1, -1};
```

```
        if (numbers == null) {
            return result;
        }

        Pair[] pairs = getSortedPairs(numbers);

        int left = 0, right = pairs.length - 1;
        while (left < right) {
            if (pairs[left].number + pairs[right].number < target) {
                left++;
            } else if (pairs[left].number + pairs[right].number > target) {
                right--;
            } else {
                result[0] = Math.min(pairs[left].index, pairs[right].index);
                result[1] = Math.max(pairs[left].index, pairs[right].index);
                return result;
            }
        }

        return result;
    }
    private Pair[] getSortedPairs(int[] numbers) {
        Pair[] pairs = new Pair[numbers.length];
        for (int i = 0; i < numbers.length; i++) {
            pairs[i] = new Pair(numbers[i], i);
        }
        Arrays.sort(pairs);

        return pairs;
    }
    class Pair implements Comparable<Pair> {
        int number, index;

        public Pair(int number, int index) {
            this.number = number;
            this.index = index;
        }

        public int compareTo(Pair other) {
            return number - other.number;
        }
    }
}
```

4、两数之和 2-输入有序数组（167）

禁止商用
公众号：
yes的练级攻略

167. 两数之和 II - 输入有序数组

难度 简单 348 收藏 分享 切换为英文 关注 反馈

给定一个已按照*升序排列*的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

- 返回的下标值（index1 和 index2）不是从零开始的。
- 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例:

```
输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。
```

方法 1：与上一题类似，用 HashMap（2ms）

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < numbers.length; i++){
            if (map.containsKey(target - numbers[i])) {
                return new int[] {map.get(target - numbers[i]) + 1, i +
 1};
            }
            map.put(numbers[i], i);
        }
        return null;
    }
}
```

方法 2：双指针（1ms）

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int left = 0, right = numbers.length - 1;
        while (left < right) {
            if (numbers[left] + numbers[right] == target) {
                return new int[] {left + 1, right + 1};
            } else {
                if (numbers[left] + numbers[right] < target){
                    left++;
                } else {
                    right--;
                }
            }
        }
```

```
            return null;
        }
    }
```

5、和为 s 的两个数字（剑指 57）【和上一题只有返回值的区别】

剑指 Offer 57. 和为s的两个数字

难度 简单 33 收藏 分享 切换为英文 关注 反馈

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们的和正好是s。如果有多对数字的和等于s，则输出任意一对即可。

示例 1:

```
输入: nums = [2,7,11,15], target = 9
输出: [2,7] 或者 [7,2]
```

示例 2:

```
输入: nums = [10,26,30,31,47,60], target = 40
输出: [10,30] 或者 [30,10]
```

限制:

- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^6

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            if (nums[left] + nums[right] == target) {
                return new int[] {nums[left], nums[right]};
            } else {
                if (nums[left] + nums[right] < target){
                    left++;
                } else {
                    right--;
                }
            }
        }
        return null;
    }
}
```

6、两数之和 2（lintcode 443）【多少对整数的和大于目标值】

443. 两数之和 II

给一组整数，问能找出多少对整数，他们的和大于一个给定的目标值。

样例
样例 1:

输入: [2, 7, 11, 15], target = 24
输出: 1
解释: 11 + 15 是唯一的一对
样例 2:

输入: [1, 1, 1, 1], target = 1
输出: 6
挑战
O(1) 额外空间以及 O(nlogn) 时间复杂度

https://www.jiuzhang.com/solution/two-sum-greater-than-target/#tag-lang-java

总体思路是排序后使用两根指针进行遍历, 而遍历的过程有多种实现方式.

本题解采用的思路是:

- 定义左右端点指针 `l = 0`, `r = n - 1` (n是数组长度)
- 若 `l` 指向元素与 `r` 指向元素的和不大于 `target`, 则 `l` 右移; 反之, `result += r - l`, 并 `r` 左移
- 终止条件为 `l >= r`

上述思路利用了单调性: 若 `nums[l] + nums[r] > target`, 则 `nums[l + k] + nums[r] > target (k >= 0)`

`result += r - l` 代表 `nums[r]` 可以与 `nums[l], nums[l + 1], ... , nums[r - 1]` 这 `r - l` 个元素成对, 加和大于 `target`.

一个小优化: `l` 左移时可以用二分/倍增加速.

```
public class Solution {
    /**
     * @param nums: an array of integer
     * @param target: An integer
     * @return: an integer
     */
    public int twoSum2(int[] nums, int target) {
        if (nums == null || nums.length < 2) {
            return 0;
        }

        Arrays.sort(nums);

        int left = 0, right = nums.length - 1;
        int count = 0;
        while (left < right) {
            if (nums[left] + nums[right] <= target) {
                left++;
            } else {
```

yes的练级攻略

```
                count += right - left;
                right--;
            }
        }
        return count;
    }
}
```

7、两数之和 3-数据结构设计（lintcode 607）【add 和 find】【哈希表】

607. 两数之和 III-数据结构设计

设计b并实现一个 TwoSum 类。他需要支持以下操作: `add` 和 `find`。
`add` -把这个数添加到内部的数据结构。
`find` -是否存在任意一对数字之和等于这个值

样例
样例 1:

```
add(1);add(3);add(5);
find(4)//返回true
find(7)//返回false
```

https://www.jiuzhang.com/solution/two-sum-iii-data-structure-design/#tag-lang-java

用HashMap来存储数据（key是num，value是num的个数）。add的时间复杂度是O(1), find的时间复杂的是O(n)，空间复杂度O(n)。 需要注意的是当在map里找到value - num时，要判断下这个值是否与num相等。如果不相等可以直接返回true，如果相等要确认num在map里不止一个

```
public class TwoSum {
    HashMap<Integer, Integer> map;

    public TwoSum() {
        map = new HashMap<>();
    }

    public void add(int number) {
        // write your code here
        // key is number and value is value's count
        map.put(number, map.getOrDefault(number, 0) + 1);
    }

    public boolean find(int value) {
        int diff;
        for (int num : map.keySet()) {
            diff = value - num;
            if (map.containsKey(diff)) {
                if (diff != num || map.get(diff) > 1)
                    return true;
            }
```

```
            }
            return false;
        }
}
```

8、两数之和-不同组成（lintcode 587）

587. 两数之和 - 不同组成

给一整数数组, 找到数组中有多少组 不同的元素对 有相同的和, 且和为给出的 target 值, 返回对数.

样例
例1:

输入: nums = [1,1,2,45,46,46], target = 47
输出: 2
解释:

1 + 46 = 47
2 + 45 = 47

例2:

输入: nums = [1,1], target = 2
输出: 1
解释:
1 + 1 = 2

https://www.jiuzhang.com/solution/two-sum-unique-pairs/#tag-lang-java

```
public class Solution {
    /**
     * @param nums: an array of integer
     * @param target: An integer
     * @return: An integer
     */
    public int twoSum6(int[] nums, int target) {
        if (nums == null || nums.length < 2)
            return 0;

        Arrays.sort(nums);
        int cnt = 0;
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int v = nums[left] + nums[right];
            if (v == target) {
                cnt ++;
                left ++;
                right --;
                while (left < right && nums[right] == nums[right + 1])
                    right --;
```

公众号：yes的练级攻略

```
                while (left < right && nums[left] == nums[left - 1])
                    left ++;
            } else if (v > target) {
                right --;
            } else {
                left ++;
            }
        }
        return cnt;
    }
}
```

9、两数之和 7（lintcode 1879）【按绝对值升序排列】【二分查找】

1879. 两数之和 VII

给定一个已经 按绝对值升序排列 的数组，找到两个数使他们加起来的和等于特定数。

函数应该返回这两个数的下标，index1必须小于index2。注意返回的值是0-based。

你不能对该数组进行排序。

样例
输入：
[0,-1,2,-3,4]
1
输出：[[1,2],[3,4]]
说明：nums[1] + nums[2] = -1 + 2 = 1, nums[3] + nums[4] = -3 + 4 = 1
你也可以返回 [[3,4],[1,2]]，系统将自动帮你排序成 [[1,2],[3,4]]。但是[[2,1],[3,4]]是不合法的。
挑战
O(*n*)的时间复杂度和O(1)的额外空间复杂度

注意事项
- 数据保证*nums*中的所有数的互不相同的。
- *nums*数组长度≤ 100 000
- *nums*内的数≤ 10^9

https://www.jiuzhang.com/solution/two-sum-vii/#tag-lang-java

遍历nums，二分寻找数组里是否有绝对值等于target - nums的元素， 时间复杂度O（nlogn）

最后考虑找到目标，但正负同时在数组的情况，因为这个程序大于等于都让end = mid，所以只需要最后判断end + 1是不是目标数字即可，无需判断start-1

```
public class Solution {
    /**
     * @param nums: the input array
     * @param target: the target number
     * @return: return the target pair
     */
    public List<List<Integer>> twoSumVII(int[] nums, int target) {
        List<List<Integer>> res = new ArrayList<>();
        for(int i=0; i<nums.length; i++) {
            int index = checkTarget(i, nums, target - nums[i]);

            if(index != -1) {
```

```
                List<Integer> list = new ArrayList<>();
                list.add(i);
                list.add(index);
                res.add(list);
            }
        }

        return res;
    }
    private int checkTarget(int current, int[] nums, int target) {
        int start = current + 1, end = nums.length - 1;
        if(start >= nums.length)    return -1;
        while(start + 1 < end) {
            int mid = (start + end) / 2;

            if(Math.abs(nums[mid]) >= Math.abs(target)) {
                end = mid;
            } else {
                start = mid;
            }
        }
        if(nums[start] == target) {
            return start;

        } else if(nums[end] == target) {
            return end;

        } else if(end < nums.length-1 && target == nums[end+1]) {
            return end + 1;

        }
        return -1;
    }
}
```

10、两数之和-BST 版本（lintcode 689）

禁止商用
公众号：
yes的练级攻略

689. 两数之和 - BST版本

给一棵二叉搜索树以及一个整数 `n`, 在树中找到和为 `n` 的两个数字

样例

样例1

输入:
{4,2,5,1,3}
3
输出: [1,2] (or [2,1])
解释:
二叉搜索树如下:

```
    4
   / \
  2   5
 / \
1   3
```

样例2

输入:
{4,2,5,1,3}
5
输出: [2,3] (or [3,2])
注意事项
没有任何额外的空间。

https://www.jiuzhang.com/solution/two-sum-iv-input-is-a-bst/#tag-lang-java

h为树的最大深度，n为树的结点个数

方法一：基于 predecessor 与 successor + 双指针 Time O(h * n) Space O(1) 448. Inorder Successor in BST 915. Inorder Predecessor in BST

方法二：基于 BST_Iterator + 双指针 Time O(h + n) Space O(h) 86. Binary Search Tree Iterator

方法三：暴力中序遍历 + 双指针 Time O(n) Space O(n)

【注】：本题是双指针从两头向中间移动，left=> <=right 相似题目901. Closest Binary Search Tree Value II可以用双指针从中间向两头移动的解法 <=left right=>

方法 1：BST_Iterator+双指针

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

class BST_Iterator {
```

```
    Stack<TreeNode> stack;
    TreeNode p;
    boolean forward;
    BST_Iterator(TreeNode root, boolean forward) {
        this.stack = new Stack<>();
        this.p = root;
        this.forward = forward;
    }

    public boolean hasNext() {
        return !stack.isEmpty() || p != null;
    }

    public TreeNode next() {
        if (!hasNext()) {
            return null;
        }

        while (p != null) {
            stack.push(p);
            p = forward ? p.left : p.right;
        }

        TreeNode ret = stack.pop();
        p = forward ? ret.right : ret.left;

        return ret;
    }
}

public class Solution {
    /*
     * @param : the root of tree
     * @param : the target sum
     * @return: two numbers from tree which sum is n
     */
    public int[] twoSum(TreeNode root, int n) {
        // write your code here
        if (root == null) {
            return null;
        }

        BST_Iterator forward = new BST_Iterator(root, true);
        BST_Iterator backward = new BST_Iterator(root, false);
```

禁止商用 公众号：yes的练级攻略

```
            TreeNode left = forward.next();
            TreeNode right = backward.next();

            while (left != right) {
                if (left.val + right.val == n) {
                    return new int[] {left.val, right.val};
                } else if (left.val + right.val < n) {
                    left = forward.next();
                } else {
                    right = backward.next();
                }
            }

            return null;
        }
}
```

方法 2：暴力中序+双指针【最短】

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /*
     * @param : the root of tree
     * @param : the target sum
     * @return: two numbers from tree which sum is n
     */
    public int[] twoSum(TreeNode root, int n) {
        // write your code here
        if (root == null) {
            return null;
        }

        List<Integer> inorder = new ArrayList<>();
        inorderTraverse(root, inorder);
```

禁止商用
公众号：
yes的练级攻略

```
            int left = 0;
            int right = inorder.size() - 1;
            while (left != right) {
                int leftVal = inorder.get(left);
                int rightVal = inorder.get(right);
                if (leftVal + rightVal == n) {
                    return new int[] {leftVal, rightVal};
                } else if (leftVal + rightVal < n) {
                    left++;
                } else {
                    right--;
                }
            }

            return null;
        }

        private void inorderTraverse(TreeNode root, List<Integer> inorder) {
            if (root == null) {
                return;
            }

            inorderTraverse(root.left, inorder);
            inorder.add(root.val);
            inorderTraverse(root.right, inorder);
        }
}
```

11、两数之和 4-输入 BST（653）

653. 两数之和 IV - 输入 BST

难度 简单 169 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树和一个目标结果，如果 BST 中存在两个元素且它们的和等于给定的目标结果，则返回 true。

案例 1:

输入：

```
    5
   / \
  3   6
 / \   \
2   4   7

Target = 9
```

输出： True

禁止商用 公众号：yes的练级攻略

案例 2:

```
输入：
    5
   / \
  3   6
 / \   \
2   4   7

Target = 28

输出：False
```

https://leetcode-cn.com/problems/two-sum-iv-input-is-a-bst/solution/liang-shu-zhi-he-iv-by-leetcode/

方法三：使用 BST【通过】

算法

在本方法中利用 BST 的性质，BST 的中序遍历结果是按升序排列的。因此，中序遍历给定的 BST，并将遍历结果存储到 $list$ 中。

遍历完成后，使用两个指针 l 和 r 作为 $list$ 的头部索引和尾部索引。然后执行以下操作：

1. 检查 l 和 r 索引处两元素之和是否等于 k。如果是，立即返回 True。
2. 如果当前两元素之和小于 k，则更新 l 指向下一个元素。这是因为当我们需要增大两数之和时，应该增大较小数。
3. 如果当前两元素之和大于 k，则更新 r 指向上一个元素。这是因为当我们需要减小两数之和时，应该减小较大数。
4. 重复步骤一至三，直到左指针 l 大于右指针 r。
5. 如果左指针 l 到右指针 r 的右边，则返回 False。

注意，在任何情况下，都不应该增大较大的数，也不应该减小较小的数。这是因为如果当前两数之和大于 k，不应该首先增大 $list[r]$ 的值。类似的，也不应该首先减小 $list[l]$ 的值。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是树中节点的数量。本方法需要中序遍历整棵树。
- 空间复杂度：$O(n)$，$list$ 中存储 n 个元素。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
```

```
    public boolean findTarget(TreeNode root, int k) {
        if (root == null) return false;
        List<Integer> in = new ArrayList<>();
        dfs(root, in);
        int left = 0, right = in.size() - 1;
        while (left != right) {
            int leftVal = in.get(left), rightVal = in.get(right);
            if (leftVal + rightVal == k) return true;
            else if (leftVal + rightVal < k) left++;
            else right--;
        }
        return false;
    }
    private void dfs(TreeNode root, List<Integer> in){
        if (root == null) return;
        dfs(root.left, in);
        in.add(root.val);
        dfs(root.right, in);
    }
}
```

12、两数和-差等于目标值（lintcode 610）

610. 两数和 - 差等于目标值

给定一个排序后的整数数组，找到两个数的 差 等于目标值。
你需要返回一个包含两个数字的列表 [num1, num2]，使得 num1与 num2 的差为target，同时 num1 必须小于 num2。

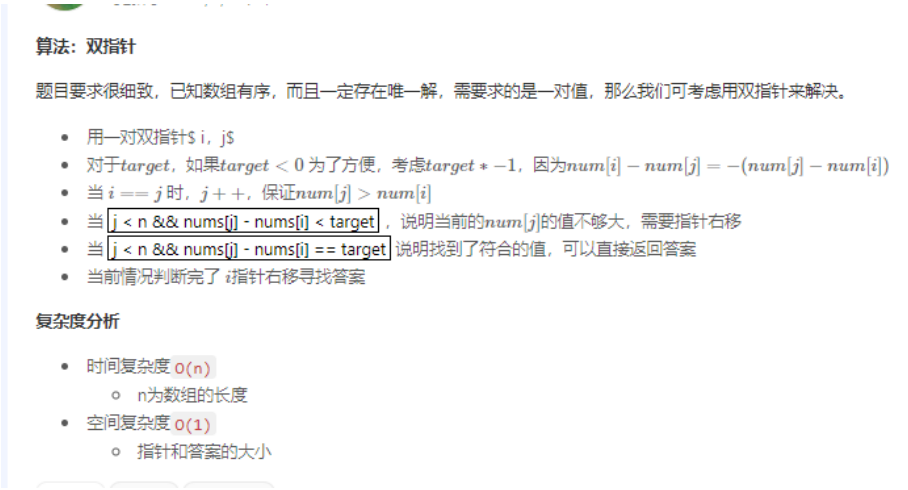
样例
例1:

```
输入: nums = [2, 7, 15, 24], target = 5
输出: [2, 7]
解释:
(7 - 2 = 5)
```

例2:

```
输入: nums = [1, 1], target = 0
输出: [1, 1]
解释:
(1 - 1 = 0)
```

注意事项
保证只有一个答案。
注意：禁止使用额外空间。



算法：双指针

题目要求很细致，已知数组有序，而且一定存在唯一解，需要求的是一对值，那么我们可考虑用双指针来解决。

- 用一对双指针$ i，j$
- 对于$target$，如果$target < 0$为了方便，考虑$target * -1$，因为$num[i] - num[j] = -(num[j] - num[i])$
- 当$i == j$时，$j++$，保证$num[j] > num[i]$
- 当`j < n && nums[j] - nums[i] < target`，说明当前的$num[j]$的值不够大，需要指针右移
- 当`j < n && nums[j] - nums[i] == target`说明找到了符合的值，可以直接返回答案
- 当前情况判断完了 i指针右移寻找答案

复杂度分析

- 时间复杂度 `O(n)`
  - n为数组的长度
- 空间复杂度 `O(1)`
  - 指针和答案的大小

https://www.jiuzhang.com/solution/two-sum-difference-equals-to-target/#tag-lang-java

```
public class Solution {
    /**
     * @param nums: an array of Integer
     * @param target: an integer
     * @return: [num1, num2] (num1 < num2)
     */
    public int[] twoSum7(int[] nums, int target) {
        if (nums == null || nums.length < 2) return new int[]{-1, -1};
        target = Math.abs(target);
        int j = 1;
        for (int i = 0; i < nums.length; i++){
            j = Math.max(j, i+1);
            while (j < nums.length && nums[j] - nums[i] < target) {
                j++;
            }
            if (j >= nums.length) break;
            if (nums[j] - nums[i] == target){
                return new int[] {nums[i], nums[j]};
            }
        }
        return new int[]{-1, -1};
    }
}
```

13、两数和的最接近值（lintcode 533）

禁止商用 公众号：yes的练级攻略

533. 两数和的最接近值

给定整数数组 num，从中找到两个数字使得他们和最接近 target，返回两数和与 target 的差的 绝对值。

样例

样例1

```
输入: nums = [-1, 2, 1, -4] 并且 target = 4
输出: 1
解释:
最小的差距是1，(4 - (2 + 1) = 1).
```

样例2

```
输入: nums = [-1, -1, -1, -4] 并且 target = 4
输出: 6
解释:
最小的差距是6，(4 - (- 1 - 1) = 6).
```

挑战

Do it in O(nlogn) time complexity.

```
public class Solution {
    /**
     * @param nums: an integer array
     * @param target: An integer
     * @return: the difference between the sum and the target
     */
    public int twoSumClosest(int[] nums, int target) {
        if (nums == null || nums.length == 0) {
            return -1;
        }

        Arrays.sort(nums);
        int left = 0;
        int right = nums.length - 1;
        int diff = Integer.MAX_VALUE;

        while (left < right) {
            if (nums[left] + nums[right] == target) {
                return 0;
            } else if (nums[left] + nums[right] < target) {
                diff = Math.min(diff, Math.abs(target - nums[left] - nums[right]));
                left++;
            } else {
                diff = Math.min(diff, Math.abs(target - nums[left] - nums[right]));
                right--;
            }
        }

        return diff;
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
    }
}
```

14、三数之和（15）

15. 三数之和

难度 中等 2423 收藏 分享 切换为英文 关注 反馈

给你一个包含 *n* 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 *a*，*b*，*c*，使得 *a* + *b* + *c* = 0？请你找出所有满足条件且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

```
给定数组 nums = [-1, 0, 1, 2, -1, -4]，

满足要求的三元组集合为：
[
  [-1, 0, 1],
  [-1, -1, 2]
]
```

https://leetcode-cn.com/problems/3sum/solution/san-shu-zhi-he-by-leetcode-solution/

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法一：排序 + 双指针

题目中要求找到所有「不重复」且和为 0 的三元组，这个「不重复」的要求使得我们无法简单地使用三重循环枚举所有的三元组。这是因为在最坏的情况下，数组中的元素全部为 0，即

```
[0, 0, 0, 0, 0, ..., 0, 0, 0]
```

任意一个三元组的和都为 0。如果我们直接使用三重循环枚举三元组，会得到 $O(N^3)$ 个满足题目要求的三元组（其中 N 是数组的长度）时间复杂度至少为 $O(N^3)$。在这之后，我们还需要使用哈希表进行去重操作，得到不包含重复三元组的最终答案，又消耗了大量的空间。这个做法的时间复杂度和空间复杂度都很高，因此我们要换一种思路来考虑这个问题。

「不重复」的本质是什么？我们保持三重循环的大框架不变，只需要保证：

- 第二重循环枚举到的元素不小于当前第一重循环枚举到的元素；
- 第三重循环枚举到的元素不小于当前第二重循环枚举到的元素。

也就是说，我们枚举的三元组 (a, b, c) 满足 $a \leq b \leq c$，保证了只有 (a, b, c) 这个顺序会被枚举到，而 (b, a, c)、(c, b, a) 等等这些不会，这样就减少了重复。要实现这一点，我们可以将数组中的元素从小到大进行排序，随后使用普通的三重循环就可以满足上面的要求。

同时，对于每一重循环而言，相邻两次枚举的元素不能相同，否则也会造成重复。举个例子，如果排完序的数组为

```
[0, 1, 2, 2, 2, 3]
 ^  ^  ^
```

我们使用三重循环枚举到的第一个三元组为 $(0, 1, 2)$，如果第三重循环继续枚举下一个元素，那么仍然是三元组 $(0, 1, 2)$，产生了重复。因此我们需要将第三重循环「跳到」下一个不相同的元素，即数组中的最后一个元素 3，枚举三元组 $(0, 1, 3)$。

下面给出了改进的方法的伪代码实现：

```
nums.sort()
for first = 0 .. n-1
    // 只有和上一次枚举的元素不相同，我们才会进行枚举
    if first == 0 or nums[first] != nums[first-1] then
        for second = first+1 .. n-1
            if second == first+1 or nums[second] != nums[second-1] then
                for third = second+1 .. n-1
                    if third == second+1 or nums[third] != nums[third-1] then
                        // 判断是否有 a+b+c==0
                        check(first, second, third)
```

这种方法的时间复杂度仍然为 $O(N^3)$，毕竟我们还是没有跳出三重循环的大框架。然而它是很容易继续优化的，可以发现，如果我们固定了前两重循环枚举到的元素 a 和 b，那么只有唯一的 c 满足 $a + b + c = 0$。当第二重循环往后枚举一个元素 b' 时，由于 $b' > b$，那么满足 $a + b' + c' = 0$ 的 c' 一定有 $c' < c$，即 c' 在数组中一定出现在 c 的左侧。也就是说，我们可以从小到大枚举 b，**同时**从大到小枚举 c，即**第二重循环和第三重循环实际上是并列的关系。**

有了这样的发现，我们就可以保持第二重循环不变，而将**第三重循环变成一个从数组最右端开始向左移动的指针**，从而得到下面的伪代码：

```
nums.sort()
for first = 0 .. n-1
    if first == 0 or nums[first] != nums[first-1] then
        // 第三重循环对应的指针
        third = n-1
        for second = first+1 .. n-1
            if second == first+1 or nums[second] != nums[second-1] then
                // 向左移动指针，直到 a+b+c 不大于 0
                while nums[first]+nums[second]+nums[third] > 0
                    third = third-1
                // 判断是否有 a+b+c==0
                check(first, second, third)
```

这个方法就是我们常说的「双指针」，当我们需要枚举数组中的两个元素时，如果我们发现随着第一个元素的递增，第二个元素是递减的，那么就可以使用双指针的方法，将枚举的时间复杂度从 $O(N^2)$ 减少至 $O(N)$。为什么是 $O(N)$ 呢？这是因为在枚举的过程每一步中，「左指针」会向右移动一个位置（也就是题目中的 b），而「右指针」会向左移动若干个位置，这个与数组的元素有关，但我们知道它一共会移动的位置数为 $O(N)$，**均摊下来**，每次也向左移动一个位置，因此时间复杂度为 $O(N)$。

注意到我们的伪代码中还有第一重循环，时间复杂度为 $O(N)$，因此枚举的总时间复杂度为 $O(N^2)$。由于排序的时间复杂度为 $O(N \log N)$，在渐进意义下小于前者，因此算法的总时间复杂度为 $O(N^2)$。

上述的伪代码中还有一些细节需要补充，例如我们需要保持左指针一直在右指针的左侧（即满足 $b \leq c$），具体可以参考下面的代码，均给出了详细的注释。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 是数组 $nums$ 的长度。
- 空间复杂度：$O(\log N)$。我们忽略存储答案的空间，额外的排序的空间复杂度为 $O(\log N)$。然而我们修改了输入的数组 $nums$，在实际情况下不一定允许，因此也可以看成使用了一个额外的数组存储了 $nums$ 的副本并进行排序，空间复杂度为 $O(N)$。

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int n = nums.length;
        Arrays.sort(nums);
        List<List<Integer>> ans = new ArrayList<List<Integer>>();
        // 枚举 a
        for (int first = 0; first < n; ++first) {
            // 需要和上一次枚举的数不相同
            if (first > 0 && nums[first] == nums[first - 1]) {
                continue;
            }
            // c 对应的指针初始指向数组的最右端
            int third = n - 1;
            int target = -nums[first];
            // 枚举 b
            for (int second = first + 1; second < n; ++second) {
                // 需要和上一次枚举的数不相同
                if (second > first + 1 && nums[second] == nums[second - 1]) {
                    continue;
                }
                // 需要保证 b 的指针在 c 的指针的左侧
                while (second < third && nums[second] + nums[third] > target) {
                    --third;
                }
                // 如果指针重合，随着 b 后续的增加
                // 就不会有满足 a+b+c=0 并且 b<c 的 c 了，可以退出循环
                if (second == third) {
                    break;
                }
                if (nums[second] + nums[third] == target) {
```

```
                    List<Integer> list = new ArrayList<Integer>();
                    list.add(nums[first]);
                    list.add(nums[second]);
                    list.add(nums[third]);
                    ans.add(list);
                }
            }
        }
        return ans;
    }
}
```

Ellipse 中测试的主函数

```
    public static void main(String[] args) {
        //int[] nums = {-1, 0, 1, 2, -1, -4};
        Scanner sc = new Scanner(System.in);
        int length = 6;
        int[] nums = new int[length];
        for (int i = 0; i < length; i++) {
            nums[i] = sc.nextInt();
        }
        TwoPointer tp = new TwoPointer();
        List<List<Integer>> res = new LinkedList<>();
        res = tp.threeSum(nums);
        System.out.println(res);

    }
```

15、最接近的三数之和（16）

16. 最接近的三数之和

难度 中等　523　收藏　分享　切换为英文　关注　反馈

给定一个包括 *n* 个整数的数组 `nums` 和 一个目标值 `target`。找出 `nums` 中的三个整数，使得它们的和与 `target` 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

示例：

```
输入：nums = [-1,2,1,-4], target = 1
输出：2
解释：与 target 最接近的和是 2 (-1 + 2 + 1 = 2) 。
```

提示：

- `3 <= nums.length <= 10^3`
- `-10^3 <= nums[i] <= 10^3`
- `-10^4 <= target <= 10^4`

https://leetcode-cn.com/problems/3sum-closest/solution/hua-jie-suan-fa-16-zui-jie-jin-de-san-shu-zhi-he-b/

禁止商用 公众号：

解题方案

思路

- 标签：排序和双指针
- 本题目因为要计算三个数，如果靠暴力枚举的话时间复杂度会到 $O(n^3)$，需要降低时间复杂度
- 首先进行数组排序，时间复杂度 $O(nlogn)$
- 在数组 nums 中，进行遍历，每遍历一个值利用其下标，形成一个固定值 nums[i]
- 再使用前指针指向 `start = i + 1` 处，后指针指向 `end = nums.length - 1` 处，也就是结尾处
- 根据 `sum = nums[i] + nums[start] + nums[end]` 的结果，判断 sum 与目标 target 的距离，如果更近则更新结果 ans
- 同时判断 sum 与 target 的大小关系，因为数组有序，如果 `sum > target` 则 `end--`，如果 `sum < target` 则 `start++`，如果 `sum == target` 则说明距离为 0 直接返回结果
- 整个遍历过程，固定值为 n 次，双指针为 n 次，时间复杂度为 $O(n^2)$
- 总时间复杂度：$O(nlogn) + O(n^2) = O(n^2)$

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int ans = nums[0] + nums[1] + nums[2];
        for(int i=0;i<nums.length;i++) {
            int start = i+1, end = nums.length - 1;
            while(start < end) {
                int sum = nums[start] + nums[end] + nums[i];
                if(Math.abs(target - sum) < Math.abs(target - ans))
                    ans = sum;
                if(sum > target)
                    end--;
                else if(sum < target)
                    start++;
                else
                    return ans;
            }
        }
        return ans;
    }
}
```

16、四数之和（18）

18. 四数之和

难度 中等 480 收藏 分享 切换为英文 关注 反馈

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

示例：

```
给定数组 nums = [1, 0, -1, 0, -2, 2]，和 target = 0。

满足要求的四元组集合为：
[
  [-1,  0, 0, 1],
  [-2, -1, 1, 2],
  [-2,  0, 0, 2]
]
```

https://leetcode-cn.com/problems/4sum/solution/ji-bai-9994de-yong-hu-you-dai-ma-you-zhu-shi-by-yo/

```
class Solution {
public List<List<Integer>> fourSum(int[] nums,int target){
        /*定义一个返回值*/
        List<List<Integer>> result=new ArrayList<>();
        /*当数组为null 或元素小于4 个时，直接返回*/
        if(nums==null||nums.length<4){
            return result;
        }
        /*对数组进行从小到大排序*/
        Arrays.sort(nums);
        /*数组长度*/
        int length=nums.length;
        /*定义4 个指针 k，i，j，h  k 从0 开始遍历，i 从k+1 开始遍历，留下j 和h，j 指向i+1，h 指向数组最大值*/
        for(int k=0;k<length-3;k++){
            /*当k 的值与前面的值相等时忽略*/
            if(k>0&&nums[k]==nums[k-1]){
                continue;
            }
            /*获取当前最小值，如果最小值比目标值大，说明后面越来越大的值根本没戏*/
            int min1=nums[k]+nums[k+1]+nums[k+2]+nums[k+3];
            if(min1>target){
                break;
            }
            /*获取当前最大值，如果最大值比目标值小，说明后面越来越小的值根本没戏，忽略*/
            int max1=nums[k]+nums[length-1]+nums[length-2]+nums[length-3];
            if(max1<target){
                continue;
```

```
            }
            /*第二层循环 i，初始值指向 k+1*/
            for(int i=k+1;i<length-2;i++){
                /*当 i 的值与前面的值相等时忽略*/
                if(i>k+1&&nums[i]==nums[i-1]){
                    continue;
                }
                /*定义指针 j 指向 i+1*/
                int j=i+1;
                /*定义指针 h 指向数组末尾*/
                int h=length-1;
                /*获取当前最小值，如果最小值比目标值大，说明后面越来越大的值根本没戏*/
                int min=nums[k]+nums[i]+nums[j]+nums[j+1];
                if(min>target){
                    break;
                }
                /*获取当前最大值，如果最大值比目标值小，说明后面越来越小的值根本没戏，
忽略*/
                int max=nums[k]+nums[i]+nums[h]+nums[h-1];
                if(max<target){
                    continue;
                }
                /*开始 j 指针和 h 指针的表演，计算当前和，如果等于目标值，j++并去重，h--
并去重，当当前和大于目标值时 h--，当当前和小于目标值时 j++*/
                while (j<h){
                    int curr=nums[k]+nums[i]+nums[j]+nums[h];
                    if(curr==target){
                        result.add(Arrays.asList(nums[k],nums[i],nums[j],nums[h
]));
                        j++;
                        while(j<h&&nums[j]==nums[j-1]){
                            j++;
                        }
                        h--;
                        while(j<h&&i<h&&nums[h]==nums[h+1]){
                            h--;
                        }
                    }else if(curr>target){
                        h--;
                    }else {
                       j++;
                    }
                }
            }
```

```
        }
        return result;
    }
}
```

17、四数相加 2（454）【哈希表】

454. 四数相加 II

难度 中等 193 收藏 分享 切换为英文 关注 反馈

给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 `(i, j, k, l)` ，使得 `A[i] + B[j] + C[k] + D[l] = 0`。

为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 0 ≤ N ≤ 500 。所有整数的范围在 -2^{28} 到 $2^{28} - 1$ 之间，最终结果不会超过 $2^{31} - 1$ 。

例如:

输入:

```
A = [ 1, 2]
B = [-2,-1]
C = [-1, 2]
D = [ 0, 2]
```

输出:

```
2
```

解释:

两个元组如下:

```
1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0
```

https://leetcode-cn.com/problems/4sum-ii/solution/chao-ji-rong-yi-li-jie-de-fang-fa-si-shu-xiang-jia/

解题思路

思路:

一.采用分为两组，HashMap存一组，另一组和HashMap进行比对。

二.这样的话情况就可以分为三种:

1.HashMap存一个数组，如A。然后计算三个数组之和，如BCD。时间复杂度为：O(n)+O(n^3),得到O(n^3).

2.HashMap存三个数组之和，如ABC。然后计算一个数组，如D。时间复杂度为：O(n^3)+O(n),得到O(n^3).

3.HashMap存两个数组之和，如AB。然后计算两个数组之和，如CD。时间复杂度为：O(n^2)+O(n^2),得到O(n^2).

三.根据第二点我们可以得出要存两个数组算两个数组。

四.我们以存AB两数组之和为例。首先求出A和B任意两数之和sumAB，以sumAB为key，sumAB出现的次数为value，存入hashmap中。

然后计算C和D中任意两数之和的相反数sumCD，在hashmap中查找是否存在key为sumCD。

算法时间复杂度为O(n2)。

```
class Solution {
    public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
        Map<Integer, Integer> map = new HashMap<>();
        int res = 0;
        for(int i = 0;i<A.length;i++){
            for(int j= 0;j<B.length;j++){
                int sumAB = A[i]+B[j];
                if(map.containsKey(sumAB)) map.put(sumAB,map.get(sumAB)+1);
                else map.put(sumAB,1);
            }
```

```
        }
        for(int i = 0;i<C.length;i++){
            for(int j = 0;j<D.length;j++){
                int sumCD = -(C[i]+D[j]);
                if(map.containsKey(sumCD)) res += map.get(sumCD);
            }
        }
        return res;
    }
}
```

18、下一个排序（31）

31. 下一个排列

难度 中等 525 收藏 分享 切换为英文 关注 反馈

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。

`1,2,3` → `1,3,2`

`3,2,1` → `1,2,3`

`1,1,5` → `1,5,1`

https://leetcode-cn.com/problems/next-permutation/solution/shuang-zhi-zhen-by-acnesu/

解题思路

需要找到数组中的两个数，前面的数比后面的数小，用两个指针去寻找，i指针在前，j指针在后，由于要寻找最小的增量，所以，i要尽可能靠右，所以从后往前遍历

```
class Solution {
    public void nextPermutation(int[] nums) {
        //i 指针在前，从后往前遍历
        for(int i = nums.length-1; i>=0; i--){
            // j 指针在后
            for(int j = nums.length-1; j>i; j--){
                if(nums[i] < nums[j]){
                    swap(nums,i,j);
                    Arrays.sort(nums,i+1,nums.length);
                    return;
                }
            }
        }
        //如果没有找到
        Arrays.sort(nums);
```

```
    }
    void swap(int[] a, int i, int j){
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

19、部分排序（面试题 16.16）

面试题 16.16. 部分排序

难度 中等 28 收藏 分享 切换为英文 关注 反馈

给定一个整数数组，编写一个函数，找出索引 m 和 n，只要将索引区间 [m,n] 的元素排好序，整个数组就是有序的。注意：n-m 尽量最小，也就是说，找出符合条件的最短序列。函数返回值为 [m,n]，若不存在这样的 m 和 n（例如整个数组是有序的），请返回 [-1,-1]。

示例：

输入： [1,2,4,7,10,11,7,12,6,7,16,18,19]
输出： [3,9]

提示：

- 0 <= len(array) <= 1000000

https://leetcode-cn.com/problems/sub-sort-lcci/solution/yi-bian-bian-li-shuang-zhi-zhen-by-chen-wei-zhe/

解题思路

默认升序（降序也只是改一点代码，不影响）

原理：如果左侧最大值大于中间的最小值，则一定会被中间序列包括；同理，如果右侧最小值大于中间的最大值，则一定会被中间序列包括。

一遍遍历 + 两个指针（两次扫描可一次遍历完成）

1、从前向后扫描数组，判断当前array[i]是否比max小，是则将last置为当前array下标i，否则更新max;

2、从后向前扫描数组，判断当前array[len - 1 - i]是否比min大，是则将first置位当前下标len - 1 - i，否则更新min;

3、返回{first， last}

```
class Solution {
    public int[] subSort(int[] array) {
     if (array == null || array.length == 0) return new int[] {-1, -1};
        int m = -1, n = -1;
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        int len = array.length;
        for (int i = 0; i < len; i++){
            if (array[i] < max) {
                n = i;
            } else {
```

```
                max = Math.max(max, array[i]);
            }
            if (array[len - 1 - i] > min) {
                m = len - 1 - i;
            } else {
                min = Math.min(min, array[len - 1 - i]);
            }
        }
        return new int[] {m, n};
    }
}
```

20、删除排序数组中的重复项（26）【删除排序数组中的重复数字（lintcode100）】

26. 删除排序数组中的重复项

难度 简单 1551 收藏 分享 切换为英文 关注 反馈

给定一个排序数组，你需要在 原地 删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 **修改输入数组** 并在使用 O(1) 额外空间的条件下完成。

示例 1:

给定数组 *nums* = [1,1,2],

函数应该返回新的长度 2，并且原数组 *nums* 的前两个元素被修改为 1，2。

你不需要考虑数组中超出新长度后面的元素。

示例 2:

给定 *nums* = [0,0,1,1,1,2,2,3,3,4],

函数应该返回新的长度 5，并且原数组 *nums* 的前五个元素被修改为 0，1，2，3，4。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**「引用」**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中该长度范围内的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/solution/shuang-zhi-zhen-shan-chu-zhong-fu-xiang-dai-you-hu/

解题思路：

解法： 双指针

首先注意数组是有序的，那么重复的元素一定会相邻。

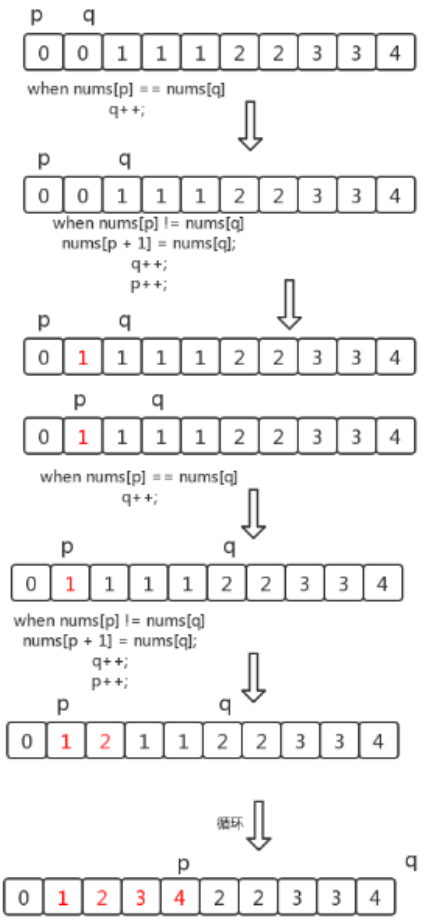
要求删除重复元素，实际上就是将不重复的元素移到数组的左侧。

考虑用 2 个指针，一个在前记作 `p`，一个在后记作 `q`，算法流程如下：

1.比较 `p` 和 `q` 位置的元素是否相等。

如果相等， `q` 后移 1 位
如果不相等，将 `q` 位置的元素复制到 `p+1` 位置上， `p` 后移一位， `q` 后移 1 位
重复上述过程，直到 `q` 等于数组长度。

返回 `p + 1`，即为新数组长度。



方法 1：未优化版本双指针

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int p = 0, q = 1;
        while (q < nums.length) {
            if (nums[p] != nums[q]) {
                nums[p+1] = nums[q];
                p++;
            }
            q++;
        }
        return p + 1;
    }
}
```

优化：

这是大部分题解都没有提出的，在这里提一下。

考虑如下数组：

此时数组中没有重复元素，按照上面的方法，每次比较时 `nums[p]` 都不等于 `nums[q]`，因此就会将 `q` 指向的元素原地复制一遍，这个操作其实是不必要的。

因此我们可以添加一个小判断，当 `q - p > 1` 时，才进行复制。

YES的练级攻略

方法 2：优化版本双指针

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int p = 0, q = 1;
        while (q < nums.length) {
            if (nums[p] != nums[q]) {
                if (q - p > 1) nums[p+1] = nums[q];//只有这里加了个判断
                p++;
            }
            q++;
        }
        return p + 1;
    }
}
```

21、寻找峰值（162）（lintcode 75）

更多硬核内容就在公众号「yes 的练级攻略」

162. 寻找峰值

难度 中等 270 收藏 分享 切换为英文 关注 反馈

峰值元素是指其值大于左右相邻值的元素。

给定一个输入数组 `nums`，其中 `nums[i] ≠ nums[i+1]`，找到峰值元素并返回其索引。

数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞`。

示例 1:

```
输入: nums = [1,2,3,1]
输出: 2
解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

示例 2:

```
输入: nums = [1,2,1,3,5,6,4]
输出: 1 或 5
解释: 你的函数可以返回索引 1，其峰值元素为 2;
     或者返回索引 5， 其峰值元素为 6。
```

说明:

你的解法应该是 *O(logN)* 时间复杂度的。

中文 English

描述

你给出一个整数数组(size为n)，其具有以下特点:

- 相邻位置的数字是不同的
- A[0] < A[1] 并且 A[n - 2] > A[n - 1]

假定P是峰值的位置则满足 `A[P] > A[P-1]` 且 `A[P] > A[P+1]`，返回数组中任意一个峰值的位置。

- 数组保证至少存在一个峰
- 如果数组存在多个峰，返回其中任意一个就行
- 数组至少包含 3 个数

您在真实的面试中是否遇到过这个题? 是 题目纠错

样例

```
样例 1:
        输入:  [1, 2, 1, 3, 4, 5, 7, 6]
        输出:  1 or 6

        解释:
        返回峰顶元素的下标


样例 2:
        输入: [1,2,3,4,1]
        输出:  3
```

挑战

Time complexity O(logN)

https://leetcode-cn.com/problems/find-peak-element/solution/xun-zhao-feng-zhi-by-leetcode/

方法三：迭代二分查找

算法

上述二分查找方法使用了递归。我们可以通过迭代达到同样的效果。本方法即为迭代实现二分查找。

Java

```java
public class Solution {
    public int findPeakElement(int[] nums) {
        int l = 0, r = nums.length - 1;
        while (l < r) {
            int mid = (l + r) / 2;
            if (nums[mid] > nums[mid + 1])
                r = mid;
            else
                l = mid + 1;
        }
        return l;
    }
}
```

复杂度分析

- 时间复杂度：$O(log_2(n))$。每一步都将搜索空间减半。因此，总的搜索空间只需要 $log_2(n)$ 步。其中 n 为 $nums$ 数组的长度。
- 空间复杂度：$O(1)$。只使用了常数空间。

```java
public class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = (left + right) / 2;
            if (nums[mid] > nums[mid + 1])
                right = mid;
            else
                left = mid + 1;
        }
        return left;
    }
}
```

22、绝对差不超过限制的三元子数组（Lintcode1529）

1529. 绝对差不超过限制的三元子数组

给定一个递增的整数数组 `nums`，和一个表示限制的整数 `limit`，请你返回满足条件的三元子数组的个数，使得该子数组中的任意两个元素之间的绝对差小于或者等于 `limit`。

如果不存在满足条件的子数组，则返回 0。

样例

样例 1:

```
输入：[1, 2, 3, 4], 3
输出：4
解释：可选方案有(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)。因此，满足条件的三元组有4个。
```

样例 2:

```
输入：[1, 10, 20, 30, 50], 19
输出：1
解释：唯一可行的三元组是(1, 10, 20)，所以答案为1。
```

挑战

你可以只用O(n)的时间复杂度解决这个问题吗?

注意事项

数据范围：1 ≤ len(nums) ≤ 1e4，1 ≤ limit ≤ 1e6，0 ≤ nums[i] ≤ 1e6

由于答案可能很大，请返回它对 `99997867` 取余后的结果。

使用双指针。初始化左指针 `left` 为 0，右指针 `right` 为 2。每次右指针移动一位，指向三元组中最后一位元素，左指针向右移动到满足 `nums[right]-nums[left]<=limit` 为止。如果 `right-left>=2`，说明可以构成三元组，从 `nums[left, right-1]`这个区间中任挑两位作为三元组的前两个元素，有 c(n,2)种选择。直到右指针遍历到最后一个元素为止。由于左右指针都只遍历了一次，所以时间复杂度为 O(n)，空间复杂度为 O(1)。

排列组合公式

$$A_n^m = n(n-1)\cdots(n-m+1) = \frac{n!}{(n-m)!}$$

$$C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!(n-m)!} = C_n^{n-m}$$

```
public class Solution {
    /**
     * @param nums: the given array
     * @param limit: the limit on the absolute difference of the subarray
     * @return: Find the number of triplet subarray with the absolute difference less than or equal to limit
     */
    public int tripletSubarray(int[] nums, int limit) {
        // write your code here
        int left = 0, res = 0;
        int mod = 99997867;
        for (int right = 2; right < nums.length; right++) {
```

```
                while (nums[right] - nums[left] > limit) {
                    left++;
                }
                if (right - left >= 2) {
                    res += (right - left) * (right - left - 1) / 2;
                }
            }
            return res % mod;
        }
}
```

23、移动零（283）

283. 移动零

难度 简单 668 收藏 分享 切换为英文 关注 反馈

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:

```
输入: [0,1,0,3,12]
输出: [1,3,12,0,0]
```

说明:

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

https://leetcode-cn.com/problems/move-zeroes/solution/javabao-li-qiu-jie-fa-he-shuang-zhi-zhen-fa-by-and/

第一步：设置指针 head 和 tail，head 默认为 0，表示非零数的依次存储位置，tail 则依次遍历的数组索引位置；

第二步：如果 tail 指针位置元素是非零元素，则与 head 指针位置元素交换；

```
class Solution {
    public void moveZeroes(int[] nums) {
        int left = 0, right = nums.length;
        for (int i = 0; i < nums.length; i++){
            if (nums[i] != 0) {
                int tmp = nums[left];
                nums[left] = nums[i];
                nums[i] = tmp;
                left++;
            }
        }
    }
}
```

24、调整数组顺序使奇数位于偶数前面（剑指 21）

剑指 Offer 21. 调整数组顺序使奇数位于偶数前面

难度 简单 38 收藏 分享 切换为英文 关注 反馈

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

示例：

```
输入：nums = [1,2,3,4]
输出：[1,3,2,4]
注：[3,1,2,4] 也是正确的答案之一。
```

提示：

1. `1 <= nums.length <= 50000`
2. `1 <= nums[i] <= 10000`

https://leetcode-cn.com/problems/diao-zheng-shu-zu-shun-xu-shi-qi-shu-wei-yu-ou-shu-qian-mian-lcof/solution/javashuang-zhi-zhen-shi-xian-by-neil2020/

1.我们就可以定义两个指针 left 和 right，left 指向数组的左边，right 指向数组的右边
2.然后先从 left 指针开始往右边扫描，当找到一个偶数时，就停下来
3.再让 right 指针从右往左开始扫描，当找到一个奇数时，就停下来
4.这时，我们就可以交换这两个数字的位置
我们交换完后就又执行第 2 步，开始找下一个需要交换的地方。
当 left 和 right 指向同一个地方时，说明已经查找完毕，满足了题目的要求。

```
class Solution {
    public int[] exchange(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            // 当找到一个偶数时，就跳出循环。
            // (这里有个求奇偶数的小技巧，就是当一个数是奇数时，它的二进制表示的最后一位肯定是 1
            while (left < right && (nums[left] & 1) == 1) {
                left++;
            }
            // 当找到一个奇数时，就跳出循环
            while (left < right && (nums[right] & 1) == 0) {
                right--;
            }
            // 如果两个指针还没有碰到一起时，说明找到了需要交换的位置
            if (left < right) {
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
            }
```

```
            }
            return nums;
        }
    }
```

25、合并两个有序数组（88）

88. 合并两个有序数组

难度 简单 577 收藏 分享 切换为英文 关注 反馈

给你两个有序整数数组 *nums1* 和 *nums2*，请你将 *nums2* 合并到 *nums1* 中，使 *nums1* 成为一个有序数组。

说明:

- 初始化 *nums1* 和 *nums2* 的元素数量分别为 *m* 和 *n* 。
- 你可以假设 *nums1* 有足够的空间（空间大小大于或等于 *m + n*）来保存 *nums2* 中的元素。

示例:

输入:
`nums1 = [1,2,3,0,0,0], m = 3`
`nums2 = [2,5,6], n = 3`

输出: `[1,2,2,3,5,6]`

https://leetcode-cn.com/problems/merge-sorted-array/solution/he-bing-liang-ge-you-xu-shu-zu-by-leetcode/

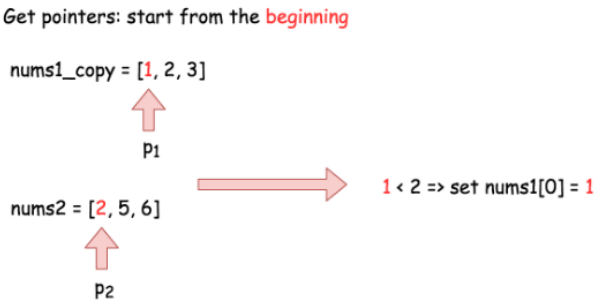
方法二：双指针 / 从前往后

直觉

一般而言，对于有序数组可以通过 *双指针法* 达到$O(n+m)$的时间复杂度。

最直接的算法实现是将指针 `p1` 置为 `nums1` 的开头， `p2` 为 `nums2` 的开头，在每一步将最小值放入输出数组中。

由于 `nums1` 是用于输出的数组，需要将 `nums1` 中的前 `m` 个元素放在其他地方，也就需要 $O(m)$ 的空间复杂度。



复杂度分析

- 时间复杂度：$O(n+m)$。
- 空间复杂度：$O(m)$。

方法 1：双指针从前往后【空间复杂度为 o(n)】

```
class Solution {
```

```
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int[] tmp = new int[nums1.length];
        int p1 = 0, p2 = 0, k = 0;
        while (p1 < m && p2 < n){
            if (nums1[p1] <= nums2[p2]) {
                tmp[k++] = nums1[p1++];
            } else {
                tmp[k++] = nums2[p2++];
            }
        }
        while (p1 < m) tmp[k++] = nums1[p1++];
        while (p2 < n) tmp[k++] = nums2[p2++];
        for (int i = 0; i < nums1.length; i++){
            nums1[i] = tmp[i];
        }
    }
}
```

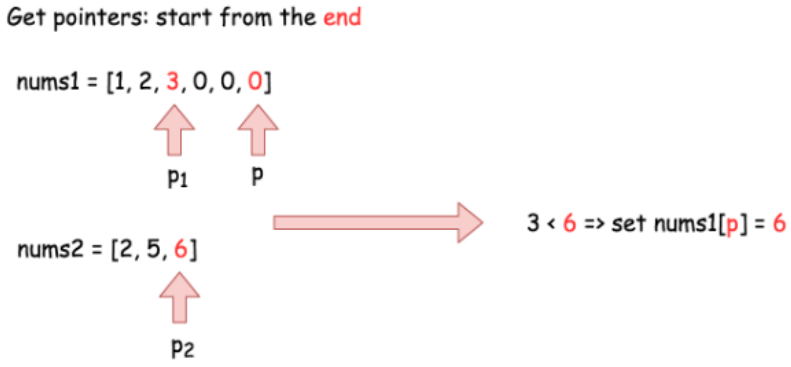
方法三：双指针 / 从后往前

直觉

方法二已经取得了最优的时间复杂度$O(n+m)$，但需要使用额外空间。这是由于在从头改变 `nums1` 的值时，需要把 `nums1` 中的元素存放在其他位置。

> 如果我们从结尾开始改写 `nums1` 的值又会如何呢？这里没有信息，因此不需要额外空间。

这里的指针 `p` 用于追踪添加元素的位置。



复杂度分析

- 时间复杂度：$O(n+m)$。
- 空间复杂度：$O(1)$。

方法 2：双指针从后往前【空间复杂度为 o(1)】

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        // two get pointers for nums1 and nums2
        int p1 = m - 1;
        int p2 = n - 1;
```

```
        // set pointer for nums1
        int p = m + n - 1;

        // while there are still elements to compare
        while ((p1 >= 0) && (p2 >= 0))
        // compare two elements from nums1 and nums2
        // and add the largest one in nums1
        nums1[p--] = (nums1[p1] < nums2[p2]) ? nums2[p2--] : nums1[p1--
];

        // add missing elements from nums2
        System.arraycopy(nums2, 0, nums1, 0, p2 + 1);
    }
}
```

26、最小差（面试题 16.06）

面试题 16.06. 最小差

难度 中等 16 收藏 分享 切换为英文 关注 反馈

给定两个整数数组 a 和 b，计算具有最小差绝对值的一对数值（每个数组中取一个值），并返回该对数值的差

示例：

```
输入：{1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}
输出： 3，即数值对(11, 8)
```

提示：

- 1 <= a.length, b.length <= 100000
- -2147483648 <= a[i], b[i] <= 2147483647
- 正确结果在区间[-2147483648, 2147483647]内

https://leetcode-cn.com/problems/smallest-difference-lcci/solution/shuang-zhi-zhen-shi-yong-longjie-jue-ce-shi-yong-l/

解题思路

```
先排序，然后使用双指针，查找最小查
1. min = min(min, abs(a[i] - b[j]));
2. 如果 a[i] < b[j] 那么 i++, 将使用a中比较大的数和b[j]比较。
假如i不变，j++， 那么差值会更大，所以我们可以当前a[i] 和b的差值应该是最小的。如果i >= a.length, 使用最后一个数即可
3. 如果 a[i] > b[j] 那么j++;
4. 如果 a[i] == b[j] 直接返回0，0是全球最小的数;

使用long解决溢出问题
```

```
class Solution {
    /**
    先排序，然后使用双指针，查找最小查
    1. min = min(min, abs(a[i] - b[j]));
    2. 如果 a[i] < b[j] 那么 i++, 将使用 a 中比较大的数和 b[j]比较。
```

```
    假如 i 不变，j++， 那么差值会更大，所以我们可以当前 a[i] 和 b 的差值应该是最小的。如果 i >= a.length, 使用最后一个数即可
    3. 如果 a[i] > b[j] 那么 j++;
    4. 如果 a[i] == b[j] 直接返回 0，0 是全球最小的数;
    使用 long 解决溢出问题
    **/
    public int smallestDifference(int[] a, int[] b) {
        Arrays.sort(a);
        Arrays.sort(b);
        int i = 0, j  = 0;
        int min = Integer.MAX_VALUE;
        while(i < a.length || j < b.length) {
            long ai = i < a.length? a[i] : a[a.length - 1]; //如果 i 溢出，那么使用最后一个元素
            long bi = j < b.length? b[j] : b[b.length - 1];
            min = (int) Math.min(min, Math.abs(ai - bi));
            if (ai == bi) return 0;
            //如果 i 溢出，j++即可
            if(i >= a.length) {
                j++;
                continue;
            }
            if(j >= b.length) {
                i++;
                continue;
            }
            if (ai < bi) i++;
            else j++;
        }
        return min;
    }
}
```

27、测试函数

```
import java.util.*;
public class Test {
    /* WaterMaxArea(11)
    public static void main(String[] args) {
        //int[] height = {1,8,6,2,5,4,8,3,7};
        Scanner sc = new Scanner(System.in);
        int length = 9;
        int[] height = new int[length];
        for (int i = 0; i < length; i++) {
            height[i] = sc.nextInt();
        }
```

```
        WaterMaxArea wam = new WaterMaxArea();
        int res = wam.maxArea(height);
        System.out.println(res);

    }
    */
    /* Num3Sum(15)
    public static void main(String[] args) {
        //int[] nums = {-1, 0, 1, 2, -1, -4};
        Scanner sc = new Scanner(System.in);
        int length = 6;
        int[] nums = new int[length];
        for (int i = 0; i < length; i++) {
            nums[i] = sc.nextInt();
        }
        Num3Sum tp = new Num3Sum();
        List<List<Integer>> res = new LinkedList<>();
        res = tp.threeSum(nums);
        System.out.println(res);

    }
    */
    /* Num4Sum(18)
    public static void main(String[] args) {

        int[] nums = {0,4,-5,2,-2,4,2,-1,4};
        int target = 12;
//      Scanner sc = new Scanner(System.in);
//      int length = 6;
//      int[] nums = new int[length];
//      for (int i = 0; i < length; i++) {
//          nums[i] = sc.nextInt();
//      }
        Num4Sum tp = new Num4Sum();
        List<List<Integer>> res = new LinkedList<>();
        res = tp.fourSum(nums,target);
        System.out.println(res);
    }
    */
    /* Closest3Num(16)
    public static void main(String[] args) {
        int[] nums = {-1,2,1,-4};
        int target = 1;
        Closest3Num cn = new Closest3Num();
```

禁止商用
公众号：
yes的练级攻略

```
    int res = cn.threeSumClosest(nums, target);
    System.out.println(res);
}
*/
/*IntersectionLsit(160)
public static void main(String[] args) {
    ListNode intersectVal = new ListNode(2);
    MyLinkList list1 = new MyLinkList();
    list1.add(new ListNode(0));
    list1.add(new ListNode(9));
    list1.add(new ListNode(1));
    list1.add(intersectVal);
    list1.add(new ListNode(4));

    MyLinkList list2 = new MyLinkList();
    list2.add(new ListNode(3));
    list2.add(intersectVal);
    list2.add(new ListNode(4));
    IntersectionLsit i = new IntersectionLsit();
    ListNode node = i.getIntersectionNode(list1.head,list2.head);
    System.out.println(node.val);
}
*/
/*RemoveNthFromEnd(19)
public static void main(String[] args) {
    MyLinkList list = new MyLinkList();
    list.add(new ListNode(1));
    list.add(new ListNode(2));
    list.add(new ListNode(3));
    list.add(new ListNode(4));
    list.add(new ListNode(5));
    RemoveNthFromEnd rnf = new RemoveNthFromEnd();
    int n = 2;
    ListNode node = rnf.removeNthFromEnd(list.head, n);
    while (node != null) {
        System.out.println(node.val+" ");
        node = node.next;
    }

}
*/
/*CycleList(141)
public static void main(String[] args) {
    ListNode node0 = new ListNode(3);
```

禁止商用 公众号：yes的练级攻略

```
        ListNode node1 = new ListNode(2);
        ListNode node2 = new ListNode(0);
        ListNode node3 = new ListNode(-4);
        node0.next = node1;
        node1.next = node2;
        node2.next = node3;
//      node3.next = node1;
//      ListNode node0 = new ListNode(1);
        CycleList c = new CycleList();
        boolean h = c.hasCycle(node0);
        System.out.println(h);
    }
    */

    public static void main(String[] args) {
        ListNode node0 = new ListNode(1);
        ListNode node1 = new ListNode(2);
        ListNode node2 = new ListNode(3);
        ListNode node3 = new ListNode(2);
        ListNode node4 = new ListNode(1);
        node0.next = node1;
        node1.next = node2;
        node2.next = node3;
        node3.next = node4;
        Palindrome p = new Palindrome();
        boolean a = p.isPalindrome(node0);
        System.out.println(a);
    }
}
//链表需要
class ListNode {
    public int val;
    public ListNode next;
    public ListNode(int x) {
        this.val = x;
        this.next = null;
    }
}
//链表需要
class MyLinkList{
    public ListNode head = new ListNode(0);

    public void add(ListNode listnode) {
        ListNode tmp = head;
```

禁止商用 公众号：yes的练级攻略

```
        while(tmp.next!=null) {
            tmp =tmp.next;
        }
        tmp.next =listnode;
    }
}
```

28、

四、动态规划，偶尔有贪心

1、斐波那契数列（剑指 10-1）

剑指 Offer 10- I. 斐波那契数列

难度 简单 48 收藏 分享 切换为英文 关注 反馈

写一个函数，输入 `n` ，求斐波那契（Fibonacci）数列的第 `n` 项。斐波那契数列的定义如下：

```
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), 其中 N > 1.
```

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

```
输入：n = 2
输出：1
```

示例 2：

```
输入：n = 5
输出：5
```

https://leetcode-cn.com/problems/fei-bo-na-qi-shu-lie-lcof/solution/mian-shi-ti-10-i-fei-bo-na-qi-shu-lie-dong-tai-gui/

动态规划解析：

- **状态定义：** 设 dp 为一维数组，其中 $dp[i]$ 的值代表 斐波那契数列第 i 个数字 。
- **转移方程：** $dp[i+1]=dp[i]+dp[i-1]$ ，即对应数列定义 $f(n+1)=f(n)+f(n-1)$ ；
- **初始状态：** $dp[0]=0, dp[1]=1$ ，即初始化前两个数字；
- **返回值：** $dp[n]$ ，即斐波那契数列的第 n 个数字。

空间复杂度优化：

> 若新建长度为 n 的 dp 列表，则空间复杂度为 $O(N)$ 。

- 由于 dp 列表第 i 项只与第 $i-1$ 和第 $i-2$ 项有关，因此只需要初始化三个整形变量 `sum` , `a` , `b` ，利用辅助变量 sum 使 a,b 两数字交替前进即可 *（具体实现见代码）* 。
- 节省了 dp 列表空间，因此空间复杂度降至 $O(1)$ 。

循环求余法：

> 大数越界： 随着 n 增大， $f(n)$ 会超过 `Int32` 甚至 `Int64` 的取值范围，导致最终的返回值错误。

- **求余运算规则：** 设正整数 x,y,p ，求余符号为 $\odot$ ，则有 $(x+y)\odot p=(x\odot p+y\odot p)\odot p$ 。
- **解析：** 根据以上规则，可推出 $f(n)\odot p=[f(n-1)\odot p+f(n-2)\odot p]\odot p$ ，从而可以在循环过程中每次计算 $sum=(a+b)\odot 1000000007$ ，此操作与最终返回前取余等价。

```
class Solution {
    public int fib(int n) {
```

```
        int res = 0, preTwo = 0, preOne = 1;
        if (n < 2) return n;
        for (int i = 2; i <= n; i++) {
            res = (preOne + preTwo) % 1000000007;
            preTwo = preOne;
            preOne = res;
        }
        return res;
    }
}
```

2、青蛙跳台阶问题（剑指 10-2）【爬楼梯（70）】

剑指 Offer 10- II. 青蛙跳台阶问题

难度 简单　54　收藏　分享　切换为英文　关注　反馈

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 `n` 级的台阶总共有多少种跳法。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

输入： `n = 2`
输出： `2`

示例 2：

输入： `n = 7`
输出： `21`

示例 3：

输入： `n = 0`
输出： `1`

提示：

- `0 <= n <= 100`

注意：本题与主站 70 题相同：https://leetcode-cn.com/problems/climbing-stairs/

https://leetcode-cn.com/problems/qing-wa-tiao-tai-jie-wen-ti-lcof/solution/mian-shi-ti-10-ii-qing-wa-tiao-tai-jie-wen-ti-dong/

动态规划解析：

- **状态定义：** 设 dp 为一维数组，其中 $dp[i]$ 的值代表 `斐波那契数列第 $i$ 个数字` 。
- **转移方程：** $dp[i+1]=dp[i]+dp[i-1]$ ，即对应数列定义 $f(n+1)=f(n)+f(n-1)$ ；
- **初始状态：** $dp[0]=1, dp[1]=1$ ，即初始化前两个数字；
- **返回值：** $dp[n]$ ，即斐波那契数列的第 n 个数字。

空间复杂度优化：

> 若新建长度为 n 的 dp 列表，则空间复杂度为 $O(N)$ 。

- 由于 dp 列表第 i 项只与第 $i-1$ 和第 $i-2$ 项有关，因此只需要初始化三个整形变量 `sum` ， `a` ， `b` ，利用辅助变量 sum 使 a,b 两数字交替前进即可 *(具体实现见代码)* 。
- 因为节省了 dp 列表空间，因此空间复杂度降至 $O(1)$ 。

循环求余法：

> **大数越界：** 随着 n 增大, $f(n)$ 会超过 `Int32` 甚至 `Int64` 的取值范围，导致最终的返回值错误。

- **求余运算规则：** 设正整数 x,y,p ，求余符号为 $\odot$ ，则有 $(x+y)\odot p=(x\odot p+y\odot p)\odot p$ 。
- **解析：** 根据以上规则，可推出 $f(n)\odot p=[f(n-1)\odot p+f(n-2)\odot p]\odot p$ ，从而可以在循环过程中每次计算 $sum=a+b\odot 1000000007$ ，此操作与最终返回前取余等价。

复杂度分析：

- **时间复杂度 $O(N)$ ：** 计算 $f(n)$ 需循环 n 次，每轮循环内计算操作使用 $O(1)$ 。
- **空间复杂度 $O(1)$ ：** 几个标志变量使用常数大小的额外空间。

```
class Solution {
    public int numWays(int n) {
        int a = 1, b = 1, sum;
        for(int i = 0; i < n; i++){
            sum = (a + b) % 1000000007;
            a = b;
            b = sum;
        }
        return a;
    }
}
```

70. 爬楼梯

难度 简单 1182 收藏 分享 切换为英文 关注 反馈

假设你正在爬楼梯。需要 *n* 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 *n* 是一个正整数。

示例 1：

```
输入： 2
输出： 2
解释： 有两种方法可以爬到楼顶。
1.  1 阶 + 1 阶
2.  2 阶
```

示例 2：

```
输入： 3
输出： 3
解释： 有三种方法可以爬到楼顶。
1.  1 阶 + 1 阶 + 1 阶
2.  1 阶 + 2 阶
3.  2 阶 + 1 阶
```

```java
class Solution {
    public int climbStairs(int n) {
        int a = 1, b = 1, sum;
        for(int i = 0; i < n; i++){
            sum = a + b; //与青蛙跳台阶问题的差别只在于这边不取模
            a = b;
            b = sum;
        }
        return a;
    }
}
```

3、三步问题（面试题 08.01）

面试题 08.01. 三步问题

难度 简单 21 收藏 分享 切换为英文 关注 反馈

三步问题。有个小孩正在上楼梯，楼梯有n阶台阶，小孩一次可以上1阶、2阶或3阶。实现一种方法，计算小孩有多少种上楼梯的方式。结果可能很大，你需要对结果模1000000007。

示例1:

输入: n = 3
输出: 4
说明: 有四种走法

示例2:

输入: n = 5
输出: 13

提示:

1. n范围在[1, 1000000]之间

```
class Solution {
    public int waysToStep(int n) {
        if(n < 3) return n;
        if(n == 3) return 4;
        int mod = 1000000007, a = 4, b = 2, c = 1;
        for(int i = 0; i < n - 3; i++){
           int temp_a = a;
           int temp_b = b;
           a = (temp_a + (b + c)% mod)% mod;
           b = temp_a;
           c = temp_b;
        }
        return a;
    }
}
```

4、变态跳台阶

题目：变态跳台阶（一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。）

```
public class Solution {
    public int JumpFloorII(int target) {
        return (int) Math.pow(2,target-1);
    }
}
```

5、通配符匹配（44）

【与正则表达式的不同：这边的'*'是匹配任意字符串包括空串的，正则里面是零至多个的前一个字符。】

44. 通配符匹配

难度 困难 482 收藏 分享 切换为英文 关注 反馈

给定一个字符串 (s) 和一个字符模式 (p)，实现一个支持 '?' 和 '*' 的通配符匹配。

```
'?' 可以匹配任何单个字符。
'*' 可以匹配任意字符串（包括空字符串）。
```

两个字符串完全匹配才算匹配成功。

说明:

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 *。

示例 1:

```
输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入:
s = "aa"
p = "*"
输出: true
解释: '*' 可以匹配任意字符串。
```

示例 3:

```
输入:
s = "cb"
p = "?a"
输出: false
解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。
```

示例 4:

```
输入:
s = "adceb"
p = "*a*b"
输出: true
解释: 第一个 '*' 可以匹配空字符串, 第二个 '*' 可以匹配字符串 "dce".
```

示例 5:

```
输入:
s = "acdcb"
p = "a*c?b"
输出: false
```

https://leetcode-cn.com/problems/wildcard-matching/solution/zi-fu-chuan-dong-tai-gui-hua-bi-xu-miao-dong-by-sw/

更多硬核内容就在公众号「yes 的练级攻略」

一、题目分析

题意：实现支持'?' 和 '*' 的通配符匹配的字符串匹配。

分析：类似 *最长公共子串*，*最长公共子序列*，*编辑距离* 等 **求 2 个字符串(或数组)之间的某种关系的题目**，一般来说都是有动态规划的解法的。

1、状态定义：

`dp[i][j]` 表示 p 的前 i 个字符和 s 的前 j 个字符是否匹配。

2、状态转移：

如果 `p[i - 1] == s[j - 1]` 或 `p[i - 1] == '?'`，表示当前的字符串是匹配的，`dp[i][j]` 可以从 `dp[i - 1][j - 1]` 转移而来。
如果 `p[i - 1] == '*'`，这个位置可以匹配 0 到 若干个字符。那么 `dp[i][j]` 可以从 `dp[i - 1][j]` 转移而来（表示当前星号没有匹配字符），也可以从 `dp[i][j - 1]` 转移而来（表示当前星号匹配了当前的位置的字符）。因为只要任意一种匹配即可，所以这里是逻辑或的关系。

状态转移方程如下：

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & p[i-1] == s[j-1] \ || \ p[i-1] == ? \\ dp[i-1][j] \ | \ dp[i][j-1] & p[i-1] == * \end{cases}$$

3、初始条件

`dp[0][0] = true` 表示空串是匹配的。

处理一下匹配串 p 以若干个星号开头的情况。因为星号是可以匹配空串的：

```
for (int i = 1; i <= p.length(); i++) {
    if (p.charAt(i - 1) != '*') {
        break;
    }
    dp[i][0] = true;
}
```

复杂度分析

- 时间复杂度：$O(mn)$，其中 m 和 n 分别是字符串 s 和模式 p 的长度。
- 空间复杂度：$O(mn)$，即为存储所有 $(m+1)(n+1)$ 个状态需要的空间。此外，在状态转移方程中，由于 $dp[i][j]$ 只会从 $dp[i][..]$ 以及 $dp[i-1][..]$ 转移而来，因此我们可以使用滚动数组对空间进行优化，即用两个长度为 $n+1$ 的一维数组代替整个二维数组进行状态转移，空间复杂度为 $O(n)$。

方法 1：二维动规（36ms）

```
class Solution {
    public boolean isMatch(String s, String p) {
        int len1 = p.length(), len2 = s.length();
        boolean[][] dp = new boolean[len1 + 1] [len2 + 1];
        dp[0][0] = true;
        for (int i = 1; i <= len1; i++) {
            if (p.charAt(i - 1) != '*') {
                break;
            }
            dp[i][0] = true;
        }
```

```
        for (int i = 1; i <= len1; i++) {
            for (int j = 1; j <= len2; j++) {
                if (p.charAt(i - 1) == s.charAt(j - 1) || p.charAt(i -
1) == '?') {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (p.charAt(i - 1) == '*') {
                    dp[i][j] = dp[i - 1][j] | dp[i][j - 1];
                }
            }
        }
        return dp[len1][len2];
    }
}
```

https://leetcode-cn.com/problems/wildcard-matching/solution/tong-pei-fu-pi-pei-by-leetcode-solution/

方法二：贪心算法

方法一的瓶颈在于对星号 * 的处理方式：使用动态规划枚举所有的情况。由于星号是「万能」的匹配字符，连续的多个星号和单个星号实际上是等价的，那么不连续的多个星号呢？

我们以 $p = *\,\text{abcd}\,*$ 为例，p 可以匹配**所有包含子串** abcd **的字符串**，也就是说，我们只需要暴力地枚举字符串 s 中的每个位置作为起始位置，并判断对应的子串是否为 abcd 即可。这种暴力方法的时间复杂度为 $O(mn)$，与动态规划一致，但不需要额外的空间。

如果 $p = *\,\text{abcd}\,*\,\text{efgh}\,*\,\text{i}\,*$ 呢？显然，p 可以匹配所有依次出现子串 abcd、efgh、i 的字符串。此时，对于任意一个字符串 s，我们首先暴力找到最早出现的 abcd，随后从下一个位置开始暴力找到最早出现的 efgh，最后找出 i，就可以判断 s 是否可以与 p 匹配。这样「贪心地」找到最早出现的子串是比较直观的，因为如果 s 中多次出现了某个子串，那么我们选择最早出现的位置，可以使得后续子串能被找到的机会更大。

因此，如果模式 p 的形式为 $*\,u_1\,*\,u_2\,*\,u_3\,*\cdots*\,u_x\,*$，即字符串（可以为空）和星号交替出现，并且首尾字符均为星号，那么我们就可以设计出下面这个基于贪心的暴力匹配算法。算法的本质是：如果在字符串 s 中首先找到 u_1，再找到 $u_2, u_3, \cdots, u_x$，那么 s 就可以与模式 p 匹配，伪代码如下：

```
// 我们用 sIndex 和 pIndex 表示当前遍历到 s 和 p 的位置
// 此时我们正在 s 中寻找某个 u_i
// 其在 s 和 p 中的起始位置为 sRecord 和 pRecord

// sIndex 和 sRecord 的初始值为 0
// 即我们从字符串 s 的首位开始匹配
sIndex = sRecord = 0

// pIndex 和 pRecord 的初始值为 1
// 这是因为模式 p 的首位是星号，那么 u_1 的起始位置为 1
pIndex = pRecord = 1

while sIndex < s.length and pIndex < p.length do
    if p[pIndex] == '*' then
        // 如果遇到星号，说明找到了 u_i，开始寻找 u_i+1
        pIndex += 1
        // 记录下起始位置
```

```
            sRecord = sIndex
            pRecord = pIndex
        else if match(s[sIndex], p[pIndex]) then
            // 如果两个字符可以匹配，就继续寻找 u_i 的下一个字符
            sIndex += 1
            pIndex += 1
        else if sRecord + 1 < s.length then
            // 如果两个字符不匹配，那么需要重新寻找 u_i
            // 枚举下一个 s 中的起始位置
            sRecord += 1
            sIndex = sRecord
            pIndex = pRecord
        else
            // 如果不匹配并且下一个起始位置不存在，那么匹配失败
            return False
        end if
    end while

    // 由于 p 的最后一个字符是星号，那么 s 未匹配完，那么没有关系
    // 但如果 p 没有匹配完，那么 p 剩余的字符必须都是星号
    return all(p[pIndex] ~ p[p.length - 1] == '*')
```

然而模式 p 并不一定是 $*u_1*u_2*u_3*\cdots*u_x*$ 的形式：

- 模式 p 的开头字符不是星号；
- 模式 p 的结尾字符不是星号。

第二种情况处理起来并不复杂。如果模式 p 的结尾字符不是星号，那么就必须与字符串 s 的结尾字符匹配。那么我们不断地匹配 s 和 p 的结尾字符，直到 p 为空或者 p 的结尾字符是星号为止。在这个过程中，如果匹配失败，或者最后 p 为空但 s 不为空，那么需要返回 False。

第一种情况的处理也很类似，我们可以不断地匹配 s 和 p 的开头字符。下面的代码中给出了另一种处理方法，即修改 $sRecord$ 和 $tRecord$ 的初始值为 -1，表示模式 p 的开头字符不是星号，并且在匹配失败时进行判断，如果它们的值仍然为 -1，说明没有「反悔」重新进行匹配的机会。

复杂度分析

- 时间复杂度：
  - 渐进复杂度：$O(mn)$，其中 m 和 n 分别是字符串 s 和模式 p 的长度。从代码中可以看出，$s[sIndex]$ 和 $p[pIndex]$ 至多只会被匹配（判断）一次，因此渐进时间复杂度为 $O(mn)$。
  - 最坏情况的例子：$s = \text{aaa}\cdots\text{aaab}$，$p = *\,\text{ab}\,*$。
  - 平均复杂度：$O(m \log n)$。直观上来看，如果 s 和 p 是随机字符串，那么暴力算法的效率会非常高，因为大部分情况下只需要匹配常数次就可以成功匹配 u_i 或者枚举下一个常数位置。具体的分析可以参考论文 On the Average-case Complexity of Pattern Matching with Wildcards，注意论文中的分析是对于每一个 u_i 而言的，即模式中只包含小写字母和问号，本题相当于多个连续模式的情况。由于超出了面试难度。这里不再赘述。
- 空间复杂度：$O(1)$。

方法 2：贪心（3ms）

```
class Solution {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    public boolean isMatch(String s, String p) {
        int sRight = s.length(), pRight = p.length();
        while (sRight > 0 && pRight > 0 && p.charAt(pRight - 1) != '*')
 {
            if (charMatch(s.charAt(sRight - 1), p.charAt(pRight - 1)))
{
                --sRight;
                --pRight;
            } else {
                return false;
            }
        }

        if (pRight == 0) {
            return sRight == 0;
        }

        int sIndex = 0, pIndex = 0;
        int sRecord = -1, pRecord = -1;

        while (sIndex < sRight && pIndex < pRight) {
            if (p.charAt(pIndex) == '*') {
                ++pIndex;
                sRecord = sIndex;
                pRecord = pIndex;
            } else if (charMatch(s.charAt(sIndex), p.charAt(pIndex))) {
                ++sIndex;
                ++pIndex;
            } else if (sRecord != -1 && sRecord + 1 < sRight) {
                ++sRecord;
                sIndex = sRecord;
                pIndex = pRecord;
            } else {
                return false;
            }
        }

        return allStars(p, pIndex, pRight);
    }

    public boolean allStars(String str, int left, int right) {
        for (int i = left; i < right; ++i) {
            if (str.charAt(i) != '*') {
                return false;
```

```
            }
        }
        return true;
    }

    public boolean charMatch(char u, char v) {
        return u == v || v == '?';
    }
}
```

6、一次编辑（面试题 01.05）

面试题 01.05. 一次编辑

难度 中等 36 收藏 分享 切换为英文 关注 反馈

字符串有三种编辑操作:插入一个字符、删除一个字符或者替换一个字符。 给定两个字符串，编写一个函数判定它们是否只需要一次(或者零次)编辑。

示例 1:

输入:
first = "pale"
second = "ple"
输出: True

示例 2:

输入:
first = "pales"
second = "pal"
输出: False

https://leetcode-cn.com/problems/one-away-lcci/solution/zi-fu-chuan-bi-jiao-by-luorong/

yes的练级攻略

解题思路

主要思路是，只有一个地方需要修改，那么不妨定位到不同字符处。有以下两种情况
（1）长度相同：leetcode 与 leetkode。
那么我们需要找到 'c' 和 'k'，然后比较 'ode' 和 'ode' 是否相同。
（2）长度不同：leetcode 与 leetode。
我们发现 'c' 和 'o' 不相同，然后比较 'ode' 和 'ode' 是否相同。

复杂度

时间复杂度：O(n)。因为要遍历字符串的每个字符。
空间复杂度：O(1)。

```
class Solution {
    public boolean oneEditAway(String first, String second) {
        if (first == null || second == null) return false;
        int len1 = first.length();
        int len2 = second.length();
        if (Math.abs(len1 - len2) > 1) return false;
        if (len2 > len1) return oneEditAway(second, first);
```

```
            // 保持第一个比第二个长
            for (int i = 0; i < len2; i++){
                if (first.charAt(i) != second.charAt(i)){
                    // 如果是长度相同字符串，那就比较下一个，如果长度不一样，那就从该字符开始进行比较。
                    return first.substring(i + 1).equals(second.substring(len1 == len2 ? i + 1 : i));
                }
            }
            return true;
        }
    }
```

7、编辑距离（72）（lintcode119）

72. 编辑距离

难度 困难 986 收藏 分享 切换为英文 关注 反馈

给你两个单词 *word1* 和 *word2*，请你计算出将 *word1* 转换成 *word2* 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

示例 1：

```
输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2：

```
输入：word1 = "intention", word2 = "execution"
输出：5
解释：
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

略

https://leetcode-cn.com/problems/edit-distance/solution/dong-tai-gui-hua-java-by-liweiwei1419/

这个问题是使用「动态规划」解决的的经典问题，一般是作为例题进行学习的，因此就直接给出「动态规划」方法的解释，主要在理解「动态规划」是如何定义「状态」和推导「状态转移方程」。

思路：

- 「动态规划」告诉我们可以「自底向上」去考虑一个问题，思路是：**先想这个问题最开始是什么情况，这个问题是两个字符串都为空字符的时候，然后逐个地，一个字符一个字符加上去，在加字符的过程中考虑「状态转移」**；
- 由于要考虑空字符，因此状态空间要多设置一行、多设置一列。

方法：动态规划

第 1 步：定义状态

状态：`dp[i][j]` 表示将 `word1[0, i)` 转换成为 `word2[0, j)` 的方案数。

思考状态的方法：1、题目问什么就将什么定义为状态；2、「状态转移方程」怎么好推导，就怎么定义状态；3、根据经验和问题的特点（只有多做题了）。

说明：**由于要考虑空字符**，这里的下标 `i` 不包括 `word[i]`，同理下标 `j` 不包括 `word[j]`，从行数和列数多设置一行、一列也可以来理解这一点，也就是状态的下标 `i` 和 `j` 和字符的下标 `i`、`j` 有一个位置的偏差。

下文有些地方是 `)` 有些地方是 `]` 。如果造成疑惑，可以暂时不管，并不影响理解思想。

第 2 步：思考状态转移方程

状态转移方程通常是在做分类讨论，而分类讨论的过程，常常利用了这个问题的「最优子结构」。

情况 1： `word1[i] == word2[j]`

如果 `word1[i] == word2[j]` 成立，则将 `word1[0, i)` 转换成为 `word2[0, j)` 的方案数就等于 将 `word1[0, i - 1)` 转换成为 `word2[0, j - 1)` 的方案数，即：

```
dp[i + 1][j + 1] = dp[i][j];
```

注意：这种情况，方案数最少，后面三种情况可以不用再讨论。

- 直觉：后面的情况要考虑的字符更多，并且还会增加 1 步操作；
- 实验：从实际代码运行的结果上看，可以通过测评，并且的确是后面的情况不会使得结果更少，大家还可以使用 `aaaa` 和 `aaaaa` 这样的情况去验证。

对于这个结论 @1dUv8logjC 这位朋友给出了使用「数学归纳法」的证明，欢迎大家在评论区围观，感谢这位朋友。

也可以参考 @gelthin 这位朋友给出的题解：《证明: 当 w1[i] == w2[j]时，必有 DP[i+1][j+1] = DP[i][j]》。

更多硬核内容就在公众号「yes 的练级攻略」

情况 2： `word1[i] != word2[j]`

（下面的文字看过去文绉绉的，意会就可以了，建议尝试自己举例去理解。）

如果 `word1[i] != word2[j]` ，则将 `word1[0, i)` 转换成为 `word2[0, j)` 的方案数就等于下面 3 种情况的最少操作数（「最优子结构」）：

1. 考虑修改 `word1[i]` 成为 `word2[j]` ；

此时 `dp[i + 1][j + 1] = dp[i][j] + 1` ，这里的 `1` 代表了将 `word1[i]` 替换成为 `word2[j]` 这一步操作。

2. 考虑将 `word1[0, i]` 的最后一个字符删除；

此时 `word1[0, i - 1]` 到 `word2[0, j]` 的最少操作数 $+1$，就是这种方案数的最少操作数，即： `dp[i + 1][j + 1] = dp[i][j + 1] + 1` ，这里的 `1` 代表了 `word1[0, i]` 的最后一个字符删除这一步操作。

3. 考虑将 `word1[0, i]` 的末尾添加一个字符使得 `word1[i + 1] == word2[j]` ；

此时考虑方案的时候，由于 `word1[i + 1] == word2[j]` ，状态转移就不应该考虑 `word2[j]` ，因此 `word1[0, i]` 到 `word2[0, j - 1]` 的最少操作数 $+1$，就是这种方案数的最少操作数，即： `dp[i + 1][j + 1] = dp[i + 1][j] + 1` ，这里的 `1` 代表了将 `word1[0, i]` 的末尾添加一个字符使得 `word1[i + 1] == word2[j]` 。（注意：可以考虑一下为什么得先讨论 `word1[i] == word2[j]` 的情况。）

在这 3 种操作中取最小值。

```
dp[i + 1][j + 1] = min(dp[i][j], dp[i][j + 1], dp[i + 1][j]) + 1
```

第 3 步：初始化

- 从一个字符串变成空字符串，非空字符串的长度就是编辑距离；
- 以下代码其实就是在填表格的第 0 行、第 0 列。

```
for (int i = 0; i <= len1; i++) {
    dp[i][0] = i;
}

for (int j = 0; j <= len2; j++) {
    dp[0][j] = j;
}
```

第 4 步： 思考输出

输出： `dp[len1][len2]` 符合语义，即 `word1[0, len)` 转换成 `word2[0, len2)` 的最小操作数。（这里 `)` 表示开区间。）

第 5 步： 思考状态压缩

我们看一下「状态转移方程」：

- 如果末尾字符相等，就「抄」左上角单元格的值；
- 如果末尾字符不相等，就从「正上方」、「左边」、「左上角」三个单元格的值中选出最小的 + 1。

因此，初看可以使用「滚动数组」，更极端一点，用 2×2 表格就可以完成操作。但是真正去做「状态压缩」的时候，由于初始化的原因，发现没有那么容易，在这里不做「状态压缩」。（事实上可以压缩，但是只要是压缩状态，必然给编码造成一定困难，并且破坏代码可读性，根据情况做吧，个人觉得在空间紧张的情况下必须压缩空间，其余不必。）

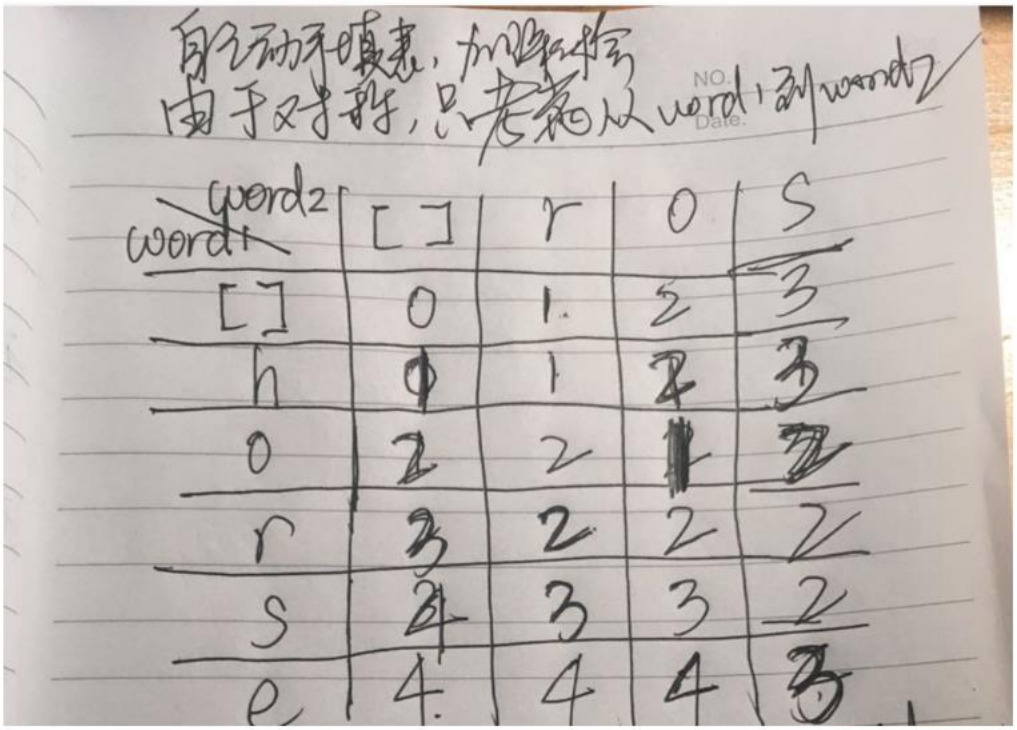
建议：自己动手填表格，加深体会「动态规划」的「自底向上」**填表格**解决问题的方法，它不是直接针对问题求解，而是从一个最小的情况开始，逐步递推，并且记录中间过程，得到最终答案的过程。

说明：手动填一下这张表格，体会：

- 在末尾的两个字符相等的情况下，直接把左上角的值抄下来；

- 在末尾的两个字符不相等的情况下，「正上方」、「左边」、「左上角」三个单元格的值中选出最小的 + 1

手填的时候一不小心很容易出错，然后去比对正确的表格，看看是哪里出错了。



输出：

```
[0, 1, 2, 3]
[1, 1, 2, 3]
[2, 2, 1, 2]
[3, 2, 2, 2]
[4, 3, 3, 2]
[5, 4, 4, 3]
```

发现有一个地方填错了，然后去思考原因。

类似问题

这其实是一类称之为区间型 dp 的问题，主要特征是：思考的时候可以先将大区间拆分成小区间，求解的时候由小区间的解得到大区间的解。大家可以做一下这些问题以加深体会做这一类问题的思路和技巧，体会可以使用「动态规划」求解的问题的三个特征：

- 重叠子问题；
- 最优子结构；
- 无后效性。

「力扣」上比较典型的区间型 dp 问题有：

- 「力扣」第 5 题：最长回文子串；
- 「力扣」第 1143 题：最长公共子串；
- 「力扣」第 877 题：石子游戏；
- 「力扣」第 10 题：正则表达式匹配。

72 具体代码

```
import java.util.Arrays;
public class Solution {
    public int minDistance(String word1, String word2) {
        // 由于 word1.charAt(i) 操作会去检查下标是否越界，因此
        // 在 Java 里，将字符串转换成字符数组是常见额操作
        char[] word1Array = word1.toCharArray();
        char[] word2Array = word2.toCharArray();

        int len1 = word1Array.length;
        int len2 = word2Array.length;

        // 多开一行一列是为了保存边界条件，即字符长度为 0 的情况，这一点在字符串的动态规划问题中比较常见
        int[][] dp = new int[len1 + 1][len2 + 1];

        // 初始化：当 word 2 长度为 0 时，将 word1 的全部删除
        for (int i = 1; i <= len1; i++) {
            dp[i][0] = i;
        }
        // 当 word1 长度为 0 时，就插入所有 word2 的字符
        for (int j = 1; j <= len2; j++) {
            dp[0][j] = j;
        }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        // 注意：填写 dp 数组的时候，由于初始化多设置了一行一列，横、纵坐标有个偏移
        for (int i = 0; i < len1; i++) {
            for (int j = 0; j < len2; j++) {
                // 这是最佳情况
                if (word1Array[i] == word2Array[j]) {
                    dp[i + 1][j + 1] = dp[i][j];
                    continue;
                }

                // 否则在以下三种情况中选出步骤最少的，这是「动态规划」的「最优子结构」
                // 1、在下标 i 处插入一个字符
                int insert = dp[i + 1][j] + 1;
                // 2、替换一个字符
                int replace = dp[i][j] + 1;
                // 3、删除一个字符
                int delete = dp[i][j + 1] + 1;
                dp[i + 1][j + 1] = Math.min(Math.min(insert, replace), delete);

            }
        }

        // 打印状态表格进行调试
//        for (int i = 0; i <=len1; i++) {
//            System.out.println(Arrays.toString(dp[i]));
//        }
        return dp[len1][len2];
    }

    public static void main(String[] args) {
        String word1 = "horse";
        String word2 = "ros";

        Solution solution = new Solution();
        int res = solution.minDistance(word1, word2);
        System.out.println(res);
    }
}
```

8、最大子矩阵（面试题 17.24）

面试题 17.24. 最大子矩阵

难度 困难 21 收藏 分享 切换为英文 关注 反馈

给定一个正整数和负整数组成的 N × M 矩阵，编写代码找出元素总和最大的子矩阵。

返回一个数组 [r1, c1, r2, c2]，其中 r1, c1 分别代表子矩阵左上角的行号和列号，r2, c2 分别代表右下角的行号和列号。若有多个满足条件的子矩阵，返回任意一个均可。

注意：本题相对书上原题稍作改动

示例：

```
输入：
[
   [-1,0],
   [0,-1]
]
输出：[0,1,0,1]
解释：输入中标粗的元素即为输出所表示的矩阵
```

说明：

- 1 <= matrix.length, matrix[0].length <= 200

https://leetcode-cn.com/problems/max-submatrix-lcci/solution/chang-gui-jie-fa-yi-yu-li-jie-by-tago-tech/

```
class Solution {
    /**
     * 最大累加和的子矩阵
     * 给定一个正整数和负整数组成的 N × M 矩阵，编写代码找出元素总和最大的子矩阵。
     *
     * 返回一个数组 [r1, c1, r2, c2]，
     * 其中 r1, c1 分别代表子矩阵左上角的行号和列号，
     * r2, c2 分别代表右下角的行号和列号。若有多个满足条件的子矩阵，返回任意一个均可。
     * */
    public int[] getMaxMatrix(int[][] matrix) {
        int n = matrix.length , m = matrix[0].length;
        int[] target = {-1,-1,-1,-1};
        int[] sum = new int[m];
        int maxOfGlobal = Integer.MIN_VALUE;
        for (int top = 0;top < n;top++) {
            for (int i = 0;i < m;i++) {
                sum[i] = 0;
            }
            for (int bottom = top;bottom < n;bottom++) {
                for (int i = 0;i < m;i++) {
                    sum[i] += matrix[bottom][i];
                }
                //计算出[top,bottom]的当前最大值
                int cur = 0, l = 0, r = -1,sl = 0,max = Integer.MIN_VALUE;
                for (int i = 0;i < m;i++) {
                    cur += sum[i];
                    if (cur > max) {
                        l = sl;
                        r = i;
```

```
                    max = cur;
                }
                if (cur < 0) {
                    cur = 0;
                    sl = i + 1;
                }
            }
            if (max > maxOfGlobal) {
                target[0] = top;
                target[1] = l;
                target[2] = bottom;
                target[3] = r;
                maxOfGlobal = max;
            }
        }
    }
    return target;
  }
}
```

9、判断子序列（392）

392. 判断子序列

难度 简单 262 收藏 分享 切换为英文 关注 反馈

给定字符串 **s** 和 **t** ，判断 **s** 是否为 **t** 的子序列。

你可以认为 **s** 和 **t** 中仅包含英文小写字母。字符串 **t** 可能会很长（长度 ~= 500,000），而 **s** 是个短字符串（长度 <=100）。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，`"ace"` 是 `"abcde"` 的一个子序列，而 `"aec"` 不是）。

示例 1:
s = `"abc"`, **t** = `"ahbgdc"`

返回 `true`.

示例 2:
s = `"axc"`, **t** = `"ahbgdc"`

返回 `false`.

后续挑战：

如果有大量输入的 S，称作S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？

致谢:

特别感谢 @pbrother 添加此问题并且创建所有测试用例。

通过次数 64,939 | 提交次数 129,376

https://leetcode-cn.com/problems/is-subsequence/solution/pan-duan-zi-xu-lie-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：双指针

思路及算法

本题询问的是，s 是否是 t 的子序列，因此只要能找到任意一种 s 在 t 中出现的方式，即可认为 s 是 t 的子序列。

而当我们从前往后匹配，可以发现每次贪心地匹配靠前的字符是最优决策。

> 假定当前需要匹配字符 c，而字符 c 在 t 中的位置 x_1 和 x_2 出现（$x_1 < x_2$），那么贪心取 x_1 是最优解，因为 x_2 后面能取到的字符，x_1 也都能取到，并且通过 x_1 与 x_2 之间的可选字符，更有希望能匹配成功。

这样，我们初始化两个指针 i 和 j，分别指向 s 和 t 的初始位置。每次贪心地匹配，匹配成功则 i 和 j 同时右移，匹配 s 的下一个位置，匹配失败则 j 右移，i 不变，尝试用 t 的下一个字符匹配 s。

最终如果 i 移动到 s 的末尾，就说明 s 是 t 的子序列。

复杂度分析

- 时间复杂度：$O(n+m)$，其中 n 为 s 的长度，m 为 t 的长度。每次无论是匹配成功还是失败，都有至少一个指针发生右移，两指针能够位移的总距离为 $n+m$。
- 空间复杂度：$O(1)$。

方法 1：双指针

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int len1 = s.length(), len2 = t.length();
        int i = 0, j = 0;
        while (i < len1 && j < len2) {
            if (s.charAt(i) == t.charAt(j)) {
                i++;
            }
            j++;
        }
        return i == len1;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

方法二：动态规划

思路及算法

考虑前面的双指针的做法，我们注意到我们有大量的时间用于在 t 中找到下一个匹配字符。

这样我们可以预处理出对于 t 的每一个位置，从该位置开始往后每一个字符第一次出现的位置。

我们可以使用动态规划的方法实现预处理，令 $f[i][j]$ 表示字符串 t 中从位置 i 开始往后字符 j 第一次出现的位置。在进行状态转移时，如果 t 中位置 i 的字符就是 j，那么 $f[i][j]=i$，否则 j 出现在位置 $i+1$ 开始往后，即 $f[i][j]=f[i+1][j]$，因此我们要倒过来进行动态规划，从后往前枚举 i。

这样我们可以写出状态转移方程：

$$f[i][j]=\begin{cases} i, & t[i]=j \\ f[i+1][j], & t[i]\neq j \end{cases}$$

假定下标从 0 开始，那么 $f[i][j]$ 中有 $0\leq i\leq m-1$，对于边界状态 $f[m-1][..]$，我们置 $f[m][..]$ 为 m，让 $f[m-1][..]$ 正常进行转移。这样如果 $f[i][j]=m$，则表示从位置 i 开始往后不存在字符 j。

这样，我们可以利用 f 数组，每次 $O(1)$ 地跳转到下一个位置，直到位置变为 m 或 s 中的每一个字符都匹配成功。

> 同时我们注意到，该解法中对 t 的处理与 s 无关，且预处理完成后，可以利用预处理数组的信息，线性地算出任意一个字符串 s 是否为 t 的子串。这样我们就可以解决「后续挑战」啦。

复杂度分析

- 时间复杂度：$O(m\times|\Sigma|+n)$，其中 n 为 s 的长度，m 为 t 的长度，Σ 为字符集，在本题中字符串只包含小写字母，$|\Sigma|=26$。预处理时间复杂度 $O(m)$，判断子序列时间复杂度 $O(n)$。
  - 如果是计算 k 个平均长度为 n 的字符串是否为 t 的子序列，则时间复杂度为 $O(m\times|\Sigma|+k\times n)$。
- 空间复杂度：$O(m\times|\Sigma|)$，为动态规划数组的开销。

方法 2：动规【适用于需要依次检查它们是否为 T 的子序列】

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int len1 = s.length(), len2 = t.length();
        int[][] dp = new int[len2+1][26];
        for (int j = 0; j < 26; j++){
            dp[len2][j] = len2;
        }
        for (int i = len2 - 1; i >= 0; --i) {
            for(int j = 0; j < 26; ++j){
                if (t.charAt(i) == j + 'a'){
                    dp[i][j] = i;
                } else {
                    dp[i][j] = dp[i+1][j];
                }
            }
        }
        int add = 0;
        for (int k = 0; k < len1; ++k){
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                if (dp[add][s.charAt(k) - 'a'] == len2){
                    return false;
                }
                add = dp[add][s.charAt(k) - 'a'] + 1;
            }
            return true;
        }
    }
```

代码仔 2020-07-27

动态 规划的，我把按他的思想，拆分一下结构吧。在项目工程中可能就能感觉到好处。 也就是题目描述到的：如果有大量输入的 S，称作S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？ 把T的处理拆分出来，一次初始化，以后就不需要再一个一个的字符去匹配了。

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        if (s == null || s.length() == 0) return true;
        if (t == null || t.length() == 0) return false;
        int[][] dp = makeDp(t);
        return match(s, dp);
    }
    private boolean match(String s, int[][] dp) {
        int sLen = s.length();
        int index = 0;
        for (int i = 0; i < sLen; i++) {
            if ((index = dp[index][s.charAt(i) - 'a']) == -1) return false;
            index++;
        }
        return true;
    }
    private int[][] makeDp(String t) {
        int tLen = t.length();
        int[][] dp = new int[tLen + 1][26];
        for (int i = 0; i < 26; i++) {
            dp[tLen][i] = -1;
        }
        for (int i = tLen - 1; i >= 0; i--) {
            for (int j = 0; j < 26; j++) {
                if (t.charAt(i) == j + 'a') {
                    dp[i][j] = i;
                } else {
                    dp[i][j] = dp[i + 1][j];
                }
            }
        }
```

```
        return dp;
    }
}
```

方法 3：内置函数

https://leetcode-cn.com/problems/is-subsequence/solution/javade-2chong-jie-fa-by-sdwwld/

java 中 String 类有这样一个方法 public int indexOf(int ch, int fromIndex) ，他表示的是在字符串中是否存在一个字符 ch，并且是从字符串的下标 fromIndex 开始查找的。我们要做的是在 t 字符串中查找 s 中的每一个字符，如果没查到，直接返回 false。如果查到，就从 t 的下一个位置继续开始查。

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int index = -1;
        for (char c : s.toCharArray()) {
            //index 表示上一次查找的位置(第一次查找的时候为-1)，所以这里要从 t 的下标(index+1)开始查找
            index = t.indexOf(c, index + 1);
            //没找到，返回 false
            if (index == -1)
                return false;
        }
        return true;
    }
}
```

10、最长公共子序列（1143）

1143. 最长公共子序列

难度 中等 168 收藏 分享 切换为英文 关注 反馈

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

一个字符串的 *子序列* 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。
例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

示例 1:

输入： text1 = "abcde", text2 = "ace"
输出： 3
解释： 最长公共子序列是 "ace"，它的长度为 3。

示例 2:

输入： text1 = "abc", text2 = "abc"
输出： 3
解释： 最长公共子序列是 "abc"，它的长度为 3。

示例 3:

输入： text1 = "abc", text2 = "def"
输出： 0
解释： 两个字符串没有公共子序列，返回 0。

https://leetcode-cn.com/problems/longest-common-subsequence/solution/marvelzhong-deng-de-xue-xi-bi-ji-1143-by-tyanyon-2/

解题思路

最长公共子序列（Longest Common Subsequence，LCS）是指：
给定两个字符串（数列），求一个子列，使得这个子列是两个给定字符串的最长公共部分。例如：

```
a b c d e f
b d a f
```

以上两个字符串的LCS就是b d f，接下来用动态规划求解。

形如最大子列和、最长上升子列等动规问题，我们都有一个dp[]数组来表示问题的状态。
例如，在最大子列和问题中，dp[i]表示以nums[i]作为序列末尾时能得到的最大子列和；最长上升子列问题中，dp[i]表示以nums[i]作为序列末尾时能得到的最长上升子列长度。**dp[i]就是问题的状态**，此外，我们会从题目中获取**边界信息**，例如`dp[0] = nums[0]`就是最大子列和问题的边界。然后我们需要写出关键的**状态转移方程**。之后就可以**从边界出发，通过状态转移方程求解状态了**。以上是解决动规问题的一般思路。

而本题我们依然会使用一个dp数组表示问题的状态，只不过这次我们用的是二维数组dp[][]。带着疑惑往下看。
我们用dp[i][j]表示第一个字符串的第1至i号位字符部分与第二个字符串的第1至j号位字符部分的LCS长度，即问题的状态。这里我们用1作为字符串第一个字符的下标。以上面给出的两个字符串为例，dp[4][2]的值就是字符串"a b c d"与字符串"b d"的LCS长度，显然为2。理解了dp数组含义后，其实我们要求解的状态就是**dp[len1][len2]**，len1为第一个字符串的长度，len2为第二个字符串的长度。

同样的，我们现在知道了状态dp[i][j]是什么，现在该获取边界信息了，即题目隐藏的或给定的一些已知信息。
显然$dp[i][0] = 0(i = 0, 1, 2, 3, ..., len1)$;
$dp[0][j] = 0(j = 0, 1, 2, 3, ..., len2)$。
由于上面提到了，字符的下标从1开始，因此0代表空串，字符串和空串的LCS长度自然为0。

边界信息有了，可以尝试一下写状态转移方程。
为了便于理解，我们先把dp[][]二维数组画出来。并且根据刚才的边界信息，我们可以在表格的特定位置填上0，如下所示。

			b	d	a	f
		0	1	2	3	4
	0	0	0	0	0	0
a	1	0				
b	2	0				
c	3	0				
d	4	0				
e	5	0				
f	6	0				

接下来，就是完善这个表格。再看一个例子：
dp[4][2]，其中，t1[4] == t2[2]，都是字符d，说明此时LCS长度可以增加1，即`dp[4][2] = dp[3][1] + 1;`。
dp[4][3]，其中，t1[4] != t2[3]，即字符串"a b c d"和"b d a"，不难得到，这两个字符串的LCS长度应该是字符串"a b c"和"b d a"的LCS长度与字符串"a b c d"和"b d"的LCS长度这两个长度的较大者。因为d != a，所以这一对字符不会增加LCS长度，所以LCS长度应该继承dp[3][3]和dp[4][2]的较大者。

按照这个思路完善表格如下：

			b	d	a	f
		0	1	2	3	4
	0	0	0	0	0	0
a	1	0	0	0	1	1
b	2	0	1	1	1	1
c	3	0	1	1	1	1
d	4	0	1	2	2	2
e	5	0	1	2	2	2
f	6	0	1	2	2	3

这样，dp[6][4]为3就是两个字符串的LCS长度。

此时可以得到状态转移方程：
$t1[i] == t2[j]$时，$dp[i][j] = dp[i-1][j-1] + 1$;
$t1[i] != t2[j]$时，$dp[i][j] = max(dp[i-1][j], dp[i][j-1])$。

时间复杂度：O(nm)，n、m分别为两个字符串的长度。
空间复杂度：O(nm)。

方法 1：二维

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        char[] t1 = text1.toCharArray();
        char[] t2 = text2.toCharArray();
        int len1 = t1.length;
        int len2 = t2.length;
        int[][] dp = new int[len1+1][len2+1];
        for(int i = 1; i <= len1; i++)
        {
            for(int j = 1; j <= len2; j++)
                dp[i][j] = t1[i-1] == t2[j-1] ? (dp[i-1][j-1] + 1) : Math.max(d
p[i-1][j], dp[i][j-1]);
        }
        return dp[len1][len2];
    }
}
```

优化

上述做法中，通过观察状态转移方程：
$t1[i] == t2[j]$时，$dp[i][j] = dp[i-1][j-1] + 1$;
$t1[i] != t2[j]$时，$dp[i][j] = max(dp[i-1][j], dp[i][j-1])$。
可以发现，计算每个dp[i][j]时（第i行第j列），我们都只用到了第i行和第i-1行、第j列和第j-1列。

i-1, j-1	i-1, j
i, j-1	i, j

如图所示，计算dp[i][j]只用到了其左方、左上方和上方的数据。从这个点出发可以将空间复杂度优化。
具体做法：逐列地计算dp[][]数组的值，这样可以把二维数组变为一维数组，只存储一列数据即可。
假设某个时刻，第j-1列的所有dp值都已计算好，存储于dp数组中，此时要计算第j列的dp值。例如要计算dp[i][j]，即此时的dp[i]，我们已经有了它的左方数据，就是dp[i]（即第j-1列第i行数据），记为temp，然后我们需要它的左上方数据，这个可以在上一轮用上一轮的temp赋值给last，last是每轮计算的对象的左上方数据，最后需要它的上方数据，即dp[i-1]，上一轮已经算好了，这样计算dp[i]所需要的信息都齐全了。
这样逐列的计算dp[]，同时更新LCS长度，最终的结果与用二维数组存储是一样的。具体细节参考下面代码。

时间复杂度：O(nm)。
空间复杂度：O(n)。

方法 2：一维

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        char[] t1 = text1.toCharArray();
        char[] t2 = text2.toCharArray();
        int[] dp = new int[t1.length + 1];
        int ans = 0;
        for(int j = 1; j <= t2.length; j++)
        {
            int last = 0;
            for(int i = 1; i <= t1.length; i++)
            {
                int temp = dp[i];
                dp[i] = t1[i-1] == t2[j-1] ? (last + 1) : Math.max(dp[i-1], dp[
i]);
                ans = Math.max(ans, dp[i]);
                last = temp;
            }
        }
        return ans;
    }
}
```

11、俄罗斯套娃信封问题（354）（lintcode 602）【贪心+二分】

更多硬核内容就在公众号「yes 的练级攻略」

354. 俄罗斯套娃信封问题

难度 困难 170 收藏 分享 切换为英文 关注 反馈

给定一些标记了宽度和高度的信封，宽度和高度以整数对形式 (w, h) 出现。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算最多能有多少个信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

说明:
不允许旋转信封。

示例:

```
输入: envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出: 3
解释: 最多信封的个数为 3, 组合为: [2,3] => [5,4] => [6,7]。
```

602. 俄罗斯套娃信封 中文

给一定数量的信封，带有整数对 (w, h) 分别代表信封宽度和高度。一个信封的宽高均大于另一个信封时可以放下另一个信封。求最大的信封嵌套层数。

样例
样例 1:

```
输入: [[5,4],[6,4],[6,7],[2,3]]
输出: 3
解释:
最大的信封嵌套层数是 3 ([2,3] => [5,4] => [6,7])。
```

样例 2:

```
输入: [[4,5],[4,6],[6,7],[2,3],[1,1]]
输出: 4
解释:
最大的信封嵌套层数是 4 ([1,1] => [2,3] => [4,5] / [4,6] => [6,7])。
```

https://leetcode-cn.com/problems/russian-doll-envelopes/solution/tan-xin-suan-fa-er-fen-cha-zhao-python-dai-ma-java/

贪心法 + 二分查找

如果按照之前的分析，原问题等价于“将信封数组按照宽“升序”排序以后，信封数组的高的“最长上升子序列”的长度就是题目所求”，在提交的时候，就会发现出错了。查看错误的测试用例不难发现问题所在，这里我们举例说明问题出在哪里。

事实上，之前我们也已经提到过了：题目要求小信封可以装入大信封的条件是“小信封宽和高分别严格小于大信封的宽和高”。

当“宽”（下图粉红色）相等的时候，如果“高度”（下图蓝色）出现“上升子序列”就有可能引入错误的结果。例如：

此时，左边粉红色宽度按照升序排序，但是第 3 个和第 4 个，在宽相等的时候，[6, 7] 和 [6, 8] 会被算法同时选取，就违背了题意（小信封的宽度和高度分别“严格”小于大信封的宽度和高度）。

解决的办法很容易想到，那就是：**在宽度相等的时候，让高度不能出现“上升的子序列”。即首先按照宽度“升序排序”，在宽度相等的时候，按照高度“降序排序”**，这样就能保证宽度相等的信封，最多只能选一个，这种策略保证了结果的正确性。这就是最开始的说明中的第 1 点缺少的正确性的另一半。

与最长上升子序列类似

```
import java.util.Arrays;
import java.util.Comparator;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
public class Solution {

    public int maxEnvelopes(int[][] envelopes) {

        int len = envelopes.length;
        if (len < 2) {
            return len;
        }
//可以用这个替代下面的
//Arrays.sort(envelopes, (o1, o2) -> o1[0] == o2[0] ? o2[1] - o1[1] : o
1[0] - o2[0]);
        Arrays.sort(envelopes, new Comparator<int[]>() {
            @Override
            public int compare(int[] envelope1, int[] envelope2) {
                if (envelope1[0] != envelope2[0]) {
                    return envelope1[0] - envelope2[0];
                }
                return envelope2[1] - envelope1[1];
            }
        });


        int[] tail = new int[len];
        tail[0] = envelopes[0][1];

        // end 表示有序数组 tail 的最后一个已经赋值元素的索引
        int end = 0;

        for (int i = 1; i < len; i++) {
            int target = envelopes[i][1];

            if (target > tail[end]) {
                end++;
                tail[end] = target;
            } else {
                int left = 0;
                int right = end;

                while (left < right) {
                    int mid = (left + right) >>> 1;
                    if (tail[mid] < target) {
                        left = mid + 1;
                    } else {
                        right = mid;
                    }
```

```
                }
                tail[left] = target;
            }
        }
        return end + 1;
    }
}
```

12、摇动序列（376）【**返回**可以变成摇摆序列的最长子序列的**长度**】

376. 摆动序列

难度 中等 217 收藏 分享 切换为英文 关注 反馈

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为**摆动序列**。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如，[1, 7, 4, 9, 2, 5] 是一个摆动序列，因为差值 (6, -3, 5, -7, 3) 是正负交替出现的。相反，[1, 4, 7, 2, 5] 和 [1, 7, 4, 5, 5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。 通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

示例 1:

输入: [1,7,4,9,2,5]
输出: 6
解释: 整个序列均为摆动序列。

示例 2:

输入: [1,17,5,10,13,15,10,5,16,8]
输出: 7
解释: 这个序列包含几个长度为 7 摆动序列，其中一个可为[1,17,10,13,10,16,8]。

示例 3:

输入: [1,2,3,4,5,6,7,8,9]
输出: 2

进阶:
你能否用 O(*n*) 时间复杂度完成此题?

https://leetcode-cn.com/problems/wiggle-subsequence/solution/bai-dong-xu-lie-by-leetcode/

题解的定义应该是这样的： up[i]是前 i 个元素中，摆动序列以上升元素结尾的最长子序列长度； down[i]是前 i 个元素中，摆动序列以下降元素结尾的最长子序列长度。 例如，对于序列【3, 2, 4, 5, 7】，有： up[] = 【1, 1, 2, 3, 3】 down[] = 【1, 2, 2, 2, 2】 以第三个元素为例（为了便于描述，数组下标从 1 开始），down[3] = 2，也就是子序列【3, 2, 4】中的摆动序列以下降元素结尾的最长子序列的长度是 2，即【3, 2】，4 并不在该最长摆动子序列中。有了上面概念，那么就容易得出题解中描述的状态转移方程了：

1、对于 nums[i] > nums[i-1]，假设 up[i-1] = x，有以下事实：如果前 i-1 个元素中以下降元素结尾的最长摆动序列以元素 i-1 结尾，显然 up[i] = down[i-1] + 1；如果前 i-1 个元素中以下降元素结尾的最长摆动子序列不以元素 i-1 结尾，那么必然存在元素 nums[j]（0 <= j < i-1），有 nums[j] < nums[i-1]，（不取等号的原因是，如果 nums[j] = nums[i-1]，可以选择元素 i-1 而不选择元素 j），必然 nums[j] < nums[i]，那么也可以得出 up[i] = down[j] + 1 = down[i-1] + 1；

2、对于 nums[i] < nums[i-1]，与上面相反论证便是了；

3、对于 nums[i] == nums[i-1]，正如 1 所说，如果存在连续两个元素相同，那么以最后一个元素结尾，而忽略前面一个元素，up 和 down 都不需要改变。

复杂度分析

- 时间复杂度：$O(n)$。只需要遍历数组一遍。
- 空间复杂度：$O(n)$。 dp需要两个相同长度的数组。

更多硬核内容就在公众号「yes 的练级攻略」

方法 1：一维动规数组

```
class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2) return nums.length;
        int n = nums.length;
        int[] up = new int[n], down = new int[n];
        up[0] = down[0] = 1;
        for (int i = 1; i < n; ++i){
            if (nums[i] > nums[i-1]){ //意味着这里在摆动上升，前一个数字肯定处于下降的位置
                up[i] = down[i-1] + 1;
                down[i] = down[i-1];
            } else if (nums[i] < nums[i-1]){ //意味着这里在摆动下降，前一个数字肯定处于上升的位置
                down[i] = up[i-1] + 1;
                up[i] = up[i-1];
            } else { //意味着这个元素不会改变任何东西因为它没有摆动。
                down[i] = down[i-1];
                up[i] = up[i-1];
            }
        }
        return Math.max(down[n - 1], up[n - 1]);
    }
}
```

方法 4： 空间优化的动态规划

算法

这个方法与方法 3 一样。但我们观察可以发现， DP 过程中更新 $up[i]$ 和 $down[i]$ ，其实只需要 $up[i-1]$ 和 $down[i-1]$ 。因此，我们可以通过只记录最后一个元素的值而不使用数组来节省空间。

复杂度分析

- 时间复杂度： $O(n)$ 。仅遍历数组一次。
- 空间复杂度： $O(1)$ 。只使用了常数级别的额外空间。

方法 2：常数级动规【面试推荐】

```
class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2) return nums.length;
        int n = nums.length;
        int up = 1, down = 1;
        for (int i = 1; i < n; ++i){
            if (nums[i] > nums[i-1]){ //意味着这里在摆动上升，前一个数字肯定处于下降的位置
                up = down + 1;
```

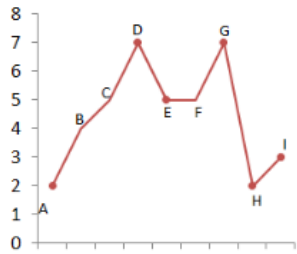
```
            } else if (nums[i] < nums[i-1]){ //意味着这里在摆动下降，前一个数字肯定处于上升的位置
                down = up + 1;
            }
        }
        return Math.max(up, down);
    }
}
```

方法 5：贪心

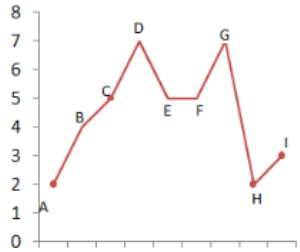
算法

我们其实不需要 dp 来解决这个问题。这个问题等价于找数组中交替的最大最小值。因此，如果我们选择中间的数字作为摆动序列的一部分，只会导致序列小于等于只选连续的最大最小元素。

下图可以更好地说明此方法：



Wiggle Subsequence (choosing D as 2nd point): ADEGHI



Wiggle Subsequence (choosing C as the 2nd point): ACHI

从上图中，我们可以看到，如果我们选择 C 而不是 D 作为摆动序列的第二个点，我们无法将点 E 也包括进最终序列。因此，我们无法得到最长摆动序列。

因此，为了解决这个问题，我们维护一个变量 prevdiff，它的作用是只是当前数字的序列是上升还是下降的。如果 $\text{prevdiff} > 0$，那么表示目前是上升序列，我们需要找一个下降的元素。因此，我们更新已找到的序列长度 diff（$nums[i] - nums[i-1]$）为负数。类似的，如果 $\text{prevdiff} < 0$，我们就更新 diff（$nums[i] - nums[i-1]$）为正数。

当整个数组都被遍历后，我们得到了最终的结果，它就是最长摆动子序列的长度。

复杂度分析

- 时间复杂度：$O(n)$。我们需要遍历数组一次。
- 空间复杂度：$O(1)$。不需要使用额外的空间。

方法 3：贪心

```
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
```

```
        int prevdiff = nums[1] - nums[0];
        int count = prevdiff != 0 ? 2 : 1;
        for (int i = 2; i < nums.length; i++) {
            int diff = nums[i] - nums[i - 1];
            if ((diff > 0 && prevdiff <= 0) || (diff < 0 && prevdiff >=
0)) {
                count++;
                prevdiff = diff;
            }
        }
        return count;
    }
}
```

13、买卖股票的最佳时机（121）【股票的最大利润（剑指 63）】【最多 1 笔交易】

121. 买卖股票的最佳时机

难度 简单 1012 收藏 分享 切换为英文 关注 反馈

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-iv/solution/yi-ge-tong-yong-fang-fa-tuan-mie-6-dao-gu-piao-w-5/

这 6 道股票买卖问题是有共性的，我们通过对第四题（限制最大交易次数为 k）的分析一道一道解决。因为第四题是一个最泛化的形式，其他的问题都是这个形式的简化。

第一题是只进行一次交易，相当于 k = 1；第二题是不限交易次数，相当于 k = +infinity（正无穷）；第三题是只进行 2 次交易，相当于 k = 2；剩下两道也是不限次数，但是加了交易「冷冻期」和「手续费」的额外条件，其实就是第二题的变种，都很容易处理。

```
class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) return 0;
        int dp0 = 0, dp1 = -prices[0];
        for (int i = 0; i < prices.length; i++){
```

```
            dp0 = Math.max(dp0, dp1 + prices[i]);
            dp1 = Math.max(dp1, -prices[i]);
        }
        return dp0;
    }
}
```

14、买票股票的最佳时机 2（122）【尽可能更多次的交易】

122. 买卖股票的最佳时机 II

难度 简单 742 收藏 分享 切换为英文 关注 反馈

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

输入： [7,1,5,3,6,4]
输出： 7
解释： 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3 。

示例 2:

输入： [1,2,3,4,5]
输出： 4
解释： 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

示例 3:

输入： [7,6,4,3,1]
输出： 0
解释： 在这种情况下，没有交易完成，所以最大利润为 0。

第二题，k = +infinity

如果 k 为正无穷，那么就可以认为 k 和 k - 1 是一样的。可以这样改写框架：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
            = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])

我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态了：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
```

方法 1：动态规划代码

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int dp_i_0 = 0, dp_i_1 = -prices[0];
        for (int i = 0; i < n; i++) {
```

```
            int temp = dp_i_0;
            dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
            dp_i_1 = Math.max(dp_i_1, temp - prices[i]);
        }
        return dp_i_0;
    }
}
```

这题的最优解是采用贪心算法

解题思路：

股票买卖策略：

- **单独交易日：** 设今天价格 p_1、明天价格 p_2，则今天买入、明天卖出可赚取金额 $p_2 - p_1$ （负值代表亏损）。
- **连续上涨交易日：** 设此上涨交易日股票价格分别为 $p_1, p_2, ..., p_n$，则第一天买最后一天卖收益最大，即 $p_n - p_1$ ；等价于每天都买卖，即 $p_n - p_1 = (p_2 - p_1) + (p_3 - p_2) + ... + (p_n - p_{n-1})$。
- **连续下降交易日：** 则不买卖收益最大，即不会亏钱。

算法流程：

- 遍历整个股票交易日价格列表 `price` ，策略是所有上涨交易日都买卖（赚到所有利润），所有下降交易日都不买卖（永不亏钱）。

1. 设 `tmp` 为第 `i-1` 日买入与第 `i` 日卖出赚取的利润，即 `tmp = prices[i] - prices[i - 1]` ；
2. 当该天利润为正 `tmp > 0` ，则将利润加入总利润 `profit` ；当利润为 0 或为负，则直接跳过；
3. 遍历完成后，返回总利润 `profit` 。

复杂度分析：

时间复杂度 O(N) ： 只需遍历一次 price；

空间复杂度 O(1)： 变量使用常数额外空间。

链接：

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/solution/best-time-to-buy-and-sell-stock-ii-zhuan-hua-fa-ji/

方法 2：贪心算法【最优解】

```
class Solution {
    public int maxProfit(int[] prices) {
        int profit = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] - prices[i - 1]> 0)
                profit += prices[i] - prices[i - 1];
        }
        return profit;
    }
}
```

15、买卖股票的最佳时机 3（123）【2 笔交易】

更多硬核内容就在公众号「yes 的练级攻略」

给定一个数组，它的第 *i* 个元素是一支给定的股票在第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 *两笔* 交易。

注意: 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [3,3,5,0,0,3,1,4]
输出: 6
解释: 在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
     随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3 。
```

示例 2:

```
输入: [1,2,3,4,5]
输出: 4
解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4 。
     注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
     因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。
```

示例 3:

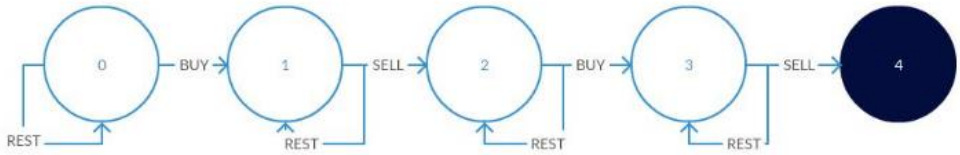
```
输入: [7,6,4,3,1]
输出: 0
解释: 在这个情况下, 没有交易完成, 所以最大利润为 0。
```

方法 1：动态规划代码

```java
class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) {
            return 0;
        }
        int dp1 = 0;
        int dp2 = 0;
        int min1 = prices[0];
        int min2 = prices[0];
        for (int i = 1; i < prices.length; i++) {
            min1 = Math.min(prices[i] - 0, min1);
            dp1 = Math.max(dp1, prices[i] - min1);

            min2 = Math.min(prices[i] - dp1, min2);
            dp2 = Math.max(dp2, prices[i] - min2);
        }
        return dp2;
    }
}
```

再分享个利用状态机的 解法，虽然不容易想到，但真的太强了，上次用状态机还是 65 题。

每天我们其实是有四个状态，买入当前价格的股票，以当前价格的股票卖出。第二次买入股票，第二次卖出股票。



`s0` 代表初始状态，初始时钱是 `0` 。 `s1` 代表第一次买入后当前的钱， `s2` 代表第一次卖出后当前的前， `s3` 代表第二次买入后当前的钱， `s4` 代表第二次卖出后当前的钱。

然后我们只需要更新每天的这四个状态即可。

方法 2：状态机代码

```
class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) {
            return 0;
        }
         //进行初始化，第一天 s1 将股票买入，其他状态全部初始化为最小值
        int s1 = -prices[0], s2 = Integer.MIN_VALUE, s3 = Integer.MIN_V
ALUE, s4 = Integer.MIN_VALUE;
        for (int i = 1; i < prices.length; i++) {
            s1 = Math.max(s1, -prices[i]); //买入价格更低的股
            s2 = Math.max(s2, s1 + prices[i]); //卖出当前股，或者不操作
            s3 = Math.max(s3, s2 - prices[i]); //第二次买入，或者不操作
            s4 = Math.max(s4, s3 + prices[i]); //第二次卖出，或者不操作
        }
        return Math.max (0, s4);
    }
}
```

16、买卖股票的最佳时机 4（188）【最多 k 笔交易】

给定一个数组，它的第 *i* 个元素是一支给定的股票在第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **k** 笔交易。

注意: 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [2,4,1], k = 2
输出: 2
解释: 在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。
```

示例 2:

```
输入: [3,2,6,5,0,3], k = 2
输出: 7
解释: 在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出，这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
```

不能用状态机

解题思路

首先需要判断k的大小，如果k >= n/2 的话，这样其实就不存在交易次数的限制，可以按照正常的买卖股票来处理
我们定义两个数组，buy 和sell 分别存放第i次交易中买入和卖出后的收益

```
class Solution {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;
        if(n < 2) return 0;
//k>= n/2，按无限次算，参考 122 的贪心算法
        if(k >= n/2){
            int profit = 0;
            for(int i = 1; i < n; i++){
                if(prices[i] > prices[i-1]){
                    profit += prices[i] - prices[i-1];
                }
            }
            return profit;
        }

        int[] buy = new int[k+1];
        int[] sell = new int[k+1];
        Arrays.fill(buy, Integer.MIN_VALUE);
        for(int cur : prices){ //等于 for (int j = 0; j < n; j++)，然后 cur
用 prices[j]表示
            for(int i = 1; i <= k; i++){
                buy[i] = Math.max(buy[i], sell[i-1] -prices[j]);
                sell[i] = Math.max(sell[i], buy[i] + prices[j]);
                // if(buy[i] < sell[i-1] - prices[j]){
                //     buy[i] = sell[i-1] -prices[j];
                // }
                // if(sell[i] < buy[i] + prices[j]){
                //     sell[i] = buy[i] + prices[j];
                // }
            }
        }
        return sell[k];
    }
}
```

17、最佳买卖股票时机含冷冻期（309）

309. 最佳买卖股票时机含冷冻期

难度 中等 350 收藏 分享 切换为英文 关注 反馈

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

示例:

```
输入: [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

```java
class Solution {
    public int maxProfit(int[] prices) {
        int len = prices.length;
        if (len < 2) return 0;
        //x 表示本不持有，y 表示持有，z 表示当天卖出，不持有
        int x = 0, y = -prices[0], z = 0;
        for (int i = 1; i < len; i++){
            int tmp = y;
            y = Math.max(y, x-prices[i]);
            x = Math.max(x,z);
            z = y+prices[i];
        }
        return Math.max(x,z);  //返回最后一天不持有股票的状态，此处可以得到收益的最大值
    }
}
```

18、买卖股票的最佳时机含手续费（714）

714. 买卖股票的最佳时机含手续费

难度 中等 199 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 `prices`，其中第 `i` 个元素代表了第 `i` 天的股票价格；非负整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1:

```
输入: prices = [1, 3, 2, 8, 4, 9], fee = 2
输出: 8
解释: 能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

注意:

- `0 < prices.length <= 50000`.
- `0 < prices[i] < 50000`.
- `0 <= fee < 50000`.

```java
class Solution {
    public int maxProfit(int[] prices, int fee) {
        if (prices == null || prices.length == 0) return 0;
        int dp_i_0 = 0, dp_i_1 = -prices[0];
        for (int i = 0; i < prices.length; i++) {
            int temp = dp_i_0;
            dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i] - fee);
            dp_i_1 = Math.max(dp_i_1, temp - prices[i]);
        }
        return dp_i_0;
    }
}
```

19、假期（腾讯 2020 校园招聘-后台第 4 题）

更多硬核内容就在公众号「yes 的练级攻略」

由于业绩优秀，公司给小Q放了 n 天的假，身为工作狂的小Q打算在在假期中工作、锻炼或者休息。他有个奇怪的习惯：不会连续两天工作或锻炼。只有当公司营业时，小Q才能去工作，只有当健身房营业时，小Q才能去健身，小Q一天只能干一件事。给出假期中公司，健身房的营业情况，求小Q最少需要休息几天。

输入描述:

第一行一个整数 $n(1 \leq n \leq 100000)$ 表示放假天数
第二行 n 个数 每个数为0或1,第 i 个数表示公司在第 i 天是否营业
第三行 n 个数 每个数为0或1,第 i 个数表示健身房在第 i 天是否营业
（1为营业 0为不营业）

输出描述:

一个整数，表示小Q休息的最少天数

输入例子1:

```
4
1 1 0 0
0 1 1 0
```

输出例子1:

```
2
```

例子说明1:

小Q可以在第一天工作，第二天或第三天健身，小Q最少休息2天

```
import java.util.Scanner;
public class Main {
    //策略：使用 dp[i][0] dp[i][1] dp[i][2]分别记录在第 i 天如果是休息、工作、健身情况下，前 i 天有事做（也就是没休息）的最大天数
    //如果第 i 天休息，那么前 i 天有事做的最大天数，实际就是 dp[i-1][0] dp[i-1][1] dp[i-1][2] 中的最大值
    //如果第 i 天工作，那么前 i 天有事做的最大天数，就是前一天休息、健身中的最大值 + 1（因为第 i 天工作了，没有休息）
    //如果第 i 天健身，那么前 i 天有事做的最大天数，就是前一天休息、工作中的最大值 + 1（因为第 i 天健身了，没有休息）
    //最后的结果，就用 day 减去最大的做事天数，就是最少的休息时间了
    public static int holiday(int day, int [] works, int [] gyms){
        int res = 0;
        int[][] dp = new int[day+1][3];
        dp[0][0] = dp[0][1] = dp[0][2] = 0;
        for (int i = 1; i < day+1; i++) {
            if(works[i] == 1){
                //第 i 天可以选择工作
                dp[i][1] = Math.max(dp[i-1][0], dp[i-1][2]) + 1;
            }
            if(gyms[i] == 1){
                //第 i 天可以选择健身
```

```
                dp[i][2] = Math.max(dp[i-1][0], dp[i-1][1]) + 1;
            }
            dp[i][0] = Math.max(dp[i-1][0], Math.max(dp[i-1][1], dp[i-1][2]));
        }
        res = day - Math.max(dp[day][0], Math.max(dp[day][1], dp[day][2]));
        return res;
    }

    public static void main(String[] args) {
        //第一步数据处理：三行，就是三个字符串，后两行进行一个分割处理，然后再调用函数转换成为整型数据
        /*  第一种数据处理方式
        Scanner in = new Scanner(System.in);
        String day_str = in.nextLine();
        String [] work_str = in.nextLine().split(" ");
        String [] gym_str = in.nextLine().split(" ");
        int day = Integer.parseInt(day_str);    //日期
        int [] works = new int[day+1];
        int [] gyms = new int[day+1];
        for(int i = 1;i < day+1;i++) {
            works[i] = Integer.parseInt(work_str[i - 1]);
            gyms[i] = Integer.parseInt(gym_str[i-1]);
        }
        int res = holiday(day, works, gyms);
        System.out.println(res);
        */
        //第二种处理方法，适用于输入参数都是 int 的时候
        Scanner sc = new Scanner(System.in);
        int day = sc.nextInt();
        int[] works = new int[day+1];
        int[] gyms = new int[day+1];
        for(int i = 1; i < day + 1; i++) {
            works[i] = sc.nextInt();
        }
        for(int i = 1; i < day + 1; i++) {
            gyms[i] = sc.nextInt();
        }
        int res = holiday(day, works, gyms);
        System.out.println(res);
    }
}
```

20、鸡蛋掉落（887）【lintcode 584 丢鸡蛋二】

禁止商用 公众号：yes的练级攻略

887. 鸡蛋掉落

难度 困难 401 收藏 分享 切换为英文 关注 反馈

你将获得 `K` 个鸡蛋，并可以使用一栋从 `1` 到 `N` 共有 `N` 层楼的建筑。

每个蛋的功能都是一样的，如果一个蛋碎了，你就不能再把它掉下去。

你知道存在楼层 `F` ，满足 `0 <= F <= N` 任何从高于 `F` 的楼层落下的鸡蛋都会碎，从 `F` 楼层或比它低的楼层落下的鸡蛋都不会破。

每次*移动*，你可以取一个鸡蛋（如果你有完整的鸡蛋）并把它从任一楼层 `X` 扔下（满足 `1 <= X <= N`）。

你的目标是**确切地**知道 `F` 的值是多少。

无论 `F` 的初始值如何，你确定 `F` 的值的最小移动次数是多少？

示例 1：

```
输入：K = 1, N = 2
输出：2
解释：
鸡蛋从 1 楼掉落。如果它碎了，我们肯定知道 F = 0 。
否则，鸡蛋从 2 楼掉落。如果它碎了，我们肯定知道 F = 1 。
如果它没碎，那么我们肯定知道 F = 2 。
因此，在最坏的情况下我们需要移动 2 次以确定 F 是多少。
```

示例 2：

```
输入：K = 2, N = 6
输出：3
```

示例 3：

```
输入：K = 3, N = 14
输出：4
```

提示：

1. `1 <= K <= 100`
2. `1 <= N <= 10000`

https://leetcode-cn.com/problems/super-egg-drop/solution/ji-dan-diao-luo-by-leetcode-solution-2/

方法三：数学法

说明

方法三涉及逆向思维，是一种没见过就不太可能想出来，读过题解也很容易忘记的方法。

思路

反过来想这个问题：如果我们可以做 T 次操作，而且有 K 个鸡蛋，那么我们能找到答案的最高的 N 是多少？我们设 $f(T,K)$ 为在上述条件下的 N。如果我们求出了所有的 $f(T,K)$，那么只需要找出最小的满足 $f(T,K) \geq N$ 的 T。

那么我们如何求出 $f(T,K)$ 呢？我们还是使用动态规划。因为我们需要找出最高的 N，因此我们不必思考到底在哪里扔这个鸡蛋，我们只需要扔出一个鸡蛋，看看到底发生了什么：

- 如果鸡蛋没有碎，那么对应的是 $f(T-1,K)$，也就是说在这一层的上方可以有 $f(T-1,K)$ 层；
- 如果鸡蛋碎了，那么对应的是 $f(T-1,K-1)$，也就是说在这一层的下方可以有 $f(T-1,\ K-1)$ 层。

因此我们就可以写出状态转移方程：

$$f(T,K) = 1 + f(T-1,K-1) + f(T-1,K)$$

边界条件为：当 $T \geq 1$ 的时候 $f(T,1) = T$，当 $K \geq 1$ 时，$f(1,K) = 1$。

那么问题来了：T **最大可以达到多少**？由于我们在进行动态规划时，T 在题目中并没有给出，那么我们需要进行到动态规划的哪一步呢？可以发现，操作次数是一定不会超过楼层数的，因此 $T \leq N$，我们只要算出在 $f(N,K)$ 内的所有 f 值即可。

复杂度分析

- 时间复杂度：$O(K*N)$。事实上，更准确的时间复杂度应当为 $O(K*T)$，我们不加证明地给出 $N = O(T^K)$，因此有 $O(K*T) = O(K*\sqrt[K]{N})$。
- 空间复杂度：$O(K*N)$。

方法 1：数学法（13ms）将 N 层楼扔多少次的问题转化为扔多少次可以覆盖 N 层楼

```
class Solution {
    public int superEggDrop(int K, int N) {
        if (N == 1) {
            return 1;
        }
        int[][] f = new int[N+1][K+1];
        for (int i = 1; i < K+1; ++i) {
            f[1][i] = 1;
        }
        int ans = -1;
        for (int i = 2; i < N+1; ++i) {
            for (int j = 1; j < K+1; ++j) {
                f[i][j] = 1 + f[i-1][j-1] + f[i-1][j];
            }
            if (f[i][K] >= N) {
                ans = i;
                break;
            }
        }
        return ans;
```

```
    }
}
```

方法 2：上述方法的优化：因为次数都是 i-1，将层数省略【K 为鸡蛋，N 为层数】

```
class Solution {
    public int superEggDrop(int K, int N) {
        if (N == 1) return 1;
        int [] dp = new int[K+1];
        int ans = 0;
        while (dp[K] < N) {
            for (int i = K; i > 0; i--) {
                dp[i] = dp[i] + dp[i-1] + 1;
            }
            ans++;
        }
        return ans;
    }
}
```

21、丢鸡蛋（lintcode 254）【只有两个鸡蛋】

254. 丢鸡蛋

楼有 n 层高，鸡蛋若从 k 层或以上扔，就会碎。从 k 层以下扔，就不会碎。

现在给两个鸡蛋，用最少的扔的次数找到 k。返回最坏情况下次数。

样例

样例 1:

输入：100
输出：14

样例 2:

输入：10
输出：4

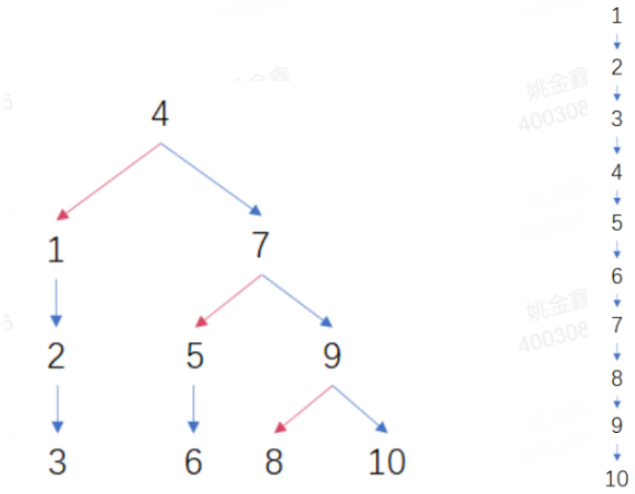
说明

对于 n = 10， 一种找 k 的初级方法是从 1、2 ... k 层不断找。但最坏情况下要扔 10 次。

注意有两个鸡蛋可以使用，所以可以从 4、7、9 层扔。这样最坏就只需要 4 次 (例如 k = 9 时)。

题意解释：

有两个鸡蛋可以扔，也就是第一个鸡蛋若是碎了，那么第二个鸡蛋必须不能出错，得从确定不会碎的楼层x开始，一层一层向上试，直到找到答案。若从1到5逐一尝试，那么最坏情况就是5次，而现在要求在最坏情况下，最少的扔鸡蛋次数。以n=10时为例，下图展示了两种策略方案。



注：红色箭头表示若在当前楼层第一个鸡蛋碎了，蓝色表示鸡蛋未碎。

上图可见，我们要求的最坏情况扔鸡蛋次数即图中情况树的深度，左图需扔4次，右图需扔10次。而题目求最小的次数，即如何安排决策可以使得这棵树的深度最小。可见，当我们安排每一种情况的所有叶子节点高度越平均，树的深度越小。

解题思路：

故可以想到，假设第一次从x层开始扔，那么若鸡蛋碎了，需要从1到x - 1逐层尝试,那么共需尝试x次; 若第一次鸡蛋未碎，第二次尝试从y层开始，碎在y层，那么需要从x + 1到y - 1逐层尝试，共需尝试 （y - x + 1）次，那么当x = y - x + 1时（也即y = x + x - 1），能够使得扔鸡蛋次数最小，以此类推，第一次从x层开始，那么若未碎第二次应当从x + x - 1层开始，第三次应当从x + x - 1 + x - 2层开始.....

那么x的取值应当满足（1 + 2 + ... + x）>= n 即 x * (x + 1) / 2 >= n 的最小值即为答案值。

复杂度分析：

- 时间复杂度：若暴力从小到大搜索找x则复杂度为$O(sqrt(n))$，若二分查找x则复杂度为$O(log(sqrt(n)))$
- 空间复杂度：$O(1)$

```
public class Solution {
    public int dropEggs(int n) {
        long ans = 0;
        for (int i = 1; ; ++i) {
            ans += (long) i;
            if (ans >= (long) n)
                return i;
        }
    }
}
```

22、戳气球（312）【区间 DP】

yes的练级攻略

312. 戳气球

难度 困难 406 收藏 分享 切换为英文 关注 反馈

有 n 个气球，编号为0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。如果你戳破气球 i ，就可以获得 nums[left] * nums[i] * nums[right] 个硬币。 这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

- 你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。
- 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

示例:

```
输入: [3,1,5,8]
输出: 167
解释: nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
     coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

通过次数 19,757 | 提交次数 30,844

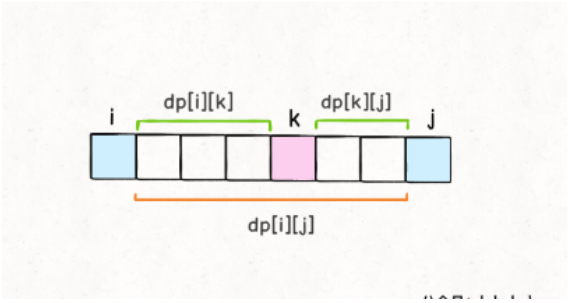
https://leetcode-cn.com/problems/burst-balloons/solution/dong-tai-gui-hua-tao-lu-jie-jue-chuo-qi-qiu-wen-ti/

根据刚才对 dp 数组的定义，如果最后一个戳破气球 k ， dp[i][j] 的值应该为：

```
dp[i][j] = dp[i][k] + dp[k][j]
         + points[i]*points[k]*points[j]
```

你不是要最后戳破气球 k 吗？那得先把开区间 (i, k) 的气球都戳破，再把开区间 (k, j) 的气球都戳破；最后剩下的气球 k ，相邻的就是气球 i 和气球 j ，这时候戳破 k 的话得到的分数就是 points[i]*points[k]*points[j] 。

那么戳破开区间 (i, k) 和开区间 (k, j) 的气球最多能得到的分数是多少呢？嘿嘿，就是 dp[i][k] 和 dp[k][j] ，这恰好就是我们对 dp 数组的定义嘛！



更多硬核内容就在公众号「yes 的练级攻略」

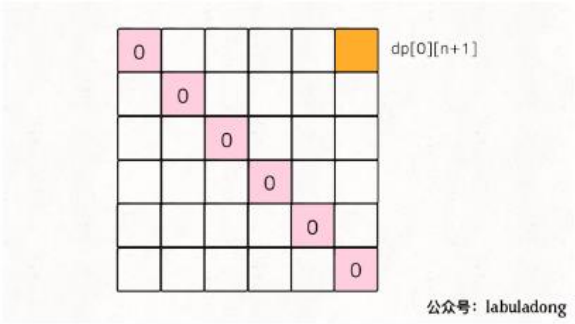
关于「状态」的穷举，最重要的一点就是：状态转移所依赖的状态必须被提前计算出来。

拿这道题举例，`dp[i][j]` 所依赖的状态是 `dp[i][k]` 和 `dp[k][j]`，那么我们必须保证：在计算 `dp[i][j]` 时，`dp[i][k]` 和 `dp[k][j]` 已经被计算出来了（其中 `i < k < j`）。

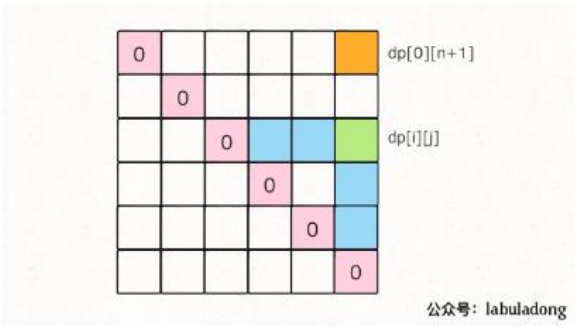
那么应该如何安排 `i` 和 `j` 的遍历顺序，来提供上述的保证呢？我们前文 动态规划答疑篇 写过处理这种问题的一个鸡贼技巧：**根据 base case 和最终状态进行推导。**

PS：最终状态就是指题目要求的结果，对于这道题目也就是 `dp[0][n+1]`。

我们先把 base case 和最终的状态在 DP table 上画出来：



对于任一 `dp[i][j]`，我们希望所有 `dp[i][k]` 和 `dp[k][j]` 已经被计算，画在图上就是这种情况：



那么，为了达到这个要求，可以有两种遍历方法，要么斜着遍历，要么从下到上从左到右遍历：

YES的练级攻略

312 具体代码

```
class Solution {
    public int maxCoins(int[] nums) {
            int n = nums.length;
            // 添加两侧的虚拟气球
            int[] points = new int[n + 2];
            points[0] = points[n + 1] = 1;
            for (int i = 1; i <= n; i++) {
                points[i] = nums[i - 1];
            }
            // base case 已经都被初始化为 0
            int[][] dp = new int[n + 2][n + 2];
            // 开始状态转移
            // i 应该从下往上
            for (int i = n; i >= 0; i--) {
                // j 应该从左往右
                for (int j = i + 1; j < n + 2; j++) {
```

```
                    // 最后戳破的气球是哪个？
                    for (int k = i + 1; k < j; k++) {
                        // 择优做选择
                        dp[i][j] = Math.max(dp[i][j], dp[i][k] + dp[k][
j] + points[i]*points[j]*points[k]);
                    }
                }
            }
        return dp[0][n + 1];
    }
}
```

23、移除盒子（546）

546. 移除盒子

难度 困难 151 收藏 分享 切换为英文 关注 反馈

给出一些不同颜色的盒子，盒子的颜色由数字表示，即不同的数字表示不同的颜色。
你将经过若干轮操作去去掉盒子，直到所有的盒子都去掉为止。每一轮你可以移除具有相同颜色的连续 k 个盒子（k >= 1），这样一轮之后你将得到 k*k 个积分。
当你将所有盒子都去掉之后，求你能获得的最大积分和。

示例：

```
输入：boxes = [1,3,2,2,2,3,4,3,1]
输出：23
解释：
[1, 3, 2, 2, 2, 3, 4, 3, 1]
----> [1, 3, 3, 4, 3, 1] (3*3=9 分)
----> [1, 3, 3, 3, 1] (1*1=1 分)
----> [1, 1] (3*3=9 分)
----> [] (2*2=4 分)
```

提示：

- 1 <= boxes.length <= 100
- 1 <= boxes[i] <= 100

https://leetcode-cn.com/problems/remove-boxes/solution/yi-chu-he-zi-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

我们很容易陷入这样一个错误的思路：用 $f(l,r)$ 来表示移除区间 $[l,r]$ 内所有的盒子能得到的最大积分，然后去探索某一种移除盒子的策略来进行状态转移。而实际上，我们并不能直接使用起始节点和结束节点决定最大分数，因为这个分数并不只依赖于子序列，也依赖于之前的移动对当前数组的影响，这可能让最终的子序列不是一个连续的子串。比如 $\{3,4,2,4,4\}$，如果先把 2 移除，3 个 4 会合并在一起，对答案的贡献是 $3^2=9$，如果先移除左右两边的 4 再移除 2 这里 3 个 4 的贡献就是 $1^2+2^2=5$，最优的办法当然是先取走 2，但是这样剩下的 3 个 4 其实并不是原串的某个连续子串。

我们可以换一种思路，用 $f(l,r,k)$ 表示移除区间 $[l,r]$ 加上该区间右边等于 a_r 的 k 个元素组成的这个序列的最大积分。例如序列 $\{6,3,6,5,6,7,6,6,8,6\}$，$l=1$（下标从 1 开始），$r=5$，那么 $f(l,r,3)$ 对应的元素就是 $\{[6,3,6,5,6],7,6,6,8,6\}$ 中标记为红色的部分。$f(l,r,k)$ 的定义是移除这个红色的序列获得的最大积分。请注意此时我们约定 7 和 8 已经先被移除，所以在这个状态下我们可以认为最后四个 6 是连续的，也就是说实际上序列是这样的：$\{[6,3,6,5,6],6,6,6\}$，此时我们可以有这样一些策略来移除盒子：

- $\{[6,3,6,5,6],6,6,6\}$，删除后面的四个 6，再删除前面的这个区间，这样获得的分数为 $f(1,4,0)+4^2$
- $\{[6,3,6],[5],6,6,6,6\}$，删除一个 5，然后后面的 5 个 6 一起删除，再删除前面的橘黄色区间，这样获得的分数是 $f(1,2,5)+f(4,4,0)$
- $\{[6],[3,6,5],6,6,6,6\}$ 删除 3、6、5 之后再删除 5 个 6，这样获得的分数是 $f(1,1,5)+f(2,4,0)$

这个就是我们转移的时候使用的策略，我们可以推导出这样的动态规划转移方程：

$$f(l,r,k)=\max\left\{f(l,r-1,0)+(k+1)^2,\max_{i=l}^{r-1}\{[f(l,i,k+1)+f(i+1,r-1,0)]\times\epsilon(a_i==a_r)\}\right\}$$

- $f(l,r-1,0)+(k+1)^2$ 代表我们把 a_r 和后面的 k 个数一起删除，再删除 $[l,r-1]$ 这个区间
- $[f(l,i,k+1)+f(i+1,r-1,0)]\times\epsilon(a_i==a_r)$ 代表当 $a_i(l\le i<r)$ 等于 a_r 的时候，考虑先删掉 $[i+1,r-1]$ 这个区间，然后再删除 $[l,i]$ 区间和后面的 $k+1$ 个和 a_r 相等的数构成的序列，其中 $\epsilon(x)$ 为选择函数：

$$\epsilon(x)=\begin{cases}1, & x==\text{True}\\ 0, & x==\text{False}\end{cases}$$

这样我们就可以计算出 $f(1,n,0)$ 的值，即为答案。

复杂度分析

- 时间复杂度：$O(n^4)$。最坏情况下每个 $f(l,r,k)$ 被计算一次，每次状态转移需要 $O(n)$ 的时间复杂度。
- 空间复杂度：$O(n^3)$。dp 数组的空间代价是 $O(n^3)$，递归使用栈空间的代价为 $O(n)$。

```
class Solution {
    public int removeBoxes(int[] boxes) {
        int[][][] dp = new int[100][100][100];
        return calculatePoints(boxes, dp, 0, boxes.length - 1, 0);
    }

    public int calculatePoints(int[] boxes, int[][][] dp, int l, int r,
 int k) {
        if (l > r) return 0;
        if (dp[l][r][k] != 0) return dp[l][r][k];
        while (r > l && boxes[r] == boxes[r - 1]) {
            r--;
            k++;
        }
        dp[l][r][k] = calculatePoints(boxes, dp, l, r - 1, 0) + (k + 1)
 * (k + 1);
```

```
            for (int i = l; i < r; i++) {
                if (boxes[i] == boxes[r]) {
                    dp[l][r][k] = Math.max(dp[l][r][k], calculatePoints(box
es, dp, l, i, k + 1) + calculatePoints(boxes, dp, i + 1, r - 1, 0));
                }
            }
            return dp[l][r][k];
        }
    }
```

24、多边形三角剖分的最低得分（1039）【区间 DP】

1039. 多边形三角剖分的最低得分

难度 中等 53 收藏 分享 切换为英文 关注 反馈

给定 N，想象一个凸 N 边多边形，其顶点按顺时针顺序依次标记为 A[0], A[i], ..., A[N-1]。

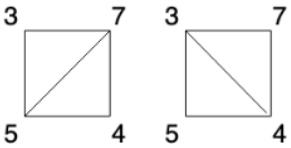
假设您将多边形剖分为 N-2 个三角形。对于每个三角形，该三角形的值是顶点标记的乘积，三角剖分的分数是进行三角剖分后所有 N-2 个三角形的值之和。

返回多边形进行三角剖分后可以得到的最低分。

示例 1：

输入： [1,2,3]
输出： 6
解释： 多边形已经三角化，唯一三角形的分数为 6。

示例 2：



输入： [3,7,4,5]
输出： 144
解释： 有两种三角剖分，可能得分分别为：3*7*5 + 4*5*7 = 245，或 3*4*5 + 3*4*7 = 144。最低分数为 144。

示例 3：

输入： [1,3,1,4,1,5]
输出： 13
解释： 最低分数三角剖分的得分情况为 1*1*3 + 1*1*4 + 1*1*5 + 1*1*1 = 13。

提示：

1. 3 <= A.length <= 50
2. 1 <= A[i] <= 100

https://leetcode-cn.com/problems/minimum-score-triangulation-of-polygon/solution/javaqu-jian-dpdian-xing-li-ti-by-im1gw0/

区间 DP 套路，枚举所有的区间和区间内的点，区间内的点和区间左右端点将整个区间又划分为两个已知的子区间。

```
class Solution {
    //环形的结构
    //从 0 ~ N-1 形成一个环
    //    1—3
    //   /    \
```

```
    //  5       1
    //   \     /
    //    1—4
    public int minScoreTriangulation(int[] A) {
        if(A == null || A.length == 0) return 0;
        int N = A.length;
        int[][] dp=new int[N][N];
        for (int len = 3; len <= N; len++) { //枚举长度,从 3 开始
            for (int left = 0;left <= N - len;left++) { //枚举左端点
                //left+len-1<N
                int right = left + len - 1;
                //init
                dp[left][right] = Integer.MAX_VALUE;
                for (int i = left+1; i < right; i++) { //枚举区间内的所有
的点(不包括端点)),将环分割成左右两部分
                    dp[left][right] = Math.min(dp[left][right], dp[left
][i] + dp[i][right] + A[i]*A[left]*A[right]);
                }
            }
        }
        return dp[0][N-1];
    }
}
```

25、奇怪的打印机（664）【刷串问题，区间 DP】

664. 奇怪的打印机

难度 困难 70 收藏 分享 切换为英文 关注 反馈

有台奇怪的打印机有以下两个特殊要求：

1. 打印机每次只能打印同一个字符序列。
2. 每次可以在任意起始和结束位置打印新字符，并且会覆盖掉原来已有的字符。

给定一个只包含小写英文字母的字符串，你的任务是计算这个打印机打印它需要的最少次数。

示例 1:

输入: "aaabbb"
输出: 2
解释: 首先打印 "aaa" 然后打印 "bbb"。

示例 2:

输入: "aba"
输出: 2
解释: 首先打印 "aaa" 然后在第二个位置打印 "b" 覆盖掉原来的字符 'a'。

提示: 输入字符串的长度不会超过 100。

https://leetcode-cn.com/problems/strange-printer/solution/java-qu-jian-dp-by-im1gw0/

思路

动态规划，`dp[left][right]` 代表打印 `left` 到 `right` 的最少打印次数，然后我们要理解一点：`dp[left][right]` 其实只可能有两个取值，`dp[left][right-1]` 或者 `dp[left][right-1]+1`，我们求最小打印次数其实就是希望增加一个字符后打印次数不变

> eg: abc b 这个最后的b就可以连同前面的bc一起顺带打印出来，打印次数不变

枚举所有的区间 `[left,right]`，然后枚举区间 `[left,right-1]` 内的字符，如果区间 `[left,right-1]` 内的某个字符 `i` 和 `right` 相等，那么 `right` 就可以跟随 `[i,right-1]` 一起顺带打印出来，这样该区间的

- `dp[left][right]=dp[left][i-1]+dp[i][right-1]`

这样就**有可能**让打印次数不变，我们求一下所有情况的最小值就可以了

```
class Solution {
    public int strangePrinter(String s) {
        if( s == null || s.length() == 0) return 0;
        int n = s.length();
        int[][] dp = new int[n][n];
        for (int i = 0; i < n; i++) {
            dp[i][i]=1;
        }
        //aa        1
        //aab       2
        //aaba      2
        //aabab     3
        //aababa    3
        for (int len = 2; len <= n; len++) {
            for (int left = 0; left + len - 1 < n; left++) { //左端点
                int right = left + len - 1; //右端点
                //和前一个区间[left,right-1]的第一个字符相等,可以直接顺带打印出来
                if(s.charAt(right) == s.charAt(left)){
                    //至少是 dp[left][right-1] 新增加一个字符不可能打印次数还减少 所以直接跳出减少循环次数
                    dp[left][right] = dp[left][right-1];
                    continue;
                }
                //最多是前一个区间+1(直接打印 right 位置的字符)
                dp[left][right] = dp[left][right-1]+1;
                //枚举[left,right)中的所有字符,如果有和 right 相等的就考虑从这个位置分割整个字符
                //比如 abc b 这个最后的 b 就可以连同前面的 bc 一起顺带打印出来
                //所以当前区间的值就是可以是 dp[a][a]+dp[b][c],我们从所有的这种情况中取最小值就 ok 了
                for (int i = left; i < right; i++) {
                    if(s.charAt(right) == s.charAt(i)){
                        dp[left][right] = Math.min(dp[left][i-1] + dp[i][right-1], dp[left][right]);
```

```
                }
            }
        }
    }
    return dp[0][n-1];
  }
}
```

26、整数拆分（343）【与下一题剪绳子类似】

343. 整数拆分

难度 中等 290 收藏 分享 切换为英文 关注 反馈

给定一个正整数 *n*，将其拆分为**至少**两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

示例 1:

```
输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。
```

示例 2:

```
输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。
```

说明: 你可以假设 *n* 不小于 2 且不大于 58。

https://leetcode-cn.com/problems/integer-break/solution/zheng-shu-chai-fen-by-leetcode-solution/

方法一：动态规划

对于的正整数 n，当 $n \geq 2$ 时，可以拆分成至少两个正整数的和。令 k 是拆分出的第一个正整数，则剩下的部分是 $n-k$，$n-k$ 可以不继续拆分，或者继续拆分成至少两个正整数的和。由于每个正整数对应的最大乘积取决于比它小的正整数对应的最大乘积，因此可以使用动态规划求解。

创建数组 dp，其中 $\mathrm{dp}[i]$ 表示将正整数 i 拆分成至少两个正整数的和之后，这些正整数的最大乘积。特别地，0 不是正整数，1 是最小的正整数，0 和 1 都不能拆分，因此 $\mathrm{dp}[0] = \mathrm{dp}[1] = 0$。

当 $i \geq 2$ 时，假设对正整数 i 拆分出的第一个正整数是 j（$1 \leq j < i$），则有以下两种方案：

- 将 i 拆分成 j 和 $i-j$ 的和，且 $i-j$ 不再拆分成多个正整数，此时的乘积是 $j \times (i-j)$；
- 将 i 拆分成 j 和 $i-j$ 的和，且 $i-j$ 继续拆分成多个正整数，此时的乘积是 $j \times \mathrm{dp}[i-j]$。

因此，当 j 固定时，有 $\mathrm{dp}[i] = \max(j \times (i-j), j \times \mathrm{dp}[i-j])$。由于 j 的取值范围是 1 到 $i-1$，需要遍历所有的 j 得到 $\mathrm{dp}[i]$ 的最大值，因此可以得到状态转移方程如下：

$$\mathrm{dp}[i] = \max_{1 \leq j < i} \{\max(j \times (i-j), j \times \mathrm{dp}[i-j])\}$$

最终得到 $\mathrm{dp}[n]$ 的值即为将正整数 n 拆分成至少两个正整数的和之后，这些正整数的最大乘积。

复杂度分析

- 时间复杂度：$O(n^2)$，其中 n 是给定的正整数。对于从 2 到 n 的每一个整数都要计算对应的 dp 值，计算一个整数对应的 dp 值需要 $O(n)$ 的时间复杂度，因此总时间复杂度是 $O(n^2)$。
- 空间复杂度：$O(n)$，其中 n 是给定的正整数。创建一个数组 dp，其长度为 $n+1$。

方法 1：朴素的动规

```
class Solution {
    public int integerBreak(int n) {
        int[] dp = new int[n + 1];
        for (int i = 2; i <= n; i++) {
            int curMax = 0;
            for (int j = 1; j < i; j++) {
                curMax = Math.max(curMax, Math.max(j * (i - j), j * dp[
i - j]));
            }
            dp[i] = curMax;
        }
        return dp[n];
    }
}
```

方法二：优化的动态规划

方法一中定义的状态转移方程如下：

$$\mathrm{dp}[i] = \max_{1 \le j < i} \{\max(j \times (i-j), j \times \mathrm{dp}[i-j])\}$$

使用上述状态转移方程，计算 $\mathrm{dp}[i]$ 时，j 的值遍历了从 1 到 $i-1$ 的所有值，因此总时间复杂度是 $O(n^2)$。是否可以降低时间复杂度？

上述状态转移方程包含两项，当 j 固定时，$\mathrm{dp}[i]$ 的值由 $j \times (i-j)$ 和 $j \times \mathrm{dp}[i-j]$ 中的较大值决定，因此需要对两项分别考虑。

首先考虑 $j \times \mathrm{dp}[i-j]$ 这一项。

注意到 dp 的定义，$\mathrm{dp}[i]$ 表示将正整数 i 拆分成至少两个正整数的和之后，这些正整数的最大乘积，因此对于任意 $1 \le j < i$，有 $\mathrm{dp}[i] \ge j \times \mathrm{dp}[i-j]$。

当 j 是奇数时，有 $j = \frac{j-1}{2} + \frac{j+1}{2}$，因此 $\mathrm{dp}[i] \ge \frac{j-1}{2} \times \mathrm{dp}[i - \frac{j-1}{2}] \ge \frac{j-1}{2} \times \frac{j+1}{2} \times \mathrm{dp}[i-j]$。

当 j 是偶数时，有 $j = \frac{j}{2} + \frac{j}{2}$，因此 $\mathrm{dp}[i] \ge \frac{j}{2} \times \mathrm{dp}[i - \frac{j}{2}] \ge \frac{j}{2} \times \frac{j}{2} \times \mathrm{dp}[i-j]$。

如果 $j \ge 4$ 且 j 是奇数，则 $\frac{j-1}{2} \times \frac{j+1}{2} > j$ 恒成立。如果 $j \ge 4$ 且 j 是偶数，则 $\frac{j}{2} \times \frac{j}{2} \ge j$ 恒成立，当且仅当 $j = 4$ 时等号成立。

由此可知，如果 $j \ge 4$，则 $\mathrm{dp}[j] \ge j$，当且仅当 $j = 4$ 时等号成立，即当 $j \ge 4$ 时一定能将 j 拆成至少两个正整数的和，这些正整数的乘积大于或等于 j。

同时也可以得到，如果 $j \ge 4$，则 $\mathrm{dp}[i] \ge j \times \mathrm{dp}[i-j]$，只有当 $j = 4$ 时等号可能成立。又由于

$$\mathrm{dp}[i] \ge 2 \times \mathrm{dp}[i-2] \ge 2 \times (2 \times \mathrm{dp}[i-4]) = 4 \times \mathrm{dp}[i-4]$$

因此取 $j = 2$ 计算得到的 $\mathrm{dp}[i]$ 一定不会小于取 $j = 4$ 计算得到的 $\mathrm{dp}[i]$。根据上述分析，$j \ge 4$ 的情况都是不需要考虑的。

更多硬核内容就在公众号「yes 的练级攻略」

那么 $j=1$ 是否需要考虑？答案是不需要。如果取 $j=1$，则有 $\mathrm{dp}[i] \geq 1 \times \mathrm{dp}[i-1] = \mathrm{dp}[i-1]$。当 $i \geq 3$ 时，$\mathrm{dp}[i-1]$ 是将正整数 $i-1$ 拆分成至少两个正整数的和之后，这些正整数的最大乘积，在拆分成的正整数中，任选一个数字加 1，则拆分成的正整数的和变成 i，且乘积一定大于 $\mathrm{dp}[i-1]$，因此必有 $\mathrm{dp}[i] > \mathrm{dp}[i-1]$，即当 $j=1$ 时不可能得到最大的 $\mathrm{dp}[i]$ 的值。

根据上述分析可知，计算 $\mathrm{dp}[i]$ 的值只需要考虑 $j=2$ 和 $j=3$ 的情况，不需要遍历从 1 到 $i-1$ 的所有值。

其次考虑 $j \times (i-j)$ 这一项。

根据上述推导可知，如果 $j \geq 4$，则 $\mathrm{dp}[j] \geq j$，当且仅当 $j=4$ 时等号成立。因此在 $i-j \geq 4$ 的情况下，有 $\mathrm{dp}[i-j] \geq i-j$，$\mathrm{dp}[i] \geq j \times \mathrm{dp}[i-j] \geq j \times (i-j)$，此时计算 $\mathrm{dp}[i]$ 的值不需要考虑 $j \times (i-j)$ 的值。

如果 $i-j<4$，在计算 $\mathrm{dp}[i]$ 的值的时候就需要考虑 $j \times (i-j)$ 的值。在考虑 $j \times \mathrm{dp}[i-j]$ 时，根据上述分析，只需要考虑 $j=2$ 和 $j=3$ 的情况。在考虑 $j \times (i-j)$ 时，需要考虑 j 的哪些值？

如果 $j=1$，则 $1 \times (i-1) = i-1$，当 $i=2$ 或 $i=3$ 时有 $\mathrm{dp}[i] = i-1$，当 $i \geq 4$ 时有 $\mathrm{dp}[i] \geq i > i-1$，显然当 $i \geq 4$ 时取 $j=1$ 不可能得到最大乘积，因此 $j=1$ 时是不需要考虑的。

如果 $j \geq 4$，$\mathrm{dp}[i]$ 是否可能等于 $j \times (i-j)$？当 i 固定时，要使得 $j \times (i-j)$ 的值最大，j 的值应该取 $j=i/2$，这里的 $/$ 表示整数除法。当 $j \geq 4$ 时，若要满足 $j=i/2$，则 $i \geq 8$，此时 $i-j \geq 4$，利用上述结论，$\mathrm{dp}[i-j] \geq i-j$，因此 $j \times \mathrm{dp}[i-j] \geq j \times (i-j)$。由此可见，当 $j \geq 4$ 时，计算 $\mathrm{dp}[i]$ 只需要考虑 $j \times \mathrm{dp}[i-j]$，不需要考虑 $j \times (i-j)$。

又由于在使用 $j \times \mathrm{dp}[i-j]$ 计算 $\mathrm{dp}[i]$ 时，$j=2$ 和 $j=3$ 的情况一定优于 $j \geq 4$ 的情况，因此无论是考虑 $j \times \mathrm{dp}[i-j]$ 还是考虑 $j \times (i-j)$，都只需要考虑 $j=2$ 和 $j=3$ 的情况。

由此可以对方法一的动态规划进行优化。

边界情况是 $n=2$，此时唯一的拆分方案是 $2=1+1$，最大乘积是 $1 \times 1 = 1$。

当 $i \geq 3$ 时，状态转移方程如下：

$$\mathrm{dp}[i] = \max(2 \times (i-2), 2 \times \mathrm{dp}[i-2], 3 \times (i-3), 3 \times \mathrm{dp}[i-3])$$

Java | C++ | Python3 | Golang | C

```java
class Solution {
    public int integerBreak(int n) {
        if (n < 4) {
            return n - 1;
        }
        int[] dp = new int[n + 1];
        dp[2] = 1;
        for (int i = 3; i <= n; i++) {
            dp[i] = Math.max(Math.max(2 * (i - 2), 2 * dp[i - 2]), Math.max(3 * (i - 3), 3 * dp[i - 3]));
        }
        return dp[n];
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是给定的正整数。和方法一相比，计算每个整数对应的 dp 的值的时间复杂度降到 $O(1)$，因此总时间复杂度降到 $O(n)$。
- 空间复杂度：$O(n)$，其中 n 是给定的正整数。创建一个数组 dp，其长度为 $n+1$。

更多硬核内容就在公众号「yes 的练级攻略」

方法三：数学

方法二中利用了数学知识降低时间复杂度。正整数 4 可以拆分成 $2+2$，乘积不变（$4=2\times2$）。对于大于 4 的正整数，总是存在一种拆分的方案，使得拆分成的两个正整数的乘积大于拆分前的正整数（例如，$5=2+3$，$2\times3=6>5$）。那么，能否利用数学知识在方法二的基础上进一步降低时间复杂度，找到最优的拆分方案呢？

下面给出两种直接得出最优拆分方案的证明方法。

函数极值证明法

显然，如果将给定的正整数拆分成尽可能多的某个特定的正整数，则这些正整数的乘积最大。

定义函数 $f(x)$ 表示将给定的正整数 n 拆分成尽可能多的正数 x 的情况下的最大乘积，则可以将 n 分成 $\frac{n}{x}$ 项，此时 $f(x)=x^{\frac{n}{x}}$，目标是求 $f(x)$ 的最大值，即

$$\max_x(f(x))$$

可以将 $f(x)$ 写成如下形式：

$$f(x)=x^{\frac{n}{x}}=e^{\frac{n\ln x}{x}}$$

令 $g(t)=e^t$，$h(x)=\frac{\ln x}{x}$，则有 $f(x)=g(n\cdot h(x))$。由于 $g(t)$ 是单调递增的，$n>0$，因此 $h(x)$ 与 $f(x)$ 的单调性相同。

计算 $h(x)$ 的驻点，即 $h'(x)=\frac{1-\ln x}{x^2}=0$ 的点，得到驻点为 $x=e$。

由于当 $0<x<e$ 时 $h'(x)>0$，当 $x>e$ 时 $h'(x)<0$，因此 $x=e$ 是 $h(x)$ 的极大值点，也是 $f(x)$ 的极大值点。由于函数 $f(x)$ 的定义域连续，因此极大值点唯一，也是最大值点。

因此，当 $x=e$ 时，$f(x)$ 取到最大值，$\max f(x)=f(e)=e^{\frac{n}{e}}$。

由于 e 不是整数，因此使用与 e 最接近的整数作为 x 的值，x 可以是 2 或 3，此时需要比较 $f(2)$ 与 $f(3)$ 的大小，可以通过计算 $\frac{f(3)}{f(2)}$ 进行比较。

$$\frac{f(3)}{f(2)}=\frac{e^{n\cdot h(3)}}{e^{n\cdot h(2)}}=e^{n\cdot h(3)-n\cdot h(2)}=e^{n\cdot(\frac{\ln3}{3}-\frac{\ln2}{2})}=e^{\frac{n}{6}\cdot(2\ln3-3\ln2)}=e^{\frac{n}{6}\cdot(\ln9-\ln8)}$$

由于 $\ln9>\ln8$，因此 $\frac{f(3)}{f(2)}>1$，即 $f(3)>f(2)$。当 $x=3$ 时，可以得到最大乘积。因此，应该将给定的正整数拆分成尽可能多的 3。

根据 n 除以 3 的余数进行分类讨论：

- 如果余数为 0，即 $n=3m(m\geq2)$，则将 n 拆分成 m 个 3；
- 如果余数为 1，即 $n=3m+1(m\geq1)$，由于 $2\times2>3\times1$，因此将 n 拆分成 $m-1$ 个 3 和 2 个 2；
- 如果余数为 2，即 $n=3m+2(m\geq1)$，则将 n 拆分成 m 个 3 和 1 个 2。

上述拆分的适用条件是 $n\geq4$。如果 $n\leq3$，则上述拆分不适用，需要单独处理。

- 如果 $n = 2$，则唯一的拆分方案是 $2 = 1 + 1$，最大乘积是 $1 \times 1 = 1$；
- 如果 $n = 3$，则拆分方案有 $3 = 1 + 2 = 1 + 1 + 1$，最大乘积对应方案 $3 = 1 + 2$，最大乘积是 $1 \times 2 = 2$。

这两种情形可以合并为：当 $n \leq 3$ 时，最大乘积是 $n - 1$。

归纳证明法

- 第一步：证明最优的拆分方案中不会出现大于 4 的整数。

> 假设出现了大于 4 的整数 x，由于 $2(x-2) > x$ 在 $x > 4$ 时恒成立，将 x 拆分成 2 和 $x - 2$ 可以增大乘积。因此最优的拆分方案中不会出现大于 4 的整数。

- 第二步：证明最优的拆分方案中可以不出现整数 4。

> 如果出现了整数 4，我们可以用 2×2 代替之，乘积不变。

此时，我们可以知道，最优的拆分方案中只会出现 1，2 和 3。

- 第三步：证明当 $n \geq 5$ 时，最优的拆分方案中不会出现整数 1。

> 当 $n \geq 5$ 时，如果出现了整数 1，那么拆分中剩余的数的和为 $n - 1 \geq 4$，对应这至少两个整数。我们将其中任意一个整数 x 加上 1，乘积就会增大。因此最优的拆分方案中不会出现整数 1。

此时，我们可以知道，当 $n \geq 5$ 时，最优的拆分方案中只会出现 2 和 3。

- 第四步：证明当 $n \geq 5$ 时，最优的拆分方案中 2 的个数不会超过 3 个。

> 如果出现了超过 3 个 2，那么将它们转换成 2 个 3，可以增大乘积，即$3 \times 3 > 2 \times 2 \times 2$。

此时，$n \geq 5$ 的最优拆分方案就唯一了。这是因为当最优的拆分方案中 2 的个数分别为 0，1，2 个时，就对应着 n 除以 3 的余数分别为 0，2，1 的情况。因此我们可以得到和「函数极值证明法」相同的分类讨论结果。

当 $n = 4$ 时，$4 = 2 \times 2$ 的最优拆分方案也可以放入分类讨论结果；当 $2 \leq n \leq 3$ 时，只有唯一的拆分方案 $1 \times (n - 1)$。

复杂度分析

- 时间复杂度：$O(1)$。涉及到的操作包括计算商和余数，以及幂次运算，时间复杂度都是常数。
- 空间复杂度：$O(1)$。只需要使用常数复杂度的额外空间。

$$res = \begin{cases} 3^{quo}, rema = 0 \\ 3^{quo-1} \times 4, rema = 1 \\ 3^{quo} \times 2, 其他 \end{cases}$$

方法 2：数学

```
class Solution {
    public int integerBreak(int n) {
        if (n <= 3) {
            return n - 1;
        }
        int quotient = n / 3; //商
        int remainder = n % 3; //余数
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            if (remainder == 0) {
                return (int) Math.pow(3, quotient);
            } else if (remainder == 1) {
                return (int) Math.pow(3, quotient - 1) * 4;
            } else {
                return (int) Math.pow(3, quotient) * 2;
            }
        }
}
```

27、剪绳子（剑指 14 - 1）【贪心】

剑指 Offer 14- I. 剪绳子

难度 中等 88 收藏 分享 切换为英文 关注 反馈

给你一根长度为 `n` 的绳子，请把绳子剪成整数长度的 `m` 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 `k[0],k[1]...k[m-1]` 。请问 `k[0]*k[1]*...*k[m-1]` 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

示例 1：

```
输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1
```

示例 2:

```
输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36
```

提示：

- `2 <= n <= 58`

注意：本题与主站 343 题相同：https://leetcode-cn.com/problems/integer-break/

https://leetcode-cn.com/problems/jian-sheng-zi-lcof/solution/mian-shi-ti-14-i-jian-sheng-zi-tan-xin-si-xiang-by/

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

- 设将长度为 n 的绳子切为 a 段：

$$n = n_1 + n_2 + ... + n_a$$

- 本题等价于求解：

$$\max(n_1 \times n_2 \times ... \times n_a)$$

> 以下数学推导总体分为两步：① 当所有绳段长度相等时，乘积最大。② 最优的绳段长度为 3 。

数学推导：

- 以下公式为“算术几何均值不等式”，等号当且仅当 $n_1 = n_2 = ... = n_a$ 时成立。

$$\frac{n_1 + n_2 + ... + n_a}{a} \geq \sqrt[a]{n_1 n_2 ... n_a}$$

> **推论一：** 将绳子 **以相等的长度等分为多段** ，得到的乘积最大。

- 设将绳子按照 x 长度等分为 a 段，即 $n = ax$ ，则乘积为 x^a 。观察以下公式，由于 n 为常数，因此当 $x^{\frac{1}{x}}$ 取最大值时， 乘积达到最大值。

$$x^a = x^{\frac{n}{x}} = (x^{\frac{1}{x}})^n$$

- 根据分析，可将问题转化为求 $y = x^{\frac{1}{x}}$ 的极大值，因此对 x 求导数。

$$\begin{aligned}
\ln y &= \frac{1}{x}\ln x && \text{取对数} \\
\frac{1}{y}\dot{y} &= \frac{1}{x^2} - \frac{1}{x^2}\ln x && \text{对 } x \text{ 求导} \\
&= \frac{1 - \ln x}{x^2} \\
\dot{y} &= \frac{1 - \ln x}{x^2} x^{\frac{1}{x}} && \text{整理得}
\end{aligned}$$

- 令 $\dot{y} = 0$ ，则 $1 - \ln x = 0$ ，易得驻点为 $x_0 = e \approx 2.7$ ；根据以下公式，可知 x_0 为极大值点。

$$\dot{y} \begin{cases} > 0 & , x \in [-\infty, e) \\ < 0 & , x \in (e, \infty] \end{cases}$$

- 由于切分长度 x 必须为整数，最接近 e 的整数为 2 或 3 。如下式所示，代入 $x = 2$ 和 $x = 3$ ，得出 $x = 3$ 时，乘积达到最大。

$$\begin{aligned}
y(3) &= 3^{1/3} \approx 1.44 \\
y(2) &= 2^{1/2} \approx 1.41
\end{aligned}$$

- 口算对比方法：给两数字同时取 6 次方，再对比。

$$\begin{aligned}
[y(3)]^6 &= (3^{1/3})^6 = 9 \\
[y(2)]^6 &= (2^{1/2})^6 = 8
\end{aligned}$$

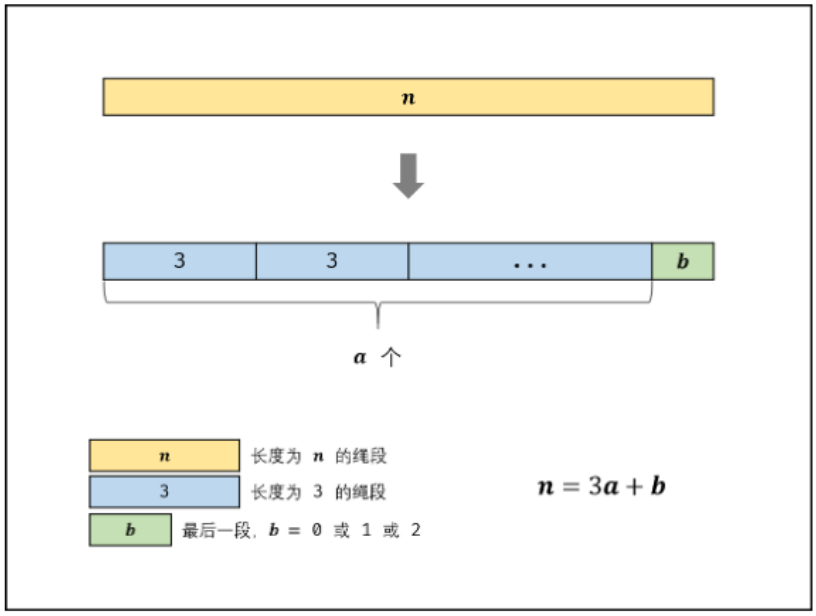
> **推论二：** 尽可能将绳子以长度 3 等分为多段时，乘积最大。

切分规则：

1. **最优：** 3 。把绳子尽可能切为多个长度为 3 的片段，留下的最后一段绳子的长度可能为 $0, 1, 2$ 三种情况。
2. **次优：** 2 。若最后一段绳子长度为 2 ；则保留，不再拆为 $1 + 1$ 。
3. **最差：** 1 。若最后一段绳子长度为 1 ；则应把一份 $3 + 1$ 替换为 $2 + 2$，因为 $2 \times 2 > 3 \times 1$。

算法流程：

1. 当 $n \leq 3$ 时，按照规则应不切分，但由于题目要求必须剪成 $m > 1$ 段，因此必须剪出一段长度为 1 的绳子，即返回 $n - 1$。
2. 当 $n > 3$ 时，求 n 除以 3 的 整数部分 a 和 余数部分 b（即 $n = 3a + b$），并分为以下三种情况：
   - 当 $b = 0$ 时，直接返回 3^a；
   - 当 $b = 1$ 时，要将一个 $1 + 3$ 转换为 $2 + 2$，因此返回 $3^{a-1} \times 4$；
   - 当 $b = 2$ 时，返回 $3^a \times 2$。



复杂度分析：

- **时间复杂度 $O(1)$：** 仅有求整、求余、次方运算。
  - 求整和求余运算：资料提到不超过机器数的整数可以看作是 $O(1)$；
  - 幂运算：查阅资料，提到浮点取幂为 $O(1)$。
- **空间复杂度 $O(1)$：** 变量 `a` 和 `b` 使用常数大小额外空间。

方法 1：数学推导

```
class Solution {
    public int cuttingRope(int n) {
        if(n <= 3) return n - 1;
        int a = n / 3, b = n % 3;
        if(b == 0) return (int)Math.pow(3, a);
        if(b == 1) return (int)Math.pow(3, a - 1) * 4;
        return (int)Math.pow(3, a) * 2;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

> 数学推导需要一定的知识基础。下面分享一种基于贪心思想的思路，个人认为适合于时间有限情况下的快速解题。

贪心思路：

> 设一绳子长度为 $n\ (n>1)$，则其必可被切分为两段 $n=n_1+n_2$。
> 根据经验推测，切分的两数字乘积往往原数字更大，即往往有 $n_1\times n_2>n_1+n_2=n$。
> - **例如绳子长度为 6：** $6=3+3<3\times 3=9$；
> - **也有少数反例，例如 2：** $2=1+1>1\times 1=1$。

- **推论一：** 合理的切分方案可以带来更大的乘积。

> 设一绳子长度为 $n\ (n>1)$，**切分为两段** $n=n_1+n_2$，**切分为三段** $n=n_1+n_2+n_3$。
> 根据经验推测，**三段** 的乘积往往更大，即往往有 $n_1n_2n_3>n_1n_2$。
> - **例如绳子长度为 9：** 两段 $9=4+5$ 和 三段 $9=3+3+3$，则有 $4\times 5<3\times 3\times 3$。
> - **也有少数反例，例如 6：** 两段 $6=3+3$ 和 三段 $6=2+2+2$，则有 $3\times 3>2\times 2\times 2$。

- **推论二：** 若切分方案合理，绳子段切分的越多，乘积越大。

> 总体上看，貌似长绳子切分为越多段乘积越大，但其实到某个长度分界点后，乘积到达最大值，就不应再切分了。
> **问题转化：** 是否有**优先级最高的长度** x 存在？若有，则应该尽可能把绳子以 x 长度切为多段，以获取最大乘积。

- **推论三：** 为使乘积最大，只有长度为 2 和 3 的绳子不应再切分，且 3 比 2 更优 *（详情见下表）*。

绳子切分方案	乘积	结论
$2=1+1$	$1\times 1=1$	2 不应切分
$3=1+2$	$1\times 2=2$	3 不应切分
$4=2+2=1+3$	$2\times 2=4>1\times 3=3$	4 和 2 等价，且 $2+2$ 比 $1+3$ 更优
$5=2+3=1+4$	$2\times 3=6>1\times 4=4$	5 应切分为 $2+3$
$6=3+3=2+2+2$	$3\times 3=9>2\times 2\times 2=8$	6 应切分为 $3+3$，进而**推出 3 比 2 更优**
>7	...	**长绳**（长度>7）可转化为多个**短绳**（长度1~6），因此肯定应切分

原文链接：https://blog.csdn.net/weixin_45405425/article/details/104416883

贪心解法

把绳长 target 剪成 i 段的最大值为：Math.pow(a, i - b) * Math.pow(a + 1, b)

如：target=8 i=3 时，ans=2^1*3^2

然后在剪成 2 段、3 段...x 段中取最大值即可。

方法 2: 贪心

```
class Solution {
    public int cuttingRope(int n) {
        int res = 0;
        for (int i = 2; i <= n; i++){
            int a = n / i, b = n % i;
            int tmp = (int) (Math.pow(a, i - b) * Math.pow(a + 1, b));
            if (tmp > res) res = tmp;
        }
        return res;
    }
}
```

28、剪绳子 2（剑指 14 - 2）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 14- II. 剪绳子 II

难度 中等 34 收藏 分享 切换为英文 关注 反馈

给你一根长度为 `n` 的绳子，请把绳子剪成整数长度的 `m` 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 `k[0],k[1]...k[m - 1]` 。请问 `k[0]*k[1]*...*k[m - 1]` 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

```
输入：2
输出：1
解释：2 = 1 + 1, 1 × 1 = 1
```

示例 2:

```
输入：10
输出：36
解释：10 = 3 + 3 + 4, 3 × 3 × 4 = 36
```

提示：

- `2 <= n <= 1000`

注意：本题与主站 343 题相同：https://leetcode-cn.com/problems/integer-break/

禁止商用

https://leetcode-cn.com/problems/jian-sheng-zi-ii-lcof/solution/mian-shi-ti-14-ii-jian-sheng-zi-iitan-xin-er-fen-f/

大数求余解法：

> **大数越界：** 当 a 增大时，最后返回的 3^a 大小以指数级别增长，可能超出 `int32` 甚至 `int64` 的取值范围，导致返回值错误。
> **大数求余问题：** 在仅使用 `int32` 类型存储的前提下，正确计算 x^a 对 p 求余（即 $x^a \odot p$）的值。
> **解决方案：** *循环求余* 、 *快速幂求余* ，其中后者的时间复杂度更低，两种方法均基于以下求余运算规则推出：
>
> $$(xy) \odot p = [(x \odot p)(y \odot p)] \odot p$$

1. 循环求余：

- 根据求余运算性质推出（∵ 本题中 $x < p$，$\therefore x \odot p = x$）：

$$x^a \odot p = [(x^{a-1} \odot p)(x \odot p)] \odot p = [(x^{a-1} \odot p)x] \odot p$$

- **解析：** 利用此公式，可通过循环操作依次求 $x^1, x^2, ..., x^{a-1}, x^a$ 对 p 的余数，保证每轮中间值 `rem` 都在 `int32` 取值范围中。封装方法代码如下所示。
- **时间复杂度 $O(N)$：** 其中 $N = a$，即循环的线性复杂度。

```
# 求 (x^a) % p —— 循环求余法
def remainder(x, a, p):
    rem = 1
    for _ in range(a):
        rem = (rem * x) % p
    return rem
```

2. 快速幂求余：

- 根据求余运算性质可推出：

$$x^a \odot p = (x^2)^{a/2} \odot p = (x^2 \odot p)^{a/2} \odot p$$

- 当 a 为奇数时 $a/2$ 不是整数，因此分为以下两种情况（"//" 代表向下取整的除法）：

$$x^a \odot p = \begin{cases} (x^2 \odot p)^{a//2} \odot p & , a \text{ 为偶数} \\ [(x \odot p)(x^{a-1} \odot p)] \odot p = [x(x^2 \odot p)^{a//2}] \odot p & , a \text{ 为奇数} \end{cases}$$

- **解析：** 利用以上公式，可通过循环操作每次把指数 a 问题降低至指数 $a//2$ 问题，只需循环 $log_2(N)$ 次，因此可将复杂度降低至对数级别。封装方法代码如下所示。

```
# 求 (x^a) % p —— 快速幂求余
def remainder(x, a, p):
    rem = 1
    while a > 0:
        if a % 2: rem = (rem * x) % p
        x = x ** 2 % p
        a //= 2
    return rem
```

- **帮助理解：** 根据下表，初始状态 $rem = 1, x = 3, a = 19, p = 1000000007$，最后会将 $rem \times (x^a \odot p)$ 化为 $rem \times (x^0 \odot p) = rem \times 1$ 的形式，即 rem 为余数答案。

n	$rem \times (x^a \odot p)$	$rem_n = rem_{n-1} \times x_{n-1} \odot p$	$x_n = x_{n-1}^2 \odot p$	$a_n = a_{n-1}//2$
1	$1 \times (3^{19} \odot p)$	1	3	19
2	$3 \times (9^9 \odot p)$	$3 = 1 \times 3 \odot p$	$9 = 3^2 \odot p$	$9 = 19//2$
3	$27 \times (81^4 \odot p)$	$27 = 3 \times 9 \odot p$	$81 = 9^2 \odot p$	$4 = 9//2$
4	$27 \times (6561^2 \odot p)$	27	$6561 = 81^2 \odot p$	$2 = 4//2$
5	$27 \times (43046721^1 \odot p)$	27	$43046721 = 6561^2 \odot p$	$1 = 2//2$
6	$162261460 \times (175880701^0 \odot p)$	$162261460 = 27 \times 43046721 \odot p$	$175880701 = 43046721^2 \odot p$	$0 = 1//2$

复杂度分析：

> 以下为二**分求余法**的复杂度。

- **时间复杂度** $O(\log_2 N)$：其中 $N = a$，二分法为对数级别复杂度，每轮仅有求整、求余、次方运算。
  - 求整和求余运算：资料提到不超过机器数的整数可以看作是 $O(1)$；
  - 幂运算：查阅资料，提到浮点取幂为 $O(1)$。
- **空间复杂度** $O(1)$：变量 `a, b, p, x, rem` 使用常数大小额外空间。

代码：

> **Python 代码（第三栏）：** 由于语言特性，理论上 Python 中的变量取值范围由系统内存大小决定（无限大），因此在 Python 中其实不用考虑大数越界问题。
> **Java 代码：** 根据二分法计算原理，至少要保证变量 `x` 和 `rem` 可以正确存储 1000000007^2，而 $2^{64} > 1000000007^2 > 2^{32}$，因此我们选取 `long` 类型。

方法 1：数学推导

```
class Solution {
    public int cuttingRope(int n) {
        if(n <= 3) return n - 1;
        int b = n % 3, p = 1000000007;
```

```
        long rem = 1, x = 3;
        for(int a = n / 3 - 1; a > 0; a /= 2) {
            if(a % 2 == 1) rem = (rem * x) % p;
            x = (x * x) % p;
        }
        if(b == 0) return (int)(rem * 3 % p);
        if(b == 1) return (int)(rem * 4 % p);
        return (int)(rem * 6 % p);
    }
}
```

https://leetcode-cn.com/problems/jian-sheng-zi-ii-lcof/solution/javatan-xin-si-lu-jiang-jie-by-henrylee4/

DP的缺陷

这个题和剪绳子I一样的描述，就是数据范围变大了。 `剪绳子` 可以用动态规划或者贪心做，这道题对于使用DP难度就增大了一些，因为数据范围变得比较大时，long已经不足以去存储中间结果的状态，但是由于DP做法是枚举各种剪的情况然后取最大值，因此只能通过使用 `BigInteger` 的方法去做，这点评论区已经有人给出了解答。

那么这个题范围变大的本意是想让我们使用贪心算法能更好的求解(毕竟 `BigInteger` 使用起来麻烦，贪心没有数据溢出的问题，它是找当下的最优解，不需要比较，**中间结果可以直接取模**)。

贪心

我们首先考虑对于一段长n的绳子，我们可以切出的结果包含什么？

`1` 会包含吗？ 不会，因为 `1 * (k - 1) < k` ，只要把1和任何一个其他的片段组合在一起就有个更大的值
`2` 可以
`3` 可以
`4` 可以吗？ 它拆成两个2的效果和本身一样，因此也不考虑
`5` 以上可以吗？ 不可以，这些绳子必须拆，因为总有一种拆法比不拆更优，比如拆成 `k / 2` 和 `k - k / 2`

综上，最后的结果只包含2和3(当然当总长度为2和3时单独处理)，那么很显然 `n >= 5时， 3*(n - 3) >= 2 * (n - 2)` ，因此我们优先拆成3，最后剩余的拆成2。最后的结果一定是由*若干个* `3` *和1或2个* `2` 组成.

方法 2：贪心

```
class Solution {
    public int cuttingRope(int n) {
        if(n <= 3) return n - 1;
        int mod = 1000000007;
        long res = 1;
        while(n > 4) {
            res *= 3;
            res %= mod;
            n -= 3;
        }
        return (int)(res * n % mod);
    }
}
```

29、分组选项（linkcode1834）

1834. 分组选项 中文 English

现在有n个人在排成一排。他们必须从左到右分成连续的m组。如果每个组的人数都必须大于等于其左边的组的人数，
请问一共有多少种不同的分组方式。对于两个分组而言，当且仅当按递增序排列以后是不同的，他才是不同的分组。例如，`[1, 1, 1, 3]`和`[1, 1, 1, 2]`不同，但和`[1, 3, 1, 1]`相同。

样例
```
Input:
8
4
Output: 5
Explanation: [1, 1, 1, 5], [1, 1, 2, 4], [1, 1, 3, 3], [1, 2, 2, 3], [2, 2, 2, 2]
```
注意事项
$1 \leq n, m \leq 200$

动态规划，$dp[i][j]$表示i个人分成j组的方案数，则dp如下：

- 边界条件：$dp[i][i] = 1, dp[i][j] = 0(i < j)$
- 转移方程：$dp[i][j] = \sum_i^j dp[i-j][k]$
- 答案：$dp[n][m]$

```
public class Solution {
    /**
     * @param n: the number of people
     * @param m: the number of groups
     * @return: the number of grouping options
     */
    public long groupingOptions(int n, int m) {
        // write your code here
        if (m > n) return 0;
        long dp[][] = new long[n+1][n+1];
        for (int i = 1; i <= n; i++) dp[i][i] = 1;
        for (int i = 2; i <= n; i++) {
            for (int j = 1; j < i; j++) {
                for (int k = 1; k <= j; k++) {
                    dp[i][j] += dp[i-j][k];
                }
            }
        }
        return dp[n][m];
    }
}
```

30、工作安排（lintcode1147）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

描述

中文 English

小美是团队的负责人，需要为团队制定工作的计划，来帮助团队产生最大的价值。每周团队都会有两项候选的任务,其中一项为简单任务,一项为复杂任务,两项任务都能在一周内完成。在第 `i` 周,团队完成简单任务的价值为low_i,完成复杂任务的价值为$high_i$。

由于复杂任务本身的技术难度较高，团队如果在第 `i` 周选择执行复杂任务的话,需要在 `i-1` 周不做任何任务来专心准备。

如果团队在第 `i` 周选择执行简单任务的话,不需要提前做任何准备。现在小美的团队收到了未来 `n` 周的候选任务列表,

请帮助小美确定每周的工作安排使得团队的工作价值最大。

$1 \leq |low|, |high| \leq 10000$
$1 \leq low[i], high[i] \leq 10000$

您在真实的面试中是否遇到过这个题？ 是 题目纠错

样例

例1:

```
输入:low=[4,2,3,7],hard=[3,5,6,9]
输出:17
解释:在第一周的时候挑选简单任务，价值=4
在第二周做准备，第三周挑选复杂任务，价值=4+6=10
在第四周挑选简单任务，价值=10+7=17
```

例2:

```
输入:low=[1,3,5,9],high=[3,5,7,9]
输出:19
```

禁止商用 公众号：yes的练级攻略

```
public class Solution {
    /**
     * @param low: the simple task
     * @param high: the complex task
     * @return: the most value
     */
    public int workPlan(int[] low, int[] high) {
        // Write your code here.
        int n = low.length;
        int[] dp = new int[n+5];
        int i;
        for (i = 0;i <= n + 1;i++){
            dp[i] = 0;
        }
        dp[1] = low[0];
        for (i = 2; i <= n; i++) {
            dp[i] = Math.max(dp[i - 1] + low[i - 1], dp[i - 2] + high[i - 1]);
        }
        return dp[n];
    }
}
```

31、方案数量（Lintcode1528）

1528. 方案数量

给定一个严格单调上升的数组$position$，两个整数m，$distance$
从$position$ 中选出长度为m的子序列作为A数组 要求：$(A[i]-A[i-1]) \leq distance \quad 1 \leq i \leq m-1$
请给出符合条件的方案数量

样例

输入:[1,2,3,4]
3
3
输出:4
解释:可选方案 (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)

注意事项
A中有一个数不同就认为是不同方案
方案数过大请对99997867取模输出
$2 \leq position.length \leq 5000$
$2 \leq m \leq 100$
$1 \leq distance \leq 10^7$
$1 \leq position[i] \leq 10^7$

助教链接：https://paste.ubuntu.com/p/ypjS3sDtwv/【不能用双指针】
now 和 last 是我们使用滚动数组来节省空间的

```
public class Solution {
    /**
     * @param position:  a  int array
     * @param m: selected number
     * @param distance: the max distance
     * @return: return the number of schemes
     */
    public int getnumberschemes(int[] position, int m, int distance) {
        // write your code here
        int mod = 99997867;
        int n = position.length;
        int[][] dp = new int[2][n+1]; //前面的是新旧区别，后面是位置
        Queue<Integer>q = new LinkedList<Integer>();
        int now = 0, last = 1; //更新下标，一直用 dp[last]去更新 dp[now]
        //dp 数组边界条件
        for (int i = 0; i < n; i++) {
            dp[now][i] = 1;
        }
        //交换 now last，一直用 dp[last]去更新 dp[now]
        now = now ^ 1;
        last = last ^ 1;
        for(int i = 1; i < m; i++) {
            //清空本次被更新的 dp[now] 避免之前的值的影响
```

```
            for(int j = 0; j < n; j++) {
                dp[now][j] = 0;
            }
            //清空队列，避免上次循环留在队列的元素的影响
            while(!q.isEmpty()) {
                q.poll();
            }
            //队列里元素的和
            int tmp = 0;
            //将 0 到 i-1 压入队列
            for(int j = 0; j < i; j++) {
                tmp = (tmp + dp[last][j]) % mod;
                q.add(j);
            }
            //j 从 i 开始，因为 j>=i  才能选出 i 个元素来
            for(int j = i; j < n; j++) {
                //判断队列头部是否能更新 j
                while(!q.isEmpty() && position[j] - position[q.peek()] > distance) {
                    //弹出队首  更新 tmp
                    tmp -= dp[last][q.peek()];
                    tmp = (tmp + mod) % mod;
                    q.poll();
                }
                //更新 dp[i][j]
                dp[now][j] = tmp;
                //更新 tmp，压入 j
                tmp = (tmp + dp[last][j]) % mod;
                q.add(j);
            }
            //交换 now last，这样下次就是 dp[last]去更新 dp[now]
            now = now ^ 1;
            last = last ^ 1;
        }
        //最终答案要枚举最后一个元素从哪里选的
        int ans = 0;
        for(int i = m - 1; i < n; i++) {
            //now last  最后交换，所以 dp[M-1]应该是 dp[last]
            ans = (ans + dp[last][i]) % mod;
        }
        return ans;
    }
}
```

32、最小调整代价（lintcode91）

禁止商用 公众号：yes的练级攻略

91. 最小调整代价 中文 English

给一个整数数组，调整每个数的大小，使得相邻的两个数的差不大于一个给定的整数target，调整每个数的代价为调整前后的差的绝对值，求调整代价之和最小是多少。

样例
样例 1:
输入: [1,4,2,3], target=1
输出: 2

样例 2:
输入: [3,5,4,7], target=2
输出: 1

注意事项
你可以假设数组中每个整数都是正整数，且小于等于100。

https://www.jiuzhang.com/solution/minimum-adjustment-cost/#tag-lang-java

算法：dp

思路

- 已知每个整数范围 `[1,100]`，那么对于每个元素，为了调整到该元素和与之相邻的元素的差不大于target，该元素调整的范围就在 `[1,100]`。所以对于数组A[]的每一位元素，我们都需要进行 `[1,100]` 范围内的可能状态的转移。
- 令 `dp[i][j]` 表示元素 `A[i]=j` 时，`A[i]` 与 `A[i-1]` 差值不大于target所需要付出的最小代价。
- 当 `A[i]=j` 时，可行的 `A[i-1]` 的范围为 `[max(1, j - target), min(100, j + target)]`。而 `dp[i][j]` 为所有可行的 `A[i-1]` 中，花费代价最小的一种可能，再加上 `A[i]` 调整到 j 所需花费 `abs(j - A[i])`。
- 当 `A[i]=j` 时，k在 `[max(1, j - target), min(100, j + target)]` 范围内时，我们可以写出以下式子：

  1.临界值：

  ```
  dp[0][j] = abs(j - A[0])
  ```

  2.状态转移方程：

  ```
  dp[i][j] = min(dp[i][j], dp[i - 1][k] + abs(j - A[i]))
  ```

- 最后在所有最后一位的可能解 `dp[n-1][i]` 中的最小值，就是我们所求的最小代价。

复杂度

- 假设数组长度为n
- 空间复杂度O(10000*n)
- 时间复杂度O(n^2)

方法 1：常规 dp

```
public class Solution {
    /*
     * @param A: An integer array
     * @param target: An integer
     * @return: An integer
     */
    public int MinAdjustmentCost(List<Integer> A, int target) {
        int n = A.size();
        // dp[i][j]表示元素 A[i]=j 时，A[i]与 A[i-1]差值不大于 target 所需要付出的最小代价
        int[][] dp = new int[n][101];
        for (int i = 0; i < n; i++) {
            for (int j = 1; j <= 100; j++) {
                // 初始化为极大值
                dp[i][j] = Integer.MAX_VALUE;
            }
```

```
        }
        for (int i = 0; i < n; i++) {
            for (int j = 1; j <= 100; j++) {
                if (i == 0) {
                    // 临界值：第一个元素 A[0]调整为 j 的代价
                    dp[0][j] = Math.abs(j - A.get(0));
                }
                else {
                    // left 为 A[i]=j 时，A[i-1]与 A[i]差值不大于 target 的 A[i-1]最小值
                    // right 为 A[i]=j 时，A[i-1]与 A[i]差值不大于 target 的 A[i-1]最大值
                    int left = Math.max(1, j - target);
                    int right = Math.min(100, j + target);
                    for (int k = left; k <= right; k++) {
                        // 当 A[i-1]=k 时，答案为 A[i-1]=k 的代价 dp[i-1][k]，加上 A[i]=j
的调整代价 abs(j-A[i])
                        dp[i][j] = Math.min(dp[i][j], dp[i - 1][k] + Math.abs(j - A.get(i)));
                    }
                }
            }
        }
        int mincost = Integer.MAX_VALUE;
        for (int i = 1; i <= 100; i++) {
            mincost = Math.min(mincost, dp[n - 1][i]);
        }
        return mincost;
    }
}
```

方法 2：利用滚动数组

滚动数组优化空间复杂度。currentj means change the ith element in A to j. Using DP, currentj = prevp+Math.abs(j-A.get(i)), p is the (i-1)th number after change in A.

```
public class Solution {
    /*
     * @param A: An integer array
     * @param target: An integer
     * @return: An integer
     */
    public int MinAdjustmentCost(List<Integer> A, int target) {
        int[] pre = new int[101];
        for (int i = 0; i < A.size(); i++){
            int[] cur = new int[101];
            for (int j = 0; j <= 100; j++){
                cur[j] = Integer.MAX_VALUE;
                int lower = Math.max (0, j - target);
```

禁止商用 公众号：yes的练级攻略

```
                int upper = Math.min(100, j + target);
                for (int p = lower; p <= upper; p++){
                    cur[j] = Math.min(cur[j], pre[p] + Math.abs(j - A.get(i)));
                }
            }
            pre = cur;
        }
        int res = Integer.MAX_VALUE;
        for (int i = 0; i <= 100; i++){
            res = Math.min(res, pre[i]);
        }
        return res;
    }
}
```

33、按摩师（面试题 17.16）

面试题 17.16. 按摩师

难度 简单 115 收藏 分享 切换为英文 关注 反馈

一个有名的按摩师会收到源源不断的预约请求，每个预约都可以选择接或不接。在每次预约服务之间要有休息时间，因此她不能接受相邻的预约。给定一个预约请求序列，替按摩师找到最优的预约集合（总预约时间最长），返回总的分钟数。

注意：本题相对原题稍作改动

示例 1：

```
输入： [1,2,3,1]
输出： 4
解释： 选择 1 号预约和 3 号预约，总时长 = 1 + 3 = 4。
```

示例 2：

```
输入： [2,7,9,3,1]
输出： 12
解释： 选择 1 号预约、 3 号预约和 5 号预约，总时长 = 2 + 9 + 1 = 12。
```

示例 3：

```
输入： [2,1,4,5,3,1,1,3]
输出： 12
解释： 选择 1 号预约、 3 号预约、 5 号预约和 8 号预约，总时长 = 2 + 4 + 3 + 3 = 12。
```

https://leetcode-cn.com/problems/the-masseuse-lcci/solution/100-kong-jian-cong-onyou-hua-dao-o1bao-hui-by-swee/

递推方程：`dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]`

2. 空间优化到 $O(1)$

```java
class Solution {
    public int massage(int[] nums) {
        int a = 0, b = 0;
        for (int i = 0; i < nums.length; i++) {
            int c = Math.max(b, a + nums[i]);
            a = b;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            b = c;
        }
        return b;
    }
}
```

34、打家劫舍（198）

198. 打家劫舍

难度 简单 920 收藏 分享 切换为英文 关注 反馈

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**

给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下** ，一夜之内能够偷窃到的最高金额。

示例 1：

输入： [1,2,3,1]
输出： 4
解释： 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
偷窃到的最高金额 = 1 + 3 = 4 。

示例 2：

输入： [2,7,9,3,1]
输出： 12
解释： 偷窃 1 号房屋 (金额 = 2)，偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
偷窃到的最高金额 = 2 + 9 + 1 = 12 。

https://leetcode-cn.com/problems/house-robber/solution/liang-ge-yue-0ji-chu-cong-an-mo-shi-zhuan-xing-zi-/

题目分析

读题知输入一个长度为 n 的数组，表示 n 个房屋，让我们求到第 n 个房屋的最大金额。

我们想一下到第n个房屋的最大金额 dp[n]:
1. 如果偷第 n 个房屋的话，由于相邻的房屋不能偷，所以 dp[n] = dp[n - 2] + nums[n]（即等于到第 n - 2 个房屋的最大金额 + 第 n 个房屋的金额）
2. 反之如果不偷第 n 个房屋的话，那么 dp[n] = dp[n - 1]（即等于到第 n - 1 个房屋的最大金额）

于是，我们得到了 **状态转移方程：dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])**

一、首先实现一个便于理解的基础版本的 dp，时间 $O(N)$，空间 $O(N)$。

ok，就是这么简单，但此时我们发现了一个问题，i 其实只依赖于 i - 1 和 i - 2 两个状态，因此我们其实不需要定义一个长度为 n 的 dp 数组，只要定义两个变量来存储 i - 1 和 i - 2 不就行了嘛，于是这样就把空间优化到了 $O(1)$ ~

方法 1：o(N)空间复杂度

```
class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        int n = nums.length;
        int[] dp = new int[n+1];
```

```
        dp[0] = 0;
        dp[1] = nums[0];

        for (int i = 2; i <= n; i++) {
            dp[i] = Math.max(dp[i-1], dp[i-2] + nums[i-1]);
        }

        return dp[n];
    }
}
```

方法 2：o(1)空间复杂度

因为只有 dp[i],dp[i-1],dp[i-2]，所以可以用常量代替。

二、因此，实现一版空间优化到 $O(1)$ 的代码。

还是参照着 dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]) 这个方程来写，

用 a 表示 i - 2，b 表示 i - 1，

因此上面的方程变成了dp[i] = max(b, a + nums[i]),

然后下一轮循环的时候：
i - 1 就变成了i - 2，因此现在需要把 b 赋值给 a；
i 就变成了 i - 1，因此现在需要把 dp[i] 赋值给 b。

```
class Solution {
    public int rob(int[] nums) {
        int a = 0, b = 0;
        for (int i = 0; i < nums.length; i++) {
            int c = Math.max(b, a + nums[i]);
            a = b;
            b = c;
        }
        return b;
    }
}
```

35、打家劫舍 2（213）

213. 打家劫舍 II

难度 中等 308 收藏 分享 切换为英文 关注 反馈

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都**围成一圈**，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**

给定一个代表每个房屋存放金额的非负整数数组，计算你**在不触动警报装置的情况下**，能够偷窃到的最高金额。

示例 1:

```
输入：[2,3,2]
输出：3
解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。
```

示例 2:

```
输入：[1,2,3,1]
输出：4
解释：你可以先偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/house-robber-ii/solution/213-da-jia-jie-she-iidong-tai-gui-hua-jie-gou-hua-/

总体思路：

- **此题是 198. 打家劫舍 的拓展版：** 唯一的区别是此题中的房间是**环状排列**的（即首尾相接），而 198. 题中的房间是**单排排列**的；而这也是此题的难点。
- **环状排列**意味着第一个房子和最后一个房子中**只能选择一个偷窃**，因此可以把此**环状排列房间**问题约化为两个**单排排列房间**子问题：
  1. 在不偷窃第一个房子的情况下（即 $nums[1:]$），最大金额是 p_1 ；
  2. 在不偷窃最后一个房子的情况下（即 $nums[:n-1]$），最大金额是 p_2 。
  - **综合偷窃最大金额：** 为以上两种情况的较大值，即 $max(p1, p2)$ 。
- 下面的任务则是解决 **单排排列房间（即 198. 打家劫舍）** 问题。推荐可以先把 198. 做完再做这道题。

198. 解题思路：

典型的动态规划，以下按照标准流程解题。

- **状态定义：**
  - 设动态规划列表 dp ， $dp[i]$ 代表前 i 个房子在满足条件下的能偷窃到的最高金额。
- **转移方程：**
  - **设：** 有 n 个房子，前 n 间能偷窃到的最高金额是 $dp[n]$ ，前 $n-1$ 间能偷窃到的最高金额是 $dp[n-1]$ ，此时向这些房子后加一间房，此房间价值为 num ；
  - **加一间房间后：** 由于不能抢相邻的房子，意味着抢第 $n+1$ 间就不能抢第 n 间；那么前 $n+1$ 间房能偷取到的最高金额 $dp[n+1]$ 一定是以下两种情况的 **较大值** ：
    1. 不抢第 $n+1$ 个房间，因此等于前 n 个房子的最高金额，即 $dp[n+1]=dp[n]$ ；
    2. 抢第 $n+1$ 个房间，此时不能抢第 n 个房间；因此等于前 $n-1$ 个房子的最高金额加上当前房间价值，即 $dp[n+1]=dp[n-1]+num$ ；
  - **细心的我们发现：** 难道在前 n 间的最高金额 $dp[n]$ 情况下，第 n 间一定被偷了吗？假设没有被偷，那 $n+1$ 间的最大值应该也可能是 $dp[n+1]=dp[n]+num$ 吧？其实这种假设的情况可以被省略，这是因为：
    1. 假设第 n 间没有被偷，那么此时 $dp[n]=dp[n-1]$ ，此时 $dp[n+1]=dp[n]+num=dp[n-1]+num$ ，即可以将 **两种情况合并为一种情况** 考虑；
    2. 假设第 n 间被偷，那么此时 $dp[n+1]=dp[n]+num$ **不可取** ，因为偷了第 n 间就不能偷第 $n+1$ 间。
  - **最终的转移方程：** $dp[n+1]=max(dp[n], dp[n-1]+num)$
- **初始状态：**
  - 前 0 间房子的最大偷窃价值为 0 ，即 $dp[0]=0$ 。
- **返回值：**
  - 返回 dp 列表最后一个元素值，即所有房间的最大偷窃价值。
- **简化空间复杂度：**
  - 我们发现 $dp[n]$ 只与 $dp[n-1]$ 和 $dp[n-2]$ 有关系，因此我们可以设两个变量 `cur` 和 `pre` 交替记录，将空间复杂度降到 $O(1)$ 。

复杂度分析：

- **时间复杂度** $O(N)$：两次遍历 `nums` 需要线性时间；
- **空间复杂度** $O(1)$：`cur` 和 `pre` 使用常数大小的额外空间。

```
class Solution {
    public int rob(int[] nums) {
        if(nums.length == 0) return 0;
        if(nums.length == 1) return nums[0];
        return Math.max(myRob(Arrays.copyOfRange(nums, 0, nums.length - 1)),
                        myRob(Arrays.copyOfRange(nums, 1, nums.length)));
    }
    private int myRob(int[] nums) {
        int pre = 0, cur = 0, tmp;
        for(int num : nums) {
            tmp = cur;
            cur = Math.max(pre + num, cur);
            pre = tmp;
        }
        return cur;
    }
}
```

36、打家劫舍 3（337）【二叉树】

更多硬核内容就在公众号「yes 的练级攻略」

337. 打家劫舍 III

难度 中等 407 收藏 分享 切换为英文 关注 反馈

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

```
输入: [3,2,3,null,3,null,1]

     3
    / \
   2   3
    \   \
     3   1

输出: 7
解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.
```

示例 2:

```
输入: [3,4,5,1,3,null,1]

     3
    / \
   4   5
  / \   \
 1   3   1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

https://leetcode-cn.com/problems/house-robber-iii/solution/shu-xing-dp-ru-men-wen-ti-by-liweiwei1419/

先说重点：

- 这里最关键的是理解：在设计状态的时候，**在后面加一维，消除后效性**，打家劫舍第 1 题，股票系列问题 6 个问题，都是这样的思路；
- 树的问题，很多时候采用**后序遍历**。

关于后效性，我在下面这两篇题解里有详细叙述：

- 152. 乘积最大子数组 题解：动态规划（理解无后效性）
- 面试题 17.16. 按摩师 题解：动态规划（经典问题，掌握如何消除后效性）。

分析：（这里略过暴力解法和记忆化递归。）

- 根据打家劫舍 I 和 II，我们有了经验，这是一个 dp 问题；
- 问题场景在「树」上，就要用到「树的遍历」，这里用「后序遍历」，这是因为：**我们的逻辑是子结点陆续汇报信息给父结点，一层一层向上汇报，最后在根结点汇总值。**

关键：当前结点「偷」或者「不偷」决定了孩子结点偷或者不偷，把这一点设计成状态，放在第 2 维，这一步叫「消除后效性」，这一点技巧非常常见。

第 1 步：状态定义

`dp[node][j]` ：这里 `node` 表示一个结点，以 `node` 为根结点的树，并且规定了 `node` 是否偷取能够获得的最大价值。

- `j = 0` 表示 `node` 结点不偷取；
- `j = 1` 表示 `node` 结点偷取。

第 2 步： 推导状态转移方程

根据当前结点偷或者不偷，就决定了需要从哪些**子结点**里的对应的状态转移过来。

- 如果当前结点不偷，左右子结点偷或者不偷都行，选最大者；
- 如果当前结点偷，左右子结点均不能偷。

（状态转移方程的表述有点复杂，请大家直接看代码。）

第 3 步： 初始化

一个结点都没有，空节点，返回 0，对应后序遍历时候的递归终止条件；

第 4 步： 输出

在根结点的时候，返回两个状态的较大者。

第 5 步： 思考优化空间

优化不了。

```
class Solution {
    public int rob(TreeNode root) {
        int[] res = dfs(root);
        return Math.max(res[0], res[1]);
    }
    private int[] dfs(TreeNode node) {
        if (node == null) return new int[2];
        // 分类讨论的标准是：当前结点偷或者不偷
        // 由于需要后序遍历，所以先计算左右子结点，然后计算当前结点的状态值
        int[] left = dfs(node.left);
        int[] right = dfs(node.right);

        // dp[0]：以当前 node 为根结点的子树能够偷取的最大价值，规定 node 结点不偷
        // dp[1]：以当前 node 为根结点的子树能够偷取的最大价值，规定 node 结点偷
        int[] dp = new int[2];

        dp[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
        dp[1] = node.val + left[0] + right[0];
        return dp;
    }
```

}

37、找零（lintcode1503）【贪心】

1503. 找零

某国的货币系统包含面值 1 元、4 元、16 元、64 元共 4 种硬币，以及面值 1024 元的纸币。
你现在使用 1024 元的纸币购买了一件价值为 $N, 0 < N <= 1024$ 元的商品，请问最少会收到多少个硬币作为找零。

样例
样例输入1:
amount = 1014

样例输出1:
4

找零 2 个 4 元硬币，和 2 个 1 元硬币。
样例输入2:
amount = 1004

样例输出2:
2

找零 1 个 16 元硬币，1 个 4 元硬币。

```
public class Solution {
    /**
     * @param amount: The amount you should pay.
     * @return: Return the minimum number of coins for change.
     */
    public int giveChange(int amount) {
        int[] coins = new int[] {1, 4, 16, 64};
        int change = 1024 - amount;
        int[] dp = new int [change + 1];
        Arrays.fill(dp, change + 1);
        dp[0] = 0;
        for (int i = 1; i < change + 1; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (coins[j] <= i) {
                    dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }
        return dp[change];
    }
}
```

这题的最优解是用贪心算法【给的零钱面值是倍数关系的用贪心】

https://www.jiuzhang.com/solution/give-change/

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

解题思路

我们可以算出要找的钱为 $1024 - amount$，由于所有的硬币的面值两两成倍数关系，所以一定尽量每次优先用最大的面值去找零，如果最大的面值大于剩下的金额数，再用第二大的面值。利用贪心思想，从大到小进行计算。

代码思路

1. 令常量 $VALUES = [64, 16, 4, 1]$。
2. 从大到小遍历，将 $number$ 加上 $change/value$，代表需要多少个面值 $value$ 的硬币，$change \mod value$ 代表用这种面值后剩下多少金额。

复杂度分析

时间复杂度

- 只需常量次数的计算，时间复杂度为 `O(1)`。

空间复杂度

- 需要常量空间进行计算，空间复杂度为 `O(1)`。

```
public class Solution {
    /**
     * @param amount: The amount you should pay.
     * @return: Return the minimum number of coins for change.
     */
    public int giveChange(int amount) {
        List<Integer> value = new ArrayList<>();
        value.add(64);
        value.add(16);
        value.add(4);
        value.add(1);
        int change = 1024 - amount;
        int number = 0;
        for (int i = 0; i < 4; i++) {
            int curValue = (int) value.get(i);
            number += change / curValue;
            change %= curValue;
        }
        return number;
    }
}
```

38、不同路径（62）求路径条数

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

62. 不同路径

难度 中等 596 收藏 分享 切换为英文 关注 反馈

一个机器人位于一个 *m x n* 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径?

例如，上图是一个7 x 3 的网格。有多少可能的路径?

示例 1:

```
输入: m = 3, n = 2
输出: 3
解释:
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右
```

示例 2:

```
输入: m = 7, n = 3
输出: 28
```

提示:

- `1 <= m, n <= 100`
- 题目数据保证答案小于等于 `2 * 10 ^ 9`

yes的练级攻略

方法 1：二维数组，用甜姨的思路，与 64 题的类似

```java
class Solution {
    public int uniquePaths(int m, int n) {
        if (m == 0 || n == 0) return 1;
        int[][] dp = new int[m][n];
        for (int i = 0; i < m; i++){
            dp[i][0] = 1;
        }
        for (int j = 0; j < n; j++){
            dp[0][j] = 1;
        }
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
            }
        }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        return dp[m-1][n-1];
    }
}
```

方法 2：甜姨的代码压缩空间

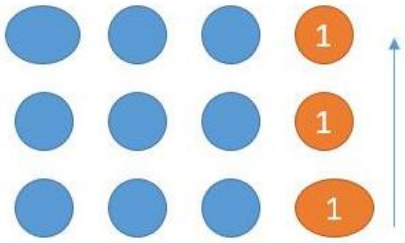
```
class Solution {
    public int uniquePaths(int m, int n) {
        if (m == 0 || n == 0) return 1;
        int[] dp = new int[n+1];
        dp[1] = 1;
        for (int i = 0; i < m; i++){
            for (int j = 1; j <= n; j++){
                dp[j] += dp[j-1];
            }
        }
        return dp[n];
    }
}
```

https://leetcode-cn.com/problems/unique-paths/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-20/

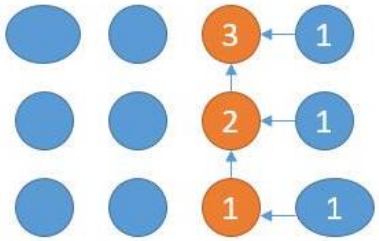
禁止商用
公众号：
yes的练级攻略

解法一是基于递归的，压栈浪费了很多时间。我们来分析一下，压栈的过程，然后我们其实完全可以省略压栈的过程，直接用迭代去实现。

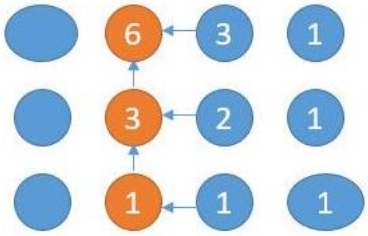
如下图，如果是递归的话，根据上边的代码，从 （0，0）点向右压栈，向右压栈，到最右边后，就向下压栈，向下压栈，到最下边以后，就开始出栈。出栈过程就是橙色部分。

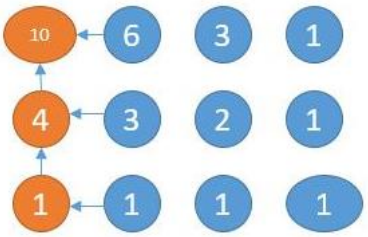


然后根据代码，继续压栈前一列，下图的橙色部分，然后到最下边后，然后开始出栈，根据它的右边的点和下边的点计算当前的点的走法。



接下来两步同理，压栈，出栈。





我们现在要做的就是要省略压栈的过程，直接出栈。很明显可以做到的，只需要初始化最后一列为 1 ，然后 1 列，1 列的向前更新就可以了。有一些动态规划的思想了。

时间复杂度：O（m * n）。

空间复杂度：O（m）。

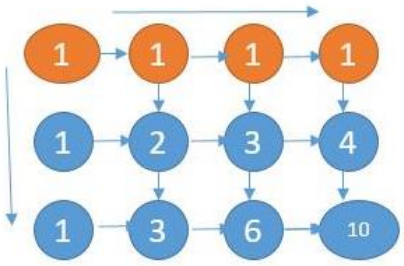
方法 3：另外一种解法，压缩空间，省略压栈的过程

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[] dp = new int[m];
        Arrays.fill(dp,1);
        for (int i = n - 2; i >= 0; i--) {
            for (int j = m - 2; j >= 0; j--) {
                dp[j] += dp[j+1];
            }
        }
        return dp[0];
    }
}
```

这里也有一个类似的想法。不过他是正向考虑的，和上边的想法刚好相反。如果把 dp[i][j]表示为从点 （0，0）到点（i，j）的走法。

上边解法公式就是 dp[i][j] = dp[i + 1][j] + dp[i][j +1]。

这里的话就是 dp[i][j] = dp[i - 1][j] + dp[i][j - 1]。就是用它左边的和上边的更新，可以结合下图。



这样的话，就是从左向右，从上到下一行一行更新（当前也可以一列一列更新）。

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[] dp = new int[n];
        Arrays.fill(dp,1);
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[j] += dp[j-1];
            }
        }
        return dp[n-1];
    }
}
```

39、不同路径 2（63）

63. 不同路径 II

难度 中等 385 收藏 分享 切换为英文 关注 反馈

一个机器人位于一个 *m x n* 网格的左上角 （起始点在下图中标记为“Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？

网格中的障碍物和空位置分别用 `1` 和 `0` 来表示。

说明： *m* 和 *n* 的值均不超过 100。

示例 1:

输入:
```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```
输出: 2
解释:
3x3 网格的正中间有一个障碍物。
从左上角到右下角一共有 2 条不同的路径：
1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右

yes的练级攻略

https://leetcode-cn.com/problems/unique-paths-ii/solution/jian-dan-dpbi-xu-miao-dong-by-sweetiee/

一、题目分析

递归思路：

假设我们定义到达右下角的走法数为 $f(m,n)$，因为右下角只能由它上方或者左方的格子走过去，因此可以很容易的写出递归求解式，即 $f(m,n)=f(m-1,n)+f(m,n-1)$，最后加上递归终止条件，SO EASY 看起来大功告成啦！

然而事情并木有结束～ 因为这样自底向上的递归会存在大量的重复计算，所以我们将其改写为在二维数组中自顶向下的递推即可，即 $dp[i,j]=dp[i-1,j]+dp[i,j-1]$。

1、状态定义：

$dp[i][j]$ 表示走到格子 (i,j) 的方法数。

2、状态转移：

如果网格 (i,j) 上有障碍物，则 $dp[i][j]$ 值为 0，表示走到该格子的方法数为 0；
否则网格 (i,j) 可以从网格 $(i-1,j)$ 或者 网格 $(i,j-1)$ 走过来，因此走到该格子的方法数为走到网格 $(i-1,j)$ 和网格 $(i,j-1)$ 的方法数之和，即 $dp[i,j]=dp[i-1,j]+dp[i,j-1]$。

状态转移方程如下：

$$dp[i][j]=\begin{cases} dp[i-1,j]+dp[i,j-1] & (i,j)\text{上无障碍物} \\ 0 & (i,j)\text{上有障碍物} \end{cases}$$

3、初始条件

第 1 列的格子只有从其上边格子走过去这一种走法，因此初始化 dp[i][0] 值为 1，存在障碍物时为 0；

第 1 行的格子只有从其左边格子走过去这一种走法，因此初始化 dp[0][j] 值为 1，存在障碍物时为 0。

```
int m = obstacleGrid.length, n = obstacleGrid[0].length;
int[][] dp = new int[m][n];
for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) {
    dp[i][0] = 1;
}
for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) {
    dp[0][j] = 1;
}
```

63 具体代码

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        if (obstacleGrid == null || obstacleGrid.length == 0 || obstacl
eGrid[0].length == 0){
            return 1;
        }
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
        int[][] dp = new int[m][n];
        for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) {
            dp[i][0] = 1;
        }
        for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) {
            dp[0][j] = 1;
        }
        for (int i = 1; i < m; i++){
            for (int j = 1; j < n; j++){
                if (obstacleGrid[i][j] == 1) {
                    dp[i][j] = 0;
                } else {
                    dp[i][j] = dp[i-1][j] + dp[i][j-1];
                }
            }
        }
        return dp[m-1][n-1];
    }
}
```

空间优化以后：

2,动态规划优化

这里可以参照这里的二维数组的优化370，最长公共子串和子序列很类似，我们发现上面的代码有个规律，就是在遍历的时候只使用了每个位置左边的值和上边的值，其他的都用不到，所以我们可以考虑把二维数组转化为一位数组，来看下代码

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
```

```
        if (obstacleGrid == null || obstacleGrid.length == 0 || obstacl
eGrid[0].length == 0){
            return 1;
        }
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
        int[] dp = new int[n+1];
        dp[1] = 1;
        for (int i = 0; i < m; i++){
            for (int j = 1; j <= n; j++){
                if (obstacleGrid[i][j-1] == 1){
                    dp[j] = 0;
                } else {
                    dp[j] += dp[j-1];
                }
            }
        }
        return dp[n];
    }
}
```

40、迷路的机器人（面试题 08.02）【是不同路径 2 的求路径题目（用回溯）】

面试题 08.02. 迷路的机器人

难度 中等 31 收藏 分享 切换为英文 关注 反馈

设想有个机器人坐在一个网格的左上角，网格 r 行 c 列。机器人只能向下或向右移动，但不能走到一些被禁止的网格（有障碍物）。设计一种算法，寻找机器人从左上角移动到右下角的路径。

网格中的障碍物和空位置分别用 1 和 0 来表示。

返回一条可行的路径，路径由经过的网格的行号和列号组成。左上角为 0 行 0 列。如果没有可行的路径，返回空数组。

示例 1:

```
输入:
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
输出: [[0,0],[0,1],[0,2],[1,2],[2,2]]
解释:
输入中标粗的位置即为输出表示的路径，即
0行0列（左上角） -> 0行1列 -> 0行2列 -> 1行2列 -> 2行2列（右下角）
```

说明： *r* 和 *c* 的值均不超过 100。

https://leetcode-cn.com/problems/robot-in-a-grid-lcci/solution/mi-lu-de-ji-qi-ren-zhu-shi-xiang-xi-po-su-si-lu-by/

目标:

找到一条从左上角走到右下角路径，如果没有这样的路径返回空数组。

规则:

从左上角走到右下角，每次只能向下走或者向右走。

明确回溯法四要素:

1. 结束条件：走到右下角就结束；右下角本身有障碍物，不可能走得到；
2. 路径：走到当前位置之前已经走过的路径。
3. 选择：每次只能向下走或者向右走。当下方有障碍物时，只能考虑向右走；当右方有障碍物时，只能考虑向下走；当下方和右方都有障碍物时，只能往回走，你从哪个地方进入这个死胡同的就回到哪个地方去。
4. 约束条件：除了在"选择中的"约束之外，我们还不能走已经走过的地方。

代码:

以x, y表示当前位置，一个与网格同大小的数组$visit$记录走过的地方。

```
class Solution {
    List<List<Integer>> path = new LinkedList<>();  // 记录路径
    int r = 0;  // 行数
    int c = 0;  // 列数
    public List<List<Integer>> pathWithObstacles(int[][] obstacleGrid)
{
        r = obstacleGrid.length;
        if (r == 0) {       // 空网格
            return path;
        }
        c = obstacleGrid[0].length;
        if (obstacleGrid[r - 1][c - 1] == 1) {  // 终点有障碍
            return path;
        }
        boolean[][] visit = new boolean[r][c];  // 记录数组
        backtrack(obstacleGrid, 0, 0, visit);
```

```
        return path;
    }

    public boolean backtrack(int[][] obstacleGrid, int x, int y, boolea
n[][] visit) {
        // 越界，有障碍，已访问
        if (x >= r || y >= c || obstacleGrid[x][y] == 1 || visit[x][y])
 {
            return false;
        }
        // 如果不是以上情况，说明这个格子值得探索，做出选择
        path.add(Arrays.asList(x, y));
        visit[x][y] = true;
        // 选择后是否到达终点
        if (x == r - 1 && y == c - 1) return true;
        // 选择后没到终点，先尝试向下，再尝试向右，神奇的或运算
        if (backtrack(obstacleGrid, x + 1, y, visit) || backtrack(obsta
cleGrid, x, y + 1, visit)) {
            return true;
        }
        // 既不能向下也不能向右，是个死胡同，撤销选择
        path.remove(path.size() - 1);
        return false;
    }
}
```

41、最小路径和（64）

https://leetcode-cn.com/problems/minimum-path-sum/solution/zui-xiao-lu-jing-he-by-leetcode-

更多硬核内容就在公众号「yes 的练级攻略」

solution/

由于路径的方向只能是向下或向右，因此网格的第一行的每个元素只能从左上角元素开始向右移动到达，网格的第一列的每个元素只能从左上角元素开始向下移动到达，此时的路径是唯一的，因此每个元素对应的最小路径和即为对应的路径上的数字总和。

对于不在第一行和第一列的元素，可以从其上方相邻元素向下移动一步到达，或者从其左方相邻元素向右移动一步到达，元素对应的最小路径和等于其上方相邻元素与其左方相邻元素两者对应的最小路径和中的最小值加上当前元素的值。由于每个元素对应的最小路径和与其相邻元素对应的最小路径和有关，因此可以使用动态规划求解。

创建二维数组 dp，与原始网格的大小相同，$dp[i][j]$ 表示从左上角出发到 (i, j) 位置的最小路径和。显然，$dp[0][0] = grid[0][0]$。对于 dp 中的其余元素，通过以下状态转移方程计算元素值。

- 当 $i > 0$ 且 $j = 0$ 时，$dp[i][0] = dp[i-1][0] + grid[i][0]$。
- 当 $i = 0$ 且 $j > 0$ 时，$dp[0][j] = dp[0][j-1] + grid[0][j]$。
- 当 $i > 0$ 且 $j > 0$ 时，$dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$。

最后得到 $dp[m-1][n-1]$ 的值即为从网格左上角到网格右下角的最小路径和。

复杂度分析

- 时间复杂度：$O(mn)$，其中 m 和 n 分别是网格的行数和列数。需要对整个网格遍历一次，计算 dp 的每个元素的值。
- 空间复杂度：$O(mn)$，其中 m 和 n 分别是网格的行数和列数。创建一个二维数组 dp，和网格大小相同。空间复杂度可以优化，例如每次只存储上一行的 dp 值，则可以将空间复杂度优化到 $O(n)$。

方法 1：二维 dp 数组

```
class Solution {
    public int minPathSum(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }
        int rows = grid.length, columns = grid[0].length;
        int[][] dp = new int[rows][columns];
        dp[0][0] = grid[0][0];
        for (int i = 1; i < rows; i++) {
            dp[i][0] = dp[i - 1][0] + grid[i][0];
        }
        for (int j = 1; j < columns; j++) {
            dp[0][j] = dp[0][j - 1] + grid[0][j];
        }
        for (int i = 1; i < rows; i++) {
            for (int j = 1; j < columns; j++) {
                dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[
i][j];
            }
        }
        return dp[rows - 1][columns - 1];
    }
}
```

方法 2：一维的

```java
class Solution {
    public int minPathSum(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }
        int m = grid.length, n = grid[0].length;
        int[] dp = new int[n];
        dp[0] = grid[0][0];
        for (int j = 1; j < n; j++) {
            dp[j] = dp[j - 1] + grid[0][j];
        }
        for (int i = 1; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (j == 0) {
                    dp[j] += grid[i][j];
                } else {
                    dp[j] = Math.min(dp[j], dp[j - 1]) + grid[i][j];
                }
            }
        }
        return dp[n - 1];
    }
}
```

方法 3：这题的空间结构优化可以利用原矩阵，直接修改原矩阵

```java
class Solution {
    public int minPathSum(int[][] grid) {
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(i == 0 && j == 0) continue;
                else if(i == 0)    grid[i][j] = grid[i][j - 1] + grid[i][j];
                else if(j == 0)    grid[i][j] = grid[i - 1][j] + grid[i][j];
                else grid[i][j] = Math.min(grid[i - 1][j], grid[i][j - 1]) + grid[i][j];
            }
        }
        return grid[grid.length - 1][grid[0].length - 1];
    }
}
```

链接：

https://leetcode-cn.com/problems/minimum-path-sum/solution/zui-xiao-lu-jing-he-dong-tai-gui-hua-gui-fan-liu-c/

42、礼物的最大价值（剑指 47）

yes 的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 47. 礼物的最大价值

难度 中等 56 收藏 分享 切换为英文 关注 反馈

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

示例 1:

```
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 12
解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物
```

提示：

- 0 < grid.length <= 200
- 0 < grid[0].length <= 200

```
class Solution {
    public int maxValue(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) return 0;
        int m = grid.length, n = grid[0].length;
        int[] dp = new int[n];
        dp[0] = grid[0][0];
        for (int j = 1; j < n; j++) {
            dp[j] = dp[j - 1] + grid[0][j];
        }
        for (int i = 1; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (j == 0) {
                    dp[j] += grid[i][j];
                } else {
                    dp[j] = Math.max(dp[j], dp[j - 1]) + grid[i][j]; //与上一题
唯一的差别
                }
            }
        }
        return dp[n - 1];
    }
}
```

43、不相交的线（1035）

更多硬核内容就在公众号「yes 的练级攻略」

1035. 不相交的线

难度 中等 70 收藏 分享 切换为英文 关注 反馈

我们在两条独立的水平线上按给定的顺序写下 `A` 和 `B` 中的整数。

现在，我们可以绘制一些连接两个数字 `A[i]` 和 `B[j]` 的直线，只要 `A[i] == B[j]`，且我们绘制的直线不与任何其他连线（非水平线）相交。

以这种方法绘制线条，并返回我们可以绘制的最大连线数。

示例 1：

```
输入：A = [1,4,2], B = [1,2,4]
输出：2
解释：
我们可以画出两条不交叉的线，如上图所示。
我们无法画出第三条不相交的直线，因为从 A[1]=4 到 B[2]=4 的直线将与从 A[2]=2 到 B[1]=2 的直线相交。
```

示例 2：

```
输入：A = [2,5,1,2,5], B = [10,5,2,1,5,2]
输出：3
```

示例 3：

```
输入：A = [1,3,7,1,7,5], B = [1,9,2,5,1]
输出：2
```

提示：

1. `1 <= A.length <= 500`
2. `1 <= B.length <= 500`
3. `1 <= A[i], B[i] <= 2000`

yes的练级攻略

https://leetcode-cn.com/problems/uncrossed-lines/solution/1035-bu-xiang-jiao-de-xian-java-dong-tai-gui-hua-b/

解题思路

本体使用动态规划求解。*其实连线不能相交我觉得可以在某些方面上理解成小人走路，只能向右和向下的那类题。*

基本思路就是：`dp[i][j]` 表示 `A` 截止到 `i`， `B` 截止到 `j` 点，此时的最大连线数。

转移方程为：

- 当 `A[i] == B[j]` 时：`dp[i][j] = dp[i - 1][j - 1] + 1`
- 当 `A[i] != B[j]` 时：`dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])`

```
class Solution {
    public int maxUncrossedLines(int[] A, int[] B) {
        int[][] dp = new int[A.length + 1][B.length + 1];
        for (int i = 1; i <= A.length; i ++) {
            for (int j = 1; j <= B.length; j ++) {
                dp[i][j] = A[i - 1] == B[j - 1] ? dp[i - 1][j - 1] + 1 : Math.m
ax(dp[i - 1][j], dp[i][j - 1]);
            }
```

```
            }
            return dp[A.length][B.length];
        }
    }
```

44、地下城游戏（174）

174. 地下城游戏

难度 困难 241 收藏 分享 切换为英文 关注 反馈

一些恶魔抓住了公主（**P**）并将她关在了地下城的右下角。地下城是由 M x N 个房间组成的二维网格。我们英勇的骑士（**K**）最初被安置在左上角的房间里，他必须穿过地下城并通过对抗恶魔来拯救公主。

骑士的初始健康点数为一个正整数。如果他的健康点数在某一时刻降至 0 或以下，他会立即死亡。

有些房间由恶魔守卫，因此骑士在进入这些房间时会失去健康点数（若房间里的值为*负整数*，则表示骑士将损失健康点数）；其他房间要么是空的（房间里的值为 *0*），要么包含增加骑士健康点数的魔法球（若房间里的值为*正整数*，则表示骑士将增加健康点数）。

为了尽快到达公主，骑士决定每次只向右或向下移动一步。

编写一个函数来计算确保骑士能够拯救到公主所需的最低初始健康点数。

例如，考虑到如下布局的地下城，如果骑士遵循最佳路径 `右 -> 右 -> 下 -> 下`，则骑士的初始健康点数至少为 **7**。

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

说明:

- 骑士的健康点数没有上限。
- 任何房间都可能对骑士的健康点数造成威胁，也可能增加骑士的健康点数，包括骑士进入的左上角房间以及公主被监禁的右下角房间。

注意：i++和++i 在 for 循环中都是放在语句最后执行的，没什么差别。差距在于先减的话，不需要用一个临时变量去储存该值。在 Java 中 i++语句是需要一个临时变量取存储返回自增前的值，而++i 不需要。

https://leetcode-cn.com/problems/dungeon-game/solution/di-xia-cheng-you-xi-by-leetcode-solution/

yes的练级攻略

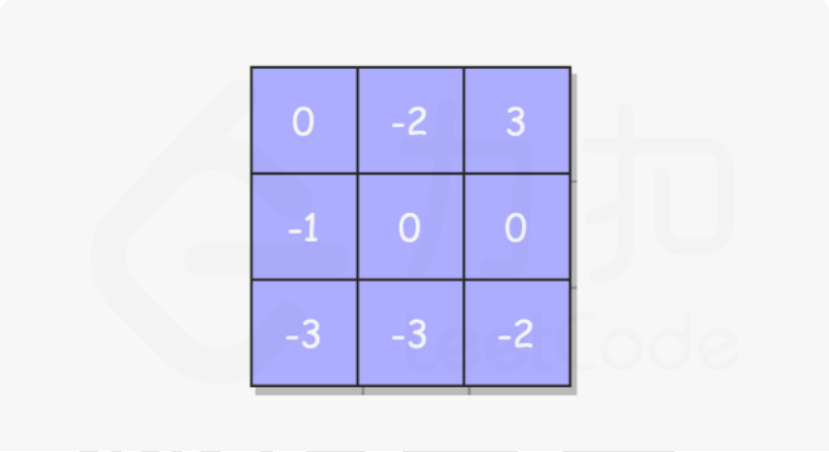
更多硬核内容就在公众号「yes 的练级攻略」

方法一：动态规划

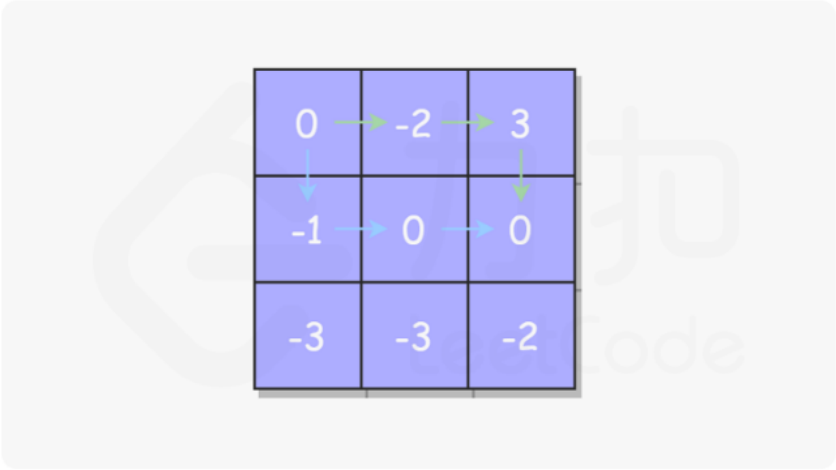
思路及算法

几个要素：「$M \times N$ 的网格」「每次只能向右或者向下移动一步」。让人很容易想到该题使用动态规划的方法。

但是我们发现，如果按照从左上往右下的顺序进行动态规划，对于每一条路径，我们需要同时记录两个值。第一个是「从出发点到当前点的路径和」，第二个是「从出发点到当前点所需的最小初始值」。而这两个值的重要程度相同，参看下面的示例：



从 $(0, 0)$ 到 $(1, 2)$ 有多条路径，我们取其中最有代表性的两条：



- 绿色路径「从出发点到当前点的路径和」为 1，「从出发点到当前点所需的最小初始值」为 3。
- 蓝色路径「从出发点到当前点的路径和」为 -1，「从出发点到当前点所需的最小初始值」为 2。

我们希望「从出发点到当前点的路径和」尽可能大，而「从出发点到当前点所需的最小初始值」尽可能小。这两条路径各有优劣。

在上图中，我们知道应该选取绿色路径，因为蓝色路径的路径和太小，使得蓝色路径需要增大初始值到 4 才能走到终点，而绿色路径只要 3 点初始值就可以直接走到终点。但是如果把终点的 -2 换为 0，蓝色路径只需要初始值 2，绿色路径仍然需要初始值 3，最优决策就变成蓝色路径了。

更多硬核内容就在公众号「yes 的练级攻略」

因此，如果按照从左上往右下的顺序进行动态规划，我们无法直接确定到达 $(1,2)$ 的方案，因为有两个重要程度相同的参数同时影响后续的决策。也就是说，这样的动态规划是不满足「无后效性」的。

于是我们考虑从右下往左上进行动态规划。令 $dp[i][j]$ 表示从坐标 (i,j) 到终点所需的最小初始值。换句话说，当我们到达坐标 (i,j) 时，如果此时我们的路径和不小于 $dp[i][j]$，我们就能到达终点。

这样一来，我们就无需担心路径和的问题，只需要关注最小初始值。对于 $dp[i][j]$，我们只要关心 $dp[i][j+1]$ 和 $dp[i+1][j]$ 的最小值 $minn$。记当前格子的值为 $dungeon(i,j)$，那么在坐标 (i,j) 的初始值只要达到 $minn - dungeon(i,j)$ 即可。同时，初始值还必须大于等于 1。这样我们就可以得到状态转移方程：

$$dp[i][j] = \max(\min(dp[i+1][j], dp[i][j+1]) - dungeon(i,j), 1)$$

最终答案即为 $dp[0][0]$。

边界条件为，当 $i = n-1$ 或者 $j = m-1$ 时，$dp[i][j]$ 转移需要用到的 $dp[i][j+1]$ 和 $dp[i+1][j]$ 中有无效值，因此代码实现中给无效值赋值为极大值。特别地，$dp[n-1][m-1]$ 转移需要用到的 $dp[n-1][m]$ 和 $dp[n][m-1]$ 均为无效值，因此我们给这两个值赋值为 1。

复杂度分析

- 时间复杂度：$O(N \times M)$，其中 N, M 为给定矩阵的长宽。
- 空间复杂度：$O(N \times M)$，其中 N, M 为给定矩阵的长宽，注意这里可以利用滚动数组进行优化，优化后空间复杂度可以达到 $O(N)$。

方法 1：二维反向动态规划

```
class Solution {
    public int calculateMinimumHP(int[][] dungeon) {
        int m = dungeon.length, n = dungeon[0].length;
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 0; i <= m; ++i) {
            Arrays.fill(dp[i], Integer.MAX_VALUE);
        }
        dp[m][n-1] = 1;
        dp[m-1][n] = 1;
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                int min = Math.min (dp[i+1][j], dp[i][j+1]);
                dp[i][j] = Math.max(min - dungeon[i][j], 1);
            }
        }
        return dp[0][0];
    }
}
```

方法 2：一维

因为 dp[i][j]只和它的右边和下方有关，所以可以省略一维数组

```
class Solution {
    public int calculateMinimumHP(int[][] dungeon) {
        int m = dungeon.length, n = dungeon[0].length;
        int[] dp = new int[n+1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[n-1] = 1;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) {
                int min = Math.min(dp[j], dp[j+1]);
                dp[j] = Math.max(min - dungeon[i][j], 1);
            }
        }
        return dp[0];
    }
}
```

https://leetcode-cn.com/problems/dungeon-game/solution/jian-dan-dfskan-yi-yan-jiu-ming-bai-e-by-sweetiee/

一、题目分析

从出发点开始，DFS 搜索到终点的所有的路径。
在每个点，我们可以选择向右搜或者向下搜（最后一行/列除外）。

因此，求从 `(0, 0)` 出发到终点的最小初始值 `dfs(0, 0)`，可以转化为先求 `dfs(1, 0)` 和 `dfs(0, 1)`，取其中较小的一个，即 `min(dfs(0, 1), dfs(1, 0))`，作为 `(0, 0)` 点的后续路径所要求的最低血量。又因为 `(0, 0)` 点本身就提供血量 `dungoen[0][0]`，因此从 `(0, 0)` 点出发所需的血量为后续路径所需要的最低血量减去 `(0, 0)` 点提供的血量，即 `min(dfs(0, 1), dfs(1, 0)) - dungoen[0][0]`（这里写代码时要注意骑士的血量不能小于 1）。

解法2：DFS + 记忆化（0ms，100%，搜索时会有大量重复计算的分支，加上记忆化即可解决）

在解法 1 的基础上定义了 `int[][] memo`。

时间复杂度：$O(m*n)$，m, n 为 dungeon 的行数和列数。
空间复杂度：$O(m*n)$，m, n 为 dungeon 的行数和列数。

方法 3：DFS+memo

```
class Solution {
    int[][] memo; // 定义记忆化数组
    public int calculateMinimumHP(int[][] dungeon) {
        memo = new int[dungeon.length][dungeon[0].length];
        return dfs(dungeon, dungeon.length, dungeon[0].length, 0, 0);
    }
    private int dfs(int[][] dungeon, int m, int n, int i, int j) {
        // 到达终点，递归终止。
        if (i == m - 1 && j == n - 1) {
            return Math.max(1 - dungeon[i][j], 1);
        }
        // 如果 memo 数组中有值，直接取出并返回，不进行后续的搜索。
        if (memo[i][j] > 0) {
            return memo[i][j];
        }
        // 同解法一，向右搜+向下搜
        int minRes = 0;
        if (i == m - 1) {
```

```
            minRes =  Math.max(dfs(dungeon, m, n, i, j + 1) - dungeon[i][j], 1)
;
        } else if (j == n - 1) {
            minRes = Math.max(dfs(dungeon, m, n, i + 1, j) - dungeon[i][j], 1);
        } else {
            minRes = Math.max(Math.min(dfs(dungeon, m, n, i + 1, j), dfs(dungeo
n, m, n, i, j + 1)) - dungeon[i][j], 1);
        }
        // 将结果存入 memo 数组
        return memo[i][j] = minRes;
    }
}
```

45、三角形最小路径和（120）

120. 三角形最小路径和

难度 中等 452 收藏 分享 切换为英文 关注 反馈

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

例如，给定三角形：

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

自顶向下的最小路径和为 11（即，**2** + **3** + **5** + **1** = 11）。

说明：

如果你可以只使用 *O*(*n*) 的额外空间（*n* 为三角形的总行数）来解决这个问题，那么你的算法会很加分。

https://leetcode-cn.com/problems/triangle/solution/di-gui-ji-yi-hua-dp-bi-xu-miao-dong-by-sweetiee/

更多硬核内容就在公众号「yes 的练级攻略」

一、题目分析

题意：
给定三角形，每次只能移动到下一行中的相邻结点，求从顶点到底边的最小路径和。

Java

```
[
[2],
[3,4],
[6,5,7],
[4,1,8,3]
]
```

相邻结点：与(i，j) 点相邻的结点为 (i + 1，j) 和 (i + 1，j + 1)。

分析：
若定义 $f(i,j)$ 为 (i,j) 点到底边的最小路径和，则易知递归求解式为：

$$f(i,j) = min(f(i+1,j), f(i+1,j+1)) + triangle[i][j]$$

由此，我们将任一点到底边的最小路径和，转化成了与该点相邻两点到底边的最小路径和中的较小值，再加上该点本身的值。这样本题的 **递归解法（解法一）** 就完成了。

暴力搜索会有大量的重复计算，导致 **超时**，因此在 **解法二** 中结合记忆化数组进行优化。

解法二：递归 + 记忆化

在解法一的基础上，定义了二维数组进行记忆化。

时间复杂度：$O(N^2)$，N 为三角形的行数。
空间复杂度：$O(N^2)$，N 为三角形的行数。

方法 1：dfs+memo

```
class Solution {
    Integer[][] memo;
    public int minimumTotal(List<List<Integer>> triangle) {
        memo = new Integer[triangle.size()][triangle.size()];
        return dfs(triangle, 0, 0);
    }
    private int dfs(List<List<Integer>> triangle, int i , int j) {
        if (i == triangle.size()){
            return 0;
        }
        if (memo[i][j] != null) {
            return memo[i][j];
        }

        return memo[i][j] = Math.min(dfs(triangle, i+1, j), dfs(triangl
e, i+1, j+1)) + triangle.get(i).get(j);
    }
}
```

解法三：动态规划

定义二维 dp 数组，将解法二中「自顶向下的递归」改为「自底向上的递推」。

1、状态定义：

$dp[i][j]$ 表示从点 (i, j) 到底边的最小路径和。

2、状态转移：

$$dp[i][j] = min(dp[i+1][j], dp[i+1][j+1]) + triangle[i][j]$$

时间复杂度：$O(N^2)$，N 为三角形的行数。
空间复杂度：$O(N^2)$，N 为三角形的行数。

4、空间优化

在上述代码中，我们定义了一个 N 行 N 列 的 dp 数组（N 是三角形的行数）。
但是在实际递推中我们发现，计算 $dp[i][j]$ 时，只用到了下一行的 $dp[i+1][j]$ 和 $dp[i+1][j+1]$。
因此 dp 数组不需要定义 N 行，只要定义 1 行就阔以啦。
所以我们稍微修改一下上述代码，将 i 所在的维度去掉（如下），就可以将 $O(N^2)$ 的空间复杂度优化成 $O(N)$ 啦~

时间复杂度：$O(N^2)$，N 为三角形的行数。
空间复杂度：$O(N)$，N 为三角形的行数。

方法 2：一维 dp

```
class Solution {
    public int minimumTotal(List<List<Integer>> triangle) {
        int n = triangle.size();
        int[] dp = new int[n + 1];
        for (int i = n - 1; i >= 0; i--) {
            for (int j = 0; j <= i; j++) {
                dp[j] = Math.min(dp[j], dp[j + 1]) + triangle.get(i).get(j);
            }
        }
        return dp[0];
    }
}
```

46、交错字符串（97）

更多硬核内容就在公众号「yes 的练级攻略」

题目描述　评论 (257)　题解(234)　提交记录

97. 交错字符串

难度 困难　261　收藏　分享　切换为英文　关注　反馈

给定三个字符串 *s1*, *s2*, *s3*, 验证 *s3* 是否是由 *s1* 和 *s2* 交错组成的。

示例 1:

```
输入: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
输出: true
```

示例 2:

```
输入: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
输出: false
```

https://leetcode-cn.com/problems/interleaving-string/solution/lei-si-lu-jing-wen-ti-zhao-zhun-zhuang-tai-fang-ch/

每次只能往右或者往下选择字符，求问是否存在target路径

s1 = bacc　　s2 = aabcce

target = abaacbccec

i:	j:	0	a	a	b	c	c	e
	0	T	T	F	F	F	F	F
	b	F	T	T	F	F	F	F
	a	F	T	T	F	F	F	F
	c	F	F	T	T	T	T	T
	c	F	F	F	F	T	F	T

图画的烂，轻喷。
不知道大家对不同路径的题还有没有印象，本题也可以利用其思想求解:
target的每个字符都是从s1（向下）或者s2（向右）拿到的,所以只要判断是否存在这条target路径即可；

于是可定义boolean[][] dp ， dp[i][j]代表s1前i个字符与s2前j个字符拼接成s3的i+j字符，也就是存在目标路径能够到达i,j;
状态方程:

边界1：dp[0][0] = true;
边界2：if i=0 : dp[0]dp[j] = s2[0-j) equals s3[0,j) 遇到false后面可以直接省略
边界3：if j=0 : dp[i]dp[0] = s1[0-i) equals s3[0,i) 遇到false后面可以直接省略

其他情况，到达（i，j）可能由（i-1,j）点向下一步，选择s1[i-1]到达；也可能由（i,j-1）点向右一步，选择s2[j-1]到达；
dp[i,j] = (dp[i-1][j] && s3[i+j-1] == s1[i-1]) || (dp[i][j-1] && s3[i+j-1] == s2[j-1])

方法 1：二维

```
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
```

```
        int m = s1.length(), n = s2.length();
        if (s3.length() != m + n) return false;
        // 动态规划，dp[i,j]表示 s1 前 i 字符能与 s2 前 j 字符组成 s3 前 i+j 个字
符;
        boolean[][] dp = new boolean[m+1][n+1];
        dp[0][0] = true;
        for (int i = 1; i <= m && s1.charAt(i-1) == s3.charAt(i-1); i++
) dp[i][0] = true; // 不相符直接终止
        for (int j = 1; j <= n && s2.charAt(j-1) == s3.charAt(j-1); j++
) dp[0][j] = true; // 不相符直接终止
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = (dp[i - 1][j] && s3.charAt(i + j - 1) == s1.
charAt(i - 1))
                    || (dp[i][j - 1] && s3.charAt(i + j - 1) == s2.char
At(j - 1));
            }
        }
        return dp[m][n];
    }
}
```

方法 2：一维

```
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        int m = s1.length(), n = s2.length();
        if (s3.length() != m + n) return false;

        boolean[] dp = new boolean[n+1];
        dp[0] = true;

        for (int j = 1; j <= n; j++) {
            dp[j] = dp[j-1] && s2.charAt(j-1) == s3.charAt(j-1) ;
        }

        for (int i = 1; i <= m; i++) {
            dp[0] = dp[0] && s3.charAt(i - 1) == s1.charAt(i - 1);
            for (int j = 1; j <= n; j++) {
                dp[j] = (dp[j] && s3.charAt(i + j - 1) == s1.charAt(i -
 1))
                    || (dp[j - 1] && s3.charAt(i + j - 1) == s2.charAt(
j - 1));
            }
        }
        return dp[n];
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    }
}
```

47、单词拆分（139）

139. 单词拆分

难度 中等 597 收藏 分享 切换为英文 关注 反馈

给定一个**非空**字符串 *s* 和一个包含**非空**单词列表的字典 *wordDict*，判定 *s* 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
     注意你可以重复使用字典中的单词。
```

示例 3：

```
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false
```

https://leetcode-cn.com/problems/word-break/solution/dan-ci-chai-fen-by-leetcode-solution/

方法一：动态规划

思路和算法

我们定义 $dp[i]$ 表示字符串 s 前 i 个字符组成的字符串 $s[0..i-1]$ 是否能被空格拆分成若干个字典中出现的单词。从前往后计算考虑转移方程，每次转移的时候我们需要枚举包含位置 $i-1$ 的最后一个单词，看它是否出现在字典中以及除去这部分的字符串是否合法即可。公式化来说，我们需要枚举 $s[0..i-1]$ 中的分割点 j，看 $s[0..j-1]$ 组成的字符串 s_1（**默认 $j=0$ 时 s_1 为空串**）和 $s[j..i-1]$ 组成的字符串 s_2 是否都合法，如果两个字符串均合法，那么按照定义 s_1 和 s_2 拼接成的字符串也同样合法。由于计算到 $dp[i]$ 时我们已经计算出了 $dp[0..i-1]$ 的值，因此字符串 s_1 是否合法可以直接由 $dp[j]$ 得知，剩下的我们只需要看 s_2 是否合法即可，因此我们可以得出如下转移方程：

$$dp[i] = dp[j] \ \&\& \ check(s[j..i-1])$$

其中 $check(s[j..i-1])$ 表示子串 $s[j..i-1]$ 是否出现在字典中。

对于检查一个字符串是否出现在给定的字符串列表里一般可以考虑哈希表来快速判断，同时也可以做一些简单的剪枝，枚举分割点的时候倒着枚举，如果分割点 j 到 i 的长度已经大于字典列表里最长的单词的长度，那么就结束枚举，但是需要注意的是下面的代码给出的是不带剪枝的写法。

对于边界条件，我们定义 $dp[0] = true$ 表示空串且合法。

有能力的读者也可以考虑怎么结合字典树 $Trie$ 来实现，这里不再展开。

复杂度分析

- 时间复杂度：$O(n^2)$，其中 n 为字符串 s 的长度。我们一共有 $O(n)$ 个状态需要计算，每次计算需要枚举 $O(n)$ 个分割点，哈希表判断一个字符串是否出现在给定的字符串列表需要 $O(1)$ 的时间，因此总时间复杂度为 $O(n^2)$。
- 空间复杂度：$O(n)$，其中 n 为字符串 s 的长度。我们需要 $O(n)$ 的空间存放 dp 值以及哈希表亦需要 $O(n)$ 的空间复杂度，因此总空间复杂度为 $O(n)$。

方法 1：hashset+动规

```
public class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> wordDictSet = new HashSet(wordDict);
        boolean[] dp = new boolean[s.length() + 1];
        dp[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && wordDictSet.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```

方法 2 的链接：

https://leetcode-cn.com/problems/word-break/solution/139ti-dan-ci-chai-fen-by-iceblood/

题意分析

(1) 给定一个非空字符串和一个非空且没有重复单词的单词表 `List` 。

(2) 要求判断字符串能否被空格拆分成一个或多个单词表中的单词，拆分时可重复使用单词表中单词。

*** 本题主要需要的考虑的是判断一个单词以后如何判断字符串除当前单词外部分。

方法一：动态规划 + 单词表遍历

思路：

(1) 定义动态数组记录每个长度的前缀字符串是否符合拆分要求。

(2) 当前缀符合并且连接前缀的单词存在于单词表，则新组成的前缀符合拆分要求。

(3) 按单词表遍历进行剪枝，状态转移直到字符串长度，返回最终状态值。

方法 2：动规+单词表遍历

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        int len = s.length();
        // 定义动态数组记录每个长度的前缀字符串是否符合拆分要求。
        boolean[] dp = new boolean[len + 1];
        // 初始化字符串开头之前部分符合拆分要求。
        dp[0] = true;
        // 从头开始遍历字符串，记录每部分前缀是否符合拆分要求。
        for (int i = 1; i <= len; i++) {
            // 从单词表中依次取出单词进行判断。
            for (String word : wordDict) {
                // 当前单词大于遍历到的字符串前缀长度，则跳过。
                if (word.length() > i) {
                    continue;
                }
                // 当前单词是遍历到前缀的结尾并且去掉当前结尾单词的部分依然符合拆分要求，则当前前缀符合。
```

```
                if (dp[i - word.length()] && s.substring(i - word.lengt
h(), i).equals(word)) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[len];
    }
}
```

方法 3 的链接

https://leetcode-cn.com/problems/word-break/solution/java-trie-bfsfan-hui-suo-you-ke-neng-de-jie-wei-by/

方法 3：字典树+BFS

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        // 构建 Trie，存入单词
        Trie trie = new Trie();
        for (String word : wordDict)
            trie.insert(word);

        // BFS 准备
        Queue<Integer> queue = new LinkedList<>();
        queue.add(0); // 从 s 的开头开始问 Trie：有哪些可能

        int n = s.length();
        boolean[] visited = new boolean[n];
        visited[0] = true;

        while (!queue.isEmpty()) {
            int from = queue.remove();
            // Trie 返回所有的可能结尾
            List<Integer> ends = trie.searchPrefixes(s.toCharArray(), f
rom);
            for (int end : ends) {
                if (end == n - 1) return true;
                if (visited[end + 1]) continue;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                // 以 end 为结尾匹配上一个单词，继续 end + 1 开始匹配
                queue.add(end + 1);
                visited[end + 1] = true;
            }
        }
        return false;
    }
}

class Trie {
    private TrieNode root;

    Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode curr = root;
        for (char c : word.toCharArray()) {
            if (!curr.contains(c))
                curr.set(c, new TrieNode());
            curr = curr.get(c);
        }
        curr.setEnd();
    }

    // 返回可能的结尾
    public List<Integer> searchPrefixes(char[] chars, int from) {
        List<Integer> res = new ArrayList<>();
        TrieNode curr = root;
        for (int i = from; i < chars.length; i++) {
            curr = curr.get(chars[i]);
            if (curr == null) break;
            else if (curr.isEnd()) res.add(i); // 能成词，记录此索引 i 为
结尾
        }
        return res;
    }
}

class TrieNode {
    private TrieNode[] arr;
    private boolean isEnd;
```

```
    TrieNode() {
        arr = new TrieNode[256];
        isEnd = false;
    }

    public TrieNode get(char c) {
        return arr[c];
    }

    public void set(char c, TrieNode next) {
        arr[c] = next;
    }

    public boolean contains(char c) {
        return arr[c] != null;
    }

    public boolean isEnd() {
        return isEnd;
    }

    public void setEnd() {
        isEnd = true;
    }
}
```

48、单词拆分 2（140）

140. 单词拆分 II

难度 困难 224 收藏 分享 切换为英文 关注 反馈

给定一个**非空**字符串 *s* 和一个包含**非空**单词列表的字典 *wordDict*，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。

说明：

- 分隔时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

输入：
```
s = "catsanddog"
wordDict = ["cat", "cats", "and", "sand", "dog"]
```
输出：
```
[
  "cats and dog",
  "cat sand dog"
]
```

更多硬核内容就在公众号「yes 的练级攻略」

示例 2：

```
输入：
s = "pineapplepenapple"
wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]
输出：
[
  "pine apple pen apple",
  "pineapple pen apple",
  "pine applepen apple"
]
解释：注意你可以重复使用字典中的单词。
```

示例 3：

```
输入：
s = "catsandog"
wordDict = ["cats", "dog", "sand", "and", "cat"]
输出：
[]
```

禁止商用

https://leetcode-cn.com/problems/word-break-ii/solution/leetcode-140-dan-ci-chai-fen-iiwo-gan-jio-jiang-de/

方法 3：使用动态规划

算法

这个方法背后的想法是对于给定的问题（s），它可以被拆分成子问题 $s1$ 和 $s2$。如果这些子问题分别都能满足条件，那么整个文字 s 也可以满足。比方说，"catsanddog" 可以被拆分成子字符串 "catsand" 和 "dog"。子问题 "catsand" 进一步可以被拆分成 "\text{cats}" 和 "\text{and}"，它们分别都是字典的一部分，所以 "catsand" 也是满足条件的。递归回来，因为 "catsand" 和 "dog" 分别都满足要求，所以原字符串 "catsanddog" 也符合要求。

现在，我们来考虑 dp 数组如何求出。我们使用长度为 $n+1$ 的数组 dp，其中 n 是给定字符串的长度。$\mathrm{dp}[k]$ 被用来存储用 $s[0:k-1]$ 可被拆分成合法单词的句子。我们同事用两个指针 i 和 j，其中 i 表示子字符串 s' 的长度（s' 是 s 的一个前缀），j 表示 s' 的拆分位置，即拆分成 $s'(0,j)$ 和 $s'(j+1,i)$。

为了求出 dp 数组，我们将 $\mathrm{dp}[0]$ 初始化为空串。我们以 i 为结尾表示的子字符串的所有前缀，通过指针 j 将 s 拆分成 $s1'$ 和 $s2'$。为了求出$\mathrm{dp}[i]$，我们检查所有 $\mathrm{dp}[j]$ 包含的所有非空字符串，也就是所有能形成 $s1'$ 的句子。如果存在，我们进一步检查 $s2'$ 是否在字典里，如果两个条件都满足，我们将子字符串 $s2'$ 添加到所有 $s1'$ 的句子的后面（这些句子已经保存在了 $dp[j]$ 里面），并将这些新形成的句子保存进（$\mathrm{dp}[i]$）。最终，$\mathrm{dp}[n]$（n 是给定字符串 s 的长度）里面保存了所有可以得到完整字符串 s 的所有句子。

复杂度分析

- 时间复杂度：$O(n^3)$，求 dp 需要两重循环，添加一个新的列表需要额外一重循环。
- 空间复杂度：$O(n^3)$，dp 数组的长度是n，dp 数组里保存了数组，数组里是一些字符串，也就是每个 dp 元素需要 n^2 的空间。

方法③、dp 优化

实现思路

使用 O(n) 时间提前预处理 字符串可以分割的位置

如果 最后 dp[len] == false，那么表示原串 s 无法进行分割，提前返回，否则，按照 方法② 进行分割

```
class Solution {
    public List<String> wordBreak(String s, List<String> wordDict) {
        int len = s.length();
        Set<String> set = new HashSet<>(wordDict);
        /*
        dp 处理，dp[i] 表示 [0, i] 是否能够 通过分割 或 不分割 使得处理后的字符串存在于字典中
        比如  s = catsand, wordDict = ["cat", "sand"]
        我们可以通过分割成 cat 和 sand 使得分割后的字符串存在于字典中, dp[i] = true;
        比如  s = catsand, wordDict = ["catsand"]
        我们可以不分割就使得原字符串存在于字典中, dp[i] = true;
        比如  s = catsand, wordDict = ["cats"]
        我们无论分割还是不分割都无法使得字符串都存在于字典中, dp[i] = false;
        */
        boolean[] dp = new boolean[len + 1];
        dp[0] = true;
        for(int i = 1; i <= len; i++){
            for(int j = 0; j < i; j++){
                if(dp[j] && set.contains(s.substring(j, i))){
                    dp[i] = true;
                }
            }
        }
        //原字符串 s 无法处理成字典中的字符串
        if(!dp[len]) return new ArrayList<>();
        List<String>[] strs = new List[len + 1];
        for(int i = 0; i <= len; i++) strs[i] = new ArrayList<>();
        for(int i = 1; i <= len; i++){
            if(!dp[i]) continue;
            for(int j = 0; j < i; j++){
                if(!dp[j]) continue;
                String str = s.substring(j, i);
                if(!set.contains(str)) continue;
                if(strs[j].isEmpty()) strs[i].add(str);
                else{
                    for(String ss : strs[j]){
                        strs[i].add(new StringBuilder(ss).append(" ").append(str).toString());
                    }
                }
            }
        }
        return strs[len];
```

```
    }
}
```

49、运动会（lintcode1291）【贪心】

1291. 运动会

运动会要来了，小红作为拉拉队长，要在每个项目中为参赛同学加油助威。
小红需要遵守以下规则：
1.给每个体育项目加油时长必须超过项目时长的一半，每个项目只能加油一次
2.体育项目的开始和结束时间都是整点，小红选择进入项目和离开的时间也必须是整点
3.不考虑往返于各个体育项目比赛场地中花费的时间
现在给出所有项目的起止时间，小红想知道自己能不能在每个项目中都为同学加油呢？
若可以则返回1，或不行则返回-1.

样例
样例1

输入：[[3,10],[1,5],[4,6]]
输出：1
说明：1-3时间参加第二个项目，3-4不参加，4-6参加第三个项目，6-10参加第一个项目
注意事项
1.题目保证运动会的项目数量不超过10个
2.若某项目时长为2，超过时长一半为2；若某项目时长为3，超过时长一半为2；若某项目时长为4，超过时长一半为3

CRRRRR
更新于 2020/6/9 下午7:04:00

解题思路

可以用贪心去做。给每个项目计算一个最迟参与时间（即小红最迟什么时候必须开始参加这个项目），每次取最早的最迟参与时间的项目

算法流程

·项目最少参与时间dur = (end - start)/2 +1, 项目最迟可参与时间 leave = end - dur

·定义结构体排序，按照leave升序排序，然后遍历取值，若当前时间大于leave时间则返回-1

复杂度分析

- 时间复杂度分析O(nlogn):需要进行排序再遍历

```
public class Solution {
    /**
     * @param Events: the start and end time
     * @return: if there has a solution return 1, otherwise return -1.
     */

    class Node implements Comparable<Node> {
        int leave, id, dur;
        public Node(int x, int y, int z) {
            this.leave = x;
            this.id = y;
            this.dur = z;
        }
        public int compareTo(Node a) {
            return this.leave - a.leave;
```

```
        }
    }

    public int CheerAll(int[][] Events) {
        // write your code here
        int num = Events.length;
        Node[] order_events = new Node[num];
        int now = 1;

        for(int i = 0; i < num; i++) {
            int leave = Events[i][1] - (Events[i][1] - Events[i][0]) / 2 - 1;
            Node event = new Node(leave, i, Events[i][1] - Events[i][0]);
            order_events[i] = event;
        }
        Arrays.sort(order_events);
        for(int i = 0; i < num; i++) {
            int id = order_events[i].id;
            int start = Events[id][0];
            now = Math.max(now,start);
            if(now > order_events[i].leave)
            {
                return -1;
            }
            now += order_events[i].dur/2 + 1;
        }
        return 1;
    }
}
```

50、课程表 3（630）【贪心】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

630. 课程表 III

难度 困难 75 收藏 分享 切换为英文 关注 反馈

这里有 `n` 门不同的在线课程，他们按从 `1` 到 `n` 编号。每一门课程有一定的持续上课时间（课程时间） `t` 以及关闭时间第 d 天。一门课要持续学习 `t` 天直到第 d 天时要完成，你将会从第 1 天开始。

给出 `n` 个在线课程用 `(t, d)` 对表示。你的任务是找出最多可以修几门课。

示例:

```
输入: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]
输出: 3
解释:
这里一共有 4 门课程，但是你最多可以修 3 门:
首先，修第一门课时，它要耗费 100 天，你会在第 100 天完成，在第 101 天准备下门课。
第二，修第三门课时，它会耗费 1000 天，所以你将在第 1100 天的时候完成它，以及在第 1101 天开始准备下门课程。
第三，修第二门课时，它会耗时 200 天，所以你将会在第 1300 天时完成它。
第四门课现在不能修，因为你将会在第 3300 天完成它，这已经超出了关闭日期。
```

提示:

1. 整数 1 <= d, t, n <= 10,000 。
2. 你不能同时修两门课程。

https://leetcode-cn.com/problems/course-schedule-iii/solution/ke-cheng-biao-iii-by-leetcode/

方法一：优先队列

我们首先可以发现，如果两门课 `(t1, d1)` 和 `(t2, d2)` 满足 `d1 <= d2`，即后者的结束时间不晚于前者，那么我们先学习前者再学习后者总是最优的。因为如果先学习前者，即需要满足 `x + t1 <= d1` 且 `x + t1 + t2 <= d2`；如果先学习后者，则需要满足 `x + t2 <= d2` 且 `x + t1 + t2 <= d1`。如果后者中的 `x + t1 + t2 <= d1` 成立，那么显然有 `x + t1 <= d1` 且 `x + t1 + t2 <= d1 <= d2`，即前者一定成立；反之如果 `x = 0, (t1, d1) = (2, 3), (t2, d2) = (5, 100)`，那么前者成立且后者不成立。因此先学习前者再学习后者总是最优的。

基于上面的结论，我们可以将课程按照完成时间 `d` 递增排序。假设我们在前 `i` 门课中最多可以选取 `k` 门课，并且这 `k` 门课的总时长最短（称为最优方案），那么有下面的不等式：

```
t1 + t2 <= d2
t1 + t2 + t3 <= d3
...
t1 + t2 + ... + tk <= dk
```

此时我们需要判断第 `i + 1` 门课 `(t0, d0)` 是否可选。如果选取的 `k` 门课的总时长 `t` 与 `t0` 之和小于等于 `d0`，即

```
t1 + t2 + ... + tk + t0 <= d0
```

那么 `(t0, d0)` 一定可选，此时前 `i + 1` 门课的最优方案是选取 `t1, t2, ..., tk, t0` 这 `k + 1` 门课。可以使用反证法来证明，假设可以选取超过 `k + 1` 门课，或者选取 `k + 1` 门课且总时长小于 `t1 + t2 + ... + tk + t0`，那么我们去除 `(t0, d0)` 这门课，剩余的选取方案一定满足条件，且优于选取 `t1, t2, ..., tk` 的方案，与之间的假设 `t1, t2, ..., tk` 是最优方案相矛盾。

如果上述不等式不满足，那么我们找出 `t1, t2, ..., tk` 中时间最长的那一门课 `tj`，如果 `tj > t0`，那么我们可以将 `tj` 用 `t0` 代替，即 `t1, t2, ..., tj-1, tj+1, ..., tk, t0` 是一种最优方案。这里同样可以使用反证法来证明。如果 `tj <= t0`，那么最优方案仍然为 `t1, t2, ..., tk`。

因此我们依次遍历每一门课程，通过上面的方法，就可以得到最优方案。我们就可以通过优先队列在若干个数中选出最大值，在遍历每一门课程 `(ti, di)` 时，依次进行如下操作：

- 如果当前优先队列中所有课程的时间之和 `t` 与 `ti` 之和小于等于 `di`，那么就把 `(ti, di)` 加入优先队列中；
- 如果当前优先队列中所有课程的时间之和 `t` 与 `ti` 之和大于 `di`，那么找到当前优先队列中课程时间最大的课程 `(tj, dj)`（即为堆顶），如果 `tj > ti`，则将它移出优先队列，并把 `(ti, di)` 加入优先队列中。

在所有的课程都判断完毕后，优先队列中包含的课程数目就代表了最多能选择的课程数目。

复杂度分析

- 时间复杂度：$O(N \log N)$，其中 N 是课程的数目。
- 空间复杂度：$O(N)$。

```
public class Solution {
    public int scheduleCourse(int[][] courses) {
        Arrays.sort(courses, (a, b) -> a[1] - b[1]);
        PriorityQueue < Integer > queue = new PriorityQueue < > ((a, b)
 -> b - a);
        int time = 0;
        for (int[] c: courses) {
            if (time + c[0] <= c[1]) {
                queue.offer(c[0]);
                time += c[0];
            } else if (!queue.isEmpty() && queue.peek() > c[0]) {
                time += c[0] - queue.poll();
                queue.offer(c[0]);
            }
        }
        return queue.size();
    }
}
```

51、翻倍（lintcode1328）【贪心】

1328. 翻倍

给定数字 A 和 B 以及系数 p 和 q。
每次操作可以将 A 变成 $A + p$ 或者将 p 变成 $p * q$。
返回最少变化次数使得 $B <= A$

样例
示例 1:
输入:
A = 2
B = 10
p = 1
q = 2
输出: 4
解释: 通过三次变换之后，p = 8，然后 A + p = 10，所以最终返回4.
示例 2:
输入:
A = 2
B = 13
p = 8
q = 4
输出: 2
解释: 通过一次变换之后，p = 32，然后 A + p = 34，所以最终返回2.
注意事项
$1 \le A \le 1e^9$
$1 \le B \le 1e^9$
$1 \le p \le 1e^9$
$2 \le q \le 10$

```
public class Solution {
    /**
     * @param A: an Integer
     * @param B: an Integer
     * @param p: an Integer
     * @param q: an Integer
     * @return: Return the minimum time
     */
    public int DoubleChange(int A, int B, int p, int q) {
        // write your code here
        int times = 0;
        Long res = new Long(A);
        Long pchange = new Long(p);

        while(res < B){
            if((res + pchange) >= B){
                res += pchange;
            }
            else{
                pchange = pchange * q;
            }
            times ++;
        }
        return times;
    }
}
```

52、跳柱子（lintcode1125）

1125. 跳柱子

中文 English

小易有 n 根柱子，第i根柱子的高度为h_i，一开始小易站在第一根柱子上。小易可以从i根柱子跳到第j根柱子，当且仅当$h_j \le h_i$并且$1 \le j - i \le k$，其中 k 为指定的一个数字。另外小易拥有一次放超能力的机会，
这个超能力能让小易从柱子i跳到任意满足$1 \le j - i \le k$的柱子而无视柱子高度的限制。
现在小易想知道小易是否能到达第 n 根柱子。

样例
例1:

```
输入:h=[3,2,7,4],k=2
输出:true
解释:你可以使用超能力，从第一根跳到第三根。然后按照规则，从第三根跳到第四根
```

例2:

```
输入:h=[3,4,5,6],k=2
输出:false
```

注意事项
$2 \le n, k \le 1000$
$1 \le h[i] \le 10^9$

利用dpi来保存状态。 dpi代表未使用超能力能到达的位置，dpi代表已使用过一次。 当满足跳跃的两个条件的时候，直接转移即可。 $dp[j][0] = max(dp[j][0], dp[i][0])$
$dp[j][1] = max(dp[j][1], dp[i][0])$ 或者使用超能力， $dp[j][1] = max(dp[j][1], dp[i][0])$

禁止商用 公众号：yes的练级攻略

```
public class Solution {
    /**
     * @param h: the height of n pillar
     * @param k: the limit
     * @return: Xiao Yi can or can't reach the n-th pillar
     */
    public boolean jumpPillar(int[] h, int k) {
        // write your code here.
        int[][] dp = new int[10500][2];
        int n = h.length;
        for (int i = 0;i < n + 1;i++){
            dp[i][0] = 0;
            dp[i][1] = 0;
        }
        dp[0][0] = 1;
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j <= Math.min(n - 1, i + k); j++) {
                if (h[i] >= h[j])  {
                    dp[j][0] = Math.max(dp[j][0], dp[i][0]);
                    dp[j][1] = Math.max(dp[j][1], dp[i][0]);
                }
                dp[j][1] = Math.max(dp[j][1], dp[i][0]);
            }
        if (dp[n - 1][0] + dp[n - 1][1] != 0) return true;
        return false;
    }
}
```

53、跳跃游戏（55）【贪心】

55. 跳跃游戏

难度 中等 752 收藏 分享 切换为英文 关注 反馈

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。
```

示例 2:

```
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/jump-game/solution/tiao-yue-you-xi-by-leetcode-solution/

文字题解

方法一：贪心

我们可以用贪心的方法解决这个问题。

设想一下，对于数组中的任意一个位置 y，我们如何判断它是否可以到达？根据题目的描述，只要存在一个位置 x，它本身可以到达，并且它跳跃的最大长度为 $x + nums[x]$，这个值大于等于 y，即 $x + nums[x] \geq y$，那么位置 y 也可以到达。

换句话说，对于每一个可以到达的位置 x，它使得 $x + 1, x + 2, \cdots, x + nums[x]$ 这些连续的位置都可以到达。

这样以来，我们依次遍历数组中的每一个位置，并实时维护 **最远可以到达的位置**。对于当前遍历到的位置 x，如果它在 **最远可以到达的位置** 的范围内，那么我们就可以从起点通过若干次跳跃到达该位置，因此我们可以用 $x + nums[x]$ 更新 **最远可以到达的位置**。

在遍历的过程中，如果 **最远可以到达的位置** 大于等于数组中的最后一个位置，那就说明最后一个位置可达，我们就可以直接返回 `True` 作为答案。反之，如果在遍历结束后，最后一个位置仍然不可达，我们就返回 `False` 作为答案。

以题目中的示例一

```
[2, 3, 1, 1, 4]
```

为例：

- 我们一开始在位置 0，可以跳跃的最大长度为 2，因此最远可以到达的位置被更新为 2；
- 我们遍历到位置 1，由于 $1 \leq 2$，因此位置 1 可达。我们用 1 加上它可以跳跃的最大长度 3，将最远可以到达的位置更新为 4。由于 4 大于等于最后一个位置 4，因此我们直接返回 `True`。

我们再来看看题目中的示例二

```
[3, 2, 1, 0, 4]
```

- 我们一开始在位置 0，可以跳跃的最大长度为 3，因此最远可以到达的位置被更新为 3；
- 我们遍历到位置 1，由于 $1 \leq 3$，因此位置 1 可达，加上它可以跳跃的最大长度 2 得到 3，没有超过最远可以到达的位置；
- 位置 2、位置 3 同理，最远可以到达的位置不会被更新；
- 我们遍历到位置 4，由于 $4 > 3$，因此位置 4 不可达，我们也就不考虑它可以跳跃的最大长度了。

在遍历完成之后，位置 4 仍然不可达，因此我们返回 `False`。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为数组的大小。只需要访问 `nums` 数组一遍，共 n 个位置。
- 空间复杂度：$O(1)$，不需要额外的空间开销。

```java
class Solution {
    public boolean canJump(int[] nums) {
        int n = nums.length;
        int right = 0;
        for (int i = 0; i < n; i++){
            if (i <= right) {
                right = Math.max(right, i + nums[i]);
```

```
                if (right >= n - 1) return true;
            }
        }
        return false;
    }
}
```

54、跳跃游戏 2（45）【贪心】

45. 跳跃游戏 II

难度 困难 647 收藏 分享 切换为英文 关注 反馈

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

输入: [2,3,1,1,4]
输出: 2
解释: 跳到最后一个位置的最小跳跃数是 2。
从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

说明:

假设你总是可以到达数组的最后一个位置。

https://leetcode-cn.com/problems/jump-game-ii/solution/tiao-yue-you-xi-ii-by-leetcode-solution/

公众号 yes的练级攻略

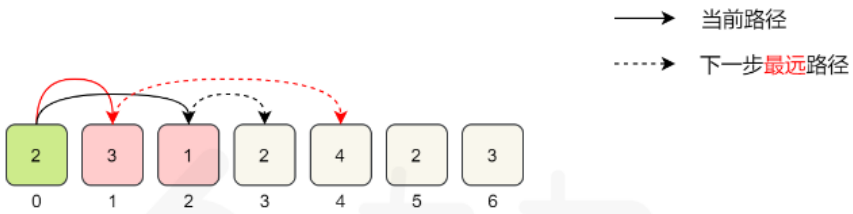
方法二：正向查找可到达的最大位置

方法一虽然直观，但是时间复杂度比较高，有没有办法降低时间复杂度呢？

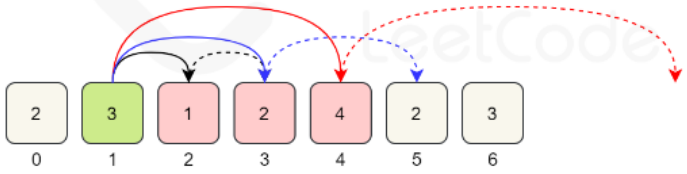
如果我们「贪心」地进行正向查找，每次找到可到达的最远位置，就可以在线性时间内得到最少的跳跃次数。

例如，对于数组 $[2,3,1,2,4,2,3]$，初始位置是下标 0，从下标 0 出发，最远可到达下标 2。下标 0 可到达的位置中，下标 1 的值是 3，从下标 1 出发可以达到更远的位置，因此第一步到达下标 1。

从下标 1 出发，最远可到达下标 4。下标 1 可到达的位置中，下标 4 的值是 4，从下标 4 出发可以达到更远的位置，因此第二步到达下标 4。



从下标 0 出发，可以跳到下标 1 和下标 2，下标 1 可以跳得更远，选择下标 1



从下标 1 出发，可以跳到下标 2、3 和 4，下标 4 可以跳得更远，选择下标 4

在具体的实现中，我们维护当前能够到达的最大下标位置，记为边界。我们从左到右遍历数组，到达边界时，更新边界并将跳跃次数增加 1。

在遍历数组时，我们不访问最后一个元素，这是因为在访问最后一个元素之前，我们的边界一定大于等于最后一个位置，否则就无法跳到最后一个位置了。如果访问最后一个元素，在边界正好为最后一个位置的情况下，我们会增加一次「不必要的跳跃次数」，因此我们不必访问最后一个元素。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是数组长度。
- 空间复杂度：$O(1)$。

```
class Solution {
    public int jump(int[] nums) {
        int n = nums.length;
        int cnt = 0, end = 0, far = 0;
        for (int i = 0; i < n - 1; i++){
            far = Math.max(far, i + nums[i]);
            if (i == end) {
                end = far;
                cnt++;
            }
        }
```

```
            return cnt;
        }
    }
```

55、跳跃游戏 5（1340）

1340. 跳跃游戏 V

难度 困难 43 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `arr` 和一个整数 `d` 。每一步你可以从下标 `i` 跳到：

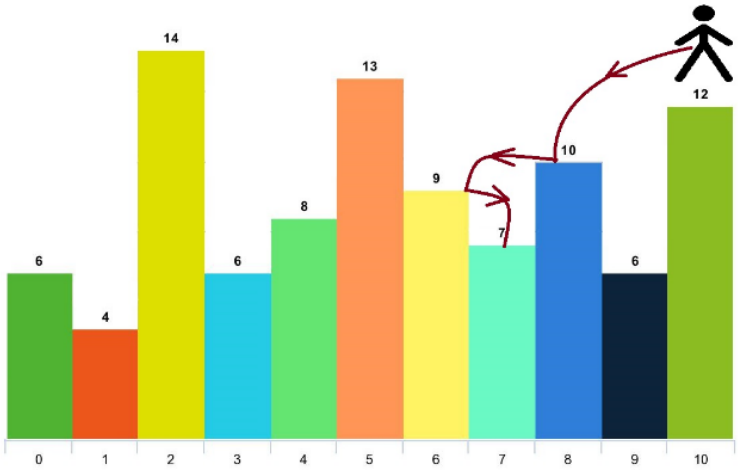
- `i + x` ，其中 `i + x < arr.length` 且 `0 < x <= d` 。
- `i - x` ，其中 `i - x >= 0` 且 `0 < x <= d` 。

除此以外，你从下标 `i` 跳到下标 `j` 需要满足：`arr[i] > arr[j]` 且 `arr[i] > arr[k]` ，其中下标 `k` 是所有 `i` 到 `j` 之间的数字（更正式的，`min(i, j) < k < max(i, j)`）。

你可以选择数组的任意下标开始跳跃。请你返回你 **最多** 可以访问多少个下标。

请注意，任何时刻你都不能跳到数组的外面。

示例 1：



yes的练级攻略

输入： arr = [6,4,14,6,8,13,9,7,10,6,12], d = 2
输出： 4
解释： 你可以从下标 10 出发，然后如上图依次经过 10 --> 8 --> 6 --> 7 。
注意，如果你从下标 6 开始，你只能跳到下标 7 处。你不能跳到下标 5 处因为 13 > 9 。你也不能跳到下标 4 处，因为下标 5 在下标 4 和 6 之间且 13 > 9 。
类似的，你不能从下标 3 处跳到下标 2 或者下标 1 处。

示例 2：

输入： arr = [3,3,3,3,3], d = 3
输出： 1
解释： 你可以从任意下标处开始且你永远无法跳到任何其他坐标。

示例 3：

输入： arr = [7,6,5,4,3,2,1], d = 1
输出： 7
解释： 从下标 0 处开始，你可以按照数值从大到小，访问所有的下标。

示例 4：

输入： arr = [7,1,7,1,7,1], d = 2
输出： 2

示例 5：

输入： arr = [66], d = 1
输出： 1

提示：

- 1 <= arr.length <= 1000
- 1 <= arr[i] <= 10^5
- 1 <= d <= arr.length

https://leetcode-cn.com/problems/jump-game-v/solution/a-fei-xue-suan-fa-zhi-si-ke-yi-dao-ti-1340-tiao-yu/

方法1：DP（从柱子的最小值到最大值）

- 先对 arr[i] 的值按从小到大排列，最后可以想象成 (k,v) 的这种形式，按 k 来排序，k 指的是 arr[i] 的值，v 的值指的是 index 索引,写入一个 idx 的数组中，存的是排序好的 index 的值 - 涉及到的排序的方式：

```
Arrays.sort(idx, Comparator.comparingInt(o -> arr[o]));//按升序排列
//        Arrays.sort(idx, (Comparator) ((o1, o2) -> arr[(int) o1] - arr[(int) o2]));
```

- 两种均可，第一种是 lambda 表达式
- 准备一个 dp ，**dp[i] 表示的是下表 i 最多可以跳跃的次数，或是访问的次数**，当 i 处于某个位置的时候，i 可以往左边跳跃，也可以往右边跳跃
  - 往左边跳跃 dp[i]=max{dp[j]+1,dp[i]} j∈[i−d,i−1]
  - 往右边跳跃 dp[i]=max{dp[j]+1,dp[i]} j∈[i+1,i+d]
  - 因为从 i 跳跃到 j 位置，都是一次跳跃，所以要 +1
  - 要判断下 arr[j]>arr[i] 的情况，此种情况表示跳跃不了的，要 break 掉
- 在 for loop 的过程中，记录下 max 值，最后返回

方法 1：动规

```
class Solution {
    public int maxJumps(int[] arr, int d) {
        int n = arr.length;
        Integer[] idx = new Integer[n]; //存储的arr 的下标
        for (int i = 0; i < n; i++){
            idx[i] = i;
        }
```

```
        Arrays.sort(idx, Comparator.comparingInt(o -> arr[o]));//按升序排列
        //Arrays.sort(idx, (Comparator) ((o1, o2) -> arr[(int) o1] - arr[(int) o2]));
        int[] dp = new int[n];
        Arrays.fill(dp, 1);
        int res = 0;
        for (int k = 0; k < n; k++){
            int i = idx[k];
            for (int j = i - 1; j >= 0 && j >= i - d; j--){
                if (arr[j] >= arr[i]) break;
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
            for (int j = i + 1; j < n && j <= i + d; j++){
                if (arr[j] >= arr[i]) break;
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}
```

https://leetcode-cn.com/problems/jump-game-v/solution/java-jian-dan-de-shu-ju-jie-gou-dong-tai-gui-hua-y/

站在一根柱子上，既可以往左跳也可以往右跳，只要在范围内。对于任意的柱子，能访问到的柱子数等于往左或往右跳到的那一个柱子所能访问的数目再加它本身。所以可以用一个一维数组 `dp[]` 来存放每个柱子最多能访问的数量。
使用递归可以利用 `dp[]` 的值来免去排序之类的处理，使得解法更加简洁快速。
在跳的过程中（ `d` 范围内），如果遇到了比当前柱子高或等的柱子，就不再往更远的地方跳了。

这里使用 dp 数组来记忆的一个技巧就是避免重复计算。对于每一根柱子，他最小的访问数目是1，就是他自己（比如左右紧挨着的柱子都比它高），数组初始化的时候每一个元素都是0，后面的遍历中，如果取到的柱子 `i` 有 `dp[i]!=0` ，说明这个柱子已经计算过了，直接取这个值就可以用，而无需重新计算；如果 `dp[i]` 是0，就要获取它左边或右边数目最大的柱子，再+1。注意一下限制条件。
通过这种记忆化的方法将递归结果存储下来的类似题目还有 329.矩阵中的最长递增路径。

方法 2：动规+memo

```
class Solution {
    int[] arr;
    int n; //数组长度
    int d;
    int[] dp;   //用来存储每个柱子的最大结果
    public int maxJumps(int[] arr, int d) {
        this.arr = arr;
        this.n = arr.length;
        this.d = d;
        dp = new int[n];
        int ans = 0;
```

```
        for(int i = 0; i < n; i++){
            ans = Math.max(ans, getMaxFromOnePoint(i));
        }
        return ans;
    }

    private int getMaxFromOnePoint(int p){
        if(dp[p] != 0)  return dp[p];   //当前柱子已经计算过，直接返回它的值
        // 如果没有，分别计算它往左和往右跳一次可以得到的最大值
        int leftMax = 0;
        int l = 1;  // 往左跳的距离
        while(p-l >= 0 && l <= d){
            if(arr[p-l] >= arr[p]) break; //遇到了高柱子挡路，只能结束
            if(dp[p-l] == 0)  dp[p-l] = getMaxFromOnePoint(p-l);
            leftMax = Math.max(leftMax, dp[p-l]);
            l++;
        }
        // 同理右边
        int rightMax = 0;
        int r = 1;
        while(p+r < n && r <= d){
            if(arr[p+r] >= arr[p]) break;
            if(dp[p+r] == 0)  dp[p+r] = getMaxFromOnePoint(p+r);
            rightMax = Math.max(rightMax, dp[p+r]);
            r++;
        }

        return Math.max(leftMax, rightMax)+1;
    }
}
```

56、最小跳跃次数（LCP 09）

LCP 09. 最小跳跃次数

难度 困难　34　收藏　分享　切换为英文　关注　反馈

为了给刷题的同学一些奖励，力扣团队引入了一个弹簧游戏机。游戏机由 `N` 个特殊弹簧排成一排，编号为 `0` 到 `N-1`。初始有一个小球在编号 `0` 的弹簧处。若小球在编号为 `i` 的弹簧处，通过按动弹簧，可以选择把小球向右弹射 `jump[i]` 的距离，或者向左弹射到任意左侧弹簧的位置。也就是说，在编号为 `i` 弹簧处按动弹簧，小球可以弹向 `0` 到 `i-1` 中任意弹簧或者 `i+jump[i]` 的弹簧（若 `i+jump[i]>=N`，则表示小球弹出了机器）。小球位于编号 0 处的弹簧时不能再向左弹。

为了获得奖励，你需要将小球弹出机器。请求出最少需要按动多少次弹簧，可以将小球从编号 `0` 弹簧弹出整个机器，即向右越过编号 `N-1` 的弹簧。

示例 1：

输入：`jump = [2, 5, 1, 1, 1, 1]`

输出：`3`

解释：小 Z 最少需要按动 3 次弹簧，小球依次到达的顺序为 0 -> 2 -> 1 -> 6，最终小球弹出了机器。

限制：

- `1 <= jump.length <= 10^6`
- `1 <= jump[i] <= 10000`

https://leetcode-cn.com/problems/zui-xiao-tiao-yue-ci-shu/solution/jian-dan-yi-dong-de-dong-tai-gui-hua-wan-fa-by-don/

从右向左计算 dp 值(从后向前)，当前位置如果为 i 则它如果直接跳到右边（前面）去就是 dp[jump[i]+i]+1（这个值已经计算过了），计算出当前位置 dp[i]之后，当前位置 i 可以影响 i+1 到 dp[j] >= dp[i]+1 位置上的值（因为某个位置可以跳到左边任意位置）注意遍历到 dp[j]>=dp[i]+1 即可。

方法 1：动规（13ms）

```
class Solution {
    public int minJump(int[] jump) {
        int n = jump.length;
        // dp[i] 表示从 i 这里跳出去要多少步
        int[] dp = new int[n];
        dp[n - 1] = 1;
        for (int i = n - 2; i >= 0; --i){
           // 如果 jump[i] + i >= n 说明可以直接跳出去更新 dp[i]
           // 否则考虑向右跳到 jump[i] + i 然后再跳出去
            dp[i] = jump[i] + i >= n ? 1 : dp[jump[i] + i] + 1;
// 一旦 dp[i] 更新了，那么对于所有的 j(i < j < n)，都多了一种选择，就是先往左挑到 i 然后再往右跳。
// 也就是说 dp[i] 一旦更新，对于右边的 dp[j] 可以往左跳，也可以往右跳，两者取小
            for (int j = i + 1; i < n && dp[j] >= dp[i] + 1; ++j){
                dp[j] = dp[i] + 1;
            }
        }
        return dp[0];
    }
}
```

方法 2：BFS（134ms）

```
class Solution {
    public int minJump(int[] jump) {
        LinkedList<Integer> list = new LinkedList();
        list.addLast(jump[0]);
        list.addLast(-1);
        //bfs 模拟下一步和往左走  当放入-1 表示这次模拟已经结束
        //通过 set 来和 max 保证不重复放入，同一个 index 重复放入肯定比之前的要步数多
        Set<Integer> set = new HashSet();
        set.add(jump[0]);
        int count = 1;
        int max = 0;
        while(list.size() > 0){
            int index = list.pollFirst();
            if(index>jump.length-1){
                return count;
```

```
            } else if(index==-1){
                count++;
                list.addLast(-1);
            } else {
                //跳到下一步
                int temp = index+jump[index];
                //处理已经进入过就不用进入了
                if(!set.contains(temp)){
                    set.add(temp);
                    list.addLast(temp);
                }
                //当前 0-index-1 可以进去
                //max 是之前已经走过的 在上一步已经放入过所以就不用在判断了
                for(int i=index-1;i>max;i--){
                    if(!set.contains(i)){
                        set.add(i);
                        list.addLast(i);
                    }
                }
                //更新 max;
                max=Math.max(index, max);
            }
        }
        return count;
    }
}
```

57、老鼠跳跃（lintcode 1861）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

1861. 老鼠跳跃 中文 Engli

有一个老鼠从高为 n 的楼梯顶部跳跃下来，这个老鼠在偶数次跳跃时可以跳1, 3或者4个台阶，奇数次可以跳跃1, 2或者4个台阶。
但是楼梯中间会有一些台阶上有胶水，如果跳到那些台阶上，老鼠就会被直接粘住，无法继续跳跃。你需要求出从这个楼梯顶部开始，
老鼠有多少种方法能够到达地面，即第0层。若超过地面，也算是可以到达。例如从1跳跃到-1，和从1跳跃到0的方案不同。
楼梯有无胶水的状态是从高往低输入的，即arr[0]代表当前老鼠所在的位置无胶水。

样例
Example1:

```
Input:[0,0,0]
Output:5
Explanation:
There are 3 steps.
The step 2  is the starting point without glue.
Step 1, no glue.
Step 0, no glue.
The mouse jump plans is:
2--odd(1)-->1--even(1)-->0
2--odd(1)-->1--even(3)-->-2
2--odd(1)-->1--even(4)-->-3
2--odd(2)-->0
2--odd(4)-->-2
```

Example2:

```
Input:[0,0,1,0]
Output:3
Explanation:
There are 4 steps.
The step 3  is the starting point without glue.
Step 2, no glue.
Step 1, have glue.
Step 0, no glue.
```

注意事项
2<=n<=50000
arr[i]=1代表台阶上有胶水，0代表没有
输入保证最高的台阶和最低的为0
答案需对1e9+7 取模

https://www.jiuzhang.com/solution/rat-jump/#tag-lang-java

```java
public class Solution {
    /**
     * @param arr: the steps whether have glue
     * @return: the sum of the answers
     */
    public int ratJump(int[] arr) {
        int[] odds = new int[] {1, 2, 4};
        int[] evens = new int[] {1, 3, 4};
        int mod = 1000000007;
        int n = arr.length;
        int[][] dp = new int[n][2];
        dp[0][0] = 1;
        dp[0][1] = 1;
        for (int i = 1; i < n; i++){
            if (arr[n - i - 1] == 1) continue;
            for (int j : odds) {
                dp[i][0] = (dp[i][0] + dp[Math.max(i - j, 0)][1]) % mod;
            }
            for (int j : evens) {
                dp[i][1] = (dp[i][1] + dp[Math.max(i - j, 0)][0]) % mod;
            }
        }
        return dp[n-1][0];
```

公众号：yes的练级攻略

```
    }
}
```

58、预测赢家（486）

486. 预测赢家

难度 中等 271 收藏 分享 切换为英文 关注 反馈

给定一个表示分数的非负整数数组。 玩家 1 从数组任意一端拿取一个分数，随后玩家 2 继续从剩余数组任意一端拿取分数，然后玩家 1 拿，……。每次一个玩家只能拿取一个分数，分数被拿取之后不再可取。直到没有剩余分数可取时游戏结束。最终获得分数总和最多的玩家获胜。

给定一个表示分数的数组，预测玩家1是否会成为赢家。你可以假设每个玩家的玩法都会使他的分数最大化。

示例 1：

输入： [1, 5, 2]
输出： False
解释： 一开始，玩家1可以从1和2中进行选择。
如果他选择 2（或者 1 ），那么玩家 2 可以从 1（或者 2 ）和 5 中进行选择。如果玩家 2 选择了 5 ，那么玩家 1 则只剩下 1（或者 2 ）可选。
所以，玩家 1 的最终分数为 1 + 2 = 3，而玩家 2 为 5 。
因此，玩家 1 永远不会成为赢家，返回 False 。

示例 2：

输入： [1, 5, 233, 7]
输出： True
解释： 玩家 1 一开始选择 1 。然后玩家 2 必须从 5 和 7 中进行选择。无论玩家 2 选择了哪个，玩家 1 都可以选择 233 。
最终，玩家 1（234 分）比玩家 2（12 分）获得更多的分数，所以返回 True，表示玩家 1 可以成为赢家。

提示：

- 1 <= 给定的数组长度 <= 20.
- 数组里所有分数都为非负数且不会大于 10000000 。
- 如果最终两个玩家的分数相等，那么玩家 1 仍为赢家。

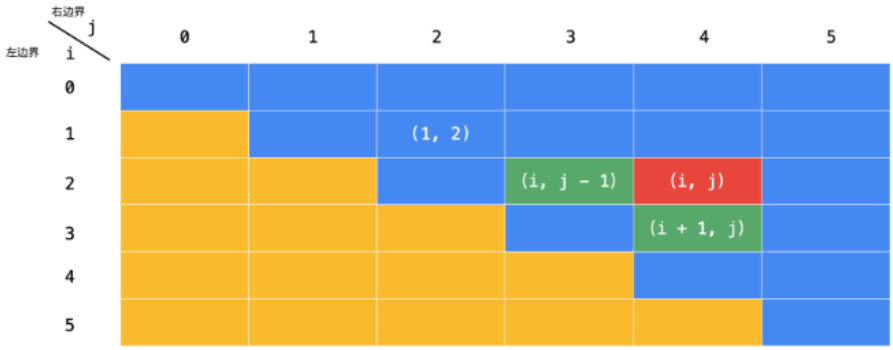
https://leetcode-cn.com/problems/predict-the-winner/solution/ji-yi-hua-di-gui-dong-tai-gui-hua-java-by-liweiwei/

方法二：动态规划

状态定义：`dp[i][j]` 表示作为先手，在区间 `nums[i..j]` 里进行选择可以获得的**相对分数**。相对分数的意思是：**当前自己的选择得分为正，对手的选择得分为负。**

☆ 收藏

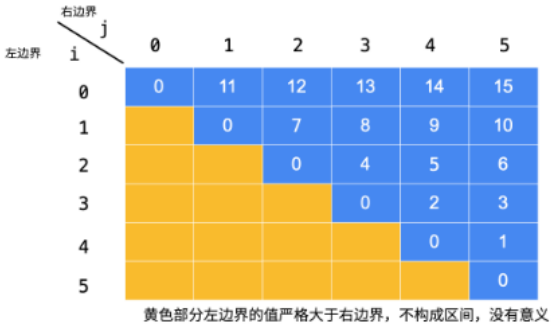
```
dp[i][j] = max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1])
```



在 i <= j 的时候才有意义，故只填这张表的上半部分。

红色部分依赖绿色，所以先填下面的对角线，再填上面的对角线。

填表方式 1：



黄色部分左边界的值严格大于右边界，不构成区间，没有意义

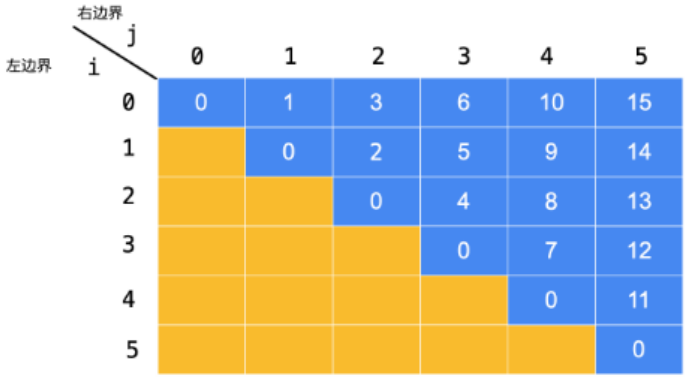
蓝色部分的数值表示计算顺序，从小到大，对角线由于很好计算，计算不分后，统一设置为 0。代码如下：

```
for (int i = len - 2; i >= 0; i--) {
    for (int j = i + 1; j < len; j++) {
        dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);
    }
}
```

方法 1:

```
public class Solution {
    // 状态转移方程：
dp[i][j] = max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1])
    public boolean PredictTheWinner(int[] nums) {
        int len = nums.length;
        int[][] dp = new int[len][len];
        // dp[i][j]：作为先手，在区间 nums[i..j] 里进行选择可以获得的相对分数
        for (int i = 0; i < len; i++) {
            dp[i][i] = nums[i];
        }
        for (int i = len - 2; i >= 0; i--) {
            for (int j = i + 1; j < len; j++) {
                dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j -
 1]);
            }
        }
        return dp[0][len - 1] >= 0;
    }
}
```

填表方式 2：



黄色部分左边界的值严格大于右边界，不构成区间，没有意义

蓝色部分的数值表示计算顺序，从小到大，对角线由于很好计算，计算不分后，统一设置为 0。代码如下：

```
for (int j = 1; j < len; j++) {
    for (int i = j - 1; i >= 0; i--) {
        dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);
    }
}
```

方法 2:

```
public class Solution {
    // 状态转移方程：
dp[i][j] = max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1])
    public boolean PredictTheWinner(int[] nums) {
        int len = nums.length;
        int[][] dp = new int[len][len];
        // dp[i][j]：作为先手，在区间 nums[i..j] 里进行选择可以获得的相对分数
        for (int i = 0; i < len; i++) {
            dp[i][i] = nums[i];
        }
        for (int j = 1; j < len; j++) {
            for (int i = j - 1; i >= 0; i--) {
                dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j -
 1]);
            }
        }
        return dp[0][len - 1] >= 0;
    }
}
```

59、石头游戏（877）

更多硬核内容就在公众号「yes 的练级攻略」

877. 石子游戏

难度 中等 160 收藏 分享 切换为英文 关注 反馈

亚历克斯和李用几堆石子在做游戏。偶数堆石子**排成一行**，每堆都有正整数颗石子 `piles[i]` 。

游戏以谁手中的石子最多来决出胜负。石子的总数是奇数，所以没有平局。

亚历克斯和李轮流进行，亚历克斯先开始。 每回合，玩家从行的开始或结束处取走整堆石头。 这种情况一直持续到没有更多的石子堆为止，此时手中石子最多的玩家获胜。

假设亚历克斯和李都发挥出最佳水平，当亚历克斯赢得比赛时返回 `true` ，当李赢得比赛时返回 `false` 。

示例：

```
输入：[5,3,4,5]
输出：true
解释：
亚历克斯先开始，只能拿前 5 颗或后 5 颗石子 。
假设他取了前 5 颗，这一行就变成了 [3,4,5] 。
如果李拿走前 3 颗，那么剩下的是 [4,5]，亚历克斯拿走后 5 颗赢得 10 分。
如果李拿走后 5 颗，那么剩下的是 [3,4]，亚历克斯拿走后 4 颗赢得 9 分。
这表明，取前 5 颗石子对亚历克斯来说是一个胜利的举动，所以我们返回 true 。
```

提示：

1. `2 <= piles.length <= 500`
2. `piles.length` 是偶数。
3. `1 <= piles[i] <= 500`
4. `sum(piles)` 是奇数。

二、石头游戏

游戏规则是这样的：你和你的朋友面前有一排石头堆，用一个数组 piles 表示，piles[i] 表示第 i 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。

假设你们都很聪明，由你第一个开始拿，请你写一个算法，输入一个数组 piles，返回你是否能赢（true 或 false）。

注意，石头的堆的数量为偶数，所以你们两人拿走的堆数一定是相同的。石头的总数为奇数，也就是你们最后不可能拥有相同多的石头，一定有胜负之分。

举个例子，`piles=[2, 1, 9, 5]` ，你先拿，可以拿 2 或者 5，你选择 2。

`piles=[1, 9, 5]` ，轮到对手，可以拿 1 或 5，他选择 5。

`piles=[1, 9]` 轮到你拿，你拿 9。

最后，你的对手只能拿 1 了。

这样下来，你总共拥有 `2 + 9 = 11` 颗石头，对手有 `5 + 1 = 6` 颗石头，你是可以赢的，所以算法应该返回 true。

你看到了，并不是简单的挑数字大的选，为什么第一次选择 2 而不是 5 呢？因为 5 后面是 9，你要是贪图一时的利益，就把 9 这堆石头暴露给对手了，那你就要输了。

这也是强调双方都很聪明的原因，算法也是求最优决策过程下你是否能赢。

这道题又涉及到两人的博弈，也可以用动态规划算法暴力试，比较麻烦。但我们只要对规则深入思考，就会大惊失色：只要你足够聪明，你是必胜无疑的，因为你是先手。

这是为什么呢，因为题目有两个条件很重要：一是石头总共有偶数堆，石头的总数是奇数。这两个看似增加游戏公平性的条件，反而使该游戏成为了一个割韭菜游戏。我们以 `piles=[2, 1, 9, 5]` 讲解，假设这四堆石头从左到右的索引分别是 1，2，3，4。

如果我们把这四堆石头按索引的奇偶分为两组，即第 1、3 堆和第 2、4 堆，那么这两组石头的数量一定不同，也就是说一堆多一堆少。因为石头的总数是奇数，不能被平分。

而作为第一个拿石头的人，你可以控制自己拿到所有偶数堆，或者所有的奇数堆。

你最开始可以选择第 1 堆或第 4 堆。如果你想要偶数堆，你就拿第 4 堆，这样留给对手的选择只有第 1、3 堆，他不管怎么拿，第 2 堆又会暴露出来，你就可以拿。同理，如果你想拿奇数堆，你就拿第 1 堆，留给对手的只有第 2、4 堆，他不管怎么拿，第 3 堆又给你暴露出来了。

也就是说，你可以在第一步就观察好，奇数堆的石头总数多，还是偶数堆的石头总数多，然后步步为营，就一切尽在掌控之中了。知道了这个漏洞，可以整一整不知情的同学了。

方法 1：数学（0ms）

```
class Solution {
    public boolean stoneGame(int[] piles) {
        return true;
    }
}
```

https://leetcode-cn.com/problems/stone-game/solution/shi-zi-you-xi-by-leetcode/

方法一：动态规划

思路

让我们改变游戏规则，使得每当李得分时，都会从亚历克斯的分数中扣除。

令 `dp(i, j)` 为亚历克斯可以获得的最大分数，其中剩下的堆中的石子数是 `piles[i], piles[i+1], ..., piles[j]`。

这在比分游戏中很自然：我们想知道游戏中每个位置的值。

我们可以根据 `dp(i + 1, j)` 和 `dp(i, j-1)` 来制定 `dp(i, j)` 的递归，我们可以使用动态编程以不重复这个递归中的工作。该方法可以输出正确的答案，因为状态形成一个DAG（有向无环图）。

算法

当剩下的堆的石子数是 `piles[i], piles[i+1], ..., piles[j]` 时，轮到的玩家最多有 2 种行为。

可以通过比较 `j-i` 和 `N modulo 2` 来找出轮到的人。

如果玩家是亚历克斯，那么她将取走 `piles[i]` 或 `piles[j]` 颗石子，增加她的分数。

之后，总分为 `piles[i] + dp(i+1, j)` 或 `piles[j] + dp(i, j-1)`；我们想要其中的最大可能得分。

如果玩家是李，那么他将取走 `piles[i]` 或 `piles[j]` 颗石子，减少亚历克斯这一数量的分数。

之后，总分为 `-piles[i] + dp(i+1, j)` 或 `-piles[j] + dp(i, j-1)`；我们想要其中的最小可能得分。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 是石子堆的数目。
- 空间复杂度：$O(N^2)$，该空间用以存储每个子游戏的中间结果。

方法 2：动规（10ms）

```
class Solution {
    public boolean stoneGame(int[] piles) {
        int n = piles.length;
        // dp[i+1][j+1] = the value of the game [piles[i], ..., piles[j]].
        int[][] dp = new int[n+2][n+2];
        for (int size = 1; size <= n; ++size){
            for (int i = 0; i+size <= n; ++i){
                int j = i + size - 1;
                int parity = (j + i + n) % 2;  // j - i - n; but +x = -x (mod 2)
                if (parity == 1)
                    dp[i+1][j+1] = Math.max(piles[i] + dp[i+2][j+1], piles[j] + dp[i+1][j]);
                else
                    dp[i+1][j+1] = Math.min(-piles[i] + dp[i+2][j+1], -piles[j] + dp[i+1][j]);
            }
        }
        return dp[1][n] > 0;
    }
}
```

60、石头游戏 2（1140）

1140. 石子游戏 II

难度 中等 57 收藏 分享 切换为英文 关注 反馈

亚历克斯和李继续他们的石子游戏。许多堆石子 **排成一行**，每堆都有正整数颗石子 `piles[i]`。游戏以谁手中的石子最多来决出胜负。

亚历克斯和李轮流进行，亚历克斯先开始。最初，`M = 1`。

在每个玩家的回合中，该玩家可以拿走剩下的 **前** `X` 堆的所有石子，其中 `1 <= X <= 2M`。然后，令 `M = max(M, X)`。

游戏一直持续到所有石子都被拿走。

假设亚历克斯和李都发挥出最佳水平，返回亚历克斯可以得到的最大数量的石头。

示例：

输入： piles = [2,7,9,4,4]
输出： 10
解释：
如果亚历克斯在开始时拿走一堆石子，李拿走两堆，接着亚历克斯也拿走两堆。在这种情况下，亚历克斯可以拿到 2 + 4 + 4 = 10 颗石子。
如果亚历克斯在开始时拿走两堆石子，那么李就可以拿走剩下全部三堆石子。在这种情况下，亚历克斯可以拿到 2 + 7 = 9 颗石子。
所以我们返回更大的 10。

提示：

- `1 <= piles.length <= 100`
- `1 <= piles[i] <= 10 ^ 4`

https://leetcode-cn.com/problems/stone-game-ii/solution/java-dong-tai-gui-hua-qing-xi-yi-dong-17xing-by-lg/

更多硬核内容就在公众号「yes 的练级攻略」

思路

本题很明显要用记忆化搜索或者动态规划来求解，如果直接使用动态规划的话，我们要想清楚有哪些子状态需要存储。

首先一定要存储的是取到**某一个位置**时，已经得到的最大值或者后面能得到的最大值，但是光有位置是不够的，相同的位置有不同数量的堆可以取，所以我们还需存储**当前的M值**。

由于本题中的状态是从后往前递推的，如：假如最后只剩一堆，一定能算出来最佳方案，但是剩很多堆时不好算（依赖后面的状态）。所以我们选择从后往前递推。

递推公式

有了思路之后，我们就能很方便地定义dp数组了：

dp[i][j]表示剩余[i : len - 1]堆时，M = j的情况下，先取的人能获得的最多石子数

1. i + 2M >= len, dp[i][M] = sum[i : len - 1], 剩下的堆数能够直接全部取走，那么最优的情况就是剩下的石子总和
2. i + 2M < len, dp[i][M] = max(dp[i][M], sum[i : len - 1] - dp[i + x][max(M, x)]), 其中 1 <= x <= 2M，剩下的堆数不能全部取走，那么最优情况就是让下一个人取的更少。对于我所有的x取值，下一个人从x开始取起，M变为max(M, x)，所以下一个人能取dp[i + x][max(M, x)]，我最多能取sum[i : len - 1] - dp[i + x][max(M, x)]。

示例演示

对于piles = [2,7,9,4,4]，我们可以得到下图所示的dp数组，结果为dp[0][1]

☆收藏

M \ i	0	1	2	3	4
1	10	16	13	8	4
2	22	24	17	8	4
3	26	24	17	8	4
4	26	24	17	8	4
5	26	24	17	8	4

是不是非常清晰呢

```
class Solution {
    public int stoneGameII(int[] piles) {
        int n = piles.length, sum = 0;
        int[][] dp = new int[n][n+1];
        for (int i = n -1; i >= 0; i--){
          sum += piles[i];
          for (int M = 1; M <= n; M++){
              if (i + 2 * M >= n) dp[i][M] = sum;
              else {
                for (int x = 1; x <= 2 * M; x++){
                  dp[i][M] = Math.max(dp[i][M], sum - dp[i+x][Math.max(
M, x)]);
                }
              }
           }
```

```
            }
            return dp[0][1];
        }
    }
```

61、石头游戏 3（1406）

1406. 石子游戏 III

难度 困难 50 收藏 分享 切换为英文 关注 反馈

Alice 和 Bob 用几堆石子在做游戏。几堆石子排成一行，每堆石子都对应一个得分，由数组 `stoneValue` 给出。

Alice 和 Bob 轮流取石子，**Alice** 总是先开始。在每个玩家的回合中，该玩家可以拿走剩下石子中的的前 **1、2 或 3 堆石子** 。比赛一直持续到所有石头都被拿走。

每个玩家的最终得分为他所拿到的每堆石子的对应得分之和。每个玩家的初始分数都是 **0** 。比赛的目标是决出最高分，得分最高的选手将会赢得比赛，比赛也可能会出现平局。

假设 Alice 和 Bob 都采取 **最优策略** 。如果 Alice 赢了就返回 *"Alice"*，Bob 赢了就返回 *"Bob"*，平局（分数相同）返回 *"Tie"*。

示例 1：

输入： values = [1,2,3,7]
输出： "Bob"
解释： Alice 总是会输，她的最佳选择是拿走前三堆，得分变成 6 。但是 Bob 的得分为 7，Bob 获胜。

示例 2：

输入： values = [1,2,3,-9]
输出： "Alice"
解释： Alice 要想获胜就必须在第一个回合拿走前三堆石子，给 Bob 留下负分。
如果 Alice 只拿走第一堆，那么她的得分为 1，接下来 Bob 拿走第二、三堆，得分为 5 。之后 Alice 只能拿到分数 -9 的石子堆，输掉比赛。
如果 Alice 拿走前两堆，那么她的得分为 3，接下来 Bob 拿走第三堆，得分为 3 。之后 Alice 只能拿到分数 -9 的石子堆，同样会输掉比赛。
注意，他们都应该采取 **最优策略** ，所以在这里 Alice 将选择能够使她获胜的方案。

示例 4：

输入： values = [1,2,3,-1,-2,-3,7]
输出： "Alice"

示例 5：

输入： values = [-1,-2,-3]
输出： "Tie"

提示：

- `1 <= values.length <= 50000`
- `-1000 <= values[i] <= 1000`

https://leetcode-cn.com/problems/stone-game-iii/solution/java-3msshi-jian-fu-za-du-onkong-jian-fu-za-du-o1-/

解题思路

定义 `dp[i]` 的意义为从 `stoneValue[i]` 到 `stoneValue[N-1]` 这堆石头里，从 `i` 开始取石头的人所能够拿到的最多石头数。因为从 `i` 开始取石头的人可以取1、2或3堆石头，然后下一个人取到的最多石头数就分别是 `dp[i+1]` 、 `dp[i+2]` 和 `dp[i+3]` ，为了使从 `i` 开始取得的石头最多，等价于使下一个人取得的石头数最少即 `min(dp[i+1], dp[i+2], dp[i+3])` ，此时从 `i` 开始取得的最多石头数为$\sum_{j=i}^{N-1} stoneValue[j] - min(dp[i+1], dp[i+2], dp[i+3])$。最后根据 `2dp[0]` 与$\sum_{j=0}^{N-1} stoneValue[j]$的大小关系即可知道最终结果。时间复杂度为$O(n)$，空间复杂度为$O(n)$。

方法 1：一维动规

```
class Solution {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    public String stoneGameIII(int[] stoneValue) {
        int N = stoneValue.length;
        int[] dp = new int[N + 1];
        dp[N] = 0;
        dp[N - 1] = stoneValue[N - 1];
        int sum = stoneValue[N - 1];
        if (N - 2 >= 0) {
            sum += stoneValue[N - 2];
            dp[N - 2] = sum - Math.min(dp[N - 1], dp[N]);
        }
        for (int i = N - 3; i >= 0; --i) {
            sum += stoneValue[i];
            dp[i] = sum - Math.min(Math.min(dp[i + 1], dp[i + 2]), dp[i
 + 3]);
        }
        if (dp[0] * 2 > sum) {
            return "Alice";
        }
        else if (dp[0] * 2 < sum) {
            return "Bob";
        }
        else {
            return "Tie";
        }
    }
}
```

进一步可以减少空间复杂度，这是因为每次的状态 `dp[i]` 最多只与另外三个状态 `dp[i+1]` 、 `dp[i+2]` 及 `dp[i+3]` 相关，因此可以用三个变量进行保存，从而将空间复杂度降到了$O(1)$。

方法 2：常数级动规

```
class Solution {
    public String stoneGameIII(int[] stoneValue) {
        int N = stoneValue.length;
        int dp0, dp1 = stoneValue[N - 1], dp2 = 0;
        int sum = stoneValue[N - 1];
        if (N - 2 >= 0) {
            sum += stoneValue[N - 2];
            dp0 = sum - Math.min(dp1, dp2);
        }
        else {
            dp0 = dp1;
        }
        for (int i = N - 3; i >= 0; --i) {
            sum += stoneValue[i];
```

```
            int temp = dp0;
            dp0 = sum - Math.min(Math.min(dp0, dp1), dp2);
            dp2 = dp1;
            dp1 = temp;
        }
        if (dp0 * 2 > sum) {
            return "Alice";
        }
        else if (dp0 * 2 < sum) {
            return "Bob";
        }
        else {
            return "Tie";
        }
    }
}
```

62、石子游戏 4（1510）

1510. 石子游戏 IV

难度 困难 7 收藏 分享 切换为英文 关注 反馈

Alice 和 Bob 两个人轮流玩一个游戏，Alice 先手。

一开始，有 `n` 个石子堆在一起。每个人轮流操作，正在操作的玩家可以从石子堆里拿走 **任意** 非零 **平方数** 个石子。

如果石子堆里没有石子了，则无法操作的玩家输掉游戏。

给你正整数 `n` ，且已知两个人都采取最优策略。如果 Alice 会赢得比赛，那么返回 `True` ，否则返回 `False` 。

示例 1：

输入： `n = 1`
输出： `true`
解释： `Alice 拿走 1 个石子并赢得胜利，因为 Bob 无法进行任何操作。`

示例 2：

输入： `n = 2`
输出： `false`
解释： `Alice 只能拿走 1 个石子，然后 Bob 拿走最后一个石子并赢得胜利（2 -> 1 -> 0）。`

示例 3：

输入： `n = 4`
输出： `true`
解释： `n 已经是一个平方数，Alice 可以一次全拿掉 4 个石子并赢得胜利（4 -> 0）。`

示例 4：

```
输入：n = 7
输出：false
解释：当 Bob 采取最优策略时，Alice 无法赢得比赛。
如果 Alice 一开始拿走 4 个石子， Bob 会拿走 1 个石子，然后 Alice 只能拿走 1 个石子，Bob 拿走最后一个石子并赢得胜利（7 -> 3 -> 2 -> 1 -> 0）。
如果 Alice 一开始拿走 1 个石子， Bob 会拿走 4 个石子，然后 Alice 只能拿走 1 个石子，Bob 拿走最后一个石子并赢得胜利（7 -> 6 -> 2 -> 1 -> 0）。
```

示例 5：

```
输入：n = 17
输出：false
解释：如果 Bob 采取最优策略，Alice 无法赢得胜利。
```

提示：

- `1 <= n <= 10^5`

https://leetcode-cn.com/problems/stone-game-iv/solution/shi-zi-you-xi-iv-by-leetcode-solution/

方法一：动态规划

我们用 $f[i]$ 表示先手在面对 i 颗石子时是否处于必胜态（会赢得比赛）。由于先手和后手都采取最优策略，那么 $f[i]$ 为必胜态，当且仅当存在某个 $f[i-k^2]$ 为必败态。也就是说，当先手在面对 i 颗石子时，可以选择取走 k^2 颗，剩余的 $i-k^2$ 颗对于后手来说是必败态，因此先手会获胜。

我们可以写出状态转移方程：

$$f[i] = \begin{cases} \text{true}, & \text{any } f[i-k^2] \text{ is false where } 1 \le k^2 \le i \\ \text{false}, & \text{otherwise} \end{cases}$$

边界条件为 $f[0] = \text{false}$，即没有石子时，先手会输掉游戏。

最终的答案即为 $f[n]$。

复杂度分析

- 时间复杂度：$O(n\sqrt{n})$。对于每一个数 i，k 的枚举上限不超过 $\sqrt{i}$，所以总时间复杂度为 $O(\sum_{i=1}^{n}\sqrt{i}) = O(n\sqrt{n})$。
- 空间复杂度：$O(n)$，即为存储所有状态需要的空间。

```java
class Solution {
    public boolean winnerSquareGame(int n) {
        boolean[] dp = new boolean[n+1];
        for (int i = 1; i <= n; i++){
            for (int k = 1; k * k <= i; ++k) {
                if (!dp[i - k * k]){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[n];
    }
}
```

63、加油站（134）（lintcode187）【贪心】

134. 加油站

难度 中等 363 收藏 分享 切换为英文 关注 反馈

在一条环路上有 *N* 个加油站，其中第 *i* 个加油站有汽油 `gas[i]` 升。

你有一辆油箱容量无限的的汽车，从第 *i* 个加油站开往第 *i+1* 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

- 如果题目有解，该答案即为唯一答案。
- 输入数组均为非空数组，且长度相同。
- 输入数组中的元素均为非负数。

示例 1:

```
输入:
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]

输出: 3

解释:
从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。
```

示例 2:

```
输入:
gas  = [2,3,4]
cost = [3,4,3]

输出: -1

解释:
你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。
我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油
开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油
开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油
你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。
因此，无论怎样，你都不可能绕环路行驶一周。
```

更多硬核内容就在公众号「yes 的练级攻略」

187. 加油站

中文 English

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i]，并且从第_i_个加油站前往第_i_+1个加油站需要消耗汽油 cost[i]。

你有一辆油箱容量无限大的汽车，现在要从某一个加油站出发绕环路一周，一开始油箱为空。

求可环绕环路一周时出发的加油站的编号，若不存在环绕一周的方案，则返回 -1。

样例

样例 1:

输入:gas[i]=[1,1,3,1],cost[i]=[2,2,1,1]
输出:2

样例 2:

输入:gas[i]=[1,1,3,1],cost[i]=[2,2,10,1]
输出:-1

挑战

O(n)时间和O(1)额外空间

注意事项

数据保证答案唯一。

https://www.cnblogs.com/yuzhangcmu/p/4179228.html

SOLUTION 1:

引自Lexi's Leetcode solutions 的解答:

1. 从i开始，j是当前station的指针，sum += gas[j] - cost[j] （从j站加了油，再算上从i开始走到j剩的油，走到j+1站还能剩下多少油）
2. 如果sum < 0，说明从i开始是不行的。那能不能从i..j中间的某个位置开始呢？既然出发到i+1是可行的， 又i~j是不可行的， 从而发现i+1~ j是不可行的。
3. 以此类推i+2~j， i+3~j，i+4~j。。。。等等都是不可行的
4. 所以一旦sum<0，index就赋成j + 1，sum归零。
5. 最后total表示能不能走一圈。

以上做法，其实是贪心的思想:

也就是说brute force的解是： 一个一个来考虑， 每一个绕一圈， 但是 实际上 我们发现 i - j不可行 直接index就跳到j+1， 这样周而复始，很快就是绕一圈 就得到解了。

算法证明:

1. 我们要证明当total > 0的时候，我们一定存在一个点，从它开始，可以绕一圈。

证明以下设定:

我们假定对任意i, j (i: 0 ~ len-1, j: 0 ~ len-1)，我们寻找sum = Σ (gas[k]-cost[k]) (k 从i到j)，使sum 最大。我们从i出发，一定可以绕一圈。

i x x x x j x x x x m x

（1）假定存在这个一个m点，从i点出发，到m点会total（i-m） < 0. 也就是说从i出发不可以走完全程。

=> 因为Total > 0 而 total（i-m）<0，则必然 total ((m+1) ~ (i-1)) > 0 ，那么我们可以得出total ((m+1) ~ j) > total(i-j)，与total(i-j)是最大sum相违背。

=> 根据反证法，上述推定不能成立。所以不会存在这样一个m点，使total（i-m） < 0。也就是说从i出发，我们可以跑完全程。

2. 从1中我们可以看到，只要找到这样一个i-j区间，我们就能使汽车从i点出发，能够跑完全程。按照我们目前使用的算法，只要找到任何一个负区间，我们就丢弃，向下一个区间寻找，这其实就是找最大sum的算法。（因为前面如果是负区间你可以不加他们）最后得到的index实际上就是i （我们并没有计算，实际上再从0 的index往后寻找一些正的station加上即是j的边界），

所以从index出发汽车一定可以跑完一圈。另外，这样的解应该不止一个，在i-j之间，从到n（n在i-j之间）都是可能的解。

实际上只要使total(n~j) >= -total(j+1 ~ i-1)即可， 同样假设假定存在这个一个m点，从n点出发，到m点会total（n-m） < 0. 则因为total(n~j) >= -total(j+1 ~ i-1)，同样可以推出total ((m+1) ~ (i-1)) > 0，与以上证明类似，同样会推翻假设。

例如：total(n~j) = 10， total(j+1 ~ i-1) = -9，如果total（n-m） = -1 ，则total ((m+1) ~ (i-1)) = 2 > 0

3. 总结：从到m只要满足total(n~j) >= -total(j+1 ~ i-1)的点，都是可能的可以出发的点

```
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        if (gas == null || gas.length == 0 || cost == null || cost.length == 0)
 return -1;
        int sum = 0, tol = 0, idx = -1;
        for (int i = 0; i < gas.length; i++){
            sum += gas[i] - cost[i];
            tol += gas[i] - cost[i];
            if (sum < 0){
                idx = i;
                sum = 0;
            }
        }
        return tol < 0 ? -1 : idx + 1;
    }
}
```

64、最低票价（983）

983. 最低票价

难度 中等 278 收藏 分享 切换为英文 关注 反馈

在一个火车旅行很受欢迎的国度，你提前一年计划了一些火车旅行。在接下来的一年里，你要旅行的日子将以一个名为 `days` 的数组给出。每一项是一个从 `1` 到 `365` 的整数。

火车票有三种不同的销售方式：

- 一张为期一天的通行证售价为 `costs[0]` 美元；
- 一张为期七天的通行证售价为 `costs[1]` 美元；
- 一张为期三十天的通行证售价为 `costs[2]` 美元。

通行证允许数天无限制的旅行。 例如，如果我们在第 2 天获得一张为期 7 天的通行证，那么我们可以连着旅行 7 天：第 2 天、第 3 天、第 4 天、第 5 天、第 6 天、第 7 天和第 8 天。

返回你想要完成在给定的列表 `days` 中列出的每一天的旅行所需要的最低消费。

示例 1：

```
输入：days = [1,4,6,7,8,20], costs = [2,7,15]
输出：11
解释：
例如，这里有一种购买通行证的方法，可以让你完成你的旅行计划：
在第 1 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 1 天生效。
在第 3 天，你花了 costs[1] = $7 买了一张为期 7 天的通行证，它将在第 3, 4, ..., 9 天生效。
在第 20 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 20 天生效。
你总共花了 $11，并完成了你计划的每一天旅行。
```

示例 2：

```
输入：days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]
输出：17
解释：
例如，这里有一种购买通行证的方法，可以让你完成你的旅行计划：
在第 1 天，你花了 costs[2] = $15 买了一张为期 30 天的通行证，它将在第 1, 2, ..., 30 天生效。
在第 31 天，你花了 costs[0] = $2 买了一张为期 1 天的通行证，它将在第 31 天生效。
你总共花了 $17，并完成了你计划的每一天旅行。
```

提示：

1. `1 <= days.length <= 365`
2. `1 <= days[i] <= 365`
3. `days` 按顺序严格递增
4. `costs.length == 3`
5. `1 <= costs[i] <= 1000`

https://leetcode-cn.com/problems/minimum-cost-for-tickets/solution/java-dong-tai-gui-hua-si-lu-bu-zou-cong-hou-xiang-/

更多硬核内容就在公众号「yes 的练级攻略」

思路

- 今天不需要出门，不用买票
- 今天如果要出门，需要买几天？
  - 看往后几天（最多 30 天内）要不要出门
    - 30 天内都没有要出行的，那只买今天就好
    - 有要出门的（不同决策）
      - 这次 和 后面几次 分开买更省
      - 这次 和 后面几次 一起买更省

细化思路

上述思路显而易见，最关键在于：「今天买多少，得看后几天怎么安排」，即「前面依赖后面」——从后向前来买。

如图所示，例 `days = [1,4,6,7,8,20]`

1. 第 `21` 及以后的日子都不需要出门，不用买票
2. 第 `20` 需要出门，需要买几天？
   - 不考虑 `20` 之前要不要出门，否则与思路相违背
   - 第 `20` 之后没有出门日，故买「一天」的 `costs[0]` 最省钱

	day/tour	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
																					c[0]	0	0	0	

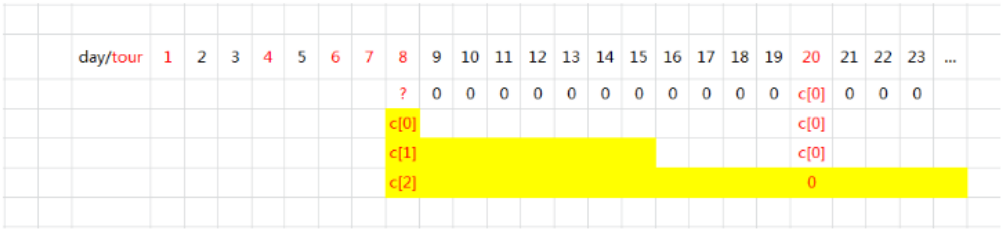
3. 第 `9` - `19` 不需要出门，则不用买

	day/tour	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
									?	0	0	0	0	0	0	0	0	0	0	0	c[0]	0	0	0	

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

4. 第 `8` 需要出门，需要买几天?
   - 往后（只需看往后 30 天）有出门的需求
     - 决策 1：买一天期，后面的不包
     - 决策 2：买七天期，包到第 `8 + 7 - 1` 天，第 `8 + 7` 天往后的不包
     - 决策 3：买三十天期，包到第 `8 + 30 - 1` 天，第 `8 + 30` 天往后的不包
     - 下图展示了三种决策所包含的日期跨度（黄色区域画多了一天...）、所花费用
       - 可见，决策 3 包三十天期的话，第 `20` 可不用花钱



假设题目只有 `[8, 20]` 两天需要出行，则

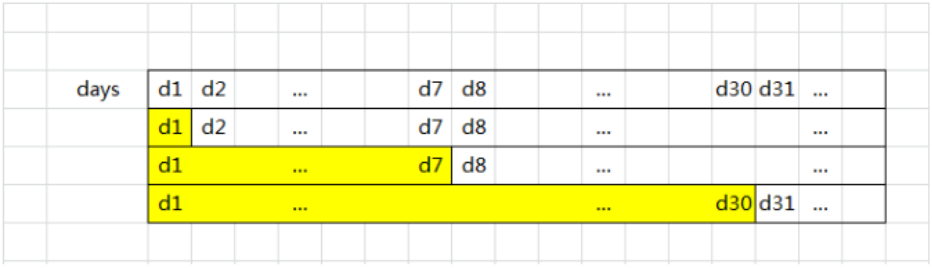
Java

```
int result8 = min(c[0] + c[0], c[1] + c[0], c[2] + 0);
// 第 8 天  = min(决策1,       决策2,       决策3);
```

5. 抽象，定义状态，确定从后向前的递推公式
   - 将上述结果换个说法：「`result` 为第 `8` 天开始，所需最小费用 **累计**」
   - 抽象，定义状态：「`dp[i]` 为第 `i` 天开始，所需最小费用 **累计**」
   - 则

Java

```
dp[i] = min(决策1, 决策2, 决策3);
      = min(c[0] + 1天后不包, c[1] + 7天后不包, c[2] + 30天不包);
      = min(c[0] + dp[i + 1], c[1] + dp[i + 7], c[2] + dp[i + 30]);
```



复杂度分析

- 时间复杂度：$O(maxDay - minDay)$
- 空间复杂度：$O(maxDay)$

再优化

- 当 `d != days[i]` 时，可直接将 `d = days[i]` 以快速跳到前一个需要出发的 `d`
- 若如此，则需将中间跳过的 `dp[d]` 都补上总费用累计

```
class Solution {
    public int mincostTickets(int[] days, int[] costs) {
        int len = days.length, maxDay = days[len - 1], minDay = days[0];
        int[] dp = new int[maxDay + 31]; // 多扩几天，省得判断 365 的限制
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            // 只需看 maxDay -> minDay，此区间外都不需要出门，不会增加费用
            for (int d = maxDay, i = len - 1; d >= minDay; d--) {
                // i 表示 days 的索引
                // 也可提前将所有 days 放入 Set，再通过 set.contains() 判断
                if (d == days[i]) {
                    dp[d] = Math.min(dp[d + 1] + costs[0], dp[d + 7] + costs[1]);
                    dp[d] = Math.min(dp[d], dp[d + 30] + costs[2]);
                    i--; // 别忘了递减一天
                } else dp[d] = dp[d + 1]; // 不需要出门
            }
            return dp[minDay]; // 从后向前遍历，返回最前的 minDay
        }
    }
```

65、区间合并（54）【贪心】

56. 合并区间

难度 中等 524 收藏 分享 切换为英文 关注 反馈

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

```
输入: [[1,3],[2,6],[8,10],[15,18]]
输出: [[1,6],[8,10],[15,18]]
解释: 区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

示例 2:

```
输入: [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

yes的练级攻略

https://leetcode-cn.com/problems/merge-intervals/solution/tan-xin-suan-fa-java-by-liweiwei1419-3/

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Stack;
public class Solution {
    public int[][] merge(int[][] intervals) {
        int len = intervals.length;
        if (len < 2) {
            return intervals;
        }
        // 按照起点排序
        Arrays.sort(intervals, Comparator.comparingInt(o -> o[0]));
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        // 也可以使用 Stack，因为我们只关心结果集的最后一个区间
        List<int[]> res = new ArrayList<>();
        res.add(intervals[0]);

        for (int i = 1; i < len; i++) {
            int[] curInterval = intervals[i];

          // 每次新遍历到的列表与当前结果集中的最后一个区间的末尾端点进行比较
            int[] peek = res.get(res.size() - 1);

            if (curInterval[0] > peek[1]) {
                res.add(curInterval);
            } else {
                // 注意，这里应该取最大
                peek[1] = Math.max(curInterval[1], peek[1]);
            }
        }
        return res.toArray(new int[res.size()][]);
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[][] intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
        int[][] res = solution.merge(intervals);
        for (int i = 0; i < res.length; i++) {
            System.out.println(Arrays.toString(res[i]));
        }
    }
}
```

一、合并 2 个区间

2 个区间的关系有以下 6 种，但是其实可以变成上面 3 种情况（只需要假设 **第一个区间的起始位置 ≤ 第二个区间的起始位置**，如果不满足这个假设，交换这两个区间）。这 3 种情况的合并的逻辑都很好写。

更多硬核内容就在公众号「yes 的练级攻略」

经验：区间类的问题，一般而言是需要画图思考的。因为只有建立直观的感觉，才能更有效的去思考解决问题的方案。

还有需要画图思考的相关算法问题有（其实绝大部分都需要打草稿，大神除外）：

- 和物理现象相关的：第 42 题：接雨水问题、第 11 题：盛最多水的容器、第 218 题：天际线问题；
- 本身问题描述就和图形相关的问题：第 84 题：柱状图中最大的矩形；
- 链表问题：穿针引线如果不画图容易把自己绕晕；
- 回溯算法问题：根据示例画图发现每一步的选择和剪枝的条件；
- 动态规划问题：画示意图发现最优子结构。

得出结论：**可以被合并的区间一定是有交集的区间**，前提是区间按照左端点排好序，这里的交集可以是一个点（例如例 2）。

至于为什么按照左端点升序排序，这里要靠一点直觉猜想，我没有办法说清楚是怎么想到的，有些问题的策略是按照右端点升序排序（也有可能是降序排序，具体问题具体分析）。

接着说，直觉上，只需要对所有的区间**按照左端点升序排序**，然后遍历。

- 如果当前遍历到的区间的左端点 > 结果集中最后一个区间的右端点，说明它们没有交集，此时把区间添加到结果集；
- 如果当前遍历到的区间的左端点 <= 结果集中最后一个区间的右端点，说明它们有交集，此时产生合并操作，即：对结果集中最后一个区间的右端点更新（取两个区间的最大值）。

复杂度分析：

- 时间复杂度：$O(N \log N)$，这里 N 是区间的长度；
- 空间复杂度：$O(N)$，保存结果集需要的空间，这里计算的是最坏情况，也就是所有的区间都没有交点的时候。

说明：`Arrays.sort(intervals, Comparator.comparingInt(o -> o[0]));` 是 Java8 以后提供的一种函数式编程语法，在这里就不展开叙述了。

这里用到的算法思想是：贪心算法。

在具体的算法描述中：

- 前提：区间按照左端点排序；
- 贪心策略：在右端点的选择中，如果产生交集，总是将右端点的数值更新成为最大的，这样就可以合并更多的区间，这种做法是符合题意的。

这道题的证明请见「官方题解」。

这里用到的算法是「贪心算法」，「贪心算法」是在基础算法领域真正很「玄」的算法。很难也很简单。它简单在只要能想到，就不难写出来，且代码一般来说逻辑都比较简单，难在证明的算法的合理性，好在绝大多数情况下不要求证明。

贪心算法就像这个世界上的很多定理和猜想一样，需要人先猜测，然后做实验去验证。贪心这个名字有一定「短视」的意味在里面，就是只注重眼前利益，就能达到全局最优。如果用于形容人的话，有一定讽刺意味，但是在算法领域里面，这种思路的确可以解决特定的一类问题。

贪心算法（Greedy Algorithm）是指：在对问题求解时，总是做出在当前看来是最好的选择。也就是不从整体最优上加以考虑，贪心算法所做出决策是在某种意义上的局部最优解。

更多硬核内容就在公众号「yes 的练级攻略」

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

> 可以适用贪心的问题就是每一步局部最优，最后导致结果全局最优。

重点：贪心策略可以使用的前提是和要解决的问题相关的。不是所有的问题都适合使用贪心算法。而判断一个问题是否可以应用贪心算法，可以从以下两个角度：

- 直觉，根据直觉描述出来的算法，具备「只考虑当前，不考虑全局」的特点，那可能就是「贪心算法」；
- 如果不能举出反例，那多半这个问题就具有「贪心算法性质」，可以使用贪心算法去做。

要严格证明「贪心算法」有效，必须使用数学相关的理论，常见的方法有：

- 数学归纳法；
- 反证法。

贪心算法的证明比较难，并且就算看证明也会给人一头雾水的感觉，就像是让你证明 $\sqrt{2}$ 是无理数一样，但是推翻「贪心算法」很简单。在这里不展开。

经验：由于贪心算法适用的场景一般都是在一组决策里选择最大或者最小值，因此常常在使用贪心算法之前，需要先对数据按照某种规则排序。

一个最简单的理解贪心算法的例子就是「选择排序」，算法描述是：每一轮选择未排定部分里最小的元素交换到未排定部分的开头。

的开头。

> 说明：对于「选择排序」是否是贪心算法，我查过资料，这一点有争议。我个人认为「选择排序」的算法描述符合「局部最优，则整体最优」，即每一步的决策并不考虑全局，只考虑当下，选这个例子的愿意只是因为它足够简单。证明「贪心算法」在「选择排序」上有效需要使用「循环不变量」，在这里不展开。

贪心算法不是对所有问题都能够每一步只看当下，选择最好的策略，就得到整体最优解，关键是贪心策略的选择。选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

具备「无后效性」其实在「动态规划」这一类问题里体现得特别明显，大家可以通过贪心算法的学习在具体去理解「无后效性」的意思。

- 当前决策对后面的决策不产生影响；
- 当前决策只需要记录一个结果，而这个决策是怎么来的不重要。

一旦贪心选择性质不成立，可以考虑的另一种算法思想就是「动态规划」。「动态规划」在每一步做决策的时候，就不只考虑当前步骤的最优解。

贪心算法的应用

- 对数据压缩编码的霍夫曼编码（Huffman Coding）
- 求最小生成树的 Prim 算法和 Kruskal 算法
- 求单源最短路径的Dijkstra算法

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Stack;


public class Solution {

    public int[][] merge(int[][] intervals) {
        int len = intervals.length;
        if (len < 2) {
            return intervals;
```

```
        }

        // 按照起点排序
        Arrays.sort(intervals, Comparator.comparingInt(o -> o[0]));

        // 也可以使用 Stack，因为我们只关心结果集的最后一个区间
        List<int[]> res = new ArrayList<>();
        res.add(intervals[0]);

        for (int i = 1; i < len; i++) {
            int[] curInterval = intervals[i];

            // 每次新遍历到的列表与当前结果集中的最后一个区间的末尾端点进行比
较
            int[] peek = res.get(res.size() - 1);

            if (curInterval[0] > peek[1]) {
                res.add(curInterval);
            } else {
                // 注意，这里应该取最大
                peek[1] = Math.max(curInterval[1], peek[1]);
            }
        }
        return res.toArray(new int[res.size()][]);
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[][] intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
        int[][] res = solution.merge(intervals);
        for (int i = 0; i < res.length; i++) {
            System.out.println(Arrays.toString(res[i]));
        }
    }
}
```

66、视野争夺（腾讯 2020 校园招聘-后台第 5 题）【贪心】

小Q在进行一场竞技游戏,这场游戏的胜负关键就在于能否能争夺一条长度为L的河道,即可以看作是[0,L]的一条数轴。
这款竞技游戏当中有n个可以提供视野的道具-真视守卫,第i个真视守卫能够覆盖区间$[x_i,y_i]$。现在小Q想知道至少用几个真视守卫就可以覆盖整段河道。

输入描述:

输入包括n+1行。

第一行包括两个正整数n和L$(1<=n<=10^5,1<=L<=10^9)$

接下来的n行,每行两个正整数$x_i,y_i$$(0<=x_i<=y_i<=10^9)$,表示第i个真视守卫覆盖的区间。

输出描述:

一个整数，表示最少需要的真视守卫数量，如果无解，输出-1。

输入例子1:

```
4 6
3 6
2 4
0 2
4 7
```

输出例子1:

```
3
```

```java
import java.util.*;
public class Main {
    public static int solutionFive(int n, int L, int[][] arr) {
        Arrays.sort(arr, new Comparator<int[]> (){
            public int compare (int[] o1, int[] o2) {
                return o1[0] == o2[0] ? o1[1]-o2[1]: o1[0]-o2[0];
            }
        });
        int index = 0, count = 0, pre = 0;
        while (pre < L) {
            if (arr[index][0] > pre) {
                break;
            }
            int max = 0;
            while (index < n && arr[index][0] <= pre) {
                max = Math.max(max,  arr[index][1]);
                index++;
            }
            count++;
            pre = max;
            if (pre >= L) {
                return count;
            }
            if (index >= n) {
```

公众号：yes的练级攻略

```
                break;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        //solutionFive
        int n = sc.nextInt();
        int L = sc.nextInt();
        int[][] arr = new int[n][2];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 2; j++) {
                arr[i][j] = sc.nextInt();
            }
        }
        int res = solutionFive(n, L, arr);
        System.out.print(res);
    }

}
```

禁止商用
公众号·

67、种花问题（605）【贪心】

605. 种花问题

难度 简单 154 收藏 分享 切换为英文 关注 反馈

假设你有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花卉不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给定一个花坛（表示为一个数组包含0和1，其中0表示没种植花，1表示种植了花），和一个数 **n** 。能否在不打破种植规则的情况下种入 **n** 朵花？能则返回True，不能则返回False。

示例 1:

```
输入: flowerbed = [1,0,0,0,1], n = 1
输出: True
```

示例 2:

```
输入: flowerbed = [1,0,0,0,1], n = 2
输出: False
```

注意:

1. 数组内已种好的花不会违反种植规则。
2. 输入的数组长度范围为 [1, 20000]。
3. **n** 是非负整数，且不会超过输入数组的大小。

https://leetcode-cn.com/problems/can-place-flowers/solution/chong-hua-wen-ti-by-leetcode/

方法一：贪心

我们从左到右扫描数组 `flowerbed`，如果数组中有一个 `0`，并且这个 `0` 的左右两侧都是 `0`，那么我们就可以在这个位置种花，即将这个位置的 `0` 修改成 `1`，并将计数器 `count` 增加 `1`。对于数组的第一个和最后一个位置，我们只需要考虑一侧是否为 `0`。

在扫描结束之后，我们将 `count` 与 `n` 进行比较。如果 `count` >= `n`，那么返回 `True`，否则返回 `False`。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是数组 `flowerbed` 的长度。
- 空间复杂度：$O(1)$。

方法 1：贪心

```
public class Solution {
    public boolean canPlaceFlowers(int[] flowerbed, int n) {
        int i = 0, count = 0;
        while (i < flowerbed.length) {
            if (flowerbed[i] == 0 && (i == 0 || flowerbed[i - 1] == 0) && (i ==
 flowerbed.length - 1 || flowerbed[i + 1] == 0)) {
                flowerbed[i] = 1;
                count++;
            }
            i++;
        }
        return count >= n;
    }
}
```

方法二：贪心 + 常数优化

我们可以优化方法一中的常数。在扫描数组 `flowerbed` 时，如果 `count` 的值已经达到 `n`，那么我们可以直接跳出循环并返回 `True`。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是数组 `flowerbed` 的长度。
- 空间复杂度：$O(1)$。

方法 2：贪心+常数优化

```
class Solution {
    public static boolean canPlaceFlowers(int[] flowerbed, int n) {
        int cnt = 0;
        for (int i = 0; i < flowerbed.length; i++){
            if (flowerbed[i] == 0 &&
            (i == 0 || flowerbed[i-1] == 0) &&
            (i == flowerbed.length - 1 || flowerbed[i+1] == 0)){
                flowerbed[i++] = 1;
                cnt++;
            }
            if (cnt >= n) return true;
        }
        return false;
```

```
    }
}
```

68、最大平均值和的分组（813）【与二分题中 410 的动规思路类似】

813. 最大平均值和的分组

难度 中等 97 收藏 分享 切换为英文 关注 反馈

我们将给定的数组 A 分成 K 个相邻的非空子数组，我们的分数由每个子数组内的平均值的总和构成。计算我们所能得到的最大分数是多少。

注意我们必须使用 A 数组中的每一个数进行分组，并且分数不一定需要是整数。

示例:
输入:
A = [9,1,2,3,9]
K = 3
输出: 20
解释:
A 的最优分组是[9], [1, 2, 3], [9]. 得到的分数是 9 + (1 + 2 + 3) / 3 + 9 = 20.
我们也可以把 A 分成[9, 1], [2], [3, 9].
这样的分组得到的分数为 5 + 2 + 6 = 13, 但不是最大值.

说明:

- 1 <= A.length <= 100.
- 1 <= A[i] <= 10000.
- 1 <= K <= A.length.
- 答案误差在 10^-6 内被视为是正确的。

https://leetcode-cn.com/problems/largest-sum-of-averages/solution/zui-da-ping-jun-zhi-he-de-fen-zu-by-leetcode/

动态规划:

我们可以使用动态规划来解决这个问题。设 dp(i, k) 表示将数组 A 中的前 i 个元素 A[:i] 分成 k 个相邻的非空子数组，可以得到的最大分数。dp(i, k) 的值可以通过 dp(j, k - 1) 转移而来，其中 j < i，状态转移方程为：

```
dp(i, k) = max(dp(j, k - 1) + average(j + 1, i))
dp(i, 0) = average(0, i)
```

其中 average(j + 1, i) 表示 A[j + 1] 到 A[i] 的平均值 (A[j + 1] + A[j + 2] + ... + A[i]) / (i - j)。我们可以通过预处理出前缀和 P[x + 1] = A[0] + A[1] + ... + A[x]，从而用 average(j + 1, i) = (P[i + 1] - P[j + 1]) / (i - j) 在常数时间内得到平均值。

我们可以继续优化动态规划的空间复杂度。可以发现，如果设 dp(i, k) 为第 k 层的结果，那么第 k 层的结果实际上只和第 k - 1 层有关，因此我们可以使用滚动数组优化空间，即只使用两个一维数组。进一步而言。如果我们从后往前进行动态规划，即设 dp(i, k) 表示数组 A 中从第 i 个元素到结尾 A[i:] 分成 k 个相邻的非空子数组，可以得到的最大分数，那么状态转移方程将变为：

```
dp(i, k) = max(dp(j, k - 1) + average(i, j - 1))
dp(i, 0) = average(i, n - 1)
```

其中 j > i，那么我们在计算第 k 层的结果，并且 i 是依次递增的时候，第 k 层的结果并不会覆盖掉第 k - 1 层的结果，因为当 dp(i, k) 被计算出并且覆盖了 dp(i, k - 1) 时，接下来的所有 dp(i0, k), i0 > i 都不会从 dp(i, k - 1) 转移而来。因此我们最终只需要用一个一维数组，就能完成动态规划。

复杂度分析

- 时间复杂度：$O(K * N^2)$，其中 N 是数组 A 的长度。
- 空间复杂度：$O(N)$。

```
class Solution {
    public double largestSumOfAverages(int[] A, int K) {
```

```
        int N = A.length;
        double[] P = new double[N+1];
        for (int i = 0; i < N; ++i)
            P[i+1] = P[i] + A[i];

        double[] dp = new double[N];
        for (int i = 0; i < N; ++i)
            dp[i] = (P[N] - P[i]) / (N - i);

        for (int k = 0; k < K-1; ++k)
            for (int i = 0; i < N; ++i)
                for (int j = i+1; j < N; ++j)
                    dp[i] = Math.max(dp[i], (P[j]-P[i]) / (j-i) + dp[j]);

        return dp[0];
    }
}
```

69、可被三整除的最大和（1262）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

1262. 可被三整除的最大和

难度 中等 68 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `nums`，请你找出并返回能被三整除的元素最大和。

示例 1：

输入： nums = [3,6,5,1,8]
输出： 18
解释： 选出数字 3, 6, 1 和 8，它们的和是 18（可被 3 整除的最大和）。

示例 2：

输入： nums = [4]
输出： 0
解释： 4 不能被 3 整除，所以无法选出数字，返回 0。

示例 3：

输入： nums = [1,2,3,4,4]
输出： 12
解释： 选出数字 1, 3, 4 以及 4，它们的和是 12（可被 3 整除的最大和）。

提示：

- `1 <= nums.length <= 4 * 10^4`
- `1 <= nums[i] <= 10^4`

https://leetcode-cn.com/problems/greatest-sum-divisible-by-three/solution/java-dong-tai-gui-hua-by-you-gu-meng-nan-3/

Java 动态规划 O(n)

幽谷猛男 发布于 2020-06-04 33 Java 动态规划

解题思路

ans为遍历到第i个元素的最大和能被3整除（余数为0）
要求出ans需要知道前i-1个元素中的 最大和除以3余1--n1 以及 最大和除以3余2--n2 最大和除以3余0的值--ans'
因为当前数字--num 除以3的余数可能会对ans造成影响
如果余1 n1+num 余2 n2+num 余0 ans'+num 余1

如果余2 n1+num 余0 n2+num 余1 ans'+num 余2

如果余0 n1+num 余1 n2+num 余2 ans'+num 余0
每个数字都可能对这三个值造成影响 仔细一点写好代码就行了

```
class Solution {
```

```
    public int maxSumDivThree(int[] nums) {
        int n1 = nums[0] % 3 == 1 ? nums[0] : 0;
        int n2 = nums[0] % 3 == 2 ? nums[0] : 0;
        int ans = nums[0] % 3 == 0 ? nums[0] : 0;
        //System.out.println(ans + ", " + n1 + ", " + n2);
        for (int i = 1; i < nums.length; i++){
            int num = nums[i];
            if (num % 3 == 0) {
                ans += num;
                if (n1 != 0) n1 += num;
                if (n2 != 0) n2 += num;
            }
            else if (num % 3 == 1) {
                int tmp = n2;
                if (n1 != 0) n2 = Math.max(n2,n1+num);
                n1 = Math.max(n1,ans+num);
                if (tmp != 0) ans = Math.max(ans,num+tmp);
            }
            else {
                int tmp = n1;
                if (n2 != 0) n1 = Math.max(n1,n2+num);
                n2 = Math.max(n2,ans+num);
                if (tmp != 0) ans = Math.max(ans,num+tmp);
            }
            //System.out.println(ans + ", " + n1 + ", " + n2);
        }
        return ans;
    }
}
```

70、寻宝（LCP 13）

LCP 13. 寻宝

难度 困难　29　收藏　分享　切换为英文　关注　反馈

我们得到了一副藏宝图，藏宝图显示，在一个迷宫中存在着未被世人发现的宝藏。

迷宫是一个二维矩阵，用一个字符串数组表示。它标识了唯一的入口（用 'S' 表示），和唯一的宝藏地点（用 'T' 表示）。但是，宝藏被一些隐蔽的机关保护了起来。在地图上有若干个机关点（用 'M' 表示），**只有所有机关均被触发，才可以拿到宝藏。**

要保持机关的触发，需要把一个重石放在上面。迷宫中有若干个石堆（用 'O' 表示），每个石堆都有**无限**个足够触发机关的重石。但是由于石头太重，我们一次只能搬**一个**石头到指定地点。

迷宫中同样有一些墙壁（用 '#' 表示），我们不能走入墙壁。剩余的都是可随意通行的点（用 '.' 表示）。石堆、机关、起点和终点（无论是否能拿到宝藏）也是可以通行的。

我们每步可以选择向上/向下/向左/向右移动一格，并且不能移出迷宫。搬起石头和放下石头不算步数。那么，从起点开始，我们最少需要多少步才能最后拿到宝藏呢？如果无法拿到宝藏，返回 -1 。

示例 1：

输入：["S#O", "M..", "M.T"]

输出：16

解释：最优路线为： S->O, cost = 4, 去搬石头 O->第二行的M, cost = 3, M机关触发 第二行的M->O, cost = 3, 我们需要继续回去 O 搬石头。 O->第三行的M, cost = 4, 此时所有机关均触发 第三行的M->T, cost = 2，去T点拿宝藏。 总步数为16。

示例 2：

输入：["S#O", "M.#", "M.T"]

输出：-1

解释：我们无法搬到石头触发机关

示例 3：

输入：["S#O", "M.T", "M.."]

输出：17

解释：注意终点也是可以通行的。

限制：

- `1 <= maze.length <= 100`
- `1 <= maze[i].length <= 100`
- `maze[i].length == maze[j].length`
- S 和 T 有且只有一个
- 0 <= M的数量 <= 16
- 0 <= O的数量 <= 40，题目保证当迷宫中存在 M 时，一定存在至少一个 O 。

https://leetcode-cn.com/problems/xun-bao/solution/xun-bao-bfs-dp-by-leetcode-solution/

方法一：状态压缩动态规划

题意概述

一个人在迷宫中，要从起点 S 走到终点 T。迷宫有两类特殊点，分别是：

- M：机关点，需要用石头触发
- O：石头点，一次可以搬一块石头

只有当所有 M 点均被触发以后，终点才可到达，问起点走到终点的最小代价。

思路与算法

虽然迷宫有很多格子，但是我们实际上的走法只有几种：

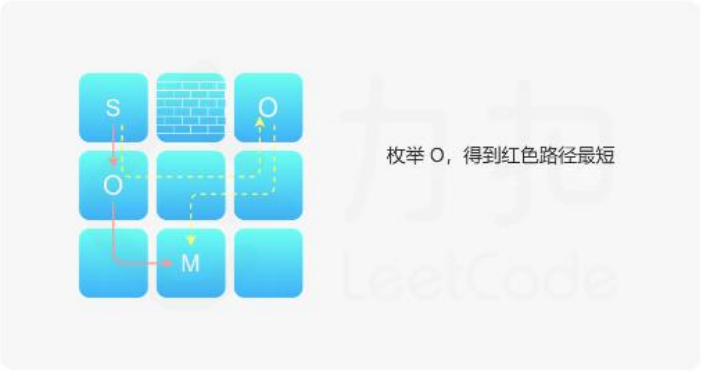
- 从 S 走到 O，我们不会从 S 直接走到 M，因为触发机关要先搬石头
- 从 O 走到 M
- 从 M 走到 O
- 从 M 走到 T

有一点性质很重要，**不论我们触发机关还是搬运石头，都不会改变迷宫的连通状态。因此，两个点的最短距离一旦计算出，就不会再改变了。** 于是第一步，我们可以做一步预处理——我们计算所有特殊点（包括 M，O，S，T）互相之间的最短距离，即对这里面的每个点做一次 BFS。这样我们就不需要考虑其他点了。为什么要预处理出这些特殊点两两之间的距离，这个问题会在在下文中解释。

解决这个问题的关键是理解我们要以什么样的策略来取石头和触发机关：

- 在最开始，我们一定会从 S，经过某一个 O，到达某一个 M。那么对于特定的 M 来说，我们枚举 O 就可以计算 $S-O-M$ 的最短距离。那么如果我们要从起点 S 到达 M，一定会选择这条距离最短的路。这样，我们首先得到了 S 到每一个 M 的最短距离。

更多硬核内容就在公众号「yes 的练级攻略」



- 假定我们已经从起点到达了某个 M 了，接下来需要去其他的 O 点搬石头接着触发其他的机关，这是一个 $M-O-M'$ 的路线。同样的道理，对于给定的 M'，中间的 O 也是固定的。即给定 M 和 M'，我们可以确定一个 O，使得 $M-O-M'$ 距离最短。我们同样可以记录下这个最短距离，即得到了 所有 M 到 M' 的最短距离。

- 最后，所有 M 到 T 的距离在前面已经计算出了。

我们需要所有的 M 都被触发，M 的触发顺序不同会导致行走的路径长度不同。假设这里一共有 n 个 M，我们用 $d(i,j)$ 表示第 i 个 M 到第 j 个 M 经过某一个 O 的最短距离。因为这里的 n 不大于 16，我们可以使用一个 16 位的二进制数表示状态，这个二进制数的第 i 位为 1 表示第 i 个 M 已经触发，为 0 表示第 i 个 M 还未被触发，记这个二进制数为 mask 。记 M_i 为第 i 个 M（下标从 1 开始），每一个 mask 都可以表示成两个集合，一个已触发集合、一个未触发集合，例如 $n=16$，$mask=$ 0000 1100 0001 0001 的已触发集合就可以表示为 $T=\{M_1,M_5,M_{11},M_{12}\}$，剩下的元素都在未触发集合 $U-T$ 中。

我们定义 $f(\text{mask},i)$ 表示当前在第 i 个 M 处，触发状态为 mask 的最小步数，如果当前 mask 代表的已触发集合为 T ，未触发集合为 $U-T$，则我们可以推出这样的动态规划转移方程：

$$f(\text{mask},i)=\min_{j\in T,j\neq i}\{f(\text{mask xor } 2^i,j)+d(j,i)\}$$

其中 mask xor 2^i 表示把 M_i 已触发的集合当中去掉，即 mask 这个状态可以由 mask xor 2^i 状态转移得到，转移时我们除了关注触发状态 mask 的变化，我们还关注是从哪一个 M 转移到了 M_i，我们可以枚举 mask 当中已触发的所有的 $M_j(j\neq i)$ 作为上一个位置，而 $d(j,i)$ 就是我们从 j 转移到 i 行走的最短步数，我们可以在预处理之后按照我们的策略得到所有的 $d(j,i)$（如果 i,j 不可达可以设为正无穷），然后 $O(1)$ 查询，这就是预处理的目的。

实际上，在实现的时候，如果我们用记忆化搜索的方式实现，那么我们用到的是上面的转移方程；如果我们使用循环实现的话，也可以使用下面的转移方程，写法类似递推：

$$f(\text{mask}|2^j,j)=\min\{f(\text{mask},i)+d(i,j)\}$$

大家可以结合代码来理解。当然，写代码的时候需要注意的是：

- 由于本题的复杂度较高，使用 Python 等性能较差的语言实现时需要注意效率问题。
- 本题边界情况较多，比如迷宫没有 M、M 不可达等。

更多硬核内容就在公众号「yes 的练级攻略」

题型小结

这道题是一个非常经典的状态压缩动态规划模型：有 n 个任务 $\{M_1, M_2 \cdots M_n\}$，每两个任务之间有一个 $c(M_i, M_j)$ 表示在 M_i 之后（下一个）做 M_j 的花费，让你求解把 n 个任务都做完需要的最小花费。通常这个 n 会非常的小，因为需要构造 2^n 种状态，$c(M_i, M_j)$ 可能是题目给出，也可能是可以在很短的时间内计算出来的一个值。这类问题的状态设计一般都是 $f(\text{mask}, i)$ 表示当前任务完成的状态是 mask，当前位置是 i，考虑转移的时候我们只需要考虑当前任务的上一个任务即可。**希望读者可以理解这里的思想，并尝试使用记忆化搜索和循环两种方式实现。**

LCP13 具体代码

```
class Solution {
    int[] dx = {1, -1, 0, 0};
    int[] dy = {0, 0, 1, -1};
    int n, m;

    public int minimalSteps(String[] maze) {
        n = maze.length;
        m = maze[0].length();
        // 机关 & 石头
        List<int[]> buttons = new ArrayList<int[]>();
        List<int[]> stones = new ArrayList<int[]>();
        // 起点 & 终点
        int sx = -1, sy = -1, tx = -1, ty = -1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (maze[i].charAt(j) == 'M') {
                    buttons.add(new int[]{i, j});
                }
                if (maze[i].charAt(j) == 'O') {
                    stones.add(new int[]{i, j});
                }
                if (maze[i].charAt(j) == 'S') {
                    sx = i;
                    sy = j;
                }
                if (maze[i].charAt(j) == 'T') {
                    tx = i;
                    ty = j;
                }
            }
        }
        int nb = buttons.size();
        int ns = stones.size();
        int[][] startDist = bfs(sx, sy, maze);

        // 边界情况：没有机关
        if (nb == 0) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            return startDist[tx][ty];
        }
        // 从某个机关到其他机关 / 起点与终点的最短距离。
        int[][] dist = new int[nb][nb + 2];
        for (int i = 0; i < nb; i++) {
            Arrays.fill(dist[i], -1);
        }
        // 中间结果
        int[][][] dd = new int[nb][][];
        for (int i = 0; i < nb; i++) {
            int[][] d = bfs(buttons.get(i)[0], buttons.get(i)[1], maze)
;
            dd[i] = d;
            // 从某个点到终点不需要拿石头
            dist[i][nb + 1] = d[tx][ty];
        }

        for (int i = 0; i < nb; i++) {
            int tmp = -1;
            for (int k = 0; k < ns; k++) {
                int midX = stones.get(k)[0], midY = stones.get(k)[1];
                if (dd[i][midX][midY] != -1 && startDist[midX][midY] !=
 -1) {
                    if (tmp == -1 || tmp > dd[i][midX][midY] + startDis
t[midX][midY]) {
                        tmp = dd[i][midX][midY] + startDist[midX][midY]
;
                    }
                }
            }
            dist[i][nb] = tmp;
            for (int j = i + 1; j < nb; j++) {
                int mn = -1;
                for (int k = 0; k < ns; k++) {
                    int midX = stones.get(k)[0], midY = stones.get(k)[1
];
                    if (dd[i][midX][midY] != -1 && dd[j][midX][midY] !=
 -1) {
                        if (mn == -1 || mn > dd[i][midX][midY] + dd[j][
midX][midY]) {
                            mn = dd[i][midX][midY] + dd[j][midX][midY];
                        }
                    }
                }
```

```
                dist[i][j] = mn;
                dist[j][i] = mn;
            }
        }

        // 无法达成的情形
        for (int i = 0; i < nb; i++) {
            if (dist[i][nb] == -1 || dist[i][nb + 1] == -1) {
                return -1;
            }
        }

        // dp 数组，  -1 代表没有遍历到
        int[][] dp = new int[1 << nb][nb];
        for (int i = 0; i < 1 << nb; i++) {
            Arrays.fill(dp[i], -1);
        }
        for (int i = 0; i < nb; i++) {
            dp[1 << i][i] = dist[i][nb];
        }

        // 由于更新的状态都比未更新的大，所以直接从小到大遍历即可
        for (int mask = 1; mask < (1 << nb); mask++) {
            for (int i = 0; i < nb; i++) {
                // 当前 dp 是合法的
                if ((mask & (1 << i)) != 0) {
                    for (int j = 0; j < nb; j++) {
                        // j 不在 mask 里
                        if ((mask & (1 << j)) == 0) {
                            int next = mask | (1 << j);
                            if (dp[next][j] == -1 || dp[next][j] > dp[m
ask][i] + dist[i][j]) {
                                dp[next][j] = dp[mask][i] + dist[i][j];
                            }
                        }
                    }
                }
            }
        }

        int ret = -1;
        int finalMask = (1 << nb) - 1;
        for (int i = 0; i < nb; i++) {
```

```
            if (ret == -1 || ret > dp[finalMask][i] + dist[i][nb + 1])
{
                ret = dp[finalMask][i] + dist[i][nb + 1];
            }
        }

        return ret;
    }

    public int[][] bfs(int x, int y, String[] maze) {
        int[][] ret = new int[n][m];
        for (int i = 0; i < n; i++) {
            Arrays.fill(ret[i], -1);
        }
        ret[x][y] = 0;
        Queue<int[]> queue = new LinkedList<int[]>();
        queue.offer(new int[]{x, y});
        while (!queue.isEmpty()) {
            int[] p = queue.poll();
            int curx = p[0], cury = p[1];
            for (int k = 0; k < 4; k++) {
                int nx = curx + dx[k], ny = cury + dy[k];
                if (inBound(nx, ny) && maze[nx].charAt(ny) != '#' && re
t[nx][ny] == -1) {
                    ret[nx][ny] = ret[curx][cury] + 1;
                    queue.offer(new int[]{nx, ny});
                }
            }
        }
        return ret;
    }

    public boolean inBound(int x, int y) {
        return x >= 0 && x < n && y >= 0 && y < m;
    }
}
```

71、比特位计数（338）

338. 比特位计数

难度 中等 363 收藏 分享 切换为英文 关注 反馈

给定一个非负整数 **num**。对于 **0 ≤ i ≤ num** 范围中的每个数字 **i** ，计算其二进制数中的 1 的数目并将它们作为数组返回。

示例 1:

```
输入: 2
输出: [0,1,1]
```

示例 2:

```
输入: 5
输出: [0,1,1,2,1,2]
```

进阶:

- 给出时间复杂度为**O(n*sizeof(integer))**的解答非常容易。但你可以在线性时间**O(n)**内用一趟扫描做到吗？
- 要求算法的空间复杂度为**O(n)**。
- 你能进一步完善解法吗？要求在C++或任何其他语言中不使用任何内置函数（如 C++ 中的 **__builtin_popcount**）来执行此操作。

https://leetcode-cn.com/problems/counting-bits/solution/bi-te-wei-ji-shu-by-leetcode/

方法二：动态规划 + 最高有效位 【通过】

直觉

利用已有的计数结果来生成新的计数结果。

算法

假设有一个整数：

$$x = (1001011101)_2 = (605)_{10}$$

我们已经计算了从 0 到 $x - 1$ 的全部结果。

我们知道，x 与 我们计算过的一个数只有一位之差：

$$x' = (1011101)_2 = (93)_{10}$$

它们只在最高有效位上不同。

让我们以二进制形式检查 $[0, 3]$ 的范围：

$$(0) = (0)_2$$

$$(1) = (1)_2$$

$$(2) = (10)_2$$

$$(3) = (11)_2$$

可以看出， 2 和 3 的二进制形式可以通过给 0 和 1 的二进制形式在前面加上 1 来得到。因此，它们的 pop count 只相差 1。

类似的，我们可以使用 $[0, 3]$ 作为蓝本来得到 $[4, 7]$。

总之，对于pop count $P(x)$，我们有以下的状态转移函数：

$$P(x + b) = P(x) + 1, b = 2^m > x$$

有了状态转移函数，我们可以利用动态规划从 0 开始生成所有结果。

复杂度分析

- 时间复杂度：$O(n)$。对每个整数 x，我们只需要常数时间。
- 空间复杂度：$O(n)$。我们需要 $O(n)$ 的空间来存储技术结果。如果排除这一点，就只需要常数空间。

方法 1：动规+最高有效位

```
public class Solution {
```

```
    public int[] countBits(int num) {
        int[] ans = new int[num + 1];
        int i = 0, b = 1;
        // [0, b) is calculated
        while (b <= num) {
            // generate [b, 2b) or [b, num) from [0, b)
            while(i < b && i + b <= num){
                ans[i + b] = ans[i] + 1;
                ++i;
            }
            i = 0;   // reset i
            b <<= 1; // b = 2b
        }
        return ans;
    }
}
```

方法三：动态规划 + 最低有效位 【通过】

直觉

只要x' 小于 x，且它们的 pop count 之间存在函数关系，就可以写出其他的状态转移函数。

算法

遵循上一方法的相同原则，我们还可以通过最低有效位来获得状态转移函数。

观察x 和 $x' = x/2$ 的关系：

$$x = (1001011101)_2 = (605)_{10}$$

$$x' = (100101110)_2 = (302)_{10}$$

可以发现 x' 与 x 只有一位不同，这是因为x' 可以看做 x 移除最低有效位的结果。

这样，我们就有了下面的状态转移函数：

$$P(x) = P(x/2) + (x \mod 2)$$

复杂度分析

- 时间复杂度：$O(n)$。对每个整数 x，我们只需要常数时间。
- 空间复杂度：$O(n)$。与方法二相同。

方法 2：动规+最低有效位【推荐】

```
class Solution {
    public int[] countBits(int num) {
        int[] dp = new int[num + 1];
        for (int i = 1; i <= num; ++i){
            dp[i] = dp[i >> 1] + (i & 1); // x / 2 is x >> 1 and x % 2
is (x & 1)
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }
        return dp;
    }
}
```

方法四：动态规划 + 最后设置位【通过】

算法

与上述方法思路相同，我们可以利用最后设置位。

最后设置位是从右到左第一个为1的位。使用 `x &= x - 1` 将该位设置为0，就可以得到以下状态转移函数：

$$P(x) = P(x \ \& \ (x-1)) + 1;$$

复杂度分析

- 时间复杂度：$O(n)$。与方法三相同。
- 空间复杂度：$$(n)$。与方法三相同。

方法 3：动规+最后设置位

```
class Solution {
    public int[] countBits(int num) {
        int[] dp = new int[num + 1];
        for (int i = 1; i <= num; ++i){
            dp[i] = dp[i & (i - 1)] + 1;
        }
        return dp;
    }
}
```

72、根据身高重建队列（406）【贪心】

406. 根据身高重建队列

难度 中等 415 收藏 分享 切换为英文 关注 反馈

假设有打乱顺序的一群人站成一个队列。每个人由一个整数对 (h, k) 表示，其中 h 是这个人的身高，k 是排在这个人前面且身高大于或等于 h 的人数。编写一个算法来重建这个队列。

注意：
总人数少于1100人。

示例

输入：
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

输出：
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

解题思路

首先解释一下题目：
给定一群人的一个队列，要求重新排序。队列中每个人由一个整数对(h, k)表示，其中h是这个人的身高，k表示在**新队列**中要求排在这个人前面且身高大于或等于h的人数。*注：队列排序即数值排序*

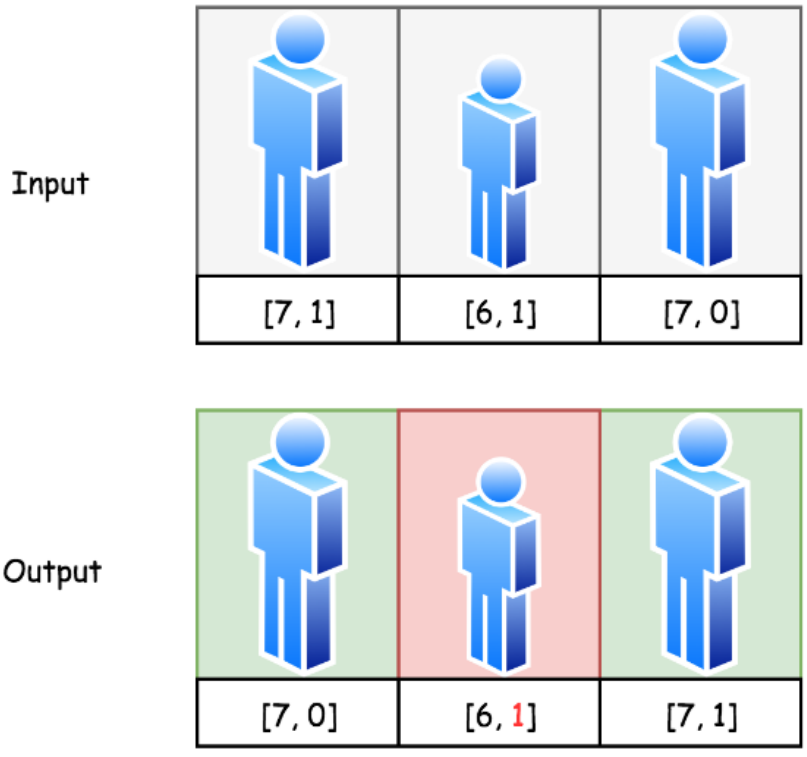
更通俗一点就是：假设有一个队列，已知每个人的身高h和排在该人前面的更高或一样高的人数k，现在给你打乱顺序后的数组，要求恢复原来的队列。

考虑将这群人依次加入新队列中，加入时需符合k的要求。注意到，一个人的k值实际上与身高更矮的人的位置无关，所以如果身高比之更高的人已经排好队了，这个人加入这个新队列的位置就可以根据k值确定了。因此，身高较高的人应该先加入，我们先对队列按身高降序排序。此外，对于相同身高的人，k值较小的人位置在前，优先加入。

链接：
https://leetcode-cn.com/problems/queue-reconstruction-by-height/solution/gen-ju-shen-gao-zhong-jian-dui-lie-by-leetcode/

让我们从最简单的情况下思考，当队列中所有人的 (h,k) 都是相同的高度 h，只有 k 不同时，解决方案很简单：每个人在队列的索引 index = k。

即使不是所有人都是同一高度，这个策略也是可行的。因为个子矮的人相对于个子高的人是“看不见”的，所以可以先安排个子高的人。



上图中我们先安排身高为 `7` 的人，将它放置在与 `k` 值相等的索引上；再安排身高为 `6` 的人，同样的将它放置在与 `k` 值相等的索引上。

该策略可以递归进行：

- 将最高的人按照 `k` 值升序排序，然后将它们放置到输出队列中与 `k` 值相等的索引位置上。
- 按降序取下一个高度，同样按 `k` 值对该身高的人升序排序，然后逐个插入到输出队列中与 `k` 值相等的索引位置上。
- 直到完成为止。

算法：

- 排序：
  - 按高度降序排列。
  - 在同一高度的人中，按 `k` 值的升序排列。
- 逐个地把它们放在输出队列中，索引等于它们的 `k` 值。
- 返回输出队列

复杂度分析

- 时间复杂度：$\mathcal{O}(N^2)$。排序使用了 $\mathcal{O}(N \log N)$ 的时间，每个人插入到输出队列中需要 $\mathcal{O}(k)$ 的时间，其中 k 是当前输出队列的元素个数。总共的时间复杂度为 $\mathcal{O}\left(\sum_{k=0}^{N-1} k\right) = \mathcal{O}(N^2)$。
- 空间复杂度：$\mathcal{O}(N)$，输出队列使用的空间。

406 具体代码

```
class Solution {
    public int[][] reconstructQueue(int[][] people) {
        Arrays.sort(people, (o1, o2) -> o1[0] == o2[0] ? o1[1] - o2[1]
: o2[0] - o1[0]);
        List<int[]> res = new LinkedList<>();
        for (int[] p : people){
            res.add(p[1], p);
        }
        return res.toArray(new int[res.size()][2]);
    }
}
```

73、我能赢吗（464）【状态压缩 DP】

464. 我能赢吗

难度 中等 142 收藏 分享 切换为英文 关注 反馈

在 "100 game" 这个游戏中，两名玩家轮流选择从 1 到 10 的任意整数，累计整数和，先使得累计整数和达到 100 的玩家，即为胜者。

如果我们将游戏规则改为 “玩家不能重复使用整数” 呢？

例如，两个玩家可以轮流从公共整数池中抽取从 1 到 15 的整数（不放回），直到累计整数和 >= 100。

给定一个整数 `maxChoosableInteger` （整数池中可选择的最大数）和另一个整数 `desiredTotal` （累计和），判断先出手的玩家是否能稳赢（假设两位玩家游戏时都表现最佳）？

你可以假设 `maxChoosableInteger` 不会大于 20， `desiredTotal` 不会大于 300。

示例：

输入：
maxChoosableInteger = 10
desiredTotal = 11

输出：
false

解释：
无论第一个玩家选择哪个整数，他都会失败。
第一个玩家可以选择从 1 到 10 的整数。
如果第一个玩家选择 1，那么第二个玩家只能选择从 2 到 10 的整数。
第二个玩家可以通过选择整数 10（那么累积和为 11 >= desiredTotal），从而取得胜利.
同样地，第一个玩家选择任意其他整数，第二个玩家都会赢。

https://leetcode-cn.com/problems/can-i-win/solution/hui-su-zhuang-ya-dp-by-8bun/

先读懂题意：

"稳赢"指的是，假设有两个人A, B，A先抽整数然后轮到B，再轮到A...，每次都根据要凑的和选择**最有利于自己赢的数**（并不一定越大越好，所以说是一种博弈），如果存在A走一步之后, B无论怎么走能还是A能赢的情况，那么就说A能稳赢（而不是所有的情况下）。
例如maxChoosableInteger = 10,desiredTotal=3 那么就存在A抽3，然后B抽其他数都赢不了的情况，因为A已经赢了（可以理解为B就不用再抽了）,所以A可以稳赢。
又如：maxChoosableInteger = 10，desiredTotal = 11 那么A第一轮选1(此时选得越小就越有利于自己赢),那么第二个玩家可以选择10（此时选择11-1就有利于自己赢）来赢得胜利，那么A就不能稳赢了。
下面提供两种解法（解决的思路一致，见注释）

```
/**
 * @Description:
 * 由于状态不可用数组进行传递【在递归当中会受到改变，不能准确定位当前状态】，故在此处用 Int 的位表示状态（1 表示用过,0 表示未用过）
 * 这里采用 DP 状态压缩的方式，思想与回溯类似，只是这里的状态被压缩成了一个 bitArray 了
 * 状态压缩，我们可以用二进制的第 i 位的 0 或者 1 来表示 i 这个数字的选取与否，这样所有数字的选取状态就可以用一个数来很方便的表示，
 * 题目说了不超过 20 位，所以这里就可以用一个 int 来表示状态 state，通过 state 来判断状态是否一致，进而进行记忆化的存取
 */
public class Solution {

    public boolean canIWin(int maxChoosableInteger, int desiredTotal) {

        if (maxChoosableInteger >= desiredTotal) return true;
        if ((1 + maxChoosableInteger) * maxChoosableInteger / 2 < desiredTotal) return false;
        /**
         *  dp 表示"每个"取数(A 和 B 共同作用的结果)状态下的输赢
         *  例如只有 1,2 两个数选择,那么 (1 << 2) - 1 = 4 - 1 = 3 种状态表示:
         *  01,10,11 分别表示：A 和 B 已经选了 1，已经选了 2，已经选了 1、2 状态下，A 的输赢情况
         *  并且可见这个表示所有状态的 dp 数组的每个状态元素的长度为 maxChoosableInteger 位的二进制数
         */
        Boolean[] dp = new Boolean[(1 << maxChoosableInteger) - 1];
        return dfs(maxChoosableInteger, desiredTotal, 0, dp);
    }

    /**
     * @param maxChoosableInteger 选择的数的范围 [1,2,...maxChoosableInteger]
     * @param desiredTotal 目标和
     * @param state 当前状态的十进制表示（记录着可能不止一个数被选择的状态）
     * @param dp 记录所有状态
```

```
     * @return
     */
    private boolean dfs(int maxChoosableInteger, int desiredTotal, int
state, Boolean[] dp) {
        if (dp[state] != null)
            return dp[state];
        /**
         * 例如maxChoosableInteger=2，选择的数只有1,2 两个，二进制只要两位就
可以表示他们的选择状态
         * 最大位为2（第2 位），也就是1 << (2 - 1)的结果，所以最大的位可以表
示为  1 << (maxChoosableInteger - 1)
         * 最小的位可以表示为 1 << (1 - 1)，也就是1（第1 位）
         * 这里i 表示括号的范围
         */
        for (int i = 1; i <= maxChoosableInteger; i++){
            //当前待抉择的位，这里的tmp 十进制只有一位为1，用来判断其为1 的位，
对于state 是否也是在该位上为1
            //用以表示该位（数字）是否被使用过
            /**
             * (&运算规则，都1 才为1)
             * 例如,i=3, tmp = 4, state = 3;
             *  100
             * &011
             * =0   表示该位没有被使用过，也就是第三位没有被使用过，即数字3 (i)
没有被使用过
             */
            int tmp = (1 << (i - 1));
            if ((tmp & state) == 0){  //该位没有被使用过
                //如果当前选了i 已经赢了或者选了i 还没赢但是后面对方选择输
了,tmp|state 表示进行状态的更新
                /**
                 * 例如
                 *  100
                 * |011
                 * =111
                 */
                //注意这里并没有像回溯一样进行状态的(赋值化的)更新、回溯
                //其实这里是隐含了回溯的，我们通过参数传递更新后的state
                //但是我们在这个调用者这里的state 还是没有进行更新的，所以
                //就相当于回溯了状态。
                if (desiredTotal - i <= 0 || !dfs(maxChoosableInteger,
desiredTotal - i, tmp|state, dp)) {
                    dp[state] = true;
                    return true;
```

```
                }
            }
        }
        //如果都赢不了
        dp[state] = false;
        return false;
    }
}
```

74、优美的排列（526）【状态压缩 DP】

526. 优美的排列

难度 中等 86 收藏 分享 切换为英文 关注 反馈

假设有从 1 到 N 的 N 个整数，如果从这 N 个数字中成功构造出一个数组，使得数组的第 i 位 (1 <= i <= N) 满足如下两个条件中的一个，我们就称这个数组为一个优美的排列。条件：

1. 第 i 位的数字能被 i 整除
2. i 能被第 i 位上的数字整除

现在给定一个整数 N，请问可以构造多少个优美的排列？

示例1:

输入: 2
输出: 2
解释:

第 1 个优美的排列是 [1, 2]:
  第 1 个位置（i=1）上的数字是1，1能被 i（i=1）整除
  第 2 个位置（i=2）上的数字是2，2能被 i（i=2）整除

第 2 个优美的排列是 [2, 1]:
  第 1 个位置（i=1）上的数字是2，2能被 i（i=1）整除
  第 2 个位置（i=2）上的数字是1，i（i=2）能被 1 整除

说明:

1. N 是一个正整数，并且不会超过15。

https://www.bilibili.com/video/BV1Lb411w7X4

```
class Solution {
    public int countArrangement(int N) {
        // dp[state]表示：如state = 11，也就是1011，代表当前使用了第一个数、第二个数、第四个数的情况下，一共有多少个不同的排列
        int[] dp = new int[1 << N];
        dp[0] = 1;
        for(int i = 0; i < (1 << N); i++){
            // 计算已经放的数字个数
            int s = 1;
            for(int j = 0; j < N; j++){
                s += i >> j & 1;// i >> j & 1 表示:i 的第 j 位是否是 1
            }
            // 放下一个数
            for(int j = 1; j <= N; j++){
                // 没有被选过（对应位置是 0），并且满足互为倍数
                if( (i >> (j-1) & 1) == 0 && (s % j == 0 || j % s == 0)
){
                    // 把放 j 的方案加上
                    // dp[010111] += dp[010101]
```

```
                    dp[i | (1 << j - 1)] += dp[i];
                }
            }
        }
        return dp[(1 << N) - 1];
    }
}
```

75、参加考试的最大学生数（1349）【状态压缩 DP】

1349. 参加考试的最大学生数

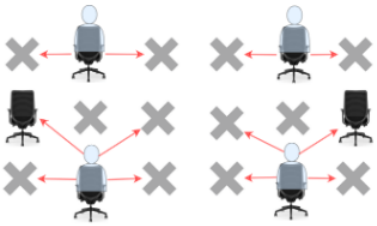
难度 困难 80 收藏 分享 切换为英文 关注 反馈

给你一个 `m * n` 的矩阵 `seats` 表示教室中的座位分布。如果座位是坏的（不可用），就用 `'#'` 表示；否则，用 `'.'` 表示。

学生可以看到左侧、右侧、左上、右上这四个方向上紧邻他的学生的答卷，但是看不到直接坐在他前面或者后面的学生的答卷。请你计算并返回该考场可以容纳的一起参加考试且无法作弊的最大学生人数。

学生必须坐在状况良好的座位上。

示例 1：



```
输入：seats = [["#",".","#","#",".","#"],
               [".","#","#","#","#","."],
               ["#",".","#","#",".","#"]]
输出：4
解释：教师可以让 4 个学生坐在可用的座位上，这样他们就无法在考试中作弊。
```

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

示例 2：

```
输入：seats = [[".","#"],
              ["#","#"],
              ["#","."],
              ["#","#"],
              [".","#"]]
输出：3
解释：让所有学生坐在可用的座位上。
```

示例 3：

```
输入：seats = [["#",".",".",".","#"],
              [".","#",".","#","."],
              [".",".","#",".","."],
              [".","#",".","#","."],
              ["#",".",".",".","#"]]
输出：10
解释：让学生坐在第 1、3 和 5 列的可用座位上。
```

提示：

- `seats` 只包含字符 `'.'` 和 `'#'`
- `m == seats.length`
- `n == seats[i].length`
- `1 <= m <= 8`
- `1 <= n <= 8`

https://leetcode-cn.com/problems/maximum-students-taking-exam/solution/can-jia-kao-shi-de-zui-da-xue-sheng-shu-ya-suo-zhu/

压缩状态DP，是将某一状态进行编码。例如一排座位有4个凳子，从右至左分别标号为0 —— 3.假设凳子有人坐表示1，没人坐表示0。那么其中一种状态是我自己坐在0号座位上(0001)，另一种状态是我坐在0号座位上，我女朋友坐在2号座位上(0101)。还有一种状态是我坐在0号座位上，我的三个前女友分别坐在123号座位上(1111)。那么可知，一共可以有16种状态。而这16种状态是由二进制表示可一一对应一个十进制的数字，那么就可以用这个十进制的数字来代表某一种状态。例如数字5表示0号和2号凳子有人坐(0101)。

回到这一题，要确定整个教室参加考试的最大人数，可将问题想成教室m横排的最大考试人数，那么前m - 1横排的最大人数加上第m排满足条件的最大人数就得到整个教室所能参加的最大人数。可是每一横排学生的排列不相同，所以我们将要遍历这一排学生可能坐的所有情况。假设这一横排有n个座位，那么就要遍历2^n次。于是我们定义数组dp[i][j] 表示第i + 1排(数组从0开始计数)的学生排列状态为十进制数j的情况下，前i排能够做最大学生人数加上j状态下对应的学生人数。(状态为十进制数j意思就是上面那个女朋友的例子)。

得到了dp数组的定义，就要看如何递推了。第i + 1排状态为j的dp[i][j]肯定和前一排有关系。如何得到dp[i][j]的最大值，就要将前一排的所有状态遍历一遍，将本排满足左上角右上角没人的前一排的状态都挑选出来，再选出一个最大值得到第i + 1排状态为j的dp[i][j]的值。

```
for(int k = 0; k < stateSize; k++){
    if(左上角右上角没有学生){
        dp[i][j] = Math.max(dp[i - 1][k] + Integer.bitCount(j) , dp[i][j]);
    }
}
```

解决了DP数组定义和状态转移方程，接下来就是具体实现细节了。
Q1：如何计算状态j情况下的学生人头数？
A：利用java函数Integer.bitCount(j),j的二进制有多少1，就返回具体个数，即学生人头数。

Q2：如何计算状态j情况下左右没有人？
A：利用位运算，将j左移一位和j进行按位与如果为0，则说明二进制的j中没有相邻的二进制1，则说明左右无人。即：`(j & (j >> 1)) == 0`。

Q3：如何计算状态j情况下左上方右上方无人？
A：将前一排的状态k右移一位和j按位与，如果等于0，则说明状态k二进制1不会存在于状态j的左上方。
将前一排的状态k左移一位和j按位与，如果等于0，则说明状态k二进制1不会存在于状态j的右上方。
`j & (k >> 1)) == 0 && ((j >> 1) & k) == 0`

Q4：如何表示好的座椅？题中桌椅的好坏是用字符表示的。
A：设置一个表示座椅是否可用的数组validity[m]，将每一排的可用桌椅表示状态并压缩。类似前面女朋友的例子。好的桌椅表示1，坏的表示0.那么当遍历不同学生状态j时，只需要将学生状态j和桌椅状态按位相遇即可，若等于j。表示当钱学生状态可坐进这一排的座位中。
`j & validity[i] == j`

```java
class Solution {
    public int maxStudents(char[][] seats) {
        int m = seats.length;
        int n = seats[0].length;
        int[] validity = new int[m]; // validity 数组用于记录每一横排位置是否能做。
        int stateSize = 1 << n; // 每一横排可由学生排布的方式有 2^n 种。
        int[][] dp = new int[m][stateSize];
        int ans = 0;
        // 初始化 validity 数组
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                validity[i] = (validity[i] << 1) + (seats[i][j] == '.' ? 1 : 0);
            }
        }
        // 初始化 dp 数组
        for(int i = 0; i < m; i++){
            for(int j = 0; j < stateSize; j++){
                dp[i][j] = -1;
            }
        }
        for(int i = 0; i < m; i++){
            for(int j = 0; j < stateSize; j++){
                // j & validity[i] == j 判断 j 的状态下能否坐下第 i 横排
```

```
                // (j & (j >> 1) == 0) 判断 j 模式左右是否没人
                if(((j & validity[i]) == j) && ((j & (j >> 1)) == 0)){
                    if(i == 0){ // 第一横排
                        dp[i][j] = Integer.bitCount(j);
                    }else{
                        // 不是第一排，就要遍历前一排，从而取得当前排的最大值。
                        for(int k = 0; k < stateSize; k++){
                            if((j & (k >> 1)) == 0 && ((j >> 1) & k) ==
 0 && (dp[i - 1][k] != -1)){
                                dp[i][j] = Math.max(dp[i - 1][k] + Inte
ger.bitCount(j) , dp[i][j]);
                            }
                        }
                    }
                    ans = Math.max(ans , dp[i][j]);
                }
            }
        }
        return ans;
    }
}
```

76、骑士拨号器（935）

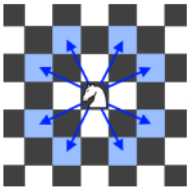
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

935. 骑士拨号器

难度 中等 40 收藏 分享 切换为英文 关注 反馈

国际象棋中的骑士可以按下图所示进行移动：



1	2	3
4	5	6
7	8	9
	0	

这一次，我们将 “骑士” 放在电话拨号盘的任意数字键（如上图所示）上，接下来，骑士将会跳 N-1 步。每一步必须是从一个数字键跳到另一个数字键。

每当它落在一个键上（包括骑士的初始位置），都会拨出键所对应的数字，总共按下 `N` 位数字。

你能用这种方式拨出多少个不同的号码？

因为答案可能很大，**所以输出答案模 `10^9 + 7`**。

示例 1：

```
输入：1
输出：10
```

示例 2：

```
输入：2
输出：20
```

示例 3：

```
输入：3
输出：46
```

提示：

- `1 <= N <= 5000`

https://leetcode-cn.com/problems/knight-dialer/solution/qi-shi-bo-hao-qi-by-leetcode/

方法一：动态规划

我们用 `f(start, n)` 表示骑士从数字 `start` 开始，跳了 `n - 1` 步得到不同的 `n` 位数字的个数。`f(start, n)` 可以从 `f(x, n - 1)` 转移而来，其中 `x` 是任意一个可以一步跳到 `start` 的数字。例如当 `start = 1` 时，`x` 可以为 `6` 或 `8`，因此有 `f(1, n) = f(6, n - 1) + f(8, n - 1)`。

最终的答案即为 `f(0, N) + f(1, N) + ... + f(9, N)`。我们可以使用滚动数组减少空间复杂度，这是因为 `f(start, n)` 只和 `f(x, n - 1)` 有关，因此在计算 `f(start, n)` 时，所有第二维小于 `n - 1` 的 `f` 值都不必存储。也就是说，我们只要实时存储当前正在计算的所有 `f` 值（`n` 位数字）以及上一个状态的 `f` 值（`n - 1` 位数字）即可。在 `Java` 代码中，我们使用 `dp[0][start]` 和 `dp[1][start]` 交替表示当前和上一个状态的 `f` 值。在 `Python` 代码中，我们使用 `dp2` 数组计算出当前的 `f` 值后，直接覆盖存储了上一个状态的 `f` 值的 `dp` 数组。

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：如果使用滚动数组，则空间复杂度为 $O(1)$，也可以看成 $O(10)$。否则空间复杂度为 $O(N)$。

方法 1：动规

```
class Solution {
    public int knightDialer(int N) {
        int MOD = 1_000_000_007;
        int[][] moves = new int[][]{
```

```
            {4,6},{6,8},{7,9},{4,8},{3,9,0},
            {},{1,7,0},{2,6},{1,3},{2,4}};

        int[][] dp = new int[2][10];
        Arrays.fill(dp[0], 1);
        for (int hops = 0; hops < N-1; ++hops) {
            Arrays.fill(dp[~hops & 1], 0);
            for (int node = 0; node < 10; ++node)
                for (int nei: moves[node]) {
                    dp[~hops & 1][nei] += dp[hops & 1][node];
                    dp[~hops & 1][nei] %= MOD;
                }
        }

        long ans = 0;
        for (int x: dp[~N & 1])
            ans += x;
        return (int) (ans % MOD);
    }
}
```

方法 2：常规

```
class Solution {
    private static final int mod = 1000000007;
    public int knightDialer(int N) {
        long[] array = new long[10];
        Arrays.fill(array, 1);
        for(int n = 2; n <= N; n++) {
            long a1 = array[6] + array[8];
            long a2 = array[7] + array[9];
            long a3 = array[4] + array[8];
            long a4 = array[3] + array[9] + array[0];
            long a6 = array[1] + array[7] + array[0];
            long a7 = array[2] + array[6];
            long a8 = array[1] + array[3];
            long a9 = array[4] + array[2];
            long a0 = array[4] + array[6];
            array[0] = a0 % mod;
            array[1] = a1 % mod;
            array[2] = a2 % mod;
            array[3] = a3 % mod;
            array[4] = a4 % mod;
            array[5] = 0;
            array[6] = a6 % mod;
            array[7] = a7 % mod;
```

```
            array[8] = a8 % mod;
            array[9] = a9 % mod;
        }

        long sum = 0;
        for(long a : array) {
            sum += a;
        }

        return (int)(sum % mod);
    }
}
```

77、最大为 N 的数字组合（902）

902. 最大为 N 的数字组合

难度 困难 40 收藏 分享 切换为英文 关注 反馈

我们有一组排序的数字 D，它是 {'1','2','3','4','5','6','7','8','9'} 的非空子集。（请注意，'0' 不包括在内。）

现在，我们用这些数字进行组合写数字，想用多少次就用多少次。例如 D = {'1','3','5'}，我们可以写出像 '13', '551', '1351315' 这样的数字。

返回可以用 D 中的数字写出的小于或等于 N 的正整数的数目。

示例 1：

输入： D = ["1","3","5","7"], N = 100
输出： 20
解释：
可写出的 20 个数字是：
1, 3, 5, 7, 11, 13, 15, 17, 31, 33, 35, 37, 51, 53, 55, 57, 71, 73, 75, 77.

示例 2：

输入： D = ["1","4","9"], N = 1000000000
输出： 29523
解释：
我们可以写 3 个一位数字，9 个两位数字，27 个三位数字，
81 个四位数字，243 个五位数字，729 个六位数字，
2187 个七位数字，6561 个八位数字和 19683 个九位数字。
总共，可以使用D中的数字写出 29523 个整数。

提示：

1. D 是按排序顺序的数字 '1'~'9' 的子集。
2. 1 <= N <= 10^9

https://leetcode-cn.com/problems/numbers-at-most-n-given-digit-set/solution/zui-da-wei-n-de-shu-zi-zu-he-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：数位动态规划

分析

我们称满足 `X <= N` 且仅包含 `D` 中出现的数字的 `X` 为合法的。我们的目标是找出所有合法的 `X` 的个数。

设 `N` 是一个 `K` 位数，那么对于任意一个小于 `K` （假设有 `k` 位，即 `k < K`）的数，如果它仅包含 `D` 中出现的数字，那么它就是合法的，并且 `k` 位数中，合法的数一共有 $|D|^k$ 个。

考虑完位数小于 `K` 的数，我们接下来考虑位数等于 `K` 的数，我们用 `N = 2345` 作为例子来考虑所有合法的 `K = 4` 位数。

- 如果第 `1` 个数位比 `N` 中对应的第 `1` 个数位（即 `2`）小，那么剩下的 `3` 个数位我们可以使用 `D` 中的任何一个数字，因此有 $|D|^{k-1}$ 个合法的数。
- 如果第 `1` 个数位和 `N` 中对应的第 `1` 个数位（即 `2`）相等，那么从第 `2` 个数位开始，它既可以比 `N` 中对应的第 `2` 个数位（即 `3`）小，也可以相等。此时相当于我们在考虑一个 `K - 1` 位数的问题。

算法

我们用 `dp[i]` 表示小于等于 `N` 中最后 `|N| - i` 位数的合法数的个数，例如当 `N = 2345` 时，`dp[0]`, `dp[1]`, `dp[2]`, `dp[3]` 分别表示小于等于 `2345`, `345`, `45`, `5` 的合法数的个数。我们从大到小计算 `dp[i]`，状态转移方程为：

```
dp[i] = (number of d in D with d < S[i]) * ((|D|) ** (|N| - i - 1))
```

即我们枚举第 `|N| - i` 位数，后面的 `|N| - i - 1` 位数可以在 `D` 中任选。如果 `N` 的第 `|N| - i` 位数在 `D` 中，上述的状态转移方程还需要加上一项 `dp[i + 1]`。

最终的答案为 `dp[0]` 加上所有 `k < K` 位的合法的数。

复杂度分析

- 时间复杂度：$O(|N|)$，也可以写作 $O(\log N)$，它们只相差一个常数，是等价的。
- 空间复杂度：$O(|N|)$。

方法 1：数位 DP（1ms）

```
class Solution {
    public int atMostNGivenDigitSet(String[] D, int N) {
        String S = String.valueOf(N);
        int K = S.length();
        int[] dp = new int[K+1];
        dp[K] = 1;

        for (int i = K-1; i >= 0; --i) {
            // compute dp[i]
            int Si = S.charAt(i) - '0';
            for (String d: D) {
                if (Integer.valueOf(d) < Si)
                    dp[i] += Math.pow(D.length, K-i-1);
                else if (Integer.valueOf(d) == Si)
                    dp[i] += dp[i+1];
            }
        }

        for (int i = 1; i < K; ++i)
            dp[0] += Math.pow(D.length, i);
```

```
            return dp[0];
        }
    }
```

方法二：数学

分析

我们令 `B = |D|`，一个合法的数仅包含 `D` 中的数字，如果我们把 `D` 中数字从小到大映射为 `[1 .. B]`，那么将对我们的计数有很大的便利。例如，当 `D` 包含 `[1, 3, 5, 7]` 时，我们将它映射为 `[1, 2, 3, 4]`，那么合法的数也从 `1, 3, 5, 7, 11, 13, 15, 17, 31, ...` 映射为 `1, 2, 3, 4, 11, 12, 13, 14, 21, ...`。这样的好处是，对于任何一个映射好的数，我们可以用类似进制转换的方式，得到它是第一个合法的数，例如 `34` 就是第 `3 * B + 4 = 3 * 4 + 4 = 16` 个合法的数。

有了这样的性质，我们可以先求出小于等于 `N` 的最大的合法的数，随后对它进行映射，如果它为第 `m` 个合法的数，就说明小于等于 `N` 的合法的数一共有 `m` 个。

算法

如果求出了小于等于 `N` 的最大的合法的数 `X`，后面的两步（映射，进制转换）的方法就都比较显然了，因此我们只重点说明求出 `X` 的方法。

我们从 `X` 的最高位开始，每次参考 `N` 中对应的数字，写下一个数位，直到写完最低位的数字。在写某一个数位时，会遇到下面的若干种情况（假设 `D` 为 `[2, 4, 6, 8]`）：

- 如果 `N` 中对应的数字在 `D` 中出现，那么在 `X` 的这一位写下与 `N` 相同的数字。例如当 `N` 为 `25123` 时，我们要参考最高位的数字 `2`，它在 `D` 中出现，因此我们也在 `X` 的最高位写下 `2`。
- 如果 `N` 中对应的数字没有在 `D` 中出现，但它大于 `D` 中最小的数字，那么就在 `X` 这一位写下 `D` 中比该数字小的最大的数字。例如当 `N` 为 `5123` 时，我们要参考最高位的数字 `5`，它没有在 `D` 中出现，我们就在 `X` 的最高位写下 `D` 中比 `5` 小的最大的数字 `4`。
- 如果 `N` 中对应的数字没有在 `D` 中出现，并且它小于 `D` 中最小的数字，那么我们需要把上一个写下的数字替换成 `D` 中一个更小的数字，如果 `D` 中没有更小的数字，那么就延续到再上一个写下的数字，只要某个数字可以替换成 `D` 中一个更小的数字，或者没有可以替换的数字。如果是前者，我们就从被替换的数字的下一位开始，将之后的所有数位都写下 `D` 中最大的数字；如果是后者，说明最大的合法的数 `X` 的位数比 `N` 小，那么我们写下一个 `|N| - 1` 位的，每一位都是 `D` 中最大的数字的数即可。下面给出了一个更加具体的例子：

  当 `N` 为 `123` 时，我们发现最高位的 `1` 比 `D` 中最小的 `2` 还要小，因为它是最高位，所以说明最大的合法的数 `X` 应该只有 `2` 位，因此 `X` 为 `88`；当 `N` 为 `4123` 时，我们在最高位填了 `4` 之后，次高位的 `1` 小于 `2`，因此将最高位换成更小的数字 `2`，再在后面都写下 `8`，得到 `2888`；当 `N` 为 `22123` 时Z，我们在前两位填了 `22` 之后，第三位的 `1` 小于 `2`，并且前两位都无法换成更小的数字，因此 `X` 应该只有 `4` 位，即 `8888`。

复杂度分析

- 时间复杂度：$O(|N|)$，也可以写作 $O(\log N)$，它们只相差一个常数，是等价的。
- 空间复杂度：$O(|N|)$。

方法 2：数学（0ms）

```
class Solution {
    public int atMostNGivenDigitSet(String[] D, int N) {
        int B = D.length; // bijective-base B
        char[] ca = String.valueOf(N).toCharArray();
        int K = ca.length;
        int[] A = new int[K];
        int t = 0;
```

```
        for (char c: ca) {
            int c_index = 0;  // Largest such that c >= D[c_index - 1]
            boolean match = false;
            for (int i = 0; i < B; ++i) {
                if (c >= D[i].charAt(0))
                    c_index = i+1;
                if (c == D[i].charAt(0))
                    match = true;
            }

            A[t++] = c_index;
            if (match) continue;

            if (c_index == 0) { // subtract 1
                for (int j = t-1; j > 0; --j) {
                    if (A[j] > 0) break;
                    A[j] += B;
                    A[j-1]--;
                }
            }

            while (t < K)
                A[t++] = B;
            break;
        }

        int ans = 0;
        for (int x: A)
            ans = ans * B + x;
        return ans;
    }
}
```

78、2 出现的次数（面试题 17.06）【数位 dp】

更多硬核内容就在公众号「yes 的练级攻略」

面试题 17.06. 2出现的次数

难度 中等 24 收藏 分享 切换为英文 关注 反馈

编写一个方法，计算从 0 到 n (含 n) 中数字 2 出现的次数。

示例:

```
输入: 25
输出: 9
解释: (2, 12, 20, 21, 22, 23, 24, 25)(注意 22 应该算作两次)
```

提示:

- n <= 10^9

https://leetcode-cn.com/problems/number-of-2s-in-range-lcci/solution/javajie-fa-shu-wei-dp-shuang-100-by-sagacioushugo/

主要思路是数位dp:
以dp[i]表示n的1~i位组成的数字所包含2的个数，关键点在于推导出dp[i]与dp[i-1]的关系

例如：n = 3478

```
dp[1] == numberOf2sInRange(8)
dp[2] == numberOf2sInRange(78)
dp[3] == numberOf2sInRange(478)
dp[4] == numberOf2sInRange(3478)

dp[i] = f(dp[i-1]) ?
```

下面来分析一下dp[i]与dp[i-1]的关系
根据第i位的取值可分为4种情况：

1. 第i位是0
   例如：n = 102, 分析dp[2]和dp[1]的关系，即numberOf2sInRange(02)与numberOf2sInRange(2) (02实际是2，写作02便于理解)
   第i位是0，该位取值范围只有这一种可能，由此可得

```
dp[2] = dp[1]
numberOf2sInRange(02) = numberOf2sInRange(2)
```

2. 第i位是1
   例如：n = 178，分析dp[3]和dp[2]的关系，即numberOf2sInRange(178)与numberOf2sInRange(78)
   第3位是1，该位可能取0,1两种情况：

```
dp[3] = 当第3位是0，1-2位取00~99时2的次数 + 当第3位是1，1-2位取00~78时2的次数
dp[3] = numberOf2sInRange(99) + dp[2]
numberOf2sInRange(178) = numberOf2sInRange(99) + numberOf2sInRange(78)
```

3. 第i位是2
例如：n = 233, 分析dp[3]和dp[2]的关系，即numberOf2sInRange(233)与numberOf2sInRange(33)

```
dp[3] = 第3位取0-1,1-2位取00~99时2的次数 + 第3位是2,1-2位取00~33时2在1-2位出现的次数 + 第3位是2,1-2位取00~33时2在第
dp[3] = 2 *numberOf2sInRange(99) + dp[2] + 33 + 1
numberOf2sInRange(233) = 2 * numberOf2sInRange(99) + numberOf2sInRange(33) + 33 + 1
```

4. 第i位大于2
以 n = 478为例，分析dp[3]和dp[2]的关系，即numberOf2sInRange(478)与numberOf2sInRange(78)

```
dp[3] = 第3位取0-3,1-2位取00-99时2出现在1-2位的次数 + 第3位取4,1-2位取00-78时2的次数 + 第3位取2,1-2位取00-99时2出现
dp[3] = 4 * numberOf2sInRange(99) + dp[2] + 100
```

总结上面4种情况：

```
dp[i]与dp[i-1]的关系，假设n的第i位的值为k
dp[i] = k * numberOf2sInRange(99..9){共i-1个9} + dp[i-1] + {n % 10^(i-1) + 1 }{若k == 2}  + { 10^(i-1) } {若k > 2}
```

根据递推公式可以发现，若计算dp[i]，不仅要知道dp[i-1]还要知道numberOf2sInRange(99..9)，所以要同时计算numberOf2sInRange(99..9)的值

```
class Solution {
    public int numberOf2sInRange(int n) {
        if(n == 0) {
            return 0;
        }
        int digit = (int)Math.log10(n) + 1;
        int[][] dp = new int[digit+1][2];
        // dp[i][0] = numberOf2sInRange(n % pow(10, i)) 保存0~n 的1-i 位组成的数包含 2 的个数
        // dp[i][1] = numberOf2sInRange(99..9) 保存 i 位均为 9 包含 2 的个数
        dp[1][0] = n % 10 >= 2 ? 1:0;
        dp[1][1] = 1;
        for(int i = 2; i <= digit; i++) {
            int k = n/ ((int)Math.pow(10, i-1)) % 10;
            dp[i][0] = k * dp[i-1][1] + dp[i-1][0];
            if(k == 2) {
                dp[i][0] += n % (int)Math.pow(10, i-1) +1;
            } else if(k > 2){
                dp[i][0] +=  (int)Math.pow(10, i-1);
            }
            dp[i][1] = 10 * dp[i-1][1] + (int)Math.pow(10, i-1); //计算 1-i 位均为 9 的值包含 2 的个数
        }
        return dp[digit][0];
    }
}
```

79、n 个骰子的点数（剑指 60）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 60. n个骰子的点数

难度 简单 90 收藏 分享 切换为英文 关注 反馈

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

示例 1:

```
输入：1
输出：[0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]
```

示例 2:

```
输入：2
输出：
[0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.05556,0.02778]
```

限制：

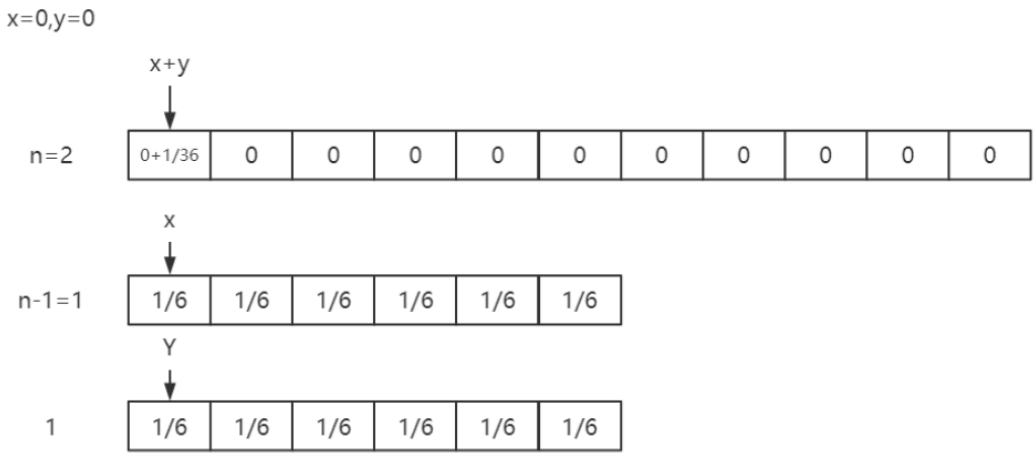
`1 <= n <= 11`

https://leetcode-cn.com/problems/nge-tou-zi-de-dian-shu-lcof/solution/java-dong-tai-gui-hua-by-zhi-xiong/

解题思路：

根据动态规划的思想分解子问题。
我们可以把n个骰子的点数分解为n-1个骰子的点数加上一个骰子的点数。
根据1个骰子的点数概率数组求出2的点数概率数组，根据2的点数概率数组求出3的点数概率数组....直到求出n的点数。
那么我们要怎么根据n-1个骰子的点数概率数组求出n个骰子的点数概率数组呢?
我们假设n=2。
则我们已知1的点数概率数组为{1/6d,1/6d,1/6d,1/6d,1/6d,1/6d}
我们要求的2个骰子，可以分解为n-1个骰子和1个骰子
同时易知2个骰子的点数概率数组长度为2*5+1。
则:
如图，x,y分别为n-1数组和1数组指针。

如图，x,y分别为n-1数组和1数组指针。



使得n-1点数概率数组和1点数概率数组元素两两相乘，并将乘积结果**加到**n点数概率数组上。
运算完成后就得到了最终的n点数概率数组。

基本思路如上，然后我们可以根据动态规划的套路：
1.构造dp数组：tmp[]为n个骰子的点数概率数组，pre[]为n-1个骰子的点数概率数组，一个骰子的点数概率数组显然是6个六分之一,不需要另设数组。
2.初始化dp数组：pre[]={1/6d,1/6d,1/6d,1/6d,1/6d,1/6d}
3.构造状态转移方程：`tmp[x+y]+=pre[x]*num[y];`

```
class Solution {
    public double[] twoSum(int n) {
        double[] pre = {1/6d, 1/6d, 1/6d, 1/6d, 1/6d, 1/6d};
        for (int i = 2; i <= n; i++){
            double[] tmp = new double[5 * i + 1];
            for (int j = 0; j < pre.length; j++){
                for (int x = 0; x < 6; x++){
                    tmp[j + x] += pre[j] / 6;
                }
            }
            pre = tmp;
        }
        return pre;
    }
}
```

80、粉刷房子（256）

256. 粉刷房子

难度 简单 59 收藏 分享 切换为英文 关注 反馈

假如有一排房子，共 n 个，每个房子可以被粉刷成红色、蓝色或者绿色这三种颜色中的一种，你需要粉刷所有的房子并且使其相邻的两个房子颜色不能相同。

当然，因为市场上不同颜色油漆的价格不同，所以房子粉刷成不同颜色的花费成本也是不同的。每个房子粉刷成不同颜色的花费是以一个 n x 3 的矩阵来表示的。

例如，costs[0][0] 表示第 0 号房子粉刷成红色的成本花费；costs[1][2] 表示第 1 号房子粉刷成绿色的花费，以此类推。请你计算出粉刷完所有房子最少的花费成本。

注意：

所有花费均为正整数。

示例：

```
输入: [[17,2,17],[16,16,5],[14,3,19]]
输出: 10
解释: 将 0 号房子粉刷成蓝色，1 号房子粉刷成绿色，2 号房子粉刷成蓝色。
     最少花费: 2 + 5 + 3 = 10。
```

https://leetcode-cn.com/problems/paint-house/solution/256-fen-shua-fang-zi-by-klb/

解题思路

本题适合动态规划的思想，即第 n 排房子刷漆方案不仅受第 n-1 排房子颜色的影响，还受 0 到 n-1 排房子的总花费影响。

假设房子有 n 排，编号分别是 0 到 n-1。

刷第 0 排房子时：
选择红色，则截至到第 0 排，总花费是 R0；
选择蓝色，则截至到第 0 排，总花费是 B0；
选择绿色，则截至到第 0 排，总花费是 G0；
总共有三种花费方案：[R0,B0,G0]

刷第 1 排房子时：
选择红色，那么上一层只能是蓝色或绿色并且截至到上一排总花费最低的，截至到本排，总话费是 R1 + min(B0,G0)
选择蓝色，那么上一层只能是红色或绿色并且截至到上一排总花费最低的，截至到本排，总话费是 B1 + min(R0,G0)
选择绿色，那么上一层只能是红色或蓝色并且截至到上一排总花费最低的，截至到本排，总话费是 G1 + min(R0,B0)
总共有三种花费方案：[R1 + min(B0,G0), B1 + min(R0,G0), G1 + min(R0,B0)]

.....

以此类推，每遍历到一排，都可以计算出当前排选择的颜色和所需的成本。

```
class Solution {
    public int minCost(int[][] costs) {
        if (costs.length == 0) return 0;
        int n = costs.length - 1;
        for (int i = 1; i <= n; i++) {
            costs[i][0] = costs[i][0] + Math.min(costs[i - 1][1], costs[i - 1][
2]);
            costs[i][1] = costs[i][1] + Math.min(costs[i - 1][0], costs[i - 1][
2]);
            costs[i][2] = costs[i][2] + Math.min(costs[i - 1][0], costs[i - 1][
1]);
        }
        return Math.min(Math.min(costs[n][0], costs[n][1]), costs[n][2]);
    }
}
```

81、粉刷房子 2（265）

265. 粉刷房子 II

难度 困难 37 收藏 分享 切换为英文 关注 反馈

假如有一排房子，共 n 个，每个房子可以被粉刷成 k 种颜色中的一种，你需要粉刷所有的房子并且使其相邻的两个房子颜色不能相同。

当然，因为市场上不同颜色油漆的价格不同，所以房子粉刷成不同颜色的花费成本也是不同的。每个房子粉刷成不同颜色的花费是以一个 $n \times k$ 的矩阵来表示的。

例如，`costs[0][0]` 表示第 0 号房子粉刷成 0 号颜色的成本花费；`costs[1][2]` 表示第 1 号房子粉刷成 2 号颜色的成本花费，以此类推。请你计算出粉刷完所有房子最少的花费成本。

注意：

所有花费均为正整数。

示例：

```
输入: [[1,5,3],[2,9,4]]
输出: 5
解释: 将 0 号房子粉刷成 0 号颜色，1 号房子粉刷成 2 号颜色。最少花费: 1 + 4 = 5;
     或者将 0 号房子粉刷成 2 号颜色，1 号房子粉刷成 0 号颜色。最少花费: 3 + 2 = 5.
```

https://leetcode-cn.com/problems/paint-house-ii/solution/dong-tai-gui-hua-by-yue-ke-2-3/

解题思路

和粉刷房子一样，只是把3种颜色变成了n种颜色，多一次遍历就完事了

优化思路

这种解法的时间复杂度是N * K * K，事实上可以简化，我遍历每一行的每一个元素时都求了一次上一行的最小价格，所以我可以把这个过程拿到外面，每一行遍历完都求出该行的最小值和第二小的值，这样dp[i][j] = j == previous ? secondMin + costs[i][j] : Min + costs[i][j]，就是N * K的是时间复杂度了

```java
class Solution {
    public int minCostII(int[][] costs) {
        int row = costs.length;
        if (row == 0) return 0;
        int col = costs[0].length;
        int[][] dp = new int[row][col];
        for (int i = 0; i < col; i++) dp[0][i] = costs[0][i];
        for (int i = 1; i < row; i++) {
            for (int j = 0; j < col; j++) {
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = 0; k < col; k++) {
                    if (k == j) continue;
                    dp[i][j] = Math.min(dp[i][j], dp[i - 1][k]);
                }
                dp[i][j] += costs[i][j];
            }
        }
        int res = Integer.MAX_VALUE;
        for (int i = 0; i < col; i++) res = Math.min(res, dp[row - 1][i]);
        return res;
    }
}
```

82、布尔运算（面试题 08.14）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 08.14. 布尔运算

难度 中等 24 收藏 分享 切换为英文 关注 反馈

给定一个布尔表达式和一个期望的布尔结果 result，布尔表达式由 `0` (false)、`1` (true)、`&` (AND)、`|` (OR) 和 `^` (XOR) 符号组成。实现一个函数，算出有几种可使该表达式得出 result 值的括号方法。

示例 1:

```
输入: s = "1^0|0|1", result = 0

输出: 2
解释: 两种可能的括号方法是
1^(0|(0|1))
1^((0|0)|1)
```

示例 2:

```
输入: s = "0&0&0&1^1|0", result = 1

输出: 10
```

提示:

- 运算符的数量不超过 19 个

https://leetcode-cn.com/problems/boolean-evaluation-lcci/solution/java-ji-yi-hua-sou-suo-by-npe_tle/

思路

设 `dp[s][e][r]` 为从索引 `s` 到索引 `e` 值为 `r` 的方案数。那么，我们可以拿一个指针 `k` (从 `s` 遍历到 `e - 1`)，将区间 `[s, e]` 分成两个部分，`[s, k]` 和 `[k + 2, e]`。其中 `k+1` 的位置是运算符。同时，由于是布尔运算，因此左右两部分的结果页要么是 `0`，要么是 `1`。组合也就是四种情况，`{00, 01, 10, 11}`。然后判断这四种情况通过 `k+1` 位置的运算符算出来的值是不是能够等于 `r` (`dp[s][e][r]` 中的 `r`)。能等的话，就将左右两部分的方案数相乘即可。

```
class Solution {
private char[] arr;
    private int[][][] dp;
    private int getBoolAns(int val1, int val2, char operator) {
        switch (operator) {
            case '&':
                return val1 & val2;
            case '|':
                return val1 | val2;
            case '^':
                return val1 ^ val2;
        }
        return val1 & val2;
    }
    /**
```

更多硬核内容就在公众号「yes 的练级攻略」

```
     * 返回从索引 start 到 end 值为 result 的不同括号方案的个数
     */
    private int rec(int start, int end, int result) {
        if (start == end) {
            return arr[start] - '0' == result ? 1 : 0;
        }

        if (dp[start][end][result] != -1) {
            return dp[start][end][result];
        }

        int ansCount = 0;
        for (int k = start; k < end; k+=2) {
            char operator = arr[k + 1];
            for (int i = 0; i <= 1; i++) {
                for (int j = 0; j <= 1; j++) {
                    if (getBoolAns(i, j, operator) == result) {
                        ansCount += rec(start, k, i) * rec(k + 2, end,
j);
                    }
                }
            }
        }

        dp[start][end][result] = ansCount;
        return ansCount;
    }
    public int countEval(String s, int result) {
        arr = s.toCharArray();
        int len = arr.length;
        dp = new int[len][len][2];
        for (int i = 0; i < len; i++) {
            for (int j = 0; j < len; j++) {
                Arrays.fill(dp[i][j], -1);
            }
        }
        return rec(0, len - 1, result);
    }
}
```

83、堆箱子（面试题 08.13）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 08.13. 堆箱子

难度 困难 22 收藏 分享 切换为英文 关注 反馈

堆箱子。给你一堆n个箱子，箱子宽 wi、深 di、高 hi。箱子不能翻转，将箱子堆起来时，下面箱子的宽度、高度和深度必须大于上面的箱子。实现一种方法，搭出最高的一堆箱子。箱堆的高度为每个箱子高度的总和。

输入使用数组 [wi, di, hi] 表示每个箱子。

示例1:

```
输入: box = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
输出: 6
```

示例2:

```
输入: box = [[1, 1, 1], [2, 3, 4], [2, 6, 7], [3, 4, 5]]
输出: 10
```

提示:

1. 箱子的数目不大于3000个。

https://leetcode-cn.com/problems/pile-box-lcci/solution/dong-tai-gui-hua-dpji-lu-mei-yi-ceng-zui-gao-gao-d/

解题思路

dp[]记录每一层最高高度, 往上迭代(取合格的box), 然后合格的box的max也得到了.
每一个(不绝对哦)box都可以做前面的相对自己合格的box的上一层, 这样记录在dp[]里面就好了.

```
class Solution {
    public int pileBox(int[][] box) {
        // 利用一个记录 array, 记录每一层的最高, 然后在这一层的基础上算上一层.
        // 先把一个固定下俩, 其他两个逆序, 这样以后重排列
        Arrays.sort(box, (a, b) -> {return a[0] == b[0] ? a[1] == b[1] ? b[2]-a
[2] : b[1]-a[1] : a[0]-b[0];});
        int[] dpVal = new int[box.length];
        int res = dpVal[0];
        // 已经排序好 width 了, 去对比 depth 和 height.
        for (int i = 0; i < box.length; i ++) {
            //
            int maxVal = 0, baseDepth = box[i][1], baseHeight = box[i][2];
            for (int j = 0; j < i; j ++) {
                // 如果这个是合格的 box(depth 和 height 都小的话), 就记录它的高度为最
大
                // 因为 i 是递增的, j 是在 i 之内的, 可以理解为, 每个 j 都可以影响之后的
以它(j)为合格的 i 记录.
                if (baseDepth > box[j][1] && baseHeight > box[j][2]) {
                    // i 记录最大的之前合格的 j, 这样加上自己(i)就可以组成 dpVal[i]
了, 然后为 i 以后的 box 做贡献
                    maxVal = Math.max(maxVal, dpVal[j]);
                }
            }
```

```
                // 加上就是当前的最大的了.
                dpVal[i] = maxVal + baseHeight;
                res = Math.max(res, dpVal[i]);
            }
            return res;
        }
}
```

84、最大黑方阵（面试题 17.23）

面试题 17.23. 最大黑方阵

难度 中等 12 收藏 分享 切换为英文 关注 反馈

给定一个方阵，其中每个单元(像素)非黑即白。设计一个算法，找出 4 条边皆为黑色像素的最大子方阵。

返回一个数组 [r, c, size] ，其中 r , c 分别代表子方阵左上角的行号和列号， size 是子方阵的边长。若有多个满足条件的子方阵，返回 r 最小的，若 r 相同，返回 c 最小的子方阵。若无满足条件的子方阵，返回空数组。

示例 1:

输入:
```
[
   [1,0,1],
   [0,0,1],
   [0,0,1]
]
```
输出: [1,0,2]
解释: 输入中 0 代表黑色，1 代表白色，标粗的元素即为满足条件的最大子方阵

示例 2:

输入:
```
[
   [0,1,1],
   [1,0,1],
   [1,1,0]
]
```
输出: [0,0,1]

提示:

- matrix.length == matrix[0].length <= 200

https://leetcode-cn.com/problems/max-black-square-lcci/solution/javadong-tai-gui-hua-by-archerxyz/

1.计 xSum[x][y] = m[0][y]+m[1][y]+...+m[x-1][y];
ySum[x][y] = m[x][0]+m[x][1]+...+m[x][y-1];
2.此时 符合条件的正方形((x1,y1),(x2,y2))判断条件为:
xSum[x2][y1]-xSum[x1][y1] = 0,
且: xSum[x2][y2] - xSum[x1][y2] = 0,
且: ySum[x1][y2] - ySum[x1][y1] = 0,
且: ySum[x2][y2] - ySum[x2][y1] = 0
也即，四个边长分别的和为零
3.扩大矩形边长做判断，取出最大边长且最接近原点的点

时间复杂度 O(n^3),空间复杂度 O(n^2);

暴力破解详见题解:https://leetcode-cn.com/problems/max-black-square-lcci/solution/javabao-li-po-jie-by-archerxyz/

https://leetcode-cn.com/problems/max-black-square-lcci/solution/javabao-li-po-jie-by-archerxyz/

```
class Solution {
    private int[][] matrix;
    private int[][] xSum;
    private int[][] ySum;
    public int[] findSquare(int[][] matrix) {
        if(matrix.length == 0)
            return new int[0];
        if(matrix.length == 1){
            if(matrix[0][0] == 0)
                return new int[] {0,0,1};
            else
                return new int[0];
        }
        int len = matrix.length;

        this.xSum = new int[len+1][len];
        this.ySum = new int[len][len+1];

        this.matrix = matrix;
        for(int i = 0; i < len; ++i){
            for(int j = 0; j < len; ++j){
                ySum[i][j+1] = ySum[i][j] + matrix[i][j];
                xSum[j+1][i] = xSum[j][i] + matrix[j][i];
            }
        }
        int[] res = new int[3];

        for(int i = 0; i < len; ++i){
            if(len-i < res[2])
                break;

            for(int j = 0; j < len; ++j){
                if(len -j < res[2])
                    continue;
                if(matrix[i][j] == 0){
                    int l = square(i,j);
                    if(res[2] < l){
                        res[0] = i;
                        res[1] = j;
                        res[2] = l;
                    }
                }
            }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }
        if(res[2] == 0)
            return new int[0];
        return res;
    }

    private int square(int x, int y){
        int l = 1;
        int ans = 1;
        while(x+l < matrix.length && y+l < matrix.length){
            if(xSum[x+l+1][y]-xSum[x][y] == 0 && ySum[x][y+l+1]-ySum[x][y] == 0
){
                if(xSum[x+l+1][y+l]-xSum[x][y+l] == 0 && ySum[x+l][y+l+1]-ySum[
x+l][y] == 0)
                    ans = l+1;
            } else
                break;
            ++l;
        }
        return ans;
    }
}
```

85、

五、前缀

https://leetcode-cn.com/leetbook/read/dynamic-programming-1-plus/5mu64d/

前缀和是一种查询数组中任意区间的元素的和的数据结构，这里数组给定之后就不变了。针对这个不变的数组，前缀和用于多次查询区间 [i, j] 上元素的和。

对于动态规划而言，前缀和的意义主要有两点：

1. 一维和二维前缀和的推导，分别用到了单串和矩阵中最经典的状态设计以及状态转移；
2. 在一些更复杂的动态规划问题中，状态转移的时候需要依赖区间和，因为状态转移是非常频繁的操作，因此必须高效地求区间和才能使得状态转移的时间复杂度可接受，此时就必须用到前缀和了。

除此之外，一些问题需要前缀和与其它数据结构配合来解决，也有两类：

1. 先预处理出前缀和数组，这一步是动态规划，然后在前缀和数组上用其它数据结构解决；
2. 还是按照动态规划的方式求前缀和，也需要额外的数据结构维护前缀和，但不是预处理好前缀和数组之后再用数据结构计算，而是每求出一个前缀和，就更新一次数据结构并维护答案。

前缀和的推导和计算隐含着动态规划的基本思想，同时它的状态设计是线性动态规划中比较简单的那一类。与线性动态规划一样，前缀和也有一维和二维两种场景。
虽然前缀和本身很简单，但需要用到它解决的问题非常多，与其它数据结构配合的变化也很多，因此需要从线性动态规划中剥离出来单独学习。

更多硬核内容就在公众号「yes 的练级攻略」

前缀和简介

考虑以下问题：

给定长度 n 的序列 a：$a_0, a_1, ..., a_{n-1}$，给每个前缀求一次和，$S_0 = 0, S_i = \sum_{j=0}^{i-1} a_j$ 。这些前缀和维护在一个长度 $n+1$ 数组 S 里, 称为**前缀和数组**。

有两类在数组 a 上的求和需求

1. 前缀和：求 a[0..i] 的和
2. 区间和：求区间 a[L, R] 的和
   对于前缀和，S[i + 1] 刚好就是答案，因为这就是前缀和的定义：$S_{i+1} = \sum_{j=0}^{i} a_j$。

对于区间和，解决此问题最直观的方法是枚举 [L, R] 上的所有元素求和：$sum(L, R) = \sum_{idx=L}^{R} nums[idx]$ 。这种方法虽然直观，但是不足是查询一次需要 $O(N)$ 的时间，并且每来一个新的查询，就要重新枚举元素求和。通过简单的画图推导可以得到答案为 $S[R+1] - S[L]$

如果查询次数很多，当新查询来时，此前的查询计算的中间结果很多是可以直接用的，新的查询不必重新枚举，例如 此前查询过 `sum(5, 10)`，现在来了新查询 `sum(4, 8)` 在计算新查询时，5 ~ 8 这一段在计算此前的查询 `sum(5, 10)` 的时候已经计算过了，新查询的计算过程可以写成 $nums[4] + sum(5, 8)$，而不用全部枚举。

如果已经有了 **前缀和数组**，那么通过简单的画图推导可以知道，两个前缀和 S[R + 1] 和 S[L] 的差刚好就是 [L, R] 上的区间和。已有前缀和数组之后，这一步操作就是 $O(1)$ 的。

现将前缀和数组预处理出来，然后在每次查询中直接通过前缀和数组来计算而不是通过原数组，这是一种缓存中间结果的思想，如果把这一思想执行彻底，可以将所有可能的问题 `sum(i, j) 其中 0<=i<=j<=n-1`，将计算它们所需的所有中间结果先算一次缓存下来。这里提到的是缓存所有所需的中间结果，而不是子问题的中间结果，因为计算 `sum(i, j)` 需要的中间结果是前 $i-1$ 个数的和 `sum(0,i)` 和前 j 个数的和 `sum(0,j+1)`。最终 `sum(i,j) = sum(0,j+1) - sum(0,i)` 这对于所有的区间查询都是适用的。

以上将中间结果缓存思路与动态规划中缓存子问题的解的思路是一致的。

定义 `sums[k]` 为 [0..k-1] 的和，其中 `sums[0]` 表示数组中没有数字被选中，`sums[1]` 表示之选中第一个数 `nums[0]`。预先计算 0 ~ k $(0 <= k <= n-1)$ 的和，这一系列的和都是从 0 开始的，因此称为前缀和。公式如下：

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

1. $k = 0 : sum(0, 0) = 0$
2. $1 <= k <= n - i : sum(0, k) = \sum_{i=0}^{k-1} nums[i]$

此后的区间查询都可以利用公式 `sum(i, j) = sums(0, j + 1) - sum(0, i)`

在上一章中，我们把线性动态规划中几种主流问题及其对应的 dp 状态设计做了总结。其中最基础的一种是单串 `dp[i]`，并且只与子问题 $i - 1$ 有关，即 `dp[i] = f(dp[i-1])`。前缀和就是这种情况，`sums[i]` 只与 `sums[i-1]` 有关。推导前缀和数组的过程是 $O(N)$ 的，如果区间和的查询次数达到了 $O(N)$ 那么计算区间和的时间复杂度是摊销 $O(1)$ 的。

以上就是前缀和的基础介绍了，其中比较关键的有两点

1. 预处理前缀和的过程是最简单且最经典的单串动态规划问题。
2. 先将所有位置的前缀和预处理出来，然后再处理区间和的查询，这是一种先缓存中间结果再处理查询的思路，因为**这些中间结果在查询时需要反复用到**，缓存之后就不用反复计算了，因此花时间预处理这些信息是有效的。

前缀和除了求区间和之外，还有一些其它的应用：

1. 在用 dp 的方式推 `sums[i]` 的时候，有时求完 `sums[i]` 需要查询以前算过的结果计算某种指标，需要用其它数据结构将前面的计算结果维护起来，例如哈希表等等，在求每个位置的前缀和的过程中，查询数据结构并更新答案，这是前缀和的一大类问题，变化比较多，力扣上这类题也有很多，在本章最后一节中汇总了一些题目，在下一节也选择了典型题目做讲解。
2. 前缀和的逆运算是差分，对原序列求出其差分序列，然后在对得到的差分序列求其前缀和序列，可以得到原序列，这在处理一些区间修改的问题时很有用，参考后面小节的题目讲解。
3. 前缀和还可以推广到二维上，并用于快速求矩形和，二维前缀和的计算过程是最经典的矩阵上的线性动态规划，参考后面一节的题目和讲解。

求区间和

利用前缀和求区间和的思想，已经求前缀和的过程在上一节中已经重点介绍，这里主要回顾一下前缀和中的动态规划思想，如下：

```
状态定义：sums[i] := [0..i-1] 的和
状态转移：sums[i] = a[i - 1] + sums[i - 1]
初始化：sums[0] = 0
```

这是最简单的单串线性动态规划，其思想在上一章重点介绍。求解该动态规划问题后得到数组 sums 。然后区间和问题就变成了两个前缀和的差的问题。

```
int rangeRum(int L, int R)
{
    return sums[R + 1] - sums[L];
}
```

题目

- 303. 区域和检索 - 数组不可变

上一节的最后提到前缀和可以推广到二维，解决的问题以及预处理时的算法思想与一维前缀和完全一样，实现二维前缀和以及利用二维前缀和与数据结构的配合解决一些问题，在力扣上有一些题目。见本章后续节。

实现前缀和问题

以下两道题是实现前缀和数据结构。问题是前缀和关心的最基本的问题：区间求和、子矩形求和

可以借助这两道题将前面介绍的前缀和的思想和计算方式实现出来，并加深理解。
中间的前缀和计算的核心逻辑在后续题目中会反复使用。

- 区域和检索 - 数组不可变
- 二维区域和检索 - 矩阵不可变

1、区域和检索-数组不可变（303）

303. 区域和检索 - 数组不可变

难度 简单 187 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 *nums*，求出数组从索引 *i* 到 *j* (*i* ≤ *j*) 范围内元素的总和，包含 *i*, *j* 两点。

示例：

```
给定 nums = [-2, 0, 3, -5, 2, -1]，求和函数为 sumRange()

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

说明：

1. 你可以假设数组不可变。
2. 会多次调用 *sumRange* 方法。

https://leetcode-cn.com/problems/range-sum-query-immutable/solution/marveljian-dan-de-xue-xi-bi-ji-303-by-tyanyonecanc/

解题思路

类似于本题这样**求一个序列指定区间内元素之和**的问题，都可以用以下方式求解：

对于序列s1、s2...sn，定义一个数组a，令数组每个元素$a[i] = s1 + s2 + ... + si$。
这样一来，区间$[i, j]$的区域和为：
$si + ... + sj = (s1 + s2 + ... + sj) - (s1 + s2 + ...si - 1) = a[j] - a[i - 1]$。

```
class NumArray {
    private int[] sum;
    public NumArray(int[] nums) {
        sum = Arrays.copyOf(nums, nums.length);
        for(int i = 1; i < sum.length; i++)
            sum[i] += sum[i - 1];
    }
    public int sumRange(int i, int j) {
        return (i == 0) ? sum[j] : (sum[j] - sum[i - 1]);
    }
}
/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = new NumArray(nums);
 * int param_1 = obj.sumRange(i,j);
```

*/

2、二维区域和检索 - 矩阵不可变（304）

304. 二维区域和检索 - 矩阵不可变

难度 中等 106 收藏 分享 切换为英文 关注 反馈

给定一个二维矩阵，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (*row1*, *col1*) ，右下角为 (*row2*, *col2*)。

上图子矩阵左上角 (row1, col1) = **(2, 1)** ，右下角(row2, col2) = **(4, 3)**，该子矩形内元素的总和为 8。

示例:

```
给定 matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]

sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

说明:

1. 你可以假设矩阵不可变。
2. 会多次调用 *sumRegion* 方法。
3. 你可以假设 *row1* ≤ *row2* 且 *col1* ≤ *col2*。

https://leetcode-cn.com/problems/range-sum-query-2d-immutable/solution/er-wei-qu-yu-he-jian-suo-ju-zhen-bu-ke-bian-by-lee/

方法二：缓存 [超过内存限制]

因为 sumregion 可以多次调用，所以我们肯定需要做一些预处理。

算法:

我们可以通过预先计算所有可能的矩形区域和并将其存储在哈希表中，从而在额外的空间中换取速度。每个 sumregion 查询现在只需要恒定的时间复杂度。

复杂度分析

- 时间复杂度：每次查询的时间 $O(1)$。$O(m^2n^2)$ 的时间预计算。每个 sumregion 查询需要 $O(1)$ 时间，因为哈希表查找的时间复杂度是恒定的。预计算将花费 $O(m^2n^2)$ 时间，因为总共需要缓存 $M^2 \times n^2$ 的可能值。
- 空间复杂度：$O(m^2n^2)$。由于矩形区域的左上方和右下方的可能性各不相同，因此所需的额外空间为 $O(m^2n^2)$。

方法三：缓存行

还记得我们使用累积和数组的一维版本吗？我们可以直接用它来解这个二维版本吗？

算法:

尝试将二维矩阵视为一维数组的 m 行。为了找到区域和，我们只需要在区域中逐行累积和。

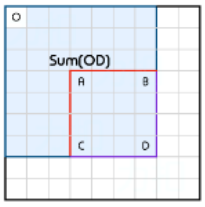
更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

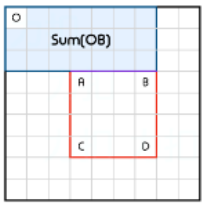
- 时间复杂度：每次查询的时间 $O(m)$。$O(mn)$ 时间预计算。构造函数中的预计算需要 $O(mn)$ 时间。sumregion 查询需要 $O(m)$ 时间。
- 空间复杂度：$O(mn)$。该算法使用 $O(mn)$ 空间存储所有行的累积和。

方法四：智能缓存

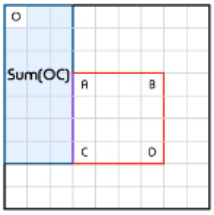
算法：
我们在一维版本中使用了累积和数组。我们注意到累积和是根据索引 0 处的原点计算的。将这个类比扩展到二维情况，我们可以预先计算出一个与原点相关的累积区域和，即 $(0,0)$。



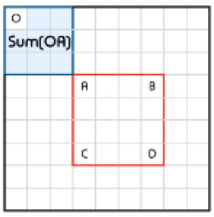
Sum(OD)是相对于原点(0,0)的累计区域和。
如何使用预先计算的累积区域和得出 $Sum(ABCD)$ 呢?



Sum(OB)是矩形顶部的累积区域和。



Sum(OC)是矩形左侧的累积区域和。



Sum(OA) 是矩形左上角的累积区域和。
区域 $Sum(OA)$ 由 $Sum(OB)$ 和 $Sum(OC)$两次覆盖。我们可以使用包含排除原则计算 $Sum(ABCD)$ 如下：

$$sum(abcd) = sum(od) - sum(ob) - sum(oc) + sum(oa)$$

复杂度分析

- 时间复杂度：每次查询时间 $O(1)$，$O(mn)$ 的时间预计算。构造函数中的预计算需要 $O(mn)$ 时间。每个 sumregion 查询需要 $O(1)$ 时间 。
- 空间复杂度：$O(mn)$，该算法使用 $O(mn)$ 空间存储累积区域和。

这道题虽然名义上属于动态规划，但我觉得这个 tag 名不副实。我更愿意用考研娱乐圈老师张宇的一句话："**来一包二重积分**！"来形容这道 LeetCode 试题。

在信号与系统中，我们学过著名的广义函数——δ 函数，这个函数的定义是：

$$\int_{-\infty}^{a}\delta(x)\,dx=\begin{cases}1,\text{if } a\geq 0^{+}\\0,\text{if } a\leq 0^{-}\end{cases}\tag{0}$$

冲激函数最重要的性质就是：抽样性质。

$$f(x)\,\delta(x)=f(0)\,\delta(x)\tag{1}$$

或者可以写成：

$$\int_{-\infty}^{+\infty}f(x)\,\delta(x)\,dx=f(0)\tag{2}$$

很容易想到一个不严谨的推广，就是将上述表达式平行地推广到二维空间上：

$$\iint_{-\infty<x<a,-\infty<y<b}\delta(x,y)\,dxdy=\begin{cases}1,\text{if } a\geq 0^{+}\text{ AND }b\geq 0^{+}\\0,\text{else}\end{cases}\tag{3}$$

抽样性质可以类似的推广成：

$$\iint_{x,y\in\mathbb{R}^2}f(x,y)\,\delta(x,y)\,dxdy=f(0,0)\tag{4}$$

而本题要计算这个封闭空间内的数的和，很容易让人想到，这就是一个二维冲激函数的背景计算二重积分的问题，其中冲激点是离散的下标。而我们观察积分区间的形式—矩形，这让我们想到了另外一个背景：利用概率论中两个变量的联合分布函数，计算给定区间内的概率的计算方法，假设二维随机变量的联合分布函数是 $F(x,y)$，那么计算给定区间 $[x_1,x_2]\times[y_1,y_2]$ 内的概率的公式是：

$$f(x_1,x_2,y_1,y_2)=F(x_2,y_2)+F(x_1,y_1)-F(x_1,y_2)-F(x_2,y_1)\tag{5}$$

而本题和这个计算的要求是几乎一致的，只是我们需要在头脑中稍微修改一下坐标范围和坐标轴的方向。维护一个二维数组 `dp`，长度为 `matrix.length+1`，高度为 `matrix[0].length+1`，其中 `dp[i][j]` 定义为在 $[0,i)\times[0,j)$ 范围内的求和，这实际上就和二重积分的表达式非常像了。很容易参考公式 (5) 得到 `dp` 的转移关系：

$$\mathrm{dp}[i][j]=\begin{cases}0,\text{if } i=0\text{ OR }j=0\\\mathrm{dp}[i-1][j]+dp[i][j-1]-\mathrm{dp}[i-1][j-1]+\mathrm{matrix}[i-1][j-1]\end{cases}\tag{6}$$

之后，每当我们要计算题目中给定的表达式的时候，只需要参考利用联合分布函数计算概率的公式 (5)，就可以快速得到结果：

$$\begin{aligned}f(row1,col1,row2,col2)=\mathrm{dp}[row2+1][col2+1]+\mathrm{dp}[row1][col1]-\\\mathrm{dp}[row2+1][col1]-\mathrm{dp}[row1][col2+1]\end{aligned}\tag{7}$$

代码如下：

这个算法的空间复杂度是 `O(matrix.length * matrix[0].length)`，初始化的时间复杂度是 `O(matrix.length * matrix[0].length)`。之后每次调用函数，只需要使用 `O(1)` 的时间即可计算出结果。

容易想到，使用这种重积分的思想，理论上可以将问题推广到高维的问题上，但是这样的情况下，上述公式 (5) 的形式就更复杂了。例如三维的情况下：

$$\begin{aligned}f(x_1,x_2,y_1,y_2,z_1,z_2)=F(x_2,y_2,z_2)+F(x_1,y_1,z_2)-F(x_1,y_2,z_2)-F(x_2,y_1,z_2)-\\F(x_2,y_2,z_1)-F(x_1,y_1,z_1)+F(x_1,y_2,z_1)+F(x_2,y_1,z_1)\end{aligned}\tag{8}$$

其中 $F(x,y,z)$ 代表的是 $[0,x)\times[0,y)\times[0,z)$ 内元素的总和。

用这个思路也可以类似地完成三维的问题。容易想到，在这种思路下，推广后的解法，constructor 的时间复杂度是 $O\left(2^{\text{dimensions}}\times\prod\text{matrix.dimension}[i]\right)$，空间复杂度也是这么多，其中 `matrix.dimension[i]` 代表的是高维矩阵中各个维度的长度，用 Matlab 的话来说，就是 `size(matrix, i)`。每次查询的时间复杂度是 $O(2^{\text{dimensions}})$，其中 `dimensions` 代表矩阵的维度。例如本题是 2 维，所以需要 4 次，如果是三维，就需要 8 次，以此类推。

```
class NumMatrix {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    int[][] dp;
    public NumMatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) retu
rn;
        dp = new int[matrix.length + 1][matrix[0].length + 1];
        for (int i = 0; i < matrix.length; i++) {
            for (int j = 0; j < matrix[0].length; j++) {
              dp[i+1][j+1] = dp[i+1][j] + dp[i][j+1] + matrix[i][j] - dp[i][j];
            }
        }
    }
    public int sumRegion(int row1, int col1, int row2, int col2) {
        return dp[row2 + 1][col2 + 1] - dp[row1][col2 + 1] - dp[row2 + 1][col1]
 + dp[row1][col1];
    }
}
/**
 * Your NumMatrix object will be instantiated and called as such:
 * NumMatrix obj = new NumMatrix(matrix);
 * int param_1 = obj.sumRegion(row1,col1,row2,col2);
 */
```

数据结构维护前缀和

在上一节中提到，在用 dp 的方式推 `sums[i]` 的时候，有时求完 `sums[i]` 需要查询以前算过的结果计算某种指标，需要用其它数据结构将前面的计算结果维护起来，以便高效查询。以下几个问题就是以上思路的直接应用。

将前缀和维护在数据结构中，以便于后续的 **多次** 查询，最常见的是维护在哈希表中。

力扣上这种题目非常多，更多题目详见本章后续节。下面考虑几个经典问题：

- 第一问：$a_0, a_1, ..., a_{n-1}$ 上有没有一个区间，其和为 target 。

计算前缀和数组 `sums[i]` 。当扫描到 i 时， $a_0, a_1, ..., a_{i-1}$ 的前缀和都已经求过了，在计算的过程中将前缀和维护在数据结构中，以便于后续的多次查询，本题在之后要查询前缀和的值是否存在，因此维护在 `unordered_set` (哈希表)里。

求完当前值 a[i] 对应的前缀和 S[i+1], 在插入到 `unordered_set` 之前先问：S[i+1] - target 在 `unordered_set` 中是否出现。

- 如果出现，说明存在以 i 结尾的某个区间，和为 target, 则找到答案。
- 如果不出现，则没有以 i 结尾的区间，和为 target，继续枚举 $i+1$。

上面的问题还可以有变种：

- 第二问：$a_0, a_1, ..., a_{n-1}$ 上有多少个区间，其和为 target 。

按照第一问的思路，把 `unordered_set` 改成 `unordered_map` 就可以

更多硬核内容就在公众号「yes 的练级攻略」

参考题目：560. 和为 K 的子数组

- 第三问：$a_0, a_1, ..., a_{n-1}$ 上有没有一个区间，其和大于/小于 target 。

整体思路与第一问相同，但是维护前缀和的数据结构需要从哈希表变为线段树

参考题目：327. 区间和的个数

- 第四问: 一棵树上有没有某个路径，其和为 target

这是第一问的树形版本，dfs（前序遍历）时，栈里存的是当前节点到根的链，这条链上的和可以作为前缀和维护在 `unordered_map` 里。从左子树跳到右子树的时候，左子树的所有节点对应的前缀和要先从 `unordered_map` 中删掉。

参考题目： 437. 路径总和 III

数据结构维护前缀和相关练习题：

- HashMap 维护 (1)，键是前缀和（状态）的值，值为第一次出现时的索引
- HashMap 维护 (2)，键是前缀和（前缀状态）的值，值为出现次数
- HashMap 维护 (3)，键是前缀和模 K 的余数（可以理解为前缀状态，状态为前缀和模K)
- 前缀和（积）与后缀和（积）均需要
- 二维前缀和

数据结构维护前缀和问题：HashMap 维护（2）

HashMap 维护（2），键是前缀和（前缀状态）的值，值为出现次数。

- 和为 K 的子数组
- 统计优美子数组 —— 前缀状态为奇数的个数

3、和为 K 的子数组（560）【前缀和+哈希表优化】

560. 和为K的子数组

难度 中等 527 收藏 分享 切换为英文 关注 反馈

给定一个整数数组和一个整数 **k**，你需要找到该数组中和为 **k** 的连续的子数组的个数。

示例 1：

```
输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。
```

说明：

1. 数组的长度为 [1, 20,000]。
2. 数组中元素的范围是 [-1000, 1000] ，且整数 **k** 的范围是 [-1e7, 1e7]。

https://leetcode-cn.com/problems/subarray-sum-equals-k/solution/bao-li-jie-fa-qian-zhui-he-qian-zhui-he-you-hua-ja/

方法四：前缀和 + 哈希表优化

- 由于只关心次数，不关心具体的解，我们可以使用哈希表加速运算；
- 由于保存了之前相同前缀和的个数，计算区间总数的时候不是一个一个地加，时间复杂度降到了 $O(N)$。

这个思路不是很容易想到，需要多做一些类似的问题慢慢培养感觉。

- 同类问题有：
  - 「力扣」第 1 题：两数之和；
  - 「力扣」第 1248 题：统计「优美子数组」；
  - 「力扣」第 454 题：四数相加 II。

解释一开始 `preSumFreq.put(0, 1);` 的意义

想一想算法的意思：

- 计算完包括了当前数前缀和以后，我们去查一查在当前数之前，有多少个前缀和等于 `preSum - k` 呢？

这是因为满足 `preSum - (preSum - k) == k` 的区间的个数是我们所关心的。

- 对于一开始的情况，下标 0 之前没有元素，可以认为前缀和为 0，个数为 1 个，因此 `preSumFreq.put(0, 1);`，这一点是必要且合理的。

具体的例子是：`nums = [3,...], k = 3`，和 `nums = [1, 1, 1,...], k = 3`。

复杂度分析：

- 时间复杂度：$O(N)$，这里 N 是数组的长度；
- 空间复杂度：$O(N)$。

```
import java.util.HashMap;
import java.util.Map;
public class Solution {
    public int subarraySum(int[] nums, int k) {
        // key：前缀和，value：key 对应的前缀和的个数
        Map<Integer, Integer> preSumFreq = new HashMap<>();
        // 对于下标为 0 的元素，前缀和为 0，个数为 1
        preSumFreq.put(0, 1);
        int preSum = 0;
        int count = 0;
        for (int num : nums) {
            preSum += num;
            // 先获得前缀和为 preSum - k 的个数，加到计数变量里
            if (preSumFreq.containsKey(preSum - k)) {
                count += preSumFreq.get(preSum - k);
            }
            // 然后维护 preSumFreq 的定义
            preSumFreq.put(preSum, preSumFreq.getOrDefault(preSum, 0) +
 1);
        }
        return count;
    }
}
```

4、统计优美子数组（1248）

更多硬核内容就在公众号「yes 的练级攻略」

1248. 统计「优美子数组」

难度 中等 127 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `nums` 和一个整数 `k`。

如果某个 **连续** 子数组中恰好有 `k` 个奇数数字，我们就认为这个子数组是「**优美子数组**」。

请返回这个数组中「优美子数组」的数目。

示例 1：

```
输入：nums = [1,1,2,1,1], k = 3
输出：2
解释：包含 3 个奇数的子数组是 [1,1,2,1] 和 [1,2,1,1] 。
```

示例 2：

```
输入：nums = [2,4,6], k = 1
输出：0
解释：数列中不包含任何奇数，所以不存在优美子数组。
```

示例 3：

```
输入：nums = [2,2,2,1,2,2,1,2,2,2], k = 2
输出：16
```

提示：

- `1 <= nums.length <= 50000`
- `1 <= nums[i] <= 10^5`
- `1 <= k <= nums.length`

https://leetcode-cn.com/problems/count-number-of-nice-subarrays/solution/hua-dong-chuang-kou-qian-zhui-he-bi-xu-miao-dong-b/

解题思路

一、滑动窗口

1. 不断右移 `right` 指针来扩大滑动窗口，使其包含 `k` 个奇数；
2. 若当前滑动窗口包含了 `k` 个奇数，则如下「计算当前窗口的优美子数组个数」：
   - 统计第 `1` 个奇数左边的偶数个数 `leftEvenCnt`。这 `leftEvenCnt` 个偶数都可以作为「优美子数组」的起点，因此起点的选择有 `leftEvenCnt + 1` 种（因为可以一个偶数都不取，因此别忘了 +1 喔）。
   - 统计第 `k` 个奇数右边的偶数个数 `rightEvenCnt`。这 `rightEvenCnt` 个偶数都可以作为「优美子数组」的终点，因此终点的选择有 `rightEvenCnt + 1` 种（因为可以一个偶数都不取，因此别忘了 +1 喔）。
   - 因此「优美子数组」左右起点的选择组合数为 `(leftEvenCnt + 1) * (rightEvenCnt + 1)`。

👇代码注释很详细👇（时间复杂度 $O(N)$，空间复杂度 $O(1)$）

方法 1：滑动窗口（6ms）

```
class Solution {
    public int numberOfSubarrays(int[] nums, int k) {
        int left = 0, right = 0, oddCnt = 0, res = 0;
        while (right < nums.length) {
            // 右指针先走，每遇到一个奇数则 oddCnt++。
            if ((nums[right++] & 1) == 1) oddCnt++;
```

```
            //  若当前滑动窗口 [left, right) 中有 k 个奇数了，进入此分支统计当前滑动
窗口中的优美子数组个数。
            if (oddCnt == k) {
                // 先将滑动窗口的右边界向右拓展，直到遇到下一个奇数（或出界）
                // rightEvenCnt 即为第 k 个奇数右边的偶数的个数
                int tmp = right;
                while (right < nums.length && (nums[right] & 1) == 0) {
                    right++;
                }
                int rightEvenCnt = right - tmp;
                // leftEvenCnt 即为第 1 个奇数左边的偶数的个数
                int leftEvenCnt = 0;
                while ((nums[left] & 1) == 0) {
                    leftEvenCnt++;
                    left++;
                }
                // 第 1 个奇数左边的 leftEvenCnt 个偶数都可以作为优美子数组的起点
                // (因为第1 个奇数左边可以 1 个偶数都不取，所以起点的选择
有 leftEvenCnt + 1 种）
                // 第 k 个奇数右边的 rightEvenCnt 个偶数都可以作为优美子数组的终点
                // (因为第k 个奇数右边可以 1 个偶数都不取，所以终点的选择
有 rightEvenCnt + 1 种）
                // 所以该滑动窗口中，优美子数组左右起点的选择组合数
为 (leftEvenCnt + 1) * (rightEvenCnt + 1)
                res += (leftEvenCnt + 1) * (rightEvenCnt + 1);

                // 此时 left 指向的是第 1 个奇数,因为该区间已经统计完了,因此 left 右
移一位，oddCnt--
                left++;
                oddCnt--;
            }
        }
        return res;
    }
}
```

二、前缀和

1. 计算前缀和数组 `arr`：遍历原数组，每遍历一个元素，计算当前的前缀和（即到当前元素为止，数组中有多少个奇数）；
2. 对上述前缀和数组，双重循环统计 `arr[j] - arr[i] == k` 的个数，这样做是 $O(N^2)$ 的（这里会超时哦）。
3. **优化**：因此，我们可以像「1. 两数之和」那样使用 `HashMap` 优化到 $O(N)$，键是「前缀和」，值是「前缀和的个数」（下面代码中具体使用的是 `int[] prefixCnt` 数组，下标是「前缀和」，值是「前缀和的个数」），因此我们可以遍历原数组，每遍历到一个元素，计算当前的前缀和 `sum`，就在 `res` 中累加上前缀和为 `sum - k` 的个数。

👇代码注释很详细👇（时间复杂度 $O(N)$，空间复杂度 $O(N)$）

方法 2：前缀和（3ms）

```java
class Solution {
    public int numberOfSubarrays(int[] nums, int k) {
        // 数组 prefixCnt 的下标是前缀和（即当前奇数的个数），值是前缀和的个数。
        int[] prefixCnt = new int[nums.length + 1];
        prefixCnt[0] = 1;
        // 遍历原数组，计算当前的前缀和，统计到 prefixCnt 数组中，
        // 并且在 res 中累加上与当前前缀和差值为 k 的前缀和的个数。
        int res = 0, sum = 0;
        for (int num: nums) {
            sum += num & 1;
            prefixCnt[sum]++;
            if (sum >= k) {
                res += prefixCnt[sum - k];
            }
        }
        return res;
    }
}
```

5、区间和的个数（327）

327. 区间和的个数

难度 困难 113 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 nums，返回区间和在 [lower, upper] 之间的个数，包含 lower 和 upper。
区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。

说明:
最直观的算法复杂度是 $O(n^2)$，请在此基础上优化你的算法。

示例:

输入: *nums* = [-2,5,-1], *lower* = -2, *upper* = 2,
输出: 3
解释: 3个区间分别是: [0,0], [2,2], [0,2]，它们表示的和分别为: -2, -1, 2。

https://leetcode-cn.com/problems/count-of-range-sum/solution/treemap-onlognfang-fa-by-wdw87/

首先想到前缀和，直接计算并统计结果划分O(n2)的时间复杂度。

前缀和数组为sum[];
满足条件的区间和为:
lower <= sum[i] - sum[j] <= upper;
将上述式子变形得到:
sum[i] - upper <= sum[j] <= sum[i] - lower;
也就是说在前缀和数组sum[0...i]中，满足上述条件的sum[j]都对应着一个满足条件的区间;

TreeMap底层实现为红黑树，是一种有序映射，查找时间复杂度为O(logN),并且可方便的进行区间查找
由于每次查找都是在sum[0...i]的范围内，所以可以边计算前缀和边插入TreeMap，同时统计满足条件的区间个数

方法 1：TreeMap（49ms）

```java
class Solution {
    public int countRangeSum(int[] nums, int lower, int upper) {
```

```
        if(nums ==  null || nums.length == 0) return 0;
        //键值为区间和和这个区间和出现的次数
        TreeMap<Long, Integer> tree = new TreeMap<>();
        tree.put(0L, 1);
        int count = 0;
        long sum = 0L;
        for(int num : nums){
            sum += num;
            //subMap()返回一个值在sum - upper 和sum - lower 之间的子集合,values()
方法获得这个映射的值得视图
            for(int cnt : tree.subMap(sum - upper, true, sum - lower, true).val
ues()){
                count += cnt; //统计满足条件的区间和个数
            }
            tree.put(sum, tree.getOrDefault(sum, 0) + 1);
        }
        return count;
    }
}
```

https://leetcode-cn.com/problems/count-of-range-sum/solution/java-ji-yu-gui-bing-pai-xu-de-fen-zhi-fa-jian-dan-/

此题解法：在归并排序算法中增加一个计数方法。

计什么数?
计算区间和在[lower,upper]之间的区间个数

怎么计数?
归并排序的特点，数组被分为两半，左右两半分别有序
对于左边数组的每一个元素在右边数组中找到两个边界:low_bound和up_bound
low_bound是第一个使得区间和>=lower的位置
up_bound是第一个使得区间和>uper的位置

up_bound-low_bound就是区间和在[lower,uper]的区间个数

方法 2：归并（8ms）

```
class Solution {
    public int countRangeSum(int[] nums, int lower, int upper) {
        int count=0;
        long []pre_sum=new long[nums.length];
        long sum=0;
        for(int i=0;i<nums.length;i++){
            sum+=nums[i];
            pre_sum[i]=sum;  // pre_sum[i]: nums[0:i]的和
        }
        count=merge_sort(pre_sum,0,nums.length-1,lower,upper);
        return count;
    }

    public int merge_sort(long []pre_sum,int left,int right,int lower,int upper
){
```

```
        if(left>right) return 0;
        if(left==right){
            if(pre_sum[left]>=lower&&pre_sum[right]<=upper) return 1;
            else return 0;
        }
        int count=0;
        int mid=left+(right-left)/2;
        //System.out.println(left+"   "+right);
        count=merge_sort(pre_sum,left,mid,lower,upper)+merge_sort(pre_sum,mid+1
,right,lower,upper);
        int low_bound=mid+1;
        int up_bound=mid+1;
        for(int i=left;i<=mid;i++){ // 计数
            // 退出 while 后 low_bound 是在右边数组中第一个使得
pre_sum[low_bound] - pre_sum[i] >= lower 的位置
            while(low_bound<=right&&pre_sum[low_bound]-pre_sum[i]<lower) low_bo
und++;
            // up_bound 是在右边数组中第一个使得
pre_sum[up_bound] - pre_sum[i] >upper 的位置
            while(up_bound<=right&&pre_sum[up_bound]-pre_sum[i]<=upper) up_boun
d++;
            count+=(up_bound-low_bound);
        }
        long []merged=new long[right-left+1];
        int l=left,r=mid+1;
        int index=0;
        while(l<=mid&&r<=right){   // 归并
            if(pre_sum[l]<=pre_sum[r]) merged[index++]=pre_sum[l++];
            else merged[index++]=pre_sum[r++];
        }
        while(l<=mid) merged[index++]=pre_sum[l++];
        while(r<=right) merged[index++]=pre_sum[r++];
        System.arraycopy(merged,0,pre_sum,left,merged.length); // 辅助空间的数据
复制到原数组中
        return count;
    }
}
```

数据结构维护前缀和问题：HashMap 维护（1）

以下若干组题目都是利用数据结构维护前缀和，实现统计数组中的某个指标的目的。HashMap 维护（1），键是前缀和（状态）的值，值为第一次出现时的索引。

- 和等于 k 的最长子数组长度
- 连续数组
- 每个元音包含偶数次的最长子字符串 —— 前缀状态为 a,e,i,o,u 的个数的奇偶

6、和等于 k 的最长子数组长度（325）【前缀和+哈希表】

325. 和等于 k 的最长子数组长度

难度 中等 56 收藏 分享 切换为英文 关注 反馈

给定一个数组 *nums* 和一个目标值 *k*，找到和等于 *k* 的最长子数组长度。如果不存在任意一个符合要求的子数组，则返回 0。

注意:
nums 数组的总和是一定在 32 位有符号整数范围之内的。

示例 1:

输入: *nums* = [1, -1, 5, -2, 3], *k* = 3
输出: 4
解释: 子数组 [1, -1, 5, -2] 和等于 3，且长度最长。

示例 2:

输入: *nums* = [-2, -1, 2, 1], *k* = 1
输出: 2
解释: 子数组 [-1, 2] 和等于 1，且长度最长。

进阶:
你能使时间复杂度在 O(*n*) 内完成此题吗?

https://leetcode-cn.com/problems/maximum-size-subarray-sum-equals-k/solution/325-he-deng-yu-k-de-zui-chang-zi-shu-zu-chang-du-j/

本题的主要思路：

1. 计算出每个索引的前缀和，利用一个数组 `sum` 保存，利用一个 `max` 储存最长子数组长度；
2. 利用HashMap储存每个前缀和和对应的索引，如果出现前缀和相同的情况，则储存较小的索引（因为要求最长子数组）；
3. 利用一个指针 `i` 作为子数组的结尾从后向前遍历，寻找map中是否储存有 `key` 为 `sum[i] - k` 的索引，如果有则更新 `max`；
4. 当指针的值小于等于 `max` 的值后，则无需再继续遍历；
5. 返回max则为最终结果。

```
class Solution {
    public int maxSubArrayLen(int[] nums, int k) {
        int len = nums.length;
        int max = 0;
        // 求前缀和，并利用 Map<前缀和, 对应索引> 储存
```

```
        int[] sum = new int[len + 1];
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 0);
        for (int i = 1; i <= len; i ++) {
            sum[i] = sum[i - 1] + nums[i - 1];
            if (!map.containsKey(sum[i]))
                map.put(sum[i], i);
        }
        // 从后向前遍历数组，i 为子数组的结尾，寻找符合条件的前缀和及其索引
        for (int i = len; i > max; i --) {
            if (map.containsKey(sum[i] - k)) {
                max = Math.max(max, i - map.get(sum[i] - k));
            }
        }
        return max;
    }
}
```

7、连续数组（525）【前缀和+哈希表】

525. 连续数组

难度 中等 163 收藏 分享 切换为英文 关注 反馈

给定一个二进制数组, 找到含有相同数量的 0 和 1 的最长连续子数组（的长度）。

示例 1:

输入: [0,1]
输出: 2
说明: [0, 1] 是具有相同数量0和1的最长连续子数组。

示例 2:

输入: [0,1,0]
输出: 2
说明: [0, 1] (或 [1, 0]) 是具有相同数量0和1的最长连续子数组。

注意: 给定的二进制数组的长度不会超过50000。

https://leetcode-cn.com/problems/contiguous-array/solution/qian-zhui-he-chai-fen-ha-xi-biao-java-by-liweiwei1/

更多硬核内容就在公众号「yes 的练级攻略」

首先应该想到的是「暴力解法」，使用两个 `for` 循环枚举所有可能的区间，然后判断区间是否符合题意，进而计算区间的长度。

这道题我个人感觉如果之前没有做过相关的问题，很难想到比较好的思路。这里我的思路来自「力扣」第 1371 题：每个元音包含偶数次的最长子字符串，周赛的时候没有做出来，然后看了 题解 学到的技巧。

- 这里引入一个概念，叫做「不变量」，题目要我们找的「不变量」是「区间里」 `0` 的个数和 `1` 的个数相等的区间的长度；
- 区间里的操作，一种思路是「滑动窗口」，但是发现很难下手，另一种思路就是「前缀和之差」，这一点在学习「线段树」的时候，是作为例子引入的；
- **重点思路**：要利用到「前缀和」还有希望能够通过一次遍历记住结果，多半要使用到哈希表，这里做的一个处理是把所有的 `0` 都看成 `-1`，如此一来：**如果一个区间的和是 `0`，就说明这个区间里的 `0`（被视为 `-1`） 和 1 的数量相同**；
- 由于求的是最长区间，因此就需要记录前缀和的数值第 1 次出现的下标，相同的前缀再次出现，就说明这一段区间的和为 `0`（把 `0` 看成 `-1` 以后），在遍历的过程中，记录最长的区间的长度；
- 区间的问题几个思路供参考：1、滑动窗口（双指针） 2、前缀和的差 3、线段树（树状数组）。

总结：

- 一开始介绍的「不变量」是比较繁琐的，把它转换成一个等价的表示，就使得解决问题变得简单；
- 在编码的过程中，还需要注意一些细节，这里放在代码注释里说明；
- 有相关问题的经验还是比较重要的，在平常刷题的过程中需要多总结。

可以做一下相似的问题，「力扣」第 352 题：和等于 k 的最长子数组长度。

复杂度分析：

- 时间复杂度：$O(N)$，这里 N 是数组的长度，算法遍历了一次数组，每一个元素的操作都是常数次的，因此整体时间复杂度是 $O(N)$；
- 空间复杂度：$O(N)$。

```
class Solution {
    public int findMaxLength(int[] nums) {
        int len = nums.length;
        int max = 0, preSum = 0;
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, -1);  // 下标 0 之前的位置是 -1
        // 把数组中的 0 都看成 -1
        for (int i = 0; i < len; i++) {
            // 把 0 视为 -1，pre 是先加了，所以后面计算的时候是 i - map.get(preSum)，
注意下标 + 1 和不加 1 的差别
            if (nums[i] == 1) preSum += 1;
            else preSum += -1;
            if (map.containsKey(preSum)) {
                max = Math.max(max, i - map.get(preSum));
            } else {
                // 只记录这个数字第 1 次出现的下标
                map.put(preSum, i);
            }
        }
        return max;
    }
```

}

8、每个元音包含偶数次的最长子字符串（1371）

1371. 每个元音包含偶数次的最长子字符串

难度 中等 272 收藏 分享 切换为英文 关注 反馈

给你一个字符串 `s` ，请你返回满足以下条件的最长子字符串的长度：每个元音字母，即 'a'，'e'，'i'，'o'，'u'，在子字符串中都恰好出现了偶数次。

示例 1：

输入： s = "eleetminicoworoep"
输出： 13
解释： 最长子字符串是 "leetminicowor" ，它包含 e，i，o 各 2 个，以及 0 个 a，u 。

示例 2：

输入： s = "leetcodeisgreat"
输出： 5
解释： 最长子字符串是 "leetc" ，其中包含 2 个 e 。

示例 3：

输入： s = "bcbcbc"
输出： 6
解释： 这个示例中，字符串 "bcbcbc" 本身就是最长的，因为所有的元音 a，e，i，o，u 都出现了 0 次。

提示：

- `1 <= s.length <= 5 x 10^5`
- `s` 只包含小写英文字母。

https://leetcode-cn.com/problems/find-the-longest-substring-containing-vowels-in-even-counts/solution/mei-ge-yuan-yin-bao-han-ou-shu-ci-de-zui-chang-z-2/

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法一：前缀和 + 状态压缩

思路和算法

我们先来考虑暴力方法怎么做。最直观的方法无非就是枚举所有子串，遍历子串中的所有字符，统计元音字母出现的个数。如果符合条件，我们就更新答案，但这样肯定会因为超时而无法通过所有测试用例。

再回顾一下上面的操作，其实每个子串对应着一个区间，那么有什么方法可以在不重复遍历子串的前提下，快速求出这个区间里元音字母出现的次数呢？答案就是前缀和，对于一个区间，我们可以用两个前缀和的差值，得到其中某个字母的出现次数。

我们对每个元音字母维护一个前缀和，定义 $pre[i][k]$ 表示在字符串前 i 个字符中，第 k 个元音字母一共出现的次数。假设我们需要求出 $[l,r]$ 这个区间的子串是否满足条件，那么我们可以用 $pre[r][k]-pre[l-1][k]$，在 $O(1)$ 的时间得到第 k 个元音字母出现的次数。对于每一个元音字母，我们都判断一下其是否出现偶数次即可。

我们利用前缀和优化了统计子串的时间复杂度，然而枚举所有子串的复杂度仍需要 $O(n^2)$，不足以通过本题，还需要继续进行优化，避免枚举所有子串。我们考虑枚举字符串的每个位置 i，计算以它结尾的满足条件的最长字符串长度。其实我们要做的就是快速找到**最小**的 $j\in[0,i)$，满足 $pre[i][k]-pre[j][k]$（即每一个元音字母出现的次数）均为偶数，那么以 i 结尾的最长字符串 $s[j+1,i]$ 长度就是 $i-j$。

有经验的读者可能马上就想到了利用哈希表来优化查找的复杂度，但是单单利用前缀和，我们无法找到 i 和 j 相关的恒等式，像 1248. 统计优美子数组 这道题我们是能明确知道两个前缀的差值是恒定的。那难道就没办法了么？其实不然，这道题我们还有一个性质没有充分利用：我们需要找的子串中，每个元音字母都恰好出现了**偶数次**。

偶数这个条件其实告诉了我们，对于满足条件的子串而言，两个前缀和 $pre[i][k]$ 和 $pre[j][k]$ 的奇偶性一定是相同的，因为小学数学的知识告诉我们：奇数减奇数等于偶数，偶数减偶数等于偶数。因此我们可以对前缀和稍作修改，从维护元音字母出现的次数改作维护元音字母出现次数的**奇偶性**。这样我们只要实时维护每个元音字母出现的奇偶性，那么 $s[j+1,i]$ 满足条件当且仅当对于所有的 k，$pre[i][k]$ 和 $pre[j][k]$ 的奇偶性都相等，此时我们就可以利用哈希表存储**每一种**奇偶性（即考虑所有的元音字母）对应**最早出现的位置**，边遍历边更新答案。

题目做到这里基本上做完了，但是我们还可以进一步优化我们的编码方式，如果直接以每个元音字母出现次数的奇偶性为哈希表中的键，难免有些冗余，我们可能需要额外定义一个状态：

```
{
  a: cnta, // a 出现次数的奇偶性
  e: cnte, // e 出现次数的奇偶性
  i: cnti, // i 出现次数的奇偶性
  o: cnto, // o 出现次数的奇偶性
  u: cntu  // u 出现次数的奇偶性
}
```

将这么一个结构当作我们哈希表存储的键值，如果题目稍作修改扩大了字符集，那么维护起来可能会比较吃力。考虑到出现次数的奇偶性其实无非就两个值，0 代表出现了偶数次，1 代表出现了奇数次，我们可以将其压缩到一个二进制数中，第 k 位的 0 或 1 代表了第 k 个元音字母出现的奇偶性。举一个例子，假如到第 i 个位置，`u o i e a` 出现的奇偶性分别为 `1 1 0 0 1`，那么我们就可以将其压成一个二进制数 $(11001)_2=(25)_{10}$ 作为它的状态。这样我们就可以将 5 个元音字母出现次数的奇偶性压缩到了一个二进制数中，且连续对应了二进制数的 $[(00000)_2,(11111)_2]$ 的范围，转成十进制数即 $[0,31]$。因此我们也不再需要使用哈希表，直接用一个长度为 32 的数组来存储对应状态出现的最早位置即可。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为字符串 s 的长度。我们只需要遍历一遍字符串即可求得答案，因此时间复杂度为 $O(n)$。
- 空间复杂度：$O(S)$，其中 S 表示元音字母压缩成一个状态数的最大值，在本题中 $S=32$。我们需要对应 S 大小的空间来存放每个状态第一次出现的位置，因此需要 $O(S)$ 的空间复杂度。

```java
class Solution {
    public int findTheLongestSubstring(String s) {
        int n = s.length();
        int[] pos = new int[1 << 5];
```

```
        Arrays.fill(pos, -1);
        int ans = 0, status = 0;
        pos[0] = 0;
        for (int i = 0; i < n; i++) {
            char ch = s.charAt(i);
            if (ch == 'a') {
                status ^= (1 << 0);
            } else if (ch == 'e') {
                status ^= (1 << 1);
            } else if (ch == 'i') {
                status ^= (1 << 2);
            } else if (ch == 'o') {
                status ^= (1 << 3);
            } else if (ch == 'u') {
                status ^= (1 << 4);
            }
            if (pos[status] >= 0) {
                ans = Math.max(ans, i + 1 - pos[status]);
            } else {
                pos[status] = i + 1;
            }
        }
        return ans;
    }
}
```

9、最长公共前缀（14）【lintcode78】

14. 最长公共前缀

难度 简单 1179 收藏 分享 切换为英文

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 `""`。

示例 1:

```
输入：["flower","flow","flight"]
输出："fl"
```

示例 2:

```
输入：["dog","racecar","car"]
输出：""
解释：输入不存在公共前缀。
```

说明:

所有输入只包含小写字母 `a-z` 。

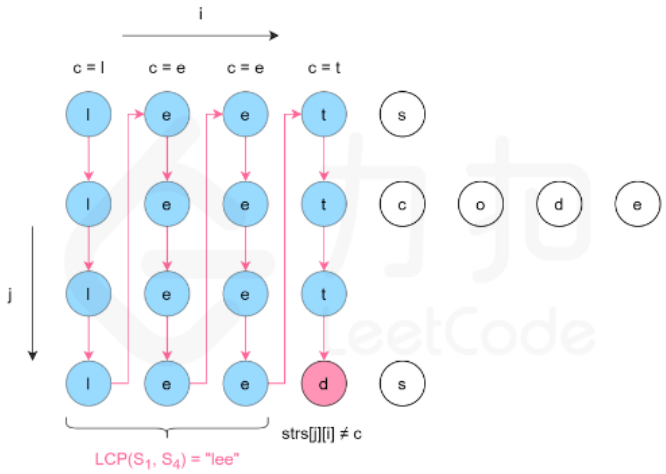
https://leetcode-cn.com/problems/longest-common-prefix/solution/zui-chang-gong-gong-qian-zhui-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：纵向扫描

方法一是横向扫描，依次遍历每个字符串，更新最长公共前缀。另一种方法是纵向扫描。纵向扫描时，从前往后遍历所有字符串的每一列，比较相同列上的字符是否相同，如果相同则继续对下一列进行比较，如果不相同则当前列不再属于公共前缀，当前列之前的部分为最长公共前缀。

☆收藏



复杂度分析

- 时间复杂度：$O(mn)$，其中 m 是字符串数组中的字符串的平均长度，n 是字符串的数量。最坏情况下，字符串数组中的每个字符串的每个字符都会被比较一次。
- 空间复杂度：$O(1)$。使用的额外空间复杂度为常数。

14 具体代码

```
class Solution {
    public String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }
        int length = strs[0].length();//第一个字符串的长度
        int count = strs.length;//所有字符串的个数
        for (int i = 0; i < length; i++) {
            char c = strs[0].charAt(i);//一个字符串的第 i 个字符
            for (int j = 1; j < count; j++) {
                if (i == strs[j].length() || strs[j].charAt(i) != c) {//如果遍历结束或者有其中一个不相等
                    return strs[0].substring(0, i);
                }
            }
        }
        return strs[0];
    }
}
```

10、最长公共前缀二（lintcode784）

更多硬核内容就在公众号「yes 的练级攻略」

784. 最长公共前缀 II

中文 English

给出 n 个字符串 dic ,和一个目标串，输出目标串与这 n 个字符串的最长公共前缀的长度的最大值。

样例
样例1

```
输入: dic = ["abcba","acc","abwsf"] and target = "abse"
输出: 2
解释:
"abse"与"abcba"的最长公共前缀为"ab"，长度为2,与"acc"最长公共前缀为"a"，长度为1，与"abwsf"最长公共前缀为"ab"，长度为2，max(2,1,2) = 2。
```

样例2

```
输入: dic = ["aaa","bbb","aabb"] and target = "aaab"
输出: 3
解释:
"aaab"与"aaa"最长公共前缀为"aaa"，长度为3，与"bbb"最长公共前缀为""，长度为0，与"aabb"最长公共前缀为"aa"，长度为2，max(3,0,2) = 3。
```

注意事项

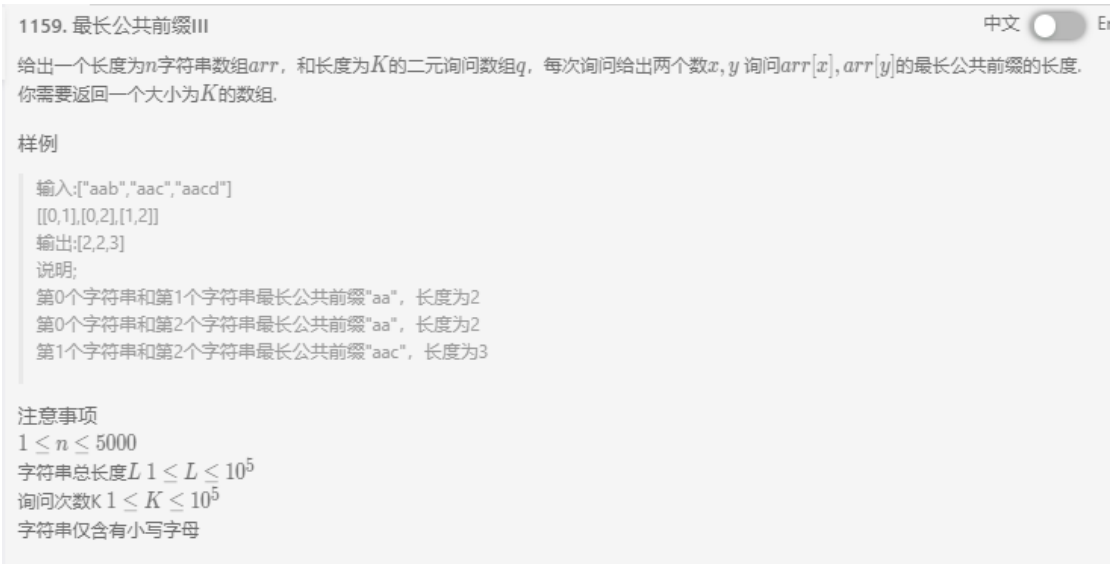
n 个字符串的长度总和sum， 1 <= sum <= 50000 .每个字符串的长度大于0(即没有空串)。

```
public class Solution {
    /**
     * @param dic: the n strings
     * @param target: the target string
     * @return: The ans
     */
    public int the_longest_common_prefix(List<String> dic, String target) {
        // write your code here
        int max = 0;
        for (String it : dic) {
            int tmp = 0;
            for (int i = 0; i < target.length(); i++){
                if (i > it.length() - 1 || target.charAt(i) != it.charAt(i)){
                    break;
                }
                tmp++;
            }
            max = Math.max(max,tmp);
        }
        return max;
    }
}
```

11、最长公共前缀三（lintcode1159）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」



1159. 最长公共前缀III

中文

给出一个长度为n字符串数组arr，和长度为K的二元询问数组q，每次询问给出两个数x, y 询问$arr[x], arr[y]$的最长公共前缀的长度.
你需要返回一个大小为K的数组.

样例

输入:["aab","aac","aacd"]
[[0,1],[0,2],[1,2]]
输出:[2,2,3]
说明:
第0个字符串和第1个字符串最长公共前缀"aa"，长度为2
第0个字符串和第2个字符串最长公共前缀"aa"，长度为2
第1个字符串和第2个字符串最长公共前缀"aac"，长度为3

注意事项
$1 \leq n \leq 5000$
字符串总长度L $1 \leq L \leq 10^5$
询问次数K $1 \leq K \leq 10^5$
字符串仅含有小写字母

```
public class Solution {
    /**
     * @param arr: string array
     * @param query: query array
     * @return: return    LCP ans array
     */
    public int[] queryLCP(String[] arr, int[][] query) {
        // write your code here
        int[] res = new int[query.length];
        for (int n = 0; n < query.length; n++) {
            if (arr[query[n][0]].length() < arr[query[n][1]].length()) {
                res[n] = longLength(arr[query[n][0]], arr[query[n][1]]);
            } else {
                res[n] = longLength(arr[query[n][1]], arr[query[n][0]]);
            }
        }
        return res;
    }
    public int longLength(String arr1, String arr2) {
        if (arr1 == null || arr2 == null) return 0;
        int ans = 0;
        for (int i = 0; i < arr1.length(); i++) {
            if (arr2.charAt(i) != arr1.charAt(i)) {
                return i;
            }
        }
        return arr1.length();
    }
}
```

12、数飞机（lintcode391）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

391. 数飞机

给出飞机的起飞和降落时间的列表，用序列 `interval` 表示. 请计算出天上同时最多有多少架飞机?

样例

样例 1:

```
输入: [(1, 10), (2, 3), (5, 8), (4, 7)]
输出: 3
解释:
第一架飞机在1时刻起飞，10时刻降落.
第二架飞机在2时刻起飞，3时刻降落.
第三架飞机在5时刻起飞，8时刻降落.
第四架飞机在4时刻起飞，7时刻降落.
在5时刻到6时刻之间，天空中有三架飞机.
```

样例 2:

```
输入: [(1, 2), (2, 3), (3, 4)]
输出: 1
解释: 降落优先于起飞.
```

注意事项

如果多架飞机降落和起飞在同一时刻，我们认为降落有优先权。

公众号：
yes的练级攻略

算法

算法一 前缀和

在开始时间位置+1架飞机，在结束时间-1架飞机，求一遍前缀和，就是对应时间飞机的数量，

前缀和算法涉及到了对时间离散化，所以这里更推荐扫描线

算法二 扫描线

扫描线，把飞机按开始时间从小到大排序，如果开始时间相同，结束时间小的在前，

扫描一遍，当扫描到开始时间，就会多一架飞机，飞机数+1，当扫描到结束时间就少一架飞机，飞机数-1

答案取扫描过程中的最大飞机数

复杂度分析

时间复杂度

算法一 前缀和

前缀和 $O(Time)$ Time表示最大时间

算法二 扫描线

扫描线 $O(NlogN)$ N是飞机数量

空间复杂度

算法一 前缀和

前缀和 $O(Time)$ Time表示最大时间

算法二 扫描线

扫描线 $O(N)$ N是飞机数量

算法一：前缀和

```
public class Solution {
    /**
     * @param airplanes: an array of meeting time airplanes
     * @return: the minimum number of conference rooms required
     */
    public int countOfAirplanes(List<Interval> airplanes) {
        int []num = new int[1110100];
        int mx = 0;
```

```
            for(int i = 0; i < airplanes.size(); i++) {
                num[airplanes.get(i).start] += 1;
                num[airplanes.get(i).end] -= 1;
                mx = Math.max(mx, airplanes.get(i).end);
            }
            int ans = 0;
            for(int i = 0; i <= mx; i++) {
                //计算前缀和
                if(i != 0) {
                    num[i] += num[i - 1];
                }
                ans = Math.max(ans, num[i]);
            }
            return ans;
        }
}
```

算法二：扫描线

```
public class Solution {
    /**
     * @param airplanes: an array of meeting time airplanes
     * @return: the minimum number of conference rooms required
     */
    static class Node {
        public int time;
        public int cost;

        public Node() {
        }
        // 时间，开始时间 cost 为 1，结束时间 cost 为-1
        public Node(int time, int cost) {
            this.time = time;
            this.cost = cost;
        }
    }
    //先按照时间升序，再按照 cost 升序排序
    static Comparator<Node> cNode = new Comparator<Node>() {
        public int compare(Node o1, Node o2) {
            if(o1.time != o2.time) {
                return o1.time - o2.time;
            }
            return o1.cost - o2.cost;
        }
    };
     public int countOfAirplanes(List<Interval> airplanes) {
```

禁止商用 公众号：yes的练级攻略

```
            //扫描线数组
            List<Node>room = new ArrayList<Node>();
            for(int i = 0; i < airplanes.size(); i++) {
                room.add(new Node(airplanes.get(i).start, 1));
                room.add(new Node(airplanes.get(i).end, -1));
            }
            //排序
            Collections.sort(room, cNode);
            int ans = 0;
            int tmp = 0;
            for(int i = 0; i < room.size(); i++) {
                tmp += room.get(i).cost;
                ans = Math.max(ans, tmp);
            }
            return ans;
    }
}
```

禁止商用

13、会议室（lintcode920）

920. 会议室

给定一系列的会议时间间隔，包括起始和结束时间[[s1,e1]，[s2,e2]，...(si < ei)，确定一个人是否可以参加所有会议。

样例

样例1

```
输入: intervals = [(0,30),(5,10),(15,20)]
输出: false
解释:
(0,30), (5,10) 和 (0,30),(15,20) 这两对会议会冲突
```

样例2

```
输入: intervals = [(5,8),(9,15)]
输出: true
解释:
这两个时间段不会冲突
```

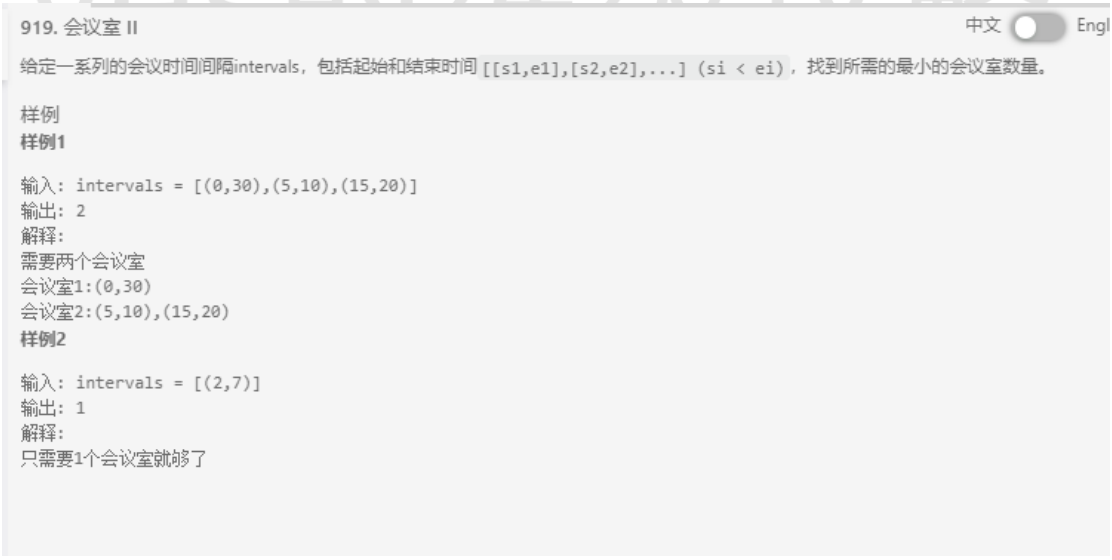
注意事项

(0,8),(8,10)在8这这一时刻不冲突

```
/**
 * Definition of Interval:
 * public class Interval {
 *     int start, end;
 *     Interval(int start, int end) {
 *         this.start = start;
 *         this.end = end;
 *     }
 * }
 */
```

```
public class Solution {
    /**
     * @param intervals: an array of meeting time intervals
     * @return: if a person could attend all meetings
     */
    public boolean canAttendMeetings(List<Interval> intervals) {
        if (intervals.size() == 0) return true;
        // Write your code here
        int[] num = new int[1110100];
        int mx = 0;
        for (int i = 0; i < intervals.size(); i++){
            num[intervals.get(i).start] ++;
            num[intervals.get(i).end] --;
            mx = Math.max(mx, intervals.get(i).end);
        }
        int ans = 0;
        for (int i = 0; i <= mx; i++){
            if (i != 0 ) {
                num[i] += num[i-1];
            }
            ans = Math.max(ans, num[i]);
        }
        return ans == 1 ? true : false;
    }
}
```

14、会议室 2（253）（lintcode919）



919. 会议室 II

中文 Engl

给定一系列的会议时间间隔intervals，包括起始和结束时间 `[[s1,e1],[s2,e2],...] (si < ei)`，找到所需的最小的会议室数量。

样例
样例1

```
输入: intervals = [(0,30),(5,10),(15,20)]
输出: 2
解释:
需要两个会议室
会议室1:(0,30)
会议室2:(5,10),(15,20)
```

样例2

```
输入: intervals = [(2,7)]
输出: 1
解释:
只需要1个会议室就够了
```

Lintcode 做法
可以转换成同一个时段最大的同时在用的会议室数目。

```
public class Solution {
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
    /**
     * @param intervals: an array of meeting time intervals
     * @return: the minimum number of conference rooms required
     */
    public int minMeetingRooms(List<Interval> intervals) {
        // Write your code here
        if (intervals.size() == 0) return 0;
        int[] num = new int[1110100];
        int mx = 0;
        for (int i = 0; i < intervals.size(); i++){
            num[intervals.get(i).start] ++;
            num[intervals.get(i).end] --;
            mx = Math.max(mx, intervals.get(i).end);
        }
        int ans = 0;
        for (int i = 0; i <= mx; i++){
            if (i != 0) num[i] += num[i-1];
            ans = Math.max(ans, num[i]);
        }
        return ans;
    }
}
```

禁止商用

253. 会议室 II

难度 中等 131 收藏 分享 切换为英文 关注 反馈

给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间 $[[s_1, e_1], [s_2, e_2], \ldots]$ $(s_i < e_i)$，为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会议安排。

示例 1:

```
输入: [[0, 30],[5, 10],[15, 20]]
输出: 2
```

示例 2:

```
输入: [[7,10],[2,4]]
输出: 1
```

Leetcode 做法

```
class Solution {
    public int minMeetingRooms(int[][] intervals) {
        int n=intervals.length;
        int[] start=new int[n];
        int[] end = new int[n];
        for (int i = 0; i < n; i++) {
            start[i]=intervals[i][0];
            end[i]=intervals[i][1];
        }
        Arrays.sort(start);
        Arrays.sort(end);
```

```
        int i=0,j=0,count=0,res=0;
        while(i<n) {
            if (start[i]<end[j]) {
                count++;
                i++;
                res=Math.max(count, res);
            }else if (start[i]>end[j]) {
                count--;
                j++;
            }else {
                i++;
                j++;
            }
        }
        return res;
    }
}
```

15、会议室 3（lintcode1897）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

描述

你有一个当前会议列表 `intervals` ，里面表明了每个会议的开始和结束时间，以及一些会议室 `rooms` 。
现在有一系列会议 `ask` 需要加入，逐个判断他们能否被安排进当前的会议列表中而不产生冲突。
一个会议室在同一时间只能进行一场会议。每个询问都相互独立。

保证输入可以被安排在 `rooms` 个会议室之中
保证任意一个会议的结束时间不大于50000
|Intervals|<=50000
|ask|<=50000
rooms<=20

您在真实的面试中是否遇到过这个题? 是

样例

例 1:

```
输入:Intervals:[[1,2],[4,5],[8,10]],rooms = 1,ask: [[2,3],[3,4]]
输出: [true,true]
解释:
对于[2,3]的询问，我们可以安排一个会议室room0。
以下是room0的会议列表:
[[1,2],[2,3],[4,5],[8,10]]
对于[3,4]的询问，我们可以安排一个会议室room0。
以下是room0的会议列表:
[[1,2],[3,4],[4,5],[8,10]]
```

例 2:

```
输入:
[[1,2],[4,5],[8,10]]
1
[[4,5],[5,6]]
输出:
[false,true]
```

```
public class Solution {
    /**
     * @param intervals: the intervals
     * @param rooms: the sum of rooms
     * @param ask: the ask
     * @return: true or false of each meeting
     */
    public boolean[] meetingRoomIII(int[][] intervals, int rooms, int[][] ask) {
        int[] sum = new int[50050];
        int[] vis = new int[50050];
        boolean[] ans = new boolean[ask.length];
        sum[0] = 0;
        int maxn = 0;
        int i;
        for (i = 0; i < intervals.length; i++) {
```

```
            vis[intervals[i][0]]++;
            vis[intervals[i][1]]--;
            maxn = Math.max(maxn, intervals[i][1]);
        }

        for (i = 0; i < ask.length; i++) {
            maxn = Math.max(maxn, ask[i][1]);
        }

        int tmp = 0;
        for (i = 1; i <= maxn; i++) {
            tmp += vis[i];
            sum[i] = (tmp < rooms) ? 0 : 1;
        }

        for (i = 1; i <= maxn; i++) {
            sum[i] += sum[i - 1];
        }

        for (i = 0; i < ask.length; i++) {
            ans[i] = (sum[ask[i][0] - 1] == sum[ask[i][1] - 1]);
        }
        return ans;
    }
}
```

16、最多可以参加的会议数目（1353）【贪心+排序】

禁止商用
公众号：
yes的练级攻略

1353. 最多可以参加的会议数目

难度 中等 86 收藏 分享 切换为英文 关注 反馈

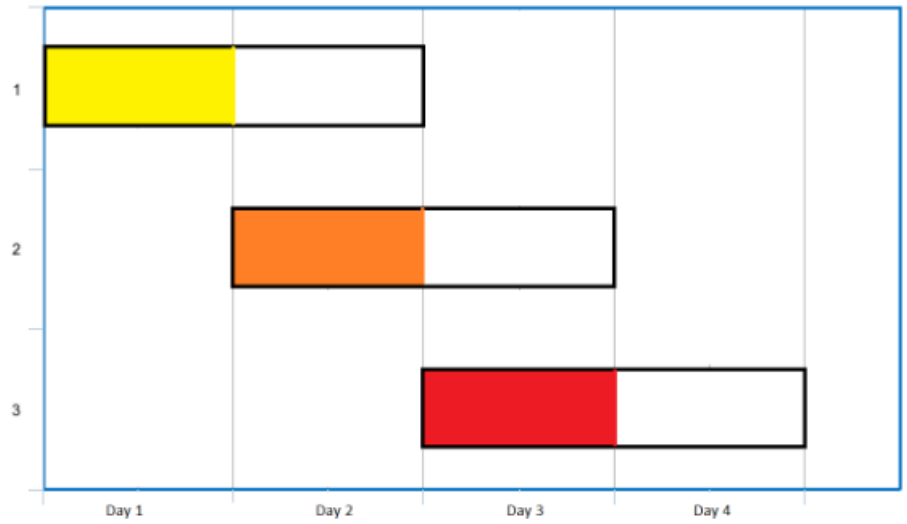
给你一个数组 `events`，其中 `events[i] = [startDay_i, endDay_i]`，表示会议 `i` 开始于 `startDay_i`，结束于 `endDay_i`。

你可以在满足 `startDay_i <= d <= endDay_i` 中的任意一天 `d` 参加会议 `i`。注意，一天只能参加一个会议。

请你返回你可以参加的 **最大** 会议数目。

示例 1：

☆ 收藏



输入： events = [[1,2],[2,3],[3,4]]
输出： 3
解释： 你可以参加所有的三个会议。
安排会议的一种方案如上图。
第 1 天参加第一个会议。
第 2 天参加第二个会议。
第 3 天参加第三个会议。

示例 2：

```
输入：events= [[1,2],[2,3],[3,4],[1,2]]
输出：4
```

示例 3：

```
输入：events = [[1,4],[4,4],[2,2],[3,4],[1,1]]
输出：4
```

示例 4：

```
输入：events = [[1,100000]]
输出：1
```

示例 5：

```
输入：events = [[1,1],[1,2],[1,3],[1,4],[1,5],[1,6],[1,7]]
输出：7
```

提示：

- `1 <= events.length <= 10^5`
- `events[i].length == 2`
- `1 <= events[i][0] <= events[i][1] <= 10^5`

https://leetcode-cn.com/problems/maximum-number-of-events-that-can-be-attended/solution/chun-cui-de-tan-xin-mei-yong-you-xian-dui-lie-dai-/

解题思路

发现这么写，比较简单。。。就按照结束时间升序排序即可，然后用个set，对区间从前往后遍历，如果这个点上没安排会议就添加进去。。对于 $100000 * [1, 100000]$这种例子不太可行，数据水（本人菜鸡大佬们轻喷

优先队列参考这位老哥的！这是本题**最好**的思路！他写的非常清楚了，我就给大家引路了！

我在这补充下我这个为什么要根据结束时间升序排序。。有的小伙伴有点模糊，假设已经安排了一些会议，我们现在再安排的会议肯定是不能和之前冲突的对吧。有这么几种选择

- 安排剩下的里面，具有最早开始时间的
- 安排剩下的里面，持续时间最短的
- 安排剩下的里面，具有最早结束时间的。

选择第一种，极端情况是开始很早，持续时间很长，选择第二种，持续时间很短，开始很晚，选择第三种最早结束时间，可以理解为最早开始时间+持续时间也最短，所以按照最早结束时间升序排序就是这么来的。

或者这么说，哪个会今天要结束了，我一定得去参加这个，因为别的会，改天能再参加啊，但是这个会，今天没了就没了，所以说这个会我一定要参加，才能保证参加的会最多（滑稽，我才不喜欢开会），这就是贪心。。

代码

思路一，时间复杂度 $O(n^2)$ 超时

改进版：在选择结束时间早的先分配的基础上，也要判断区间的可行性，也就是根据右端点选择区间分配，需要用到优先队列，时间复杂度 $O(nlogn)$

```
class Solution {
    public int maxEvents(int[][] events) {
        //首先排序：开始时间小的在前。这样是方便我们顺序遍历，把开始时间一样的都放进堆
        Arrays.sort(events, (o1, o2) -> o1[0] - o2[0]);
        //小顶堆
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        //结果、开始时间、events 下标、有多少组数据
        int res = 0, last = 1, i = 0, n = events.length;
        while (i < n || !pq.isEmpty()) {
            //将 start 相同的会议都放进堆里
            while (i < n && events[i][0] == last) {
                pq.offer(events[i++][1]);
            }
            //pop 掉当前天数之前的
            while (!pq.isEmpty() && pq.peek() < last) {
                pq.poll();
            }
            //顶上的就是俺们要参加的
            if (!pq.isEmpty()) {
                pq.poll();
                res++;
            }
            last++;
        }
        return res;
    }
}
```

17、生存人数（面试题 16.10）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.10. 生存人数

难度 中等 17 收藏 分享 切换为英文 关注 反馈

给定N个人的出生年份和死亡年份，第 i 个人的出生年份为 birth[i]，死亡年份为 death[i]，实现一个方法以计算生存人数最多的年份。

你可以假设所有人都出生于1900年至2000年（含1900和2000）之间。如果一个人在某一年的任意时期都处于生存状态，那么他们应该被纳入那一年的统计中。例如，生于1908年、死于1909年的人应当被列入1908年和1909年的计数。

如果有多个年份生存人数相同且均为最大值，输出其中最小的年份。

示例：

```
输入：
birth = {1900, 1901, 1950}
death = {1948, 1951, 2000}
输出： 1901
```

提示：

- 0 < birth.length == death.length <= 10000
- birth[i] <= death[i]

禁止商用

https://leetcode-cn.com/problems/living-people-lcci/solution/java2msyong-shu-zu-bao-cun-mei-nian-xin-zeng-ren-s/

解题思路

我们只关心每年有多少人出生，有多少人死亡，总共101年，这里用一个大小102的数组记录每年新增人数。
新增人数等于该年出生人数减去死亡人数，可以为负值。
因为某年死亡的人数实际上也计入生存人数统计，因此death[i]+1才是实际死亡年份，这也是总共101年要用一个大小102的数组来记录的原因。

```
class Solution {
    public static int maxAliveYear(int[] birth, int[] death) {
        //数组用于保存每年新增人数
        int[] r = new int[102];
        for(int i = 0; i < birth.length; i++){
            //该年出生人数
            r[birth[i] - 1900]++;
            //该年死亡人数
            r[death[i] - 1900 + 1]--;
        }
        //从 1900 年起，遍历过程中每年存活的人数
        int l = 0;
        //从 1900 年起，遍历过程中每年存活的人数的最大值
        int p = 0;
        //存活人数最多的年份
        int y = 0;
        //这里缺一个答案为 2000 的测试用例，实际上每次遍历到 i<75 就能通过验证了（笑）
        for(int i = 0; i < 101; i++){
            l += r[i];
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                if(l > p){
                    p = l;
                    y = i + 1900;
                }
            }
            return y;
        }
    }
```

18、全零子矩阵（lintcode1330）

1330. 全零子矩阵

给定一个非负二维矩阵*Matrix*，元素均大于等于0
问是否存在*c*行*d*列 元素都为0 的子矩阵

样例

输入:[[0,0,0],[1,0,0]]
2
2
输出：true
你可以选择(0,1),(0,2),(1,1),(1,2) 这四个点

注意事项
$2 \le n \le 2000$
$2 \le m \le 2000$
$2 \le c \le 500$
$2 \le d \le 500$
$0 \le Matrix[i][j] \le 10 \quad 0 \le i \le n-1 \quad 0 \le j \le m-1$
n为矩阵行数，m为矩阵列数

算法二 前缀和

二维前缀和知识

num数组是输入给出的要求和的矩阵 我们令sum数组为左上端点为(0,0) 右下端点为(i,j)的子矩阵所有元素之和 $sum[i][j] = \sum_{k1=1}^{k1=i}\sum_{k2=1}^{k2=j} num[k1][k2]$ 由容斥原理，我们可以遍历一遍预处理出前缀和数组，减去$sum[i-1][j-1]$的原因是$sum[i][j-1], sum[i-1][j]$均还有$sum[i-1][j-1]$的全部元素，所以把$sum[i-1][j-1]$加重复了一次，所以要减去 $sum[i][j] = num[i][j] + sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1]$ 如果我们想求左上端点为(lx,ly)右下端点(rx,ry)子矩阵的所有元素之和，也可以借鉴预处理的容斥原理 $\sum_{k1=lx}^{k1=rx}\sum_{k2=ly}^{k2=ry} num[k1][k2] = \$sum[rx][ry] - sum[lx-1][ry] - sum[rx][ly-1] - sum[lx-1][ly-1]\$$ 减去$sum[lx-1][ly-1]$也是因为这一部分在$[lx-1][ry]$和$[rx][ly-1]$加重复了一次

1. 本题因为题目矩阵内元素均是非负整数，若一个子矩阵的是全零的，等价于这个子矩阵求和是0，我们可以通过二维前缀和的知识快速算出这个矩阵的和是不是0 即可判断
2. 做法： 枚举矩阵左上角，通过 `c` `d` 算出整个矩阵，通过二位前缀和快速判断这个矩阵是不是全零的

复杂度分析

- 时间复杂度 n是矩阵长，m是矩阵宽 二维前缀和构造$O(nm)$ 枚举左上角$O(nm)$ 判断是否全零$O(1)$ 所以总时间复杂度$O(nm)$
- 空间复杂度 二维前缀和数组$O(nm)$

```
public class Solution {
    /**
     * @param Matrix:
     * @param c: the zero submatrix rows
     * @param d: the zero submatrix columns
     * @return: return if there exists zero submatrix
     */
    public boolean isZeroMatrix(int[][] Matrix, int c, int d) {
```

```
        // write your code here
        //行数
        int row = Matrix.length;
        //列数
        int col = Matrix[0].length;
        //  前缀和数组
        int [][] prefix_sum = new int[row + 1][col + 1];
        //计算前缀和
        for(int i = 1; i <= row; i++) {
            for(int j = 1; j <= col; j++) {
                prefix_sum[i][j] = Matrix[i - 1][j - 1];
                prefix_sum[i][j] += prefix_sum[i - 1][j] + prefix_sum[i][j - 1] - prefix_sum[i - 1][j
- 1];
            }
        }
        //枚举子矩阵左上角
        for(int i = 1; i <= row; i++) {
            for(int j = 1; j <= col; j++) {
                //计算得到子矩阵右下角
                int nx = i + c - 1;
                int ny = j + d - 1;
                //越界非法
                if(nx > row || ny > col) {
                    continue;
                }
                //得到子矩阵元素和
                int sum = prefix_sum[nx][ny] - prefix_sum[i - 1][ny] - prefix_sum[nx][j - 1] +
prefix_sum[i - 1][j - 1];
                //元素和为 0，存在全零子矩阵
                if(sum == 0) {
                    return true;
                }
            }
        }
        //不存在全零子矩阵
        return false;
    }
}
```

19、实现 Tire（前缀树）（208）

禁止商用
公众号：
yes的练级攻略

208. 实现 Trie (前缀树)

难度 中等 354 收藏 分享 切换为英文 关注 反馈

实现一个 Trie (前缀树)，包含 `insert`, `search`, 和 `startsWith` 这三个操作。

示例:

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // 返回 true
trie.search("app");     // 返回 false
trie.startsWith("app"); // 返回 true
trie.insert("app");
trie.search("app");     // 返回 true
```

说明:

- 你可以假设所有的输入都是由小写字母 `a-z` 构成的。
- 保证所有输入均为非空字符串。

https://leetcode-cn.com/problems/implement-trie-prefix-tree/solution/shi-xian-trie-qian-zhui-shu-by-leetcode/

正文

应用

Trie (发音为 "try") 或前缀树是一种树数据结构，用于检索字符串数据集中的键。这一高效的数据结构有多种应用：

1. 自动补全

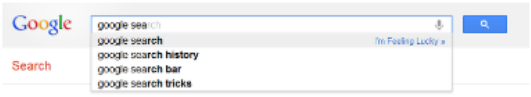


图 1. 谷歌的搜索建议

2. 拼写检查

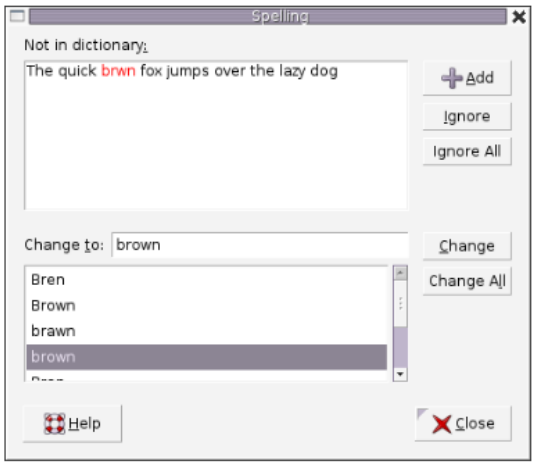


图2. 文字处理软件中的拼写检查

3. IP 路由 (最长前缀匹配)

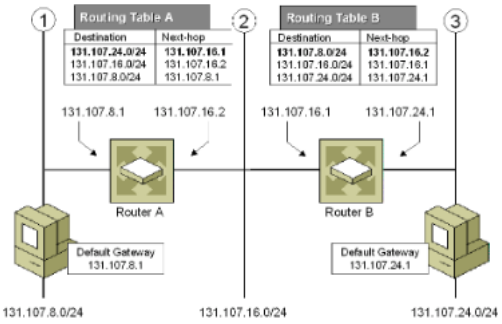


图 3. 使用Trie树的最长前缀匹配算法，Internet 协议（IP）路由中利用转发表选择路径。

4. T9 (九宫格) 打字预测

图 4. T9（九宫格输入），在 20 世纪 90 年代常用于手机输入

5. 单词游戏

图 5. Trie 树可通过剪枝搜索空间来高效解决 Boggle 单词游戏

还有其他的数据结构，如平衡树和哈希表，使我们能够在字符串数据集中搜索单词。为什么我们还需要 Trie 树呢？尽管哈希表可以在 $O(1)$ 时间内寻找键值，却无法高效的完成以下操作：

- 找到具有同一前缀的全部键值。
- 按词典序枚举字符串的数据集。

Trie 树优于哈希表的另一个理由是，随着哈希表大小增加，会出现大量的冲突，时间复杂度可能增加到 $O(n)$，其中 n 是插入的键的数量。与哈希表相比，Trie 树在存储多个具有相同前缀的键时可以使用较少的空间。此时 Trie 树只需要 $O(m)$ 的时间复杂度，其中 m 为键长。而在平衡树中查找键值需要 $O(m \log n)$ 时间复杂度。

Trie 树的结点结构

Trie 树是一个有根的树，其结点具有以下字段：。

- 最多 R 个指向子结点的链接，其中每个链接对应字母表数据集中的一个字母。
  本文中假定 R 为 26，小写拉丁字母的数量。
- 布尔字段，以指定节点是对应键的结尾还是只是键前缀。

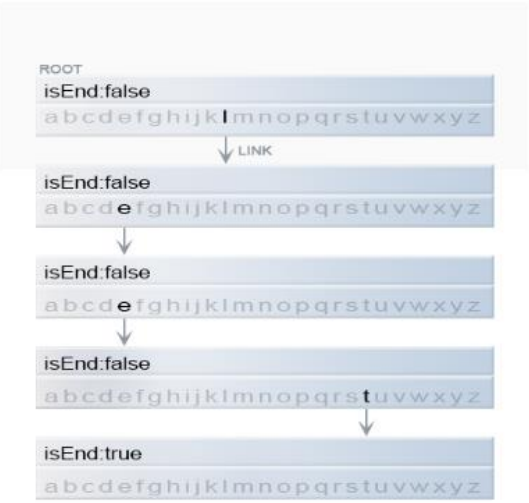


图 6. 单词 "leet" 在 Trie 树中的表示

Trie 树中最常见的两个操作是键的插入和查找。

向 Trie 树中插入键

我们通过搜索 Trie 树来插入一个键。我们从根开始搜索它对应于第一个键字符的链接。有两种情况：

- 链接存在。沿着链接移动到树的下一个子层。算法继续搜索下一个键字符。
- 链接不存在。创建一个新的节点，并将它与父节点的链接相连，该链接与当前的键字符相匹配。

重复以上步骤，直到到达键的最后一个字符，然后将当前节点标记为结束节点，算法完成。

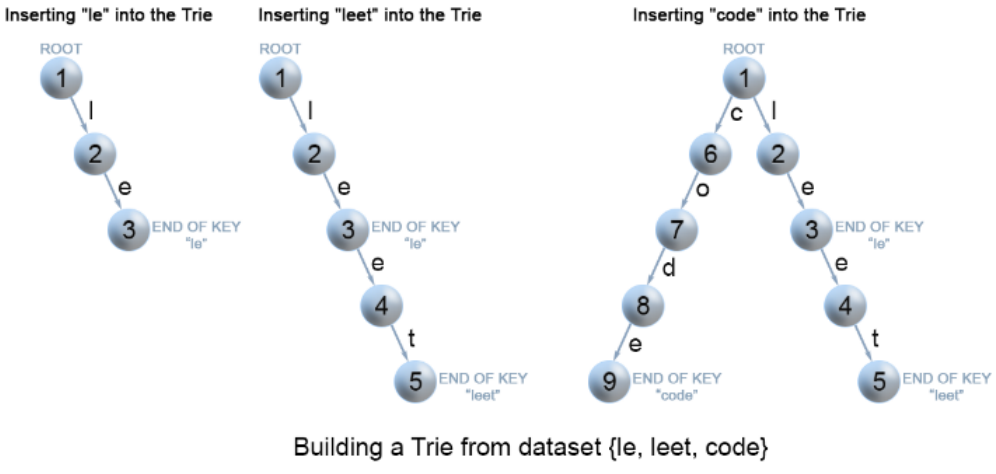


图 7. 向 Trie 树中插入键

更多硬核内容就在公众号「yes 的练级攻略」

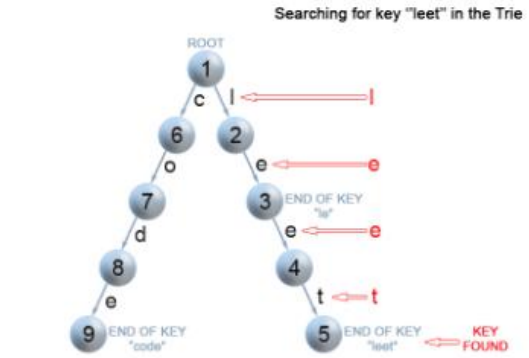
复杂度分析

- 时间复杂度：$O(m)$，其中 m 为键长。在算法的每次迭代中，我们要么检查要么创建一个节点，直到到达键尾。只需要 m 次操作。
- 空间复杂度：$O(m)$。最坏的情况下，新插入的键和 Trie 树中已有的键没有公共前缀。此时需要添加 m 个结点，使用 $O(m)$ 空间。

在 Trie 树中查找键

每个键在 trie 中表示为从根到内部节点或叶的路径。我们用第一个键字符从根开始，。检查当前节点中与键字符对应的链接。有两种情况：

- 存在链接。我们移动到该链接后面路径中的下一个节点，并继续搜索下一个键字符。
- 不存在链接。若已无键字符，且当前结点标记为 `isEnd`，则返回 true。否则有两种可能，均返回 false：
  - 还有键字符剩余，但无法跟随 Trie 树的键路径，找不到键。
  - 没有键字符剩余，但当前结点没有标记为 `isEnd`。也就是说，待查找键只是Trie树中另一个键的前缀。



Searching for a key in a Trie from dataset {le, leet, code}

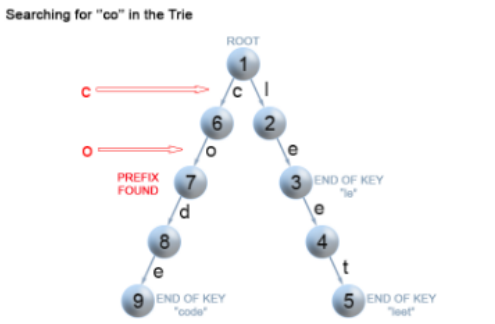
图 8. 在 Trie 树中查找键

复杂度分析

- 时间复杂度：$O(m)$。算法的每一步均搜索下一个键字符。最坏的情况下需要 m 次操作。
- 空间复杂度：$O(1)$。

查找 Trie 树中的键前缀

该方法与在 Trie 树中搜索键时使用的方法非常相似。我们从根遍历 Trie 树，直到键前缀中没有字符，或者无法用当前的键字符继续 Trie 中的路径。与上面提到的"搜索键"算法唯一的区别是，到达键前缀的末尾时，总是返回 true。我们不需要考虑当前 Trie 节点是否用 "isend" 标记，因为我们搜索的是键的前缀，而不是整个键。



Searching for a prefix in a Trie from dataset {le, leet, code}

图 9. 查找 Trie 树中的键前缀

复杂度分析

- 时间复杂度：$O(m)$。
- 空间复杂度：$O(1)$。

208 具体代码

```
class TrieNode {
    // R links to node children
    private TrieNode[] links;
    private final int R = 26;
    private boolean isEnd;
    public TrieNode() {
        links = new TrieNode[R];
    }
    public boolean containsKey(char ch) {
        return links[ch -'a'] != null;
    }
    public TrieNode get(char ch) {
        return links[ch -'a'];
    }
    public void put(char ch, TrieNode node) {
        links[ch -'a'] = node;
    }
    public void setEnd() {
        isEnd = true;
    }
    public boolean isEnd() {
        return isEnd;
    }
}
public class Trie {
    private TrieNode root;
    /** Initialize your data structure here. */
    public Trie() {
        root = new TrieNode();
    }
    /** Inserts a word into the trie. */
    public void insert(String word) {
        TrieNode node = root;
        for (int i = 0; i < word.length(); i++) {
            char currentChar = word.charAt(i);
            if (!node.containsKey(currentChar)) {
                node.put(currentChar, new TrieNode());
            }
            node = node.get(currentChar);
```

```
            }
            node.setEnd();
        }
        /** Returns if the word is in the trie. */
        public boolean search(String word) {
            TrieNode node = searchPrefix(word);
            return node != null && node.isEnd();
        }
            // search a prefix or whole key in trie and
        // returns the node where search ends
        private TrieNode searchPrefix(String word) {
            TrieNode node = root;
            for (int i = 0; i < word.length(); i++) {
               char curLetter = word.charAt(i);
               if (node.containsKey(curLetter)) {
                   node = node.get(curLetter);
               } else {
                   return null;
               }
            }
            return node;
        }
        /** Returns if there is any word in the trie that starts with the g
iven prefix. */
        public boolean startsWith(String prefix) {
            TrieNode node = searchPrefix(prefix);
            return node != null;
        }
}

/**
 * Your Trie object will be instantiated and called as such:
 * Trie obj = new Trie();
 * obj.insert(word);
 * boolean param_2 = obj.search(word);
 * boolean param_3 = obj.startsWith(prefix);
 */
```

20、添加与搜索单词（211）【只能用前缀树，其他方法超时】

211. 添加与搜索单词 - 数据结构设计

难度 中等 140 收藏 分享 切换为英文 关注 反馈

设计一个支持以下两种操作的数据结构：

```
void addWord(word)
bool search(word)
```

search(word) 可以搜索文字或正则表达式字符串，字符串只包含字母 . 或 a-z 。 . 可以表示任何一个字母。

示例:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

说明:

你可以假设所有单词都是由小写字母 a-z 组成的。

禁止商用

https://leetcode-cn.com/problems/add-and-search-word-data-structure-design/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by--43/

```
class WordDictionary {
    class TrieNode {
        TrieNode[] children;
        boolean flag;

        public TrieNode() {
            children = new TrieNode[26];
            flag = false;
            for (int i = 0; i < 26; i++) {
                children[i] = null;
            }
        }

    }
    TrieNode root;

    /** Initialize your data structure here. */
    public WordDictionary() {
        root = new TrieNode();
    }

    /** Adds a word into the data structure. */
    public void addWord(String word) {
        char[] array = word.toCharArray();
```

```
        TrieNode cur = root;
        for (int i = 0; i < array.length; i++) {
            // 当前孩子是否存在
            if (cur.children[array[i] - 'a'] == null) {
                cur.children[array[i] - 'a'] = new TrieNode();
            }
            cur = cur.children[array[i] - 'a'];
        }
        // 当前节点代表结束
        cur.flag = true;
    }

    /**
     * Returns if the word is in the data structure. A word could conta
in the
     * dot character '.' to represent any one letter.
     */
    public boolean search(String word) {
        return searchHelp(word, root);
    }

    private boolean searchHelp(String word, TrieNode root) {
        char[] array = word.toCharArray();
        TrieNode cur = root;
        for (int i = 0; i < array.length; i++) {
            // 对于 . , 递归的判断所有不为空的孩子
            if(array[i] == '.'){
                for(int j = 0;j < 26; j++){
                    if(cur.children[j] != null){
                        if(searchHelp(word.substring(i + 1),cur.childre
n[j])){
                            return true;
                        }
                    }
                }
                return false;
            }
            // 不含有当前节点
            if (cur.children[array[i] - 'a'] == null) {
                return false;
            }
            cur = cur.children[array[i] - 'a'];
        }
                                        // 当前节点是否为是某个单词的结束
```

```
            return cur.flag;
        }
    }

/**
 * Your WordDictionary object will be instantiated and called as such:
 * WordDictionary obj = new WordDictionary();
 * obj.addWord(word);
 * boolean param_2 = obj.search(word);
 */
```

21、字典序的第 k 小数字（440）

440. 字典序的第K小数字

难度 困难 147 收藏 分享 切换为英文 关注 反馈

给定整数 `n` 和 `k`，找到 `1` 到 `n` 中字典序第 `k` 小的数字。

注意：$1 \le k \le n \le 10^9$。

示例：

输入：
`n: 13  k: 2`

输出：
`10`

解释：
字典序的排列是 `[1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9]`，所以第二小的数字是 `10`。

https://leetcode-cn.com/problems/k-th-smallest-in-lexicographical-order/solution/an-zi-dian-xu-chu-li-qian-zhui-java-by-benboby/

题目

给定整数 n 和 k，找到 1 到 n 中字典序第 k 小的数字。

注意：1 ≤ k ≤ n ≤ 109。

Solution

我们可以把每个字符看作是字符串，这样我们只需要考虑前缀即可。显然最小的字符串一定是0，然后是所有以1开头的数。。。

我们只需要枚举所有首位数字 $1-9$，计算每个前缀在区间 $[0,n]$ 内有多少个数就好了，当累加和超过 n 时，说明答案一定是以这个首位数字开头的，然后向下枚举。比如当以得知答案以3开头时，扣除3本身，然后向下枚举前缀30,31重复上述操作。。。直到 k 为0。

```
class Solution {
    public int findKthNumber(int n, int k) {
        int pre = 1;
```

```
            k--; // 扣除数字0
            while (k > 0) {
                int cur = count(n, pre, pre + 1);
                if (k >= cur) {// 说明不在这个前缀区间里
                    pre++;  // 找下一个字典序前缀
                    k -= cur;  // 扣除这个前缀的所有数
                } else {  // 说明答案是这个前缀
                    pre *= 10;  // 往下找
                    k--; // 扣除当前这个数
                }
            }
            return pre;
        }
        public int count(int n, long x, long y) {
            int sum = 0;
            while (x <= n) {
                sum += Math.min(n + 1, y) - x; // 如n=15,则sum+=min(16,20)-10
                x *= 10;
                y *= 10;
            }
            return sum;
        }
}
```

22、字典序排数（386）

386. 字典序排数

难度 中等　115　收藏　分享　切换为英文　关注　反馈

给定一个整数 *n*, 返回从 *1* 到 *n* 的字典顺序。

例如，

给定 *n* =1 3，返回 [1,10,11,12,13,2,3,4,5,6,7,8,9] 。

请尽可能的优化算法的时间复杂度和空间复杂度。 输入的数据 *n* 小于等于 5,000,000。

方法 1：用上一题的解法进行递归

```
class Solution {
    public List<Integer> lexicalOrder(int n) {
        List<Integer> res = new ArrayList<>();
        for (int i = 1; i <= n; i++) {
            res.add(findKthNumber(n,i));
        }
        return res;
    }
```

```
    public int findKthNumber(int n, int k) {
        int pre = 1;
        k--; // 扣除数字0
        while (k > 0) {
            int cur = count(n, pre, pre + 1);
            if (k >= cur) {// 说明不在这个前缀区间里
                pre++;  // 找下一个字典序前缀
                k -= cur;  // 扣除这个前缀的所有数
            } else {  // 说明答案是这个前缀
                pre *= 10;  // 往下找
                k--; // 扣除当前这个数
            }
        }
        return pre;
    }
    public int count(int n, long x, long y) {
        int sum = 0;
        while (x <= n) {
            sum += Math.min(n + 1, y) - x; // 如n=15,则sum+=min(16,20)-10
            x *= 10;
            y *= 10;
        }
        return sum;
    }
}
```

https://leetcode-cn.com/problems/lexicographical-numbers/solution/ncha-shu-de-qian-xu-bian-li-by-huhudaya/

解题思路

没啥好说的，N叉树前序遍历一下。二叉树怎么弄的，N叉树一样

方法 2：N 叉树前序遍历

```
class Solution {
    List<Integer> res = new ArrayList<>();
    public List<Integer> lexicalOrder(int n) {
        for (int i = 1; i < 10; i ++){
            dfs(n, i);
        }
        return res;
    }
    private void dfs(int n, int cur){
        if (cur > n) return;
        res.add(cur);
        for (int i = 0; i < 10; i ++) {
            dfs(n, cur * 10 + i);
        }
```

```
    }
}
```

23、字母与数字（面试题 17.05）

面试题 17.05. 字母与数字

难度 中等 20 收藏 分享 切换为英文 关注 反馈

给定一个放有字符和数字的数组，找到最长的子数组，且包含的字符和数字的个数相同。

返回该子数组，若存在多个最长子数组，返回左端点最小的。若不存在这样的数组，返回一个空数组。

示例 1:

```
输入：["A","1","B","C","D","2","3","4","E","5","F","G","6","7","H","I","J","K","L","M"]

输出：["A","1","B","C","D","2","3","4","E","5","F","G","6","7"]
```

示例 2:

```
输入：["A","A"]

输出：[]
```

提示:

- `array.length <= 100000`

https://leetcode-cn.com/problems/find-longest-subarray-lcci/solution/shu-zi-1-zi-mu-1-by-gfu/

思路

数字看成 `-1`，**字母**看成 `1`，再计算**前缀和**。
前缀和 相同则计算下标的差值。

技巧：使用 `int[]` 数组 `memo` 存储 该**前缀和** 第 `1` 次出现时 的**下标**。
为何是**第1次出现时**的下标?
因为要求**最长**子数组。

比如:
`["A","1","B","C","D","2","3","4","E","5","F","G","6","7","H","I","J","K","L","M"]`

转化为 `[1, -1, 1, 1, 1, -1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1]`

再转为**前缀和形式** `[1, 0, 1, 2, 3, 2, 1, 0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 4, 5, 6]` 。

前缀和相同是什么意思呢?

比如看**加粗**的 `2` 个 `1` ：[**1**, 0, **1**, 2, 3, 2, 1, 0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 4, 5, 6]

说明这 `2` 个 `1` 的**下标**所构成的**区间内**的**数字和**为 `0` ，而**数字和为0**说明有**相同个数**的 `1` 和 `-1` ，即**相同个数**的**字母**和**数字**。

具体来说：第 `1` 个加粗的 `1` 下标为 `0` ，第 `2` 个加粗的 `1` 下标为 `2` ，构成**区间** `(0, 2]` (注意：是**半开半闭**)，
区间 `(0, 2]` 内元素的和即**下标为 `1` 和下标为 `2` 的元素之和**，[1, **-1**, **1**, 1, 1, -1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 1, 1, 1, 1, 1]
很明显 `-1 + 1 = 0`

理解了这一点，剩余的工作就是找**前缀和相同**，且**相隔最远**的2个元素。

什么才是相隔最远?

[**1**, 0, **1**, 2, 3, 2, **1**, 0, **1**, 0, **1**, 2, **1**, 0, **1**, 2, 3, 4, 5, 6]

单只看**前缀和为1**的情况，毫无疑问，前缀和为 `1` 时，相隔最远的肯定是**最左边**的 `1` 和**最右边**的 `1` .

因此我们只需要记录前缀和为1**第一次出现时**的下标即可(也就是记录了最左边的 `1` 的位置)，以后再遇到前缀和为 `1` 时，只需要将其下标 `-` 所记录的最左边的 `1` 的位置 即可。

至于 `memo` 数组大小为何初始化为 `(len << 1) + 1` ，考虑最极端的情况，要么全为 `-1` ，要么全为 `1` ，对应前缀和为 `-array.length` 和 `array.length` 。

特别注意:
当**前缀和为0**时，假设其下标为 `n` ，则说明区间 `[0, n]` 内所有元素的和为 `0` ，**区间长度**为 `n + 1` 。
因此将 `memo[0 + array.length]` 的值设为 `-1` ，因为 `n - (-1) = n + 1` 。

同时由于数组下标不能为负数，因此需要映射处理:
假设数组有 `2` 个元素，边界值为 `-2` 和 `2` ，即 `[-2, -1, 0, 1, 2]` 。
将 `[-2, -1, 0, 1, 2]` 集体右移 `nums.length` 个单位，映射为 `[0, 1, 2, 3, 4]` 。

```
class Solution {
    public String[] findLongestSubarray(String[] array) {
        int len = array.length;
        int[] memo = new int[(len << 1) + 1];
        Arrays.fill(memo, -2);
        memo[len] = -1;
        int res = 0, count = 0, begin = 0, end = 0;
        for (int i = 0; i < len; ++i) {
            boolean is_num = true;
            for (char c : array[i].toCharArray())
                if (c < '0' || c > '9') {
                    is_num = false;
                    break;
                }
            count += is_num ? -1 : 1;
            //未曾见过该前缀和(即 memo 数组中没有记录)
            if (memo[count + len] <= -2)
                memo[count + len] = i;  //记录该前缀和的下标
            //曾见过该前缀和(即 memo 数组中已有记录)
            else if (i - memo[count + len] > res) {
                begin = memo[count + len] + 1;
                end = i + 1;
                res = i - memo[count + len];
            }
        }
        return Arrays.copyOfRange(array, begin, end);
    }
}
```

六、背包问题

AcWing 题库链接：https://www.acwing.com/problem/

背包九讲学习视频（b 站的大雪菜）：

https://www.bilibili.com/video/BV1qt411Z7nE?from=search&seid=13053729582382786869

1. 0-1 背包

描述：有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

- 01 背包问题（awing 2）【与背包问题 2 一样】

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$

输入样例

```
4 5
1 2
2 4
3 4
4 5
```

输出样例：

```
8
```

方法 1：适用于数据量比较小的时候

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
```

```
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w));
    }
    private static int maxValue(int N, int V, int[] v, int[] w){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        for (int i = 0; i < N; i++){
            for (int j = V; j >= v[i]; j--){
                dp[j] = Math.max(dp[j], dp[j-v[i]] + w[i]);
            }
        }
        return dp[V];
    }
}
```

方法 2：适用于数据量比较大的时候【二进制】

```
import java.util.Scanner;
import java.util.ArrayList;
public class Main{
    static class ValueHolder{
        int v,w;
        public ValueHolder(int v, int w) {
            this.v = v;
            this.w = w;
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w));
    }
    private static int maxValue(int N, int V, int[] v, int[] w){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        ArrayList<ValueHolder> list = new ArrayList<>();
        for (int i = 0; i < N; i++){
```

禁止商用 公众号：yes的练级攻略

```
            list.add(new ValueHolder(v[i], w[i]));
        }
        for (ValueHolder holder : list){
            for (int j = V; j >= holder.v; j--){
                dp[j] = Math.max(dp[j], dp[j - holder.v] + holder.w);
            }
        }
        return dp[V];
    }
}
```

- 背包问题（lintcode92）

92. 背包问题

在n个物品中挑选若干物品装入背包，最多能装多满？假设背包的大小为m，每个物品的大小为A[i]

样例
样例 1:
输入: [3,4,8,5], backpack size=10
输出: 9

样例 2:
输入: [2,3,5,7], backpack size=12
输出: 12

挑战
O(n x m) 的时间复杂度 and O(m) 空间复杂度
如果不知道如何优化空间O(n x m) 的空间复杂度也可以通过.

注意事项
你不可以将物品进行切割。

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @return: The maximum size
     */
    public int backPack(int m, int[] A) {
        int[] dp = new int[m + 1];//动态规划矩阵
        for (int i = 0; i < A.length; i++){
            for (int j = m; j >= A[i]; j--){
                dp[j] = Math.max(dp[j], dp[j - A[i]] + A[i]);
            }
        }
        return dp[m];
    }
}
```

- 背包问题 2（lintcode125）

更多硬核内容就在公众号「yes 的练级攻略」

描述

中文 English

有 `n` 个物品和一个大小为 `m` 的背包. 给定数组 `A` 表示每个物品的大小和数组 `V` 表示每个物品的价值.

问最多能装入背包的总价值是多大?

1. `A[i], V[i], n, m` 均为整数
2. 你不能将物品进行切分
3. 你所挑选的要装入背包的物品的总大小不能超过 `m`
4. 每个物品只能取一次

您在真实的面试中是否遇到过这个题? 是 题目纠错

样例

样例 1:

```
输入: m = 10, A = [2, 3, 5, 7], V = [1, 5, 2, 4]
输出: 9
解释: 装入 A[1] 和 A[3] 可以得到最大价值, V[1] + V[3] = 9
```

样例 2:

```
输入: m = 10, A = [2, 3, 8], V = [2, 5, 8]
输出: 10
解释: 装入 A[0] 和 A[2] 可以得到最大价值, V[0] + V[2] = 10
```

挑战

O(nm) 空间复杂度可以通过, 不过你可以尝试 O(m) 空间复杂度吗?

```
public class Solution {
    /**
     * @param m: An integer m denotes the size of a backpack
     * @param A: Given n items with size A[i]
     * @param V: Given n items with value V[i]
     * @return: The maximum value
     */
    public int backPackII(int m, int[] A, int[] V) {
        if (m == 0 || A.length == 0) return 0;
        int[] dp = new int[m + 1];//动态规划矩阵
        for (int i = 0; i < A.length; i++){
            for (int j = m; j >= A[i]; j--){
                dp[j] = Math.max(dp[j], dp[j - A[i]] + V[i]);
            }
        }
        return dp[m];
    }
}
```

- 背包问题 9（lintcode 800）

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

800. 背包问题 IX

中文 English

你总共有 n 万元，希望申请国外的大学，要申请的话需要交一定的申请费用，给出每个大学的申请费用以及你得到这个大学offer的成功概率，大学的数量是 m 。如果经济条件允许，你可以申请多所大学。找到获得至少一份工作的最高可能性。

样例
样例 1:

```
输入：
        n = 10
        prices = [4,4,5]
        probability = [0.1,0.2,0.3]
输出：  0.440

解释：
选择第2和第3个学校。
```

样例 2:

```
输入：
        n = 10
        prices = [4,5,6]
        probability = [0.1,0.2,0.3]
输出：  0.370

解释：
选择第1和第3个学校。
```

注意事项
0<=n<=10000,0<=m<=10000

https://www.jiuzhang.com/solution/backpack-ix/#tag-lang-java

题目描述

n 表示你有多少万元；prices `i` 表示申请第 `i` 个学校所需的费用；probability `i` 表示申请第 `i` 个学校成功的可能写；求花费不超过n万元时搜能申请到至少一个大学offer的最大可能性。

算法思路

- 根据题目描述得知每所学校只能申请一次
- 根据题意描述我们可以得知这题是一题01背包变形
- 至少一所大学录取，那么就有 1、2、.....、m所录取的可能性，这种情况太多了，所以我们计算一所都没有录取的最小概率，那么 `1 - 一所都未录取的最小概率` 就是至少有一所大学录取的最大概率
- 我们定义状态 `dp[i]` 来表示花费 `i` 万元申请学校一所都没录取的概率
- 状态转移方程为：`dp[j] = min(dp[j],dp[j-prices[i]] * 第 i 所大学未录取的概率)`

代码思路

1. 定义一个二维数组 `dp[i]` 来表示花费 `i` 万元申请学校一所都没录取的概率
2. 初始化dp数组，先假设录取概率都为0
3. 操作probability数组，变成未录取的概率 `probability[i] = 1.0 - probability[i]`
4. 第一层循环枚举 `i` 表示申请前 `i` 个学校
5. 第二层循环枚举 `j` 表示花费 `j` 万元
6. 根据算法思路中的状态转移方程来实现 `dp[j] = min(dp[j],dp[j-prices[i]] * probability[i])`
7. `dp[n]` 表示花费 n 万元一所学校都未录取的最小概率
8. return `1 - dp[n]`

复杂度分析

- 空间复杂度：$O(N)$
- 时间复杂度：$O(N * M)$

```
public class Solution {
    /**
     * @param n: Your money
     * @param prices: Cost of each university application
     * @param probability: Probability of getting the University's offer
     * @return: the    highest probability
```

```
     */
    public double backpackIX(int n, int[] prices, double[] probability) {
        //正难则反。 计算一个 offer 都收不到的概率，然后减掉。 使用 0-1 背包来计算收到 0 个 offer 的最小概率。
        if (n == 0 || prices.length == 0) return 0.0;
        double[] dp = new double[n+1];
        for (int i = 0; i <= n; i++) dp[i] = 1.0;
        for (int i = 0; i < probability.length; i++) {
            probability[i] = 1 - probability[i];
        }
        dp[0] = 1.0;
        for (int i = 0; i < prices.length; i++){
            for (int j = n; j >= prices[i]; j--) {
                dp[j] = Math.min(dp[j], dp[j - prices[i]] * probability[i]);
            }
        }
        return 1 - dp[n];
    }
}
```

禁止商用

- 分割等和子集（416）【0-1 背包】

416. 分割等和子集

难度 中等 319 收藏 分享 切换为英文 关注 反馈

给定一个**只包含正整数**的**非空**数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

注意:

1. 每个数组中的元素不会超过 100
2. 数组的大小不会超过 200

示例 1:

```
输入: [1, 5, 11, 5]

输出: true

解释: 数组可以分割成 [1, 5, 5] 和 [11].
```

示例 2:

```
输入: [1, 2, 3, 5]

输出: false

解释: 数组不能分割成两个元素和相等的子集.
```

https://leetcode-cn.com/problems/partition-equal-subset-sum/solution/0-1-bei-bao-wen-ti-xiang-jie-zhen-dui-ben-ti-de-yo/

关于背包问题的介绍，大家可以在互联网上搜索「背包九讲」进行学习，其中「0-1」背包问题是这些问题的基础。「力扣」上涉及的背包问题有「0-1」背包问题、完全背包问题、多重背包问题。

本题解有些地方使用了「0-1」背包问题的描述，因此会不加解释的使用「背包」、「容量」这样的名词。

说明：这里感谢很多朋友在这篇题解下提出的建议，对我的启发很大。本题解的阅读建议是：先浏览代码，然后再看代码之前的分析，能更有效理解知识点和整个问题的思考路径。题解后也增加了「总结」，供大家参考。

转换为「0 - 1」背包问题

这道问题是我学习「背包」问题的入门问题，做这道题需要做一个等价转换：是否可以从输入数组中挑选出一些正整数，使得这些数的和 **等于** 整个数组元素的和的一半。很坦白地说，如果不是老师告诉我可以这样想，我很难想出来。容易知道：数组的和一定得是偶数。

本题与 0-1 背包问题有一个很大的不同，即：

- 0-1 背包问题选取的物品的容积总量 **不能超过** 规定的总量；
- 本题选取的数字之和需要 **恰好等于** 规定的和的一半。

这一点区别，决定了在初始化的时候，所有的值应该初始化为 `false`。（《背包九讲》的作者在介绍「0-1 背包」问题的时候，有强调过这点区别。）

「0 - 1」背包问题的解决策略

作为「0-1 背包问题」，它的特点是：「每个数只能用一次」。解决的基本思路是：**物品一个一个选，容量也一点一点增加去考虑**，这一点是「动态规划」的思想，特别重要。
在实际生活中，我们也是这样做的，一个一个地尝试把候选物品放入「背包」，通过比较得出一个物品要不要拿走。

具体做法是：画一个 `len` 行，`target + 1` 列的表格。这里 `len` 是物品的个数，`target` 是背包的容量。`len` 行表示一个一个物品考虑，`target + 1` 多出来的那 `1` 列，表示背包容量从 `0` 开始考虑。很多时候，我们需要考虑这个容量为 `0` 的数值。

状态与状态转移方程

- 状态定义：`dp[i][j]` 表示从数组的 `[0, i]` 这个子区间内挑选一些正整数，每个数只能用一次，使得这些数的和**恰好等于** `j`。
- 状态转移方程：很多时候，状态转移方程思考的角度是「分类讨论」，对于「0-1 背包问题」而言就是「当前考虑到的数字选与不选」。
  - 不选择 `nums[i]`，如果在 `[0, i - 1]` 这个子区间内已经有一部分元素，使得它们的和为 `j`，那么 `dp[i][j] = true`；
  - 选择 `nums[i]`，如果在 `[0, i - 1]` 这个子区间内就得找到一部分元素，使得它们的和为 `j - nums[i]`。

状态转移方程是：

```
dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i]]
```

一般写出状态转移方程以后，就需要考虑初始化条件。

- `j - nums[i]` 作为数组的下标，一定得保证大于等于 `0`，因此 `nums[i] <= j`；
- 注意到一种非常特殊的情况：`j` 恰好等于 `nums[i]`，即单独 `nums[j]` 这个数恰好等于此时「背包的容积」`j`，这也是符合题意的。

因此完整的状态转移方程是：

$$\mathrm{dp}[i][j] = \begin{cases} \mathrm{dp}[i-1][j], & \text{至少是这个答案，如果 } \mathrm{dp}[i-1][j] \text{ 为真，直接计算下一个状态} \\ \text{true}, & \text{nums[i]} = \text{j} \\ \mathrm{dp}[i-1][j-nums[i]]. & \text{nums[i]} < \text{j} \end{cases}$$

说明：虽然写成花括号，但是它们的关系是 **或者** 。

- 初始化：`dp[0][0] = false`，因为是正整数，当然凑不出和为 0；
- 输出：`dp[len - 1][target]`，这里 `len` 表示数组的长度，`target` 是数组的元素之和（必须是偶数）的一半。

复杂度分析：

- 时间复杂度：$O(NC)$：这里 N 是数组元素的个数，C 是数组元素的和的一半。
- 空间复杂度：$O(NC)$。

解释设置 `dp[0][0] = true` 的合理性（重点）

修改状态数组初始化的定义：`dp[0][0] = true`。考虑容量为 0 的时候，即 `dp[i][0]`。按照本意来说，应该设置为 `false`，但是注意到状态转移方程（代码中）：

```
dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i]];
```

当 `j - nums[i] == 0` 成立的时候，根据上面分析，就说明单独的 `nums[i]` 这个数就恰好能够在被分割为单独的一组，其余的数分割成为另外一组。因此，我们把初始化的 `dp[i][0]` 设置成为 `true` 是没有问题的。

注意：观察状态转移方程，`or` 的结果只要为真，表格 **这一列** 下面所有的值都为真。因此在填表的时候，只要表格的最后一列是 `true`，代码就可以结束，直接返回 `true`。

考虑空间优化（重要）

说明：这个技巧很常见、很基础，请一定要掌握。

「0-1 背包问题」常规优化：「状态数组」从二维降到一维，减少空间复杂度。

- 在「填表格」的时候，当前行只参考了上一行的值，因此状态数组可以只设置 2 行，使用「滚动数组」的技巧「填表格」即可；
- 实际上，在「滚动数组」的基础上还可以优化，在「填表格」的时候，**当前行总是参考了它上面一行「头顶上」那个位置和「左上角」某个位置的值。**因此，我们可以只开一个一维数组，从后向前依次填表即可。

> 友情提示：这一点在刚开始学习的时候，可能会觉得很奇怪。理解的办法是：拿题目中的示例，画一个表格，自己模拟一遍程序是如何「填表」的行为，就很清楚为什么状态数组降到 1 行的时候，需要「从后前向」填表。

- 「从后向前」写的过程中，一旦 `nums[i] <= j` 不满足，可以马上退出当前循环，因为后面的 `j` 的值肯定越来越小，没有必要继续做判断，直接进入外层循环的下一层。相当于也是一个剪枝，这一点是「从前向后」填表所不具备的。

说明：如果对空间优化技巧还有疑惑的朋友，本题解下的精选评论也解释了如何理解这个空间优化的技巧，请大家前往观看。

参考代码 3：只展示了使用一维表格，并且「从后向前」填表格的代码。

考虑空间优化（重要）

说明：这个技巧很常见、很基础，请一定要掌握。

「0-1 背包问题」常规优化：「状态数组」从二维降到一维，减少空间复杂度。

- 在「填表格」的时候，当前行只参考了上一行的值，因此状态数组可以只设置 2 行，使用「滚动数组」的技巧「填表格」即可；
- 实际上，在「滚动数组」的基础上还可以优化，在「填表格」的时候，**当前行总是参考了它上面一行「头顶上」那个位置和「左上角」某个位置的值**。因此，我们可以只开一个一维数组，从后向前依次填表即可。

> 友情提示：这一点在刚开始学习的时候，可能会觉得很奇怪。理解的办法是：拿题目中的示例，画一个表格，自己模拟一遍程序是如何「填表」的行为，就很清楚为什么状态数组降到 1 行的时候，需要「从后前向」填表。

- 「从后向前」写的过程中，一旦 `nums[i] <= j` 不满足，可以马上退出当前循环，因为后面的 `j` 的值肯定越来越小，没有必要继续做判断，直接进入外层循环的下一层。相当于也是一个剪枝，这一点是「从前向后」填表所不具备的。

说明：如果对空间优化技巧还有疑惑的朋友，本题解下的精选评论也解释了如何理解这个空间优化的技巧，请大家前往观看。

参考代码 3：只展示了使用一维表格，并且「从后向前」填表格的代码。

总结

「0-1 背包」问题是一类非常重要的动态规划问题，一开始学习的时候，可能会觉得比较陌生。建议动笔计算，手动模拟填表的过程，其实就是画表格。这个过程非常重要，自己动手填过表，更能加深体会程序是如何执行的，也能更好地理解「空间优化」技巧的思路和好处。

```
public class Solution {

    public boolean canPartition(int[] nums) {
        int len = nums.length;
        if (len == 0) {
            return false;
        }

        int sum = 0;
        for (int num : nums) {
            sum += num;
```

```
        }
        // 特判：如果是奇数，就不符合要求
        if ((sum & 1) == 1) {
            return false;
        }

        int target = sum / 2;

        boolean[] dp = new boolean[target + 1];
        dp[0] = true;
        // 先填表格第 0 行，第 1 个数只能让容积为它自己的背包恰好装满
        if (nums[0] <= target) {
            dp[nums[0]] = true;
        }

        for (int i = 1; i < len; i++) {
            for (int j = target; nums[i] <= j; j--) {
                //剪枝操作，加快运行速度，可以删掉。
                if (dp[target]) {
                    return true;
                }

                dp[j] = dp[j] || dp[j - nums[i]];
            }
        }
        return dp[target];
    }
}
```

- 最小划分（lintcode 724）【两个子集中的和的差的绝对值最小】

724. 最小划分

中文 English

给出一个正整数数组，写一个程序把这个整数数组分成S1跟S2两部分，使S1中的和跟S2中的和的差的绝对值最小。换句话讲，如果有一个一个整数数组 S 有 n 个数，如果Subset1有 m 个数，Subset2必须有 n-m 个数并且 abs(sum(Subset1) - sum(Subset2)) 应该最小

样例

样例1

输入：nums = [1, 6, 11, 5]
输出：1
解释：
Subset1 = [1, 5, 6]，和是12
Subset2 = [11]，和是11
abs(11 - 12) = 1

样例2

输入：nums = [1, 2, 3, 4]
输出：0
解释：
Subset1 = [1, 4]，和是 5
Subset2 = [2, 3]，和是5
abs(5 - 5) = 0

更多硬核内容就在公众号「yes 的练级攻略」

https://www.jiuzhang.com/solution/minimum-partition/#tag-lang-java

```
public class Solution {
    /**
     * @param nums: the given array
     * @return: the minimum difference between their sums
     */
    public int findMin(int[] nums) {
        int sum = 0;
        for (int num : nums) {
            sum += num;
        }
        boolean[] dp = new boolean[sum+1];
        dp[0] = true;
        int min = Integer.MAX_VALUE;
        for (int num: nums){
            for (int j = sum; j >= num; j--){
                dp[j] |= dp[j - num];
                if (dp[j]) {
                    int diff = Math.abs(sum - j * 2);
                    min = Math.min(min, diff);
                }
            }
        }
        return min;
    }
}
```

- 目标和（494）【0-1 背包，与上一题类似】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

494. 目标和

难度 中等 308 收藏 分享 切换为英文 关注 反馈

给定一个非负整数数组，a1, a2, ..., an, 和一个目标数，S。现在你有两个符号 + 和 -。对于数组中的任意一个整数，你都可以从 + 或 - 中选择一个符号添加在前面。

返回可以使最终数组和为目标数 S 的所有添加符号的方法数。

示例：

```
输入：nums: [1, 1, 1, 1, 1], S: 3
输出：5
解释：

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3

一共有5种方法让最终目标和为3。
```

提示：

- 数组非空，且长度不会超过 20 。
- 初始的数组的和不会超过 1000 。
- 保证返回的最终结果能被 32 位整数存下。

https://leetcode-cn.com/problems/target-sum/solution/bei-bao-da-qia-ti-20-1bei-bao-qia-hao-zhuang-man-b/

该问题可以转换为 Subset Sum 问题，从而使用 0-1 背包的方法来求解。

可以将这组数看成两部分，P 和 N，其中 P 使用正号，N 使用负号，有以下推导：

```
                  sum(P) - sum(N) = target
sum(P) + sum(N) + sum(P) - sum(N) = target + sum(P) + sum(N)
                      2 * sum(P) = target + sum(nums)
```

因此只要找到一个子集，令它们都取正号，并且和等于 (target + sum(nums))/2，就证明存在解。

```
class Solution {
    public int findTargetSumWays(int[] nums, int S) {
        int len = nums.length;
        int sum = 0;
        for (int num: nums) {
            sum += num;
        }

        if (sum < S || ((sum + S) & 1) == 1) {
            return 0;
        }

        int target = (sum + S) / 2;
        int[] dp = new int[target + 1];
        dp[0] = 1;
```

```
            for (int num : nums) {
                for (int j = target; num <= j; j--){
                    dp[j] = dp[j] + dp[j - num];
                }
            }

            return dp[target];
        }
}
```

- 货物转移（lintcode1161）【背包】

本题可以使用背包 dp 的方法求解 我们用 `dp[i]`来表示当前容量为 i 时所需要的最少背包数。 状态转移方程为 `dp[j] = min(dp[j],dp[max(0, j-b[i])]+1)` 最短的时间 `time` 是由货物的转移来决定的。所以我们用一个额外的数组 `weight[i]`来表示货物的重量。 所以 `time = sum_good-max_weight`

```
public class Solution {
    /**
     * @param n: the number of backpacks
     * @param a: the number of goods each backpack carries
     * @param b:    the maximum capacity of each backpack
     * @return: given n, ai and bi,return the minimum number of backpacks and the minimum time
     */
    public int[] goodsTransfer(int n, int[] a, int[] b) {
        final int INF = 0x3f3f3f3f; //0x3f3f3f3f 的十进制是 1061109567，是 10^9 级别的（和 0x7fffffff 一个数量级），而一般场合下的数据都是小于 10^9 的，所以它可以作为无穷大使用而不致出现数据大于无穷大的情形。
        int max_weight = 0, sum_good = 0, sum_cap = 0, k = INF;
        for (int i = 0; i < a.length; i++) {
```

```
            sum_good += a[i];
            sum_cap += b[i];
        }
        int[] dp = new int[sum_cap + 1];
        int[] weight = new int[sum_cap + 1];
        for (int i = 1; i <=sum_cap; i++) {
            dp[i] = INF;
        }
        for (int i = 0; i < n; i++) {
            for (int j = sum_cap; j > 0; j--) {
                int res = Math.max(j-b[i], 0);
                if (dp[j] < dp[res] + 1) {
                    continue;
                } else if (dp[j] > dp[res] + 1) {
                    dp[j] = dp[res] + 1;
                    weight[j] = weight[res] + a[i];
                } else {
                    weight[j] = Math.max(weight[j], weight[res] + a[i]);
                }
            }
        }
        for (int i = sum_good; i <= sum_cap; i++) {
            if (dp[i] < k) {
                k = dp[i];
                max_weight = weight[i];
            } else if (dp[i] == k) {
                max_weight = Math.max(max_weight, weight[i]);
            }
        }
        return new int[] {k , sum_good - max_weight};
    }
}
```

- 最后一块石头的重量 2（1049）

禁止商用
公众号：
yes的练级攻略

1049. 最后一块石头的重量 II

难度 中等 81 收藏 分享 切换为英文 关注 反馈

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出**任意两块石头**，然后将它们一起粉碎。假设石头的重量分别为 `x` 和 `y`，且 `x <= y`。那么粉碎的可能结果如下：

- 如果 `x == y`，那么两块石头都会被完全粉碎；
- 如果 `x != y`，那么重量为 `x` 的石头将会完全粉碎，而重量为 `y` 的石头新重量为 `y-x`。

最后，最多只会剩下一块石头。返回此石头**最小的可能重量**。如果没有石头剩下，就返回 `0`。

示例：

```
输入：[2,7,4,1,8,1]
输出：1
解释：
组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，
组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。
```

提示：

1. `1 <= stones.length <= 30`
2. `1 <= stones[i] <= 1000`

https://leetcode-cn.com/problems/last-stone-weight-ii/solution/01bei-bao-zhuan-huan-wen-ti-by-pen999/

```
class Solution {
    //0,1 背包的转换，相当于把整个数组分成两堆，尽可能的平均分，这样最后剩下的最少
    public int lastStoneWeightII(int[] stones) {
        int n = stones.length;
        int sum = 0;
        for (int s : stones){
            sum += s;
        }
        int size = sum / 2;
        int[] dp = new int[size + 1];
        dp[0] = 0;
        //第一维是物品个数
        for (int i = 0; i < n; i++){
            //第二维是剩余容量，此处从大到小遍历，为了简化一维数组，从后往前使用的数据是上一层未更新的
            for (int j = size; j >= stones[i]; j--) {
                dp[j] = Math.max(dp[j], dp[j - stones[i]] + stones[i]);
            }
        }
        //两堆分别为sum-dp，dp,两者相减就是最小差
        return sum - 2 * dp[size];
```

```
    }
}
```

2. 完全背包

描述：有 N 种物品和一个容量为 V 的背包，**每种物品都有无限件可用**。第 i 种物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

主要代码：

```
for (int i = 1; i <= N; i++){
    for (int j = v[i - 1]; j<= V; j++){
        dp[j] = Math.max(dp[j], dp[j-v[i-1]] + w[i-1]);
    }
}
```

- 完全背包问题（acwing 3）

有 N 种物品和一个容量是 V 的背包，每种物品都有无限件可用。

第 i 种物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 种物品的体积和价值。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$

输入样例

```
4 5
1 2
2 4
3 4
4 5
```

输出样例：

```
10
```

方法 1：适用于数据量比较小的时候

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
```

```
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w));
    }
    private static int maxValue(int N, int V, int[] v, int[] w){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        for (int i = 0; i < N; i++){
            for (int j = v[i]; j<= V; j++){
                dp[j] = Math.max(dp[j], dp[j-v[i]] + w[i]);
            }
        }
        return dp[V];
    }
}
```

方法 2：适用于数据量比较大的时候【二进制】

```
import java.util.Scanner;
import java.util.ArrayList;
public class Main{
    static class ValueHolder{
        int v,w;
        public ValueHolder(int v, int w) {
            this.v = v;
            this.w = w;
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w));
    }
```

禁止商用
公众号：
yes的练级攻略

```
    private static int maxValue(int N, int V, int[] v, int[] w){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        ArrayList<ValueHolder> list = new ArrayList<>();
        for (int i = 0; i < N; i++){
            list.add(new ValueHolder(v[i], w[i]));
        }
        for (ValueHolder holder : list){
            for (int j = holder.v; j <= V ; j++){
                dp[j] = Math.max(dp[j], dp[j - holder.v] + holder.w);
            }
        }
        return dp[V];
    }
}
```

- 背包问题 3（lintcode 440）

440. 背包问题 III

给定 `n` 种物品, 每种物品都有无限个. 第 `i` 个物品的体积为 `A[i]`, 价值为 `V[i]`.

再给定一个容量为 `m` 的背包. 问可以装入背包的最大价值是多少?

样例
样例 1:

```
输入: A = [2, 3, 5, 7], V = [1, 5, 2, 4], m = 10
输出: 15
解释: 装入三个物品 1 (A[1] = 3, V[1] = 5), 总价值 15.
```

样例 2:

```
输入: A = [1, 2, 3], V = [1, 2, 3], m = 5
输出: 5
解释: 策略不唯一. 比如, 装入五个物品 0 (A[0] = 1, V[0] = 1).
```

注意事项
1. 不能将一个物品分成小块.
2. 放入背包的物品的总大小不能超过 `m`.

```
public class Solution {
    /**
     * @param A: an integer array
     * @param V: an integer array
     * @param m: An integer
     * @return: an array
     */
    public int backPackIII(int[] A, int[] V, int m) {
        if (A.length == 0 || m == 0) return 0;
        int[] dp = new int[m+1];
        for (int i = 0; i < A.length; i++){
```

```
                for (int j = A[i]; j <= m; j++){
                    dp[j] = Math.max(dp[j], dp[j-A[i]] + V[i]);
                }
            }
            return dp[m];
        }
}
```

- 背包问题 10（lintcode801）

801. 背包问题X

你总共有 n 元，商人总共有三种商品，它们的价格分别是150元,250元,350元，三种商品的数量可以认为是无限多的，购买完商品以后需要将剩下的钱给商人作为小费，求最少需要给商人多少小费

样例

样例 1:

输入： n = 900
输出： 0

样例 2:

输入： n = 800
输出： 0

```
public class Solution {
    /**
     * @param n: the money you have
     * @return: the minimum money you have to give
     */
    public int backPackX(int n) {
        int[] prices = {150, 250, 350};
        int[] dp = new int[n+1];
        for (int i = 0; i < 3; i++){
            for (int j = prices[i]; j <= n; j++){
                dp[j] = Math.max(dp[j], dp[j - prices[i]] + prices[i]);
            }
        }
        return n - dp[n];
    }
}
```

- 剩余价值背包（lintcode 971）

禁止商用 公众号：yes的练级攻略

971. 剩余价值背包

有一个容量为 `c` 的背包。
有 `n` 个 A 类物品，第 `i` 个 A 类物品的体积为 `a[i]`，物品的价值为装入该物品后背包剩余容量 * `k1`。
有 `m` 个 B 类物品，第 `i` 个 `B` 类物品的体积为 `b[i]`，物品的价值为装入该物品后背包剩余容量 * `k2`。
求最大可以获得的价值。

样例
Example 1:

```
Given k1 = `3`,k2 = `2`,c = ` 7`,n = `2`,m = `3`,a = `[4,3]`,b = `[1,3,2]`, return `23`.
Input:
3 2 7 2 3
[4,3]
[1,3,2]
Output:
23

Explanation:
2 * (7-1)+2*(6-2) + 3 * (4-3) = 23
```

Example 2:

```
Given k1 = `1`,k2 = `2`,c = ` 5`,n = `1`,m = `1`,a = `[2]`,b = `[1]`, return `10`.
Input:
1 2 5 1 1
[2]
[1]
Output:
10

Explanation:
2 * (5-1)+1*(4-2) = 10
```

注意事项

- `1 <= k1, k2, c, a[i], b[i] <= 10^7`
- `1 <= n, m <= 1000`

yes的练级攻略

https://www.jiuzhang.com/solution/surplus-value-backpack/#tag-lang-java

更多硬核内容就在公众号「yes 的练级攻略」

解题思路

- 通过分析可知，同类物品只有先装重量小的再装重量大的，策略是最优的。
- 我们可以将两个数组先排序，然后使用动态规划，分别计算取A类的前 `i` 个和B类的前 `j` 个物品的总价值，通过遍历找到最大值。

算法流程

- 设 `dp[i][j]` 表示取A类的前i个和B类的前j个物品放入背包得到的最大总价值。初始条件：`f[0][0] = 0`
- 设背包的剩余容量为 `surplus_c` ，那么有

$$surplus_c = c - a[0] - a[1] - \ldots - a[i-1] - b[0] - b[1] \ldots - b[j-1]$$

- 状态转移情况可能有两种：
  - 上一状态取A类的前 `i-1` 个和B类的前 `j` 个物品，这次选取了A类的第 `i` 个物品，那么新增价值为 $k1 * surplus_c$
  - 上一状态取A类的前 `i` 个和B类的前 `j-1` 个物品，这次选取了B类的第 `j` 个物品，那么新增价值为 $k2 * surplus_c$
- 我们写出状态转移方程

$$dp[i][j] = max \begin{cases} dp[i-1][j] + k1 * surplus_c \\ dp[i][j-1] + k2 * surplus_c \end{cases}$$

- 遍历得到所有的 `dp[i][j]` ，其中的最大值就是我们的答案。
- 另外，剩余容量可以用前缀和来计算，有效降低了时间复杂度。

复杂度分析

- 时间复杂度：$O(mn)$。对 `dp` 数组遍历。
- 时间复杂度：$O(mn)$。 `dp` 数组的大小。

```
public class Solution {
    /**
     * @param k1: The coefficient of A
     * @param k2: The    coefficient of B
     * @param c: The volume of backpack
     * @param n: The amount of A
     * @param m: The amount of B
     * @param a: The volume of A
     * @param b: The volume of B
     * @return: Return the max value you can get
     */
    public long getMaxValue(int k1, int k2, int c, int n, int m, int[] a, int[] b) {
        // step1: 对 a， b 数组分别排序
        Arrays.sort(a);
        Arrays.sort(b);
        // step2: 求 a, b 数组的前缀和
        long[] prefix_sum_a = new long[n + 1];
        long[] prefix_sum_b = new long[m + 1];
        for (int i = 0; i < n; i ++){
            prefix_sum_a[i + 1] = prefix_sum_a[i] + a[i];
        }
        for (int i = 0; i < m; i ++){
            prefix_sum_b[i + 1] = prefix_sum_b[i] + b[i];
        }
```

公众号：yes的练级攻略

```
            // step3: 动态规划
            long dp[][] = new long[n + 1][m + 1];
            long res = 0;
            for (int i = 0; i < n + 1; i ++){
                for (int j = 0; j < m + 1; j ++){
                    long surplus_c = c - prefix_sum_a[i] - prefix_sum_b[j];
                    if (surplus_c < 0) break;
                    if (i > 0) dp[i][j] = dp[i - 1][j] + k1 * surplus_c;
                    if (j > 0) dp[i][j] = Math.max(dp[i][j], dp[i][j - 1] + k2 * surplus_c);
                    res = Math.max(res, dp[i][j]);
                }
            }
            return res;
        }
}
```

- 零钱兑换（322）

322. 零钱兑换

难度 中等 693 收藏 分享 切换为英文 关注 反馈

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1 。

示例 1:

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

示例 2:

```
输入: coins = [2], amount = 3
输出: -1
```

https://leetcode-cn.com/problems/coin-change/solution/322-ling-qian-dui-huan-by-leetcode-solution/

算法：

我们采用自下而上的方式进行思考。仍定义 $F(i)$ 为组成金额 i 所需最少的硬币数量，假设在计算 $F(i)$ 之前，我们已经计算出 $F(0)-F(i-1)$ 的答案。 则 $F(i)$ 对应的转移方程应为

$$F(i)=\min_{j=0\ldots n-1}F(i-c_j)+1$$

其中 c_j 代表的是第 j 枚硬币的面值，即我们枚举最后一枚硬币面额是 c_j，那么需要从 $i-c_j$ 这个金额的状态 $F(i-c_j)$ 转移过来，再算上枚举的这枚硬币数量 1 的贡献，由于要硬币数量最少，所以 $F(i)$ 为前面能转移过来的状态的最小值加上枚举的硬币数量 1 。

例子1：假设

```
coins = [1, 2, 5], amount = 11
```

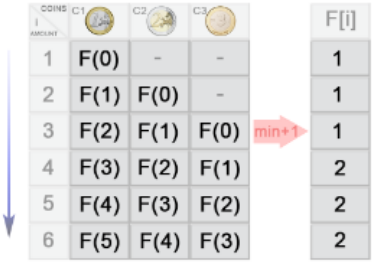
则，当 $i==0$ 时无法用硬币组成，为 0 。当 $i<0$ 时，忽略 $F(i)$

F(i)	最小硬币数量
F(0)	0 //金额为0不能由硬币组成
F(1)	1 //$F(1)=min(F(1-1),F(1-2),F(1-5))+1=1$
F(2)	1 //$F(2)=min(F(2-1),F(2-2),F(2-5))+1=1$
F(3)	2 //$F(3)=min(F(3-1),F(3-2),F(3-5))+1=2$
F(4)	2 //$F(4)=min(F(4-1),F(4-2),F(4-5))+1=2$
...	...
F(11)	3 //$F(11)=min(F(11-1),F(11-2),F(11-5))+1=3$

我们可以看到问题的答案是通过子问题的最优解得到的。

例子2：假设

```
coins = [1, 2, 3], amount = 6
```



Finding minimal number of coins for amount i = 6.
F[6] = 2

在上图中，可以看到：

$$\begin{aligned}F(3)&=\min(F(3-c_1),F(3-c_2),F(3-c_3))+1\\&=\min(F(3-1),F(3-2),F(3-3))+1\\&=\min(F(2),F(1),F(0))+1\\&=\min(1,1,0)+1\\&=1\end{aligned}$$

复杂度分析

- 时间复杂度：$O(Sn)$，其中 S 是金额，n 是面额数。我们一共需要计算 $O(S)$ 个状态，S 为题目所给的总金额。对于每个状态，每次需要枚举 n 个面额来转移状态，所以一共需要 $O(Sn)$ 的时间复杂度。
- 空间复杂度：$O(S)$。DP 数组需要开长度为总金额 S 的空间。

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        if (coins == null || coins.length == 0) return -1;

        int max = amount + 1;
        int[] dp = new int[max];
        Arrays.fill(dp, max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (coins[j] <= i) {
                    dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }

        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

- 零钱兑换 2（518）

518. 零钱兑换 II

难度 中等 188 收藏 分享 切换为英文 关注 反馈

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

示例 3:

```
输入: amount = 10, coins = [10]
输出: 1
```

https://leetcode-cn.com/problems/coin-change-2/solution/ling-qian-dui-huan-ii-by-leetcode/

策略为：

- 从基本情况没有***开始，一一添加。***
- 对于每个添加的***，我们从金额 `0` 到 `amount` 递归的计算组合数量。

算法：

- 以基本情况没有***开始组合数量。 `dp[0] = 1` ，然后其余等于 `0` 。
- 遍历所有***面值：
  - 对于每个***，我们将从金额 `0` 遍历到 `amount` ：
    - 对于每个 `x` ，计算组合数： `dp[x] += dp[x - coin]` 。
- 返回 `dp[amount]` 。

复杂度分析

- 时间复杂度：$\mathcal{O}(N \times \text{amount})$。其中 `N` 为 `coins` 数组的长度。
- 空间复杂度：$\mathcal{O}(\text{amount})$， `dp` 数组使用的空间。

```
class Solution {
    public int change(int amount, int[] coins) {
        int[] dp = new int[amount + 1];
        dp[0] = 1;

        for (int coin : coins) {
        for (int x = coin; x < amount + 1; ++x) {
            dp[x] += dp[x - coin];
        }
        }
        return dp[amount];
    }
}
```

- 组合总和四（377）（lintcode 564）【和上一题的内外循环不能相反】

更多硬核内容就在公众号「yes 的练级攻略」

377. 组合总和 Ⅳ

难度 中等 165 收藏 分享 切换为英文 关注 反馈

给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

示例:

```
nums = [1, 2, 3]
target = 4

所有可能的组合为:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)

请注意，顺序不同的序列被视作不同的组合。

因此输出为 7。
```

进阶:
如果给定的数组中含有负数会怎么样?
问题会产生什么变化?
我们需要在题目中添加什么限制来允许负数的出现?

https://leetcode-cn.com/problems/combination-sum-iv/solution/dong-tai-gui-hua-python-dai-ma-by-liweiwei1419/

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

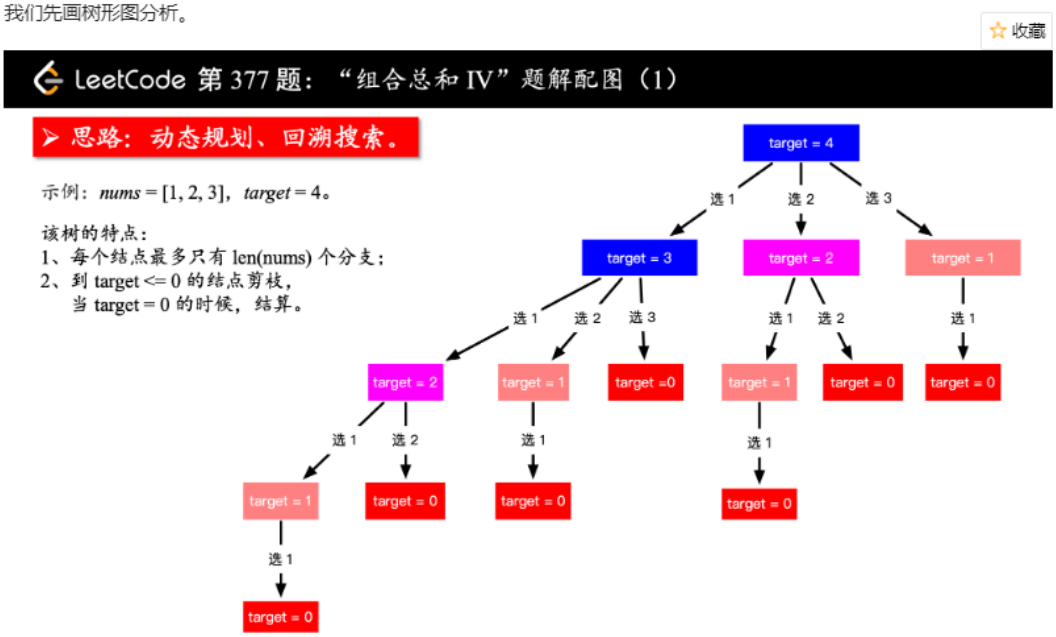
题意分析：

- 输入数组的每个元素可以使用多次，这一点和「完全背包」问题有点像；
- 顺序不同的序列被视作不同的组合，这一点和所有的「背包问题」都不同，与 518. 零钱兑换 II 问题不同的地方就在这一点。

思路分析：

- 遇到这一类问题，做一件事情有很多种做法，每一种做法有若干个步骤，脑子里能想到的常规思路大概有「回溯搜索」、「动态规划」；
- 由于不用得到具体的组合表示，因此考虑使用「动态规划」来解。

我们先画树形图分析。



很容易发现「重复问题」，因此，我们可以使用「动态规划」来做，如果题目问具体的解，那么用「回溯搜索」做（「力扣」第 39 题：组合之和）。

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

对上图的解释：

☆ 收藏

我们就以 nums = [1,2,3]，target = 4 为例：
下面我们分析 4 可以有几种分解：
4 = 1 + 3，其中 3 应该看成用 [1,2,3] 组成之和为 3 的组合数：{1,1,1,1},...，先略过：
4 = 2 + 2，其中 2 应该看成用 [1,2,3] 组成之和为 2 的组合数：{2,1,1},{2,2}，共 2 种；
4 = 3 + 1，其中 1 应该看成用 [1,2,3] 组成之和为 1 的组合数：{3,1}，共 1 种。
所以，4 一共有（3 的分解种数）种 + 2 种 + 1 种。

下面我们分析 3 可以有几种分解：
3 = 1 + 2，其中 2 应该看成用 [1,2,3] 组成之和为 2 的组合数：{1,1,1},{1,2}，共 2 种；
3 = 2 + 1，其中 1 应该看成用 [1,2,3] 组成之和为 1 的组合数：{2,1}，共 1 种。
3 = 3 + 0，其中 0 应该看成用 [1,2,3] 组成之和为 0 的组合数：{3,{}}，共 1 种。
所以 3 一共有 2 种 + 1 种 + 1 种 = 4 种。再回到 4 的分解可知 4 有 (4+2+1=) 7 种。

2 = 1 + 1，其中 1 应该看成用 [1,2,3] 组成之和为 1 的组合数：{1,1}，共 1 种；
2 = 2 + 0，其中 0 应该看成用 [1,2,3] 组成之和为 0 的组合数：{2,{}}，共 1 种；
所以 2 一共有 1 种 + 1 种 = 2 种。

1 = 1 + 0，其中 0 应该看成用 [1,2,3] 组成之和为 0 的组合数：{1,{}}，共 1 种；
所以 1 一共有 1 种 。

怎么写代码呢？

- 递归求解：由于有大量「重复子问题」，因此必须使用缓存，以避免相同问题重复求解，这个方法叫「记忆化搜索」，在《算法导论》这本书上也把它归入到「动态规划」的定义中。这种思考问题的方式是「从上到下」的，直接面对问题求解，遇到什么问题，就解决什么问题，同时记住结果；
- 「动态规划」告诉了我们另一种思考问题的方式：「从底向上」，可以不直接面对问题求解，从这个问题最小的样子开始，通过逐步递推，至到得到所求的问题的答案。

虽然这个问题没有明显的「最优子结构」，但这种「从底向上」递推的思路是很深刻的，我们也把它归纳到「动态规划」的解法中。

方法：动态规划

“动态规划”的两个步骤是思考“状态”以及“状态转移方程”。

1、状态

对于“状态”，我们首先思考能不能就用问题当中问的方式定义状态，上面递归树都画出来了。当然就用问题问的方式。

`dp[i]` ：对于给定的由正整数组成且不存在重复数字的数组，和为 `i` 的组合的个数。

思考输出什么？因为状态就是问题当中问的方式而定义的，因此输出就是最后一个状态 `dp[n]` 。

2、状态转移方程

由上面的树形图，可以很容易地写出状态转移方程：

```
dp[i] = sum(dp[i - num] for num in nums and if i >= num)
```

注意：在 0 这一点，我们定义 `dp[0] = 1` 的，它表示如果 `nums` 里有一个数恰好等于 `target` ，它单独成为 1 种可能。

对于进阶问题的思考

1、如果给定的数组中含有负数会怎么样？问题会产生什么变化？

如果有负数，相当于给定数组中的元素有了更多的组合，特别是出现了一对相反数的时候，例如题目中的示例 `[-4, 1, 2, 3, 4]`，`target = 4` 的时候，`-4` 和 `4` 可以无限次地、成对添加到题目中的示例中，成为新的组合，那么这道问题就没有什么意义了。

仔细思考，负数我只要不选它就行了。但由于这道问题的问法是“组合”，因此我们要保证有负数参与进来，不能够与已有的正数的组合之和为 0 即可。

2、我们需要在题目中添加什么限制来允许负数的出现？

- 如果有负数参与进来，不能够与已有的正数的组合之和为 `0`；
- 或者限制负数的使用次数，设计成类似 0-1 背包问题的样子。

可能有考虑不完全的地方，欢迎讨论。

```
public class Solution {
    /**
     * 这里状态定义就是题目要求的，并不难，状态转移方程要动点脑子，也不难：
     * 状态转移方程：
dp[i]= dp[i - nums[0]] + dp[i - nums[1]] + dp[i - nums[2]] + ... （当 [] 里面的数 >= 0）
     * 特别注意：dp[0] = 1，表示，如果那个硬币的面值刚刚好等于需要凑出的价值，这个就成为 1 种组合方案
     * 再举一个具体的例子：nums=[1, 3, 4], target=7;
     * dp[7] = dp[6] + dp[4] + dp[3]
     * 即：7 的组合数可以由三部分组成，1 和 dp[6]，3 和 dp[4]，4 和dp[3];
     *
     * @param nums
     * @param target
     * @return
     */
    public int combinationSum4(int[] nums, int target) {
        int[] dp = new int[target + 1];
        // 这个值被其它状态参考，设置为 1 是合理的
        dp[0] = 1;

        for (int i = 1; i <= target; i++) {
            for (int num : nums) {
                if (num <= i) {
                    dp[i] += dp[i - num];
                }
            }
        }
        return dp[target];
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

如果是重复数字的话，可以进行去重以后用零钱兑换 2，去重代码

```
import java.util.*;
public class RemoveRepetition {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int[] str = {5, 6, 6, 6, 8, 8, 7, 4};
        List<Integer> list = new ArrayList<Integer>();
        for (int i=0; i<str.length; i++) {
            if(!list.contains(str[i])) {
                list.add(str[i]);
            }
        }
        int[] res = new int[list.size()];
        for (int j = 0; j < list.size(); j++){
            res[j] = list.get(j);
            System.out.println(res[j]);
        }
    }
}
```

禁止商用

- 硬币（面试题 08.11）

面试题 08.11. 硬币

难度 中等 160 收藏 分享 切换为英文 关注 反馈

硬币。给定数量不限的硬币，币值为25分、10分、5分和1分，编写代码计算n分有几种表示法。(结果可能会很大，你需要将结果模上1000000007)

示例1:

```
 输入: n = 5
 输出: 2
 解释: 有两种方式可以凑成总金额:
5=5
5=1+1+1+1+1
```

示例2:

```
 输入: n = 10
 输出: 4
 解释: 有四种方式可以凑成总金额:
10=10
10=5+5
10=5+1+1+1+1+1
10=1+1+1+1+1+1+1+1+1+1
```

说明:

注意:

你可以假设:

- 0 <= n (总金额) <= 1000000

https://leetcode-cn.com/problems/coin-lcci/solution/dong-tai-gui-hua-wan-quan-bei-bao-wen-ti-by-eddiev/

改进后：

正确答案：

于是我们先遍历硬币，保证在考虑一枚硬币的情况时，没有较大的硬币影响，这样，我们最终每种组合情况，都是以硬币的面额大小非递减组合。保证了同样的情况，调换顺序后重复计算的情况。

这时候，我们求6的硬币情况数时，我们观察一下流程：

```
coin = 1:
前面5种情况数：dp[1,5] = [1,1,1,1,1];

dp[6] += (dp[6 - coin] => dp[5] => 1);
即拿到coin(1)的情况有一种 :
        coin(1,1,1,1,1) + coin(1);

coin = 5:
前面5种情况数：dp[1,5] = [1,1,1,1,2];

dp[6] += (dp[6 - coin] => dp[1] => 1);
即拿到coin（5）的情况有一种:
        coin(1) + coin(5);
```

此时，硬币组合情况，的确为正确的情况。

```
class Solution {
    public int waysToChange(int n) {
        int[] coins = {1,5,10,25};
        int[] dp = new int[n+1];
        dp[0] = 1;
        for (int coin : coins) {
            for (int i = coin; i <= n; i++){
                dp[i] = (dp[i] + dp[i - coin]) % 1000000007;
            }
        }
        return dp[n];
    }
}
```

3. 多重背包

描述：有 N 种物品和一个容量为 V 的背包。**第 i 种物品最多有 n[i]件可用**，每件费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

主要代码（只适用于数据量较少的时候）

```
for (int i = 1; i <= N; i++){
    for (int j = V; j >= 0; j--){
        for (int k = 0; k <= s[i - 1]; k++){
            if (j >= k * v[i - 1]) {
                dp[j] = Math.max(dp[j], dp[j - k * v[i - 1]] + k * w[i - 1]);
            }
        }
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

- 多重背包问题 1 2 3（acwing 4 5 6）

4. 多重背包问题 I

题目 | 提交记录 | 讨论 | 题解 | 视频讲解

有 N 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行三个整数 v_i, w_i, s_i，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 100$
$0 < v_i, w_i, s_i \leq 100$

输入样例

```
4 5
1 2 3
2 4 1
3 4 3
4 5 2
```

输出样例：

```
10
```

yes的练级攻略

5. 多重背包问题 II

题目 | 提交记录 | 讨论 | 题解 | 视频讲解

有 N 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行三个整数 v_i, w_i, s_i，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N \leq 1000$
$0 < V \leq 2000$
$0 < v_i, w_i, s_i \leq 2000$

提示:

本题考查多重背包的二进制优化方法。

输入样例

```
4 5
1 2 3
2 4 1
3 4 3
4 5 2
```

输出样例:

```
10
```

6. 多重背包问题 III

题目 | 提交记录 | 讨论 | 题解 | 视频讲解

有 N 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

输入格式

第一行两个整数，N，V $(0 < N \leq 1000, 0 < V \leq 20000)$，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行三个整数 v_i, w_i, s_i，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N \leq 1000$
$0 < V \leq 20000$
$0 < v_i, w_i, s_i \leq 20000$

提示

本题考查多重背包的单调队列优化方法。

输入样例

```
4 5
1 2 3
2 4 1
3 4 3
4 5 2
```

输出样例：

```
10
```

方法 1：适用于数据量比较小的时候

```
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        int[] s = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
            s[i] = sc.nextInt();
```

```
        }
        System.out.println(maxValue(N, V, v, w, s));
    }
    private static int maxValue(int N, int V, int[] v, int[] w, int[] s){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        for (int i = 0; i < N; i++){
            for (int j = V; j >= 0; j--){
                for (int k = 0; k <= s[i]; k++){
                    if (j >= k * v[i]) {
                        dp[j] = Math.max(dp[j], dp[j - k * v[i]] + k * w[i]);
                    }
                }
            }
        }
        return dp[V];
    }
}
```

方法 2：适用于数据量比较大的时候【二进制】

```
import java.util.Scanner;
import java.util.ArrayList;
public class Main{
    static int MAX = 101;
    static class ValueHolder{
        int v,w;
        public ValueHolder(int v, int w) {
            this.v = v;
            this.w = w;
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        int[] s = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
            s[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w, s));
    }
```

禁止商用
公众号：
yes的练级攻略

```
    public static int maxValue(int N, int V, int[] v, int[] w, int[] s){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        ArrayList<ValueHolder> list = new ArrayList<>();
        for (int i = 0; i < N; i++){
            int cnt = s[i];
            for (int k = 1; k < cnt; k *= 2){
                cnt -= k;
                list.add(new ValueHolder(v[i] * k, w[i] * k));
            }
            if (cnt >= 0) {
                list.add(new ValueHolder(v[i] * cnt, w[i] * cnt));
            }
        }
        for (ValueHolder holder : list){
            for (int j = V; j >= holder.v; j--){
                dp[j] = Math.max(dp[j], dp[j - holder.v] + holder.w);
            }
        }
        return dp[V];
    }
}
```

方法 3：适用于数据量比较大的时候【单调队列】

https://www.acwing.com/solution/content/16038/

pinkfloyda 1个月前 回复

花了点时间 优化了一下代码，其实可以用一个单调队列，然后循环每个slice，只要每次重置这个单调队列就行了
而且不需要二维数组，可以用一维数组，因为之前的最值都保存在了单调队列里面，从左向右每次dp[j]都表示上面的i-1th的值，拿来用就好了，保留了Deque class没有优化成数组，更直观，下面是优化过的代码，供参考

```
import java.util.*;
public class Main{
    static int MAX = 101;
    static class ValueHolder{
        int v,w;
        public ValueHolder(int v, int w) {
            this.v = v;
            this.w = w;
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
```

禁止商用 公众号：yes的练级攻略

```
            int[] s = new int[N];
            for (int i = 0; i < N; i++){
                v[i] = sc.nextInt();
                w[i] = sc.nextInt();
                s[i] = sc.nextInt();
            }
            System.out.println(maxValue(N, V, v, w, s));
        }
        private static int maxValue(int N, int C, int[] ss, int[] vs, int[] qs) {
            int[] dp = new int[C+1];
            Deque<int[]> dq = new LinkedList<>();
            int ws = 0; // window start index
            int we = 0; // window end index
            for (int i=0; i<N; i++) {
                int s = ss[i], v = vs[i], q = qs[i];
                for (int k=0; k<s; k++) { // loop s slices
                    dq.clear();
                    ws = we = k;
                    for (int j=k; j<=C; j+=s) { // increment each slice by s
                        if ((we-ws)/s == q) { // slide window already max out
                            if (dq.size() > 0 && dq.peekFirst()[0] == ws) { // if window start already the max, will eject it
                                dq.removeFirst();
                            }
                            ws+=s;
                        }
                        int m = (j-k)/s; // multiplier
                        while (!dq.isEmpty() && dq.peekLast()[1] <= dp[j]-m*v) { // try to add the current index
                            dq.removeLast();
                        }
                        dq.addLast(new int[]{j, dp[j]-m*v});
                        we=j;
                        dp[j] = dq.peekFirst()[1]+m*v;
                    }
                }
            }
            return dp[C];
        }
}
```

- 背包问题 7（lintcode 798）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

798. 背包问题VII 中文

假设你身上有 n 元，超市里有多种大米可以选择，每种大米都是袋装的，必须整袋购买，给出每种大米的重量，价格以及数量，求最多能买多少公斤的大米

样例
样例 1:

```
输入：  n = 8, prices = [3,2], weights = [300,160], amounts = [1,6]
输出：  640
解释：  全买价格为2的米。
```

样例 2:

```
输入：  n = 8, prices  = [2,4], weight = [100,100], amounts = [4,2 ]
输出：  400
解释：  全买价格为2的米
```

```
public class Solution {
    /**
     * @param n: the money of you
     * @param prices: the price of rice[i]
     * @param weight: the weight of rice[i]
     * @param amounts: the amount of rice[i]
     * @return: the maximum weight
     */
    public int backPackVII(int n, int[] prices, int[] weight, int[] amounts) {
        if (n == 0 || prices.length == 0) return 0;
        int[] dp = new int[n+1];
        for (int i = 0; i < prices.length; i++){
            for (int j = n; j >= 0; j--){
                for (int k = 0; k <= amounts[i]; k++){
                    if (j >= k * prices[i]){
                        dp[j] = Math.max(dp[j], dp[j - k * prices[i]] + k * weight[i]);
                    }
                }
            }
        }
        return dp[n];
    }
}
```

- 背包问题 8（lintcode 799）

禁止商用 公众号：yes的练级攻略

799. 背包问题VIII

给一些不同价值和数量的硬币。找出这些硬币可以组合在 1 ~ n 范围内的值的数量

样例
样例 1:
输入:
n = 5
value = [1,4]
amount = [2,1]
输出: 4

解释:
可以组合出1，2，4，5

样例 2:
输入:
n = 10
value = [1,2,4]
amount = [2,1,1]
输出: 8

解释:
可以组合出1-8所有数字。

```
public class Solution {
    /**
     * @param n: the value from 1 - n
     * @param value: the value of coins
     * @param amount: the number of coins
     * @return: how many different value
     */
    public int backPackVIII(int n, int[] value, int[] amount) {
        int[] dp = new int[n+1];
        for (int i = 1; i <= n; i++) dp[i] = -1;
        int res = 0;
        for (int i = 0; i < value.length; i++){
            for (int j = 0; j <= n; j++){
                //已经可以组成的情况
                if (dp[j] >= 0) dp[j] = amount[i];
                //用一个就超出了或者比 j 小的数都用完了
                else if (j < value[i] || dp[j - value[i]] < 0) dp[j] = -1;
                //用上了第 i 个硬币的情况
                else dp[j] = dp[j - value[i]] - 1;
            }
        }
        for (int i = 1; i <= n; i++){
            //剩下硬币数量大于等于 0 是合法的
```

公众号：yes的练级攻略

```
                if (dp[i] >= 0) res++;
            }
            return res;
        }
}
```

- 有限纸币找零

有 1、5、10、50、100 元，分别 a、b、c、d、e 张纸币。编程实现支付 k 元，需要最少多少张纸币。

输入为两行，第一行为 5 个数字，分别代表 1、5、10、50、100 元纸币的数量

如：5 2 2 3 5

第二行为要支付的金额 k

如：55

输出描述：

输出最少需要张数，如：2

当无解时输出：-1

```
public void solution() {
    Scanner in = new Scanner(System.in);
    long[][] coins = new long[5][2];
    coins[0] = new long[]{1, in.nextLong()};
    coins[1] = new long[]{5, in.nextLong()};
    coins[2] = new long[]{10, in.nextLong()};
    coins[3] = new long[]{50, in.nextLong()};
    coins[4] = new long[]{100, in.nextLong()};
    long k = in.nextLong();
    System.out.println(minCost(coins, k));
  }

  public static long minCost(long[][] coins, long k) {
    long count = 0;
    for (int i = coins.length - 1; i >= 0; i--) {
      while (coins[i][0] <= k && coins[i][1] > 0) {
        k -= coins[i][0];
        coins[i][1]--;
        count++;
      }
    }
    if (k != 0) return -1;
    return count;
  }
}
```

4. 混合三种背包

描述：有的物品只可以取一次（01 背包），有的物品可以取无限次（完全背包），有的物品可以取的次数有一个上限（多重背包）。

- 混合背包问题（acwing 7）

禁止商用 公众号：yes的练级攻略

7. 混合背包问题

题目 | 提交记录 | 讨论 | 题解 | 视频讲解

有 N 种物品和一个容量是 V 的背包。

物品一共有三类：

- 第一类物品只能用1次（01背包）；
- 第二类物品可以用无限次（完全背包）；
- 第三类物品最多只能用 s_i 次（多重背包）；

每种体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行三个整数 v_i, w_i, s_i，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

- $s_i = -1$ 表示第 i 种物品只能用1次；
- $s_i = 0$ 表示第 i 种物品可以用无限次；
- $s_i > 0$ 表示第 i 种物品可以使用 s_i 次；

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$
$-1 \leq s_i \leq 1000$

输入样例

```
4 5
1 2 -1
2 4 1
3 4 0
4 5 2
```

输出样例：

```
8
```

```
import java.util.Scanner;
import java.util.ArrayList;
public class Main{
    static class ValueHolder{
        int v,w;
        public ValueHolder(int v, int w) {
            this.v = v;
```

```
            this.w = w;
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        int[] s = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
            s[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w, s));
    }
    public static int maxValue(int N, int V, int[] v, int[] w, int[] s){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
        ArrayList<ValueHolder> list = new ArrayList<>();
        for (int i = 0; i < N; i++){
            int cnt = s[i];
            if (cnt <= 0) { //完全背包和 01 背包
                list.add(new ValueHolder(v[i], w[i]));
                continue;
            }
            //多重背包
            for (int k = 1; k < cnt; k *= 2){
                cnt -= k;
                list.add(new ValueHolder(v[i] * k, w[i] * k));
            }
            if (cnt > 0) {
                list.add(new ValueHolder(v[i] * cnt, w[i] * cnt));
            }

        }
        for (ValueHolder holder : list){
            for (int j = V; j >= holder.v; j--){
                dp[j] = Math.max(dp[j], dp[j - holder.v] + holder.w);
            }
        }
        return dp[V];
    }
```

禁止商用 公众号：yes的练级攻略

}

5. 二维费用的背包问题

描述：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价 1 和代价 2，第 i 件物品所需的两种代价分别为 a[i]和 b[i]。两种代价可付出的最大值（两种背包容量）分别为 V 和 U。物品的价值为 w[i]。

- 二维费用的背包问题（acwing 8）

有 N 件物品和一个容量是 V 的背包，背包能承受的最大重量是 M。

每件物品只能用一次。体积是 v_i，重量是 m_i，价值是 w_i。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，总重量不超过背包可承受的最大重量，且价值总和最大。
输出最大价值。

输入格式

第一行两个整数，N，V,M，用空格隔开，分别表示物品件数、背包容积和背包可承受的最大重量。

接下来有 N 行，每行三个整数 v_i,m_i,w_i，用空格隔开，分别表示第 i 件物品的体积、重量和价值。

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N \leq 1000$
$0 < V, M \leq 100$
$0 < v_i, m_i \leq 100$
$0 < w_i \leq 1000$

输入样例

```
4 5 6
1 2 3
2 4 4
3 4 5
4 5 6
```

输出样例:

```
8
```

```
import java.util.Scanner;
import java.util.ArrayList;
public class Main{
    static class ValueHolder{
        int v, m, w;
        public ValueHolder(int v, int m, int w) {
            this.v = v;
            this.m = m;
            this.w = w;
        }
    }
    public static void main(String[] args){
```

```
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int M = sc.nextInt();
        int[] v = new int[N];
        int[] m = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            m[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, M, v, m, w));
    }
    public static int maxValue(int N, int V, int M, int[] v, int[] m, int[] w){
        if (N == 0 || V == 0 || M == 0) return 0;
        int[][] dp = new int[V+1][M+1];
        ArrayList<ValueHolder> list = new ArrayList<>();
        for (int i = 0; i < N; i++){
            list.add(new ValueHolder(v[i], m[i], w[i]));
        }
        for (ValueHolder holder : list){
            for (int j = V; j >= holder.v; j--){
                for (int k = M; k >= holder.m; k--){
                    dp[j][k] = Math.max(dp[j][k], dp[j-holder.v][k-holder.m] + holder.w);
                }
            }
        }
        return dp[V][M];
    }
}
```

- 一和零（474）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

474. 一和零

难度 中等 175 收藏 分享 切换为英文 关注 反馈

在计算机界中，我们总是追求用有限的资源获取最大的收益。

现在，假设你分别支配着 **m** 个 `0` 和 **n** 个 `1`。另外，还有一个仅包含 `0` 和 `1` 字符串的数组。

你的任务是使用给定的 **m** 个 `0` 和 **n** 个 `1` ，找到能拼出存在于数组中的字符串的最大数量。每个 `0` 和 `1` 至多被使用一次。

注意:

1. 给定 `0` 和 `1` 的数量都不会超过 `100`。
2. 给定字符串数组的长度不会超过 `600`。

示例 1:

```
输入: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
输出: 4

解释: 总共 4 个字符串可以通过 5 个 0 和 3 个 1 拼出，即 "10","0001","1","0" 。
```

示例 2:

```
输入: Array = {"10", "0", "1"}, m = 1, n = 1
输出: 2

解释: 你可以拼出 "10"，但之后就没有剩余数字了。更好的选择是拼出 "0" 和 "1" 。
```

https://leetcode-cn.com/problems/ones-and-zeroes/solution/javadong-tai-gui-hua-zhi-bei-bao-wen-ti-474-yi-he-/

解题思路

m个0，n个1 可以看作是背包，而字符串数组strs是物品列表

则对于每一个物品(str)，都有放进背包(背包的容量要变成m-zeros,n-ones)和不放进背包两种选择，
其中zeros表示str中0的个数，ones表示str中1的个数

则有状态: dp(i,j,k)代表用j个0，k个1组装strs[0...i]的最大个数
动态转移方程: dp(i,j,k) = max(dp(i-1,j,k),dp(i-1,j-zeros,k-ones))

```
class Solution {
    public int findMaxForm(String[] strs, int m, int n) {
        if (strs.length == 0 || (m == 0 && n == 0)) return 0;
        int[][] dp = new int[m + 1][n + 1];
        for (String str : strs) {
            int zeros = 0;
            int ones = 0;
            for (int i = 0; i < str.toCharArray().length; i++) {
                if (str.toCharArray()[i] == '0') {
                    zeros++;
                }else {
                    ones++;
                }
            }
```

```
                for (int i = m; i >= zeros; i--) {
                    for (int j = n; j >= ones; j--) {
                        dp[i][j] = Math.max(dp[i][j], 1 + dp[i - zeros][j - ones]);
                    }
                }
            }
            return dp[m][n];
        }
}
```

6. 分组的背包问题

描述：有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

- 分组背包问题（acwing 9）

有 N 组物品和一个容量是 V 的背包。

每组物品有若干个，同一组内的物品最多只能选一个。
每件物品的体积是 v_{ij}，价值是 w_{ij}，其中 i 是组号，j 是组内编号。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

输入格式

第一行有两个整数 N，V，用空格隔开，分别表示物品组数和背包容量。

接下来有 N 组数据：

- 每组数据第一行有一个整数 S_i，表示第 i 个物品组的物品数量；
- 每组数据接下来有 S_i 行，每行有两个整数 v_{ij},w_{ij}，用空格隔开，分别表示第 i 个物品组的第 j 个物品的体积和价值；

输出格式

输出一个整数，表示最大价值。

数据范围

$0 < N, V \leq 100$
$0 < S_i \leq 100$
$0 < v_{ij}, w_{ij} \leq 100$

输入样例

```
3 5
2
1 2
2 4
1
3 4
1
4 5
```

输出样例：

```
8
```

```
import java.util.Scanner;
public class Main{
```

```
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] dp = new int[V+1];
        for (int i = 0; i < N; i++){
            int S = sc.nextInt();
            int[] v = new int[S];
            int[] w = new int[S];
            for (int j = 0; j < S; j++){
                v[j] = sc.nextInt();
                w[j] = sc.nextInt();
            }
            dp[V] = maxValue(N, V, S, v, w, dp);
        }
        System.out.println(dp[V]);
    }
    private static int maxValue(int N, int V, int S, int[] v, int[] w, int[] dp){
        if (N == 0 || V == 0) return 0;
        for (int j = V; j >= 0; j--){
            for (int k = 0; k < S; k++){
                if (j >= v[k]) {
                    dp[j] = Math.max(dp[j], dp[j - v[k]] + w[k]);
                }
            }
        }
        return dp[V];
    }
}
```

禁止商用
公众号：
yes的练级攻略

7. 有依赖的背包问题

描述：这种背包问题的物品间存在某种“依赖”的关系。也就是说，i 依赖于 j，表示若选物品 i，则必须选物品 j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

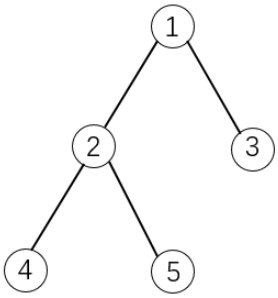
- 有依赖的背包问题（acwing 10）

更多硬核内容就在公众号「yes 的练级攻略」

有 N 个物品和一个容量是 V 的背包。

物品之间具有依赖关系，且依赖关系组成一棵树的形状。如果选择一个物品，则必须选择它的父节点。

如下图所示：



如果选择物品5，则必须选择物品1和2。这是因为2是5的父节点，1是2的父节点。

每件物品的编号是 i，体积是 v_i，价值是 w_i，依赖的父节点编号是 p_i。物品的下标范围是 $1 \dots N$。

求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。

输出最大价值。

输入格式

第一行有两个整数 N，V，用空格隔开，分别表示物品个数和背包容量。

接下来有 N 行数据，每行数据表示一个物品。
第 i 行有三个整数 v_i, w_i, p_i，用空格隔开，分别表示物品的体积、价值和依赖的物品编号。
如果 $p_i = -1$，表示根节点。 **数据保证所有物品构成一棵树。**

输出格式

输出一个整数，表示最大价值。

数据范围

$1 \le N, V \le 100$
$1 \le v_i, w_i \le 100$

父节点编号范围：

- 内部结点：$1 \le p_i \le N$;
- 根节点 $p_i = -1$;

输入样例

```
5 7
2 3 -1
2 2 1
3 5 1
4 7 2
3 6 2
```

输出样例：

```
11
```

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        int[] p = new int[N];
        List<Integer>[] tree = new List[N+1];
```

```
        Arrays.setAll(tree, e -> new ArrayList<>());
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
            p[i] = sc.nextInt();
            tree[p[i] < 0 ? 0 : p[i]].add(i);
        }
        System.out.println(maxValue(N, V, v, w, tree));
    }
    private static int maxValue(int N, int V, int[] v, int[] w, List<Integer>[] tree){
        if (N == 0 || V == 0) return 0;
        int[][] memo = new int[N+1][V+1];
        for (int i = 0; i <= N; i++){
            for (int j = 0; j <= V; j++){
                memo[i][j] = -1;
            }
        }
        return dfs(V, v, w, 0, tree, memo);
    }
    private static int dfs(int V, int[] v, int[] w, int root, List<Integer>[] tree, int[][] memo){
        if (memo != null && memo[root][V] >= 0) {
            return memo[root][V];
        }
        List<Integer> nodes = tree[root];
        int N = nodes.size();
        int[][] values = new int[N][V+1];//每个节点每个容量的最大价值
        for (int i = 0; i < N; i++){
            int vi = v[nodes.get(i)];
            int wi = w[nodes.get(i)];
            for (int j = vi; j <= V; j++){
                values[i][j] = dfs(j-vi, v, w, nodes.get(i)+1, tree, memo) + wi;
            }
        }
        int[] dp = new int[V+1];
        for (int i = 0; i < N; i++){
            for (int j = V; j >= 1; j--){
                for (int k = 1; k <= j; k++){
                    dp[j] = Math.max(dp[j], dp[j-k] + values[i][k]);
                }
            }
        }
        memo[root][V] = dp[V];
        return dp[V];
    }
```

禁止商用
公众号：
yes的练级攻略

}

8. 泛化物品

描述：考虑这样一种物品，它并没有固定的费用和价值，而是它的价值随着你分配给它的费用而变化。这就是泛化物品的概念。更严格的定义之。在背包容量为 V 的背包问题中，泛化物品是一个定义域为 0..V 中的整数的函数 h，当分配给它的费用为 v 时，能得到的价值就是 h(v)。这个定义有一点点抽象，另一种理解是一个泛化物品就是一个数组 h[0..V]，给它费用 v，可得到价值 h[V]。一个费用为 c 价值为 w 的物品，如果它是 01 背包中的物品，那么把它看成泛化物品，它就是除了 h(c)=w 其它函数值都为 0 的一个函数。如果它是完全背包中的物品，那么它可以看成这样一个函数，仅当 v 被 c 整除时有 h(v)=v/c*w，其它函数值均为 0。如果它是多重背包中重复次数最多为 n 的物品，那么它对应的泛化物品的函数有 h(v)=v/c*w 仅当 v 被 c 整除且 v/c<=n，其它情况函数值均为 0。一个物品组可以看作一个泛化物品 h。对于一个 0..V 中的 v，若物品组中不存在费用为 v 的的物品，则 h(v)=0，否则 h(v)为所有费用为 v 的物品的最大价值。P07 中每个主件及其附件集合等价于一个物品组，自然也可看作一个泛化物品。

9. 背包问题问法的变化

以上涉及的各种背包问题都是要求在背包容量（费用）的限制下求可以取到的最大价值，但背包问题还有很多种灵活的问法，在这里值得提一下。但是我认为，只要深入理解了求背包问题最大价值的方法，即使问法变化了，也是不难想出算法的。

例如，求解最多可以放多少件物品或者最多可以装满多少背包的空间。这都可以根据具体问题利用前面的方程求出所有状态的值（f 数组）之后得到。

还有，如果要求的是“总价值最小”“总件数最小”，只需简单的将上面的状态转移方程中的 max 改成 min 即可。

下面说一些变化更大的问法。

（1）输出方案

一般而言，背包问题是要求一个最优值，如果要求输出这个最优值的方案，可以参照一般动态规划问题输出方案的方法：记录下每个状态的最优值是由状态转移方程的哪一项推出来的，换句话说，记录下它是由哪一个策略推出来的。便可根据这条策略找到上一个状态，从上一个状态接着向前推即可。

还是以01背包为例，方程为f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。再用一个数组g[i][v]，设g[i][v]=0表示推出f[i][v]的值时是采用了方程的前一项（也即f[i][v]=f[i-1][v]），g[i][v]表示采用了方程的后一项。注意这两项分别表示了两种策略：未选第i个物品及选了第i个物品。那么输出方案的伪代码可以这样写（设最终状态为f[N][V]）：

```
i=N
v=V
while(i>0)
  if(g[i][v]==0)
    print " 未选第i项物品"
  else if(g[i][v]==1)
    print " 选了第i项物品"
    v=v-c[i]
```

另外，采用方程的前一项或后一项也可以在输出方案的过程中根据f[i][v]的值实时地求出来，也即不须纪录g数组，将上述代码中的g[i][v]==0改成f[i][v]==f[i-1][v]，g[i][v]==1改成f[i][v]==f[i-1][v-c[i]]+w[i]也可。

- 背包问题求方案数（acwing 11）

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

输出 **最优选法的方案数**。注意答案可能很大，请输出答案模 10^9+7 的结果。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

输出格式

输出一个整数，表示 **方案数** 模 10^9+7 的结果。

数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$

输入样例

```
4 5
1 2
2 4
3 4
4 6
```

输出样例：

```
2
```

yes的练级攻略

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        System.out.println(maxValue(N, V, v, w));
    }
    private static int maxValue(int N, int V, int[] v, int[] w){
        if (N == 0 || V == 0) return 0;
        int[] dp = new int[V+1];
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            int[] count = new int[V+1];
            // 初始化，此时的 count 代表 i=0，即没有物品可以选择时
            // 什么都不选，也是一种方案
            Arrays.fill(count, 1);
            final int mod = 1000000007;
            for (int i = 0; i < N; i++){
                for (int j = V; j >= v[i]; j--){
                    if (dp[j] < dp[j - v[i]] + w[i]){
                        // 选择当前物品的收益要高，因此这里 dp 的选择必然是选择当前物品
                        count[j] = count[j - v[i]] % mod;
                    } else if (dp[j] == dp[j - v[i]] + w[i]){
                        count[j] = (count[j] + count[j - v[i]]) % mod;
                    }
                    dp[j] = Math.max(dp[j],dp[j-v[i]] + w[i]);
                }
            }
            return count[V];
        }
    }
```

（2）输出字典序最小的最优方案

这里“字典序最小”的意思是1..N号物品的选择方案排列出来以后字典序最小。以输出01背包最小字典序的方案为例。

一般而言，求一个字典序最小的最优方案，只需要在转移时注意策略。首先，子问题的定义要略改一些。我们注意到，如果存在一个选了物品1的最优方案，那么答案一定包含物品1，原问题转化为一个背包容量为v-c[1]，物品为2..N的子问题。反之，如果答案不包含物品1，则转化成背包容量仍为V，物品为2..N的子问题。不管答案怎样，子问题的物品都是以i..N而非前所述的1..i的形式来定义的，所以状态的定义和转移方程都需要改一下。但也许更简易的方法是先把物品逆序排列一下，以下按物品已被逆序排列来叙述。

在这种情况下，可以按照前面经典的状态转移方程来求值，只是输出方案的时候要注意：从N到1输入时，如果f[i][v]==f[i-v]及f[i][v]==f[i-1][f-c[i]]+w[i]同时成立，应该按照后者（即选择了物品i）来输出方案。

- 背包问题求具体方案（acwing12）

禁止商用
yes的练级攻略

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。

输出 **字典序最小的方案**。这里的字典序是指：所选物品的编号所构成的序列。物品的编号范围是 $1 \dots N$。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品数量和背包容积。

接下来有 N 行，每行两个整数 v_i, w_i，用空格隔开，分别表示第 i 件物品的体积和价值。

输出格式

输出一行，包含若干个用空格隔开的整数，表示最优解中所选物品的编号序列，且该编号序列的字典序最小。

物品编号范围是 $1 \dots N$。

数据范围

$0 < N, V \leq 1000$
$0 < v_i, w_i \leq 1000$

输入样例

```
4 5
1 2
2 4
3 4
4 6
```

输出样例：

```
1 4
```

方法 1：一维动规，但是需要二维的记忆数组（1131ms）

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        List<Integer> res = maxValue(N, V, v, w);
        for (int i = 0; i < res.size(); i++){
            if (i == res.size() - 1 ) {
                System.out.print(res.get(i));
                break;
            }
```

yes的练级攻略

```
                System.out.print(res.get(i));
                System.out.print(" ");
            }
        }
        private static List<Integer> maxValue(int N, int V, int[] v, int[] w){
            int[] dp = new int[V+1];
            int[][] memo = new int[N+1][V+1];
            for (int i = 1; i <= N; i++){
                int k = N - i;
                for (int j = V; j >= v[k]; j--){
                    int pick = dp[j - v[k]] + w[k];
                    int noPick = dp[j];
                    memo[i][j] = noPick > pick ? 0 : 1; // 选择是 1， 不选是 0
                    dp[j] = Math.max(pick, noPick);
                }
            }
            List<Integer> res = new ArrayList<>();
            int c = V;
            for (int i = N; i >= 1; i--){
                if (memo[i][c] == 1) {
                    res.add(N - i + 1);
                    c -= v[N-i];
                }
            }
            return res;
        }
}
```

方法 2：直接用二维 dp（1367ms）

```
import java.util.*;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int V = sc.nextInt();
        int[] v = new int[N];
        int[] w = new int[N];
        for (int i = 0; i < N; i++){
            v[i] = sc.nextInt();
            w[i] = sc.nextInt();
        }
        List<Integer> res = maxValue(N, V, v, w);
        for (int i = 0; i < res.size(); i++){
            if (i == res.size() - 1 ) {
                System.out.print(res.get(i));
```

禁止商用 公众号：yes的练级攻略

```
                    break;
                }
                System.out.print(res.get(i));
                System.out.print(" ");
            }
        }
        private static List<Integer> maxValue(int N, int V, int[] v, int[] w){
            int[][] dp = new int[N+1][V+1];
            for (int i = 1; i <= N; i++){
                for (int j = 1; j <= V; j++){
                    dp[i][j] = Math.max(dp[i-1][j], j >= v[N-i] ? dp[i-1][j-v[N-i]] + w[N-i] : 0);
                }
            }
            List<Integer> res = new ArrayList<>();
            int c = V;
            for (int i = N; i >= 1; i--){
                int k = N - i;
                if (c >= v[k] && dp[i-1][c-v[k]] + w[k] >= dp[i-1][c]){
                    res.add(k+1);
                    c -= v[k];
                }
            }
            return res;
        }
}
```

（3）求方案总数

对于一个给定了背包容量、物品费用、物品间相互关系（分组、依赖等）的背包问题，除了再给定每个物品的价值后求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

对于这类改变问法的问题，一般只需将状态转移方程中的max改成sum即可。例如若每件物品均是完全背包中的物品，转移方程即为

f[i][v]=sum{f[i-1][v],f[i][v-c[i]]}

初始条件f[0][0]=1。

事实上，这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

- 背包问题 4（lintcode562）【完全背包求方案数】

禁止商用 公众号：

562. 背包问题 IV

中文

给出 n 个物品, 以及一个数组, `nums[i]` 代表第i个物品的大小, 保证大小均为正数并且没有重复, 正整数 `target` 表示背包的大小, 找到能填满背包的方案数。

每一个物品可以使用无数次

样例

样例1

```
输入: nums = [2,3,6,7] 和 target = 7
输出: 2
解释:
方案有:
[7]
[2, 2, 3]
```

样例2

```
输入: nums = [2,3,4,5] 和 target = 7
输出: 3
解释:
方案有:
[2, 5]
[3, 4]
[2, 2, 3]
```

```
public class Solution {
    /**
     * @param nums: an integer array and all positive numbers, no duplicates
     * @param target: An integer
     * @return: An integer
     */
    public int backPackIV(int[] nums, int target) {
        if (nums.length == 0 || target == 0) return 0;
        int[] dp = new int[target+1];
        dp[0] = 1;
        for (int i = 0; i < nums.length; i++){
            for (int j = nums[i]; j <= target; j++) {
                dp[j] += dp[j-nums[i]];
            }
        }
        return dp[target];
    }
}
```

- 背包问题 5（lintcode563）【01 背包求方案数】

禁止商用 公众号：yes的练级攻略

563. 背包问题 V

给出 n 个物品, 以及一个数组, `nums[i]` 代表第i个物品的大小, 保证大小均为正数, 正整数 `target` 表示背包的大小, 找到能填满背包的方案数。
每一个物品只能使用一次

样例
给出候选物品集合 `[1,2,3,3,7]` 以及 target `7`

结果的集合为:
```
[7]
[1,3,3]
```
返回 `2`

```
public class Solution {
    /**
     * @param nums: an integer array and all positive numbers
     * @param target: An integer
     * @return: An integer
     */
    public int backPackV(int[] nums, int target) {
        if (nums.length == 0 || target == 0) return 0;
        int[] dp = new int[target+1];
        dp[0] = 1;
        for (int i = 0; i < nums.length; i++){
            for (int j = target; j >= nums[i]; j--) {
                dp[j] += dp[j-nums[i]];
            }
        }
        return dp[target];
    }
}
```

（4）最优方案的总数

这里的最优方案是指物品总价值最大的方案。以01背包为例。

结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求：f[i][v]意义同前述，g[i][v]表示这个子问题的最优方案的总数，则在求f[i][v]的同时求g[i][v]的伪代码如下：

```
for i=1..N
  for v=0..V
    f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}
    g[i][v]=0
    if(f[i][v]==f[i-1][v])
      inc(g[i][v],g[i-1][v]
    if(f[i][v]==f[i-1][v-c[i]]+w[i])
      inc(g[i][v],g[i-1][v-c[i]])
```

如果你是第一次看到这样的问题，请仔细体会上面的伪代码。

（5）求次优解、第 K 优解

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

对于求次优解、第K优解类的问题，如果相应的最优解问题能写出状态转移方程、用动态规划解决，那么求次优解往往可以相同的复杂度解决，第K优解则比求最优解的复杂度上多一个系数K。

其基本思想是将每个状态都表示成有序队列，将状态转移方程中的max/min转化成有序队列的合并。这里仍然以01背包为例讲解一下。

首先看01背包求最优解的状态转移方程：f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。如果要求第K优解，那么状态f[i][v]就应该是一个大小为K的数组f[i][v][1..K]。其中f[i][v][k]表示前i个物品、背包大小为v时，第k优解的值。“f[i][v]是一个大小为K的数组”这一句，熟悉C语言的同学可能比较好理解，或者也可以简单地理解为在原来的方程中加了一维。显然f[i][v][1..K]这K个数是由大到小排列的，所以我们把它认为是一个有序队列。

然后原方程就可以解释为：f[i][v]这个有序队列是由f[i-1][v]和f[i-1][v-c[i]]+w[i]这两个有序队列合并得到的。有序队列f[i-1][v]即f[i-1][v][1..K]，f[i-1][v-c[i]]+w[i]则理解为在f[i-1][v-c[i]][1..K]的每个数上加上w[i]后得到的有序队列。合并这两个有序队列并将结果（的前K项）储存到f[i][v][1..K]中的复杂度是O(K)。最后的答案是f[N][V][K]。总的复杂度是O(NVK)。

为什么这个方法正确呢？实际上，一个正确的状态转移方程的求解过程遍历了所有可用的策略，也就覆盖了问题的所有方案。只不过由于是求最优解，所以其它在任何一个策略上达不到最优的方案都被忽略了。如果把每个状态表示成一个大小为K的数组，并在这个数组中有序的保存该状态可取到的前K个最优值。那么，对于任两个状态的max运算等价于两个由大到小的有序队列的合并。

另外还要注意题目对于“第K优解”的定义，将策略不同但权值相同的两个方案是看作同一个解还是不同的解。如果是前者，则维护有序队列时要保证队列里的数没有重复的。

七、HashMap

构建实例：HashMap<包装类 1，包装类 2> map = new HashMap<>();

包装类如下：【如果是 String，可以直接填写在<>中】

Integer (int); Character (char); Byte (byte); Boolean (boolean);

Short (short); Long (long); Float (float); Double (double).

HahMap 常见方法

(1) put(K key, V value) 将键（key）/值（value）映射存放到 Map 集合中，向 HashMap 中添加元素【key 不允许重复，否则会覆盖已有的 key 对应的值】

(2) putIfAbsent(K key, V value)： 如果传入 key 对应的 value 已经存在，就返回存在的 value，不进行替换。如果不存在，就添加 key 和 value，返回 null

(3) get(Object key) 返回指定键所映射的值，没有该 key 对应的值则返回 null

(4) getOrDefault(Object key, V defaultValue)， 如果没有这个 key 的话，会返回 defaultValue

(5) size() 返回 Map 集合中数据数量

(6) clear() 清空 Map 集合

(7) isEmpty () 判断 Map 集合中是否有数据，如果没有则返回 true，否则返回 false

(8) remove(Object key) 删除 Map 集合中键为 key 的数据并返回其所对应 value 值。

(9) values() 返回 Map 集合中所有 value 组成的以 Collection 数据类型格式数据。

(10) containsKey(Object key) 判断集合中是否包含指定键，包含返回 true，否则返回 false

(11) containsValue(Object value) 判断集合中是否包含指定值，包含返回 true，否则返回 false

(12) keySet() 返回 Map 集合中所有 key 组成的 Set 集合【Set<Integer> set=map.keySet(); Iterator<Integer> it = set.iterator(); while(it.hasNext()){System.out.print(it.next() + “ “);】

禁止商用 公众号 yes的练级攻略

```
package map;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.Set;
public class Test {
        public static void main(String[] args) {
                HashMap<String,Integer> map=new HashMap<String,Integer>()
                map.put("Tom",100);
                map.put("Jim", 90);
                Set<String> set=map.keySet();
                for (String key: set) {
                        System.out.println(key+" "+map.get(key));

                }
                Iterator<String> iterator=set.iterator();
                while(iterator.hasNext()) {
                        String key=iterator.next();
                        System.out.println(key+" "+map.get(key));
                }
        }
}
```

(13) entrySet() 将 Map 集合每个 key-value 转换为一个 Entry 对象并返回由所有的 Entry 对象组成的 Set 集合

```
1  package map;
2  import java.util.HashMap;
3  import java.util.Iterator;
4  import java.util.Map.Entry;
5  import java.util.Set;
6  public class Test {
7          public static void main(String[] args) {
8                  HashMap<String,Integer> map=new HashMap<String,Integer>();
9                  map.put("Tom",100);
10                 map.put("Jim", 90);
11                 Set<Entry<String,Integer>> set=map.entrySet();
12                 for (Entry<String, Integer> entry : set) {
13                         System.out.println(entry.getKey()+" "+entry.getValue());
14                 }
15                 Iterator<Entry<String,Integer>> iterator=set.iterator();
16                 while(iterator.hasNext()) {
17                         Entry<String,Integer> entry=iterator.next();
18                         System.out.println(entry.getKey()+" "+entry.getValue());
19                 }
20         }
21 }
```

如何按值排序

```
public class TestHashMapCollections
{
    public static void main(String[] args)
    {
        Map<String, Integer> map = new HashMap<>();
        map.put("王二", 8);
        map.put("沈吴", 2);
        map.put("小菜", 3);
        map.put("大鸟", 1);

        Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
        for (Map.Entry s : entrySet)
        {
            System.out.println(s.getKey()+"--"+s.getValue());
        }

        System.out.println("============排序后============");

        //////借助list实现hashMap排序//////

        //注意 ArrayList<>() 括号里要传入map.entrySet()
        List<Map.Entry<String, Integer>> list = new ArrayList<>(map.entrySet());
```

```
        Collections.sort(list, new Comparator<Map.Entry<String, Integer>>()
        {
            @Override
            public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2)
            {
                //按照value值，重小到大排序
//                return o1.getValue() - o2.getValue();

                //按照value值，从大到小排序
//                return o2.getValue() - o1.getValue();

                //按照value值，用compareTo()方法默认是从小到大排序
                return o1.getValue().compareTo(o2.getValue());
            }
        });

        //注意这里遍历的是list，也就是我们将map.Entry放进了list，排序后的集合
        for (Map.Entry s : list)
        {
            System.out.println(s.getKey()+"--"+s.getValue());
        }

    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

输出结果如下：

沈昊--2

大鸟--1

小菜--3

王二--8

==========排序后==========

大鸟--1

沈昊--2

小菜--3

王二--8

1、有效的数独（36）

判断一个 9x9 的数独是否有效。只需要**根据以下规则**，验证已经填入的数字是否有效即可。

1. 数字 1-9 在每一行只能出现一次。
2. 数字 1-9 在每一列只能出现一次。
3. 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用 '.' 表示。

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

示例 1:

```
输入:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".",".","2",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
输出: true
```

示例 2:

```
输入:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".",".","2",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
输出: false
解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。
     但由于位于左上角的 3x3 宫内有两个 8 存在，因此这个数独是无效的。
```

略

说明:

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 给定数独序列只包含数字 `1-9` 和字符 `'.'` 。
- 给定数独永远是 `9x9` 形式的。

方法 1：HashMap（4ms）

```
class Solution {
    public boolean isValidSudoku(char[][] board) {
        //二维 hashmap 的构建
        HashMap<Character, Integer> row[] = new HashMap [9];
        HashMap<Character, Integer> col[] = new HashMap [9];
        HashMap<Character, Integer> box[] = new HashMap [9];
        //二维中的每一维的构建
        for (int i = 0; i < 9 ;i++) {
            row[i] = new HashMap<Character, Integer>();
            col[i] = new HashMap<Character, Integer>();
            box[i] = new HashMap<Character, Integer>();
```

```
            }
            for (int i = 0; i < 9 ;i++) {
                for (int j = 0; j < 9 ;j++) {
                    if (board[i][j] == '.') {
                        continue;
                    }
                    // getOrDefault (Object key, V defaultValue), 存在 key 时候用 key，没有的话就使用默认值 defaultValue
                    row[i].put(board[i][j], row[i].getOrDefault(board[i][j], 0) + 1); //value 比 key 多 1
                    col[j].put(board[i][j], col[j].getOrDefault(board[i][j], 0) + 1);
                    int numOfBox = (i/3) * 3 + (j/3);
                    box[numOfBox].put(board[i][j], box[numOfBox].getOrDefault(board[i][j], 0) + 1);
                    if (row[i].get(board[i][j]) > 1 || col[j].get(board[i][j]) > 1 || box[numOfBox].get(board[i][j]) > 1) {
                        return false;
                    }
                }
            }
            return true;
        }
    }
```

https://leetcode-cn.com/problems/valid-sudoku/solution/java-wei-yun-suan-xiang-jie-miao-dong-zuo-biao-bia/

1. 题目分析

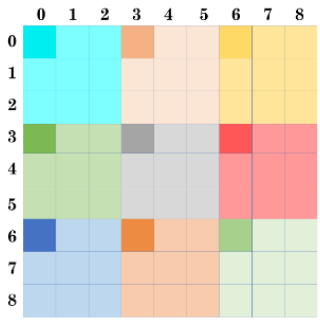
- 由于数独为9×9结构，故为了验证是否有效需要两轮循环
- 优化的关键就在于**如何通过一次两轮循环，完成横、纵、小宫的有效性判断**，需要简单推导一下坐标之间的关系

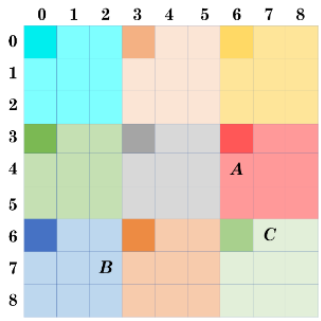
2. 简单推导

我们假设外层循环变量为i，内层循环变量为j，那么：

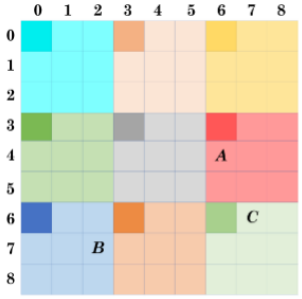
- $board[i][.]$表示第i行，$board[.][j]$表示第j列
- $board[i][j]$唯一表示第i行，第j列的元素
- 如何用坐标表示小宫内的格子？
  - 由于一共有9个小宫，每个小宫有9格，每次遍历时都会依次检查小宫里的每一格。相当于**固定一个小宫i，遍历小宫内的格子j。**
  - 如图所示，九个小宫用不同的颜色表示，每个小宫的左上角为小宫的第0个格子(深色标记)。
  - 我们设标号顺序为从左往右，从上往下。那么：
    - 格子A可以记为：第5宫，第3格$(i=5, j=3)$，坐标为$(4, 6)$
    - 格子B可以记为：第6宫，第5格$(i=6, j=5)$，坐标为$(7, 2)$
    - 格子C可以记为：第8宫，第1格$(i=8, j=1)$，坐标为$(6, 7)$

更多硬核内容就在公众号「yes 的练级攻略」





- 下面我们求出小宫的标号(第i宫)、小宫内格子的标号(第j格)的坐标



以点A(第5宫，第3格)为例。

(1)第i宫，第0格的坐标

由于点A在第5宫，第5宫相对于第0宫：

- 横坐标(行)：向下偏移(行偏移)了1 (5/3)个宫，一个宫占3行，故行偏移为3*1=3。
- 纵坐标(列)：向右偏移(列偏移)了2(5%3)个宫，一个宫占3列，故列偏移为 3*2=6 。
- 故第 5 宫第 0 格坐标为 (3,6)

(2)第j格相对于第0格的偏移

- 横坐标(行)：向下偏移(行偏移)了 1 (3/3)行
- 纵坐标(列)：向右偏移(列偏移)了 0 (3%3)列
- 故第3格相对于第0格的偏移为 (1,0)

(3)第i宫，第j格的坐标

第 5 宫第 3 格坐标为 (3+1,6+0)，即 (4,6)

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

- 首先求第i宫，第0格的坐标
  - 横坐标(行)：第i宫向下偏移(行偏移)了$i/3$个宫，一个小宫占3行，故横坐标为$3*(i/3)$
  - 纵坐标(列)：第i宫向右偏移(列偏移)了$i\%3$个宫，一个小宫占3列，故横坐标为$3*(i\%3)$
  - 故第i宫，第0格的坐标为

$$(3*(i/3),3*(i\%3))$$

- 然后求第j格相对于第0格的偏移
  - 横坐标(行)：第j格相对于第0格，偏移了$j/3$行
  - 纵坐标(列)：第j格相对于第0格，偏移了$j\%3$列
  - 故第j格相对于第0格的偏移为：

$$(j/3,j\%3)$$

- 综上所述，第i宫，第j格的坐标为：

$$(3*(i/3)+j/3,3*(i\%3)+j\%3)$$

- 由嵌套循环的知识(**外层固定，内层遍历**)：
  - $borad[i][j]$表示了对固定的一行i，遍历列j(即每次检查一整行)
  - $board[j][i]$表示了对固定的一列i，遍历行j(即每次检查一整列)
  - $board[3*(i/3)+j/3][3*(i\%3)+j\%3]$表示了对固定的宫$i$，遍历格$j$(即每次检查一个宫)

3. 位运算

- 基本知识：
  - 与运算$(a\&b)$：a,b均为1时，返回1，否则返回0
  - 异或运算$(a\ \hat{}\ b)$：a,b不同时为0或1时，返回1，否则返回0
- 本题可以使用一个9位二进制数判断数字是否被访问。第k位数为1代表已加入，为0代表未加入
- 更新方式(记九位数为val，传入的数字为n)：
  - 判断是否加入：将九位数右移位n位，与1进行与运算
    - 结果为0：未加入，将传入的数字加入九位数
    - 结果为1：已加入，返回 `false`
  - 将传入的数字加入九位数：将1左移位n位，与val异或即可
- 例子：对于数字1010010000，其第$4,7,9$位为1，表示当前$4,7,9$已经访问过了
  - 新来数字为3：
    - 将1010010000右移3位得到1010010，与1进行与运算，结果为0，未访问过。
    - 将1左移位3位得到1000，异或后得到1010011000
  - 新来数字为4：
    - 将1010010000右移4位得到101001，与1进行与运算，结果为1，访问过。
    - 返回 `false`

时间复杂度$O(1)$，空间复杂度$O(1)$。

方法 2：位运算（2ms）

```
class Solution {
    public boolean isValidSudoku(char[][] board) {
        for (int i = 0; i < 9; i++){
            int row = 0, col = 0, sqr = 0;
            for (int j = 0; j < 9; j++){
                // 由于传入为char，需要转换为int，减48
                int r = board[i][j] - 48;
                int c = board[j][i] - 48;
                int s = board[3 * (i / 3) + j / 3][3 * (i % 3) + j % 3] - 48;
                // "."的ASCII 码为46，故小于0 代表着当前符号位"."，不用讨论
                if (r > 0) row = sodokuer(r, row);
                if (c > 0) col = sodokuer(c, col);
                if (s > 0) sqr = sodokuer(s, sqr);
                if (row == -1 || col == -1 || sqr == -1) return false;
```

```
            }
        }
        return true;
    }
    private int sodokuer(int n, int val){
        return ((val >> n) & 1) == 1 ? -1 : val ^ (1 << n);
    }
}
```

2、猜数字游戏（299）

你正在和你的朋友玩 猜数字（Bulls and Cows）游戏：你写下一个数字让你的朋友猜。每次他猜测后，你给他一个提示，告诉他有多少位数字和确切位置都猜对了（称为“Bulls”，公牛），有多少位数字猜对了但是位置不对（称为“Cows”，奶牛）。你的朋友将会根据提示继续猜，直到猜出秘密数字。

请写出一个根据秘密数字和朋友的猜测数返回提示的函数，用 A 表示公牛，用 B 表示奶牛。

请注意秘密数字和朋友的猜测数都可能含有重复数字。

示例 1:

输入: secret = "1807", guess = "7810"

输出: "1A3B"

解释: 1 公牛和 3 奶牛。公牛是 8，奶牛是 0, 1 和 7。

示例 2:

输入: secret = "1123", guess = "0111"

输出: "1A1B"

解释: 朋友猜测数中的第一个 1 是公牛，第二个或第三个 1 可被视为奶牛。

说明: 你可以假设秘密数字和朋友的猜测数都只包含数字，并且它们的长度永远相等。

通过次数 14,395 | 提交次数 29,894

在真实的面试中遇到过这道题? 是 否

https://leetcode-cn.com/problems/bulls-and-cows/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by--55/

第一点，因为 map 中的 key 只有 0 到 9，所以我们可以用数组取代 map 。

第二点，我们可以把第二步用 map 统计个数的代码放到第一个 for 循环中，当位置不相等的时候再统计个数，这样最后就不用 B - A 了。

方法 1：map 数组

```
class Solution {
    public String getHint(String secret, String guess) {
        int A = 0;
        int[] mapS = new int[10];
        int[] mapG = new int[10];
        for (int i = 0; i < guess.length(); i++) {
            if (secret.charAt(i) == guess.charAt(i)) {
                A++;
            } else {
                mapS[secret.charAt(i) - '0']++;
                mapG[guess.charAt(i) - '0']++;
```

```
            }
        }
        int B = 0;
        for (int i = 0; i < 10; i++) {
            B += Math.min(mapS[i], mapG[i]);
        }
        return A + "A" + B + "B";
    }
}
```

方法 2：只有一个 numbers 数组

```
class Solution {
    public String getHint(String secret, String guess) {
        int bulls = 0;
        int cows = 0;
        int[] numbers = new int[10];
        for (int i = 0; i<secret.length(); i++) {
            int s = secret.charAt(i) - '0';
            int g = guess.charAt(i) - '0';
            if (s == g) bulls++;
            else {
                //当前数小于 0，说明之前在 guess 中出现过，和 secret 当前的数匹配
                if (numbers[s] < 0) cows++;
                //当前数大于 0，说明之前在 secret 中出现过，和 guess 当前的数匹配
                if (numbers[g] > 0) cows++;
                //secret 中的数，计数加 1
                numbers[s] ++;
                //guess 中的数，计数减 1
                numbers[g] --;
            }
        }
        return bulls + "A" + cows + "B";
    }
}
```

3、珠玑妙算（面试题 16.15）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.15. 珠玑妙算

难度 简单 14 收藏 分享 切换为英文 关注 反馈

珠玑妙算游戏（the game of master mind）的玩法如下。

计算机有4个槽，每个槽放一个球，颜色可能是红色（R）、黄色（Y）、绿色（G）或蓝色（B）。例如，计算机可能有RGGB 4种（槽1为红色，槽2、3为绿色，槽4为蓝色）。作为用户，你试图猜出颜色组合。打个比方，你可能会猜YRGB。要是猜对某个槽的颜色，则算一次“猜中”；要是只猜对颜色但槽位猜错了，则算一次“伪猜中”。注意，“猜中”不能算入“伪猜中”。

给定一种颜色组合 `solution` 和一个猜测 `guess`，编写一个方法，返回猜中和伪猜中的次数 `answer`，其中 `answer[0]` 为猜中的次数，`answer[1]` 为伪猜中的次数。

示例：

```
输入： solution="RGBY",guess="GGRR"
输出： [1,1]
解释： 猜中1次，伪猜中1次。
```

提示：

- `len(solution) = len(guess) = 4`
- `solution` 和 `guess` 仅包含 `"R"`, `"G"`, `"B"`, `"Y"` 这4种字符

https://leetcode-cn.com/problems/master-mind-lcci/solution/bie-dian-kai-zhe-yi-pian-ji-bai-100-by-hengzai-28/

解题思路

方法零：因为solution和guess的长度都只有4，所以嵌套for循环暴力解的话，也很快。

方法一：

1. 设置一个长26的数组map（目的是将RYGB对应到数组的index中）
2. for循环遍历solution和guess
3. 如果solution和guess对应元素相等，则直接real++
4. 若不相等，判断map中sol元素是否小于0（代表之前存过guess的元素），存在则fake++，然后更新map[sol - 'A']++;
5. 对map中的gue元素做同等判断
6. 返回答案ans。

方法 1：数组

```
class Solution {
    public int[] masterMind(String solution, String guess) {
        int fake = 0, real = 0;
        int[] map = new int[26];
        for(int i = 0; i < 4; i++){
            char sol = solution.charAt(i), gue = guess.charAt(i);
            if(sol == gue) real++;
            else{
                if(map[sol - 'A'] < 0) fake++;
                map[sol - 'A']++;

                if(map[gue - 'A'] > 0) fake++;
                map[gue - 'A']--;
            }
        }
        int[] ans = {real, fake};
        return ans;
```

```
    }
}
```

方法二：

1. 使用了HashMap，将solution的元素保存到map中（包含元素数量）
2. 然后判断map中是否有guess的元素，有则fake++，注意要更新元素数量
3. 在来个for循环判断一致的数据real，最后fake - real等于伪猜对

这些题目都是简单的，但方法和思路就在那，还是要多思考和多做优化的工作。

方法 2：hashmap

```
class Solution {
    public int[] masterMind(String solution, String guess) {
        HashMap<Character, Integer> map = new HashMap<Character, Integer>();
        for(char c : solution.toCharArray()){
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        int fake = 0, real = 0;
        for(char c : guess.toCharArray()){
            if(map.containsKey(c) && map.get(c) > 0){
                fake++;
                map.put(c, map.get(c) - 1);
            }
        }
        for(int i = 0; i < 4; i++){
            if(solution.charAt(i) == guess.charAt(i))
                real++;
        }
        int[] ans = {real, fake - real};
        return ans;
    }
}
```

4、存在重复元素（217）

更多硬核内容就在公众号「yes 的练级攻略」

217. 存在重复元素

难度 简单 275 收藏 分享 切换为英文 关注 反馈

给定一个整数数组，判断是否存在重复元素。

如果任意一值在数组中出现至少两次，函数返回 `true` 。如果数组中每个元素都不相同，则返回 `false` 。

示例 1:

```
输入: [1,2,3,1]
输出: true
```

示例 2:

```
输入: [1,2,3,4]
输出: false
```

示例 3:

```
输入: [1,1,1,3,3,4,3,2,4,2]
输出: true
```

https://leetcode-cn.com/problems/contains-duplicate/solution/cun-zai-zhong-fu-yuan-su-by-leetcode/

方法二：排序 【通过】

直觉

如果存在重复元素，排序后它们应该相邻。

算法

本方法使用排序算法。由于比较排序算法，如**堆排序**，可以在最坏情况下具有 $O(n \log n)$ 的时间复杂度。因此，排序经常是很好的预处理方法。排序之后，我们可以扫描已排序的数组,以查找是否有任何连续的重复元素。

Java

```
public boolean containsDuplicate(int[] nums) {
    Arrays.sort(nums);
    for (int i = 0; i < nums.length - 1; ++i) {
        if (nums[i] == nums[i + 1]) return true;
    }
    return false;
}
```

复杂度分析

- 时间复杂度：$O(n \log n)$。
  排序的复杂度是 $O(n \log n)$，扫描的复杂度是 $O(n)$。整个算法主要由排序过程决定，因此是 $O(n \log n)$。
- 空间复杂度：$O(1)$。
  这取决于具体的排序算法实现，通常而言，使用 `堆排序` 的话，是 $O(1)$。

注意

此处的算法实现对原始数组进行排序，修改了原始数组。通常，除非调用方清楚输入数据将被修改，否则不应该随意修改输入数据。可以先复制 `nums` ，然后对副本进行操作。

更多硬核内容就在公众号「yes 的练级攻略」

方法 1：排序

```
class Solution {
    public boolean containsDuplicate(int[] nums) {
        if (nums.length <= 1) return false;
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 1; ++i){
            if (nums[i] == nums[i + 1]) return true;
        }
        return false;
    }
}
```

方法三：哈希表 【通过】

直觉

利用支持快速搜索和插入操作的动态数据结构。

算法

从方法一中我们知道，对无序数组的查找操作的时间复杂度为 $O(n)$，而我们会重复调用查找操作。因此，使用搜索时间更快的数据结构将加快整个算法的速度。

有许多数据结构常用作动态集合,如二进制搜索树和哈希表。这里我们需要的操作是 `search` 和 `insert`。对于平衡二叉搜索树（Java 中的 TreeSet 或 TreeMap），`search` 和 `insert` 的时间复杂度均为 $O(\log n)$。对于哈希表（Java 中的 HashSet 或 HashMap），`search` 和 `insert` 的平均时间复杂度为 $O(1)$。因此，通过使用哈希表，我们可以达到在线性时间复杂度解决问题。

Java

```
public boolean containsDuplicate(int[] nums) {
    Set<Integer> set = new HashSet<>(nums.length);
    for (int x: nums) {
        if (set.contains(x)) return true;
        set.add(x);
    }
    return false;
}
```

复杂度分析

- 时间复杂度：$O(n)$。
  `search()` 和 `insert()` 各自使用 n 次，每个操作耗费常数时间。
- 空间复杂度：$O(n)$。哈希表占用的空间与元素数量是线性关系。

注意

对于一些特定的 n 不太大的测试样例，本方法的运行速度可能会比方法二更慢。这是因为哈希表在维护其属性时有一些开销。要注意，程序的实际运行表现和 Big-O 符号表示可能有所不同。Big-O 只是告诉我们在 充分大的输入下，算法的相对快慢。因此，在 n 不够大的情况下，$O(n)$ 的算法也可以比 $O(n \log n)$的更慢。

方法 2：哈希表

```
class Solution {
    public boolean containsDuplicate(int[] nums) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int num : nums) {
            if (map.put(num, 1) != null) return true;
        }
        return false;
```

```
    }
}
```

5、两个数组的交集（349）

349. 两个数组的交集

难度 简单 192 收藏 分享 切换为英文 关注 反馈

给定两个数组，编写一个函数来计算它们的交集。

示例 1：

```
输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]
```

示例 2：

```
输入：nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出：[9,4]
```

说明：

- 输出结果中的每个元素一定是唯一的。
- 我们可以不考虑输出结果的顺序。

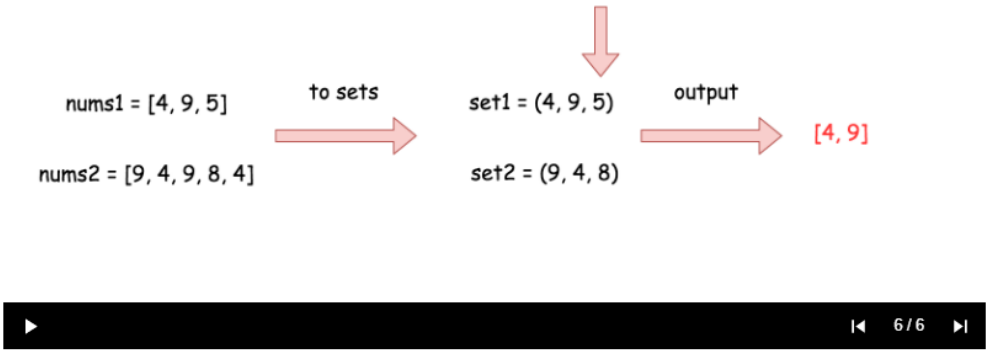
https://leetcode-cn.com/problems/intersection-of-two-arrays/solution/liang-ge-shu-zu-de-jiao-ji-by-leetcode/

方法一：两个set

最直观的方法是迭代并检查第一个数组 `nums1` 中的每个值是否也存在于 `nums2` 中。如果存在，则将值添加到输出。这种方法的时间复杂度为 $O(n \times m)$，其中 `n` 和 `m` 分别为数组 `nums1` 和 `nums2` 的长度。

为了在线性时间内解决这个问题，我们使用集合 `set` 这一数据结构，该结构可以提供平均时间复杂度为 $O(1)$ 的 `in/contains` 操作（用于测试某一元素是否为该集合的成员）。

本解法先将两个数组都转换为集合，然后迭代较小的集合，检查其中的每个元素是否同样存在于较大的集合中。平均情况下，这种方法的时间复杂度为 $O(n+m)$。



更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(m+n)$，其中 `n` 和 `m` 是数组的长度。将 `nums1` 转换为集合需要 $O(n)$ 的时间，类似地，将 `nums2` 转换为集合需要 $O(m)$ 的时间。而在平均情况下，集合的 `in/contains` 操作只需要 $O(1)$ 的时间。
- 空间复杂度：$O(m+n)$，最坏的情况是数组中的所有元素都不同。

方法 1：两个 Set

```
class Solution {
    public int[] set_intersection(HashSet<Integer> set1, HashSet<Integer> set2)
 {
        int [] output = new int[set1.size()];
        int idx = 0;
        for (Integer s : set1)
        if (set2.contains(s)) output[idx++] = s;

        return Arrays.copyOf(output, idx);
    }
    public int[] intersection(int[] nums1, int[] nums2) {
        HashSet<Integer> set1 = new HashSet<Integer>();
        for (Integer n : nums1) set1.add(n);
        HashSet<Integer> set2 = new HashSet<Integer>();
        for (Integer n : nums2) set2.add(n);

        if (set1.size() < set2.size()) return set_intersection(set1, set2);
        else return set_intersection(set2, set1);
    }
}
```

方法二：内置函数

如果使用内置函数：那么平均情况下，时间复杂度为 $O(n+m)$；而最坏的情况下，时间复杂度是 $O(n \times m)$。

复杂度分析

- 时间复杂度：一般情况下是 $O(m+n)$，最坏情况下是 $O(m \times n)$。
- 空间复杂度：最坏的情况是 $O(m+n)$，数组中的所有元素都不同。

方法 2：内置函数

```
class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
        HashSet<Integer> set1 = new HashSet<Integer>();
        for (Integer n : nums1) set1.add(n);
        HashSet<Integer> set2 = new HashSet<Integer>();
        for (Integer n : nums2) set2.add(n);

        set1.retainAll(set2);

        int [] output = new int[set1.size()];
```

```
        int idx = 0;
        for (int s : set1) output[idx++] = s;
        return output;
    }
}
```

https://leetcode-cn.com/problems/intersection-of-two-arrays/solution/duo-chong-jie-fa-jie-jue-349-liang-ge-shu-zu-de-ji/

方法2：双指针

1. 先将$nums1$ 与$nums2$ 排序，然后游走两个指针，情况都写出来了，没有用$else$
2. 时间复杂度：$O(nlogn)$

方法 3：双指针

```
class Solution {
    public int[] intersection(int[] nums1, int[] nums2) {
        Set<Integer> set = new HashSet<>();
        Arrays.sort(nums1);
        Arrays.sort(nums2);
        int i = 0, j = 0;
        while (i < nums1.length && j < nums2.length) {
            if (nums1[i] == nums2[j]) {
                set.add(nums1[i]);
                i++;
                j++;
            } else if (nums1[i] < nums2[j]) i++;
            else if (nums1[i] > nums2[j]) j++;
        }
        int[] res = new int[set.size()];
        int index = 0;
        for (int num : set) res[index++] = num;
        return res;
    }
}
```

6、两个数组的交集二（350）

更多硬核内容就在公众号「yes 的练级攻略」

350. 两个数组的交集 II

难度 简单 285 收藏 分享 切换为英文 关注 反馈

给定两个数组，编写一个函数来计算它们的交集。

示例 1:

```
输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2,2]
```

示例 2:

```
输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [4,9]
```

说明:

- 输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。
- 我们可以不考虑输出结果的顺序。

进阶:

- 如果给定的数组已经排好序呢？你将如何优化你的算法？
- 如果 *nums1* 的大小比 *nums2* 小很多，哪种方法更优？
- 如果 *nums2* 的元素存储在磁盘上，磁盘内存是有限的，并且你不能一次加载所有的元素到内存中，你该怎么办？

https://leetcode-cn.com/problems/intersection-of-two-arrays/solution/duo-chong-jie-fa-jie-jue-349-liang-ge-shu-zu-de-ji/

方法 1：hashmap

复杂度分析:

- 时间复杂度：$O(M+N)$ 其中M和 N是两个数组的长度，$HashMap$操作的时间复杂度为$O(1)$
- 空间复杂度：$O(min(M|N))$ 其中M和 N是两个数组的长度,准备了一个$list$来装重复数组，其最大的长度不超过 $min(M|N)$

```java
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        List<Integer> list = new ArrayList<>();
        Map<Integer, Integer> map = new HashMap<>();
        for (int num : nums1) {
            map.put(num, map.getOrDefault(num, 0) + 1);
        }
        for (int num : nums2) {
            if (map.containsKey(num) && map.get(num) > 0) {
                list.add(num);
                map.put(num, map.get(num) - 1);
            }
        }
        int[] res = new int[list.size()];
        for (int i = 0; i < list.size(); i++) {
            res[i] = list.get(i);
        }
```

```
        return res;
    }
}
```

https://leetcode-cn.com/problems/intersection-of-two-arrays-ii/solution/liang-chong-jie-fa-javashi-xian-by-lyl0724-2/

思路二：时间复杂度为O(nlogn)，额外空间复杂度为O(1)。

首先对两个数组进行排序，然后就是双指针，p1指向nums1，p2指向nums2：

1. 如果nums1[p1] == nums2[p2]，说明俩数组中都有这个数，是其交集，所以将它丢入list中。
2. 如果不等，则移动小的那个指针。

方法 2：双指针

```
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        Arrays.sort(nums1);
        Arrays.sort(nums2);
        List<Integer> list = new ArrayList<>();
        int p1 = 0, p2 = 0;
        while(p1 < nums1.length && p2 < nums2.length) {
            if(nums1[p1] < nums2[p2]) p1++;
            else if(nums1[p1] > nums2[p2]) p2++;
            else {
                list.add(nums1[p1]);
                p1++;
                p2++;
            }
        }
        int[] res = new int[list.size()];
        for(int i = 0; i < res.length; i++) res[i] = list.get(i);
        return res;
    }
}
```

7、设计推特（355）

355. 设计推特

难度 中等 134 收藏 分享 切换为英文 关注 反馈

设计一个简化版的推特(Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近十条推文。你的设计需要支持以下的几个功能：

1. **postTweet(userId, tweetId)**: 创建一条新的推文
2. **getNewsFeed(userId)**: 检索最近的十条推文。每个推文都必须是由此用户关注的人或者是用户自己发出的。推文必须按照时间顺序由最近的开始排序。
3. **follow(followerId, followeeId)**: 关注一个用户
4. **unfollow(followerId, followeeId)**: 取消关注一个用户

示例:

```
Twitter twitter = new Twitter();

// 用户1发送了一条新推文 (用户id = 1, 推文id = 5).
twitter.postTweet(1, 5);

// 用户1的获取推文应当返回一个列表，其中包含一个id为5的推文.
twitter.getNewsFeed(1);

// 用户1关注了用户2.
twitter.follow(1, 2);

// 用户2发送了一个新推文 (推文id = 6).
twitter.postTweet(2, 6);

// 用户1的获取推文应当返回一个列表，其中包含两个推文，id分别为 -> [6, 5].
// 推文id6应当在推文id5之前，因为它是在5之后发送的.
twitter.getNewsFeed(1);

// 用户1取消关注了用户2.
twitter.unfollow(1, 2);

// 用户1的获取推文应当返回一个列表，其中包含一个id为5的推文.
// 因为用户1已经不再关注用户2.
twitter.getNewsFeed(1);
```

这里「推特」，可以理解为中国的「微博」、「朋友圈」、「力扣」，真正的数据数需要存在数据库里的，并且还要加上一些非关系型的数据库（redis 等），不能是放在内存里的，这里只是简化了需求。

分析：

- 这是一类系统设计问题（上周我们做过的 LFU 缓存也是属于这一类问题），通常简化了很多需求，只要题目意思理解清楚，一般情况下不难写出，难在编码的细节和调试；
- 这里需求 3 和需求 4，只需要维护「我关注的人的 id 列表」 即可，不需要维护「谁关注了我」，由于不需要维护有序性，为了删除和添加方便， 「我关注的人的 id 列表」需要设计成哈希表（HashSet），而每一个人的和对应的他关注的列表存在一个哈希映射（HashMap）里；
- 最复杂的是需求 2 getNewsFeed(userId):

  每一个人的推文和他的 id 的关系，依然是存放在一个哈希表里；

  对于每一个人的推文，只有顺序添加的需求，没有查找、修改、删除操作，因此可以使用线性数据结构，链表或者数组均可；

  使用数组就需要在尾部添加元素，还需要考虑扩容的问题（使用动态数组）；

  使用链表就得在头部添加元素，由于链表本身就是动态的，无需考虑扩容；

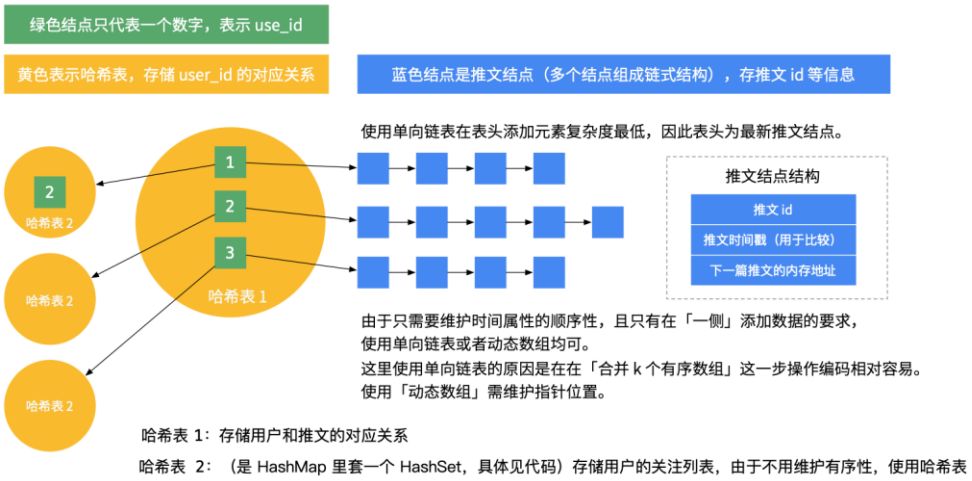
检索最近的十条推文，需要先把这个用户关注的人的列表拿出来（实际上不是这么做的，请看具体代码，或者是「多路归并」的过程），然后再合并，排序以后选出 Top10，这其实是非常经典的「多路归并」的问题（「力扣」第 23 题：合并 K 个排序链表），这里需要使用的数据结构是优先队列，所以在上一步在存储推文列表的时候使用单链表是合理的，并且

应该存储一个时间戳字段，用于比较哪一队的队头元素先出列。

剩下的就是一些细节问题了，例如需要查询关注人（包括自己）的最近十条推文，所以要把自己的推文也放进优先队列里。在出队（优先队列）、入队的时候需要考虑是否为空。
编写对这一类问题，需要仔细调试，并且养成良好的编码习惯，是很不错的编程练习问题。
总结：

- 如果需要维护数据的时间有序性，链表在特殊场景下可以胜任。因为时间属性通常来说是相对固定的，而不必去维护顺序性；
- 如果需要动态维护数据有序性，「优先队列」（堆）是可以胜任的，「力扣」上搜索「堆」（heap）标签，可以查看类似的问题；
- 设计类问题也是一类算法和数据结构的问题，并且做这一类问题有助于我们了解一些数据结构的大致思想和细节，「力扣」上搜索「设计」标签，可以查看类似的问题；
- 做完这个问题，不妨仔细思考一下这里使用链表存储推文的原因。

推文设计示意图



参考链接：
https://leetcode-cn.com/problems/design-twitter/solution/ha-xi-biao-lian-biao-you-xian-dui-lie-java-by-liwe/
355 具体代码

```
import java.util.*;
public class Twitter {
    // 推文类，是一个单链表
    private class Tweet {
        private int id;
        private int timestamp;
        private Tweet next;
        public Tweet(int id, int timestamp) {
            this.id = id;
            this.timestamp = timestamp;
        }
    }
    //用户 id 和推文的对应关系
```

```
    private Map<Integer, Tweet> twitter;
    //用户 id 和他关注的用户列表的对应关系
    private Map<Integer, Set<Integer>> followings;
    //全局使用的时间戳字段，用户每发布一条推文之前 + 1
    private static int timestamp = 0;
    //合并 k 组推文使用的数据结构（可以在方法里创建使用），声明成全局变量非必需，视个人情况使用
    private static PriorityQueue<Tweet> maxHeap;
    /* Initialize your data structure here. */
    public Twitter() {
        twitter =  new HashMap<>();
        followings = new HashMap<>();
        maxHeap =  new PriorityQueue<>((o1, o2) -> -o1.timestamp + o2.timestamp);
    }
    /* Compose a new tweet. */
    public void postTweet(int userId, int tweetId) {
        timestamp++;
        if (twitter.containsKey(userId)) {
            Tweet oldHead = twitter.get(userId);
            Tweet newHead = new Tweet(tweetId, timestamp);
            newHead.next = oldHead;
            twitter.put(userId, newHead);
        } else {
            twitter.put(userId, new Tweet(tweetId, timestamp));
        }
    }

    /* Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent. */
    public List<Integer> getNewsFeed(int userId) {
        //全局变量使用前需要清空
        maxHeap.clear();
        if (twitter.containsKey(userId)) {
            maxHeap.offer(twitter.get(userId));
        }
        Set<Integer> followingsList = followings.get(userId);
        if(followingsList != null && followingsList.size() > 0) {
            for (Integer followingId: followingsList) {
                Tweet t = twitter.get(followingId);
                if (t != null) {
                    maxHeap.offer(t);
```

```
                }
            }
        }
        List<Integer> res = new ArrayList<>(10);
        int count = 0;
        while (! maxHeap.isEmpty() && count < 10) {
            Tweet head = maxHeap.poll();
            res.add(head.id);
            // 这里最好的操作应该是 replace，但是 Java 没有提供
            if (head.next != null) {
                maxHeap.offer(head.next);
            }
            count++;
        }
        return res;
    }

    /* Follower follows a followee. If the operation is invalid, it sho
uld be a no-op.
     * followerId : 发起关注者 id; followeeId 被关注者 id*/
    public void follow(int followerId, int followeeId) {
        if (followerId == followeeId) {
            return;
        }
        // 获取我自己的关注列表
        Set<Integer> followingList = followings.get(followerId);
        if (followingList == null) {
            Set<Integer> init = new HashSet<>();
            init.add(followeeId);
            followings.put(followerId, init);
        } else {
            if (followingList.contains(followeeId)) {
                return;
            }
            followingList.add(followeeId);
        }
    }

    /* Follower unfollows a followee. If the operation is invalid, it s
hould be a no-op. */
    public void unfollow(int followerId, int followeeId) {
        if (followerId == followeeId) {
            return;
        }
```

```
        // 获取我自己的关注列表
        Set<Integer> followingsList = followings.get(followerId);
        if (followingsList == null) {
            return;
        }
        // 这里删除之前无需做判断，因为查找是否存在以后，就可以删除，反正删除之前都要查找
        followingsList.remove(followeeId);
    }
    // public static void main(String[] args) {
    //  // TODO Auto-generated method stub
    //   Twitter twitter = new Twitter();
    //   // 用户 1 发送了一条新推文 (用户 id = 1, 推文 id = 5).
    //   twitter.postTweet(1, 5);
    //  // 用户 1 的获取推文应当返回一个列表，其中包含一个 id 为 5 的推文.
    //   List<Integer> res1 = twitter.getNewsFeed(1);
    //      System.out.println(res1);
    //     // 用户 1 关注了用户 2.
    //      twitter.follow(1, 2);
    //     // 用户 2 发送了一个新推文 (推文 id = 6).
    //      twitter.postTweet(2, 6);
    //      // 用户 1 的获取推文应当返回一个列表，其中包含两个推文，id 分别为 -> [6, 5].
    //      // 推文 id6 应当在推文 id5 之前，因为它是在 5 之后发送的.
    //      List<Integer> res2 = twitter.getNewsFeed(1);
    //      System.out.println(res2);
    //      twitter.unfollow(1, 2);
    //      List<Integer> res3 = twitter.getNewsFeed(1);
    //      System.out.println(res3);
    // }

}
```

8、O（1）时间插入、删除和获取随机元素（380）

更多硬核内容就在公众号「yes 的练级攻略」

380. 常数时间插入、删除和获取随机元素

难度 中等 134 收藏 分享 切换为英文 关注 反馈

设计一个支持在*平均*时间复杂度 **O(1)** 下，执行以下操作的数据结构。

1. `insert(val)`：当元素 val 不存在时，向集合中插入该项。
2. `remove(val)`：元素 val 存在时，从集合中移除该项。
3. `getRandom`：随机返回现有集合中的一项。每个元素应该有**相同的概率**被返回。

示例：

```
// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();

// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);

// 返回 false ，表示集合中不存在 2 。
randomSet.remove(2);

// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);

// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();

// 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomSet.remove(1);

// 2 已在集合中，所以返回 false 。
randomSet.insert(2);

// 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
randomSet.getRandom();
```

https://leetcode-cn.com/problems/insert-delete-getrandom-o1/solution/chang-shu-shi-jian-cha-ru-shan-chu-he-huo-qu-sui-j/

```
class RandomizedSet {
  Map<Integer, Integer> dict;
  List<Integer> list;
  Random rand;

  /** Initialize your data structure here. */
  public RandomizedSet() {
    dict = new HashMap();
    list = new ArrayList();
    rand = new Random();
  }

  /** Inserts a value to the set. Returns true if the set did not alrea
dy contain the specified element. */
  public boolean insert(int val) {
    if (dict.containsKey(val)) return false;
    dict.put(val, list.size());
```

```
    list.add(list.size(), val);
    return true;
  }

  /** Removes a value from the set. Returns true if the set contained t
he specified element. */
  public boolean remove(int val) {
    if (! dict.containsKey(val)) return false;

    // move the last element to the place idx of the element to delete
    int lastElement = list.get(list.size() - 1);
    int idx = dict.get(val);
    list.set(idx, lastElement);
    dict.put(lastElement, idx);
    // delete the last element
    list.remove(list.size() - 1);
    dict.remove(val);
    return true;
  }

  /** Get a random element from the set. */
  public int getRandom() {
    return list.get(rand.nextInt(list.size()));
  }
}
```

9、O（1）时间插入、删除和获取随机元素 - 允许重复（381）

设计一个支持在*平均*时间复杂度 **O(1)** 下，执行以下操作的数据结构。

注意: 允许出现重复元素。

1. `insert(val)`：向集合中插入元素 val。
2. `remove(val)`：当 val 存在时，从集合中移除一个 val。
3. `getRandom`：从现有集合中随机获取一个元素。每个元素被返回的概率应该与其在集合中的数量呈线性相关。

示例:

```
// 初始化一个空的集合。
RandomizedCollection collection = new RandomizedCollection();

// 向集合中插入 1 。返回 true 表示集合不包含 1 。
collection.insert(1);

// 向集合中插入另一个 1 。返回 false 表示集合包含 1 。集合现在包含 [1,1] 。
collection.insert(1);

// 向集合中插入 2 ，返回 true 。集合现在包含 [1,1,2] 。
collection.insert(2);

// getRandom 应当有 2/3 的概率返回 1 ，1/3 的概率返回 2 。
collection.getRandom();

// 从集合中删除 1 ，返回 true 。集合现在包含 [1,2] 。
collection.remove(1);

// getRandom 应有相同概率返回 1 和 2 。
collection.getRandom();
```

https://leetcode-cn.com/problems/insert-delete-getrandom-o1-duplicates-allowed/solution/hashmaphashsetzui-he-li-jie-fa-by-mei-ming-zi-/

```
import java.util.*;

/**
 * Created by zlw on 2020/5/14.
 */
public class RandomizedCollection {
    private List<Integer> vals;
    private Map<Integer,Set<Integer>> indexs;
    private int index;

    /** Initialize your data structure here. */
    public RandomizedCollection() {
        vals = new ArrayList<>();
        indexs = new HashMap<>();
        index = 0;
    }

    /** Inserts a value to the collection. Returns true if the collection did n
ot already contain the specified element. */
    public boolean insert(int val) {
        vals.add(val);
        //插入下标
        if(!indexs.containsKey(val)){
            indexs.put(val,new HashSet<>());
        }
        indexs.get(val).add(index ++);
        return indexs.get(val).size() == 1;
    }

    /** Removes a value from the collection. Returns true if the collection con
tained the specified element. */
    public boolean remove(int val) {
        //找到最后一个删除的下标
        if(!indexs.containsKey(val) || indexs.get(val).size() == 0) return fals
e;
        Set<Integer> is = indexs.get(val);
        //得到一个下标
        int randomIndex = is.iterator().next();
        is.remove(randomIndex);
        //用最后一个值覆盖他
        int last = vals.get(index - 1);
        vals.set(randomIndex,last);
        //如果要删除的下标不等于最后一个元素下标，需要修改最后元素的下标映射
        if(randomIndex != index - 1){
```

```
            //修改最后元素的下标映射
            Set<Integer> set = indexs.get(last);
            set.remove(index - 1);
            set.add(randomIndex);
        }
        //删除最后一个元素
        vals.remove(--index);
        return true;
    }
    /** Get a random element from the collection. */
    public int getRandom() {
        if(index <= 0) throw new RuntimeException("空");
        int i = (int) Math.floor(Math.random() * index);
        return vals.get(i);
    }
}
/**
 * Your RandomizedCollection object will be instantiated and called as
such:
 * RandomizedCollection obj = new RandomizedCollection();
 * boolean param_1 = obj.insert(val);
 * boolean param_2 = obj.remove(val);
 * int param_3 = obj.getRandom();
 */
```

10、元音拼写检查器（966）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

966. 元音拼写检查器

难度 中等 15 收藏 分享 切换为英文 关注 反馈

在给定单词列表 `wordlist` 的情况下，我们希望实现一个拼写检查器，将查询单词转换为正确的单词。

对于给定的查询单词 `query`，拼写检查器将会处理两类拼写错误：

- 大小写：如果查询匹配单词列表中的某个单词（**不区分大小写**），则返回的正确单词与单词列表中的大小写相同。
  - 例如：`wordlist = ["yellow"], query = "YellOw": correct = "yellow"`
  - 例如：`wordlist = ["Yellow"], query = "yellow": correct = "Yellow"`
  - 例如：`wordlist = ["yellow"], query = "yellow": correct = "yellow"`
- 元音错误：如果在将查询单词中的元音（`'a'`、`'e'`、`'i'`、`'o'`、`'u'`）分别替换为任何元音后，能与单词列表中的单词匹配（**不区分大小写**），则返回的正确单词与单词列表中的匹配项大小写相同。
  - 例如：`wordlist = ["YellOw"], query = "yollow": correct = "YellOw"`
  - 例如：`wordlist = ["YellOw"], query = "yeellow": correct = ""` (无匹配项)
  - 例如：`wordlist = ["YellOw"], query = "yllw": correct = ""` (无匹配项)

此外，拼写检查器还按照以下优先级规则操作：

- 当查询完全匹配单词列表中的某个单词（**区分大小写**）时，应返回相同的单词。
- 当查询匹配到大小写问题的单词时，您应该返回单词列表中的第一个这样的匹配项。
- 当查询匹配到元音错误的单词时，您应该返回单词列表中的第一个这样的匹配项。
- 如果该查询在单词列表中没有匹配项，则应返回空字符串。

给出一些查询 `queries`，返回一个单词列表 `answer`，其中 `answer[i]` 是由查询 `query = queries[i]` 得到的正确单词。

示例：

输入： `wordlist = ["KiTe","kite","hare","Hare"], queries = ["kite","Kite","KiTe","Hare","HARE","Hear","hear","keti","keet","keto"]`
输出： `["kite","KiTe","KiTe","Hare","hare","","","KiTe","","KiTe"]`

提示：

1. `1 <= wordlist.length <= 5000`
2. `1 <= queries.length <= 5000`
3. `1 <= wordlist[i].length <= 7`
4. `1 <= queries[i].length <= 7`
5. `wordlist` 和 `queries` 中的所有字符串仅由英文字母组成。

https://leetcode-cn.com/problems/vowel-spellchecker/solution/jian-dan-xie-fa-yi-kan-jiu-dong-by-keytian/

```
class Solution {
    public String[] spellchecker(String[] wordlist, String[] queries) {
        HashSet<String> set = new HashSet<>();
        HashMap<String,String> map1 = new HashMap<>();
        HashMap<String,String> map2 = new HashMap<>();
        String[] ans = new String[queries.length];
        for (String s : wordlist) {
            set.add(s);
            //toLowerCase() 将大写字符转换为小写。
            String s1 = s.toLowerCase();
            map1.putIfAbsent(s1,s);
            //makeVowel(s1) 元音字母替换
            String s2 = makeVowel(s1);
            map2.putIfAbsent(s2,s);
        }
        for (int i = 0; i < queries.length; i++) {
            if (set.contains(queries[i])) {
                ans[i] = queries[i];
```

```
                continue;
            }
            String s1 = queries[i].toLowerCase();
            if (map1.containsKey(s1)) {
                ans[i] = map1.get(s1);
                continue;
            }
            String s2 = makeVowel(s1);
            if (map2.containsKey(s2)) {
                ans[i] = map2.get(s2);
                continue;
            }
            ans[i] = "";
        }
        return ans;
    }
    public String makeVowel (String s1){
        StringBuffer s2 = new StringBuffer();
        for (char c : s1.toCharArray()){
            if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u
') {
                s2.append("*");
            } else {
                s2.append(c);
            }
        }
        return s2.toString();
    }
}
```

11、面试题 16.24 数对和

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.24. 数对和

难度 中等 7 收藏 分享 切换为英文 关注 反馈

设计一个算法，找出数组中两数之和为指定值的所有整数对。一个数只能属于一个数对。

示例 1:

```
输入: nums = [5,6,5], target = 11
输出: [[5,6]]
```

示例 2:

```
输入: nums = [5,6,5,6], target = 11
输出: [[5,6],[5,6]]
```

提示:

- `nums.length <= 100000`

https://leetcode-cn.com/problems/pairs-with-sum-lcci/solution/ha-xi-biao-shuang-zhi-zhen-liang-chong-fang-fa-shi/

第一种实现方法：利用哈希表辅助实现。
思路：把出现过的元素当成key，元素出现的次数当成value存到map中，如果遍历到某一个元素num，查看target - num元素是否出现过，如果出现过，那么可以组成一个整数对，注意这里一个数只能属于一个数对，所以判断target - num元素出现的次数，如果为1，直接remove即可，否则更新target - num元素出现的次数-1。

方法 1：hashmap

```
class Solution {
    public List<List<Integer>> pairSums(int[] nums, int target) {
        //key:数组的元素;value:该元素出现的次数
        HashMap <Integer, Integer> map = new HashMap<>();
        List<List<Integer>> ans = new ArrayList<>();
        for (int num : nums) {
            Integer count = map.get(target-num);
            if (count != null) {
                ans.add(Arrays.asList(num, target-num));
                if (count == 1) {
                    map.remove(target-num);
                } else {
                    map.put(target-num, --count);
                }
            } else {
                map.put(num,map.getOrDefault(num,0) + 1);
            }
        }
        return ans;
    }
}
```

第二种实现方法：使用双指针实现，注意要先对数组进行排序处理。

方法 2：双指针

```
class Solution {
    public List<List<Integer>> pairSums(int[] nums, int target) {
        //对数组进行排序
        Arrays.sort(nums);
        List<List<Integer>> ans = new LinkedList<>();
        int left = 0, right = nums.length - 1;
        while (left < right) {
            //两个指针所指的两个元素和
            int sum = nums[left] + nums[right];
            //如果两个的和小于目标值，那么 left 指针向右走一步继续寻找
            if (sum < target)
                ++left;
            //如果两个的和大于目标值，那么 right 指针向左走一步继续寻找
            else if (sum > target)
                --right;
            //如果刚好等于要找的 target 值，那么加入结果集中，并且 left 指针和 right 指针分别向右和向左走一步(因为一个数只能属于一个数对)
            else
                ans.add(Arrays.asList(nums[left++], nums[right--]));

        }
        return ans;
    }
}
```

12、特征提取（lintcode1135）

1135. 特征提取　　中文 English

小明想从猫咪的视频里挖掘一些猫咪的运动信息。为了提取运动信息，他需要从视频的每一帧提取"猫咪特征"。
一个猫咪特征是一个两维的vector<x, y>。如果x_1=x_2 and y_1=y_2，那么这俩是同一个特征。
如果喵咪特征连续一致，可以认为喵咪在运动。也就是说，如果特征<a, b>在持续帧里出现，那么它将构成特征运动。比如，特征<a, b>在第2/3/4/7/8帧出现，那么该特征将形成两个特征运动2-3-4 和7-8。
现在，给定每一帧的特征，特征的数量可能不一样。小明期望能找到最长的特征运动。

样例
示例1:

输入:
features:
[[2,1,1,2,2],
[2,1,1,1,4],
[2,1,1,2,2],
[2,2,2,1,4],
[0],
[0],
[1,1,1],
[1,1,1]]
输出:3
解释:特征<1,1>在连续的帧中连续出现3次，相比其他特征连续出现的次数大，所以输出3
注意事项
每行代表一帧。其中，第一个数字是该帧的特征个数，接下来的数字每两个为一对<x,y>
特征总数不超过 10^5

```
public class Solution {
    /**
     * @param frames: A series of frames
     * @return: Find the longest feature movement
     */
    public int FeatureExtraction(int[][] frames) {
        //存储上一行键值对出现的次数信息
        HashMap<String,Integer>         preFeaTimes         =         new
HashMap<String,Integer>();
        //存储本行键值对在上一行的基础上出现的次数信息
        HashMap<String,Integer>         curFeaTimes         =         new
HashMap<String,Integer>();
        //保存结果
        int count = 0;
        for (int i = 0; i < frames.length; i++) {
            for(int j = 0; j < frames[i][0]; j++) {
                String  xy  =  String.valueOf(frames[i][j  *  2  +  1])  +
String.valueOf(frames[i][j * 2 + 2]);
                if (preFeaTimes.containsKey(xy)){
                    //判断如果本次输入的键值对在上一行出现过，那么该键值对的次
数
                    //等于上一行中记录的出现次数+1
                    curFeaTimes.put(xy, preFeaTimes.get(xy) + 1);
                } else {
                    //如果该键值对没有出现过或者没在上一行出现过，说明该键值对
                    //要么根本没出现过要么中间断了，所以重新置为 1
                    curFeaTimes.put(xy, 1);
                }
                count = Math.max(count, curFeaTimes.get(xy));
            }
            //清理：是因为本行已经遍历完了，键值对的更新信息已经保存在 feaTimes
中了
            preFeaTimes.clear();
            //因为要继续判断下一行的键值对信息，下一行是依赖本行的
            preFeaTimes.putAll(curFeaTimes);
            //本行的清理
            curFeaTimes.clear();
        }
        return count;
    }
}
```

13、LRU 缓存机制（146）（面试题 16.25）（最近最少使用 LRU）

146. LRU缓存机制

难度 中等 733 收藏 分享 切换为英文 关注 反馈

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果关键字 (key) 存在于缓存中，则获取关键字的值（总是正数），否则返回 -1。
写入数据 put(key, value) - 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字/值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶:

你是否可以在 **O(1)** 时间复杂度内完成这两种操作?

示例:

```
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得关键字 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得关键字 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

链接：
https://leetcode-cn.com/problems/lru-cache/solution/lruhuan-cun-ji-zhi-by-leetcode-solution/

LRU 缓存机制可以通过哈希表辅以双向链表实现，我们用一个哈希表和一个双向链表维护所有在缓存中的键值对。

- 双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的。
- 哈希表即为普通的哈希映射（HashMap），通过缓存数据的键映射到其在双向链表中的位置。

这样以来，我们首先使用哈希表进行定位，找出缓存项在双向链表中的位置，随后将其移动到双向链表的头部，即可在 O(1)的时间内完成 get 或者 put 操作。具体的方法如下：

- 对于 get 操作，首先判断 key 是否存在：
  - 如果 key 不存在，则返回 -1；
  - 如果 key 存在，则 key 对应的节点是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。
- 对于 put 操作，首先判断 key 是否存在：
  - 如果 key 不存在，使用 key 和 value 创建一个新的节点，在双向链表的头部添加该节点，并将 key 和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；
  - 如果 key 存在，则与 get 操作类似，先通过哈希表定位，再将对应的节点的值更新为 value，并将该节点移到双向链表的头部。

上述各项操作中，访问哈希表的时间复杂度为 O(1)，在双向链表的头部添加节点、在双向链表的尾部删除节点的复杂度也为 O(1)。而将一个节点移到双向链表的头部，可以分成「删除该节点」和「在双向链表的头部添加节点」两步操作，都可以在 O(1)时间内完成。

小贴士：在双向链表的实现中，使用一个伪头部（dummy head）和伪尾部（dummy tail）标记界限，这样在添加节点和删除节点的时候就不需要检查相邻的节点是否存在。

作者：LeetCode-Solution

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：对于 `put` 和 `get` 都是 $O(1)$。
- 空间复杂度：$O(\text{capacity})$，因为哈希表和双向链表最多存储 $\text{capacity}+1$ 个元素。

方法 1：hashmap+双向链表

```
public class LRUCache {
    class DLinkedNode {
        int key;
        int value;
        DLinkedNode prev;
        DLinkedNode next;
        public DLinkedNode() {}
        public DLinkedNode(int _key, int _value) {key = _key; value = _
value;}
    }

    private Map<Integer, DLinkedNode> cache = new HashMap<Integer, DLin
kedNode>();
    private int size;
    private int capacity;
    private DLinkedNode head, tail;

    public LRUCache(int capacity) {
        this.size = 0;
        this.capacity = capacity;
        // 使用伪头部和伪尾部节点
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            return -1;
        }
        // 如果 key 存在，先通过哈希表定位，再移到头部
        moveToHead(node);
        return node.value;
    }
```

```
public void put(int key, int value) {
    DLinkedNode node = cache.get(key);
    if (node == null) {
        // 如果 key 不存在，创建一个新的节点
        DLinkedNode newNode = new DLinkedNode(key, value);
        // 添加进哈希表
        cache.put(key, newNode);
        // 添加至双向链表的头部
        addToHead(newNode);
        ++size;
        if (size > capacity) {
            // 如果超出容量，删除双向链表的尾部节点
            DLinkedNode tail = removeTail();
            // 删除哈希表中对应的项
            cache.remove(tail.key);
            --size;
        }
    }
    else {
        // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
        node.value = value;
        moveToHead(node);
    }
}

private void addToHead(DLinkedNode node) {
    node.prev = head;
    node.next = head.next;
    head.next.prev = node;
    head.next = node;
}

private void removeNode(DLinkedNode node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

private void moveToHead(DLinkedNode node) {
    removeNode(node);
    addToHead(node);
}

private DLinkedNode removeTail() {
    DLinkedNode res = tail.prev;
```

```
            removeNode(res);
            return res;
        }
    }
```

https://leetcode-cn.com/problems/lru-cache/solution/yuan-yu-linkedhashmapyuan-ma-by-jeromememory/

简单介绍LinkedHashMap（跟题目有关的知识点）

HashMap 大家都清楚，底层是 数组 + 红黑树 + 链表 （不清楚也没有关系），同时其是无序的，而 LinkedHashMap 刚好就比 HashMap 多这一个功能，就是其提供 **有序**，并且，LinkedHashMap的有序可以按两种顺序排列，一种是按照插入的顺序，一种是按照**读取**的顺序（这个题目的示例就是告诉我们要按照读取的顺序进行排序），而其内部是靠 **建立一个双向链表** 来维护这个顺序的，在每次插入、删除后，都会调用一个函数来进行 **双向链表的维护** ，准确的来说，是有三个函数来做这件事，这三个函数都统称为 **回调函数** ，这三个函数分别是：

- **void afterNodeAccess(Node<K,V> p) { }**
  其作用就是在访问元素之后，将该元素放到双向链表的尾巴处(所以这个函数只有在按照读取的顺序的时候才会执行)，之所以提这个，是建议大家去看看，如何优美的实现在双向链表中将指定元素放入链尾！
- **void afterNodeRemoval(Node<K,V> p) { }**
  其作用就是在删除元素之后，将元素从双向链表中删除，还是非常建议大家去看看这个函数的，很优美的方式在双向链表中删除节点！
- **void afterNodeInsertion(boolean evict) { }**
  这个才是我们题目中会用到的，在插入新元素之后，需要回调函数判断是否需要移除一直不用的某些元素！

其次，我再介绍一下 LinkedHashMap 的构造函数！
其主要是两个构造方法，一个是继承 HashMap ，一个是可以选择 accessOrder 的值(默认 false，代表按照插入顺序排序)来确定是按插入顺序还是读取顺序排序。

```
/**
 * //调用父类HashMap的构造方法。
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the default initial capacity (16) and load factor (0.75).
 */
public LinkedHashMap() {
    super();
    accessOrder = false;
}
// 这里的 accessOrder 默认是为false，如果要按读取顺序排序需要将其设为 true
// initialCapacity 代表 map 的 容量，loadFactor 代表加载因子（默认即可）
public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}
```

思路 & 代码

下面是我自己在分析 LinkedHashMap 源码时做的一些笔记，应该是比较清楚的，主体意思就是我们要继承 LinkedHashMap，然后复写 removeEldestEntry()函数，就能拥有我们自己的缓存策略！

Java

```
// 在插入一个新元素之后，如果是按插入顺序排序，即调用newNode()中的linkNodeLast()完成
// 如果是按照读取顺序排序，即调用 afterNodeAccess()完成
// 那么这个方法是干嘛的呢，这个就是著名的 LRU 算法啦
// 在插入完成之后，需要回调函数判断是否需要移除某些元素！
// LinkedHashMap 函数部分源码

/**
 * 插入新节点才会触发该方法，因为只有插入新节点才需要内存
 * 根据 HashMap 的 putVal 方法，evict 一直是 true
 * removeEldestEntry 方法表示移除规则，在 LinkedHashMap 里一直返回 false
 * 所以在 LinkedHashMap 里这个方法相当于什么都不做
 */
void afterNodeInsertion(boolean evict) { // possibly remove eldest
    LinkedHashMap.Entry<K,V> first;
    // 根据条件判断是否移除最近最少被访问的节点
    if (evict && (first = head) != null && removeEldestEntry(first)) {
        K key = first.key;
        removeNode(hash(key), key, null, false, true);
    }
}

// 移除最近最少被访问条件之一，通过覆盖此方法可实现不同策略的缓存
// LinkedHashMap是默认返回false的，我们可以继承LinkedHashMap然后复写该方法即可
// 例如 LeetCode 第 146 题就是采用该种方法，直接 return size() > capacity;
protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return false;
}
```

通过上述代码，我们就已经知道了只要复写 removeEldestEntry() 即可，而条件就是 map 的大小不超过 给定的容量，超过了就得使用 LRU 了！然后根据题目给定的语句构造和调用：

Java

```
/**
 * LRUCache 对象会以如下语句构造和调用:
 * LRUCache obj = new LRUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

很明显我们只需要直接继承父类的put函数即可，因为题目没有特殊要求，故可以不写！至于 get() 函数，题目是有要求的！

> 获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

所以我们可以调用 LinkedHashMap 中的 getOrDefault()，完美符合这个要求，即当key不存在时会返回默认值 -1。

至此，我们就基本完成了本题的要求，只要写一个构造函数即可，答案的 `super(capacity, 0.75F, true);`，没看过源码的小伙伴可能不太清楚这个构造函数，这就是我上文讲的 LinkedHashMap 中的常用的第二个构造方法，具体大家可以看我上面代码的注释！

//标准的如何在双向链表中将指定元素放入队尾
// LinkedHashMap 中覆写
//访问元素之后的回调方法

/**
 * 1. 使用 get 方法会访问到节点，从而触发调用这个方法

```
 * 2. 使用 put 方法插入节点，如果 key 存在，也算要访问节点，从而触发该方法
 * 3. 只有 accessOrder 是 true 才会调用该方法
 * 4. 这个方法会把访问到的最后节点重新插入到双向链表结尾
 */
void afterNodeAccess(Node<K,V> e) { // move node to last
    // 用 last 表示插入 e 前的尾节点
    // 插入 e 后 e 是尾节点，所以也是表示 e 的前一个节点
    LinkedHashMap.Entry<K,V> last;
    //如果是访问序，且当前节点并不是尾节点
    //将该节点置为双向链表的尾部
    if (accessOrder && (last = tail) != e) {
        // p: 当前节点
        // b: 前一个节点
        // a: 后一个节点
        // 结构为: b <=> p <=> a
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        // 结构变成: b <=> p <- a
        p.after = null;

        // 如果当前节点 p 本身是头节点，那么头结点要改成 a
        if (b == null)
            head = a;
        // 如果 p 不是头尾节点，把前后节点连接，变成: b -> a
        else
            b.after = a;

        // a 非空，和 b 连接，变成: b <- a
        if (a != null)
            a.before = b;
        // 如果 a 为空，说明 p 是尾节点, b 就是它的前一个节点，符合 last 的定义
        // 这个 else 没有意义，因为最开头 if 已经确保了 p 不是尾结点了，自然 after 不
会是 null
        else
            last = b;

        // 如果这是空链表, p 改成头结点
        if (last == null)
            head = p;
        // 否则把 p 插入到链表尾部
        else {
            p.before = last;
            last.after = p;
        }
```

```
            tail = p;
            ++modCount;
        }
    }
    void afterNodeRemoval(Node<K,V> e) { // 优美的一笔，学习一波如何在双向链表中删除节点
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        // 将 p 节点的前驱后后继引用置空
        p.before = p.after = null;
        // b 为 null，表明 p 是头节点
        if (b == null)
            head = a;
        else
            b.after = a;
        // a 为 null，表明 p 是尾节点
        if (a == null)
            tail = b;
        else
            a.before = b;
    }
```

方法 2：源于 LinkedHashMap 源码

```
class LRUCache extends LinkedHashMap<Integer, Integer>{
    private int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75F, true);
        this.capacity = capacity;
    }

    public int get(int key) {
        return super.getOrDefault(key, -1);
    }

    // 这个可不写
    public void put(int key, int value) {
        super.put(key, value);
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {
        return size() > capacity;
    }
}
```

14、LFU 缓存（460）【双向链表】（最不经常使用 LFU）

更多硬核内容就在公众号「yes 的练级攻略」

460. LFU缓存

难度 困难 248 收藏 分享 切换为英文 关注 反馈

请你为 最不经常使用（LFU）缓存算法设计并实现数据结构。它应该支持以下操作：`get` 和 `put`。

- `get(key)` - 如果键存在于缓存中，则获取键的值（总是正数），否则返回 -1。
- `put(key, value)` - 如果键已存在，则变更其值；如果键不存在，请插入键值对。当缓存达到其容量时，则应该在插入新项之前，使最不经常使用的项无效。在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除最久未使用的键。

「项的使用次数」就是自插入该项以来对其调用 `get` 和 `put` 函数的次数之和。使用次数会在对应项被移除后置为 0 。

进阶：
你是否可以在 **O(1)** 时间复杂度内执行两项操作？

示例：

```
LFUCache cache = new LFUCache( 2 /* capacity (缓存容量) */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回 1
cache.put(3, 3);    // 去除 key 2
cache.get(2);       // 返回 -1 (未找到key 2)
cache.get(3);       // 返回 3
cache.put(4, 4);    // 去除 key 1
cache.get(1);       // 返回 -1 (未找到 key 1)
cache.get(3);       // 返回 3
cache.get(4);       // 返回 4
```

禁止商用

https://leetcode-cn.com/problems/lfu-cache/solution/java-13ms-shuang-100-shuang-xiang-lian-biao-duo-ji/

一、$O(1)$ 解法

下面要说的 $O(1)$ 的 3 种 Java 写法其实是 1 种解法，因为具体实现细节中使用的数据结构不同，导致性能有所差异。为方便理解，下面 3 个实现，**性能从略挫逐步优化**：

1. `HashMap<Integer, Node> cache` 存缓存的内容；`min` 是最小访问频次；`HashMap<Integer, LinkedHashSet<Node>> freqMap` 存每个访问频次对应的 Node 的双向链表（写法 1 为了方便，直接用了 JDK 现有的 LinkedHashSet，其实现了 1 条双向链表贯穿哈希表中的所有 Entry，支持以插入的先后顺序对原本无序的 HashSet 进行迭代）
2. `HashMap<Integer, Node> cache` 存缓存的内容；`min` 是最小访问频次；`HashMap<Integer, DoublyLinkedList>freqMap` 存每个访问频次对应的 Node 的双向链表（写法 2 与写法 1 一样，只不过将 JDK 自带的 LinkedHashSet 双向链表实现改成了自定义的双向链表 DoublyLinkedList，减少了一些哈希相关的耗时）
3. `HashMap<Integer, Node> cache` 存缓存的内容；将写法 1 写法 2 中的 freqMap 不再用 HashMap 来表示，而是直接用双向链表 `DoublyLinkedList firstLinkedList; DoublyLinkedList lastLinkedList`，省去了一些哈希相关的耗时，也不需要用 min 来存储最小频次了，lastLinkedList.pre 这条 DoublyLinkedList 即为最小频次对应的 Node 双向链表，lastLinkedList.pre.tail.pre 这个 Node 即为最小频次的双向链表中的所有 Node 中最先访问的 Node，即容量满了后要删除的 Node。

方法 1：甜姨

```
class LFUCache {
    Map<Integer, Node> cache;  // 存储缓存的内容，Node 中除了 value 值外，还有 key、freq、所在 doublyLinkedList、所在 doublyLinkedList 中的 postNode、所在 doublyLinkedList 中的 preNode，具体定义在下方。
    DoublyLinkedList firstLinkedList; // firstLinkedList.post 是频次最大的双向链表
    DoublyLinkedList lastLinkedList;  // lastLinkedList.pre 是频次最小的双向链表，满了之后删除 lastLinkedList.pre.tail.pre 这个 Node 即为频次最小且访问最早的 Node
    int size;
    int capacity;
```

```
    public LFUCache(int capacity) {
        cache = new HashMap<>(capacity);
        firstLinkedList = new DoublyLinkedList();
        lastLinkedList = new DoublyLinkedList();
        firstLinkedList.post = lastLinkedList;
        lastLinkedList.pre = firstLinkedList;
        this.capacity = capacity;
    }
    public int get(int key) {
        Node node = cache.get(key);
        if (node == null) return -1;
        // 该 key 访问频次+1
        freqInc(node);
        return node.value;
    }
    public void put(int key, int value) {
        if (capacity == 0) return;
        Node node = cache.get(key);
        // 若 key 存在，则更新 value，访问频次+1
        if (node != null) {
            node.value = value;
            freqInc(node);
        } else {
            // 若 key 不存在
            if (size == capacity) {
                // 如果缓存满了，删除 lastLinkedList.pre 这个链表（即表示最小频次的链表）中的 tail.pre 这个 Node（即最小频次链表中最先访问的 Node），如果该链表中的元素删空了，则删掉该链表。
                cache.remove(lastLinkedList.pre.tail.pre.key);
                lastLinkedList.removeNode(lastLinkedList.pre.tail.pre);
                size--;
                if (lastLinkedList.pre.head.post == lastLinkedList.pre.tail) {
                    removeDoublyLinkedList(lastLinkedList.pre);
                }
            }
            // cache 中 put 新 Key-Node 对儿，并将新 node 加入表示 freq 为 1 的 DoublyLinkedList 中，若不存在 freq 为 1 的 DoublyLinkedList 则新建。
            Node newNode = new Node(key, value);
            cache.put(key, newNode);
            if (lastLinkedList.pre.freq != 1) {
                DoublyLinkedList newDoublyLinedList = new DoublyLinkedList(1);
                addDoublyLinkedList(newDoublyLinedList, lastLinkedList.pre);
                newDoublyLinedList.addNode(newNode);
            } else {
```

```
                lastLinkedList.pre.addNode(newNode);
            }
            size++;
        }
    }
    /**
     * node 的访问频次 + 1
     */
    void freqInc(Node node) {
        // 将 node 从原 freq 对应的双向链表里移除，如果链表空了则删除链表。
        DoublyLinkedList linkedList = node.doublyLinkedList;
        DoublyLinkedList preLinkedList = linkedList.pre;
        linkedList.removeNode(node);
        if (linkedList.head.post == linkedList.tail) {
            removeDoublyLinkedList(linkedList);
        }
        // 将 node 加入新 freq 对应的双向链表，若该链表不存在，则先创建该链表。
        node.freq++;
        if (preLinkedList.freq != node.freq) {
            DoublyLinkedList newDoublyLinedList = new DoublyLinkedList(node.fre
q);
            addDoublyLinkedList(newDoublyLinedList, preLinkedList);
            newDoublyLinedList.addNode(node);
        } else {
            preLinkedList.addNode(node);
        }
    }
    /**
     * 增加代表某 1 频次的双向链表
     */
    void addDoublyLinkedList(DoublyLinkedList newDoublyLinedList, DoublyLinkedL
ist preLinkedList) {
        newDoublyLinedList.post = preLinkedList.post;
        newDoublyLinedList.post.pre = newDoublyLinedList;
        newDoublyLinedList.pre = preLinkedList;
        preLinkedList.post = newDoublyLinedList;
    }
    /**
     * 删除代表某 1 频次的双向链表
     */
    void removeDoublyLinkedList(DoublyLinkedList doublyLinkedList) {
        doublyLinkedList.pre.post = doublyLinkedList.post;
        doublyLinkedList.post.pre = doublyLinkedList.pre;
    }
```

```
}
class Node {
    int key;
    int value;
    int freq = 1;
    Node pre; // Node 所在频次的双向链表的前继 Node
    Node post; // Node 所在频次的双向链表的后继 Node
    DoublyLinkedList doublyLinkedList;  // Node 所在频次的双向链表
    public Node() {}
    public Node(int key, int value) {
        this.key = key;
        this.value = value;
    }
}
class DoublyLinkedList {
    int freq; // 该双向链表表示的频次
    DoublyLinkedList pre;  // 该双向链表的前继链表（pre.freq < this.freq）
    DoublyLinkedList post; // 该双向链表的后继链表 (post.freq > this.freq)
    Node head; // 该双向链表的头节点，新节点从头部加入，表示最近访问
    Node tail; // 该双向链表的尾节点，删除节点从尾部删除，表示最久访问
    public DoublyLinkedList() {
        head = new Node();
        tail = new Node();
        head.post = tail;
        tail.pre = head;
    }
    public DoublyLinkedList(int freq) {
        head = new Node();
        tail = new Node();
        head.post = tail;
        tail.pre = head;
        this.freq = freq;
    }
    void removeNode(Node node) {
        node.pre.post = node.post;
        node.post.pre = node.pre;
    }
    void addNode(Node node) {
        node.post = head.post;
        head.post.pre = node;
        head.post = node;
        node.pre = head;
        node.doublyLinkedList = this;
    }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
}
/**
 * Your LFUCache object will be instantiated and called as such:
 * LFUCache obj = new LFUCache(capacity);
 * int param_1 = obj.get(key);
 * obj.put(key,value);
 */
```

LFU （Least Frequently Used）缓存机制（看访问次数）

- 在缓存满的时候，删除缓存里使用次数最少的元素，然后在缓存中放入新元素；
- 数据的访问次数很重要，**访问次数越多**，就越不容易被删除；
- 根据题意，「当存在平局（即两个或更多个键具有相同使用频率）时，最近最少使用的键将被去除」，即在「访问次数」相同的情况下，按照时间顺序，先删除在缓存里时间最久的数据。

说明：本题其实就是在「力扣」第 146 题：LRU缓存机制 的基础上，在删除策略里多考虑了一个维度（「访问次数」）的信息。

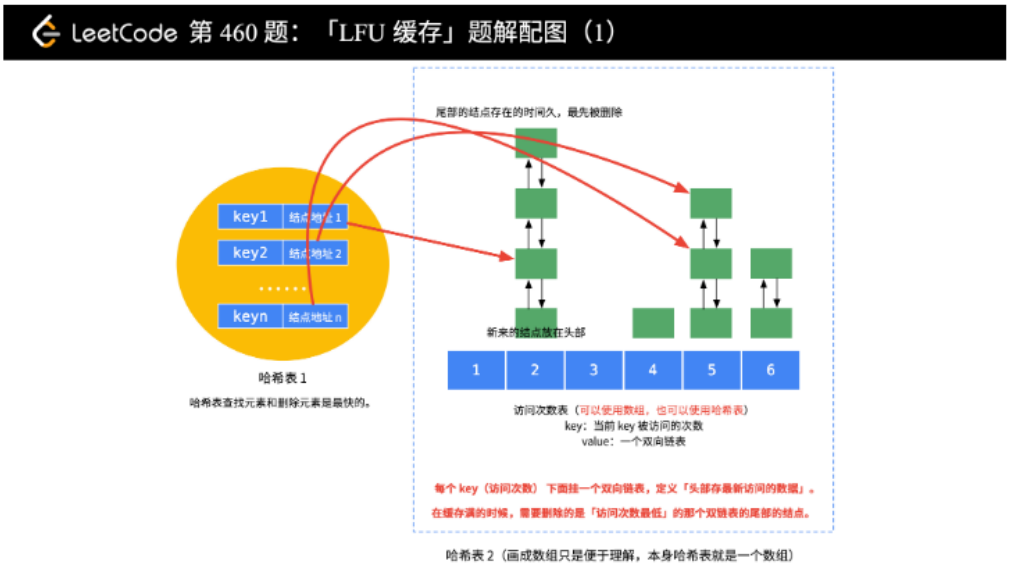
核心思想：先考虑访问次数，在访问次数相同的情况下，再考虑缓存的时间。

思路

- 由于题目的时间复杂度要求 $O(1)$，空间肯定不能省，存取数据时间性能最好的就是哈希表，因此底层的数据结构一定是一个哈希表；
- 又由于缓存大小有限制，删除策略是「先看访问频次，再看访问时间」，所以需要记录每个数据访问的频次；
- 「删除某个数据」得 $O(1)$，访问某个数据，时间优先级得提前（提前到当前频次最高），**这样的数据结构符合在头尾访问数据最快，并且删除其中一个结点也得是 $O(1)$，这种数据结构是「双向链表」**；
- 「链表」结点得记录：1、value，2、key（在哈希表里删除的时候用得上），3、访问次数信息，以便知道下一个访问次数是多少；4、前驱结点引用，5、后继结点引用；
- 哈希表存储的 key 就是题目的 key，方便快速查询和删除，value 是结点的引用，方便对结点进行操作。

这样一套设计下来，题目中要求的操作就是 $O(1)$ 了。

yes的练级攻略



(图是用软件 OmniGraffle 画的。不太复杂的规则图形，用 PPT 也可以画出来。)

- 每次访问一个已经存在的元素的时候：应该先把结点类从当前所属的访问次数双链表里删除，然后再添加到它「下一个访问次数」的双向链表的头部。

编码说明：

- 应该先弄清楚思路，再编码；
- 可以设计成「双向链表」的尾部存储较新访问的结点，头部是当前频次最旧的结点。双向链表在结构上是对称的，编码的时候注意保持语义一致；
- 「双向链表」的常见操作是使用两个虚拟结点，一个访问头部最快，另一个访问尾部最快，这个技巧其实在「单链表」中我们已经见过，叫「哨兵结点」；
- 链表中的相关操作建议画图去思考实现，否则凭空想象一些指针变量的指向操作容易出错；
- 如果在编码的过程中，发现一些数据需要维护，可以适当增加属性；
- 在一些操作中相同的操作，应该考虑抽取成公用的方法；
- 在编码完成以后，需要注意调试，这一步是很花时间的，也没有过多的技巧，添加打印输出语句。

注意：

1. 以下代码由于本人水平有限，封装还不够好，仅供参考；
2. 下面的代码细节特别多，读者**浏览即可，不建议模仿，应该尝试自己编写完成，相信是一个不错的编程练习**；
3. `ListNode` 结点类，哈希表要用到，双向链表也会用到，因此设计在外面；
4. Java 里的 `LinkedList` 就是双向链表，这里只是为了练习，定制化一些操作，因此手写。

参考代码：

感谢 @zhangkyou 和 @lzhlyle 朋友的提醒，原来我的代码维护的是 `maxFrequent`，时间复杂度没有到 $O(1)$，应该维护的是 `minFrequent`。需要在以下两处维护这个变量：

- 插入一个新结点的时候，因为这个结点之前肯定没有被访问过，此时设置 `minFrequent = 1`，相当于归位了；
- 当 `put` 和 `get` 的时候，都有把当前链表里的结点移到另一个链表结点的操作。如果移除的那个链表恰好是访问次数最小链表，并且移除以后链表的结点个数为 `0`，`minFrequent` 需要加 1。

https://leetcode-cn.com/problems/lfu-cache/solution/ha-xi-biao-shuang-xiang-lian-biao-java-by-liweiwei/

方法 2：weiwei【包含输入输出】

```
import java.util.HashMap;
import java.util.Map;
public class LFUCache {
    private Map<Integer, ListNode> map; //key 就是题目中的 key, value 是结点类
```

```
    private Map<Integer, DoubleLinkedList> frequentMap; //访问次数哈希表，使用 ListNode[] 也可以，不过要占用很多空间
    private Integer capacity; //外部传入的容量大小
    private Integer minFrequent = 1; //全局最高访问次数，删除最少使用访问次数的结点时会用到
    public LFUCache(int capacity) {
        // 显式设置哈希表的长度 = capacity 和加载因子 = 1 是为了防止哈希表扩容带来的性能消耗
        // 这一步操作在理论上的可行之处待讨论，实验下来效果是比较好的
        map = new HashMap<>(capacity, 1);
        frequentMap = new HashMap<>();
        this.capacity = capacity;
    }
    //get 一次操作，访问次数就增加 1；从原来的链表调整到访问次数更高的链表的表头
    public int get(int key) {
        // 测试测出来的，capacity 可能传 0
        if (capacity == 0) {
            return -1;
        }
        if (map.containsKey(key)) {
            // 获得结点类
            ListNode listNode = removeListNode(key);
            // 挂接到新的访问次数的双向链表的头部
            int frequent = listNode.frequent;
            addListNode2Head(frequent, listNode);
            return listNode.value;
        } else {
            return -1;
        }
    }
    public void put(int key, int value) {
        if (capacity == 0) return;
        // 如果 key 存在，就更新访问次数 + 1，更新值
        if (map.containsKey(key)) {
            ListNode listNode = removeListNode(key);

            // 更新 value
            listNode.value = value;
            int frequent = listNode.frequent;
            addListNode2Head(frequent, listNode);
            return;
        }
        // 如果 key 不存在
        // 1、如果满了，先删除访问次数最小的的末尾结点，再删除 map 里对应的 key
```

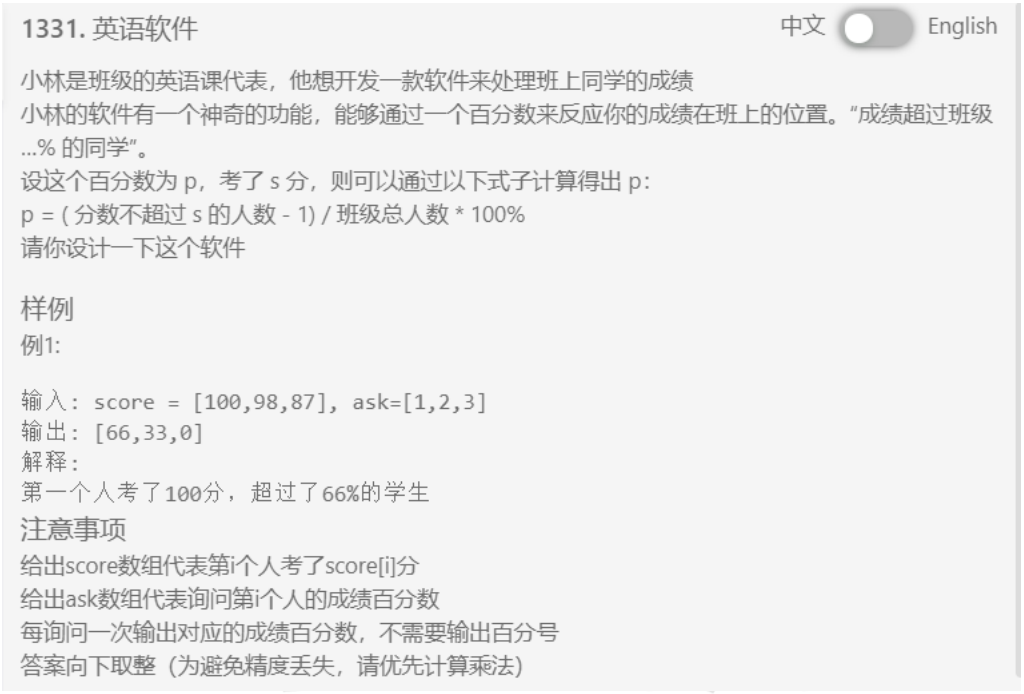
```
        if (map.size() == capacity) {
            // 1、从双链表里删除结点
            DoubleLinkedList doubleLinkedList = frequentMap.get(minFrequent);
            ListNode removeNode = doubleLinkedList.removeTail();

            // 2、删除 map 里对应的 key
            map.remove(removeNode.key);
        }
        // 2、再创建新结点放在访问次数为 1 的双向链表的前面
        ListNode newListNode = new ListNode(key, value);
        addListNode2Head(1, newListNode);
        map.put(key, newListNode);
        // 【注意】因为这个结点是刚刚创建的，最少访问次数一定为 1
        this.minFrequent = 1;
    }
    // 以下部分主要是结点类和双向链表的操作
    //结点类，是双向链表的组成部分
    private class ListNode {
        private int key;
        private int value;
        private int frequent = 1;
        private ListNode pre;
        private ListNode next;
        public ListNode() {
        }
        public ListNode(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }
    //双向链表
    private class DoubleLinkedList {
        private ListNode dummyHead; //虚拟头结点，它无前驱结点
        private ListNode dummyTail; //虚拟尾结点，它无后继结点
        private int count; //当前双向链表的有效结点数
        public DoubleLinkedList() {
            // 虚拟头尾结点赋值多少无所谓
            this.dummyHead = new ListNode(-1, -1);
            this.dummyTail = new ListNode(-2, -2);
            dummyHead.next = dummyTail;
            dummyTail.pre = dummyHead;
            count = 0;
        }
        //把一个结点类添加到双向链表的开头（头部是最新使用数据）
```

```
        public void addNode2Head(ListNode addNode) {
            ListNode oldHead = dummyHead.next;
            // 两侧结点指向它
            dummyHead.next = addNode;
            oldHead.pre = addNode;
            // 它的前驱和后继指向两侧结点
            addNode.pre = dummyHead;
            addNode.next = oldHead;
            count++;
        }
        //把双向链表的末尾结点删除（尾部是最旧的数据，在缓存满的时候淘汰）
        public ListNode removeTail() {
            ListNode oldTail = dummyTail.pre;
            ListNode newTail = oldTail.pre;
            // 两侧结点建立连接
            newTail.next = dummyTail;
            dummyTail.pre = newTail;
            // 它的两个属性切断连接
            oldTail.pre = null;
            oldTail.next = null;
            // 重要：删除一个结点，当前双向链表的结点个数少 1
            count--;
            // 维护
            return oldTail;
        }
    }
    //将原来访问次数的结点，从双向链表里脱离出来
    private ListNode removeListNode(int key) {
        // 获得结点类
        ListNode deleteNode = map.get(key);
        ListNode preNode = deleteNode.pre;
        ListNode nextNode = deleteNode.next;
        // 两侧结点建立连接
        preNode.next = nextNode;
        nextNode.pre = preNode;
        // 删除去原来两侧结点的连接
        deleteNode.pre = null;
        deleteNode.next = null;
        // 维护双链表结点数
        frequentMap.get(deleteNode.frequent).count--;
        // 【注意】维护 minFrequent
        // 如果当前结点正好在最小访问次数的链表上，并且移除以后结点数为 0，最小访问次数
需要加 1
```

```
        if (deleteNode.frequent == minFrequent && frequentMap.get(deleteNode.fr
equent).count == 0) {
            // 这一步需要仔细想一下，经过测试是正确的
            minFrequent++;
        }
        // 访问次数加 1
        deleteNode.frequent++;
        return deleteNode;
    }
    //把结点放在对应访问次数的双向链表的头部
    private void addListNode2Head(int frequent, ListNode addNode) {
        DoubleLinkedList doubleLinkedList;
        // 如果不存在，就初始化
        if (frequentMap.containsKey(frequent)) {
            doubleLinkedList = frequentMap.get(frequent);
        } else {
            doubleLinkedList = new DoubleLinkedList();
        }
        // 添加到 DoubleLinkedList 的表头
        doubleLinkedList.addNode2Head(addNode);
        frequentMap.put(frequent, doubleLinkedList);
    }

    public static void main(String[] args) {
        LFUCache cache = new LFUCache(2);
        cache.put(1, 1);
        cache.put(2, 2);
        System.out.println(cache.map.keySet());
        int res1 = cache.get(1);
        System.out.println(res1);
        cache.put(3, 3);
        System.out.println(cache.map.keySet());
        int res2 = cache.get(2);
        System.out.println(res2);
        int res3 = cache.get(3);
        System.out.println(res3);
        cache.put(4, 4);
        System.out.println(cache.map.keySet());
        int res4 = cache.get(1);
        System.out.println(res4);
        int res5 = cache.get(3);
        System.out.println(res5);
        int res6 = cache.get(4);
        System.out.println(res6);
```

```
    }
}
```

15、英语软件（lintcode1331）



1331. 英语软件

中文 English

小林是班级的英语课代表，他想开发一款软件来处理班上同学的成绩
小林的软件有一个神奇的功能，能够通过一个百分数来反应你的成绩在班上的位置。"成绩超过班级 ...% 的同学"。
设这个百分数为 p，考了 s 分，则可以通过以下式子计算得出 p:
p = (分数不超过 s 的人数 - 1) / 班级总人数 * 100%
请你设计一下这个软件

样例
例1:

```
输入: score = [100,98,87], ask=[1,2,3]
输出: [66,33,0]
解释:
第一个人考了100分，超过了66%的学生
```

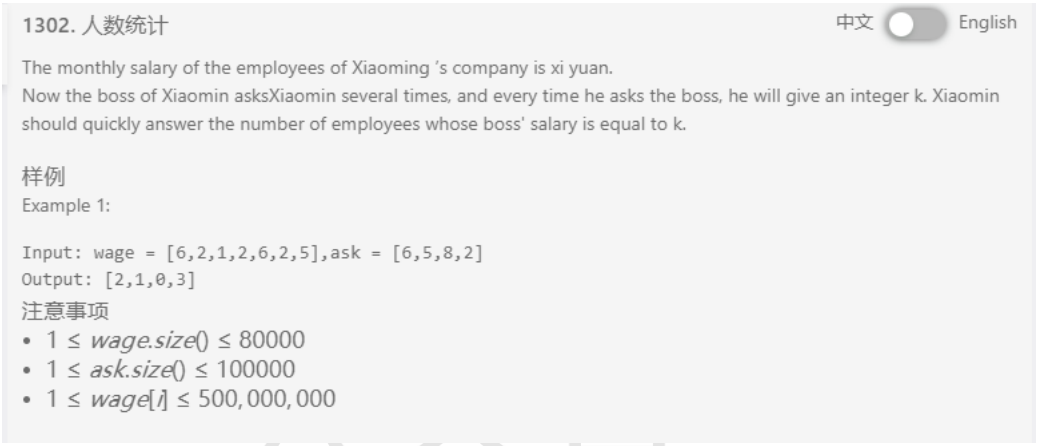
注意事项
给出score数组代表第i个人考了score[i]分
给出ask数组代表询问第i个人的成绩百分数
每询问一次输出对应的成绩百分数，不需要输出百分号
答案向下取整（为避免精度丢失，请优先计算乘法）

```
class stu{
    public int id;
    public double score;
    public stu(int id,double score){
        this.id = id;
        this.score = score;
    }
}
public class Solution {
    /**
     * @param score: Each student's grades
     * @param ask: A series of inquiries
     * @return: Find the percentage of each question asked
     */
    public List<Integer> englishSoftware(List<Integer> score, List<Integer> ask) {
        Map<Integer, Double> vec = new HashMap<Integer, Double>();
        List<stu> sortVec = new ArrayList<stu>();
        for (int i = 0; i < score.size(); i++) {
            vec.put(i + 1, (double)score.get(i));
            sortVec.add(new stu(i, (double)score.get(i)));
        }

        sortVec.sort((o1, o2) -> (int)o1.score - (int)o2.score);
        Map<Double, Double> um = new HashMap<Double,Double>();
```

公众号：yes的练级攻略

```
            for (int i = 0; i < sortVec.size(); i++) {
                um.put(sortVec.get(i).score, (double)i*100 / sortVec.size());
            }
            List<Integer> ans = new ArrayList<Integer>(){};
            for (int k:ask) {
                ans.add((int)Math.floor(um.get(vec.get(k))));
            }
            return ans;
        }
}
```

16、人数统计（lintcode1302）



1302. 人数统计

中文 English

The monthly salary of the employees of Xiaoming 's company is xi yuan.
Now the boss of Xiaomin asksXiaomin several times, and every time he asks the boss, he will give an integer k. Xiaomin should quickly answer the number of employees whose boss' salary is equal to k.

样例
Example 1:

```
Input: wage = [6,2,1,2,6,2,5],ask = [6,5,8,2]
Output: [2,1,0,3]
```

注意事项
- $1 \le wage.size() \le 80000$
- $1 \le ask.size() \le 100000$
- $1 \le wage[i] \le 500,000,000$

```
public class Solution {
    /**
     * @param wage: Salaries of all employees
     * @param ask: Number of inquiries
     * @return: Every time an answer is asked
     */
    public List<Integer> PeopleCounting(List<Integer> wage, List<Integer> ask) {
        HashMap<Integer,Integer> um = new HashMap<>();
        for (int i = 0; i < wage.size(); i++) {
            um.put(wage.get(i),um.getOrDefault(wage.get(i),0) + 1);
        }
        List<Integer> res = new ArrayList<Integer>();
        for (int i = 0; i < ask.size(); i++) {
            if(um.get(ask.get(i)) == null) {
                res.add(0);
            }
            else {
                res.add(um.get(ask.get(i)));
            }
        }
        return res;
    }
```

公众号：yes的练级攻略

}

17、婴儿名字（面试题 17.07）【并查集】

面试题 17.07. 婴儿名字

难度 中等 17 收藏 分享 切换为英文 关注 反馈

每年，政府都会公布一万个最常见的婴儿名字和它们出现的频率，也就是同名婴儿的数量。有些名字有多种拼法，例如，John 和 Jon 本质上是相同的名字，但被当成了两个名字公布出来。给定两个列表，一个是名字及对应的频率，另一个是本质相同的名字对。设计一个算法打印出每个真实名字的实际频率。注意，如果 John 和 Jon 是相同的，并且 Jon 和 Johnny 相同，则 John 与 Johnny 也相同，即它们有传递和对称性。

在结果列表中，选择**字典序最小**的名字作为真实名字。

示例：

```
输入：names = ["John(15)","Jon(12)","Chris(13)","Kris(4)","Christopher(19)"], synonyms = ["(Jon,John)","(John,Johnny)","(Chris,Kris)","(Chris,Christopher)"]
输出：["John(27)","Chris(36)"]
```

提示：

- `names.length <= 100000`

https://leetcode-cn.com/problems/baby-names-lcci/solution/bing-cha-ji-si-lu-yong-hashmapzuo-95ms-by-accounto/

```
class Solution {
    public String[] trulyMostPopular(String[] names, String[] synonyms) {
        Map<String, Integer> map = new HashMap<>();
        Map<String, String> unionMap = new HashMap<>();     //并查集， key(子孙)->value(祖宗)
        for (String name : names) {     //统计频率
            int idx1 = name.indexOf('(');
            int idx2 = name.indexOf(')');
            int frequency = Integer.valueOf(name.substring(idx1 + 1, idx2));
            map.put(name.substring(0, idx1), frequency);
        }
        for (String pair : synonyms) {  //union 同义词
            int idx = pair.indexOf(',');
            String name1 = pair.substring(1, idx);
            String name2 = pair.substring(idx + 1, pair.length() - 1);
            while (unionMap.containsKey(name1)) {   //找 name1 祖宗
                name1 = unionMap.get(name1);
            }
            while (unionMap.containsKey(name2)) {   //找 name2 祖宗
                name2 = unionMap.get(name2);
            }
            if(!name1.equals(name2)){   //祖宗不同，要合并
                int frequency = map.getOrDefault(name1, 0) + map.getOrDefault(name2, 0);    //出现次数是两者之和
                String trulyName = name1.compareTo(name2) < 0 ? name1 : name2;
                String nickName = name1.compareTo(name2) < 0 ? name2 : name1;
```

```
                unionMap.put(nickName, trulyName);      //小名作为大名的分支，即大名是小名的祖宗
                map.remove(nickName);       //更新一下数据
                map.put(trulyName, frequency);
            }
        }
        String[] res = new String[map.size()];
        int index = 0;
        for (String name : map.keySet()) {
            StringBuilder sb = new StringBuilder(name);
            sb.append('(');
            sb.append(map.get(name));
            sb.append(')');
            res[index++] = sb.toString();
        }
        return res;
    }
}
```

18、最短超串（面试题 17.18）

面试题 17.18. 最短超串

难度 中等 10 收藏 分享 切换为英文 关注 反馈

假设你有两个数组，一个长一个短，短的元素均不相同。找到长数组中包含短数组所有的元素的最短子数组，其出现顺序无关紧要。

返回最短子数组的左端点和右端点，如有多个满足条件的子数组，返回左端点最小的一个。若不存在，返回空数组。

示例 1:

```
输入:
big = [7,5,9,0,2,1,3,5,7,9,1,1,5,8,8,9,7]
small = [1,5,9]
输出: [7,10]
```

示例 2:

```
输入:
big = [1,2,3]
small = [4]
输出: []
```

提示:

- `big.length <= 100000`
- `1 <= small.length <= 100000`

https://leetcode-cn.com/problems/shortest-supersequence-lcci/solution/hua-dong-chuang-kou-ha-xi-by-feng-669/

解题思路

维护一个滑动窗口，右窗口滑动计数窗口中含有的small的元素的数量，当达到small的长度时这个区间就是一个解，这时滑动左窗口判断窗口中的数量是否仍能保持为small的数量，若能则继续更新答案，直到窗口中的数量不足以维持住small的数量，此时轮到滑动右窗口以保存元素数量

```
class Solution {
    public int[] shortestSeq(int[] big, int[] small) {
        if(big.length==0 || small.length==0 || small.length>big.length) return
new int[0];
        Set<Integer> original=new HashSet<>();
        Map<Integer,Integer> map=new HashMap<>();
        for(int a:small) original.add(a);
        int[] ans=new int[] {0,Integer.MAX_VALUE};
        int count=0,left=0;
        for(int right=0;right<big.length;++right) {
            if(original.contains(big[right])) {
                if(!map.containsKey(big[right]) || map.get(big[right])<1) ++cou
nt;
                map.put(big[right],map.getOrDefault(big[right],0)+1);
            }
            while(count==small.length) {
                if(original.contains(big[left]) && map.get(big[left])<=1) --cou
nt;
                if(original.contains(big[left])) map.put(big[left],map.get(big[
left])-1);
                if(right-left<ans[1]-ans[0]) {
                    ans[0]=left;
                    ans[1]=right;
                }
                ++left;
            }
        }
        return ans[1]==Integer.MAX_VALUE ? new int[0]:ans;
    }
}
```

19、T9 键盘（面试题 16.20）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.20. T9键盘

难度 中等 13 收藏 分享 切换为英文 关注 反馈

在老式手机上，用户通过数字键盘输入，手机将提供与这些数字相匹配的单词列表。每个数字映射到0至4个字母。给定一个数字序列，实现一个算法来返回匹配单词的列表。你会得到一张含有有效单词的列表。映射如下图所示：

示例 1:

```
输入: num = "8733", words = ["tree", "used"]
输出: ["tree", "used"]
```

示例 2:

```
输入: num = "2", words = ["a", "b", "c", "d"]
输出: ["a", "b", "c"]
```

提示:

- `num.length <= 1000`
- `words.length <= 500`
- `words[i].length == num.length`
- `num` 中不会出现 0, 1 这两个数字

https://leetcode-cn.com/problems/t9-lcci/solution/ha-xi-biao-qiu-jie-bing-tian-jia-yi-ge-zhi-zhen-bi/

```
class Solution {
    public List<String> getValidT9Words(String num, String[] words) {
        List<String> res = new ArrayList<>();
        char[] map = {'2','2','2','3','3','3','4','4','4','5','5','5','6','6','6','7','7','7','7','8','8','8','9','9','9','9'};
        for (String word : words) {
            int index = 0;
            boolean flag = true;
            for (char c : word.toCharArray()) {
                char n = map[c - 'a'];
                if (n != num.charAt(index ++)) {
                    flag = false;
                    break;
                }
            }
            if (flag) res.add(word);
        }
        return res;
    }
```

}

八、HashSet

1、分糖果（575）

575. 分糖果

难度 简单 72 收藏 分享 切换为英文 关注 反馈

给定一个偶数长度的数组，其中不同的数字代表着不同种类的糖果，每一个数字代表一个糖果。你需要把这些糖果**平均**分给一个弟弟和一个妹妹。返回妹妹可以获得的最大糖果的种类数。

示例 1:

输入: candies = [1,1,2,2,3,3]
输出: 3
解析: 一共有三种种类的糖果，每一种都有两个。
最优分配方案：妹妹获得[1,2,3],弟弟也获得[1,2,3]。这样使妹妹获得糖果的种类数最多。

示例 2 :

输入: candies = [1,1,2,3]
输出: 2
解析: 妹妹获得糖果[2,3],弟弟获得糖果[1,1]，妹妹有两种不同的糖果，弟弟只有一种。这样使得妹妹可以获得的糖果种类数最多。

注意:

1. 数组的长度为[2, 10,000]，并且确定为偶数。
2. 数组中数字的大小在范围[-100,000, 100,000]内。

https://leetcode-cn.com/problems/distribute-candies/solution/fen-tang-guo-by-leetcode/

方法四：集合set

算法：

找到唯一元素数量的另一种方法是遍历给定 $candies$ 数组的所有元素，并继续将元素放入集合中。通过集合的属性，它将只包含唯一的元素。最后，我们可以计算集合中元素的数量，例如 $count$。要返回的值将再次由 $\min(count, n/2)$ 给出，如前面的方法所述。其中 n 表示 $candies$ 数组的大小。

- 时间复杂度：$O(n)$。整个 $candies$ 数组只遍历一次。这里，n 表示 $candies$ 数组的大小。
- 空间复杂度：$O(n)$,在最坏的情况下，set 的大小为 n。

```
class Solution {
    public int distributeCandies(int[] candies) {
        HashSet <Integer> set = new HashSet<>();
        for (int i: candies) {
            set.add(i);
        }
        return Math.min(set.size(), candies.length/2);
    }
}
```

2、最长连续序列（128）

128. 最长连续序列

难度 困难 445 收藏 分享 切换为英文 关注

给定一个未排序的整数数组，找出最长连续序列的长度。

要求算法的时间复杂度为 *O(n)*。

示例:

```
输入: [100, 4, 200, 1, 3, 2]
输出: 4
解释: 最长连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

https://leetcode-cn.com/problems/longest-consecutive-sequence/solution/zui-chang-lian-xu-xu-lie-by-leetcode-solution/

方法一：哈希表

思路和算法

我们考虑枚举数组中的每个数 x，考虑以其为起点，不断尝试匹配 $x+1, x+2, \cdots$ 是否存在，假设最长匹配到了 $x+y$，那么以 x 为起点的最长连续序列即为 $x, x+1, x+2, \cdots, x+y$，其长度为 $y+1$，我们不断枚举并更新答案即可。

对于匹配的过程，暴力的方法是 $O(n)$ 遍历数组去看是否存在这个数，但其实更高效的方法是用一个哈希表存储数组中的数，这样查看一个数是否存在即能优化至 $O(1)$ 的时间复杂度。

仅仅是这样我们的算法时间复杂度最坏情况下还是会达到 $O(n^2)$（即外层需要枚举 $O(n)$ 个数，内层需要暴力匹配 $O(n)$ 次），无法满足题目的要求。但仔细分析这个过程，我们会发现其中执行了很多不必要的枚举，如果已知有一个 $x, x+1, x+2, \cdots, x+y$ 的连续序列，而我们却重新从 $x+1$，$x+2$ 或者是 $x+y$ 处开始尝试匹配，那么得到的结果肯定不会优于枚举 x 为起点的答案，因此我们在外层循环的时候碰到这种情况跳过即可。

那么怎么判断是否跳过呢？由于我们要枚举的数 x 一定是在数组中不存在前驱数 $x-1$ 的，不然按照上面的分析我们会从 $x-1$ 开始尝试匹配，因此我们每次在哈希表中检查是否存在 $x-1$ 即能判断是否需要跳过了。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为数组的长度。具体分析已在上面正文中给出。
- 空间复杂度：$O(n)$。哈希表存储数组中所有的数需要 $O(n)$ 的空间。

```java
class Solution {
    public int longestConsecutive(int[] nums) {
        Set<Integer> num_set = new HashSet<Integer>();
        for (int num : nums) {
            num_set.add(num);
        }

        int max = 0;

        for (int num : num_set) {
            if (!num_set.contains(num - 1)) { //判断num_set不包含当前元素-1的值，跳过已经计算的最长递增序列
                int currentNum = num;
                int currentStreak = 1;

                while (num_set.contains(currentNum + 1)) {
```

```
                    currentNum += 1;
                    currentStreak += 1;
                }

                max = Math.max(max, currentStreak);
            }
        }
        return max;
    }
}
```

3、判断数独是否合法（lintcode389）

389. 判断数独是否合法

中文 English

请判定一个数独是否有效。

该数独可能只填充了部分数字，其中缺少的数字用 . 表示。

样例

样例1:

输入:
`["53..7....","6..195...",".98....6.","8...6...3","4..8.3..1","7...2...6",".6....28.","...419..5","....8..79"]`
输出: `true`
样例说明:
这个数独如下图所示，他是合法的。

样例 2:

输入:
`["53..75...","6..195...",".98....6.","8...6...3","4..8.3..1","7...2...6",".6....28.","...419..5","....8..79"]`
输出: `false`
样例说明:
这个数独如下图所示。他是不合法的因为他的第一行和第六列有两个5。

说明

What is `Sudoku`?

- http://sudoku.com.au/TheRules.aspx
- https://zh.wikipedia.org/wiki/數獨
- https://en.wikipedia.org/wiki/Sudoku
- http://baike.baidu.com/subview/961/10842669.htm

注意事项

一个合法的数独（仅部分填充）并不一定是可解的。我们仅需使填充的空格有效即可。

方法 1：hashset

更多硬核内容就在公众号「yes 的练级攻略」

解题思路

这题考查的是二维数组的遍历顺序。

判断每一行是否合法，外层循环枚举行，内层循环枚举列。

判断每一列是否合法，外层循环枚举列，内层循环枚举行。

判断每一块是否合法，每一块左上角的坐标都是`(0, 0), (3, 0), (6, 0), (3, 0), (3, 3) ...`，可以发现都是`3`的倍数，可以总结出规律，先枚举`i = [0, 1, 2]`，再枚举`j = [0, 1, 2]`，左上角坐标就是`(i**3, j****3)`。

代码思路

利用一个`set`来记录已经被用过的数，每次遍历一行，一列，一块，将访问过的元素加入`set`，冲突的话，返回`false`退出。最后返回`true`。

复杂度分析

假设这是一个`N * N`的数独，不止是`9 * 9`。

时间复杂度

- 要遍历`3`次这个数独，时间复杂度为`O(N^2)`。

空间复杂度

- 需要`O(N)`的空间，记录以用过的数。

源代码

```
public class Solution {
    /**
     * @param board: the board
     * @return: whether the Sudoku is valid
     */
    public boolean isValidSudoku(char[][] board) {
        // write your code here
        Set<Character> used = new HashSet();
        //枚举行
        for (int row = 0; row < 9; row++) {
            used.clear();
            for (int col = 0; col < 9; col++) {
                if (! checkValid(board[row][col], used)){
                    return false;
                }
            }
        }
        //枚举列
        for (int col = 0; col < 9; col++) {
            used.clear();
            for (int row = 0; row < 9; row++) {
                if (! checkValid(board[row][col], used)){
                    return false;
                }
            }
        }
        //每个分块的左上角的坐标为（i*3, j*3）
        for (int i = 0; i < 3; i++){
```

禁止商用
公众号：
yes的练级攻略

```
                for (int j = 0; j < 3; j++){
                    used.clear();
                    for (int row = i * 3; row < i * 3 + 3; row++) {
                        for (int col = j * 3; col < j * 3 + 3; col++){
                            if (! checkValid(board[row][col], used)){
                                return false;
                            }
                        }
                    }
                }
            }
            return true;
        }

        boolean checkValid(char c, Set<Character> used) {
            if (c == '.') {
                return true;
            }
            if (used.contains(c)) {
                return false;
            }
            used.add(c);
            return true;
        }
}
```

方法 2：直接用数组

```
public class Solution {
    /**
     * @param board: the board
     * @return: whether the Sudoku is valid
     */
    public boolean isValidSudoku(char[][] board) {
        // write your code here
        boolean[] visited = new boolean[9];
        //枚举行
        for (int row = 0; row < 9; row++) {
            Arrays.fill(visited, false);
            for (int col = 0; col < 9; col++) {
                if(!process(visited, board[row][col])){
                    return false;
                }
            }
        }
        //枚举列
```

禁止商用
公众号：
yes的练级攻略

```
        for (int col = 0; col < 9; col++) {
            Arrays.fill(visited, false);
            for (int row = 0; row < 9; row++) {
                if(!process(visited, board[row][col])){
                    return false;
                }
            }
        }
        //每个分块的左上角的坐标为（i*3, j*3）
        for (int i = 0; i < 3; i++){
            for (int j = 0; j < 3; j++){
                Arrays.fill(visited, false);
                for (int row = i * 3; row < i * 3 + 3; row++) {
                    for (int col = j * 3; col < j * 3 + 3; col++){
                        if (!process(visited, board[row][col])){
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }

    private boolean process(boolean[] visited, char digit){
        if(digit == '.'){
            return true;
        }
        int num = digit - '0';
        if ( num < 1 || num > 9 || visited[num-1]){
            return false;
        }
        visited[num-1] = true;
        return true;
    }
}
```

4、存在重复元素 2（219）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

219. 存在重复元素 II

难度 简单 189 收藏 分享 切换为英文 关注 反馈

给定一个整数数组和一个整数 k，判断数组中是否存在两个不同的索引 i 和 j，使得 **nums [i] = nums [j]**，并且 i 和 j 的差的 **绝对值** 至多为 k。

示例 1:

```
输入: nums = [1,2,3,1], k = 3
输出: true
```

示例 2:

```
输入: nums = [1,0,1,1], k = 1
输出: true
```

示例 3:

```
输入: nums = [1,2,3,1,2,3], k = 2
输出: false
```

https://leetcode-cn.com/problems/contains-duplicate-ii/solution/hua-jie-suan-fa-219-cun-zai-zhong-fu-yuan-su-ii-by/

解题方案

思路

- 标签：哈希
- 维护一个哈希表，里面始终最多包含 `k` 个元素，当出现重复值时则说明在 `k` 距离内存在重复元素
- 每次遍历一个元素则将其加入哈希表中，如果哈希表的大小大于 `k` ，则移除最前面的数字
- 时间复杂度：$O(n)$，n 为数组长度

复杂度分析

- 时间复杂度：$O(n)$
  我们会做 n 次 `搜索` ， `删除` ， `插入` 操作，每次操作都耗费常数时间。
- 空间复杂度：$O(\min(n,k))$
  开辟的额外空间取决于散列表中存储的元素的个数，也就是滑动窗口的大小 $O(\min(n,k))$。

```
class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        HashSet<Integer> set = new HashSet<>();
        for (int i = 0; i < nums.length; i++){
            if (set.contains(nums[i])) return true;
            set.add(nums[i]);
            if (set.size() > k) set.remove(nums[i-k]);
        }
        return false;
    }
}
```

5、存在重复元素 3（220）

更多硬核内容就在公众号「yes 的练级攻略」

220. 存在重复元素 III

难度 中等 205 收藏 分享 切换为英文 关注 反馈

在整数数组 `nums` 中，是否存在两个下标 *i* 和 *j*，使得 **nums [i]** 和 **nums [j]** 的差的绝对值小于等于 *t*，且满足 *i* 和 *j* 的差的绝对值也小于等于 *k*。

如果存在则返回 `true`，不存在返回 `false`。

示例 1:

```
输入: nums = [1,2,3,1], k = 3, t = 0
输出: true
```

示例 2:

```
输入: nums = [1,0,1,1], k = 1, t = 2
输出: true
```

示例 3:

```
输入: nums = [1,5,9,1,5,9], k = 2, t = 3
输出: false
```

https://leetcode-cn.com/problems/contains-duplicate-iii/solution/hua-dong-chuang-kou-er-fen-sou-suo-shu-zhao-shang-/

方法二：滑动窗口（以空间换时间）

题目意思翻译一下：在数组 `nums[i]` 中，在任意有效区间 `[i, i + k]` 里是否存在两个数的绝对值小于等于 `t`，即

$$|nums[i] - nums[j]| <= t$$

等价于 $nums[i] - nums[j] <= t$ 并且 $nums[i] - nums[j] >= -t$，

整理得：$nums[i] <= t + nums[j]$ 并且 $nums[i] >= nums[j] - t$。

于是算法可以在遍历的时候，把遍历到的数存到 `BST` 中，边遍历边查找。

根据刚才得到式子 $nums[i] >= nums[j] - t$，假设当前遍历到的数是 $nums[j]$，$nums[i]$ 就是 $nums[j] - t$ 的天花板数（`ceiling`），这个数同时还要小于等于 $t + nums[j]$。根据这一点写滑动窗口的代码。

- `ceiling(key)` 函数：返回大于等于 `key` 的最小元素，如果不存在，返回空。下面的是这个函数的文档（通过 Intellij IDEA 查看）。

复杂度分析：

- 时间复杂度：$O(N \log K)$，遍历数组使用 $O(N)$，在遍历的同时向二叉搜索树中插入元素和移除元素的时间复杂度是 $O(\log K)$。
- 空间复杂度：$O(1)$。

```
import java.util.TreeSet;
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int
 t) {
        // 滑动窗口结合查找表，此时滑动窗口即为查找表本身（控制查找表的大小即可控制窗口大小）
        TreeSet<Long> set = new TreeSet<>();
        for (int i = 0; i < nums.length; i++) {
            // 边添加边查找
```

```
            // 查找表中是否有大于等于 nums[i] - t 且小于等于 nums[i] + t 的
值
            Long ceiling = set.ceiling((long) nums[i] - (long) t);
            if (ceiling != null && ceiling <= ((long) nums[i] + (long)
t)) {
                return true;
            }
            // 添加后，控制查找表（窗口）大小，移除窗口最左边元素
            set.add((long) nums[i]);
            if (set.size() == k + 1) {
                set.remove((long) nums[i - k]);
            }
        }
        return false;
    }
}
```

6、扑克牌中的顺子（剑指 61）

剑指 Offer 61. 扑克牌中的顺子

难度 简单 55 收藏 分享 切换为英文 关注 反馈

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为 0 ，可以看成任意数字。A 不能视为 14。

示例 1:

```
输入: [1,2,3,4,5]
输出: True
```

示例 2:

```
输入: [0,0,1,2,5]
输出: True
```

限制:

数组长度为 5

数组的数取值为 [0, 13] .

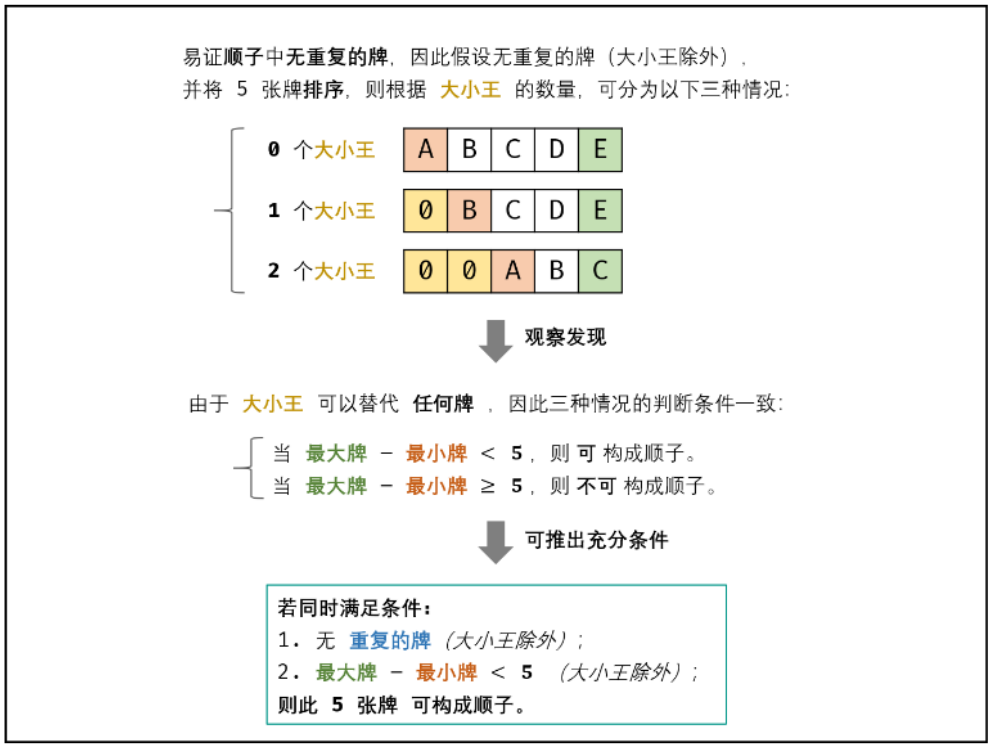
https://leetcode-cn.com/problems/bu-ke-pai-zhong-de-shun-zi-lcof/solution/mian-shi-ti-61-bu-ke-pai-zhong-de-shun-zi-ji-he-se/

根据题意，此 5 张牌是顺子的 **充分条件** 如下：

1. 除大小王外，所有牌 **无重复** ；
2. 设此 5 张牌中最大的牌为 max ，最小的牌为 min （大小王除外），则需满足：

$$max - min < 5$$

因而，可将问题转化为：此 5 张牌是否满足以上两个条件？



方法一： 集合 Set + 遍历

- 遍历五张牌，遇到大小王（即 0 ）直接跳过。
- **判别重复：** 利用 Set 实现遍历判重， Set 的查找方法的时间复杂度为 $O(1)$ ；
- **获取最大 / 最小的牌：** 借助辅助变量 ma 和 mi ，遍历统计即可。

复杂度分析：

- **时间复杂度** $O(N) = O(5) = O(1)$ ： 其中 N 为 $nums$ 长度，本题中 $N = 5$ ；遍历数组使用 $O(N)$ 时间。
- **空间复杂度** $O(N) = O(5) = O(1)$ ： 用于判重的辅助 Set 使用 $O(N)$ 额外空间。

方法 1：hashset

```
class Solution {
    public boolean isStraight(int[] nums) {
        Set<Integer> a = new HashSet<>();
        int max = 0, min = 14;
        for (int num : nums){
            if (num == 0) continue;
            max = Math.max(max, num);
            min = Math.min(min, num);
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            if (a.contains(num)) return false;
            a.add(num);
        }
        return max - min < 5;
    }
}
```

方法二：排序 + 遍历

- 先对数组执行排序。
- **判别重复：** 排序数组中的相同元素位置相邻，因此可通过遍历数组，判断 $nums[i] = nums[i+1]$ 是否成立来判重。
- **获取最大 / 最小的牌：** 排序后，数组末位元素 $nums[4]$ 为最大牌；元素 $nums[joker]$ 为最小牌，其中 $joker$ 为大小王的数量。

复杂度分析：

- **时间复杂度** $O(N\log N) = O(5\log 5) = O(1)$ ： 其中 N 为 $nums$ 长度，本题中 $N = 5$ ；数组排序使用 $O(N\log N)$ 时间。
- **空间复杂度** $O(1)$ ： 变量 $joker$ 使用 $O(1)$ 大小的额外空间。

方法 2：排序+遍历

```
class Solution {
    public boolean isStraight(int[] nums) {
        int joker = 0;
        Arrays.sort(nums); // 数组排序
        for(int i = 0; i < 4; i++) {
            if(nums[i] == 0) joker++; // 统计大小王数量
            else if(nums[i] == nums[i + 1]) return false; // 若有重复，提前返回 false
        }
        return nums[4] - nums[joker] < 5;
    }
}
```

九、各种排序算法

- 各种排序

排序方式	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
希尔排序	$O(n^{1.3})$			O(1)	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	简单
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定	较复杂
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定	较复杂
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(r)	稳定	较复杂

更多硬核内容就在公众号「yes 的练级攻略」

算　法	最坏情况运行时间	平均情况/期望运行时间
插入排序	$\Theta(n^2)$	$\Theta(n^2)$
归并排序	$\Theta(n\lg n)$	$\Theta(n\lg n)$
堆排序	$O(n\lg n)$	—
快速排序	$\Theta(n^2)$	$\Theta(n\lg n)$(期望)
计数排序	$\Theta(k+n)$	$\Theta(k+n)$
基数排序	$\Theta(d(n+k))$	$\Theta(d(n+k))$
桶排序	$\Theta(n^2)$	$\Theta(n)$(平均情况)

(1)　插入排序

平均和最坏的时间复杂度 o(n^2)，最好的时间复杂度 o(n)【有序的情况下】，空间复杂度 o(1)，稳定。

思路：i 从 1 开始，如果下标为 i 的值小于前一个数字，把下标为 i 的值赋值给 tmp，设置 j 为 i 前一个下标，如果 tmp 还是小于下标为 j 的值，则将下标 j 的值赋值给 j 后面（右）一个下标并将 j 指向前（左）一个下标。最后把 tmp 赋值给 j 后面的一个下标。

```
public class InsertSort {
  public static void insertSort (int[] arr) {
      if(arr == null || arr.length == 1) {
           return;
      }
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int tmp = arr[i];
              int j;
              for (j = i-1; j >= 0 && tmp < arr[j]; j--) {
                  //把数据向后移，把索引向前移
                  arr[j+1]=arr[j];
              }
              arr[j+1]=tmp;
          }
      }
  }
}
```

对链表进行插入排序（147）

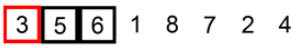
禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

147. 对链表进行插入排序

难度 中等 180 收藏 分享 切换为英文 关注 反馈

对链表进行插入排序。



插入排序的动画演示如上。从第一个元素开始，该链表可以被认为已经部分排序（用黑色表示）。
每次迭代时，从输入数据中移除一个元素（用红色表示），并原地将其插入到已排好序的链表中。

插入排序算法：

1. 插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。
2. 每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。
3. 重复直到所有输入数据插入完为止。

示例 1：

```
输入：4->2->1->3
输出：1->2->3->4
```

示例 2：

```
输入：-1->5->3->4->0
输出：-1->0->3->4->5
```

https://leetcode-cn.com/problems/insertion-sort-list/solution/java-jian-ji-dai-ma-shi-xian-by-serenade0816/

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode insertionSortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode dummy = new ListNode(0);
        ListNode pre = head;
        ListNode cur = head.next;
        dummy.next = head;
        while (cur != null) {
            if (pre.val <= cur.val) {// 本来就有序
                pre = cur;
                cur = cur.next;
            } else {
                ListNode p = dummy;
```

```
                // 找到一个位置使得 p < cur < p.next
                while (p.next != cur && p.next.val < cur.val) {
                    p = p.next;
                }
                // 将 cur 插入到 p 和 p.next 之间
                // 因为 cur 被插到前面，pre 的指针需要跳过 cur
                pre.next = cur.next;
                cur.next = p.next;
                p.next = cur;
                // 完成插入后，cur 回到 pre 后面
                cur = pre.next;
            }
        }
        return dummy.next;
    }
}
```

(2) 希尔排序

是插入排序的一种，又称为缩小增量排序。时间复杂度 o($n^{1.3}$)，空间复杂度 o(1)，不稳定。

思路：

设置一个增量 s,记录每次 arr 的长度除以 2 的值

设置一个 i 等于这个 s，i 从这个数组的一半开始每次循环加一

设置一个 j 等于 i - s，每次减 s 个增量，减到小于 0 为止

在这里进行下标 i 和下标 j 的值的对比：当增量序列为 dlta[k]=2t-k+1-1（0≤k≤t≤|log2(n+1)|）时，可以获得不错的效率。

```
public class ShellSort {
  public static void shellSort(int[] arr) {
      if(arr == null || arr.length == 1) {
          return;
      }
      for (int s = arr.length / 2; s > 0; s /=2) {
          for (int i = s; i < arr.length; i++) {
              int tmp = arr[i];
              int j = i - s;
              while (j>=0 && tmp < arr[j]) {
                  arr[j+s] = arr[j];
                  j -= s;
              }
              arr[j+s] = tmp;
          }
      }
  }
}
```

(3) 冒泡排序

平均和最坏的时间复杂度 o(n^2)，最好的时间复杂度 o(n)【有序的情况下】，空间复杂度 o(1)，

禁止商用 公众号：yes的练级攻略

稳定。

思路：依次比较相邻的两个数，将比较小的数放在前面，比较大的数放在后面。

```
public class BubbleSort {
//冒泡排序，从小到大
  public static void bubbleSort(int [] arr) {
      if(arr == null || arr.length == 1) {
          return;
       }
      for (int i = 0; i < arr.length-1; i++) {
        //第 i 趟比较
          for (int j = 0; j < arr.length - 1 - i; j++) {
            //开始进行比较，如果 arr[j]比 arr[j+1]的值大，那就交换位置
              if (arr[j] > arr[j+1]) {
                  int tmp = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = tmp;
              }
          }
      }
  }
  //逆冒泡排序，从大到小
  public static void oppositeBubbleSort(int[] arr) {
      if(arr == null || arr.length == 1) {
          return;
      }
      for (int i = 0; i < arr.length-1; i++) {
        //第 i 趟比较
          for (int j = 0; j < arr.length - 1 - i; j++) {
            //开始进行比较，如果 arr[j]比 arr[j+1]的值小，那就交换位置
              if (arr[j] < arr[j+1]) {
                  int tmp = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = tmp;
              }
          }
      }
  }
}
```

(4) 快速排序

平均和最好时间复杂度 o(nlogn)，最坏的时间复杂度 $o(n^2)$，空间复杂度 o(logn)，不稳定。

禁止商用 公众号：ye的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

对于包含 n 个数的输入数组来说，快速排序是一种最坏情况时间复杂度为 $\Theta(n^2)$ 的排序算法。虽然最坏情况时间复杂度很差，但是快速排序通常是实际排序应用中最好的选择，因为它的平均性能非常好：它的期望时间复杂度是 $\Theta(n\lg n)$，而且 $\Theta(n\lg n)$ 中隐含的常数因子非常小。另外，它还能够进行原址排序(见 2.1 节)，甚至在虚存环境中也能很好地工作。

7.1 节将描述快速排序算法及它的一个重要的划分子程序。因为快速排序的运行情况比较复杂，在 7.2 节中，我们先对其性能进行一个直观的讨论，在本章的最后会给出一个准确的分析。在 7.3 节中，我们会介绍一个基于随机抽样的快速排序算法。这一算法的期望时间复杂度较好，而且没有什么特殊的输入会导致最坏情况的发生。7.4 节对这一随机算法的分析表明，其最坏情况时间复杂度是 $\Theta(n^2)$；在元素互异的情况下，期望时间复杂度 $O(n\lg n)$。

7.2 快速排序的性能

快速排序的运行时间依赖于划分是否平衡，而平衡与否又依赖于用于划分的元素。如果划分是平衡的，那么快速排序算法性能与归并排序一样。如果划分是不平衡的，那么快速排序的性能就接近于插入排序了。在本节中，我们将给出划分为平衡或不平衡时快速排序性能的非形式化的分析。

最坏情况划分

当划分产生的两个子问题分别包含了 $n-1$ 个元素和 0 个元素时，快速排序的最坏情况发生了(证明见 7.4.1 节)。不妨假设算法的每一次递归调用中都出现了这种不平衡划分。划分操作的时间复杂度是 $\Theta(n)$。由于对一个大小为 0 的数组进行递归调用会直接返回，因此 $T(0)=\Theta(1)$，

于是算法运行时间的递归式可以表示为：

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

从直观上来看，每一层递归的代价可以被累加起来，从而得到一个算术级数(公式(A.2))，其结果为 $\Theta(n^2)$。实际上，利用代入法可以直接得到递归式 $T(n)=T(n-1)+\Theta(n)$ 的解为 $T(n)=\Theta(n^2)$(见练习 7.2-1)。

因此，如果在算法的每一层递归上，划分都是最大程度不平衡的，那么算法的时间复杂度就是 $\Theta(n^2)$。也就是说，在最坏情况下，快速排序算法的运行时间并不比插入排序更好。此外，当输入数组已经完全有序时，快速排序的时间复杂度仍然为 $\Theta(n^2)$。而在同样情况下，插入排序的时间复杂度为 $O(n)$。

最好情况划分

在可能的最平衡的划分中，PARTITION 得到的两个子问题的规模都不大于 $n/2$。这是因为其中一个子问题的规模为 $\lfloor n/2 \rfloor$，而另一个子问题的规模为 $\lceil n/2 \rceil - 1$。在这种情况下，快速排序的性能非常好。此时，算法运行时间的递归式为：

$$T(n) = 2T(n/2) + \Theta(n)$$

在上式中，我们忽略了一些余项以及减 1 操作的影响。根据主定理(定理 4.1)的情况 2，上述递归式的解为 $T(n)=\Theta(n\lg n)$。通过在每一层递归中都平衡划分子数组，我们得到了一个渐近时间上更快的算法。

更多硬核内容就在公众号「yes 的练级攻略」

平衡的划分

快速排序的平均运行时间更接近于其最好情况，而非最坏情况。详细的分析可以参看本书 7.4 节。理解这一点的关键就是理解划分的平衡性是如何反映到描述运行时间的递归式上的。

例如，假设划分算法总是产生 9：1 的划分，乍一看，这种划分是很不平衡的。此时，我们得到的快速排序时间复杂度的递归式为：

$$T(n) = T(9n/10) + T(n/10) + cn$$

这里，我们显式地写出了 $\Theta(n)$项中所隐含的常数 c。图 7-4 显示了这一递归调用所对应的递归树。注意，树中每一层的代价都是 cn，直到在深度 $\log_{10} n = \Theta(\lg n)$处达到递归的边界条件时为

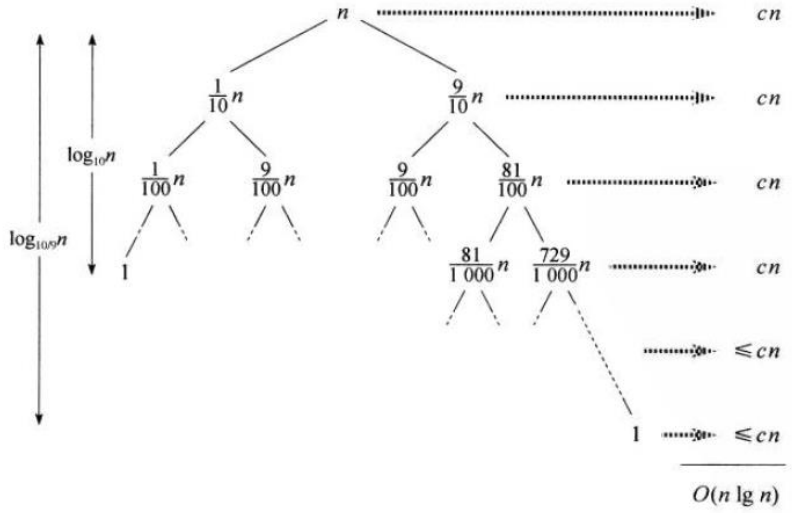


图 7-4 QUICKSORT 的一棵递归树，其中 PARTITION 总是产生 9：1 的划分。该树的时间复杂度为 $O(n \lg n)$。每个结点的值表示子问题的规模，每一层的代价显示在最右边。每一层的代价包含了 $\Theta(n)$项中隐含的常数 c

止，之后每层代价至多为 cn。递归在深度为 $\log_{10/9} n = \Theta(\lg n)$处终止。因此，快速排序的总代价为 $O(n \lg n)$。因此，即使在递归的每一层上都是 9：1 的划分，直观上看起来非常不平衡，但快速排序的运行时间是 $O(n \lg n)$，与恰好在中间划分的渐近运行时间是一样的。实际上，即使是 99：1的划分，其时间复杂度仍然是 $O(n \lg n)$。事实上，任何一种常数比例的划分都会产生深度为 $\Theta(\lg n)$的递归树，其中每一层的时间代价都是 $O(n)$。因此，只要划分是常数比例的，算法的运行时间总是 $O(n \lg n)$。

① 需要形参有开头 start 和结尾 end

② 如果 start < end 就递归自己

③ 把下标为 0（也可以是其他下标）的值设为标准值，把开头和结束分别设置为低位（low）和高位（high）。

④ 当 low<high 时循环执行

（low<high 时)高位的值大于标准值，把高位减一位，直到小于时把当前高位的值赋值给低位

（low<high 时)低位的值小于标准值，把低位减一位，直到大于时把当前低位的值赋值给高位

⑤ 执行完后此时的高位和低位相等，把标准值赋值给低位 or 高位的值，（low<high 时)保证高位一定等于低位出循环。

⑥ 此时的数组分为（low 右边）大于标准值和（low 左边）小于标准值，然后分两边递归调用自己。

⑦ 直到 start >= end（low >= high）不会递归下去

```
public class QuickSort {
```

```
    public static void quickSort(int[] arr, int start, int end) {
        if (start < end) {
            int low = start;
            int high = end;
            int standard = arr[low];
            while (low < high) {
                // 当队尾的元素大于等于基准数据时,向前挪动 high 指针
                while (low < high && standard < arr[high]) {
                    high--;
                }
                if (low < high) {
                    arr[low++] = arr[high];
                }
                while (low < high && standard > arr[low]) {
                    low++;
                }
                if (low < high) {
                    arr[high--] = arr[low];
                }
            }
            arr[low] = standard;
            quickSort (arr, start, low-1);
            quickSort (arr, low +1, end);
        }
    }
}
```

- 字符串排序（lintcode1153）【由一些逗号隔开的字符串，返回字典序排序】

1153. 字符串排序

给定一些由 , 隔开的字符串，请将他们按字典序排列。

样例
样例 1

输入: "bb,aa,lintcode,c"
输出: "aa,bb,c,lintcode"
说明: 在字典序中，"aa" < "bb" < "c" < "lintcode"
挑战
你可以自己写一个排序函数吗?

注意事项
- 字符串仅包含小写字母。
- 字符串数量≤ 1 000且总长度≤ 10^5

```
public class Solution {
    /**
     * @param s: string
     * @return: sort string in lexicographical order
     */
    public String sorting(String s) {
```

禁止商用 公众号：yes的练级攻略

```
        // write your code here
        String[] arr = s.split(",");
        qsort(arr,0, arr.length-1);
        String ans = "";
        for (int i = 0; i < arr.length; i++) {
            if (i > 0) {
                ans += ",";
            }
            ans += arr[i];
        }
        return ans;
    }
    public void qsort(String[] arr, int start, int end) {
        if (start < end) {
            int low = start;
            int high = end;
            String standard = arr[low];
            while (low < high) {
                while (low < high && arr[high].compareTo(standard) > 0) {
                    high--;
                }
                if (low < high) {
                    arr[low++] = arr[high];
                }
                while (low < high && arr[low].compareTo(standard) < 0) {
                    low++;
                }
                if (low < high) {
                    arr[high--] = arr[low];
                }
            }
            arr[low] = standard;
            qsort(arr, start, low-1);
            qsort(arr, low + 1, end);
        }
    }
}
```

- 把数组排成最小的数（剑指 45）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 45. 把数组排成最小的数

难度 中等 78

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

示例 1:

```
输入：[10,2]
输出："102"
```

示例 2:

```
输入：[3,30,34,5,9]
输出："3033459"
```

提示:

- `0 < nums.length <= 100`

说明:

- 输出结果可能非常大，所以你需要返回一个字符串而不是整数
- 拼接起来的数字可能会有前导 0，最后结果不需要去掉前导 0

https://leetcode-cn.com/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/solution/mian-shi-ti-45-ba-shu-zu-pai-cheng-zui-xiao-de-s-4/

算法流程:

1. **初始化：** 字符串列表 $strs$，保存各数字的字符串格式；
2. **列表排序：** 应用以上 “排序判断规则”，对 $strs$ 执行排序；
3. **返回值：** 拼接 $strs$ 中的所有字符串，并返回。

复杂度分析:

- **时间复杂度** $O(N \log N)$： N 为最终返回值的字符数量（$strs$ 列表的长度 $\leq N$）；使用快排或内置函数的平均时间复杂度为 $O(N \log N)$，最差为 $O(N^2)$。
- **空间复杂度** $O(N)$： 字符串列表 $strs$ 占用线性大小的额外空间。

方法 1：使用内置函数

```
class Solution {
    public String minNumber(int[] nums) {
        if (nums.length == 0) return "";
        int len = nums.length;
        String[] str = new String[len];
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < len; i++){
            str[i] = String.valueOf(nums[i]);
        }
        //第一种
        Arrays.sort(str, (o1, o2) -> (o1 + o2).compareTo(o2 + o1));
        //第二种
        // Arrays.sort(str, new Comparator<String>(){
```

```
        //     @Override
        //     public int compare(String s1, String s2){
        //         String c1 = s1 + s2;
        //         String c2 = s2 + s1;
        //         return c1.compareTo(c2);
        //     }
        // });
        for (String s : str){
            sb.append(s);
        }
        return sb.toString();
    }
}
```

方法 2：快排

```
class Solution {
    public String minNumber(int[] nums) {
        if (nums.length == 0) return "";
        int len = nums.length;
        String[] str = new String[len];
        for (int i = 0; i < len; i++){
            str[i] = String.valueOf(nums[i]);
        }
        fastSort(str, 0, str.length - 1);
        StringBuffer sb = new StringBuffer();
        for (String s : str){
            sb.append(s);
        }
        return sb.toString();
    }
    public void fastSort(String[] str, int start, int end){
        if (start >= end) return;
        int i = start, j = end;
        String tmp = str[i];
        while (i < j) {
            while (i < j && (str[j] + str[start]).compareTo(str[start]
+ str[j]) >= 0) j--;
            while (i < j && (str[i] + str[start]).compareTo(str[start]
+ str[i]) <= 0) i++;
            tmp = str[i];
            str[i] = str[j];
            str[j] = tmp;
        }
        str[i] = str[start];
        str[start] = tmp;
```

```
            fastSort(str, start, i - 1);
            fastSort(str, i + 1, end);
        }
    }
```

- 最小 k 个数（面试题 17.14）【剑指 40 题】（包含前 k 小和第 k 小）

面试题 17.14. 最小K个数

难度 中等 18 收藏 分享 切换为英文 关注 反馈

设计一个算法，找出数组中最小的k个数。以任意顺序返回这k个数均可。

示例：

```
输入： arr = [1,3,5,7,2,4,6,8], k = 4
输出： [1,2,3,4]
```

提示：

- 0 <= len(arr) <= 100000
- 0 <= k <= min(100000, len(arr))

通过次数 9,221 | 提交次数 16,545

剑指 Offer 40. 最小的k个数

难度 简单 110 收藏 分享 切换为英文 关注 反馈

输入整数数组 arr ，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

```
输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]
```

示例 2：

```
输入：arr = [0,1,2,1], k = 1
输出：[0]
```

限制：

- 0 <= k <= arr.length <= 10000
- 0 <= arr[i] <= 10000

https://zhuanlan.zhihu.com/p/114699207

TopK问题，不管是**求前K大/前K小/第K大/第K小**等，都有4种不错的方法喔：

1. O(N)：用快排变形最最最高效解决TopK问题

2. O(NlogK)：大根堆（前K小）/小根堆（前K大）

3. O(NlogK)：二叉搜索树

4. O(N): 对于数据范围有限的情况例如本题，可以直接计数排序O(N)高效解决～

下面针对本题，求数组中的前K小的数，用上面的4种方法分别实现一遍。

1. 用快排变形最最最高效解决TopK问题 O(N)

注意找前K大/前K小/第K大/第K小，是不需要对整个数组进行O(NlogN)的排序的！因为可以通过快排切分直接O(N)找到第K大的数（比如求中位数就可以用本方法O(N)找到第mid大的数，如果只会先排序再找的话，那啥...基本上就交代了叭(ﾟ ʖ ﾟ)...）。

因此本题先通过快排切分排好第K小的数，根据快排切分的性质，它左边的K - 1个数都小于等于它，因此它以及它左边的数就是我们要找的前K小的数。

下面代码给出了详细的注释，没啥好啰嗦的，就是快排模版要记牢哈～

快排变形时间复杂度分析：
因为我们是要找下标为k的元素，第一次切分的时候需要遍历整个数组(0 ~ n)找到了下标是j的元素，假如k比j小的话，那么我们下次切分只要遍历数组(0~k-1)的元素就行啦，反之如果k比j大的话，那下次切分只要遍历数组(k+1 ~ n)的元素就行啦，总之平均情况下，可以看作每次调用partition遍历的元素数目都是上一次遍历的1/2，因此时间复杂度是N + N/2 + N/4 + ... + N/N = 2N, 因此时间复杂度是**O(N)**。

7.3 快速排序的随机化版本

在讨论快速排序的平均情况性能的时候，我们的前提假设是：输入数据的所有排列都是等概率的。但是在实际工程中，这个假设并不会总是成立(见练习 7.2-4)。正如在 5.3 节中我们所看到的那样，有时我们可以通过在算法中引入随机性，从而使得算法对于所有的输入都能获得较好的期望性能。很多人都选择随机化版本的快速排序作为大数据输入情况下的排序算法。

在 5.3 节中，我们通过显式地对输入进行重新排列，使得算法实现随机化。当然，对于快速排序我们也可以这么做。但如果采用一种称为**随机抽样**(random sampling)的随机化技术，那么可以使得分析变得更加简单。与始终采用 $A[r]$作为主元的方法不同，随机抽样是从子数组 $A[p..r]$中随机选择一个元素作为主元。为达到这一目的，首先将 $A[r]$与从 $A[p..r]$中随机选出的一个元素交换。通过对序列 p，…，r 的随机抽样，我们可以保证主元元素 $x=A[r]$是等概率地从子数组的 $r-p+1$ 个元素中选取的。因为主元元素是随机选取的，我们期望在平均情况下，对输入数组的划分是比较均衡的。

方法 1：快排变形 o(N)【顺序按大小顺序】【随机版本的快排】

```
class Solution {
    public int[] smallestK(int[] arr, int k) {
        if (k == 0 || arr.length == 0) return new int[0];
```

```
        // 最后一个参数传入我们要找的下标（第 k 小的数下标是 k-1）
        return quickSort(arr, 0, arr.length - 1, k -1);
    }
    private int[] quickSort(int[] arr, int low, int high, int k){
        int j = partition(arr, low, high);
        if (j == k) return Arrays.copyOf(arr, j+1);
        return j > k ? quickSort(arr, low, j - 1, k) : quickSort(arr, j
 + 1, high, k);
    }
    // 快排切分，返回下标 j，使得比 nums[j] 小的数都在 j 的左边，比 nums[j] 大的数
都在 j 的右边。
    private int partition(int[] arr, int low, int high){
        int v = arr[low];
        int i = low, j = high + 1;
        while (true) {
            while (++i <= high && arr[i] < v);
            while (--j >= low && arr[j] > v);
            if (i >= j) break;
            int t = arr[j];
            arr[j] = arr[i];
            arr[i] = t;
        }
        arr[low] = arr[j];
        arr[j] = v;
        return j;
    }
}
```

方法 1 完整代码如下：

```
import java.util.*;
public class NumK {
    //acwing 786
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int k = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++){
            arr[i] = sc.nextInt();
        }
        sc.close();
        int[] sortArr = quickSearch(arr,0,n-1,k - 1);
        System.out.println("前 k 小的数为：");
        for (int i = 0; i < sortArr.length; i++) {
            System.out.print(sortArr[i]);
```

```
            System.out.print(" ");
        }
        System.out.println();
        System.out.println("第 k 小的数为：" + sortArr[k - 1]);
    }
    //输出数组
    private static int[] quickSearch(int[] nums, int low, int high, int k)
{
        // 每快排切分 1 次，找到排序后下标为 j 的元素，如果 j 恰好等于 k 就返回 j 以
及 j 左边所有的数；
        int j = partition(nums, low, high);
        if (j == k) {
            return Arrays.copyOf(nums, j+1);
        }
        // 否则根据下标 j 与 k 的大小关系来决定继续切分左段还是右段。
        return j > k? quickSearch(nums, low, j - 1, k): quickSearch(nums,
j + 1, high, k);
    }

    // 快排切分，返回下标 j，使得比 nums[j]小的数都在 j 的左边，比 nums[j]大的数
都在 j 的右边。
    private static int partition(int[] nums, int low, int high) {
        int v = nums[low];
        int i = low, j = high + 1;
        while (true) {
            while (++i <= high && nums[i] < v);
            while (--j >= low && nums[j] > v);
            if (i >= j) {
                break;
            }
            int t = nums[j];
            nums[j] = nums[i];
            nums[i] = t;
        }
        nums[low] = nums[j];
        nums[j] = v;
        return j;
    }
}
```

2. 大根堆（前K小）/小根堆（前K大） O(NlogK)

用堆虽然时间复杂度比快排变形慢了点，但是因为Java中提供了现成的PriorityQueue（默认小根堆），所以不需要自己写大段的模版代码，因此实现起来最简单，没几行代码，写起来很快~~面试的时候可以先快点写出这个方案

注意本题是求**前K小**，因此用一个容量为K的**大根堆**（每次poll出最大的数，那堆中保留的就是前K小啦）。注意不是小根堆嗷！小根堆的话需要把全部的元素都入堆，那是O(NlogN)，就不是O(NlogK)啦~~

方法 2：大根堆（前 K 小）/大根堆（前 K 大） o（NlogK）【顺序与原数组相同】

```
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if (k == 0 || arr.length == 0) return new int[0];
        Queue<Integer> pq = new PriorityQueue<>((v1, v2) -> v2 - v1);
        for (int num: arr) {
            if (pq.size() < k) {
                pq.offer(num);
            } else if (num < pq.peek()){
                pq.poll();
                pq.offer(num);
            }
        }
        int[] res = new int[pq.size()];
        int idx = 0;
        for (int num : pq) {
            res[idx++] = num;
        }
        return res;
    }
}
```

其中 main 函数主要代码

```
public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int k = sc.nextInt();
    int[] arr = new int[n];
    for (int i = 0; i < n; i++){
        arr[i] = sc.nextInt();
    }
    sc.close();
    int[] bh = BigHeap(arr, k);
    System.out.println("大根堆版本：前 k 小的数为：");
    for (int i = 0; i < sortArr.length; i++) {
```

```
            System.out.print(bh[i]);
            System.out.print(" ");
        }
        System.out.println();
    }
```

3. 二叉搜索树也可以O(NlogK)解决TopK问题嗷～

BST相对没有前两种方法辣么热门，但是也很简单，和大根堆的思路差不多～不得不提的是，与前两种方法相比，**BST的优势就是求得的前K个数字保证是有序的。**

因为有重复的数字，所以用的是TreeMap而不是TreeSet（有的语言的标准库自带TreeMultiset，也是可以的）。TreeMap的key是数字，value是该数字的个数。我们遍历数组中的数字，维护一个数字总个数为K的TreeMap，每遍历一个元素：

1. 若目前map中数字个数小于K，则将map中当前数字对应的个数+1；

2. 否则，判断当前数字与map中最大的数字的大小关系：若当前数字大于等于map中的最大数字，就直接跳过该数字；若当前数字小于map中的最大数字，则将map中当前数字对应的个数+1，并将map中最大数字对应的个数减1.

方法 3：二叉搜索树

```
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if (k == 0 || arr.length == 0) {
            return new int[0];
        }
        // TreeMap 的 key 是数字，value 是该数字的个数。
        // cnt 表示当前 map 总共存了多少个数字。
        TreeMap<Integer, Integer> map = new TreeMap<>();
        int cnt = 0;
        for (int num: arr) {
            // 1. 遍历数组，若当前 map 中的数字个数小于 k，则 map 中当前数字对应个数+1
            if (cnt < k) {
                map.put(num, map.getOrDefault(num, 0) + 1);
                cnt++;
                continue;
            }
            // 2. 否则，取出 map 中最大的 Key（即最大的数字），判断当前数字与 map 中最大数字的大小关系：
            //    若当前数字比 map 中最大的数字还大(或等于)，就直接忽略；
            //    若当前数字比 map 中最大的数字小，则将当前数字加入 map 中，并将 map 中的最大数字的个数-1。
            Map.Entry<Integer, Integer> entry = map.lastEntry();
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            if (entry.getKey() > num) {
                map.put(num, map.getOrDefault(num, 0) + 1);
                if (entry.getValue() == 1) {
                    map.pollLastEntry();
                } else {
                    map.put(entry.getKey(), entry.getValue() - 1);
                }
            }

        }

        // 最后返回 map 中的元素
        int[] res = new int[k];
        int idx = 0;
        for (Map.Entry<Integer, Integer> entry: map.entrySet()) {
            int freq = entry.getValue();
            while (freq-- > 0) {
                res[idx++] = entry.getKey();
            }
        }
        return res;
    }
}
```

说到这里了，就额外再说一下吧，Java的标准库中虽然没有TreeMultiset，但是Guava提供了TreeMultiset，因此也可以通过引入Guava包来实现嗷，不是考察重点，简单贴下大概代码。

yes的练级攻略

```
// ✘leetcode不支持引入第三方包哦
public int[] getLeastNumbers(int[] arr, int k) {
    if (k == 0 || arr.length == 0) {
        return new int[0];
    }
    // TreeMultiset 中允许有重复元素，所以就不需要用TreeMap了。
    TreeMultiset<Integer> set = TreeMultiset.create();
    for(int num: arr) {
        // 1. 遍历每个数字，如果set中的数量小于K，则直接将当前数字加入set中。
        if (set.size() < k) {
            set.add(num);
            continue;
        }
        // 2. 否则判断当前数字与set中最大数字的大小关系：
        //    若当前数字大于等于set中的最大数字，则直接跳过该数字；
        //    若当前数字小于set中的最大数字，则将当前数字加入set，并将set中最大数字的个数-1
        Multiset.Entry<Integer> lastEntry = set.lastEntry();
        if (num < lastEntry.getElement()) {
            set.remove(lastEntry.getElement(), 1);
            set.add(num);
        }
    }
```

```
    // 返回set中的元素
    int[] res = new int[k];
    int idx = 0;
    for(int num: set) {
        res[idx++] = num;
    }
    return res;
}
```

方法 4：桶排序（2ms）

4. 对于数据范围有限的情况例如本题，可以直接计数排序O(N)高效解决～

```
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if (k == 0 || arr.length == 0) {
            return new int[0];
        }
        // 统计每个数字出现的次数
        int[] counter = new int[10001];
        for (int num: arr) {
            counter[num]++;
        }
        // 根据 counter 数组从头找出 k 个数作为返回结果
```

```
            int[] res = new int[k];
            int idx = 0;
            for (int num = 0; num < counter.length; num++) {
                while (counter[num]-- > 0 && idx < k) {
                    res[idx++] = num;
                }
                if (idx == k) {
                    break;
                }
            }
            return res;
        }
}
```

- 数组中第 k 个最大元素（215）【快排，堆排，桶排】

> 215. 数组中的第K个最大元素
>
> 难度 中等 624 收藏 分享 切换为英文 关注 反馈
>
> 在未排序的数组中找到第 **k** 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
>
> **示例 1:**
>
> **输入:** [3,2,1,5,6,4] 和 k = 2
> **输出:** 5
>
> **示例 2:**
>
> **输入:** [3,2,3,1,2,4,5,5,6] 和 k = 4
> **输出:** 4
>
> **说明:**
>
> 你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

1. 冒泡排序：时间复杂度为 O(n^2)，空间复杂度为 O(1)，稳定排序；因为时间复杂度是 O(n^2)，所以对于大规模数据肯定是不适用的； ps.实际开发中冒泡排序很少使用，更多地是使用插入排序，虽然时间复杂度都是 O(n^2)，但是冒牌排序每次交换元素对，需要使用三次赋值，而插入排序只需要一次。还有一个问题，实际开发中为什么有时不用 O(nlogn)的快排和归并，因为快排虽然是原地排序算法，但是不是稳定排序算法；而归并排序虽然是稳定的排序算法，但并不是原地排序算法；所以当需要原地的稳定排序算法时，就必须用 O(n^2)的排序算法了，而 O(n^2)的排序算法中，插入排序是最优的。至于为什么实际开发中为什么需要稳定的排序算法，这个可以参考《数据结构与算法之美》上的例子哈。
2. 桶排序：(感觉从你的实现来看更应该称之为计数排序 CountingSort，它应该算是桶排序的一种特殊情况）虽然它时间复杂度为 O(n)，线性时间复杂度，看上去很完美，但它更适用于待排序的数据范围不是很大的情况，比如：年龄、分数等，当待排序的数据范围很大时，比如-23829839484~+1847242938292222，它就需要一个很大的计数数组来计数，占用的内存空间很大，所以用在这感觉不是很合适；
3. 堆排序：固然时间复杂度是 O(nlogn)，空间复杂度为 O(k)，这里可以近似为 O(1)，用在这里确实没问题。这里仅提醒一点，堆更适合用来处理动态的数据流，而非是这种静态的数组数据；

更多硬核内容就在公众号「yes 的练级攻略」

4. 借助快速排序中的 partition 函数：这题我觉得最好的解法就是这个。首先，时间复杂度 O(n)，其次，原地算法。

方法 1：快排

第一种：partition（1ms）

Weiwei 全题解

https://leetcode-cn.com/problems/kth-largest-element-in-an-array/solution/partitionfen-er-zhi-zhi-you-xian-dui-lie-java-dai-/

方法二：借助 partition 操作定位到最终排定以后索引为 `len - k` 的那个元素（特别注意：随机化切分元素）

以下是注意事项，因为很重要，所以放在前面说：

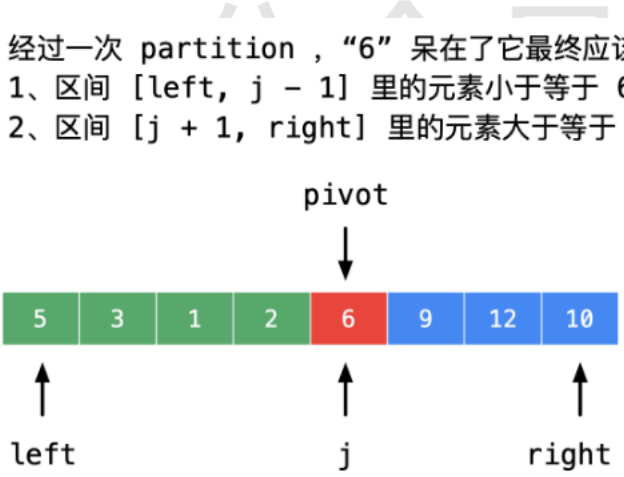
> 快速排序虽然快，但是如果实现得不好，在遇到特殊测试用例的时候，时间复杂度会变得很高。如果你使用 `partition` 的方法完成这道题，时间排名不太理想，可以考虑一下是什么问题，这个问题很常见。

以下的描述基于“快速排序”算法知识的学习，如果忘记的朋友们可以翻一翻自己的《数据结构与算法》教材，复习一下，partition 过程、分治思想和“快速排序”算法的优化。

分析：我们在学习“快速排序”的时候，接触的第 1 个操作就是 partition（切分），简单介绍如下：

partition（切分）操作，使得：

- 对于某个索引 `j`，`nums[j]` 已经排定，即 `nums[j]` 经过 partition（切分）操作以后会放置在它“最终应该放置的地方”；
- `nums[left]` 到 `nums[j - 1]` 中的所有元素都不大于 `nums[j]`；
- `nums[j + 1]` 到 `nums[right]` 中的所有元素都不小于 `nums[j]`。



注意：选择 pivot 的时候，应该先在数组中随机选择一个元素，然后再执行 partition。以避免递归树加深、复杂度最坏的情况出现。

partition（切分）操作总能排定一个元素，还能够知道这个元素它最终所在的位置，这样每经过一次 partition（切分）操作就能缩小搜索的范围，这样的思想叫做“减而治之”（是“分而治之”思想的特例）。

切分过程可以不借助额外的数组空间，仅通过交换数组元素实现。下面是参考代码：

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析：

- 时间复杂度：$O(N)$，这里 N 是数组的长度，理由可以参考本题解下用户 @ZLW 的评论，需要使用主定理进行分析。
- 空间复杂度：$O(1)$，原地排序，没有借助额外的辅助空间。

> 注意：本题必须随机初始化 `pivot` 元素，否则通过时间会很慢，因为测试用例中有极端测试用例。

为了应对极端测试用例，使得递归树加深，可以在循环一开始的时候，随机交换第 1 个元素与它后面的任意 1 个元素的位置；

说明：最极端的是顺序数组与倒序数组，此时递归树画出来是链表，时间复杂度是 $O(N^2)$，根本达不到减治的效果。

快排的时间复杂度我觉得是 T(n) = T(n/2) + n，首先不是一个完全的递归树，按照每次排序刚好定位到中间来想，递归树的下一层只有 T(n/2)，不考虑另一半；其次，每一层递归树都要遍历对应的, "分而治之"的”分“后的数组长度，所以+n 这样来算用 Master method，T(n) = aT(n/b) + f(n), f(n) = n,a=1,b=2, f(n) = Ω(n 的 (以 b 为底 a 的对数 + x) 次方)，x>0, 所以 T(n) = O(f(n))=O(n)。

```
import java.util.Random;
public class Solution {
    private static Random random = new Random(System.currentTimeMillis(
));
    public int findKthLargest(int[] nums, int k) {
        if (nums.length < k) return -1;
        int len = nums.length;
        int target = len - k;
        int left = 0;
        int right = len - 1;
        while (left <= right) {
            int index = partition(nums, left, right);
            if (index < target) left = index + 1;
            else if (index > target) right = index - 1;
            else return nums[index];
        }
        return -1;
    }

    // 在区间 [left, right] 这个区间执行 partition 操作

    private int partition(int[] nums, int left, int right) {
        // 在区间随机选择一个元素作为标定点
        if (right > left) {
            int randomIndex = left + 1 + random.nextInt(right - left);
            swap(nums, left, randomIndex);
        }

        int tmp = nums[left];
        int j = left;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        for (int i = left + 1; i <= right; i++) {
            if (nums[i] < tmp ) {
                j++;
                swap(nums, j, i);
            }
        }
        swap(nums, left, j);
        return j;
    }

    private void swap(int[] nums, int index1, int index2) {
        int temp = nums[index1];
        nums[index1] = nums[index2];
        nums[index2] = temp;
    }
}
```

前 K 大的主函数

```
public static void main(String[] args){
    int[] nums = {3,2,1,5,6,4};
    int k = 2;
    int[] res = new int[k];
    for (int i = 0; i < k; i++){
        res[i] = findKthLargest(nums, i+1);
        System.out.print(res[i]);
        System.out.print(" ");
    }
}
```

第二种（13ms）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法3：基于快速排序的方法，时间复杂度为o(n)

选一个枢轴点，用快排的方法将数组分为两部分，位于枢轴点左边的数都比它大，位于枢轴点右边的数都比它小，

1）如果枢轴点的索引刚好是k-1,则此时它对应的就是数组的第k大的数；

2）如果比k-1大，那么第k大的数位于它的左边部分；

3）如果比k-1小，那么第k大的数位于它的右边部分；

图示一下partition快排的情况：

求下数组中第2大的数。

3	1	2	5	4	7	6

temp=nums[low]=3,以3为枢轴点，进行快排。

第一次，low=0,high=6,temp=3,经过快排后的结果：

6	7	4	5	3	2	1

此时partition排序后获得的index=4，4>2，那么第2大的数位于枢轴点的左边。

第二次，只针对上图中左半边进行排序。

temp=nums[low]=6,以6为枢轴点进行第2次快排。

6	7	4	5	3	2	1

第2次，low=0，high=4-1=3，temp=6；

排序后的结果：

7	6	4	5	3	2	1

此时快排后的index=1,1=2-1，所以枢轴点对应的就是第k大的数。

结束！

https://blog.csdn.net/c

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        return findTopK(nums, 0, nums.length - 1, k);
    }
    private int findTopK(int[] nums, int start, int end, int k){
        int low = start, high = end;
        int tmp = nums[low];
        while (low < high){
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            while (low < high && nums[high] <= tmp) high--;
            nums[low] = nums[high];
            while (low < high && nums[low] >= tmp) low++;
            nums[high] = nums[low];
        }
        nums[high] = tmp;
        if (high == k -1) return tmp;
        else if (high > k-1) return findTopK(nums, start, high - 1, k);
        else return findTopK(nums, high + 1, end, k);
    }
}
```

方法 2：堆排序

方法三：优先队列

优先队列的思路是很朴素的。因为第 `K` 大元素，其实就是整个数组排序以后后半部分最小的那个元素。因此，我们可以维护一个有 `K` 个元素的最小堆：

1、如果当前堆不满，直接添加；

2、堆满的时候，如果新读到的数小于等于堆顶，肯定不是我们要找的元素，只有新都到的数大于堆顶的时候，才将堆顶拿出，然后放入新读到的数，进而让堆自己去调整内部结构。

说明：这里最合适的操作其实是 `replace`，即直接把新读进来的元素放在堆顶，然后执行下沉（`siftDown`）操作。Java 当中的 `PriorityQueue` 没有提供这个操作，只好先 `poll()` 再 `offer()`。

优先队列的写法就很多了，这里例举一下我能想到的（以下的写法大同小异，没有本质差别）。

假设数组有 `len` 个元素。

思路1：把 `len` 个元素都放入一个最小堆中，然后再 `pop()` 出 `len - k` 个元素，此时最小堆只剩下 `k` 个元素，堆顶元素就是数组中的第 `k` 个最大元素。

思路2：把 `len` 个元素都放入一个最大堆中，然后再 `pop()` 出 `k - 1` 个元素，因为前 `k - 1` 大的元素都被弹出了，此时最大堆的堆顶元素就是数组中的第 `k` 个最大元素。

```
import java.util.PriorityQueue;
public class Solution {
    // 所以分界点就是 k = len - k
    public int findKthLargest(int[] nums, int k) {
        int len = nums.length;
        if (k <= len - k) {
            // System.out.println("使用最小堆");
            // 特例：k = 1，用容量为 k 的最小堆
            // 使用一个含有 k 个元素的最小堆
            PriorityQueue<Integer> minHeap = new PriorityQueue<>(k, (a,
 b) -> a - b);
            for (int i = 0; i < k; i++) {
                minHeap.add(nums[i]);
            }
            for (int i = k; i < len; i++) {
                // 看一眼，不拿出，因为有可能没有必要替换
                Integer topEle = minHeap.peek();
```

```
                // 只要当前遍历的元素比堆顶元素大，堆顶弹出，遍历的元素进去
                if (nums[i] > topEle) {
                    minHeap.poll();
                    minHeap.add(nums[i]);
                }
            }
            return minHeap.peek();

        } else {
            // System.out.println("使用最大堆");
            assert k > len - k;
            // 特例：k = 100，用容量为 len - k + 1 的最大堆
            int capacity = len - k + 1;
            PriorityQueue<Integer> maxHeap = new PriorityQueue<>(capaci
ty, (a, b) -> b - a);
            for (int i = 0; i < capacity; i++) {
                maxHeap.add(nums[i]);
            }
            for (int i = capacity; i < len; i++) {
                // 看一眼，不拿出，因为有可能没有必要替换
                Integer topEle = maxHeap.peek();
                // 只要当前遍历的元素比堆顶元素大，堆顶弹出，遍历的元素进去
                if (nums[i] < topEle) {
                    maxHeap.poll();
                    maxHeap.add(nums[i]);
                }
            }
            return maxHeap.peek();
        }
    }
}
```

本地的代码，包含用自带函数写的小根堆

```
import java.util.*;
public class NumK {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int k = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++){
            arr[i] = sc.nextInt();
        }
        sc.close();
        int TopK = findTopK(arr, 0, arr.length - 1, k);
```

```
        System.out.println("快排版本：第 k 大的数为：" + TopK);

        int[] resTopK = new int[k];
        System.out.println("快排版本：前 k 大的数为：");
        for (int i = 0; i < k; i++){
            resTopK[i] = findTopK(arr, 0, arr.length - 1, i + 1);
            System.out.print(resTopK[i]);
            System.out.print(" ");
        }
        System.out.println();

        int tsh = TopKSmallHeap(arr,k);
        System.out.println("小根堆版本：第 k 大的数为：" + tsh);

        int[] smallArr = SmallHeap(arr, k);
        System.out.println("小根堆版本：前 k 大的数为：");
        for (int i = 0; i < smallArr.length; i++) {
            System.out.print(smallArr[i]);
            System.out.print(" ");
        }

    }
    //返回第 k 大
    private static int findTopK(int[] nums, int start, int end, int k) {
        int low = start, high = end;
        int tmp = nums[low];//枢轴点
        while (low < high) {
            while (low < high && nums[high] <= tmp) high--;
            //高处遇到比 tmp 小的数，将它放在地位
            nums[low] = nums[high];
            while (low < high && nums[low] >= tmp) low++;
            //低处遇到比 tmp 大的数，把它放到高位
            nums[high] = nums[low];
        }
        nums[high] = tmp; //枢轴点归位
        if (high == k -1) return tmp;
        else if (high > k -1) return findTopK(nums, start, high - 1, k);
        else return findTopK(nums,high + 1, end, k);
    }
    //小根堆前 K 大
前 K 大小根堆
    private static int[] SmallHeap (int[] nums, int k){
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int num : nums) {
```

禁止商用 公众号：yes的练级攻略

```
            pq.add(num);
            if (pq.size() > k) pq.poll();
        }
        int[] res = new int[pq.size()];
        int idx = 0;
        for (int num : pq) {
            res[idx++] = num;
        }
        return res;
    }

    //小根堆第 K 大
    private static int TopKSmallHeap (int[] nums, int k){
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int num : nums) {
            pq.add(num);
            if (pq.size() > k) pq.poll();
        }
        return pq.peek();
    }
```

禁止商用

小根堆第 k 大的优化

```
//上面的多一次查询比较，过滤排除本身就无需入堆的元素，相较于后者多了一次查询代价 O（1）,但是可能节省 O（K）
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int num : nums) {
            if (pq.size() < k) {
                pq.offer(num);
            }else if (num > pq.peek()) {
                pq.poll();
                pq.offer(num);
            }
        }
        return pq.peek();
    }
}
```

方法 3：桶排序

先找到数组中的最大最小值，然后创建 bucket 数组，数组大小为 max-min+1；遍历 nums 数组，以 nums[i]-min 作为 bucket 索引，该值出现次数为 bucket 数组值。从后往前遍历 bucket 数组，设置计数器 counts，每次 counts+bucket[j] ,当 counts>=k 时，输出 j+min，即是要找的第 k 个最大的数组的值。

时间复杂度 O(d(n+r))，其中 d 是关键码数目，r 是关键码取值范围，n 是记录。空间复杂度 o(r)

原文链接：https://blog.csdn.net/qq_34538534/java/article/details/93377862

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        int max = Integer.MIN_VALUE, min = Integer.MAX_VALUE;
        for (int num : nums) {
            max = Math.max(max, num);
            min = Math.min(min, num);
        }
        int[] bucket = new int[max - min + 1];
        for (int i = 0; i < nums.length; i++) {
            bucket[nums[i] - min]++;
        }
        int cnt = 0;
        for (int j = bucket.length - 1; j>= 0; j--){
            cnt += bucket[j];
            if (cnt >= k) return j + min;
        }
        return -1;
    }
}
```

- 第 k 个数（acwing786）【快排求第 k 小的数】

786. 第k个数

题目 讨论 题解 视频讲解

给定一个长度为n的整数数列，以及一个整数k，请用快速选择算法求出数列的第k小的数是多少。

输入格式

第一行包含两个整数 n 和 k。

第二行包含 n 个整数（所有整数均在1~10^9范围内），表示整数数列。

输出格式

输出一个整数，表示数列的第k小数。

数据范围

$1 \le n \le 100000$,
$1 \le k \le n$

输入样例：

```
5 3
2 4 1 5 3
```

输出样例：

```
3
```

```java
import java.util.*;
public class Main {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int k = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++){
```

```
            arr[i] = sc.nextInt();
        }
        sc.close();

        int small = SmallKquickSearch(arr, 0, arr.length - 1, k - 1);
        System.out.println(small);
    }

    //前第小
    private static int SmallKquickSearch(int[] nums, int low, int high, int k) {
        // 每快排切分 1 次，找到排序后下标为 j 的元素，如果 j 恰好等于 k 就返回 j 以及 j 左边所有的数；
        int j = partition(nums, low, high);
        if (j == k) {
            return nums[j];
        }
        // 否则根据下标 j 与 k 的大小关系来决定继续切分左段还是右段。
        return j > k? SmallKquickSearch(nums, low, j - 1, k): SmallKquickSearch(nums, j + 1, high, k);
    }

    // 快排切分，返回下标 j，使得比 nums[j]小的数都在 j 的左边，比 nums[j]大的数都在 j 的右边。
    private static int partition(int[] nums, int low, int high) {
        int v = nums[low];
        int i = low, j = high + 1;
        while (true) {
            while (++i <= high && nums[i] < v);
            while (--j >= low && nums[j] > v);
            if (i >= j) {
                break;
            }
            int t = nums[j];
            nums[j] = nums[i];
            nums[i] = t;
        }
        nums[low] = nums[j];
        nums[j] = v;
        return j;
    }
}
```

- 颜色分类（75）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

75. 颜色分类

难度 中等 515 收藏 分享 切换为英文 关注 反馈

给定一个包含红色、白色和蓝色，一共 *n* 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

注意:
不能使用代码库中的排序函数来解决这道题。

示例:

```
输入: [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]
```

进阶:

- 一个直观的解决方案是使用计数排序的两趟扫描算法。
  首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。
- 你能想出一个仅使用常数空间的一趟扫描算法吗?

https://leetcode-cn.com/problems/sort-colors/solution/kuai-su-pai-xu-partition-guo-cheng-she-ji-xun-huan/

做对这道题需要熟悉快速排序的 partition 过程。

- partition 过程经过一次扫描把整个数组分成三个部分，正好符合这个问题的场景。
- 写对这道题的方法是，把我们对循环不变量的定义作为注释先写出来，然后再编码。

（以下是复习 partition 过程，与本题解法无直接关系，直接跳过不影响后续内容的阅读。）

下面的动画，从一个中间的状态开始演示，直到 partition 完成。

如果循环不变量是这样定义的：

- 所有在子区间 `[0, zero)` 的元素都等于 `0`；
- 所有在子区间 `[zero, i)` 的元素都等于 `1`；
- 所有在子区间 `[two, len - 1]` 的元素都等于 `2`。

说明：设计循环不变量的原则是“不重不漏”。

1、`len` 是数组的长度；
2、变量 `zero` 是前两个子区间的分界点，一个是闭区间，另一个就必须是开区间；
3、变量 `i` 是循环变量，一般设置为开区间，表示 `i` 之前的元素是遍历过的；
4、`two` 是另一个分界线，我设计成闭区间。

于是编码要解决以下三个问题：

> 变量初始化应该如何定义。
> 在遍历的时候，是先加减还是先交换。
> 什么时候循环终止。

处理这三个问题，完全看循环不变量的定义。

- 编码的时候，`zero` 和 `two` 初始化的值就应该保证上面的三个子区间全为空。
- 在遍历的过程中，“索引先加减再交换”、还是“先交换再加减”就看初始化的时候变量在哪里。
- 退出循环的条件也看上面定义的循环不变量，在 `i == two` 成立的时候，上面的三个子区间就正好“不重不漏”地覆盖了整个数组，并且给出的性质成立，题目的任务也就完成了。

以下给出不同的写法，循环不变量的定义写在了注释中。使用“循环不变量”编码是为了便于我们处理细节，也方便别人看懂你的算法。

方法 1：快排 partition

```
import java.util.Arrays;
public class Solution {
```

```
    public void sortColors(int[] nums) {
        int len = nums.length;
        if (len < 2) return;
        // all in [0, zero) = 0
        // all in [zero, i) = 1
        // all in [two, len - 1] = 2

        // 循环终止条件是 i == two，那么循环可以继续的条件是 i < two
        // 为了保证初始化的时候 [0, zero) 为空，设置 zero = 0，
        // 所以下面遍历到 0 的时候，先交换，再加
        int zero = 0;

        // 为了保证初始化的时候 [two, len - 1] 为空，设置 two = len
        // 所以下面遍历到 2 的时候，先减，再交换
        int two = len;
        int i = 0;
        // 当 i == two 上面的三个子区间正好覆盖了全部数组
        // 因此，循环可以继续的条件是 i < two
        while (i < two) {
            if (nums[i] == 0) {
                swap(nums, i, zero);
                zero++;
                i++;
            } else if (nums[i] == 1) {
                i++;
            } else {
                two--;
                swap(nums, i, two);
            }
        }
    }

    private void swap(int[] nums, int index1, int index2) {
        int temp = nums[index1];
        nums[index1] = nums[index2];
        nums[index2] = temp;
    }
}
```

https://leetcode-cn.com/problems/sort-colors/solution/yan-se-fen-lei-by-leetcode/

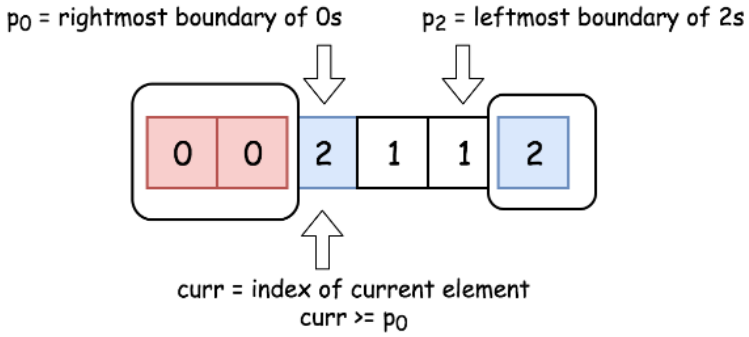
方法一：一次遍历

直觉

本问题被称为 荷兰国旗问题
，最初由 Edsger W. Dijkstra提出。
其主要思想是给每个数字设定一种颜色，并按照荷兰国旗颜色的顺序进行调整。

Dutch National Flag Problem

我们用三个指针（p0, p2 和curr）来分别追踪0的最右边界，2的最左边界和当前考虑的元素。



本解法的思路是沿着数组移动 `curr` 指针，若 `nums[curr] = 0` ，则将其与 `nums[p0]` 互换；若 `nums[curr] = 2` ，则与 `nums[p2]` 互换。

算法

- 初始化0的最右边界： `p0 = 0` 。在整个算法执行过程中 `nums[idx < p0] = 0` .
- 初始化2的最左边界： `p2 = n - 1` 。在整个算法执行过程中 `nums[idx > p2] = 2` .
- 初始化当前考虑的元素序号： `curr = 0` .
- While `curr <= p2` ：
  - 若 `nums[curr] = 0` ：交换第 `curr` 个和第 `p0` 个元素，并将指针都向右移。
  - 若 `nums[curr] = 2` ：交换第 `curr` 个和第 `p2` 个元素，并将 `p2` 指针左移。
  - 若 `nums[curr] = 1` ：将指针 `curr` 右移。

方法 2：三指针【荷兰三色旗】

```
class Solution {
  /*
  荷兰三色旗问题解
  */
```

```
    public void sortColors(int[] nums) {
        // 对于所有 idx < i : nums[idx < i] = 0
        // j 是当前考虑元素的下标
        int p0 = 0, cur = 0;
        // 对于所有 idx > k : nums[idx > k] = 2
        int p2 = nums.length - 1;

        while (cur <= p2) {
            if (nums[cur] == 0) {
                // 交换第 p0 个和第 cur 个元素
                swap(nums, p0, cur);
                p0++;
                cur++;
            }
            else if (nums[cur] == 2) {
                // 交换第 k 个和第 cur 个元素
                // p2--
                swap(nums, cur, p2);
                p2--;
            }
            else cur++;
        }
    }
    private void swap(int[] nums, int index1, int index2) {
        int temp = nums[index1];
        nums[index1] = nums[index2];
        nums[index2] = temp;
    }
}
```

(5) 双路快排

https://blog.csdn.net/qq_37113604/article/details/96110218?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase

```
public class QuickSort3 {

    public static void main(String[] args) {
        int[] array = ArrayUtil.bigArray();
        long time1 = System.currentTimeMillis();
        QuickSort(array,0,array.length-1);
        long time2 = System.currentTimeMillis();
        System.out.println(time2 - time1);
    }
        private static void QuickSort(int[] arr, int l, int r) {
            if(l>=r)
```

```
            return;

        swap(arr,l,(int)Math.random()*(r-l+1)+l);
        //满足 arr[l+1,i)<=v,arr(j,r]>=v
        int v=arr[l];
        int i=l+1,j=r;
        while(true) {
            //从左到右扫描，扫描出第一个比 base 大的元素，然后 i 停在那里
            while(arr[i]<v&&i<r) //arr[i]不能=v，会导致 v 聚集在一边
                i++;
            //从右到左扫描，扫描出第一个比 base 小的元素，然后 j 停在那里
            while(arr[j]>v&&j>l)
                j--;
            if(i>=j)
                break;
            swap(arr, i, j);
            i++;
            j--;
        }
        //将基准值交换到合适位置
         //因为 j 能遍历到 0，而 i 只能从第 1 个往后找，如果第 0 个是最小的数，把会最小的数换到中间
        swap(arr, l, j);

        QuickSort(arr, l, j-1);
        QuickSort(arr, j+1, r);
    }
    private static void swap(int[] arr, int i, int j) {
        if(i!=j) {
            int temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
        }
    }
}
```

(6) 三路快排

```
private static void quickSort3Ways(int[] arr,int left,int right){
    if (left >= right){
        return;
    }
    //基数
    int v = arr[left];
    //左边最接近 v 且等于 v
    int lt = left;
```

禁止商用 公众号：yes的练级攻略

```
        //右边最接近 v 且等于 v
        int rt = right;
        //从第一个数开始比较
        int i=left+1;
        while (i<=rt){
            if(arr[i] < v){
                //把小的数往左放  如果第 arr[1]<arr[0]   交换 i 和 0 上的数
                swap(arr,lt,i);
                i++;
                lt++;
            }else if (arr[i]>v){
                //大数放右边
                swap(arr,rt,i);
                rt--;
            }else{
                i++;
            }
        }
        //为什么这里是 lt-1 与 rt+1  因为 lt 与 rt 索引指向的都是等于中间重复值的索引，中间的重复值已经是中值了
        quickSort3Ways(arr,left,lt-1);
        quickSort3Ways(arr,rt+1,right);
}
```

(7) 选择排序

时间复杂度 $o(n^2)$，空间复杂度 o(1)，不稳定。

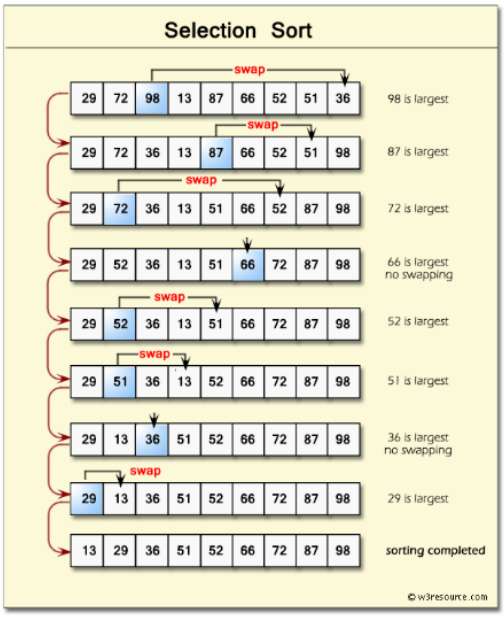
思路 1：遍历全数组，把下标为 0 设置成最小值的下标，开始遍历该下标以后是否有比它小的元素，如果有的话，就调换顺序。

```
public class SelectSort {
  public static void selectSort(int[] arr) {
     if(arr == null || arr.length == 1) {
         return;
     }
     for (int i = 0; i<arr.length-1; i++) {
        for (int j = i+1; j<arr.length; j++) {
           if (arr[i] > arr[j]) {
               int tmp = arr[i];
               arr[i] = arr[j];
               arr[j] = tmp;
           }
        }
     }
  }
}
```

思路 2：与最小值交换

禁止商用 公众号：yes的练级攻略

选择排序（Select Sorting）属于内部排序算法。是从待排序的数据中，按指定的规则选出某一元素，再依规定交换位置后达到排序的目的。



基本思想：

第 1 次从 arr[0] ~ arr[n-1] 中选取最小值 min，与 arr[0] 进行交换；

第 2 次从 arr[1] ~ arr[n-1] 中选取最小值 min，与 arr[1] 进行交换；

.....

第 i 次从 arr[i-1] ~ arr[n-1] 中选取最小值 min，与 arr[i-1] 进行交换。

选择排序思路：

1.选择排序共进行 数组大小 - 1 （arr.length - 1）轮排序；

2.每一轮排序中，又是一个循环（因此选择排序时间复杂度是 O(n^2)），循环的规则如下：

a。先假定当前这个数是最小数min;

b。然后将假定的这个最小数min和后面的每一个数进行比较，如果发现有比当前数更小的数，就重新确定最小数，并得到该最小数的下标；

c。当遍历到数组的最后时，就得到本轮最小数和下标；

d。然后将确定的最小数与该轮排序的第一个数进行交换

以数组 arr = {101, 34, 119, 1} 为例说明：

第一轮排序：{1，34，119， 101}

第一轮排序：{1， 34，119，101} //本轮没有发生未知变化，因为假定的第一个数 34 就是最小值

第一轮排序：{1， 34，101，119}

```
import java.util.Arrays;
public class SelectSort {
    public static void main(String[] args){
        //选择排序 {111,34,119,1}
        int[] arr = {111,34,119,1};
        selectSort(arr);
        System.out.println(Arrays.toString(arr));
    }
    public static void selectSort(int[] arr){
        int min = 0;
        int minIndex = 0;
        //选择排序时间复杂度是 O(n^2)
        //算法 就是先简单 让后复杂。先把一个复杂算法，拆分成简单的问题，然后逐步解决
        for(int i=0; i< arr.length-1; i++){
            min = arr[i]; //假定最小值
            minIndex = i; //假定最小值的索引
            for(int j=i+1; j<arr.length; j++){ //这里是整个数组的长度，因为要遍历整个数组的所有数，找到最小的那个，从第 i+1 个元素开始
                if(min > arr[j]){ //说明 min 不是假定的最小值 arr[j]比 min 还小
                    min = arr[j]; //重置 min，让 min 重返最小值，找到该轮的最小值
                    minIndex = j; //重置 minIndex，找到该轮的最小值索引
                }
            }
            //将最小值 放在 arr[0]，即交换
            //如第一轮下来 i=0 时，min = 1, minIndex = 3
            if(minIndex != i){ //这里是优化的代码，如果 minIndex 没有发生改变，就不执行里面得的代码
                arr[minIndex] = arr[i]; //把 第 i 轮循环开始的，第一个元素 放到下标 minIndex 最小值那个位置上(第一轮 arr[3] = arr[0], 将 111 放在 1 那个位置)
                arr[i] = min; //第一轮的时候，此时 min = 1 是最小值，arr[0] = 1, 把第一个元素设置成最小值
            }
            System.out.println("第" + (i+1) +"轮候后：");
            System.out.println(Arrays.toString(arr));
        }
    }
}
```

(8) 堆排序

时间复杂度 o(nlogn)，空间复杂度 o(1)，不稳定。

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

如果把堆看成是一棵树，我们定义一个堆中的结点的**高度**就为该结点到叶结点最长简单路径上边的数目；进而我们可以把堆的高度定义为根结点的高度。既然一个包含 n 个元素的队可以看做一棵完全二叉树，那么该堆的高度是 $\Theta(\lg n)$（见练习 6.1-2）。我们会发现，堆结构上的一些基本操作的运行时间至多与树的高度成正比，即时间复杂度为 $O(\lg n)$。在本章的剩余部分中，我们将介绍一些基本过程，并说明如何在排序算法和优先队列中应用它们。

- MAX-HEAPIFY 过程：其时间复杂度为 $O(\lg n)$，它是维护最大堆性质的关键。
- BUILD-MAX-HEAP 过程：具有线性时间复杂度，功能是从无序的输入数据数组中构造一个最大堆。
- HEAPSORT 过程：其时间复杂度为 $O(n \lg n)$，功能是对一个数组进行原址排序。
- MAX-HEAP-INSERT、HEAP-EXTRACT-MAX、HEAP-INCREASE-KEY 和 HEAP-MAXIMUM 过程：时间复杂度为 $O(\lg n)$，功能是利用堆实现一个优先队列。

适用于海量数据下最小的 k 个数和最大的 k 个数的题目。

大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

堆排序思路：创建大顶堆/小顶堆，把堆首和堆尾交换位置，把堆的尺寸缩小 1，然后再创建新的大顶堆/小顶堆，直到堆尺寸变为 1.

```
public class HeapSort {
        //堆排序
        public static void heapSort(int [] arr) {
            //边界判断：空集和只有一个节点的判断
            if(arr == null || arr.length == 1) {
                return;
            }
             //构建大顶堆
             for (int i = arr.length/2 - 1; i >= 0 ; i--) {
                 //从第一个非叶子结点从下至上，从右至左调整结构
                 adjustBigHeap(arr, i, arr.length);
             }
             //调整堆结构+交换堆顶元素与末尾元素
             for(int j = arr.length-1; j>0; j--) {
                 //将堆顶元素与末尾元素进行交换
                 int temp = arr[0];
                 arr[0] = arr[j];
                 arr[j] = temp;
                 //重新对堆进行调整
                 adjustBigHeap(arr, 0, j);
             }
         }
         //调整大顶堆
         public static void adjustBigHeap(int [] arr, int i, int length) {
             int tmp = arr[i];
             for (int k = i*2+1; k < length; k = k*2+1) {
                 //如果左子结点小于右子结点，k 指向右子结点
                 if (k+1<length && arr[k]<arr[k+1]) {
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                k++;
            }
            //如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）
            if (arr[k] > tmp) {
                arr[i] = arr[k];
                i = k;
            } else {
                break;
            }
        }
        arr[i] = tmp;
    }

    //逆堆排序
    public static void oppositeHeapSort(int [] arr) {
        if(arr == null || arr.length == 1) {
            return;
        }
        //构建小顶堆
        for (int i = arr.length/2 - 1; i >= 0 ; i--) {
            //从第一个非叶子结点从下至上，从右至左调整结构
            adjustSmallHeap(arr, i, arr.length);
        }
        //调整堆结构+交换堆顶元素与末尾元素
        for(int j = arr.length-1; j>0; j--) {
            //将堆顶元素与末尾元素进行交换
            int temp = arr[0];
            arr[0] = arr[j];
            arr[j] = temp;
            //重新对堆进行调整
            adjustSmallHeap(arr, 0, j);
        }
    }
    //调整小顶堆
    public static void adjustSmallHeap(int [] arr, int i, int length)
{
        int tmp = arr[i];
        for (int k = i*2+1; k < length; k = k*2+1) {
            if (k+1<length && arr[k]>arr[k+1]) {
                k++;
            }
            if (arr[k] < tmp) {
                arr[i] = arr[k];
                i = k;
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                } else {
                    break;
                }
            }
            arr[i] = tmp;
        }
}
```

优先队列实现大根堆和小根堆

```
PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>(); //小顶堆，默认容量为 11
PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(11,new Comparator<Integer>(){ //大顶堆，容量 11
    @Override
    public int compare(Integer i1,Integer i2){
        return i2-i1;
    }
});
```

大根堆可以简写为：

```
PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>((a, b) -> b - a);
```

- 前 K 个高频元素（347）

禁止商用

347. 前 K 个高频元素

难度 中等 361 收藏 分享 切换为英文 关注 反馈

给定一个非空的整数数组，返回其中出现频率前 ***k*** 高的元素。

示例 1:

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

提示:

- 你可以假设给定的 *k* 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
- 你的算法的时间复杂度**必须**优于 O(*n* log *n*) , *n* 是数组的大小。
- 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
- 你可以按任意顺序返回答案。

https://leetcode-cn.com/problems/top-k-frequent-elements/solution/4-chong-fang-fa-miao-sha-topkji-shu-pai-xu-kuai-pa/

更多硬核内容就在公众号「yes 的练级攻略」

一、前言

众所周知，**TopK** 是面试必问的题目，本文提供 4 种解法，其中 **堆** 和 **二叉搜索树** 的时间复杂度是 $O(NlogK)$，**计数排序（桶排序）** 和 **快排变形** 的时间复杂度是 $O(N)$。
因为所有的 TopK 问题的解法都是一样的，所以具体讲解可以看我之前的一篇题解 【Leetcode每日打卡】4种解法秒杀TopK（快排变形/堆/二叉搜索树/计数排序）。

TopK问题，不管是**求前K大/前K小/第K大/第K小**等，都有4种不错的方法喔：

1. O(N)：用快排变形最最最高效解决TopK问题

2. O(NlogK)：大根堆（前K小）/小根堆（前K大）

3. O(NlogK)：二叉搜索树

4. O(N): 对于数据范围有限的情况例如本题，可以直接计数排序O(N)高效解决～

方法 1：小根堆

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // 统计每个数字出现的次数
        Map<Integer, Integer> counter = IntStream.of(nums).boxed().coll
ect(Collectors.toMap(e -> e, e -> 1, Integer::sum));
        // 定义小根堆，根据数字频率自小到大排序
        Queue<Integer> pq = new PriorityQueue<>((v1, v2) -> counter.get
(v1) - counter.get(v2));
        // 遍历数组，维护一个大小为 k 的小根堆：
        // 不足 k 个直接将当前数字加入到堆中；否则判断堆中的最小次数是否小于当
前数字的出现次数，若是，则删掉堆中出现次数最少的一个数字，将当前数字加入堆中。
        counter.forEach((num, cnt) -> {
            if (pq.size() < k) {
                pq.offer(num);
            } else if (counter.get(pq.peek()) < cnt) {
                pq.poll();
                pq.offer(num);
            }
        });
        // 构造返回结果
        int[] res = new int[k];
        int idx = 0;
        for (int num: pq) {
            res[idx++] = num;
        }
        return res;
    }
}
```

方法 2：二叉搜索树

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // 统计每个数字出现的次数
        Map<Integer, Integer> counterMap = IntStream.of(nums).boxed().c
ollect(Collectors.toMap(e -> e, e -> 1, Integer::sum));
        // 定义二叉搜索树：key 是数字出现的次数，value 是出现了 key 次的数字列
表。
        TreeMap<Integer, List<Integer>> treeMap = new TreeMap<>();
        // 维护一个有 k 个数字的二叉搜索树：
        // 不足 k 个直接将当前数字加入到树中；否则判断当前树中的最小次数是否小
于当前数字的出现次数，若是，则删掉树中出现次数最少的一个数字，将当前数字加入树中。
        int count = 0;
        for(Map.Entry<Integer, Integer> entry: counterMap.entrySet()) {
            int num = entry.getKey();
            int cnt = entry.getValue();
            if (count < k) {
                treeMap.computeIfAbsent(cnt, ArrayList::new).add(num);
                count++;
            } else {
                Map.Entry<Integer, List<Integer>> firstEntry = treeMap.
firstEntry();
                if (cnt > firstEntry.getKey()) {
                    treeMap.computeIfAbsent(cnt, ArrayList::new).add(nu
m);
                    List<Integer> list = firstEntry.getValue();
                    if (list.size() == 1) {
                        treeMap.pollFirstEntry();
                    } else {
                        list.remove(list.size() - 1);
                    }
                }
            }
        }
        // 构造返回结果
        int[] res = new int[k];
        int idx = 0;
        for (List<Integer> list: treeMap.values()) {
            for (int num: list) {
                res[idx++] = num;
            }
        }
        return res;
    }
```

```
}
```

方法 3：桶排序

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // 统计每个数字出现的次数
        Map<Integer, Integer> counterMap = IntStream.of(nums).boxed().c
ollect(Collectors.toMap(e -> e, e -> 1, Integer::sum));
        // 一个数字最多出现 nums.length 次，因此定义一个长度
为 nums.length + 1 的数组，freqList[i] 中存储出现次数为 i 的所有数字。
        List<Integer>[] freqList = new List[nums.length + 1];
        for (int i = 0; i < freqList.length; i++) {
            freqList[i] = new ArrayList<>();
        }
        counterMap.forEach((num, freq) -> {
            freqList[freq].add(num);
        });
        // 按照出现频次，从大到小遍历频次数组，构造返回结果。
        int[] res = new int[k];
        int idx = 0;
        for (int freq = freqList.length - 1; freq > 0; freq--) {
            for (int num: freqList[freq]) {
                res[idx++] = num;
                if (idx == k) {
                    return res;
                }
            }
        }
        return res;
    }
}
```

方法 4：快排

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // 统计每个数字出现的次数
        Map<Integer, Integer> counterMap = IntStream.of(nums).boxed().c
ollect(Collectors.toMap(e -> e, e -> 1, Integer::sum));
        // 构造 Pair 数组，Pair.num 表示数值，Pair.freq 表示数字出现的次数
        Pair[] pairs = IntStream.of(nums).distinct().boxed().map(num ->
 new Pair(num, counterMap.get(num))).toArray(Pair[]::new);
        // 使用快排变形 O(N) 获取 Pair 数组中频次最大的 k 个元素（第 4 个参数
是下标，因此是 k - 1）。
        Pair[] topKPairs = quickSelect(pairs, 0, pairs.length - 1, k -
1);
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        // 构造返回结果
        int[] res = new int[k];
        int idx = 0;
        for (Pair pair: topKPairs) {
            res[idx++] = pair.num;
        }
        return res;
    }

    private Pair[] quickSelect(Pair[] pairs, int lo, int hi, int idx) {
        if (lo > hi) {
            return new Pair[0];
        }
        // 快排切分后，j 左边的数字出现的次数都大于等于 pairs[j].freq，j 右边
的数字出现的次数都小于等于 pairs[j].freq。
        int j = partition(pairs, lo, hi);
        // 如果 j 正好等于目标 idx，说明当前 pairs 数组中的[0, idx]就是出现次
数最大的 K 个元素。
        if (j == idx) {
            return Arrays.copyOf(pairs, idx + 1);
        }
        // 否则根据 j 与 idx 的大小关系判断找左段还是右段
        return j < idx? quickSelect(pairs, j + 1, hi, idx): quickSelect
(pairs, lo, j - 1, idx);
    }

    private int partition(Pair[] pairs, int lo, int hi) {
        Pair v = pairs[lo];
        int i = lo, j = hi + 1;
        while (true) {
            while(++i <= hi && pairs[i].freq > v.freq);
            while(--j >= lo && pairs[j].freq < v.freq);
            if (i >= j) {
                break;
            }
            Pair tmp = pairs[i];
            pairs[i] = pairs[j];
            pairs[j] = tmp;
        }
        pairs[lo] = pairs[j];
        pairs[j] = v;
        return j;
    }
}
```

```
class Pair {
    int num;
    int freq;

    public Pair(int num, int freq) {
        this.num = num;
        this.freq = freq;
    }
}
```

- 前 K 个高频单词（692）

692. 前K个高频单词

难度 中等 136 收藏 分享 切换为英文 关注 反馈

给一非空的单词列表，返回前 *k* 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。

示例 1：

```
输入：["i", "love", "leetcode", "i", "love", "coding"], k = 2
输出：["i", "love"]
解析："i" 和 "love" 为出现次数最多的两个单词，均为2次。
    注意，按字母顺序 "i" 在 "love" 之前。
```

示例 2：

```
输入：["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4
输出：["the", "is", "sunny", "day"]
解析："the", "is", "sunny" 和 "day" 是出现次数最多的四个单词，
    出现次数依次为 4, 3, 2 和 1 次。
```

注意：

1. 假定 *k* 总为有效值， 1 ≤ *k* ≤ 集合元素数。
2. 输入的单词均由小写字母组成。

扩展练习：

1. 尝试以 *O*(*n* log *k*) 时间复杂度和 *O*(*n*) 空间复杂度解决。

https://leetcode-cn.com/problems/top-k-frequent-words/solution/qian-kge-gao-pin-dan-ci-by-leetcode/

方法二：堆

算法：

- 计算每个单词的频率，然后将其添加到存储到大小为 `k` 的小根堆中。它将频率最小的候选项放在堆的顶部。最后，我们从堆中弹出最多 `k` 次，并反转结果，就可以得到前 `k` 个高频单词。
- 在 Python 中，我们使用 `heapq\heapify`，它可以在线性时间内将列表转换为堆，从而简化了我们的工作。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(N \log k)$。其中 N 是 `words` 的长度。我们用 $O(N)$ 的时间计算每个单词的频率，然后将 N 个单词添加到堆中，添加每个单词的时间为 $O(\log k)$。最后，我们从堆中弹出最多 k 次。因为 $k \leq N$ 的值，总共是 $O(N \log k)$。
- 空间复杂度：$O(N)$，用于存储我们计数的空间

```
class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        Map<String, Integer> count = new HashMap();
        for (String word: words) {
            count.put(word, count.getOrDefault(word, 0) + 1);
        }
        PriorityQueue<String> heap = new PriorityQueue<String>(
                (w1, w2) -> count.get(w1).equals(count.get(w2)) ?
                w2.compareTo(w1) : count.get(w1) - count.get(w2) );

        for (String word: count.keySet()) {
            heap.offer(word);
            if (heap.size() > k) heap.poll();
        }

        List<String> ans = new ArrayList();
        while (!heap.isEmpty()) ans.add(heap.poll());
        Collections.reverse(ans);
        return ans;
    }
}
```

- 单词频率（面试题 16.02）【hashmap 或者字典序】

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.02. 单词频率

难度 中等 7 收藏 分享 切换为英文 关注 反馈

设计一个方法，找出任意指定单词在一本书中的出现频率。

你的实现应该支持如下操作：

- `WordsFrequency(book)` 构造函数，参数为字符串数组构成的一本书
- `get(word)` 查询指定单词在书中出现的频率

示例：

```
WordsFrequency wordsFrequency = new WordsFrequency({"i", "have", "an", "apple", "he", "have", "a",
"pen"});
wordsFrequency.get("you"); //返回0，"you"没有出现过
wordsFrequency.get("have"); //返回2，"have"出现2次
wordsFrequency.get("an"); //返回1
wordsFrequency.get("apple"); //返回1
wordsFrequency.get("pen"); //返回1
```

提示：

- `book[i]` 中只包含小写字母
- `1 <= book.length <= 100000`
- `1 <= book[i].length <= 10`
- `get` 函数的调用次数不会超过100000

https://leetcode-cn.com/problems/words-frequency-lcci/solution/java-hashmap-by-songhouhou/

```java
class WordsFrequency {
    private HashMap<String, Integer> map = new HashMap<>();
    public WordsFrequency(String[] book) {
        for (String word : book) {
            map.put(word, map.getOrDefault(word, 0) + 1);
        }
    }

    public int get(String word) {
        return map.getOrDefault(word, 0);
    }
}
/**
 * Your WordsFrequency object will be instantiated and called as such:
 * WordsFrequency obj = new WordsFrequency(book);
 * int param_1 = obj.get(word);
 */
```

- 最接近原点的 K 个点（973）

更多硬核内容就在公众号「yes 的练级攻略」

973. 最接近原点的 K 个点

难度 中等 102 收藏 分享 切换为英文 关注 反馈

我们有一个由平面上的点组成的列表 points。需要从中找出 K 个距离原点 (0, 0) 最近的点。

（这里，平面上两点之间的距离是欧几里德距离。）

你可以按任何顺序返回答案。除了点坐标的顺序之外，答案确保是唯一的。

示例 1：

```
输入：points = [[1,3],[-2,2]], K = 1
输出：[[-2,2]]
解释：
(1, 3) 和原点之间的距离为 sqrt(10)，
(-2, 2) 和原点之间的距离为 sqrt(8)，
由于 sqrt(8) < sqrt(10)，(-2, 2) 离原点更近。
我们只需要距离原点最近的 K = 1 个点，所以答案就是 [[-2,2]]。
```

示例 2：

```
输入：points = [[3,3],[5,-1],[-2,4]], K = 2
输出：[[3,3],[-2,4]]
（答案 [[-2,4],[3,3]] 也会被接受。）
```

提示：

1. 1 <= K <= points.length <= 10000
2. -10000 < points[i][0] < 10000
3. -10000 < points[i][1] < 10000

YES的练级攻略

https://leetcode-cn.com/problems/k-closest-points-to-origin/solution/java-top-k-3-by-donoghl/

计算哪个点离原点最近，利用我们众所周知的距离公式来计算，但是我们大可不必用外面的那一层根号，因为仅仅内部平方和就可以判断了。但是整体思路与前两个题几乎一毛一样。

思路：

1. 同样适用maxHeap，因为我们要找的是smallest one，我们需要不断判断堆顶的元素与新元素的大小关系。
2. 先加入k个元素（计算后的）到maxheap中。
3. 进行逻辑判断，堆顶和新元素，实施必要的替换，保证heap的size。
4. 简单循环，取出heap中的值到array中，输出。

time: **O(n) = N * log(k)**。
space: **O(k)**。

方法 1：堆排

```
class Solution {
    public int[][] kClosest(int[][] points, int K) {
        PriorityQueue<int[]> maxHeap = new PriorityQueue<>((a,b) -> (cal(b) - cal(a)));
```

```
        int[][] result = new int[K][2];
        for(int i = 0; i < K; i++){
            maxHeap.offer(points[i]);
        }
        for(int i = K; i < points.length; i++){
            if(cal(maxHeap.peek()) > cal(points[i])){
                maxHeap.poll();
                maxHeap.offer(points[i]);
            }
        }
        for(int i = 0; i < K; i++){
            result[i] = maxHeap.poll();
        }
        return result;
    }

    public int cal(int a[]){
        return (a[0] * a[0]) + (a[1] * a[1]);
    }
}
```

https://leetcode-cn.com/problems/k-closest-points-to-origin/solution/tou-lan-yu-kuai-pai-by-jalan/

解二：快速排序

这里的快速排序无需使整个数组有序，只需要筛选出最小的 K 个值即可。

假设第一次排序后，哨兵值 `pivot` 将原数组分为两个部分：

1. 左侧部分，元素值均小于 `pivot`，假设下标范围是 `[begin, i - 1]`，长度为 `left_length`
2. 右侧部分，元素值均大于或等于 `pivot`，假设下标范围是 `[i + 1, end]`，长度为 `right_length`

此时：

- 如果 `left_length >= K`，最小的 `K` 个值均在左侧，因此下轮递归只需对左侧部分进行排序
- 如果 `left_length < K`，我们已经获得了 `left_length` 个最小值，因此下轮递归只需对右侧部分进行排序

python | java | php

复杂度分析

- 时间复杂度：$O(N)$，这是在*平均情况下*的时间复杂度，其中 N 是给定点的数量。
- 空间复杂度：$O(N)$。

方法 2：快排

```
class Solution {
     public int[][] kClosest(int[][] points, int K) {
        int start = 0;
        int end = points.length - 1;
        while (start < end) {
```

```
            int index = patition(points, start, end);
            if (index == K) {
                break;
            } else if (index < K) {
                start = index + 1;
            } else {
                end = index - 1;
            }
        }

        return Arrays.copyOf(points, K);
    }

    private int patition(int[][] points, int start, int end) {
        int i = start;
        int j = end + 1;
        int mid = distance(points[i][0], points[i][1]);
        while (true) {
            while (distance(points[++i][0], points[i][1]) < mid && i <
end);
            while (distance(points[--j][0], points[j][1]) > mid && j >
start);
            if (i >= j) {
                break;
            }
            swap(points, i, j);
        }
        swap(points, start, j);
        return j;
    }
    private int distance(int a, int b) {
        return a * a + b * b;
    }
    private void swap(int[][] points, int a, int b) {
        int[] temp = points[a];
        points[a] = points[b];
        points[b] = temp;
    }
}
```

- 连接棒材的最低费用（1167）

更多硬核内容就在公众号「yes 的练级攻略」

1167. 连接棒材的最低费用

难度 中等 17 收藏 分享 切换为英文 关注 反馈

为了装修新房，你需要加工一些长度为正整数的棒材 `sticks`。

如果要将长度分别为 `X` 和 `Y` 的两根棒材连接在一起，你需要支付 `X + Y` 的费用。 由于施工需要，你必须将所有棒材连接成一根。

返回你把所有棒材 `sticks` 连成一根所需要的最低费用。注意你可以任意选择棒材连接的顺序。

示例 1：

```
输入：sticks = [2,4,3]
输出：14
解释：先将 2 和 3 连接成 5，花费 5；再将 5 和 4 连接成 9；总花费为 14。
```

示例 2：

```
输入：sticks = [1,8,3,5]
输出：30
```

提示：

- `1 <= sticks.length <= 10^4`
- `1 <= sticks[i] <= 10^4`

https://leetcode-cn.com/problems/minimum-cost-to-connect-sticks/solution/java-top-k-4-by-donoghl/

连接各个棒子，要求我们每次连接的都是cost最小的。
并不是说依次连接就ok，我们要全部连接起来并且保证cost，例如1 2 连接后为3， 但是下几段为 4 5 6 那么我们必然是要用新得出的3来进行连接4，而不是直接连接4 5。简而言之，我们每次连接后的cost也要作为接下来连接的考虑。

思路：

1. 因为是每次都要用最小的来进行连接，所以我们用minheap，保证堆顶就是最小的。
2. 先将我们的stick全部放进heap中。
3. 将最小的两个值弹出做计算，然后加入到结果中，并且一定要记得把这个和还要加回到heap中，作为下面连接的元素。
4. 最后返回即可。

time: 因为每次我们都要弹出两个，插入一个，所以这就意味着我们有n段的话，就会有n步，每一步的耗费为log，所以最终为**O(n) = n * log(n)**。
space: 因为要存n段，所以所使用的额外空间就是**O(n) = n**

```
class Solution {
    public int connectSticks(int[] sticks) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>((a,b) -> a
 - b);
        int result = 0;
        for(int i : sticks){
            minHeap.offer(i);
        }
```

```
        while(minHeap.size() > 1){
            int temp = minHeap.poll() + minHeap.poll();
            result += temp;
            minHeap.offer(temp);
        }
        return result;
    }
}
```

(9) 归并排序：

时间复杂度 o(nlogn)，空间复杂度 o(n)，稳定。

思路：分为左右两路进行递归排序，合并时都是从未比较的两个子序列的最小值中选出一个更小值。如果未被检测完，直接加到合并序列中。

```
public class MergeSort {
    public static void mergeSort(int[] arr, int start, int end) {
        if(arr == null || arr.length == 1) {
              return;
        }
        if (start < end) {
            int mid = (start + end) / 2;
            mergeSort (arr, start, mid); //对左侧子序列进行递归排序
            mergeSort (arr, mid+1, end); //对右侧子序列进行递归排序
            merge (arr, start, mid, end); //合并
        }
    }

    public static void  merge (int[] arr, int left, int mid, int right) {
        int[] tmp = new int[arr.length]; //辅助数组
        int p1 = left, p2 = mid + 1, k = left; //p1、p2 是检测指针，k 是存放指针
        while (p1 <= mid && p2 <= right) {
            if (arr[p1] <= arr[p2]) {
                tmp[k++] = arr[p1++];
            } else {
                tmp[k++] = arr[p2++];
            }
        }
        //如果第一个序列未检测完，直接将后面所有元素加到合并的序列中
        while (p1 <= mid) {
            tmp[k++] = arr[p1++];
        }
        //如果第二个序列未检测完，直接将后面所有元素加到合并的序列中
        while (p2 <= right) {
            tmp[k++] = arr[p2++];
        }
```

禁止商用 公众号：yes的练级攻略

```
        //复制回原素组
        for (int i = left; i <=right; i++)
            arr[i]=tmp[i];
    }
}
```

- 计算右侧小于当前元素的个数（315）

给定一个整数数组 *nums*，按要求返回一个新数组 *counts*。数组 *counts* 有该性质： `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例:

```
输入: [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1).
2 的右侧仅有 1 个更小的元素 (1).
6 的右侧有 1 个更小的元素 (1).
1 的右侧有 0 个更小的元素.
```

https://leetcode-cn.com/problems/count-of-smaller-numbers-after-self/solution/gui-bing-pai-xu-suo-yin-shu-zu-python-dai-ma-java-/

归并排序计算逆序数 + 索引数组

使用「归并排序」求解这道问题，需要有求解「逆序对」的经验。可以先做一下「力扣」剑指 Offer 51. 数组中的逆序对;

- 求解「逆序对」的思想：当其中一个数字放进最终归并以后的有序数组中的时候，这个数字与之前看过的数字个数（或者是未看过的数字个数）可以直接统计出来，而不必一个一个数。这样排序完成以后，原数组的逆序数也就计算出来了;
- 具体来说，本题让我们求「在一个数组的某个元素的右边，比自己小的元素的个数」，因此，需要在「前有序数组」的元素归并的时候，数一数「后有序数组」已经归并回去的元素的个数，因为这些已经出列的元素都比当前出列的元素要（严格）小;
- 但是在「归并」的过程中，元素的位置会发生变化，因此下一步需要思考如何定位元素；根据「索引堆」的学习经验，一个元素在算法的执行过程中位置发生变化，我们还想定位它，可以使用「索引数组」，技巧在于：**「原始数组」不变，用于比较两个元素的大小，真正位置变化的是「索引数组」的位置**;
- 「索引数组」技巧建立了一个一一对应的关系，**记录了当前操作的数对应的「原始数组」的下标**，「索引数组」技巧想法的来源是「索引堆」（《算法（第 4 版）》第 2.4 节 练习）；
- 「归并排序」还需要一个用于归并的辅助数组，这个时候拷贝的就是索引数组的值了。

复杂度分析:

- 时间复杂度：$O(N \log N)$，数组的元素个数是 N，递归执行分治法，时间复杂度是对数级别的，因此时间复杂度是 $O(N \log N)$。
- 空间复杂度：$O(N)$，需要 3 个数组，一个索引数组，一个临时数组用于索引数组的归并，还有一个结果数组，它们的长度都是 N，故空间复杂度是 $O(N)$。

```
import java.util.ArrayList;
import java.util.List;
public class Solution {
    private int[] temp;
    private int[] counter;
    private int[] indexes;
    public List<Integer> countSmaller(int[] nums) {
        List<Integer> res = new ArrayList<>();
        int len = nums.length;
        if (len == 0) {
```

```
            return res;
        }
        temp = new int[len];
        counter = new int[len];
        indexes = new int[len];
        for (int i = 0; i < len; i++) {
            indexes[i] = i;
        }
        mergeAndCountSmaller(nums, 0, len - 1);
        for (int i = 0; i < len; i++) {
            res.add(counter[i]);
        }
        return res;
    }

    /**
     * 针对数组 nums 指定的区间 [l, r] 进行归并排序，在排序的过程中完成统计任
务
     *
     * @param nums
     * @param l
     * @param r
     */
    private void mergeAndCountSmaller(int[] nums, int l, int r) {
        if (l == r) {
            // 数组只有一个元素的时候，没有比较，不统计
            return;
        }
        int mid = l + (r - l) / 2;
        mergeAndCountSmaller(nums, l, mid);
        mergeAndCountSmaller(nums, mid + 1, r);
        // 归并排序的优化，同样适用于该问题
        // 如果索引数组有序，就没有必要再继续计算了
        if (nums[indexes[mid]] > nums[indexes[mid + 1]]) {
            mergeOfTwoSortedArrAndCountSmaller(nums, l, mid, r);
        }
    }

    /**
     * [l, mid] 是排好序的
     * [mid + 1, r] 是排好序的
     *
     * @param nums
     * @param l
```

```
     * @param mid
     * @param r
     */
    private void mergeOfTwoSortedArrAndCountSmaller(int[] nums, int l,
int mid, int r) {
        // 3,4  1,2
        for (int i = l; i <= r; i++) {
            temp[i] = indexes[i];
        }
        int i = l;
        int j = mid + 1;
        // 左边出列的时候，计数
        for (int k = l; k <= r; k++) {
            if (i > mid) {
                indexes[k] = temp[j];
                j++;
            } else if (j > r) {
                indexes[k] = temp[i];
                i++;
                // 此时 j 用完了，[7,8,9 | 1,2,3]
                // 之前的数就和后面的区间长度构成逆序
                counter[indexes[k]] += (r - mid);
            } else if (nums[temp[i]] <= nums[temp[j]]) {
                indexes[k] = temp[i];
                i++;
                // 此时 [4,5, 6   | 1,2,3 10 12 13]
                //           mid          j
                counter[indexes[k]] += (j - mid - 1);
            } else {
                // nums[indexes[i]] > nums[indexes[j]] 构成逆序
                indexes[k] = temp[j];
                j++;
            }
        }
    }

    public static void main(String[] args) {
        int[] nums = new int[]{5, 2, 6, 1};
        Solution solution = new Solution();
        List<Integer> countSmaller = solution.countSmaller(nums);
        System.out.println(countSmaller);
    }
}
```

- 数组中的逆序对（剑指 51）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 51. 数组中的逆序对

难度 困难 216 收藏 分享 切换为英文 关注 反馈

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

示例 1:

```
输入: [7,5,6,4]
输出: 5
```

限制:

`0 <= 数组长度 <= 50000`

https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/bao-li-jie-fa-fen-zhi-si-xiang-shu-zhuang-shu-zu-b/

方法二：分治思想（借助归并排序统计逆序数）

说明：理解这个算法需要对「归并排序」比较熟悉。掌握如果编写递归函数，每一次都一分为二拆分数组的子区间，然后在方法栈弹出的时候，一步一步合并两个有序数组，最后完成排序工作。

而计算逆序数就发生在排序的过程中，利用了「排序」以后数组的有序性。

- 利用「归并排序」计算逆序对，是非常经典的做法；
- **关键在于「合并两个有序数组」的步骤，利用数组的部分有序性**，一下子计算出一个数之前或者之后元素的逆序的个数；
- **前面「分」的时候什么都不做，「合」的过程中计算「逆序对」的个数**；
- 「排序」的工作是必要的，正是因为「排序」才能在下一轮利用顺序关系加快逆序数的计算，也能避免重复计算；
- 在代码实现上，只需要在「归并排序」代码的基础上，加上「逆序对」个数的计算，计算公式需要自己在草稿纸上推导。

思想是「分治算法」，所有的「逆序对」来源于 3 个部分：

- 左边区间的逆序对；
- 右边区间的逆序对；
- 横跨两个区间的逆序对。

下面提供两种写法：

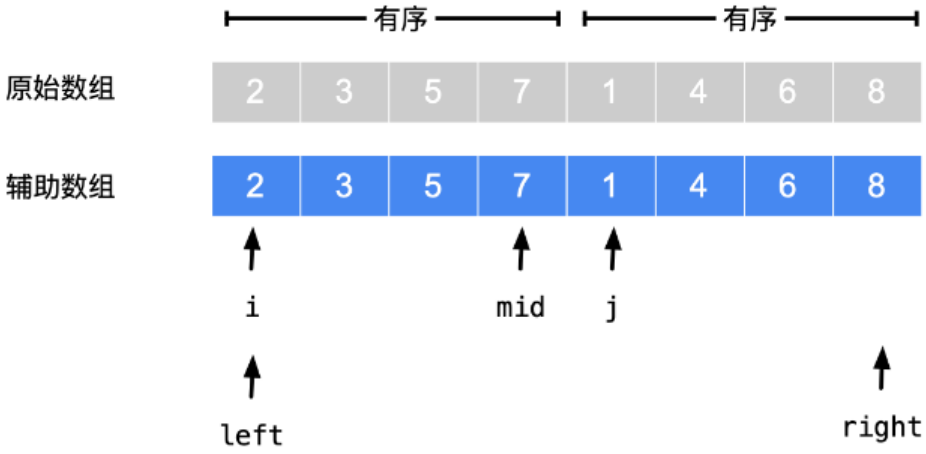
1、在第 2 个子区间元素归并回去的时候，计算逆序对的个数（参考代码 2）；
2、在第 1 个子区间元素归并回去的时候，计算逆序对的个数（参考代码 3）。

注意：两者不能同时计算，否则会计算重复。

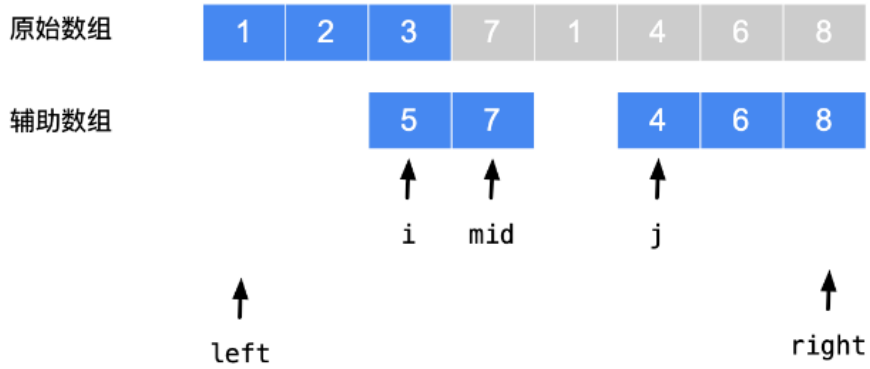
更多硬核内容就在公众号「yes 的练级攻略」

参考代码 2：在第 2 个子区间元素归并回去的时候，计算逆序对的个数。

任务：将 [2, 3, 5, 7, 1, 4, 6, 8] 合并成为一个更长的有序数组
思路：将原始数组拷贝到辅助数组，再使用双指针，每次较小元素归并回去。



在指针 j 指向的元素归并回去的同时，计算逆序数。
逆序数的个数就等于前面还没有归并回去的元素的个数：(mid - i + 1)。



这一步 4 归并回去，4 与之前的 5 和 7 构成逆序对，个数为 (mid - i + 1)。

即在 `j` 指向的元素赋值回去的时候，给计数器加上 `mid - i + 1`。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析：

- 时间复杂度：$O(N \log N)$，这里 N 是数组的长度。复杂度是归并排序的时间复杂度，直接看递归树的结点个数或者使用主定理分析，归并的回收每一步计算逆序对的个数是 $O(1)$ 的；
- 空间复杂度：$O(N)$。

主定理分析：数据列均分为两部分，分别排序，之后以 $O(N)$ 的复杂度进行合并。根据主定理（递归函数的时间复杂度分析工具）：

$$T(N) = 2T(\frac{N}{2}) + O(N)$$

这里 $a = 2$，$b = 2$，$N^{\log_b a} = N^{\log_2 2} = N$，于是得到 $T(N) = O(N \log N)$。

至于为什么是这样，不在我能解释的范围内，大家可以在互联网上搜索「主定理」、「归并排序」获得相关的知识。

```java
public class Solution {
    public int reversePairs(int[] nums) {
        int len = nums.length;

        if (len < 2) {
            return 0;
        }

        int[] copy = new int[len];
        for (int i = 0; i < len; i++) {
            copy[i] = nums[i];
        }

        int[] temp = new int[len];
        return reversePairs(copy, 0, len - 1, temp);
    }
    private int reversePairs(int[] nums, int left, int right, int[] temp) {
        if (left == right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        int leftPairs = reversePairs(nums, left, mid, temp);
        int rightPairs = reversePairs(nums, mid + 1, right, temp);

        if (nums[mid] <= nums[mid + 1]) {
            return leftPairs + rightPairs;
        }

        int crossPairs = mergeAndCount(nums, left, mid, right, temp);
        return leftPairs + rightPairs + crossPairs;
    }
```

```
    private int mergeAndCount(int[] nums, int left, int mid, int right, int[] t
emp) {
        for (int i = left; i <= right; i++) {
            temp[i] = nums[i];
        }

        int i = left;
        int j = mid + 1;
        int count = 0;
        for (int k = left; k <= right; k++) {

            if (i == mid + 1) {
                nums[k] = temp[j];
                j++;
            } else if (j == right + 1) {
                nums[k] = temp[i];
                i++;
                count += (right - mid);
            } else if (temp[i] <= temp[j]) {
                nums[k] = temp[i];
                i++;
                count += (j - mid - 1);
            } else {
                nums[k] = temp[j];
                j++;
            }
        }
        return count;
    }
}
```

- 逆序对（腾讯 2020 校园招聘-后台第 3 题）

作为程序员的小Q，他的数列和其他人的不太一样，他有2^n个数。

老板问了小Q一共 m次，每次给出一个整数$q_i(1 <= i <= m)$，要求小Q把这些数每2^{q_i}分为一组，然后把每组进行翻转，小Q想知道每次操作后整个序列中的逆序对个数是多少呢？

例如:
对于序列1 3 4 2，逆序对有(4, 2),(3, 2),总数量为2。
翻转之后为2 4 3 1，逆序对有(2, 1),(4, 3), (4, 1), (3, 1),总数量为4。

输入描述:

> 第一行一个数$n(0 \leq n \leq 20)$
> 第二行2^n个数，表示初始的序列$(1 \leq 初始序列 \leq 10^9)$
> 第三行一个数$m(1 \leq m \leq 10^6)$
> 第四行m个数表示$q_i(0 \leq q_i \leq n)$

输出描述:

> m行 每行一个数表示答案。

禁止商用
公众号：
yes的练级攻略

输入例子1:

```
2
2 1 4 3
4
1 2 0 2
```

输出例子1:

```
0
6
6
0
```

例子说明1:

初始序列2 1 4 3

$2^{q_1} = 2$ ->

第一次: 1 2 3 4 -> 逆序对数为0

$2^{q_2} = 4$ ->

第二次: 4 3 2 1 -> 逆序对数为6

$2^{q_3} = 1$ ->

第三次: 4 3 2 1 -> 逆序对数为6

$2^{q_4} = 4$ ->

第四次: 1 2 3 4 -> 逆序对数为0

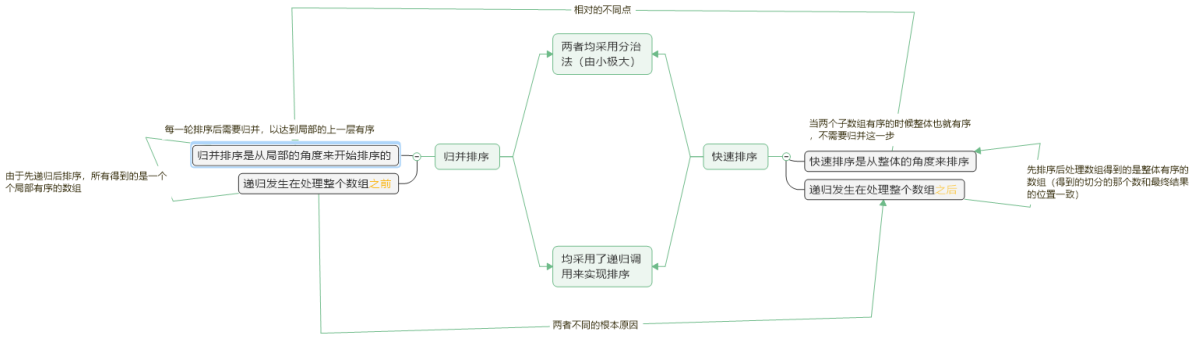
```
import java.util.Scanner;
//就是首先一次归并求逆序对数，dp 分别存的是 2 的 1 到 n 次方为一组的逆序对的个数和顺序对的个数，
//dp[index][0]是逆序对的个数，dp[index][1]是顺序对的个数，2 的 qi 次方为一组计算逆序对数，
//只需要调整 1 到 qi 次方的逆序对数，也就是 原来的逆序对数+dp[index][1]-dp[index][0]
public class Main {
    public static void resolve(int arr[], int left, int right, int[] temp,
                               long[] result, long[][] dp, int index) {
        if (left < right) {
            int mid = left + right >> 1;
```

```
            resolve(arr, left, mid, temp, result, dp, index - 1);
            resolve(arr, mid + 1, right, temp, result, dp, index - 1);
            merge(arr, left, mid, right, temp, result, dp, index);
        }
    }

    public static void merge(int arr[], int left, int mid, int right, int[] temp,
                             long[] result, long[][] dp, int index) {
        int i = left;
        int j = mid + 1;
        int tempLeft = 0;

        long max = 0;
        long equal = 0;
        while (i <= mid && j <= right)
            if (arr[i] <= arr[j]) {
                max += j - mid - 1;
                if (arr[i] == arr[j]) {
                    equal++;
                    int ind = j + 1;
                    while (ind <= right && arr[j] == arr[ind++])
                        equal++;
                }
                temp[tempLeft++] = arr[i++];
            } else {
                temp[tempLeft++] = arr[j++];
            }
        while (i <= mid) {
            max += right - mid;
            temp[tempLeft++] = arr[i++];
        }
        while (j <= right) {
            temp[tempLeft++] = arr[j++];
        }
        result[0] += max;
        dp[index][0] += max;
        dp[index][1] += (long)(right - mid) * (right - mid) - max - equal;

        tempLeft = left;
        i = 0;
        while (tempLeft <= right)
            arr[tempLeft++] = temp[i++];
    }
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();

        int total = n == 0 ? 1 : 2 << n - 1;
        int[] arr = new int[total];
        for (int i = 0; i < total; i++) {
            arr[i] = scanner.nextInt();
        }

        int m = scanner.nextInt();

        long[] result = {0};
        long[][] dp = new long[n][2];

        int[] temp = new int[total];
        resolve(arr, 0, total - 1, temp, result, dp, n - 1);

        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < m; i++) {
            int qn = scanner.nextInt() - 1;
            if (i == 0 || qn != -1) {
                int index = 0;
                while (index <= qn) {
                    long tem = dp[index][0];
                    dp[index][0] = dp[index][1];
                    dp[index][1] = tem;
                    result[0] += dp[index][0] - dp[index][1];
                    index++;
                }
            }
            builder.append(result[0] + "\n");
        }
        System.out.print(builder);
    }
}
```

- 快速排序和归并排序的区别与联系

更多硬核内容就在公众号「yes 的练级攻略」



联系：均采用了分治法和递归调用。

区别：

1、根本区别：**归并**排序的递归发生在处理整个数组之**前（先递归后排序）**，**快速**排序的递归发生在处理整个数组之**后（先排序后处理）**。

2、相对区别：**归并**排序是从**局部**角度开始排序的（每一轮排序后都需要归并，以达到局部的上一层有序），**快速**排序是从**整体**角度来排序的（当两个子数组有序时整体也有序，不需要归并这一步）。

3、**归并**排序是一种**稳定**的排序，相等元素的位置不会变化，**快速**排序**不稳定**。

4、定义：

快速排序：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

归并排序：建立在归并操作上的一种有效的排序算法,该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

(10) 桶排序（基数排序）：

属于分配式排序，又称桶排序，时间复杂度 O(d(n+r))，其中 d 是关键码数目，r 是关键码取值范围，n 是记录。空间复杂度 o(r)，稳定。

思路：基数排序采用最低优先的方式，将所有待排序数列元素统一为同样的数位长度，数列元素数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

① 找到数列元素的个位数，然后以个位数为桶的索引，将数列元素存放在桶中。

② 遍历桶，将桶的数据元素取出依次存放在原数组中。

③ 找到数列元素的高位数，重复 1,2 操作，直到找操作完最高位数，这时数列就有序了。

注意测试时，生成的随机数组必须是二维的！否则无法跑。

更多硬核内容就在公众号「yes 的练级攻略」

桶排序思想

桶排序，是一种基于非比较的排序方式，时间复杂度O(n)，因此它是是属于一种“线性排序”。

思想：桶排序的思想是将一组数据分到几个有序的桶里，每个桶里的数据再单独进行快速排序。每个桶内都排序完成后，再加上本身桶之间已经是有序的，那么按照桶的顺序依次取出每个桶内的数据，最终组成的序列就是有序的。

桶排序的时间复杂度分析：

假如排序数组的元素个数为n，均匀分布在个数为m的桶中，那么每个桶中的元素个数为k=n/m,因为每个桶快速排序的时间复杂度为klogk即n/mlog(n/m),那么m个桶一起就是nlog(n/m),假如桶的个数m跟元素个数n十分接近，那么最终的时间复杂度为O(n).

如果极端情况下，所有的元素n个都分在一个桶里呢？这种情况下时间复杂度就退化成O(nlogn)了。

所以，可以看出桶排序对数据及其分布的要求是十分苛刻的：

```
public class RadixSort {
    //arr 是要排序的数组，max 是数组中最大的数有几位
    public static void radixSort(int[] ary)
    {
        int aryLen=ary.length;
        int len=aryLen;
        if(aryLen<=10) {
            len=10;
        }
        int bucket [][] =new int [10][len];//用于存放每次排序存放的结果。
        int bucketCap []=new int[len];//每个桶子存放的数据量。bucketCap[i] 表示 i 个桶有多少个数据量，所以 len 必须大于等于 10
        int loop=1;//排序的轮数，第一次按个位数上的数字排序，第二次按照十位数上的数字排序
        int maxLoop=getMaxLoop(ary);//排序需要的最大轮数，即是数组元素中最大元素是几位数。
        int k=0;//每轮排序好的数据存放的索引位置
        while(loop<=maxLoop) {
            for (int i = 0; i < aryLen; i++) {//将数组元素存放在相应的桶里面
                int position=(int) (ary[i]/(Math.pow(10, loop-1))%10);//取出要排的数据，第一轮取出个位数，第二轮取出十位数...,，也是数组元素存放在哪个桶子中，桶子的索引值。
                bucket[position][bucketCap[position]]=ary[i];
                bucketCap[position]++;
            }

            for (int i = 0; i < 10; i++) {//遍历所有的桶子，取出桶子的值，存放在原数组完成一轮排序
                if(bucketCap[i]>0) {
                    for (int j = 0; j < bucketCap[i]; j++) {
                        ary[k]=bucket[i][j];
                        k++;
                    }
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                    bucketCap[i]=0;//将桶子清空
                }
            }
            loop++;//轮数加+
            k=0;
        }
    }
    public static int getMaxLoop(int[] ary) {
        int max=ary[0];
        for (int i = 1; i < ary.length; i++) {
            if(max<ary[i]) {
                max=ary[i];
            }
        }
        int temp=max;
        int n=1;
        while((temp=temp/10)>0) {
            n++;
        }

        return n;
    }
}
```

- 任务调度器（621）

621. 任务调度器

难度 中等 321 收藏 分享 切换为英文 关注 反馈

给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A - Z 字母表示的26 种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。

然而，两个**相同种类**的任务之间必须有长度为 **n** 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的**最短时间**。

示例：

输入： tasks = ["A","A","A","B","B","B"], n = 2
输出： 8
解释： A -> B -> (待命) -> A -> B -> (待命) -> A -> B.
在本示例中，两个相同类型任务之间必须间隔长度为 n = 2 的冷却时间，而执行一个任务只需要一个单位时间，所以中间出现了（待命）状态。

提示：

1. 任务的总个数为 [1, 10000]。
2. n 的取值范围为 [0, 100]。

https://leetcode-cn.com/problems/task-scheduler/solution/621-ren-wu-diao-du-qi-java-jie-ti-zhu-shi-ying-gai/

```
class Solution {
    /**
     * 解题思路:
     * 1、将任务按类型分组，正好A-Z 用一个int[26]保存任务类型个数
```

```
     * 2、对数组进行排序，优先排列个数（count）最大的任务，
     *      如题得到的时间至少为 retCount =（count-1）
* (n+1) + 1 ==> A->X->X->A->X->X->A(X 为其他任务或者待命)
     * 3、再排序下一个任务，如果下一个任务 B 个数和最大任务数一致，
     *      则 retCount++ ==> A->B->X->A->B->X->A->B
     * 4、如果空位都插满之后还有任务，那就随便在这些间隔里面插入就可以，因为间隔长度肯定会大于 n，在这种情况下就是任务的总数是最小所需时间
     *
     * @param tasks
     * @param n
     * @return
     */
    public int leastInterval(char[] tasks, int n) {
        if (tasks.length <= 1 || n < 1) return tasks.length;
        //步骤 1
        int[] counts = new int[26];
        for (int i = 0; i < tasks.length; i++) {
            counts[tasks[i] - 'A']++;
        }
        //步骤 2
        Arrays.sort(counts);
        int maxCount = counts[25];
        int retCount = (maxCount - 1) * (n + 1) + 1;
        int i = 24;
        //步骤 3
        while (i >= 0 && counts[i] == maxCount) {
            retCount++;
            i--;
        }
        //步骤 4
        return Math.max(retCount, tasks.length);
    }
}
```

(11) 计数排序

计数排序是一个非基于比较的排序算法，该算法于 1954 年由 Harold H. Seward 提出。它的优势在于在对一定范围内的整数排序时，它的复杂度为 O(n+k)（其中 k 是整数的范围），快于任何比较排序算法。[1] 当然这是一种牺牲空间换取时间的做法，而且当 O(k)>O(n*log(n)) 的时候其效率反而不如基于比较的排序（基于比较的排序的时间复杂度在理论上的下限是 O(n*log(n)), 如归并排序，堆排序），是稳定的。

```
//针对 c 数组的大小，优化过的计数排序
public class Main {
    public static void main(String[] args){
        int[] a = {25,84,21,47,15,27,68,35,20};
        int[] b = countSort(a);
```

```
        for(int i : b){
            System.out.print(i +" ");
        }
        System.out.println();
    }
    public static int[] countSort(int[] a){
        int b[] = new int[a.length];
        int max = a[0],min = a[0];
        for(int i:a){
            if(i>max){
                max=i;
            }
            if(i<min){
                min=i;
            }
        }
        //这里 k 的大小是要排序的数组中，元素大小的极值差+1
        int k=max-min+1;
        int c[]=new int[k];
        for(int i=0;i<a.length;++i){
            c[a[i]-min]+=1;//优化过的地方，减小了数组 c 的大小
        }
        for(int i=1;i<c.length;++i){
            c[i]=c[i]+c[i-1];
        }
        for(int i=a.length-1;i>=0;--i){
            b[--c[a[i]-min]]=a[i];//按存取的方式取出 c 的元素
        }
        return b;
    }
}
```

- 排序测试代码

```
import java.util.Arrays;
public class Test {
    public static void main(String[] args) {
        int testTime=10000;
        int size = 10;
        int value=100;
        //堆排序测试
        boolean succeed_heap = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
            int[] arr2 = copyArray(arr1);
            HeapSort.heapSort(arr1);
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
            rightMethod(arr2);
            if(!isEqual(arr1,arr2)){
                succeed_heap=false;
                break;
            }
        }
        System.out.println(succeed_heap?"HeapSort succeeds!":"HeapSort
fails!");
        //原始数据
        int[] arr_heap = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_heap));
        //堆排序，从小到大
        HeapSort.heapSort(arr_heap);
        System.out.println("堆排序后：  "+Arrays.toString(arr_heap));
        //逆堆排序，从大到小
        HeapSort.oppositeHeapSort(arr_heap);
        System.out.println("逆堆排序后：  "+Arrays.toString(arr_heap));
        //冒泡排序测试
        boolean succeed_bubble = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
            int[] arr2 = copyArray(arr1);
            BubbleSort.bubbleSort(arr1);
            rightMethod(arr2);
            if(!isEqual(arr1,arr2)){
                succeed_bubble=false;
                break;
            }
        }
        System.out.println(succeed_bubble?"BubbleSort
succeeds!":"BubbleSort fails!");
        //原始数据
        int[] arr_bubble = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_bubble));
        //冒泡排序，从小到大
        BubbleSort.bubbleSort(arr_bubble);
        System.out.println("冒泡排序后：  "+Arrays.toString(arr_bubble));
        //逆冒泡排序，从大到小
        BubbleSort.oppositeBubbleSort(arr_bubble);
        System.out.println("逆泡排序后：  "+Arrays.toString(arr_bubble));
        //快速排序测试
        boolean succeed_quick = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
```

禁止商用 公众号：yes的练级攻略

```
            int[] arr2 = copyArray(arr1);
            QuickSort.quickSort(arr1,0,arr1.length-1);
            rightMethod(arr2);
            if(!isEqual(arr1,arr2)){
                succeed_quick=false;
                break;
            }
        }
        System.out.println(succeed_quick?"QuickSort succeeds!":"QuickSort
fails!");
        //原始数据
        int[] arr_quick = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_quick));
        //快速排序，从小到大
        QuickSort.quickSort(arr_quick,0,arr_quick.length-1);
        System.out.println("快速排序后：  "+Arrays.toString(arr_quick));

        //插入排序测试
        boolean succeed_insert = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
            int[] arr2 = copyArray(arr1);
            InsertSort.insertSort(arr1);
            rightMethod(arr2);
            if(!isEqual(arr1,arr2)){
                succeed_insert=false;
                break;
            }
        }
        System.out.println(succeed_insert?"InsertSort
succeeds!":"InsertSort fails!");
        //原始数据
        int[] arr_insert = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_insert));
        //插入排序，从小到大
        InsertSort.insertSort(arr_insert);
        System.out.println("插入排序后：  "+Arrays.toString(arr_insert));
      //希尔排序测试
        boolean succeed_shell = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
            int[] arr2 = copyArray(arr1);
            ShellSort.shellSort(arr1);
            rightMethod(arr2);
```

禁止商用 公众号：yes的练级攻略

```
            if(!isEqual(arr1,arr2)){
                succeed_shell=false;
                break;
            }
        }
        System.out.println(succeed_shell?"ShellSort succeeds!":"ShellSort
fails!");
        //原始数据
        int[] arr_shell = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_shell));
        //希尔排序，从小到大
        ShellSort.shellSort(arr_shell);
        System.out.println("希尔排序后：  "+Arrays.toString(arr_shell));

      //选择排序测试
        boolean succeed_select = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
            int[] arr2 = copyArray(arr1);
            SelectSort.selectSort(arr1);
            rightMethod(arr2);
            if(!isEqual(arr1,arr2)){
                succeed_select=false;
                break;
            }
        }
        System.out.println(succeed_select?"SelectSort
succeeds!":"SelectSort fails!");
        //原始数据
        int[] arr_select = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_select));
        //选择排序，从小到大
        SelectSort.selectSort(arr_select);
        System.out.println("选择排序后：  "+Arrays.toString(arr_select));

      //归并排序测试
        boolean succeed_merge = true;
        for(int i = 0 ;i<testTime;i++){
            int[] arr1 = generateRandomArray(size,value);
            int[] arr2 = copyArray(arr1);
            MergeSort. mergeSort(arr1,0,arr1.length-1);
            rightMethod(arr2);
            if(!isEqual(arr1,arr2)){
                succeed_merge=false;
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                break;
            }
        }
        System.out.println(succeed_merge?"MergeSort succeeds!":"MergeSort
fails!");
        //原始数据
        int[] arr_merge = generateRandomArray(size,value);
        System.out.println("排序前：  "+Arrays.toString(arr_merge));
        //归并排序，从小到大
        MergeSort. mergeSort(arr_merge,0,arr_merge.length-1);
        System.out.println("归并排序后：  "+Arrays.toString(arr_merge));

      //基数排序测试
        //原始数据
        int[] arr_radix = new int
[]{6,5,7,3,1,12,33,53,25,89,108,149,44,87,36};
        System.out.println("排序前：  "+Arrays.toString(arr_radix));
        //基数排序，从小到大
        RadixSort.radixSort(arr_radix);
        System.out.println("基数排序后：  "+Arrays.toString(arr_radix));
    }
    //产生一个随机数组，数组的长度和值都是随机的，
    public static  int[] generateRandomArray(int size,int value){
        //在 java 中，Math.random() ->double(0,1)
        //(int)((size+1)*Math.random())--->产生的是[0,size]之间的整数
        //生成长度随机的数组，数组的最大长度是 size 的长度
        int[] arr = new int[(int)((size+1)*Math.random())];
        for(int i = 0 ;i<arr.length;i++){
            //针对数组中的每个值都可以随机一下，一个随机数减去另外一个随机数，
可能产生正数，也可能产生负数

arr[i]=(int)((value+1)*Math.random())-(int)(value*Math.random());//值也
可以是随机的
            //只产生正数的代码如下
            //arr[i]=Math.abs
((int)((value+1)*Math.random())-(int)(value*Math.random()));
        }
        return arr;
    }
    //复制数组
    public static int[] copyArray(int[] arr){
        if(arr==null){
            return null;
        }
```

禁止商用 公众号：yes的练级攻略

```
        int[] res = new int[arr.length];
        for(int i = 0 ;i<arr.length;i++){
            res[i]=arr[i]  ;
        }
        return res;
    }
    //绝对正确的方法,这个方法可以时间复杂度很差，但是要保证其准确性
    public static void rightMethod(int[] arr){
        Arrays.sort(arr);
    }
    //对比两个数组是否相等
    public static boolean isEqual(int[] arr1,int[] arr2){
        if(arr1==null&&arr2!=null||arr1!=null&&arr2==null){
            return false;
        }
        if (arr1==null&&arr2==null){
            return true;
        }
        if (arr1.length!=arr2.length){
            return false;
        }
        for(int i = 0;i<arr1.length;i++){
            if(arr1[i]!=arr2[i]){
                return false;
            }
        }
        return true;
    }
}
```

十、数组

1、滑动窗口的中位数（linkcode360）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

描述 中文 English

给定一个包含 n 个整数的数组，和一个大小为 *k* 的滑动窗口,从左到右在数组中滑动这个窗口，找到数组中每个窗口内的中位数。
(如果数组个数是偶数，则在该窗口排序数字后，返回第 N/2 个数字。)

您在真实的面试中是否遇到过这个题? 是 题目纠错

样例

样例 1:

```
输入:
[1,2,7,8,5]
3
输出:
[2,7,7]

解释:
最初，窗口的数组是这样的: `[ | 1,2,7 | ,8,5]` , 返回中位数 `2`;
接着，窗口继续向前滑动一次。`[1, | 2,7,8 | ,5]`, 返回中位数 `7`;
接着，窗口继续向前滑动一次。`[1,2, | 7,8,5 | ]`, 返回中位数 `7`;
```

样例 2:

```
输入:
[1,2,3,4,5,6,7]
4
输出:
[2,3,4,5]

解释:
最初，窗口数组是这样的: `[ | 1,2,3,4, | 5,6,7]` , 返回中位数 `2`;
接着，窗口向前滑动一次.`[1,| 2,3,4,5 | 6,7]`,返回中位数 `3`;
接着，窗口向前滑动一次.`[1,2, | 3,4,5,6 | 7 ]`, 返回中位数 `4`;
接着，窗口向前滑动一次`[1,2,3,| 4,5,6,7 ]`, 返回中位数 `5`;
```

挑战

时间复杂度为 O(nlog(n))

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: An integer
     * @return: The median of the element inside the window at each moving
     */
     PriorityQueue<Integer> maxHeap, minHeap;
    public List<Integer> medianSlidingWindow(int[] nums, int k) {
        // write your code here
        List<Integer> res = new ArrayList<>();
        if (nums.length == 0) return res;
        maxHeap = new PriorityQueue<Integer>(nums.length, Collections.reverseOrder());
        minHeap = new PriorityQueue<Integer>(nums.length);
        for (int i = 0; i < nums.length; ++i) {
            if (maxHeap.size() == 0 || nums[i] <= maxHeap.peek()) {
                maxHeap.offer(nums[i]);
            } else {
                minHeap.offer(nums[i]);
            }
            balance();
```

yes的练级攻略

```
            if (i - k >= 0) {
                if (nums[i-k] > maxHeap.peek()) {
                    minHeap.remove(nums[i-k]);
                }else{
                    maxHeap.remove(nums[i-k]);
                }
            }
            balance();
            if (i >= k-1) {
                res.add(maxHeap.peek());
            }
        }
        return res;
    }
    private void balance (){
            while (maxHeap.size() < minHeap.size() ) {
                maxHeap.offer(minHeap.poll());
            }
            while (minHeap.size() < maxHeap.size() - 1) {
                minHeap.offer(maxHeap.poll());
            }
    }
}
```

2、滑动窗口的最大值（239）（lintcode362）（剑指 59-1）【单调队列】

362. 滑动窗口的最大值

中文

给出一个可能包含重复的整数数组，和一个大小为 *k* 的滑动窗口, 从左到右在数组中滑动这个窗口，找到数组中每个窗口内的最大值。

样例

样例 1:

输入:
[1,2,7,7,8]
3
输出:
[7,7,8]

解释:
最开始，窗口的状态如下: `[|1, 2 ,7| ,7 , 8]`, 最大值为 `7`;
然后窗口向右移动一位: `[1, |2, 7, 7|, 8]`, 最大值为 `7`;
最后窗口再向右移动一位: `[1, 2, |7, 7, 8|]`, 最大值为 `8`.

样例 2:

输入:
[1,2,3,1,2,3]
5
输出:
[3,3]

解释:
最开始，窗口的状态如下: `[|1,2,3,1,2 | ,3]` , 最大值为`3`;
然后窗口向右移动一位.`[1, |2,3,1,2,3]`, 最大值为 `3`;

挑战

O(n)时间，O(k)的额外空间

禁止商用 公众号：

更多硬核内容就在公众号「yes 的练级攻略」

算法二 单调队列

- 单调队列中的元素是严格单调的。我们在求解这个问题的时候需要维护他的单调性。
- 队首元素即为当前位置的最大值。假设要求滑动窗口中的最大值。我们就需要确保滑动窗口中的元素从队首到队尾是递减的。
- 每滑动一次就判断当前元素和队尾元素的关系，如果放入队尾满足单调递减，那么放入即可；如果放入不满足，就需要删除队尾元素直到放入当前元素之后满足队列单调递减。同时要确保已经出窗口的最大值（队首元素）被删除掉。

注意事项：

我们在本题中 往单调队列里压入的是元素对应的下标，因为我们要通过下标来判断该元素有没有离开了滑动窗口，如果离开了滑动窗口就得从队列里给删除了

流程

序列a是 1 2 7 7 6 5 3

滑动窗口的大小是3

遍历一遍 for i in 0：7

- i==0 队列里没有元素，直接添加（value=1,index=0） value是权值，index是下标（这里为了表达清晰，就把权值和下标都注明出来了）

队列里 (1,0)

- i==1 a[1]比队尾元素大，添加a[1]后不满足单调递减性质，所以需要把队尾给弹出，a[1]比队尾大，而且队尾元素肯定在a[1]的左面（因为队尾元素先压入的，它的下标肯定小），而且队尾元素还没a[1]大，对于**后面**的窗口，队尾元素肯定不是最大值了（我们已经找到了 a[1]比，而且队尾在窗口里面，a[1]肯定也在窗口里面，因为a[1]的下标更大）
  - 弹掉队尾后，队列为空
  - 压入(2,1)
  - **队列里 (2,1)**
- i==2 a[2]比队尾元素大，弹掉队尾，压入(7,2)

队列里 (7,2)

这里i> =k-1了 ，当前队首为当前[0,2]这个滑动窗口的答案，这个窗口答案是7

- i==3 a[3]不比队尾元素大，不用弹出队尾 ，压入新元素

队列里 (7,2) (7,3)

统计[1,3]滑动窗口的答案，也是7

- i==4 a[4]不比队尾元素大，不用弹出队尾，压入新元素

队列里 (7,2) (7,3) (6，4)

统计[2，4]滑动窗口的答案，也是7

- i==5 a[5]不比队尾元素大，不用弹出队尾，压入新元素

队列里 (7,2) (7,3) (6，4) (5，5)

这时候i和队首位置差已经超过了滑动窗口长度，弹出队首

队列里 (7,3) (6，4) (5，5)

这个窗口答案也为7

复杂度分析

时间复杂度

每个元素进队出队都只有一次，所以复杂度是$O(N)$的

空间复杂度

队列里元素最多为N个,所以空间复杂度也是$O(N)$

Lintcode 版本

```
public class Solution {
    /**
     * @param nums: A list of integers.
     * @param k: An integer
```

更多硬核内容就在公众号「yes 的练级攻略」

```
     * @return: The maximum number inside the window at each moving.
     */
    public List<Integer> maxSlidingWindow(int[] nums, int k) {
        // 答案数组和单调队列
        List<Integer> res = new ArrayList<>();
        Deque<Integer> max = new LinkedList<>();
        for (int i = 0; i < nums.length; i++){
            //维护单调性
            while (! max.isEmpty() && nums[i] > nums[max.getLast()]){
                max.removeLast();
            }
            //滑动窗口的长度超过了 k 删掉超过的部分
            if (! max.isEmpty() && i - max.getFirst() >= k) {
                max.removeFirst();
            }
            max.addLast(i);
            if (i >= k - 1){
                res.add(nums[max.getFirst()]);
            }
        }
        return res;
    }
}
```

Leecode 版本

239. 滑动窗口最大值

难度 困难 472 收藏 分享 切换为英文 关注 反馈

给定一个数组 *nums*，有一个大小为 *k* 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 *k* 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

进阶：

你能在线性时间复杂度内解决此题吗？

示例：

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

提示：

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4
- 1 <= k <= nums.length

禁止商用
公众号：

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        List<Integer> list = new ArrayList<>();
        Deque<Integer> max = new LinkedList<>();
        for (int i = 0; i < nums.length; ++i){
            while (!max.isEmpty() && nums[i] > nums[max.getLast()]){
                max.removeLast();
            }
            if (!max.isEmpty() && i - max.getFirst() >= k) {
                max.removeFirst();
            }
            max.addLast(i);
            if (i >= k -1) list.add(nums[max.getFirst()]);
        }
        int[] res = new int[list.size()];
        for (int i = 0; i < list.size(); i++) {
            res[i] = list.get(i);
        }
        return res;
    }
}
```

3、队列的最大值（剑指 59-2）

剑指 Offer 59 - II. 队列的最大值

难度 中等 132 收藏 分享 切换为英文 关注 反馈

请定义一个队列并实现函数 max_value 得到队列里的最大值，要求函数 max_value、push_back 和 pop_front 的均摊时间复杂度都是O(1)。

若队列为空，pop_front 和 max_value 需要返回 -1

示例 1：

```
输入：
["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
[[],[1],[2],[],[],[]]
输出：[null,null,null,2,1,2]
```

示例 2：

```
输入：
["MaxQueue","pop_front","max_value"]
[[],[],[]]
输出：[null,-1,-1]
```

限制：

- 1 <= push_back,pop_front,max_value的总操作数 <= 10000
- 1 <= value <= 10^5

https://leetcode-cn.com/problems/dui-lie-de-zui-da-zhi-lcof/solution/mian-shi-ti-59-ii-javashi-xian-yuan-li-he-mian-shi/

思路和剑指Offer59-2基本完全一致。
这里仅仅对涉及到的知识点进行复习：

ArrayList是顺序结构，所以定位很快，但插入，删除数据慢。
LinkedList 是链表结构，定位慢，但插入，删除数据快。

ArrayList实现了List接口，常见方法有：
add(); contains(); get(); indexOf():定位对象所处的位置; remove(); size(); toArray(); toString();//转换为字符串

LinkedList也实现了List接口外，可以实现上述ArrayList中的常用方法，此外：
1.LinkedList还实现了双向链表结构Deque，可以很方便的在头尾插入删除数据。
LinkedList<class> link = new LinkedList<>();
常用方法：addFirst(); addLast(); getFirst(); getLast(); removeFirst(); removeLast();

2.LinkedList除了实现了List和Deque外，还实现了Queue接口(队列),Queue是先进先出队列 FIFO。
Queue<class> queue = new LinkedList<>();
常用方法：poll()取出第一个元素; peek()查看第一个元素; offer()在最后添加元素,可用add()替换;

先进后出FILO Stack栈：
Stack<class> stack = new Stack<>();
常用方法：push();可用add();代替 pop();输出末尾的元素相当于LinkedList中的removeLast(); peek();查看最后一个元素，相当于getLast();

```
public class MaxQueue {

    Queue<Integer> queue;
    LinkedList<Integer> max;
    public MaxQueue() {
        queue = new LinkedList<>();
        max = new LinkedList<>();//LinkedList 是双端链表
    }

    public int max_value() {
        return max.size() == 0 ? -1 : max.getFirst();
    }

    public void push_back(int value) {
        queue.add(value);
        while(max.size() != 0 && max.getLast() < value){//注意：这里第二个判断条件不能带等号，即 max 中对于当前 queue 中的具有相同值的元素会全部存储，而不是存储最近的那个。
            max.removeLast();
        }
        max.add(value);
    }

    public int pop_front() {
        if(max.size() != 0 && queue.peek().equals(max.getFirst()))//Integer 类型的值的比较不能直接使用==
            max.removeFirst();
        return queue.size() == 0 ? -1 : queue.poll();
    }
```

```
    /**
     * Your MaxQueue object will be instantiated and called as such:
     * MaxQueue obj = new MaxQueue();
     * int param_1 = obj.max_value();
     * obj.push_back(value);
     * int param_3 = obj.pop_front();
     */
}
```

4、滑动窗口内数的和（lintcode 604）

604. 滑动窗口内数的和

中文 Eng

给你一个大小为n的整型数组和一个大小为k的滑动窗口，将滑动窗口从头移到尾，输出从开始到结束每一个时刻滑动窗口内的数的和。

样例

样例 1

```
输入: array = [1,2,7,8,5], k = 3
输出: [10,17,20]
解析:
1 + 2 + 7 = 10
2 + 7 + 8 = 17
7 + 8 + 5 = 20
```

算法：前缀和，滑动窗口

- 前缀和：

前缀和就是一个数列的前n项和，前缀和的计算非常简单，我们只需要额外开一个数组来记录它就好了。每次有新的元素出现我们令 `sum[i] = sum[i - 1] + nums[i]`，其中sum是我们用来记录前缀和的数组，nums是我们要输入的数列，i代表数列nums中的第i个数。这样的话我们仅仅需要O(N)的复杂度就可以完成计算，在查询宽度为k的区间之间的值的时候，只需要计算 `sum[i] - sum[i - k] + nums[i-k]` 即可

- 滑动窗口：

可以理解成是一个会滑动的窗口，每次记录下窗口的状态，再找出符合条件的适合的窗口，这题要求长度为k，我们维护一个区间[l,r]，先计算这个区间的和，然后每次l，r都向右移动一个单位，并减去nums[l]，加上nums[r+1]，得到新的区间的和

复杂度分析

- 时间复杂度 `O(n)`
  - n为数组的大小
- 空间复杂度 `O(n)`
  - n为数组的大小

方法 1：前缀和

```
public class Solution {
    /**
     * @param nums: a list of integers.
     * @param k: length of window.
     * @return: the sum of the element inside the window at each moving.
     */
    public int[] winSum(int[] nums, int k) {
        if (k == 0) return new int[0];
        int n = nums.length;
        int[] sum = new int[n];
        sum[0] = nums[0];
        for (int i = 1; i < n; i++){
            sum[i] = sum[i-1] + nums[i];
```

```
            }
            k--;
            int[] res = new int[n - k];
            for (int i = k; i < n; i++){
                res[i - k] = sum[i] - sum[i-k] + nums[i - k];
            }
            return res;
        }
}
```

方法 2：滑动窗口

```
public class Solution {
    public int[] winSum(int[] nums, int k) {
        if (k == 0) return new int[0];
        int n = nums.length;
        int left = 0, right = k - 1, sum = 0;
        for (int i = 0; i < k; i++){
            sum += nums[i];
        }
        int[] res = new int[n - k + 1];
        int idx = 0;
        res[idx++] = sum;
        while (right < n - 1) {
            sum -= nums[left];
            left++;
            right++;
            sum += nums[right];
            res[idx++] = sum;
        }
        return res;
    }
}
```

5、数据流的中位数（lintcode81）【优先队列维护最大堆和最小堆】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

81. 数据流中位数

数字是不断进入数组的，在每次添加一个新的数进入数组的同时返回当前新数组的中位数。

样例
样例1

```
输入：[1,2,3,4,5]
输出：[1,1,2,2,3]
样例说明：
[1] 和 [1,2] 的中位数是 1.
[1,2,3] 和 [1,2,3,4] 的中位数是 2.
[1,2,3,4,5] 的中位数是 3.
```

样例2

```
输入：[4,5,1,3,2,6,0]
输出：[4,4,4,3,3,3,3]
样例说明：
[4], [4,5] 和 [4,5,1] 的中位数是 4.
[4,5,1,3], [4,5,1,3,2], [4,5,1,3,2,6] 和 [4,5,1,3,2,6,0] 的中位数是 3.
```

挑战
时间复杂度为O(nlogn)

说明
中位数的定义：

- 这里的 **中位数** 不等同于数学定义里的 **中位数**。
- 中位数是排序后数组的中间值，如果有数组中有n个数，则中位数为$A[(n-1)/2]$。
- 比如：数组A=[1,2,3]的中位数是2，数组A=[1,19]的中位数是1。

https://www.jiuzhang.com/solution/find-median-from-data-stream/#tag-lang-java

按照题意中位数为 A(n-1)/2，即如果数组长度为奇数，中位数是最中间的那个，如果长度为偶数是中间偏左的那个元素，我们假设以中位数作为数组的分割点， 且将中位数始终划分到左半部分，那么数组的左半部分也可以看成一个最大堆，只需 poll 一次就可弹出当前的中位数，而右半部分就是最小堆，因为将中位数 划分在左半部分，那么左半部分的长度永远等于或者比右半部分大 1。用 maxheap 保存左半部分的数，用 minheap 保存右半部分的数。 把所有的数一左一右的加入到每个部分。左边部分最大的数就一直都是 median。 这个过程中，可能会出现左边部分并不完全都 <= 右边部分的情况。这种情况发生的时候，交换左边最大和右边最小的数即可。维护一个最大堆和一个最小堆，最大堆放数组前半部份的数（假设从小到大排序），最小堆放数组后半部分的数 如果数组个数为偶数，则最大堆和最小堆大小相同，按照题目要求，返回 num(n - 1) / 2，则是最大堆的 peek 如果数组个数为奇数，则最大堆元素个数= 最小堆+1, 则也正好是最大堆的 peek。

方法 1：复杂版本

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: the median of numbers
     */
    public int[] medianII(int[] nums) {
        if (nums == null || nums.length == 0) return new int[0];
```

```
        PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>(); //上升的顺序
        // PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(new
Comparator<Integer>(){
        //     public int compare(Integer a, Integer b) {
        //         return b - a; //下降的顺序
        //     }
        // });
        int n = nums.length;
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(n, (o1, o2) -> -o1+o2);
        int[] res = new int[n];
        for (int i = 0; i < n; i++){
            int cur = nums[i];
            if (maxHeap.isEmpty() || maxHeap.peek() >= cur){
                maxHeap.offer(cur);
            } else {
                minHeap.offer(cur);
            }
            //要保证 maxHeap.size() == minHeap.size() || maxHeap.size() == minHeap.size() +
1
            if (maxHeap.size() > minHeap.size() + 1){
                minHeap.offer(maxHeap.poll());
            }
            if (minHeap.size() > maxHeap.size()){
                maxHeap.offer(minHeap.poll());
            }
            res[i] = maxHeap.peek();
        }
        return res;
    }
}
```

方法 2：简单版本

不断维持最大堆和最小堆，而且变化的元素只可能在刚进入的元素和两个堆顶的元素，所以每次交换堆顶元素，剩下的活都留给 pq，这样代码就很简洁。

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: the median of numbers
     */
    public int[] medianII(int[] nums) {
        if (nums == null || nums.length == 0) return new int[0];
        PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>(); //上升的顺序
        // PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(new
Comparator<Integer>(){
        //     public int compare(Integer a, Integer b) {
```

禁止商用 公众号：yes的练级攻略

```
            //                return b - a; //下降的顺序
            //            }
            // }); //最大堆的第 1 种创建方式
            //PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(n, (o1, o2) ->
-o1+o2); //最大堆的第 2 种创建方式
            PriorityQueue<Integer> maxHeap = new
PriorityQueue<Integer>(Collections.reverseOrder());
            int n = nums.length;

            int[] res = new int[n];
            for (int i = 0; i < n; i++){
                int cur = nums[i];
                maxHeap.offer(cur);
                minHeap.offer(maxHeap.poll());
                maxHeap.offer(minHeap.poll());
                if (maxHeap.size() > minHeap.size() + 1) {
                    minHeap.offer(maxHeap.poll());
                }
                res[i] = maxHeap.peek();
            }
            return res;
        }
}
```

6、数据流的中位数（295）【剑指 41】【连续中值（面试题 17.20）】

295. 数据流的中位数

难度 困难 233 收藏 分享 切换为英文 关注 反馈

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

进阶：

1. 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？
2. 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

https://leetcode-cn.com/problems/shu-ju-liu-zhong-de-zhong-wei-shu-lcof/solution/mian-shi-ti-41-shu-ju-liu-zhong-de-zhong-wei-shu-y/

禁止商用 公众号：攻略

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

> 给定一长度为 N 的无序数组，其中位数的计算方法：首先对数组执行排序（使用 $O(N \log N)$ 时间），然后返回中间元素即可（使用 $O(1)$ 时间）。

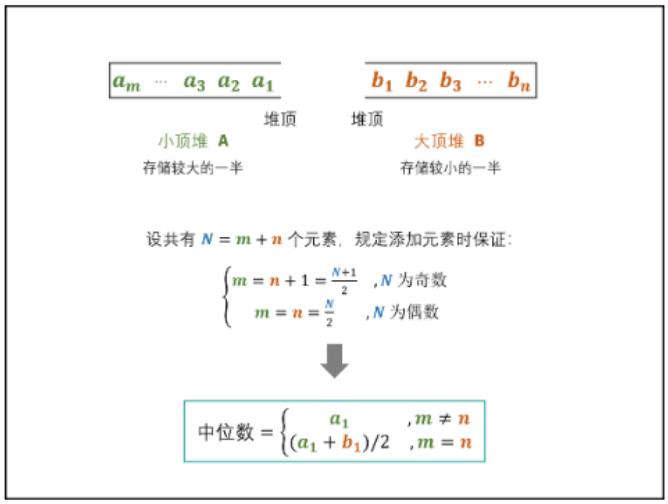
针对本题，根据以上思路，可以将数据流保存在一个列表中，并在添加元素时 **保持数组有序** 。此方法的时间复杂度为 $O(N)$ ，其中包括： 查找元素插入位置 $O(\log N)$ （二分查找）、向数组某位置插入元素 $O(N)$ （插入位置之后的元素都需要向后移动一位）。

> 借助 堆 可进一步优化时间复杂度。

建立一个 **小顶堆** A 和 **大顶堆** B ，各保存列表的一半元素，且规定：

- A 保存 **较大** 的一半，长度为 $\frac{N}{2}$ （ N 为偶数）或 $\frac{N+1}{2}$ （ N 为奇数）；
- B 保存 **较小** 的一半，长度为 $\frac{N}{2}$ （ N 为偶数）或 $\frac{N-1}{2}$ （ N 为奇数）；

随后，中位数可仅根据 A, B 的堆顶元素计算得到。



算法流程：

> 设元素总数为 $N = m + n$ ，其中 m 和 n 分别为 A 和 B 中的元素个数。

`addNum(num)` **函数：**

1. 当 $m = n$ （即 N 为 **偶数**）：需向 A 添加一个元素。实现方法：将新元素 num 插入至 B ，再将 B 堆顶元素插入至 A ；
2. 当 $m \neq n$ （即 N 为 **奇数**）：需向 B 添加一个元素。实现方法：将新元素 num 插入至 A ，再将 A 堆顶元素插入至 B ；

> 假设插入数字 num 遇到情况 `1.` 。由于 num 可能属于 “较小的一半” （即属于 B ），因此不能将 $nums$ 直接插入至 A 。而应先将 num 插入至 B ，再将 B 堆顶元素插入至 A 。这样就可以始终保持 A 保存较大一半、 B 保存较小一半。

`findMedian()` **函数：**

1. 当 $m = n$ （ N 为 **偶数**）：则中位数为 (A 的堆顶元素 + B 的堆顶元素)/2。
2. 当 $m \neq n$ （ N 为 **奇数**）：则中位数为 A 的堆顶元素。

☆收藏

复杂度分析：

- **时间复杂度：**
  - **查找中位数** $O(1)$：获取堆顶元素使用 $O(1)$ 时间；
  - **添加数字** $O(\log N)$：堆的插入和弹出操作使用 $O(\log N)$ 时间。
- **空间复杂度** $O(N)$：其中 N 为数据流中的元素数量，小顶堆 A 和大顶堆 B 最多同时保存 N 个元素。

代码：

Python 中 heapq 模块是小顶堆。实现 **大顶堆** 方法：小顶堆的插入和弹出操作均将元素 **取反** 即可。
Java 使用 `PriorityQueue<>((x, y) -> (y - x))` 可方便实现大顶堆。

```
class MedianFinder {
    Queue<Integer> A, B;
    public MedianFinder() {
        A = new PriorityQueue<>(); // 小顶堆，保存较大的一半
        B = new PriorityQueue<>((x, y) -> (y - x)); // 大顶堆，保存较小的一半
    }
    public void addNum(int num) {
        if(A.size() != B.size()) {
            A.add(num);
            B.add(A.poll());
        } else {
            B.add(num);
            A.add(B.poll());
        }
    }
    public double findMedian() {
        return A.size() != B.size() ? A.peek() : (A.peek() + B.peek()) / 2.0;
    }
}
/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder obj = new MedianFinder();
 * obj.addNum(num);
 * double param_2 = obj.findMedian();
 */
```

7、在排序数组中查找数组 1（剑指 53）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 53 - I. 在排序数组中查找数字 I

难度 简单 47 收藏 分享 切换为英文 关注 反馈

统计一个数字在排序数组中出现的次数。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: 2
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: 0
```

限制:

`0 <= 数组长度 <= 50000`

注意: 本题与主站 34 题相同（仅返回值不同）：https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/

https://leetcode-cn.com/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof/solution/er-fen-cha-zhao-fa-zhao-dao-di-yi-ci-chu-xian-de-w/

总结一下写二分查找法的思路：

- `while(left < right)` 在退出循环的时候，一定有 `left == right` 成立，因此，在循环中，不断排除不是目标元素的区间，以确定下一轮搜索的区间，进而确定边界如何设置，在退出循环的时候，根据情况，需要单独做一次判断；
- 在循环体里，对 `nums[mid]` 与 `target` 的大小关系的判断，可以二分，也可以三分，本题采用三分去判断，只是为了演示怎么设置边界：先得到目标元素可能在哪个区间里，然后再设置边界，这样不容易出错，建议写成注释；可以二分的好处是只要一边判断是正确的，另一边就是其反面，考虑的内容会少一些；
- 看到边界是 `left = mid` 的时候，取中间数的计算表达式应该上取整，即在括号里加 `1`；
- 编码时应该先写主干逻辑，然后分支逻辑作为私有函数。

```
class Solution {
    public int search(int[] nums, int target) {
        if (nums == null || nums.length == 0) return 0;
        int firstPosition = findFirstPosition(nums, target);
        if (firstPosition == -1) return 0 ;
        int lastPosition = findLastPosition(nums, target);
        return lastPosition - firstPosition + 1;
    }
    private int findFirstPosition (int[] nums, int target){
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid =  left + (right - left) / 2;
            if (nums[mid] < target) {
```

```
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        if (nums[left] == target) {
            return left;
        } else {
            return -1;
        }
    }
    private int findLastPosition (int[] nums, int target){
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid =  left + (right - left + 1) / 2;
            if (nums[mid] > target) {
                right = mid - 1;
            } else {
                left = mid;
            }
        }
        return left;
    }
}
```

8、合并排序的数组（面试题 10.01）

面试题 10.01. 合并排序的数组

难度 简单 69 收藏 分享 切换为英文 关注 反馈

给定两个排序后的数组 A 和 B，其中 A 的末端有足够的缓冲空间容纳 B。 编写一个方法，将 B 合并入 A 并排序。

初始化 A 和 B 的元素数量分别为 *m* 和 *n*。

示例:

```
输入:
A = [1,2,3,0,0,0], m = 3
B = [2,5,6],       n = 3

输出: [1,2,2,3,5,6]
```

说明:

- `A.length == n + m`

https://leetcode-cn.com/problems/sorted-merge-lcci/solution/5xing-dai-ma-shuang-bai-dao-xu-fa-shi-jian-fu-za-d/

解题思路

需要比较 $m+n$ 个数的大小，所以先保证其中至少 m 或 n 个数比较完毕并放在正确位置，再考虑剩下的数。
注意 m 和 n 是未遍历的、有效的数字的个数，不是下标，作为下标要 -1 。写的时候偷懒，可能有些地方可读性比较差。

```
class Solution {
    public void merge(int[] A, int m, int[] B, int n) {
        // 先确保将其中一个数组中的数字遍历完
        while (m > 0 && n > 0) {
            // 对比选出较大的数放在 m + n - 1 的位置，并将选出此数的指针向前移动
            A[m + n - 1] = A[m - 1] > B[n - 1] ? A[m-- - 1] : B[n-- - 1];
        }
        // 剩下的数都比已经遍历过的数小
        // 如果 m 不为 0，则 A 没遍历完，都已经在 A 中不用再管
        // 如果 n 不为 0，则 B 没遍历完，直接全移到 A 中相同的位置
        while (n > 0) {
            A[n - 1] = B[n - 1];
            n--;
        }
    }

    // 测试一下
    public static void main(String[] args) {
        int[] A = new int[] {1, 2, 3, 0, 0, 0};
        int[] B = new int[] {2, 5, 6};
        new Solution().merge(A, 3, B, 3);
        for (int i = 0; i < A.length; i++)
            System.out.print(A[i] + ",");
    }
}
```

9、多数元素（169）【数组中出现次数超过一半的数字（剑指 39）】【主要元素（面试题 17.10）】

169. 多数元素

难度 简单 632 收藏 分享 切换为英文 关注 反馈

给定一个大小为 *n* 的数组，找到其中的多数元素。多数元素是指在数组中出现次数**大于** `⌊ n/2 ⌋` 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

```
输入: [3,2,3]
输出: 3
```

示例 2:

```
输入: [2,2,1,1,1,2,2]
输出: 2
```

面试题 17.10. 主要元素

难度 简单 27 收藏 分享 切换为英文 关注 反馈

数组中占比超过一半的元素称之为主要元素。给定一个**整数**数组，找到它的主要元素。若没有，返回-1。

示例 1:

```
输入: [1,2,5,9,5,9,5,5,5]
输出: 5
```

示例 2:

```
输入: [3,2]
输出: -1
```

示例 3:

```
输入: [2,2,1,1,1,2,2]
输出: 2
```

说明：
你有办法在时间复杂度为 O(N)，空间复杂度为 O(1) 内完成吗?

通过次数 12,990 提交次数 20,129

https://leetcode-cn.com/problems/majority-element/solution/duo-shu-yuan-su-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法五：Boyer-Moore 投票算法

思路

如果我们把众数记为 $+1$，把其他数记为 -1，将它们全部加起来，显然和大于 `0`，从结果本身我们可以看出众数比其他数多。

算法

Boyer-Moore 算法的本质和方法四中的分治十分类似。我们首先给出 Boyer-Moore 算法的详细步骤：

- 我们维护一个候选众数 `candidate` 和它出现的次数 `count`。初始时 `candidate` 可以为任意值，`count` 为 `0`；
- 我们遍历数组 `nums` 中的所有元素，对于每个元素 `x`，在判断 `x` 之前，如果 `count` 的值为 `0`，我们先将 `x` 的值赋予 `candidate`，随后我们判断 `x`：
  - 如果 `x` 与 `candidate` 相等，那么计数器 `count` 的值增加 `1`；
  - 如果 `x` 与 `candidate` 不等，那么计数器 `count` 的值减少 `1`。
- 在遍历完成后，`candidate` 即为整个数组的众数。

我们举一个具体的例子，例如下面的这个数组：

```
[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]
```

在遍历到数组中的第一个元素以及每个在 `|` 之后的元素时，`candidate` 都会因为 `count` 的值变为 `0` 而发生改变。最后一次 `candidate` 的值从 `5` 变为 `7`，也就是这个数组中的众数。

Boyer-Moore 算法的正确性较难证明，这里给出一种较为详细的用例子辅助证明的思路，供读者参考：

首先我们根据算法步骤中对 `count` 的定义，可以发现：在对整个数组进行遍历的过程中，`count` 的值一定非负。这是因为如果 `count` 的值为 `0`，那么在这一轮遍历的开始时刻，我们会将 `x` 的值赋予 `candidate` 并在接下来的一步中将 `count` 的值增加 `1`。因此 `count` 的值在遍历的过程中一直保持非负。

那么 `count` 本身除了计数器之外，还有什么更深层次的意义呢？我们还是以数组

```
[7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]
```

作为例子，首先写下它在每一步遍历时 `candidate` 和 `count` 的值：

```
nums:      [7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]
candidate:  7  7  7  7  7  7   5  5   5  5  5  5   7  7  7  7
count:      1  2  1  2  1  0   1  0   1  2  1  0   1  2  3  4
```

更多硬核内容就在公众号「yes 的练级攻略」

我们再定义一个变量 `value`，它和真正的众数 `maj` 绑定。在每一步遍历时，如果当前的数 `x` 和 `maj` 相等，那么 `value` 的值加 `1`，否则减 `1`。`value` 的实际意义即为：到当前的这一步遍历为止，众数出现的次数比非众数多出了多少次。我们将 `value` 的值也写在下方：

```
nums:     [7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]
value:     1  2  1  2  1  0  -1  0  -1 -2 -1  0  1  2  3  4
```

有没有发现什么？我们将 `count` 和 `value` 放在一起：

```
nums:     [7, 7, 5, 7, 5, 1 | 5, 7 | 5, 5, 7, 7 | 7, 7, 7, 7]
count:     1  2  1  2  1  0  1  0  1  2  1  0  1  2  3  4
value:     1  2  1  2  1  0 -1  0 -1 -2 -1  0  1  2  3  4
```

发现在每一步遍历中，`count` 和 `value` 要么相等，要么互为相反数！并且在候选众数 `candidate` 就是 `maj` 时，它们相等，`candidate` 是其它的数时，它们互为相反数！

为什么会有这么奇妙的性质呢？这并不难证明：我们将候选众数 `candidate` 保持不变的连续的遍历称为「一段」。在同一段中，`count` 的值是根据 `candidate == x` 的判断进行加减的。那么如果 `candidate` 恰好为 `maj`，那么在这一段中，`count` 和 `value` 的变化是同步的；如果 `candidate` 不为 `maj`，那么在这一段中 `count` 和 `value` 的变化是相反的。因此就有了这样一个奇妙的性质。

这样以来，由于：

- 我们证明了 `count` 的值一直为非负，在最后一步遍历结束后也是如此；
- 由于 `value` 的值与真正的众数 `maj` 绑定，并且它表示「众数出现的次数比非众数多出了多少次」，那么在最后一步遍历结束后，`value` 的值为正数；

在最后一步遍历结束后，`count` 非负，`value` 为正数，所以它们不可能互为相反数，只可能相等，即 `count == value`。因此在最后「一段」中，`count` 的 `value` 的变化是同步的，也就是说，`candidate` 中存储的候选众数就是真正的众数 `maj`。

复杂度分析

- 时间复杂度：$O(n)$。Boyer-Moore 算法只对数组进行了一次遍历。
- 空间复杂度：$O(1)$。Boyer-Moore 算法只需要常数级别的额外空间。

```
class Solution {
    public int majorityElement(int[] nums) {
        int count = 0;
        Integer candidate = null;
        for (int num : nums) {
            if (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }

        return candidate;
    }
}
```

其实上面这种没有拆开写的快，下面这种更快。

```
class Solution {
```

```
    public int majorityElement(int[] nums) {
        int preValue = nums[0];
        int count = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] == preValue) {
                count++;
            } else {
                count--;
                if (count == 0) {
                    preValue = nums[i];
                    count = 1;
                }
            }
        }
        return preValue;
    }
}
```

10、求众数 2（229）

229. 求众数 II

难度 中等 226 收藏 分享 切换为英文 关注

给定一个大小为 *n* 的数组，找出其中所有出现超过 ⌊ n/3 ⌋ 次的元素。

说明: 要求算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

示例 1:

```
输入: [3,2,3]
输出: [3]
```

示例 2:

```
输入: [1,1,1,3,3,2,2,2]
输出: [1,2]
```

https://leetcode-cn.com/problems/majority-element-ii/solution/liang-fu-dong-hua-yan-shi-mo-er-tou-piao-fa-zui-zh/

摩尔投票法，解决的问题是如何在任意多的候选人中，选出票数超过一半的那个人。注意，是超出一半票数的那个人。

假设投票是这样的，[A, C, A, A, B]，ABC是指三个候选人。

第一张票与第二张票进行对坑，如果票不同则互相抵消掉；

接着第三票与第四票进行对坑，如果票相同，则增加这个候选人的可抵消票数；

这个候选人拿着可抵消票数与第五张票对坑，如果票不同，则互相抵消掉，即候选人的可抵消票数-1。

如果最后得到的抵消票数不为0的话，那说明他可能希望的，这是我们需要一个阶段来验证这个候选人的票数是否超过一半——**计数阶段**。

所以摩尔投票法分为两个阶段：抵消阶段和计数阶段。

抵消阶段：两个不同投票进行对坑，并且同时抵消掉各一张票，如果两个投票相同，则累加可抵消的次数；

计数阶段：在抵消阶段最后得到的抵消计数只要不为0，那这个候选人是有可能超过一半的票数的，为了验证，则需要遍历一次，统计票数，才可确定。

摩尔投票法经历两个阶段最多消耗O(2n)的时间，也属于O(n)的线性时间复杂度，另外空间复杂度也达到常量级。

理解摩尔投票法之后，我们再回到题目描述，题目可以看作是：在任意多的候选人中，选出票数超过 `[ 1/3 ]` 的候选人。

我们可以这样理解，假设投票是这样的[A, B, C, A, A, B, C]，ABC是指三个候选人。

第1张票，第2张票和第3张票进行对坑，如果票都不同，则互相抵消掉；

第4张票，第5张票和第6张票进行对坑，如果有部分相同，则累计增加他们的可抵消票数，如[A, 2]和[B, 1]；

接着将[A, 2]和[B, 1]与第7张票进行对坑，如果票都没匹配上，则互相抵消掉，变成[A, 1]和[B, 0]。

```
class Solution {
    public List<Integer> majorityElement(int[] nums) {
        // 创建返回值
        List<Integer> res = new ArrayList<>();
        if (nums == null || nums.length == 0) return res;
        // 初始化两个候选人 candidate，和他们的计票
        int cand1 = nums[0], count1 = 0;
        int cand2 = nums[0], count2 = 0;
        // 摩尔投票法，分为两个阶段：配对阶段和计数阶段
        // 配对阶段
        for (int num : nums) {
            // 投票
            if (cand1 == num) {
                count1++;
                continue;
            }
            if (cand2 == num) {
                count2++;
                continue;
            }
            // 第 1 个候选人配对
            if (count1 == 0) {
                cand1 = num;
                count1++;
                continue;
            }
            // 第 2 个候选人配对
            if (count2 == 0) {
                cand2 = num;
                count2++;
```

```
                continue;
            }
            count1--;
            count2--;
        }
        // 计数阶段
        // 找到了两个候选人之后，需要确定票数是否满足大于 N/3
        count1 = 0;
        count2 = 0;
        for (int num : nums) {
            if (cand1 == num) count1++;
            else if (cand2 == num) count2++;
        }
        if (count1 > nums.length / 3) res.add(cand1);
        if (count2 > nums.length / 3) res.add(cand2);
        return res;
    }
}
```

11、数组的相对排序（1122）【桶排序】

1122. 数组的相对排序

难度 简单 59 收藏 分享 切换为英文 关注 反馈

给你两个数组，arr1 和 arr2，

- arr2 中的元素各不相同
- arr2 中的每个元素都出现在 arr1 中

对 arr1 中的元素进行排序，使 arr1 中项的相对顺序和 arr2 中的相对顺序相同。未在 arr2 中出现过的元素需要按照升序放在 arr1 的末尾。

示例：

输入：arr1 = [2,3,1,3,2,4,6,7,9,2,19], arr2 = [2,1,4,3,9,6]
输出：[2,2,2,1,4,3,3,9,6,7,19]

提示：

- arr1.length, arr2.length <= 1000
- 0 <= arr1[i], arr2[i] <= 1000
- arr2 中的元素 arr2[i] 各不相同
- arr2 中的每个元素 arr2[i] 都出现在 arr1 中

通过次数 17,477 | 提交次数 26,430

https://leetcode-cn.com/problems/relative-sort-array/solution/java-tong-shi-pai-xu-ji-shu-pai-xu-shuang-100-by-e/

```
class Solution {
    public int[] relativeSortArray(int[] arr1, int[] arr2) {
        //由于arr1 的可能取值为0-1000，共1001 个取值，不算一个很大的取值范围，所以可以利用桶式排序
        int[] bucket = new int[1001];
        //计数每个桶有多少个数，也就是每个值在数组中有几个
        for(int num:arr1){
```

```
            bucket[num]++;
        }
        //由于重新排序不会改变数组的长度，所以可以利用原数组，可以节省空间复杂度
        int i = 0;
        //由于排序是按照相对顺序进行排序，所以，首先根据 arr2 中的桶的顺序，依次从对应的桶中取数到 arr1 中
        //并注意，每拿出一个数，需要将对桶中的数的个数进行-1 操作
        for(int num:arr2){
            while(bucket[num]-- > 0){
                arr1[i++] = num;
            }
        }
        //未在 arr2 中的桶中的数，按照桶号升序进行输出，所以进行二次遍历
        for(int j = 0; j < 1001; ++j){
            while(bucket[j]-- > 0){
                arr1[i++] = j;
            }
        }
        return arr1;
    }
}
```

12、雀魂启动（lintcode1527）

公众号：
yes的练级攻略

1527. 雀魂启动!

根据一款叫做雀魂的麻将游戏，简化了游戏规则，具体如下：
1.总共有36张牌，每张牌是1~9。每个数字4张牌。
2.如果取出14张牌满足如下条件，即算作和牌：
①14张牌中有2张相同数字的牌，称为雀头。
②除去上述2张牌，剩下12张牌可以组成4个顺子或刻子。

注：顺子的意思是递增的连续3个数字牌（例如234,567等），刻子的意思是相同数字的3个数字牌（例如111,777）。

从36张牌中抽取了13张牌，那么在剩下的23张牌中再取一张牌，取到哪几种数字牌可以和牌?

样例
样例 1:

输入:
[1, 1, 1, 2, 2, 2, 5, 5, 5, 6, 6, 6, 9]
输出:
[9]
说明：可以组成1,2,5,6的4个刻子和9的雀头。
样例 2:

输入:
[1, 1, 1, 1, 2, 2, 3, 3, 5, 6, 7, 8, 9]
输出:
[4, 7]
说明:
用1做雀头，组123,123,567或456,789的四个顺子。
注意事项
若满足条件的有多种牌，请按从小到大的顺序返回。若没有满足条件的牌，请返回0。

```
public class Solution {
    /**
     * @param cards: A list of cards.
     * @return: A list of feasible solution.
     */
    public List<Integer> getTheNumber(int[] cards) {
        // write your code here
        int[] times = new int[10];
        List<Integer> res = new ArrayList<>();
        for (int i = 0; i < 13; i++) {
            times[cards[i]]++;
        }
        // 和牌：雀头+4*顺子/刻子
        for (int i = 1; i < 10; i++) {
            if (times[i] < 4) {
                times[i]++;
                if (check(times)){
                    res.add(i);
                }
                times[i]--;
            }
        }
```

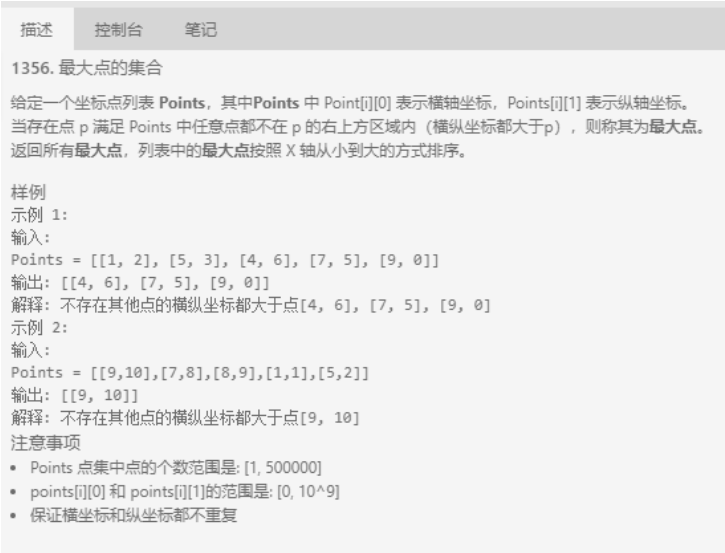
公众号：yes的练级攻略

```
        if (res.size() == 0) res.add(0);
        return res;
    }
    // times[i]表示数字 i 出现的次数
    private boolean check(int[] times){
        int[] tmp = new int[10];
        //i 为雀头
        for (int i = 1; i < 10; i++) {
            if (times[i] < 2) {
                continue;
            }
            //ok 标记是否可以和牌的状态
            boolean ok = true;
            for (int k = 1; k <10; k++) {
                tmp[k] = times[k];
            }
            tmp[i] -= 2;
            for (int j = 1; j < 10 && ok; j++) {
                //  刻子 jjj
                if (tmp[j] >= 3 ){
                    tmp[j] -= 3;
                }
                while (tmp[j] > 0 && ok) {
                    if (j + 2 > 9) {
                        ok = false;
                        break;
                    }
                    //  顺子(j, j + 1, j + 2)
                    if (tmp[j+1] >0 && tmp[j+2] >0) {
                        tmp[j]--;
                        tmp[j+1]--;
                        tmp[j+2]--;
                    }else {
                        ok = false;
                    }
                }
            }
            if (ok) return true;
        }
        return false;
    }
}
```

13、最大点的集合（lintcode1356）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」



描述 控制台 笔记

1356. 最大点的集合

给定一个坐标点列表 **Points**，其中**Points** 中 Point[i][0] 表示横轴坐标，Points[i][1] 表示纵轴坐标。
当存在点 p 满足 Points 中任意点都不在 p 的右上方区域内（横纵坐标都大于p），则称其为**最大点**。
返回所有**最大点**，列表中的**最大点**按照 X 轴从小到大的方式排序。

样例
示例 1:
输入:
Points = [[1, 2], [5, 3], [4, 6], [7, 5], [9, 0]]
输出: [[4, 6], [7, 5], [9, 0]]
解释: 不存在其他点的横纵坐标都大于点[4, 6], [7, 5], [9, 0]
示例 2:
输入:
Points = [[9,10],[7,8],[8,9],[1,1],[5,2]]
输出: [[9, 10]]
解释: 不存在其他点的横纵坐标都大于点[9, 10]
注意事项

- Points 点集中点的个数范围是: [1, 500000]
- points[i][0] 和 points[i][1]的范围是: [0, 10^9]
- 保证横坐标和纵坐标都不重复

https://www.jiuzhang.com/solution/maximum-points-set/

首先对于所有的点进行 x 坐标从大到小排序。 然后用一个变量 max_y 来存储目前最大的 y。 当你遍历这个排好序的数组的时候，只要该点的 y 大于 max_y 就证明是最大点。

```
public class Solution {
    /**
     * @param Points: a list of [x, y]
     * @return: return a list of points
     */
    Comparator<Node> comparator = new Comparator<Node>(){
        public int compare(Node a, Node b) {
            return b.x - a.x;
        }
    };
    public List<List<Integer>> MaximumPointsSet(int[][] Points) {
        int n = Points.length, max_y = -1;
        int[] vis = new int[n];
        List<Node> nums = new ArrayList<Node>();
        List<List<Integer>> result = new ArrayList();

        for(int i = 0;i < n;i++){
            nums.add(new Node(Points[i][0], Points[i][1]));
        }

        Collections.sort(nums,comparator);

        for(int i = 0; i < n; i++){
            if(nums.get(i).y > max_y){
                vis[i] = 1;
            }
            max_y = Math.max(nums.get(i).y, max_y);
```

禁止商用 公众号：yes的练级攻略

```
            }

            for(int i = n - 1; i >= 0; i--){
                if(vis[i] == 1){
                    List<Integer> temp = new ArrayList<Integer>();
                    temp.add(nums.get(i).x);
                    temp.add(nums.get(i).y);
                    result.add(temp);
                }
            }
            return result;
        }
}
class Node{
        public int x, y;
        Node(){}
        Node(int a, int b){
            x = a;
            y = b;
        }
}
```

禁止商用

14、顺时针打印矩阵（剑指 29）【与下一题只有返回值不同】

剑指 Offer 29. 顺时针打印矩阵

难度 简单 101 收藏 分享 切换为英文 关注 反馈

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。

示例 1：

```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]
```

示例 2：

```
输入：matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
输出：[1,2,3,4,8,12,11,10,9,5,6,7]
```

限制：

- `0 <= matrix.length <= 100`
- `0 <= matrix[i].length <= 100`

注意：本题与主站 54 题相同：https://leetcode-cn.com/problems/spiral-matrix/

更多硬核内容就在公众号「yes 的练级攻略」

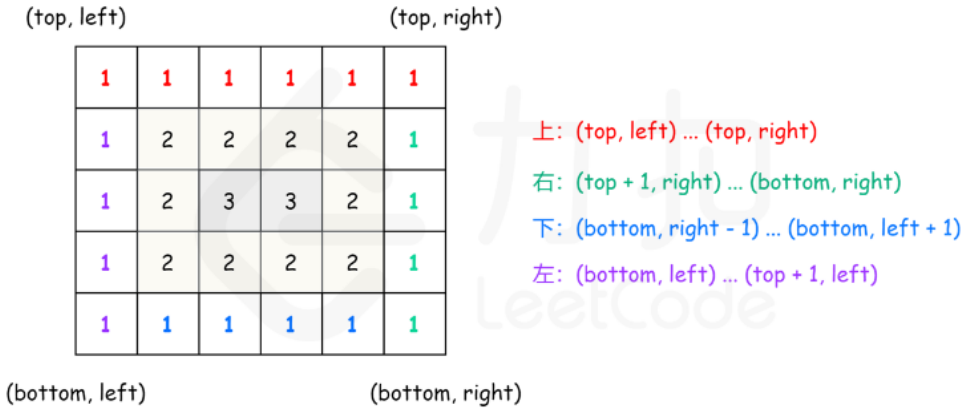
https://leetcode-cn.com/problems/shun-shi-zhen-da-yin-ju-zhen-lcof/solution/shun-shi-zhen-da-yin-ju-zhen-by-leetcode-solution/

对于每层，从左上方开始以顺时针的顺序遍历所有元素。假设当前层的左上角位于 $(top, left)$，右下角位于 $(bottom, right)$，按照如下顺序遍历当前层的元素。

1. 从左到右遍历上侧元素，依次为 $(top, left)$ 到 $(top, right)$。
2. 从上到下遍历右侧元素，依次为 $(top + 1, right)$ 到 $(bottom, right)$。
3. 如果 $left < right$ 且 $top < bottom$，则从右到左遍历下侧元素，依次为 $(bottom, right - 1)$ 到 $(bottom, left + 1)$，以及从下到上遍历左侧元素，依次为 $(bottom, left)$ 到 $(top + 1, left)$。

遍历完当前层的元素之后，将 $left$ 和 top 分别增加 1，将 $right$ 和 $bottom$ 分别减少 1，进入下一层继续遍历，直到遍历完所有元素为止。

☆收藏




```
class Solution {
    public int[] spiralOrder(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length ==
 0) {
            return new int[0];
        }
        int rows = matrix.length, columns = matrix[0].length;
        int[] order = new int[rows * columns];
        int index = 0;
        int left = 0, right = columns - 1, top = 0, bottom = rows - 1;
        while (left <= right && top <= bottom) {
            for (int column = left; column <= right; column++) {
                order[index++] = matrix[top][column];
            }
            for (int row = top + 1; row <= bottom; row++) {
                order[index++] = matrix[row][right];
            }
            if (left < right && top < bottom) {
                for (int column = right - 1; column > left; column--) {
                    order[index++] = matrix[bottom][column];
                }
                for (int row = bottom; row > top; row--) {
```

```
                    order[index++] = matrix[row][left];
                }
            }
            left++;
            right--;
            top++;
            bottom--;
        }
        return order;
    }
}
```

思考题：顺时针打印矩阵的最后一个点的坐标（假设坐标从 0 开始）

每走一行 行数-1，每走一列 列数-1，行列交替走，那么数量少的那个就先走完。

由于不管行还是列都是左右左右依次消除的 所以少的那个最后一次是走在中间的位置。

假设少的那个是行 那么走完最后一行 下一次要走哪一列也可以直接求出来 这个列就是另一个坐标。

假设 m 为总行数，n 为总列数， x 是终点横坐标，y 是终点纵坐标

（1）m <= n 时，且 m 为奇数时候：x = m/2，y = n - m/2 - 1;

（2）m <= n 时，且 m 为偶数时候：x = m/2，y = m/2 - 1;

（3）m > n 时，且 n 为奇数时候：x = m - n/2 - 1，y = n/2;

（4）m > n 时，且 n 为偶数时候：x = n/2，y = n/2 - 1。

测试代码

```
import java.util.*;
public class Main {
    public static void main(String[] args){
        int[][] matrix = {{1},{5},{9},{13},{17},{21}};
        for (int i = 0; i < matrix.length; i++){
            for (int j = 0; j < matrix[0].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
        int[] order = spiralOrder(matrix);
        for (int i = 0; i < order.length; i++) {
            System.out.print(order[i] + " ");
        }
        int[] xy = spiralOrderXY(matrix);
        System.out.println();
        System.out.println(xy[0] + "," + xy[1]);
        System.out.println(matrix[xy[0]][xy[1]]);
    }
    public static int[] spiralOrder(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return new int[0];
```

```
        }
        int rows = matrix.length, columns = matrix[0].length;
        int[] order = new int[rows * columns];
        int index = 0;
        int left = 0, right = columns - 1, top = 0, bottom = rows - 1;
        while (left <= right && top <= bottom) {
            for (int column = left; column <= right; column++) {
                order[index++] = matrix[top][column];
            }
            for (int row = top + 1; row <= bottom; row++) {
                order[index++] = matrix[row][right];
            }
            if (left < right && top < bottom) {
                for (int column = right - 1; column > left; column--) {
                    order[index++] = matrix[bottom][column];
                }
                for (int row = bottom; row > top; row--) {
                    order[index++] = matrix[row][left];
                }
            }
            left++;
            right--;
            top++;
            bottom--;
        }
        return order;
    }
    public static int[] spiralOrderXY(int[][] matrix) {
        int[] xy = new int[2];
        int m = matrix.length, n = matrix[0].length;
        if (m <= n) {
            xy[0] = m / 2;
            if (m % 2 == 1) {
                xy[1] = n - m / 2 - 1;
            } else {
                xy[1] = m / 2 - 1;
            }
        } else {
            if (n % 2 == 1) {
                xy[0] = m - n / 2 - 1;
                xy[1] = n / 2;
            } else {
                xy[0] = n / 2;
                xy[1] = n / 2 - 1;
```

```
            }
        }
        return xy;
    }
}
```

15、螺旋矩阵（54）【与上一题只有返回值不同】

54. 螺旋矩阵

难度 中等 437 收藏 分享 切换为英文 关注 反馈

给定一个包含 m x n 个元素的矩阵（m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

示例 1:

```
输入:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
输出: [1,2,3,6,9,8,7,4,5]
```

示例 2:

```
输入:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
```

```java
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> res = new ArrayList<>();
        if (matrix == null || matrix.length == 0 || matrix[0].length ==
 0) {
            return res;
        }
        int rows = matrix.length, cols = matrix[0].length;
        int left = 0, right = cols - 1, top = 0, bottom = rows - 1;
        while (left <= right && top <= bottom){
            for (int col = left; col <= right; col++){
                res.add(matrix[top][col]);
            }
            for (int row = top + 1; row <= bottom; row++){
                res.add(matrix[row][right]);
            }
```

```
            if (left < right && top < bottom){
                for (int col = right - 1; col > left; col--){
                    res.add(matrix[bottom][col]);
                }
                for (int row = bottom; row > top; row--){
                    res.add(matrix[row][left]);
                }
            }
            left++;
            right--;
            top++;
            bottom--;
        }
        return res;
    }
}
```

16、螺旋矩阵 2（59）【类比上面两题】

59. 螺旋矩阵 II

难度 中等 212 收藏 分享 切换为英文 关注 反馈

给定一个正整数 n，生成一个包含 1 到 n^2 所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。

示例:

输入: 3
输出:

```
[
 [ 1, 2, 3 ],
 [ 8, 9, 4 ],
 [ 7, 6, 5 ]
]
```

```
class Solution {
    public int[][] generateMatrix(int n) {
        int[][] res = new int[n][n];
        int num = 1;
        int rows = n, cols = n;
        int left = 0, right = cols - 1, top = 0, bottom = rows - 1;
        while (left <= right && top <= bottom){
            for (int col = left; col <= right; col++){
                res[top][col] = num;
                num++;
            }
            for (int row = top + 1; row <= bottom; row++){
```

```
                res[row][right] = num;
                num++;
            }
            if (left < right && top < bottom) {
                for (int col = right - 1; col > left; col--) {
                    res[bottom][col] = num;
                    num++;
                }
                for (int row = bottom; row > top; row--){
                    res[row][left] = num;
                    num++;
                }
            }
            left++;
            right--;
            top++;
            bottom--;
        }
        return res;
    }
}
```

17、旋转图像（48）【旋转矩阵（面试题 01.07）】

48. 旋转图像

难度 中等 488 收藏 分享 切换为英文 关注 反馈

给定一个 *n* × *n* 的二维矩阵表示一个图像。

将图像顺时针旋转 90 度。

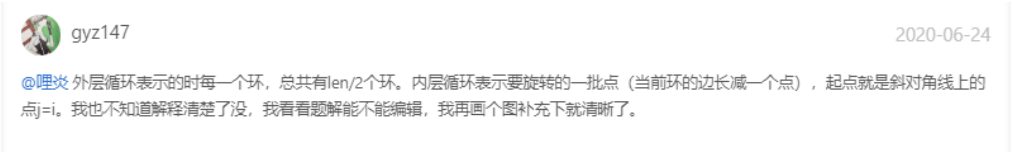
说明：

你必须在原地旋转图像，这意味着你需要直接修改输入的二维矩阵。**请不要**使用另一个矩阵来旋转图像。

示例 1:

```
给定 matrix =
[
  [1,2,3],
  [4,5,6],
  [7,8,9]
],

原地旋转输入矩阵，使其变为:
[
  [7,4,1],
  [8,5,2],
  [9,6,3]
]
```

https://leetcode-cn.com/problems/rotate-image/solution/zhu-ceng-zhu-ge-fan-zhuan-javachao-guo-100-by-gyz1/




```
class Solution {
    public void rotate(int[][] matrix) {
        if (matrix == null || matrix.length <= 1) return;
        int len = matrix.length;
        int tmp = 0;
        for (int i = 0; i < len >> 1; i++) {
            for (int j = i; j < len - i - 1; j++) {
             tmp = matrix[i][j];
             matrix[i][j] = matrix[len - j - 1][i];
             matrix[len - j - 1][i] = matrix[len - i - 1][len - j - 1];
             matrix[len - i - 1][len - j - 1] = matrix[j][len - i - 1];
             matrix[j][len - i - 1] = tmp;
            }
        }
    }
}
```

18、跳水板（面试题 16.11）

面试题 16.11. 跳水板

难度 简单 24 收藏 分享 切换为英文 关注 反馈

你正在使用一堆木板建造跳水板。有两种类型的木板，其中长度较短的木板长度为 `shorter`，长度较长的木板长度为 `longer`。你必须正好使用 `k` 块木板。编写一个方法，生成跳水板所有可能的长度。

返回的长度需要从小到大排列。

示例：

```
输入：
shorter = 1
longer = 2
k = 3
输出： {3,4,5,6}
```

提示：

- 0 < shorter <= longer
- 0 <= k <= 100000

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/diving-board-lcci/solution/tiao-shui-ban-by-leetcode-solution/

方法一：数学

首先考虑两种边界情况。

- 如果 $k = 0$，则不能建造任何跳水板，因此返回空数组。
- 如果 $shorter$ 和 $longer$ 相等，则建造的跳水板的长度是唯一的，都等于 $shorter * k$，因此返回长度为 1 的数组，数组中的元素为 $shorter * k$。

然后考虑一般情况，即 $shorter < longer$ 且 $k > 0$。由于短木板和长木板一共使用 k 块，因此一共有 $k + 1$ 种组合，每种组合下建造的跳水板长度都是不一样的，一共有 $k + 1$ 种不同的长度。

为什么每种组合下建造的跳水板长度都是不一样的？考虑以下两种不同的组合：第一种组合，有 i 块长木板，则跳水板的长度是 $shorter * (k - i) + longer * i$；第二种组合，有 j 块长木板，则跳水板的长度是 $shorter * (k - j) + longer * j$。其中 $0 \le i < j \le k$。则两种不同的组合下的跳水板长度之差为：

$$(shorter * (k - i) + longer * i) - (shorter * (k - j) + longer * j) = (longer - shorter) * (i - j)$$

由于 $longer > shorter$ 且 $i < j$，因此上式的值小于 0。由此可见，任意两种不同的组合下的跳水板长度都是不一样的，而且使用的长木板越多，跳水板的长度越大。

因此创建长度为 $k + 1$ 的数组 lengths，对于 $0 \le i \le k$，令 $\text{lengths}[i] = shorter * (k - i) + longer * i$，则 lengths 包含跳水板所有可能的长度，且长度为升序排序。

复杂度分析

- 时间复杂度：$O(k)$，其中 k 是木板数量。短木板和长木板一共使用 k 块，一共有 $k + 1$ 种组合，对于每种组合都要计算跳水板的长度。
- 空间复杂度：$O(1)$。除了返回值以外，额外使用的空间复杂度为常数。

```
class Solution {
    public int[] divingBoard(int shorter, int longer, int k) {
        if (k == 0) return new int[] {};
        if (shorter == longer) {
            return new int[] {k * shorter};
        }

        int[] res = new int[k + 1];
        for (int i = 0; i <= k; i++) {
            res[i] = i * longer + (k - i) * shorter;
        }

        return res;
    }
}
```

19、序列维护（lintcode1329）

1329. 序列维护

有一个长度为 n 的序列 a，在序列上进行 q 次操作。
每次操作有一个查询的数字 bi，将序列中所有大于等于 bi 的数字减一。
返回每次操作后有多少数字被减一。

样例
样例1

输入
4
3
[1,2,3,4]
[4,3,1]
输出
[1,2,4]
样例2

输入
3
2
[1,2,3]
[3,3]
输出
[1,0]
注意事项
$1 \le n, q \le 10000, 1 \le a_i, x \le n$

首先将数组排序,因为修改的限制,所以数组只会是一个不减的数组. 那么我们就可以二分确定第一个>=b[i]的位置 pos 如果不存在这个位置,就说明没有一个元素符合条件。否则输出 n-l+1.然后更新线段树即可。

```
public class Solution {
    /**
     * @param n: length of sequence
     * @param q: Operating frequency
     * @param a: the sequence
     * @param b: the standard of each operation
     * @return: How many numbers are subtracted by one after each operation
     */
    public static void build(int[] sum,int[] a,int l,int r,int rt)
    {
        if(l==r)
        {
            sum[rt]=a[l-1];
            return;
        }
        int m=(l+r)>>1;
        build(sum,a,l,m,rt<<1);
        build(sum,a,m+1,r,rt<<1|1);
    }
    void pushDown(int[] sum,int[] add,int rt,int ln,int rn)
    {
        if(add[rt]!=0)
```

公众号：yes的练级攻略

```
        {
            add[rt<<1]+=add[rt];
            add[rt<<1|1]+=add[rt];
            sum[rt<<1]+=add[rt]*ln;
            sum[rt<<1|1]+=add[rt]*rn;
            add[rt]=0;
        }
}
void update(int[] sum,int[] add,int L,int R,int C,int l,int r,int rt)
{
        if(L <= l && r <= R)
        {
            sum[rt]+=C*(r-l+1);
            add[rt]+=C;
            return ;
        }
        int m=(l+r)>>1;
        pushDown(sum,add,rt,m-l+1,r-m);
        if(L <= m) update(sum,add,L,R,C,l,m,rt<<1);
        if(R >    m) update(sum,add,L,R,C,m+1,r,rt<<1|1);
}
int query(int[] sum,int[] add,int L,int R,int l,int r,int rt)
{
        if(L <= l && r <= R)
        {
            return sum[rt];
        }
        int m=(l+r)>>1;
        pushDown(sum,add,rt,m-l+1,r-m);
        int ans=0;
        if(L <= m) ans+=query(sum,add,L,R,l,m,rt<<1);
        if(R >    m) ans+=query(sum,add,L,R,m+1,r,rt<<1|1);
        return ans;
}
public int[] sequenceMaintenance(int n, int q, int[] a, int[] b) {
        Arrays.sort(a);
        int[] sum=new int[n<<2];
        int[] add=new int[n<<2];
        int[] ans=new int[q];
        int cnt=0;
        build(sum,a,1,n,1);
        for (int i=0;i<q;i++){
            int l=1,r=n,pos=-1;
            while (l<=r){
```

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                int m=(l+r)>>1;
                if (query(sum,add,m,m,1,n,1)>=b[i]){
                    r=m-1;
                    pos=m;
                }else{
                    l=m+1;
                }
            }
            if (pos==-1) {
                ans[cnt++]=0;
            }
            else {
                ans[cnt++]=n-pos+1;
                update(sum,add,pos,n,-1,1,n,1);
            }
        }
        return ans;
    }
}
```

20、乘积为 B（lintcode1322）

1322. 乘积为B

给定一个长度为n的正整数序列A_i。
你每次可以花费1的代价，将序列中某一个数加一或者减一。
你希望用最小的代价使得序列所有数的乘积为B
每次操作结束后 每个数都要是正整数

样例

输入:12
[1,3,5]
输出:1
解释：可以把A[2] 减一变为4

- 预处理 先将B的所有因子求出来 存到$factor$数组
- 算法一 动态规划 方程 $dp[i][j]$表示考虑A数组前i个数修改之后的乘积为j的最小花费 边界条件 $dp[0][0]=0$ 转移 $dp[i][j]=min(dp[i-1][k]+abs(j/k-A[i]))$,即枚举前i-1个数修改后的乘积，再算出$A[i]$如何修改 算出$dp[i][j]$

时间复杂度：状态量$O(NB)$ 转移$O(B)$ 总复杂度$O(NB^2)$ 空间复杂度：$O(NB)$ N是A的长度 B是最终乘积

- 算法二 动态规划+状态优化

因为要求乘积是B 那么j应该是B的因子 例如：要求$B=8$ 若 $dp[1][5]=2$ 5不是8的因子，那么从2到n，A数组不管怎么修改都不能使得 $5x=8$ (x是A1后面的数的乘积) 那么j只需要枚举B的因子就可以了

方程 $dp[i][j]$表示考虑A数组前i个数修改之后的乘积为$factor[j]$的最小花费 边界条件 $dp[0][0]=0$ 转移
$dp[i][j]=min(dp[i-1][k]+abs(factor[j]/factor[k]-A[i]))$ $factor[j]$, 即枚举前i-1个数修改后的乘积，再算出$A[i]$如何修改 算出$dp[i][j]$

时间复杂度：状态量$O(NM)$ 转移$O(M)$ 总复杂度$O(NM^2)$ 空间复杂度：$O(NM)$ N是A的长度 M是最终乘积B的因子数量

10^5内因子数最多的为83160和98280，因子数均为128

- 算法三 动态规划+状态优化+滚动数组优化

滚动数组是DP中的一种编程思想。简单的理解就是让数组滚动起来， 每次都使用固定的几个存储空间，来达到压缩，节省存储空间的作用。起到优化空间，主要应用在递推或动态规划中（如01背包问题）。 因为DP题目是一个自底向上的扩展过程，我们常常需要用到的是连续的解，前面的解往往可以舍去。所以用滚动数组优化是很有效的。 利用滚动数组的话在N很大的情况下可以达到压缩存储的作用。 例如

禁止商用
号：
级攻略

```
    //斐波那契
    d[0]=1;
d[1]=1;
for(i=2;i<80;i++)
{
    d[i]=d[i-1]+d[i-2];
}
//改写成
d[1]=1;
d[2]=1;
for(i=2;i<80;i++)
{
    d[0]=d[1];
    d[1]=d[2];
    d[2]=d[0]+d[1];
}

int i, j, d[100][100];
    for(i = 1; i < 100; i++)
    for(j = 0; j < 100; j++)
            d[i][j] = d[i - 1][j] + d[i][j - 1];

//上面的d[i][j]只依赖于d[i - 1][j], d[i][j - 1];

//运用滚动数组

    int i, j, d[2][100];
    for(i = 1; i < 100; i++)
    for(j = 0; j < 100; j++)
            d[i % 2][j] = d[(i - 1) % 2][j] + d[i % 2][j - 1];
//i%2 只有0和1 两种结果，那么我们可以用d[0]去更新d[1] 再d[1]更新d[0]
//这样数组交替滚动使用，来降低空间复杂度
```

对于本题，我们只用第i-1次的dp答案去更新第i次的tmp的答案,再把tmp赋值给dp，去做第i+1次更新就好了

```
public class Solution {
    /**
     * @param B: the all Ai product equal to B
     * @param A: the positive int array
     * @return: return the minium cost
     */
    public int getMinCost(int B, int[] A) {
        // write your code here
        //A 的长度
        int n = A.length;

        //存 B 的因数的数组
        int []factor = new int[200];

        // B 因子的数量
        int m = 0;
        //找出 B 的因子 并存入 factor
        for(int i = 1; i <= B; i++) {
            if(B % i == 0) {
                factor[m++] = i;
            }
```

公众号：yes的练级攻略

```
        }

        //初始化 dp 数组
        int []dp = new int[m];
        for (int i = 0; i < m; i++) {
            dp[i] = Integer.MAX_VALUE;
        }

        //初始化 dp 数组
        dp[0] = 0;

        for(int i = 0; i < n; i++) {
            //初始化本次循环中，被更新的 tmp 数组
            //用 dp 数组去更新 tmp 数组
            int []tmp = new int[m];
            for (int j = 0; j < m; j++) {
                tmp[j] = Integer.MAX_VALUE;
            }
            for(int j = 0; j < m; j++) {
                //dp[j]为无穷，无法去更新
                if(dp[j] == Integer.MAX_VALUE) {
                    continue;
                }
                //枚举 factor[j]的倍数，将前 i 个数的乘积由 factor[j]变换成 factor[k]
                for(int k = j; k < m; k++) {
                    //倍数关系约束
                    if(factor[k] % factor[j] != 0) {
                        continue;
                    }
                    //dp 方程更新
                    tmp[k] = Math.min(tmp[k], dp[j] + Math.abs(factor[k] / factor[j] - A[i]));
                }
            }

            //本次循环的 tmp 数组去更新的下次的 tmp 数组
            dp = tmp;
        }
        return dp[m - 1];
    }
}
```

21、除自身以外数组的乘积（238）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

238. 除自身以外数组的乘积

难度 中等 537 收藏 分享 切换为英文 关注 反馈

给你一个长度为 n 的整数数组 `nums`，其中 $n > 1$，返回输出数组 `output` ，其中 `output[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

示例:

```
输入: [1,2,3,4]
输出: [24,12,8,6]
```

提示：题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

说明: 请**不要使用除法**，且在 O(n) 时间复杂度内完成此题。

进阶：
你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组**不被视为**额外空间。）

https://leetcode-cn.com/problems/product-of-array-except-self/solution/product-of-array-except-self-shang-san-jiao-xia-sa/

解题思路：

- 因为空间复杂度要求 $O(1)$，不能使用 `除法` ，因此一定需要在 `乘法` 过程中得到所有答案；
- 我们可以将 `res` 数组列成乘积形式，不同的 n 组成每行内容，形成一个矩阵，可以发现矩阵 `主对角线` 全部为 1（当前数字不相乘，等价为乘 1）；
- 因此，我们分别计算矩阵的 `下三角` 和 `上三角` ，并且在计算过程中储存过程值，最终可以在遍历 2 遍 `nums` 下完成结果计算。

res					
res[0] =	1	num[1]	...	num[n-2]	num[n-1]
res[1] =	num[0]	1	...	num[n-2]	num[n-1]
...	...	...	...	num[n-2]	num[n-1]
res[n-2] =	num[0]	num[1]	...	1	num[n-1]
res[n-1] =	num[0]	num[1]	...	num[n-2]	1

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int[] res = new int[nums.length];
        int p = 1, q = 1;
//下三角中自上而下对每一行做连乘
        for (int i = 0; i < nums.length; i++) {
            res[i] = p;
            p *= nums[i];
        }
//上三角自下而上的连乘
        for (int i = nums.length - 1; i > 0 ; i--) {
            q *= nums[i];
            res[i - 1] *= q;
        }
        return res;
    }
}
```

22、构建乘积数组（剑指 66）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 66. 构建乘积数组

难度 简单 35 收藏 分享 切换为英文 关注 反馈

给定一个数组 `A[0, 1, …, n-1]`，请构建一个数组 `B[0, 1, …, n-1]`，其中 `B` 中的元素 `B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]`。不能使用除法。

示例:

```
输入: [1,2,3,4,5]
输出: [120,60,40,30,24]
```

提示:

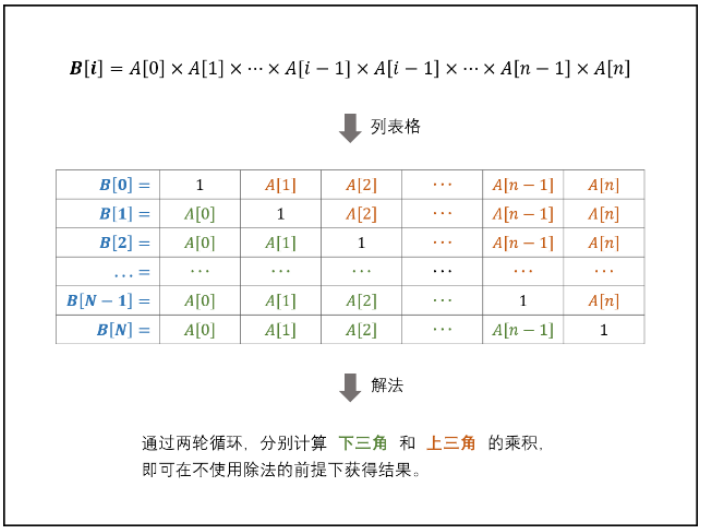
- 所有元素乘积之和不会溢出 32 位整数
- `a.length <= 100000`

https://leetcode-cn.com/problems/gou-jian-cheng-ji-shu-zu-lcof/solution/mian-shi-ti-66-gou-jian-cheng-ji-shu-zu-biao-ge-fe/

解题思路:

本题的难点在于 **不能使用除法**，即需要 **只用乘法** 生成数组 B。根据题目对 $B[i]$ 的定义，可列表格，如下图所示。

根据表格的主对角线（全为 1），可将表格分为 **上三角** 和 **下三角** 两部分。分别迭代计算下三角和上三角两部分的乘积，即可 **不使用除法** 就获得结果。



算法流程:

1. 初始化：数组 B，其中 $B[0] = 1$；辅助变量 $tmp = 1$；
2. 计算 $B[i]$ 的 **下三角** 各元素的乘积，直接乘入 $B[i]$；
3. 计算 $B[i]$ 的 **上三角** 各元素的乘积，记为 tmp，并乘入 $B[i]$；
4. 返回 B。

复杂度分析:

- **时间复杂度** $O(N)$：其中 N 为数组长度，两轮遍历数组 a，使用 $O(N)$ 时间。
- **空间复杂度** $O(1)$：变量 tmp 使用常数大小额外空间（数组 b 作为返回值，不计入复杂度考虑）。

更多硬核内容就在公众号「yes 的练级攻略」

```java
class Solution {
    public int[] constructArr(int[] a) {
        int[] res = new int[a.length];
        if (a.length == 0) return res;
        res[0] = 1;
        int tmp = 1;
        for (int i = 1; i < a.length; i++){
            res[i] = res[i-1] * a[i-1];
        }
        for (int i = a.length - 2; i >= 0; i--){
            tmp *= a[i+1];
            res[i] *= tmp;
        }
        return res;
    }
}
```

86、乘积最大子数组（152）【找到一个乘积最大的子数组】

152. 乘积最大子数组

难度 中等 677 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 nums ，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

示例 1:

输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。

示例 2:

输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。

https://leetcode-cn.com/problems/maximum-product-subarray/solution/dong-tai-gui-hua-li-jie-wu-hou-xiao-xing-by-liweiw/

更多硬核内容就在公众号「yes 的练级攻略」

思路

- 这个问题很像「力扣」第 53 题：最大子序和，只不过当前这个问题求的是乘积的最大值；
- 「连续」这个概念很重要，可以参考第 53 题的状态设计，将状态设计为：**以 `nums[i]` 结尾的连续子数组的最大值**；
- 类似状态设计的问题还有「力扣」第 300 题：最长上升子序列，「子数组」、「子序列」问题的状态设计的特点是：**以 `nums[i]` 结尾**，这是一个经验，可以简化讨论。

> 提示：**以 `nums[i]` 结尾**这件事情很重要，贯穿整个解题过程始终，请大家留意。

分析与第 53 题的差异

- 求乘积的最大值，示例中负数的出现，告诉我们这题和 53 题不一样了，一个正数乘以负数就变成负数，即：**最大值乘以负数就变成了最小值**；
- 因此：**最大值和最小值是相互转换的，这一点提示我们可以把这种转换关系设计到「状态转移方程」里去**；
- 如何解决这个问题呢？这里常见的技巧是在「状态设计」的时候，在原始的状态设计后面多加一个维度，减少分类讨论，降低解决问题的难度。

这里是百度百科的「无后效性」词条的解释：

> 无后效性是指如果在某个阶段上过程的状态已知，则从此阶段以后过程的发展变化仅与此阶段的状态有关，而与过程在此阶段以前的阶段所经历过的状态无关。利用动态规划方法求解多阶段决策过程问题，过程的状态必须具备无后效性。

再翻译一下就是：「动态规划」通常不关心过程，只关心「阶段结果」，这个「阶段结果」就是我们设计的「状态」。什么算法关心过程呢？「回溯算法」，「回溯算法」需要记录过程，复杂度通常较高。

而将状态定义得更具体，通常来说对于一个问题的解决是满足「无后效性」的。这一点的叙述很理论化，不熟悉朋友可以通过多做相关的问题来理解「无后效性」这个概念。

第 1 步：状态设计（特别重要）

- `dp[i][j]`：以 `nums[i]` 结尾的连续子数组的最值，计算最大值还是最小值由 `j` 来表示，`j` 就两个值；
  - 当 `j = 0` 的时候，表示计算的是最小值；
  - 当 `j = 1` 的时候，表示计算的是最大值。

这样一来，状态转移方程就容易写出。

第 2 步：推导状态转移方程（特别重要）

- 由于状态的设计 `nums[i]` 必须被选取（请大家体会这一点，这一点恰恰好也是使得子数组、子序列问题更加简单的原因：当情况复杂、分类讨论比较多的时候，需要固定一些量，以简化计算）；
- `nums[i]` 的正负和之前的状态值（正负）就产生了联系，由此关系写出状态转移方程：
  - 当 `nums[i] > 0` 时，由于是乘积关系：
    - 最大值乘以正数依然是最大值；
    - 最小值乘以**同一个**正数依然是最小值；
  - 当 `nums[i] < 0` 时，依然是由于乘积关系：
    - 最大值乘以负数变成了最小值；
    - 最小值乘以**同一个**负数变成最大值；
  - 当 `nums[i] = 0` 的时候，由于 `nums[i]` 必须被选取，最大值和最小值都变成 0，合并到上面任意一种情况均成立。

更多硬核内容就在公众号「yes 的练级攻略」

- 但是，还要注意一点，之前状态值的正负也要考虑：例如，在考虑最大值的时候，当 `nums[i] > 0` 是，如果 `dp[i - 1][1] < 0`（之前的状态最大值），此时 `nums[i]` 可以另起炉灶（这里依然是第 53 题的思想），此时 `dp[i][1] = nums[i]`，合起来写就是：

```
dp[i][1] = max(nums[i], nums[i] * dp[i - 1][1]) if nums[i] >= 0
```

其它三种情况可以类似写出，状态转移方程如下：

```
dp[i][0] = min(nums[i], nums[i] * dp[i - 1][0]) if nums[i] >= 0
dp[i][1] = max(nums[i], nums[i] * dp[i - 1][1]) if nums[i] >= 0

dp[i][0] = min(nums[i], nums[i] * dp[i - 1][1]) if nums[i] < 0
dp[i][1] = max(nums[i], nums[i] * dp[i - 1][0]) if nums[i] < 0
```

第 3 步：考虑初始化

由于 `nums[i]` 必须被选取，那么 `dp[i][0] = nums[0]`，`dp[i][1] = nums[0]`。

第 4 步：考虑输出

题目问连续子数组的乘积最大值，这些值需要遍历 `dp[i][1]` 获得。

复杂度分析：

- 时间复杂度：$O(N)$，这里 N 是数组的长度，遍历 2 次数组；
- 空间复杂度：$O(N)$，状态数组的长度为 $2N$。

方法 1：二维滚动数组

```java
class Solution {
    public int maxProduct(int[] nums) {
        if (nums.length == 0) return 0;
        int n = nums.length;
        int[][] dp = new int[n][2];
        dp[0][0] = nums[0]; //dp[i][0]计算的是最小值
        dp[0][1] = nums[0]; //dp[i][1]计算的是最大值
        for (int i = 1; i < n; i++){
            if (nums[i] >= 0) {
                dp[i][0] = Math.min(nums[i], nums[i] * dp[i - 1][0]);
                dp[i][1] = Math.max(nums[i], nums[i] * dp[i - 1][1]);
            } else{
                dp[i][0] = Math.min(nums[i], nums[i] * dp[i - 1][1]);
                dp[i][1] = Math.max(nums[i], nums[i] * dp[i - 1][0]);
            }
        }
        int res = dp[0][1];
        for (int i = 1; i < n; i++){
            res = Math.max(res, dp[i][1]);
        }
        return res;
    }
```

更多硬核内容就在公众号「yes 的练级攻略」

}

第 5 步：考虑表格复用

- 动态规划问题，基于「自底向上」、「空间换时间」的思想，通常是「填表格」，本题也不例外；
- 由于通常只关心最后一个状态值，或者在状态转移的时候，当前值只参考了上一行的值，因此在填表的过程中，表格可以复用，常用的技巧有：
  - 1、滚动数组（当前行只参考了上一行的时候，可以只用 2 行表格完成全部的计算）；
  - 2、滚动变量（斐波拉契数列问题）。
- 掌握非常重要的「表格复用」技巧，来自「0-1 背包问题」（弄清楚为什么要倒序填表）和「完全背包问题」（弄清楚为什么可以正向填表）；
- 「表格复用」的合理性，**只由「状态转移方程」决定，即当前状态值只参考了哪些部分的值。**

复杂度分析：

- 时间复杂度：$O(N)$，这里 N 是数组的长度，最值也在一次遍历的过程中计算了出来；
- 空间复杂度：$O(1)$，只使用了常数变量。

这里说一点题外话：除了基础的「0-1」背包问题和「完全背包」问题，需要掌握「表格复用」的技巧以外。在绝大多数情况下，在「力扣」上做的「动态规划」问题都可以不考虑「表格复用」。

做题通常可以不先考虑优化空间（个人观点，仅供参考），理由如下：

- 空间通常来说是用户不敏感的，并且在绝大多数情况下，空间成本低，我们写程序通常需要**优先考虑时间复杂度最优**；
- 时间复杂度和空间复杂度通常来说不可能同时最优，所以我们经常看到的是优化解法思路都是「空间换时间」，这一点几乎贯穿了基础算法领域的绝大多数的算法设计思想；
- 限制空间的思路，通常来说比较难，一般是在优化的过程中才考虑优化空间，在一些限制答题时间的场景下（例如面试），先写出一版正确的代码是更重要的，并且不优化空间的代码一般来说，可读性和可解释性更强。

总结

动态规划问题通常用于计算多阶段决策问题的最优解。

- 多阶段，是指解决一个问题有多个步骤；
- 最优解，是指「最优子结构」。

动态规划有三个概念很重要：

- 重复子问题：因为重复计算，所以需要「空间换时间」，记录子问题的最优解；
- 最优子结构：规模较大的问题的最优解，由各个子问题的最优解得到；
- 无后效性（上面已经解释）。

动态规划有两个特别关键的步骤：

- 设计状态：
  - 有些题目问啥，就设计成什么；
  - 如果不行，只要有利于状态转移，很多时候，就可以设计成状态；
  - 根据过往经验；
  - 还有一部分问题是需要在思考的过程中调整的，例如本题。
- 推导状态转移方程：通常是由问题本身决定的。

动态规划问题思考的两个方向：

- 自顶向下：即「递归 + 记忆化」，入门的时候优先考虑这样做；
- 自底向上：即「递归」，从一个最小的问题开始，逐步得到最终规模问题的解。后面问题见得多了，优先考虑这样做，绝大部分动态规划问题可以「自底向上」通过递推得到。

方法 2：表格复用（常量级别）

```
class Solution {
    public int maxProduct(int[] nums) {
        if (nums.length == 0) return 0;
```

```
        int n = nums.length;
        int preMin = nums[0];
        int preMax = nums[0];
        int curMin = 0, curMax = 0;
        int res = nums[0];
        for (int i = 1; i < n; i++){
            if (nums[i] >= 0) {
                curMin = Math.min(nums[i], nums[i] * preMin);
                curMax = Math.max(nums[i], nums[i] * preMax);
            } else{
                curMin = Math.min(nums[i], nums[i] * preMax);
                curMax = Math.max(nums[i], nums[i] * preMin);
            }
            preMin = curMin;
            preMax = curMax;
            res = Math.max(res, curMax);
        }
        return res;
    }
}
```

23、将数组重新排序以构造最小值（lintcode379）

379. 将数组重新排序以构造最小值

给定一个整数数组，请将其重新排序，以构造最小值。

样例
样例 1:

```
输入：[2, 1]
输出：[1, 2]
解释：通过将数组重新排序，可构造 2 个可能性数字：
          1+2=12
          2+1=21
其中，最小值为 12，所以，将数组重新排序后，该数组变为 [1, 2]。
```

样例 2:

```
输入：[3, 32, 321]
输出：[321, 32, 3]
解释：通过将数组重新排序，可构造 6 个可能性数字：
        3+32+321=332321
        3+321+32=332132
        32+3+321=323321
        32+321+3=323213
        321+3+32=321332
        321+32+3=321323
其中，最小值为 321323，所以，将数组重新排序后，该数组变为 [321, 32, 3]。
```

挑战
在原数组上完成，不使用额外空间。

注意事项
因为答案非常大，所以你需要返回一个字符串而非一个整数。

```
public class Solution {
    /**
     * @param nums n non-negative integer array
     * @return a string
     */
    public String minNumber(int[] nums) {
```

```
        // Write your code here
        int n = nums.length;
        if (n < 1) return "";

        String[] strs = new String[n];
        for (int i = 0; i < n; i++) {
            strs[i] = String.valueOf(nums[i]);
        }

        Arrays.sort(strs, new Cmp());

        String ans = "";
        for (int i = n - 1; i >= 0; i--) {
            ans = ans.concat(strs[i]);
        }

        int i = 0;
        while (i < n && ans.charAt(i) == '0')
            i ++;

        if (i == n) return "0";
        return ans.substring(i);
    }
}
class Cmp implements Comparator<String>{
    @Override
    public int compare(String a, String b) {
        String ab = a.concat(b);
        String ba = b.concat(a);
        return ba.compareTo(ab);
    }
}
```

24、最大数（179）

禁止商用
公众号：
yes的练级攻略

179. 最大数

难度 中等 368 收藏 分享 切换为英文 关注

给定一组非负整数，重新排列它们的顺序使之组成一个最大的整数。

示例 1:

```
输入: [10,2]
输出: 210
```

示例 2:

```
输入: [3,30,34,5,9]
输出: 9534330
```

说明: 输出结果可能非常大，所以你需要返回一个字符串而不是整数。

```java
class Solution {
    public String largestNumber(int[] nums) {
        int n = nums.length;
        if (n < 1) return "";
        String[] strs = new String[n];
        for (int i = 0; i < n; i++){
            strs[i] = String.valueOf(nums[i]);
        }
        Arrays.sort(strs, new Cmp());
        String ans = "";
        for (int i = 0; i < n; i++) { //与上一题唯一的差别
            ans = ans.concat(strs[i]);
        }
        int i = 0;
        while (i < n && ans.charAt(i) == '0') i++;
        if (i == n) return "0";
        return ans.substring(i);
    }
    class Cmp implements Comparator<String> {
        @Override
        public int compare(String a, String b) {
            String ab = a.concat(b);
            String ba = b.concat(a);
            return ba.compareTo(ab);
        }
    }
}
```

25、拼接最大数（321）

更多硬核内容就在公众号「yes 的练级攻略」

321. 拼接最大数

难度 困难 120 收藏 分享 切换为英文 关注 反馈

给定长度分别为 m 和 n 的两个数组，其元素由 0-9 构成，表示两个自然数各位上的数字。现在从这两个数组中选出 k (k <= m + n) 个数字拼接成一个新的数，要求从同一个数组中取出的数字保持其在原数组中的相对顺序。

求满足该条件的最大数。结果返回一个表示该最大数的长度为 k 的数组。

说明: 请尽可能地优化你算法的时间和空间复杂度。

示例 1:

```
输入:
nums1 = [3, 4, 6, 5]
nums2 = [9, 1, 2, 5, 8, 3]
k = 5
输出:
[9, 8, 6, 5, 3]
```

示例 2:

```
输入:
nums1 = [6, 7]
nums2 = [6, 0, 4]
k = 5
输出:
[6, 7, 6, 0, 4]
```

示例 3:

```
输入:
nums1 = [3, 9]
nums2 = [8, 9]
k = 3
输出:
[9, 8, 9]
```

https://leetcode-cn.com/problems/create-maximum-number/solution/qiu-jie-zi-wen-ti-zai-he-bing-zi-wen-ti-de-jie-by-/

```
public class Solution {
    /**
     * 假设数组一为[3,4,6,5]、数组二为[9,1,2,5,8,3]、k = 5;
     * 组合情况有0 + 5、1 + 4、2 + 3、3 + 2、4 + 1 五种情况,就是从此五种情况取出组合最大的一种;
     * 表示若数组二的元素个数>=k,则数组一的元素个数可以从0 开始取,否则在数组二的大小基础上补。
     * @param nums1
     * @param nums2
     * @param k
     * @return
     */
    public int[] maxNumber(int[] nums1, int[] nums2, int k) {
        int len1=nums1.length,len2=nums2.length;
        int[] res=new int[k];
        for (int arr1_num_select=Math.max(0,k-len2);arr1_num_select<=len1 && arr1_num_select<=k;arr1_num_select++)        {
```

```
            int[] arr=mergeMaxArr(maxArr(nums1,arr1_num_select),maxArr(nums2,k-arr1_num_select),k);
            if(compare(arr,res,k))
                res=arr;
        }
        return res;
    }
    /**
     * 从 num 数组中找 pos 个数（按相对顺序）组建成新数组，使该数组构成的数最大
     * 维护一个单调递减栈：栈大小为目标个数的长度
     * 记录栈中的元素、外面还未遍历的元素的个数，来决定是否按单调递减栈的规则让小的出栈，大的进栈
     * 这里要注意，小于当前待入栈的栈中元素的个数要记录，因为有可能把全部出栈完了再入栈，而后面剩余还没被的元素再
     * 也填补不满栈，所以要出栈的个数受剩余未入栈的个数的限制，出栈到外面还未遍历的元素个数刚好能补充栈中空缺的位，
     * 后面的元素直接按顺序进栈
     * 为了方便记录栈中比当前要入栈的元素小的元素个数，使用数组来方便遍历，因为栈只能访问栈顶元素，要访问栈顶下的元素必须
     * 要出栈，比较麻烦。
     * @param num
     * @param pos
     * @return
     */
    private int[] maxArr(int[] num,int pos){
        int max_len=num.length;
        if (max_len==pos)
            return num;
        int[] res=new int[pos];
        int res_top_index=-1; //模拟栈顶指针，指向下一个 res 待填补的位置,res_top_index+1 表示栈中现在已有的元素个数
        //max_len-cur_to_In 得到当前还未进行遍历出栈入栈的元素个数：例如
        //num[]={1,2,8,9,10,2}，cur_to_In=2，那么 6-2=4，表示包括当前正在访问的元素 8，还未进行出栈入栈的元素个数
        //那么这些元素一定要能填满栈，就要保证 res_top_index+1+max_len-cur_to_In>k，此时可以利用
        // 单调递减栈的规则来进出栈，直到该式=k 就一直加入栈即可，所以可以利用这个来设计一个 while 循环
        //但是如果此时栈是空的，也就是 res_top_index=-1，那也是直接进栈即可
        for (int cur_to_In=0;cur_to_In<max_len;cur_to_In++){
            //一个元素进栈前，先运用规则让比他小的元素出栈，直到满足一定的关系
            while (res_top_index+1>0 && num[cur_to_In]>res[res_top_index] && res_top_index+1+max_len-cur_to_In>pos)
                res_top_index--;  //回退表示出栈
```

```
            if(res_top_index<pos-1) //元素进栈，栈还没满，塞入下标为res_top_index+1的位置
                res[++res_top_index]=num[cur_to_In];
        }
        return res;
    }

    /**
     * 假设数组一最大组合为[6,5],数组二最大组合为[9,8,3],（数组不一定是递减的，也就是栈不一定是单调减的，后面的几个元素是直接加入的）
     * 进行双指针排序,排序后为[9,8,6,5,3]
     * pos=len1+len2=5
     * 与归并类似，但是这个排序后的结果不改变元素在原数组的相对顺序，
     * 使用双指针遍历两个数组，指针指向的元素大的填到新数组，然后右移动，继续比较
     * 这里要注意一个数组先遍历到最后的时候，
     * int[] num={2,5,6,4,4,0};
     * int[] num2={7,3,8,0,6,5,7,6,2};
     * 此时p1走到了最后一个指向0的地方，p2也指向0，那么下一位应该先取p2的0，因为0657620>0065762
     * 为什么选择p2指向的0先呢？我们发现是因为p2的下一位6大于了p1指向的0，所以p2优先级更高，也就是此时p1==len1-1，p2=某个值
     * p1指向数组短的那方
     * int[] num={2,5,6,4,4,3};
     * int[] num2={7,3,8,3,2,5,7,6,2};
     * @param maxArr2
     * @param maxArr1
     * @return
     */
    private int[] mergeMaxArr(int[] maxArr1,int[] maxArr2,int pos){
        int p1=0,p2=0;
        int[] res=new int[pos];
        int index=0;
        while (index<pos){
            if (compare(maxArr1,maxArr2,p1,p2))
                res[index++]=maxArr1[p1++];
            else res[index++]=maxArr2[p2++];
        }
        return res;
    }

    /**
     * 获取结果组合中最大的
     * 例如
     * 6,7,6,0,4
```

```
     * 6,5,1,3,4
     * @param arr1
     * @param arr2
     * @return
     */
    private boolean compare(int[] arr1,int[] arr2,int k){
        //也是双指针进行比较即可，对位遍历
        int p=0;
        while (p<k&&arr1[p]==arr2[p])
            p++;
        return p==k||arr1[p]>=arr2[p];
    }

    /**
     * 如果遍历的过程中遇到相等的情况，我们就需要一直往后比，直到它们不相等分出大小为止
     * 此函数可与前面的compare 重载进行合并
     * @param arr1
     * @param arr2
     * @param p1
     * @param p2
     * @return
     */
    private boolean compare(int[] arr1,int[] arr2,int p1,int p2){

        int len1=arr1.length,len2=arr2.length;
        while (p1<len1&&p2<len2&&arr1[p1]==arr2[p2]){
            p1++;p2++;
        }
        //此时p1,p2 是否走过末尾，
        // 假设p2 走过了末尾，那么也就是如
        //811113 和811 的情况，最大合并是8(1111)3(11)，先拿前面的1，先走过末尾的后面
拿
        //p1，p2 都没走过末尾，有下面两种情况，[]内的数字为此时p1 和p2 走到的数字
        //81[3]和711[8]0，最大合并是87(118)(13)(0),先拿后面的1，因为3<8
        //81[3]和711[2]，最大合并是87(1)(3)(11)(2)，先拿前面的1，因为3>2
        //也就是大的先拿
        return p2==len2||(p1<len1&&arr1[p1]>arr2[p2]);
    }
}
```

26、最小区间（632）

更多硬核内容就在公众号「yes 的练级攻略」

632. 最小区间

难度 困难 131 收藏 分享 切换为英文 关注 反馈

你有 `k` 个升序排列的整数数组。找到一个**最小**区间，使得 `k` 个列表中的每个列表至少有一个数包含在其中。

我们定义如果 `b-a < d-c` 或者在 `b-a == d-c` 时 `a < c`，则区间 [a,b] 比 [c,d] 小。

示例 1:

```
输入:[[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]
输出: [20,24]
解释:
列表 1：[4, 10, 15, 24, 26]，24 在区间 [20,24] 中。
列表 2：[0, 9, 12, 20]，20 在区间 [20,24] 中。
列表 3：[5, 18, 22, 30]，22 在区间 [20,24] 中。
```

注意:

1. 给定的列表可能包含重复元素，所以在这里升序表示 >= 。
2. 1 <= `k` <= 3500
3. -10^5 <= 元素的值 <= 10^5
4. **对于使用Java的用户，请注意传入类型已修改为List<List<Integer>>。重置代码模板后可以看到这项改动。**

https://leetcode-cn.com/problems/smallest-range-covering-elements-from-k-lists/solution/tan-xin-duo-zhi-zhen-you-xian-ji-dui-lie-by-girapa/

解题思路

首先，我们想一下这样一个问题：题目要求是寻找最小区间，最小区间满足下面两点：

- 长度最小(首要)
- 长度相同时起点最小

长度最小的区间必然是需要我们在计算中找到的，但是起点最小的区间我们是可以知道的。就是从每个区间中找最小的元素，组成的新的区间，我们称其为起始区间。如果之后没有长度比起始区间长度更短的，那么起始区间就是我们所求的最小区间，因为起始区间的起点是最小的。

那么我们所需要做的就是搜索是否有比起始区间还要短的区间了。那么该如何搜索呢?
答案是**每次都将当前区间中最小的元素丢弃，换成其原始数组中的下一个元素**。就是说，如果当前我们从每个区间中选取的元素分别是a1_1,a2_1,a3_1...ak_1(前面的数字代表来自第几个数组，后面的数字表示该数是该数组的第几个元素),若此时最小的元素是ai_1,最大元素是aj_1,那么区间就为[ai_1, aj_1],区间中最小元素是ai_1,那么我们就将ai_1丢弃,将ai_2拿出来放进去。

原因：如果不换ai_1,那么区间的起点就一直是ai_1,而且区间长度不可能缩小，区间长度取决于起点和终点，而终点是不可能变小的。因为我们是从小到大进行元素的选取，我们每次丢弃一个元素，就要选择它在原数组中的下一位，而原数组是升序排列的。故终点不可能减小。
那么，我们只能够通过让起点增大来是区间缩小，即每次丢弃最小的元素，换成它原数组的下一个元素。然后再计算当前所选区间的长度，如果小于之前的区间长度就更新即可。

结束条件：当当前最小元素是其原来整数数组的最后一个元素时，就是结束的时候。因为之后的操作不可能更改起点了，只会让终点变大，即区间变长。

方法 1：贪心+多指针+优先队列

```
class Solution {
```

```
    class NumGroup{
        public NumGroup(int num, int grp){
            this.num = num;
            this.grp = grp;
        }
        int num; //数值
        int grp; //组号
    }
    public int[] smallestRange(List<List<Integer>> nums) {
        //因为每次都要找最小元素，所以维护一个最小堆比较合适
        PriorityQueue<NumGroup> numgrp = new PriorityQueue<>(new Compar
ator<NumGroup>(){
            @Override
            public int compare(NumGroup n1, NumGroup n2){
                return n1.num - n2.num;
            }
        });

        int end = Integer.MIN_VALUE;
        //记录每个数组当前的指针位置，一开始都指向第 0 个元素，即每个区间的最小
元素
        int[] index = new int[nums.size()];

        //起始区间
        for(int i = 0; i < nums.size(); i++){
            if(nums.get(i).get(0) > end) end = nums.get(i).get(0);
            NumGroup num = new NumGroup(nums.get(i).get(0), i);
            numgrp.offer(num);
        }

        int max = end;
        int start = numgrp.peek().num;
        int min = start;
        int len = end - start + 1;

        while(true){
            //grp 为当前最小元素的原数组号
            int grp = numgrp.poll().grp;
            //如果当前最小元素已经是原数组最大元素了，则退出
            if(index[grp]+1 == nums.get(grp).size()) break;

            //索引++
            index[grp]++;
           NumGroup n =new NumGroup(nums.get(grp).get(index[grp]),grp);
```

```
            numgrp.offer(n);
            //当前最大值
            if(n.num > max) max = n.num;
            min = numgrp.peek().num;
            //长度变小
            if(max-min+1 < len){
                start = min;
                end = max;
                len = max-min+1;
            }
        }

        return new int[]{start, end};
    }
}
```

https://leetcode-cn.com/problems/smallest-range-covering-elements-from-k-lists/solution/hua-dong-chuang-kou-jie-fa-te-bie-rong-yi-li-jie-b/

解题思路

首先把所有的数字和该数字所在的第k个list凑成一个大的list,这个list里面存放的是新构造的Node类,node.pos指的是该元素原本在第几个list里面,node.val就是该元素值,map[k]指的是在滑动窗口[i,j]里面,第k个小list里面的元素出现过几次,count指的是,在滑动窗口[i,j]里面,出现的小list的种类个数,比如如果数据中有3500个小list,count=3499就代表滑动窗口中出现了3499个list,还差一个就圆满了^_^,len就是迭代到当前位置时的最终res范围的长度,其他的不是太难理解

方法 2：滑动窗口

```
class Solution {
    public int[] smallestRange(List<List<Integer>> nums) {
        int[] res = new int[2];
        int len = Integer.MAX_VALUE;
        int[] map = new int[nums.size()];
        int count = 0;

        int j = 0;
        List<Node> list = new ArrayList<>();
        for (int i = 0; i < nums.size(); i++) {
            for (int x : nums.get(i)) {
                list.add(new Node(x, i));
            }
        }
        list.sort(new Comparator<Node>() {
            @Override
            public int compare(Node o1, Node o2) {
                return o1.val - o2.val;
            }
        });
```

```
        for (int i = 0; i < list.size(); i++) {
            Node node = list.get(i);
            map[node.pos]++;
            if (map[node.pos] == 1) {
                count++;
            }
            if (count == nums.size()) {
                for (; j < list.size(); j++) {//move the left side
                    if (map[list.get(j).pos] == 1) {

                        if (node.val - list.get(j).val < len) {
                            len = node.val - list.get(j).val;
                            res[0] = list.get(j).val;
                            res[1] = node.val;
                        }
                        count--;
                        map[list.get(j).pos] = 0;
                        j++;
                        break;
                    } else {
                        map[list.get(j).pos]--;
                    }
                }
            }
        }
        return res;
    }
}
class Node {
    int val;
    int pos;

    public Node(int val, int pos) {
        this.val = val;
        this.pos = pos;
    }
}
```

https://leetcode-cn.com/problems/smallest-range-covering-elements-from-k-lists/solution/zui-xiao-qu-jian-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：堆

给定 k 个列表，需要找到最小区间，使得每个列表都至少有一个数在该区间中。该问题可以转化为，从 k 个列表中各取一个数，使得这 k 个数中的最大值与最小值的差最小。

假设这 k 个数中的最小值是第 i 个列表中的 x，对于任意 $j \neq i$，设第 j 个列表中被选为 k 个数之一的数是 y，则为了找到最小区间，y 应该取第 j 个列表中大于等于 x 的最小的数。简单证明如下：假设 z 也是第 j 个列表中的数，且 $z > y$，则有 $z - x > y - x$，同时包含 x 和 z 的区间一定不会小于同时包含 x 和 y 的区间。因此，其余 $k - 1$ 个列表中应该取大于等于 x 的最小的数。

由于 k 个列表都是升序排列的，因此对每个列表维护一个指针，通过指针得到列表中的元素，指针右移之后指向的元素一定大于或等于之前的元素。

使用最小堆维护 k 个指针指向的元素中的最小值，同时维护堆中元素的最大值。初始时，k 个指针都指向下标 0，最大元素即为所有列表的下标 0 位置的元素中的最大值。每次从堆中取出最小值，根据最大值和最小值计算当前区间，如果当前区间小于最小区间则用当前区间更新最小区间，然后将对应列表的指针右移，将新元素加入堆中，并更新堆中元素的最大值。

如果一个列表的指针超出该列表的下标范围，则说明该列表中的所有元素都被遍历过，堆中不会再有该列表中的元素，因此退出循环。

复杂度分析

- 时间复杂度：$O(nk \log k)$，其中 n 是所有列表的平均长度，k 是列表数量。所有的指针移动的总次数最多是 nk 次，每次从堆中取出元素和添加元素都需要更新堆，时间复杂度是 $O(\log k)$，因此总时间复杂度是 $O(nk \log k)$。
- 空间复杂度：$O(k)$，其中 k 是列表数量。空间复杂度取决于堆的大小，堆中维护 k 个元素。

方法 3：堆

```
class Solution {
    public int[] smallestRange(List<List<Integer>> nums) {
        int rangeLeft = 0, rangeRight = Integer.MAX_VALUE;
        int minRange = rangeRight - rangeLeft;
        int max = Integer.MIN_VALUE;
        int size = nums.size();
        int[] next = new int[size];
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<Intege
r>(new Comparator<Integer>() {
            public int compare(Integer index1, Integer index2) {
                return nums.get(index1).get(next[index1]) - nums.get(in
dex2).get(next[index2]);
            }
        });
        for (int i = 0; i < size; i++) {
            priorityQueue.offer(i);
            max = Math.max(max, nums.get(i).get(0));
        }
        while (true) {
            int minIndex = priorityQueue.poll();
           int curRange = max - nums.get(minIndex).get(next[minIndex]);
            if (curRange < minRange) {
                minRange = curRange;
                rangeLeft = nums.get(minIndex).get(next[minIndex]);
```

```
                rangeRight = max;
            }
            next[minIndex]++;
            if (next[minIndex] == nums.get(minIndex).size()) {
                break;
            }
            priorityQueue.offer(minIndex);
           max = Math.max(max, nums.get(minIndex).get(next[minIndex]));
        }
        return new int[]{rangeLeft, rangeRight};
    }
}
```

27、判断能否成为等差数列（1502）

1502. 判断能否形成等差数列

难度 简单 2 收藏 分享 切换为英文 关注 反馈

给你一个数字数组 `arr` 。

如果一个数列中，任意相邻两项的差总等于同一个常数，那么这个数列就称为 **等差数列** 。

如果可以重新排列数组形成等差数列，请返回 `true` ；否则，返回 `false` 。

示例 1：

输入： arr = [3,5,1]
输出： true
解释： 对数组重新排序得到 [1,3,5] 或者 [5,3,1] ，任意相邻两项的差分别为 2 或 -2 ，可以形成等差数列。

示例 2：

输入： arr = [1,2,4]
输出： false
解释： 无法通过重新排序得到等差数列。

略

提示：

- `2 <= arr.length <= 1000`
- `-10^6 <= arr[i] <= 10^6`

https://leetcode-cn.com/problems/can-make-arithmetic-progression-from-sequence/solution/qiao-yong-pai-xu-by-supperbug/

方法 1：使用内置函数

```
class Solution {
    public boolean canMakeArithmeticProgression(int[] arr) {
        Arrays.sort(arr);//先对数组进行排序，默认是升序
        int temp = arr[1] - arr[0];//求出前两个元素之间的差值 temp
        for(int i = 1;i < arr.length;i++){
```

```
            if(arr[i] != arr[i - 1] + temp){//判断后续所有相信元素之间的差值是否为
temp
                return false;//只有要一组不符合，就证明不能组成等差数列
            }
        }
        return true;

    }
}
```

https://leetcode-cn.com/problems/can-make-arithmetic-progression-from-sequence/solution/shuang-bai-jie-ti-si-lu-by-codewithwind/

等差数列只需要找到最小值min和间隔gap
(1) 如果gap=0,那么arr中所有的值都必须一样
(2) 如果gap!=0,那么min+gap*N必须都在arr中存在，其中N=1,2...
时间复杂度:O(n)
空间复杂度:O(n)

方法 2：HashSet

```
class Solution {
    public boolean canMakeArithmeticProgression(int[] arr) {
        if(null == arr){
            return false;
        }
        if(arr.length <= 2){
            return true;
        }
        Set<Integer> store = new HashSet<Integer>(1024);
        int minest = 10000000;
        int min = 10000000;
        for(int i = 0; i < arr.length; i++){
            //找到最小的两个
            store.add(arr[i]);
            if(arr[i] < minest){
                int temp = minest;
                minest = arr[i];
                if(temp < min){
                    min = temp;
                }
                continue;
            }else if(arr[i] < min){
                min = arr[i];
                continue;
            }
        }
        int gap = min - minest;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            if(gap == 0){
                return store.size() == 1;
            }
            for(int i = 2; i < arr.length; i ++){
                if(store.contains(min + gap*(i-1))){
                    continue;
                }else{
                    return false;
                }
            }
            return true;
        }
    }
```

28、柠檬水找零（860）

860. 柠檬水找零

难度 简单 126 收藏 分享 切换为英文 关注 反馈

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。

顾客排队购买你的产品，（按账单 bills 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

示例 1：

输入： [5,5,5,10,20]
输出： true
解释：
前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。
第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。
第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。
由于所有客户都得到了正确的找零，所以我们输出 true。

示例 2：

输入： [5,5,10]
输出： true

示例 3：

输入： [10,10]
输出： false

示例 4：

输入： [5,5,10,10,20]
输出： false
解释：
前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。
对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。
对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。
由于不是每位顾客都得到了正确的找零，所以答案是 false。

提示：

- 0 <= bills.length <= 10000
- bills[i] 不是 5 就是 10 或是 20

https://leetcode-cn.com/problems/lemonade-change/solution/ning-meng-shui-zhao-ling-by-leet

code/

方法：模拟情景

思路和算法

让我们尝试模拟给每个购买柠檬水的顾客进行找零的过程。最初，我们既没有 `5` 美元钞票也没有 `10` 美元钞票。

- 如果顾客支付了 `5` 美元钞票，那么我们就得到 `5` 美元的钞票。
- 如果顾客支付了 `10` 美元钞票，我们必须找回一张 `5` 美元钞票。如果我们没有 `5` 美元的钞票，答案就是 `False` ，因为我们无法正确找零。
- 如果顾客支付了 `20` 美元钞票，我们必须找回 `15` 美元。
  - 如果我们有一张 `10` 美元和一张 `5` 美元，那么我们总会更愿意这样找零，这比用三张 `5` 美元进行找零更有利。
  - 否则，如果我们有三张 `5` 美元的钞票，那么我们将这样找零。
  - 否则，我们将无法给出总面值为 `15` 美元的零钱，答案是 `False` 。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是 `bills` 的长度。
- 空间复杂度：$O(1)$。

```
class Solution {
    public boolean lemonadeChange(int[] bills) {
        int five = 0, ten = 0;
        for (int bill: bills) {
            if (bill == 5)
                five++;
            else if (bill == 10) {
                if (five == 0) return false;
                five--;
                ten++;
            } else {
                if (five > 0 && ten > 0) {
                    five--;
                    ten--;
                } else if (five >= 3) {
                    five -= 3;
                } else {
                    return false;
                }
            }
        }
        return true;
    }
}
```

29、分发糖果（135）

更多硬核内容就在公众号「yes 的练级攻略」

135. 分发糖果

难度 困难 242 收藏 分享 切换为英文 关注 反馈

老师想给孩子们分发糖果，有 *N* 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻的孩子中，评分高的孩子必须获得更多的糖果。

那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1:

```
输入: [1,0,2]
输出: 5
解释: 你可以分别给这三个孩子分发 2、1、2 颗糖果。
```

示例 2:

```
输入: [1,2,2]
输出: 4
解释: 你可以分别给这三个孩子分发 1、2、1 颗糖果。
     第三个孩子只得到 1 颗糖果，这已满足上述两个条件。
```

https://leetcode-cn.com/problems/candy/solution/fen-fa-tang-guo-by-powcai/

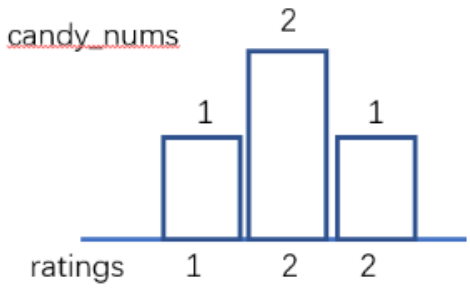
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

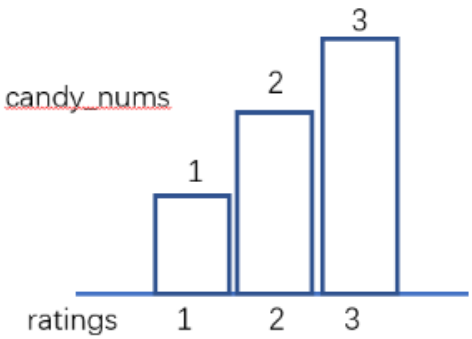
思路二:
只需扫一遍数组。

如何判断 `i` 位置需要多少糖果，我们需要处理有三种情况:

1. `ratings[i - 1]` == `ratings[i]`，那么我们只需要 `1` 糖果



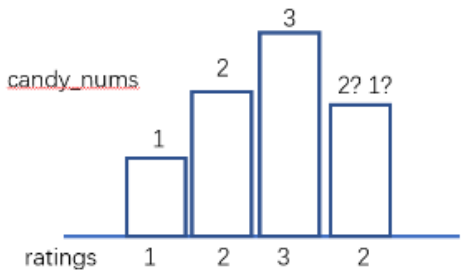
2. `ratings[i - 1]` < `ratings[i]`，那么我们只需要比前一个多一块糖果



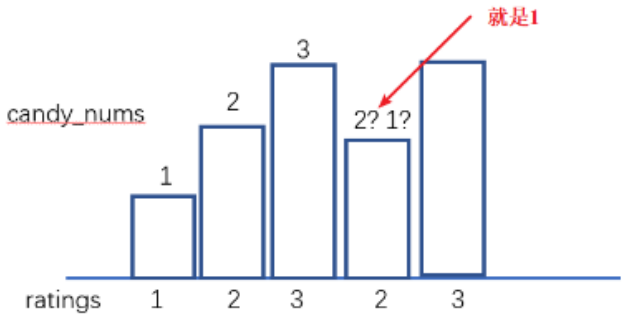
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

3. `ratings[i - 1] > ratings[i]`，那么我们不知道如何判断了，比如：

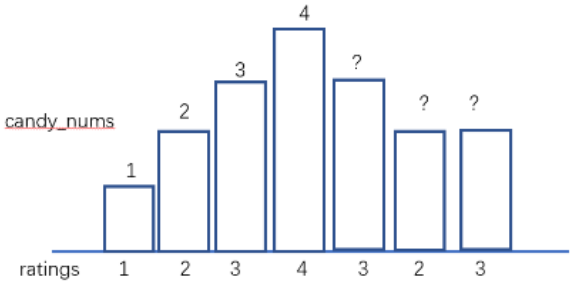


但是，如果知道**递减的个数**，我们就能判断最少的糖果了



公众号：
yes的练级攻略

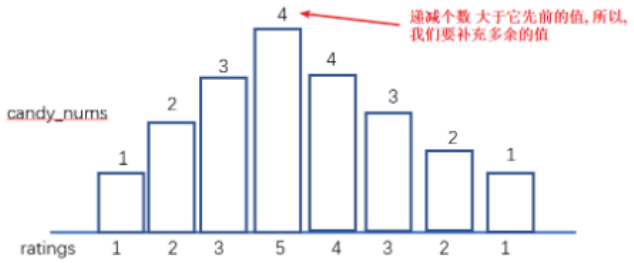
所以我们保证，递减序列，是从 `1` 开始加的(方向加)，再举个例子:



如何判读 `?` 的糖果多少，我们发现 `ratings` 是 `3 - 2` 递减的，递减序列个数 `des_num` 为`，我们**反向加**，

有公差为 `1` 的求和公式首项尾项项数$\frac{(首项+尾项)*项数}{2}$，所以我们先假设 `rating` 在等于 `4` 时候也是满足等差的，所以，有 $\frac{(1+des_num)*des_num}{2} = \frac{(1+2)*2}{2}$ 糖果,所以 `ratings` 是 `3 - 2` 对于的糖果说 `2 - 1`

但是还有一种可能，如下图所示:



所以

时间复杂度 : $O(n)$

空间复杂度 : $O(1)$

```
class Solution {
    public int candy(int[] ratings) {
        if (ratings == null || ratings.length == 0) return 0;
        int n = ratings.length;
        int[] candy_nums = new int[n];
        Arrays.fill(candy_nums, 1);
        for (int i = 1; i < n; i++) {
            if (ratings[i] > ratings[i - 1]) candy_nums[i] = candy_nums[i - 1]
+ 1;
        }
        for (int i = n - 1; i > 0; i--) {
            if (ratings[i - 1] > ratings[i]) candy_nums[i - 1] = Math.max(candy
_nums[i - 1], candy_nums[i] + 1);
        }
        int res = 0;
        for (int i = 0; i < n; i++) res += candy_nums[i];
        return res;
    }
```

}

30、分糖果 2（1103）

1103. 分糖果 II

难度 简单 75 收藏 分享 切换为英文 关注 反馈

排排坐，分糖果。

我们买了一些糖果 candies，打算把它们分给排好队的 n = num_people 个小朋友。

给第一个小朋友 1 颗糖果，第二个小朋友 2 颗，依此类推，直到给最后一个小朋友 n 颗糖果。

然后，我们再回到队伍的起点，给第一个小朋友 n + 1 颗糖果，第二个小朋友 n + 2 颗，依此类推，直到给最后一个小朋友 2 * n 颗糖果。

重复上述过程（每次都比上一次多给出一颗糖果，当到达队伍终点后再次从队伍起点开始），直到我们分完所有的糖果。注意，就算我们手中的剩下糖果数不够（不比前一次发出的糖果多），这些糖果也会全部发给当前的小朋友。

返回一个长度为 num_people、元素之和为 candies 的数组，以表示糖果的最终分发情况（即 ans[i] 表示第 i 个小朋友分到的糖果数）。

示例 1：

```
输入：candies = 7, num_people = 4
输出：[1,2,3,1]
解释：
第一次，ans[0] += 1，数组变为 [1,0,0,0]。
第二次，ans[1] += 2，数组变为 [1,2,0,0]。
第三次，ans[2] += 3，数组变为 [1,2,3,0]。
第四次，ans[3] += 1（因为此时只剩下 1 颗糖果），最终数组变为 [1,2,3,1]。
```

示例 2：

```
输入：candies = 10, num_people = 3
输出：[5,2,3]
解释：
第一次，ans[0] += 1，数组变为 [1,0,0]。
第二次，ans[1] += 2，数组变为 [1,2,0]。
第三次，ans[2] += 3，数组变为 [1,2,3]。
第四次，ans[0] += 4，最终数组变为 [5,2,3]。
```

提示：

- 1 <= candies <= 10^9
- 1 <= num_people <= 1000

https://leetcode-cn.com/problems/distribute-candies-to-people/solution/fen-tang-guo-ii-by-linmingjian/

解题思路

1. 定义 $nums$ 数组存放每个小朋友获得的糖果数量
2. 定义变量 i 递增并对人数取模赋值给 k 表示当前第几个小朋友获得糖果
3. 定义变量 n 表示获得的糖果数量
4. 如果 n 大于糖果数量则把所有糖果给下标为 k 的小朋友，否则糖果总数 $-n$

```
class Solution {
    public int[] distributeCandies(int candies, int num_people) {
        int[] nums = new int[num_people];
        int i = 0;
        while(candies!=0){
            int k = i%num_people;
            int n = i+1;
            if(n<=candies){
                nums[k] = nums[k]+n;
                candies = candies-n;
            }else{
                nums[k] = nums[k]+candies;
```

```
                break;
            }
            i++;
        }
        return nums;
    }
}
```

31、分发饼干（455）

455. 分发饼干

难度 简单 190 收藏 分享 切换为英文 关注 反馈

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i，都有一个胃口值 g_i，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j，都有一个尺寸 s_j。如果 $s_j >= g_i$，我们可以将这个饼干 j 分配给孩子 i，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

注意：

你可以假设胃口值为正。
一个小朋友最多只能拥有一块饼干。

示例 1:

输入: [1,2,3], [1,1]

输出: 1

解释:
你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。
虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。
所以你应该输出1。

示例 2:

输入: [1,2], [1,2,3]

输出: 2

解释:
你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。
你拥有的饼干数量和尺寸都足以让所有孩子满足。
所以你应该输出2.

https://leetcode-cn.com/problems/assign-cookies/solution/tan-xin-jie-fa-by-cyc2018/

1. 给一个孩子的饼干应当尽量小并且又能满足该孩子，这样大饼干才能拿来给满足度比较大的孩子。
2. 因为满足度最小的孩子最容易得到满足，所以先满足满足度最小的孩子。

在以上的解法中，我们只在每次分配时饼干时选择一种看起来是当前最优的分配方法，但无法保证这种局部最优的分配方法最后能得到全局最优解。我们假设能得到全局最优解，并使用反证法进行证明，即假设存在一种比我们使用的贪心策略更优的最优策略。如果不存在这种最优策略，表示贪心策略就是最优策略，得到的解也就是全局最优解。

证明：假设在某次选择中，贪心策略选择给当前满足度最小的孩子分配第 m 个饼干，第 m 个饼干为可以满足该孩子的最小饼干。假设存在一种最优策略，可以给该孩子分配第 n 个饼干，并且 m < n。我们可以发现，经过这一轮分配，贪心策略分配后剩下的饼干一定有一个比最优策略来得大。因此在后续的分配中，贪心策略一定能满足更多的孩子。也就是说不存在比贪心策略更优的策略，即贪心策略就是最优策略。

```
class Solution {
    public int findContentChildren(int[] grid, int[] size) {
        if (grid == null || size == null) return 0;
        Arrays.sort(grid);
        Arrays.sort(size);
        int gi = 0, si = 0;
```

```
            while (gi < grid.length && si < size.length) {
                if (grid[gi] <= size[si]) {
                    gi++;
                }
                si++;
            }
            return gi;
        }
    }
```

32、和为 s 的连续正数序列（剑指 57）

剑指 Offer 57 - II. 和为s的连续正数序列

难度 简单 137 收藏 分享 切换为英文 关注 反馈

输入一个正整数 `target` ，输出所有和为 `target` 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例 1：

```
输入：target = 9
输出：[[2,3,4],[4,5]]
```

示例 2：

```
输入：target = 15
输出：[[1,2,3,4,5],[4,5,6],[7,8]]
```

限制：

- `1 <= target <= 10^5`

https://leetcode-cn.com/problems/he-wei-sde-lian-xu-zheng-shu-xu-lie-lcof/solution/shi-yao-shi-hua-dong-chuang-kou-yi-ji-ru-he-yong-h/

如何用滑动窗口解这道题

要用滑动窗口解这道题，我们要回答两个问题：

- 第一个问题，窗口何时扩大，何时缩小？
- 第二个问题，滑动窗口能找到全部的解吗？

对于第一个问题，回答非常简单：

- 当窗口的和小于 `target` 的时候，窗口的和需要增加，所以要扩大窗口，窗口的右边界向右移动
- 当窗口的和大于 `target` 的时候，窗口的和需要减少，所以要缩小窗口，窗口的左边界向右移动
- 当窗口的和恰好等于 `target` 的时候，我们需要记录此时的结果。设此时的窗口为 $[i, j)$，那么我们已经找到了一个 i 开头的序列，也是唯一一个 i 开头的序列，接下来需要找 $i+1$ 开头的序列，所以窗口的左边界要向右移动

我们一开始要找的是 1 开头的序列，只要窗口的和小于 `target`，窗口的右边界会一直向右移动。假设 $1+2+\cdots+8$ 小于 `target`，再加上一个 9 之后，发现 $1+2+\cdots+8+9$ 又大于 `target` 了。这说明 1 开头的序列找不到解。此时滑动窗口的最右元素是 9。

接下来，我们需要找 2 开头的序列，我们发现，$2+\cdots+8<1+2+\cdots+8<\text{target}$。这说明 2 开头的序列至少要加到 9。那么，我们只需要把原先 1~9 的滑动窗口的左边界向右移动，变成 2~9 的滑动窗口，然后继续寻找。而右边界完全不需要向左移动。

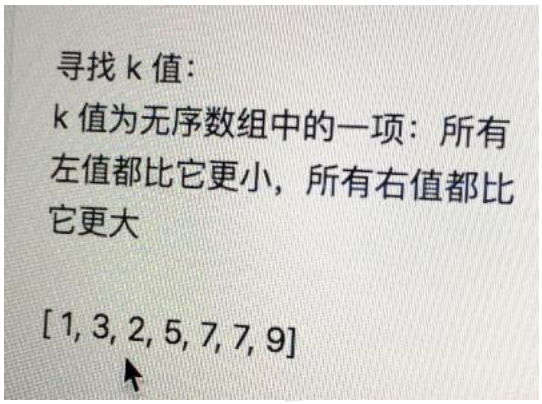
以此类推，滑动窗口的左右边界都不需要向左移动，所以这道题用滑动窗口一定可以得到所有的解。时间复杂度是 $O(n)$。

注：这道题当前可以用等差数列的求和公式来计算滑动窗口的和。不过我这里没有使用求和公式，是为了展示更通用的解题思路。实际上，把题目中的正整数序列换成任意的递增整数序列，这个方法都可以解。

```
class Solution {
    public int[][] findContinuousSequence(int target) {
        int i = 1; // 滑动窗口的左边界
        int j = 1; // 滑动窗口的右边界
        int sum = 0; // 滑动窗口中数字的和
        List<int[]> res = new ArrayList<>();
        while (i <= target / 2) {
            if (sum < target) {
                // 右边界向右移动
                sum += j;
                j++;
            } else if (sum > target) {
                // 左边界向右移动
                sum -= i;
                i++;
            } else {
                // 记录结果
                int[] arr = new int[j-i];
                for (int k = i; k < j; k++) {
                    arr[k-i] = k;
                }
```

```
                res.add(arr);
                // 左边界向右移动
                sum -= i;
                i++;
            }
        }
        return res.toArray(new int[res.size()][]);
    }
}
```

33、寻找 k 值（字节面试题）




```
import java.util.*;
public class Main {
    public static void main(String[] args){
        int[] arr = {1,3,2,5,7,7,9};
        List<Integer> res = findK(arr);
        for (int i = 0; i < res.size(); i++){
            System.out.print(res.get(i) + " ");
        }
    }
    public static List<Integer> findK(int[] arr) {
        List<Integer> list = new LinkedList<>();
        int n = arr.length;
        int[] dpMax = new int[n];
        int[] dpMin = new int[n];
        dpMax[0] = arr[0];
        dpMin[n - 1] = arr[n - 1];
        for (int i = 1; i < n; i++) {
            //if (arr[i] > dpMax[i-1]) dpMax[i] = arr[i];
            //else dpMax[i] = dpMax[i-1];
            dpMax[i] = Math.max(arr[i], dpMax[i - 1]);
        }
        for (int i = n - 2; i >= 0; i--) {
            //if (arr[i] < dpMin[i+1]) dpMin[i] = arr[i];
            //else dpMin[i] = dpMin[i+1];
```

商用 公众号：yes的练级攻略

```
                dpMin[i] = Math.min(arr[i], dpMin[i + 1]);
            }
            for (int i = 0; i < n; i++) {
                if (dpMax[i] == dpMin[i] && (i == 0 || arr[i-1] < arr[i]) && (i == n - 1 || arr[i] < arr[i+1])) {
                    list.add(dpMax[i]);
                }
            }
            return list;
        }
}
```

34、打乱数组（384）【洗牌算法】

384. 打乱数组

难度 中等 87 收藏 分享 切换为英文 关注 反馈

打乱一个没有重复元素的数组。

示例:

```
// 以数字集合 1, 2 和 3 初始化数组。
int[] nums = {1,2,3};
Solution solution = new Solution(nums);

// 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3]的排列返回的概率应该相同。
solution.shuffle();

// 重设数组到它的初始状态[1,2,3]。
solution.reset();

// 随机返回数组[1,2,3]打乱后的结果。
solution.shuffle();
```

https://leetcode-cn.com/problems/shuffle-an-array/solution/da-luan-shu-zu-by-leetcode/

方法二： Fisher-Yates 洗牌算法 【通过】

思路

我们可以用一个简单的技巧来降低之前算法的时间复杂度和空间复杂度，那就是让数组中的元素互相交换，这样就可以避免掉每次迭代中用于修改列表的时间了。

算法

Fisher-Yates 洗牌算法跟暴力算法很像。在每次迭代中，生成一个范围在当前下标到数组末尾元素下标之间的随机整数。接下来，将当前元素和随机选出的下标所指的元素互相交换 - 这一步模拟了每次从“帽子”里面摸一个元素的过程，其中选取下标范围的依据在于每个被摸出的元素都不可能再被摸出来了。此外还有一个需要注意的细节，当前元素是可以和它本身互相交换的 - 否则生成最后的排列组合的概率就不对了。为了更清楚地理解这一过程，可以看下面这些动画：

复杂度分析

- 时间复杂度：$O(n)$
  Fisher-Yates 洗牌算法时间复杂度是线性的，因为算法中生成随机序列，交换两个元素这两种操作都是常数时间复杂度的。
- 空间复杂度：$O(n)$
  因为要实现 `重置` 功能，原始数组必须得保存一份，因此空间复杂度并没有优化。

```
class Solution {
        static int originalarr[];
        static int resarr[];
        public Solution(int[] nums) {
            this.originalarr = nums.clone();
            this.resarr = nums;
        }
        /** Resets the array to its original configuration and return i
t. */
        public int[] reset() {
            return originalarr;
        }
        /** Returns a random shuffling of the array. */
        public int[] shuffle() {
            int len = resarr.length;
            Random random = new Random();
            for (int i = 0; i < len; i++) {
                int a = random.nextInt(len) ;
                int tmp = resarr[i];
                resarr[i] = resarr[a];
                resarr[a] = tmp;
            }
            return resarr;
        }
}
/**
 * Your Solution object will be instantiated and called as such:
 * Solution obj = new Solution(nums);
 * int[] param_1 = obj.reset();
 * int[] param_2 = obj.shuffle();
 */
```

35、生成数组（1420）

更多硬核内容就在公众号「yes 的练级攻略」

1420. 生成数组

难度 困难 44 收藏 分享 切换为英文 关注 反馈

给你三个整数 `n`、`m` 和 `k` 。下图描述的算法用于找出正整数数组中最大的元素。

收藏

```
maximum_value = -1
maximum_index = -1
search_cost = 0
n = arr.length
for (i = 0; i < n; i++) {
    if (maximum_value < arr[i]) {
        maximum_value = arr[i]
        maximum_index = i
        search_cost = search_cost + 1
    }
}
return maximum_index
```

请你生成一个具有下述属性的数组 `arr` ：

- `arr` 中有 `n` 个整数。
- `1 <= arr[i] <= m` 其中 `(0 <= i < n)` 。
- 将上面提到的算法应用于 `arr` ， `search_cost` 的值等于 `k` 。

返回上述条件下生成数组 `arr` 的 **方法数** ，由于答案可能会很大，所以 **必须** 对 `10^9 + 7` 取余。

示例 1：

输入： `n = 2, m = 3, k = 1`
输出： `6`
解释： 可能的数组分别为 `[1, 1], [2, 1], [2, 2], [3, 1], [3, 2] [3, 3]`

示例 2：

输入： `n = 5, m = 2, k = 3`
输出： `0`
解释： 没有数组可以满足上述条件

示例 3：

输入： `n = 9, m = 1, k = 1`
输出： `1`
解释： 可能的数组只有 `[1, 1, 1, 1, 1, 1, 1, 1, 1]`

示例 4：

输入： `n = 50, m = 100, k = 25`
输出： `34549172`
解释： 不要忘了对 `1000000007` 取余

示例 5：

输入： `n = 37, m = 17, k = 7`
输出： `418930126`

提示：

- `1 <= n <= 50`
- `1 <= m <= 100`
- `0 <= k <= n`

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/build-array-where-you-can-find-the-maximum-exactly-k-comparisons/solution/java-onmkjie-fa-10ms-by-cjjohn/

解题思路

定义 `dp[i][j][p]` 代表对于前 `i` 个数，最大的数为 `j` ，然后search cost为 `p` 时的符合条件的数组总数。那么对于前 `i` 个数且最大值为 `j` 、search cost为 `k` 的这种情况，又可以分为两种情况：

- 最大值 `j` 在位置 `1~i-1` 之间，则第 `i` 个位置上的数可以是 `1~j` ，则此时的总数为 `j * dp[i-1][j][k]` ，这里的 `dp[i-1][j][k]` 就是前 `i-1` 个数里刚好最大值为 `j` 且search cost为 `k` 时的总数；
- 最大值 `j` 在第 `i` 个位置，则此时的方法数就是前 `i-1` 个数最大值为 `1~j-1` 这 `j-1` 种情况、search cost为 `k-1` 的 `j-1` 个总数之和：$\sum_{p=1}^{j-1} dp[i-1][p][k-1]$

总结一下，就是 $dp[i][j][k] = j \times dp[i-1][j][k] + \sum_{p=1}^{j-1} dp[i-1][p][k-1]$ 。然后边界条件为 `dp[i][1][1]=1` （对应于全部 `i` 个数全为1的情况）以及 `dp[1][j][1]=1` （对应于只有一个数的情况，这个数只能是 `j` ）。

时间复杂度为$O(nm^2k)$，空间复杂度为$O(mnk)$。

上面的复杂度是基于三层循环的顺序分别为 `i` 、 `j` 、 `p` 而得到的，但其实只要我们稍微改一下三个循环的顺序，把对 `p` 的循环提到对 `j` 的循环之前就可以借助于对 `j` 的循环求得$\sum_{p=1}^{j-1} dp[i-1][p][k-1]$。从而减少时间复杂度为$O(nmk)$，空间复杂度还是$O(mnk)$。

```java
class Solution {
    public int numOfArrays(int n, int m, int k) {
        if (k == 0 || m < k) return 0;
        int mod = 1000000007;
        // dp[i][j][p]:前 i 个数，最大的为 j，search cost 为 p
        int[][][] dp = new int[n + 1][m + 1][k + 1];
        for (int i = 1; i <= n; i++){
            dp[i][1][1] = 1;
        }
        for (int i = 1; i <= m; i++){
            dp[1][i][1] = 1;
        }
        for (int i = 2; i <= n; ++i) {
            for (int p = 1; p <= k; ++p) {
                int sum = 0;
                for(int j = 2; j <= m; ++j) {
                    sum = (sum + dp[i - 1][j - 1][p - 1]) % mod;
                    int temp = (int)((dp[i - 1][j][p] * 1L * j) % mod);
                    dp[i][j][p] = (sum + temp) % mod;
                }
            }
        }
        int result = 0;
        for (int j = 1; j <= m; ++j) {
            result = (result + dp[n][j][k]) % mod;
        }
        return result % mod;
    }
```

}

36、递增子序列（491）

491. 递增子序列

难度 中等 118 收藏 分享 切换为英文 关注 反馈

给定一个整型数组, 你的任务是找到所有该数组的递增子序列，递增子序列的长度至少是2。

示例:

```
输入: [4, 6, 7, 7]
输出: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]
```

说明:

1. 给定数组的长度不会超过15。
2. 数组中的整数范围是 [-100,100]。
3. 给定数组中可能包含重复数字，相等的数字应该被视为递增的一种情况。

https://leetcode-cn.com/problems/increasing-subsequences/solution/jin-tian-wo-you-shuang-ruo-zhuo-neng-miao-dong-la-/

题目分析

- 定义全局变量 `List<List<Integer>> res` 保存结果；
- 定义 `dfs(int[] nums, int idx, List<Integer> curList)` 进行搜索；
  - 参数 `int idx` 表示当前搜索到的数组下标，初始化为 -1，每次在 `[idx + 1, nums.length - 1]` 范围内遍历搜索递增序列的下一个值。
  - 参数 `List<Integer> curList` 表示当前的递增序列，只要长度大于 1，就拷贝一份存入结果列表 res 中。
- 每次在 `[idx + 1, nums.length - 1]` 范围内遍历搜索递增序列的下一个值时，借助 set 进行去重，如果当前的值已经在 set 中了，则终止此路径继续搜索。

```
class Solution {
    // 定义全局变量保存结果
    List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> findSubsequences(int[] nums) {
        // idx 初始化为 -1，开始 dfs 搜索。
        dfs(nums, -1, new ArrayList<>());
        return res;
    }
    private void dfs(int[] nums, int idx, List<Integer> curList) {
        // 只要当前的递增序列长度大于 1，就加入到结果 res 中，然后继续搜索递增序列的下一个值。
        if (curList.size() > 1)  res.add(new ArrayList<>(curList));
        // 在 [idx + 1, nums.length - 1] 范围内遍历搜索递增序列的下一个值。
        // 借助 set 对 [idx + 1, nums.length - 1] 范围内的数去重。
        Set<Integer> set = new HashSet<>();
        for (int i = idx + 1; i < nums.length; i++) {
```

```
            // 1. 如果 set 中已经有与 nums[i] 相同的值了，说明加上 nums[i] 后的所有可能的递增序列之前已经被搜过一遍了，因此停止继续搜索。
            if (set.contains(nums[i])) continue;
            set.add(nums[i]);
            // 2. 如果 nums[i] >= nums[idx] 的话，说明出现了新的递增序列，因此继续 dfs 搜索（因为 curList 在这里是复用的，因此别忘了 remove 哦）
            if (idx == -1 || nums[i] >= nums[idx]) {
                curList.add(nums[i]);
                dfs(nums, i, curList);
                curList.remove(curList.size() - 1);
            }
        }
    }
}
```

如果要输出长度最长的子序列

```
import java.util.*;
public class Main {
    public static int lengthOfLIS(int[] nums) {
        int[] tails = new int[nums.length];
        int res = 0;
        for(int num : nums) {
            int i = 0, j = res;
            while(i < j) {
                int m = (i + j) / 2;
                if(tails[m] < num) i = m + 1; //如果是最长非减的话，这边加等号
                    else j = m;
            }
            tails[i] = num;
            if(res == j) res++;
        }
        return res;
    }
    public static List<List<Integer>> findSubsequences(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        int n = nums.length;
        if (n == 0) return res;
        dfs(nums, new int[n], 0, -1, Integer.MIN_VALUE, res, n);
        return res;
    }
    public static void dfs(int[] nums, int[] tmp, int curp, int prep, int prev, List<List<Integer>> res, int n){
        if (curp > 1) {
            List<Integer> tres = new ArrayList<>();
            for (int i = 0; i < curp; i++) {
                tres.add(tmp[i]);
```

```
                }
                res.add(tres);
            }
            Set<Integer> set = new HashSet<>();
            for (int i = prep + 1; i < n; i++) {
                if (!set.contains(nums[i]) && nums[i] > prev) {
                    set.add(nums[i]);
                    tmp[curp] = nums[i];
                    dfs(nums, tmp, curp + 1, i, nums[i], res, n);
                }
            }
        }
        public static void main(String[] args) {
            //int[] nums = new int[]{3, 5, 6, 2, 5, 4, 19, 5, 6, 7, 12};
              int[] nums = new int[]{10,9,2,5,3,7,101,102,200,67,297,168,2,2,3,1,1,3,2,1,2,3,4,5,5,6,7,8,18};
            int lengthOfLIS = lengthOfLIS(nums);
            System.out.println("最长上升子序列的长度：" + lengthOfLIS);
            List<List<Integer>> res = findSubsequences(nums);
            List<Integer> maxLen = res.get(0);
            for (int i = 1; i < res.size(); i++) {
                List<Integer> tres = res.get(i);
                if (tres.size() > maxLen.size()) {
                    maxLen = tres;
                }
            }
            int[] max = maxLen.stream().mapToInt(Integer::valueOf).toArray();
            System.out.println("最长上升子序列是：");
            for (int i = 0; i < max.length; i++) {
                if (i != max.length - 1) System.out.print(max[i] + ",");
                else System.out.print(max[i]);
            }
        }
    }
```

37、最长上升子序列（300）

更多硬核内容就在公众号「yes 的练级攻略」

300. 最长上升子序列

难度 中等 795 收藏 分享 切换为英文 关注 反馈

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

```
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

说明:

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 $O(n^2)$ 。

进阶: 你能将算法的时间复杂度降低到 $O(n \log n)$ 吗?

https://leetcode-cn.com/problems/longest-increasing-subsequence/solution/dong-tai-gui-hua-er-fen-cha-zhao-tan-xin-suan-fa-p/

方法二：动态规划

- 首先考虑题目问什么，就把什么定义成状态。
- 题目问最长上升子序列的长度，其实可以把「子序列的长度」定义成状态，但是发现「状态转移」不好做；
- 把「子序列的长度」定义成状态，事实上也可以，只是目前这样定义状态，**没有定义得很清晰**，具体做法在下文「方法三」；
- 为了从一个较短的上升子序列得到一个较长的上升子序列，我们主要关心这个较短的上升子序列结尾的元素。由于要保证子序列的相对顺序，在程序读到一个新的数的时候，如果比已经得到的子序列的最后一个数还大，那么就可以放在这个子序列的最后，形成一个更长的子序列；

于是我们可以这样定义状态：

第 1 步：定义状态

由于一个子序列一定会以一个数结尾，于是将状态定义成：`dp[i]` 表示**以** `nums[i]` **结尾**的「上升子序列」的长度。注意：这个定义中 `nums[i]` **必须被选取，且必须是这个子序列的最后一个元素。**

第 2 步：考虑状态转移方程

- 遍历到 `nums[i]` 时，需要把下标 `i` 之前的所有的数都看一遍；
- 只要 `nums[i]` 严格大于在它位置之前的某个数，那么 `nums[i]` 就可以接在这个数后面形成一个更长的上升子序列；
- 因此，`dp[i]` 就等于下标 `i` 之前**严格**小于 `nums[i]` 的状态值的最大者 $+1$。

语言描述：在下标 `i` 之前严格小于 `nums[i]` 的所有状态值中的最大者 $+1$。

符号描述：

$$dp[i] = \max_{0 \le j < i, nums[j] < nums[i]} dp[j] + 1$$

更多硬核内容就在公众号「yes 的练级攻略」

第 3 步：考虑初始化

- `dp[i] = 1`，1 个字符显然是长度为 1 的上升子序列。

第 4 步：考虑输出

- 这里要注意，不能返回最后一个状态值；
- 还是根据定义，最后一个状态值只是以 `nums[len - 1]` 结尾的「上升子序列」的长度；
- 状态数组 `dp` 的最大值才是最后要输出的值。

$$\max_{1 \leq i \leq N} dp[i]$$

第 5 步：考虑状态压缩。

- 遍历到一个新数的时候，之前所有的状态值都得保留，因此无法压缩。

看下面的例子理解「动态规划」的执行流程。

提示：由于幻灯片上的文字比较多，可以先让幻灯片动起来，从整体把握思想。

复杂度分析：

- 时间复杂度：$O(N^2)$，这里 N 是数组的长度，我们写了两个 `for` 循环，每个 `for` 循环的时间复杂度都是线性的。
- 空间复杂度：$O(N)$，要使用和输入数组长度相等的状态数组，因此空间复杂度是 $O(N)$。

这个「动态规划」的方法在计算一个新的状态的时候，需要考虑到之前所有小于 `nums[i]` 的那些位置的状态。

事实上，这个算法还有改进的空间。我们是从「状态」的定义开始考虑的。

（1）普通的动态规划代码（12ms）

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums.length < 2) return nums.length;
        int[] dp = new int[nums.length];
        dp[0] = 1;
        int res = 1;
        for (int i = 1; i < nums.length; i++) {
            int max = 0;
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]){
                    max = Math.max(max, dp[j]);
                }
            }
            dp[i] = max + 1;
            res = Math.max(res,dp[i]);
        }
        return res;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/longest-increasing-subsequence/solution/zui-chang-shang-sheng-zi-xu-lie-dong-tai-gui-hua-2/

解法二：动态规划 + 二分查找

解题思路：

- **降低复杂度切入点：** 解法一中，遍历计算 dp 列表需 $O(N)$，计算每个 $dp[k]$ 需 $O(N)$。
  1. 动态规划中，通过线性遍历来计算 dp 的复杂度无法降低；
  2. 每轮计算中，需要通过线性遍历 $[0,k)$ 区间元素来得到 $dp[k]$ 。我们考虑：是否可以通过重新设计**状态定义**，使整个 dp 为一个**排序列表**；这样在计算每个 $dp[k]$ 时，就可以通过二分法遍历 $[0,k)$ 区间元素，将此部分复杂度由 $O(N)$ 降至 $O(logN)$。
- **设计思路：**
  - **新的状态定义：**
    - 我们考虑维护一个列表 $tails$，其中每个元素 $tails[k]$ 的值代表 **长度为 $k+1$ 的子序列尾部元素的值。**
    - 如 $[1,4,6]$ 序列，长度为 $1,2,3$ 的子序列尾部元素值分别为 $tails=[1,4,6]$。
  - **状态转移设计：**
    - 设常量数字 N，和随机数字 x，我们可以容易推出：当 N 越小时，$N<x$ 的几率越大。例如： $N=0$ 肯定比 $N=1000$ 更可能满足 $N<x$。
    - 在遍历计算每个 $tails[k]$，不断更新长度为 $[1,k]$ 的子序列尾部元素值，**始终保持每个尾部元素值最小** （例如 $[1,5,3]$），遍历到元素 5 时，长度为 2 的子序列尾部元素值为 5；当遍历到元素 3 时，尾部元素值应更新至 3，因为 3 遇到比它大的数字的几率更大）。
  - $tails$ **列表一定是严格递增的：** 即当尽可能使每个子序列尾部元素值最小的前提下，子序列越长，其序列尾部元素值一定更大。
    - **反证法证明：** 当 $k<i$，若 $tails[k]>=tails[i]$，代表较短子序列的尾部元素的值 $>$ 较长子序列的尾部元素的值。这是不可能的，因为从长度为 i 的子序列尾部倒序删除 $i-1$ 个元素，剩下的为长度为 k 的子序列，设此序列尾部元素值为 v，则一定有 $v<tails[i]$ （即长度为 k 的子序列尾部元素值一定更小），这和 $tails[k]>=tails[i]$ 矛盾。
    - 既然严格递增，每轮计算 $tails[k]$ 时就可以使用二分法查找需要更新的尾部元素值的对应索引 i。
- **算法流程：**
  - **状态定义：**
    - $tails[k]$ 的值代表 长度为 $k+1$ 子序列 的尾部元素值。
  - **转移方程：** 设 res 为 $tails$ 当前长度，代表直到当前的最长上升子序列长度。设 $j\in[0,res)$，考虑每轮遍历 $nums[k]$ 时，通过二分法遍历 $[0,res)$ 列表区间，找出 $nums[k]$ 的大小分界点，会出现两种情况：
    - **区间中存在** $tails[i]>nums[k]$ **：** 将第一个满足 $tails[i]>nums[k]$ 执行 $tails[i]=nums[k]$ ；因为更小的 $nums[k]$ 后更可能接一个比它大的数字（前面分析过）。
    - **区间中不存在** $tails[i]>nums[k]$ **：** 意味着 $nums[k]$ 可以接在前面所有长度的子序列之后，因此肯定是接到最长的后面（长度为 res ），新子序列长度为 $res+1$。
  - **初始状态：**
    - 令 $tails$ 列表所有值 $=0$。
  - **返回值：**
    - 返回 res ，即最长上升子序列长度。

复杂度分析：

- **时间复杂度** $O(NlogN)$ **：** 遍历 $nums$ 列表需 $O(N)$，在每个 $nums[i]$ 二分法需 $O(logN)$。
- **空间复杂度** $O(N)$ **：** $tails$ 列表占用线性大小额外空间。

（2）用到了贪心算法和二分查找

```
public class Solution {
    public int lengthOfLIS(int[] nums) {
```

```
        int[] tails = new int[nums.length];
        int res = 0;
        for(int num : nums) {
            int i = 0, j = res;
            while(i < j) {
                int m = (i + j) / 2;
                if(tails[m] < num) i = m + 1; //如果是最长非减的话，这边加
等号
                else j = m;
            }
            tails[i] = num;
            if(res == j) res++;
        }
        return res;
    }

    // 调试方法，以观察是否运行正确
    private void printArray(int num, int[] tail) {
        System.out.print("当前数字：" + num);
        System.out.print("\t 当前 tail 数组：");
        int len = tail.length;
        for (int i = 0; i < len; i++) {
            if (tail[i] == 0) {
                break;
            }
            System.out.print(tail[i] + ", ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] nums = new int[]{3, 5, 6, 2, 5, 4, 19, 5, 6, 7, 12};
        Solution solution = new Solution8();
        int lengthOfLIS = solution8.lengthOfLIS(nums);
        System.out.println("最长上升子序列的长度：" + lengthOfLIS);
    }
}
```

38、最长递增子数组

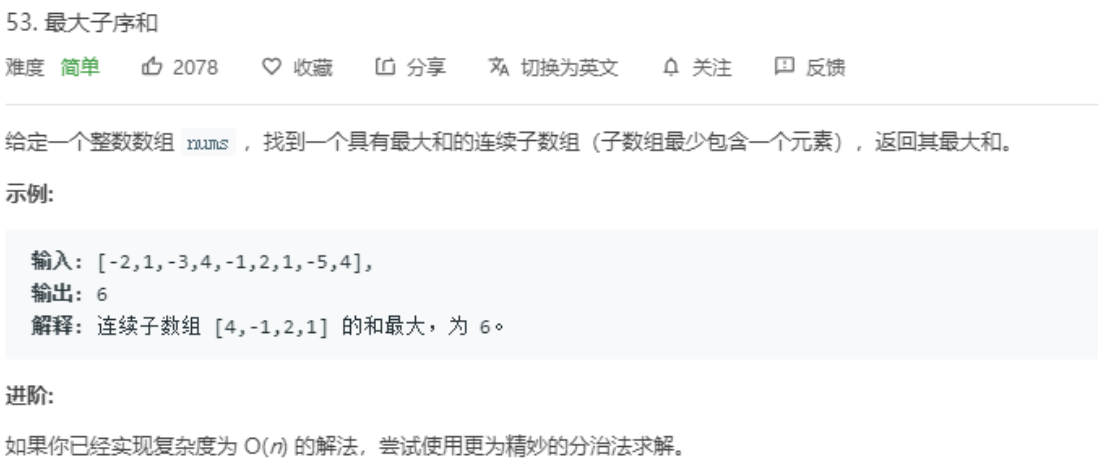
最长递增子数组长度，最长递增子数组具体数组，最长非递减子数组长度，最长非递减子数组具体数组，最长递减子数组长度，最长递减子数组具体数组，最长非递增子数组长度，最长非递增子数组具体数组。

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
```

```
        //int a[] = {3,2,1};
        int a[] = {4,1,2,6,8,2,5,1,1,6,7,8,1};
        int res[] = findLongestArray(a);
        int maxLen = findLongestLength(a);
        System.out.println("最长递增子数组的长度是：" + maxLen);
        System.out.println("最长递增子数组是：");
        if (res.length == 1) System.out.print(""); //没有递增序列，都是递减的情况下，写这行可以将结果清空，如果不写这行，会输出任意一个长度为1 的元素。
            else {
              for (int i = 0; i < res.length; i++){
                  if (i != res.length - 1) System.out.print(res[i] + " ");
                  else System.out.print(res[i]);
              }
        }
    }
    public static int findLongestLength(int[] a) {
        int size = a.length;
        int length = 1, maxlen = 1;
        int i;
        for(i = 1; i < size; i++) {
            if(a[i-1] < a[i]) { //递增"<",递减">",非递减用“<=”， 非递增用">="
                    length++;
            }
            else {
                maxlen = Math.max(maxlen, length);
                length = 1;
            }
        }
        return maxlen;
    }
    public static int[] findLongestArray(int[] a) {
        int size = a.length;
        int length = 1, maxlen = 1, startpoint = 0;
        int i;
        for(i = 1; i < size; i++) {
            if(a[i-1] < a[i]) { //递增"<",递减">",非递减用“<=”， 非递增用">="
                    length++;
            }
            else {
                if(length >= maxlen) { //如果为“>”，就是输出第一个最长递增数组；如果是”>=”，就是输出最后一个最长递增数组
                        maxlen = length;
                    startpoint = i - length;
                }
```

```
                length = 1;
            }
        }
        if(length >= maxlen) { //如果为">"，就是输出第一个最长递增数组；如果是">="，就是输出最后一个最长递增数组
            maxlen = length;
            startpoint = i - length;
        }
        int[] res = new int[maxlen];
        for(i = 0; i < maxlen; i++) {
            res[i] = a[startpoint+i];
        }
        return res;
    }
}
```

39、最大子序和（53）【最大子数组（lintcode 41）】【最大和的子数组】【连续数列（面试题 16.17）】



53. 最大子序和

难度 简单 2078 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例:

```
输入: [-2,1,-3,4,-1,2,1,-5,4],
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。
```

进阶:

如果你已经实现复杂度为 O(*n*) 的解法，尝试使用更为精妙的分治法求解。

https://leetcode-cn.com/problems/maximum-subarray/solution/53-zui-da-zi-xu-he-java-dong-tai-gui-hua-by-flagma/

```
class Solution {
    public int maxSubArray(int[] nums) {
        int len = nums.length;
        int[] dp = new int[len];
        dp[0] = nums[0];
        int max = nums[0];
        for (int i = 1; i < len; i++) {
            dp[i] = Math.max(dp[i-1] + nums[i], nums[i]);
            if (max < dp[i]) {
                max = dp[i];
            }
        }
        return max;
```

```
        }
    }
```

输出具体的子数组

```
public static int[] maxSubArray(int[] nums) {
        int len = nums.length;
        int[] dp = new int[len];
        dp[0] = nums[0];
        int maxValue = nums[0], head = 0, tail = 0, tmp_head = 0, tmp_tail = 0;
        for (int i = 1; i < len; i++) {
            if (dp[i-1] + nums[i] >= nums[i]) {
                dp[i] = dp[i-1] + nums[i];
                tmp_tail = i;
            }else {
                dp[i] = nums[i];
                tmp_head = i;
            }
            if (maxValue < dp[i]) {
                maxValue = dp[i];
                head = tmp_head;
                tail = tmp_tail;
            }
        }
        int[] res = new int[tail - head + 1];
        for (int i = 0; i < tail - head + 1; i++) {
            res[i] = nums[i+head];
        }
        return res;
}
```

40、第 k 大的子数组（lintcode1139）

1139. 第k大的子数组 中文 English

给定一个长度为n的数组a，它有$\frac{n(n+1)}{2}$个子数组。请计算这些子数组的和，然后按照升序排列，并返回排序后第k个数。

样例

Example1

```
Input:
[2,3,1,4]
6
Output:5
Explanation:
我们可以得到所有子数组的和是 [1,2,3,4,4(3 + 1), 5(1 + 4), 5(2 + 3), 6(2 + 3 + 1), 8(3 + 1 + 4), 10(2 + 3 + 1 + 4)]。
其中第六个是5。
```

注意事项

- $1 \le n \le 10^5$
- $1 \le a_i \le 10^9$
- $1 \le k \le \frac{n(n+1)}{2}$

https://mp.weixin.qq.com/s/UdrJjPl8tMCnJFhg_mkg3A

更多硬核内容就在公众号「yes 的练级攻略」

【题解】

算法

二分+two pointer

算法分析

我们可以看到，题目需要求和第k大的子区间，而我们的区间总个数共有n(n+1)/2个，当n为10^5时这个数高达10^10级别。我们显然不能暴力的枚举每一个区间和然后排序。

算法思路

我们注意到，所有数字的和不超过10^14，这个范围可以让我们想到使用二分最终的答案进行求解。

二分要求解的问题是：对于给定的和x，求有多少个区间的和小于x，小于等于x。这需要我们在O(n)的时间复杂度内进行求解。由于数组内所有数都是正数，我们自然的可以想到同向双指针求解。当当前区间的和大于k，就移动左指针，否则移动右指针。

时间复杂度

O(nlog(n))

```
public class Solution {
    /**
     * @param a: an array
     * @param k: the kth
     * @return: return the kth subarray
     */
    public long thekthSubarray(int[] a, long k) {
        int n = a.length;
        long sum = 0;
        for (int i = 0; i < n; i++) sum += a[i];
        long left = 1, right = sum;
        while (left <= right){
            long mid = left + (right - left) / 2;
            int flag = check(mid, a, k);
            if (flag == 0) return mid;
            if (flag == 1) right = mid - 1;
            else left = mid + 1;
        }
        return -1;
    }
    public int check(long x, int[] a, long k){
        long tmp1 = 0, tmp2 = 0, cur = a[0];
        int left = -1, right = 0, n = a.length;
        long all = (long) n * (n + 1) / 2;
```

公众号：yes的练级攻略

```
            while (left <= right && right < n){
                if (cur >= x) {
                    if (cur == x) tmp2++;
                    else tmp1++;
                    tmp1 += n - right - 1;
                    left++;
                    cur -= a[left];
                } else {
                    right++;
                    if (right < n) cur += a[right];
                }
            }
            if (all - tmp1 - tmp2 < k && all - tmp1 >= k) return 0;
            if (all - tmp1 - tmp2 >= k) return 1;
            else return -1;
        }
}
```

41、和至少为 K 的最短子数组（862）【滑动窗口】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

862. 和至少为 K 的最短子数组

难度 困难 194 收藏 分享 切换为英文 关注 反馈

返回 `A` 的最短的非空连续子数组的长度，该子数组的和至少为 `K` 。

如果没有和至少为 `K` 的非空子数组，返回 `-1` 。

示例 1：

```
输入：A = [1], K = 1
输出：1
```

示例 2：

```
输入：A = [1,2], K = 4
输出：-1
```

示例 3：

```
输入：A = [2,-1,2], K = 3
输出：3
```

提示：

1. `1 <= A.length <= 50000`
2. `-10 ^ 5 <= A[i] <= 10 ^ 5`
3. `1 <= K <= 10 ^ 9`

YES的练级攻略

https://leetcode-cn.com/problems/shortest-subarray-with-sum-at-least-k/solution/he-zhi-shao-wei-k-de-zui-duan-zi-shu-zu-by-leetcod/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：滑动窗口

分析

我们用数组 `P` 表示数组 `A` 的前缀和，即 `P[i] = A[0] + A[1] + ... + A[i - 1]`。我们需要找到 `x` 和 `y`，使得 `P[y] - P[x] >= K` 且 `y - x` 最小。

我们用 `opt(y)` 表示对于固定的 `y`，最大的满足 `P[x] <= P[y] - K` 的 `x`，这样所有 `y - opt(y)` 中的最小值即为答案。我们可以发现两条性质：

- 如果 `x1 < x2` 且 `P[x2] <= P[x1]`，那么 `opt(y)` 的值不可能为 `x1`，这是因为 `x2` 比 `x1` 大，并且如果 `x1` 满足了 `P[x1] <= P[y] - K`，那么 `P[x2] <= P[x1] <= P[y] - K`，即 `x2` 同样满足 `P[x2] <= P[y] - K`。
- 如果 `opt(y1)` 的值为 `x`，那么我们以后就不用再考虑 `x` 了。这是因为如果有 `y2 > y1` 且 `opt(y2)` 的值也为 `x`，但此时 `y2 - x` 显然大于 `y1 - x`，不会作为所有 `y - opt(y)` 中的最小值。

算法

我们维护一个关于前缀和数组 `P` 的单调队列，它是一个双端队列（deque），其中存放了下标 `x`：`x0, x1, ...` 满足 `P[x0], P[x1], ...` 单调递增。这是为了满足性质一。

当我们遇到了一个新的下标 `y` 时，我们会在队尾移除若干元素，直到 `P[x0], P[x1], ..., P[y]` 单调递增。这同样是为了满足性质一。

同时，我们会在队首也移除若干元素，如果 `P[y] >= P[x0] + K`，则将队首元素移除，直到该不等式不满足。这是为了满足性质二。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是数组 `A` 的长度。
- 空间复杂度：$O(N)$。

我来翻一下解答：

首先解释P:
P[n]=sum(A[:n-1])
举例： A = [2,-1,2]
P = [0, 2, 1, 3]

如果我们想要A[1:3]的和，也就是[-1,2]的和，结果就是P[3]-P[1]=1

队列维护：
官方答案就是通过维持一个队列，从头到尾遍历。对于每个y，找出满足，P[y]-P[x]>=K的最大的x。如果y-x比之前的长度要小就记录新的最小值。

维持队列涉及两个规则：

1. 对于新加入的y，前面的P[?]都要比新加入的P[y]要小，比P[y]大的P[?]都要pop掉，甚至如果都比P[y]大，整个队列都要清空。
   为什么?
   因为只有比P[y]小的P[?]，才能跟y组成，(y,x)组合，使得P[y]-P[x]>=K。那些不比P[y]小的P[?]，起不到任何作用，不可能存在一个x让当前的y或者之后的新y满足P[y]-P[x]>=K，也就不可能去更新最小长度。因此，只有比 P[y]小的P[?]才有保留的必要。
2. 为什么当队列里第一个x满足P[y]-P[x]>=K的时候，第一个x可以被pop掉?
   因为此时我们构成了一个P[y]-P[x]>=K，之后这个x就没有作用了。
   为什么这个x没有用了?
   因为即使之后存在某个其他的y'，也可以跟这个x构成P[y]-P[x]>=K，但是因为y'>y，因此次新的长度一定比当前的长度y-x要长，因为可以不用考虑。

```
class Solution {
    public int shortestSubarray(int[] A, int K) {
        int N = A.length;
        long[] P = new long[N+1];
        for (int i = 0; i < N; ++i)
            P[i+1] = P[i] + (long) A[i];
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        // Want smallest y-x with P[y] - P[x] >= K
        int ans = N+1; // N+1 is impossible
        Deque<Integer> monoq = new LinkedList(); //opt(y) candidates, as indice
s of P
        for (int y = 0; y < P.length; ++y) {
            // Want opt(y) = largest x with P[x] <= P[y] - K;
            while (!monoq.isEmpty() && P[y] <= P[monoq.getLast()])
                monoq.removeLast();
            while (!monoq.isEmpty() && P[y] >= P[monoq.getFirst()] + K)
                ans = Math.min(ans, y - monoq.removeFirst());

            monoq.addLast(y);
        }
        return ans < N+1 ? ans : -1;
    }
}
```

42、长度最小的子数组（209）

209. 长度最小的子数组

难度 中等 406 收藏 分享 切换为英文 关注 反馈

给定一个含有 n 个正整数的数组和一个正整数 s，找出该数组中满足其和 ≥ s 的长度最小的 连续 子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

示例：

```
输入：s = 7, nums = [2,3,1,2,4,3]
输出：2
解释：子数组 [4,3] 是该条件下的长度最小的子数组。
```

进阶：

- 如果你已经完成了 O(n) 时间复杂度的解法, 请尝试 O(n log n) 时间复杂度的解法。

https://leetcode-cn.com/problems/minimum-size-subarray-sum/solution/chang-du-zui-xiao-de-zi-shu-zu-by-leetcode-solutio/

方法二：前缀和 + 二分查找

方法一的时间复杂度是 $O(n^2)$，因为在确定每个子数组的开始下标后，找到长度最小的子数组需要 $O(n)$ 的时间。如果使用二分查找，则可以将时间优化到 $O(\log n)$。

为了使用二分查找，需要额外创建一个数组 sums 用于存储数组 nums 的前缀和，其中 $\text{sums}[i]$ 表示从 $\text{nums}[0]$ 到 $\text{nums}[i-1]$ 的元素和。得到前缀和之后，对于每个开始下标 i，可通过二分查找得到大于或等于 i 的最小下标 $bound$，使得 $\text{sums}[bound] - \text{sums}[i-1] \geq s$，并更新子数组的最小长度（此时子数组的长度是 $bound - (i-1)$）。

因为这道题保证了数组中每个元素都为正，所以前缀和一定是递增的，这一点保证了二分的正确性。如果题目没有说明数组中每个元素都为正，这里就不能使用二分来查找这个位置了。

复杂度分析

- 时间复杂度：$O(n \log n)$，其中 n 是数组的长度。需要遍历每个下标作为子数组的开始下标，遍历的时间复杂度是 $O(n)$，对于每个开始下标，需要通过二分查找得到长度最小的子数组，二分查找得时间复杂度是 $O(\log n)$，因此总时间复杂度是 $O(n \log n)$。
- 空间复杂度：$O(n)$，其中 n 是数组的长度。额外创建数组 sums 存储前缀和。

更多硬核内容就在公众号「yes 的练级攻略」

方法 1：前缀和+二分

```
class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int n = nums.length, res = Integer.MAX_VALUE;
        int[] preSum = new int[n+1];
        for (int i = 1; i <= n; i++) {
            preSum[i] = preSum[i-1] + nums[i-1];
        }
        for (int i = 1; i <= n; i++){
            int target = s + preSum[i-1];
            int end = Arrays.binarySearch(preSum, target);
            if (end < 0) end = -(end + 1);
            if (end <= n) res = Math.min(res, end - i + 1);
        }
        return res == Integer.MAX_VALUE ? 0 : res;
    }
}
```

方法三：双指针

在方法一和方法二中，都是每次确定子数组的开始下标，然后得到长度最小的子数组，因此时间复杂度较高。为了降低时间复杂度，可以使用双指针的方法。

定义两个指针 $start$ 和 end 分别表示子数组的开始位置和结束位置，维护变量 sum 存储子数组中的元素和（即从 $\text{nums}[start]$ 到 $\text{nums}[end]$ 的元素和）。

初始状态下，$start$ 和 end 都指向下标 0，sum 的值为 0。

每一轮迭代，将 $\text{nums}[end]$ 加到 sum，如果 $sum \geq s$，则更新子数组的最小长度（此时子数组的长度是 $end - start + 1$），然后将 $\text{nums}[start]$ 从 sum 中减去并将 $start$ 右移，直到 $sum < s$，在此过程中同样更新子数组的最小长度。在每一轮迭代的最后，将 end 右移。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是数组的长度。指针 $start$ 和 end 最多各移动 n 次。
- 空间复杂度：$O(1)$。

方法 2：双指针

```
class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int n = nums.length, res = Integer.MAX_VALUE;
        int start = 0, end = 0, sum = 0;
        while (end < n){
            sum += nums[end];
            while (sum >= s) {
                res = Math.min(res, end - start + 1);
                sum -= nums[start];
                start++;
```

```
            }
            end++;
        }
        return res == Integer.MAX_VALUE ? 0 : res;
    }
}
```

43、最短无序连续子数组（581）【内置函数】【单调栈】【双指针】

> 581. 最短无序连续子数组
>
> 难度 简单 354 收藏 分享 切换为英文 关注 反馈
>
> 给定一个整数数组，你需要寻找一个**连续的子数组**，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。
>
> 你找到的子数组应是**最短**的，请输出它的长度。
>
> **示例 1:**
>
> **输入:** [2, 6, 4, 8, 10, 9, 15]
> **输出:** 5
> **解释:** 你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。
>
> **说明 :**
>
> 1. 输入的数组长度范围在 [1, 10,000]。
> 2. 输入的数组可能包含**重复**元素 ，所以**升序**的意思是<=。

https://leetcode-cn.com/problems/shortest-unsorted-continuous-subarray/solution/zui-duan-wu-xu-lian-xu-zi-shu-zu-by-leetcode/

> **方法 3：排序**
>
> **算法**
>
> 另一个简单的想法是：我们将数组 $nums$ 进行排序，记为 $nums_sorted$ 。然后我们比较 $nums$ 和 $nums_sorted$ 的元素来决定最左边和最右边不匹配的元素。它们之间的子数组就是要求的最短无序子数组。
>
> **复杂度分析**
>
> - 时间复杂度：$O(n \log n)$ 。排序消耗 $n \log n$ 的时间。
> - 空间复杂度：$O(n)$ 。我们拷贝了一份原数组来进行排序。

方法 1：用内置排序（8ms）

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        int[] arr = nums.clone();
        Arrays.sort(arr);
        int start = arr.length, end = 0;
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] != nums[i]) {
                start = Math.min(start, i);
                end = Math.max(end, i);
            }
        }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            return (end - start) >= 0 ? end - start + 1 : 0;
        }
    }
```

这个方法背后的想法仍然是选择排序。我们需要找到无序子数组中最小元素和最大元素分别对应的正确位置，来求得我们想要的无序子数组的边界。

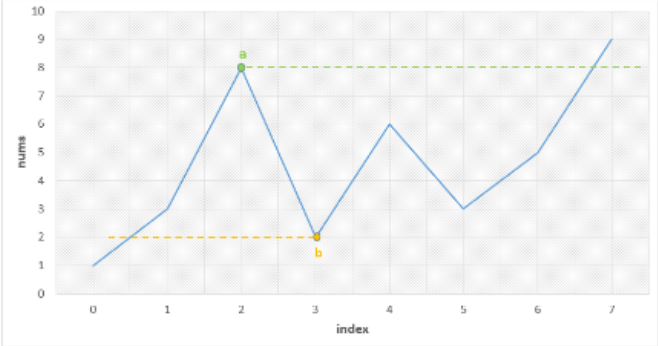
为了达到这一目的，此方法中，我们使用 栈 。我们从头遍历 $nums$ 数组，如果遇到的数字大小一直是升序的，我们就不断把对应的下标压入栈中，这么做的目的是因为这些元素在目前都是处于正确的位置上。一旦我们遇到前面的数比后面的数大，也就是 $nums[j]$ 比栈顶元素小，我们可以知道 $nums[j]$ 一定不在正确的位置上。

为了找到 $nums[j]$ 的正确位置，我们不断将栈顶元素弹出，直到栈顶元素比 $nums[j]$ 小，我们假设栈顶元素对应的下标为 k ，那么我们知道 $nums[j]$ 的正确位置下标应该是 $k+1$ 。

我们重复这一过程并遍历完整个数组，这样我们可以找到最小的 k， 它也是无序子数组的左边界。

类似的，我们逆序遍历一遍 $nums$ 数组来找到无序子数组的右边界。这一次我们将降序的元素压入栈中，如果遇到一个升序的元素，我们像上面所述的方法一样不断将栈顶元素弹出，直到找到一个更大的元素，以此找到无序子数组的右边界。

我们可以看下图作为参考。我们观察到上升还是下降决定了相对顺序，我们还可以观察到指针 b 在下标 0 后面标记着无序子数组的左边界，指针 a 在下标 7 前面标记着无序子数组的右边界。



复杂度分析

- 时间复杂度：$O(n)$。需要遍历数组一遍，栈的时间复杂度也为 $O(n)$。
- 空间复杂度：$O(n)$。栈的大小最大达到 n。

方法 2：单调栈（10s）

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        Deque<Integer> stack = new ArrayDeque<>();
        int left = nums.length, right = 0;
        for (int i = 0; i < nums.length; i++){
            while (!stack.isEmpty() && nums[stack.peek()] > nums[i])
                left = Math.min(left, stack.pop());
            stack.push(i);
        }
        stack.clear();
        for (int i = nums.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && nums[stack.peek()] < nums[i])
                right = Math.max(right, stack.pop());
            stack.push(i);
```

```
        }
        return right - left > 0 ? right - left + 1 : 0;
    }
}
```

方法 3：双指针，同时遍历

同时从前往后和从后往前遍历，分别得到要排序数组的右边界和左边界；
寻找右边界：
从前往后遍历的过程中，用**max记录遍历过的最大值**，如果max大于当前的nums[i]，说明nums[i]的位置不正确，属于需要排序的数组，因此将右边界更新为i，然后更新max；这样最终可以找到需要排序的数组的右边界，右边界之后的元素都大于max；
寻找左边界：
从后往前遍历的过程中，用**min记录遍历过的最小值**，如果min小于当前的nums[j]，说明nums[j]的位置不正确，应该属于需要排序的数组，因此将左边界更新为j，然后更新min；这样最终可以找到需要排序的数组的左边界，左边界之前的元素都小于min；
（从前往后遍历和从后往前遍历两个过程可以分两次循环完成，也可以放一起完成，这样的话就有：j=len-i-1）

```
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        int len = nums.length;
        int max = nums[0], min = nums[len - 1];
        int left = 0, right = -1;
        for (int i = 0; i < len; i++) {
            if (max > nums[i]) right = i;
            else max = nums[i];
            if (min < nums[len - 1 - i]) left = len - 1 - i;
            else min = nums[len - 1 - i];
        }
        return right - left + 1;
    }
}
```

44、分割数组的最大值（410）【二分+贪心】【动规】

410. 分割数组的最大值

难度 困难 205 收藏 分享 切换为英文 关注 反馈

给定一个非负整数数组和一个整数 *m*，你需要将这个数组分成 *m* 个非空的连续子数组。设计一个算法使得这 *m* 个子数组各自和的最大值最小。

注意:
数组长度 *n* 满足以下条件:

- 1 ≤ *n* ≤ 1000
- 1 ≤ *m* ≤ min(50, *n*)

示例:

```
输入:
nums = [7,2,5,10,8]
m = 2

输出:
18

解释:
一共有四种方法将nums分割为2个子数组。
其中最好的方式是将其分为[7,2,5] 和 [10,8]，
因为此时这两个子数组各自的和的最大值为18，在所有情况中最小。
```

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/split-array-largest-sum/solution/er-fen-cha-zhao-by-liweiwei1419-4/

重点理解：

1. **大于的时候舍弃，小于等于的时候保留，这样左右向中间逼近能找到 `M` 的最小值**，具体可以看题解中唯一的那张图；
2. 理解贪心算法的应用，在从左向右划分组的时候，我们的方案是尽量让一个组有更多的元素，直至超过了设定的临界值（很遗憾，证明已经超出了我的能力范围，欢迎知道的朋友们指正）。

注意题目中给出的这 3 个条件：

1. 数组中的元素均是「非负整数」；
2. 子数组的特点是：「非空」且「连续」；
3. 恰好分成 `m` 个非空「非空连续子数组」。

题目中还给出了一个概念：「连续子数组各自和的最大值」，我们用一个变量 `maxIntervalSum` 表示。不难知道：

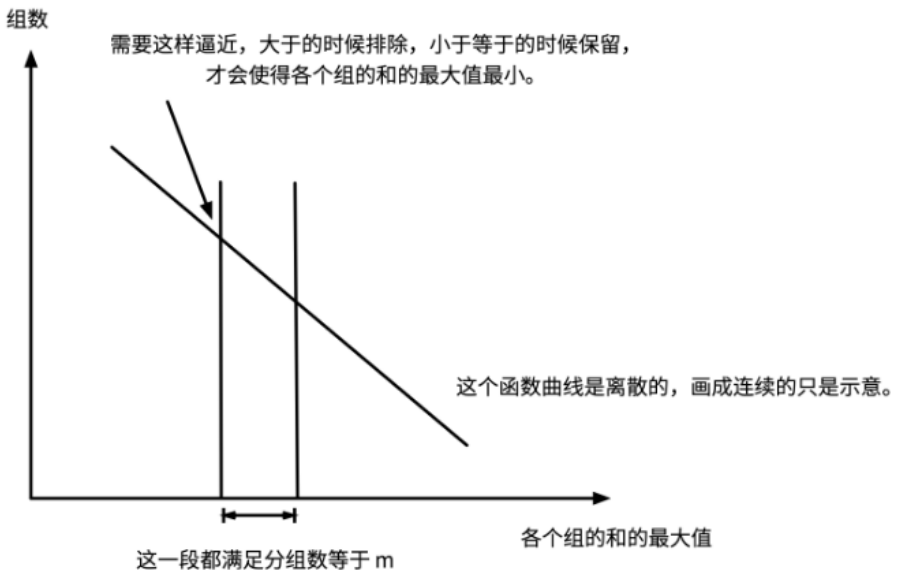
- 每一个「非空连续子数组」如果包含的元素个数越多，那么 `maxIntervalSum` 就可能越大（非负整数保证）；
- **一个 `maxIntervalSum` 的数值就唯一对应了一个分出的「非空连续子数组」的组数 `M`，它们是函数关系（一一对应），`maxIntervalSum` 是自变量，`M` 是因变量，可以写成：**

$$M = function(maxIntervalSum)$$

如何找到一个 `maxIntervalSum`，使得它对应的 `M` 恰好等于题目给出的 `m`。但是我们容易分析出，**这个函数是一个单调递减的函数：**

- 如果 `maxIntervalSum` 越小，分出的「非空连续子数组」的组数 `M` 就越大；
- 如果 `maxIntervalSum` 越大，分出的「非空连续子数组」的组数 `M` 就越小。

因此这个函数是单调不增的函数。



原因就在于上面强调的题目中给出的 2 个条件：非负整数和非空连续子数组，**由于这种单调性，可以使用二分查找，找到与 `m` 对应的 `maxIntervalSum`。**

参考代码：

下面是一些细节：

- **只要连续加起来的数值超过了 `maxIntervalSum`，就新产生一个新的连续子数组；**
- `maxIntervalSum` 的最小值是这个数组中的最大值，这是因为 `max(nums)` 一定会被分到其中一组；
- `maxIntervalSum` 的最大值是这个数组中所有元素的和，极端情况就是题目中给出 $m = 1$ 的时候。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析：

- 时间复杂度：$O(N \log \sum nums)$，这里 N 表示输入数组的长度，$\sum nums$ 表示输入数组的和，代码在 $[\max(nums), \sum nums]$ 区间里使用二分查找找到目标元素，而每一次判断分支需要遍历一遍数组，时间复杂度为 $O(N)$；
- 空间复杂度：$O(1)$，只使用到常数个临时变量。

总结

理解单调性

- 一个重要的性质：分割数越大，「子数组各自的和的最大值」就越小（非递增，满足单调性），因此可以使用二分查找，定位分割数；
- 一种「分割方案（分成几个子数组）」对应了一个「子数组各自的和的最大值」；
- 反过来，一个「子数组各自的和的最大值」对应了一种「分割方案（分成几个子数组）」；
- 它们是一一对应的关系（关键）。

思路整理（绕了个弯子去逼近）

- 先找「子数组各自的和的最大值」的中间数（尝试得到的一个值），看看它对应的「分割方案（分成几个子数组）」是多少；
- 如果「分割方案（分成几个子数组）」比题目要求的 `m` 还多，说明「子数组各自的和的最大值」还太小，所以下一次搜索应该至少是中位数 `+ 1`（`left = mid + 1`），它的反面即至多是中位数（`right = mid`）。

方法 1：二分+贪心

```
class Solution {
    public int splitArray(int[] nums, int m) {
        int max = 0, sum = 0;
         // 计算「子数组各自的和的最大值」的上下界
        for (int num : nums){
            max = Math.max(num, max);
            sum += num;
        }
        // 使用「二分查找」确定一个恰当的「子数组各自的和的最大值」，
        // 使得它对应的「子数组的分割数」恰好等于 m
        int left = max, right = sum;
        while (left < right){
            int mid = left + (right - left) / 2;
            int splits = split(nums, mid);
            if (splits > m) {
                // 如果分割数太多，说明「子数组各自的和的最大值」太小，此时需要将「子数组各自的和的最大值」调大
                // 下一轮搜索的区间是 [mid + 1, right]
                left = mid + 1;
            } else {
                // 下一轮搜索的区间是上一轮的反面区间 [left, mid]
                right = mid;
            }
        }
        return left;
```

```
    }
    /***
     *
     * @param nums 原始数组
     * @param maxIntervalSum 子数组各自的和的最大值
     * @return 满足不超过「子数组各自的和的最大值」的分割数
     */
    private int split(int[] nums, int maxIntervalSum) {
        // 至少是一个分割
        int splits = 1;
        // 当前区间的和
        int curIntervalSum = 0;
        for (int num : nums) {
            // 尝试加上当前遍历的这个数，如果加上去超过了「子数组各自的和的最大值」，就不加这个数，另起炉灶
            if (curIntervalSum + num > maxIntervalSum) {
                curIntervalSum = 0;
                splits++;
            }
            curIntervalSum += num;
        }
        return splits;
    }
}
```

方法 2 链接：

https://leetcode-cn.com/problems/split-array-largest-sum/solution/fen-ge-shu-zu-de-zui-da-zhi-by-leetcode-solution/

方法一：动态规划

思路与算法

「将数组分割为 m 段，求……」是动态规划题目常见的问法。

本题中，我们可以令 $f[i][j]$ 表示将数组的前 i 个数分割为 j 段所能得到的最大连续子数组和的最小值。在进行状态转移时，我们可以考虑第 j 段的具体范围，即我们可以枚举 k，其中前 k 个数被分割为 $j-1$ 段，而第 $k+1$ 到第 i 个数为第 j 段。此时，这 j 段子数组中和的最大值，就等于 $f[k][j-1]$ 与 $sub(k+1,i)$ 中的较大值，其中 $sub(i,j)$ 表示数组 $nums$ 中下标落在区间 $[i,j]$ 内的数的和。

由于我们要使得子数组中和的最大值最小，因此可以列出如下的状态转移方程：

$$f[i][j]=\min_{k=0}^{i-1}\left\{\max(f[k][j-1],sub(k+1,i))\right\}$$

对于状态 $f[i][j]$，由于我们不能分出空的子数组，因此合法的状态必须有 $i \geq j$。对于不合法（$i<j$）的状态，由于我们的目标是求出最小值，因此可以将这些状态全部初始化为一个很大的数。在上述的状态转移方程中，一旦我们尝试从不合法的状态 $f[k][j-1]$ 进行转移，那么 $\max(\cdots)$ 将会是一个很大的数，就不会对最外层的 $\min\{\cdots\}$ 产生任何影响。

此外，我们还需要将 $f[0][0]$ 的值初始化为 0。在上述的状态转移方程中，当 $j=1$ 时，唯一的可能性就是前 i 个数被分成了一段。如果枚举的 $k=0$，那么就代表着这种情况；如果 $k \neq 0$，对应的状态 $f[k][0]$ 是一个不合法的状态，无法进行转移。因此我们需要令 $f[0][0]=0$。

最终的答案即为 $f[n][m]$。

方法 2：动态规划

```
class Solution {
    public int splitArray(int[] nums, int m) {
        int n = nums.length;
        int[][] f = new int[n + 1][m + 1];
        for (int i = 0; i <= n; i++) {
            Arrays.fill(f[i], Integer.MAX_VALUE);
        }
        int[] sub = new int[n + 1];
        for (int i = 0; i < n; i++) {
            sub[i + 1] = sub[i] + nums[i];
        }
        f[0][0] = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= Math.min(i, m); j++) {
                for (int k = 0; k < i; k++) {
                    f[i][j] = Math.min(f[i][j], Math.max(f[k][j - 1], sub[i] - sub[k]));
                }
            }
        }
        return f[n][m];
    }
}
```

87、最长重复子数组（718）

718. 最长重复子数组

难度 中等 250 收藏 分享 切换为英文 关注 反馈

给两个整数数组 A 和 B ，返回两个数组中公共的、长度最长的子数组的长度。

示例：

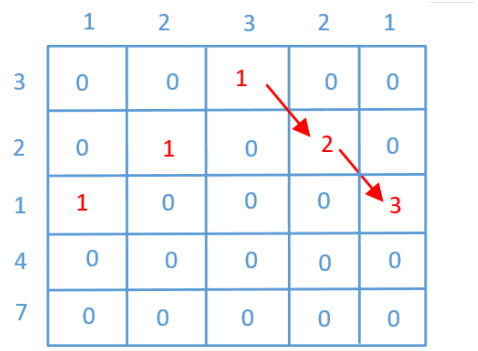
输入：
A: [1,2,3,2,1]
B: [3,2,1,4,7]
输出： 3
解释：
长度最长的公共子数组是 [3, 2, 1] 。

提示：

- 1 <= len(A), len(B) <= 1000
- 0 <= A[i], B[i] < 100

https://leetcode-cn.com/problems/maximum-length-of-repeated-subarray/solution/javadong-tai-gui-hua-jie-jue-qi-shi-jiu-shi-zui-ch/

更多硬核内容就在公众号「yes 的练级攻略」



方法 1：用二维数组（82ms）

```
class Solution {
    public int findLength(int[] A, int[] B) {
        int max = 0;
        int[][] dp = new int[A.length + 1][B.length + 1];
        for (int i = 1; i <= A.length; i++) {
            for (int j = 1; j <= B.length; j++) {
                if (A[i - 1] == B[j - 1])
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                else
                    dp[i][j] = 0;
                max = Math.max(max, dp[i][j]);
            }
        }
        return max;
    }
}
```

方法 2：缩减 dp 成一位数组（48ms）

```
class Solution {
    public int findLength(int[] A, int[] B) {
        int max = 0;
        int[] dp = new int[B.length + 1];
        for (int i = 1; i <= A.length; i++) {
            for (int j = B.length; j > 0 ; j--) {
                if (A[i - 1] == B[j - 1])
                    dp[j] = dp[j - 1] + 1;
                else
                    dp[j] = 0;
//如果用 if (dp[j] > max) max = dp[j];可以省去每次都赋值的时间
                max = Math.max(max, dp[j]);
            }
        }
        return max;
    }
}
```

方法 3：不是逆序的，而是存放临时值，与下一题类似（63ms）

```
class Solution {
    public int findLength(int[] A, int[] B) {
        int max = 0;
        int[] dp = new int[B.length + 1];
        int tmp = 0;
        for (int i = 1; i <= A.length; i++) {
            int last = 0; ////存放上一次的数据
            for (int j = 1; j <= B.length ; j++) {
                tmp = dp[j]; //dp[j]这个值会被替换，所以替换之前要把他保存下
来
                if (A[i - 1] == B[j - 1])
                    dp[j] = last + 1;
                else
                    dp[j] = 0;
                if (dp[j] > max) max = dp[j];
                last = tmp; // 更新 last
            }
        }
        return max;
    }
}
```

45、电话目录管理系统（379）

公众号：
yes的练级攻略

379. 电话目录管理系统

难度 中等 16 收藏 分享 切换为英文 关注

设计一个电话目录管理系统，让它支持以下功能：

1. `get`：分配给用户一个未被使用的电话号码，获取失败请返回 -1
2. `check`：检查指定的电话号码是否被使用
3. `release`：释放掉一个电话号码，使其能够重新被分配

示例：

```
// 初始化电话目录，包括 3 个电话号码：0，1 和 2。
PhoneDirectory directory = new PhoneDirectory(3);

// 可以返回任意未分配的号码，这里我们假设它返回 0。
directory.get();

// 假设，函数返回 1。
directory.get();

// 号码 2 未分配，所以返回为 true。
directory.check(2);

// 返回 2，分配后，只剩一个号码未被分配。
directory.get();

// 此时，号码 2 已经被分配，所以返回 false。
directory.check(2);

// 释放号码 2，将该号码变回未分配状态。
directory.release(2);

// 号码 2 现在是未分配状态，所以返回 true。
directory.check(2);
```

提示：

- `1 <= maxNumbers <= 10^4`
- `0 <= number < maxNumbers`
- 调用方法的总数处于区间 `[0 - 20000]` 之内

https://leetcode-cn.com/problems/design-phone-directory/solution/379-dian-hua-mu-lu-guan-li-xi-tong-by-klb/

解题思路

题目的构造器是输入一个 int 类型数字，则电话号码指的是：0、1、...、n-1 这些数字。

定义一个布尔类型的数组 phone[]，容量为电话号码数量，这个数组的索引就是电话号码，值为 true 表示可未被使用，值为 false 表示已被使用；

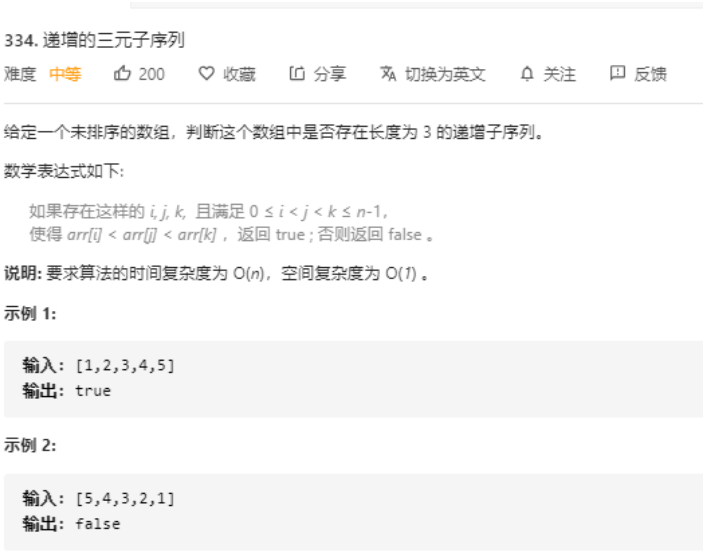
1、get: 遍历 phone[]，返回第一个值为 true 的索引；

2、check：先 phone[i] = false，再返回 phone[i]；

3、release：phone[i] = true。

```
class PhoneDirectory {
    /** Initialize your data structure here
        @param maxNumbers - The maximum numbers that can be stored in the phone
 directory. */
    private boolean[] phone;
    private int size;
    public PhoneDirectory(int maxNumbers) {
        this.size = maxNumbers;
        phone = new boolean[maxNumbers];
        Arrays.fill(phone, true); // true 表示可以使用
    }
    /** Provide a number which is not assigned to anyone.
        @return - Return an available number. Return -1 if none is available. *
/
    public int get() {
        for (int i = 0; i < size; i++) {
            if (phone[i]) {
                phone[i] = false;
                return i;
            }
        }
        return -1;
    }
    /** Check if a number is available or not. */
    public boolean check(int number) {
        return phone[number];
    }
    /** Recycle or release a number. */
    public void release(int number) {
        phone[number] = true;
    }
}
/**
 * Your PhoneDirectory object will be instantiated and called as such:
 * PhoneDirectory obj = new PhoneDirectory(maxNumbers);
 * int param_1 = obj.get();
 * boolean param_2 = obj.check(number);
 * obj.release(number);
 */
```

46、递增的三元子序列（334）



334. 递增的三元子序列

难度 中等 200 收藏 分享 切换为英文 关注 反馈

给定一个未排序的数组，判断这个数组中是否存在长度为 3 的递增子序列。

数学表达式如下:

如果存在这样的 i, j, k, 且满足 0 ≤ i < j < k ≤ n-1，
使得 arr[i] < arr[j] < arr[k] ，返回 true ; 否则返回 false 。

说明: 要求算法的时间复杂度为 O(n)，空间复杂度为 O(1) 。

示例 1:

输入: [1,2,3,4,5]
输出: true

示例 2:

输入: [5,4,3,2,1]
输出: false

https://leetcode-cn.com/problems/increasing-triplet-subsequence/solution/java-0ms-100-by-shi-wen/

```
public class Solution {
    public boolean increasingTriplet(int[] nums) {
        int len = nums.length;
        if(len < 3) return false;
        int min = Integer.MAX_VALUE; //最小值
        int second = Integer.MAX_VALUE;//第二元素最小的递增对的第二元素
        for(int num : nums){
            if(num <= min){//num 比 min 小或相等，肯定不存在递增三元素。由于不存在以 num 结尾的递增对，故只需更新 min
                min = num;
            }else if(num <= second){//num 比 second 小或相对，肯定不存在递增三元素。由于存在以 num 结尾的递增对且 num 不大于 second，因此可以更新 second
                second = num;
            }else {//num 比 second 大，那就存在递增三元素，因为 second 是一个递增对的第二元素，加上 num 后就是递增三元素了
                return true;
            }
        }
        return false;
    }
}
```

47、摆动排序（280）

280. 摆动排序

难度 中等 29 收藏 分享 切换为英文 关注 反馈

给你一个无序的数组 nums，将该数字 原地 重排后使得 nums[0] <= nums[1] >= nums[2] <= nums[3]...。

示例:

输入: nums = [3,5,2,1,6,4]
输出: 一个可能的解答是 [3,5,1,6,2,4]

https://leetcode-cn.com/problems/wiggle-sort/solution/java-liang-chong-jie-fa-by-npe_tle-2/

方法二 直接交换

思路

还是看奇数索引，如果奇数索引的数字比前面一个数字小，则两者进行交换，交换完之后，再跟后面一个元素比，如果小于后面一个元素的话，则二者再进行交换。这样遍历完所有的奇数索引就会生成一个摆动排序后的结果。如$[2,1,3]$，我们判断1比2小，交换后变成$[1,2,3]$，这时候又发现2比3小，再进行交换变成$[1,3,2]$，这样就满足要求了。

```
class Solution {
    private void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
    public void wiggleSort(int[] nums) {
        int len = nums.length;
        for (int i = 1; i < len; i += 2) {
            if (nums[i] < nums[i-1]) swap(nums, i, i-1);
            if (i < len - 1 && nums[i] < nums[i+1]) swap(nums, i, i+1);
        }
    }
}
```

48、摆动排序 2（376）【峰与谷（面试题 10.11）】

更多硬核内容就在公众号「yes 的练级攻略」

324. 摆动排序 II

难度 中等 174 收藏 分享 切换为英文 关注 反馈

给定一个无序的数组 `nums`，将它重新排列成 `nums[0] < nums[1] > nums[2] < nums[3]...` 的顺序。

示例 1:

```
输入: nums = [1, 5, 1, 1, 6, 4]
输出: 一个可能的答案是 [1, 4, 1, 5, 1, 6]
```

示例 2:

```
输入: nums = [1, 3, 2, 2, 3, 1]
输出: 一个可能的答案是 [2, 3, 1, 3, 1, 2]
```

说明:
你可以假设所有输入都会得到有效的结果。

进阶:
你能用 O(n) 时间复杂度和 / 或原地 O(1) 额外空间来实现吗?

面试题 10.11. 峰与谷

难度 中等 18 收藏 分享 切换为英文 关注 反馈

在一个整数数组中，“峰”是大于或等于相邻整数的元素，相应地，“谷”是小于或等于相邻整数的元素。例如，在数组{5, 8, 6, 2, 3, 4, 6}中，{8, 6}是峰， {5, 2}是谷。现在给定一个整数数组，将该数组按峰与谷的交替顺序排序。

示例:

```
输入: [5, 3, 1, 2, 3]
输出: [5, 1, 3, 2, 3]
```

提示:

- `nums.length <= 10000`

https://leetcode-cn.com/problems/wiggle-sort-ii/solution/javaxiang-xi-ti-jie-shuo-ming-by-heator/

yes的练级攻略

有一个数组[a1,a2,...,an]，我们怎么把它排成摇摆序列呢?

由摇摆序列的定义: `nums[0] < nums[1] > nums[2] < nums[3]...` ，我们知道了可以分成较大一部分的数和较小一部分数，然后互相穿插即可。比如一个数组排序后为： `A=[a1,a2,...,an]` `(a1<=a2<=....<=an)` ，然后分成较小和较大的两部分 `[a1,a2,...,a(n/2)]` ， `[a(n/2+1),...,an]` (数组长度为奇数时不影响) ，再进行穿插操作。

那是不是穿插成[a1,a(n/2+1),a2,a(n/2+2),...,an]就行了呢?

其实不对，可以验证特殊情况：n比较小时且为偶数时，穿插后的序列需要满足 `a(n/2+1)>a2` ，如果 `a1<a2<=a(n/2+1)<an` ， `a(n/2+1)` 正好是 `a2` 的后一项且与 `a2` 相等呢? 即如果是 `[4,5,5,6]` 的情况呢?
那就分成了 `[4,5]` ， `[5,6]` 两部分，之后穿插成的是 `[4,5,5,6]` 并不是摇摆序列。

应该怎样排列呢?

1. 把较小和较大的两部分 `[a1,a2,...,a(n/2)]` ， `[a(n/2+1),...,an]` 翻转成 `[a(n/2),...,a1]` ， `[an,...,a(n/2+1)]`
2. 排列成 `B=[a(n/2),a(n),a(n/2-1),a(n-1),...,a1,a(n/2+1)]` 即可。

这样相邻两个数一定不存在相等的情况，而且满足摇摆序列的要求。

- `n=1` 时显然正确
- `n=2`或`n=3` 时也显然正确。
- `n>=4`时 :
  1. 首先，判断相邻的情况，如果B中现在相邻两数在原来的有序序列A也相邻，此时令 `n-n/2=1` ， `(n-1)-(n/2-1)=1` ，... 都解得 `n=2` ，意思是 `n=2` 是他们的公共解，而 `n>=4` 时， `n-n/2` ， `(n-1)-(n/2-1)` 等于 `n/2>=2` ，说明B中相邻两数一定在原来的有序序列A中不相邻;

  2. 有人会说也存在原本(在A中)不相邻但是相等的情况，即几个数连续相等，但其中两个相等的数并不相邻，比如 `aq=a(q+1)=a(q+2)` ， `aq=a(q+2)` 相等但是不相邻，但是我们知道如果 `a(n/2-k)=a(n-k)` 即不相邻相等，那么由于他们中间的所有数都要相等，而且与这两个数相等才能保证不相邻相等：即 `a(n/2-k)=a(n/2-k+1)=a(n/2-k+2)=...=a(n-k-2)=a(n-k-1)=a(n-k)` ，一共 `n/2+1` 个数相等。

  3. 而不相邻相等这种情况是不行的：虽然只有 `a(n/2-k),a(n-k)` (在B中相邻) 在原序列A中不相邻相等，但也违背了摇摆序列的定义，所以这道题有一个隐藏的条件：输入序列在排序后最多只有序列长度一半(`n/2`)的相邻的数连续相等。

- 在这个条件下：这道题按此排列就没问题。大家可以用✓验证一下。

```
class Solution {
    /**
     * 先排序再穿插
     * O(nlogn)+O(n)=O(nlogn)
     * @param nums
     */
    public void wiggleSort(int[] nums) {
        //排序
        Arrays.sort(nums);
        int len = nums.length, i = 0;
        int[] smaller=new int[len%2==0?len/2:(len/2+1)],bigger=new int[len/2];
        //复制
        System.arraycopy(nums,0,smaller,0,smaller.length);
        System.arraycopy(nums,smaller.length,bigger,0,len/2);
        //穿插
        for (; i < len / 2; i++) {
            nums[2*i]=smaller[smaller.length-1-i];
            nums[2*i+1]=bigger[len/2-1-i];
        }
```

```
            if (len %2!=0) nums[2*i]=smaller[smaller.length-1-i];
        }
    }
```

49、矩阵置零（73）【零矩阵（面试题 01.08）】

73. 矩阵置零

难度 中等 290 收藏 分享 切换为英文 关注 反馈

给定一个 *m* x *n* 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用原地算法。

示例 1:

输入：

```
[
  [1,1,1],
  [1,0,1],
  [1,1,1]
]
```

输出：

```
[
  [1,0,1],
  [0,0,0],
  [1,0,1]
]
```

示例 2:

输入：

```
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
```

输出：

```
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]
```

进阶:

- 一个直接的解决方案是使用 O(*mn*) 的额外空间，但这并不是一个好的解决方案。
- 一个简单的改进方案是使用 O(*m* + *n*) 的额外空间，但这仍然不是最好的解决方案。
- 你能想出一个常数空间的解决方案吗?

https://leetcode-cn.com/problems/zero-matrix-lcci/solution/huo-yong-shou-xing-shou-lie-yuan-su-zuo-wei-gai-xi/

和73题撞题了

步骤如下：

1、矩阵中某个数为零，则将该数所在行的第一个数置零，所在列的第一个数置零，即matrix[0][j] = matrix[i][0] = 0

2、遍历上面处理过的首行和首列的每个元素（[0][0]位置除外），如果为零，则说明该行或者该列应该都为零，将该行或列全部元素置零

3、第一步操作会让首行首列是否有零这个信息损失掉，因为首行首列被用来存储其他信息了，会改变他们的取值，所以再定义两个变量row0,col0记录首行首列是否有零，最后根据row0和col0来处理首行和首列自己

```
class Solution {
```

```
    public void setZeroes(int[][] matrix) {
  int m=matrix.length;
        int n=matrix[0].length;
        boolean row0=false,cow0=false;
        //因为第一行和第一列被征用了，所以原始状态需要提前记录下来
        for(int i=0;i<m;i++){//首列
            if(matrix[i][0]==0) cow0=true;
        }
        for(int j=0;j<n;j++){//首行
            if(matrix[0][j]==0) row0=true;
        }
        for(int i=1;i<m;i++){
            for(int j=1;j<n;j++){
                if(matrix[i][j]==0){//第 i 行，第 j 列都需要置零，所以把首列第
i 行和首行第 j 列标记上
                    matrix[i][0]=0;
                    matrix[0][j]=0;
                }
            }
        }
        for(int i=1;i<m;i++){
            if(matrix[i][0]==0) matrix[i]=new int[n];
        }
        for(int j=1;j<n;j++){
            if(matrix[0][j]==0){
                for(int i=0;i<m;i++) matrix[i][j]=0;
            }
        }
        if(row0) matrix[0]=new int[n];
        if(cow0){
            for(int i=0;i<m;i++) matrix[i][0]=0;
        }
    }
}
```

十一、二分

1. 二分的总结【找数字，找左边界，找右边界】

labuladong 已关注

东哥手把手带你刷 LeetCode，感受支配算法的乐趣！

收起 ^

前导部分来源于这个公众号

更多硬核内容就在公众号「yes 的练级攻略」

第一个，最基本的二分查找算法：

```
因为我们初始化 right = nums.length - 1
所以决定了我们的「搜索区间」是 [left, right]
所以决定了 while (left <= right)
同时也决定了 left = mid+1 和 right = mid-1

因为我们只需找到一个 target 的索引即可
所以当 nums[mid] == target 时可以立即返回
```

```
int binarySearch(int[] nums, int target) {
    int left = 0;
```

67

我写了首诗，让你闭着眼睛也能写对二分搜索

```
    int right = nums.length - 1; // 注意

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
    }
    return -1;
}
```

第二个，寻找左侧边界的二分查找：

```
因为我们初始化 right = nums.length
```

78

我写了首诗，让你闭着眼睛也能写对二分搜索

```
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid + 1 和 right = mid

因为我们需找到 target 的最左侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧右侧边界以锁定左侧边界
```

用
：
级攻略

更多硬核内容就在公众号「yes 的练级攻略」

二、寻找左侧边界的二分搜索

以下是最常见的代码形式，其中的标记是需要注意的细节：

```
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length; // 注意

    while (left < right) { // 注意
        int mid = (left + right) / 2;
        if (nums[mid] == target) {
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid; // 注意
        }
    }
    return left;
}
```

改成两端都是闭区间以后

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定左侧边界
            right = mid - 1;
        }
    }
    // 最后要检查 left 越界的情况
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}
```

第三个，寻找右侧边界的二分查找：

```
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid + 1 和 right = mid

因为我们需找到 target 的最右侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧左侧边界以锁定右侧边界

又因为收紧左侧边界时必须 left = mid + 1
所以最后无论返回 left 还是 right，必须减一
```

三、寻找右侧边界的二分查找

类似寻找左侧边界的算法，这里也会提供两种写法，还是先写常见的左闭右开的写法，只有两处和搜索左侧边界不同，已标注：

```
int right_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = (left + right) / 2;
        if (nums[mid] == target) {
            left = mid + 1; // 注意
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left - 1; // 注意
}
```

改成两端都是闭区间以后

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回，锁定右侧边界
            left = mid + 1;
        }
    }
    // 最后要检查 right 越界的情况
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

2. 寻找两个正序数组的中位数（4）

4. 寻找两个正序数组的中位数

难度 困难 2767 收藏 分享 切换为英文 关注 反馈

给定两个大小为 m 和 n 的正序（从小到大）数组 `nums1` 和 `nums2`。

请你找出这两个正序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

你可以假设 `nums1` 和 `nums2` 不会同时为空。

示例 1:

```
nums1 = [1, 3]
nums2 = [2]

则中位数是 2.0
```

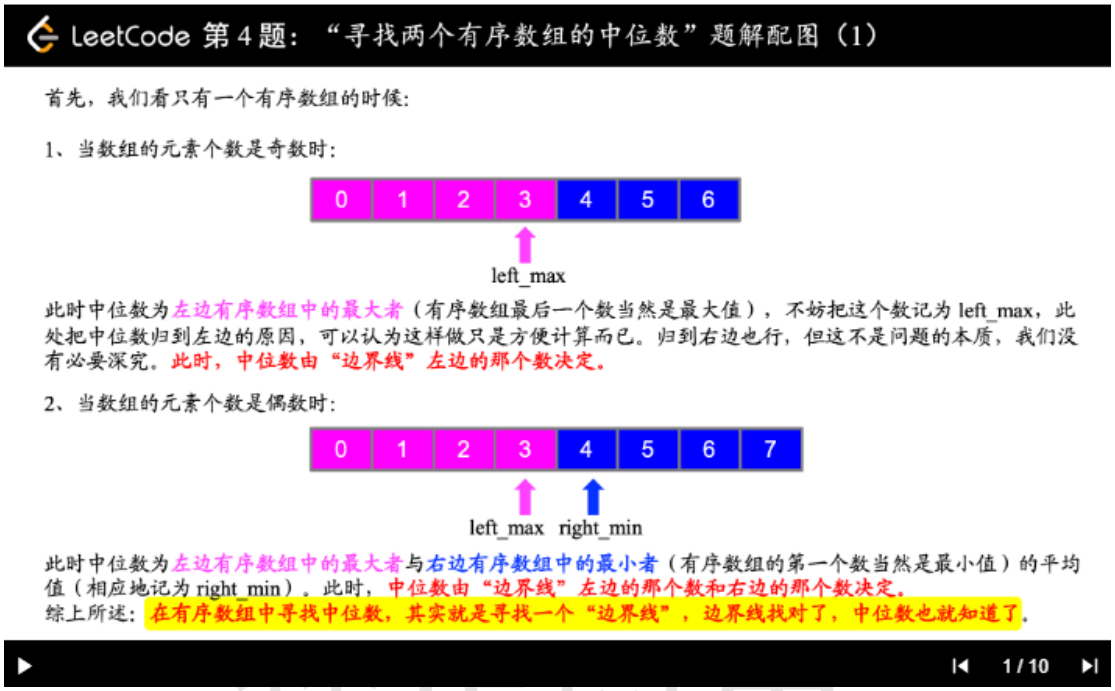
示例 2:

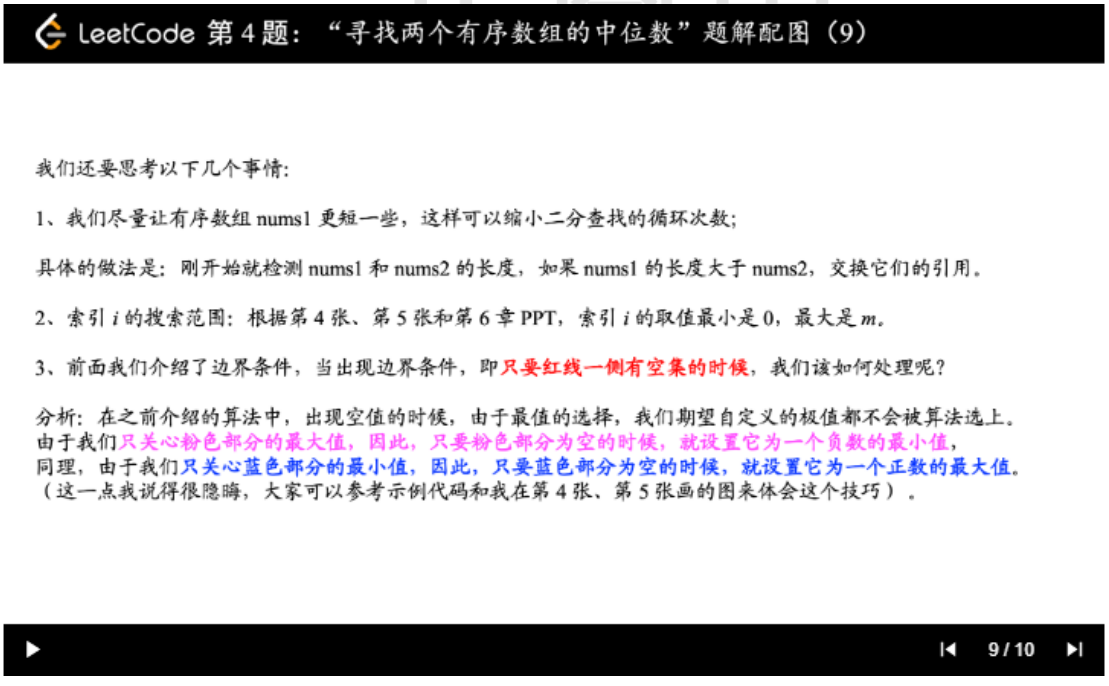
```
nums1 = [1, 2]
nums2 = [3, 4]

则中位数是 (2 + 3)/2 = 2.5
```

https://leetcode-cn.com/problems/median-of-two-sorted-arrays/solution/he-bing-yi-hou-zhao-g

更多硬核内容就在公众号「yes 的练级攻略」

ui-bing-guo-cheng-zhong-zhao-/





索引 i 的搜索范围

更多硬核内容就在公众号「yes 的练级攻略」

LeetCode 第 4 题：“寻找两个有序数组的中位数”题解配图（4）

核心思想：

给两个有序数组中间画一条红线，这条红线得满足如下条件：

条件 ①：红线左边所有的数都小于等于红线右边所有的数；

条件 ②：如果记红线左边所有的数的个数为 $size_{左}$，红线右边所有的数的个数为 $size_{右}$，则

（1）当两个数组的元素个数之和为奇数的时候，有：$size_{左} = size_{右} + 1$，如图 2；
（2）当两个数组的元素个数之和为偶数的时候，有：$size_{左} = size_{右}$，如图 1。

说明：以下以及以后的幻灯片中，
使用 m 表示数组 nums1 的元素个数，即 m = len(nums1)。
使用 n 表示数组 nums2 的元素个数，即 n = len(nums2)。

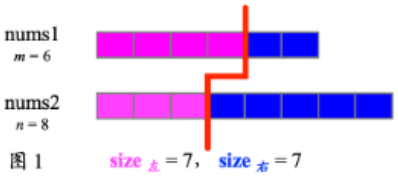


图 1 $size_{左} = 7$，$size_{右} = 7$

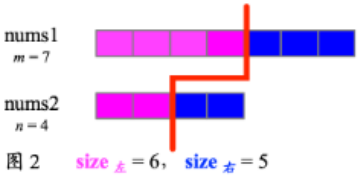


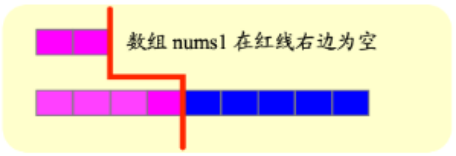
图 2 $size_{左} = 6$，$size_{右} = 5$

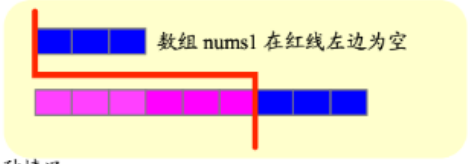
4 / 10

LeetCode 第 4 题：“寻找两个有序数组的中位数”题解配图（5）

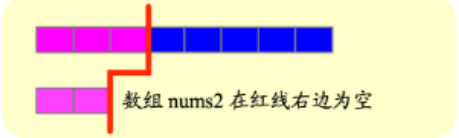
以下是一些边界条件，相信读者心里也能想到，我们只标注边界的部分，在编码的时候会使用一个小技巧进行处理。

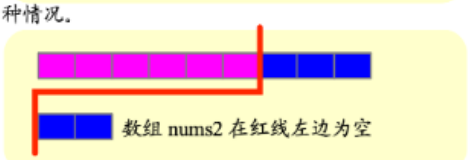
边界情况 ①：数组 nums1 长度短的情况下，又分为以下两种情况。



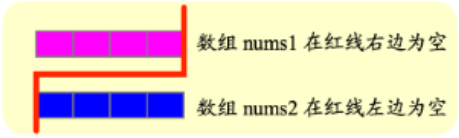


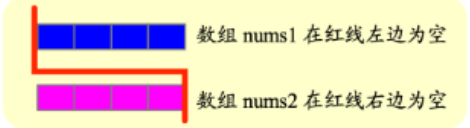
边界情况 ②：数组 nums1 长度长的情况下，又分为以下两种情况。





边界情况 ③：数组 nums1 长度与数组 nums2 的长度相等时，又分为以下两种情况（最麻烦）。





5 / 10

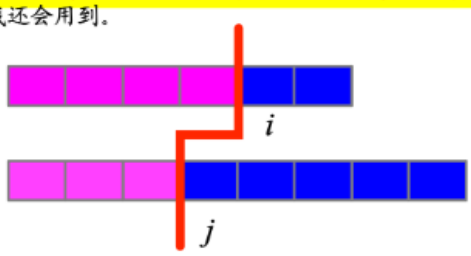
更多硬核内容就在公众号「yes 的练级攻略」

LeetCode 第 4 题：“寻找两个有序数组的中位数”题解配图（6）

画“红线”，等价于在两个有序数组中分别寻找两个索引 i 和 j，那么，索引 i 和 j 有没有什么关系呢？
根据条件②：
（1）当两个数组的元素个数之和为奇数的时候，有：$size_{左} = size_{右} + 1$；
（2）当两个数组的元素个数之和为偶数的时候，有：$size_{左} = size_{右}$。

我们记
数组 nums1 在“红线”右边的第 1 个数的索引为 i，它前面的数的个数也为 i，索引编号从 0 开始；
数组 nums2 在“红线”右边的第 1 个数的索引为 j，它前面的数的个数也为 j，索引编号从 0 开始。

此时有：$size_{左} = i + j$，这一点是不管两个数组的元素个数之和为奇数或者偶数时都成立的，这个结论请读者先有个印象，下个 PPT 我还会用到。

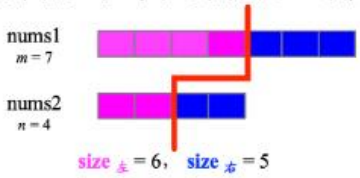


6 / 10

j 的取值

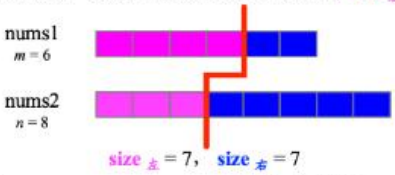
LeetCode 第 4 题：“寻找两个有序数组的中位数”题解配图（7）

（1）当两个数组的元素个数之和为奇数的时候，可以拿下边的图验证，此时有 $size_{左} = \frac{m+n+1}{2}$



$size_{左} = 6$，$size_{右} = 5$

（2）当两个数组的元素个数之和为偶数的时候，可以拿下边的图验证，此时有 $size_{左} = \frac{m+n}{2}$



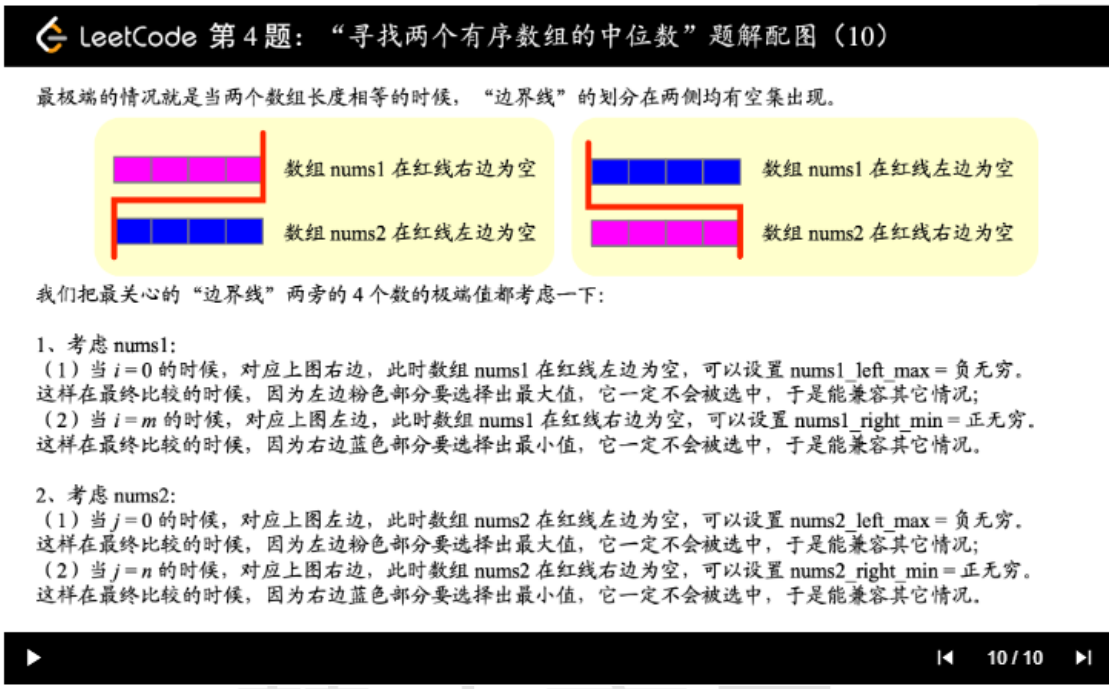
$size_{左} = 7$，$size_{右} = 7$

乍一看，这两个情况不一样，但是仔细一想，如果 $m+n$ 是偶数，$\frac{m+n}{2} = \frac{m+n+1}{2}$，可以代入上图中的数据验证。
因此以上两种情况可以合并，即不管 $m+n$ 的奇偶性，均有 $size_{左} = \frac{m+n+1}{2}$ 成立。
由上一页 PPT，知道 $size_{左} = i + j$。故 $j = \frac{m+n+1}{2} - i$，即索引 i 确定了，索引 j 也就确定了，我们可以只关心索引 i。

7 / 10

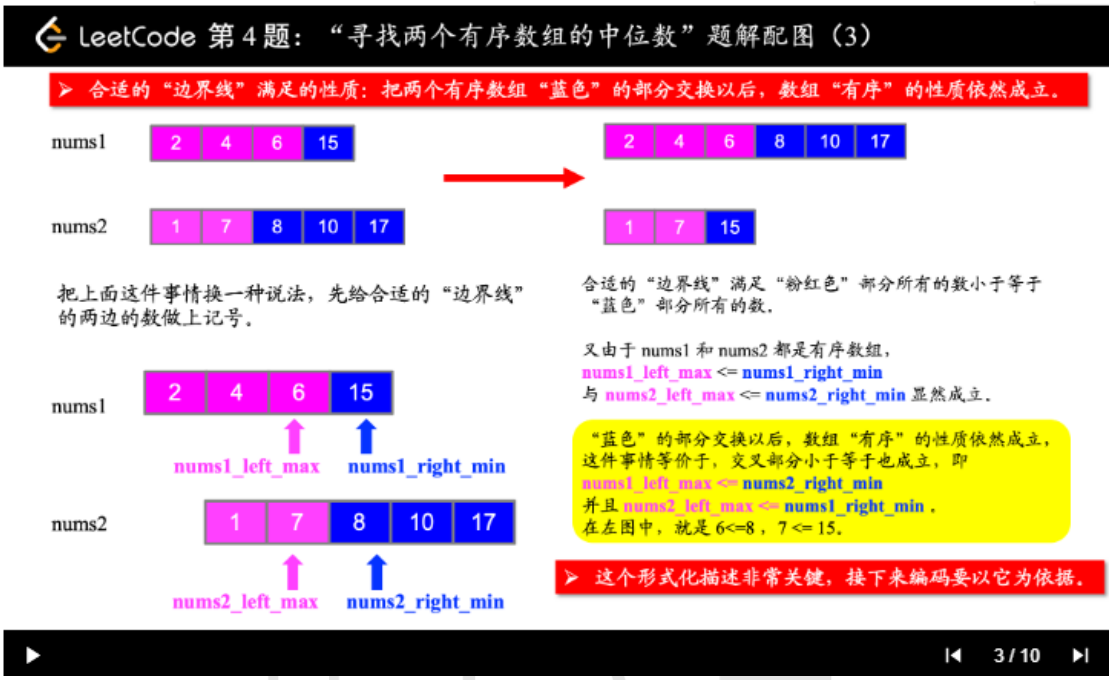
边界值的特殊取法的原因

更多硬核内容就在公众号「yes 的练级攻略」



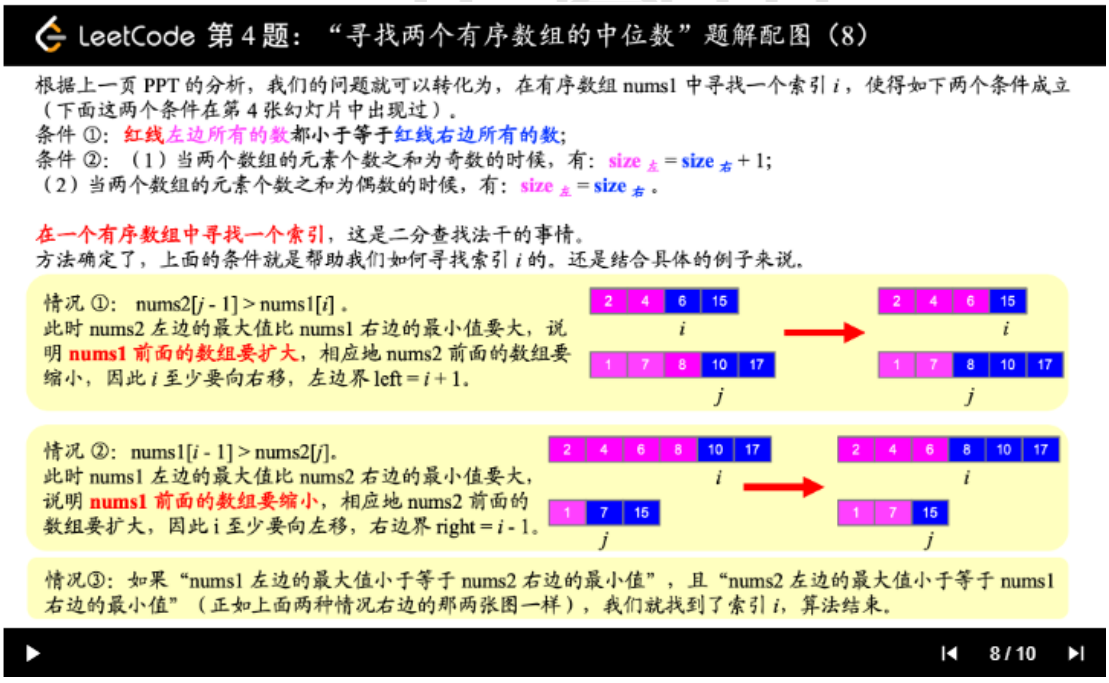
交叉小于等于关系成立，那么中位数就可以从"边界线"两边的数得到

更多硬核内容就在公众号「yes 的练级攻略」



分支缩短边界的原因

禁止商用



方法 1：weiwei 的参考代码 4

```
public class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        // 为了让搜索范围更小，我们始终让 num1 是那个更短的数组，PPT 第 9 张
        if (nums1.length > nums2.length) {
            int[] temp = nums1;
            nums1 = nums2;
            nums2 = temp;
        }
```

```
        // 上述交换保证了 m <= n，在更短的区间 [0, m] 中搜索，会更快一些
        int m = nums1.length;
        int n = nums2.length;

        // 使用二分查找算法在数组 nums1 中搜索一个索引 i，PPT 第 9 张
        int left = 0;
        int right = m;
        // 这里使用的是最简单的、"传统"的二分查找法模板，使用"高级的"二分查找法模板在退出循环时候处理不方便
        while (left <= right) {
            // 尝试要找的索引，在区间里完成二分，为了保证语义，这里就不定义成 mid 了
            // 用加号和右移是安全的做法，即使在溢出的时候都能保证结果正确，但是 Python 中不存在溢出
            // 参考：https://leetcode-cn.com/problems/guess-number-higher-or-lower/solution/shi-fen-hao-yong-de-er-fen-cha-zhao-fa-mo-ban-pyth/
            int i = (left + right) >>> 1;
            // j 的取值在 PPT 第 7 张
            int j = ((m + n + 1) >>> 1) - i;

            // 边界值的特殊取法的原因在 PPT 第 10 张
            int nums1LeftMax = i == 0 ? Integer.MIN_VALUE : nums1[i - 1];
            int nums1RightMin = i == m ? Integer.MAX_VALUE : nums1[i];
            int nums2LeftMax = j == 0 ? Integer.MIN_VALUE : nums2[j - 1];
            int nums2RightMin = j == n ? Integer.MAX_VALUE : nums2[j];

            // 交叉小于等于关系成立，那么中位数就可以从"边界线"两边的数得到，原因在 PPT 第 2 张、第 3 张
            if (nums1LeftMax <= nums2RightMin && nums2LeftMax <= nums1RightMin) {
                // 已经找到解了，分数组之和是奇数还是偶数得到不同的结果，原因在 PPT 第 2 张
                if (((m + n) & 1) == 1) {
                    return Math.max(nums1LeftMax, nums2LeftMax);
                } else {
                    return (double) ((Math.max(nums1LeftMax, nums2LeftMax) + Math.min(nums1RightMin, nums2RightMin))) / 2;
                }
            } else if (nums2LeftMax > nums1RightMin) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                // 这个分支缩短边界的原因在 PPT 第 8 张
                left = i + 1;
            } else {
                // 这个分支缩短边界的原因在 PPT 第 8 张
                right = i - 1;
            }
        }
        throw new IllegalArgumentException("传入无效的参数，输入的数组不是有序数组，算法失效");
    }
}
```

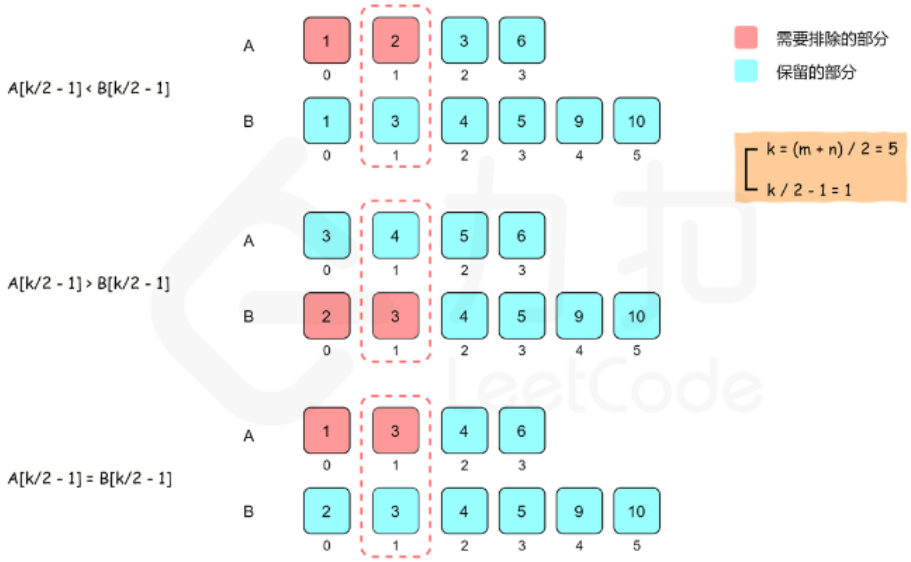
如何把时间复杂度降低到 $O(\log(m+n))$ 呢？如果对时间复杂度的要求有 log，通常都需要用到二分查找，这道题也可以通过二分查找实现。

根据中位数的定义，当 $m+n$ 是奇数时，中位数是两个有序数组中的第 $(m+n)/2$ 个元素，当 $m+n$ 是偶数时，中位数是两个有序数组中的第 $(m+n)/2$ 个元素和第 $(m+n)/2+1$ 个元素的平均值。因此，这道题可以转化成寻找两个有序数组中的第 k 小的数，其中 k 为 $(m+n)/2$ 或 $(m+n)/2+1$。

假设两个有序数组分别是 A 和 B。要找到第 k 个元素，我们可以比较 $A[k/2-1]$ 和 $B[k/2-1]$，其中 / 表示整数除法。由于 $A[k/2-1]$ 和 $B[k/2-1]$ 的前面分别有 $A[0..k/2-2]$ 和 $B[0..k/2-2]$，即 $k/2-1$ 个元素，对于 $A[k/2-1]$ 和 $B[k/2-1]$ 中的较小值，最多只会有 $(k/2-1)+(k/2-1) \le k/2-2$ 个元素比它小，那么它就不能是第 k 小的数了。

因此我们可以归纳出三种情况：

- 如果 $A[k/2-1] < B[k/2-1]$，则比 $A[k/2-1]$ 小的数最多只有 A 的前 $k/2-1$ 个数和 B 的前 $k/2-1$ 个数，即比 $A[k/2-1]$ 小的数最多只有 $k-2$ 个，因此 $A[k/2-1]$ 不可能是第 k 个数，$A[0]$ 到 $A[k/2-1]$ 也都不可能是第 k 个数，可以全部排除。
- 如果 $A[k/2-1] > B[k/2-1]$，则可以排除 $B[0]$ 到 $B[k/2-1]$。
- 如果 $A[k/2-1] = B[k/2-1]$，则可以归入第一种情况处理。



更多硬核内容就在公众号「yes 的练级攻略」

可以看到，比较 $A[k/2-1]$ 和 $B[k/2-1]$ 之后，可以排除 $k/2$ 个不可能是第 k 小的数，查找范围缩小了一半。同时，我们将在排除后的新数组上继续进行二分查找，并且根据我们排除数的个数，减少 k 的值，这是因为我们排除的数都不大于第 k 小的数。

有以下三种情况需要特殊处理：

- 如果 $A[k/2-1]$ 或者 $B[k/2-1]$ 越界，那么我们可以选取对应数组中的最后一个元素。在这种情况下，我们**必须根据排除数的个数减少 k 的值**，而不能直接将 k 减去 $k/2$。
- 如果一个数组为空，说明该数组中的所有元素都被排除，我们可以直接返回另一个数组中第 k 小的元素。
- 如果 $k=1$，我们只要返回两个数组首元素的最小值即可。

用一个例子说明上述算法。假设两个有序数组如下：

```
A: 1 3 4 9
B: 1 2 3 4 5 6 7 8 9
```

两个有序数组的长度分别是 4 和 9，长度之和是 13，中位数是两个有序数组中的第 7 个元素，因此需要找到第 $k=7$ 个元素。

比较两个有序数组中下标为 $k/2-1=2$ 的数，即 $A[2]$ 和 $B[2]$，如下面所示：

```
A: 1 3 4 9
       ↑
B: 1 2 3 4 5 6 7 8 9
       ↑
```

由于 $A[2]>B[2]$，因此排除 $B[0]$ 到 $B[2]$，即数组 B 的下标偏移（offset）变为 3，同时更新 k 的值：$k=k-k/2=4$。

下一步寻找，比较两个有序数组中下标为 $k/2-1=1$ 的数，即 $A[1]$ 和 $B[4]$，如下面所示，其中方括号部分表示已经被排除的数。

```
A: 1 3 4 9
     ↑
B: [1 2 3] 4 5 6 7 8 9
           ↑
```

由于 $A[1]<B[4]$，因此排除 $A[0]$ 到 $A[1]$，即数组 A 的下标偏移变为 2，同时更新 k 的值：$k=k-k/2=2$。

下一步寻找，比较两个有序数组中下标为 $k/2-1=0$ 的数，即比较 $A[2]$ 和 $B[3]$，如下面所示，其中方括号部分表示已经被排除的数。

```
A: [1 3] 4 9
        ↑
B: [1 2 3] 4 5 6 7 8 9
          ↑
```

由于 $A[2]=B[3]$，根据之前的规则，排除 A 中的元素，因此排除 $A[2]$，即数组 A 的下标偏移变为 3，同时更新 k 的值：$k=k-k/2=1$。

由于 k 的值变成 1，因此比较两个有序数组中的未排除下标范围内的第一个数，其中较小的数即为第 k 个数，由于 $A[3]>B[3]$，因此第 k 个数是 $B[3]=4$。

```
A: [1 3 4] 9
          ↑
B: [1 2 3] 4 5 6 7 8 9
          ↑
```

方法 2

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int length1 = nums1.length, length2 = nums2.length;
        int totalLength = length1 + length2;
        if (totalLength % 2 == 1) {
            int midIndex = totalLength / 2;
            double median = getKthElement(nums1, nums2, midIndex + 1);
            return median;
        } else {
            int midIndex1 = totalLength / 2 - 1, midIndex2 = totalLengt
h / 2;
            double median = (getKthElement(nums1, nums2, midIndex1 + 1)
 + getKthElement(nums1, nums2, midIndex2 + 1)) / 2.0;
            return median;
        }
    }

    public int getKthElement(int[] nums1, int[] nums2, int k) {
        /* 主要思路：要找到第 k (k>1) 小的元素，那么就
取 pivot1 = nums1[k/2-1] 和 pivot2 = nums2[k/2-1] 进行比较
         * 这里的 "/" 表示整除
         * nums1 中小于等于 pivot1 的元素有 nums1[0 .. k/2-2] 共
计 k/2-1 个
         * nums2 中小于等于 pivot2 的元素有 nums2[0 .. k/2-2] 共
计 k/2-1 个
         * 取 pivot = min(pivot1, pivot2)，两个数组中小于等于 pivot 的元素
共计不会超过 (k/2-1) + (k/2-1) <= k-2 个
         * 这样 pivot 本身最大也只能是第 k-1 小的元素
         * 如果 pivot = pivot1，那么 nums1[0 .. k/2-1] 都不可能是第 k 小的
元素。把这些元素全部 "删除"，剩下的作为新的 nums1 数组
         * 如果 pivot = pivot2，那么 nums2[0 .. k/2-1] 都不可能是第 k 小的
元素。把这些元素全部 "删除"，剩下的作为新的 nums2 数组
         * 由于我们 "删除" 了一些元素（这些元素都比第 k 小的元素要小），因此需
要修改 k 的值，减去删除的数的个数
         */

        int length1 = nums1.length, length2 = nums2.length;
        int index1 = 0, index2 = 0;
        int kthElement = 0;

        while (true) {
            // 边界情况
            if (index1 == length1) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                return nums2[index2 + k - 1];
            }
            if (index2 == length2) {
                return nums1[index1 + k - 1];
            }
            if (k == 1) {
                return Math.min(nums1[index1], nums2[index2]);
            }

            // 正常情况
            int half = k / 2;
            int newIndex1 = Math.min(index1 + half, length1) - 1;
            int newIndex2 = Math.min(index2 + half, length2) - 1;
            int pivot1 = nums1[newIndex1], pivot2 = nums2[newIndex2];
            if (pivot1 <= pivot2) {
                k -= (newIndex1 - index1 + 1);
                index1 = newIndex1 + 1;
            } else {
                k -= (newIndex2 - index2 + 1);
                index2 = newIndex2 + 1;
            }
        }
    }
}
```

3. 搜索旋转排序数组（33）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

33. 搜索旋转排序数组

难度 中等 788 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0, 1, 2, 4, 5, 6, 7]` 可能变为 `[4, 5, 6, 7, 0, 1, 2]`)。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 `-1` 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

示例 1:

```
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
```

示例 2:

```
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

- **方法二：在英文版看到了一个有意思的思路。**

  对于旋转数组 `nums = [4,5,6,7,0,1,2]`
  首先根据 `nums[0]` 与 `target` 的关系判断 `target` 是在左段还是右段。

  - 例如 `target = 5`，目标值在左半段，因此在 `[4, 5, 6, 7, inf, inf, inf]` 这个有序数组里找就行了；
  - 例如 `target = 1`，目标值在右半段，因此在 `[-inf, -inf, -inf, -inf, 0, 1, 2]` 这个有序数组里找就行了。

  如此，我们又双叒叕将「旋转数组中找目标值」转化成了「有序数组中找目标值」

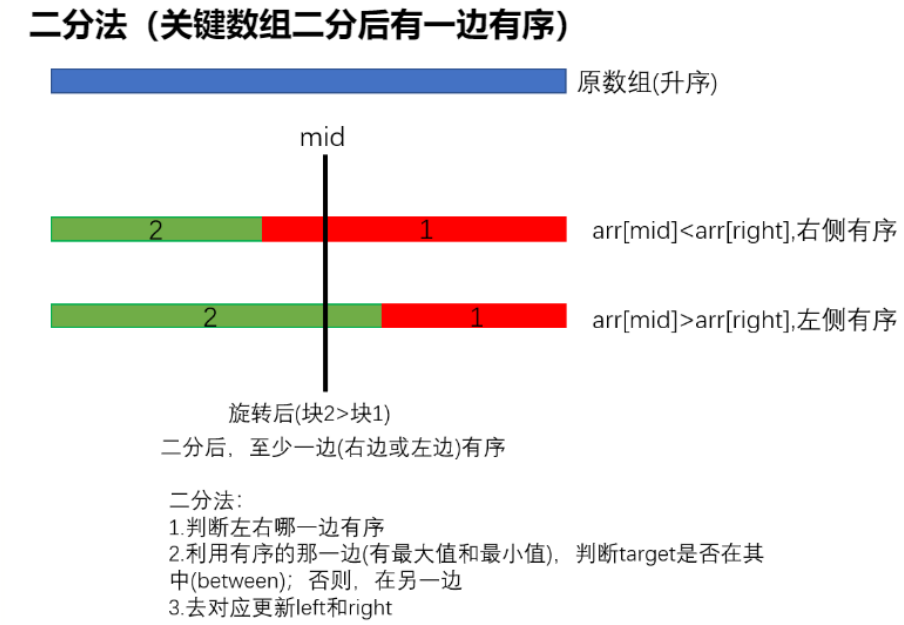
参考链接：
https://leetcode-cn.com/problems/search-in-rotated-sorted-array/solution/duo-si-lu-wan-quan-gong-lue-bi-xu-miao-dong-by-swe/

方法 1

```
class Solution {
    public int search(int[] nums, int target) {
        int low = 0, high = nums.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) {
                return mid;
            }
            // 先根据 nums[0] 与 target 的关系判断目标值是在左半段还是右半段
            if (target >= nums[0]) {
```

```
                // 目标值在左半段时，若 mid 在右半段，则将 mid 索引的值改
成 inf
                if (nums[mid] < nums[0]) {
                    nums[mid] = Integer.MAX_VALUE;
                }
            } else {
                // 目标值在右半段时，若 mid 在左半段，则将 mid 索引的值改
成 -inf
                if (nums[mid] >= nums[0]) {
                    nums[mid] = Integer.MIN_VALUE;
                }
            }
            if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }
}
```

方法 2：另外一种解法（也是 0ms）




```
public class Solution {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target) {
```

```
                return mid;
            } else if (nums[mid] < nums[right]) { // 如果中间的数小于右边的数，则右边是有序的;反之，左边是有序的
                if (nums[mid] < target && target <= nums[right]) { // target 在右侧有序的区间
                    left = mid + 1;
                } else { // 在左侧无序区间
                    right = mid - 1;
                }

            } else { // 在左侧有序区间
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
        }
        return -1;
    }
}
```

4. 搜索旋转排序数组二（81）

81. 搜索旋转排序数组 II

难度 中等 175 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

（例如，数组 `[0, 0, 1, 2, 2, 5, 6]` 可能变为 `[2, 5, 6, 0, 0, 1, 2]` ）。

编写一个函数来判断给定的目标值是否存在于数组中。若存在返回 `true`，否则返回 `false`。

示例 1:

```
输入: nums = [2,5,6,0,0,1,2], target = 0
输出: true
```

示例 2:

```
输入: nums = [2,5,6,0,0,1,2], target = 3
输出: false
```

进阶:

- 这是 搜索旋转排序数组 的延伸题目，本题中的 `nums` 可能包含重复元素。
- 这会影响到程序的时间复杂度吗？会有怎样的影响，为什么？

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/search-in-rotated-sorted-array-ii/solution/er-fen-cha-zhao-by-liweiwei1419/

题目给出的是“旋转排序数组”，是“部分有序数组”，**我们同样可以一次排除一半或者一半以上元素的方法**，即二分查找法。

> 二分查找法的本质是排除法，二分只是手段，二分保证了“熵”最大，即在没有任何有效信息的时候，平分是最好的方案。

思路清楚了以后，我们就得确定“有序数组”存在在“原始数组”的哪个子区间里，下面提供了两个比较标准：

1、中间元素和左边界比较；

2、中间元素和右边界比较。

由这两个比较标准就能写出两版不同的代码。

方法：二分查找

二分查找法的模板我写在了「力扣」第 35 题：“搜索插入位置”的题解《特别好用的二分查找法模板（Python 代码、Java 代码）》。

情况 1：中间元素和左边界比较

```
public class Solution {
    public boolean search(int[] nums, int target) {
        int len = nums.length;
        if (len == 0) {
            return false;
        }
        int left = 0;
        int right = len - 1;
        while (left < right) {
            int mid = (left + right) >>> 1;
            if (nums[mid] > nums[left]) {
                if (nums[left] <= target && target <= nums[mid]) {
                    // 落在前有序数组里
                    right = mid;
                } else {
                    left = mid + 1;
                }
            } else if (nums[mid] < nums[left]) {
                // 让分支和上面分支一样
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid;
                }
            } else {
                // 要排除掉左边界之前，先看一看左边界可以不可以排除
                if (nums[left] == target) {
                    return true;
                } else {
```

```
                left = left + 1;
            }
        }

    }
    // 后处理，夹逼以后，还要判断一下，是不是 target
    return nums[left] == target;
  }
}
```

5. 搜索旋转数组（面试题 10.03）

面试题 10.03. 搜索旋转数组

难度 中等 26 收藏 分享 切换为英文 关注 反馈

搜索旋转数组。给定一个排序后的数组，包含n个整数，但这个数组已被旋转过很多次了，次数不详。请编写代码找出数组中的某个元素，假设数组元素原先是按升序排列的。若有多个相同元素，返回索引值最小的一个。

示例1:

```
输入：arr = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14], target = 5
输出：8（元素5在该数组中的索引）
```

示例2:

```
输入：arr = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14], target = 11
输出：-1 （没有找到）
```

提示:

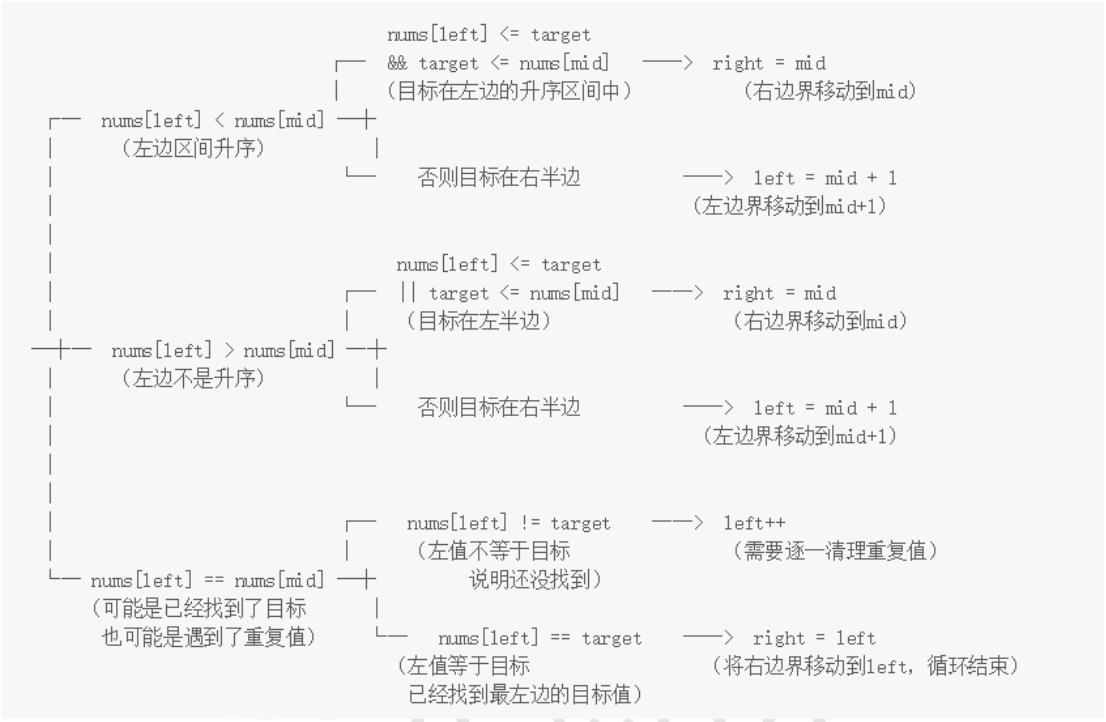
1. arr 长度范围在[1, 1000000]之间

https://leetcode-cn.com/problems/search-rotate-array-lcci/solution/er-fen-fa-by-armeria-program/

对于能判断升序区间的情况，根据目标值的大小移动边界。
对于不能判断升序区间的情况，需要逐步清除重复值。

没有重复值的最优情况时间复杂度是O(log N)，全部或几乎全部是重复值的最差情况时间复杂度是O(N)。

更多硬核内容就在公众号「yes 的练级攻略」




```
class Solution {
    public int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        if (right == -1)
            return -1;
        while (left < right) {
// 循环结束条件 left==right
            int mid = left + (right - left) / 2;
            if (nums[left] < nums[mid]) {
// 如果左值小于中值，说明左边区间升序
                if (nums[left] <= target && target <= nums[mid]) {
// 如果目标在左边的升序区间中，右边界移动到 mid
                    right = mid;
                } else {
// 否则目标在右半边，左边界移动到 mid+1
                    left = mid + 1;
                }
            } else if (nums[left] > nums[mid]) {
// 如果左值大于中值，说明左边不是升序，右半边升序
                if (nums[left] <= target || target <= nums[mid]) {
// 如果目标在左边，右边界移动到 mid
                    right = mid;
                } else {
// 否则目标在右半边，左边界移动到 mid+1
                    left = mid + 1;
```

```
                }
            } else if (nums[left] == nums[mid]) {
// 如果左值等于中值，可能是已经找到了目标，也可能是遇到了重复值
                if (nums[left] != target) {
// 如果左值不等于目标，说明还没找到，需要逐一清理重复值。
                    left++;
                } else {
// 如果左值等于目标，说明已经找到最左边的目标值
                    right = left;
// 将右边界移动到 left，循环结束
                }
            }
        }
        return (nums[left] == target) ? left : -1;
// 返回 left，或者-1
    }
}
```

6. 寻找旋转排序数组中的最小值（153）

153. 寻找旋转排序数组中的最小值

难度 中等 197 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

（例如，数组 [0, 1, 2, 4, 5, 6, 7] 可能变为 [4, 5, 6, 7, 0, 1, 2] ）。

请找出其中最小的元素。

你可以假设数组中不存在重复元素。

示例 1:

```
输入：[3,4,5,1,2]
输出：1
```

示例 2:

```
输入：[4,5,6,7,0,1,2]
输出：0
```

https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/solution/er-fen-fa-fen-zhi-fa-python-dai-ma-java-dai-ma-by-/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：二分查找

思路分析：二分法也可以解决这个问题，因为「旋转排序数组」，几乎有序的数组，也可以通过比较特定位置的元素的值的判断达到「减治」的效果（**逐渐缩小搜索区间**）。

很自然地，我们会看 **中间数**（位于待搜索区间中间位置的元素，由于不是有序数组，因此不能称之为中位数）。

另外，待搜索区间头和尾的元素是位置特殊的元素。有两个比较自然的思路是：

- 思路 1：看看当前搜索区间的 **左边界** 和「中间数」（注意这里不是中位数），是不是可以缩小搜索区间的范围；
- 思路 2：看看当前搜索区间的 **右边界** 和「中间数」（注意这里不是中位数），是不是可以缩小搜索区间的范围；

要想清楚这个问题，我们不妨举几个例子。

- 针对思路 1：

例 1：`[1, 2, 3, 4, 5]`
例 2：`[2, 3, 4, 5, 1]`

这两个例子的「中间数」都比左边界大，但是「旋转排序数组」的最小值一个在「中间数」的左边，一个在「中间数」的右边，因此思路 1 不可行。

- 针对思路 2，依然写两个例子，这两个例子分别是「中间数比右边界大」和「中间数比右边界小」，看看能不能推导出一般化的结论。

例 3：`[7, 8, 9, 10, 11, 12, 1, 2, 3]`

「中间数」 11 比右边界 3 大，因此中间数左边的数（包括中间数）都不是「旋转排序数组」的最小值，因此，下一轮搜索的区间是 `[mid + 1, right]`，将下一轮搜索的左边界设置成中间数位置 + 1，即 `left = mid + 1`。

例 4：`[7, 8, 1, 2, 3]`

「中间数」 1 比右边界 3 小，说明，中间数到右边界是递增的，那么中间数右边的（不包括中间数）一定不是「旋转排序数组」的最小值，可以把它们排除，但中间数有可能是整个数组中的最小值，就如本例，因此，在下一轮搜索区间是 `[left, mid]`，于是把右边界设置为 `right = mid`。

从例 3 和例 4 可以看出，不论中间数比右边界大，还是中间数比右边界小，我们都可以排除掉将近一半的元素，把原始问题转换成一个规模更小的子问题，这正是「减而治之」思想的体现，因此思路 2 可行。

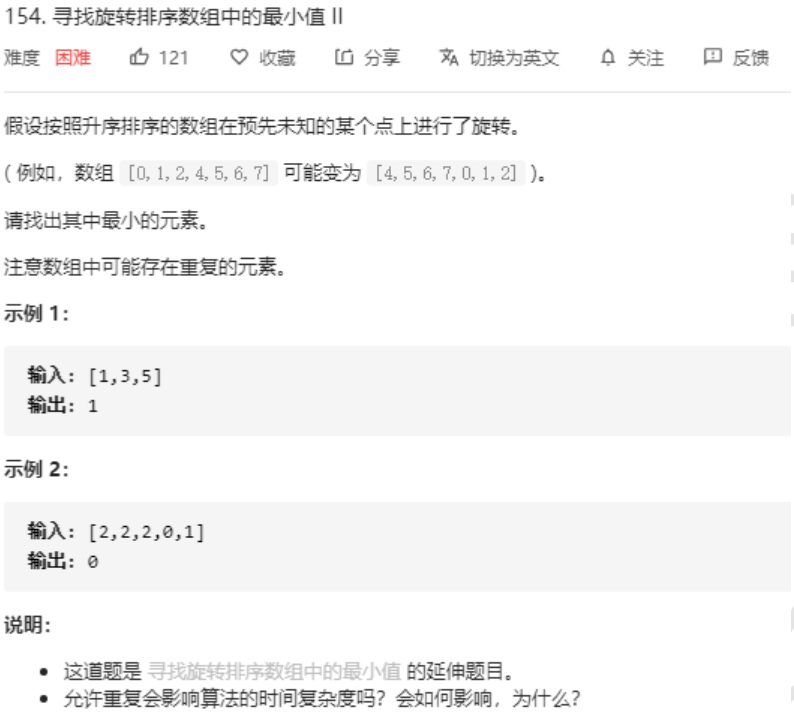
复杂度分析：

- 时间复杂度：$O(\log N)$，这里 N 是输入数组的长度；
- 空间复杂度：$O(1)$，只使用了常数个临时变量。

```java
public class Solution {
    public int findMin(int[] nums) {
        int len = nums.length;
        if (len == 0) {
            throw new IllegalArgumentException("数组为空，无最小元素");
        }
        int left = 0;
        int right = len - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            // int mid = (left + right) >>> 1;
            if (nums[mid] > nums[right]) {
                left = mid + 1;
```

```
            } else {
                // 因为题目中说：你可以假设数组中不存在重复元素。
                // 此时一定有 nums[mid] < nums[right]
                right = mid;
            }
        }
        // 一定存在最小元素，因此无需再做判断
        return nums[left];
    }
}
```

7. 寻找旋转排序数组中的最小值 2（154）（剑指 offer11 旋转数组的最小数字）【二分】



154. 寻找旋转排序数组中的最小值 II

难度 困难 121 收藏 分享 切换为英文 关注 反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0, 1, 2, 4, 5, 6, 7] 可能变为 [4, 5, 6, 7, 0, 1, 2])。

请找出其中最小的元素。

注意数组中可能存在重复的元素。

示例 1：

```
输入：[1,3,5]
输出：1
```

示例 2：

```
输入：[2,2,2,0,1]
输出：0
```

说明：

- 这道题是 寻找旋转排序数组中的最小值 的延伸题目。
- 允许重复会影响算法的时间复杂度吗？会如何影响，为什么？

https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array-ii/solution/er-fen-fa-fen-zhi-fa-python-dai-ma-by-liweiwei1419/

方法一：二分法

只要是一次可以排除一半，都可以用二分法。那么如何使用二分法呢？我们很自然会想到使用边界的值和中间位置的值进行比较。

注意：这里的提法是「中间数」，即位于中间的那个数，不是中位数，请注意区分。

可以分为以下两种情况：

1、中间数与左边界比较

尝试在纸上写出几个例子：

例 1：`[1, 2, 3, 4, 5]`

例 2：`[2, 3, 4, 5, 1]`

以上这两个例子，中间数都比左边界大，但是旋转排序数组的最小值可能在中间数的左边（例 1），也可能在中间数的右边（例 2），因此**不能使用中间数与左边界比较作为二分法的讨论依据。**

接下来，看看另一种讨论依据。

2、中间数与右边界比较：

（1）当中间数比右边界表示的数大的时候，中间数就一定不是目标数（旋转排序数组的最小值）。

还是尝试举个例子：

例 3：`[7, 8, 9, 10, 11, 12, 1, 2, 3]`

中间数 11 比右边界 3 大，因此中间数以及中间数前面的数都不是目标数，把左边界设置中间数位置 + 1，即 `left = mid + 1`；

（2）当中间数比右边界表示的数小的时候，中间数就可能是目标数（旋转排序数组的最小值），举个例子：

例 4：`[7, 8, 1, 2, 3]`

中间数 1 比右边界表示的数小的时候，说明，中间数到右边界是递增的（对于这道题是非递减），那么中间数右边的（不包括中间数）就一定不是目标数，可以把它们排除，不过中间数有可能是目标数，就如本例，因此，把右边界设置为 `right = mid`。

（3）当中间数与右边界表示的数相等的时候，看下面两个例子：

例 5：`[0, 1, 1, 1, 1, 1, 1]`

例 6：`[1, 1, 1, 1, 0, 1, 1]`

目标值可能在中间数的左边，也可能在中间数的右边，那么该怎么办呢？很简单，此时你看到的是右边界，就把只右边界排除掉就好了。正是因为右边界和中间数相等，你去掉了右边界，中间数还在，就让中间数在后面的循环中被发现吧。

因此，根据中间数和右边界的大小关系，可以使用二分法搜索到目标值。

关键：

> **解本题的关键在于举例，在尝试举例的过程中，考虑到不同的情况，得到解题思路。**

```java
public class Solution {
    public int findMin(int[] nums) {
        int len = nums.length;
        if (len == 0) {
            return 0;
        }
        int left = 0;
        int right = len - 1;
```

```
            while (left < right) {
                // int mid = left + (right - left) / 2;
                int mid = (left + right) >>> 1;
                if (nums[mid] > nums[right]) {
                    left = mid + 1;
                } else if (nums[mid] < nums[right]) {
                    right = mid;
                } else {
                    assert nums[mid] == nums[right];
                    right--;
                }
            }
            return nums[left];
        }
}
```

8. 在排序数组中查找元素的第一个和最后一个位置（34）

34. 在排序数组中查找元素的第一个和最后一个位置

难度 中等 467 收藏 分享 切换为英文 关注 反馈

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

如果数组中不存在目标值，返回 `[-1, -1]`。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
```

官方题解：

https://leetcode-cn.com/problems/find-first-and-last-position-of-element-in-sorted-array/solution/zai-pai-xu-shu-zu-zhong-cha-zhao-yuan-su-de-di-yi-/

想法

因为数组已经排过序了，我们可以使用二分查找的方法去定位左右下标。

算法

总体算法工作过程与线性扫描方法类似，除了找最左和最右下标的方法。这里我们仅仅做几个微小的调整，用这种修改过的二分查找方法去搜索这个排过序的数组。首先，为了找到最左边（或者最右边）包含 `target` 的下标（而不是找到的话就返回 `true`），所以算法在我们找到一个 `target` 后不能马上停止。我们需要继续搜索，直到 `lo = hi` 且它们在某个 `target` 值处下标相同。

另一个改变是 `left` 参数的引入，它是一个 boolean 类型的变量，指示我们在遇到 `target = nums[mid]` 时应该做什么。如果 `left` 为 `true`，那么我们递归查询左区间，否则递归右区间。考虑如果我们在下标为 `i` 处遇到了 `target`，最左边的 `target` 一定不会出现在下标大于 `i` 的位置，所以我们永远不需要考虑右子区间。当求最右下标时，道理同样适用。

☆收藏

更多硬核内容就在公众号「yes 的练级攻略」

方法 1

```
class Solution {
    // returns leftmost (or rightmost) index at which `target` should b
e
    // inserted in sorted array `nums` via binary search.
    private int extremeInsertionIndex(int[] nums, int target, boolean l
eft) {
        int lo = 0;
        int hi = nums.length;

        while (lo < hi) {
            int mid = (lo + hi) / 2;
            if (nums[mid] > target || (left && target == nums[mid])) {
                hi = mid;
            }
            else {
                lo = mid+1;
            }
        }

        return lo;
    }

    public int[] searchRange(int[] nums, int target) {
        int[] targetRange = {-1, -1};

        int leftIdx = extremeInsertionIndex(nums, target, true);

        // assert that `leftIdx` is within the array bounds and that `t
arget`
        // is actually in `nums`.
        if (leftIdx == nums.length || nums[leftIdx] != target) {
            return targetRange;
        }

        targetRange[0] = leftIdx;
        targetRange[1] = extremeInsertionIndex(nums, target, false)-1;

        return targetRange;
    }
}
```

方法 2

```
class Solution {
```

```
    //这题要求给定target 的左右边界 一种思路是分别求出target 的左右边界然后组合一下
    //我想的另一种思路是也是用二分来做 只不过添加终止条件 所谓的终止条件是 nums[left] == nums[right]
    public int[] searchRange(int[] nums, int target) {
        int[] res = new int[] {-1,-1};
        //先判空
        if(nums.length == 0) return res;
        int left = 0;
        int right = nums.length-1;
        while(left < right){
            int mid = left + ((right - left) >> 1);
            //int mid = (left+right) / 2;  >>>1 是无符号右移1 位， 与 /2 相同
            if(nums[mid] > target){
                right = mid - 1;
            }else if(nums[mid] < target){
                left = mid + 1;
            }else{
                if(nums[left] == target && nums[right] == target){
                    break;
                }else if(nums[left] == target){
                    right--;
                }else if(nums[right] == target){
                    left++;
                }else{
                    left++;
                    right--;
                }
            }
        }
        //退出的时候只有两种可能 1. 从 break 处跳出 此时说明上面找到了区间 同时满足 left != right 且 left < right
        //2.left >= right
        //一直认为while(left < right)的退出条件只有一个 left == right，其实并不是，left < right 的反面是 left >= right
        //像测试用例[2,2] target= 1 时推出，left = 0.right = -1.此时 left > right 也满足推出的条件。
        //因此下面分为两种情况 1.left > right  2.left != right(break 退出的) 3.left == right
        //1.left > right 此时直接返回[-1,-1]
        // if(left > right){
        //     return res;
        // }
```

```
        // //2.left != right
        // if(left != right){
        //     return new int[]{left,right};
        // }
        //3.left == right  这时候要判断 left 是否等于 target 如果是返回
new int[]{left,right}.否则返回 new int[]{-1,-1}
        res = (nums[left] == target && left <= right) ? new int[]{left,
right} : res;
        return res;
    }
}
```

9. 搜索插入位置（35）

35. 搜索插入位置

难度 简单 575 收藏 分享 切换为英文 关注 反馈

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

你可以假设数组中无重复元素。

示例 1:

```
输入: [1,3,5,6], 5
输出: 2
```

示例 2:

```
输入: [1,3,5,6], 2
输出: 1
```

示例 3:

```
输入: [1,3,5,6], 7
输出: 4
```

示例 4:

```
输入: [1,3,5,6], 0
输出: 0
```

方法 1：零神写的用语言自带 API

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int pos = Arrays.binarySearch(nums, target);
        return (pos >= 0 ? pos : -pos - 1);
    }
}
```

https://leetcode-cn.com/problems/search-insert-position/solution/te-bie-hao-yong-de-er-fen-cha-fa-fa-mo-ban-python-/

思路分析：

- 首先，插入位置有可能在数组的末尾（题目中的示例 3），需要单独判断。如果在数组的末尾，插入位置的下标就是数组的长度；
- 否则，根据示例和暴力解法的分析，插入位置的下标是 **大于等于** `target` 的第 1 个元素的位置。

因此，**严格小于** `target` **的元素一定不是解**，在循环体中将左右边界 `left` 和 `right` 逐渐向中间靠拢，最后 `left` 和 `right` 相遇，则找到了插入元素的位置。根据这个思路，可以写出如下代码。

复杂度分析：

- 时间复杂度：$O(\log N)$，这里 N 是数组的长度，每一次都将问题的规模缩减为原来的一半，因此时间复杂度是对数级别的；
- 空间复杂度：$O(1)$，使用到常数个临时变量。

方法 3：Weiwei 写的

```
public class Solution {

    public int searchInsert(int[] nums, int target) {
        int len = nums.length;
        if (len == 0) {
            return 0;
        }
        // 特判
        if (nums[len - 1] < target) {
            return len;
        }
        int left = 0;
        int right = len - 1;
        while (left < right) {
            int mid = left + (right - left) >> 1;
//等于 mid = left + (right - left)/2
            // 严格小于 target 的元素一定不是解
            if (nums[mid] < target) {
                // 下一轮搜索区间是 [mid + 1, right]
                left = mid + 1;
            } else {
                // 下一轮搜索区间是 [left, mid],收缩右边界不影响 first
equals

                right = mid;
            }
        }
        return left;
    }
}
```

10. 二分查找（704）

更多硬核内容就在公众号「yes 的练级攻略」

题目描述 | 评论 (246) | 题解(230) | 提交记录

704. 二分查找

难度 简单 | 140 | 收藏 | 分享 | 切换为英文 | 关注 | 反馈

给定一个 `n` 个元素有序的（升序）整型数组 `nums` 和一个目标值 `target` ，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。

示例 1:

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

示例 2:

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不存在 nums 中因此返回 -1
```

提示：

1. 你可以假设 `nums` 中的所有元素是不重复的。
2. `n` 将在 `[1, 10000]` 之间。
3. `nums` 的每个元素都将在 `[-9999, 9999]` 之间。

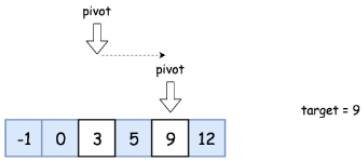
通过次数 54,276 | 提交次数 99,658

https://leetcode-cn.com/problems/binary-search/solution/er-fen-cha-zhao-by-leetcode/

方法：二分查找

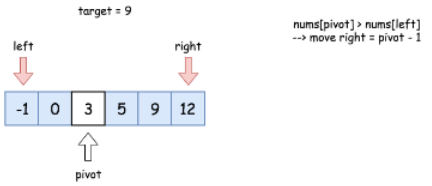
二分查找是一种基于比较目标值和数组中间元素的教科书式算法。

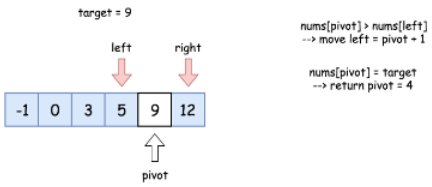
- 如果目标值等于中间元素，则找到目标值。
- 如果目标值较小，继续在左侧搜索。
- 如果目标值较大，则继续在右侧搜索。



算法：

- 初始化指针 `left = 0`，`right = n - 1` 。
- 当 `left <= right`：
  - 比较中间元素 `nums[pivot]` 和目标值 `target` 。
    - 如果 `target = nums[pivot]`，返回 `pivot` 。
    - 如果 `target < nums[pivot]`，则在左侧继续搜索 `right = pivot - 1` 。
    - 如果 `target > nums[pivot]`，则在右侧继续搜索 `left = pivot + 1` 。





更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$\mathcal{O}(\log N)$。
- 空间复杂度：$\mathcal{O}(1)$。

```
class Solution {
    public int search(int[] nums, int target) {
        int pivot, left = 0, right = nums.length - 1;
        while (left <= right) {
            pivot = left + (right - left) / 2;
            if (nums[pivot] == target) return pivot;
            if (target < nums[pivot]) right = pivot - 1;
            else left = pivot + 1;
        }
        return -1;
    }
}
```

11. 魔术索引（面试题 08.03）

面试题 08.03. 魔术索引

难度 简单 34 收藏 分享 切换为英文 关注 反馈

魔术索引。 在数组 A[0...n-1] 中，有所谓的魔术索引，满足条件 A[i] = i。给定一个有序整数数组，编写一种方法找出魔术索引，若有的话，在数组A中找出一个魔术索引，如果没有，则返回-1。若有多个魔术索引，返回索引值最小的一个。

示例1:

```
输入: nums = [0, 2, 3, 4, 5]
输出: 0
说明: 0下标的元素为0
```

示例2:

```
输入: nums = [1, 1, 1]
输出: 1
```

提示:

1. nums长度在[1, 1000000]之间

https://leetcode-cn.com/problems/magic-index-lcci/solution/er-fen-fa-zhi-mo-zhu-suo-yin-adelie-penguin-by-a-f/

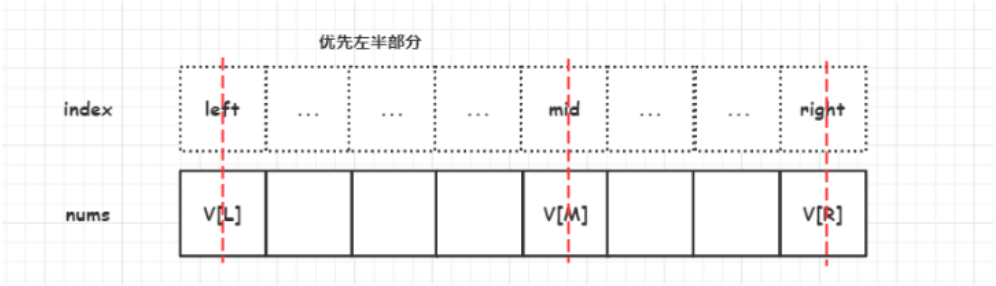
更多硬核内容就在公众号「yes 的练级攻略」

方法2:二分+剪枝

准备一个函数$helper(nums, left, right)$,这个函数求$nums[left...right]$区间范围内第一个符合条件的值（值与索引相等时返回），没有符合的值时，返回-1,

一个前提：数组是有序的，从左到右是上升的

- 每次优先搜索$[left...mid-1]$部分，如果这部分的返回值不为-1,返回这个值即可（注意$helper$函数的定义），如果这个值为-1，说明$[left...mid-1]$部分没有符合条件的值，判断下$nums[mid]$与mid的值是否相等，相等则返回，不相等？进入下一步
- 搜索$[mid+1...right]$,逻辑和搜索左半部分的逻辑相同，在这个大区间上分出左右部分



复杂度分析

- 时间复杂度：最坏情况下会达到 $O(n)$ 的时间复杂度，其中 n 为数组的长度。具体分析已经在上文中讲述，这里不再赘述。
- 空间复杂度：递归函数的空间取决于调用的栈深度，而最坏情况下我们会递归 n 层，即栈深度为 $O(n)$，因此空间复杂度最坏情况下为 $O(n)$。

方法 1：二分加剪枝

```
class Solution {
    public int findMagicIndex(int[] nums) {
        if (nums.length == 0) return -1;
        return helper(nums, 0, nums.length - 1);
    }
    public int helper(int[] nums, int left, int right){
        if(left > right) return -1;
        int mid = left + (right - left) / 2;
        int leftRes = helper(nums, left, mid - 1);
        if (leftRes != -1) return leftRes;
        else if (nums[mid] == mid) return mid;
        return helper(nums, mid + 1, right);
    }
}
```

方法 2：暴力

```
class Solution {
    public int findMagicIndex(int[] nums) {
        for(int i = 0; i < nums.length; ++i){
            if(i == nums[i]) return i;
        }
        return -1;
```

```
    }
}
```

12. 山脉数组的峰顶索引（852）【山形数组的顶峰坐标（lintcode1476）】

852. 山脉数组的峰顶索引

难度 简单 94 收藏 分享 切换为英文 关注 反馈

我们把符合下列属性的数组 A 称作山脉：

- A.length >= 3
- 存在 0 < i < A.length - 1 使得A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1]

给定一个确定为山脉的数组，返回任何满足 A[0] < A[1] < ... A[i-1] < A[i] > A[i+1] > ... > A[A.length - 1] 的 i 的值。

示例 1：

输入： [0,1,0]
输出： 1

示例 2：

输入： [0,2,1,0]
输出： 1

提示：

1. 3 <= A.length <= 10000
2. 0 <= A[i] <= 10^6
3. A 是如上定义的山脉

https://leetcode-cn.com/problems/peak-index-in-a-mountain-array/solution/shan-mai-shu-zu-de-feng-ding-suo-yin-by-leetcode/

方法二：二分查找

思路和算法

将山脉数组中所有满足 A[i] < A[i+1] 的 i 点标记为 True，不满足的点标记为 False。则一个山脉数组可以标记为：[True, True, True, ..., True, False, False, ..., False]。例如山脉数组 [1, 2, 3, 4, 1] 可以标记为 True, True, True, False。

在山脉数组上使用二分查找，找出满足 A[i] < A[i+1] 的最大 i。更多关于 *二分查找* 的知识，请访问 Leetcode 探索。

复杂度分析

- 时间复杂度：$O(\log N)$，其中 N 是 A 的长度。
- 空间复杂度：$O(1)$。

```
class Solution {
    public int peakIndexInMountainArray(int[] A) {
        int lo = 0, hi = A.length - 1;
        while (lo < hi) {
            int mi = lo + (hi - lo) / 2;
            if (A[mi] < A[mi + 1])
```

```
                lo = mi + 1;
            else
                hi = mi;
        }
        return lo;
    }
}
```

13. 山脉序列中的最大值（lintcode585）

585. 山脉序列中的最大值

给 n 个整数的山脉数组，即先增后减的序列，找到山顶（最大值）

样例
例1:

输入: nums = [1, 2, 4, 8, 6, 3]
输出: 8
例2:

输入: nums = [10, 9, 8, 7],
输出: 10
注意事项
数组严格递增，严格递减

```
public class Solution {
    /**
     * @param nums: a mountain sequence which increase firstly and then decrease
     * @return: then mountain top
     */
    public int mountainSequence(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < nums[mid + 1]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return nums[left]; //与上一题方法 2 唯一的不同
    }
}
```

14. 山脉数组中查找目标值（1095）

公众号：yes的练级攻略

1095. 山脉数组中查找目标值

难度 困难 84 收藏 分享 切换为英文 关注 反馈

（这是一个 **交互式问题** ）

给你一个 **山脉数组** `mountainArr` ，请你返回能够使得 `mountainArr.get(index)` **等于** `target` **最小** 的下标 `index` 值。

如果不存在这样的下标 `index` ，就请返回 `-1` 。

何为山脉数组？如果数组 `A` 是一个山脉数组的话，那它满足如下条件：

首先，`A.length >= 3`

其次，在 `0 < i < A.length - 1` 条件下，存在 `i` 使得：

- `A[0] < A[1] < ... A[i-1] < A[i]`
- `A[i] > A[i+1] > ... > A[A.length - 1]`

你将 **不能直接访问该山脉数组**，必须通过 `MountainArray` 接口来获取数据：

- `MountainArray.get(k)` - 会返回数组中索引为`k` 的元素（下标从 0 开始）
- `MountainArray.length()` - 会返回该数组的长度

注意：

对 `MountainArray.get` 发起超过 `100` 次调用的提交将被视为错误答案。此外，任何试图规避判题系统的解决方案都将会导致比赛资格被取消。

为了帮助大家更好地理解交互式问题，我们准备了一个样例 “**答案**”：https://leetcode-cn.com/playground/RKhe3ave，请注意这 **不是一个正确答案。**

示例 1：

```
输入：array = [1,2,3,4,5,3,1], target = 3
输出：2
解释：3 在数组中出现了两次，下标分别为 2 和 5，我们返回最小的下标 2。
```

示例 2：

```
输入：array = [0,1,2,4,2,1], target = 3
输出：-1
解释：3 在数组中没有出现，返回 -1。
```

提示：

- `3 <= mountain_arr.length() <= 10000`
- `0 <= target <= 10^9`
- `0 <= mountain_arr.get(index) <= 10^9`

https://leetcode-cn.com/problems/find-in-mountain-array/solution/shi-yong-chao-hao-yong-de-er-fen-fa-mo-ban-python-/

更多硬核内容就在公众号「yes 的练级攻略」

方法：二分查找法

自然地，求解这道题可以分为 3 步：

第 1 步：先找到山顶元素 `mountaintop` 所在的索引。

> 说到 mountaintop，你是不是跟我一样，很想唱起来："mountaintop，就我一起来"，嘻嘻嘻 ^_^ ，调皮一下。

第 2 步：在前有序且升序数组中找 `target` 所在的索引，如果找到了，就返回，如果没有找到，就执行第 3 步；

第 3 步：如果步骤 2 找不到，就在后有序且降序数组中找 `target` 所在的索引。

> 注意： 具体编码实现的时候，每一步写一个辅助方法就可以了。这 3 个辅助方法都是二分查找法。

在这里我要向大家强烈推荐我使用了很久的二分查找法模板。我专门把这个二分法模板好用的地方、使用它的技巧和注意事项整理在了 「 力扣 」第 35 题：搜索插入位置的题解《特别好用的二分查找法模板（Python 代码、Java 代码）》，希望能对大家有所帮助。

如果你会了这个模板，你就会发现使用这个模板，写出来的 3 个辅助方法的分支逻辑出奇地一样，在取中位数的时候，都取左中位数，才不会发生死循环。

下面给出的参考代码包括了抽象类（接口）和我的简单实现类，还有一些调试的代码。

1095 具体代码

```
/**
 * // This is MountainArray's API interface.
 * // You should not implement it, or speculate about its implementation
 */

interface MountainArray {
    public int get(int index);

    public int length();
}


class MountainArrayImpl implements MountainArray {
    private int[] arr;
    private int size;

    public MountainArrayImpl(int[] arr) {
        this.arr = arr;
        this.size = this.arr.length;
    }

    @Override
    public int get(int index) {
        return this.arr[index];
    }

    @Override
    public int length() {
```

禁止商用 公众号：yes的练级攻略

```
            return this.size;
        }

    }

    class Solution {

        // 特别注意：3 个辅助方法的分支出奇地一样，因此选中位数均选左中位数，才不会发生死循环

        public int findInMountainArray(int target, MountainArray mountainArr) {
            int size = mountainArr.length();
            // 步骤 1：先找到山顶元素所在的索引
            int mountaintop = findMountaintop(mountainArr, 0, size - 1);
            // 步骤 2：在前有序且升序数组中找 target 所在的索引
            int res = findFromSortedArr(mountainArr, 0, mountaintop, target);
            if (res != -1) {
                return res;
            }
            // 步骤 3：如果步骤 2 找不到，就在后有序且降序数组中找 target 所在的索引
            return findFromInversedArr(mountainArr, mountaintop + 1, size - 1, target);
        }

        private int findMountaintop(MountainArray mountainArr, int l, int r) {
            // 返回山顶元素
            while (l < r) {
                int mid = l + (r - l) / 2;
                // 取左中位数，因为进入循环，数组一定至少有 2 个元素
                // 因此，左中位数一定有右边元素，数组下标不会发生越界
                if (mountainArr.get(mid) < mountainArr.get(mid + 1)) {
                    // 如果当前的数比右边的数小，它一定不是山顶
                    l = mid + 1;
                } else {
                    r = mid;
                }
            }
            // 根据题意，山顶元素一定存在，因此退出 while 循环的时候，不用再单独作判断
            return l;
        }

        private int findFromSortedArr(MountainArray mountainArr, int l, int r, int target) {
            // 在前有序且升序数组中找 target 所在的索引
            while (l < r) {
```

禁止商用 公众号：yes 的练级攻略

```
            int mid = l + (r - l) / 2;
            if (mountainArr.get(mid) < target) {
                l = mid + 1;
            } else {
                r = mid;
            }

        }
        // 因为不确定区间收缩成 1 个数以后，这个数是不是要找的数，因此单独做一次判断
        if (mountainArr.get(l) == target) {
            return l;
        }
        return -1;
    }

    private int findFromInversedArr(MountainArray mountainArr, int l, int r, int target) {
        // 在后有序且降序数组中找 target 所在的索引
        while (l < r) {
            int mid = l + (r - l) / 2;
            // 与 findFromSortedArr 方法不同的地方仅仅在于由原来的小于号改成大于好
            if (mountainArr.get(mid) > target) {
                l = mid + 1;
            } else {
                r = mid;
            }
        }
        // 因为不确定区间收缩成 1 个数以后，这个数是不是要找的数，因此单独做一次判断
        if (mountainArr.get(l) == target) {
            return l;
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 3, 1};
        int target = 3;
        MountainArray mountainArray = new MountainArrayImpl(arr);

        Solution solution = new Solution();
        int res = solution.findInMountainArray(target, mountainArray);
```

禁止商用 公众号：yes的练级攻略

```
            System.out.println(res);
        }
}
```

15. 在 D 天内送达包裹的能力（1011）【只能用二分+贪心，动规会超出时间限制】

1011. 在 D 天内送达包裹的能力

难度 中等 95 收藏 分享 切换为英文 关注 反馈

传送带上的包裹必须在 D 天内从一个港口运送到另一个港口。

传送带上的第 `i` 个包裹的重量为 `weights[i]`。每一天，我们都会按给出重量的顺序往传送带上装载包裹。我们装载的重量不会超过船的最大运载重量。

返回能在 `D` 天内将传送带上的所有包裹送达的船的最低运载能力。

示例 1：

输入： weights = [1,2,3,4,5,6,7,8,9,10], D = 5
输出： 15
解释：
船舶最低载重 15 就能够在 5 天内送达所有包裹，如下所示：
第 1 天：1, 2, 3, 4, 5
第 2 天：6, 7
第 3 天：8
第 4 天：9
第 5 天：10

请注意，货物必须按照给定的顺序装运，因此使用载重能力为 14 的船舶并将包装分成 (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) 是不允许的。

示例 2：

输入： weights = [3,2,2,4,1,4], D = 3
输出： 6
解释：
船舶最低载重 6 就能够在 3 天内送达所有包裹，如下所示：
第 1 天：3, 2
第 2 天：2, 4
第 3 天：1, 4

示例 3：

输入： weights = [1,2,3,1,1], D = 4
输出： 3
解释：
第 1 天：1
第 2 天：2
第 3 天：3
第 4 天：1, 1

提示：

1. `1 <= D <= weights.length <= 50000`
2. `1 <= weights[i] <= 500`

https://leetcode-cn.com/problems/capacity-to-ship-packages-within-d-days/solution/zai-dtian-nei-song-da-bao-guo-de-neng-li-by-lenn12/

方法一：二分查找

思路：

- 对于一艘承载力为K船来说，我们必然会在不超过其承载力的前提下贪心地往上装载货物，这样才能使得运送包裹所花费的时间最短。
- 如果船在承载力为K的条件下可以完成在D天内送达包裹的任务，那么任何承载力大于K的条件下依然也能完成任务。
- 我们可以让这个承载力K从$max(weights)$开始（即所有包裹中质量最大包裹的重量，低于这个重量我们不可能完成任务），逐渐增大承载力K，直到K可以让我们在D天内送达包裹。此时K即为我们所要求的最低承载力。
- 逐渐增大承载力K的方法效率过低，让我们用二分查找的方法来优化它。

相似题目：

875. 爱吃香蕉的珂珂

算法：

- 定义函数$canShip(D, K)$,来判断在最低承载力为K的情形下能否在D天内送达所有包裹。我们所要做的就是按照传送带上货物的顺序，依次且尽可能多地往船上装载货物，当该艘船无法装下更多货物时，我们换一艘船，同时将天数加1。当运输完所有货物后，我们判断所用的天数是否小于等于D。
- 用二分查找的方式，来查找这个最低承载力，如果mid可以完成任务，我们把查找范围缩减至$[lo, mid]$（注意不是$mid+1$,因为mid可能是我们所求的解），否则我们去$[mid+1, hi]$区间中继续查找，详情见代码。

复杂度分析：

- 时间复杂度：$O(NlogN)$
- 空间复杂度：$O(1)$

```
class Solution {
    public int shipWithinDays(int[] weights, int D) {
        int lo = 0, hi = Integer.MAX_VALUE;
        while (lo < hi) {
            int mid = lo + (hi - lo) / 2;
            if (canShip(weights, D, mid)) {
                hi = mid;
            } else {
                lo = mid+1;
            }
        }
        return lo;
    }
    private boolean canShip(int[] weights, int D, int K) {
        int cur = K; // cur 表示当前船的可用承载量
        for (int weight: weights) {
            if (weight > K) return false;
            if (cur < weight) {
                cur = K;
                D--;
            }
            cur -= weight;
        }
        return D > 0; // 能否在 D 天内运完
```

```
    }
}
```

16. Shopee 的零食柜（Shopee2019 年第 2 题）

2/5 [编程题]Shopee的零食柜

时间限制：C/C++ 1秒，其他语言2秒
空间限制：C/C++ 32M，其他语言64M

shopee的零食柜，有着各式各样的零食，但是因为贪吃，小虾同学体重日益增加，终于被人叫为小胖了，他终于下定决心减肥了，他决定每天晚上去操场跑两圈，但是跑步太累人了，他想转移注意力，忘记痛苦，正在听着音乐的他，突然有个想法，他想跟着音乐的节奏来跑步，音乐有7种音符，对应的是1到7，那么他对应的步长就可以是1-7分米，这样的话他就可以转移注意力了，但是他想保持自己跑步的速度，在规定时间m分钟跑完。为了避免被累死，他需要规划他每分钟需要跑过的音符，这些音符的步长总和要尽量小。下面是小虾同学听的歌曲的音符，以及规定的时间，你能告诉他每分钟他应该跑多少步长？

输入描述:

输入的第一行输入 n（1 ≤ n ≤ 1000000，表示音符数），m（1<=m< 1000000, m <= n）组成，

第二行有 n 个数，表示每个音符（1<= f <= 7）

输出描述:

输出每分钟应该跑的步长

输入例子1:

8 5 6 5 6 7 6 6 3 1

输出例子1:

11

与上一题主要的区别在于需要处理大数问题，然后二分法<和等于的是不一样的

```
import java.util.*;
public class Main{
    //soTwo
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int m = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        sc.close();
        long res = soTwo(arr, m);
        System.out.println(res);
    }
    public static long soTwo(int[] arr, int m){
        long max = 0, sum = 0;
        for (int a: arr){
```

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
            max = Math.max(a, max);
            sum += a;
        }
        long left = max, right = sum;
        while (left < right){
            long mid = left + (right - left) / 2;
            long splits = split(arr, mid);
            if (splits > m) {
                left = mid + 1;
            } else if (splits < m){
                right = mid - 1;
            } else {
                right = mid;
            }
        }
        return left;
    }
    private static long split(int[] nums, long maxIntervalSum) {
        long splits = 1;
        long curIntervalSum = 0;
        for (int num : nums) {
            if (curIntervalSum + num > maxIntervalSum) {
                curIntervalSum = 0;
                splits++;
            }
            curIntervalSum += num;
        }
        return splits;
    }

}
```

17. 小张刷题计划（LCP 12）【二分+贪心】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

LCP 12. 小张刷题计划

难度 中等 27 收藏 分享 切换为英文 关注 反馈

为了提高自己的代码能力，小张制定了 LeetCode 刷题计划，他选中了 LeetCode 题库中的 n 道题，编号从 0 到 n-1，并计划在 m 天内**按照题目编号顺序**刷完所有的题目（注意，小张不能用多天完成同一题）。

在小张刷题计划中，小张需要用 time[i] 的时间完成编号 i 的题目。此外，小张还可以使用场外求助功能，通过询问他的好朋友小杨题目的解法，可以省去该题的做题时间。为了防止“小张刷题计划”变成“小杨刷题计划”，小张每天最多使用一次求助。

我们定义 m 天中做题时间最多的一天耗时为 T（小杨完成的题目不计入做题总时间）。请你帮小张求出最小的 T 是多少。

示例 1：

输入：time = [1,2,3,3], m = 2

输出：3

解释：第一天小张完成前三题，其中第三题找小杨帮忙；第二天完成第四题，并且找小杨帮忙。这样做题时间最多的一天花费了 3 的时间，并且这个值是最小的。

示例 2：

输入：time = [999,999,999], m = 4

输出：0

解释：在前三天中，小张每天求助小杨一次，这样他可以在三天内完成所有的题目并不花任何时间。

限制：

- 1 <= time.length <= 10^5
- 1 <= time[i] <= 10000
- 1 <= m <= 1000

https://leetcode-cn.com/problems/xiao-zhang-shua-ti-ji-hua/solution/er-fen-jia-bi-zui-xiao-hua-zui-da-zhi-shuang-100-b/

刷这道题之前，建议先刷一下 力扣 410.分割数组的最大值。

- 时间复杂度：$O(n \times log^{\sum_{i=1}^{n} time[i]})$
- 空间复杂度：$O(1)$
- 思路：典型的二分求最大值最小化。因为要求分割成子数组中各自和减去该子数组中最大值后的最大值最小，所以我们只需在 [min(nums),sum(nums)] 中二分查找 mid，判断其是否可以将整个数组划分成不超过 m 的子数组个数，并且不断夹逼最小化 mid 即可。判断子数组和时需要减去该子数组中最大的元素值。

```
class Solution {
    public int minTime(int[] time, int m) {
        int len = time.length, lt = Integer.MAX_VALUE, rt = 0, mid, res = 0;
        if (m >= len)
            return 0;
        for (int i = 0; i < len; i++) {
            lt = Math.min(lt, time[i]);
            rt += time[i];
        }
        while (lt <= rt) {
            mid = (lt + rt) >> 1;
            if (check(time, len, mid, m)) { // 切割得多了，调大下界
                lt = mid + 1;
            } else { // 切割得可能少了，调小上界
                rt = mid - 1;
                res = mid; // 同时记录夹逼值
            }
        }
        return res;
```

```
    }
    private boolean check(int[] time, int len, int limit, int m) {
        int cnt = 1, sum = time[0], maxVal = time[0]; // 初始化为1，且当前子数组最大值初始化为第一个元素
        for (int i = 1; i < len; i++) { // 从第二个元素开始遍历
            sum += time[i];
            maxVal = Math.max(maxVal, time[i]);
            if (sum - maxVal > limit) { // 划分 第 cnt + 1 个子数组（新的子数组的第一个元素）
                cnt++;
                maxVal = sum = time[i];
            }
            if (cnt > m) // 当划分的子数组个数超过 m 时，直接返回 true
                return true;
        }
        return false; // 找到一种可能的分割方案
    }
}
```

18. 爱吃香蕉的珂珂（875）

875. 爱吃香蕉的珂珂

难度 中等 81 收藏 分享 切换为英文 关注 反馈

珂珂喜欢吃香蕉。这里有 `N` 堆香蕉，第 `i` 堆中有 `piles[i]` 根香蕉。警卫已经离开了，将在 `H` 小时后回来。

珂珂可以决定她吃香蕉的速度 `K` （单位：根/小时）。每个小时，她将会选择一堆香蕉，从中吃掉 `K` 根。如果这堆香蕉少于 `K` 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。

返回她可以在 `H` 小时内吃掉所有香蕉的最小速度 `K` （`K` 为整数）。

示例 1：

```
输入: piles = [3,6,7,11], H = 8
输出: 4
```

示例 2：

```
输入: piles = [30,11,23,4,20], H = 5
输出: 30
```

示例 3：

```
输入: piles = [30,11,23,4,20], H = 6
输出: 23
```

提示：

- `1 <= piles.length <= 10^4`
- `piles.length <= H <= 10^9`
- `1 <= piles[i] <= 10^9`

https://leetcode-cn.com/problems/koko-eating-bananas/solution/er-fen-cha-zhao-ding-wei-su-du-by-liweiwei1419/

由于速度是一个有范围的整数，因此可以使用二分查找法解决这个问题。

思路：

1、速度越小，耗时越多；
2、搜索的是速度。因为题目限制了珂珂一个小时之内只能选择一堆香蕉吃，因此速度最大值就是这几堆香蕉中，数量最多的那一堆。速度的最小值是 1（其实还可以再分析一下下界是多少）；

3、还是因为珂珂一个小时之内只能选择一堆香蕉吃，因此：**每堆香蕉吃完的耗时 = 这堆香蕉的数量 / 珂珂一小时吃香蕉的数量**，这里的 `/` 在不能整除的时候，需要上取整。

复杂度分析：

- 时间复杂度：$O(N \log \max(piles))$，这里 N 表示数组 `piles` 的长度。我们在 $[1, \max piles]$ 里使用二分查找定位最小速度，而每一次执行判别函数的时间复杂度是 $O(N)$；
- 空间复杂度：$O(1)$，算法只使用了常数个临时变量。

```
public class Solution {

    public int minEatingSpeed(int[] piles, int H) {
        int maxVal = 1;
        for (int pile : piles) {
            maxVal = Math.max(maxVal, pile);
        }
        // 速度最小的时候，耗时最长
        int left = 1;
        // 速度最大的时候，耗时最短
        int right = maxVal;
        while (left < right) {
            int mid = (left + right) >>> 1;
            if (calculateSum(piles, mid) > H) {
                // 耗时太多，说明速度太慢了，下一轮搜索区间在
                // [mid + 1, right]
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
    }

    /**
     * 如果返回的小时数严格大于 H，就不符合题意
     *
     * @param piles
     * @param speed
     * @return 需要的小时数
     */
    private int calculateSum(int[] piles, int speed) {
```

```
            int sum = 0;
            for (int pile : piles) {

                // 上取整可以这样写，第一种方法快，第二种方法慢
                sum += (pile + speed - 1) / speed;
               //sum += Math.ceil(pile * 1.0 / speed);

            }
            return sum;
        }
}
```

19. 制作 m 束花所需的最少天数（1482）

1482. 制作 m 束花所需的最少天数

难度 中等 50 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `bloomDay`，以及两个整数 `m` 和 `k` 。

现需要制作 `m` 束花。制作花束时，需要使用花园中 **相邻的** `k` **朵花** 。

花园中有 `n` 朵花，第 `i` 朵花会在 `bloomDay[i]` 时盛开，**恰好** 可以用于 **一束** 花中。

请你返回从花园中摘 `m` 束花需要等待的最少的天数。如果不能摘到 `m` 束花则返回 **-1** 。

示例 1：

输入： bloomDay = [1,10,3,10,2], m = 3, k = 1
输出： 3
解释： 让我们一起观察这三天的花开过程，x 表示花开，而 _ 表示花还未开。
现在需要制作 3 束花，每束只需要 1 朵。
1 天后：[x, _, _, _, _] // 只能制作 1 束花
2 天后：[x, _, _, _, x] // 只能制作 2 束花
3 天后：[x, _, x, _, x] // 可以制作 3 束花，答案为 3

示例 2：

输入： bloomDay = [1,10,3,10,2], m = 3, k = 2
输出： -1
解释： 要制作 3 束花，每束需要 2 朵花，也就是一共需要 6 朵花。而花园中只有 5 朵花，无法满足制作要求，返回 -1 。

示例 3：

输入： bloomDay = [7,7,7,7,12,7,7], m = 2, k = 3
输出： 12
解释： 要制作 2 束花，每束需要 3 朵。
花园在 7 天后和 12 天后的情况如下：
7 天后：[x, x, x, x, _, x, x]
可以用前 3 朵盛开的花制作第一束花。但不能使用后 3 朵盛开的花，因为它们不相邻。
12 天后：[x, x, x, x, x, x, x]
显然，我们可以用不同的方式制作两束花。

示例 4：

```
输入：bloomDay = [1000000000,1000000000], m = 1, k = 1
输出：1000000000
解释：需要等 1000000000 天才能采到花来制作花束
```

示例 5：

```
输入：bloomDay = [1,10,2,9,3,8,4,7,5,6], m = 4, k = 2
输出：9
```

提示：

- `bloomDay.length == n`
- `1 <= n <= 10^5`
- `1 <= bloomDay[i] <= 10^9`
- `1 <= m <= 10^6`
- `1 <= k <= n`

https://leetcode-cn.com/problems/minimum-number-of-days-to-make-m-bouquets/solution/java-er-fen-dai-zhu-shi-by-feng-qi-feng-luo/

解题思路

二分查找，看似简单的人人都懂的二分，很少人能够灵活运用。
我们可以用天数作为指标，0天到数组里的最大值最为区间。

```java
class Solution {
    public int minDays(int[] bloomDay, int m, int k) {
        if  (bloomDay.length < m * k) return -1;
        // 最大等待的天数是数组里的最大值
        int max = 0;
        for (int bd : bloomDay){
            max = Math.max(bd, max);
        }
        int left = 0, right = max;
        while (left < right){
            int mid = (left + right) >>> 1;
            // 等待 mid 天，有多少组连续的 k 朵花已经开花了
            int count = getCount(bloomDay, mid, k);
            if (count < m){
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        return left;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    }
    // 返回等待 day 天，有多少组连续的 k 天<=day   这里用的贪心
    public int getCount(int[] arr, int day, int k){
        int res = 0, count = 0;
        for (int i = 0; i < arr.length; i++){
            if (arr[i] > day){
                count = 0;
            } else {
                count++;
            }
            //  连续的 k 朵花开了
            if (count == k) {
                res++;
                count = 0;
            }
        }
        return res;
    }
}
```

20. x 的平方根（69）

69. x 的平方根

难度 简单 455 收藏 分享 切换为英文 关注 反馈

实现 `int sqrt(int x)` 函数。

计算并返回 *x* 的平方根，其中 *x* 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

```
输入: 4
输出: 2
```

示例 2:

```
输入: 8
输出: 2
说明: 8 的平方根是 2.82842...,
     由于返回类型是整数，小数部分将被舍去。
```

https://leetcode-cn.com/problems/sqrtx/solution/er-fen-cha-zhao-niu-dun-fa-python-dai-ma-by-liweiw/

更多硬核内容就在公众号「yes 的练级攻略」

二分查找法应用于搜索平方根的思想很简单，其实就是“猜”，但是是有策略的“猜”，用“排除法”在有限的区间里，一次排除一半的区间元素，最后只剩下一个数，这个数就是题目要求的向下取整的平方根整数。

牛顿法最初提出的时候，是用于求解方程的根，它的基本思想是“以直代曲”，在迭代中搜索得到方程的近似解。

方法一：二分法

思路分析：使用二分法搜索平方根的思想很简单，就类似于小时候我们看的电视节目中的“猜价格”游戏，高了就往低了猜，低了就往高了猜，范围越来越小。因此，使用二分法猜算术平方根就很自然。

一个数的平方根肯定不会超过它自己，不过直觉还告诉我们，一个数的平方根最多不会超过它的一半，例如 8 的平方根，8 的一半是 4，$4^2 = 16 > 8$，如果这个数越大越是如此，因此我们要计算一下，这个边界是多少。为此，解如下不等式：

$$\left(\frac{a}{2}\right)^2 \geq a$$

意即：如果一个数的一半的平方大于它自己，那么这个数的取值范围。解以上不等式得 $a \geq 4$ 或者 $a \leq 0$。

于是边界值就是 4，那么对 0、1、2、3 分别计算结果，很容易知道，这 4 个数的平方根依次是 0、1、1、1。

注意：这 4 个特值如果没有考虑到，有可能导致你设置的搜索边界不正确。在使用二分法寻找平方根的时候，要特别注意边界值的选择，以下给出两个参考代码。

参考代码 1：所有的数都放在一起考虑，为了照顾到 0 把左边界设置为 0，为了照顾到 1 把右边界设置为 `x // 2 + 1`。

复杂度分析：

- 时间复杂度：$O(\log N)$，二分法的时间复杂度是对数级别的。
- 空间复杂度：$O(1)$，使用了常数个数的辅助空间用于存储和比较。

总结：使用二分查找法搜索，注意特值对搜索边界的影响。

以下这部分内容是根据与用户 @lukas 在本题解下的讨论而添加的。

在这里补充一下，如果你实在不太想分析 `a` 的平方根可能的上界，之前说了，它肯定不会超过 `a` 自己，即使你把上界写成一个很大的数，例如 999999，这个数必须得是力扣的测试用例都达不到的数，在二分查找的过程中，不符合要求的数每次会被很快砍掉一半。

yes的练级攻略

方法 1：二分

```
public class Solution {
    public int mySqrt(int x) {
        if (x == 0) return 0;
        // 注意：针对特殊测试用例，例如 2147395599
        // 要把搜索的范围设置成长整型
        long left = 1;
        long right = x / 2;
        while (left < right) {
            // 注意：这里一定取右中位数，如果取左中位数，代码会进入死循环
            // long mid = left + (right - left + 1) / 2;
            long mid = (left + right + 1) >>> 1;
            long square = mid * mid;
            if (square > x) {
                right = mid - 1;
            } else {
                left = mid;
```

```
            }
        }
        // 因为一定存在，因此无需后处理
        return (int) left;
    }
}
```

说明：`1e-6` 是科学计数法，表示 1 乘以 10 的负 6 次方，也就是 0.000001。有的地方使用 `epsilon` (ϵ) 表示 `1e-6`，用来抵消浮点运算中因为误差造成的相等无法判断的情况，它通常是一个非常小的数字，具体多小要根据你的精度需求来设置。

复杂度分析：

- 时间复杂度：（待讨论，反正很快很快就是了 ^_^，调皮一下显示我的无知）。
- 空间复杂度：$O(1)$，使用了常数个数的辅助空间用于存储和比较。

方法 2：牛顿法

```
public class Solution {
    public int mySqrt(int a) {
        long x = a;
        while (x * x > a) {
            x = (x + a / x) / 2;
        }
        return (int) x;
    }
}
```

21. 有序矩阵中第 k 小的元素（378）【可以转为有序矩阵中第 k 大的元素】

378. 有序矩阵中第K小的元素

难度 中等　377　收藏　分享　切换为英文　关注　反馈

给定一个 $n \times n$ 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 `k` 小的元素。
请注意，它是排序后的第 `k` 小元素，而不是第 `k` 个不同的元素。

示例：

```
matrix = [
   [ 1,  5,  9],
   [10, 11, 13],
   [12, 13, 15]
],
k = 8,

返回 13。
```

提示：
你可以假设 k 的值永远是有效的，$1 \leqslant k \leqslant n^2$ 。

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/kth-smallest-element-in-a-sorted-matrix/solution/you-xu-ju-zhen-zhong-di-kxiao-de-yuan-su-by-leetco/

方法三：二分查找

思路及算法

由题目给出的性质可知，这个矩阵内的元素是从左上到右下递增的（假设矩阵左上角为 $matrix[0][0]$）。以下图为例：

1	3	5	7	9	11
2	4	6	8	10	12
3	5	7	9	11	13
4	6	8	10	12	14
5	7	9	11	13	15
6	8	10	12	14	16

我们知道整个二维数组中 $matrix[0][0]$ 为最小值，$matrix[n-1][n-1]$ 为最大值，现在我们将其分别记作 l 和 r。

可以发现一个性质：任取一个数 mid 满足 $l \leq mid \leq r$，那么矩阵中不大于 mid 的数，肯定全部分布在矩阵的左上角。

例如下图，取 $mid = 8$：

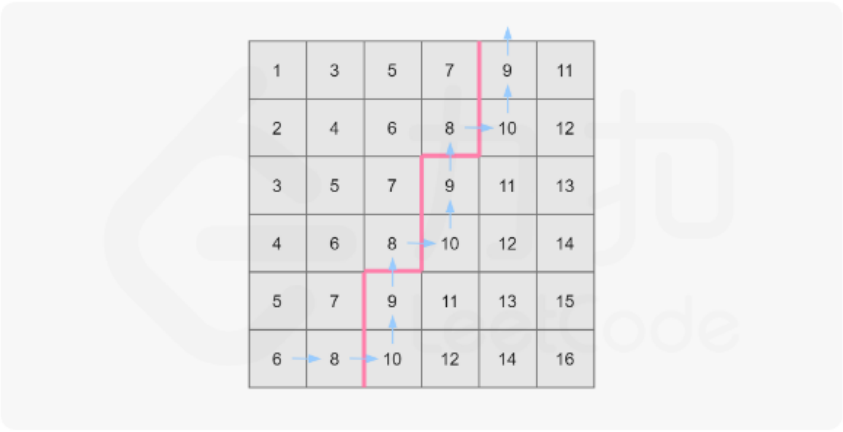
1	3	5	7	9	11
2	4	6	8	10	12
3	5	7	9	11	13
4	6	8	10	12	14
5	7	9	11	13	15
6	8	10	12	14	16

我们可以看到，矩阵中大于 mid 的数就和不大于 mid 的数分别形成了两个板块，沿着一条锯齿线将这个矩形分开。其中左上角板块的大小即为矩阵中不大于 mid 的数的数量。

读者也可以自己取一些 mid 值，通过画图以加深理解。

我们只要沿着这条锯齿线走一遍即可计算出这两个板块的大小，也自然就统计出了这个矩阵中不大于 mid 的数的个数了。

走法演示如下，依然取 $mid = 8$：



可以这样描述走法：

- 初始位置在 $matrix[n-1][0]$（即左下角）；
- 设当前位置为 $matrix[i][j]$。若 $matrix[i][j] \leq mid$，则将当前所在列的不大于 mid 的数的数量（即 $i+1$）累加到答案中，并向右移动，否则向上移动；
- 不断移动直到走出格子为止。

我们发现这样的走法时间复杂度为 $O(n)$，即我们可以线性计算对于任意一个 mid，矩阵中有多少数不大于它。这满足了二分查找的性质。

不妨假设答案为 x，那么可以知道 $l \leq x \leq r$，这样就确定了二分查找的上下界。

每次对于「猜测」的答案 mid，计算矩阵中有多少数不大于 mid：

- 如果数量不少于 k，那么说明最终答案 x 不大于 mid；
- 如果数量少于 k，那么说明最终答案 x 大于 mid。

这样我们就可以计算出最终的结果 x 了。

复杂度分析

- 时间复杂度：$O(n \log(r-l))$，二分查找进行次数为 $O(\log(r-l))$，每次操作时间复杂度为 $O(n)$。
- 空间复杂度：$O(1)$。

写在最后

上述三种解法，第一种没有利用矩阵的性质，所以时间复杂度最差；第二种解法只利用了一部分性质（每一行是一个有序数列，而忽视了列之间的关系）；第三种解法则利用了全部性质，所以时间复杂度最佳。

这也启示我们要认真把握题目中的条件与性质，更有利于我们解题。

378 具体代码

```
class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        int left = matrix[0][0];
        int right = matrix[n - 1][n - 1];
        while (left < right) {
            int mid = left + ((right - left) >> 1);
            if (check(matrix, mid, k, n)) {
```

```
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }

    public boolean check(int[][] matrix, int mid, int k, int n) {
        int i = n - 1;
        int j = 0;
        int num = 0;
        while (i >= 0 && j < n) {
            if (matrix[i][j] <= mid) {
                num += i + 1;
                j++;
            } else {
                i--;
            }
        }
        return num >= k;
    }
}
```

可运行版本的第 k 大元素

```
import java.util.*;
public class Main {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] matrix = new int[n][n];
        for (int i = 0; i < n; i++){
            for (int j = 0; j < n; j++){
                matrix[i][j] = sc.nextInt();
            }
        }
        int k = sc.nextInt();
        int topK = SmallK(n, matrix, n * n - k + 1); //总的元素数目是 n * n
        System.out.println(topK);
    }
    public static int SmallK(int n, int[][] matrix, int k){
        int left = matrix[0][0], right = matrix[n-1][n-1];
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (check(n, matrix, k, mid)) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
    public static boolean check(int n, int[][] matrix, int k, int mid){
        int i = n - 1, j = 0;
        int num = 0;
        while (i >= 0 && j < n) {
            if (matrix[i][j] <= mid) {
                num += i + 1;
                j++;
            } else {
                i--;
            }
        }
        return num >= k;
    }
}
```

22. 有序矩阵中第 k 个最小数组和（1439）

公众号：
yes的练级攻略

1439. 有序矩阵中的第 k 个最小数组和

难度 困难 29 收藏 分享 切换为英文 关注 反馈

给你一个 `m * n` 的矩阵 `mat`，以及一个整数 `k` ，矩阵中的每一行都以非递减的顺序排列。

你可以从每一行中选出 1 个元素形成一个数组。返回所有可能数组中的第 k 个 **最小** 数组和。

示例 1：

输入： mat = [[1,3,11],[2,4,6]], k = 5
输出： 7
解释： 从每一行中选出一个元素，前 k 个和最小的数组分别是：
[1,2], [1,4], [3,2], [3,4], [1,6]。其中第 5 个的和是 7 。

示例 2：

输入： mat = [[1,3,11],[2,4,6]], k = 9
输出： 17

示例 3：

输入： mat = [[1,10,10],[1,4,5],[2,3,6]], k = 7
输出： 9
解释： 从每一行中选出一个元素，前 k 个和最小的数组分别是：
[1,1,2], [1,1,3], [1,4,2], [1,4,3], [1,1,6], [1,5,2], [1,5,3]。其中第 7 个的和是 9 。

示例 4：

输入： mat = [[1,1,10],[2,2,9]], k = 7
输出： 12

提示：

- `m == mat.length`
- `n == mat.length[i]`
- `1 <= m, n <= 40`
- `1 <= k <= min(200, n ^ m)`
- `1 <= mat[i][j] <= 5000`
- `mat[i]` 是一个非递减数组

https://leetcode-cn.com/problems/find-the-kth-smallest-sum-of-a-matrix-with-sorted-rows/solution/er-fen-fa-hui-su-java-by-zper/

```
class Solution {
    int cnt=0;
    public void help(int[][] mat,int mid,int cur_sum,int index,int k){
        //如果已经超过mid，则总和不可能小于等于mid，直接返回
        if(cur_sum > mid) return;
```

```
        //如果组合数已经超过k，则当前k 不可能为第k 个数组和，直接返回(只要知道>k
即可，不管大于多少)
        if(cnt > k) return;
        //满足条件，cnt++，并返回
        if(index == mat.length){
            cnt++;
            return;
        }
        //在第 index 行进行遍历，因为 cur_sum 默认是每一行第一个元素的和，所以
遍历到第 j 个数时，
        //需要减去 index 行第一个数
        for(int j = 0; j < mat[0].length; j++){
            help(mat, mid, cur_sum+mat[index][j] - mat[index][0], index
 + 1, k);
        }
    }

    public int kthSmallest(int[][] mat, int k) {
        int m = mat.length, n = mat[0].length;
        //求 max 中最小和最大数组和
        int lo = 0, hi = 0;
        for(int i = 0; i < m; i++){
            lo += mat[i][0];
            hi += mat[i][n-1];
        }
        //low 记录最小数组和，作为回溯 cur_sum 的初始值。
        //如果 cur_sum 从 0 开始，则分支过多，会超时，所以使用最小数组和
        int low = lo;
        while(lo < hi){
            cnt = 0;
            int mid = (hi-lo)/2 + lo;
            help(mat, mid, low, 0, k);
            if(cnt >= k) hi = mid;
            else lo = mid+1;
        }
        return lo;
    }
}
```

23. 搜索二维矩阵（74）

更多硬核内容就在公众号「yes 的练级攻略」

74. 搜索二维矩阵

难度 中等 215 收藏 分享 切换为英文 关注 反馈

编写一个高效的算法来判断 $m \times n$ 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1:

```
输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 3
输出: true
```

示例 2:

```
输入:
matrix = [
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
target = 13
输出: false
```

https://leetcode-cn.com/problems/search-a-2d-matrix/solution/sou-suo-er-wei-ju-zhen-by-leetcode/

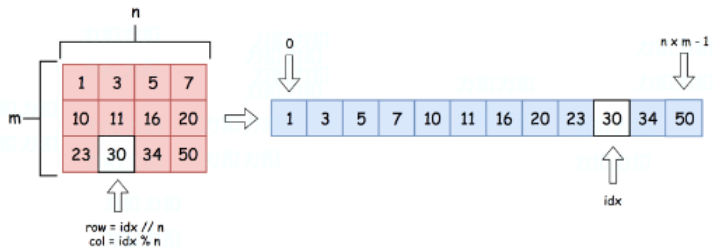
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法：二分查找

直觉

注意到输入的 `m x n` 矩阵可以视为长度为 `m x n` 的有序数组。



由于该 *虚* 数组的序号可以由下式方便地转化为原矩阵中的行和列 (*我们当然不会真的创建一个新数组*)，该有序数组非常适合二分查找。

> `row = idx // n` ， `col = idx % n` 。

算法

这是一个标准二分查找算法：

- 初始化左右序号
  `left = 0` 和 `right = m x n - 1` 。
- While `left < right` ：
  - 选取虚数组最中间的序号作为中间序号：`pivot_idx = (left + right) / 2` 。
  - 该序号对应于原矩阵中的 `row = pivot_idx // n` 行 `col = pivot_idx % n` 列，由此可以拿到中间元素 `pivot_element` 。该元素将虚数组分为两部分。
  - 比较 `pivot_element` 与 `target` 以确定在哪一部分进行进一步查找。

复杂度分析

- 时间复杂度：由于是标准的二分查找，时间复杂度为$O(\log(mn))$。
- 空间复杂度：$O(1)$。

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int m = matrix.length;
        if (m == 0) return false;
        int n = matrix[0].length;

        // 二分查找
        int left = 0, right = m * n - 1;
        int pivotIdx, pivotElement;
        while (left <= right) {
        pivotIdx = (left + right) >>> 1;
        pivotElement = matrix[pivotIdx / n][pivotIdx % n];
        if (target == pivotElement) return true;
        else {
            if (target < pivotElement) right = pivotIdx - 1;
            else left = pivotIdx + 1;
```

```
            }
            }
            return false;
        }
}
```

24. 搜索二维矩阵 2（240）【二维数组中的查找（剑指 offer 4）】【排序矩阵查找（面试题 10.09）】

240. 搜索二维矩阵 II

难度 中等 384 收藏 分享 切换为英文 关注 反馈

编写一个高效的算法来搜索 *m* x *n* 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例:

现有矩阵 matrix 如下：

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = `5`，返回 `true`。

给定 target = `20`，返回 `false`。

剑指 Offer 04. 二维数组中的查找

难度 简单 117 收藏 分享 切换为英文 关注 反馈

在一个 n * m 的二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

示例:

现有矩阵 matrix 如下：

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = `5`，返回 `true`。

给定 target = `20`，返回 `false`。

限制:

`0 <= n <= 1000`

`0 <= m <= 1000`

注意：本题与主站 240 题相同：https://leetcode-cn.com/problems/search-a-2d-matrix-ii/

更多硬核内容就在公众号「yes 的练级攻略」

面试题 10.09. 排序矩阵查找

难度 中等 10 收藏 分享 切换为英文 关注

给定M×N矩阵，每一行、每一列都按升序排列，请编写代码找出某元素。

示例:

现有矩阵 matrix 如下：

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = `5`，返回 `true`。

给定 target = `20`，返回 `false`。

https://leetcode-cn.com/problems/search-a-2d-matrix-ii/solution/sou-suo-er-wei-ju-zhen-ii-by-leetcode-2/

禁止商用

方法四：

因为矩阵的行和列是排序的（分别从左到右和从上到下），所以在查看任何特定值时，我们可以修剪$O(m)$或$O(n)$元素。

算法：

首先，我们初始化一个指向矩阵左下角的 (row, col) 指针。然后，直到找到目标并返回 `true`（或者指针指向矩阵维度之外的 (row, col) 为止，我们执行以下操作：如果当前指向的值大于目标值，则可以“向上”移动一行。否则，如果当前指向的值小于目标值，则可以移动一列。不难理解为什么这样做永远不会删减正确的答案：因为行是从左到右排序的，所以我们知道当前值右侧的每个值都较大。因此，如果当前值已经大于目标值，我们知道它右边的每个值会比较大。也可以对列进行非常类似的论证，因此这种搜索方式将始终在矩阵中找到目标（如果存在）。

复杂度分析

- 时间复杂度：$O(n+m)$。
  时间复杂度分析的关键是注意到在每次迭代（我们不返回 true）时，行或列都会精确地递减/递增一次。由于行只能减少 m 次，而列只能增加 n 次，因此在导致 `while` 循环终止之前，循环不能运行超过 $n+m$ 次。因为所有其他的工作都是常数，所以总的时间复杂度在矩阵维数之和中是线性的。
- 空间复杂度：$O(1)$，因为这种方法只处理几个指针，所以它的内存占用是恒定的。

```java
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        // 从左下角开始
        int row = matrix.length-1;
        int col = 0;
        while (row >= 0 && col < matrix[0].length) {
            //如果当前指向的值大于目标值，则可以“向上”移动一行。
            if (matrix[row][col] > target) {
                row--;
            } else if (matrix[row][col] < target) {//如果当前指向的值小于目标值，则可以移动一列。
```

```
                col++;
            } else { //找到目标
                return true;
            }
        }

        return false;
    }
}
```

25. 最后一块石头的重量（1046）

1046. 最后一块石头的重量

难度 简单 65 收藏 分享 切换为英文 关注 反馈

有一堆石头，每块石头的重量都是正整数。

每一回合，从中选出两块 **最重的** 石头，然后将它们一起粉碎。假设石头的重量分别为 `x` 和 `y`，且 `x <= y`。那么粉碎的可能结果如下：

- 如果 `x == y`，那么两块石头都会被完全粉碎；
- 如果 `x != y`，那么重量为 `x` 的石头将会完全粉碎，而重量为 `y` 的石头新重量为 `y-x`。

最后，最多只会剩下一块石头。返回此石头的重量。如果没有石头剩下，就返回 `0`。

示例：

输入： [2,7,4,1,8,1]
输出： 1
解释：
先选出 7 和 8，得到 1，所以数组转换为 [2,4,1,1,1]，
再选出 2 和 4，得到 2，所以数组转换为 [2,1,1,1]，
接着是 2 和 1，得到 1，所以数组转换为 [1,1,1]，
最后选出 1 和 1，得到 0，最终数组转换为 [1]，这就是最后剩下那块石头的重量。

提示：

1. `1 <= stones.length <= 30`
2. `1 <= stones[i] <= 1000`

https://leetcode-cn.com/problems/last-stone-weight/solution/pai-xu-zhan-dui-lie-zhi-xing-yong-shi-0-ms-zai-suo/

方法 1：维护一个最大堆

```
class Solution {
    public int lastStoneWeight(int[] stones) {
        int n = stones.length;
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(n, (o1, o2
) -> -o1 + o2);
        for (int stone : stones){
            maxHeap.add(stone);
        }
        while (maxHeap.size() > 1) {
            Integer head1 = maxHeap.poll();
            Integer head2 = maxHeap.poll();
```

```
                maxHeap.offer(head1 - head2);
            }
            return maxHeap.poll();
        }
    }
```

方法 2：使用内置函数 Arrays.sort

```
class Solution {
    public int lastStoneWeight(int[] stones) {
        if (stones == null || stones.length == 0) return 0;
        int n = stones.length;
        for (int i = 0; i < n - 1; ++i){
            Arrays.sort(stones);
            stones[n - 1] -= stones[n - 2];
            stones[n - 2] = 0;
        }
        return stones[n - 1];
    }
}
```

26. 马戏团人塔（面试题 17.08）

面试题 17.08. 马戏团人塔

难度 中等 42 收藏 分享 切换为英文 关注 反馈

有个马戏团正在设计叠罗汉的表演节目，一个人要站在另一人的肩膀上。出于实际和美观的考虑，在上面的人要比下面的人矮一点且轻一点。已知马戏团每个人的身高和体重，请编写代码计算叠罗汉最多能叠几个人。

示例：

```
输入：height = [65,70,56,75,60,68] weight = [100,150,90,190,95,110]
输出：6
解释：从上往下数，叠罗汉最多能叠 6 层：(56,90), (60,95), (65,100), (68,110), (70,150), (75,190)
```

提示：

- `height.length == weight.length <= 10000`

https://leetcode-cn.com/problems/circus-tower-lcci/solution/xian-gen-ju-shen-gao-pai-xu-ruo-shen-gao-yi-yang-z/

更多硬核内容就在公众号「yes 的练级攻略」

温馨提示：
题目要求是"矮一点 且 轻一点"，是严格要求 `<` 而不是 `<=` 。

官方测试用例

自己可以用来小测的测试用例(答案为 `3` 而不是 `5`):

```
[1, 2, 2, 2, 3]
[4, 5, 6, 7, 8]
```

思路

题目要求在 `2` 个维度上(即身高 + 体重)同时保持严格递增。

那么我们可以先将其中一个维度排好序，以保证在一个维度上保持递增（此时并非严格递增）；
之后就可以专注于处理另一个维度。

具体而言：
先根据**身高 升序排序**，若身高一样则根据**体重 降序排序。**
身高排序好之后，剩余待处理的就是体重。
处理体重的问题就是处理最长递增子序列的问题。

那么仔细想想为什么身高相同时，**体重需要降序排序**呢?

其实很简单的道理，将**身高相同**的人看成 `1` 个集合，若他们都按照**体重升序**来排序，则之后的二分法处理中，**有一定的概率**会在这集合中取 `>= 2` **个人**作为最终结果。
为什么说是"**有一定的概率**会在这集合中取 `>= 2` **个人**作为最终结果"呢?
比如:
身高: `[1, 2, 2, 3, 4]`
体重: `[1, 3, 4, 5, 7]`
显然，在身高升序排序，且身高相同则体重**升序排序**的**预处理**后，对体重进行二分查找得到的最长递增子序列的结果是 `[1, 3, 4, 5, 7]` ，而体重为3、4的那2个人身高相同，不符合题意。
比如:
身高: `[1, 2, 2, 3, 4]`
体重: `[4, 1, 2, 5, 7]`
显然，在身高升序排序，且身高相同则体重升序排序的预处理后，对体重进行二分查找得到的最长递增子序列的结果是 `[4, 5, 7]` ，这其中并没有包含身高为 `2` 的那 `2` 个人。(简单说就是运气好罢了，结果刚好符合题意)。

因为是**体重升序**，二分法处理时(从左往右遍历)，身高相同的人的集合 **自己本身**也**有可能**构成上升子序列，就使得最终答案内可能包含 身高相同，体重递增的结果。

同样，将**身高相同**的人看成 `1` 个集合，若他们都按照**体重降序**来排序，在二分法顺序遍历的过程中，是**无法**在这个集合中取 `>= 2` **个人**作为最终结果(即，要么取只能取1人，要么不取)。
因为是**体重降序**，二分法处理时(从左往右遍历)，身高相同的人的集合 **自己本身**是**无法构成**上升子序列，也就**不会出现**最终答案内包含 身高相同，体重递增的情况。

```
class Solution {
    public int bestSeqAtIndex(int[] height, int[] weight) {
        int len = height.length;
        int[][] person = new int[len][2];
        for (int i = 0; i < len; ++i)
            person[i] = new int[]{height[i], weight[i]};
        Arrays.sort(person, (a, b) -> a[0] == b[0] ? b[1] - a[1] : a[0]
 - b[0]);
```

```
            int[] dp = new int[len];
            int res = 0;
            for (int[] pair : person) {
                int i = Arrays.binarySearch(dp, 0, res, pair[1]);
                if (i < 0)
                    i = -(i + 1);
                dp[i] = pair[1];
                if (i == res)
                    ++res;
            }
            return res;
        }
}
```

27. 交换和（面试题 16.21）

面试题 16.21. 交换和

难度 中等 11 收藏 分享 切换为英文 关注 反馈

给定两个整数数组，请交换一对数值（每个数组中取一个数值），使得两个数组所有元素的和相等。

返回一个数组，第一个元素是第一个数组中要交换的元素，第二个元素是第二个数组中要交换的元素。若有多个答案，返回任意一个均可。若无满足条件的数值，返回空数组。

示例:

```
输入: array1 = [4, 1, 2, 1, 1, 2], array2 = [3, 6, 3, 3]
输出: [1, 3]
```

示例:

```
输入: array1 = [1, 2, 3], array2 = [4, 5, 6]
输出: []
```

提示:

- `1 <= array1.length, array2.length <= 100000`

https://leetcode-cn.com/problems/sum-swap-lcci/solution/java-er-fen-cha-zhao-by-gogo-49/

```
class Solution {
    public int[] findSwapValues(int[] array1, int[] array2) {
        int sum1 = 0, sum2 = 0;
        for(int c: array1) sum1 += c;
        for(int c: array2) sum2 += c;
        int in = Math.abs(sum1 - sum2);//计算数组和的差
        if(in % 2 != 0) return new int[]{};
        in /= 2;
        Arrays.sort(array2);//先排序再二分搜索
        if(sum1 < sum2)//第一个数组是需要换一个大的数字
        {
            for(int i=0;i<array1.length;i++)
            {
```

```
                int idx = Arrays.binarySearch(array2,array1[i]+in);//二分查找
                if(idx >= 0) return new int[]{ array1[i],array2[idx]};//找到了符合的
            }
        }else {
            for(int i=0;i<array1.length;i++)
            {
                int idx=Arrays.binarySearch(array2,array1[i]-in);
                if(idx>=0) return new int[]{ array1[i],array2[idx]};
            }
        }
         return new int[]{};
    }
}
```

28. 稀疏数组搜索（面试题 10.05）

面试题 10.05. 稀疏数组搜索

难度 简单 18 收藏 分享 切换为英文 关注 反馈

稀疏数组搜索。有个排好序的字符串数组，其中散布着一些空字符串，编写一种方法，找出给定字符串的位置。

示例1:

```
输入: words = ["at", "", "", "", "ball", "", "", "car", "", "","dad", "", ""], s = "ta"
输出: -1
说明: 不存在返回-1。
```

示例2:

```
输入: words = ["at", "", "", "", "ball", "", "", "car", "", "","dad", "", ""], s = "ball"
输出: 4
```

提示:

1. words的长度在[1, 1000000]之间

https://leetcode-cn.com/problems/sparse-array-search-lcci/solution/java-er-fen-cha-zhao-yong-shi-0mschao-guo-100-by-w/

解题思路

除了空字符串外剩余字符串已经排好序，要想利用已排序的特性最好的方式就是二分查找(空串特殊处理)。
1）将左右边界移动至最近的非空串位置；
2）折半遇到空串则向右移动至第1个非空串，根据这个非空串确定左右边界如何变化；

折半查找会涉及终止条件(i<j还是i<=j)、移动幅度(如i=mid还是i=mid+1)的细节问题；要想不出错的话需要关注两点：
1）只要目标存在绝对不能超出[i,j]的范围；
2）当i和j邻近的时候不能出现死循环。

```
class Solution {
    public int findString(String[] words, String s) {
        if(words.length == 1 && words[0].equals(s))
```

```
            return 0;
        int i = 0, j = words.length - 1;
        while(i <= j){
            while(i < j && words[i].equals("")) i ++;
            while(i < j && words[j].equals("")) j --;
            if(i <= j){
                int mid = (i + j) / 2;
                int tmp = mid;
                while(tmp < j && words[tmp].equals("")) tmp ++;
                if(words[tmp].compareTo(s) > 0){
                    j = mid - 1;
                }else if(words[tmp].compareTo(s) < 0){
                    i = tmp + 1;
                }else{
                    return tmp;
                }
            }
        }
        return -1;
    }
}
```

29. 稀疏相似度（面试题 17.26）

面试题 17.26. 稀疏相似度

难度 困难 13 收藏 分享 切换为英文 关注 反馈

两个(具有不同单词的)文档的交集(intersection)中元素的个数除以并集(union)中元素的个数，就是这两个文档的相似度。例如，{1, 5, 3} 和 {1, 7, 2, 3} 的相似度是 0.4，其中，交集的元素有 2 个，并集的元素有 5 个。给定一系列的长篇文档，每个文档元素各不相同，并与一个 ID 相关联。它们的相似度非常"稀疏"，也就是说任选 2 个文档，相似度都很接近 0。请设计一个算法返回每对文档的 ID 及其相似度。只需输出相似度大于 0 的组合。请忽略空文档。为简单起见，可以假定每个文档由一个含有不同整数的数组表示。

输入为一个二维数组 `docs`，`docs[i]` 表示 id 为 `i` 的文档。返回一个数组，其中每个元素是一个字符串，代表每对相似度大于 0 的文档，其格式为 `{id1},{id2}: {similarity}`，其中 `id1` 为两个文档中较小的 id，`similarity` 为相似度，精确到小数点后 4 位。以任意顺序返回数组均可。

示例:

输入:

```
[
  [14, 15, 100, 9, 3],
  [32, 1, 9, 3, 5],
  [15, 29, 2, 6, 8, 7],
  [7, 10]
]
```

输出:

```
[
  "0,1: 0.2500",
  "0,2: 0.1000",
  "2,3: 0.1429"
]
```

提示:

- `docs.length <= 500`
- `docs[i].length <= 500`

https://leetcode-cn.com/problems/sparse-similarity-lcci/solution/chao-xiang-xi-ti-jie-qing-song-kan-dong-shuang-bai/

【核心思想】

- 这道题对于做过搜索引擎、学过信息检索的读者来说，应该不会陌生感，这道题的核心思想就是：**倒排索引**
- 思维转变一下，不是把两个数组比较，看有没有重复的。而是每次的数字出现，都去检查一下是否出现过，在哪个数组里出现的

【思路】

- 遍历 `docs[][]` ，将数字【在哪个数组中出现的】保存在map中，map的key就是遍历到的数字，value是一个由数组index组成的list
- 用一个 `help` 矩阵来存储数组与数组之间有多少交集
- 最后遍历 `help` 数组，按指定格式输出

【备注】

- 代码不难，可以轻松看懂
- 思想很经典
- 双百通过

```
class Solution {
    public List<String> computeSimilarities(int[][] docs) {
        List<String> ans = new ArrayList<>();
        Map<Integer, List<Integer>> map = new HashMap<>();
        int[][] help = new int[docs.length][docs.length];
        for (int i = 0; i < docs.length; i++) {
            for (int j = 0; j < docs[i].length; j++) {
                List<Integer> list = map.get(docs[i][j]);
                if (list == null) {
                    list = new ArrayList<>();
                    map.put(docs[i][j], list);
                } else {
                    for (Integer k : list) {
                        help[i][k]++;
                    }
                }
                list.add(i);
            }

            for (int k = 0; k < docs.length; k++) {
                if (help[i][k] > 0) {
                    ans.add(k + "," + i + ": " + String.format("%.4f", (double)
 help[i][k] / (docs[i].length + docs[k].length - help[i][k])));
                }
            }
        }
        return ans;
    }
```

}

30. 数字流的秩（面试题 10.10）

面试题 10.10. 数字流的秩

难度 中等 10 收藏 分享 切换为英文 关注 反馈

假设你正在读取一串整数。每隔一段时间，你希望能找出数字 x 的秩(小于或等于 x 的值的个数)。请实现数据结构和算法来支持这些操作，也就是说：

实现 `track(int x)` 方法，每读入一个数字都会调用该方法；

实现 `getRankOfNumber(int x)` 方法，返回小于或等于 x 的值的个数。

注意：本题相对原题稍作改动

示例：

```
输入:
["StreamRank", "getRankOfNumber", "track", "getRankOfNumber"]
[[], [1], [0], [0]]
输出:
[null,0,null,1]
```

提示：

- `x <= 50000`
- `track` 和 `getRankOfNumber` 方法的调用次数均不超过 2000 次

https://leetcode-cn.com/problems/rank-from-stream-lcci/solution/er-fen-cha-zhao-arraylist-by-gfu/

```java
class StreamRank {
    private ArrayList<Integer> list;

    public StreamRank() {
        list = new ArrayList<>(50000);
    }

    public void track(int x) {
        int idx = Collections.binarySearch(list, x);
        if (idx < 0) idx = -idx - 1;
        list.add(idx, x);
    }

    public int getRankOfNumber(int x) {
        int idx = Collections.binarySearch(list, x);
        if (idx < 0) idx = -idx - 1;
        while (idx < list.size() && list.get(idx) <= x)
            ++idx;
        return idx;
    }
}
/**
 * Your StreamRank object will be instantiated and called as such:
 * StreamRank obj = new StreamRank();
```

```
* obj.track(x);
* int param_2 = obj.getRankOfNumber(x);
*/
```

十二、数字问题

1、寻找重复数（287）

287. 寻找重复数

难度 中等 782 收藏 分享 切换为英文 关注 反馈

给定一个包含 $n + 1$ 个整数的数组 *nums*，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

说明:

1. 不能更改原数组（假设数组是只读的）。
2. 只能使用额外的 $O(1)$ 的空间。
3. 时间复杂度小于 $O(n^2)$。
4. 数组中只有一个重复的数字，但它可能不止重复出现一次。

东北树小

https://leetcode-cn.com/problems/find-the-duplicate-number/solution/er-fen-fa-si-lu-ji-dai-ma-python-by-liweiwei1419/

注意：这个方法使用的前提是，题目中说：

> 给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n）。

如果测试数据不在这个范围里，二分法失效。

个人意见：

- 通过这个方法知道**二分法还可以用于确定一个有范围的整数**（这个思路很常见）；
- 本题的场景和限制是极其特殊的，实际工作中和绝大多数算法问题都不会用「时间换空间」。

这题二分和快慢指针都不是常规思路。面试的时候最好提一嘴，因为有各种限制，才用二分这种耗时的做法，用快慢指针是因为做过类似的问题。

预备知识：

> 抽屉原理：桌上有十个苹果，要把这十个苹果放到九个抽屉里，无论怎样放，我们会发现至少会有一个抽屉里面放不少于两个苹果。

思路：如果题目不限制：

1. 不能更改原数组（假设数组是只读的）；
2. 只能使用额外的 `O(1)` 的空间。

容易想到的方法有：

1. 使用哈希表判重，这违反了限制 2；
2. 将原始数组排序，排序以后，重复的数相邻，即找到了重复数，这违反了限制 1；
3. 使用类似「力扣」第 41 题：缺失的第一个正数 （原地哈希）的思路，当两个数发现要放在同一个地方的时候，就发现了这个重复的元素，这违反了限制 1；
4. 既然要定位数，这个数恰好是一个整数，可以在「整数的有效范围内」做二分查找，但是比较烦的一点是得反复看整个数组好几次，本题解就介绍通过二分法定位一个有范围的整数；
5. 还可以使用「快慢指针」来完成，不过这种做法太有技巧性了，不是通用的做法，可以查看官方题解。

方法：二分查找

思路：

- 这道题要求我们查找的数是一个整数，并且给出了这个整数的范围（在 1 和 n 之间，包括 `1` 和 `n`），并且给出了一些限制，于是可以使用二分查找法定位在一个区间里的整数；
- 二分法的思路是先猜一个数（有效范围 `[left, right]` 里的中间数 `mid`），然后统计原始数组中**小于等于**这个中间数的元素的个数 `cnt`，如果 `cnt` **严格大于** `mid`，（注意我加了着重号的部分「小于等于」、「严格大于」）。根据抽屉原理，重复元素就在区间 `[left, mid]` 里；
- 与绝大多数二分法问题的不同点是：正着思考是容易的，即：思考哪边区间存在重复数是容易的，因为有抽屉原理做保证。我们通过一个具体的例子来分析应该如何编写代码；

以 `[2, 4, 5, 2, 3, 1, 6, 7]` 为例，一共 `8` 个数，`n + 1 = 8`，`n = 7`，根据题目意思，每个数都在 `1` 和 `7` 之间。

例如：**区间 $[1, 7]$ 的中位数是 `4`，遍历整个数组，统计小于等于 `4` 的整数的个数，如果不存在重复元素，最多为 4 个。等于 4 的时候区间 $[1, 4]$ 内也可能有重复元素。但是，如果整个数组里小于等于 `4` 的整数的个数严格大于 4 的时候，就可以说明重复的数存在于区间 $[1, 4]$。**

说明：由于这个算法是空间敏感的，「用时间换空间」是反常规做法，算法的运行效率肯定不会高。

复杂度分析：

- 时间复杂度：$O(N \log N)$，二分法的时间复杂度为 $O(\log N)$，在二分法的内部，执行了一次 `for` 循环，时间复杂度为 $O(N)$，故时间复杂度为 $O(N \log N)$。
- 空间复杂度：$O(1)$，使用了一个 `cnt` 变量，因此空间复杂度为 $O(1)$。

下面这一版写法有一点点小的不同。本质上与参考代码 1 没有区别，只是在一些边界的地方，供大家参考。

287 具体代码

```
public class Solution {

    public int findDuplicate(int[] nums) {
        int len = nums.length;
        int left = 1;
        int right = len - 1;
        while (left < right) {
            // 在 Java 里可以这么用，当 left + right 溢出的时候，无符号右移保证结果依然正确
            int mid = (left + right) >>> 1;

            int cnt = 0;
            for (int num : nums) {
                if (num <= mid) {
                    cnt += 1;
                }
            }

            // 根据抽屉原理，小于等于 4 的个数如果严格大于 4 个
            // 此时重复元素一定出现在 [1, 4] 区间里
            if (cnt > mid) {
                // 重复元素位于区间 [left, mid]
                right = mid;
```

```
            } else {
                // if 分析正确了以后，else 搜索的区间就是 if 的反面
                // [mid + 1, right]
                left = mid + 1;
            }
        }
        return left;
    }
}
```

2、数组中重复的数字（剑指 3）

剑指 Offer 03. 数组中重复的数字

难度 简单 142 收藏 分享 切换为英文 关注 反馈

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3

限制：

2 <= n <= 100000

https://leetcode-cn.com/problems/shu-zu-zhong-zhong-fu-de-shu-zi-lcof/solution/java-de-4chong-jie-fa-by-sdwwld/

1，使用集合set

最简单的方式就是把数组中的元素一个个加入到集合set中，加入的时候如果有重复的，则直接返回

方法 1：HashSet

```
class Solution {
    public int findRepeatNumber(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for (int num : nums){
            if (!set.add(num)) return num;
        }
        return -1;
    }
}
```

4，放到指定的位置

我们还可以不使用临时数组，我们在遍历的时候把数组nums中的值放到对应的位置上，比如某个元素是5，我们就把他放到nums[5]中，每次放入的时候查看一下这个位置是否放入了正确的值，如果已经放入了正确的值，就说明重复了，我们直接返回即可。

方法 2：原地

```
class Solution {
    public int findRepeatNumber(int[] nums) {
        for (int i = 0; i < nums.length; i++){
            if (i == nums[i]) continue; //位置正确，先不用管
            if (nums[i] == nums[nums[i]]) return nums[i]; //出现了重复，直接返回
            //交换
            int tmp = nums[nums[i]];
            nums[nums[i]] = nums[i];
            nums[i] = tmp;
            //这里的 i--是为了抵消掉上面的 i++，
            //交换之后需要原地再比较
            i--;
        }
        return -1;
    }
}
```

3、缺失的第一个正数（41）

41. 缺失的第一个正数

难度 困难 663 收藏 分享 切换为英文 关注 反馈

给你一个未排序的整数数组，请你找出其中没有出现的最小的正整数。

示例 1:

```
输入：[1,2,0]
输出：3
```

示例 2:

```
输入：[3,4,-1,1]
输出：2
```

示例 3:

```
输入：[7,8,9,11,12]
输出：1
```

提示:

你的算法的时间复杂度应为O(*n*)，并且只能使用常数级别的额外空间。

https://leetcode-cn.com/problems/first-missing-positive/solution/que-shi-de-di-yi-ge-zheng-shu-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：置换

除了打标记以外，我们还可以使用置换的方法，将给定的数组「恢复」成下面的形式：

> 如果数组中包含 $x \in [1, N]$，那么恢复后，数组的第 $x - 1$ 个元素为 x。

在恢复后，数组应当有 `[1, 2, ..., N]` 的形式，但其中有若干个位置上的数是错误的，每一个错误的位置就代表了一个缺失的正数。以题目中的示例二 `[3, 4, -1, 1]` 为例，恢复后的数组应当为 `[1, -1, 3, 4]`，我们就可以知道缺失的数为 2。

那么我们如何将数组进行恢复呢？我们可以对数组进行一次遍历，对于遍历到的数 $x = nums[i]$，如果 $x \in [1, N]$，我们就知道 x 应当出现在数组中的 $x - 1$ 的位置，因此交换 $nums[i]$ 和 $nums[x - 1]$，这样 x 就出现在了正确的位置。在完成交换后，新的 $nums[i]$ 可能还在 $[1, N]$ 的范围内，我们需要继续进行交换操作，直到 $x \notin [1, N]$。

注意到上面的方法可能会陷入死循环。如果 $nums[i]$ 恰好与 $nums[x - 1]$ 相等，那么就会无限交换下去。此时我们有 $nums[i] = x = nums[x - 1]$，说明 x 已经出现在了正确的位置。因此我们可以跳出循环，开始遍历下一个数。

由于每次的交换操作都会使得某一个数交换到正确的位置，因此交换的次数最多为 N，整个方法的时间复杂度为 $O(N)$。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是数组的长度。
- 空间复杂度：$O(1)$。

```
class Solution {
    public int firstMissingPositive(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; ++i) {
            while (nums[i] > 0 && nums[i] <= n && nums[nums[i] - 1] !=
nums[i]) {
                int temp = nums[nums[i] - 1];
                nums[nums[i] - 1] = nums[i];
                nums[i] = temp;
            }
        }
        for (int i = 0; i < n; ++i) {
            if (nums[i] != i + 1) {
                return i + 1;
            }
        }
        return n + 1;
    }
}
```

4、缺失数字（268）【异或】【0~n-1 中缺失的数字（剑指 53-2）】【消失的数字（面试题 17.04）】

剑指 Offer 53 - II. 0～n-1中缺失的数字

难度 简单 51 收藏 分享 切换为英文 关注 反馈

一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

示例 1:

```
输入: [0,1,3]
输出: 2
```

示例 2:

```
输入: [0,1,2,3,4,5,6,7,9]
输出: 8
```

限制:

`1 <= 数组长度 <= 10000`

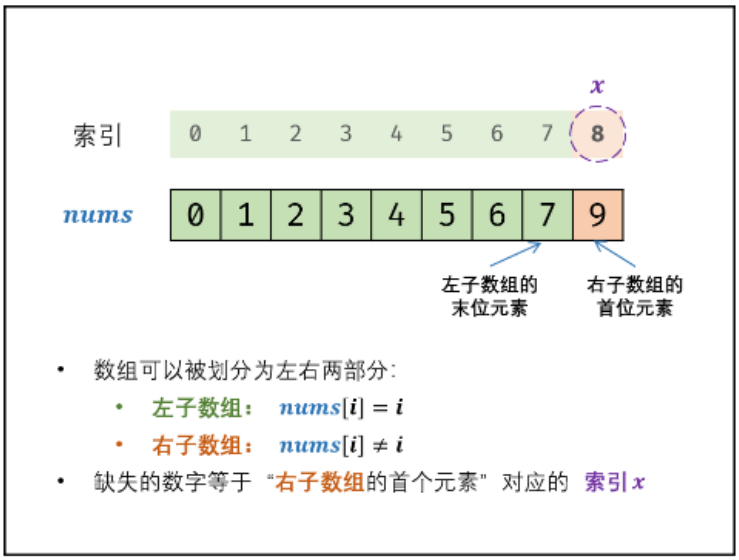
方法 1：置换（时间复杂度 o(n)）

```
class Solution {
    public int missingNumber(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++){
            while (nums[i] > 0 && nums[i] < n && nums[nums[i]] != nums[i]){
                int tmp = nums[nums[i]];
                nums[nums[i]] = nums[i];
                nums[i] = tmp;
            }
        }
        for (int i = 0; i < n; i++){
            if (nums[i] != i){
                return i;
            }
        }
        return n;
    }
}
```

https://leetcode-cn.com/problems/que-shi-de-shu-zi-lcof/solution/mian-shi-ti-53-ii-0n-1zhong-que-shi-de-shu-zi-er-f/

解题思路：

- 排序数组中的搜索问题，首先想到 **二分法** 解决。
- 根据题意，数组可以按照以下规则划分为两部分。
  - **左子数组：** $nums[i] = i$；
  - **右子数组：** $nums[i] \neq i$；
- 缺失的数字等于 **“右子数组的首位元素”** 对应的索引；因此考虑使用二分法查找 “右子数组的首位元素” 。



算法解析：

1. **初始化：** 左边界 $i = 0$，右边界 $j = len(nums) - 1$；代表闭区间 $[i, j]$。
2. **循环二分：** 当 $i \leq j$ 时循环 *（即当闭区间 $[i, j]$ 为空时跳出）*：
   1. 计算中点 $m = (i + j)//2$，其中 "//" 为向下取整除法；
   2. 若 $nums[m] = m$，则 “右子数组的首位元素” 一定在闭区间 $[m + 1, j]$ 中，因此执行 $i = m + 1$；
   3. 若 $nums[m] \neq m$，则 “左子数组的末位元素” 一定在闭区间 $[i, m - 1]$ 中，因此执行 $j = m - 1$；
3. **返回值：** 跳出时，变量 i 和 j 分别指向 “右子数组的首位元素” 和 “左子数组的末位元素” 。因此返回 i 即可。

复杂度分析：

- **时间复杂度** $O(logN)$：二分法为对数级别复杂度。
- **空间复杂度** $O(1)$：几个变量使用常数大小的额外空间。

方法 2：二分法（时间复杂度 o(logn)）

```
class Solution {
    public int missingNumber(int[] nums) {
        int i = 0, j = nums.length - 1;
        while(i <= j) {
            int m = (i + j) / 2;
            if(nums[m] == m) i = m + 1;
            else j = m - 1;
        }
        return i;
    }
```

更多硬核内容就在公众号「yes 的练级攻略」

}

https://leetcode-cn.com/problems/missing-number/solution/que-shi-shu-zi-by-leetcode/

方法三：位运算

分析

由于异或运算（XOR）满足结合律，并且对一个数进行两次完全相同的异或运算会得到原来的数，因此我们可以通过异或运算找到缺失的数字。

算法

我们知道数组中有 n 个数，并且缺失的数在 $[0..n]$ 中。因此我们可以先得到 $[0..n]$ 的异或值，再将结果对数组中的每一个数进行一次异或运算。未缺失的数在 $[0..n]$ 和数组中各出现一次，因此异或后得到 0。而缺失的数字只在 $[0..n]$ 中出现了一次，在数组中没有出现，因此最终的异或结果即为这个缺失的数字。

在编写代码时，由于 $[0..n]$ 恰好是这个数组的下标加上 n，因此可以用一次循环完成所有的异或运算，例如下面这个例子：

下标	0	1	2	3
数字	0	1	3	4

可以将结果的初始值设为 n，再对数组中的每一个数以及它的下标进行一个异或运算，即：

$$\begin{aligned}\text{missing} &= 4 \wedge (0 \wedge 0) \wedge (1 \wedge 1) \wedge (2 \wedge 3) \wedge (3 \wedge 4) \\ &= (4 \wedge 4) \wedge (0 \wedge 0) \wedge (1 \wedge 1) \wedge (3 \wedge 3) \wedge 2 \\ &= 0 \wedge 0 \wedge 0 \wedge 0 \wedge 2 \\ &= 2\end{aligned}$$

就得到了缺失的数字为 2。

复杂度分析

- 时间复杂度：$O(n)$。这里假设异或运算的时间复杂度是常数的，总共会进行 $O(n)$ 次异或运算，因此总的时间复杂度为 $O(n)$。
- 空间复杂度：$O(1)$。算法中只用到了 $O(1)$ 的额外空间，用来存储答案。

方法 3：异或

```
class Solution {
    public int missingNumber(int[] nums) {
        int missing = nums.length;
        for (int i = 0; i < nums.length; i++) {
            missing ^= i ^ nums[i];
        }
        return missing;
    }
}
```

方法四：数学

分析

我们可以用 高斯求和公式 求出 $[0..n]$ 的和，减去数组中所有数的和，就得到了缺失的数字。高斯求和公式即

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

算法

我们在线性时间内可以求出数组中所有数的和，并在常数时间内求出前 $n+1$ 个自然数（包括 0）的和，将后者减去前者，就得到了缺失的数字。

Java | Python

```
class Solution {
    public int missingNumber(int[] nums) {
        int expectedSum = nums.length*(nums.length + 1)/2;
        int actualSum = 0;
        for (int num : nums) actualSum += num;
        return expectedSum - actualSum;
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$。求出数组中所有数的和的时间复杂度为 $O(n)$，高斯求和公式的时间复杂度为 $O(1)$，因此总的时间复杂度为 $O(n)$。
- 空间复杂度：$O(1)$。算法中只用到了 $O(1)$ 的额外空间，用来存储答案。

方法 2：数学（高斯求和公式）

```
class Solution {
    public int missingNumber(int[] nums) {
        int expectedSum = nums.length*(nums.length + 1)/2;
        int actualSum = 0;
        for (int num : nums) actualSum += num;
        return expectedSum - actualSum;
    }
}
```

5、找到所有数组中消失的数字（448）

448. 找到所有数组中消失的数字

难度 简单 409 收藏 分享 切换为英文 关注 反馈

给定一个范围在 1 ≤ a[i] ≤ n (n = 数组大小) 的 整型数组，数组中的元素一些出现了两次，另一些只出现一次。

找到所有在 [1, n] 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为 O(n) 的情况下完成这个任务吗？你可以假定返回的数组不算在额外空间内。

示例:

输入:
[4,3,2,7,8,2,3,1]

输出:
[5,6]

https://leetcode-cn.com/problems/find-all-numbers-disappeared-in-an-array/solution/zhao-dao-

suo-you-shu-zu-zhong-xiao-shi-de-shu-zi-2/

方法二：原地修改

- 我们需要知道数组中存在的数字，由于数组的元素取值范围是 `[1, N]`，所以我们可以不使用额外的空间去解决它。
- 我们可以在输入数组本身以某种方式标记已访问过的数字，然后再找到缺失的数字。

算法：

- 遍历输入数组的每个元素一次。
- 我们将把 `|nums[i]|-1` 索引位置的元素标记为负数。即 $nums[|nums[i]| - 1] \times -1$。
- 然后遍历数组，若当前数组元素 `nums[i]` 为负数，说明我们在数组中存在数字 `i+1`。

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：$O(1)$，因为我们在原地修改数组，没有使用额外的空间。

https://leetcode-cn.com/problems/find-all-numbers-disappeared-in-an-array/solution/ji-yu-guan-fang-fang-fa-er-de-xiang-xi-jiang-jie-b/

```
/*
 * 这个题目要求找出  1—nums.lenth 缺少的数
 * 举个例子:
 * int[] nums={2,3,1,4,4};
 * 这个完整的数组应该包含这些元素{1,2,3,4,5};
 * 所以nums它这个里面缺少5
 * 我们先看2，它的下标本来是1,所以把自己下标的元素（也就是数组nums的3）变成负的
 *  然后看3，  它的下标本来是2,所以把自己下标的元素（也就是数组nums的1）也变成负的
 *  然后是1，  它的下标本来是0,所以把自己下标的元素（也就是数组nums的0）也变成负的
 *  然后是4，  它的下标本来是3,所以把自己下标的元素（也就是数组nums的4）也变成负的
 *  然后是4，  它的下标本来是3,所以把自己下标的元素（也就是数组nums的4）也变成负的
 * 到最后这个数组下标为4的元素（也就是第二个4）还是正数，因为它缺少元素5来使它变成负的
 */
```

```java
class Solution {
    public List<Integer> findDisappearedNumbers(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            int newIndex = Math.abs(nums[i]) - 1;
            if (nums[newIndex] > 0) {
                nums[newIndex] *= -1;
            }
        }
        List<Integer> result = new LinkedList<Integer>();
        for (int i = 1; i <= nums.length; i++) {

            if (nums[i - 1] > 0) {
                result.add(i);
            }
        }
        return result;
    }
}
```

6、消失的两个数字（面试 17.19）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 17.19. 消失的两个数字

难度 困难 21 收藏 分享 切换为英文 关注 反馈

给定一个数组，包含从 1 到 N 所有的整数，但其中缺了两个数字。你能在 O(N) 时间内只用 O(1) 的空间找到它们吗？

以任意顺序返回这两个数字均可。

示例 1:

```
输入: [1]
输出: [2,3]
```

示例 2:

```
输入: [2,3]
输出: [1,4]
```

提示：

- `nums.length <= 30000`

https://leetcode-cn.com/problems/missing-two-lcci/solution/java-shi-jian-fu-za-du-onkong-jian-fu-za-du-o1-b-3/

计 squareSub = miss1^2 + miss2^2, sub = miss1+miss2;
也即记录两个缺失数字的 平方和 和 普通和
然后从1 到能取到的最大数字sub遍历，找到Math.sqrt(squareSub-i*i)==Integer 并且该两数和为sub

注：平方和公式 1^2+2^2+...+n^2 = n(n+1)(2n+1)/6

方法 1：数学

```java
class Solution {
    public int[] missingTwo(int[] nums) {
        if(nums == null || nums.length == 0)
            return new int[]{1,2};
        long sum = 0;
        long squareSum = 0;
        for(int i = 0; i < nums.length; ++i){
            squareSum+=(long)nums[i]*(long)nums[i];
            sum += (long)nums[i];
        }
        long n = nums.length+2;
        long squareTotal = n*(n+1)*(2*n+1)/6;
        long squareSub = squareTotal - squareSum;  //缺失两数的平方和
        long total = (n*(n+1)/2);
        long sub = (total - sum);   //缺失两数的普通和
        int[] ans = new int[2];
        for(long i = 1; i < sub; ++i){
            double d = Math.sqrt(squareSub-i*i);
            if(d==(int)d && sub == i+(int)d){
                ans[0] = (int)i;
```

```
                ans[1] = (int)d;
                break;
            }
        }
        return ans;
    }
}
```

https://leetcode-cn.com/problems/missing-two-lcci/solution/xiang-dang-yu-yi-huo-qiu-jie-1-nge-shu-zi-zhong-zh/

解题思路

题意中, 1到N的所有正整数中, 缺失了两个数字，也就是在随后给定的数组nums中包含的数字个数为 N-2, 即 nums.length = N-1, 得到最大的数字N=nums.length+2;

这个题相当于给定两个数组, 一个数组为nums, 数组nums中的元素的最大值为N, 另一个数组为{1, 2, ... , N}, 记该数组为temp, 求这两个数组中只出现一次的数字;

声明变量res=0, 先遍历一遍数组nums和temp, 并将这两个数组中的元素进行异或运算, 异或运算结果记录为res;

异或运算的原则是二进制中相同位的值相同, 则该位的值为0, 相同位的值不同, 则该位为1;

异或运算结果res就是缺失两个数字的异或运算结果;

找到res在二进制下值为1的位，当该位的值为1时, 说明缺失的两个数字在该位的值不同, 以此来给数组nums和temp中的数字分组, 结果分为两组, 记为nums1, nums2;

再分别对nums1 和 nums2的元素进行异或运算, 剩下的两个异或运算结果num1和num2即为缺失的两个数字;

方法 2：位运算

```
class Solution {
    public int[] missingTwo(int[] nums) {
        int[] out = new int[2];
        int res = 0;
        int len = nums.length;
        for(int i=0;i<len;i++){
            res ^= nums[i];
        }
        for(int i=1;i<=len+2;i++){
            res ^= i;
        }
        int cnt = 0;
        while((res&1)!=1){
            cnt++;
            res = res>>1;
        }
        int num1 = 0;
        int num2 = 0;
        for(int i=0;i<len;i++){
            if(((nums[i]>>cnt)&1)==1){
                num1 ^=nums[i];
            }else{
```

```
                num2 ^=nums[i];
            }
        }
        for(int i=1;i<=len+2;i++){
            if(((i>>cnt)&1)==1){
                num1 ^= i;
            }else{
                num2 ^= i;
            }
        }
        out[0] = num1;
        out[1] = num2;
        return out;
    }
}
```

7、缺失区间（163）

163. 缺失的区间

难度 中等 25 收藏 分享 切换为英文 关注 反馈

给定一个排序的整数数组 ***nums***，其中元素的范围在 **闭区间 [*lower, upper*]** 当中，返回不包含在数组中的缺失区间。

示例：

输入： ***nums*** = [0, 1, 3, 50, 75], ***lower*** = 0 和 ***upper*** = 99,
输出： ["2", "4->49", "51->74", "76->99"]

https://leetcode-cn.com/problems/missing-ranges/solution/163-que-shi-de-qu-jian-by-klb/

解题思路

从题意可知， nums 肯定在 [lower, upper] 闭区间内。

遍历 nums 数组，定义一个变量 pre 用来表示遍历到的数字的前一个数字。

pre 初始值为 lower - 1.

每遍历一个数字：

1、如果 nums[i] - pre == 2，说明 pre 和 nums[i] 中间缺失了一个数字，这个缺失的数字就是 pre + 1，也可以用 nums[i] - 1 表示。

2、如果 nums[i] - pre >= 3，说明 pre 和 nums[i] 之间缺失了两个以上的数字，缺失的数字的范围是 pre + 1 到 nums[i] - 1。

3、每处理完一个 nums[i]，更新 pre 为 nums[i]。即，对于 nums[i+1] 来说，它的前一个数字就是 nums[i]。

4、遍历完毕 nums[] 数组，此时 pre == nums[nums.length - 1],

4.1 如果 upper - pre == 1，说明 nums[] 数组最后一个数字尚未达到边界，还差一个，这个数字为 pre + 1，也可以用 upper - 1 表示；

4.2 如果 upper - pre >= 2，说明 nums[] 数组最后一个数字尚未到达边界，还差两个以上的数字，区间为 pre + 1 到 upper。

```
class Solution {
```

```
    public List<String> findMissingRanges(int[] nums, int lower, int up
per) {
        List<String> ans = new ArrayList<>();
        long pre = (long) lower - 1;    // 防止 lower 就是 int 最小值，减
一就直接溢出了
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] - pre == 2) {
                ans.add(String.valueOf(pre + 1));
            } else if (nums[i] - pre >= 3) {
                ans.add((pre + 1) + "->" + (nums[i] - 1));
            }
            pre = nums[i]; // 'int' to 'long'
        }
        if (upper - pre == 1) {
            ans.add(String.valueOf(pre + 1));
        } else if (upper - pre >= 2) {
            ans.add((pre + 1) + "->" + upper);
        }
        return ans;
    }
}
```

8、只出现一次的数字（136）【异或】

136. 只出现一次的数字

难度 简单 1324 收藏 分享 切换为英文 关注 反馈

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

```
输入: [2,2,1]
输出: 1
```

示例 2:

```
输入: [4,1,2,1,2]
输出: 4
```

相似但是不一样的题目

剑指 50，第一个只出现一次的字符

https://leetcode-cn.com/problems/single-number/solution/xue-suan-fa-jie-guo-xiang-dui-yu-guo-cheng-bu-na-y/

方法 1：时间复杂度 o(n)，空间复杂度 o(1)，要 4ms

```
class Solution {
    public int singleNumber(int[] nums) {
```

```
        Arrays.sort(nums);
        for (int i = 1; i  < nums.length; i+=2) {
            if (nums[i] != nums[i-1]) {
                return nums[i-1];
            }
        }
        return nums[nums.length - 1];
    }
}
```

方法 2：更快的做法，用异或，只要 1ms

```
class Solution {
    public int singleNumber(int[] nums) {
        int ans = nums[0];
        for (int i = 1; i < nums.length; i++) {
            ans = ans ^ nums[i];
        }
        return ans;
    }
}
```

9、只出现一次的数字 2（137）【异或】【数组中数字出现的次数 2（剑指 56-2）】

137. 只出现一次的数字 II

难度 中等 395 收藏 分享 切换为英文 关注 反馈

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现了三次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

```
输入: [2,2,3,2]
输出: 3
```

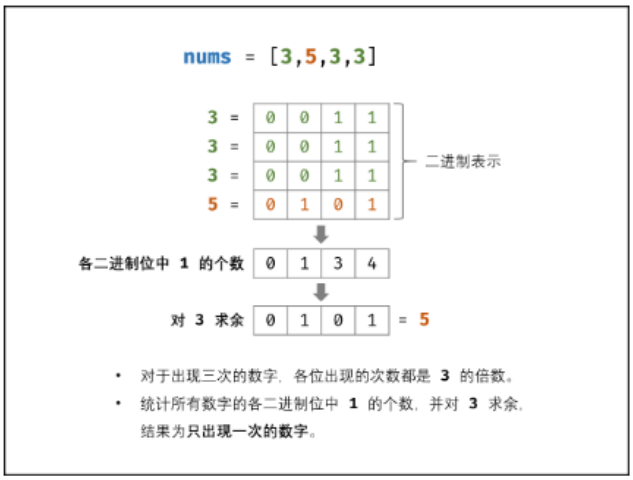
示例 2:

```
输入: [0,1,0,1,0,1,99]
输出: 99
```

https://leetcode-cn.com/problems/single-number-ii/solution/single-number-ii-mo-ni-san-jin-zhi-fa-by-jin407891/

解题思路：

如下图所示，考虑数字的二进制形式，对于出现三次的数字，各 **二进制位** 出现的次数都是 3 的倍数。
因此，统计所有数字的各二进制位中 1 的出现次数，并对 3 求余，结果则为只出现一次的数字。

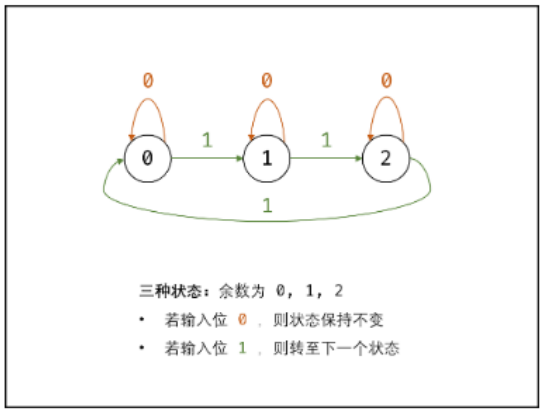


方法一：有限状态自动机

各二进制位的 **位运算规则相同** ，因此只需考虑一位即可。如下图所示，对于所有数字中的某二进制位 1 的个数，存在 3 种状态，即对 3 余数为 $0, 1, 2$ 。

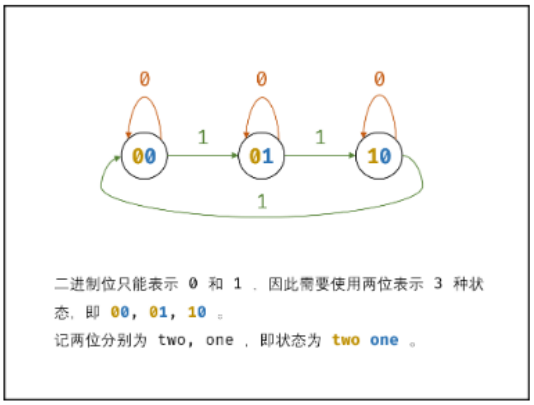
- 若输入二进制位 1 ，则状态按照以下顺序转换；
- 若输入二进制位 0 ，则状态不变。

$$0 \to 1 \to 2 \to 0 \to \cdots$$



如下图所示，由于二进制只能表示 $0, 1$ ，因此需要使用两个二进制位来表示 3 个状态。设此两位分别为 two , one ，则状态转换变为：

$$00 \rightarrow 01 \rightarrow 10 \rightarrow 00 \rightarrow \cdots$$



接下来，需要通过 **状态转换表** 导出 **状态转换的计算公式** 。首先回忆一下位运算特点，对于任意二进制位 x ，有：

- 异或运算： `x ^ 0 = x` ， `x ^ 1 = ~x`
- 与运算： `x & 0 = 0` ， `x & 1 = x`

计算 one 方法：

设当前状态为 $two\ one$ ，此时输入二进制位 n 。如下图所示，通过对状态表的情况拆分，可推出 one 的计算方法为：

```
if two == 0:
  if n == 0:
    one = one
  if n == 1:
    one = ~one
if two == 1:
    one = 0
```

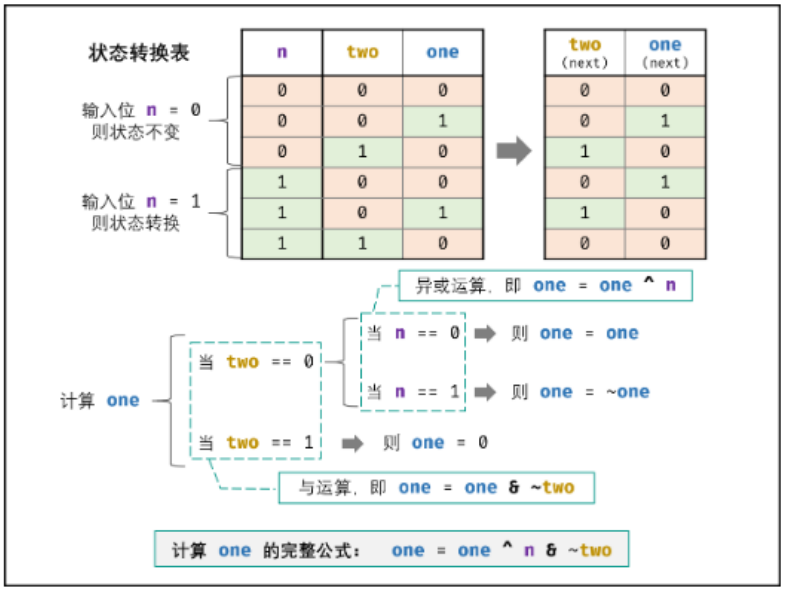
引入 **异或运算** ，可将以上拆分简化为：

```
if two == 0:
    one = one ^ n
if two == 1:
    one = 0
```

引入 **与运算** ，可继续简化为：

```
one = one ^ n & ~two
```

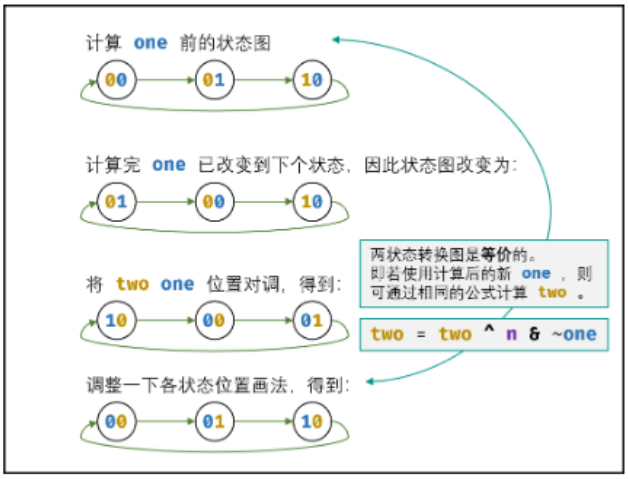
更多硬核内容就在公众号「yes 的练级攻略」



计算 two 方法：

由于是先计算 one ，因此应在新 one 的基础上计算 two 。
如下图所示，修改为新 one 后，得到了新的状态图。观察发现，可以使用同样的方法计算 two ，即：

```
two = two ^ n & ~one
```



返回值：

以上是对数字的二进制中“一位”的分析，而 `int` 类型的其他 31 位具有相同的运算规则，因此可将以上公式直接套用在 32 位数上。

遍历完所有数字后，各二进制位都处于状态 00 和状态 01 （取决于“只出现一次的数字”的各二进制位是 1 还是 0 ），而此两状态是由 one 来记录的（此两状态下 $twos$ 恒为 0 ），因此返回 $ones$ 即可。

复杂度分析：

- **时间复杂度 $O(N)$ ：** 其中 N 位数组 $nums$ 的长度；遍历数组占用 $O(N)$ ，每轮中的常数个位运算操作占用 $O(32 \times 3 \times 2) = O(1)$ 。
- **空间复杂度 $O(1)$ ：** 变量 $ones$, $twos$ 使用常数大小的额外空间。

☆ 收藏

137 具体代码

```
class Solution {
    public int singleNumber(int[] nums) {
        int one = 0, two = 0;
        for (int num : nums){
            one = one ^ num & ~two;
            two = two ^ num & ~one;
        }
        return one;
    }
}
```

10、只出现一次的数字 3（260）【异或】【数组中数字出现的次数（剑指 56-1）】

260. 只出现一次的数字 III

难度 中等 268 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。 找出只出现一次的那两个元素。

示例：

输入：[1,2,1,3,2,5]
输出：[3,5]

注意：

1. 结果输出的顺序并不重要，对于上面的例子， [5, 3] 也是正确答案。
2. 你的算法应该具有线性时间复杂度。你能否仅使用常数空间复杂度来实现？

通过次数 27,283 提交次数 37,198

https://leetcode-cn.com/problems/single-number-iii/solution/java-yi-huo-100-yao-shi-kan-bu-dong-wo-jiu-qu-chu-/

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

思路：

现在数组中有两个数字只出现1次，直接异或一次只能得到这两个数字的异或结果，但光从这个结果肯定无法得到这个两个数字。因此基于single number I 的思路——**数组只能有一个数字出现1次。**

设题目中这两个只出现1次的数字分别为A和B，如果能将A，B分开到二个数组中，那显然符合“异或”解法的关键点了。因此这个题目的关键点就是将A，B分开到二个数组中。由于A，B肯定是不相等的，因此在二进制上必定有一位是不同的。**根据这一位是0还是1可以将A，B分开到A组和B组。**而这个数组中其它数字要么就属于A组，要么就属于B组。再对A组和B组分别执行“异或”解法就可以得到A，B了。而要判断A，B在哪一位上不相同，只要根据A异或B的结果就可以知道了，**这个结果在二进制上为1的位就说明A，B在这一位上是不相同的。**

比如

```
int a[] = {1, 1, 3, 5, 2, 2}
```

整个数组异或的结果为3^5，即 **0x0011 ^ 0x0101 = 0x0110**

对0x0110，第1位（由低向高，从0开始）就是1。因此整个数组根据第1位是0还是1分成两组。

```
a[0] =1  0x0001  第一组

a[1] =1  0x0001  第一组

a[2] =3  0x0011  第二组

a[3] =5  0x0101  第一组

a[4] =2  0x0010  第二组

a[5] =2  0x0010  第二组
```

第一组有{1,1,5}，第二组有{3,2,2}，然后对这二组分别执行“异或”解法就可以得到5和3了。

代码：

时间复杂度O(n)，空间复杂度O(1)

```java
class Solution {
    public int[] singleNumber(int[] nums) {
        int[] res = new int[2];
        if (nums == null || nums.length < 2) return res;
        // 得到 两个只出现 1 次的元素异或以后的结果
        int XOR = 0;
        for (int num : nums){
            XOR ^= num;
        }
        // 对 XOR 后的值取反加 1 并按位与，找出最右侧的 1
        int Diff = XOR & (~XOR + 1);
        // 依据 Diff 对数据分组
        for (int num : nums) {
            if ((num & Diff) == 0) {
                res[0] ^= num;
            } else {
                res[1] ^= num;
            }
```

```
            }
            return res;
        }
    }
```

11、错误的集合（645）

645. 错误的集合

难度 简单 112 收藏 分享 切换为英文 关注 反馈

集合 `S` 包含从1到 `n` 的整数。不幸的是，因为数据错误，导致集合里面某一个元素复制了成了集合里面的另外一个元素的值，导致集合丢失了一个整数并且有一个元素重复。

给定一个数组 `nums` 代表了集合 `S` 发生错误后的结果。你的任务是首先寻找到重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。

示例 1:

```
输入: nums = [1,2,2,4]
输出: [2,3]
```

注意:

1. 给定数组的长度范围是 [2, 10000]。
2. 给定的数组是无序的。

https://leetcode-cn.com/problems/set-mismatch/solution/cuo-wu-de-ji-he-by-leetcode/

方法七：使用异或运算

算法

思考一个简单的问题，一个长度为 $n-1$ 的数组包含 1 到 n 中的 $n-1$ 个数字，有一个数字缺失，如何找出这个缺失的数字。首先使用 1 到 n 的所有数字做异或运算，然后再使用数组中的所有数字与这个数字异或，最后得到的值就是缺失数字。因为数字与其本身做异或运算结果为 0，因此所有数字做完异或后，剩余的就是缺失的数字。

按照这个方法，将 $nums$ 中所有的数字与 1 到 n 的所有数字做异或运算，得到的结果为 $x \hat{} y$，x 和 y 分别表示 $nums$ 中重复数字和缺失数字。

在异或结果 xor 的二进制中，值为 1 的位置表示 x 和 y 在该位置的值不同，值为 0 的位置表示 x 和 y 在该位置的值相同。我们称 xor 最右边比特位为 $rightmostbit$，且使该位置为 1。

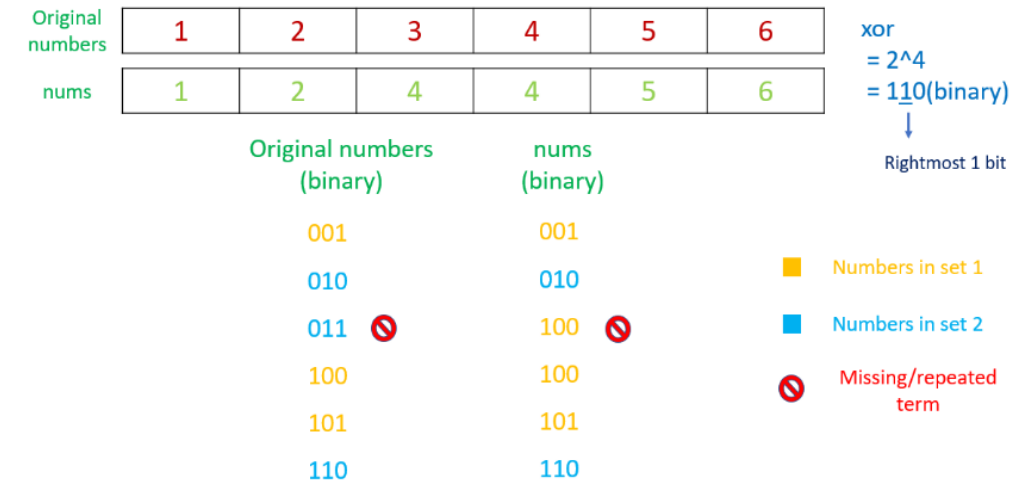
根据 $rightmostbit$ 不同将数组 $nums$ 分为两部分。第一部分所有数字的 $rightmostbit$ 为 1，第二部分所有数字的 $rightmostbit$ 为 0。那么 x 和 y 会被分配到不同的部分。此时问题就简化为最开始的简单问题。

根据 $rightmostbit$ 的不同，将 1 到 n 的所有元素分为两部分。

现在分别使用从 $nums$ 中分出来 $rightmostbit$ 为 1 的部分与 1 到 n 中分出来 $rightmostbit$ 为 1 的部分做异或。在结果中，相同的元素异或为 0，最终只会剩下重复数字或缺失数字，即 x 或 y。同理也对 $rightmostbit$ 为 0 的部分异或。

更多硬核内容就在公众号「yes 的练级攻略」

观察一个简单的例子 `[1 2 4 4 5 6]`。



最后遍历 $nums$，确定两个数字中哪个为重复数字，哪个为缺失数字。

复杂度分析

- 时间复杂度：$O(n)$。遍历 n 个元素 5 次。
- 空间复杂度：$O(1)$。使用恒定的额外空间。

方法 1：异或

```
public class Solution {
    public int[] findErrorNums(int[] nums) {
        int xor = 0, xor0 = 0, xor1 = 0;
        for (int n: nums)
            xor ^= n;
        for (int i = 1; i <= nums.length; i++)
            xor ^= i;
        int rightmostbit = xor & ~(xor - 1);
        for (int n: nums) {
            if ((n & rightmostbit) != 0)
                xor1 ^= n;
            else
                xor0 ^= n;
        }
        for (int i = 1; i <= nums.length; i++) {
            if ((i & rightmostbit) != 0)
                xor1 ^= i;
            else
                xor0 ^= i;
        }
        for (int i = 0; i < nums.length; i++) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            if (nums[i] == xor0)
                return new int[]{xor0, xor1};
        }
        return new int[]{xor1, xor0};
    }
}
```

方法六：使用额外空间

算法

已知 $nums$ 中所有数字都是正数，且处于 1 到 n 之间。遍历 $nums$ 中的所有数字，根据数字 i 找到 $nums[|i|]$，如果是第一次访问 $nums[|i|]$，将它反转为负数。如果是第二次访问，则会发现它已经是负数。因此，可以根据访问一个数字时它是否为负数找出重复数字。

完成上述操作后，所有出现过的数字对应索引处的数字都是负数，只有缺失数字 j 对应的索引处仍然是正数。

复杂度分析

- 时间复杂度：$O(n)$。在 $nums$ 上完成了两次遍历。
- 空间复杂度：$O(1)$。使用恒定的额外空间。

方法 2：使用额外空间

```
public class Solution {
    public int[] findErrorNums(int[] nums) {
        int dup = -1, missing = 1;
        for (int n: nums) {
            if (nums[Math.abs(n) - 1] < 0)
                dup = Math.abs(n);
            else
                nums[Math.abs(n) - 1] *= -1;
        }
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > 0)
                missing = i + 1;
        }
        return new int[]{dup, missing};
    }
}
```

12、情侣牵手（765）

更多硬核内容就在公众号「yes 的练级攻略」

765. 情侣牵手

难度 困难 91 收藏 分享 切换为英文 关注 反馈

N 对情侣坐在连续排列的 2N 个座位上，想要牵到对方的手。 计算最少交换座位的次数，以便每对情侣可以并肩坐在一起。 一次交换可选择任意两人，让他们站起来交换座位。

人和座位用 0 到 2N-1 的整数表示，情侣们按顺序编号，第一对是 (0, 1)，第二对是 (2, 3)，以此类推，最后一对是 (2N-2, 2N-1)。

这些情侣的初始座位 row[i] 是由最初始坐在第 i 个座位上的人决定的。

示例 1:

```
输入: row = [0, 2, 1, 3]
输出: 1
解释: 我们只需要交换row[1]和row[2]的位置即可。
```

示例 2:

```
输入: row = [3, 2, 0, 1]
输出: 0
解释: 无需交换座位，所有的情侣都已经可以手牵手了。
```

说明:

1. len(row) 是偶数且数值在 [4, 60]范围内。
2. 可以保证 row 是序列 0...len(row)-1 的一个全排列。

https://leetcode-cn.com/problems/couples-holding-hands/solution/gui-lu-zhao-dao-jiu-hen-jian-dan-shi-jian-he-kong-/

思路

这道题的前提就是N对情侣（$2N$个人）坐在$2N$个座位上，也就是说每个位置上都有人，没有空位。那么最后想要让情侣并肩坐在一起，只能是每两个座位一对情侣。而且情侣的编号是$(0,\ 1),(2,\ 3),...,(2N-2,2N-1)$,也就是说每对情侣编号中，奇数编号比偶数编号大1。发现这个规律之后就很简单了，我们从第0位开始，判断第0、2、4、...、$2N-2$位置上的人跟他们下一个位置上的人是否为情侣，如果不是，则找到他们的情侣与他们的下一个位置上的人进行交换即可。最终交换的总次数就是答案。具体代码如下：

复杂度分析

通过上述代码，明显可得
时间复杂度 $O(N)$
空间复杂度 $O(N)$

方法 1：找规律

```
class Solution {
    public int minSwapsCouples(int[] row) {
        //记录交换次数
        int cnt = 0;
        int n = row.length;
        //row 是下标-值，idx 是值-下标，因为值和下标都在 0-2N
        // 可以通过 idx 快速找到 row 某个值的下标
        int[] idx = new int[n];
        //给 idx 按其定义赋值
        for (int i = 0; i < n; i++) {
            idx[row[i]] = i;
        }

        //一对情侣，我们知道其中一个编号，就能知道另一个编号，由 next 代表
        int next = 0;
```

```
        for (int i = 0; i < n; i += 2) {
            //每两个作为一对，它们必然是紧邻的奇偶数，且偶数更小，故可推出另一个
            next = (row[i] & 1) == 0 ? row[i] + 1 : row[i] - 1;
            //row[i],row[i+1]已然一对，就跳过本轮
            if (next == row[i + 1]) {
                continue;
            }
            //否之，进行交换，使row[i+1]能和row[i]为一对
            //即将当前row[i+1]的值放到next 所在下标去，注意交换顺序
            int nextIdx = idx[next];
            idx[row[i + 1]] = nextIdx;
            row[nextIdx] = row[i + 1];
            cnt++;
        }
        return cnt;
    }
}
```

方法三： 贪心 【通过】

思路

根据我们的假设，可以制定按顺序让每张沙发上情侣开心的策略。对于每张沙发，找到沙发上第一个人的情侣，如果不在同一个沙发上，就把沙发上的第二人换成第一个人的情侣。

算法

如果一个人的编号为 `x`，那么他的情侣的编号为 `x ^ 1`，`^` 在这里是异或操作。对于每张沙发上的第一个人 `x = row[i]`，找到他们的同伴所在的位置 `row[j]`，将 `row[j]` 和 `row[i + 1]` 互相交换。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 为情侣对的数量。
- 空间复杂度：$O(1)$，互相交换不需要开辟额外的空间。

方法 2：贪心+异或

```
class Solution {
    public int minSwapsCouples(int[] row) {
        int ans = 0;
        for (int i = 0; i < row.length; i += 2) {
            int x = row[i];
            if (row[i+1] == (x ^ 1)) continue;
            ans++;
            for (int j = i+1; j < row.length; ++j) {
                if (row[j] == (x^1)) {
                    row[j] = row[i+1];
                    row[i+1] = x^1;
                    break;
                }
            }
```

```
        }
        return ans;
    }
}
```

13、中心对称数（246）

246. 中心对称数

难度 简单 16 收藏 分享 切换为英文 关注 反馈

中心对称数是指一个数字在旋转了 180 度之后看起来依旧相同的数字（或者上下颠倒地看）。

请写一个函数来判断该数字是否是中心对称数，其输入将会以一个字符串的形式来表达数字。

示例 1:

```
输入: num = "69"
输出: true
```

示例 2:

```
输入: num = "88"
输出: true
```

示例 3:

```
输入: num = "962"
输出: false
```

示例 4:

```
输入: num = "1"
输出: true
```

yes的练级攻略

https://leetcode-cn.com/problems/strobogrammatic-number/solution/java-shuang-100-ha-xi-shuang-zhi-zhen-by-siguo/

```
class Solution {
    public boolean isStrobogrammatic(String num) {
        Map<Character, Character> map = new HashMap<>();
        map.put('0', '0');
        map.put('1', '1');
        map.put('6', '9');
        map.put('8', '8');
        map.put('9', '6');  //满足要求的数字对
        int left = 0, right = num.length() - 1;     //双指针初始化
        while(left <= right){
            if(map.get(num.charAt(left)) == null || map.get(num.charAt(right)) == null)     //不满足要求的数字
                return false;
            if(map.get(num.charAt(left)) != num.charAt(right))  //check 一下是否真的中心对称
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                return false;
            left++;
            right--;
        }
        return true;
    }
}
```

14、中心对称数 2（247）

247. 中心对称数 II

难度 中等 31 收藏 分享 切换为英文 关注 反馈

中心对称数是指一个数字在旋转了 180 度之后看起来依旧相同的数字（或者上下颠倒地看）。

找到所有长度为 n 的中心对称数。

示例：

输入： n = 2
输出： ["11","69","88","96"]

禁止商用

https://leetcode-cn.com/problems/strobogrammatic-number-ii/solution/zhong-xin-dui-cheng-shu-ii-247-by-tang-yuan-27/

解题模板

```
function fn(n) {
    // 第一步：判断输入或者状态是否非法？
    if (input/state is invalid) {
        return;
    }

    // 第二步：判读递归是否应当结束?
    if (match condition) {
        return some value;
    }

    // 第三步：缩小问题规模
    result1 = fn(n1)
    result2 = fn(n2)
    ...

    // 第四步：整合结果
    return combine(result1, result2)
}
```

各

更多硬核内容就在公众号「yes 的练级攻略」

1. 判断当前情况是否非法，如果非法就立即返回，这一步也被称为完整性检查（Sanity Check）。例如，看看当前处理的情况是否越界，是否出现了不满足条件的情况。通常，这一部分代码都是写在最前面的。
2. 判断是否满足结束递归的条件。在这一步当中，处理的基本上都是一些推导过程当中所定义的初始情况。
3. 将问题的规模缩小，递归调用。在归并排序和快速排序中，我们将问题的规模缩小了一半，而在汉诺塔和解码（LeetCode91题）的例子中，我们将问题的规模缩小了一个。
4. 利用在小规模问题中的答案，结合当前的数据进行整合，得出最终的答案。

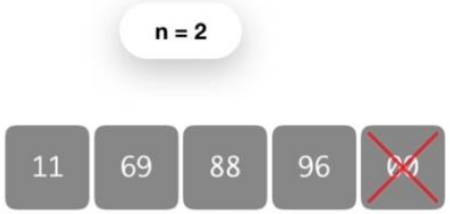
解题思路

1. 当 n=0 的时候，应该输出空字符串：“”。
2. 当 n=1 的时候，也就是长度为 1 的中心对称数有：0，1，8。
3. 当 n=2 的时候，长度为 2 的中心对称数有：11，69，88，96。注意：00 并不是一个合法的结果。
4. 当 n=3 的时候，只需要在长度为 1 的合法中心对称数的基础上，不断地在两边添加 11，69，88，96 就可以了。
   [101, 609, 808, 906, 111, 619, 818, 916, 181, 689, 888, 986]
   随着 n 不断地增长，我们只需要在长度为 n-2 的中心对称数两边添加 11，69，88，96 即可。

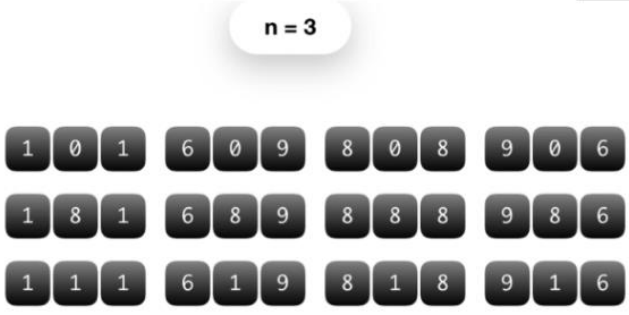
- n=0

- n=1

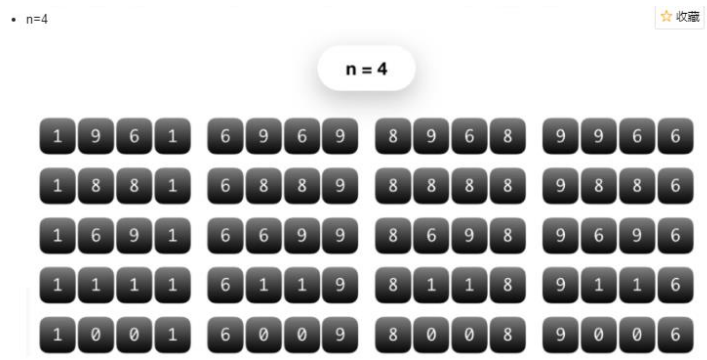
- n=2



- n=3



收藏




```java
class Solution{
    public List<String> findStrobogrammatic(int n) {
        return helper(n,n);
    }
    // n 表示，当前循环中，求得字符串长度； m 表示题目中要求的字符串长度
    public List<String> helper(int n, int m){
        // 第一步：判断输入或者状态是否合法
        if(n<0 || m<0 || n>m){
            throw  new IllegalArgumentException("invalid input");
        }
        // 第二步：判断递归是否应当结束
        if (n==0)
            return new ArrayList<>(Arrays.asList(""));
        if (n==1)
            return new ArrayList<>(Arrays.asList("0","1","8"));
        // 第三步：缩小问题规模
        List<String> list = helper(n-2, m);
        // 第四步：整合结果
        List<String> res = new ArrayList<>();
        for (String s : list){
            if (n!=m)
                // n=m 时，表示最外层处理。
                // 例如：原始需求 n=m=2， '00' 不合法
                // 若原始需求 n=m=4， 内层循环 n=2,m=4,'00';最外层循环，n=m=4 时，
'1001'
                res.add("0"+s+"0");
            res.add("1"+s+"1");
            res.add("6"+s+"9");
            res.add("8"+s+"8");
            res.add("9"+s+"6");
        }
        return res;
    }
}
```

15、中心对称数 3（248）

更多硬核内容就在公众号「yes 的练级攻略」

248. 中心对称数 III

难度 困难 24 收藏 分享 切换为英文 关注 反馈

中心对称数是指一个数字在旋转了 180 度之后看起来依旧相同的数字（或者上下颠倒地看）。

写一个函数来计算范围在 [low, high] 之间中心对称数的个数。

示例:

```
输入: low = "50", high = "100"
输出: 3
解释: 69，88 和 96 是三个在该范围内的中心对称数
```

https://leetcode-cn.com/problems/strobogrammatic-number-iii/solution/java-14msgen-ju-247de-tong-su-yi-dong-ban-ben-by-1/

```
class Solution {
    private final char[][] mapping = {{'0', '0'}, {'1', '1'}, {'8', '8'}, {'6',
 '9'}, {'9', '6'}};
    public int count = 0;
    public int strobogrammaticInRange(String low, String high) {
        int lo = low.length();
        int hi = high.length();
        for(int n = lo; n <= hi; n++) {
            char[] chs = new char[n];
            getStrobogrammatic(chs, 0, chs.length - 1, low, high);
        }
        return count;
    }
    public void getStrobogrammatic(char[] chs, int lo, int hi, String low, Stri
ng high) {
        if(lo > hi) {
            if(chs.length == 1 || chs[0] != '0') {
                String str = String.valueOf(chs);
                if(compare(str, low) && compare(high, str)) {
                    count++;
                }
            }
            return;
        }
        for(char[] map : mapping) {
            if(lo == hi && map[0] != map[1]) continue;
            chs[lo] = map[0];
            chs[hi] = map[1];
            getStrobogrammatic(chs, lo + 1, hi - 1, low, high);
        }
    }
```

```
    public boolean compare(String s1, String s2) {
        if(s1.length() == s2.length()) {
            if(s1.compareTo(s2) >= 0) {
                return true;
            }else {
                return false;
            }
        }
        return true;
    }
}
```

16、下一个数（面试题 05.04）【二进制表达式中 1 个数相同且大小最接近的两个数（一个略大，一个略小）】

面试题 05.04. 下一个数

难度 中等 12 收藏 分享 切换为英文 关注 反馈

下一个数。给定一个正整数，找出与其二进制表达式中1的个数相同且大小最接近的那两个数（一个略大，一个略小）。

示例1:

```
输入: num = 2（或者0b10）
输出: [4, 1] 或者（[0b100, 0b1]）
```

示例2:

```
输入: num = 1
输出: [2, -1]
```

提示:

1. num 的范围在[1, 2147483647]之间;
2. 如果找不到前一个或者后一个满足条件的正数，那么输出 -1。

https://leetcode-cn.com/problems/closed-number-lcci/solution/java-shuang-100-by-liu-zhao-lei-de-yu/

最接近的较小值:二进制表示中从右向左找前面有0的第一个1,"10"中的1,0换位,剩下右边的二进制位前后翻转
解释:从低位向高位找第一个可以被低位0替换的1使数变小,同时低位在保持0,1个数不变时要尽可能大,分析发现在满足前面查找条件右边的0,1分布:空,"0000"(全是0),"0011"(全1开始全0结束)和"11111"(全是1),只需要把他们前后翻转就能把所有1移动到高位实现剩余右边值最大
最接近的较大值:二进制表示中从右向左找前面有1的第一个0,"01"中的1,0换位,剩下右边的二进制位前后翻转
解释:从低位向高位找第一个可以被低位1替换的0使数变大,同时低位在保持0,1个数不变时要尽可能小,分析发现在满足前面查找条件右边的0,1分布:空,"0000"(全是0),"1100"(全0开始全1结束)和"11111"(全是1),只需要把他们前后翻转就能把所有0移动到高位实现剩余边右值最小

```
class Solution {
    public int[] findClosedNumbers(int num) {
        int [] re={-1,-1};
        int flag1=0;
        int flag2=0;
        int temp=num;
```

```
        int cur;
        for(int i=0;i<31&&(flag1<2 || flag2<2);i++){
            cur=temp&1;
            temp=temp>>1;
            if(flag1==0&&cur==0){
                flag1=1;
            }else if(flag1==1 && cur==1){
                flag1=2;
                re[1]=(int)(num-Math.pow(2,i)+Math.pow(2,i-1));
                int temp1=re[1]&(((int)Math.pow(2,i-1))-1);
                re[1]=re[1]>>(i-1);
                for(int j=0;j<i-1;j++){
                    re[1]=((re[1]<<1)|(temp1&1));
                    temp1=temp1>>1;
                }
            }
            if(flag2==0&&cur==1){
                flag2=1;
            }else if(flag2==1 && cur==0){
                flag2=2;
                re[0]=(int)(num-Math.pow(2,i-1)+Math.pow(2,i));
                int temp1=re[0]&(((int)Math.pow(2,i-1))-1);
                re[0]=re[0]>>(i-1);
                for(int j=0;j<i-1;j++){
                    re[0]=((re[0]<<1)|(temp1&1));
                    temp1=temp1>>1;
                }
            }
        }
        return re;
    }
}
```

17、翻转数位（面试题 05.03）

面试题 05.03. 翻转数位

难度 简单 15 收藏 分享 切换为英文 关注 反馈

给定一个32位整数 num，你可以将一个数位从0变为1。请编写一个程序，找出你能够获得的最长的一串1的长度。

示例 1:

输入: num = 1775(11011101111_2)
输出: 8

示例 2:

输入: num = 7(0111_2)
输出: 4

https://leetcode-cn.com/problems/reverse-bits-lcci/solution/si-lu-qing-xi-shi-yong-jian-dan-de-hui-su-suan-fa-/

运用循环遍历，进行简单的回溯。从数字的二进制的最右边进行遍历，即从i=0;到i=31;
当碰到第一个零的时候，令x=i,不要理会，接着遍历；当遇到第二个零时，令i=x;即从第二个零的下一位接着遍历。
直至结束。

```
class Solution {
    public static int reverseBits(int num) {
        int count = 0, flag = 0, max = 0, i = 0, x = 0;
        while(i <= 31){
            if(((num >> i) & 1) == 1) count++;
            else if((((num >> i) & 1) != 1) && (flag == 0)){
                count++;
                flag = 1;
                x = i;
            }
            else if((((num >> i) & 1) != 1)&&(flag == 1))
            {
                i = x;
                count = 0;
                flag = 0;
            }
            if(count > max) max = count;
            i++;
        }
        return max;
    }
}
```

十三、字符串

- 常见数据类型的转换

字符串转 int、long、double、float 的方法

```
String a = "12345";
int i1 = Integer.parseInt(a); //String 转成 int 的方法 1
System.out.println(i1); //输出 int
int i2 = Integer.valueOf(a).intValue();; //String 转成 int 的方法 2
System.out.println(i2); //输出 int
Long l1 = Long.parseLong(a); //String 转成 long 的方法 1
System.out.println(l1); //输出 long
Long l2 = Long.valueOf(a).longValue(); //String 转成 long 的方法 2
System.out.println(l2); //输出 long
double d1 = Double.parseDouble(a); //String 转成 double 的方法 1
System.out.println(d1); //输出 double
System.out.println(String.format("%.2f", d1)); //输出 double，并保留 2 位小数
double d2 = Double.valueOf(a).doubleValue(); //String 转成 double 的方法 2
System.out.println(d2); //输出 double
System.out.println(String.format("%.4f", d2)); //输出 double，并保留 4 位小数
float f1 = Float.parseFloat(a); //String 转成 float 的方法 1
System.out.println(f1); //输出 float
System.out.println(String.format("%.2f", f1)); //输出 float，并保留 2 位小数
float f2 = Float.valueOf(a).floatValue(); //String 转成 float 的方法 2
System.out.println(f2); //输出 float
System.out.println(String.format("%.4f", f2)); //输出 float，并保留 4 位小数
```

int、long、double、float 转字符串的方法

```
int i = 23556;
String is1 = String.valueOf(i); //int 转成 String 的方法 1
System.out.println(is1);
String is2 = "" + i; //int 转成 String 的方法 2
System.out.println(is2);
String is3 = Integer.toString(i); //int 转成 String 的方法 3
System.out.println(is2);
long l = 1544676767867454687L;
String ls1 = String.valueOf(l); //long 转成 String 的方法 1
System.out.println(ls1);
String ls2 = "" + l; //long 转成 String 的方法 2
System.out.println(ls2);
String ls3 = Long.toString(l); //long 转成 String 的方法 3
System.out.println(ls3);
double d = 1602.26746;
String ds1 = String.valueOf(d); //double 转成 String 的方法 1
System.out.println(ds1);
String ds2 = "" + d; //double 转成 String 的方法 2
System.out.println(ds2);
String ds3 = Double.toString(d); //double 转成 String 的方法 3
System.out.println(ds3);
float f = 2659.12F;
```

禁止商用 公众号：yes的练级攻略

```
String fs1 = String.valueOf(f); //float 转成 String 的方法 1
System.out.println(fs1);
String fs2 = "" + f; //float 转成 String 的方法 2
System.out.println(fs2);
String fs3 = Float.toString(f); //float 转成 String 的方法 3
System.out.println(fs3);
```

- String 类中常见的函数

1. 字符串拼接

```
String str1="content1";
String str2="content2";
String str3=str1+"----"+str2;
System.out.println(str3);//content1----content2
System.out.println(str1.concat(str2).concat(str3));//content1content2content1----content2
```

2. 把字符串转换成字符数组

```
Char[] ch = str1.toCharArray();
```

3. 获得字符串的子串

```
public String substring(int beginIndex)
public String substring(int beginIndex, int endIndex)
```

beginIndex -- 起始索引（包括），索引从 0 开始。endIndex -- 结束索引（不包括）。

```
"AbCA".substring(1); //bCA
"AbCA".substring(1,3); //bC
```

4. 字符串分割

split() 方法根据匹配给定的正则表达式来拆分字符串。

注意： . 、 $、 | 和 * 等转义字符，必须得加 \\。

注意：多个分隔符，可以用 | 作为连字符。

```
String strarr[]="1*2*3*4*5-6-7".split("\\*|-");
for(int i=0;i<strarr.length;i++)
System.out.println(strarr[i]);
String [] arr = str.split("\\s+"); //去除字符串中所有的空白，包括一个空格，多个空格和 tab 制表符
```

5. 获得指定位置的字符

```
str1.charAt(0)
```

6. 去掉字符串首尾的空格

```
str1.trim()
```

7. 大小写转换

```
System.out.print("AbCd".toUpperCase());//ABCD
System.out.print("AbCd".toLowerCase());//abcd
```

8. 判断字符串是否以某个字符开头或结尾

```
System.out.println("AbCd".startsWith("A"));//true
System.out.println("AbCd".endsWith("A"));//false
System.out.println("AbCd".startsWith("C",2));//true//指定开始位置
```

9. 字符串替换

```
System.out.println("AbCdA".replace('A','8'));//8bCd8
System.out.println("AbCd".replace("A","123456"));//123456bCd
```

禁止商用 公众号：yes的练级攻略

```
System.out.println("AbCA".replaceFirst("A","123456"));//123456bCA
System.out.println("AbCA".replaceAll("A","123456"));//123456bC123456
```

10. 获取字符串的索引

public int indexOf(int ch): 返回指定字符在字符串中第一次出现处的索引，如果此字符串中没有这样的字符，则返回 -1。

public int indexOf(int ch, int fromIndex): 返回从 fromIndex 位置开始查找指定字符在字符串中第一次出现处的索引，如果此字符串中没有这样的字符，则返回 -1。

int indexOf(String str): 返回指定字符在字符串中第一次出现处的索引，如果此字符串中没有这样的字符，则返回 -1。

int indexOf(String str, int fromIndex): 返回从 fromIndex 位置开始查找指定字符在字符串中第一次出现处的索引，如果此字符串中没有这样的字符，则返回 -1。

```
System.out.println("A123A456A789".indexOf('A'));//0
System.out.println("A123A456A789".indexOf('A',1));//4
System.out.println("A123A456A789".indexOf("A"));//0
System.out.println("A123A456A789".indexOf("A",1));//4

System.out.println("A123A456A789".lastIndexOf('A'));//8
System.out.println("A123A456A789".lastIndexOf('A',8));//8 //从指定索引反向搜索
System.out.println("A123A456A789".lastIndexOf("A"));//8
System.out.println("A123A456A789".lastIndexOf("A",8));//8
System.out.println("A123A456A789".lastIndexOf("100"));//-1   //查找失败
```

- StringBuffer 类和 StringBuilder 类中常见的函数

```
StringBuffer sb = new StringBuffer("东小东");
```

1. 字符串拼接

```
sb.append("123");
System.out.println(sb);//东小东 123
```

2. 删除指定位置的字符

sb. deleteCharAt(1); //删除字符串对象 sb 中索引值为 1 的字符.然后将剩余的内容形成新的字符串

sb. delete (1,4); //删除索引值 1(包括)到索引值 4(不包括)之间的所有字符，剩余的字符形成新的字符串。

3. 插入内容

sb.insert(4,fa); //该示例代码的作用是在对象 sb 的索引值 4 的位置插入 fa 值，形成新的字符串

4. 字符串反转

```
sb.reverse();
```

5. 修改对象中索引值为 indext 位置的字符

```
StringBuffer sb = new StringBuffer(“abc”);
sb.setCharAt(1,’D’);
```

则对象 sb 的值将变成”aDc”。

6. 将对象中的存储空间缩小到和字符串长度一样的长度

```
sb.trimToSize(); //将 StringBuffer 对象的中存储空间缩小到和字符串长度一样的长度.
```

7. 设置字符串长度

禁止商用 公众号：yes的练级攻略

StringBuffer sb=new StringBuffer();

sb.setlength(100);

如果用小于当前字符串长度的值调用 setlength()方法，则新长度后面的字符将丢失。

8. 获得字符串的子串

getChars(int start,int end,char chars[],int charStart);

StringBuffer sb = new StringBuffer("I love You");

int begin = 0;

int end = 5;

//注意 ch 字符数组的长度一定要大于等于 begin 到 end 之间字符的长度

//小于的话会报 ArrayIndexOutOfBoundsException

//如果大于的话，大于的字符会以空格补齐

char[] ch = new char[end-begin];

sb.getChars(begin, end, ch, 0);

System.out.println(ch); //结果：I lov

9. 字符串替换

sb.replace(1, 3, "778899");

//原来：小 dongxiaodong 东 123，现在：小 778899ngxiaodong 东 123

1、自动校对程序（lintcode1531）

1531. 自动校对程序 中文

有一些单词中有拼写错误，请编写一个程序，能够自动校对单词的拼写，并且返回正确值。
校对的规则如下：

1. 如果有三个相同的字符连在一起，一定是拼写错误，去掉其中一个，如：$ooops -> oops$。
2. 如果有两对一样的字符(AABB的形式)连在一起，一定是拼写错误，去掉第二对中的一个字符，如：$helloo -> hello$。
3. 以上两条规则要优先从左往右处理，如：$aabbcc$ 中 $aabb$ 和 $bbcc$ 都是拼写错误，应该优先考虑修复$aabb$，结果是$aabcc$。

样例
样例输入 1:

```
str = "helloo"
```

样例输出 1:

```
"hello"
```

"lloo" 拼写错误，去掉一个'o'。

样例输入 2:

```
str = "woooow"
```

样例输出 2:

```
"woow"
```

"oooo"拼写错误，先删去一个'o'，"ooo"还是拼写错误，再删去一个'o'。
注意事项
输入字符串的长度为 n，$1 <= n <= 10^5$。
字符串均由小写字符组成。

```
public class Solution {
    /**
     * @param str: The string before proofreading.
     * @return: Return the string after proofreading.
     */
    public String automaticProofreadingProgram(String str) {
        // Write your code here.
```

```
        if (str == null || str.length() < 3) return str;
        List<Character> res = new ArrayList<Character>();
        for (int i = 0; i < str.length(); i++) {
            res.add(str.charAt(i));
            int n = res.size();
            if (n < 3) {
                continue;
            }
            if (res.get(n-2) == res.get(n-1)){
                //规则 1
                if (n >= 3 && res.get(n-3) == res.get(n-2)) {
                        res.remove(n-1);
                        continue;
                }
                n = res.size(); //更新 n
                //规则 2
                if (n >= 4 && res.get(n-3) == res.get(n-4)) {
                    res.remove(n-1);
                }
            }
        }
        StringBuilder resStr = new StringBuilder();
        for (int i = 0; i < res.size(); i++) {
            resStr.append(res.get(i));
        }
        return resStr.toString();
    }
}
```

2、字符串的排序（567）

禁止商用
公众号：
yes的练级攻略

567. 字符串的排列

难度 中等 158 收藏 分享 切换为英文 关注 反馈

给定两个字符串 **s1** 和 **s2**，写一个函数来判断 **s2** 是否包含 **s1** 的排列。

换句话说，第一个字符串的排列之一是第二个字符串的子串。

示例1:

输入: s1 = "ab" s2 = "eidbaooo"
输出: True
解释: s2 包含 s1 的排列之一 ("ba").

示例2:

输入: s1= "ab" s2 = "eidboaoo"
输出: False

注意:

1. 输入的字符串只包含小写字母
2. 两个字符串的长度都在 [1, 10,000] 之间

https://leetcode-cn.com/problems/permutation-in-string/solution/567-zi-fu-chuan-de-pai-lie-java-hua-dong-chuang-ko/

利用滑窗机制，确定开始位置和结束位置，之间为窗口。先使用数组对 s1 中出现的字符数进行统计，再使用滑窗遍历 s2 数组，确定是否符合题目要求。具体思路如代码所示。

```
class Solution {
    public boolean checkInclusion(String s1, String s2) {
        int len1 = s1.length(), len2 = s2.length();
        if (len1 == 0) return true;
        if (len2 < len1) return false;

        char[] ss1 = s1.toCharArray(), ss2 = s2.toCharArray();

        // 对 s1 中的字符计数，没出现过的字符为-1
        int[] count = new int[26];
        Arrays.fill(count, -1);
        for (char c : ss1) {
            if (count[c - 'a'] == -1)
                count[c - 'a'] = 1;
            else
                count[c - 'a'] ++;
        }

        // 使用滑窗进行遍历
```

```
            int start = 0, end = 0;
            while (start <= len2 - len1) {      // while 1
                while (end < start + len1) {    // while 2

                    // 如果对应位置为0，则移动start 寻找使其不为0 的点作为新起点
                    if (count[ss2[end] - 'a'] == 0) {
                        while (count[ss2[end] - 'a'] == 0)
                            count[ss2[start ++] - 'a'] ++;
                        break;
                    }

                    // 如果对应位置为-1，则移动start 至当前end 的下一个位置，并令
end=start
                    if (count[ss2[end] - 'a'] == -1) {
                        while (start < end)
                            count[ss2[start ++] - 'a'] ++;
                        start ++;
                        end = start;
                        break;
                    }

                    // 没有以上两种情况，则对应位置计数-1
                    count[ss2[end ++] - 'a'] --;
                }

                // 如果end==start+len1，则说明while2 遍历完成，未被break，因此
返回true
                if (end == start + len1)
                    return true;
            }
            return false;
        }
    }
```

如果是字典序排序，在快排中

3、自定义字符串排序（791）【桶排序】

更多硬核内容就在公众号「yes 的练级攻略」

791. 自定义字符串排序

难度 中等 50 收藏 分享 切换为英文 关注 反馈

字符串 S 和 T 只包含小写字符。在 S 中，所有字符只会出现一次。

S 已经根据某种规则进行了排序。我们要根据 S 中的字符顺序对 T 进行排序。更具体地说，如果 S 中 x 在 y 之前出现，那么返回的字符串中 x 也应出现在 y 之前。

返回任意一种符合条件的字符串 T。

示例:
输入:
S = "cba"
T = "abcd"
输出: "cbad"
解释:
S中出现了字符 "a", "b", "c", 所以 "a", "b", "c" 的顺序应该是 "c", "b", "a".
由于 "d" 没有在S中出现，它可以放在T的任意位置. "dcba", "cdba", "cbda" 都是合法的输出。

注意:

- S 的最大长度为 26，其中没有重复的字符。
- T 的最大长度为 200。
- S 和 T 只包含小写字符。

桶排序思想

桶排序，是一种基于非比较的排序方式，时间复杂度O(n)，因此它是是属于一种“线性排序”。

思想：桶排序的思想是将一组数据分到几个有序的桶里，每个桶里的数据再单独进行快速排序。每个桶内都排序完成后，再加上本身桶之间已经是有序的，那么按照桶的顺序依次取出每个桶内的数据，最终组成的序列就是有序的。

桶排序的时间复杂度分析：

假如排序数组的元素个数为n，均匀分布在个数为m的桶中，那么每个桶中的元素个数为k=n/m,因为每个桶快速排序的时间复杂度为klogk即n/mlog(n/m),那么m个桶一起就是nlog(n/m),假如桶的个数m跟元素个数n十分接近，那么最终的时间复杂度为O(n).

如果极端情况下，所有的元素n个都分在一个桶里呢？这种情况下时间复杂度就退化成O(nlogn)了。

所以，可以看出桶排序对数据及其分布的要求是十分苛刻的：

https://leetcode-cn.com/problems/custom-sort-string/solution/zi-ding-yi-zi-fu-chuan-pai-xu-by-leetcode/

方法一：统计字符数量重新构造字符串【通过】

思路

首先找出在 T 中出现的所有的 S 的元素，并且将这些元素按照 S 中出现的相对顺序排序，然后把 T 中出现的但不在 S 中的元素添加到排好序的字符串中，就得到了我们想要的结果。

在将 T 中出现的但不在 S 中的元素添加到字符串时，无序关注顺序，因为这些元素并没有在 S 中出现，不需要满足排序关系。

算法

一种巧妙的实现方法是统计 T 中每个字符出现的次数，把结果存储在数组 count 中，count[char] 表示字符 char 出现的次数。然后把在 S 中出现的字符按照在 S 中的相对顺序排列，剩余字符添加到当前字符串的后面，最终排好序的字符串顺序为 S + (未在 S 中出现的字符)。

复杂度分析

- 时间复杂度：$O(S.\text{length} + T.\text{length})$，遍历 S 和 T 花费的时间。
- 空间复杂度：$O(T.\text{length})$，统计 26 个小写字母的空间和存储最终排好序的字符串 T 的空间。

```
class Solution {
    public String customSortString(String S, String T) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        int[] count = new int[26];
        for (char c: T.toCharArray()){
            count[c - 'a']++;
        }
        StringBuilder ans = new StringBuilder();
        for (char c: S.toCharArray()) {
            for (int i = 0; i < count[c - 'a']; ++i){
                ans.append(c);
            }
            count[c - 'a'] = 0;
        }
        for (char c = 'a'; c <= 'z'; ++c){
            for (int i = 0; i < count[c - 'a']; ++i){
                ans.append(c);
            }
        }
        return ans.toString();
    }
}
```

4、验证外星语词典（953）

953. 验证外星语词典

难度 简单 46 收藏 分享 切换为英文 关注 反馈

某种外星语也使用英文小写字母，但可能顺序 `order` 不同。字母表的顺序（`order`）是一些小写字母的排列。

给定一组用外星语书写的单词 `words`，以及其字母表的顺序 `order`，只有当给定的单词在这种外星语中按字典序排列时，返回 `true`；否则，返回 `false`。

示例 1：

输入： words = ["hello","leetcode"], order = "hlabcdefgijkmnopqrstuvwxyz"
输出： true
解释： 在该语言的字母表中，'h' 位于 'l' 之前，所以单词序列是按字典序排列的。

示例 2：

输入： words = ["word","world","row"], order = "worldabcefghijkmnpqstuvxyz"
输出： false
解释： 在该语言的字母表中，'d' 位于 'l' 之后，那么 words[0] > words[1]，因此单词序列不是按字典序排列的。

示例 3：

```
输入：words = ["apple","app"], order = "abcdefghijklmnopqrstuvwxyz"
输出：false
解释：当前三个字符 "app" 匹配时，第二个字符串相对短一些，然后根据词典编纂规则 "apple" > "app"，因为 'l' > '∅'，其中 '∅' 是空白字符，定义为比任何其他字符都小（更多信息）。
```

提示：

1. `1 <= words.length <= 100`
2. `1 <= words[i].length <= 20`
3. `order.length == 26`
4. 在 `words[i]` 和 `order` 中的所有字符都是英文小写字母。

https://leetcode-cn.com/problems/verifying-an-alien-dictionary/solution/yan-zheng-wai-xing-yu-ci-dian-by-leetcode/

方法一： 检查相邻单词

思路

只有每对相邻单词都是有序的，那么整个 `words` 才是有序的。因为有序性是可以传递的，例如，`a <= b` 和 `b <= c` 可以推出 `a <= c`。

算法

检查相邻单词 `a` 和 `b` 是否满足 `a <= b`。

为了检查相邻单词 `a`，`b` 是否满足 `a <= b`，只需要检查它们第一个不同的字母就可以了。例如，对于 `"applying"` 和 `"apples"`，第一个不同的字母是 `y` 和 `e`。之后只需要比较这两个字母在 `order` 中的下标就可以了。

还需要考虑两个单词长度不等的情况。例如，当比较 `"app"` 和 `"apply"` 的时候，前三个字母都是相等的，但 `"app"` 比 `"apply"` 更短，所以满足 `a <= b`。

复杂度分析

- 时间复杂度：$O(C)$，其中 C 是 `words` 中单词总长度和。
- 空间复杂度：$O(1)$。

```
class Solution {
    public boolean isAlienSorted(String[] words, String order) {
        int[] index = new int[26];
        for (int i = 0; i < order.length(); ++i)
            index[order.charAt(i) - 'a'] = i;
        search: for (int i = 0; i < words.length - 1; ++i) {
            String word1 = words[i];
            String word2 = words[i+1];
            // Find the first difference word1[k] != word2[k].
            for (int k = 0; k < Math.min(word1.length(), word2.length()
); ++k) {
                if (word1.charAt(k) != word2.charAt(k)) {
                    // If they compare badly, it's not sorted.
```

```
                    if (index[word1.charAt(k) - 'a'] > index[word2.char
At(k) - 'a'])
                        return false;
                    continue search;
                }
            }
            // If we didn't find a first difference, the
            // words are like ("app", "apple").
            if (word1.length() > word2.length())
                return false;
        }
        return true;
    }
}
```

5、根据字符出现频率排序（451）

451. 根据字符出现频率排序

难度 中等 153 收藏 分享 切换为英文 关注 反馈

给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

示例 1:

输入:
"tree"

输出:
"eert"

解释:
'e'出现两次，'r'和't'都只出现一次。
因此'e'必须出现在'r'和't'之前。此外，"eetr"也是一个有效的答案。

示例 2:

```
输入:
"cccaaa"

输出:
"cccaaa"

解释:
'c'和'a'都出现三次。此外，"aaaccc"也是有效的答案。
注意"cacaca"是不正确的，因为相同的字母必须放在一起。
```

示例 3:

```
输入:
"Aabb"

输出:
"bbAa"

解释:
此外，"bbaA"也是一个有效的答案，但"Aabb"是不正确的。
注意'A'和'a'被认为是两种不同的字符。
```

https://leetcode-cn.com/problems/sort-characters-by-frequency/solution/java-heap-by-chris_wing/

方法 1：堆排序（o(nlogn)）

```
class Solution {
    public String frequencySort(String s) {
        int[] letters = new int[128];
        for (char c : s.toCharArray()) {
            letters[c]++;
        }
        PriorityQueue<Character> heap = new PriorityQueue<>(128, (a, b)
 -> Integer.compare(letters[b], letters[a]));
        StringBuilder res = new StringBuilder();

        for (int i = 0; i < letters.length; ++i) {
            if (letters[i] != 0) heap.offer((char)i);
        }

        while (!heap.isEmpty()) {
            char c = heap.poll();
            while (letters[c]-- > 0)
                res.append(c);
        }
        return res.toString();
```

```
    }
}
```

https://leetcode-cn.com/problems/sort-characters-by-frequency/solution/fu-za-du-wei-onde-tong-pai-xu-de-javashi-xian-by-l/

方法 2：桶排序（o(n)）

```
class Solution {
    public String frequencySort(String s) {
        char[] chs = s.toCharArray();
        Map<Character, Integer> map = new HashMap<>();
        int maxTimes = -1;
        //统计每个字母的频次，并存入哈希表
        for(char c : chs){
            if(!map.containsKey(c)){
                map.put(c, 1);
            }else{
                map.put(c, map.get(c) + 1);
            }
            maxTimes = map.get(c) > maxTimes ? map.get(c) : maxTimes;
        }
        //新建一个桶，将字母存入索引为它的频次的桶里
        ArrayList<Character>[] buckets = new ArrayList[maxTimes + 1];
        for(char c : map.keySet()){
            int frequency = map.get(c);
            if(buckets[frequency] == null){
                buckets[frequency] = new ArrayList<>();
            }
            buckets[frequency].add(c);
        }
        //倒着遍历桶，将桶里的字母取出来，并按照它的频次插入字符数组中
        int p = 0;
        for(int i = maxTimes; i >= 0; i--){
            if(buckets[i] != null){
                for(char c : buckets[i]){
                    //buckets[i]这个桶里的字母的频次为 i，因此要插入 i 个到结果集中
                    for(int j = 0; j < i; j++){
                        //复用 chs 作为结果集
                        chs[p++] = c;
                    }
                }
            }
        }
        return new String(chs);
    }
```

}

6、单词规律（290）

290. 单词规律

难度 简单 160 收藏 分享 切换为英文 关注 反馈

给定一种规律 `pattern` 和一个字符串 `str` ，判断 `str` 是否遵循相同的规律。

这里的 **遵循** 指完全匹配，例如， `pattern` 里的每个字母和字符串 `str` 中的每个非空单词之间存在着双向连接的对应规律。

示例1:

```
输入: pattern = "abba", str = "dog cat cat dog"
输出: true
```

示例 2:

```
输入:pattern = "abba", str = "dog cat cat fish"
输出: false
```

示例 3:

```
输入: pattern = "aaaa", str = "dog cat cat dog"
输出: false
```

示例 4:

```
输入: pattern = "abba", str = "dog dog dog dog"
输出: false
```

说明:
你可以假设 `pattern` 只包含小写字母， `str` 包含了由单个空格分隔的小写字母。

https://leetcode-cn.com/problems/word-pattern/solution/split-hashmap-by-huowuyan/

这个解法主要是找字符和字符串的映射关系：

1. 先以split分隔str得到字符串数组s
2. 如果pattern中的字符个数和数组s.length不相等，说明无法一一对应，返回false
3. 当pattern.charAt(i)还没有对应的字符串时，判断s[i]是否已经与其他字符构建了映射关系，是则返回false,否则与s[i]构建映射
4. 当pattern.charAt(i)有对应的字符串时，判断是否与s[i]相等，不相等则返回false
5. 遍历完pattern，说明一一对应，返回true

方法 1：hashmap

```
class Solution {
    public boolean wordPattern(String pattern, String str) {
        if (pattern == null || pattern.length() == 0 || str == null ||
str.length() == 0) {
            return false;
        }
        HashMap <Character, String> map = new HashMap<>();
        String[] strArr = str.split(" ");
        char[] patArr = pattern.toCharArray();
        if (patArr.length != strArr.length) {
            return false;
        }
```

```
        for ( int i = 0; i < strArr.length; i++) {
            if (map.containsKey(patArr[i])) {
                String v = map.get(patArr[i]);
                if (!v.equals(strArr[i])){
                    return false;
                }
            } else {
                if (map.containsValue(strArr[i])) {
                    return false;
                }
                map.put(patArr[i], strArr[i]);
            }
        }
        return true;
    }
}
```

方法 2：双指针

```
class Solution {
    public boolean wordPattern(String pattern, String str) {
        if (pattern == null || pattern.length() == 0 || str == null ||
str.length() == 0) {
            return false;
        }
        String[] strArr = str.split(" ");
        char[] patArr = pattern.toCharArray();
        if (strArr.length != patArr.length) return false;
        for (int i = 0; i < patArr.length; i++){
            for (int j = i + 1; j < patArr.length; j++){
                if ((patArr[i] == patArr[j]) && !(strArr[i].equals(strA
rr[j])) || (patArr[i] != patArr[j]) && (strArr[i].equals(strArr[j]))) r
eturn false;
            }
        }
        return true;
    }
}
```

7、单词规律 2（291）

更多硬核内容就在公众号「yes 的练级攻略」

291. 单词规律 II

难度 困难 24 收藏 分享 切换为英文 关注 反馈

给你一种规律 pattern 和一个字符串 str，请你判断 str 是否遵循其相同的规律。

这里我们指的是 **完全遵循**，例如 pattern 里的每个字母和字符串 str 中每个 **非空** 单词之间，存在着双向连接的对应规律。

示例1:

```
输入: pattern = "abab", str = "redblueredblue"
输出: true
```

示例2:

```
输入: pattern = "aaaa", str = "asdasdasdasd"
输出: true
```

示例2:

```
输入: pattern = "aabb", str = "xyzabcxzyabc"
输出: false
```

提示:

- 你可以默认 pattern 和 str 都只会包含小写字母。

https://leetcode-cn.com/problems/word-pattern-ii/solution/291-dan-ci-gui-lu-ii-by-klb/

```
class Solution {
    Map<Character, String> map = new HashMap<Character, String>();
    public boolean wordPatternMatch(String pattern, String str) {
        //边界条件，如果 pattern 读完了字符串也正好读完就 true，如果字符串没读完就 false
        if (pattern.length() == 0) {
            return (str.length() == 0) ? true : false;
        }
        char letter = pattern.charAt(0);
        for (int i = 1; i <= str.length() - pattern.length() + 1; i++) {
            //从 1 位开始尝试是否有映射，由于每个 pattern 至少得对应一个字符，所以如果字符串剩下的字符少于 pattern 剩下的字符数就可以停止循环了
            String substr = str.substring(0, i), mapStr = map.get(letter);//mapStr 是 letter 的映射，有则返回映射，没有则等于 null
```

```
                if ((mapStr != null && substr.equals(mapStr)) || (mapStr ==
 null && !map.containsValue(substr))) {//这个 pattern 有映射，并且等于这段字
符；或者这段字符不是 pattern 的映射并且没有其他映射，就可以假设这个映射成立并继
续尝试匹配剩下的字符
                    map.put(letter, substr);//不管是否是正确答案，先放进 map 里
面尝试
                    if (wordPatternMatch(pattern.substring(1), str.substrin
g(i))) return true;//如果正好对了就返回 true
                    else if (mapStr == null) map.remove(letter);//如果不对那
就把这个映射取消继续下一个循环进行尝试
                }
            }
            return false;//循环跑完都没有结果当然就返回 false 啦
        }
}
```

8、多次搜索（面试题 17.17）

面试题 17.17. 多次搜索

难度 中等 12 收藏 分享 切换为英文 关注 反馈

给定一个较长字符串 big 和一个包含较短字符串的数组 smalls，设计一个方法，根据 smalls 中的每一个较短字符串，对 big 进行搜索。输出 smalls 中的字符串在 big 里出现的所有位置 positions，其中 positions[i] 为 smalls[i] 出现的所有位置。

示例：

输入：
big = "mississippi"
smalls = ["is","ppi","hi","sis","i","ssippi"]
输出：[[1,4],[8],[],[3],[1,4,7,10],[5]]

提示：

- 0 <= len(big) <= 1000
- 0 <= len(smalls[i]) <= 1000
- smalls 的总字符数不会超过 100000。
- 你可以认为 smalls 中没有重复字符串。
- 所有出现的字符均为英文小写字母。

https://leetcode-cn.com/problems/multi-search-lcci/solution/zi-dian-shu-xiang-xi-zhu-shi-by-zachary443/

```
class Solution {
    //设计字典树数据结构
    class Trie {
        Trie[] levels;
        int index;
        boolean isEnd;

        Trie() {
            levels = new Trie[26];
            index = -1;
            isEnd = false;
```

```
    }
}
//字典树的根节点
Trie root = new Trie();
public int[][] multiSearch(String big, String[] smalls) {
    if (big.length() == 0) {
        return new int[smalls.length][0];
    }
    //利用 smalls 字符串数组构建字典树
    for (int i = 0; i < smalls.length; i++) {
        Trie cur = root;
        char[] chars = smalls[i].toCharArray();
        for (int j = 0; j < chars.length; j++) {
            if (cur.levels[chars[j] - 'a'] == null) {
                cur.levels[chars[j] - 'a'] = new Trie();
            }
            cur = cur.levels[chars[j] - 'a'];
        }
        cur.isEnd = true;//设置当前这个字符串的结束标志为 true
        cur.index = i;//保存当前这个字符串在 smalls 中的位置
    }
    //用来暂时保存结果
    ArrayList<Integer>[] ans = new ArrayList[smalls.length];
    //初始化
    for (int i = 0; i < ans.length; i++) {
        ans[i] = new ArrayList<>();
    }
    //在 big 字符串中进行查询
    char[] bigChars = big.toCharArray();
    for (int i = 0; i < bigChars.length; i++) {
        Trie cur = root;
        for (int j = i; j < bigChars.length; j++) {
            if (cur.levels[bigChars[j] - 'a'] == null) {
                break;
            }
            cur = cur.levels[bigChars[j] - 'a'];
            //如果标志位为 true，说明这构成一个 smalls 中的字符串
            //根据该字符串在 smalls 中的位置,找到它在 ans 中的位置,然后添加它在 big
字符串中的起始位置
            if (cur.isEnd) {
                ans[cur.index].add(i);
            }
        }
    }
```

```
        //真正的二维数组结果，根据 ans 进行一个简单转化即可
        int[][] res = new int[smalls.length][];
        for (int i = 0; i < smalls.length; i++) {
            res[i] = new int[ans[i].size()];
            for (int j = 0; j < ans[i].size(); j++) {
                res[i][j] = ans[i].get(j);
            }
        }
        //返回结果
        return res;
    }
}
```

9、模式匹配（面试题 16.18）【pattern 只有 a 和 b】

面试题 16.18. 模式匹配

难度 中等　93　收藏　分享　切换为英文　关注　反馈

你有两个字符串，即 pattern 和 value。 pattern 字符串由字母 "a" 和 "b" 组成，用于描述字符串中的模式。例如，字符串 "catcatgocatgo" 匹配模式 "aabab" (其中 "cat" 是 "a"，"go" 是 "b")，该字符串也匹配像 "a"、"ab" 和 "b" 这样的模式。但需注意 "a" 和 "b" 不能同时表示相同的字符串。编写一个方法判断 value 字符串是否匹配 pattern 字符串。

示例 1:

```
输入： pattern = "abba", value = "dogcatcatdog"
输出： true
```

示例 2:

```
输入： pattern = "abba", value = "dogcatcatfish"
输出： false
```

示例 3:

```
输入： pattern = "aaaa", value = "dogcatcatdog"
输出： false
```

示例 4:

```
输入： pattern = "abba", value = "dogdogdogdog"
输出： true
解释： "a"="dogdog",b="", 反之也符合规则
```

提示:

- 1 <= len(pattern) <= 1000
- 0 <= len(value) <= 1000
- 你可以假设 pattern 只包含字母 "a" 和 "b"，value 仅包含小写字母。

https://leetcode-cn.com/problems/pattern-matching-lcci/solution/javashuang-bai-by-black_lii/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路

主要是围绕这个方程 `lv=ca*la+cb*lb` ，判断各种出口情况

- lv:value的长度
- ca:匹配串中(第一个出现的字符视为a)a的个数
- la:a的长度
- cb:匹配串中b的个数
- lb:b的长度

其中只有la和lb是未知的，那么确定其中一个，就可以得出另外一个的值，只需要判断满不满足条件即可。
满足上述条件以外，还需要判断子串的值是否相同，所以内部还有一个匹配串的遍历
具体请看注释

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

我们设 $pattern$ 的长度为 ℓ_p，$value$ 的长度为 ℓ_v。根据题目描述，我们需要给字母 a 和 b 分配不同的字符串值（可以为空字符串），使得将 $pattern$ 中的字母替换成对应的字符串后，结果与 $value$ 相同。

在分配字符串之前，我们不妨先分配 a 和 b 对应字符串的长度。如果确定了长度，那么我们只要将 $value$ 按照 $pattern$ 中出现字母的顺序，划分成 ℓ_p 个子串，并判断其中 a 对应的子串是否相同，以及 b 对应的子串是否相同即可。具体地，假设 $pattern$ 中出现了 c_a 个 a 以及 $\ell_p - c_a$ 个 b，并且 a 和 b 对应字符串的长度分别为 ℓ_a 和 ℓ_b，那么必须要满足：

$$c_a * \ell_a + (\ell_p - c_a) * \ell_b = \ell_v$$

其中 c_a 是已知的常量，ℓ_a 和 ℓ_b 是未知数。这是一个二元一次方程，可能无解、有唯一解或者无数解。然而我们需要的仅仅是**自然数解**，也就是 ℓ_a 和 ℓ_b 都大于等于 0 的解，因此我们可以直接枚举 ℓ_a 的值，它必须是 $[0, \frac{\ell_v}{c_a}]$ 之间的自然数，否则 ℓ_b 就不会大于等于 0 了。在枚举 ℓ_a 之后，我们将其带入等式并解出 ℓ_b。如果 ℓ_b 是整数，我们就枚举了一组 a 和 b 的可能长度。

在枚举了长度之后，我们就可以根据 $pattern$ 来将 $value$ 划分成 ℓ_p 个子串。具体地，我们遍历 $pattern$，并用一个指针 pos 来帮助我们进行划分。当我们遍历到一个 a 时，我们取出从 pos 开始，长度为 ℓ_a 的子串。如果这是我们第一次遇到字母 a，我们就得到了 a 应该对应的子串；否则，我们将取出的子串与 a 应该对应的子串进行比较，如果不相同，说明模式匹配失败。

同理，当我们遍历到一个 b 时，我们取出从 pos 开始，长度为 ℓ_b 的子串，根据是否第一次遇到字母 b 来进行比较。在比较结束后，我们将 pos 向后移动，进行下一个字母的匹配。

在遍历完成之后，如果匹配没有失败，我们还需要判断一下 a 和 b 是否对应了不同的子串。只有它们对应的子串不同时，才是一种满足题目要求的模式匹配。

细节

上面的算法看上去不是很复杂：我们只需要用一重循环枚举 ℓ_a，计算出 ℓ_b，再用一重循环遍历 $pattern$ 以及移动 pos 即可。但就像我们在「前言」部分所说的，本题有非常多的细节需要考虑。

我们回到二元一次方程：

$$c_a * \ell_a + (\ell_p - c_a) * \ell_b = \ell_v$$

如果我们枚举 ℓ_a，那么必须要求 $c_a \neq 0$，因为在 $c_a = 0$ 的情况下，原方程如果有解，那么一定有无数解（因为 ℓ_a 可以取任意值）。因此如果 $c_a = 0$，我们就必须枚举 ℓ_b。这无疑增加了编码的复杂度，因为需要根据 c_a 的值选择对 ℓ_a 或 ℓ_b 进行枚举，失去了统一性。并且，如果 $\ell_p - c_a$ 也为 0，那么我们连 ℓ_b 都无法枚举。

因此，我们必须梳理一下判断的逻辑：

1. 如果 $pattern$ 为空，那么只有在 $value$ 也为空时，它们才能匹配；
2. 如果 $value$ 为空，那么如果 $pattern$ 也为空，就和第一条的情况相同；如果 $pattern$ 中只出现了一种字母，我们可以令该字母为空，另一没有出现的字母为任意非空串，就可以正确匹配；如果 $pattern$ 中出现了两种字母，那么就无法正确匹配，因为这两种字母都必须为空串，而题目描述中规定它们不能表示相同的字符串；
3. 如果 $pattern$ 和 $value$ 均非空，那么我们需要枚举 $pattern$ 中出现的那个字母（如果两个字母均出现，可以枚举任意一个）对应的长度，使用上面提到的算法进行判断。

对于上面的第三条，我们可以根据「对称性」减少代码的编写的复杂度：我们还是固定枚举 ℓ_a，但如果 $c_a < \ell_p - c_a$，即 a 出现的次数少于 b 出现的次数，那么我们就将 $pattern$ 中所有的 a 替换成 b，b 替换成 a。这样做就保证了 a **出现了至少一次（$c_a > 0$），枚举 ℓ_a 就不会有任何问题**，同时不会影响答案的正确性。

这样一来，我们就可以优化判断的逻辑：

1. 我们首先保证 $pattern$ 中 a 出现的次数不少于 b 出现的次数。如果不满足，我们就将 a 和 b 互相替换；
2. 如果 $value$ 为空，那么要求 $pattern$ 也为空（$\ell_p = 0$）或者只出现了字母 a（$\ell_p - c_a = 0$），这两种情况均等同于 $\ell_p - c_a = 0$。在其余情况下，都无法匹配成功；
3. 如果 $pattern$ 为空且 $value$ 不为空，那么无法匹配成功；
4. 如果 $pattern$ 和 $value$ 均非空，我们就可以枚举 ℓ_a 并使用上面提到的算法进行判断。

下面给出的代码遵循了这样的逻辑。读者也可以尝试自己归纳出一套逻辑并编写代码。

```
class Solution {
    public boolean patternMatching(String pattern, String value) {
        //字符串长度为 0 时
        if (value.length() == 0) {
            //匹配串长度不为 0
            if (pattern.length() != 0) {
                for (int i = 1; i < pattern.length(); i++) {
                    //如果匹配串不是只有 a 或者只有 b
                    if (pattern.charAt(i - 1) != pattern.charAt(i))
                        return false;
                }
            }
            //匹配串长度为 0
            return true;
        }
        //匹配串长度为 0，或者 1 时，不需要多余判断
        if (pattern.length() == 0) return false;
        if (pattern.length() == 1) return true;
        //出现在首个位置的，无论是 a 还是 b，都视为 a
        char a = pattern.charAt(0);
        //匹配串中 a，b 的个数
        int ca = 0, cb = 0;
        for (int i = 0; i < pattern.length(); i++) {
            if (pattern.charAt(i) == a) ca++;
            else cb++;
        }
        int lv = value.length(), lp = pattern.length();
        //b 没有出现，此处的真实情况可能为'a'没出现，也可能为'b'没出现，但只会
被视为 b 没出现
        if (cb == 0) {
            //满足 ca*la=lv 即可
            if (lv % ca != 0) return false;
            int la = lv / ca;
            String aa = value.substring(0, la);
            for (int i = aa.length(); i < value.length(); i += aa.lengt
h()) {
                if (!value.substring(i, i + aa.length()).equals(aa)) re
turn false;
            }
            return true;
        }
        //lv=ca*la+cb*lb
        for (int la = 0; la * ca <= value.length(); ++la) {
            //以上方程中 lv,ca,la,cb 均已知，那么就可以计算出 lb
```

```
            int rest = lv - ca * la;
            //la 的值不满足方程，跳过此次判定
            if (rest % cb != 0) continue;
            int lb = rest / cb, ci = 0;
            //匹配标识
            boolean isMatch = true;
            //aa，bb 设定默认值
            String aa = "", bb = "";
            //开始匹配
            for (char c : pattern.toCharArray()) {
                if (c == a) {
                    String cur_a = value.substring(ci, ci += la);
                    //此处理论上有两个入口会进这个 if
                    // 1.初次进入
                    // 2.不是初次进入，但 la 长度为 0
                    if (aa.length() == 0)
                        aa = cur_a;
                        //字符串不相同，退出匹配串遍历
                    else if (!aa.equals(cur_a)) {
                        isMatch = false;
                        break;
                    }
                } else {
                    //同上
                    String cur_b = value.substring(ci, ci += lb);
                    if (bb.length() == 0)
                        bb = cur_b;
                    else if (!bb.equals(cur_b)) {
                        isMatch = false;
                        break;
                    }
                }
            }
            //如果此次已经全部匹配，那么就不用继续枚举，通过匹配
            if (isMatch && !aa.equals(bb)) return true;
        }
        //都不满足，默认出口
        return false;
    }
}
```

10、判断是否互为字符重排（面试题 01.02）【有效的字母异位词（242）】

面试题 01.02. 判定是否互为字符重排

难度 简单 12 收藏 分享 切换为英文 关注 反馈

给定两个字符串 `s1` 和 `s2`，请编写一个程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。

示例 1:

```
输入: s1 = "abc", s2 = "bca"
输出: true
```

示例 2:

```
输入: s1 = "abc", s2 = "bad"
输出: false
```

说明:

- `0 <= len(s1) <= 100`
- `0 <= len(s2) <= 100`

242. 有效的字母异位词

难度 简单 247 收藏 分享 切换为英文 关注 反馈

给定两个字符串 *s* 和 *t*，编写一个函数来判断 *t* 是否是 *s* 的字母异位词。

示例 1:

```
输入: s = "anagram", t = "nagaram"
输出: true
```

示例 2:

```
输入: s = "rat", t = "car"
输出: false
```

说明:
你可以假设字符串只包含小写字母。

进阶:
如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

https://leetcode-cn.com/problems/valid-anagram/solution/you-xiao-de-zi-mu-yi-wei-ci-by-leetcode/

方法 1：排序

```
class Solution {
    public boolean CheckPermutation(String s1, String s2) {
        if (s1.length() != s2.length()) return false;
```

```
            s1 = sortArray(s1);
            s2 = sortArray(s2);
            return s1.equals(s2) ? true : false;
        }
        private String sortArray(String before) {
            char[] tmp = before.toCharArray();
            Arrays.sort(tmp);
            return String.valueOf(tmp);
        }
}
```

https://leetcode-cn.com/problems/check-permutation-lcci/solution/hei-ke-ji-shuang-bai-by-zbw666/

方法 2：错位的字符串中每个字符的和相等，乘积相等

```
class Solution {
    public boolean CheckPermutation(String s1, String s2) {
        if (s1.length() != s2.length()) return false;
        int sum1 = 0, sum2 = 0, mul1 = 1, mul2 = 1;
        for (int i = 0; i < s1.length(); i++){
            sum1 += s1.charAt(i);
            sum2 += s2.charAt(i);
            mul1 *= s1.charAt(i);
            mul2 *= s2.charAt(i);
        }
        return (sum1 == sum2 && mul1 == mul2) ? true : false;
    }
}
```

方法二：哈希表

算法：

1. 为了检查 t 是否是 s 的重新排列，我们可以计算两个字符串中每个字母的出现次数并进行比较。因为 S 和 T 都只包含 $A-Z$ 的字母，所以一个简单的 26 位计数器表就足够了。
2. 我们需要两个计数器数表进行比较吗？实际上不是，因为我们可以用一个计数器表计算 s 字母的频率，用 t 减少计数器表中的每个字母的计数器，然后检查计数器是否回到零。

Java

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
        return false;
    }
    int[] counter = new int[26];
    for (int i = 0; i < s.length(); i++) {
        counter[s.charAt(i) - 'a']++;
        counter[t.charAt(i) - 'a']--;
    }
    for (int count : counter) {
        if (count != 0) {
            return false;
        }
    }
    return true;
}
```

3. 或者我们可以先用计数器表计算 s，然后用 t 减少计数器表中的每个字母的计数器。如果在任何时候计数器低于零，我们知道 t 包含一个不在 s 中的额外字母，并立即返回 FALSE。

Java

```
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
        return false;
    }
    int[] table = new int[26];
    for (int i = 0; i < s.length(); i++) {
        table[s.charAt(i) - 'a']++;
    }
    for (int i = 0; i < t.length(); i++) {
        table[t.charAt(i) - 'a']--;
        if (table[t.charAt(i) - 'a'] < 0) {
            return false;
        }
    }
    return true;
}
```

复杂度分析

- 时间复杂度：$O(n)$。时间复杂度为 $O(n)$ 因为访问计数器表是一个固定的时间操作。
- 空间复杂度：$O(1)$。尽管我们使用了额外的空间，但是空间的复杂性是 $O(1)$，因为无论 N 有多大，表的大小都保持不变。

进阶：
如果输入字符串包含 unicode 字符怎么办？你能否调整你的解法来应对这种情况？

解答：
使用哈希表而不是固定大小的计数器。想象一下，分配一个大的数组来适应整个 Unicode 字符范围，这个范围可能超过 100 万。哈希表是一种更通用的解决方案，可以适应任何字符范围。

方法 3：哈希表（适用于整个 Unicode 字符范围）

```
class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }
        int[] table = new int[26];
        for (int i = 0; i < s.length(); i++) {
            table[s.charAt(i) - 'a']++;
```

```
        }
        for (int i = 0; i < t.length(); i++) {
            table[t.charAt(i) - 'a']--;
            if (table[t.charAt(i) - 'a'] < 0) {
                return false;
            }
        }
        return true;
    }
}
```

11、字母异位词分组（49）【变位词组（面试题 10.02）】

49. 字母异位词分组

难度 中等 400 收藏 分享 切换为英文 关注 反馈

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

```
输入: ["eat", "tea", "tan", "ate", "nat", "bat"]
输出:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

说明:

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

YES的练级攻略

https://leetcode-cn.com/problems/group-anagrams/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by--16/

时间复杂度：虽然看起来外层用了两个 for 循环，但是我们通过 used 数组保证了每个字符串只会访问 1 次，所以外层的复杂度是字符串数组的长度 $O(n)$，判断两个字符串相等的函数 equal 函数，时间复杂度是字符串的最长长度 $O(K)$。所以总共就是 $O(nK)$。

空间复杂度：$O(NK)$，用来存储结果。

解法一算是比较通用的解法，不管字符串里边是大写字母，小写字母，数字，都可以用这个算法解决。这道题的话，题目告诉我们字符串中只有小写字母，针对这个限制，我们可以再用一些针对性强的算法。

下边的算法本质是，我们只要把一类的字符串用某一种方法唯一的映射到同一个位置就可以。

方法 1：通用方法，用到上面的判断两两相等（765ms）

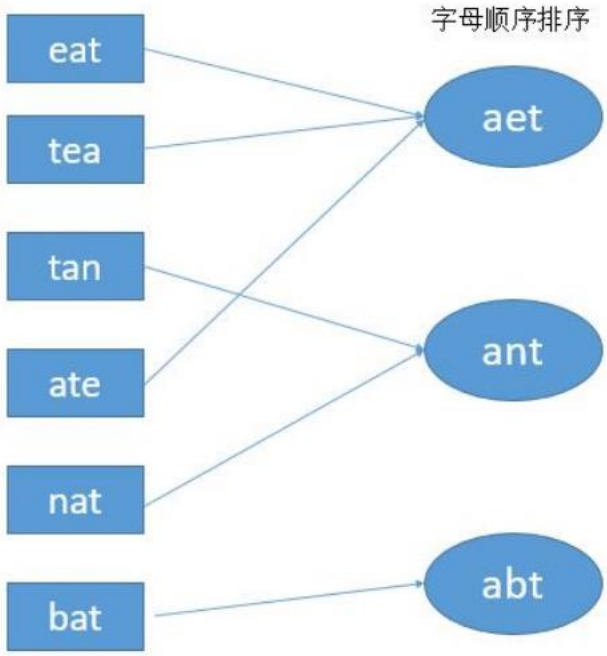
```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> ans = new ArrayList<>();
        boolean[] used = new boolean[strs.length];
        for (int i = 0; i < strs.length; i++) {
            List<String> temp = null;
            if (!used[i]) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                temp = new ArrayList<String>();
                temp.add(strs[i]);
                //两两比较判断字符串是否符合
                for (int j = i + 1; j < strs.length; j++) {
                    if (equals(strs[i], strs[j])) {
                        used[j] = true;
                        temp.add(strs[j]);
                    }
                }
            }
            if (temp != null) {
                ans.add(temp);
            }
        }
        return ans;

    }
    private boolean equals(String s1, String s2) {
        if (s1.length() != s2.length()) return false;
        int sum1 = 0, sum2 = 0, mul1 = 1, mul2 = 1;
        for (int i = 0; i < s1.length(); i++){
            sum1 += s1.charAt(i);
            sum2 += s2.charAt(i);
            mul1 *= s1.charAt(i);
            mul2 *= s2.charAt(i);
        }
        return (sum1 == sum2 && mul1 == mul2) ? true : false;
    }
}
```

解法二

我们将每个字符串按照字母顺序排序，这样的话就可以把 eat，tea，ate 都映射到 aet。其他的类似。



时间复杂度：排序的话算作 $O（Klog（K））$，最外层的 for 循环，所以就是 $O（nKlog（K））$。

空间复杂度：$O（NK）$，用来存储结果。

方法 2：按字母顺序排序（9ms）

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        HashMap<String, List<String>> hash = new HashMap<>();
        for (int i = 0; i < strs.length; i++) {
            char[] s_arr = strs[i].toCharArray();
            //排序
            Arrays.sort(s_arr);
            //映射到 key
            String key = String.valueOf(s_arr);
            //添加到对应的类中
            if (hash.containsKey(key)) {
                hash.get(key).add(strs[i]);
            } else {
                List<String> temp = new ArrayList<String>();
                temp.add(strs[i]);
                hash.put(key, temp);
            }
```

```
        }
        return new ArrayList<List<String>>(hash.values());
    }
}
```

12、找到字符串中所有字母异位词（438）

438. 找到字符串中所有字母异位词

难度 中等 325 收藏 分享 切换为英文 关注 反馈

给定一个字符串 **s** 和一个非空字符串 **p**，找到 **s** 中所有是 **p** 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 **s** 和 **p** 的长度都不超过 20100。

说明：

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

示例 1:

输入:
s: "cbaebabacd" p: "abc"

输出:
[0, 6]

解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。

示例 2:

输入:
s: "abab" p: "ab"

输出:
[0, 1, 2]

解释:
起始索引等于 0 的子串是 "ab", 它是 "ab" 的字母异位词。
起始索引等于 1 的子串是 "ba", 它是 "ab" 的字母异位词。
起始索引等于 2 的子串是 "ab", 它是 "ab" 的字母异位词。

路

https://leetcode-cn.com/problems/find-all-anagrams-in-a-string/solution/hua-dong-chuang-kou-tong-yong-si-xiang-jie-jue-zi-/

关于双指针的快慢指针和左右指针的用法，可以参见前文 双指针技巧汇总，本文就解决一类最难掌握的双指针技巧：滑动窗口技巧。总结出一套框架，可以保你闭着眼睛都能写出正确的解法。

说起滑动窗口算法，很多读者都会头疼。这个算法技巧的思路非常简单，就是维护一个窗口，不断滑动，然后更新答案么。LeetCode 上有起码 10 道运用滑动窗口算法的题目，难度都是中等和困难。该算法的大致逻辑如下：

```
int left = 0, right = 0;

while (right < s.size()) {
    // 增大窗口
    window.add(s[right]);
    right++;

    while (window needs shrink) {
        // 缩小窗口
        window.remove(s[left]);
        left++;
    }
}
```

这个算法技巧的时间复杂度是 O(N)，比字符串暴力算法要高效得多。

更多硬核内容就在公众号「yes 的练级攻略」

其实困扰大家的，不是算法的思路，而是各种细节问题。比如说如何向窗口中添加新元素，如何缩小窗口，在窗口滑动的哪个阶段更新结果。即便你明白了这些细节，也容易出 bug，找 bug 还不知道怎么找，真的挺让人心烦的。

所以今天我就写一套滑动窗口算法的代码框架，我连再哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，还不会出 bug：

```
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

其中两处 `...` 表示的更新窗口数据的地方，到时候你直接往里面填就行了。

而且，这两个 `...` 处的操作分别是右移和左移窗口更新操作，等会你会发现它们操作是完全对称的。

说句题外话，我发现很多人喜欢执着于表象，不喜欢探求问题的本质。比如说有很多人评论我这个框架，说什么散列表速度慢，不如用数组代替散列表；还有很多人喜欢把代码写得特别短小，说我这样代码太多余，影响编译速度，LeetCode 上速度不够快。

我服了。算法看的是时间复杂度，你能确保自己的时间复杂度最优，就行了。至于 LeetCode 所谓的运行速度，那个都是玄学，只要不是慢的离谱就没啥问题，根本不值得你从编译层面优化，不要舍本逐末……

labuladong 公众号的重点在于算法思想，你把框架思维了然于心，然后随你魔改代码好吧，你高兴就好。

言归正传，下面就直接上**四道** LeetCode 原题来套这个框架，其中第一道题会详细说明其原理，后面四道就直接闭眼睛秒杀了。

本文代码为 C++ 实现，不会用到什么编程方面的奇技淫巧，但是还是简单介绍一下一些用到的数据结构，以免有的读者因为语言的细节问题阻碍对算法思想的理解：

`unordered_map` 就是哈希表（字典），它的一个方法 `count(key)` 相当于 Java 的 `containsKey(key)` 可以判断键 key 是否存在。

可以使用方括号访问键对应的值 `map[key]`。需要注意的是，如果该 `key` 不存在，C++ 会自动创建这个 key，并把 `map[key]` 赋值为 0。

所以代码中多次出现的 `map[key]++` 相当于 Java 的 `map.put(key, map.getOrDefault(key, 0) + 1)`。

这是 LeetCode 第 438 题，Find All Anagrams in a String，难度 Medium：

给定一个字符串 **s** 和一个非空字符串 **p**，找到 **s** 中所有是 **p** 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 **s** 和 **p** 的长度都不超过 20100。

说明：

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

示例 1:

输入:
`s: "cbaebabacd" p: "abc"`

输出:
`[0, 6]`

解释:
起始索引等于 0 的子串是 "cba"，它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac"，它是 "abc" 的字母异位词。

呵呵，这个所谓的字母异位词，不就是排列吗，搞个高端的说法就能糊弄人了吗？**相当于，输入一个串 `S`，一个串 `T`，找到 `S` 中所有 `T` 的排列，返回它们的起始索引。**

跟寻找字符串的排列一样，只是找到一个合法异位词（排列）之后将起始索引加入 `res` 即可。

438 具体代码

```
class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        if(s == null || s.length() == 0) return new ArrayList<>();
        List<Integer> res = new ArrayList<>();
        int[] needs = new int[26]; //由于都是小写字母，因此直接用 26 个长度的数组代替原来的 HashMap
        int[] window = new int[26];
        int left = 0, right = 0, total = p.length(); //用 total 检测窗口中是否已经涵盖了 p 中的字符
        for(char ch : p.toCharArray()){
            needs[ch - 'a'] ++;
        }
        while(right < s.length()){
            char chr = s.charAt(right);
            if(needs[chr - 'a'] > 0){
                window[chr - 'a'] ++;
                if(window[chr - 'a'] <= needs[chr - 'a']){
                    total --;
                }
            }
            while(total == 0){
                if(right-left+1 == p.length()){
                    res.add(left);
                }
                char chl = s.charAt(left);
                if(needs[chl - 'a'] > 0){
                    window[chl - 'a'] --;
                    if(window[chl - 'a'] < needs[chl - 'a']){
```

```
                    total ++;
                }
            }
            left ++;
        }
        right ++;
    }
    return res;
  }
}
```

13、字符串压缩（面试题 01.06）

面试题 01.06. 字符串压缩

难度 简单 45 收藏 分享 切换为英文 关注 反馈

字符串压缩。利用字符重复出现的次数，编写一种方法，实现基本的字符串压缩功能。比如，字符串 `aabcccccaaa` 会变为 `a2b1c5a3` 。若“压缩”后的字符串没有变短，则返回原先的字符串。你可以假设字符串中只包含大小写英文字母（a至z）。

示例1:

```
输入："aabcccccaaa"
输出："a2b1c5a3"
```

示例2:

```
输入："abbccd"
输出："abbccd"
解释："abbccd"压缩后为"a1b2c2d1"，比原字符串长度更长。
```

提示:

1. 字符串长度在[0, 50000]范围内。

https://leetcode-cn.com/problems/compress-string-lcci/solution/java-bian-li-1bian-1yan-jiu-ming-bai-by-sweetiee/

```
class Solution {
    public String compressString(String S) {
        if (S == null || S.length() < 3) {
            return S;
        }
        char[] s = S.toCharArray();
        StringBuilder sb = new StringBuilder().append(s[0]);
        int cnt = 1;
        for (int i = 1; i < S.length(); i++){
            if (s[i] == s[i-1]){
                cnt++;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            } else {
                sb.append(cnt).append(s[i]);
                cnt = 1;
            }
        }
        return sb.append(cnt).length() < S.length() ? sb.toString(): S;
    }
}
```

14、压缩字符串（443）

443. 压缩字符串

难度 简单 134 收藏 分享 切换为英文 关注 反馈

给定一组字符，使用原地算法将其压缩。

压缩后的长度必须始终小于或等于原数组长度。

数组的每个元素应该是长度为1 的字符（不是 int 整数类型）。

在完成原地修改输入数组后，返回数组的新长度。

进阶：
你能否仅使用O(1) 空间解决问题?

示例 1：

输入：
["a","a","b","b","c","c","c"]

输出：
返回 6 ，输入数组的前 6 个字符应该是：["a","2","b","2","c","3"]

说明：
"aa" 被 "a2" 替代。"bb" 被 "b2" 替代。"ccc" 被 "c3" 替代。

yes的练级攻略

示例 2：

```
输入：
["a"]

输出：
返回 1 ，输入数组的前 1 个字符应该是：["a"]

解释：
没有任何字符串被替代。
```

示例 3：

```
输入：
["a","b","b","b","b","b","b","b","b","b","b","b","b"]

输出：
返回 4 ，输入数组的前4个字符应该是：["a","b","1","2"]。

解释：
由于字符 "a" 不重复，所以不会被压缩。"bbbbbbbbbbbb" 被 “b12” 替代。
注意每个数字在数组中都有它自己的位置。
```

提示：

- 所有字符都有一个ASCII值在[35, 126]区间内。
- 1 <= len(chars) <= 1000。

https://leetcode-cn.com/problems/string-compression/solution/hua-dong-chuang-kou-fa-ya-suo-zi-fu-chuan-java-by-/

算法说明

利用滑动窗口法，左右指针起点都为原数组的首位。
实现步骤如下：

1. 不断右移右指针，使窗口不断增大；
2. 当窗口内出现一个不同的元素时，就可以将元素及其数量（等于左右指针之差）更新在数组中，然后让左指针指向右指针；
3. 遍历到最后一个字符时也需要结算，因此将右指针的终点设为数组之外一格。当右指针移到终点时也要更新数组。

```
class Solution {
    public int compress(char[] chars) {     // 数组大小范
围：  1 <= chars.length <= 1000
        int left = 0;
        int size = 0;

        // 由于最后一个字符也需要判断，所以将右指针终点放到数组之外一格
        for (int right = 0; right <= chars.length; right++) {
            // 当遍历完成，或右指针元素不等于左指针元素时，更新数组
            if (right == chars.length || chars[right] != chars[left]) {
                chars[size++] = chars[left];    // 更新字符
                if (right - left > 1) {         // 更新计数，当个数大于 1 时才更新
                    for(char c : String.valueOf(right - left).toCharArray()) {
                        chars[size++] = c;
```

```
                }
            }
            left = right;
        }
    }

    return size;
  }
}
```

15、编码最短长度的字符串（471）

471. 编码最短长度的字符串

难度 困难 24 收藏 分享 切换为英文 关注 反馈

给定一个 **非空** 字符串，将其编码为具有最短长度的字符串。

编码规则是：`k[encoded_string]`，其中在方括号 *encoded_string* 中的内容重复 *k* 次。

注：

1. *k* 为正整数且编码后的字符串不能为空或有额外的空格。
2. 你可以假定输入的字符串只包含小写的英文字母。字符串长度不超过 160。
3. 如果编码的过程不能使字符串缩短，则不要对其进行编码。如果有多种编码方式，返回任意一种即可。

示例 1：

```
输入： "aaa"
输出： "aaa"
解释： 无法将其编码为更短的字符串，因此不进行编码。
```

示例 2：

```
输入： "aaaaa"
输出： "5[a]"
解释： "5[a]" 比 "aaaaa" 短 1 个字符。
```

示例 3：

```
输入： "aaaaaaaaaa"
输出： "10[a]"
解释： "a9[a]" 或 "9[a]a" 都是合法的编码，和 "10[a]" 一样长度都为 5。
```

示例 4：

```
输入：  "aabcaabcd"
输出：  "2[aabc]d"
解释：  "aabc" 出现两次，因此一种答案可以是 "2[aabc]d"。
```

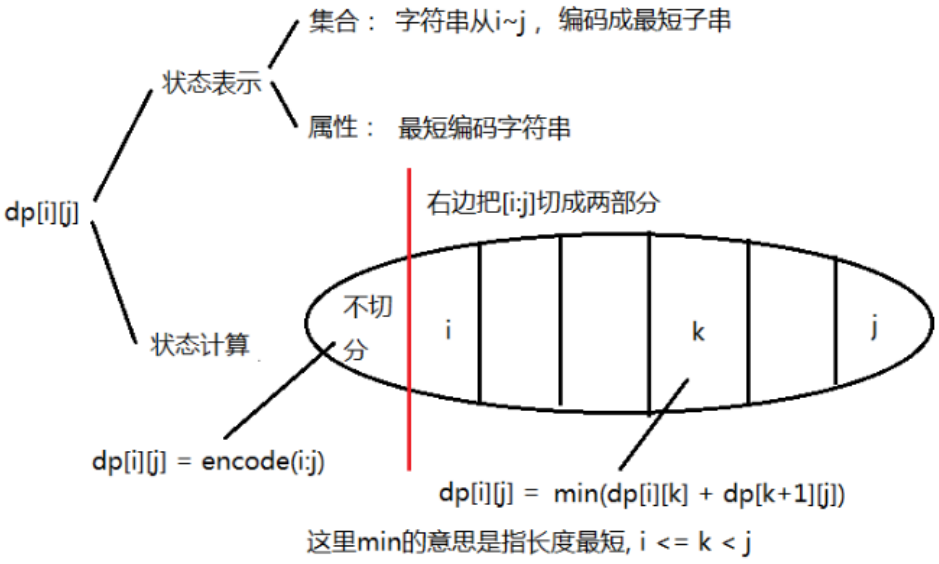
示例 5：

```
输入：  "abbbabbbcabbbabbbc"
输出：  "2[2[abbb]c]"
解释：  "abbbabbbc" 出现两次，但是 "abbbabbbc" 可以编码为 "2[abbb]c"，因此一种答案可以是 "2[2[abbb]c]"。
```

https://leetcode-cn.com/problems/encode-string-with-shortest-length/solution/471-bian-ma-zui-duan-chang-du-de-zi-fu-chuan-qu-ji/

AC本题的关键， 就是如何判断一个字符串能否进行压缩。
可参考简单题459.重复的子字符串。

使用题目459的方法， 能快速找到连续重复子字符串，
找到连续重复的子字符串，我们才能进行编码(压缩)。
题目459的思路大致如下：
另 t = s + s, 从下标1的字符开始查找字符串s， 找到下标p，
如果p < n, 存在连续重复的子字符串ps = s.substring(0, p), 个数为 n / p
否则， 不存在连续重复子字符串， 无法进行编码。




```
class Solution {
    public String encode(String s) {
        char[] cs=s.toCharArray();
        // hp 数组保存的是从 i 到 j 的字符串，从第 i 个位置开始连续重复出现了多少次，最小为 1：
        int[][] hp = new int[cs.length][cs.length];
        // dp 数组保存窗口大小为 w，起点为 j 的子串编码后的最短长度：
        int[][] dp = new int[cs.length+1][cs.length];
```

```
// dp2 数组保存的是窗口大小为w，起点为j 的子串切割的最佳位置，默认为j+w-1：
int[][] dp2 = new int[cs.length+1][cs.length];
// 首先初始化 hp 数组：
for (int i=0; i<cs.length; i++) {
    for (int j=i; j<cs.length; j++) {
        int len=(j-i+1);
        int k=j+1;
        while (cs.length-k>=len) {
            int m=0;
            for (;m<len;m++) {
                if (cs[i+m]!=cs[k+m]) {
                    break;
                }
            }
            if (m==len) {
                k+=len;
            } else {
                break;
            }
        }
        hp[i][j]=1+(k-j-1)/len;
    }
}
// 构建 dp 数组和 dp2 数组：
for (int w=1; w<=cs.length; w++) {
    for (int j=0; j<cs.length-w+1; j++) {
        dp[w][j]=w;
        dp2[w][j]=j+w-1;
        int minLen=w;
        for (int k=j; k<j+w/2; k++) {
            if (w%(k-j+1)==0 && hp[j][k]>=(w/(k-j+1))) {
                int tmp=w/(k-j+1);
                if (2+help(tmp)+dp[k-j+1][j] < w) {
                    dp[w][j]=2+help(tmp)+dp[k-j+1][j];
                    dp2[w][j]=k;
                    break;
                }
            }
        }
        for (int k=j; k<j+w-1; k++) {
            if (dp[w][j] > (dp[k-j+1][j]+dp[w-k+j-1][k+1])) {
                dp[w][j]=dp[k-j+1][j]+dp[w-k+j-1][k+1];
                dp2[w][j]=k;
            }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                }
            }
        }

        // 根据 dp 数组和 dp2 数组构建编码后的字符串：
        StringBuilder sb=new StringBuilder();
        help2(s, 0, cs.length-1, hp, dp, dp2, sb);
        return sb.toString();
    }

    public void help2(String s, int start, int end, int[][] hp, int[][] dp, int
[][] dp2, StringBuilder sb) {

        int wnd=end-start+1;
        if (dp[wnd][start] == wnd) {
            sb.append(s.substring(start, end+1));
            return;
        }
        int sp = dp2[wnd][start];
        int n1 = end-start+1, n2 = sp-start+1;
        if (n1%n2==0 && hp[start][sp]>=n1/n2) {
            sb.append(n1/n2);
            sb.append("[");
            help2(s, start, sp, hp, dp, dp2, sb);
            sb.append("]");
            return;
        }
        help2(s, start, sp, hp, dp, dp2, sb);
        help2(s, sp+1, end, hp, dp, dp2, sb);
    }

    public int help(int count) {
        if (count<10) {
            return 1;
        }
        if (count<100) {
            return 2;
        }
        return 3;
    }
}
```

16、字符串解码（394）

更多硬核内容就在公众号「yes 的练级攻略」

394. 字符串解码

难度 中等 429 收藏 分享 切换为英文 关注 反馈

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: `k[encoded_string]`，表示其中方括号内部的 *encoded_string* 正好重复 *k* 次。注意 *k* 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 *k*，例如不会出现像 `3a` 或 `2[4]` 的输入。

示例 1:

```
输入: s = "3[a]2[bc]"
输出: "aaabcbc"
```

示例 2:

```
输入: s = "3[a2[c]]"
输出: "accaccacc"
```

示例 3:

```
输入: s = "2[abc]3[cd]ef"
输出: "abcabccdcdcdef"
```

示例 4:

```
输入: s = "abc3[cd]xyz"
输出: "abccdcdcdxyz"
```

https://leetcode-cn.com/problems/decode-string/solution/decode-string-fu-zhu-zhan-fa-di-gui-fa-by-jyd/

解法一：辅助栈法

- 本题难点在于括号内嵌套括号，需要**从内向外**生成与拼接字符串，这与栈的**先入后出**特性对应。
- **算法流程：**
  1. 构建辅助栈 `stack`， 遍历字符串 `s` 中每个字符 `c`；
     - 当 `c` 为数字时，将数字字符转化为数字 `multi`，用于后续倍数计算；
     - 当 `c` 为字母时，在 `res` 尾部添加 `c`；
     - 当 `c` 为 `[` 时，将当前 `multi` 和 `res` 入栈，并分别置空置 0：
       - 记录此 `[` 前的临时结果 `res` 至栈，用于发现对应 `]` 后的拼接操作；
       - 记录此 `[` 前的倍数 `multi` 至栈，用于发现对应 `]` 后，获取 `multi` × `[...]` 字符串。
       - 进入到新 `[` 后， `res` 和 `multi` 重新记录。
     - 当 `c` 为 `]` 时， `stack` 出栈，拼接字符串 `res = last_res + cur_multi * res`，其中:
       - `last_res` 是上个 `[` 到当前 `[` 的字符串，例如 `"3[a2[c]]"` 中的 `a`；
       - `cur_multi` 是当前 `[` 到 `]` 内字符串的重复倍数，例如 `"3[a2[c]]"` 中的 `2`。
  2. 返回字符串 `res`。
- **复杂度分析：**
  - 时间复杂度 $O(N)$，一次遍历 `s`；
  - 空间复杂度 $O(N)$，辅助栈在极端情况下需要线性空间，例如 `2[2[2[a]]]`。

更多硬核内容就在公众号「yes 的练级攻略」

方法 1：辅助栈（2ms）

```
class Solution {
    public String decodeString(String s) {
        StringBuilder res = new StringBuilder();
        int multi = 0;
        LinkedList<Integer> stack_multi = new LinkedList<>();
        LinkedList<String> stack_res = new LinkedList<>();
        for(Character c : s.toCharArray()) {
            if(c == '[') {
                stack_multi.addLast(multi);
                stack_res.addLast(res.toString());
                multi = 0;
                res = new StringBuilder();
            }
            else if(c == ']') {
                StringBuilder tmp = new StringBuilder();
                int cur_multi = stack_multi.removeLast();
                for(int i = 0; i < cur_multi; i++) tmp.append(res);
                res = new StringBuilder(stack_res.removeLast() + tmp);
            }
            else if(c >= '0' && c <= '9') multi = multi * 10 + Integer.
parseInt(c + "");
            else res.append(c);
        }
        return res.toString();
    }
}
```

解法二：递归法

- 总体思路与辅助栈法一致，不同点在于将 `[` 和 `]` 分别作为递归的开启与终止条件：
  - 当 `s[i] == ']'` 时，返回当前括号内记录的 `res` 字符串与 `]` 的索引 `i`（更新上层递归指针位置）；
  - 当 `s[i] == '['` 时，开启新一层递归，记录此 `[...]` 内字符串 `tmp` 和递归后的最新索引 `i`，并执行 `res + multi * tmp` 拼接字符串。
  - 遍历完毕后返回 `res`。
- **复杂度分析：**
  - 时间复杂度 $O(N)$，递归会更新索引，因此实际上还是一次遍历 `s`；
  - 空间复杂度 $O(N)$，极端情况下递归深度将会达到线性级别。

方法 2：递归（0ms）

```
class Solution {
    public String decodeString(String s) {
        return dfs(s, 0)[0];
    }
    private String[] dfs(String s, int i) {
        StringBuilder res = new StringBuilder();
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        int multi = 0;
        while(i < s.length()) {
            if(s.charAt(i) >= '0' && s.charAt(i) <= '9')
                multi = multi * 10 + Integer.parseInt(String.valueOf(s.
charAt(i)));
            else if(s.charAt(i) == '[') {
                String[] tmp = dfs(s, i + 1);
                i = Integer.parseInt(tmp[0]);
                while(multi > 0) {
                    res.append(tmp[1]);
                    multi--;
                }
            }
            else if(s.charAt(i) == ']')
                return new String[] { String.valueOf(i), res.toString()
 };
            else
                res.append(String.valueOf(s.charAt(i)));
            i++;
        }
        return new String[] { res.toString() };
    }
}
```

17、压缩算法（腾讯 2020 招聘-后台第一题）【解压缩】

yes的练级攻略

小Q想要给他的朋友发送一个神秘字符串，但是他发现字符串的过于长了，于是小Q发明了一种压缩算法对字符串中重复的部分进行了压缩，对于字符串中连续的m个相同字符串S将会压缩为[m|S](m为一个整数且1<=m<=100)，例如字符串ABCABCABC将会被压缩为[3|ABC]，现在小Q的同学收到了小Q发送过来的字符串，你能帮助他进行解压缩么?

输入描述:

```
输入第一行包含一个字符串s，代表压缩后的字符串。
s的长度<=1000;
s仅包含大写字母、[、]、|;
解压后的字符串长度不超过100000;
压缩递归层数不超过10层;
```

输出描述:

```
输出一个字符串，代表解压后的字符串。
```

输入例子1:

```
HG[3|B[2|CA]]F
```

输出例子1:

```
HGBCACABCACABCACAF
```

例子说明1:

```
HG[3|B[2|CA]]F->HG[3|BCACA]F->HGBCACABCACABCACAF
```

【String 中常见的函数】

java.lang.String.contains() 方法返回 true，当且仅当此字符串包含指定的 char 值序列。

Java 中字符串中子串的查找共有四种方法，如下：

1、int indexOf(String str) ：返回第一次出现的指定子字符串在此字符串中的索引。

2、int indexOf(String str, int startIndex)：从指定的索引处开始，返回第一次出现的指定子字符串在此字符串中的索引。

3、int lastIndexOf(String str) ：返回在此字符串中最右边出现的指定子字符串的索引。

4、int lastIndexOf(String str, int startIndex) ：从指定的索引处开始向后搜索，返回在此字符串中最后一次出现的指定子字符串的索引。

public String substring(int beginIndex)：返回一个新的字符串，它是此字符串的一个子字符串。该子字符串从指定索引处的字符开始，直到此字符串末尾。

public String substring(int beginIndex, int endIndex)：返回一个新字符串，它是此字符串的一个子字符串。该子字符串从指定的 beginIndex 处开始，直到索引 endIndex - 1 处的字符。因此，该子字符串的长度为 endIndex-beginIndex。

beginIndex -- 起始索引（包括）；endIndex -- 结束索引（不包括）。

java 中 String 类中的 replace 方法， replace(String old, String new);

String.join(CharSequence delimiter, CharSequence... elements);

java.util.Collections.ncopies(int n, T o)：n 是元素的个数，o 是反复出现的元素。

Integer.parseInt(String)的作用就是将 String 字符类型数据转换为 Integer 整型数据。

具体代码：

```
import java.util.*;
public class Main{
    public String decode (String s) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (s.length() < 2) return s;
        while (s.contains("]")){
            int right = s.indexOf("]");
            int left = s.lastIndexOf("[", right);
            String repeatStr = s.substring(left + 1, right);
            String[] split = repeatStr.split("\\|");
            s = s.replace("[" + repeatStr + "]",
                String.join("", Collections.nCopies(Integer.parseInt(split[0]), split[1])));
        }
        return s;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        //String next = sc.next();
        //sc.close();
        String next = sc.nextLine();
        Main m = new Main();
        System.out.println(m.decode(next));
    }
}
```

18、反转数组

3、反转数组

```
void reverse(int[] nums) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        // swap(nums[left], nums[right])
        int temp = nums[left];
        nums[left] = nums[right];
        nums[right] = temp;
        left++; right--;
    }
}
```

19、旋转数组（189）

禁止商用

更多硬核内容就在公众号「yes 的练级攻略」

189. 旋转数组

难度 简单 674 收藏 分享 切换为英文 关注 反馈

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

示例 1:

```
输入: [1,2,3,4,5,6,7] 和 k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右旋转 1 步: [7,1,2,3,4,5,6]
向右旋转 2 步: [6,7,1,2,3,4,5]
向右旋转 3 步: [5,6,7,1,2,3,4]
```

示例 2:

```
输入: [-1,-100,3,99] 和 k = 2
输出: [3,99,-1,-100]
解释:
向右旋转 1 步: [99,-1,-100,3]
向右旋转 2 步: [3,99,-1,-100]
```

说明:

- 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
- 要求使用空间复杂度为 O(1) 的 原地 算法。

https://leetcode-cn.com/problems/rotate-array/solution/xuan-zhuan-shu-zu-by-leetcode/

方法 4：使用反转

算法

这个方法基于这个事实：当我们旋转数组 k 次，$k\%n$ 个尾部元素会被移动到头部，剩下的元素会被向后移动。

在这个方法中，我们首先将所有元素反转。然后反转前 k 个元素，再反转后面 $n-k$ 个元素，就能得到想要的结果。

假设 $n=7$ 且 $k=3$。

```
原始数组                  : 1 2 3 4 5 6 7
反转所有数字后            : 7 6 5 4 3 2 1
反转前 k 个数字后         : 5 6 7 4 3 2 1
反转后 n-k 个数字后       : 5 6 7 1 2 3 4 --> 结果
```

复杂度分析

- 时间复杂度：$O(n)$。n 个元素被反转了总共 3 次。
- 空间复杂度：$O(1)$。没有使用额外的空间。

方法 1：反转 3 次

```
class Solution {
    public void rotate(int[] nums, int k) {
        if (nums.length == 0) return;
        k %= nums.length;
        reverse(nums, 0, nums.length - 1);
        reverse(nums, 0, k - 1);
        reverse(nums, k, nums.length - 1);
    }
```

```
    public void reverse(int[] nums, int left, int right){
        while (left < right) {
            int tmp = nums[left];
            nums[left] = nums[right];
            nums[right] = tmp;
            left++;
            right--;
        }
    }
}
```

方法 2：使用环状替换

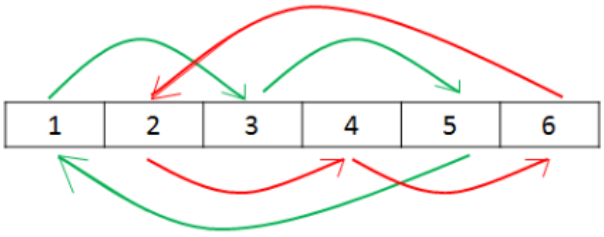
算法

如果我们直接把每一个数字放到它最后的位置，但这样的后果是遗失原来的元素。因此，我们需要把被替换的数字保存在变量 $temp$ 里面。然后，我们将被替换数字（$temp$）放到它正确的位置，并继续这个过程 n 次， n 是数组的长度。这是因为我们需要将数组里所有的元素都移动。但是，这种方法可能会有个问题，如果 $n\%k == 0$，其中 $k = k\%n$ （因为如果 k 大于 n ，移动 k 次实际上相当于移动 $k\%n$ 次）。这种情况下，我们会发现在没有遍历所有数字的情况下回到出发数字。此时，我们应该从下一个数字开始再重复相同的过程。

现在，我们看看上面方法的证明。假设，数组里我们有 n 个元素并且 k 是要求移动的次数。更进一步，假设 $n\%k = 0$ 。第一轮中，所有移动数字的下标 i 满足 $i\%k == 0$ 。这是因为我们每跳 k 步，我们只会到达相距为 k 个位置下标的数。每一轮，我们都会移动 $\frac{n}{k}$ 个元素。下一轮中，我们会移动满足 $i\%k == 1$ 的位置的数。这样的轮次会一直持续到我们再次遇到 $i\%k == 0$ 的地方为止，此时 $i = k$ 。此时在正确位置上的数字共有 $k \times \frac{n}{k} = n$ 个。因此所有数字都在正确位置上。

让我们看一下接下来的例子，以更好地说明这个过程：

```
nums: [1, 2, 3, 4, 5, 6]
k: 2
```



k = 2

复杂度分析

- 时间复杂度：$O(n)$ 。只遍历了每个元素一次。
- 空间复杂度：$O(1)$ 。使用了常数个额外空间。

```
public class Solution {
    public void rotate(int[] nums, int k) {
        k = k % nums.length;
        int count = 0;
        for (int start = 0; count < nums.length; start++) {
            int current = start;
            int prev = nums[start];
            do {
```

```
                int next = (current + k) % nums.length;
                int temp = nums[next];
                nums[next] = prev;
                prev = temp;
                current = next;
                count++;
            } while (start != current);
        }
    }
}
```

20、反转字符串（344）

344. 反转字符串

难度 简单 263

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

不要给另外的数组分配额外的空间，你必须原地**修改输入数组**、使用 O(1) 的额外空间解决这一问题。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

示例 1：

```
输入：["h","e","l","l","o"]
输出：["o","l","l","e","h"]
```

示例 2：

```
输入：["H","a","n","n","a","h"]
输出：["h","a","n","n","a","H"]
```

用 yes的练级攻略

https://leetcode-cn.com/problems/reverse-string/solution/jian-dan-de-si-lu-by-c-wwb-20/

```
class Solution {
    public void reverseString(char[] s) {
        int left = 0, right = s.length - 1;
        while (left < right){
            char tmp = s[left];
            s[left] = s[right];
            s[right] = tmp;
            left++;
            right--;
        }
    }
}
```

21、反转字符串 2（541）【反转从字符串开头算起的每隔 2k 个字符的前 k 个字符】

541. 反转字符串 II

难度 简单 83 收藏 分享 切换为英文 关注 反馈

给定一个字符串 `s` 和一个整数 `k`，你需要对从字符串开头算起的每隔 `2k` 个字符的前 `k` 个字符进行反转。

- 如果剩余字符少于 `k` 个，则将剩余字符全部反转。
- 如果剩余字符小于 `2k` 但大于或等于 `k` 个，则反转前 `k` 个字符，其余字符保持原样。

示例:

```
输入: s = "abcdefg", k = 2
输出: "bacdfeg"
```

提示:

1. 该字符串只包含小写英文字母。
2. 给定字符串的长度和 `k` 在 `[1, 10000]` 范围内。

https://leetcode-cn.com/problems/reverse-string-ii/solution/java-shuang-zhi-zhen-jian-dan-yi-dong-bu-dong-ni-l/

```java
class Solution {
    public String reverseStr(String s, int k) {
        char[] ch = s.toCharArray();
        int n = ch.length;
        for (int i = 0; i < n; i += 2*k){
            int left = i, right = (i + k - 1 < n) ? i + k - 1 : n - 1;
            while (left <= right){
                char tmp = ch[left];
                ch[left] = ch[right];
                ch[right] = tmp;
                left++;
                right--;
            }
        }
        return new String(ch);
    }
}
```

22、旋转字符串（796）【字符串轮转（面试题 01.09）】

更多硬核内容就在公众号「yes 的练级攻略」

796. 旋转字符串

难度 简单 96 收藏 分享 切换为英文 关注 反馈

给定两个字符串, `A` 和 `B`。

`A` 的旋转操作就是将 `A` 最左边的字符移动到最右边。 例如, 若 `A = 'abcde'`，在移动一次之后结果就是`'bcdea'` 。如果在若干次旋转操作之后，`A` 能变成`B`，那么返回`True`。

示例 1:
输入: A = 'abcde', B = 'cdeab'
输出: true

示例 2:
输入: A = 'abcde', B = 'abced'
输出: false

注意:

- `A` 和 `B` 长度不超过 `100`。

https://leetcode-cn.com/problems/rotate-string/solution/xuan-zhuan-zi-fu-chuan-by-leetcode/

方法二：判断子串

由于 `A + A` 包含了所有可以通过旋转操作从 `A` 得到的字符串，因此我们只需要判断 `B` 是否为 `A + A` 的子串即可。

Java Python

```
class Solution {
    public boolean rotateString(String A, String B) {
        return A.length() == B.length() && (A + A).contains(B);
    }
}
```

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 是字符串 `A` 的长度。
- 空间复杂度：$O(N)$，即 `A + A` 需要的空间。

方法 1：判断子串

```
class Solution {
    public boolean rotateString(String A, String B) {
        return A.length() == B.length() && (A + A).contains(B);
    }
}
```

方法四：KMP 算法

判断一个串是否为另一个串的子串的最优时间复杂度的算法是 KMP 算法。和方法二相同，我们只需要用 KMP 算法判断 `B` 是否为 `A + A` 的子串即可。KMP 算法较难理解，这里给出了力扣第 28 题 实现 strstr() 讨论区中一个高赞 KMP 算法详解，以及著名博主 Matrix67 的 KMP 算法详解。

https://leetcode-cn.com/problems/implement-strstr/solution/kmp-suan-fa-xiang-jie-by-labuladong/

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是字符串 `A` 的长度。
- 空间复杂度：$O(N)$。

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

首先还是简单介绍一下 KMP 算法和暴力匹配算法的不同在哪里，难点在哪里，和动态规划有啥关系。

暴力的字符串匹配算法很容易写，看一下它的运行逻辑：

```
// 暴力匹配（伪码）
int search(String pat, String txt) {
    int M = pat.length;
    int N = txt.length;
    for (int i = 0; i < N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (pat[j] != txt[i+j])
                break;
        }
        // pat 全都匹配了
        if (j == M) return i;
    }
    // txt 中不存在 pat 子串
    return -1;
}
```

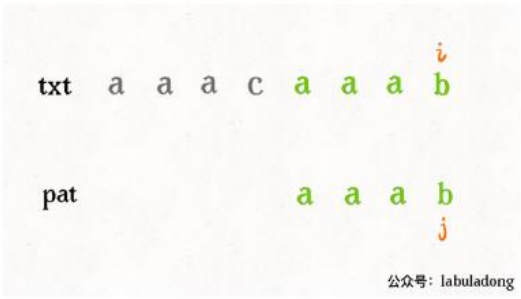
对于暴力算法，如果出现不匹配字符，同时回退 `txt` 和 `pat` 的指针，嵌套 for 循环，时间复杂度 $O(MN)$，空间复杂度$O(1)$。最主要的问题是，如果字符串中重复的字符比较多，该算法就显得很蠢。

比如 txt = "aaacaaab" pat = "aaab"：

很明显，`pat` 中根本没有字符 c，根本没必要回退指针 `i`，暴力解法明显多做了很多不必要的操作。

KMP 算法的不同之处在于，它会花费空间来记录一些信息，在上述情况中就会显得很聪明：

更多硬核内容就在公众号「yes 的练级攻略」



再比如类似的 txt = "aaaaaaab" pat = "aaab"，暴力解法还会和上面那个例子一样蠢蠢地回退指针 `i`，而 KMP 算法又会要聪明：

因为 KMP 算法知道字符 b 之前的字符 a 都是匹配的，所以每次只需要比较字符 b 是否被匹配就行了。

KMP 算法永不回退 `txt` 的指针 `i`，不走回头路（不会重复扫描 `txt`），而是借助 `dp` 数组中储存的信息把 `pat` 移到正确的位置继续匹配，时间复杂度只需 O(N)，用空间换时间，所以我认为它是一种动态规划算法。

KMP 算法的难点在于，如何计算 `dp` 数组中的信息？如何根据这些信息正确地移动 `pat` 的指针？这个就需要**确定有限状态自动机**来辅助了，别怕这种高大上的文学词汇，其实和动态规划的 `dp` 数组如出一辙，等你学会了也可以拿这个词去吓唬别人。

还有一点需要明确的是：**计算这个 `dp` 数组，只和 `pat` 串有关**。意思是说，只要给我个 `pat`，我就能通过这个模式串计算出 `dp` 数组，然后你可以给我不同的 `txt`，我都不怕，利用这个 `dp` 数组我都能在 O(N) 时间完成字符串匹配。

具体来说，比如上文举的两个例子：

方法 2：KMP

```
class Solution {
    public boolean rotateString(String A, String B) {
        int N = A.length();
        if (N != B.length()) return false;
        if (N == 0) return true;
        //Compute shift table
        int[] shifts = new int[N+1];
        Arrays.fill(shifts, 1);
        int left = -1;
        for (int right = 0; right < N; ++right) {
            while (left >= 0 && (B.charAt(left) != B.charAt(right)))
                left -= shifts[left];
            shifts[right + 1] = right - left++;
        }
        //Find match of B in A+A
        int matchLen = 0;
        for (char c: (A+A).toCharArray()) {
```

```
            while (matchLen >= 0 && B.charAt(matchLen) != c)
                matchLen -= shifts[matchLen];
            if (++matchLen == N) return true;
        }
        return false;
    }
}
```

23、左旋转字符串（剑指 58-2）

剑指 Offer 58 - II. 左旋转字符串

难度 简单 46 收藏 分享 切换为英文 关注 反馈

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

示例 1：

```
输入: s = "abcdefg", k = 2
输出: "cdefgab"
```

示例 2：

```
输入: s = "lrloseumgh", k = 6
输出: "umghlrlose"
```

限制：

- `1 <= k < s.length <= 10000`

https://leetcode-cn.com/problems/zuo-xuan-zhuan-zi-fu-chuan-lcof/solution/mian-shi-ti-58-ii-zuo-xuan-zhuan-zi-fu-chuan-qie-p/

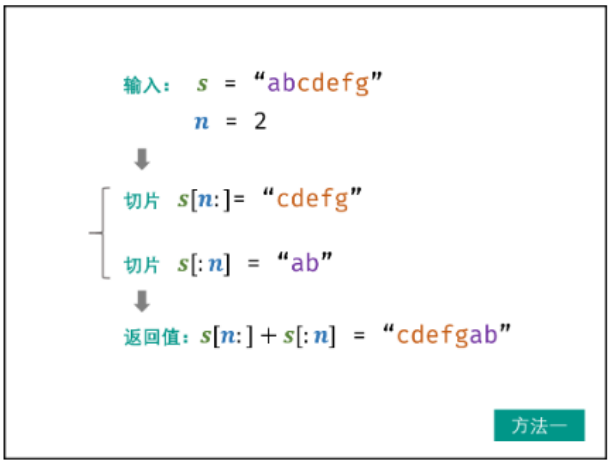
方法一：字符串切片

> 应用字符串切片函数，可方便实现左旋转字符串。

获取字符串 $s[n:]$ 切片和 $s[:n]$ 切片，使用 "+" 运算符拼接并返回即可。

复杂度分析：

- **时间复杂度** $O(N)$：其中 N 为字符串 s 的长度，字符串切片函数为线性时间复杂度（参考资料）；
- **空间复杂度** $O(N)$：两个字符串切片的总长度为 N。



方法 1：字符串切片

```
class Solution {
    public String reverseLeftWords(String s, int n) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        return s.substring(n, s.length()) + s.substring(0, n);
    }
}
```

方法 2：三次翻转字符串

```
class Solution {
    public String reverseLeftWords(String s, int n) {
        if (s.length() <= 1) return s;
        char[] ch = s.toCharArray();
        reverse(ch, 0, n-1);
        reverse(ch, n, s.length() - 1);
        reverse(ch, 0, s.length() - 1);
        return new String(ch);
    }
    public void reverse(char[] ch, int start, int end){
        while (start < end){
            char tmp = ch[start];
            ch[start] = ch[end];
            ch[end] = tmp;
            start++;
            end--;
        }
    }
}
```

公众号：
yes的练级攻略

方法二：列表遍历拼接

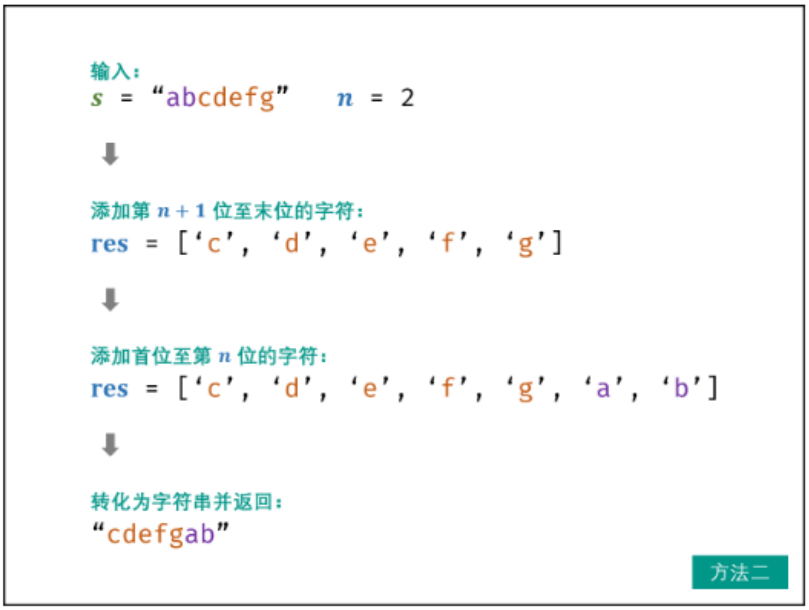
> 若面试规定不允许使用 **切片函数** ，则使用此方法。

算法流程：

1. 新建一个 list(Python)、StringBuilder(Java) ，记为 res ；
2. 先向 res 添加 “第 $n+1$ 位至末位的字符” ；
3. 再向 res 添加 “首位至第 n 位的字符” ；
4. 将 res 转化为字符串并返回。

复杂度分析：

- **时间复杂度** $O(N)$ ： 线性遍历 s 并添加，使用线性时间；
- **空间复杂度** $O(N)$ ： 新建的辅助 res 使用 $O(N)$ 大小的额外空间。



方法 3：列表遍历拼接

```
class Solution {
    public String reverseLeftWords(String s, int n) {
        StringBuilder res = new StringBuilder();
        for(int i = n; i < n + s.length(); i++)
            res.append(s.charAt(i % s.length()));
        return res.toString();
    }
}
```

24、整数反转（7）【不需要双指针】

更多硬核内容就在公众号「yes 的练级攻略」

7. 整数反转

难度 简单 2063 收藏 分享 切换为英文 关注 反馈

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1:

```
输入: 123
输出: 321
```

示例 2:

```
输入: -123
输出: -321
```

示例 3:

```
输入: 120
输出: 21
```

注意:

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为 $[-2^{31}, 2^{31} - 1]$。请根据这个假设，如果反转后整数溢出那么就返回 0。

https://leetcode-cn.com/problems/reverse-integer/solution/zheng-shu-fan-zhuan-by-leetcode/

方法：弹出和推入数字 & 溢出前进行检查

思路

我们可以一次构建反转整数的一位数字。在这样做的时候，我们可以预先检查向原整数附加另一位数字是否会导致溢出。

算法

反转整数的方法可以与反转字符串进行类比。

我们想重复"弹出" x 的最后一位数字，并将它"推入"到 rev 的后面。最后，rev 将与 x 相反。

要在没有辅助堆栈 / 数组的帮助下 "弹出" 和 "推入" 数字，我们可以使用数学方法。

```
//pop operation:
pop = x % 10;
x /= 10;

//push operation:
temp = rev * 10 + pop;
rev = temp;
```

但是，这种方法很危险，因为当 $\text{temp} = \text{rev} \cdot 10 + \text{pop}$ 时会导致溢出。

幸运的是，事先检查这个语句是否会导致溢出很容易。

为了便于解释，我们假设 rev 是正数。

1. 如果 $temp = \text{rev} \cdot 10 + \text{pop}$ 导致溢出，那么一定有 $\text{rev} \geq \frac{INTMAX}{10}$。
2. 如果 $\text{rev} > \frac{INTMAX}{10}$，那么 $temp = \text{rev} \cdot 10 + \text{pop}$ 一定会溢出。
3. 如果 $\text{rev} == \frac{INTMAX}{10}$，那么只要 $\text{pop} > 7$，$temp = \text{rev} \cdot 10 + \text{pop}$ 就会溢出。

当 rev 为负时可以应用类似的逻辑。

```java
class Solution {
    public int reverse(int x) {
        int rev = 0;
        while (x != 0) {
            int pop = x % 10;
            x /= 10;
```

```
            if (rev > Integer.MAX_VALUE/10 || (rev == Integer.MAX_VALUE
 / 10 && pop > 7)) return 0;
            if (rev < Integer.MIN_VALUE/10 || (rev == Integer.MIN_VALUE
 / 10 && pop < -8)) return 0;
            rev = rev * 10 + pop;
        }
        return rev;
    }
}
```

25、翻转字符串中的单词（151）（lintcode53）（剑指 58-1）【快手真题】

53. 翻转字符串中的单词

给定一个字符串，逐个翻转字符串中的每个单词。

样例
样例 1:
输入: "the sky is blue"
输出: "blue is sky the"

样例解释:
返回逐字反转的字符串.

样例 2:
输入: "hello world"
输出: "world hello"

样例解释:
返回逐字反转的字符串.

说明

- 单词的构成：无空格字母构成一个单词，有些单词末尾会带有标点符号
- 输入字符串是否包括前导或者尾随空格？可以包括，但是反转后的字符不能包括
- 如何处理两个单词间的多个空格？在反转字符串中间空格减少到只含一个

```
public class Solution {
    /*
     * @param s: A string
     * @return: A string
     */
    public String reverseWords(String s) {
        if(s == null || s.length() == 0) return s;
        String[] arr = s.split(" ");
        StringBuilder sb = new StringBuilder();
        for (int i = arr.length - 1; i >= 0; i--) {
            if (!arr[i].equals("")){
                if (sb.length() > 0) {
```

```
                    sb.append(" ");
                }
                sb.append(arr[i]);
            }
        }
        return sb.toString();
    }
}
```

【快手真题】需要结尾依旧是"."，且结尾的"."与前一个单词之间无空格。
输入一个英文句子，词之间有 1 个或者若干个空格，句子以英文标点"."结尾。
要求颠倒该句子中的词语顺序，并且词之间有且只有一个空格，结尾仍然是"."，结尾的"."与前一个单词之间无空格。

```
import java.util.*;
public class ReverseWord {
    public String reverseWords (String s) {
        if (s == null || s.length() < 2) return s;
        String[] arr = s.split(" ");
        StringBuilder sb = new StringBuilder();

        arr[arr.length - 1] = removePoint(arr[arr.length - 1]);
        sb.append(arr[arr.length - 1]);

        for (int i = arr.length - 2; i >= 0; i--) {
            if ( ! arr[i].equals("")) {
                if (sb.length() > 0) {
                    sb.append(" ");
                }
                sb.append(arr[i]);
            }
        }
        sb.append(".");
        return sb.toString();
    }

    public String removePoint (String before) {
        char[] tmp = before.toCharArray();
        StringBuilder after = new StringBuilder();
        for (int i = 0; i < tmp.length; i++) {
            if (tmp[i] != '.') {
                after.append(tmp[i]);
            }
        }
        return after.toString();
    }
```

禁止商用 公众号：yes的练级攻略

```
    public static void main(String[] args) {
        while (true) {
            System.out.println("请输入数据： ");
            Scanner sc = new Scanner(System.in);
            String s = sc.nextLine();
            ReverseWord rw = new ReverseWord();
            String s_reverse = rw.reverseWords(s);
            System.out.println(s_reverse);
        }
    }
}
```

26、翻转字符串 3（lintcode 927）

927. 翻转字符串II

给定输入的 字符数组 ，逐词翻转数组。单词被定义为不包含空格的字符串.
输入字符数组不包含前导或尾部空格，单词总是用单个空格分隔。

样例
样例1

输入: s = "the sky is blue"
输出: "blue is sky the"
样例2

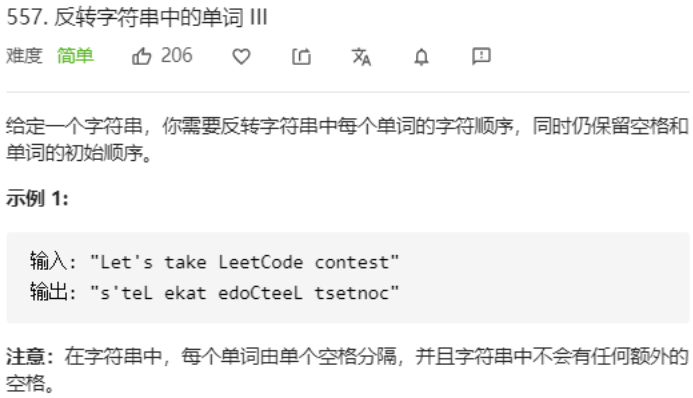
输入: "a b c"
输出: "c b a"
挑战
你能在不分配额外空间的情况下原地解决这个问题吗?

yes的练级攻略

```
public class Solution {
    /**
     * @param str: a string
     * @return: return a string
     */
    public char[] reverseWords(char[] str) {
        int pre = 0, n = str.length;
        for (int i = 0; i <= n; i++){
            if (i == n || str[i] == ' '){
                reverse(str, pre, i - 1);
                pre = i + 1;
            }
        }
        reverse(str, 0, n-1);
        return str;
    }
```

```
    public void reverse(char[] str, int left, int right){
        while (left < right) {
            char tmp = str[left];
            str[left] = str[right];
            str[right] = tmp;
            left++;
            right--;
        }
    }
}
```

27、反转字符串中的单词 3（557）【反转字符串 3（lintcode 1173）】



557. 反转字符串中的单词 III

难度 简单 206

给定一个字符串，你需要反转字符串中每个单词的字符顺序，同时仍保留空格和单词的初始顺序。

示例 1:

```
输入: "Let's take LeetCode contest"
输出: "s'teL ekat edoCteeL tsetnoc"
```

注意：在字符串中，每个单词由单个空格分隔，并且字符串中不会有任何额外的空格。

方法 1：调用自身函数

```
class Solution {
    public String reverseWords(String s) {
        String[] words = s.split(" ");
        StringBuilder reverseWord = new StringBuilder();
        for(int index = 0;index < words.length;index++){
                if(index != 0 && index != (words.length )){
                reverseWord.append(" ");
            }
            StringBuilder newWord = new StringBuilder(words[index]);
            newWord.reverse();
            reverseWord.append(newWord);
        }
        return reverseWord.toString();
    }
}
```

方法 2：自己写反转函数

```
class Solution {
    /**
     * @param s: a string
     * @return: reverse the order of characters in each word within a sentence
while still preserving whitespace and initial word order
```

```
     */
    public String reverseWords(String s) {
        char[] sentence = s.toCharArray();
        int i = 0, j = 0;
        while (j < sentence.length){
            while (i < sentence.length && sentence[i] == ' ') ++i;
            while (j < sentence.length && sentence[j] != ' ') ++j;
            reverse(sentence, i, j-1);
            i = j;
            ++j;
        }
        return String.valueOf(sentence);
    }
    public void reverse(char[] str, int left, int right){
        while (left < right) {
            char tmp = str[left];
            str[left] = str[right];
            str[right] = tmp;
            left++;
            right--;
        }
    }
}
```

28、恢复空格（面试题 17.13）

面试题 17.13. 恢复空格

难度 中等 147 收藏 分享 切换为英文 关注 反馈

哦，不！你不小心把一个长篇文章中的空格、标点都删掉了，并且大写也弄成了小写。像句子“I reset the computer. It still didn’t boot!”已经变成了“iresetthecomputeritstilldidntboot”。在处理标点符号和大小写之前，你得先把它断成词语。当然了，你有一本厚厚的词典 dictionary，不过，有些词没在词典里。假设文章用 sentence 表示，设计一个算法，把文章断开，要求未识别的字符最少，返回未识别的字符数。

注意：本题相对原题稍作改动，只需返回未识别的字符数

示例：

输入：
dictionary = ["looked","just","like","her","brother"]
sentence = "jesslookedjustliketimherbrother"
输出： 7
解释： 断句后为"**jess** looked just like **tim** her brother"，共7个未识别字符。

提示：

- 0 <= len(sentence) <= 1000
- dictionary 中总字符数不超过 150000。
- 你可以认为 dictionary 和 sentence 中只包含小写字母。

https://leetcode-cn.com/problems/re-space-lcci/solution/jian-dan-dp-trieshu-bi-xu-miao-dong-b

更多硬核内容就在公众号「yes 的练级攻略」

y-sweetiee/

题目分析

题意：给定字符串，尽可能多地匹配字典内的单词，即最少未匹配数。

分：拿到题目，先想贪心能不能做，显然不行。比如给定字符串：`abcdef`，字典：`["abcd", "ab", "def"]`，贪心匹配是 **abcd** ef，有2个字符未匹配。显然有更优匹配策略： **ab** c **def**，只有 1 个字符未匹配。所以这题需要用动态规划来解决（怎么思考一个题目可不可以用 dp 来解决，先假设 dp 表达含义，再想转移方程，很多时候自然而然就出来了）。

1、状态定义：

`dp[i]` 表示字符串的前 `i` 个字符的最少未匹配数。

2、状态转移：

假设当前我们已经考虑完了前 `i` 个字符了，对于前 `i + 1` 个字符对应的最少未匹配数：

1. 第 `i + 1` 个字符未匹配，则 `dp[i + 1] = dp[i] + 1`，即不匹配数加 1；
2. 遍历前 `i` 个字符，若以其中某一个下标 `idx` 为开头、以第 `i + 1` 个字符为结尾的字符串正好在词典里，则 `dp[i] = min(dp[i], dp[idx])` 更新 `dp[i]`。

于是，有了**解法一**。

3、解法一代码：

时间复杂度是 $O(n^2)$，n 为待匹配字符串的长度。

方法 1：DP（885ms）

```
class Solution {
    public int respace(String[] dictionary, String sentence) {
        Set<String> dict = new HashSet<>(Arrays.asList(dictionary));
        int n = sentence.length();
        int[] dp = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            dp[i] = dp[i - 1] + 1;
            for (int idx = 0; idx < i; idx++) {
                if (dict.contains(sentence.substring(idx, i))) {
                    dp[i] = Math.min(dp[i], dp[idx]);
                }
            }
        }
        return dp[n];
    }
}
```

4、使用 Trie 进行优化：

对于上述解法，计算 `dp[i + 1]` 时，我们需要用 `idx` 来遍历前 `i` 个字符，逐个判断以 `idx` 为开头，以第 `i + 1` 个字符为结尾的字符串是否在字典里。
这一步可以利用字典树来加速，通过字典树我们可以查询以第 `i + 1` 个字符为结尾的单词有哪些（构建字典树时将单词逆序插入即可）。
关于字典树的学习，可以看下 这篇题解，这就是**解法二**。
时间复杂度是 $O(m + n^2)$，m 是字典长度，n 为待匹配字符串的长度。为什么还是 n^2 呢？因为有可能状态转移的时候，每个位置都需要转移，这是最坏情况，绝大多数情况下远小于 n，所以解法二最终耗时会远小于解法一。

方法 2：字典树（16ms）

```
class TrieNode {
    boolean isWord;
    TrieNode[] children = new TrieNode[26];
    public TrieNode() {}
}
class Solution {
    public int respace(String[] dictionary, String sentence) {
        // 构建字典树
        Trie trie = new Trie();
        for (String word: dictionary) {
            trie.insert(word);
        }
        // 状态转移，dp[i] 表示字符串的前 i 个字符的最少未匹配数
        int n = sentence.length();
        int[] dp = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            dp[i] = dp[i - 1] + 1;
            for (int idx: trie.search(sentence, i - 1)) {
                dp[i] = Math.min(dp[i], dp[idx]);
            }
        }
        return dp[n];
    }
}
class Trie {
    TrieNode root;
    public Trie() {
        root = new TrieNode();
    }
    // 将单词倒序插入字典树
    public void insert(String word) {
        TrieNode cur = root;
        for (int i = word.length() - 1; i >= 0; i--) {
            int c = word.charAt(i) - 'a';
            if (cur.children[c] == null) {
                cur.children[c] = new TrieNode();
            }
            cur = cur.children[c];
        }
        cur.isWord = true;
    }
    // 找到 sentence 中以 endPos 为结尾的单词，返回这些单词的开头下标。
    public List<Integer> search(String sentence, int endPos) {
```

```
        List<Integer> indices = new ArrayList<>();
        TrieNode cur = root;
        for (int i = endPos; i >= 0; i--) {
            int c = sentence.charAt(i) - 'a';
            if (cur.children[c] == null) {
                break;
            }
            cur = cur.children[c];
            if (cur.isWord) {
                indices.add(i);
            }
        }
        return indices;
    }
}
```

29、最长单词（面试题 17.15）

面试题 17.15. 最长单词

难度 中等 8 收藏 分享 切换为英文 关注 反馈

给定一组单词 words，编写一个程序，找出其中的最长单词，且该单词由这组单词中的其他单词组合而成。若有多个长度相同的结果，返回其中字典序最小的一项，若没有符合要求的单词则返回空字符串。

示例：

```
输入： ["cat","banana","dog","nana","walk","walker","dogwalker"]
输出： "dogwalker"
解释： "dogwalker"可由"dog"和"walker"组成。
```

提示：

- 0 <= len(words) <= 200
- 1 <= len(words[i]) <= 100

https://leetcode-cn.com/problems/longest-word-lcci/solution/zui-chang-dan-ci-by-hao-hou-de-yun/

解题思路

此处撰写解题思路
先把字符串数组排序，字符串长的在前面，相同长度的字典序小的在前面，排好序后加入到set里判断是否包含，从第一个字符串开始判断，看是否由其它字符串组成，这里可以用递归
递归出口：如果字符串长的长度为0，说明遍历完了，之前的都满足条件，返回true
递归操作：遍历字符串的第0个位置开始，判断set里是否有，如果0到i的字符串正好包含在set里，下次从i+1的位置开始判断，直到遍历完了，字符串长度为0，没找到则返回false

```
class Solution {
    public String longestWord(String[] words) {
        Arrays.sort(words,(o1,o2)->{
            if(o1.length() == o2.length())
                return o1.compareTo(o2);
            else{
                return Integer.compare(o2.length(),o1.length());
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            }
        });

        Set<String> set = new HashSet<>(Arrays.asList(words));
        for(String word : words){
            set.remove(word);
            if(find(set,word))
                 return word;
        }
        return "";
    }

    public boolean find(Set<String> set, String word){
        if(word.length() == 0)
            return true;
        for(int i = 0; i < word.length(); i++){
            if(set.contains(word.substring(0,i+1)) && find(set,word.substring(i
+1)))
                return true;
        }
        return false;
    }
}
```

30、词典中最长的单词（720）

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

720. 词典中最长的单词

难度 简单 99 收藏 分享 切换为英文 关注 反馈

给出一个字符串数组 `words` 组成的一本英语词典。从中找出最长的一个单词，该单词是由 `words` 词典中其他单词逐步添加一个字母组成。若其中有多个可行的答案，则返回答案中字典序最小的单词。

若无答案，则返回空字符串。

示例 1：

```
输入：
words = ["w","wo","wor","worl", "world"]
输出："world"
解释：
单词"world"可由"w", "wo", "wor", 和 "worl"添加一个字母组成。
```

示例 2：

```
输入：
words = ["a", "banana", "app", "appl", "ap", "apply", "apple"]
输出："apple"
解释：
"apply"和"apple"都能由词典中的单词组成。但是"apple"的字典序小于"apply"。
```

提示：

- 所有输入的字符串都只包含小写字母。
- `words` 数组长度范围为 `[1,1000]`。
- `words[i]` 的长度范围为 `[1,30]`。

https://leetcode-cn.com/problems/longest-word-in-dictionary/solution/ci-dian-zhong-zui-chang-de-dan-ci-by-leetcode/

方法一：暴力法

对于每个单词，我们可以检查它的全部前缀是否存在，可以通过 `Set` 数据结构来加快查找

算法：

- 当我们找到一个单词它的长度更长且它的全部前缀都存在，我们将更改答案。
- 或者，我们可以事先将单词排序，这样当我们找到一个符合条件的单词就可以认定它是答案。

复杂度分析

- 时间复杂度：$O(\sum w_i^2)$。w_i 指的是 `words[i]` 的长度，在 `Set` 中检查 `words[i]` 全部前缀是否均存在的时间复杂度是 $O(\sum w_i^2)$。
- 空间复杂度：$O(\sum w_i^2)$ 用来存放子串的空间。

```java
class Solution {
    public String longestWord(String[] words) {
        String ans = "";
        Set<String> wordset = new HashSet();
        for (String word: words) wordset.add(word);
        for (String word: words) {
            if (word.length() > ans.length() ||
```

```
                    word.length() == ans.length() && word.compareTo(ans) < 0) {
                boolean good = true;
                for (int k = 1; k < word.length(); ++k) {
                    if (!wordset.contains(word.substring(0, k))) {
                        good = false;
                        break;
                    }
                }
                if (good) ans = word;
            }
        }
        return ans;
    }
}
```

31、最常见的单词（819）

819. 最常见的单词

难度 简单 67 收藏 分享 切换为英文 关注 反馈

给定一个段落 (paragraph) 和一个禁用单词列表 (banned)。返回出现次数最多，同时不在禁用列表中的单词。

题目保证至少有一个词不在禁用列表中，而且答案唯一。

禁用列表中的单词用小写字母表示，不含标点符号。段落中的单词不区分大小写。答案都是小写字母。

示例：

输入：
paragraph = "Bob hit a ball, the hit BALL flew far after it was hit."
banned = ["hit"]
输出： "ball"
解释：
"hit" 出现了3次，但它是一个禁用的单词。
"ball" 出现了2次 (同时没有其他单词出现2次)，所以它是段落里出现次数最多的，且不在禁用列表中的单词。
注意，所有这些单词在段落里不区分大小写，标点符号需要忽略（即使是紧挨着单词也忽略， 比如 "ball,"），
"hit"不是最终的答案，虽然它出现次数更多，但它在禁用单词列表中。

提示：

- 1 <= 段落长度 <= 1000
- 0 <= 禁用单词个数 <= 100
- 1 <= 禁用单词长度 <= 10
- 答案是唯一的, 且都是小写字母 (即使在 paragraph 里是大写的，即使是一些特定的名词，答案都是小写的。)
- paragraph 只包含字母、空格和下列标点符号 !?',;.
- 不存在没有连字符或者带有连字符的单词。
- 单词里只包含字母，不会出现省略号或者其他标点符号。

https://leetcode-cn.com/problems/most-common-word/solution/zui-chang-jian-de-dan-ci-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

简单计数：

我们统计出每个单词出现的次数，忽略所有的标点符号和大小写，答案即为出现次数最多且不在禁用列表中的那个单词。

统计单词的方法有两种。在第一种方法中，我们首先对整个段落按照空格进行分词（split），然后对于分出的每个单词，我们移除标点符号并忽略大小写。在第二种方法中，我们逐字符扫描整个段落，如果遇到一个非字母的符号，那就把之前遇到的字母作为一个单词。

对于每一个单词，我们会放入哈希映射（`Java` 中的 `HashMap` 或者 `Python` 中的 `Counter`）中进行计数。在每次放入单词之后，如果这个单词不在禁用列表中，我们就可以更新一次答案。

复杂度分析

- 时间复杂度：$O(P + B)$，其中 P 是段落 `paragraph` 的长度，B 是禁用列表 `banned` 的长度。
- 空间复杂度：$O(P + B)$，用来进行计数以及存储禁用列表 `banned`。

```
class Solution {
    public String mostCommonWord(String paragraph, String[] banned) {
        paragraph += ".";

        Set<String> banset = new HashSet();
        for (String word: banned) banset.add(word);
        Map<String, Integer> count = new HashMap();

        String ans = "";
        int ansfreq = 0;

        StringBuilder word = new StringBuilder();
        for (char c: paragraph.toCharArray()) {
            if (Character.isLetter(c)) {
                word.append(Character.toLowerCase(c));
            } else if (word.length() > 0) {
                String finalword = word.toString();
                if (!banset.contains(finalword)) {
                    count.put(finalword, count.getOrDefault(finalword,
0) + 1);
                    if (count.get(finalword) > ansfreq) {
                        ans = finalword;
                        ansfreq = count.get(finalword);
                    }
                }
                word = new StringBuilder();
            }
        }

        return ans;
    }
```

}

32、单词转换（面试题 17.22）

面试题 17.22. 单词转换

难度 中等 19 收藏 分享 切换为英文 关注 反馈

给定字典中的两个词，长度相等。写一个方法，把一个词转换成另一个词， 但是一次只能改变一个字符。每一步得到的新词都必须能在字典中找到。

编写一个程序，返回一个可能的转换序列。如有多个可能的转换序列，你可以返回任何一个。

示例 1:

```
输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出:
["hit","hot","dot","lot","log","cog"]
```

示例 2:

```
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: []

解释: endWord "cog" 不在字典中，所以不存在符合要求的转换序列。
```

https://leetcode-cn.com/problems/word-transformer-lcci/solution/95100-by-atlanta-wan/

解题思路

1. 首先构建一个单词的后继集合，key是单词，value是后继列表。找后继的方法：每次变换单词的一个字符，查找单词集中是否存在变换后的单词，存在即加入后继列表。
2. 根据后继列表进行BFS过程。在遍历过程中通过Map记录遍历过程，key是当前单词，value是上一个单词。最后从endWord到beginWord反向索引得到最短的
变换路径。

```
class Solution {
    // 构造后继图，每个单词的后继列表
    public void buildNextWordsMap(HashMap<String, List<String>> nextWordsMap, S
et<String> wordSet) {
        char[] wordArr;
        for (var word : wordSet) {
            wordArr = word.toCharArray();
            List<String> nextWords = new ArrayList<>();
            //对于每个单词，每次变换一位，然后查看 worSet 中是否含有，有的话就是后继。
            for (int i = 0; i < wordArr.length; i++) {
                for (char j = 'a'; j <= 'z'; j++) {
                    if (wordArr[i] != j) {
                        char temp = wordArr[i];
                        wordArr[i] = j;
                        String tempWord = String.valueOf(wordArr);
```

```
                    if (wordSet.contains(tempWord)) {
                        nextWords.add(tempWord);
                    }
                    //还原
                    wordArr[i] = temp;
                }
            }
        }
        nextWordsMap.put(word, nextWords);
    }
}

//执行广度优先搜索
public List<String> BFS(HashMap<String, List<String>> nextWordsMap, String
beginWord, String endWord) {
    //保存结果
    List<String> changePaths = new LinkedList<>();
    //保存搜索路径，key 表示当前节点，value 是当前节点的前驱节点，最后从 endWord 反
向索引至 beginWord 即得到最短的变换路径
    HashMap<String, String> findPathMap = new HashMap<>();

    Queue<String> queue = new LinkedList<>();
    Set<String> isVisited = new HashSet<>();
    queue.offer(beginWord);
    isVisited.add(beginWord);
    outer:
    while (!queue.isEmpty()) {
        String word = queue.poll();
        //遍历后继
        for (var nextWord : nextWordsMap.get(word)) {
            if (!isVisited.contains(nextWord)) {
                queue.add(nextWord);
                findPathMap.put(nextWord, word);
                isVisited.add(nextWord);
                if (nextWord.equals(endWord)) {
                    break outer;
                }
            }
        }
    }
    //反向遍历得到索引路径，因为每个节点只有一个前驱，如果起点到终点有路径就可以到
达。
    String cur = endWord;
    changePaths.add(cur);
```

```
        String pre;
        do {
            pre = findPathMap.get(cur);
            if (pre == null) {
                return new ArrayList<>();
            }
            changePaths.add(pre);
            cur = pre;
        } while (!cur.equals(beginWord));
        //反转
        Collections.reverse(changePaths);
        return changePaths;
    }

    public List<String> findLadders(String beginWord, String endWord, List<Stri
ng> wordList) {
        // 保存单词集
        Set<String> wordSet = new HashSet<>();
        wordSet.add(beginWord);
        wordSet.addAll(wordList);
        if (!wordSet.contains(endWord)) {
            return new ArrayList<>();
        }
        // 保存后继图
        HashMap<String, List<String>> nextWordsMap = new HashMap<>();
        buildNextWordsMap(nextWordsMap, wordSet);
        //执行广度优先遍历
        return BFS(nextWordsMap, beginWord, endWord);
    }
}
```

33、最短单词距离（243）

243. 最短单词距离

难度 简单 30 收藏 分享 切换为英文 关注 反馈

给定一个单词列表和两个单词 *word1* 和 *word2*，返回列表中这两个单词之间的最短距离。

示例:
假设 words = ["practice", "makes", "perfect", "coding", "makes"]

输入: *word1* = "coding", *word2* = "practice"
输出: 3

输入: *word1* = "makes", *word2* = "coding"
输出: 1

注意:
你可以假设 *word1* 不等于 *word2*, 并且 *word1* 和 *word2* 都在列表里。

https://leetcode-cn.com/problems/shortest-word-distance/solution/zui-duan-dan-ci-ju-chi-by-leetcode/

方法 2：遍历一次

算法

我们可以记录两个下标 `i1` 和 `i2` 来显著提高暴力的时间复杂度，我们保存 `word1` 和 `word2` 的 *最近* 出现位置。每次我们发现一个新的单词出现位置，我们不需要遍历整个数组去找到另一个单词，因为我们已经记录了最近出现位置的下标。

复杂度分析

- 时间复杂度：$O(n)$。这个问题的解法是线性的，我们无法比 $O(n)$ 更快是因为我们至少要遍历每个元素一次。
- 空间复杂度：$O(1)$。没有使用额外空间。

```
class Solution {
    public int shortestDistance(String[] words, String word1, String word2) {
     int result = words.length;
        int index1 = -1;
        int index2 = -1;
        for (int i = 0; i < words.length; i++) {
            if (words[i].equals(word1)) {
                // 如果当前的位置为word1,更新 word1 的位置
                index1 = i;
                if (index2 >= 0) {
                    // 计算距离
                    result = Math.min(index1 - index2, result);
                }
            } else if (words[i].equals(word2)) {
                // 如果当前的位置为word2, 更新word2 的位置
                index2 = i;
                if (index1 >= 0) {
                    // 计算距离
                    result = Math.min(index2 - index1, result);
                }
            }
        }
        return result;
    }
}
```

34、最短单词距离 2（244）

更多硬核内容就在公众号「yes 的练级攻略」

244. 最短单词距离 II

难度 中等 21 收藏 分享 切换为英文 关注 反馈

请设计一个类，使该类的构造函数能够接收一个单词列表。然后再实现一个方法，该方法能够分别接收两个单词 *word1* 和 *word2*，并返回列表中这两个单词之间的最短距离。您的方法将被以不同的参数调用 *多次*。

示例:
假设 words = ["practice", "makes", "perfect", "coding", "makes"]

输入: *word1* = "coding", *word2* = "practice"
输出: 3

输入: *word1* = "makes", *word2* = "coding"
输出: 1

注意:
你可以假设 *word1* **不等于** *word2*, 并且 *word1* 和 *word2* 都在列表里。

https://leetcode-cn.com/problems/shortest-word-distance-ii/solution/zui-duan-dan-ci-ju-chi-ii-by-leetcode/

解法

在开始本题的题解之前，我们首先看一下这个题要求我们做的事情的简单版本。我们需要设计一个类，在构造器中接收一个单词列表。这个类有一个我们需要去实现的函数，这个函数的名字叫 `shortest` ，它接收 2 个单词作为输入并返回这两个单词的最近距离。

两个单词的距离定义为它们在单词列表里的下标距离。比方说第一个单词在位置 `i` 出现第二个单词在位置 `j` 出现，那么两者的距离为 `abs(i - j)` 。

问题问我们的单词间 `最小` 不同距离显然说明单词可能会出现在多个不同的位置。如果单词 `word1` 出现了 `K` 次，`word2` 出现了 `L` 次，我们重复检查每一对下标需要 $O(N^2)$ 的时间复杂度，这样没法进行优化，我们在此不会讨论这种比较显然的做法。

暴力方法会考虑所有的下标对（ `word1_location` , `word2_location` ），并看看能产生的最小距离。我们基于这个想法，并尝试用预处理来降低暴力解法的时间复杂度。

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法：使用预处理并将下标排序

想法

一个给定的单词在原单词列表里出现几次。我们假设第一个单词 `word1` 在原数组中出现的位置为 `[i1, i2, i3, i4]` 。类似的，我们假设第二个单词 `word2` 出现的位置在 `[j1, j2, j3]` 。

现在，给定下标列表，我们将尝试找到绝对距离最小的下标对 `(i, j)` 。

> 这个方法的核心想法是如果这些下标对是有序的，我们可以用线性的时间找到距离最小的下标对。

这个思路使用两个指针来实现。对于一个排好序的下标对列表，我们用一个指针 `i` 代表 `word1` 的下标，指针 `j` 表示 `word2` 中的下标。每一次迭代，我们记录下标的距离，也就是 `abs(word1[i] - word2[j])` 。一旦我们更新答案后，我们有两种可能的下一步选择。

$$word1[i] < word2[j]$$

这种情况下，意味着我们往后移动 `j` 是没有意义的。由于单词出现的位置是有序的，我们知道 `word2[j + 1] > word2[j]` ，所以如果我们移动 `j` ，那么 `abs(word1[i] - word2[j + 1])` 只会变得比 `abs(word1[i] - word2[j])` 更大，这对于找最小距离毫无帮助。

> 所以，如果我们 `word1[i] < word2[j]` ，我们往后移动指针 `i` ，也就是我们去尝试第 `(i+1)` 个单词出现的位置的贡献 `abs(word1[i + 1] - word2[j])` 是否能比 `abs(word1[i] - word2[j])` 贡献一个更小距离。我们说尝试的原因是因为答案不一定更优。

让我们来看看两个不同的例子。在第一个例子中，我们看到向后移动 `i` 会给我们全局最优解。而在第二个例子中，我们可以看到向后移动 `i` 并不会求出更好的解。

例子-1

```
word1 = [2,4,5,9]
word2 = [4,10,11]

i, j = 0, 0
min_diff = 2 (abs(2 - 4))
word1[i] < word2[j] 即 2 < 4
  将 i 往后移动一步

i, j = 1, 0 (abs(4 - 4))
min_diff = 0 （我们得到了最优解！）
```

例子-2

```
word1 = [2,7,15,16]
word2 = [4,10,11]

i, j = 0, 0
min_diff = 2 (abs(2 - 4))
word1[i] < word2[j] 即 2 < 4
  将 i 往后移动一步

i, j = 1, 0
min_diff = 2 (2 < abs(7 - 4))

这里，我们没有更新全局结果。这就是我们前面说的将 'i' 移动或许不会得到更优解
但把 'j' 往后移动会让结果变得更差（或者保持不变）
```

更多硬核内容就在公众号「yes 的练级攻略」

下面让我们来讨论第二种情况。

$$word1[i] > word2[j]$$

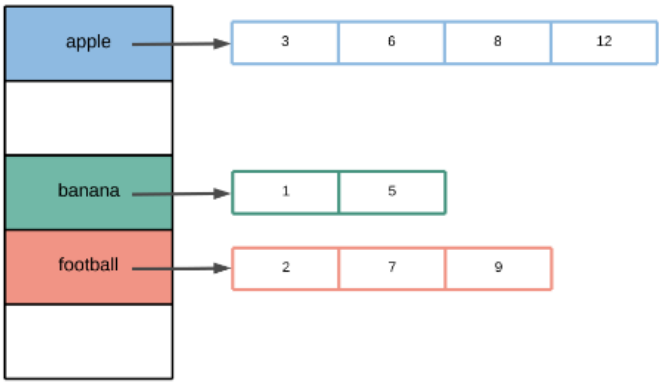
这种情况下，将 `i` 往后移动没有意义，由于下标都排过序了，所以 `word1[i + 1] > word1[i]` 。如果我们将 `i` 往后移动，差 `abs(word1[i + 1] - word2[j])` 会比 `abs(word1[i] - word2[j])` 更大，对找最优解也毫无帮助。

> 所以，与前一种情况类似，如果 `word1[i] > word2[j]` ，我们会把 `j` 往后移动，并看是否能得到更优解。

现在让我们重新看一下这道题的完整解法。

算法

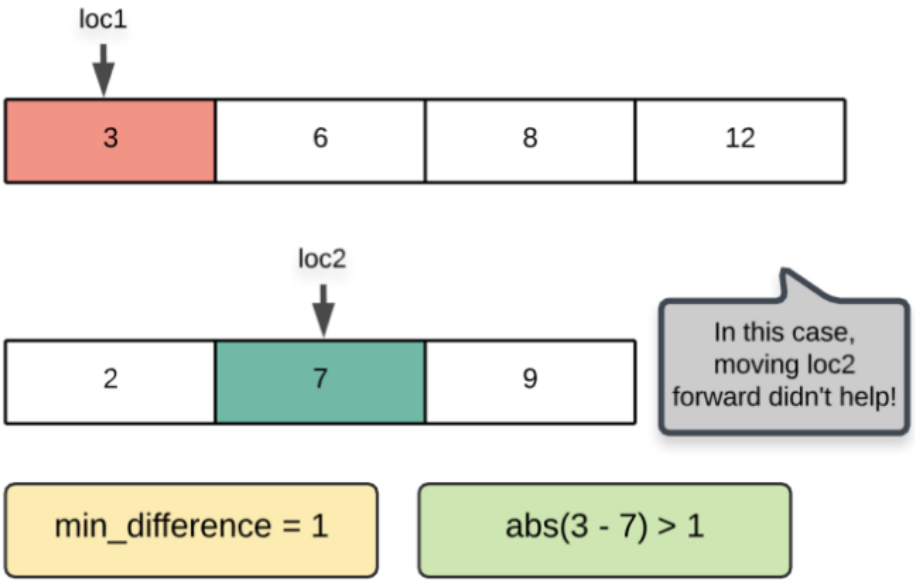
1. 在类的 `构造函数` 中，我们仅仅将单词列表遍历一遍，并用一个字典，将单词与它在数组中出现位置做一个映射。
2. 由于我们遍历单词列表的时候是从左到右的，我们得到的出现序列本身就是有序的，所以我们不需要额外排序。
3. 我们把这个字典叫做 `locations` 。
4. 对于给定的单词对，我们获取它们分别对应的列表数组，记作 `loc1` 和 `loc2` 。
5. 初始化两个指针变量 `l1 = 0` 和 `l2 = 0` 。
6. 对于给定的 `l1` 和 `l2` ，我们首先更新目前为止的最近距离（如果更优），即 `dist = min(dist, abs(loc1[l1] - loc2[l2]))` 。然后，我们判断如果 `loc1[l1] < loc2[l2]` ，那么移动 `l1` 一步，即 `l1 = l1 + 1` ，否则我们移动 `l2` 一步，即 `l2 = l2 + 1` 。
7. 我们继续执行知道有一个指针到达了数组的末尾。
8. 返回全局最优解。



yes的练级攻略

上图展示了在构造函数中我们构建的位置字典。键代表的是单词，值是一个升序包含所有出现位置的列表。我们看一看 `apple` 和 `football` 之间的最近距离。我们需要用到的两个有序列表是 `[3, 6, 8, 12]` 和 `[2, 7, 9]` 。

☆ 收藏



复杂度分析

- 时间复杂度：构造函数的时间复杂度为 $O(N)$ ，其中 N 是原本列表中单词的个数。我们逐一迭代遍历它们，并用一个键值对字典保存起来。然后，对于每一次查询最近距离，时间复杂度是 $O(max(K, L))$ 其中 K 和 L 分别表示两个单词对应数组的长度。同时我们知道， $K = O(N)$ 且 $L = O(N)$ 。因此，总时间复杂度是 $O(N)$ 的。时间复杂度是 $O(max(K, L))$ **而不是** $O(min(K, L))$ 的原因是，有可能较短列表的元素比较长列表的元素都要大，这种情况下，较短列表不会被迭代而较长列表所有元素都会被遍历一次。
- 空间复杂度：$O(N)$ 。 我们构造函数里产生的字典大小为 $O(N)$ 。键是单词列表里所有不同的单词，值是 $0 \ldots N$ 的所有下标。

时间复杂度的说明有点问题：

- 最坏情况下，两个数组的元素按大小正好交叉排列，因此遍历次数是K + L，所以O(K + L)
- 最好情况下，短数组的元素全部比长数组元素小，因此遍历次数是min(K, L)，所以O(min(k,L))
- 评价情况下，时间复杂度为O(N / 2)

```
class WordDistance {
    HashMap<String, ArrayList<Integer>> locations;
    public WordDistance(String[] words) {
        this.locations = new HashMap<String, ArrayList<Integer>>();
        for (int i = 0; i < words.length; i++) {
            ArrayList<Integer> loc = this.locations.getOrDefault(words[i], new
ArrayList<Integer>());
            loc.add(i);
            this.locations.put(words[i], loc);
        }
    }
```

```
    public int shortest(String word1, String word2) {
        ArrayList<Integer> loc1, loc2;
        loc1 = this.locations.get(word1);
        loc2 = this.locations.get(word2);
        int l1 = 0, l2 = 0, minDiff = Integer.MAX_VALUE;
        while (l1 < loc1.size() && l2 < loc2.size()) {
            minDiff = Math.min(minDiff, Math.abs(loc1.get(l1) - loc2.get(l2)));
            if (loc1.get(l1) < loc2.get(l2)) {
                l1++;
            } else {
                l2++;
            }
        }
        return minDiff;
    }
}
/**
 * Your WordDistance object will be instantiated and called as such:
 * WordDistance obj = new WordDistance(words);
 * int param_1 = obj.shortest(word1,word2);
 */
```

35、最短单词距离 3（245）【单词距离（面试题 17.11）】

245. 最短单词距离 III

难度 中等 17 收藏 分享 切换为英文 关注 反馈

给定一个单词列表和两个单词 *word1* 和 *word2*，返回列表中这两个单词之间的最短距离。

word1 和 *word2* 是有可能相同的，并且它们将分别表示为列表中两个独立的单词。

示例:
假设 words = `["practice", "makes", "perfect", "coding", "makes"]`.

输入: *word1* = "makes", *word2* = "coding"
输出: 1

输入: *word1* = "makes", *word2* = "makes"
输出: 3

注意:
你可以假设 *word1* 和 *word2* 都在列表里。

面试题 17.11. 单词距离

难度 中等 10 收藏 分享 切换为英文 关注 反馈

有个内含单词的超大文本文件，给定任意两个单词，找出在这个文件中这两个单词的最短距离(相隔单词数)。如果寻找过程在这个文件中会重复多次，而每次寻找的单词不同，你能对此优化吗?

示例:

输入: words = ["I","am","a","student","from","a","university","in","a","city"], word1 = "a", word2 = "student"
输出: 1

提示:

- words.length <= 100000

https://leetcode-cn.com/problems/shortest-word-distance-iii/solution/javaji-bai-100qu-qiao-de-jie-fa-zai-243de-ji-chu-s/

> 根据题意所给条件，当所给的两个 word 相同时，计算的则是相邻 word 的距离的最小值，则只需要增加一个判断，当两个 word 相同的时候，计算出所有 word 的 index，再进行一次遍历，计算出间隔最小值即可，当然，也可以仅记录上一个 word 的 index 的位置，计算与当前 index 的差值，并在记录下标时就更新 minDist 的值，可以节省 $O(N)$ 的空间开销和一次循环，不过懒得写了哈哈。

```
class Solution {
    public int shortestWordDistance(String[] words, String word1, String word2)
 {
        int minDist = Integer.MAX_VALUE;
        int curDist;
        if(word1.equals(word2)){
            List<Integer> list = new ArrayList<>();
            for (int i = 0; i < words.length; i++) {
                if (words[i].equals(word1)){
                    list.add(i);
                }
            }
            for (int i = 1; i < list.size(); i++) {
                curDist = Math.abs(list.get(i)- list.get(i-1));
                minDist = minDist<curDist?minDist:curDist;
            }
            return minDist;
        }

        int index1=-1;
        int index2=-1;
        for (int i = 0; i < words.length; i++) {
            if (words[i].equals(word1)){
                index1=i;
            }
            if (words[i].equals(word2)){
                index2=i;
```

```
            }
            if(index1!=-1 && index2!=-1){
                curDist = Math.abs(index1-index2);
                minDist = minDist<curDist?minDist:curDist;
            }
        }
        return minDist;
    }
}
```

36、重新排列日志文件（937）

937. 重新排列日志文件

难度 简单 41 收藏 分享 切换为英文 关注 反馈

你有一个日志数组 `logs`。每条日志都是以空格分隔的字串。

对于每条日志，其第一个字为字母与数字混合的 **标识符**，除标识符之外的所有字为这一条日志的 **内容**。

- 除标识符之外，所有字均由小写字母组成的，称为 **字母日志**
- 除标识符之外，所有字均由数字组成的，称为 **数字日志**

题目所用数据保证每个日志在其标识符后面至少有一个字。

请按下述规则将日志重新排序：

- 所有 **字母日志** 都排在 **数字日志** 之前。
- **字母日志** 在内容不同时，忽略标识符后，按内容字母顺序排序；在内容相同时，按标识符排序；
- **数字日志** 应该按原来的顺序排列。

返回日志的最终顺序。

示例：

```
输入：["a1 9 2 3 1","g1 act car","zo4 4 7","ab1 off key dog","a8 act zoo"]
输出：["g1 act car","a8 act zoo","ab1 off key dog","a1 9 2 3 1","zo4 4 7"]
```

提示：

- `0 <= logs.length <= 100`
- `3 <= logs[i].length <= 100`
- `logs[i]` 保证有一个标识符，并且标识符后面有一个字。

https://leetcode-cn.com/problems/reorder-data-in-log-files/solution/zhong-xin-pai-lie-ri-zhi-wen-jian-by-leetcode/

方法：自定义排序

思路和算法

我们按照指定的自定义顺序进行排序，而不是按默认顺序排序。

排序规则如下：

- 字母日志先于数字日志；
- 字母日志按字母数字顺序排列，先按内容排序，再按标识符排序；
- 数字日志的顺序保持不变。

这些想法很容易转化为代码。

```
class Solution {
    public String[] reorderLogFiles(String[] logs) {
        Arrays.sort(logs, (log1, log2) -> {
            String[] split1 = log1.split(" ", 2);
            String[] split2 = log2.split(" ", 2);
            boolean isDigit1 = Character.isDigit(split1[1].charAt(0));
            boolean isDigit2 = Character.isDigit(split2[1].charAt(0));
            if (!isDigit1 && !isDigit2) {
                int cmp = split1[1].compareTo(split2[1]);
                if (cmp != 0) return cmp;
                return split1[0].compareTo(split2[0]);
            }
            return isDigit1 ? (isDigit2 ? 0 : 1) : -1;
        });
        return logs;
    }
}
```

37、字符串切分（lintcode1587）

1587. 字符串切分 中文

现在有一个字符串，首字符代表一级分隔符，分隔不同的键值对key-value；第二个字符代表二级分隔符，分隔key和value；后面的字符串表示待处理的字符串。请给出所有的有效键值对。

样例

样例1

输入:"#:a:3#b:8#c:9"
输出:[["a","3"],["b","8"],["c","9"]]

样例2

输入:"#:aa:3#b:853#:9"
输出:[["aa","3"],["b","853"]]

注意事项

有效键值对即key和value均不为空的键值对。
题目保证分隔符不为字母或数字，待处理的字符串中只包含两种分隔符、小写字母和数字，
且两个一级分隔符中间最多只出现一个二级分隔符。
题目保证所给字符串长度不超过1000。

更多硬核内容就在公众号「yes 的练级攻略」

解题思路 将两个分隔符记录下来，遍历字符串，若遇到二级分隔符则将遍历得到的字符串加入key-value并清空字符串继续遍历，若遇到一级分隔符或是已经结束遍历，则将遍历所得的字符串加入key-value，若key-value中有两个非空字符串则加入答案值 **复杂度分析**

- 时间复杂度 O(n):n为字符串长度，每个字符遍历一次
- 空间复杂度 O(n):需要等大小的数组记录答案

```
public class Solution {
    /**
     * @param str: the string need to be processed
     * @return: all the valid key-value pairs.
     */
    public List<List<String>> StringSeg(char[] str) {
        char separator_1, separator_2;
        List<List<String>> ans = new ArrayList<>();
        List<String> key_value = new ArrayList<>();
        String tmp = "";

        separator_1 = str[0];
        separator_2 = str[1];
        for(int i = 2; i < str.length; i++) {
            if(str[i] == separator_1) {
                if(tmp != ""){
                    key_value.add(tmp);
                }
                tmp = "";
                if(key_value.size() == 2) {
                    List<String> N = new ArrayList<>();
                    N.add(key_value.get(0));
                    N.add(key_value.get(1));
                    ans.add(N);
                }
                key_value.clear();
            }
            else if(str[i] == separator_2) {
                if(tmp != ""){
                    key_value.add(tmp);
                }
                tmp = "";
            }
            else {
                tmp = tmp + str[i];
            }
        }
        if(tmp != "") {
            key_value.add(tmp);
        }
```

禁止商用 公众号：yes的练级攻略

```
            if(key_value.size() == 2) {
                ans.add(key_value);
            }

            return ans;
        }
}
```

38、整数转罗马数字（12）【贪心】

12. 整数转罗马数字

难度 中等 381 收藏 分享 切换为英文 关注 反馈

罗马数字包含以下七种字符： I， V， X， L，C，D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII，即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个整数，将其转为罗马数字。输入确保在 1 到 3999 的范围内。

示例 1:

```
输入: 3
输出: "III"
```

示例 2:

```
输入: 4
输出: "IV"
```

示例 3:

```
输入: 9
输出: "IX"
```

示例 4:

```
输入: 58
输出: "LVIII"
解释: L = 50, V = 5, III = 3.
```

示例 5:

```
输入: 1994
输出: "MCMXCIV"
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

https://leetcode-cn.com/problems/integer-to-roman/solution/tan-xin-suan-fa-by-liweiwei1419/

更多硬核内容就在公众号「yes 的练级攻略」

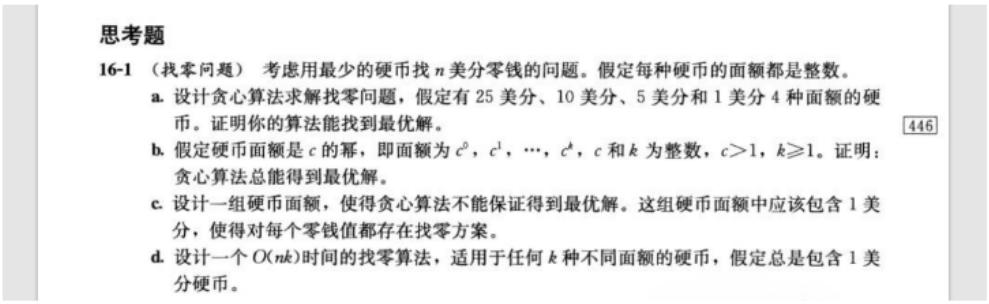
说明：

这道题能想到用贪心算法，是来源于生活中的经验，使用贪心算法有一定取巧的因素：

- 这个问题直觉是贪心去做大概不会有问题；
- 想举出反例推翻贪心算法，但是举不出来；
- 真的用贪心算法去验证，可以通过测评。

如果证明不出来贪心算法的有效性，我个人觉得在绝大多数情况下是没有关系的。有贪心选择的直觉，举不出反例，能编码实现还能过测评，就很棒了。

相同背景的问题是「找零钱问题」，这在《算法导论》（第 3 版）是有介绍的，但是已经不在我能解释的范围内了。可以肯定的一个结论是：这道题使用贪心算法之所有有效，**完全是跟给出的数据有关，换一些数值就不一定可以使用，请大家留意。**



思考题

16-1 （找零问题） 考虑用最少的硬币找 n 美分零钱的问题。假定每种硬币的面额都是整数。

a. 设计贪心算法求解找零问题，假定有 25 美分、10 美分、5 美分和 1 美分 4 种面额的硬币。证明你的算法能找到最优解。

446

b. 假定硬币面额是 c 的幂，即面额为 c^0，c^1，…，c^k，c 和 k 为整数，$c>1$，$k\geqslant 1$。证明：贪心算法总能得到最优解。

c. 设计一组硬币面额，使得贪心算法不能保证得到最优解。这组硬币面额中应该包含 1 美分，使得对每个零钱值都存在找零方案。

d. 设计一个 $O(nk)$时间的找零算法，适用于任何 k 种不同面额的硬币，假定总是包含 1 美分硬币。

（说明：翻译版第 255 页）

生活中的经验：

> 在以前还使用现金购物的时候，**如果我们不想让对方找钱**，付款的时候我们会**尽量选择面值大的纸币**给对方，这样才会使得我们给对方的纸币张数最少，对方点钱的时候也最方便。

本题“整数转罗马数字”也有类似的思想：在表示一个较大整数的时候，“罗马数字”的设计者不会让你都用 1 加起来，我们总是希望写出来的“罗马数字”的个数越少越好，以方便表示，并且这种表示方式还应该是唯一的。

思路分析：

本题中，首先给出“罗马数字”与“阿拉伯数字”的对应关系如下：

罗马数字	阿拉伯数字
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

题目还给了一些特例，我们需要推导出"罗马数字"与"阿拉伯数字"其它的对应关系。为此，从头开始举例子，以发现规律：

阿拉伯数字	转换规则	罗马数字
1	直接看表	I
2	$2 = 1 + 1$，相同数字简单叠加	II
3	$3 = 1 + 1 + 1$，相同数字简单叠加	III
4	题目中说的特例：不能写成 $4 = 1 + 1 + 1 + 1$，4 应该看做 $4 = 5 - 1$	IV
5	直接看表	V
6	$6 = 5 + 1$，大数字在前，小数字在后	VI
7	$7 = 5 + 1 + 1$，大数字在前，小数字在后，相同数字简单叠加	VII
8	$8 = 5 + 1 + 1 + 1$，大数字在前，小数字在后，相同数字简单叠加	VIII
9	题目中说的特例：不能写成 $9 = 5 + 1 + 1 + 1 + 1$，9 应该看做 $9 = 10 - 1$	IX
10	直接看表	X

说明：其实在题目中已经强调了一些特例，出现 4、9、40、90、400、900（4000、9000 不讨论了，题目测试用例中有说，不会超过 3999）的情况比较特殊一些，做的是减法，**把它们也加入到"罗马数字"与阿拉伯数字的对应关系表中，并且按照从大到小的顺序排列。**

罗马数字	阿拉伯数字
M	1000
CM	900
D	500
CD	400
C	100
XC	90
L	50
XL	40
X	10
IX	9
V	5
IV	4
I	1

于是，"将整数转换为罗马数字"的过程，就是用上面这张表中右边的数字作为"加法因子"去分解一个整数，目的是"分解的整数个数"尽可能少，因此，对于这道问题，类似于用最少的纸币凑成一个整数，贪心算法的规则如下：

> 每一步都使用当前较大的罗马数字作为加法因子，最后得到罗马数字表示就是长度最少的。

```
class Solution {
    public String intToRoman(int num) {
        // 把阿拉伯数字与罗马数字可能出现的所有情况和对应关系，放在两个数组中
        // 并且按照阿拉伯数字的大小降序排列，这是贪心选择思想
        int[] nums = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4
, 1};
```

```
        String[] romans = {"M", "CM", "D", "CD", "C", "XC", "L", "XL",
"X", "IX", "V", "IV", "I"};
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 13; i++){
            while (num >= nums[i]){
                sb.append(romans[i]);
                num -= nums[i];
            }
        }
        return sb.toString();
    }
}
```

39、罗马数字转整数（13）

13. 罗马数字转整数

难度 简单 972 收藏 分享 切换为英文 关注 反馈

罗马数字包含以下七种字符: I， V， X， L，C，D 和 M。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II 。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

- I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
- X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
- C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。输入确保在 1 到 3999 的范围内。

示例 1:

```
输入: "III"
输出: 3
```

示例 2:

```
输入: "IV"
输出: 4
```

示例 3:

```
输入: "IX"
输出: 9
```

示例 4:

```
输入: "LVIII"
输出: 58
解释: L = 50, V= 5, III = 3.
```

示例 5:

```
输入: "MCMXCIV"
输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4.
```

https://leetcode-cn.com/problems/roman-to-integer/solution/yong-shi-9993nei-cun-9873jian-dan-jie-fa-by-donesp/

按照题目的描述，可以总结如下规则：

1. 罗马数字由 I, V, X, L, C, D, M 构成；
2. 当小值在大值的左边，则减小值，如 IV=5-1=4；
3. 当小值在大值的右边，则加小值，如 VI=5+1=6；
4. 由上可知，右值永远为正，因此最后一位必然为正。

一言蔽之，**把一个小值放在大值的左边，就是做减法，否则为加法。**

X	X	V	I	I	27
10	10	5	1	1	27

X	X	I	V	24
10	10	-1	5	24

在代码实现上，可以往后看多一位，对比当前位与后一位的大小关系，从而确定当前位是加还是减法。当没有下一位时，做加法即可。

也可保留当前位的值，当遍历到下一位的时，对比保留值与遍历位的大小关系，再确定保留值为加还是减。最后一位做加法即可。

方法 1：根据罗马数字规律求解

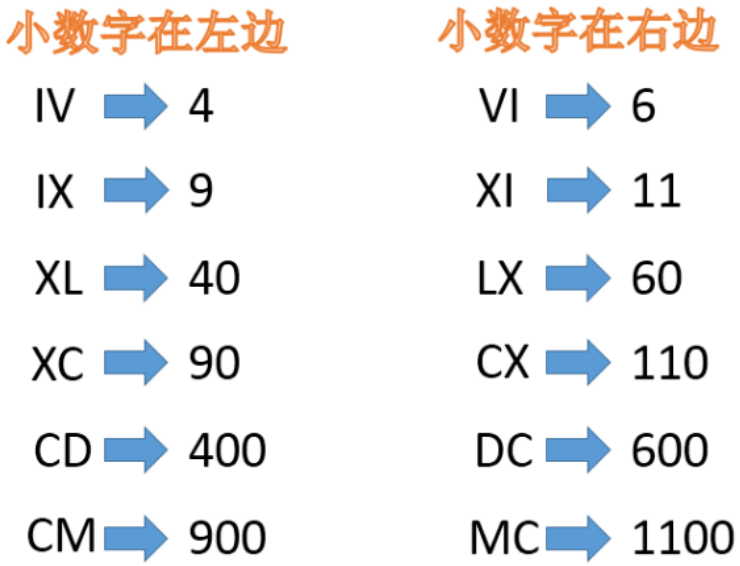
```
class Solution {
    public int romanToInt(String s) {
        int sum = 0;
        int preNum = getValue(s.charAt(0));
        for (int i = 1; i < s.length(); i++){
            int curNum = getValue(s.charAt(i));
            if (curNum > preNum){
                sum -= preNum;
            } else {
                sum += preNum;
            }
            preNum = curNum;
        }
        sum += preNum;
        return sum;
    }
    private int getValue(char ch){
        switch (ch) {
            case 'I' : return 1;
            case 'V' : return 5;
            case 'X' : return 10;
            case 'L' : return 50;
```

```
                case 'C' : return 100;
                case 'D' : return 500;
                case 'M' : return 1000;
                default : return 0;
            }
        }
}
```

https://leetcode-cn.com/problems/roman-to-integer/solution/zi-ding-yi-hashchao-guo-9997-duo-chong-jie-fa-pei-/

题解一

先从一个简单的入手，我们来看看数字 I 和数字V，他们两的组合在一起会有两种结果：
IV 表示 4 ，即左边的I是 1 ，右边的V是 5 ，右边数字减去左边数字也就是 5-1 得到 4 。
VI 表示 6 ，即左边的V是 5 ，右边的I是 1 ，左边数字加上右边数字，也就是 5+1 得到 6 。
也就是小的数字在左边就是两数字相减，小的数字在右边就是两数相加。
我们不妨把不同字母的两两组合都列出来观察一下，按照题目的意思，两两组合其实有很多限制，像IM就不是一个合法的组合，小数字在左边的只有6种情况，我们把这六种情况都列出来，同时再列出小数字在右边的情况对比一下。



更多硬核内容就在公众号「yes 的练级攻略」

通过这个图你就可以发现，这其实就是一个加减的问题。

那么对于这题，我们有两种实现方式

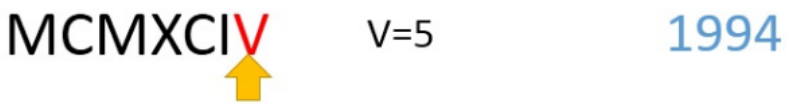
第一种方式，每次只匹配一个然后不断累加，如果前一个字母的值**小于**后一个字母的值，这次就不是累加了，而是做一个减法，也就是将当前字母的值减掉，比如**CM**，我们先减去 `100` ，再加上 `1000` 这样就得到了 `900` 。

详细执行过程见下图：

☆ 收藏

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」



第二种，我们可以给每个字母，以及6种不同的两两组合都建一个哈希映射，
也就是将I IV V IX X XL L XC C CD D CM M 这些字母都放到哈希表中，如I=5，IV=4，这样。
具体执行过程如下图所示:

然后每次遍历的时候尽可能多的匹配，先匹配2个，如果两个不行就匹配1。
像上面那个罗马数字中，MC就找不到对应的匹配，于是就只匹配一个M，第二次发现CM在哈希表中，于是一次匹配了两个。

方法 2：普通的 hashMap

```
class Solution {
    public int romanToInt(String s) {
        if(s == null) return 0;
        //建立罗马数字 到 阿拉伯数字的映射关系
        Map<Character,Integer> str2int = new HashMap<Character,Integer>
();
        str2int.put('I',1);
        str2int.put('V',5);
        str2int.put('X',10);
        str2int.put('L',50);
        str2int.put('C',100);
        str2int.put('D',500);
        str2int.put('M',1000);
        int ans = 0;
        //每次我们都匹配一个字母，注意因为要获取 i+1 位置的字母，所以循环终止
条件
        //是字符串的最大长度-1
        for(int i = 0; i < s.length() - 1; ++i) {
            if( str2int.get(s.charAt(i)) < str2int.get(s.charAt(i+1)) )
 {
```

```
                ans -= str2int.get(s.charAt(i));
            }
            else {
                ans += str2int.get(s.charAt(i));
            }
        }
        //最后不要忘记收尾操作，把遗留的字母也加上
        ans += str2int.get(s.charAt(s.length() - 1));
        return ans;
    }
}
```

方法 3：自定义 hashMap

题解一中根据罗马数字获取对应的整数是这么做的(以java为例)

```
str2int.get(s.charAt(i))
```

这里是获取字符串中第 `i` 个位置上的字符，返回的是**char**类型，因为java的Map键值对都必须是对象类型，所以**char**类型会被自动包装成对象类型，去map中查找对应的值。
值的类型是**Integer**，也是对象类型，之后**Integer**对象中的值会被自动转换成整型也就是**int**，这一切都是虚拟机帮我们做的，我们无需关心。

这就是相当于来回倒腾，当然虚拟机可以在运行期间自动优化，提高一些性能，但也免不了来回倒腾。
另外语言本身提供的哈希表虽然很强大，但是我们这里要的其实很简单，就是给出几个字符，然后能返回出对应的数字就行了，语言本身提供的哈希表显得有点重了。

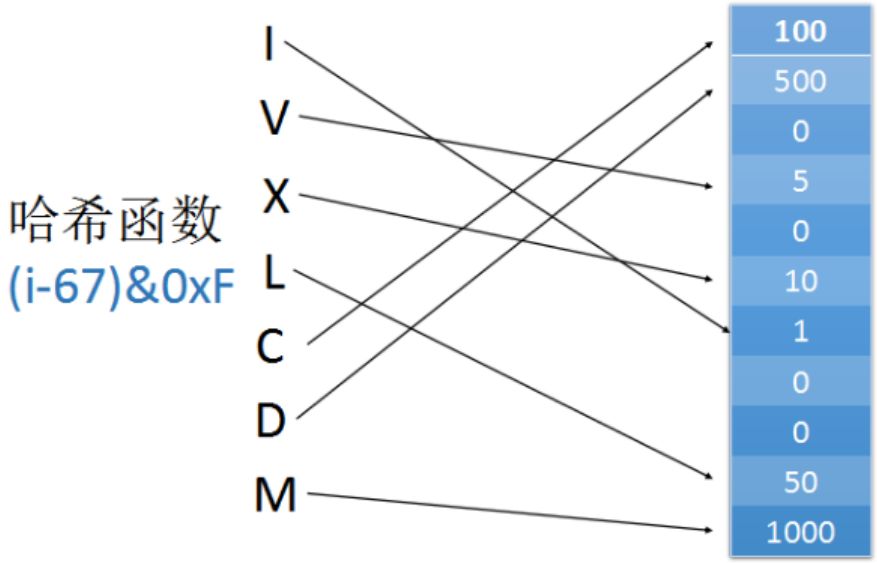
现在我们换一个思路，不再用语言提供的哈希表，我们自己实现一个，以此来加速查找过程。

首先我们得想想怎么根据一个字母获取到对应的整数，我们可以用字母本身做文章，每个字母在实际存储的时，存的是对应的ASCII码

ASCII码	十进制值	ASCII码	十进制值
A	65	N	78
B	66	O	79
C	67	P	80
D	68	Q	81
E	69	R	82
F	70	S	83
G	71	T	84
H	72	U	85
I	73	V	86
J	74	W	87
K	75	X	88
L	76	Y	89
M	77	Z	90

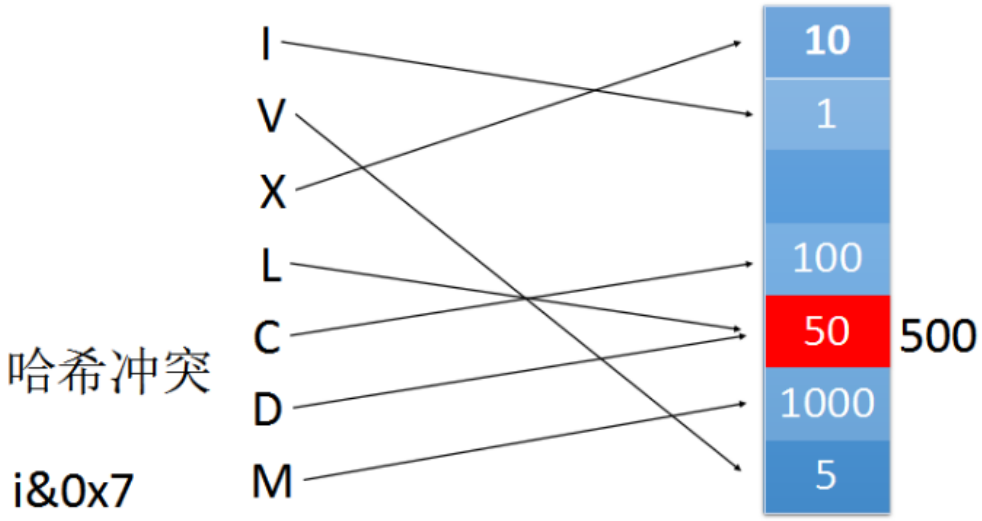
上图中罗马数字相关的几个字母都被我标记出来了，比如I的ASCII码是73，我们拿到I也就知道实际存储的是73这个十进制值。

之后我们用自己实现的哈希函数来将 73 映射成 1



上图就是一个哈希映射规则，I是73，经过(73-67)&0xF结果是 6 ，于是就去索引下标 6 的位置去查找，得到的值是 1 。
这样就建立了I -> 1的映射关系。
其他几个字母也是类似的。

但是，罗马数字只有7个，这里的数组大小为什么是11。
这是因为防止哈希冲突导致的，7个字母最理想情况是映射到长度为7的数组中，字母和数字一一对应。
但是这很难，比如下面这样的哈希函数就会冲突了，因为数组太小了，就容易出现冲突。



最终我们用上面的哈希函数，以及大小为10的整数数组，替换掉原先的哈希表，就完成了自定义的映射规则，代码实现跟解法一是一样的，只是映射规则变了。

```
class Solution {
    public int romanToInt(String s) {
        if(s == null) return 0;
        int[] str2int = {100,500,0,5,0,10,1,0,0,50,1000};
        int ans = 0;
        for(int i = 0; i < s.length() - 1; ++i) {
            //自定义的哈希映射规则，其他操作跟题解一中的代码都一样
```

```
            if( str2int[(s.charAt(i)-67) & 0xF] < str2int[(s.charAt(i+1
)-67) & 0xF] ) {
                ans -= str2int[(s.charAt(i)-67) & 0xF];
            }
            else {
                ans += str2int[(s.charAt(i)-67) & 0xF];
            }
        }
        ans += str2int[(s.charAt(s.length()-1)-67) & 0xF];
        return ans;
    }
}
```

40、整数转换英文表示（273）【整数的英语表示（面试题 16.08）】

273. 整数转换英文表示

难度 困难 92 收藏 分享 切换为英文 关注 反馈

将非负整数转换为其对应的英文表示。可以保证给定输入小于 $2^{31} - 1$ 。

示例 1:

```
输入：123
输出："One Hundred Twenty Three"
```

示例 2:

```
输入：12345
输出："Twelve Thousand Three Hundred Forty Five"
```

示例 3:

```
输入：1234567
输出："One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"
```

示例 4:

```
输入：1234567891
输出："One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand
Eight Hundred Ninety One"
```

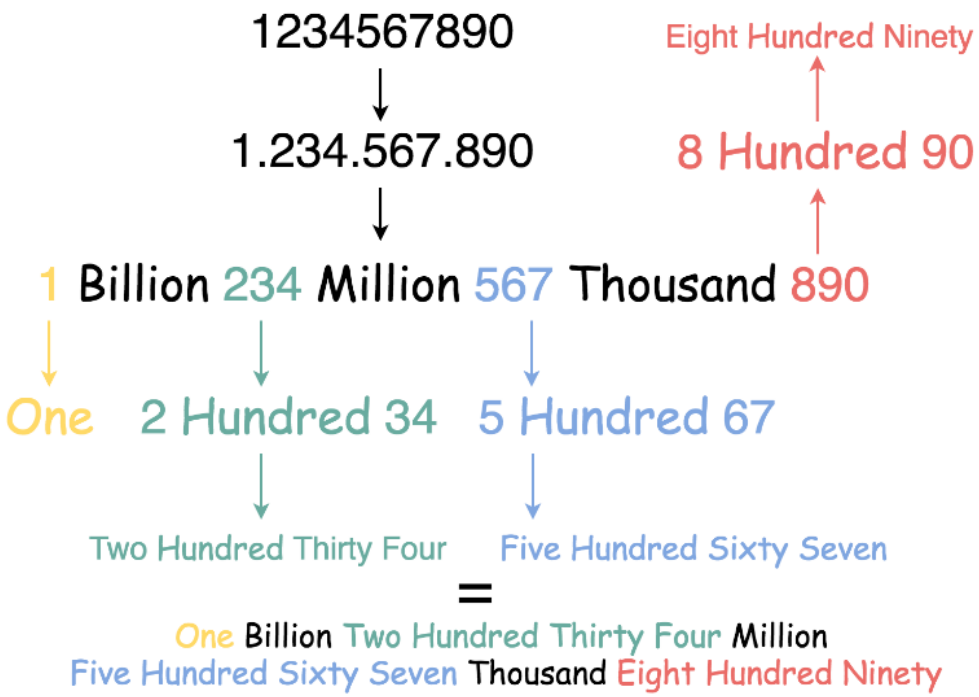
https://leetcode-cn.com/problems/integer-to-english-words/solution/zheng-shu-zhuan-huan-ying-wen-biao-shi-by-leetcode/

方法一：分治

我们将这个问题分解成一系列子问题。例如，对于数字 `1234567890`，我们将它从低位开始每三个分成一组，得到 `1,234,567,890`，它的英文表示为 `1 Billion 234 Million 567 Thousand 890`。这样我们就将原问题分解成若干个三位整数转换为英文表示的问题了。

接下来，我们可以继续将三位整数分解，例如数字 `234` 可以分别成百位 `2` 和十位个位 `34`，它的英文表示为 `2 Hundred 34`。这样我们继续将原问题分解成一位整数和两位整数的英文表示。其中一位整数的表示是很容易的，而两位整数中除了 `10` 到 `19` 以外，其余整数的的表示可以分解成两个一位整数的表示，这样问题就被圆满地解决了。下面的幻灯片中给出了 `1234567890` 得到英文表示的例子。

更多硬核内容就在公众号「yes 的练级攻略」



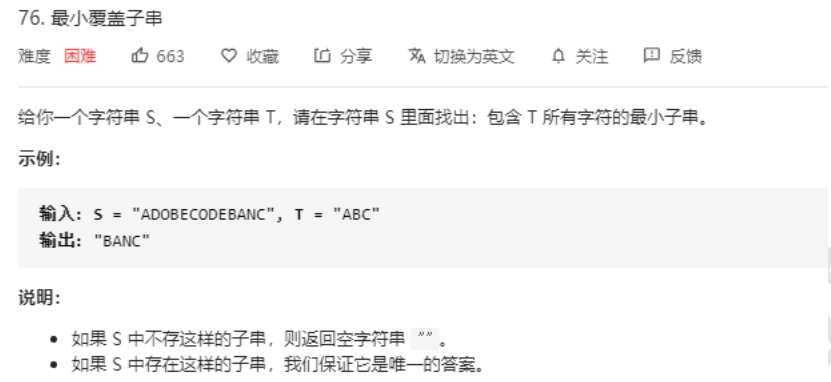
复杂度分析

- 时间复杂度：$O(N)$。其中 N 是输入整数的长度。由于输出的英文表示长度和输入整数的长度是成正比的，因此时间复杂度为 $O(N)$。
- 空间复杂度：$O(1)$。

```
class Solution {
    private final String[] THOUSAND = {"", "Thousand", "Million", "Bill
ion"};
    private final String[] LESS_THAN_TWENTY = {"", "One", "Two", "Three
", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "T
welve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eig
hteen", "Nineteen"};
    private final String[] HUNDRED = {"", "", "Twenty", "Thirty", "Fort
y", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
    public String numberToWords(int num) {
        if(num == 0) return "Zero";

        StringBuilder sb = new StringBuilder();
        int index = 0;
        while(num > 0) {
            if(num % 1000 != 0) {
                StringBuilder tmp = new StringBuilder();
                helper(num % 1000, tmp);
                sb.insert(0, tmp.append(THOUSAND[index]).append(" "));
            }
            index++;
            num /= 1000;
```

```
        }
        return sb.toString().trim();
    }

    private void helper(int num, StringBuilder tmp) {
        if(num == 0) return;
        if(num < 20) {
            tmp.append(LESS_THAN_TWENTY[num]).append(" ");
        }else if(num < 100) {
            tmp.append(HUNDRED[num / 10]).append(" ");
            helper(num % 10, tmp);
        }else {
            tmp.append(LESS_THAN_TWENTY[num / 100]).append(" Hundred").
append(" ");
            helper(num % 100, tmp);
        }
    }
}
```

41、最小覆盖子串（76）



76. 最小覆盖子串

难度 困难 663 收藏 分享 切换为英文 关注 反馈

给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字符的最小子串。

示例：

输入： S = "ADOBECODEBANC", T = "ABC"
输出： "BANC"

说明：

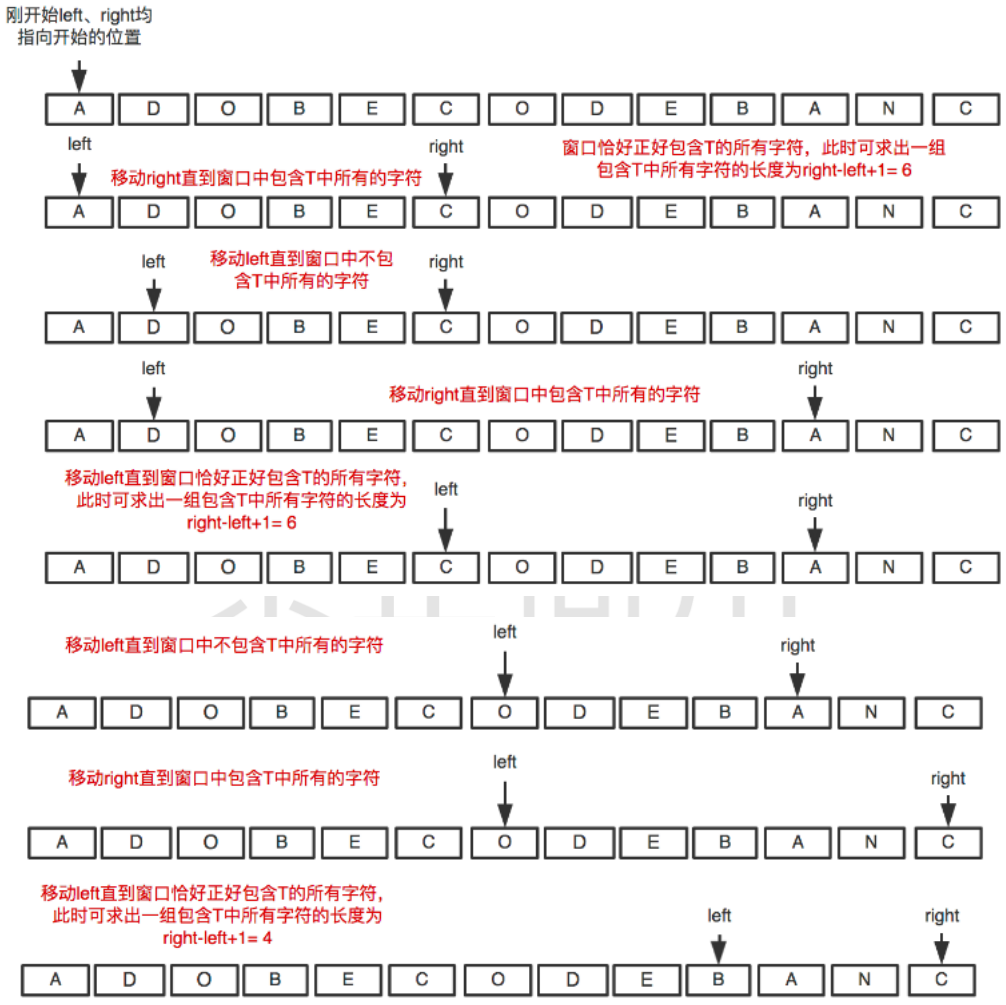
- 如果 S 中不存这样的子串，则返回空字符串 ""。
- 如果 S 中存在这样的子串，我们保证它是唯一的答案。

https://leetcode-cn.com/problems/minimum-window-substring/solution/hua-dong-chuang-kou-ji-bai-liao-100de-javayong-hu-/

分析

新建一个needs[128] 用来统计t中每个字符出现次数，新建一个 window[128]用来统计滑动窗口中每个字符出现次数。首先统计出T中每个字母出现的次数. 新建两个变量left和right分别用来表示滑动窗口的左边和右边。新建一个变量count来表示目前窗口中已经找到了多少个字符。

以S = "ADOBECODEBANC", T = "ABC"为例，然后按照如图所示的规律滑动窗口。



即可得出最小子串的长度为4，最小子串为"BANC"。

76 具体代码

```
class Solution {
    public String minWindow(String s, String t) {
        if (s == null || s.length() == 0 || t == null || t.length() ==
0 || s.length() < t.length()) return "";
        int[] cnt = new int[128];
        int[] window = new int[128];
        for (int i = 0; i < t.length(); ++i){
            cnt[t.charAt(i)]++;
        }
        int left = 0, right = 0;
        String res = "";
        int count = 0;//目前有多少个字符
        int minLen = s.length() + 1;//用来记录最短需要多少个字符。
```

```
        while (right < s.length()){
            char ch = s.charAt(right);
            window[ch]++;
            if (cnt[ch] > 0 && cnt[ch] >= window[ch]){
                count++;
            }
            //移动到不满足条件为止
            while (count == t.length()){
                ch = s.charAt(left);
                if (cnt[ch] > 0 && cnt[ch] >= window[ch]) count--;
                if (right - left + 1 < minLen) {
                    minLen = right - left + 1;
                    res = s.substring(left, right + 1);
                }
                window[ch]--;
                left++;
            }
            right++;
        }
        return res;
    }
}
```

42、替换空格（剑指 5）

剑指 Offer 05. 替换空格

难度 简单 31 收藏 分享 切换为英文 关注 反馈

请实现一个函数，把字符串 `s` 中的每个空格替换成"%20"。

示例 1：

输入： `s = "We are happy."`
输出： `"We%20are%20happy."`

限制：

`0 <= s 的长度 <= 10000`

https://leetcode-cn.com/problems/ti-huan-kong-ge-lcof/solution/mian-shi-ti-05-ti-huan-kong-ge-ji-jian-qing-xi-tu-/

方法 1：StringBuffer

```
class Solution {
    public String replaceSpace(String s) {
        StringBuilder res = new StringBuilder();
        for(Character c : s.toCharArray())
        {
            if(c == ' ') res.append("%20");
```

```
            else res.append(c);
        }
        return res.toString();
    }
}
```

https://leetcode-cn.com/problems/ti-huan-kong-ge-lcof/solution/mian-shi-ti-05-ti-huan-kong-ge-by-leetcode-solutio/

方法 2：char 数组

```
class Solution {
    public String replaceSpace(String s) {
        int len = s.length();
        if (s == null || s.length() == 0) return s;
        char[] ch = new char[len * 3];
        int j = 0;
        for (int i = 0; i < len; i++) {
            char c = s.charAt(i);
            if (c == ' ') {
                ch[j++] = '%';
                ch[j++] = '2';
                ch[j++] = '0';
            } else ch[j++] = c;
        }
        return new String(ch, 0, j);
    }
}
```

43、URL 化（面试题 01.03）【上一题代码改一改】

面试题 01.03. URL化

难度 简单　18　收藏　分享　切换为英文　关注　反馈

URL化。编写一种方法，将字符串中的空格全部替换为 %20 。假定该字符串尾部有足够的空间存放新增字符，并且知道字符串的"真实"长度。（注：用 Java 实现的话，请使用字符数组实现，以便直接在数组上操作。）

示例1:

```
输入: "Mr John Smith    ", 13
输出: "Mr%20John%20Smith"
```

示例2:

```
输入: "               ", 5
输出: "%20%20%20%20%20"
```

提示:

1. 字符串长度在[0, 500000]范围内。

```
class Solution {
    public String replaceSpaces(String S, int length) {
        char[] cs = new char[length * 3];
        int j = 0;
```

```
        for(int i = 0; i < length; i++){
            char c = S.charAt(i);
            if(c == ' '){
                cs[j++] = '%';
                cs[j++] = '2';
                cs[j++] = '0';
            } else {
                cs[j++] = c;
            }
        }
        return new String(cs, 0, j);
    }
}
```

44、无重复字符的最长子串（3）【最长无重复字符的子串（lintcode384）】

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:

输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
请注意，你的答案必须是 **子串** 的长度，"pwke" 是一个*子序列*，不是子串。

略

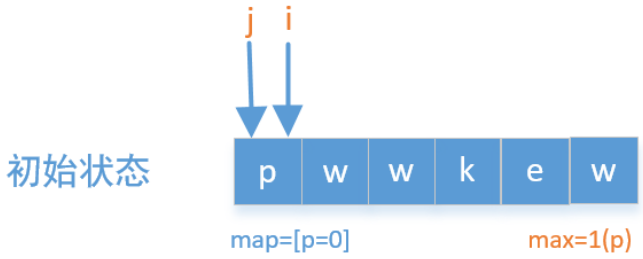
HashMap 方法：使用两个指针，一个 i 一个 j，最开始的时候 i 和 j 指向第一个元素，然后 i 往后移，把扫描过的元素都放到 map 中，如果 i 扫描过多元素没有重复的就一直往后移，顺便记录一下最大值 max，如果 i 扫描过的元素有重复的，就改变 j 的位置，我们就以 pwwkew 为例画个图看一下

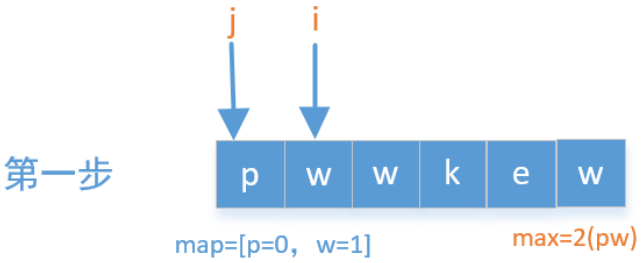
链接：

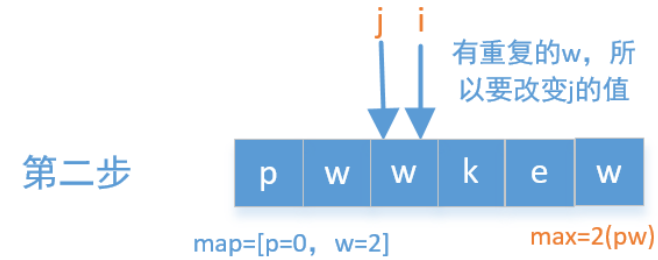
https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/solution/java-de-6chong-jie-fa-by-sdwwld/

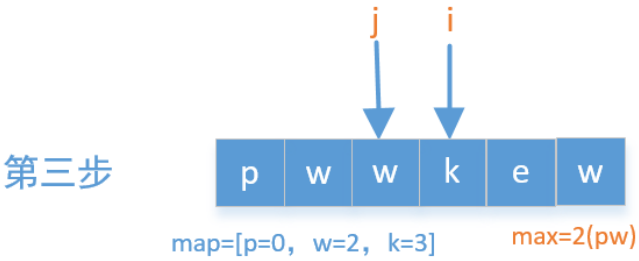
图解：

更多硬核内容就在公众号「yes 的练级攻略」

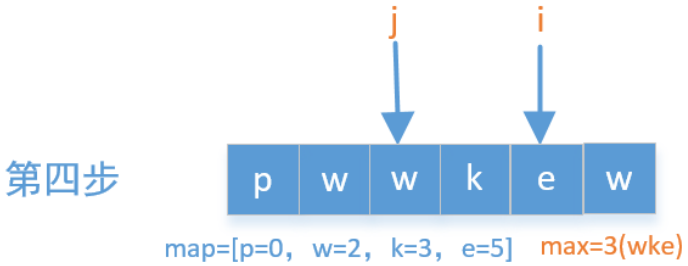


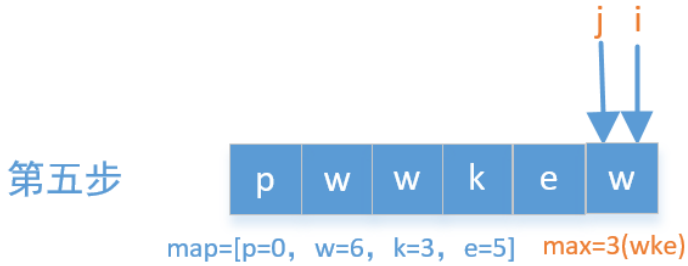






更多硬核内容就在公众号「yes 的练级攻略」





方法三：双指针 + 哈希表

> 本质上与方法一类似，不同点在于左边界 i 的定义。

- **哈希表** dic **统计：** 指针 j 遍历字符 s ，哈希表统计字符 $s[j]$ **最后一次出现的索引** 。
- **更新左指针** i **：** 根据上轮左指针 i 和 $dic[s[j]]$ ，每轮更新左边界 i ，保证区间 $[i+1, j]$ 内无重复字符且最大。

$$i = \max(dic[s[j]], i)$$

- **更新结果** res **：** 取上轮 res 和本轮双指针区间 $[i+1, j]$ 的宽度（即 $j - i$ ）中的最大值。

$$res = \max(res, j - i)$$

复杂度分析：

- **时间复杂度** $O(N)$ **：** 其中 N 为字符串长度，动态规划需遍历计算 dp 列表。
- **空间复杂度** $O(1)$ **：** 字符的 ASCII 码范围为 $0 \sim 127$ ，哈希表 dic 最多使用 $O(128) = O(1)$ 大小的额外空间。

yes的练级攻略

方法 1：双指针加哈希表（9ms）

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s.length() == 0) {
            return 0;
        }
        HashMap<Character, Integer> map = new HashMap<>();
        int max = 0;
        for (int i = 0, j = 0; i < s.length(); ++i) {
            if (map.containsKey(s.charAt(i))){
                j = Math.max(j,map.get(s.charAt(i)) + 1);
            }
            map.put(s.charAt(i),i);
            max = Math.max(max, i - j + 1);
        }
        return max;
```

```
    }
}
```

方法 2：直接用数组（3ms）

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s.length() == 0) return 0;
        int[] idx = new int[128];
        Arrays.fill(idx, -1);
        int max = 0;
        for (int i = 0, j = 0; i < s.length(); ++i) {
            j = Math.max(j, idx[s.charAt(i)] + 1);
            idx[s.charAt(i)] = i;
            if (max < i - j + 1) max = i - j + 1;
            //max = Math.max(max, i - j + 1);//用 Math.max 更慢
        }
        return max;
    }
}
```

输出具体子串的可运行版本

```
import java.util.*;
public class LongestSubstring {
    //求最长子串方法 1-hashMap
    public static String longestSubstring(String s){
        if (s.length() == 0) return s;
        HashMap<Character, Integer> map = new HashMap<>();
        int max = 0;
        String res = "";
        for (int i = 0, j = 0; i < s.length(); ++i){
            if (map.containsKey(s.charAt(i))){
                j = Math.max(j, map.get(s.charAt(i)) + 1);
            }
            map.put(s.charAt(i), i);
            if (max < i - j + 1) {
                max = i - j + 1;
                res = s.substring(j, i + 1);
            }
        }
        return res;
    }
    //求最长子串方法 2-char 数组
    public static String longestSubstring2(String s){
        if (s.length() == 0) return s;
        int[] idx = new int[128];
        Arrays.fill(idx, -1);
```

```
        int max = 0;
        String res = "";
        for (int i = 0, j = 0; i < s.length(); ++i){
            j = Math.max(j, idx[s.charAt(i)] + 1);
            idx[s.charAt(i)] = i;
            if (max < i - j + 1) {
                max = i - j + 1;
                res = s.substring(j, i + 1);
            }
        }
        return res;
    }
    //求所有的最长子串方法 1-hashMap
    public static List<String> longestSubstringList(String s){
        List<String> res = new ArrayList<>();
        if (s.length() == 0) return res;
        HashMap<Character, Integer> map = new HashMap<>();
        int max = 0;
        for (int i = 0, j = 0; i < s.length(); ++i){
            if (map.containsKey(s.charAt(i))){
                j = Math.max(j, map.get(s.charAt(i)) + 1);
            }
            map.put(s.charAt(i), i);
            if (max < i - j + 1) {
                max = i - j + 1;
                res.clear();
                res.add(s.substring(j, i+1));
            } else if (max == i - j + 1){
                res.add(s.substring(j, i+1));
            }
        }
        return res;
    }
    //求所有的最长子串方法 2-char 数组
    public static List<String> longestSubstringList2(String s){
        List<String> res = new ArrayList<>();
        if (s.length() == 0) return res;
        int[] idx = new int[128];
        Arrays.fill(idx, -1);
        int max = 0;
        for (int i = 0, j = 0; i < s.length(); ++i){
            j = Math.max(j, idx[s.charAt(i)] + 1);
            idx[s.charAt(i)] = i;
            if (max < i - j + 1) {
```

```
                    max = i - j + 1;
                    res.clear();
                    res.add(s.substring(j, i+1));
                }else if (max == i - j + 1){
                    res.add(s.substring(j, i+1));
                }
            }
            return res;
        }
        public static void main(String[] args){
            Scanner sc = new Scanner(System.in);
            String s = sc.nextLine();
            System.out.println("求最长子串方法 1-hashMap");
            System.out.println(longestSubstring(s));
            System.out.println("求最长子串方法 2-char 数组");
            System.out.println(longestSubstring2(s));
            System.out.println("求所有的最长子串方法 1-hashMap");
            List<String> strs1 = longestSubstringList(s);
            for (int i = 0; i < strs1.size(); i++){
                System.out.println(strs1.get(i));
            }
            System.out.println("求所有的最长子串方法 2-char 数组");
            List<String> strs2 = longestSubstringList2(s);
            for (int i = 0; i < strs2.size(); i++){
                System.out.println(strs2.get(i));
            }
        }
}
```

45、最长重复子串（1044）

更多硬核内容就在公众号「yes 的练级攻略」

1044. 最长重复子串

难度 困难 68 收藏 分享 切换为英文 关注 反馈

给出一个字符串 `S`，考虑其所有**重复子串**（`S` 的连续子串，出现两次或多次，可能会有重叠）。

返回**任何具有最长可能长度的重复子串**。（如果 `S` 不含重复子串，那么答案为 `""`。）

示例 1：

```
输入："banana"
输出："ana"
```

示例 2：

```
输入："abcd"
输出：""
```

提示：

1. `2 <= S.length <= 10^5`
2. `S` 由小写英文字母组成。

禁止商用

https://leetcode-cn.com/problems/longest-duplicate-substring/solution/zui-chang-zhong-fu-zi-chuan-by-leetcode/

方法一：二分查找 + Rabin-Karp 字符串编码

分析

我们可以把这个问题分解成两个子问题：

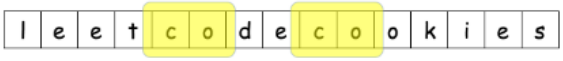
1. 从 `1` 到 `N` 中选取子串的长度 `L`；
2. 检查字符串中是否存在长度为 `L` 的重复子串。

子任务一：二分查找

解决子问题一的最简单的方法是，我们从 `L = N - 1` 开始，依次递减选取子串的长度，并进行判断。如果发现存在长度为 `L` 的重复子串，就表示 `L` 是最长的可能长度。

但我们发现，如果字符串中存在长度为 `L` 的重复子串，那么一定存在长度为 `L0 < L` 的重复子串（选取长度为 `L` 的重复子串的某个长度为 `L0` 的子串即可），因此我们可以使用二分查找的方法，找到最大的 `L`。

☆收藏



Binary search by string length:

left = 1, right = 15, check substrings of length 8 --> no duplicates.

left = 1, right = 8, check substring of length 4 --> no duplicates.

left = 1, right = 4, check substrings of length 2 --> yeees!

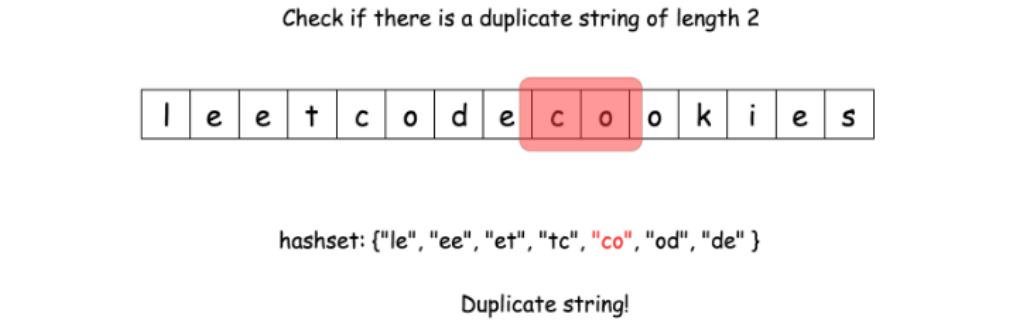
left = 3, right = 4, check substrings of length 3 --> no duplicates.

left = 3, right = 3 --> stop. Return duplicate string of length 3 - 1 = 2.

子任务二：Rabin-Karp 字符串编码

我们可以使用 `Rabin-Karp` 算法将字符串进行编码，这样只要有两个编码相同，就说明存在重复子串。对于选取的长度 `L`：

- 使用长度为 `L` 的滑动窗口在长度为 `N` 的字符串上从左向右滑动；
- 检查当前处于滑动窗口中的子串的编码是否已经出现过（用一个集合存储已经出现过的编码）；
  - 若已经出现过，就说明找到了长度为 `L` 的重复子串；
  - 若没有出现过，就把当前子串的编码加入到集合中。



可以看出，`Rabin-Karp` 字符串编码的本质是对字符串进行哈希，将字符串之间的比较转化为编码之间的比较。接下来的问题是，在滑动窗口从左向右滑动时，如何快速计算出当前子串的编码呢？如果需要在 $O(L)$ 的时间内算出编码，这种方法就没有意义了，因为这个直接进行字符串比较需要的时间相同。

为了能够快速计算出字符串编码，我们可以将字符串看成一个 26 进制的数（因为字符串中仅包含小写字母），它对应的 10 进制的值就是字符串的编码值。首先将字符转换为 26 进制中的 0-25，即通过 `arr[i] = (int)S.charAt(i) - (int)'a'`，可以将字符串 `abcd` 转换为 `[0, 1, 2, 3]`，它对应的 10 进制值为：

$$h_0 = 0 \times 26^3 + 1 \times 26^2 + 2 \times 26^1 + 3 \times 26^0$$

我们将这个表达式写得更通用一些，设 c_i 为字符串中第 `i` 个字符对应的数字，$a = 26$ 为字符串的进制，那么有：

$$\begin{aligned} h_0 &= c_0a^{L-1} + c_1a^{L-2} + ... + c_ia^{L-1-i} + ... + c_{L-1}a^1 + c_La^0 \\ &= \sum_{i=0}^{L-1} c_ia^{L-1-i} \end{aligned}$$

当我们向右移动滑动窗口时，例如从 `abcd` 变成 `bcde`，此时字符串对应的值从 `[0, 1, 2, 3]` 变成 `[1, 2, 3, 4]`，移除了最高位的 `0`，并且在最低位添加了 `4`，那么我们可以快速地计算出新的字符串的编码：

$$h_1 = (h_0 - 0 \times 26^3) \times 26 + 4 \times 26^0$$

更加通用的写法是：

$$h_1 = (h_0a - c_0a^L) + c_{L+1}$$

这样，我们只需要 $O(L)$ 的时间复杂度计算出最左侧子串的编码，这个 $O(L)$ 和滑动窗口的循环是独立的。在滑动窗口向右滑动时，计算新的子串的编码的时间复杂度仅为 $O(1)$。

最后一个需要解决的问题是，在实际的编码计算中，a^L 的值可能会非常大，在 C++ 和 Java 语言中，会导致整数的上溢出。一般的解决方法时，对编码值进行取模，将编码控制在一定的范围内，防止溢出，即 `h = h % modulus`。根据 生日悖论，模数一般需要和被编码的信息数量的平方根的数量级相同，在本题中，相同长度的子串的数量不会超过 N 个，因此选取模数是 2^{32}（无符号整型数的最大值）是足够的。在 Java 中可以用如下的代码实现取模：

```
h = (h * a - nums[start - 1] * aL % modulus + modulus) % modulus;
h = (h + nums[start + L - 1]) % modulus;
```

而在 Python 中，整型数没有最大值，因此可以在运算的最后再取模：

```
h = (h * a - nums[start - 1] * aL + nums[start + L - 1]) % modulus
```

在解决算法题时，我们只要判断两个编码是否相同，就表示它们对应的字符串是否相同。但在实际的应用场景中，会出现字符串不同但编码相同的情况，因此在实际场景中使用 `Rabin-Karp` 字符串编码时，推荐在编码相同时再对字符串进行比较，防止出现错误。

复杂度分析

- 时间复杂度：$O(N \log N)$，二分查找的时间复杂度为 $O(\log N)$，Rabin-Karp 字符串编码的时间复杂度为 $O(N)$。
- 空间复杂度：$O(N)$，用来存储字符串编码的集合。

呜呜呜 （编辑过）2020-08-19

- 将变量名人性化了一下
- 如果没有硬记住modulus，在写一次还是写不出来。

> 模数一般需要和被编码的信息数量的平方根的数量级相同，在本题中，相同长度的子串的数量不会超过 N 个，因此选取模数是 2^{32}（无符号整型数的最大值）是足够的。2 <= S.length <= 10^5

> 因为小写字母26个，所以系数使用26

```java
class Solution {
    public String longestDupSubstring(String str) {
        int[] arr = new int[str.length()];
        for(int i=0; i<str.length();i++) {
            arr[i] = str.charAt(i) - 'a';
        }
        // 二分法+RK 法
        int left = 0;
        int right = arr.length;
        while (left < right) {
            int mid = left + (right - left + 1) / 2;
            if(checkRK(arr, mid) > 0) {
                left = mid;
            } else {
                right = mid-1;
            }
        }
        int start = checkRK(arr, left);
        if(start == -1) return "";
        else return str.substring(start, start + left);
    }

    final long modulus = (long)Math.pow(2,32); // 取模的模值
    final int a = 26; // 系数

    private int checkRK(int[] arr, int length) {
        long curHash = 0;
```

```
        HashSet<Long> seen = new HashSet<>();
        // 首先统计 0----length-1 这个长为 length 的的哈希值
        for (int i=0; i<length; i++) curHash = (curHash*a+arr[i]) % modulus;
        seen.add(curHash);
        // aL 代表 a 的 L 次方
        long aL = 1;
        for (int i=1; i<=length; i++) aL = (aL*a) % modulus;
        // 统计
(1----length) (2------length+1) ..... (arr.length-length -----arr.length
-1) 的哈希值
        for(int start=1; start<=arr.length-length; start++) {
            curHash = (curHash * a - arr[start-1]*aL%modulus + modulus) % modul
us;
            curHash = curHash + arr[start+length-1] % modulus;
            if(seen.contains(curHash)) return start;
            seen.add(curHash);
        }
        return -1;
    }
}
```

46、重复叠加字符串匹配（686）

686. 重复叠加字符串匹配

难度 中等 104 收藏 分享 切换为英文 关注 反馈

给定两个字符串 A 和 B, 寻找重复叠加字符串A的最小次数，使得字符串B成为叠加后的字符串A的子串，如果不存在则返回 -1。

举个例子，A = "abcd"，B = "cdabcdab"。

答案为 3， 因为 A 重复叠加三遍后为 “abcdabcdabcd”，此时 B 是其子串；A 重复叠加两遍后为"abcdabcd"，B 并不是其子串。

注意:

A 与 B 字符串的长度在1和10000区间范围内。

https://leetcode-cn.com/problems/repeated-string-match/solution/zhong-fu-die-jia-zi-fu-chuan-pi-pei-by-leetcode/

方法一：

这个问题可以概括为"最小的 `k` 是什么，`B` 是 `A*k` 的子串？"我们可以试试每一个 `k`。

算法：

- 假设我们写了 `S = A+A+A+...` ,如果 `B` 是 `S` 的子串，我们只需要检查一些 `S[0:], S[1:], ..., S[len(A) - 1:]` 是否以 `B` 开头，因为 `S` 的长度足以包含 `B`。
- 现在，假设 `q` 是 `len(B)<=len(A*q)` 的最小数。我们只需要检查 `B` 是 `A*q` 的子串还是 `A*(q+1)` 的子串。如果我们尝试 `k<q`，那么 `B` 的长度大于 `A*q`，因此不能是子字符串。当 `k=q+1` 时，`A*k` 已经足够大，可以尝试 `B` 的所有位置，即 `A[i:i+len(B)] == B`，`i = 0, 1, ..., len(A) - 1`。

复杂度分析

- 时间复杂度：$O(N*(N+M))$。其中 M, N 是字符串 `A,B` 的长度。我们创建了两个字符串 `A*q`，`A*(q+1)`，其复杂度最多为 `O(M+N)`。当检查 `B` 是否是 `A` 的子串时，复杂度为 $O(N)$。
- 空间复杂度：如上所述，我们创建了使用 $O(M+N)$ 空间的字符串。

https://leetcode-cn.com/problems/repeated-string-match/solution/an-guan-fang-ti-jie-si-lu-xie-de-dai-ma-by-seerjjj/

```
class Solution {
    public int repeatedStringMatch(String A, String B) {
        int q = B.length() / A.length();
        if(B.length() % A.length() != 0) q++;
        StringBuilder result = new StringBuilder();
        for(int i = 0;i < q;i++)
            result.append(A);
        if(result.toString().indexOf(B) != -1) return q;
        result.append(A);
        if(result.toString().indexOf(B) != -1) return q+1;
        return -1;
    }
}
```

47、串联所有单词的子串（30）

yes的练级攻略

30. 串联所有单词的子串

难度 困难 277 收藏 分享 切换为英文 关注 反馈

给定一个字符串 **s** 和一些长度相同的单词 **words**。找出 **s** 中恰好可以由 **words** 中所有单词串联形成的子串的起始位置。

注意子串要与 **words** 中的单词完全匹配，中间不能有其他字符，但不需要考虑 **words** 中单词串联的顺序。

示例 1：

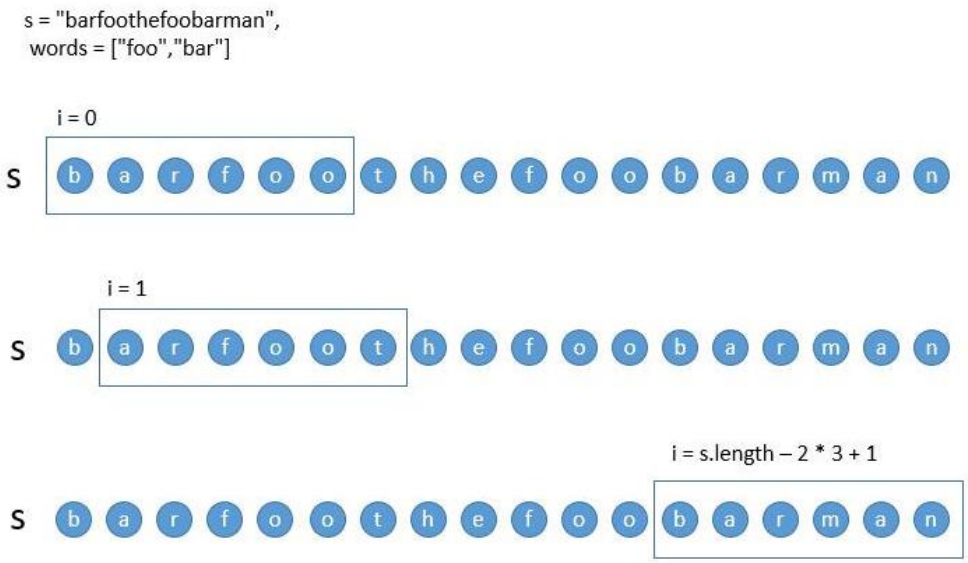
```
输入：
  s = "barfoothefoobarman",
  words = ["foo","bar"]
输出：[0,9]
解释：
从索引 0 和 9 开始的子串分别是 "barfoo" 和 "foobar" 。
输出的顺序不重要，[9,0] 也是有效答案。
```

示例 2：

```
输入：
  s = "wordgoodgoodgoodbestword",
  words = ["word","good","best","word"]
输出：[]
```

https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-w-6/

首先，最直接的思路，判断每个子串是否符合，符合就把下标保存起来，最后返回即可。



如上图，利用循环变量 i，依次后移，判断每个子串是否符合即可。

怎么判断子串是否符合？这也是这个题的难点了，由于子串包含的单词顺序并不需要固定，如果是两个单词 A，B，我们只需要判断子串是否是 AB 或者 BA 即可。如果是三个单词 A，B，C 也还好，只需要判断子串是否是 ABC，或者 ACB，BAC，BCA，CAB，CBA 就可以了，但如果更多单词呢？那就崩溃了。

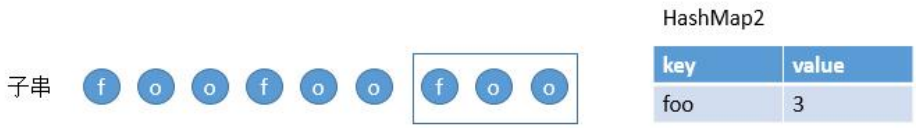
更多硬核内容就在公众号「yes 的练级攻略」

用两个 HashMap 来解决。首先，我们把所有的单词存到 HashMap 里，key 直接存单词，value 存单词出现的个数（因为给出的单词可能会有重复的，所以可能是 1 或 2 或者其他）。然后扫描子串的单词，如果当前扫描的单词在之前的 HashMap 中，就把该单词存到新的 HashMap 中，并判断新的 HashMap 中该单词的 value 是不是大于之前的 HashMap 该单词的 value，如果大了，就代表该子串不是我们要找的，接着判断下一个子串就可以了。如果不大于，那么我们接着判断下一个单词的情况。子串扫描结束，如果子串的全部单词都符合，那么该子串就是我们找的其中一个。看下具体的例子。

看下图，我们把 words 存到一个 HashMap 中。

然后遍历子串的每个单词。

第二个单词也在 HashMap1 中，然后把 foo 存到 HashMap2 中，因为之前已经存过了，所以更新它的 value 为 2，然后继续比较此时 foo 的 value 和 HashMap1 中 foo 的 value，2 <= 2，所以继续扫描下一个单词。



第三个单词也在 HashMap1 中，然后把 foo 存到 HashMap2 中，因为之前已经存过了，所以更新它的 value 为 3，然后继续比较此时 foo 的 value 和 HashMap1 中 foo 的 value，3 > 2，所以表明该字符串不符合。然后判断下个子串就好了。

当然上边的情况都是单词在 HashMap1 中，如果不在的话就更好说了，不在就表明当前子串肯定不符合了，直接判断下个子串就好了。

时间复杂度：假设 s 的长度是 n，words 里有 m 个单词，那么时间复杂度就是 O（n * m）。

空间复杂度：两个 HashMap，假设 words 里有 m 个单词，就是 O（m）。

30 具体代码

```
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        List<Integer> res = new ArrayList<>();
        if (s == null || s.length() == 0 || words == null || words.leng
th == 0) {
            return res;
        }
        int numOfWords = words.length;
        int lengthOfWord = words[0].length();
        int len = numOfWords * lengthOfWord;
```

```
//存着所有单词
        HashMap<String, Integer> map = new HashMap<>();
        for (String str: words) {
            map.put(str, map.getOrDefault(str,0) + 1);
        }
//遍历所有子串
        for (int i = 0; i <= s.length() - len; i++) {
            HashMap<String, Integer> strCount = new HashMap<>();
            int j = 0;
            while(j < numOfWords) {
                String subs = s.substring(i + j * lengthOfWord, i + (j
+ 1) * lengthOfWord);
//判断子串是否在所有单词中
                if(! map.containsKey(subs)){
                    break;
                }
                strCount.put(subs, strCount.getOrDefault(subs,0) + 1);
                if (strCount.get(subs) > map.get(subs)) {
                    break;
                }
                j++;
            }
            if (j == numOfWords) {
                res.add(i);
            }
        }
        return res;
    }
}
```

48、计数二进制子串（696）

更多硬核内容就在公众号「yes 的练级攻略」

696. 计数二进制子串

难度 简单 183 收藏 分享 切换为英文 关注 反馈

给定一个字符串 `s`，计算具有相同数量0和1的非空(连续)子字符串的数量，并且这些子字符串中的所有0和所有1都是组合在一起的。

重复出现的子串要计算它们出现的次数。

示例 1：

```
输入："00110011"
输出：6
解释：有6个子串具有相同数量的连续1和0：“0011”，“01”，“1100”，“10”，“0011” 和 “01”。

请注意，一些重复出现的子串要计算它们出现的次数。

另外，“00110011”不是有效的子串，因为所有的0（和1）没有组合在一起。
```

示例 2：

```
输入："10101"
输出：4
解释：有4个子串：“10”，“01”，“10”，“01”，它们具有相同数量的连续1和0。
```

注意：

- `s.length` 在1到50,000之间。
- `s` 只包含 “0” 或 “1” 字符。

https://leetcode-cn.com/problems/count-binary-substrings/solution/yi-ge-bi-jiao-qiao-miao-de-ban-fa-by-uygn9i8c6n/

解题思路

此处撰写解题思路
定义两个变量preLen和curLen，分别表示前面连续字符串中字符的个数，现在连续字符串中字符的个数。
当前字符和上一个字符相等时curLen++,不相等时preLen=curLen,然后curLen设为1。
如果preLen>=curLen,那么结果+1。
例如0011,应该输出为2。
1.curLen=1,preLen=0,ret=0;
2.curLen=2,preLen=0,ret=0;
3.curLen=1,preLen=2,ret=1;//001中的01
4.curLen=2,preLen=2,ret=2;//0011中的0011

```java
class Solution {
    public int countBinarySubstrings(String s) {
        int pre = 0, cur = 1, res = 0;
        for (int i = 1; i < s.length(); i++){
            if (s.charAt(i) == s.charAt(i - 1)) cur++;
            else {
                pre = cur;
                cur = 1;
            }
            if (pre >= cur) res++;
        }
```

```
            return res;
        }
    }
```

49、至少有 K 个重复字符的最长子串（395）

395. 至少有K个重复字符的最长子串

难度 中等 188 收藏 分享 切换为英文 关注 反馈

找到给定字符串（由小写字符组成）中的最长子串 *T*， 要求 *T* 中的每一字符出现次数都不少于 *k*。输出 *T* 的长度。

示例 1:

```
输入:
s = "aaabb", k = 3

输出:
3

最长子串为 "aaa" ，其中 'a' 重复了 3 次。
```

示例 2:

```
输入:
s = "ababbc", k = 2

输出:
5

最长子串为 "ababb" ，其中 'a' 重复了 2 次， 'b' 重复了 3 次。
```

https://leetcode-cn.com/problems/longest-substring-with-at-least-k-repeating-characters/solution/zhi-shao-you-kge-zhong-fu-zi-fu-de-zui-chang-zi-ch/

公众号 yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

基本思路
思路的话主要就是分治，相信大家都很容易理解。
先遍历一遍统计字符串S中每个字母出现的次数,然后再遍历一遍找到出现次数小于k的一个字母对应的位置（假设S[mid]对应的字母出现次数小于k）。那么包含S[mid]的子串显然不可能符合题目要求。

所以原问题求S[start,end]字符串对应的答案就等价于求S[start,mid-1]和S[mid+1,end]对应的2个答案的最大值。也即longestSubstring(S[start,end])=max(longestSubstring(S[start,mid-1]) , longestSubstring(S[mid+1,end]))

(如果mid不存在，则说明原字符串S完全符合要求，所以答案即为S.size())

最初跑248ms的3个原因
我一开始就是完全按照那个基本思路直接递归求解，但是这样的话会非常慢。可能原因如下：
(以下列举的字符串中，'数字' 对应数量小于k的字符)

.....1234..... 若字符串如左边所示，我们不做处理直接递归求解时，找到的第一个数量小于k的字符为1，以此为基准点分割出来的左右两个子串分别为
..... 和 234.....，在右子串中，前3个字符均为数量小于k的字符，那么在右子串继续递归计算的时候，又不得不重复计算多三次每个字符出现的次数。
所以在第一次递归时，如果能直接把1234这个整体作为一个分隔点(而不是只把1作为分隔点)，将原字符串分隔成为 ..... 和 ..... 再递归，则能节省时间，尤其是这个整体的长度越长时，节省的时间越多。

.....1.....2.....3.....4..... 若字符串如左边所示，如果我们不做任何处理直接递归求解时，找到的第一个数量小于k的字符为1，以此为分隔点分隔出来的左右两个子串分别为 ..... 和 .....2.....3.....4.....
这样的话右子串在继续递归计算时，则不得不重复计算多几次每个字符出现的次数，然后找到下一个字符 2作为分隔点继续递归，而这些实际在第一次递归时，已经求出来了。
所以在第一次递归时，如果能以1 2 3 4作为分隔点分割出5个子串，再分5次去递归，则能节约不少时间。

.....1..... 若字符串如左边所示，假设以1作为分隔点，左子串递归求解得到的答案是left_ans，如果此时右子串的长度right_length<=ans，那么显然这个时候右子串得到的答案right_ans显然<=right_length，那么这时候递归右子串没有意义。
所以如果能提前判断是否有必要递归计算这个子串，不必要时不递归计算该子串则能节省一些时间

解题思路：递归拆分子串，分治。先统计出每个字符出现的频次，维护一对双指针，从首尾开始统计，从首尾往中间排除，如果出现次数小于k则不可能出现在最终子串中，排除并挪动指针，然后得到临时子串，依次从头遍历，一旦发现出现频次小于k的字符，以该字符为分割线，分别递归求其最大值返回

```
class Solution {
  public int longestSubstring(String s, int k) {
        int len = s.length();
        if(len == 0 || len < k) return 0;
        if (k < 2) return len;
        return dtgh(s.toCharArray(),k,0,len);
    }
    public int dtgh(char[] chars, int k, int p1, int len){
        //动态规划后若长度小于k 就不是
        if (len - p1 < k) return 0;
        //统计字符出现的次数
        int[] times = new int[26];
        for(int i = p1; i < len; i++){
            times[chars[i] - 'a']++;
        }
        //设置两个指针优化算法，不设置也行
        //若头指针小于k 就右移
        //写len-p1>=k 是因为要p1++若len-p1>0 才行
        while(len - p1 >= k && times[chars[p1] - 'a'] < k){
            p1++;
        }
```

```
            //若尾指针小于k 就左移
            while(len - p1 >= k && times[chars[len-1] - 'a'] < k){
                len--;
            }

            if(len - p1 < k)return 0;
            //如果次数小于k，那就分割动态规划
            for(int i = p1; i < len; i++){
                if(times[chars[i] - 'a']<k){
                    //System.out.println(times[chars[i]-'a']);
                  return Math.max(dtgh(chars, k, p1, i), dtgh(chars, k, i+1
, len));
                }
            }
            return len - p1;
        }
}
```

50、至少 K 个不同字符的子串（lintcode 1375）

1375. 至少K个不同字符的子串

给定一个仅包含小写字母的字符串 `S`.

返回 `S` 中至少包含 `k` 个不同字符的子串的数量.

样例
样例 1:

```
输入: S = "abcabcabca", k = 4
输出: 0
解释: 字符串中一共就只有 3 个不同的字符.
```

样例 2:

```
输入: S = "abcabcabcabc", k = 3
输出: 55
解释: 任意长度不小于 3 的子串都含有 a, b, c 这三个字符.
    比如,长度为 3 的子串共有 10 个, "abc", "bca", "cab" ... "abc"
    长度为 4 的子串共有 9 个, "abca", "bcab", "cabc" ... "cabc"
    ...
    长度为 12 的子串有 1 个, 就是 S 本身.
    所以答案是 1 + 2 + ... + 10 = 55.
```

注意事项

1. 10 ≤ length(S) ≤ 1,000,000
2. 1 ≤ k ≤ 26

https://www.jiuzhang.com/solution/substring-with-at-least-k-distinct-characters/#tag-lang-java

方法 1：滑动窗口

1）套路，滑动窗口并套住包含 k 个不同字符的子串

2）更新答案的时候注意去重，重复的解集就是包含了之前的小窗口的情况，所以用一个 start 变量记录上一次窗口的起始位置

```
public class Solution {
    /**
     * @param s: a string
```

```
     * @param k: an integer
     * @return: the number of substrings there are that contain at least k distinct characters
     */
    public long kDistinctCharacters(String s, int k) {
        if (s == null || s.length() < k) return 0;
        final int n = s.length();
        int[] freq = new int[26]; //记录每个字符在滑动过程中出现的次数，由此判断是否当前含有 k 个不同字符
        long res = 0; //返回值，滑动过程中更新
        //cnt: 当前窗口中包含的不同字符数量; start : 当前窗口起始位置; (l, r) 窗口左右位置，满足条件的位置为[l-1, r-1]
        int cnt = 0, start = 0, l = 0, r = 0;
        while (r < n) {
            //套路
            while (r < n && cnt < k) {
                int c = s.charAt(r++) - 'a';
                if (freq[c]++ == 0) cnt++;
            }
            while (l < n && cnt == k) {
                int p = s.charAt(l++) - 'a';
                if (--freq[p] == 0) cnt--;
            }
            //更新结果，去重
            if (l <= r) {
                int lnum = l - start - 1;
                int rnum = n - r;
                res += lnum + rnum + lnum * rnum + 1;

                start = l;
            }
        }
        return res;
    }
}
```

方法 2：HashSet

```
public class Solution {
    /**
     * @param s: a string
     * @param k: an integer
     * @return: the number of substrings there are that contain at least k distinct characters
     */
    public long kDistinctCharacters(String s, int k) {
        int len = s.length();
        long res = 0;
```

禁止商用 公众号：yes的练级攻略

```
            int l = 0;
            int r = 0;
            int[] charArr = new int[26];
            Set<Character> set = new HashSet();
            for (l = 0; l < len - k + 1; l++) {
                while (r < len && set.size() < k) {
                    set.add(s.charAt(r));
                    charArr[s.charAt(r) - 'a']++;
                    r++;
                }
                if (set.size() == k) {
                    res += (len - r + 1);
                }
                charArr[s.charAt(l) - 'a']--;
                if (charArr[s.charAt(l) - 'a'] == 0) {
                    set.remove(s.charAt(l));
                }
            }
            return res;
        }
}
```

51、按字典序排在最后的子串（1163）

1163. 按字典序排在最后的子串

难度 困难 27 收藏 分享 切换为英文 关注 反馈

给你一个字符串 `s`，找出它的所有子串并按字典序排列，返回排在最后的那个子串。

示例 1：

```
输入："abab"
输出："bab"
解释：我们可以找出 7 个子串 ["a", "ab", "aba", "abab", "b", "ba", "bab"]。按字典序排在最后的子串是 "bab"。
```

示例 2：

```
输入："leetcode"
输出："tcode"
```

提示：

1. `1 <= s.length <= 4 * 10^5`
2. `s` 仅含有小写英文字符。

https://leetcode-cn.com/problems/last-substring-in-lexicographical-order/solution/java8ms-by-guyul/

解题思路

8ms，97%
从后往前遍历，遇到更大的字符交换指针。
遇到相同的字符，先不着急比较后续的字符，看看它前面的一个是不是一样的，一样的就continue，直到不一样或者到头了，再比较。

禁止商用

```
class Solution {
    public String lastSubstring(String s) {
        char[] str = s.toCharArray();
        int index = str.length - 1;
        int max = 0;
        for(int i = str.length - 1; i >= 0; i--){
            if(str[i] - 'a' > max){
                index = i;
                max = str[i] - 'a';
            }else if(str[i] - 'a' == max){
                if(i - 1 >= 0 && str[i] == str[i-1]) continue;//非常关键！！！
                int temp = index;
                index = i;
                max = str[i] - 'a';
                for(int j = i, k = temp; j < str.length && k < str.length; j++,
 k++){
                    if(str[k] == str[j]) continue;
                    if(str[k] > str[j]){
                        index = temp;
                        max = str[index]-'a';
                        break;
                    }else if(str[k] < str[j]){
                        break;
                    }
                }
            }
        }

        return s.substring(index);
    }
}
```

52、按字典序排列最小的等效字符串（1061）【并查集】

更多硬核内容就在公众号「yes 的练级攻略」

1061. 按字典序排列最小的等效字符串

难度 中等 9 收藏 分享 切换为英文 关注 反馈

给出长度相同的两个字符串：`A` 和 `B`，其中 A[i] 和 B[i] 是一组等价字符。举个例子，如果 `A = "abc"` 且 `B = "cde"`，那么就有 `'a' == 'c'`, `'b' == 'd'`, `'c' == 'e'`。

等价字符遵循任何等价关系的一般规则：

- 自反性：'a' == 'a'
- 对称性：'a' == 'b' 则必定有 'b' == 'a'
- 传递性：'a' == 'b' 且 'b' == 'c' 就表明 'a' == 'c'

例如，`A` 和 `B` 的等价信息和之前的例子一样，那么 `S = "eed"`, `"acd"` 或 `"aab"`，这三个字符串都是等价的，而 `"aab"` 是 `S` 的按字典序最小的等价字符串

利用 `A` 和 `B` 的等价信息，找出并返回 `S` 的按字典序排列最小的等价字符串。

示例 1：

```
输入：A = "parker", B = "morris", S = "parser"
输出："makkek"
解释：根据 A 和 B 中的等价信息，我们可以将这些字符分为 [m,p], [a,o], [k,r,s], [e,i] 共 4 组。每组中的字符都是等价的，并按字典序排列。所以答案是 "makkek"。
```

示例 2：

```
输入：A = "hello", B = "world", S = "hold"
输出："hdld"
解释：根据 A 和 B 中的等价信息，我们可以将这些字符分为 [h,w], [d,e,o], [l,r] 共 3 组。所以只有 S 中的第二个字符 'o' 变成 'd'，最后答案为 "hdld"。
```

示例 3：

```
输入：A = "leetcode", B = "programs", S = "sourcecode"
输出："aauaaaaada"
解释：我们可以把 A 和 B 中的等价字符分为 [a,o,e,r,s,c], [l,p], [g,t] 和 [d,m] 共 4 组，因此 S 中除了 'u' 和 'd' 之外的所有字母都转化成了 'a'，最后答案为 "aauaaaaada"。
```

提示：

1. 字符串 `A`, `B` 和 `S` 仅有从 `'a'` 到 `'z'` 的小写英文字母组成。
2. 字符串 `A`, `B` 和 `S` 的长度在 `1` 到 `1000` 之间。
3. 字符串 `A` 和 `B` 长度相同。

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

并查集被很多OIer认为是最简洁而优雅的数据结构之一，主要用于解决一些**元素分组**的问题。它管理一系列**不相交的集合**，并支持两种操作：

- **合并**（Union）：把两个不相交的集合合并为一个集合。
- **查询**（Find）：查询两个元素是否在同一个集合中。

当然，这样的定义未免太过学术化，看完后恐怕不太能理解它具体有什么用。所以我们先来看看并查集最直接的一个应用场景：**亲戚问题**。

（洛谷P1551）亲戚

> **题目背景**
> 若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。
> **题目描述**
> 规定：x和y是亲戚，y和z是亲戚，那么x和z也是亲戚。如果x,y是亲戚，那么x的亲戚都是y的亲戚，y的亲戚也都是x的亲戚。
> **输入格式**
> 第一行：三个整数n,m,p，（n<=5000,m<=5000,p<=5000），分别表示有n个人，m个亲戚关系，询问p对亲戚关系。
> 以下m行：每行两个数Mi，Mj，1<=Mi，Mj<=N，表示Mi和Mj具有亲戚关系。
> 接下来p行：每行两个数Pi，Pj，询问Pi和Pj是否具有亲戚关系。
> **输出格式**
> P行，每行一个’Yes’或’No’。表示第i个询问的答案为“具有”或“不具有”亲戚关系。

这其实是一个很有现实意义的问题。我们可以建立模型，把所有人划分到若干个不相交的集合中，每个集合里的人彼此是亲戚。为了判断两个人是否为亲戚，只需看它们是否属于同一个集合即可。因此，这里就可以考虑用并查集进行维护了。

yes的练级攻略

初始化

```
int fa[MAXN];
inline void init(int n)
{
    for (int i = 1; i <= n; ++i)
        fa[i] = i;
}
```

假如有编号为1, 2, 3, ..., n的n个元素，我们用一个数组fa[]来存储每个元素的父节点（因为每个元素有且只有一个父节点，所以这是可行的）。一开始，我们先将它们的父节点设为自己。

查询

```
int find(int x)
{
    if(fa[x] == x)
        return x;
    else
        return find(fa[x]);
}
```

我们用递归的写法实现对代表元素的查询：一层一层访问父节点，直至根节点（根节点的标志就是父节点是本身）。要判断两个元素是否属于同一个集合，只需要看它们的根节点是否相同即可。

合并

```
inline void merge(int i, int j)
{
    fa[find(i)] = find(j);
}
```

合并操作也是很简单的，先找到两个集合的代表元素，然后将前者的父节点设为后者即可。当然也可以将后者的父节点设为前者，这里暂时不重要。本文末尾会给出一个更合理的比较方法。

更多硬核内容就在公众号「yes 的练级攻略」

其实这说来也很好实现。只要我们在查询的过程中，**把沿途的每个节点的父节点都设为根节点**即可。下一次再查询时，我们就可以省很多事。这用递归的写法很容易实现：

合并（路径压缩）

```
int find(int x)
{
    if(x == fa[x])
        return x;
    else{
        fa[x] = find(fa[x]);  //父节点设为根节点
        return fa[x];         //返回父节点
    }
}
```

以上代码常常简写为一行：

```
int find(int x)
{
    return x == fa[x] ? x : (fa[x] = find(fa[x]));
}
```

注意赋值运算符=的优先级没有三元运算符?:高，这里要加括号。

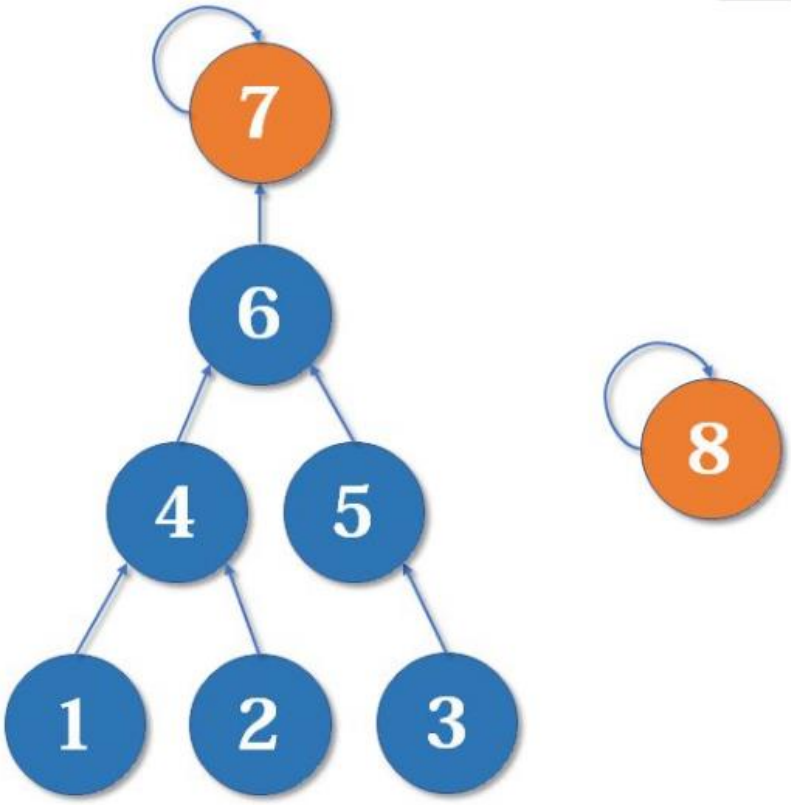
路径压缩优化后，并查集的时间复杂度已经比较低了，绝大多数不相交集合的合并查询问题都能够解决。然而，对于某些时间卡得很紧的题目，我们还可以进一步优化。

yes的练级攻略

按秩合并

有些人可能有一个误解，以为路径压缩优化后，并查集始终都是一个**菊花图**（只有两层的树的俗称）。但其实，由于路径压缩只在查询时进行，也只压缩一条路径，所以并查集最终的结构仍然可能是比较复杂的。例如，现在我们有一棵较复杂的树需要与一个单元素的集合合并：

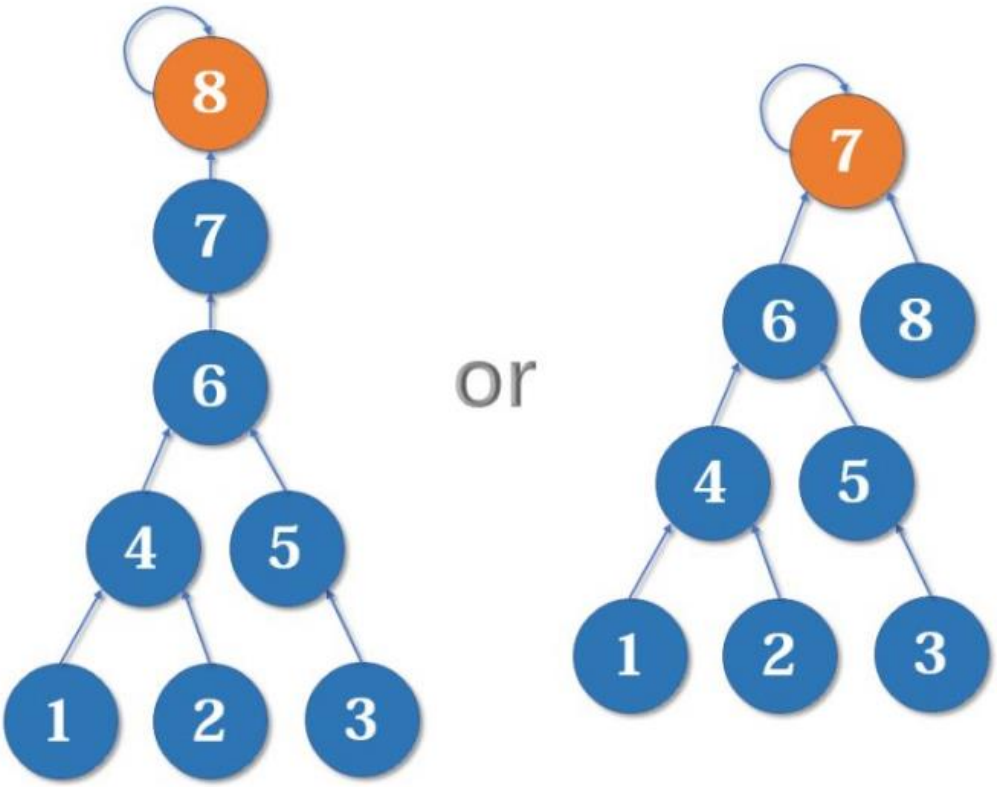
☆ 收藏



假如这时我们要merge(7,8)，如果我们可以选择的话，是把7的父节点设为8好，还是把8的父节点设为7好呢？

更多硬核内容就在公众号「yes 的练级攻略」

当然是后者。因为如果把7的父节点设为8，会使树的**深度**（树中最长链的长度）加深，原来的树中每个元素到根节点的距离都变长了，之后我们寻找根节点的路径也就会相应变长。虽然我们有路径压缩，但路径压缩也是会消耗时间的。而把8的父节点设为7，则不会有这个问题，因为它没有影响到不相关的节点。



这启发我们：我们应该把简单的树往复杂的树上合并，而不是相反。因为这样合并后，到根节点距离变长的节点个数比较少。

yes的练级攻略

我们用一个数组rank[]记录每个根节点对应的树的深度（如果不是根节点，其rank相当于以它作为根节点的**子树**的深度）。一开始，把所有元素的rank（**秩**）设为1。合并时比较两个根节点，把rank较小者往较大者上合并。路径压缩和按秩合并如果一起使用，时间复杂度接近 $O(n)$ ，但是很可能会破坏rank的准确性。

值得注意的是，按秩合并会带来额外的**空间复杂度**，可能被一些卡空间的毒瘤题卡掉。

初始化（按秩合并）

```
inline void init(int n)
{
    for (int i = 1; i <= n; ++i)
    {
        fa[i] = i;
        rank[i] = 1;
    }
}
```

合并（按秩合并）

```
inline void merge(int i, int j)
{
    int x = find(i), y = find(j);    //先找到两个根节点
    if (rank[x] <= rank[y])
        fa[x] = y;
    else
        fa[y] = x;
    if (rank[x] == rank[y] && x != y)
        rank[y]++;                   //如果深度相同且根节点不同，则新的根节点的深度+1
}
```

为什么深度相同，新的根节点深度要+1？如下图，我们有两个深度均为2的树，现在要merge(2,5)：

https://leetcode-cn.com/problems/lexicographically-smallest-equivalent-string/solution/bing-cha-ji-he-bing-gen-jie-dian-shi-qu-xiao-de-na/

解题思路

1. 通过A与B的关系，刷新并查集
2. union(x,y)时，判断parent[x]和parent[y]谁的值大，值大的一方的parent修改为值小的一方。确保根节点是最小值
3. 遍历S的每一个字符，通过find(x)获取最小的根节点，并刷新新S中的该字符。

```
class Solution {
    public String smallestEquivalentString(String A, String B, String S) {
        if(null==A||A.length()==0||B==null||B.length()==0||null==S||S.length()==0)
        {
            return S;
```

```
        }
        // 并查集，并取最小字母为根节点
        int length = A.length();
        DSU dsu = new DSU(26);
        for(int i=0;i<length;i++)
        {
            dsu.union(A.charAt(i)-'a',B.charAt(i)-'a');
        }
        char[] sChar = S.toCharArray();
        for(int i=0;i<sChar.length;i++)
        {
            int charInt = dsu.find(sChar[i]-'a')+'a';
            sChar[i]=(char)charInt;
        }
        return String.valueOf(sChar);

    }
    class DSU
    {
        int[] parent;
        DSU(int size)
        {
            init(size);
        }
        private void init(int size)
        {
            parent = new int[size];
            for(int i=0;i<parent.length;i++)
            {
                parent[i]=i;
            }
        }
        private void union(int x,int y)
        {
            x=find(x);
            y=find(y);
            if(x==y)
            {
                return;
            }
            // 取最小的节点为root 节点
            if(x>y)
            {
                parent[x]=y;
```

```
            }else {
                parent[y]=x;
            }
        }
        private int find(int x)
        {
            if(x==parent[x])
            {
                return x;
            }
            parent[x] = find(parent[x]);
            return parent[x];
        }
    }
}
```

53、替换后的最长重复字符（424）

424. 替换后的最长重复字符

难度 中等 141 收藏 分享 切换为英文 关注 反馈

给你一个仅由大写英文字母组成的字符串，你可以将任意位置上的字符替换成另外的字符，总共可最多替换 *k* 次。在执行上述操作后，找到包含重复字母的最长子串的长度。

注意:
字符串长度 和 *k* 不会超过 10^4。

示例 1:

输入:
`s = "ABAB", k = 2`

输出:
4

解释:
用两个'A'替换为两个'B',反之亦然。

示例 2:

输入:
`s = "AABABBA", k = 1`

输出:
4

解释:
将中间的一个'A'替换为'B',字符串变为 "AABBBBA"。
子串 "BBBB" 有最长重复字母，答案为 4。

https://leetcode-cn.com/problems/longest-repeating-character-replacement/solution/tong-guo-ci-ti-liao-jie-yi-xia-shi-yao-shi-hua-don/

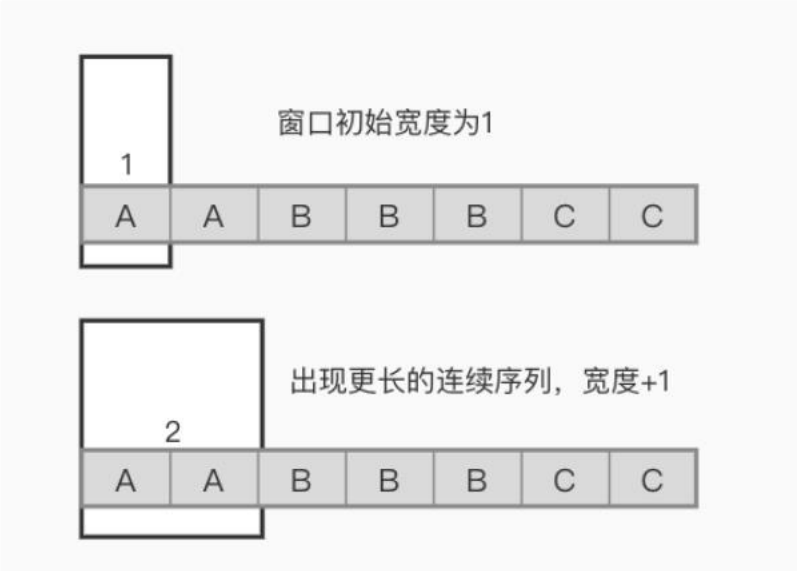
更多硬核内容就在公众号「yes 的练级攻略」

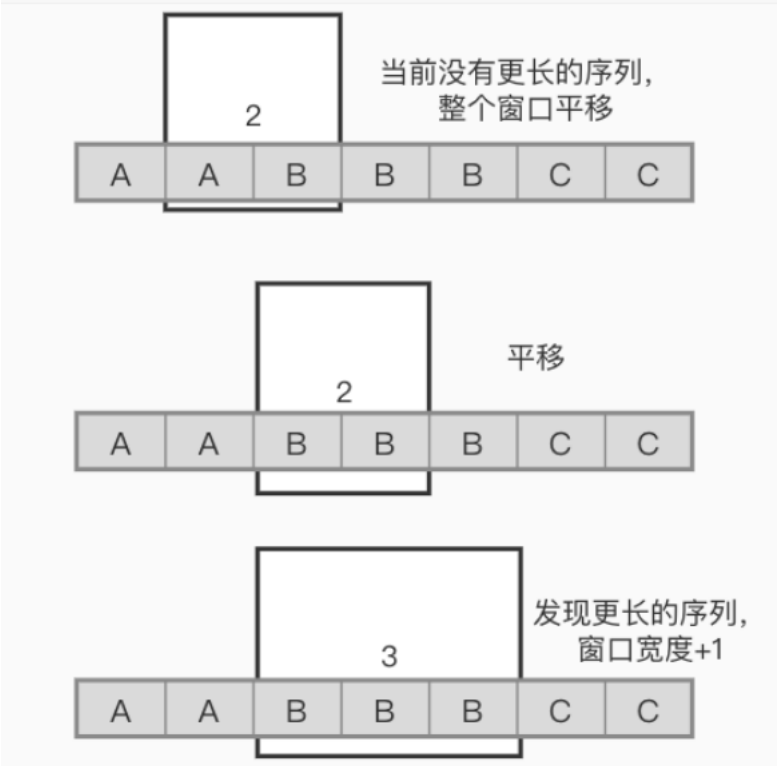
本题是比较典型的滑动窗口问题

这类问题一般通过一个滑动窗口就能在O(N)的时间复杂度下求解

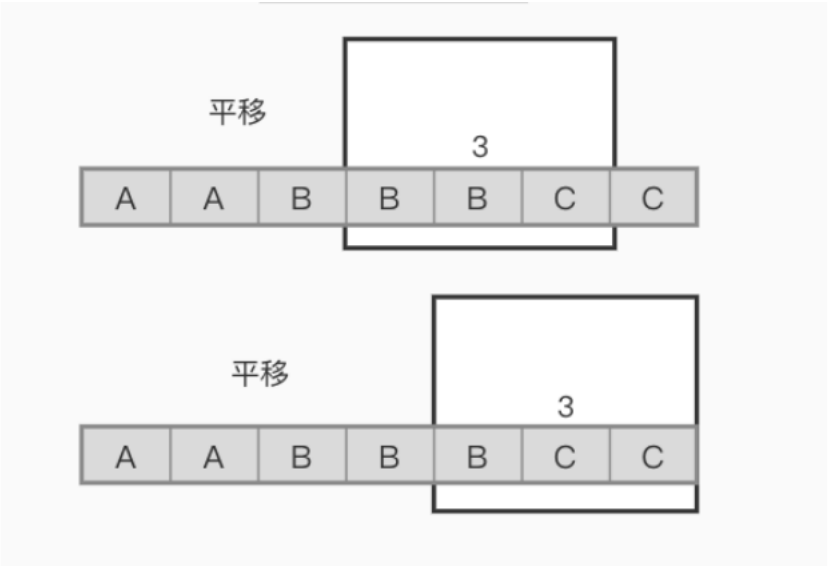
本题可以先退化成考虑K=0的情况，此时题目就变成了求解字符串中最长连续子串长度问题了

我们先可以通过这个特例先了解一下滑动窗口的求解过程





攻略



上图的求解过程展示中，窗口从左至右不断扩张/滑动，当窗口触达字符串末尾字符时，运算结束，窗口的宽度为最终结果。初始窗口的宽度为1，我们不断的通过向当前窗口覆盖的子串后面追加一个字符看是否能满足我们的要求，如果满足窗口扩张，如果不满足，窗口向右滑动。

当K>0时，子串的条件变成了允许我们变换子串中的K个字符使其变成一个连续子串

那么这个题的关键点就是我们如何判断一个字符串改变K个字符，能够变成一个连续串

如果当前字符串中的出现次数最多的字母个数+K大于串长度，那么这个串就是满足条件的

我们维护一个数组int[26]来存储当前窗口中各个字母的出现次数，left表示窗口的左边界，right表示窗口右边界

- 窗口扩张：left不变，right++
- 窗口滑动：left++, right++

historyCharMax保存滑动窗口内相同字母出现次数的**历史**最大值，通过判断窗口宽度(right - left + 1)是否大于historyCharMax + K来决定窗口是否做滑动，否则窗口就扩张

滑动窗口法具体代码

```
class Solution {
    public int characterReplacement(String s, int k) {
        int[] map = new int[26];
        if (s == null) return 0;
        char[] ch = s.toCharArray();
        int left = 0, right = 0, max = 0;
        for (right = 0; right < ch.length; right++){
            int idx = ch[right] - 'A';
            map[idx]++;
            max = Math.max(max, map[idx]);
            if (right - left + 1 > max + k){
                map[ch[left] - 'A']--;
                left++;
            }
        }
        return ch.length - left;
    }
```

}

54、重复的子字符串（459）

459. 重复的子字符串

难度 简单 291 收藏 分享 切换为英文 关注 反馈

给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。

示例 1:

```
输入："abab"

输出：True

解释：可由子字符串 "ab" 重复两次构成。
```

示例 2:

```
输入："aba"

输出：False
```

示例 3:

```
输入："abcabcabcabc"

输出：True

解释：可由子字符串 "abc" 重复四次构成。（或者子字符串 "abcabc" 重复两次构成。）
```

https://leetcode-cn.com/problems/repeated-substring-pattern/solution/zhong-fu-de-zi-zi-fu-chuan-by-leetcode-solution/

方法二：字符串匹配

思路与算法

我们可以把字符串 s 写成

$$s's'\cdots s's'$$

的形式，总计 $\frac{n}{n'}$ 个 s'。但我们如何在不枚举 n' 的情况下，判断 s 是否能写成上述的形式呢？

如果我们移除字符串 s 的前 n' 个字符（即一个完整的 s'），再将这些字符保持顺序添加到剩余字符串的末尾，那么得到的字符串仍然是 s。由于 $1 \leq n' < n$，那么如果将两个 s 连在一起，并移除第一个和最后一个字符，那么得到的字符串一定包含 s，即 s 是它的一个子串。

因此我们可以考虑这种方法：我们将两个 s 连在一起，并移除第一个和最后一个字符。如果 s 是该字符串的子串，那么 s 就满足题目要求。

注意到我们证明的是**如果 s 满足题目要求，那么 s 有这样的性质**，而我们使用的方法却是**如果 s 有这样的性质，那么 s 满足题目要求**。因此，只证明了充分性是远远不够的，我们还需要证明必要性。

> 题解区的很多题解都忽略了这一点，但它是非常重要的。

证明需要使用一些同余运算的小技巧，可以见方法三之后的「正确性证明」部分。这里先假设我们已经完成了证明，这样就可以使用非常简短的代码完成本题。在下面的代码中，我们可以从位置 1 开始查询，并希望查询结果不为位置 n，这与移除字符串的第一个和最后一个字符是等价的。

更多硬核内容就在公众号「yes 的练级攻略」

正确性证明

一方面，如果长度为 n 的字符串 s 是字符串 $t = s + s$ 的子串，并且 s 在 t 中的起始位置不为 0 或 n，那么 s 就满足题目的要求。证明过程如下：

- 我们设 s 在 t 中的起始位置为 i，$i \in (0, n)$。也就是说，t 中从位置 i 开始的 n 个连续的字符，恰好就是字符串 s。那么我们有：

  $$s[0:n-1] = t[i:n+i-1]$$

  由于 t 是由两个 s 拼接而成的，我们可以将 $t[i:n+i-1]$ 分成位置 $n-1$ 左侧和右侧两部分：

  $$\begin{cases} s[0:n-i-1] = t[i:n-1] \\ s[n-i:n-1] = t[n:n+i-1] = t[0:i-1] \end{cases}$$

  每一部分都可以对应回 s：

  $$\begin{cases} s[0:n-i-1] = s[i:n-1] \\ s[n-i:n-1] = s[0:i-1] \end{cases}$$

  这说明，s 是一个「可旋转」的字符串：将 s 的前 i 个字符保持顺序，移动到 s 的末尾，得到的新字符串与 s 相同。也就是说，**在模 n 的意义下**，

  $$s[j] = s[j+i]$$

  对于任意的 j 恒成立。

  > 「在模 n 的意义下」可以理解为，所有的加法运算的结果都需要对 n 取模，使得结果保持在 $[0, n)$ 中，这样加法就自带了「旋转」的效果。

  如果我们不断地连写这个等式：

  $$s[j] = s[j+i] = s[j+2i] = s[j+3i] = \cdots$$

  那么所有满足 $j_0 = j + k \cdot i$ 的位置 j_0 都有 $s[j] = s[j_0]$，j 和 j_0 在模 i 的意义下等价。由于我们已经在模 n 的意义下讨论这个问题，因此 j 和 j_0 在模 $\gcd(n, i)$ 的意义下等价，其中 $\gcd$ 表示最大公约数。也就是说，字符串 s 中的两个位置如果在模 $\gcd(n, i)$ 的意义下等价，那么它们对应的字符必然是相同的。

  由于 $\gcd(n, i)$ 一定是 n 的约数，那么字符串 s 一定可以由其长度为 $\gcd(n, i)$ 的前缀重复 $\frac{n}{\gcd(n,i)}$ 次构成。

另一方面，如果 s 满足题目的要求，那么 s 包含若干个「部分」，$t = s + s$ 包含两倍数量的「部分」，因此 s 显然是 t 的子串，并且起始位置可以不为 0 或 n：我们只需要选择 t 中第一个「部分」的起始位置即可。

综上所述，我们证明了：长度为 n 的字符串 s 是字符串 $t = s + s$ 的子串，并且 s 在 t 中的起始位置不为 0 或 n，**当且仅当** s 满足题目的要求。因此，

更多硬核内容就在公众号「yes 的练级攻略」

思考题答案

- 设查询串的的长度为 n，模式串的长度为 m，我们需要判断模式串是否为查询串的子串。那么使用 KMP 算法处理该问题时的时间复杂度是多少？在分析时间复杂度时使用了哪一种分析方法？
  - 时间复杂度为 $O(n+m)$，用到了均摊分析（摊还分析）的方法。
  - 具体地，无论在预处理过程还是查询过程中，虽然匹配失败时，指针会不断地根据 $fail$ 数组向左回退，看似时间复杂度会很高。但考虑匹配成功时，指针会向右移动一个位置，这一部分对应的时间复杂度为 $O(n+m)$。又因为向左移动的次数不会超过向右移动的次数，因此总时间复杂度仍然为 $O(n+m)$。
- 如果有多个查询串，平均长度为 n，数量为 k，那么总时间复杂度是多少？
  - 时间复杂度为 $O(nk+m)$。模式串只需要预处理一次。
- 在 KMP 算法中，对于模式串，我们需要预处理出一个 $fail$ 数组（有时也称为 $next$ 数组、π 数组等）。这个数组到底表示了什么？
  - $fail[i]$ 等于满足下述要求的 x 的最大值：$s[0:i]$ 具有长度为 $x+1$ 的完全相同的前缀和后缀。这也是 KMP 算法最重要的一部分。

方法 1：indexof

```
class Solution {
    public boolean repeatedSubstringPattern(String s) {
        return (s + s).indexOf(s, 1) != s.length();
    }
}
```

方法三：KMP 算法

思路与算法

在方法二中，我们使用了语言自带的字符串查找函数。同样我们也可以自己实现这个函数，例如使用比较经典的 KMP 算法。

读者需要注意以下几点：

- KMP 算法虽然有着良好的理论时间复杂度上限，但大部分语言自带的字符串查找函数并不是用 KMP 算法实现的。这是因为在实现 API 时，我们需要在平均时间复杂度和最坏时间复杂度二者之间权衡。普通的暴力匹配算法以及优化的 BM 算法拥有比 KMP 算法更为优秀的平均时间复杂度；
- 学习 KMP 算法时，一定要理解其本质。如果放弃阅读晦涩难懂的材料（即使大部分讲解 KMP 算法的材料都包含大量的图，但图毕竟只能描述特殊而非一般情况）而是直接去阅读代码，是永远无法学会 KMP 算法的。读者甚至无法理解 KMP 算法关键代码中的任意一行。

由于本题就是在一个字符串中查询另一个字符串是否出现，可以直接套用 KMP 算法。因此这里对 KMP 算法本身不再赘述。读者可以自行查阅资料进行学习。这里留了三个思考题，读者可以在学习完毕后尝试回答这三个问题，检验自己的学习成果：

- 设查询串的的长度为 n，模式串的长度为 m，我们需要判断模式串是否为查询串的子串。那么使用 KMP 算法处理该问题时的时间复杂度是多少？在分析时间复杂度时使用了哪一种分析方法？
- 如果有多个查询串，平均长度为 n，数量为 k，那么总时间复杂度是多少？
- 在 KMP 算法中，对于模式串，我们需要预处理出一个 $fail$ 数组（有时也称为 $next$ 数组、π 数组等）。这个数组到底表示了什么？

更多硬核内容就在公众号「yes 的练级攻略」

方法四：优化的 KMP 算法

思路与算法

如果读者能够看懂「正确性证明」和「思考题答案」这两部分，那么一定已经发现了方法三中的 KMP 算法有可以优化的地方。即：

- 在「正确性证明」部分，如果我们设 i 为**最小的**起始位置，那么一定有 $\gcd(n, i) = i$，即 n 是 i 的倍数。这说明字符串 s 是由长度为 i 的前缀重复 $\frac{n}{i}$ 次构成；
- 由于 $fail[n-1]$ 表示 s 具有长度为 $fail[n-1]+1$ 的完全相同的（且最长的）前缀和后缀。那么对于满足题目要求的字符串，一定有 $fail[n-1] = n-i-1$，即 $i = n - fail[n-1] - 1$；
- 对于不满足题目要求的字符串，n 一定不是 $n - fail[n-1] - 1$ 的倍数。

> 上述所有的结论都可以很容易地使用反证法证出。

因此，我们在预处理出 $fail$ 数组后，只需要判断 n 是否为 $n - fail[n-1] - 1$ 的倍数即可。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是字符串 s 的长度。
- 空间复杂度：$O(n)$。

方法 2：优化的 KMP

```
class Solution {
    public boolean repeatedSubstringPattern(String s) {
        return kmp(s);
    }

    public boolean kmp(String pattern) {
        int n = pattern.length();
        int[] fail = new int[n];
        Arrays.fill(fail, -1);
        for (int i = 1; i < n; ++i) {
            int j = fail[i - 1];
            while (j != -1 && pattern.charAt(j + 1) != pattern.charAt(i
)) {
                j = fail[j];
            }
            if (pattern.charAt(j + 1) == pattern.charAt(i)) {
                fail[i] = j + 1;
            }
        }
        return fail[n - 1] != -1 && n % (n - fail[n - 1] - 1) == 0;
    }
}
```

55、字符串中的第一个唯一字符（387）

387. 字符串中的第一个唯一字符

难度 简单 258 收藏 分享 切换为英文 关注 反馈

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

示例：

```
s = "leetcode"
返回 0

s = "loveleetcode"
返回 2
```

提示：你可以假定该字符串只包含小写字母。

禁止商用

https://leetcode-cn.com/problems/first-unique-character-in-a-string/solution/javasi-chong-jie-fa-zhu-bu-you-hua-ji-bai-100-by-k/

方法 1：哈希表（42ms）

```
class Solution {
    public int firstUniqChar(String s) {
        Map<Character, Integer> map = new HashMap<>();
        for (char cs : s.toCharArray()) {
            map.put(cs, map.getOrDefault(cs, 0) + 1);
        }
        for (int i = 0; i < s.length(); ++i) {
            if (map.get(s.charAt(i)) == 1) {
                return i;
            }
        }
        return -1;
    }
}
```

方法 2：用数组替换哈希表（6ms）

```
class Solution {
    public int firstUniqChar(String s) {
        int[] chars = new int[26];
        for (char c : s.toCharArray()) {
            chars[c - 'a'] += 1;
        }
        for (int i = 0; i < s.length(); ++i) {
            if (chars[s.charAt(i) - 'a'] == 1) {
                return i;
```

```
            }
        }
        return -1;
    }
}
```

方法 3：根据给定字符串长度分情况优化（2ms）

```
class Solution {
    public int firstUniqChar(String s) {
        // 字符串长度不超过 26，按照原先的方式遍历
        if (s.length() <= 26) {
            int[] chars = new int[26];
            for (char c : s.toCharArray()) {
                chars[c - 'a'] += 1;
            }
            for (int i = 0; i < s.length(); ++i) {
                if (chars[s.charAt(i) - 'a'] == 1) {
                    return i;
                }
            }
        }
        //只遍历 26 个字母，使用 indexOf 函数检查字符索引
        int result = -1;
        for (char c = 'a'; c <= 'z'; ++c) {
            int pre = s.indexOf(c);
            // s 包含该字符并且只出现一次
            if (pre != -1 && pre == s.lastIndexOf(c)) {
                // 取最前面的位置
                result = (result == -1 || result > pre) ? pre : result;
            }
        }
        return result;
    }
}
```

56、判定字符串是否唯一（面试题 01.01）【位运算】

更多硬核内容就在公众号「yes 的练级攻略」

面试题 01.01. 判定字符是否唯一

难度 简单 54 收藏 分享 切换为英文 关注 反馈

实现一个算法，确定一个字符串 s 的所有字符是否全都不同。

示例 1：

```
输入：s = "leetcode"
输出：false
```

示例 2：

```
输入：s = "abc"
输出：true
```

限制：

- 0 <= len(s) <= 100
- 如果你不使用额外的数据结构，会很加分。

禁止转用

https://leetcode-cn.com/problems/is-unique-lcci/solution/ji-chong-jie-fa-zong-jie-by-lian-zhou/

4，使用位运算。字符总共128个ASCII码，那就用两个long类型的值128个bit位表示这128个字符是否出现过。1代表出现过，0代表未出现。

```
class Solution {
    public boolean isUnique(String astr) {
        long high64 = 0;
        long low64 = 0;
        for (char ch : astr.toCharArray()) {
            if (ch >= 64) {
                long bitIndex = 1L << ch - 64;
                if ((high64 & bitIndex) != 0) {
                    return false;
                }
                high64 |= bitIndex;
            } else {
                long bitIndex = 1L << ch;
                if ((low64 & bitIndex) != 0) {
                    return false;
                }
                low64 |= bitIndex;
            }
        }
        return true;
    }
}
```

57、实现 strStr()函数（28）【方法 2 有 KMP 详解】

更多硬核内容就在公众号「yes 的练级攻略」

28. 实现 strStr()

难度 简单 515 收藏 分享 切换为英文 关注 反馈

实现 strStr() 函数。

给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。

示例 1:

```
输入: haystack = "hello", needle = "ll"
输出: 2
```

示例 2:

```
输入: haystack = "aaaaa", needle = "bba"
输出: -1
```

说明:

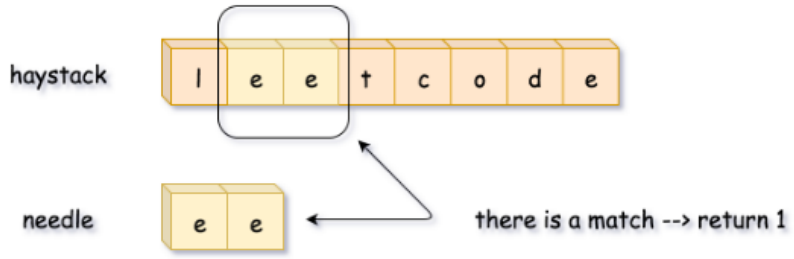
当 needle 是空字符串时，我们应当返回什么值呢？这是一个在面试中很好的问题。

对于本题而言，当 needle 是空字符串时我们应当返回 0 。这与C语言的 strstr() 以及 Java的 indexOf() 定义相符。

https://leetcode-cn.com/problems/implement-strstr/solution/shi-xian-strstr-by-leetcode/

strstr(str1,str2) 函数用于判断字符串 str2 是否是 str1 的子串。如果是，则该函数返回 str1 字符串从 str2 第一次出现的位置开始到 str1 结尾的字符串；否则，返回 NULL。

最直接的方法 - 沿着字符换逐步移动滑动窗口，将窗口内的子串与 needle 字符串比较。



复杂度分析

- 时间复杂度：$O((N-L)L)$，其中 N 为 haystack 字符串的长度，L 为 needle 字符串的长度。内循环中比较字符串的复杂度为 L，总共需要比较 (N - L) 次。
- 空间复杂度：$O(1)$。

方法 1：子串逐一比较（1ms）

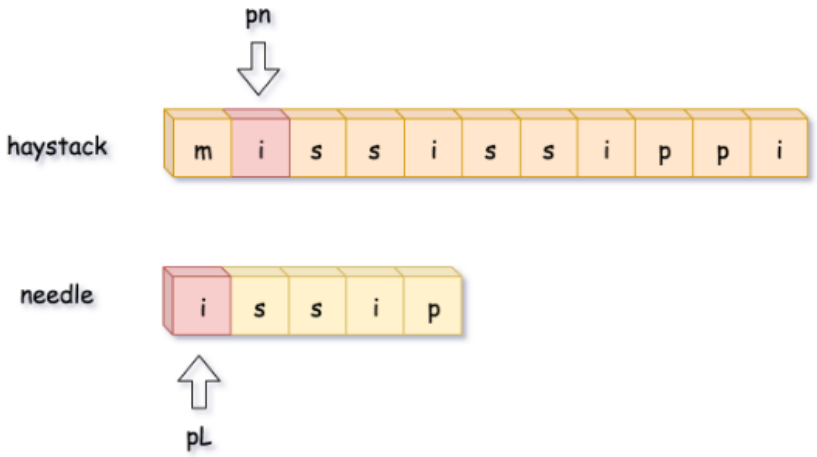
```
class Solution {
  public int strStr(String haystack, String needle) {
    int L = needle.length(), n = haystack.length();

    for (int start = 0; start < n - L + 1; ++start) {
      if (haystack.substring(start, start + L).equals(needle)) {
        return start;
      }
    }
    return -1;
  }
```

}

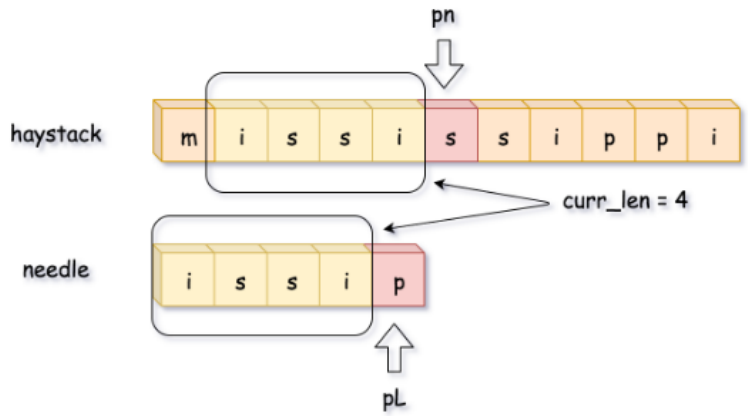
方法二：双指针 - 线性时间复杂度

上一个方法的缺陷是会将 haystack 所有长度为 L 的子串都与 needle 字符串比较，实际上是不需要这么做的。

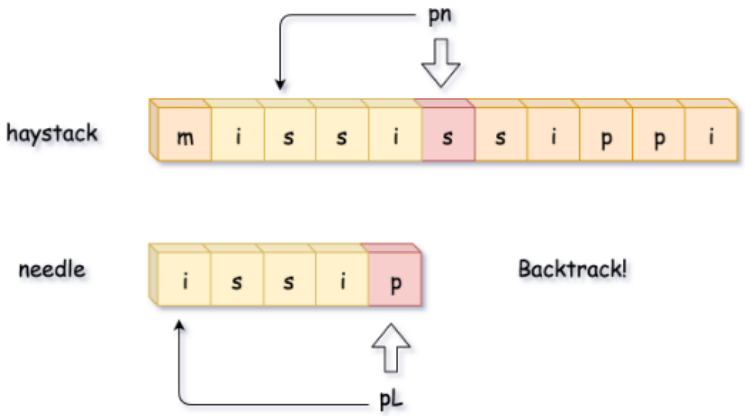
首先，只有子串的第一个字符跟 needle 字符串第一个字符相同的时候才需要比较。



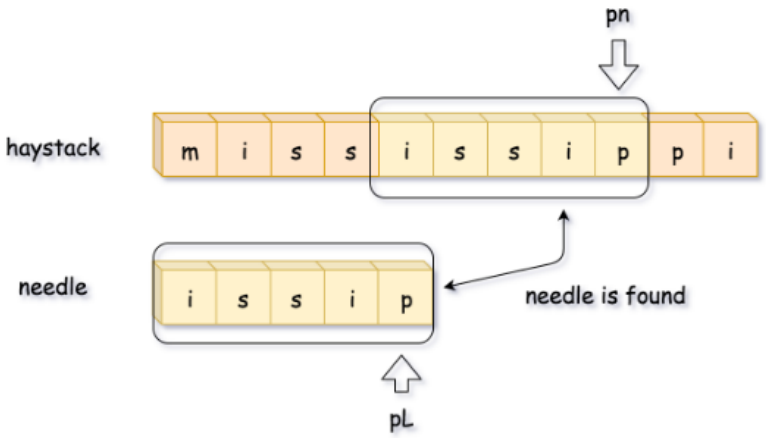
其次，可以一个字符一个字符比较，一旦不匹配了就立刻终止。



如下图所示，比较到最后一位时发现不匹配，这时候开始回溯。需要注意的是，pn 指针是移动到 pn = pn - curr_len + 1 的位置，而 *不是* pn = pn - curr_len 的位置。



这时候再比较一次，就找到了完整匹配的子串，直接返回子串的开始位置 pn - L。



算法

- 移动 pn 指针，直到 pn 所指向位置的字符与 needle 字符串第一个字符相等。
- 通过 pn，pL，curr_len 计算匹配长度。
- 如果完全匹配（即 curr_len == L），返回匹配子串的起始坐标（即 pn - L）。
- 如果不完全匹配，回溯。使 pn = pn - curr_len + 1， pL = 0， curr_len = 0。

复杂度分析

- 时间复杂度：最坏时间复杂度为 $O((N-L)L)$，最优时间复杂度为 $O(N)$。
- 空间复杂度：$O(1)$。

方法 2：双指针（2ms）

```
class Solution {
    public int strStr(String haystack, String needle) {
        int sourceLen = haystack.length();
        int targetLen = needle.length();
        // 注意特判
        if(targetLen == 0) return 0;
        // 若 target 长度大于 source，则不可能匹配
        if(targetLen > sourceLen) return -1;
        for (int i = 0; i < sourceLen - targetLen + 1; i++) {
            // 定义 k 来与 target 进行挨个比对
            int k = i;
            for (int j = 0; j < targetLen; j++) {
                if (haystack.charAt(k) == needle.charAt(j)) {
                    // 最后一个字符匹配完成，输出答案
                    if (j == targetLen - 1) return i;
                    k++;
                }
                // 其中一个字符无法匹配，直接退出做下一次循环
                else break;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            }
        }
        return -1;
    }
}
```

方法 3：KMP【含有详解】

https://leetcode-cn.com/problems/implement-strstr/solution/kmp-suan-fa-xiang-jie-by-labuladong/

一、KMP 算法概述

首先还是简单介绍一下 KMP 算法和暴力匹配算法的不同在哪里，难点在哪里，和动态规划有啥关系。

暴力的字符串匹配算法很容易写，看一下它的运行逻辑：

```
// 暴力匹配（伪码）
int search(String pat, String txt) {
    int M = pat.length;
    int N = txt.length;
    for (int i = 0; i < N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (pat[j] != txt[i+j])
                break;
        }
        // pat 全都匹配了
        if (j == M) return i;
    }
    // txt 中不存在 pat 子串
    return -1;
}
```

对于暴力算法，如果出现不匹配字符，同时回退 `txt` 和 `pat` 的指针，嵌套 for 循环，时间复杂度 $O(MN)$，空间复杂度 $O(1)$。最主要的问题是，如果字符串中重复的字符比较多，该算法就显得很蠢。

yes的练级攻略

比如 txt = "aaacaaab" pat = "aaab"：

很明显，`pat` 中根本没有字符 c，根本没必要回退指针 `i`，暴力解法明显多做了很多不必要的操作。

KMP 算法的不同之处在于，它会花费空间来记录一些信息，在上述情况中就会显得很聪明：

再比如类似的 txt = "aaaaaaab" pat = "aaab"，暴力解法还会和上面那个例子一样蠢蠢地回退指针 `i`，而 KMP 算法又会要聪明：

因为 KMP 算法知道字符 b 之前的字符 a 都是匹配的，所以每次只需要比较字符 b 是否被匹配就行了。

KMP 算法永不回退 `txt` 的指针 `i`，不走回头路（不会重复扫描 `txt`），而是借助 `dp` 数组中储存的信息把 `pat` 移到正确的位置继续匹配，时间复杂度只需 O(N)，用空间换时间，所以我认为它是一种动态规划算法。

KMP 算法的难点在于，如何计算 `dp` 数组中的信息？如何根据这些信息正确地移动 `pat` 的指针？这个就需要**确定有限状态自动机**来辅助了，别怕这种高大上的文学词汇，其实和动态规划的 `dp` 数组如出一辙，等你学会了也可以拿这个词去吓唬别人。

还有一点需要明确的是：**计算这个 `dp` 数组，只和 `pat` 串有关。**意思是说，只要给我个 `pat`，我就能通过这个模式串计算出 `dp` 数组，然后你可以给我不同的 `txt`，我都不怕，利用这个 `dp` 数组我都能在 O(N) 时间完成字符串匹配。

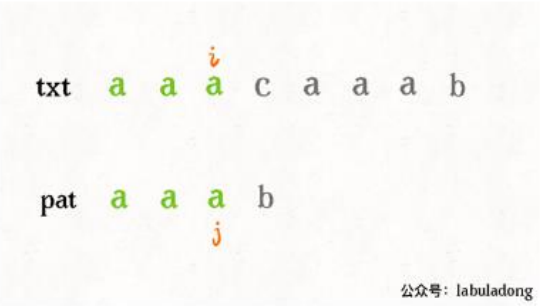
更多硬核内容就在公众号「yes 的练级攻略」

具体来说，比如上文举的两个例子：

```
txt1 = "aaacaaab"
pat = "aaab"
txt2 = "aaaaaaab"
pat = "aaab"
```

我们的 `txt` 不同，但是 `pat` 是一样的，所以 KMP 算法使用的 `dp` 数组是同一个。

只不过对于 `txt1` 的下面这个即将出现的未匹配情况：



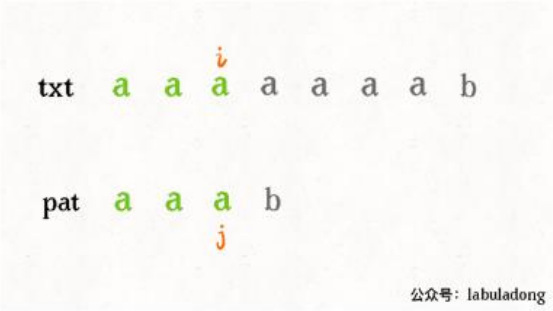
`dp` 数组指示 `pat` 这样移动：

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

PS：这个 `j` 不要理解为索引，它的含义更准确地说应该是**状态**（state），所以它会出现这个奇怪的位置，后文会详述。

而对于 `txt2` 的下面这个即将出现的未匹配情况：



`dp` 数组指示 `pat` 这样移动：

明白了 `dp` 数组只和 `pat` 有关，那么我们这样设计 KMP 算法就会比较漂亮：

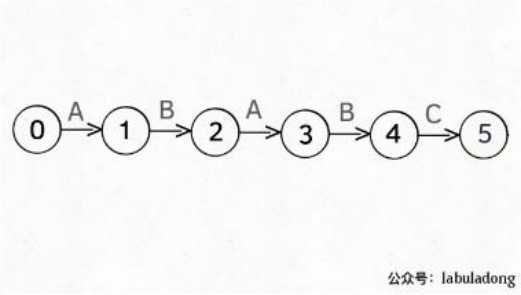
```
public class KMP {
    private int[][] dp;
    private String pat;

    public KMP(String pat) {
        this.pat = pat;
        // 通过 pat 构建 dp 数组
        // 需要 O(M) 时间
    }

    public int search(String txt) {
        // 借助 dp 数组去匹配 txt
        // 需要 O(N) 时间
    }
}
```

这样，当我们需要用同一 `pat` 去匹配不同 `txt` 时，就不需要浪费时间构造 `dp` 数组了：

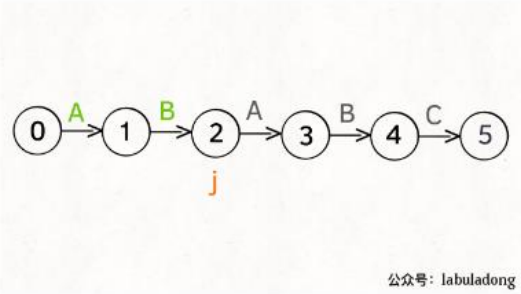
```
KMP kmp = new KMP("aaab");
int pos1 = kmp.search("aaacaaab"); //4
int pos2 = kmp.search("aaaaaaab"); //4
```

二、状态机概述

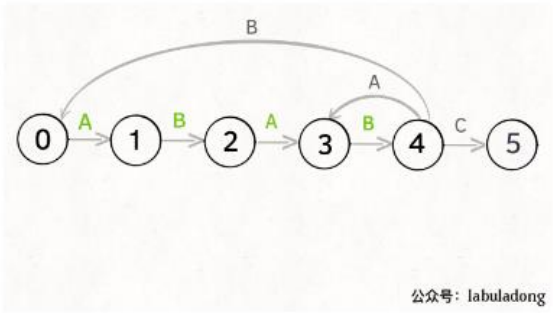
为什么说 KMP 算法和状态机有关呢？是这样的，我们可以认为 `pat` 的匹配就是状态的转移。比如当 pat = "ABABC"：



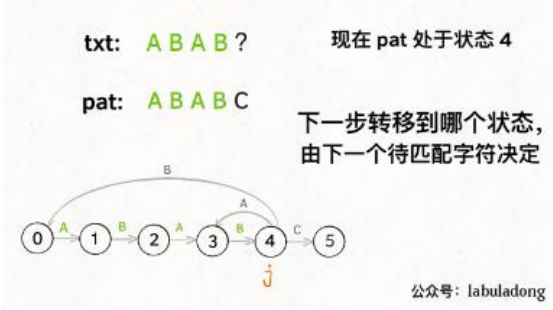
如上图，圆圈内的数字就是状态，状态 0 是起始状态，状态 5（`pat.length`）是终止状态。开始匹配时 `pat` 处于起始状态，一旦转移到终止状态，就说明在 `txt` 中找到了 `pat`。比如说当前处于状态 2，就说明字符 "AB" 被匹配：



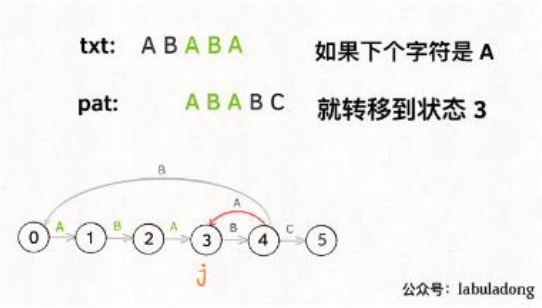
另外，处于不同状态时，`pat` 状态转移的行为也不同。比如说假设现在匹配到了状态 4，如果遇到字符 A 就应该转移到状态 3，遇到字符 C 就应该转移到状态 5，如果遇到字符 B 就应该转移到状态 0：



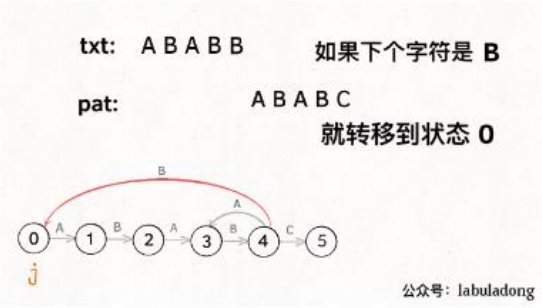
具体什么意思呢，我们来一个个举例看看。用变量 `j` 表示指向当前状态的指针，当前 `pat` 匹配到了状态 4：



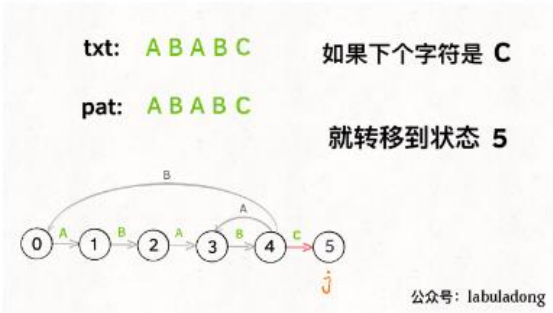
如果遇到了字符 "A"，根据箭头指示，转移到状态 3 是最聪明的：



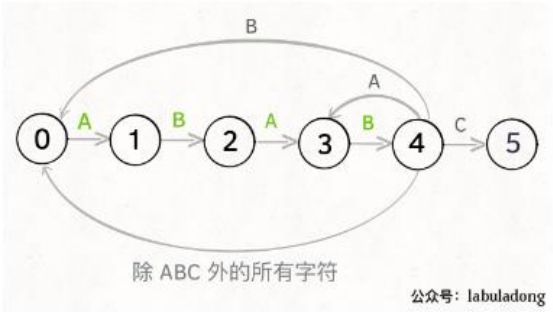
如果遇到了字符 "B"，根据箭头指示，只能转移到状态 0（一夜回到解放前）：



如果遇到了字符 "C"，根据箭头指示，应该转移到终止状态 5，这也就意味着匹配完成：

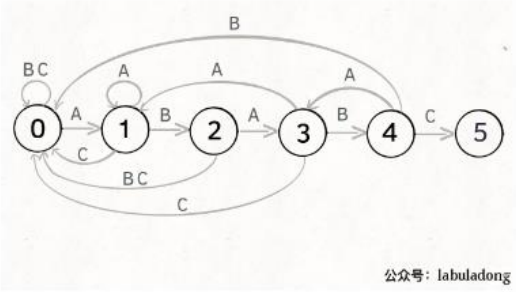


当然了，还可能遇到其他字符，比如 Z，但是显然应该转移到起始状态 0，因为 `pat` 中根本都没有字符 Z：



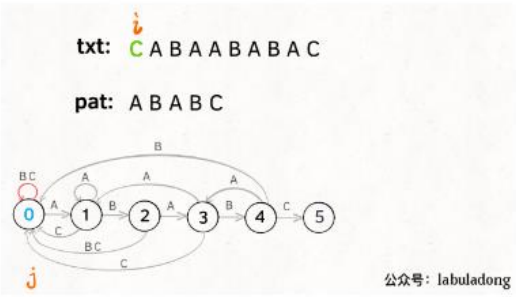
更多硬核内容就在公众号「yes 的练级攻略」

这里为了清晰起见，我们画状态图时就把其他字符转移到状态 0 的箭头省略，只画 `pat` 中出现的字符的状态转移：



KMP 算法最关键的步骤就是构造这个状态转移图。**要确定状态转移的行为，得明确两个变量，一个是当前的匹配状态，另一个是遇到的字符**；确定了这两个变量后，就可以知道这个情况下应该转移到哪个状态。

下面看一下 KMP 算法根据这幅状态转移图匹配字符串 `txt` 的过程：



请记住这个 GIF 的匹配过程，这就是 KMP 算法的核心逻辑！

为了描述状态转移图，我们定义一个二维 dp 数组，它的含义如下：

```
dp[j][c] = next
0 <= j < M，代表当前的状态
0 <= c < 256，代表遇到的字符（ASCII 码）
0 <= next <= M，代表下一个状态

dp[4]['A'] = 3 表示：
当前是状态 4，如果遇到字符 A，
pat 应该转移到状态 3

dp[1]['B'] = 2 表示：
当前是状态 1，如果遇到字符 B，
pat 应该转移到状态 2
```

更多硬核内容就在公众号「yes 的练级攻略」

根据我们这个 dp 数组的定义和刚才状态转移的过程，我们可以先写出 KMP 算法的 search 函数代码：

```
public int search(String txt) {
    int M = pat.length();
    int N = txt.length();
    // pat 的初始态为 0
    int j = 0;
    for (int i = 0; i < N; i++) {
        // 当前是状态 j，遇到字符 txt[i]，
        // pat 应该转移到哪个状态？
        j = dp[j][txt.charAt(i)];
        // 如果达到终止态，返回匹配开头的索引
        if (j == M) return i - M + 1;
    }
    // 没到达终止态，匹配失败
    return -1;
}
```

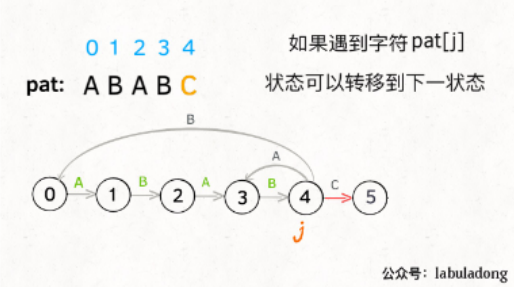
到这里，应该还是很好理解的吧，`dp` 数组就是我们刚才画的那幅状态转移图，如果不清楚的话回去看下 GIF 的算法演进过程。下面讲解：如何通过 `pat` 构建这个 `dp` 数组？

三、构建状态转移图

回想刚才说的：**要确定状态转移的行为，必须明确两个变量，一个是当前的匹配状态，另一个是遇到的字符**，而且我们已经根据这个逻辑确定了 `dp` 数组的含义，那么构造 `dp` 数组的框架就是这样：

```
for 0 <= j < M: # 状态
    for 0 <= c < 256: # 字符
        dp[j][c] = next
```

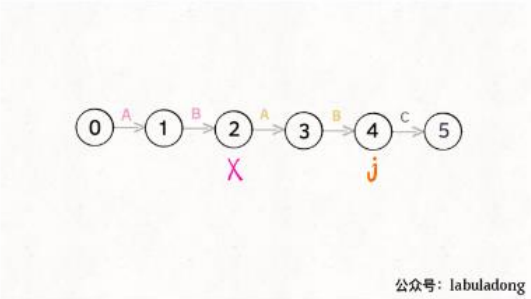
这个 next 状态应该怎么求呢？显然，**如果遇到的字符 `c` 和 `pat[j]` 匹配的话**，状态就应该向前推进一个，也就是说 `next = j + 1`，我们不妨称这种情况为**状态推进**：



更多硬核内容就在公众号「yes 的练级攻略」

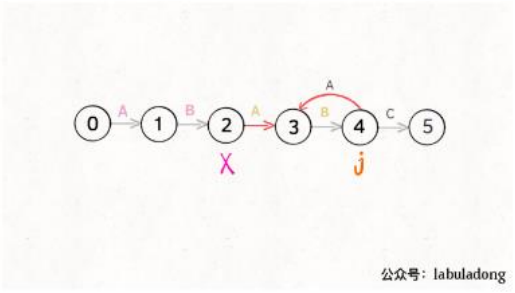
如果字符 `c` 和 `pat[j]` 不匹配的话，状态就要回退（或者原地不动），我们不妨称这种情况为**状态重启**：

那么，如何得知在哪个状态重启呢？解答这个问题之前，我们再定义一个名字：**影子状态**（我编的名字），用变量 `X` 表示。**所谓影子状态，就是和当前状态具有相同的前缀。**比如下面这种情况：

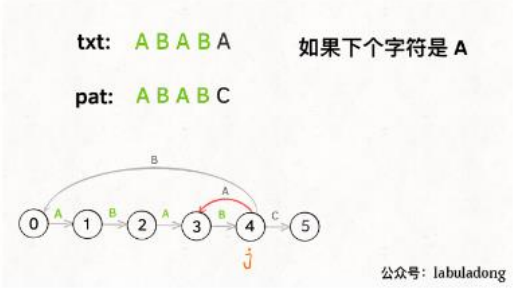


当前状态 `j = 4`，其影子状态为 `X = 2`，它们都有相同的前缀 "AB"。因为状态 `X` 和状态 `j` 存在相同的前缀，所以当状态 `j` 准备进行状态重启的时候（遇到的字符 `c` 和 `pat[j]` 不匹配），可以通过 `X` 的状态转移图来获得**最近的重启位置**。

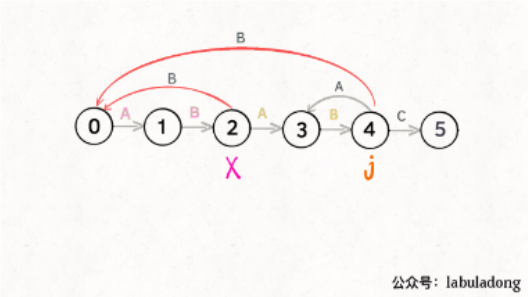
比如说刚才的情况，如果状态 `j` 遇到一个字符 "A"，应该转移到哪里呢？首先只有遇到 "C" 才能推进状态，遇到 "A" 显然只能进行状态重启。**状态 `j` 会把这个字符委托给状态 `X` 处理，也就是 `dp[j]['A'] = dp[X]['A']`**：



为什么这样可以呢？因为：既然 `j` 这边已经确定字符 "A" 无法推进状态，**只能回退**，而且 KMP 就是要**尽可能少的回退**，以免多余的计算。那么 `j` 就可以去问问和自己具有相同前缀的 `X`，如果 `X` 遇见 "A" 可以进行「状态推进」，那就转移过去，因为这样回退最少。



当然，如果遇到的字符是 "B"，状态 `X` 也不能进行「状态推进」，只能回退，`j` 只要跟着 `X` 指引的方向回退就行了：



你也许会问，这个 `X` 怎么知道遇到字符 "B" 要回退到状态 0 呢？因为 `X` 永远跟在 `j` 的身后，状态 `X` 如何转移，在之前就已经算出来了。动态规划算法不就是利用过去的结果解决现在的问题吗？

这样，我们就细化一下刚才的框架代码：

```
int X # 影子状态
for 0 <= j < M:
    for 0 <= c < 256:
        if c == pat[j]:
            # 状态推进
            dp[j][c] = j + 1
        else:
            # 状态重启
            # 委托 X 计算重启位置
            dp[j][c] = dp[X][c]
```

四、代码实现

如果之前的内容你都能理解，恭喜你，现在就剩下一个问题：影子状态 `X` 是如何得到的呢？下面先直接看完整代码吧。

```
public class KMP {
    private int[][] dp;
    private String pat;

    public KMP(String pat) {
        this.pat = pat;
        int M = pat.length();
        // dp[状态][字符] = 下个状态
        dp = new int[M][256];
        // base case
        dp[0][pat.charAt(0)] = 1;
        // 影子状态 X 初始为 0
        int X = 0;
        // 当前状态 j 从 1 开始
        for (int j = 1; j < M; j++) {
            for (int c = 0; c < 256; c++) {
                if (pat.charAt(j) == c)
                    dp[j][c] = j + 1;
                else
                    dp[j][c] = dp[X][c];
            }
            // 更新影子状态
            X = dp[X][pat.charAt(j)];
        }
    }

    public int search(String txt) {...}
}
```

先解释一下这一行代码：

```
// base case
dp[0][pat.charAt(0)] = 1;
```

这行代码是 base case，只有遇到 pat[0] 这个字符才能使状态从 0 转移到 1，遇到其它字符的话还是停留在状态 0（Java 默认初始化数组全为 0）。

影子状态 `X` 是先初始化为 0，然后随着 `j` 的前进而不断更新的。下面看看到底应该**如何更新影子状态 `X`**：

```
int X = 0;
for (int j = 1; j < M; j++) {
    ...
    // 更新影子状态
    // 当前是状态 X，遇到字符 pat[j]，
    // pat 应该转移到哪个状态？
    X = dp[X][pat.charAt(j)];
}
```

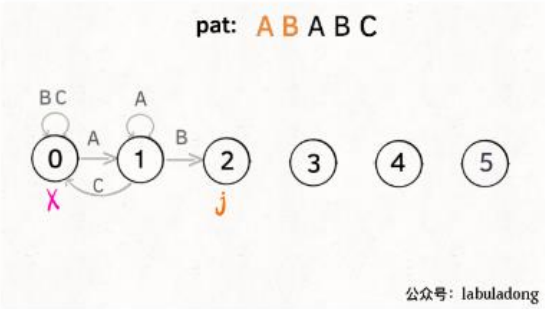
更新 `X` 其实和 `search` 函数中更新状态 `j` 的过程是非常相似的：

```
int j = 0;
for (int i = 0; i < N; i++) {
    // 当前是状态 j，遇到字符 txt[i]，
    // pat 应该转移到哪个状态？
    j = dp[j][txt.charAt(i)];
    ...
}
```

其中的原理非常微妙，注意代码中 for 循环的变量初始值，可以这样理解：后者是在 `txt` 中匹配 `pat`，前者是在 `pat` 中匹配 `pat[1..end]`，状态 `X` 总是落后状态 `j` 一个状态，与 `j` 具有最长的相同前缀。所以我把 `X` 比喻为影子状态，似乎也有一点贴切。

另外，构建 dp 数组是根据 base case `dp[0][..]` 向后推演。这就是我认为 KMP 算法就是一种动态规划算法的原因。

下面来看一下状态转移图的完整构造过程，你就能理解状态 `X` 作用之精妙了：



至此，KMP 算法的核心终于写完啦啦啦啦！看下 KMP 算法的完整代码吧：

KMP 算法完整代码

```
public class KMP {
    private int[][] dp;
    private String pat;

    public KMP(String pat) {
        this.pat = pat;
        int M = pat.length();
```

```
        // dp[状态][字符] = 下个状态
        dp = new int[M][256];
        // base case
        dp[0][pat.charAt(0)] = 1;
        // 影子状态 X 初始为 0
        int X = 0;
        // 构建状态转移图（稍改的更紧凑了）
        for (int j = 1; j < M; j++) {
            for (int c = 0; c < 256; c++) {
                dp[j][c] = dp[X][c];
            dp[j][pat.charAt(j)] = j + 1;
            // 更新影子状态
            X = dp[X][pat.charAt(j)];
        }
    }

    public int search(String txt) {
        int M = pat.length();
        int N = txt.length();
        // pat 的初始态为 0
        int j = 0;
        for (int i = 0; i < N; i++) {
            // 计算 pat 的下一个状态
            j = dp[j][txt.charAt(i)];
            // 到达终止态，返回结果
            if (j == M) return i - M + 1;
        }
        // 没到达终止态，匹配失败
        return -1;
    }
}
```

禁止商用
公众号：
yes的练级攻略

经过之前的详细举例讲解，你应该可以理解这段代码的含义了，当然你也可以把 KMP 算法写成一个函数。核心代码也就是两个函数中 for 循环的部分，数一下有超过十行吗？

五、最后总结

传统的 KMP 算法是使用一个一维数组 `next` 记录前缀信息，而本文是使用一个二维数组 `dp` 以状态转移的角度解决字符匹配问题，但是空间复杂度仍然是 O(256M) = O(M)。

在 `pat` 匹配 `txt` 的过程中，只要明确了「当前处在哪个状态」和「遇到的字符是什么」这两个问题，就可以确定应该转移到哪个状态（推进或回退）。

对于一个模式串 `pat`，其总共就有 M 个状态，对于 ASCII 字符，总共不会超过 256 种。所以我们就构造一个数组 `dp[M][256]` 来包含所有情况，并且明确 `dp` 数组的含义：

`dp[j][c] = next` 表示，当前是状态 `j`，遇到了字符 `c`，应该转移到状态 `next`。

明确了其含义，就可以很容易写出 search 函数的代码。

对于如何构建这个 `dp` 数组，需要一个辅助状态 `X`，它永远比当前状态 `j` 落后一个状态，拥有和 `j` 最长的相同前缀，我们给它起了个名字叫「影子状态」。

在构建当前状态 `j` 的转移方向时，只有字符 `pat[j]` 才能使状态推进（`dp[j][pat[j]] = j+1`）；而对于其他字符只能进行状态回退，应该去请教影子状态 `X` 应该回退到哪里（`dp[j][other] = dp[X][other]`，其中 `other` 是除了 `pat[j]` 之外所有字符）。

对于影子状态 `X`，我们把它初始化为 0，并且随着 `j` 的前进进行更新，更新的方式和 search 过程更新 `j` 的过程非常相似（`X = dp[X][pat[j]]`）。

KMP 算法也就是动态规划那点事，我们的公众号文章目录有一系列专门讲动态规划的，而且都是按照一套框架来的，无非就是描述问题逻辑，明确 `dp` 数组含义，定义 base case 这点破事。希望这篇文章能让大家对动态规划有更深的理解。

最后，点击我的头像可以查看更多详细题解，希望读者多多点赞，让我感受到你的认可~

这题具体的 KMP 代码（24ms）

```
class Solution {
    public int strStr(String haystack, String needle) {
        if(needle.equals("")) {
            return 0;
        }
        if(haystack.equals("")) {
            return -1;
        }

        // 构造 KMP 中的 dp 矩阵
        int m = needle.length();
        // 各个状态(行)遇到下一个字符(列)跳转到哪个状态
        int[][] dp = new int[m][256];
        // 影子状态
        int X = 0;
        dp[0][needle.charAt(0)] = 1;
        for (int i = 1; i < m; i++) {
            for (int j = 0; j < 256; j++) {
                //假设下个字符不匹配，此时要回去看影子状态，从而得知跳转到哪个状态
                dp[i][j] = dp[X][j];
            }
```

```
            // 只有 pat 上 i 的字符匹配，跳转到下个状态
            dp[i][needle.charAt(i)] = i + 1;
            // 更新影子状态
            X = dp[X][needle.charAt(i)];
        }

        // 构造 dp 完成后，开始 search
        // 初始状态为 0
        int s = 0;
        for (int i = 0; i < haystack.length(); i++) {
            s = dp[s][haystack.charAt(i)];
            if (s == m) {
                return i - m + 1;
            }
        }

        // 匹配失败，返回-1
        return -1;
    }
}
```

58、找到所有好字符串（1397）

1397. 找到所有好字符串

难度 困难 40 收藏 分享 切换为英文 关注 反馈

给你两个长度为 `n` 的字符串 `s1` 和 `s2` ，以及一个字符串 `evil` 。请你返回 **好字符串** 的数目。

好字符串 的定义为：它的长度为 `n` ，字典序大于等于 `s1` ，字典序小于等于 `s2` ，且不包含 `evil` 为子字符串。

由于答案可能很大，请你返回答案对 10^9 + 7 取余的结果。

示例 1：

输入： n = 2, s1 = "aa", s2 = "da", evil = "b"
输出： 51
解释： 总共有 25 个以 'a' 开头的好字符串："aa", "ac", "ad", ..., "az"。还有 25 个以 'c' 开头的好字符串："ca", "cc", "cd", ..., "cz"。最后，还有一个以 'd' 开头的好字符串："da"。

示例 2：

输入： n = 8, s1 = "leetcode", s2 = "leetgoes", evil = "leet"
输出： 0
解释： 所有字典序大于等于 s1 且小于等于 s2 的字符串都以 evil 字符串 "leet" 开头。所以没有好字符串。

示例 3：

输入： n = 2, s1 = "gx", s2 = "gz", evil = "x"
输出： 2

提示：

- `s1.length == n`
- `s2.length == n`
- `s1 <= s2`
- `1 <= n <= 500`
- `1 <= evil.length <= 50`
- 所有字符串都只包含小写英文字母。

https://leetcode-cn.com/problems/find-all-good-strings/solution/java-ji-yi-hua-di-gui-kmp-pi-pei-

by-aerysn/

前面已经有人指出此题涉及到数位 DP 的知识，数位 DP 的介绍和模板可以看此链接：https://oi-wiki.org/dp/number/。

对于字典序大于等于 `s1` 小于等 `s2` 区间内查找好字符串的问题，可以将其转化为字典序小于等于 `s2` 的区间内好字符串的个数减去字典序小于 `s1` 的区间内好字符串的个数。

方便起见我们将字典序上界字符串称为 `s`，坏字符串称为 `evil`，当前正在尝试的字符串为 `current`。我们维护一个 `dp[index][match]` 二维表来表示当前可能的好字符串的构建状态，其中 `index` 表示当前字符串 `current` 正待确定 `current[index]` 位（即 `[0, ..., index-1]` 已经确定）。`match` 表示 `evil` 字符串正待匹配 `evil[match]` 位（即 `evil[0, ..., match-1]` 位已经与 `current[index-match, ..., index-1]` 匹配）。除此以外数位 DP 还有一个重要的属性 `threshold`（也就是上面链接里模板的 `op`），表示 `current` 与 `s` 当前确定的前缀之间的关系，如果 `current[0, ..., index-1] == s[0, ..., index-1]` 则 `threshold` 为真，表示当前确定的前缀相等。

基于上述构造，我们可以很容易地写出状态间的转移关系，首先通过 `evil` 字符串求出其 KMP 算法中的 `next` 数组，`nextMatchPoint` 表示当前尝试的匹配位点为 `match` 时下一个尝试的匹配位点，当 `match` 匹配到 `evil` 尾部时表示 `current` 为坏字符串，不用再递归其子问题了。所以对于 `current` 字符串当前 `index` 位的任意可行字符 `c`，`dp[index][match]` 的值为 `dp[index+1][nextMatchIndex(evil, match, c)]` 求和。

完整代码如下，注意并没有显式地构造 `current`，因为其内容不重要，我们关心的是 `index` 和 `match`：

```
class Solution {
    int[][] dp;
    int[] next; // KMP 算法中的 next 数组
    int mod = (int) 1e9 + 7;
    int[] getNext(String s) {
        int[] next = new int[s.length() + 1];
        int i = 0, j = -1;
        next[0] = -1;
        while (i < s.length()) {
        if (j == -1 || s.charAt(j) == s.charAt(i))
            next[++i] = ++j;
        else
            j = next[j];
        }
        next[0] = 0; // 此处将 next[0] 置为 0 方便逻辑统一
        return next;
    }
    // 对于字符串 pattern, 当前正在尝试用 c 匹配 pattern[index], 求下一个尝试的匹配位点
    int nextMatchPoint(char[] pattern, int index, char c) {
        while (index > 0 && pattern[index] != c)
        index = next[index]; // 不匹配, 下一个匹配位点为 next[index]
        if (pattern[index] == c)
        index++; // 匹配, 下一个匹配位点为 index + 1
        return index;
    }
    int dfs(char[] s, int index, char[] evil, int match, boolean threshold) {
        int n = s.length;
        if (index == n)
        return 1; // current
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (!threshold && dp[index][match] != 0)
        return dp[index][match];
        char max = threshold ? s[index] : 'z'; // 如果现有前缀小于 s 中对应前
缀，则 a~z 都可以取，否则只能取 a~s[index]
        int result = 0;
        for (char c = 'a'; c <= max; c++) {
        if (evil[match] == c && match == evil.length - 1) // current 中出现了坏
字符串
            continue;
        result += dfs(s, index + 1, evil, nextMatchPoint(evil, match, c), thres
hold && c == max);
        result %= mod;
        }
        if (!threshold)
        dp[index][match] = result; // 记忆化
        return result;
    }
    public int findGoodStrings(int n, String s1, String s2, String evil) {
        next = getNext(evil);
        dp = new int[s1.length()][evil.length()];
        int v1 = dfs(s1.toCharArray(), 0, evil.toCharArray(), 0, true);
        dp = new int[s2.length()][evil.length()];
        int v2 = dfs(s2.toCharArray(), 0, evil.toCharArray(), 0, true);
        return (v2 - v1 + (s1.contains(evil) ? 0 : 1) + mod) % mod; // 因
为 v2 - v1 是左开右闭，需要特判 s1 是否可行
    }
}
```

59、外观数列（38）

yes的练级攻略

38. 外观数列

难度 简单 513 收藏 分享 切换为英文 关注 反馈

给定一个正整数 n（$1 \le n \le 30$），输出外观数列的第 n 项。

注意：整数序列中的每一项将表示为一个字符串。

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。前五项如下：

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

第一项是数字 1

描述前一项，这个数是 `1` 即 “一个 1”，记作 `11`

描述前一项，这个数是 `11` 即 “两个 1”，记作 `21`

描述前一项，这个数是 `21` 即 “一个 2 一个 1”，记作 `1211`

描述前一项，这个数是 `1211` 即 “一个 1 一个 2 两个 1”，记作 `111221`

示例 1:

输入： 1
输出： "1"
解释： 这是一个基本样例。

示例 2:

输入： 4
输出： "1211"
解释： 当 n = 3 时，序列是 "21"，其中我们有 "2" 和 "1" 两组，"2" 可以读作 "12"，也就是出现频次 = 1 而 值 = 2；类似 "1" 可以读作 "11"。所以答案是 "12" 和 "11" 组合在一起，也就是 "1211"。

https://leetcode-cn.com/problems/count-and-say/solution/1-ms-zai-suo-you-java-ti-jiao-zhong-ji-bai-liao-97/

方法 1：递归（1ms）

```
class Solution {
    public String countAndSay(int n) {
        if (n == 1) return "1";
        StringBuffer res = new StringBuffer();
        String str = countAndSay(n - 1);
        int length = str.length();
        int a = 0;
        for (int i = 1; i < length + 1; i++) {
            if (i == length) {
                return res.append(i - a).append(str.charAt(a)).toString
();
            } else if (str.charAt(i) != str.charAt(a) ) {
                res.append(i - a).append(str.charAt(a));
                a = i;
            }
        }
        return res.toString();
    }
```

```
}
```

方法 2：迭代（4ms）

```
class Solution {
    public  String countAndSay(int n) {
        // 只需要知道前一个就可以进行描述 - 要得到n 项，只需要知道n-1 项
        String str = "1"; // 存储前一项所描述的信息
        StringBuilder stringBuilder = new StringBuilder();
        int i, j = 0;
        for(int k = 1; k < n; k++){
            for(i = j + 1; i < str.length(); i++){
                if (str.charAt(i) != str.charAt(j)){
                    // 进行添加
                    stringBuilder.append(i - j).append(str.charAt(j));
                    j = i;
                }
            }
            // 处理只有一个元素和最后一个元素
            stringBuilder.append(i - j).append(str.charAt(j));
            j = 0;
            str = stringBuilder.toString();
            stringBuilder.delete(0, stringBuilder.length());
        }
        System.out.println(str);
        return str;
    }
}
```

60、机器人能否返回原点（657）

657. 机器人能否返回原点

难度 简单 149 收藏 分享 切换为英文 关注 反馈

在二维平面上，有一个机器人从原点 (0, 0) 开始。给出它的移动顺序，判断这个机器人在完成移动后是否在 **(0, 0) 处结束**。

移动顺序由字符串表示。字符 move[i] 表示其第 i 次移动。机器人的有效动作有 `R`（右），`L`（左），`U`（上）和 `D`（下）。如果机器人在完成所有动作后返回原点，则返回 true。否则，返回 false。

注意：机器人"面朝"的方向无关紧要。 "R" 将始终使机器人向右移动一次，"L" 将始终向左移动等。此外，假设每次移动机器人的移动幅度相同。

示例 1:

```
输入："UD"
输出：true
解释：机器人向上移动一次，然后向下移动一次。所有动作都具有相同的幅度，因此它最终回到它开始的原点。因此，我们返回 true。
```

示例 2:

```
输入："LL"
输出：false
解释：机器人向左移动两次。它最终位于原点的左侧，距原点有两次 "移动" 的距离。我们返回 false，因为它在移动结束时没有返回原点。
```

```
class Solution {
    public boolean judgeCircle(String moves) {
        if (moves.length() == 0) return true;
```

```
        if ((moves.length() & 1) == 1) return false;
        int R = 0, U = 0;
        for (char item : moves.toCharArray()) {
            if (item == 'R') R++;
            else if (item == 'L') R--;
            else if (item == 'U') U++;
            else if (item == 'D') U--;
        }
        return R == 0 && U == 0;
    }
}
```

61、字符串相加（415）

415. 字符串相加

难度 简单 189 收藏 分享 切换为英文 关注 反馈

给定两个字符串形式的非负整数 num1 和 num2 ，计算它们的和。

注意：

1. num1 和 num2 的长度都小于 5100.
2. num1 和 num2 都只包含数字 0-9 .
3. num1 和 num2 都不包含任何前导零。
4. 你不能使用任何内建 BigInteger 库， 也不能直接将输入的字符串转换为整数形式。

https://leetcode-cn.com/problems/add-strings/solution/add-strings-shuang-zhi-zhen-fa-by-jyd/

```
class Solution {
    public String addStrings(String num1, String num2) {
        StringBuilder sb = new StringBuilder();
        int carry = 0;
        for (int i = num1.length() - 1, j = num2.length() - 1;
        i >= 0 || j >= 0 || carry != 0;
        i--, j--){
            int x = i < 0 ? 0 : num1.charAt(i) - '0';
            int y = j < 0 ? 0 : num2.charAt(j) - '0';
            int sum = (x + y + carry) % 10;
            sb.append(sum);
            carry = (x + y + carry) / 10;
        }
        return sb.reverse().toString();
    }
}
```

62、字符串相乘（43）

更多硬核内容就在公众号「yes 的练级攻略」

43. 字符串相乘

难度 中等 399 收藏 分享 切换为英文 关注 反馈

给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

示例 1:

```
输入: num1 = "2", num2 = "3"
输出: "6"
```

示例 2:

```
输入: num1 = "123", num2 = "456"
输出: "56088"
```

说明:

1. num1 和 num2 的长度小于110。
2. num1 和 num2 只包含数字 0-9。
3. num1 和 num2 均不以零开头，除非是数字 0 本身。
4. **不能使用任何标准库的大数类型（比如 BigInteger）或直接将输入转换为整数来处理。**

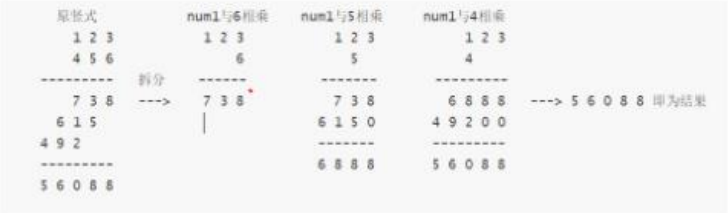
https://leetcode-cn.com/problems/multiply-strings/solution/you-hua-ban-shu-shi-da-bai-994-by-breezean/

禁止商用

方法一：普通竖式

思路

竖式运算思想，以 num1 为 123，num2 为 456 为例分析：



遍历 num2 每一位与 num1 进行相乘，将每一步的结果进行累加。

注意:

- num2 除了第一位的其他位与 num1 运算的结果需要 补0
- 计算字符串数字累加其实就是 415. 字符串相加

复杂度分析

- 时间复杂度：$O(MN)$。M, N 分别为 num1 和 num2 的长度。
- 空间复杂度：O(M+N)。用于存储计算结果。

方法 1：普通竖式

```
class Solution {
    public String multiply(String num1, String num2) {
        if (num1.equals("0") || num2.equals("0")) return "0";
```

更多硬核内容就在公众号「yes 的练级攻略」

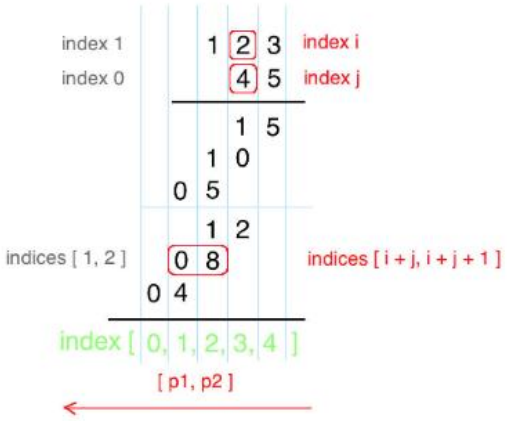
```
        String res = "0";
        for (int i = num2.length() - 1; i >= 0; i--){
            int carry = 0;
            StringBuilder tmp = new StringBuilder();
            for (int k = 0; k < num2.length() - 1 - i; k++){
                tmp.append(0);
            }
            int n2 = num2.charAt(i) - '0';
            for (int j = num1.length() - 1; j >= 0 || carry != 0; j--){
                int n1 = j < 0 ? 0 : num1.charAt(j) - '0';
                int product = (n1 * n2 + carry) % 10;
                tmp.append(product);
                carry = (n1 * n2 + carry) / 10;
            }
            res = addStrings(res, tmp.reverse().toString());
        }
        return res;
    }
    //与415 中字符串相加的代码一样
    public String addStrings(String num1, String num2) {
        StringBuilder sb = new StringBuilder();
        int carry = 0;
        for (int i = num1.length() - 1, j = num2.length() - 1;
        i >= 0 || j >= 0 || carry != 0;
        i--, j--){
            int x = i < 0 ? 0 : num1.charAt(i) - '0';
            int y = j < 0 ? 0 : num2.charAt(j) - '0';
            int sum = (x + y + carry) % 10;
            sb.append(sum);
            carry = (x + y + carry) / 10;
        }
        return sb.reverse().toString();
    }
}
```

方法二：优化竖式

该算法是通过两数相乘时，乘数某位与被乘数某位相乘，与产生结果的位置的规律来完成。具体规律如下：

- 乘数 `num1` 位数为 M，被乘数 `num2` 位数为 N，`num1 x num2` 结果 `res` 最大总位数为 **M+N**
- `num1[i] x num2[j]` 的结果为 `tmp`（位数为两位，"0x","xy"的形式），其第一位位于 `res[i+j]`，第二位位于 `res[i+j+1]`。

结合下图更容易理解



复杂度分析

- 时间复杂度：$O*(*MN*)$。M, N 分别为 `num1` 和 `num2` 的长度。
- 空间复杂度：$O*(M+N)$。用于存储计算结果。

算法比较

算法	提交结果	执行用时	内存消耗	语言
普通竖式	通过	48 ms	43.8 MB	Java
优化竖式	通过	4 ms	36.6 MB	Java

虽然两者时间复杂度和空间复杂度相同，但优化竖式执行速度提高很明显，普通竖式耗时主要还是对每步计算的字符串相加这个过程。

方法 2：优化的竖式（去除相加的函数）

```
class Solution {
    public String multiply(String num1, String num2) {
        if (num1.equals("0") || num2.equals("0")) return "0";
        int[] res = new int[num1.length() + num2.length()];
        for (int i = num1.length() - 1; i >= 0; i--){
            int n1 = num1.charAt(i) - '0';
            for (int j = num2.length() - 1; j >= 0; j--){
                int n2 = num2.charAt(j) - '0';
                int sum = (n1 * n2 + res[i + j + 1]);
                res[i + j + 1] = sum % 10;
                res[i + j] += sum /10;
            }
        }
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < res.length; i++){
```

```
            if (i == 0 && res[i] == 0) continue;
            sb.append(res[i]);
        }
        return sb.toString();
    }
}
```

63、井字游戏（面试题 16.04）

面试题 16.04. 井字游戏

难度 中等 13 收藏 分享 切换为英文 关注 反馈

设计一个算法，判断玩家是否赢了井字游戏。输入是一个 N x N 的数组棋盘，由字符" "，"X"和"O"组成，其中字符" "代表一个空位。

以下是井字游戏的规则：

- 玩家轮流将字符放入空位（" "）中。
- 第一个玩家总是放字符"O"，且第二个玩家总是放字符"X"。
- "X"和"O"只允许放置在空位中，不允许对已放有字符的位置进行填充。
- 当有N个相同（且非空）的字符填充任何行、列或对角线时，游戏结束，对应该字符的玩家获胜。
- 当所有位置非空时，也算为游戏结束。
- 如果游戏结束，玩家不允许再放置字符。

如果游戏存在获胜者，就返回该游戏的获胜者使用的字符（"X"或"O"）；如果游戏以平局结束，则返回 "Draw"；如果仍会有行动（游戏未结束），则返回 "Pending"。

示例 1：

```
输入： board = ["O X"," XO","X O"]
输出： "X"
```

示例 2：

```
输入： board = ["OOX","XXO","OXO"]
输出： "Draw"
解释： 没有玩家获胜且不存在空位
```

示例 3：

```
输入： board = ["OOX","XXO","OX "]
输出： "Pending"
解释： 没有玩家获胜且仍存在空位
```

提示：

- `1 <= board.length == board[i].length <= 100`
- 输入一定遵循井字棋规则

https://leetcode-cn.com/problems/tic-tac-toe-lcci/solution/javanei-cun-zhan-sheng-100rong-yi-li-jie-suan-fa-b/

```
class Solution {
    public String tictactoe(String[] board) {
        int len = board.length;
        if(len == 1) return String.valueOf(board[0].charAt(0));
        for(int i = 0;i<len;i++){//行
            for(int j = 0;j<len-1;j++){
                if(!String.valueOf(board[i].charAt(j)).equals(String.va
lueOf(board[i].charAt(j+1)))
                ||String.valueOf(board[i].charAt(j)).equals(" ")){
                    break;
                }
```

```
                if(j == len-2) return String.valueOf(board[i].charAt(0)
);
            }
        }
        for(int i = 0;i<len;i++) {//列
            for (int j = 0; j < len - 1; j++) {
                if(!String.valueOf(board[j].charAt(i)).equals(String.va
lueOf(board[j+1].charAt(i)))
                ||String.valueOf(board[j].charAt(i)).equals(" ")){
                    break;
                }
                if(j+1 == len-1) return String.valueOf(board[0].charAt(
i));
            }

        }
        for(int i = 0;i<len-1;i++){//一三象限角平分线
            if(!String.valueOf(board[i].charAt(i)).equals(String.valueO
f(board[i+1].charAt(i+1)))
            ||String.valueOf(board[i].charAt(i)).equals(" ")){
                break;
            }
            if(i+1 == len-1) return String.valueOf(board[0].charAt(0));
        }
        for(int i = 0;i<len-1;i++){//二四象限角平分线
            if(!String.valueOf(board[len-i-1].charAt(i)).equals(String.
valueOf(board[len-i-2].charAt(i+1)))
            ||String.valueOf(board[len-i-1].charAt(i)).equals(" ")){
                break;
            }
            int k = i+1;
            if(k == len-1) return String.valueOf(board[len-1].charAt(0)
);

        }
        for(int i = 0;i<len;i++){
            for(int j = 0;j<len;j++) {
                if (String.valueOf(board[i].charAt(j)).equals(" "))
                    return "Pending";
            }
        }
        return "Draw";
    }
}
```

64、Z 字形变换（6）

6. Z 字形变换

难度 中等 776 收藏 分享 切换为英文 关注 反馈

将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 "LEETCODEISHIRING" 行数为 3 时，排列如下：

```
L   C   I   R
E T O E S I I G
E   D   H   N
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如："LCIRETOESIIGEDHN"。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1:

```
输入: s = "LEETCODEISHIRING", numRows = 3
输出: "LCIRETOESIIGEDHN"
```

示例 2:

```
输入: s = "LEETCODEISHIRING", numRows = 4
输出: "LDREOEIIECIHNTSG"
解释:

L     D     R
E   O E   I I
E C   I H   N
T     S     G
```

https://leetcode-cn.com/problems/zigzag-conversion/solution/zzi-xing-bian-huan-by-jyd/

解题思路：

- **题目理解：**
  - 字符串 `s` 是以 Z 字形为顺序存储的字符串，目标是按行打印。
  - 设 `numRows` 行字符串分别为 s_1 , s_2 ,..., s_n，则容易发现：按顺序遍历字符串 `s` 时，每个字符 `c` 在 Z 字形中对应的 **行索引** 先从 s_1 增大至 s_n，再从 s_n 减小至 s_1 ..... 如此反复。
  - 因此，解决方案为：模拟这个行索引的变化，在遍历 `s` 中把每个字符填到正确的行 `res[i]` 。
- **算法流程：** 按顺序遍历字符串 `s` ；
  1. `res[i] += c` ： 把每个字符 `c` 填入对应行 s_i；
  2. `i += flag` ： 更新当前字符 `c` 对应的行索引；
  3. `flag = - flag` ： 在达到 Z 字形转折点时，执行反向。
- **复杂度分析：**
  - 时间复杂度 $O(N)$ ： 遍历一遍字符串 `s` ；
  - 空间复杂度 $O(N)$ ： 各行字符串共占用 $O(N)$ 额外空间。

```java
class Solution {
    public String convert(String s, int numRows) {
        if (numRows < 2) return s;
        List<StringBuilder> rows = new ArrayList<>();
        for (int i = 0; i < numRows; i++) {
            rows.add(new StringBuilder());
        }
```

```
            int i = 0, flag = -1;
            for (char c : s.toCharArray()){
                rows.get(i).append(c);
                if (i == 0 || i == numRows - 1) flag = - flag;
                i += flag;
            }
            StringBuilder res = new StringBuilder();
            for (StringBuilder row : rows) {
                res.append(row);
            }
            return res.toString();
        }
}
```

65、比较版本号（165）

165. 比较版本号

难度 中等 96 收藏 分享 切换为英文 关注 反馈

比较两个版本号 *version1* 和 *version2*。
如果 *version1* > *version2* 返回 1，如果 *version1* < *version2* 返回 -1，除此之外返回 0。

你可以假设版本字符串非空，并且只包含数字和 . 字符。

. 字符不代表小数点，而是用于分隔数字序列。

例如，2.5 不是"两个半"，也不是"差一半到三"，而是第二版中的第五个小版本。

你可以假设版本号的每一级的默认修订版号为 0。例如，版本号 3.4 的第一级（大版本）和第二级（小版本）修订号分别为 3 和 4。其第三级和第四级修订号均为 0。

示例 1:

```
输入: version1 = "0.1", version2 = "1.1"
输出: -1
```

示例 2:

```
输入: version1 = "1.0.1", version2 = "1"
输出: 1
```

示例 3:

```
输入: version1 = "7.5.2.4", version2 = "7.5.3"
输出: -1
```

示例 4:

```
输入: version1 = "1.01", version2 = "1.001"
输出: 0
解释: 忽略前导零，“01” 和 “001” 表示相同的数字 “1”。
```

示例 5:

```
输入: version1 = "1.0", version2 = "1.0.0"
输出: 0
解释: version1 没有第三级修订号，这意味着它的第三级修订号默认为 “0”。
```

提示:

1. 版本字符串由以点 （.） 分隔的数字字符串组成。这个数字字符串可能有前导零。
2. 版本字符串不以点开始或结束，并且其中不会有两个连续的点。

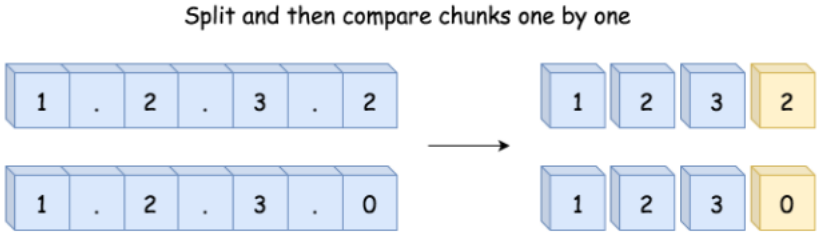
https://leetcode-cn.com/problems/compare-version-numbers/solution/bi-jiao-ban-ben-hao-by-l

更多硬核内容就在公众号「yes 的练级攻略」

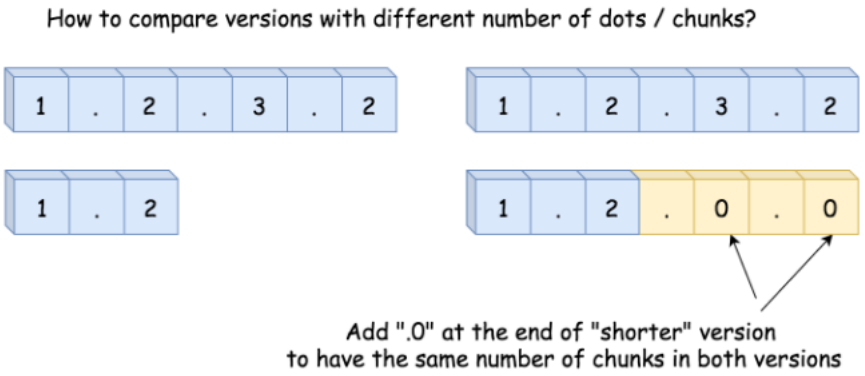
eetcode/

方法一：分割+解析，两次遍历，线性空间

第一个想法是将两个字符串按点字符分割成块，然后逐个比较这些块。



如果两个版本号的块数相同，则可以有效工作。如果不同，则需要在较短字符串末尾补充相应的 `.0` 块数使得块数相同。



算法：

- 根据点分割两个字符串将分割的结果存储到数组中。
- 遍历较长数组并逐个比较块。如果其中一个数组结束了，实际上可以根据需要添加尽可能多的零，以继续与较长的数组进行比较。
  - 如果两个版本号不同，则返回 1 或 -1。
- 版本号相同，返回 0。

复杂度分析

- 时间复杂度：$\mathcal{O}(N + M + \max(N, M))$。其中 N 和 M 指的是输入字符串的长度。
- 空间复杂度：$\mathcal{O}(N + M)$，使用了两个数组 `nums1` 和 `nums2` 存储两个字符串的块。

方法 1：分割+解析

```
class Solution {
    public int compareVersion(String version1, String version2) {
        String[] nums1 = version1.split("\\.");
        String[] nums2 = version2.split("\\.");
        int n1 = nums1.length, n2 = nums2.length;
        int i1, i2;
        for (int i = 0; i < Math.max(n1, n2); ++i) {
        i1 = i < n1 ? Integer.parseInt(nums1[i]) : 0;
        i2 = i < n2 ? Integer.parseInt(nums2[i]) : 0;
        if (i1 != i2) {
            return i1 > i2 ? 1 : -1;
```

```
        }
        }
        return 0;
    }
}
```

方法 2：双指针

方法二：双指针，一次遍历，常数空间

方法一有两个缺点：

- 是两次遍历的解决方法。
- 消耗线性空间。

我们能否实现一个只有一次遍历和消耗常数空间的解决方法呢？

其思想是在每个字符串上使用两个指针，跟踪每个数组的开始和结束。

这样，可以并行地沿着两个字符串移动，检索并比较相应的块。一旦两个字符串都被解析，比较也就完成了。

算法：

首先，我们定义了一个名为 `get_next_chunk(version, n, p)` 的函数，用于检索字符串中的下一个块。这个函数有三个参数：输入字符串 `version`，它的长度 `n`，以及指针 `p` 为要检索块的第一个字符。它在指针 `p` 和下一个点之间返回一个整数块。为了帮助迭代，返回的是下一个快的第一个字符的指针。下面是如何使用此函数解决问题的方法：

- 指针 `p1` 和 `p2` 分别指向 `version1` 和 `version2` 的起始位置：`p1=p2=0`。
- 并行遍历两个字符串。当 `p1 < n1 or p2 < n2`：
  - 使用 `get_next_chunk` 函数获取 `version1` 和 `version2` 的下一个块 `i1` 和 `i2`。
  - 比较 `i1` 和 `i2`。如果不相同，则返回 1 或 -1。
- 如果到了这里，说明版本号相同，则返回 0。

下面实现 `get_next_chunk(version, n, p)` 函数：

- 块的开头由指针 `p` 标记。如果 `p` 设置为字符串的结尾，则字符串解析完成。若要继续比较，则在添加 `.0` 返回。
- 如果 `p` 不在字符串的末尾，则沿字符串移动指针 `p_end` 以查找块的结尾。
- 返回块 `version.substring(p, p_end)`。

复杂度分析

- 时间复杂度：$\mathcal{O}(\max(N, M))$。其中 N 和 M 指的是输入字符串的长度。
- 空间复杂度：$\mathcal{O}(1)$，没有使用额外的数据结构。

```
class Solution {
    public Pair<Integer, Integer> getNextChunk(String version, int n, int p) {
        // if pointer is set to the end of string
        // return 0
        if (p > n - 1) {
        return new Pair(0, p);
        }
        // find the end of chunk
        int i, pEnd = p;
        while (pEnd < n && version.charAt(pEnd) != '.') {
        ++pEnd;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }
        // retrieve the chunk
        if (pEnd != n - 1) {
        i = Integer.parseInt(version.substring(p, pEnd));
        } else {
        i = Integer.parseInt(version.substring(p, n));
        }
        // find the beginning of next chunk
        p = pEnd + 1;

        return new Pair(i, p);
    }

    public int compareVersion(String version1, String version2) {
        int p1 = 0, p2 = 0;
        int n1 = version1.length(), n2 = version2.length();

        // compare versions
        int i1, i2;
        Pair<Integer, Integer> pair;
        while (p1 < n1 || p2 < n2) {
        pair = getNextChunk(version1, n1, p1);
        i1 = pair.getKey();
        p1 = pair.getValue();

        pair = getNextChunk(version2, n2, p2);
        i2 = pair.getKey();
        p2 = pair.getValue();
        if (i1 != i2) {
            return i1 > i2 ? 1 : -1;
        }
        }
        // the versions are equal
        return 0;
    }
}
```

66、把数字翻译成字符串（剑指 46）

剑指 Offer 46. 把数字翻译成字符串

难度 中等 116 收藏 分享 切换为英文 关注 反馈

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a”，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

示例 1:

```
输入：12258
输出：5
解释：12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"mzi"
```

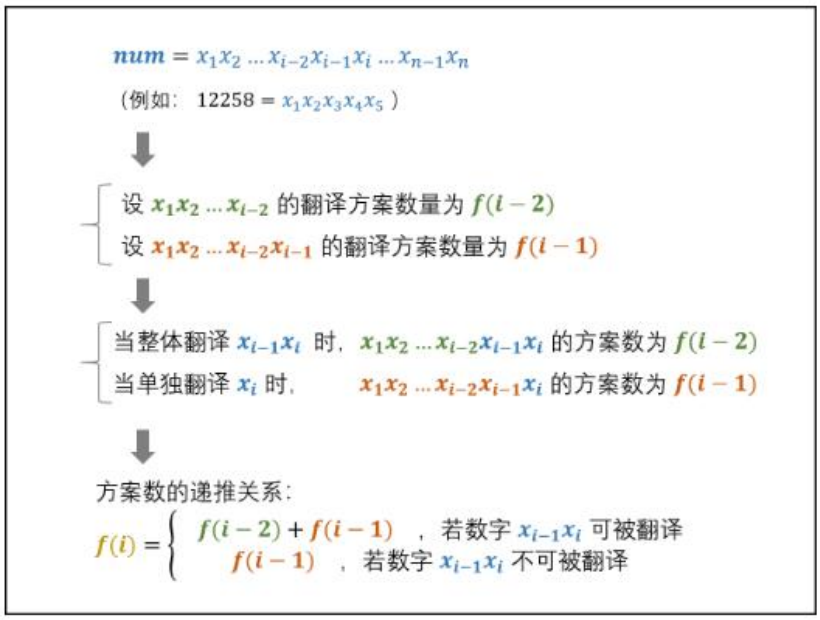
提示：

- $0 <= num < 2^{31}$

https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/solution/mian-shi-ti-46-ba-shu-zi-fan-yi-cheng-zi-fu-chua-6/

解题思路：

根据题意，可按照下图的思路，总结出 “递推公式” （即转移方程）。
因此，此题可用动态规划解决，以下按照流程解题。



更多硬核内容就在公众号「yes 的练级攻略」

动态规划解析：

> 记数字 num 第 i 位数字为 x_i，数字 num 的位数为 n；
> 例如：$num = 12258$ 的 $n = 5$，$x_1 = 1$。

- **状态定义：** 设动态规划列表 dp，$dp[i]$ 代表以 x_i 为结尾的数字的翻译方案数量。
- **转移方程：** 若 x_i 和 x_{i-1} 组成的两位数字可以被翻译，则 $dp[i] = dp[i-1] + dp[i-2]$；否则 $dp[i] = dp[i-1]$。
  - 可被翻译的两位数区间：当 $x_{i-1} = 0$ 时，组成的两位数是无法被翻译的（例如 $00, 01, 02, \cdots$），因此区间为 $[10, 25]$。

$$dp[i] = \begin{cases} dp[i-1] + dp[i-2] & ,10x_{i-1} + x_i \in [10, 25] \\ dp[i-1] & ,10x_{i-1} + x_i \in [0, 10) \cup (25, 99] \end{cases}$$

- **初始状态：** $dp[0] = dp[1] = 1$，即“无数字”和“第 1 位数字”的翻译方法数量均为 1；
- **返回值：** $dp[n]$，即此数字的翻译方案数量。

> Q：无数字情况 $dp[0] = 1$ 从何而来？
> A：当 num 第 $1, 2$ 位的组成的数字 $\in [10, 25]$ 时，显然应有 2 种翻译方法，即 $dp[2] = dp[1] + dp[0] = 2$，而显然 $dp[1] = 1$，因此推出 $dp[0] = 1$。

方法一：字符串遍历

- 为方便获取数字的各位 x_i，考虑先将数字 num 转化为字符串 s，通过遍历 s 实现动态规划。
- 通过字符串切片 $s[i-2:i]$ 获取数字组合 $10x_{i-1} + x_i$，通过对比字符串 ASCII 码判断字符串对应的数字区间。
- **空间使用优化：** 由于 $dp[i]$ 只与 $dp[i-1]$ 有关，因此可使用两个变量 a, b 分别记录 $dp[i], dp[i-1]$，两变量交替前进即可。此方法可省去 dp 列表使用的 $O(N)$ 的额外空间。

复杂度分析：

- **时间复杂度 $O(N)$：** N 为字符串 s 的长度（即数字 num 的位数 $\log(num)$），其决定了循环次数。
- **空间复杂度 $O(N)$：** 字符串 s 使用 $O(N)$ 大小的额外空间。

方法 1：动规

```
class Solution {
    public int translateNum(int num) {
        String s = String.valueOf(num);
        int a = 1, b = 1;
        for(int i = 2; i <= s.length(); i++) {
            String tmp = s.substring(i - 2, i);
            int c = tmp.compareTo("10") >= 0 && tmp.compareTo("25") <= 0 ? a +
b : a;
            b = a;
            a = c;
        }
        return a;
    }
}
```

方法二：数字求余

- 上述方法虽然已经节省了 dp 列表的空间占用，但字符串 s 仍使用了 $O(N)$ 大小的额外空间。

空间复杂度优化：

- 利用求余运算 $num\%10$ 和求整运算 $num//10$ ，可获取数字 num 的各位数字（获取顺序为个位、十位、百位...）。
- 因此，可通过 **求余** 和 **求整** 运算实现 **从右向左** 的遍历计算。而根据上述动态规划“对称性”，可知从右向左的计算是正确的。
- 自此，字符串 s 的空间占用也被省去，空间复杂度从 $O(N)$ 降至 $O(1)$ 。

复杂度分析：

- **时间复杂度** $O(N)$ ： N 为字符串 s 的长度（即数字 num 的位数 $\log(num)$ ），其决定了循环次数。
- **空间复杂度** $O(1)$ ： 几个变量使用常数大小的额外空间。

方法 2：数字求余

```
class Solution {
    public int translateNum(int num) {
        int a = 1, b = 1, x = 0, y = num % 10;
        while (num != 0){
            num /= 10;
            x = num % 10;
            int tmp = 10 * x + y;
            int c = (tmp >= 10 && tmp <= 25) ? a + b : a;
            b = a;
            a = c;
            y = x;
        }
        return a;
    }
}
```

67、第一个只出现一次的字符（剑指 50）

剑指 Offer 50. 第一个只出现一次的字符

难度 简单 39 收藏 分享 切换为英文 关注 反馈

在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

示例:

```
s = "abaccdeff"
返回 "b"

s = ""
返回 " "
```

限制:

0 <= s 的长度 <= 50000

https://leetcode-cn.com/problems/di-yi-ge-zhi-chu-xian-yi-ci-de-zi-fu-lcof/solution/mian-shi-ti-50-java-shi-yong-shu-zu-dai-ti-ha-xi-bi/

思路分析：

- 要求只出现一次的第一个字符，就需要去计数每一个字符出现了几次。一般都是用哈希表，但是字符往多了数按照扩展的ASCII码也就256个，就可以使用 `int[] count` 来代替哈希表：键就是字符对应的数，值就是该字符出现的次数。
- 第一次遍历计数每个字符出现的次数 `for(char c : chars) count[c]++;` 。第二次遍历按顺序去查看该字符出现了几次，如果该字符出现了1次，它就是第一个仅出现一次的字符，直接返回。
- 如果第二次遍历没有返回，就说明没有仅出现一次的字符，返回 `' '` 。
- 时间复杂度为$O(n)$，空间复杂度为$O(n)$（因为使用了辅助的字符数组）。

```
class Solution {
    public char firstUniqChar(String s) {
        int[] count = new int[256];
        char[] chars = s.toCharArray();
        for(char c : chars)
            count[c]++;
        for(char c : chars){
            if(count[c] == 1)
                return c;
        }
        return ' ';
    }
}
```

在牛客网剑指的版本

题目：在一个字符串(0<=字符串长度<=10000，全部由字母组成)中找到第一个只出现一次的字符,并返回它的位置， 如果没有则返回 -1（需要区分大小写）.（从 0 开始计数）

```
import java.util.*;
public class Solution {
    public int FirstNotRepeatingChar(String str) {
        if (str == null) {
            return -1;
        }
        char[] c = str.toCharArray();
        LinkedHashMap<Character, Integer> hash = new LinkedHashMap<Character, Integer>();
        // 让 item 遍历数组 c 中的每个值
        for (char item : c) {
            if (hash.containsKey(item)){
                hash.put(item, hash.get(item)+1);
            } else {
                hash.put(item,1);
            }
        }
        for (int i = 0; i<str.length(); i++){
            if (hash.get(str.charAt(i))==1){
                return i;
            }
```

公众号：yes的练级攻略

```
            }
            return -1;
        }
    }
```

68、统计重复个数（466）

466. 统计重复个数

难度 困难 112 收藏 分享 切换为英文 关注 反馈

由 n 个连接的字符串 s 组成字符串 S，记作 `S = [s,n]`。例如，`["abc",3]`="abcabcabc"。

如果我们可以从 s2 中删除某些字符使其变为 s1，则称字符串 s1 可以从字符串 s2 获得。例如，根据定义，"abc" 可以从 "abdbec" 获得，但不能从 "acbbe" 获得。

现在给你两个非空字符串 s1 和 s2（每个最多 100 个字符长）和两个整数 $0 \le n1 \le 10^6$ 和 $1 \le n2 \le 10^6$。现在考虑字符串 S1 和 S2，其中 `S1=[s1,n1]` 、 `S2=[s2,n2]` 。

请你找出一个可以满足使 `[S2,M]` 从 `S1` 获得的最大整数 M 。

示例：

```
输入：
s1 ="acb",n1 = 4
s2 ="ab",n2 = 2

返回：
2
```

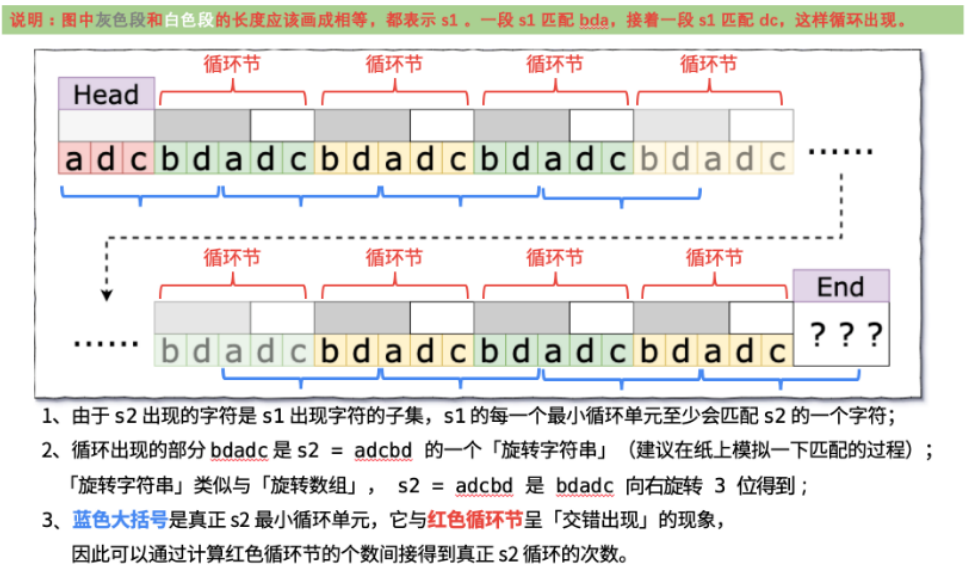
https://leetcode-cn.com/problems/count-the-repetitions/solution/bao-li-jie-fa-you-hua-jie-fa-java-dai-ma-by-liweiw/

请读者根据上面两篇题解的示意图理解以下事实：

- 由于 `s2` 的字母出现的顺序在 `s1` 中是乱序的，因此在循环匹配的时候，会出现「交错匹配」的现象：

1. 本身 `s1` 和 `s2` 就是循环多次出现的；
2. 在上一轮 `s2` 还没匹配完，`s1` 又开始下一轮匹配的时候，只要 `s1` 和 `s2` 循环次数够多，一定会出现一种特殊的「循环现象」，官方题解解释为「抽屉原理」。我个人不是很明白，我是通过具体例子（还有上面两篇题解的图去）理解的。

更多硬核内容就在公众号「yes 的练级攻略」



例如下面的例子：`s2 = "bea"`，但是产生「循环现象」的字符串 `eab`（观察下面代码的注释），注意：**它是 `s2` 的一个「旋转字符数组」。**

- 循环到 `aceb` 的时候，先找到 `e`，接着再下一轮找 `ab`，这样的过程是反复进行的。

- 循环到 `aceb` 的时候，先找到 `e`，接着再下一轮找 `ab`，这样的过程是反复进行的。

```
public static void main(String[] args) {
    String s1 = "aceb";
    int n1 = 7;

    // aceb aceb aceb aceb aceb aceb aceb
    //    b   e a b   e a b   e a b ea

    // 因为顺序不同，会出现「交叉循环」的现象，只要 s1 足够长，一定可以把 s2 覆盖
    String s2 = "bea"; // 相当于给 s2 调整了一个顺序

    // 每一次一定可以找到一个

    int n2 = 1;

    Solution solution = new Solution();
    int res = solution.getMaxRepetitions(s1, n1, s2, n2);
    System.out.println(res);
}
```

因此，在遍历的时候，我们就需要想办法观测出这种「循环现象」，这里需要引入两个变量，可以理解为「空间换时间」。

- `next[i]`：表示进行 `s1` 循环了 `i` 次以后（刚刚好在循环以后的那个时间点，`i` 从 `0` 开始），在 `s2` 上遍历到的下标（这里是上一次匹配完的下一个下标，注意结合代码理解）；
- `times[i]`：表示进行 `s1` 循环了 `i` 次以后（刚刚好在循环以后的那个时间点，`i` 从 `0` 开始），在 `s2` 上已经循环了几次（依然是结合代码理解）；

什么时候观测到循环节呢？

- 一定是在 `s1` 至少已经遍历完一次以后，因此 `i > 0`；
- 当前 `index2` 指向的下标恰好等于， `s1` 第 1 次循环完成以后的下标，即 `next[0]`。

这个时候，发现了循环节，根据「循环交错」现象（注意看上面两篇题解的图），就可以直接计算整体的 `loopTimesOnS2`，来自三个部分：

- 交错前面的部分；
- 中间「循环节」出现的部分；
- 交错后面的部分；

比较容易理解的代码是题解 找循环做优化，0ms 2MB 跑双百 给出的代码，建议多看一看，很好理解。

在这一版代码的基础上，我结合了题解 一个注释工 做了一些修改。

```
class Solution {
    public int getMaxRepetitions(String s1, int n1, String s2, int n2)
{
        int len1 = s1.length();
        int len2 = s2.length();
        // 特判
        if (len1 == 0 || len2 == 0 || n1 == 0 || n2 == 0) {
            return 0;
        }
        char[] chars1 = s1.toCharArray();
        char[] chars2 = s2.toCharArray();
        // 记录下一个要匹配的 s2 中字符的下标
        int index2 = 0;
        // 记录 s2 循环的次数，loopTimesOnS2 是 index2 的函数
        int loopTimesOnS2 = 0;
        // 当 s1 循环了 i 次以后，s2 循环了几次
        int[] times = new int[n1];
        // 当 s1 循环了 i 次以后，s2 下一个字符匹配的下标（上一个字符匹配完以
后 index2++，所以这里是下一个）
        int[] next = new int[n1];
        // 暴力解法的优化，一边遍历，一边找循环节，找到就直接计算出结果
        for (int i = 0; i < n1; i++) {
            // 指针 index1 在 s1 上循环游走
            for (int index1 = 0; index1 < len1; index1++) {
                // 如果匹配到 s2 中字符的字符
                if (chars1[index1] == chars2[index2]) {
                    index2++;
                }
                // 匹配完一个 s2，计数器 + 1，重置 s2 下标
                if (index2 == len2) {
                    // 循环扫描，因此 index2 重置为 0
                    index2 = 0;
                    loopTimesOnS2++;
```

```
                }
            }
            // 记录了 s1 遍历了 i 次（从 0 开始）以后，s2 已经完整地遍历了多少次
            times[i] = loopTimesOnS2;
            // 记录了 s1 遍历了 i 次（从 0 开始）以后，s2 上 index2 的位置（由于匹配完成以后执行了 index2++ ，此时 index2 指向下一个要匹配的字符）
            next[i] = index2;
            // 循环节一定是在：s1 至少循环了一次以后
            // s2 即将匹配的字符的下标恰好等于 s1 刚刚循环完一次以后 s2 即将要匹配的下标，就是 index2 == next[0] 这行代码的含义
          // 下面的代码如果不好理解的话，一定要结合 next、times、i、n1 的定义
            if (i > 0 && index2 == next[0]) {
                // 说明出现了循环节，直接计算结果，来自 3 个部分
                // 第 1 部分：开头
                int headCount = times[0];
                // 第 2 部分：中间循环部分
                // 这里难理解的话，一定要结合定义，看图理解，
                // (n1 - 1)，要减去 1 个，因为从 s1 已经循环了一次开始，才出现的循环节
                // (n1 - 1) / i 表示剩下的部分有多少个红色大括号段（见图）
                // times[i] - times[0] 表示：每个循环节里出现了几个 s2
                // ((n1 - 1) / i) * (times[i] - times[0]) 就表示中间那部分里面 s2 出现的次数
                int circulateCount = ((n1 - 1) / i) * (times[i] - times[0]);

                // 第 3 部分：结尾部分
                // (n1 - 1) % i 相对于 (n1 - 1) / i 而言，就是不能整除的部分
                // 减去 times[0] 是因为计算 times[i] 的时候计算的是前缀和，headCount 这部分已经计算过了，要把它删掉（写成两部分的和也可以）
                int endCount = times[(n1 - 1) % i] - times[0];

                // 总结：一下子计算出 loopTimesOnS2，是这个解法优化的地方
                return (headCount + circulateCount + endCount) / n2;
            }
        }
        // 走到这里，没有发现循环节，依据定义计算，n1 - 1 是最后一次遍历 s1 的下标
        return times[n1 - 1] / n2;
    }
    public static void main(String[] args) {
        String s1 = "abcde";
```

```
        int n1 = 5;
        // abcde abcde abcde abcde abcde
        //   c   a   e   c   a   e
        String s2 = "cae";
        int n2 = 1;
        Solution solution = new Solution();
        int res = solution.getMaxRepetitions(s1, n1, s2, n2);
        System.out.println(res);
    }
}
```

69、括号生成（22）【括号（面试题 08.09）】

22. 括号生成

难度 中等 1106 收藏 分享 切换为英文 关注 反馈

数字 *n* 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

示例：

```
输入：n = 3
输出：[
       "((()))",
       "(()())",
       "(())()",
       "()(())",
       "()()()"
     ]
```

通过次数 137,863 | 提交次数 182,417

https://leetcode-cn.com/problems/generate-parentheses/solution/jian-dan-dfsmiao-dong-by-sweetiee/

用 DFS 加剪枝

原理 1：左括号永远是第一个；

原理 2：右括号插入的时候剩余数量永远应该比左括号剩余数量多；

原理 3：左右括号总数最终等于有效括号总数

```
class Solution {
    List<String> res = new ArrayList<>();
    public List<String> generateParenthesis(int n) {
        dfs(n,n,"");
        return res;
    }
    public void dfs(int left, int right, String curStr) {
        // 左右括号都不剩余了，递归终止
        if (left == 0 && right == 0) {
            res.add(curStr);
            return;
        }
```

yes的练级攻略

```
            // 如果左括号还剩余的话，可以拼接左括号
            if (left > 0) {
                dfs (left - 1, right, curStr + "(");
            }
            // 如果右括号剩余多于左括号剩余的话，可以拼接右括号
            if (right > left) {
                dfs(left, right-1,curStr+")");
            }
        }
    }
```

70、有效的括号（20）

20. 有效的括号

难度 简单 1627 收藏 分享 切换为英文 关注 反馈

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入: "{[]}"
输出: true
```

解题思路：

- **算法原理**
  - 栈先入后出特点恰好与本题括号排序特点一致，即若遇到左括号入栈，遇到右括号时将对应栈顶左括号出栈，则遍历完所有括号后 `stack` 仍然为空；
  - 建立哈希表 `dic` 构建左右括号对应关系：key 左括号，$value$ 右括号；这样查询 2 个括号是否对应只需 $O(1)$ 时间复杂度；建立栈 `stack` ，遍历字符串 `s` 并按照算法流程一一判断。
- **算法流程**
  1. 如果 `c` 是左括号，则入栈 $push$；
  2. 否则通过哈希表判断括号对应关系，若 `stack` 栈顶出栈括号 `stack.pop()` 与当前遍历括号 `c` 不对应，则提前返回 $false$。
- **提前返回 $false$**
  - **提前返回优点：** 在迭代过程中，提前发现不符合的括号并且返回，提升算法效率。
  - **解决边界问题：**
    - **栈 `stack` 为空：** 此时 `stack.pop()` 操作会报错；因此，我们采用一个取巧方法，给 `stack` 赋初值 `?` ，并在哈希表 `dic` 中建立 key :'?'，$value$:'?' 的对应关系予以配合。此时当 `stack` 为空且 `c` 为右括号时，可以正常提前返回 $false$；
    - **字符串 `s` 以左括号结尾：** 此情况下可以正常遍历完整个 `s` ，但 `stack` 中遗留未出栈的左括号；因此，最后需返回 `len(stack) == 1` ，以判断是否是有效的括号组合。
- **复杂度分析**
  - 时间复杂度 $O(N)$：正确的括号组合需要遍历 1 遍 `s` ；
  - 空间复杂度 $O(N)$：哈希表和栈使用线性的空间大小。

```
class Solution {
    private static final Map<Character,Character> map = new HashMap<Character,C
haracter>(){{
        put('{','}'); put('[',']'); put('(',')'); put('?','?');
    }};
    public boolean isValid(String s) {
        if(s.length() > 0 && !map.containsKey(s.charAt(0))) return false;
        LinkedList<Character> stack = new LinkedList<Character>() {{ add('?');
}};
        for(Character c : s.toCharArray()){
            if(map.containsKey(c)) stack.addLast(c);
            else if(map.get(stack.removeLast()) != c) return false;
        }
        return stack.size() == 1;
    }
}
```

71、有效的括号字符串（678）

678. 有效的括号字符串

难度 中等 146 收藏 分享 切换为英文 关注 反馈

给定一个只包含三种字符的字符串：`(` ，`)` 和 `*`，写一个函数来检验这个字符串是否为有效字符串。有效字符串具有如下规则：

1. 任何左括号 `(` 必须有相应的右括号 `)`。
2. 任何右括号 `)` 必须有相应的左括号 `(` 。
3. 左括号 `(` 必须在对应的右括号之前 `)`。
4. `*` 可以被视为单个右括号 `)` ，或单个左括号 `(` ，或一个空字符串。
5. 一个空字符串也被视为有效字符串。

示例 1:

```
输入："()"
输出：True
```

示例 2:

```
输入："(*)"
输出：True
```

示例 3:

```
输入："(*))"
输出：True
```

注意:

1. 字符串大小将在 [1，100] 范围内。

https://leetcode-cn.com/problems/valid-parenthesis-string/solution/fen-zhi-tan-xin-shuang-xiang-bian-li-shuang-zhan-j/

方法二：贪心

思路

- 与方法一神似
- 无需每种情况都细分考虑，只需关注 **左括号至少几个、至多几个**
- 遇左则至多/至少都增加
- 遇右则至多/至少都减少
  - 若至多 `max` 都不够抵右括号，则 `return false`
- 遇 `'*'` 则可能减少/不变/增多，即 `min--; max++;`
  - 若至少 `min` 小于 `0` ，说明此分支走不通，需剪枝
    - 类似方法一中的递归终止条件 `if (count < 0) return false;`
- 最终要求左括号必须无剩余 `min == 0;`
  - `min < 0` 的情况在上一条思路中剪枝了

方法三：贪心算法

- 在检查字符串是否有效时，我们只关心是否 “balance” ：当我们分析字符串时计算未配对的左括号的数量。例如，在检查 “(()())” 是否有效时，我们在分析字符串时得到了 1, 2, 1, 2, 1, 0 的平衡过程： “(” 有1个左括号， “(” 有2个左括号， “(()” 剩下一个左括号，依此类推。
- 例如，如果我们有字符串 (***) ，那么在分析每个符号时， balance 的可能值是 ’(’ 对应 [1] ，’(*’ 对应 [0, 1, 2] ，’(**’ 对应 [0, 1, 2, 3] ，’(***’ 对应 [0, 1, 2, 3, 4] ，’(***)’ 对应 [0, 1, 2, 3] 。(指的是可能的未匹配的左括号数量)
- 此外，我们可以证明这些状态总是形成一个连续的区间。因此，我们只需要知道这个区间的左右边界。也就是说，我们将把上面描述的中间状态保持为 [lo, hi] = [1, 1], [0, 2], [0, 3], [0, 4], [0, 3]

算法：

- 在处理字符串中的当前字符时，让 lo、hi 分别为可能的最小和最大左括号数。
- 如果我们遇到左括号（ c == ’(’ ），那么 lo++ ，若遇到右括号，则 lo-- 。如果我们遇到 * 号且可以转换成左括号（ c != ’)’ ），则 hi++ ，否则我们转换成右括号，所以 hi-- 。如果 hi<0 ，那么无论我们选择什么，当前情况都无法生成有效的括号。而且，左括号不能小于0。最后，我们应该检查一下是否可以有 0 个左括号。

复杂度分析

- 时间复杂度：$O(N)$。其中 N 是字符串的长度,我们对字符串迭代一次。
- 空间复杂度：$O(1)$，我们使用 lo 和 hi 指针占用常数空间。但是，创建一个新的字符数组需要 $O(N)$ 空间。

```
class Solution {
    public boolean checkValidString(String s) {
        // possible range
        int min = 0, max = 0; // 维护当前左括号的数量范围：[min, max]
        for (char c : s.toCharArray()) {
            if (c == '(') {
                ++min;
                ++max;
            } else if (c == ')') {
                if (min > 0) min--;
                if (max-- == 0) return false;// 左括号不够
            } else {
                if (min > 0) min--; // 可作为右括号，抵消
                ++max; // 可作为左括号
            }
        }
        return min == 0;
    }
}
```

72、最长有效括号（32）

更多硬核内容就在公众号「yes 的练级攻略」

32. 最长有效括号

难度 困难 694 收藏 分享 切换为英文 关注 反馈

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

示例 1:

```
输入："(()"
输出：2
解释：最长有效括号子串为 "()"
```

示例 2:

```
输入：")()())"
输出：4
解释：最长有效括号子串为 "()()"
```

https://leetcode-cn.com/problems/longest-valid-parentheses/solution/zui-chang-you-xiao-gua-hao-by-leetcode-solution/

方法一：动态规划

思路和算法

我们定义 $dp[i]$ 表示以下标 i 字符结尾的最长有效括号的长度。我们将 dp 数组全部初始化为 0 。显然有效的子串一定以 ‘)’ 结尾，因此我们可以知道以 ‘(’ 结尾的子串对应的 dp 值必定为 0 ，我们只需要求解 ‘)’ 在 dp 数组中对应位置的值。

我们从前往后遍历字符串求解 dp 值，我们每两个字符检查一次：

1. $s[i] = $ ‘)’ 且 $s[i-1] = $ ‘(’，也就是字符串形如 “……()”，我们可以推出：

$$dp[i] = dp[i-2] + 2$$

我们可以进行这样的转移，是因为结束部分的 "()" 是一个有效子字符串，并且将之前有效子字符串的长度增加了 2 。

2. $s[i] = $ ‘)’ 且 $s[i-1] = $ ‘)’，也就是字符串形如 “……))”，我们可以推出：
如果 $s[i - dp[i-1] - 1] = $ ‘(’，那么

$$dp[i] = dp[i-1] + dp[i - dp[i-1] - 2] + 2$$

我们考虑如果倒数第二个 ‘)’ 是一个有效子字符串的一部分（记作 sub_s），对于最后一个 ‘)’，如果它是一个更长子字符串的一部分，那么它一定有一个对应的 ‘(’，且它的位置在倒数第二个 ‘)’ 所在的有效子字符串的前面（也就是 sub_s 的前面）。因此，如果子字符串 sub_s 的前面恰好是 ‘(’，那么我们就用 2 加上 sub_s 的长度（$dp[i-1]$）去更新 $dp[i]$。同时，我们也会把有效子串 “(sub_s)” 之前的有效子串的长度也加上，也就是再加上 $dp[i - dp[i-1] - 2]$。

最后的答案即为 dp 数组中的最大值。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为字符串的长度。我们只需遍历整个字符串一次，即可将 dp 数组求出来。
- 空间复杂度：$O(n)$。我们需要一个大小为 n 的 dp 数组。

```
public class Solution {
    public int longestValidParentheses(String s) {
        int maxans = 0;
        int dp[] = new int[s.length()];
        for (int i = 1; i < s.length(); i++) {
            if (s.charAt(i) == ')') {
                if (s.charAt(i - 1) == '(') {
```

```
                    dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;
                } else if (i - dp[i - 1] > 0 && s.charAt(i - dp[i - 1] - 1) ==
'(') {
                    dp[i] = dp[i - 1] + ((i - dp[i - 1]) >= 2 ? dp[i - dp[i - 1
] - 2] : 0) + 2;
                }
                maxans = Math.max(maxans, dp[i]);
            }
        }
        return maxans;
    }
}
```

73、有效括号的嵌套深度（1111）

1111. 有效括号的嵌套深度

难度 中等 120 收藏 分享 切换为英文 关注 反馈

有效括号字符串 定义：对于每个左括号，都能找到与之对应的右括号，反之亦然。详情参见题末「**有效括号字符串**」部分。

嵌套深度 `depth` 定义：即有效括号字符串嵌套的层数，`depth(A)` 表示有效括号字符串 `A` 的嵌套深度。详情参见题末「**嵌套深度**」部分。

有效括号字符串类型与对应的嵌套深度计算方法如下图所示：

1. 空字符串	2. AB 型	3. (A) 型
""	(()) ()	((()()))
depth = 0	depth = max(2, 1) = 2	depth = 1 + depth(A) = 3

给你一个「有效括号字符串」 `seq`，请你将其分成两个不相交的有效括号字符串，`A` 和 `B`，并使这两个字符串的深度最小。

- 不相交：每个 `seq[i]` 只能分给 `A` 和 `B` 二者中的一个，不能既属于 `A` 也属于 `B`。
- `A` 或 `B` 中的元素在原字符串中可以不连续。
- `A.length + B.length = seq.length`
- 深度最小：`max(depth(A), depth(B))` 的可能取值最小。

划分方案用一个长度为 `seq.length` 的答案数组 `answer` 表示，编码规则如下：

- `answer[i] = 0`，`seq[i]` 分给 `A`。
- `answer[i] = 1`，`seq[i]` 分给 `B`。

如果存在多个满足要求的答案，只需返回其中任意 **一个** 即可。

示例 1：

输入： `seq = "(()())"`
输出： `[0,1,1,1,1,0]`

示例 2：

```
输入：seq = "()(())()"
输出：[0,0,0,1,1,0,1,1]
解释：本示例答案不唯一。
按此输出 A = "()()", B = "()()", max(depth(A), depth(B)) = 1，它们的深度最小。
像 [1,1,1,0,0,1,1,1]，也是正确结果，其中 A = "(()())", B = "()", max(depth(A), depth(B)) = 1 。
```

提示：

- `1 <  seq.size <= 10000`

有效括号字符串：

```
仅由 "(" 和 ")" 构成的字符串，对于每个左括号，都能找到与之对应的右括号，反之亦然。
下述几种情况同样属于有效括号字符串：

  1. 空字符串
  2. 连接，可以记作 AB（A 与 B 连接），其中 A 和 B 都是有效括号字符串
  3. 嵌套，可以记作 (A)，其中 A 是有效括号字符串
```

嵌套深度：

```
类似地，我们可以定义任意有效括号字符串 s 的 嵌套深度 depth(S)：

  1. s 为空时，depth("") = 0
  2. s 为 A 与 B 连接时，depth(A + B) = max(depth(A), depth(B))，其中 A 和 B 都是有效括号字符串
  3. s 为嵌套情况，depth("(" + A + ")") = 1 + depth(A)，其中 A 是有效括号字符串

例如："", "()()"，和 "()(()())" 都是有效括号字符串，嵌套深度分别为 0，1，2，而 ")(" 和 "(()" 都不是有效括号字符串。
```

https://leetcode-cn.com/problems/maximum-nesting-depth-of-two-valid-parentheses-strings/solution/qian-tao-shen-du-wan-cheng-gua-hao-pi-pei-wen-ti-s/

分析题意

1、首先需要弄清楚「有效括号」的意思。

- 如果做过「力扣」第 20 题：有效的括号，就不难明白有效括号的含义：就像我们写代码一样，`{` 与 `}`（本题是 `(` 与 `)`）需要成对出现，才叫匹配；
- 由于匹配操作符合「后进先出」的特点：遇到进来的右括号 `)`，它匹配后进来的左括号 `(`，它相对于之前扫描到的左括号 `(` 先完成匹配，这种匹配工作需要使用的数据结构是「栈」；
- 在完成括号匹配的过程中，只需要把遍历到的左括号进栈，在遍历到右括号的时候，左括号出栈，这个思路是经典且重要的。

2、看题目：

> 连接，可以记作 `AB`（`A` 与 `B` 连接），其中 `A` 和 `B` 都是有效括号字符串。

说明连在一起的时候，以「有效括号字符串」为最小单位。例如：`A = ")("` 与 `B = "()"` 就不能连接在一起，因为 `A` 不是有效括号字符串。

3、看题目：

> 嵌套，可以记作 `(A)`，其中 A 是有效括号字符串；
> `s` 为 `A` 与 `B` 连接时，`depth(A + B) = max(depth(A), depth(B))`，其中 `A` 和 `B` 都是有效括号字符串；
> 例如：`""`，`"()()"`，和 `"()(()())"` 都是有效括号字符串，嵌套深度分别为 `0`，`1`，`2`，而 `")("` 和 `"(()"` 都不是有效括号字符串。

从这些信息中，可以知道：

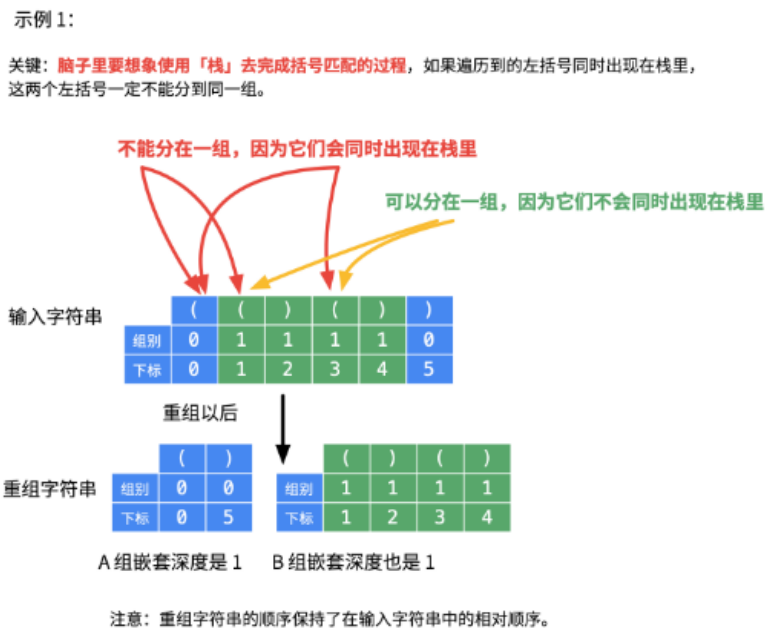
更多硬核内容就在公众号「yes 的练级攻略」

- **「嵌套」才会产生「深度」**，所以题目的定义是**嵌套深度**，极端例子 1：`()()()()`，这个字符串的「嵌套深度」为 1。极端例子 2：`((((()))))` 这个字符串的「嵌套深度」为 4。而这个深度恰好与在匹配过程中，栈的最大高度是一致的；
- 在「有效括号字符串」连接的时候，`depth(A + B) = max(depth(A), depth(B))` 的意思是大吃小，即在两个「有效括号字符串」连接的时候，整体深度取决于部分「有效括号字符串」的深度，这一点特别像**在二叉树中的计算二叉树的高度**；
- 其实仔细观察就会发现，这个「嵌套深度」就是输入字符串，**使用栈完成括号匹配，栈中最多连续出现的左括号 `(` 的个数。**
- 而题目要求我们把输入的整体有效字符串做一个重组，要求是只拆成两个部分 `A` 和 `B`，每个字符要么分到 `A` 要么分到 `B`，分到 `A` 标记为 `0`，分到 `B` 标记为 `1`。**而每个部分的字符又要保持在输入字符串中的顺序不变。**因此输出是一个与原始字符串等长的整数数组，这个整数数组里只有 `0` 和 `1`。

例如示例 1：

输入：`seq = "(()())"`，嵌套深度为 2，最多连续出现了 2 个左括号。现在要拆分以后再重组，其实思路就有了，把这两个连续出现的左括号分到不同组即可。

输出：`[0, 1, 1, 1, 1, 0]` 对应了输入字符串的每个字符对应的组号，`0` 分到 `A` 组，`1` 分到 `B` 组。



标记为 `0` 的组的「嵌套深度」是 1；标记为 1 的组的「嵌套深度」是 1，因此总的「嵌套深度」就是 1。

从这个示例中我们知道，重组以后的嵌套深度是原始嵌套深度的一半（不能整除的时候上取整）。

- 如果原始嵌套深度是偶数，例如 6，重组的嵌套深度 = 3；
- 如果原始嵌套深度是奇数，例如 7，重组的嵌套深度 = 4。

示例 2：

☆收藏

说明：**结果不唯一，但一定要保证连续的左括号 `((` 不被分在同一组**，具体来说，就是完成「括号匹配」问题使用的栈里，同时出现的左括号，不能在同一个分组里，这样就不会增加嵌套的深度。

思路分析

- 根据 `depth(A + B) = max(depth(A), depth(B))` 这个定义，整体的「嵌套深度」取决于子序列的「嵌套深度」的最大者；
- 要使得 `max(depth(A), depth(B))` 的可能取值最小，分析示例的时候提到，这很像一棵二叉树，要使得二叉树的深度最小，那么就需要该二叉树平衡，一个可行的做法是：把栈中连续出现的左括号 `(` 根据奇偶性分到不同的组，**右括号随与之匹配左括号的组号**；
- 如果出现 `()` 这种子序列，即左括号后面连着出现了右括号，其实分在那一组都是没有关系的，因为它的存在不会使得「嵌套深度」更深。

感觉核心思想就是：连续的 `(` 会造成嵌套深度的增加，因此对于这种要把他们分到不同的组中。而连续的标准不仅仅是原始字符串中的左右相邻，应该还包括中间成对的 `()` 消除后的连续。

```
class Solution {
    public int[] maxDepthAfterSplit(String seq) {
        int n = seq.length();
        int[] res = new int[n];
        int depth = 0;
        // 在 Java 里，seq.charAt(i) 函数会做下标越界检查，
        // 因此先转换成字符数组是常见的做法
        char[] ch = seq.toCharArray();
        for (int i = 0; i < n; i++) {
            // 遍历到左括号，连续括号个数加 1
            if (ch[i] == '(') {
                depth++;
                res[i] = depth % 2; // % 2 也可以写成 & 1，为了保证语义清楚，写 % 2
            } else {
                // 遍历到右括号，与当前栈顶左括号分在一组，因此先取模，再 --
```

```
                res[i] = depth % 2;
                depth--;
            }
        }
        return res;
    }
}
```

74、括号的分数（856）

856. 括号的分数

难度 中等 134 收藏 分享 切换为英文 关注

给定一个平衡括号字符串 `S`，按下述规则计算该字符串的分数：

- `()` 得 1 分。
- `AB` 得 `A + B` 分，其中 A 和 B 是平衡括号字符串。
- `(A)` 得 `2 * A` 分，其中 A 是平衡括号字符串。

示例 1：

```
输入： "()"
输出： 1
```

示例 2：

```
输入： "(())"
输出： 2
```

示例 3：

```
输入： "()()"
输出： 2
```

示例 4：

```
输入： "((()(())))"
输出： 6
```

提示：

1. `S` 是平衡括号字符串，且只含有 `(` 和 `)`。
2. `2 <= S.length <= 50`

https://leetcode-cn.com/problems/score-of-parentheses/solution/gua-hao-de-fen-shu-by-leetcode/

方法三：统计核心的数目

事实上，我们可以发现，只有 `()` 会对字符串 `S` 贡献实质的分数，其它的括号只会将分数乘二或者将分数累加。因此，我们可以找到每一个 `()` 对应的深度 `x`，那么答案就是 `2^x` 的累加和。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是字符串 `S` 的长度。
- 空间复杂度：$O(1)$。

方法 1：数学

```
class Solution {
    public int scoreOfParentheses(String S) {
        int ans = 0, bal = 0;
        for (int i = 0; i < S.length(); ++i) {
            if (S.charAt(i) == '(') {
                bal++;
            } else {
                bal--;
                if (S.charAt(i-1) == '(')
                    ans += 1 << bal;
            }
        }
        return ans;
    }
}
```

方法二：栈

字符串 `S` 中的每一个位置都有一个“深度”，即该位置外侧嵌套的括号数目。例如，字符串 `(()(.))` 中的 `.` 的深度为 `2`，因为它外侧嵌套了 `2` 层括号：`(__(.__))`。

我们用一个栈来维护当前所在的深度，以及每一层深度的得分。当我们遇到一个左括号 `(` 时，我们将深度加一，并且新的深度的得分置为 `0`。当我们遇到一个右括号 `)` 时，我们将当前深度的得分乘二并加到上一层的深度。这里有一种例外情况，如果遇到的是 `()`，那么只将得分加一。

下面给出了字符串 `(()(()))` 每次对应的栈的情况：

- `[0, 0]` `(`
- `[0, 0, 0]` `((`
- `[0, 1]` `(()`
- `[0, 1, 0]` `(()(`
- `[0, 1, 0, 0]` `(()((`
- `[0, 1, 1]` `(()(()`
- `[0, 3]` `(()(())`
- `[6]` `(()(()))`

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是字符串 `S` 的长度。
- 空间复杂度：$O(N)$，为栈的大小。

方法 2：栈

```
class Solution {
```

```
    public int scoreOfParentheses(String S) {
        int ans = 0, bal = 0;
        for (int i = 0; i < S.length(); ++i) {
            if (S.charAt(i) == '(') {
                bal++;
            } else {
                bal--;
                if (S.charAt(i-1) == '(')
                    ans += 1 << bal;
            }
        }
        return ans;
    }
}
```

75、删除无效的括号（301）

301. 删除无效的括号

难度 困难 214 收藏 分享 切换为英文 关注 反馈

删除最小数量的无效括号，使得输入的字符串有效，返回所有可能的结果。

说明: 输入可能包含了除 (和) 以外的字符。

示例 1:

输入: "()())()"
输出: ["()()()", "(())()"]

示例 2:

输入: "(a)())()"
输出: ["(a)()()", "(a())()"]

示例 3:

输入: ")("
输出: [""]

方法 1：DFS（快，讲解在后面）

```
class Solution {
    // 用集合存储所有正确的字符串，可避免重复
    private Set<String> set = new HashSet<>();
    public List<String> removeInvalidParentheses(String s) {
        char[] ss = s.toCharArray();
        int open = 0, close = 0;
        // 获取应该去除的左右括号数
        for (char c : ss) {
            if (c == '(') open ++;
            else if (c == ')')
                if (open > 0) open --;
                else close ++;
        }
```

```
        // 回溯
        backTracking(ss, new StringBuilder(), 0, 0, 0, open, close);
        return new ArrayList(set);
    }

    public void backTracking(char[] ss, StringBuilder sb, int index, in
t open, int close, int openRem, int closeRem) {
        /**
         * 回溯函数
         * 分别对字符串中的每一位置的字符进行处理,最终获得符合要求的字符串加入
集合中
         * @param ss 字符串对应的字符数组
         * @param sb 储存当前处理过且未去除字符的字符串
         * @param index 当前处理的字符位置
         * @param open 当前 sb 中储存的左括号数
         * @param close 当前 sb 中储存的右括号数
         * @param openRem 当前需要去除的左括号数
         * @param closeRem 当前需要去除的右括号数
         */
        // 所有字符都处理完毕
        if (index == ss.length) {
            // 如果应去除的左右括号数都变为0，则将 sb 插入 set
            if (openRem == 0 && closeRem == 0)
                set.add(sb.toString());
            return;
        }
        // 去掉当前位置的字符（括号），并处理下一个字符
        if (ss[index] == '(' && openRem > 0 || ss[index] == ')' && clos
eRem > 0)
            backTracking(ss, sb, index + 1, open, close, openRem - (ss[
index] == '(' ? 1 : 0), closeRem - (ss[index] == ')' ? 1 : 0));
        // 不去掉当前位置字符
        // 将当前位置字符插入 sb
        sb.append(ss[index]);
        // 当前位置不为括号，则直接处理下一个字符
        if (ss[index] != '(' && ss[index] != ')')
            backTracking(ss, sb, index + 1, open, close, openRem, close
Rem);
        // 当前位置为左括号，增加左括号计数，处理下一个字符
        else if (ss[index] == '(')
          backTracking(ss, sb, index + 1, open + 1, close, openRem, clo
seRem);
        // 当前位置为右括号，且当前左括号计数大于右括号计数，则增加右括号计数，
处理下一个字符
```

```
            else if (open > close)
                backTracking(ss, sb, index + 1, open, close + 1, openRem, clo
seRem);
            // 去除当前加入 sb 的字符
            sb.deleteCharAt(sb.length() - 1);
        }
}
```

https://leetcode-cn.com/problems/remove-invalid-parentheses/solution/shan-chu-wu-xiao-de-gua-hao-by-leetcode/

https://leetcode-cn.com/problems/remove-invalid-parentheses/solution/301-shan-chu-wu-xiao-de-gua-hao-java-hui-su-xiang-/

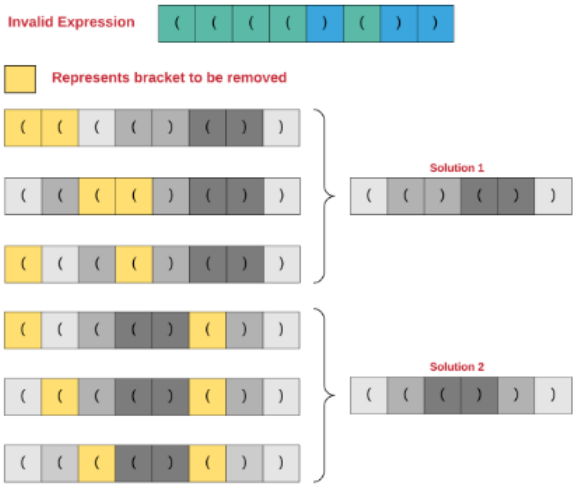
方法一：回溯

对于这个问题，我们得到了一个由括号组成的表达式，并且表达式中可能有一些错误的括号或额外的括号，导致它无效。只有当每个右括号都有对应的左括号时，由括号组成的表达式才被视为有效，反之亦然。

这意味着，如果我们从左到右查看每个括号，一旦遇到右括号，就应该有一个左括号来匹配它。否则表达式将变为无效。如果左括号的数目大于右括号的数目则表达式也无效。

让我们看看一个无效的表达式和所有可能的有效表达式，这些表达式可以通过删除一些方括号从中形成。我们可以删除哪些括号没有限制。我们只需要使表达式有效。

> 唯一的条件是，我们应该删除最小的括号数，使一个无效的表达式有效。如果不存在这个条件，我们可以潜在地删除大部分括号。



在上图中需要注意的一件重要的事情是，有多种方法可以达到相同的解，也就是说，为了使原始表达式有效，需要删除的括号的最佳数目是 k。我们可以删除多组不同的 k 括号，这些括号最终将给出相同的最终表达式。但是，每个有效表达式只应记录一次。我们必须在解决方案中解决这个问题。请注意，还有其他可能的方法可以达到上面所示的两个有效表达式之一。对于这两个有效表达式，我们只演示了三种方法。

回到我们的问题上来，现在出现的问题是，如何决定要删除哪些括号？

> 因为我们不知道哪一个括号可能被删除，所以我们尝试了所有的选项！

对于每个括号，我们有两个选择：

- 它可以被视为最终表达式的一部分
- 它可以被忽略，也就是说，我们可以从最终表达式中删除它。

> 这样的问题，我们有多个选择，我们没有战略或指标来贪婪地决定选择哪一个选择，我们尝试了所有的选择，看看哪一个导致了答案。

算法：

1. 初始化最终将存储所有有效表达式的数组。
2. 从给定序列中最左边的括号开始，然后向右递归
3. 递归状态由我们当前在原始表达式中处理的索引定义。让这个索引用字符 `i` 来表示。另外，我们有两个不同的变量 `left_count` 和 `right_count` ，它们表示我们到目前为止添加到表达式中的左括号和右括号的数目。这些是被考虑的括号。
4. 如果当前字符，即 `s[i]` （考虑 s 是表达式字符串）既不是右括号也不是左括号，那么我们只需将此字符添加到当前递归的最终解决方案字符串中。
5. 但是，如果当前字符是两个方括号中的一个，即 `S[i] == '(' or S[i] == ')'` ，则我们有两个选项。我们可以通过将此字符标记为无效字符来丢弃它，也可以将此括号视为最终表达式的一部分。
6. 当原始表达式中的所有括号都被处理后，我们只需检查 `expr` 表示的表达式是否有效，即到目前为止形成的表达式是否有效。我们检查最后一个表达式是否有效的方法是通过查看 `left_count` 和 `right_count` 的值。表达式必须是有效的 `left_count == right_count` 。如果它确实有效，那么它可能是我们可能的解决方案之一。
   - 即使我们有一个有效的表达式，我们也需要跟踪我们为获得这个表达式所做的删除操作的数量。这是由另一个名为 `rem_count` 的递归中传递的变量完成的。
   - 一旦递归完成，我们将检查 `rem_count` 的当前值是否小于我们迄今为止为形成有效表达式所采取的最少步骤数，即全局最小值。如果不是这样的话，我们不会记录新的表达式，否则我们会记录它。

从实现的角度来看，我们可以做的一个小的优化就是在我们的算法中引入某种修剪。现在，我们只需到最后一步，即处理所有的括号，当我们处理完所有的括号后，我们检查我们的表达式是否可以被考虑。

我们必须等到最后才决定递归中形成的表达式是否是有效的表达式。有没有一种方法可以让我们从早期的一些递归路径中切断，因为它们不会导致一个解决方案？答案是肯定的！优化基于以下思想。

对于递归过程中遇到的左括号，如果我们决定考虑它，那么它可能会导致或可能不会导致无效的最终表达式。如果后面没有匹配的右括号，最终可能导致表达式无效。但是，我们不确定这是否会发生。

> 但是，对于右括号，如果我们决定将其作为最终表达式的一部分保留（请记住，对于每个括号，我们都有两个选项，要么保留它，要么删除它并进一步递归），并且到目前为止表达式中没有对应的左括号来匹配它，那么无论之后做什么，它都肯定会导致无效表达式。

```
( ( ) ) )
```

在这种情况下，第三个右括号将使表达式无效。不管之后会发生什么，这都会给我们一个无效的表达式，如果发生这种情况，我们不应该进一步递归，只需要修剪递归树。

这就是为什么，除了在我们当前正在处理的原始字符串/表达式中具有索引之外，并且到目前为止已经形成了表达式字符串之外，我们还跟踪左括号和右括号的数量。每当我们在表达式中保留左括号时，我们就增加它的计数器。对于右括号，我们检查 `right_count < left_count` 。如果是这种情况，那么我们只考虑右括号，并进一步递归。否则，我们不知道它会使表达式无效。这个简单的优化节省了很多运行时间。

现在，让我们看看这个算法的实现。

复杂度分析

- 时间复杂度：$O(2^N)$。因为在最坏的情况下，表达式中只有左括号，对于每个括号，我们都有两个选项，即是删除还是考虑它。考虑到表达式有 N 括号，时间复杂性将为$O(2^N)$。
- 空间复杂度：$O(N)$，因为我们使用的是递归解决方案，对于递归解决方案，在递归过程中，始终有堆栈空间用作内部函数状态保存到堆栈中。递归的最大深度决定了所使用的堆栈空间。因为我们一次处理一个字符，而递归的基本情况是当我们处理完表达式字符串的所有字符时，堆栈的大小将为 $O(n)$。注意，我们不考虑存储有效表达式所需的空间。我们只计算中间过程的空间。

方法 2：BFS（较慢）

```
class Solution {
    public List<String> removeInvalidParentheses(String s) {
        Set<String> set = new HashSet<>();
        List<String> ans = new ArrayList<>();
        set.add(s);
        while (true) {
```

```
            for (String str : set) {
                if (isRegular(str))
                    ans.add(str);
            }
            if (ans.size() > 0) return ans;
            Set<String> nextSet = new HashSet<>();
            for (String str : set) {
                for (int i = 0; i < str.length(); i ++) {
                    if (str.charAt(i) == '(' || str.charAt(i) == ')')
                        nextSet.add(str.substring(0, i) + str.substring
(i + 1, str.length()));
                }
            }
            set = nextSet;
        }
    }

    public boolean isRegular(String s) {
        char[] ss = s.toCharArray();
        int count = 0;
        for (char c : ss) {
            if (c == '(') count ++;
            else if (c == ')') count --;
            if (count < 0) return false;
        }
        return count == 0;
    }
}
```

76、移除无效的括号（1249）

更多硬核内容就在公众号「yes 的练级攻略」

1249. 移除无效的括号

难度 中等 49 收藏 分享 切换为英文 关注 反馈

给你一个由 '('、')' 和小写字母组成的字符串 s。

你需要从字符串中删除最少数目的 '(' 或者 ')' (可以删除任意位置的括号)，使得剩下的「括号字符串」有效。

请返回任意一个合法字符串。

有效「括号字符串」应当符合以下 **任意一条** 要求：

- 空字符串或只包含小写字母的字符串
- 可以被写作 AB（A 连接 B）的字符串，其中 A 和 B 都是有效「括号字符串」
- 可以被写作 (A) 的字符串，其中 A 是一个有效的「括号字符串」

示例 1：

```
输入：s = "lee(t(c)o)de)"
输出："lee(t(c)o)de"
解释："lee(t(co)de)" , "lee(t(c)ode)" 也是一个可行答案。
```

示例 2：

```
输入：s = "a)b(c)d"
输出："ab(c)d"
```

示例 3：

```
输入：s = "))(("
输出：""
解释：空字符串也是有效的
```

示例 4：

```
输入：s = "(a(b(c)d)"
输出："a(b(c)d)"
```

提示：

- 1 <= s.length <= 10^5
- s[i] 可能是 '('、')' 或英文小写字母

https://leetcode-cn.com/problems/minimum-remove-to-make-valid-parentheses/solution/java-zhan-shi-xian-xiang-jie-by-amanehayashi/

更多硬核内容就在公众号「yes 的练级攻略」

1. 题目分析

- 移除无效的括号→通过栈实现
- 我们可以使用**一个栈存储括号，一个** `StringBuilder` **进行字符串重组**，其具体思路为：
  - 扫描字符串，分类讨论：
    - 如果字符为小写字母，直接加入 `StringBuilder`
    - 如果字符为左括号，入栈，加入 `StringBuilder`
    - 如果字符为右括号：
      - 如果栈为空，该右括号一定非法，不加入
      - 如果栈不为空，**栈顶一定为左括号**，栈顶出栈，将右括号加入 `StringBuilder`
  - 扫描结束后：
    - 如果栈不为空，栈中一定全部是非法的左括号。假设此时栈大小为k，**从后向前删除k个左括号即可**
    - 如果栈为空，说明字符串本身合法
  - 将 `StringBuilder` 转换为字符串，返回

时间复杂度：$O(n)$，空间复杂度：$O(n)$

```
class Solution {
    public String minRemoveToMakeValid(String s) {
        Stack<Character> stack = new Stack<>();
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < s.length(); i ++){
            char c = s.charAt(i);
            if(c >= 97){
                sb.append(c);
            }
            else if(c == 41){
                if(stack.isEmpty()){
                    continue;
                }
                else {
                    stack.pop();
                    sb.append(c);
                }
            }
            else {
                stack.push(c);
                sb.append(c);
            }
        }
        for(int i = 0; i < stack.size(); i ++){
            sb.deleteCharAt(sb.lastIndexOf("("));
        }
        return sb.toString();
    }
}
```

77、强密码检验器（420）

420. 强密码检验器

难度 困难 47 收藏 分享 切换为英文 关注 反馈

一个强密码应满足以下所有条件：

1. 由至少6个，至多20个字符组成。
2. 至少包含一个小写字母，一个大写字母，和一个数字。
3. 同一字符不能连续出现三次 (比如 "...aaa..." 是不允许的, 但是 "...aa...a..." 是可以的)。

编写函数 strongPasswordChecker(s)，s 代表输入字符串，如果 s 已经符合强密码条件，则返回0；否则返回要将 s 修改为满足强密码条件的字符串所需要进行修改的最小步数。

插入、删除、替换任一字符都算作一次修改。

https://leetcode-cn.com/problems/strong-password-checker/solution/java-map-xiang-xi-zhu-shi-by-jake-song/

```java
class Solution {
    public int strongPasswordChecker(String s) {
        int upper = 1;
        int lower = 1;
        int digit = 1;
        int n = s.length();
        Map<Integer, Integer> m = new HashMap<Integer, Integer>();
        for (int i = 0; i < n; ) {
            // 统计缺少的大小写或者数字的数量
            if (Character.isLowerCase(s.charAt(i))) lower = 0;
            if (Character.isUpperCase(s.charAt(i))) upper = 0;
            if (Character.isDigit(s.charAt(i))) digit = 0;
            int j = i;
            // 记录在该位置上存在多少个一样的字符
            while (i < n && s.charAt(j) == s.charAt(i)) i ++;
            int length = i - j;
            if (length >= 3) m.put(j, i - j);
        }
        // 缺少的大小写或者数字的数量
        int miss = lower + upper + digit;
        int res = 0;
        if (n < 6) {
            int diff = 6 - n;
            // 当n < 6 返回和6之间的差值以及缺少的数量当中最大的值
            res = Math.max(diff, miss);
        } else {
            // over 是超出20部分的数量
            int over = Math.max(n - 20, 0);
            res += over;
            // 以下这个for循环的目的是让所有超过3的连续数字的大小是3 的倍数 + 2
            for (int k = 1; k < 3; k ++) {
                for (int p : m.keySet()) {
                    if (m.get(p) >= 3 && m.get(p) % 3 == k - 1 && over > 0) {
                        m.put(p, m.get(p) - Math.min(k, over));
```

```
                    over -= k;
                }
            }
        }
        // 如果此时 over 还是大于 0，  大于 3 的连续数列减到 2 为止
        for (int p : m.keySet()) {
            if (m.get(p) >= 3 && over > 0) {
                int need = m.get(p) - 2;
                m.put(p, m.get(p) - over);
                over -= need;
            }
        }
        int left = 0;
        // 连续数列的长度 / 3 就是需要替换操作的步数
        for (int p : m.keySet()) {
            if (m.get(p) >= 3) {
                left += m.get(p) / 3;
            }
        }
        res += Math.max(left, miss);
    }
    return res;
  }
}
```

78、基本计算器（224）

224. 基本计算器

难度 困难 253 收藏 分享 切换为英文 关注 反馈

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式可以包含左括号 (，右括号)，加号 + ，减号 -，非负整数和空格 。

示例 1:

```
输入: "1 + 1"
输出: 2
```

示例 2:

```
输入: " 2-1 + 2 "
输出: 3
```

示例 3:

```
输入: "(1+(4+5+2)-3)+(6+8)"
输出: 23
```

说明:

- 你可以假设所给定的表达式都是有效的。
- 请不要使用内置的库函数 eval 。

https://leetcode-cn.com/problems/basic-calculator/solution/javadie-dai-jie-fa-li-yong-zhan-shi-xian-yun-suan-/

```
class Solution {
```

```
public int calculate(String s) {
    LinkedList<Boolean> stack = new LinkedList<>();
    int result = 0, opr = 0; //result: 当前的结果值; opr: 当前的被加/被减数
    Character op = null; //当前要执行的运算符
    for (char c : s.toCharArray()) {
        if (c == '+' || c == '-') {
            if (op == null) {
                //遇到第一个运算符时，将 result 置为 opr（即第一个运算符左边的数字）
                result = opr;
            } else {
                //result = result +/- opr;
                result = cal(op, result, opr);
            }
            //根据栈顶元素决定是否反转运算符
            op = swap(stack.peek() == null ? false : stack.peek(), c);
            opr = 0;
        } else if (c == '(') {
            stack.push(op != null && op == '-');
        } else if (c == ')') {
            stack.pop();
        } else if (c != ' ') {
            opr = opr * 10 + c - '0';
        }
    }
    if (op == null) {
        //算式中没有运算符时，opr 就是最终结果
        return opr;
    } else {
        //否则将 result 与 opr（即算式中最右边的数字）执行一次运算
        return cal(op, result, opr);
    }
}
char swap(boolean swap, char c) {
    if (swap) {
        return c == '+' ? '-' : '+';
    } else {
        return c;
    }
}
private int cal(char op, int opr1, int opr2) {
    switch (op) {
        case '+':
        return opr1 + opr2;
        case '-':
```

```
                return opr1 - opr2;
                default:
                return 0;
            }
        }
}
```

79、基本计算器 2（227）【计算器（面试题 16.26）】

227. 基本计算器 II

难度 中等 175 收藏 分享 切换为英文 关注 反馈

实现一个基本的计算器来计算一个简单的字符串表达式的值。

字符串表达式仅包含非负整数，`+`， `-`，`*`，`/` 四种运算符和空格 `  `。整数除法仅保留整数部分。

示例 1:

```
输入: "3+2*2"
输出: 7
```

示例 2:

```
输入: " 3/2 "
输出: 1
```

示例 3:

```
输入: " 3+5 / 2 "
输出: 5
```

说明:

- 你可以假设所给定的表达式都是有效的。
- 请**不要**使用内置的库函数 `eval`。

yes的练级攻略

https://leetcode-cn.com/problems/basic-calculator-ii/solution/li-yong-zhan-jiang-si-ze-yun-suan-hua-jian-cheng-j/

此算法的思路很简单，先把乘除法的值计算出来，最终将所有的运算简化成只有加法。

1. 先跳过空格
2. 出现了数字则记录整个数字是多少，然后根据之前的运算符决定下一步：

- 如果是加号'+'，说明前面的运算独立于以后的运算，可以将结果暂时放入栈；
- 如果是减号'-'，可以看成 `-1 * tempNum`，然后将 `-tempNum` 入栈；
- 如果是乘号'*'或者除号'/'，由于前面的运算独立于此，可以先计算 `lastNum` 和 `tempNum` 积，然后结果入栈。

3. 最后将栈中的所有元素相加就是答案。

```
class Solution {
    public int calculate(String s) {
        Stack<Integer> numStack = new Stack<>();
        char lastOp = '+';
        char[] arr = s.toCharArray();
        for(int i = 0; i < arr.length; i ++){
            if(arr[i] == ' ') continue;
```

```
            if(Character.isDigit(arr[i])){
                int tempNum = arr[i] - '0';
                while(++i < arr.length && Character.isDigit(arr[i])){
                    tempNum = tempNum * 10 + (arr[i] - '0');
                }
                i--;
                if(lastOp == '+') numStack.push(tempNum);
                else if(lastOp == '-') numStack.push(-tempNum);
                else numStack.push(res(lastOp, numStack.pop(), tempNum));
            } else lastOp = arr[i];
        }
        int ans = 0;
        for(int num : numStack) ans += num;
        return ans;
    }
    private int res(char op, int a, int b){
        if(op == '*') return a * b;
        else if(op == '/') return a / b;
        else if(op == '+') return a + b; //其实加减运算可以忽略
        else return a - b;
    }
}
```

80、基本计算器 3（772）

772. 基本计算器 III

难度 困难 32 收藏 分享 切换为英文 关注 反馈

实现一个基本的计算器来计算简单的表达式字符串。

表达式字符串可以包含左括号 (和右括号)，加号 + 和减号 -，**非负** 整数和空格 。

表达式字符串只包含非负整数，+，-，*，/ 操作符，左括号 (，右括号) 和空格 。整数除法需要向下截断。

你可以假定给定的字符串总是有效的。所有的中间结果的范围为 [-2147483648, 2147483647]。

一些例子：

```
"1 + 1" = 2
" 6-4 / 2 " = 4
"2*(5+5*2)/3+(6/2+8)" = 21
"(2+6* 3+5- (3*14/7+2)*5)+3"=-12
```

注：**不要** 使用内置库函数 eval 。

https://leetcode-cn.com/problems/basic-calculator-iii/solution/di-gui-chai-jie-wen-ti-by-jinnzenn/

```
class Solution {
    private int index = 0;
```

```
public int calculate(String s) {
    char[] ch = s.toCharArray();
    return cal(ch);
}
private int cal(char[] ch){
    Deque<Integer> stack = new ArrayDeque<>();
    int num = 0;
    char sign = '+';
    for(; index < ch.length; index++){
        char c = ch[index];
        if(Character.isDigit(c)){
            num = num*10 + (c-'0');
        }
        if(c == '('){
            index++;//index 指针指到下一个字符
            num = cal(ch);
        }
        //当遇到了新的运算符，就要对上一个运算符 sign 和累计的数字 num 作运算
        //空格直接无视，i 继续前进
        //遇到字符串末尾，肯定是要结算的
        if(!Character.isDigit(c)&& c != ' ' || index == ch.length-1){
            int pre = 0;
            switch(sign){

                case '+':
                    stack.push(num);
                    break;
                case '-':
                    stack.push(-num);
                    break;
                case '*':
                    pre = stack.pop();
                    stack.push(pre * num);
                    break;
                case '/':
                    pre = stack.pop();
                    stack.push(pre / num);
                    break;
            }
            sign = c;
            num = 0;//计数归位
        }
        if(c == ')') break;//阶段，后面开始计算局部结果，返回
    }
```

```
        int res = 0;
        while(!stack.isEmpty()){
            res += stack.pop();
        }
        return res;
    }
}
```

81、基本计算器 4（770）

770. 基本计算器 IV

难度 困难 24 收藏 分享 切换为英文 关注 反馈

给定一个表达式 `expression` 如 `expression = "e + 8 - a + 5"` 和一个求值映射，如 `{"e": 1}`（给定的形式为 `evalvars = ["e"]` 和 `evalints = [1]`），返回表示简化表达式的标记列表，例如 `["-1*a","14"]`

- 表达式交替使用块和符号，每个块和符号之间有一个空格。
- 块要么是括号中的表达式，要么是变量，要么是非负整数。
- 块是括号中的表达式，变量或非负整数。
- 变量是一个由小写字母组成的字符串（不包括数字）。请注意，变量可以是多个字母，并注意变量从不具有像 `"2x"` 或 `"-x"` 这样的前导系数或一元运算符 。

表达式按通常顺序进行求值：先是括号，然后求乘法，再计算加法和减法。例如，`expression = "1 + 2 * 3"` 的答案是 `["7"]`。

输出格式如下：

- 对于系数非零的每个自变量项，我们按字典排序的顺序将自变量写在一个项中。例如，我们永远不会写像 `"b*a*c"` 这样的项，只写 `"a*b*c"`。
- 项的次数等于被乘的自变量的数目，并计算重复项。(例如，`"a*a*b*c"` 的次数为 4。)。我们先写出答案的最大次数项，用字典顺序打破关系，此时忽略词的前导系数。
- 项的前导系数直接放在左边，用星号将它与变量分隔开(如果存在的话)。前导系数 1 仍然要打印出来。
- 格式良好的一个示例答案是 `["-2*a*a*a", "3*a*a*b", "3*b*b", "4*a", "5*c", "-6"]`。
- 系数为 0 的项（包括常数项）不包括在内。例如，“0” 的表达式输出为 []。

示例：

```
输入：expression = "e + 8 - a + 5", evalvars = ["e"], evalints = [1]
输出：["-1*a","14"]

输入：expression = "e - 8 + temperature - pressure",
evalvars = ["e", "temperature"], evalints = [1, 12]
输出：["-1*pressure","5"]

输入：expression = "(e + 8) * (e - 8)", evalvars = [], evalints = []
输出：["1*e*e","-64"]

输入：expression = "7 - 7", evalvars = [], evalints = []
输出：[]

输入：expression = "a * b * c + b * a * c * 4", evalvars = [], evalints = []
输出：["5*a*b*c"]

输入：expression = "((a - b) * (b - c) + (c - a)) * ((a - b) + (b - c) * (c - a))",
evalvars = [], evalints = []
输出：
["-1*a*a*b*b","2*a*a*b*c","-1*a*a*c*c","1*a*b*b*b","-1*a*b*b*c","-1*a*b*c*c","1*a*c*c*c","-1*b*b*b*c","2*b
```

提示：

1. `expression` 的长度在 [1, 250] 范围内。
2. `evalvars, evalints` 在范围 [0, 100] 内，且长度相同。

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/basic-calculator-iv/solution/basic-calculator-iv-by-leetcode/

方法一：多项式类 【通过】

思路

构建一个 `Poly` 多项式类，实现这个多项式类的一些数学操作。

算法

单独实现每个方法都很直接，这里先列一下要实现的方法：

- `Poly:add(this, that)` 返回 `this + that` 的结果。
- `Poly:sub(this, that)` 返回 `this - that` 的结果。
- `Poly:mul(this, that)` 返回 `this * that` 的结果。
- `Poly:evaluate(this, evalmap)` 返回将所有的自由变量替换成 `evalmap` 指定常数之后的结果。
- `Poly:toList(this)` 返回正确的多项式输出格式。
- `Solution::combine(left, right, symbol)` 返回对 `左边（left)` 和 `右边(left)` 进行 `symobl` 操作之后的结果。
- `Solution::make(expr)` 创造一个新的 `Poly` 实例来表示常数或 `expr` 指定的变量。
- `Solution::parse(expr)` 将 `expr` 解析成一个 `Poly` 实例。

复杂度分析

- 时间复杂度：时间复杂度即为 $O(2^N + M)$，其中 N 为 `expression` 的长度，M 为 `evalvars` 和 `evalints` 的长度。
- 空间复杂度：$O(N + M)$。

```
class Solution {
    public List<String> basicCalculatorIV(String expression, String[] evalVars,
 int[] evalInts) {
        Map<String, Integer> evalMap = new HashMap();
        for (int i = 0; i < evalVars.length; ++i)
            evalMap.put(evalVars[i], evalInts[i]);

        return parse(expression).evaluate(evalMap).toList();
    }

    public Poly make(String expr) {
        Poly ans = new Poly();
        List<String> list = new ArrayList();
        if (Character.isDigit(expr.charAt(0))) {
            ans.update(list, Integer.valueOf(expr));
        } else {
            list.add(expr);
            ans.update(list, 1);
        }
        return ans;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    }

    public Poly combine(Poly left, Poly right, char symbol) {
        if (symbol == '+') return left.add(right);
        if (symbol == '-') return left.sub(right);
        if (symbol == '*') return left.mul(right);
        throw null;
    }

    public Poly parse(String expr) {
        List<Poly> bucket = new ArrayList();
        List<Character> symbols = new ArrayList();
        int i = 0;
        while (i < expr.length()) {
            if (expr.charAt(i) == '(') {
                int bal = 0, j = i;
                for (; j < expr.length(); ++j) {
                    if (expr.charAt(j) == '(') bal++;
                    if (expr.charAt(j) == ')') bal--;
                    if (bal == 0) break;
                }
                bucket.add(parse(expr.substring(i+1, j)));
                i = j;
            } else if (Character.isLetterOrDigit(expr.charAt(i))) {
                int j = i;
                search : {
                    for (; j < expr.length(); ++j)
                        if (expr.charAt(j) == ' ') {
                            bucket.add(make(expr.substring(i, j)));
                            break search;
                        }
                    bucket.add(make(expr.substring(i)));
                }
                i = j;
            } else if (expr.charAt(i) != ' ') {
                symbols.add(expr.charAt(i));
            }
            i++;
        }

        for (int j = symbols.size() - 1; j >= 0; --j)
            if (symbols.get(j) == '*')
                bucket.set(j, combine(bucket.get(j), bucket.remove(j+1), symbol
s.remove(j)));
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (bucket.isEmpty()) return new Poly();
        Poly ans = bucket.get(0);
        for (int j = 0; j < symbols.size(); ++j)
            ans = combine(ans, bucket.get(j+1), symbols.get(j));

        return ans;
    }
}

class Poly {
    HashMap<List<String>, Integer> count;
    Poly() {count = new HashMap();}

    void update(List<String> key, int val) {
        this.count.put(key, this.count.getOrDefault(key, 0) + val);
    }

    Poly add(Poly that) {
        Poly ans = new Poly();
        for (List<String> k: this.count.keySet())
            ans.update(k, this.count.get(k));
        for (List<String> k: that.count.keySet())
            ans.update(k, that.count.get(k));
        return ans;
    }

    Poly sub(Poly that) {
        Poly ans = new Poly();
        for (List<String> k: this.count.keySet())
            ans.update(k, this.count.get(k));
        for (List<String> k: that.count.keySet())
            ans.update(k, -that.count.get(k));
        return ans;
    }

    Poly mul(Poly that) {
        Poly ans = new Poly();
        for (List<String> k1: this.count.keySet())
            for (List<String> k2: that.count.keySet()) {
                List<String> kNew = new ArrayList();
                for (String x: k1) kNew.add(x);
                for (String x: k2) kNew.add(x);
                Collections.sort(kNew);
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                ans.update(kNew, this.count.get(k1) * that.count.get(k2));
            }
        return ans;
    }

    Poly evaluate(Map<String, Integer> evalMap) {
        Poly ans = new Poly();
        for (List<String> k: this.count.keySet()) {
            int c = this.count.get(k);
            List<String> free = new ArrayList();
            for (String token: k) {
                if (evalMap.containsKey(token))
                    c *= evalMap.get(token);
                else
                    free.add(token);
            }
            ans.update(free, c);
        }
        return ans;
    }

    int compareList(List<String> A, List<String> B) {
        int i = 0;
        for (String x: A) {
            String y = B.get(i++);
            if (x.compareTo(y) != 0) return x.compareTo(y);
        }
        return 0;
    }
    List<String> toList() {
        List<String> ans = new ArrayList();
        List<List<String>> keys = new ArrayList(this.count.keySet());
        Collections.sort(keys, (a, b) ->
            a.size() != b.size() ? b.size() - a.size() : compareList(a, b));

        for (List<String> key: keys) {
            int v = this.count.get(key);
            if (v == 0) continue;
            StringBuilder word = new StringBuilder();
            word.append("" + v);
            for (String token: key) {
                word.append('*');
                word.append(token);
            }
```

```
                ans.add(word.toString());
            }
            return ans;
        }
}
```

82、逆波兰表达式求值（150）

150. 逆波兰表达式求值

难度 中等 182 收藏 分享 切换为英文 关注 反馈

根据 逆波兰表示法，求表达式的值。

有效的运算符包括 `+`, `-`, `*`, `/` 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
输入: ["2", "1", "+", "3", "*"]
输出: 9
解释: 该算式转化为常见的中缀算术表达式为: ((2 + 1) * 3) = 9
```

示例 2：

```
输入: ["4", "13", "5", "/", "+"]
输出: 6
解释: 该算式转化为常见的中缀算术表达式为: (4 + (13 / 5)) = 6
```

示例 3：

```
输入: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
输出: 22
解释:
该算式转化为常见的中缀算术表达式为:
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

逆波兰表达式：

逆波兰表达式是一种后缀表达式，所谓后缀就是指算符写在后面。

- 平常使用的算式则是一种中缀表达式，如 `( 1 + 2 ) * ( 3 + 4 )` 。
- 该算式的逆波兰表达式写法为 `( ( 1 2 + ) ( 3 4 + ) * )` 。

逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成 `1 2 + 3 4 + *` 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

https://leetcode-cn.com/problems/evaluate-reverse-polish-notation/solution/javachun-shu-zu-he-zhan-liang-chong-shi-xian-jian-/

其他Java相关优化操作：

1. 数组最大长度为 `tokens.length / 2 + 1`
2. switch代替if-else，效率优化
3. Integer.parseInt代替Integer.valueOf,减少自动拆箱装箱操作

附两种方法：
纯数组模拟栈实现(推荐)：

方法 1：纯数组模拟栈

```
class Solution {
    //纯数组模拟栈实现(推荐)    3 ms    36 MB
    public static int evalRPN(String[] tokens) {
        int[] numStack = new int[tokens.length / 2 + 1];
        int index = 0;
        for (String s : tokens) {
            switch (s) {
            case "+":
                numStack[index - 2] += numStack[--index];
                break;
            case "-":
                numStack[index - 2] -= numStack[--index];
                break;
            case "*":
                numStack[index - 2] *= numStack[--index];
                break;
            case "/":
                numStack[index - 2] /= numStack[--index];
                break;
            default:
                // numStack[index++] = Integer.valueOf(s);
                //valueOf 改为parseInt，减少自动拆箱装箱操作
                numStack[index++] = Integer.parseInt(s);
                break;
            }
        }
        return numStack[0];
    }
}
```

方法 2：用栈

```
class Solution {
    // 栈实现    8 ms    36.7 MB
    public static int evalRPN(String[] tokens) {
        Stack<Integer> numStack = new Stack<>();
        Integer op1, op2;
        for (String s : tokens) {
            switch (s) {
```

```
                case "+":
                    op2 = numStack.pop();
                    op1 = numStack.pop();
                    numStack.push(op1 + op2);
                    break;
                case "-":
                    op2 = numStack.pop();
                    op1 = numStack.pop();
                    numStack.push(op1 - op2);
                    break;
                case "*":
                    op2 = numStack.pop();
                    op1 = numStack.pop();
                    numStack.push(op1 * op2);
                    break;
                case "/":
                    op2 = numStack.pop();
                    op1 = numStack.pop();
                    numStack.push(op1 / op2);
                    break;
                default:
                    numStack.push(Integer.valueOf(s));
                    break;
                }
            }
            return numStack.pop();
        }
    }
```

83、Excel 表列序号（171）

171. Excel表列序号

难度 简单 171 收藏 分享 切换为英文 关注 反馈

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```
A -> 1
B -> 2
C -> 3
...
Z -> 26
AA -> 27
AB -> 28
...
```

示例 1:

```
输入: "A"
输出: 1
```

示例 2:

```
输入: "AB"
输出: 28
```

示例 3:

```
输入: "ZY"
输出: 701
```

致谢:
特别感谢 @ts 添加此问题并创建所有测试用例。

https://leetcode-cn.com/problems/excel-sheet-column-number/solution/hua-jie-suan-fa-171-excelbiao-lie-xu-hao-by-guanpe/

解题方案

思路

- 标签：字符串遍历，进制转换
- 初始化结果 `ans = 0` ，遍历时将每个字母与A做减法，因为A表示1，所以减法后需要每个数加1，计算其代表的数值 `num = 字母 - 'A' + 1`
- 因为有26个字母，所以相当于26进制，每26个数则向前进一位
- 所以每遍历一位则 `ans = ans * 26 + num`
- 以ZY为例，Z的值为26，Y的值为25，则结果为 `26 * 26 + 25=701`
- 时间复杂度：O(n)

```java
class Solution {
    public int titleToNumber(String s) {
        int ans = 0;
        for(int i=0;i<s.length();i++) {
            int num = s.charAt(i) - 'A' + 1;
            ans = ans * 26 + num;
        }
        return ans;
```

```
    }
}
```

84、Excel 表列名称（168）

168. Excel表列名称

难度 简单 261 收藏 分享 切换为英文 关注 反馈

给定一个正整数，返回它在 Excel 表中相对应的列名称。

例如，

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
...
```

示例 1:

```
输入: 1
输出: "A"
```

示例 2:

```
输入: 28
输出: "AB"
```

示例 3:

```
输入: 701
输出: "ZY"
```

https://leetcode-cn.com/problems/excel-sheet-column-title/solution/jian-yi-li-jie-by-quan-zhi-pi-pei/

解题思路

正常情况：二十六进制的范围是 0-25 ，所以除以 26 可以把 末尾元素 去掉，
但这道题的范围是 1-26 即位上元素是A~Z，不存在0！
为了转成我们熟悉的进制运算，我们需要把原本的数值减一进行运算
这样原本的26减成了25，也就是Z('A'+25)，
1减成了0,存入了A('A'+0)

```
class Solution {
    public String convertToTitle(int n) {
        StringBuilder stringBuilder = new StringBuilder();
        while (n != 0) {
            n--;
            stringBuilder.append((char) ('A' + n % 26));
            n /= 26;
        }
        return stringBuilder.reverse().toString();
    }
}
```

85、二进制数转字符串（面试题 05.02）

面试题 05.02. 二进制数转字符串

难度 中等 9 收藏 分享 切换为英文 关注 反馈

二进制数转字符串。给定一个介于0和1之间的实数（如0.72），类型为double，打印它的二进制表达式。如果该数字不在0和1之间，或者无法精确地用32位以内的二进制表示，则打印“ERROR”。

示例1:

```
输入：0.625
输出："0.101"
```

示例2:

```
输入：0.1
输出："ERROR"
提示：0.1无法被二进制准确表示
```

提示:

1. 32位包括输出中的"0."这两位。

https://leetcode-cn.com/problems/bianry-number-to-string-lcci/solution/shuang-bai-by-dushi/

```java
class Solution {
    public String printBin(double num) {
        StringBuffer sb = new StringBuffer();
        sb.append("0.");
        while(num != 0) {
            num *= 2;
            if(num >= 1) {
                sb.append(1);
                num -= 1;
            }else {
                sb.append(0);
            }
            if(sb.length() > 32)
                return "ERROR";
        }
        return sb.toString();
    }
}
```

86、兰顿蚂蚁（面试题 16.22）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.22. 兰顿蚂蚁

难度 中等 9 收藏 分享 切换为英文 关注 反馈

一只蚂蚁坐在由白色和黑色方格构成的无限网格上。开始时，网格全白，蚂蚁面向右侧。每行走一步，蚂蚁执行以下操作。

(1) 如果在白色方格上，则翻转方格的颜色，向右(顺时针)转 90 度，并向前移动一个单位。
(2) 如果在黑色方格上，则翻转方格的颜色，向左(逆时针方向)转 90 度，并向前移动一个单位。

编写程序来模拟蚂蚁执行的前 K 个动作，并返回最终的网格。

网格由数组表示，每个元素是一个字符串，代表网格中的一行，黑色方格由 `'X'` 表示，白色方格由 `'_'` 表示，蚂蚁所在的位置由 `'L'`, `'U'`, `'R'`, `'D'` 表示，分别表示蚂蚁 左、上、右、下 的朝向。只需要返回能够包含蚂蚁走过的所有方格的最小矩形。

示例 1:

```
输入: 0
输出: ["R"]
```

示例 2:

```
输入: 2
输出:
[
  "_X",
  "LX"
]
```

示例 3:

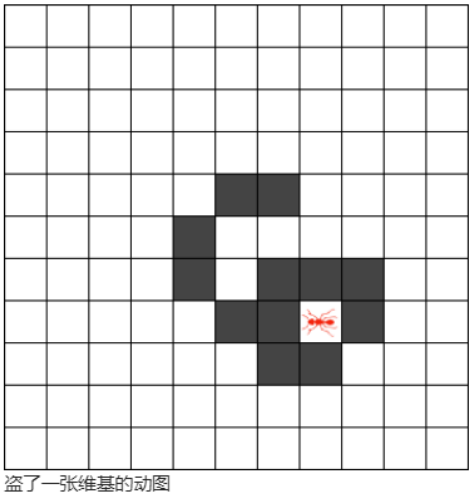
```
输入: 5
输出:
[
  "_U",
  "X_",
  "XX"
]
```

说明:

- `K <= 100000`

https://leetcode-cn.com/problems/langtons-ant-lcci/solution/28miao-shuang-100nei-bu-lei-san-lie-ji-he-by-1ujin/

目前比执行用时分布图表里最快的还快。
多看题目下方的提示，“并不真的需要一个网格”，蚂蚁的下一步坐标可以由当前状态计算出，所以保存并时刻更新蚂蚁的状态，用集合保存块的位置，最后再通过集合生成字符串列表。详细思路全在注释中了。



盗了一张维基的动图

```
class Solution {
    private class Position {
        // 横坐标 x 纵坐标 y
        int x, y;
        public Position(int x, int y) {
            this.x = x;
            this.y = y;
        }
        @Override
        public boolean equals(Object obj) {
            if (obj == this) return true;
            if (!(obj instanceof Position)) return false;
            Position o = (Position) obj;
            return x == o.x && y == o.y;
        }

        // 改写哈希算法，使两个 Position 对象可以比较坐标而不是内存地址
        @Override
        public int hashCode() {
            int result = x;
            result = 31 * result + y;
            return result;
        }
    }

    public List<String> printKMoves(int K) {
        char[] direction = {'L', 'U', 'R', 'D'};
```

```
        // 用“向量”记录方向，顺序与上一行方向的字符顺序保持一致，每个元素的后一个元素都是可以 90° 向右变换得到的
        int[][] offset = {{-1, 0}, {0, -1}, {1, 0}, {0, 1}};
        // 蚂蚁的位置
        Position antPos = new Position(0, 0);
        // 蚂蚁方向的向量序号
        int antDir = 2;
        // 用集合存储所有黑块的坐标，一开始想再定义一个路径的坐标集合，发现可以直接用黑块+蚂蚁位置也能过
        Set<Position> blackSet = new HashSet<>();
        while (K > 0) {
            // 新的坐标对象用于放入集合
            Position t = new Position(antPos.x, antPos.y);
            // 如果黑块集合能存入，说明脚下的块不在集合中，也就意味着是白色，方向序号循环自增 1
            if (blackSet.add(t)) antDir = (antDir + 1) % 4;
            else {
                // 否则说明脚下的块已经在集合中，也就意味着是黑色，方向序号循环自增 3，相当于自减 1，但是 Math.floorMod 取模可能消耗大？用+3 替代
                antDir = (antDir + 3) % 4;
                // 别忘了删除，即将黑块变白
                blackSet.remove(t);
            }
            // 蚂蚁移动位置
            antPos.x += offset[antDir][0];
            antPos.y += offset[antDir][1];
            K--;
        }
        // 计算边界，即输出网格的行数和列数
        int left = antPos.x, top = antPos.y, right = antPos.x, bottom = antPos.y;
        for (Position pos : blackSet) {
            left = pos.x < left ? pos.x : left;
            top = pos.y < top ? pos.y : top;
            right = pos.x > right ? pos.x : right;
            bottom = pos.y > bottom ? pos.y : bottom;
        }
        char[][] grid = new char[bottom - top + 1][right - left + 1];
        // 填充白块
        for (char[] row : grid)
            Arrays.fill(row, '_');
        // 替换黑块
        for (Position pos : blackSet)
            grid[pos.y - top][pos.x - left] = 'X';
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        // 替换蚂蚁
        grid[antPos.y - top][antPos.x - left] = direction[antDir];
        // 利用网格生成字符串列表
        List<String> result = new ArrayList<>();
        for (char[] row : grid)
            result.add(String.valueOf(row));
        return result;
    }
}
```

十四、回文

1、最长回文子串（5）

5. 最长回文子串

难度 中等 2318 收藏 分享 切换为英文 关注 反馈

给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

示例 1：

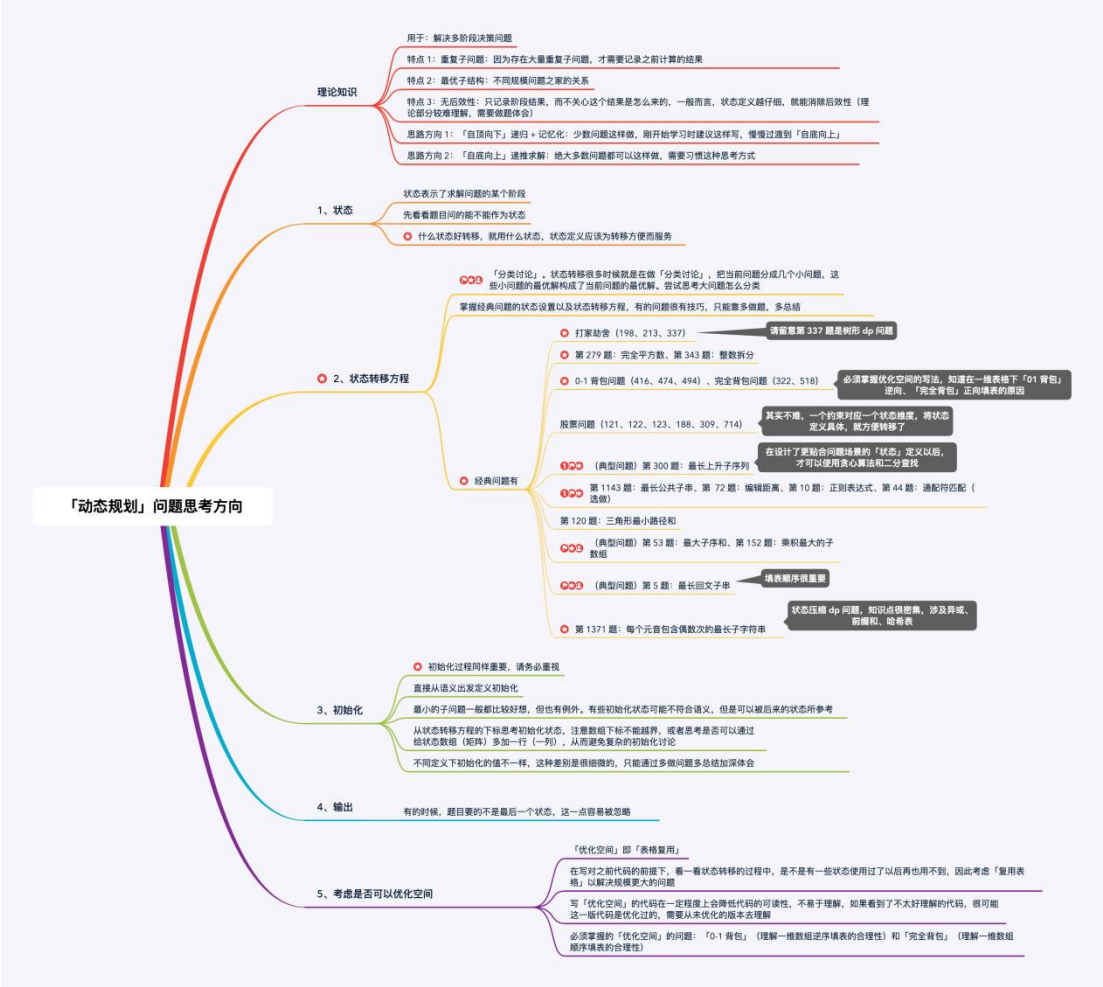
```
输入："babad"
输出："bab"
注意："aba" 也是一个有效答案。
```

示例 2：

```
输入："cbbd"
输出："bb"
```

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」



参考链接：

https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zhong-xin-kuo-san-dong-tai-gui-hua-by-liweiwei1419/

1、思考状态（重点）

- 状态的定义，先尝试「题目问什么，就把什么设置为状态」；
- 然后思考「状态如何转移」，如果「状态转移方程」不容易得到，尝试修改定义，目的依然是为了方便得到「状态转移方程」。

「状态转移方程」是原始问题的不同规模的子问题的联系。即大问题的最优解如何由小问题的最优解得到。

2、思考状态转移方程（核心、难点）

- 状态转移方程是非常重要的，是动态规划的核心，也是难点；
- 常见的推导技巧是：分类讨论。即：对状态空间进行分类；
- 归纳「状态转移方程」是一个很灵活的事情，通常是具体问题具体分析；
- 除了掌握经典的动态规划问题以外，还需要多做题；
- 如果是针对面试，请自行把握难度。掌握常见问题的动态规划解法，理解动态规划解决问题，是从一个小规模问题出发，逐步得到大问题的解，并记录中间过程；
- 动态规划方法依然是「空间换时间」思想的体现，常见的解决问题的过程很像在「填表」。

3、思考初始化

初始化是非常重要的，一步错，步步错。初始化状态一定要设置对，才可能得到正确的结果。

- 角度 1：直接从状态的语义出发；

- 角度 2：如果状态的语义不好思考，就考虑「状态转移方程」的边界需要什么样初始化的条件；
- 角度 3：从「状态转移方程」方程的下标看是否需要多设置一行、一列表示「哨兵」（sentinel），这样可以避免一些特殊情况的讨论。

4、思考输出

有些时候是最后一个状态，有些时候可能会综合之前所有计算过的状态。

5、思考优化空间（也可以叫做表格复用）

- 「优化空间」会使得代码难于理解，且是的「状态」丢失原来的语义，初学的时候可以不一步到位。先把代码写正确是更重要；
- 「优化空间」在有一种情况下是很有必要的，那就是状态空间非常庞大的时候（处理海量数据），此时空间不够用，就必须「优化空间」；
- 非常经典的「优化空间」的典型问题是「0-1 背包」问题和「完全背包」问题。

（下面是这道问题「动态规划」方法的分析）

这道题比较烦人的是判断回文子串。因此需要一种能够快速判断原字符串的所有子串是否是回文子串的方法，于是想到了「动态规划」。

「动态规划」的一个关键的步骤是想清楚「状态如何转移」。事实上，「回文」天然具有「状态转移」性质。

- 一个回文去掉两头以后，剩下的部分依然是回文（这里暂不讨论边界情况）；

依然从回文串的定义展开讨论：

- 如果一个字符串的头尾两个字符都不相等，那么这个字符串一定不是回文串；
- 如果一个字符串的头尾两个字符相等，才有必要继续判断下去。
    - 如果里面的子串是回文，整体就是回文串；
    - 如果里面的子串不是回文串，整体就不是回文串。

即：在头尾字符相等的情况下，里面子串的回文性质据定了整个子串的回文性质，这就是状态转移。因此可以把「状态」定义为原字符串的一个子串是否为回文子串。

禁止商用
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

第 1 步：定义状态

`dp[i][j]` 表示子串 `s[i..j]` 是否为回文子串，这里子串 `s[i..j]` 定义为左闭右闭区间，可以取到 `s[i]` 和 `s[j]`。

第 2 步：思考状态转移方程

在这一步分类讨论（根据头尾字符是否相等），根据上面的分析得到：

```
dp[i][j] = (s[i] == s[j]) and dp[i + 1][j - 1]
```

说明：

- 「动态规划」事实上是在填一张二维表格，由于构成子串，因此 `i` 和 `j` 的关系是 `i <= j`，因此，只需要填这张表格对角线以上的部分。
- 看到 `dp[i + 1][j - 1]` 就得考虑边界情况。

边界条件是：表达式 `[i + 1, j - 1]` 不构成区间，即长度严格小于 `2`，即 `j - 1 - (i + 1) + 1 < 2`，整理得 `j - i < 3`。

这个结论很显然：`j - i < 3` 等价于 `j - i + 1 < 4`，即当子串 `s[i..j]` 的长度等于 `2` 或者等于 `3` 的时候，其实只需要判断一下头尾两个字符是否相等就可以直接下结论了。

- 如果子串 `s[i + 1..j - 1]` 只有 1 个字符，即去掉两头，剩下中间部分只有 1 个字符，显然是回文；
- 如果子串 `s[i + 1..j - 1]` 为空串，那么子串 `s[i, j]` 一定是回文子串。

因此，在 `s[i] == s[j]` 成立和 `j - i < 3` 的前提下，直接可以下结论，`dp[i][j] = true`，否则才执行状态转移。

第 3 步：考虑初始化

初始化的时候，单个字符一定是回文串，因此把对角线先初始化为 `true`，即 `dp[i][i] = true`。

事实上，初始化的部分都可以省去。因为只有一个字符的时候一定是回文，`dp[i][i]` 根本不会被其它状态值所参考。

第 4 步：考虑输出

只要一得到 `dp[i][j] = true`，就记录子串的长度和起始位置，没有必要截取，这是因为截取字符串也要消耗性能，记录此时的回文子串的「起始位置」和「回文长度」即可。

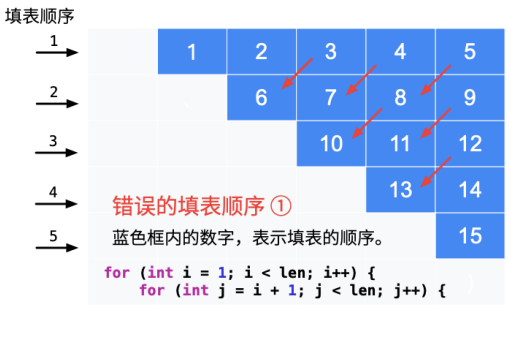
第 5 步：考虑优化空间

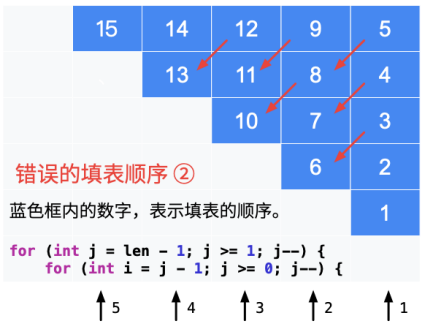
因为在填表的过程中，只参考了左下方的数值。事实上可以优化，但是增加了代码编写和理解的难度，丢失可读和可解释性。在这里不优化空间。

注意事项：总是先得到小子串的回文判定，然后大子串才能参考小子串的判断结果，即填表顺序很重要。

大家能够可以自己动手，画一下表格，相信会对「动态规划」作为一种「表格法」有一个更好的理解。

更多硬核内容就在公众号「yes 的练级攻略」



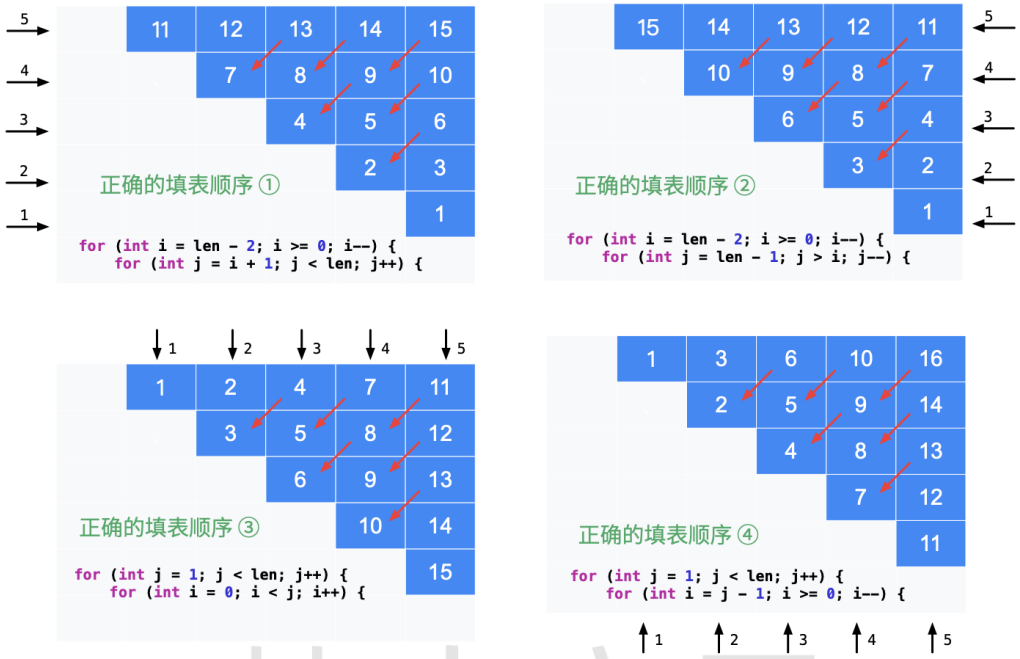


填表的时候参考的值还未计算出来，这叫“有后效性”，
即：后面将要计算的状态值需要参考的状态值还未计算出来。此时状态转移是错误的。

图中的代码是 Java 代码，只有两层 `for` 循环的部分，
其中 `i` 是左边界下标，可以取到；
其中 `j` 是右边界下标，也可以取到。

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」



新值的计算参考值是正确的，满足“无后效性”。
从填表的过程中，也可以看出：

1、对角线的元素不会被其它状态值所参考，所以可以不赋值；
2、对于“是否可以取等号”等边界问题，也可以从图中看出。

“动态规划”的“规划”（programming）就是“填表格”的意思，所以“动态规划”也叫“表格法”，它就是“以空间换时间”思想的体现。

而“动态”（dynamic）的意思是“从一个最基本的问题出发”。它不像我们熟悉的“递归 + 记忆化”的方法。“递归 + 记忆化”的方法是直接对问题求解，遇到的新问题，记录一下，再遇到的时候直接读取，这种方式叫“自顶向下”。

“自顶向下”的方式是人类的思维，就像我们日常的工作，在开发中遇到的问题，一定要记下来，记在笔记本或者博客里，下次再遇到的时候，就再查一下，而不用再计算一遍。

“动态”（dynamic）的思路是“自底向上”，我们通过发现这个问题“最初的样子”，即从“**从一个最基本的问题出发，逐步计算中间过程，最后得到要求问题的解**”。

“动态”的思想就体现在，我们不是直接面对要解决的问题。这种感觉就很像：一个学霸在考试的时候，他（她）遇到的题目全是他（她）做过的，并且他（她）都记在了脑子里（记住很重要，不记住是不行的），可以直接写出答案。所以有些资料上说“动态规划”像是开启了上帝视角，我觉得是很“贴切”的。

编程的世界就是这么神奇，人在现实世界里实现不了的事情，在编程里都能实现，例如“回溯算法”，人不可能“回到过去，回到过去一模一样的状态”，但是编程就可以啊。

禁止商用
公众号：yes的练级攻略

方法 1：动态规划（时间复杂度 o(n^2)，空间复杂度 o(n^2)）

```
class Solution {
    public String longestPalindrome(String s) {
        int len = s.length();
        if (len < 2) {
            return s;
        }
        boolean[][] dp = new boolean[len][len];
        for (int i = 0; i < len; i++) {
            Arrays.fill(dp[i], false);
```

```
        }
        for (int i = 0; i < len; i++) {
            dp[i][i] = true;
        }
        char[] c = s.toCharArray();
        int maxLen = 1;
        int start = 0;
        for (int j = 1; j < len; j++) {
            for (int i = j - 1; i >= 0; i--) {
                if (c[i] == c[j]) {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                } else {
                    dp[i][j] = false;
                }
                // 只要 dp[i][j] == true 成立，就表示子串 s[i, j] 是回文，此时记录回文长度和起始位置
                if (dp[i][j]) {
                    int curLen = j - i + 1;
                    if (curLen > maxLen) {
                        maxLen = curLen;
                        start = i;
                    }
                }
            }
        }
        return s.substring(start, start + maxLen);
    }
}
```

暴力法采用双指针两边夹，验证是否是回文子串。

除了枚举字符串的左右边界以外，比较容易想到的是枚举可能出现的回文子串的“中心位置”，从“中心位置”尝试尽可能扩散出去，得到一个回文串。

因此中心扩散法的思路是：遍历每一个索引，以这个索引为中心，利用“回文串”中心对称的特点，往两边扩散，看最多能扩散多远。

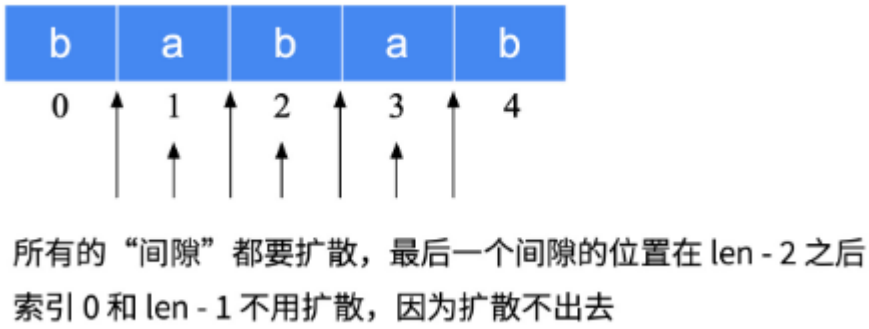
枚举“中心位置”时间复杂度为 O(N)，从“中心位置”扩散得到“回文子串”的时间复杂度为 O(N)，因此时间复杂度可以降到 O(N^2)。

在这里要注意一个细节：回文串在长度为奇数和偶数的时候，“回文中心”的形式是不一样的。

奇数回文串的“中心”是一个具体的字符，例如：回文串 "aba" 的中心是字符 "b"；

偶数回文串的“中心”是位于中间的两个字符的“空隙”，例如：回文串串 "abba" 的中心是两个 "b" 中间的那个“空隙”。

我们看一下一个字符串可能的回文子串的中心在哪里？



我们可以设计一个方法，兼容以上两种情况：
（1）如果传入重合的索引编码，进行中心扩散，此时得到的回文子串的长度是奇数；
（2）如果传入相邻的索引编码，进行中心扩散，此时得到的回文子串的长度是偶数。
具体编码细节在以下的代码的注释中体现。
参考链接：
https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zhong-xin-kuo-san-dong-tai-gui-hua-by-liweiwei1419/
方法 2：中心扩散法（时间复杂度 o(n^2)，空间复杂度 o(1)）

```
class Solution {
    public String longestPalindrome(String s) {
        if (s.length() < 2 ) {
            return s;
        }
        int maxLen = 1;
        String res = s.substring(0,1);
        // 中心位置枚举到 len - 2 即可
        for (int i = 0; i < s.length() - 1; i++){
            String oddStr = center(s,i,i);
            String evenStr = center(s,i,i+1);
            String maxLenStr = oddStr.length() > evenStr.length() ? odd
Str : evenStr;
            if (maxLenStr.length() > maxLen){
                maxLen = maxLenStr.length();
                res = maxLenStr;
            }
        }
```

```
        return res;
    }
    public String center(String s, int left, int right) {
        while (left >= 0 && right < s.length()) {
            if (s.charAt(left) == s.charAt(right)) {
                left--;
                right++;
            } else {
                break;
            }
        }
        // 这里要小心，跳出 while 循环时，恰好满
足 s.charAt(left) != s.charAt(right)，因此不能取 left，不能取 right
        return s.substring(left+1,right);
    }
}
```

方法 3：马拉车算法（Manacher 算法）【时间复杂度 o(n)，是最优解，但是比较难】

```
class Solution {
    public String longestPalindrome(String s) {
        if (s.length() < 2 ) {
            return s;
        }
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++){
            int len1 = center(s,i,i); //从一个字符扩展
            int len2 = center(s,i,i+1); //从两个字符之间扩展
            int len = Math.max(len1, len2);
            //根据 i 和 len 求得字符串的相应下标
            if (len > end - start) {
                start = i - (len-1)/2;
                end = i + len/2;
            }
        }
        return s.substring(start, end + 1);
    }
    //中间字符的索引
    public int center(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.c
harAt(right)) {
                left--;
                right++;
        }
        return right - left - 1;
    }
```

}

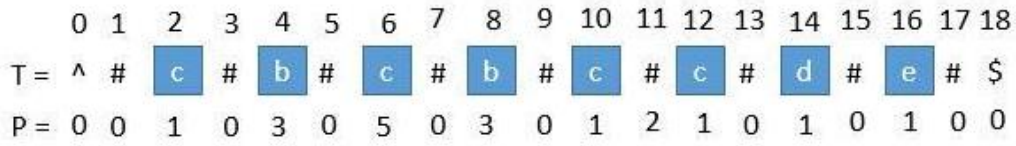
参考链接：

https://zhuanlan.zhihu.com/p/70532099

马拉车算法：

首先我们解决下奇数和偶数的问题，在每个字符间插入"#"，并且为了使得扩展的过程中，到边界后自动结束，在两端分别插入 "^" 和 "$"，两个不可能在字符串中出现的字符，这样中心扩展的时候，判断两端字符是否相等的时候，如果到了边界就一定会不相等，从而出了循环。经过处理，字符串的长度永远都是奇数了。

首先我们用一个数组 P 保存从中心扩展的最大个数，而它刚好也是去掉 "#" 的原字符串的总长度。例如下图中下标是 6 的地方。可以看到 P[6] 等于 5，所以它是从左边扩展 5 个字符，相应的右边也是扩展 5 个字符，也就是 "#c#b#c#b#c#"。而去掉 # 恢复到原来的字符串，变成 "cbcbc"，它的长度刚好也就是 5。



T：处理后的数组 P：从中心扩展的长度

求原字符串下标

用 P 的下标 i 减去 P[i]，再除以 2，就是原字符串的开头下标了。

例如我们找到 P[i] 的最大值为 5，也就是回文串的最大长度是 5，对应的下标是 6，所以原字符串的开头下标是 （6 - 5 ）/ 2 = 0 。所以我们只需要返回原字符串的第 0 到 第（5 - 1）位就可以了。

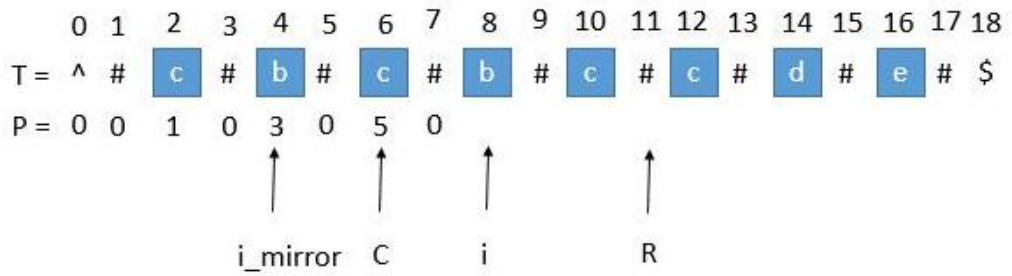
求每个 P[i]

接下来是算法的关键了，它充分利用了回文串的对称性。

我们用 C 表示回文串的中心，用 R 表示回文串的右边半径。所以 R = C + P[i] 。C 和 R 所对应的回文串是当前循环中 R 最靠右的回文串。

让我们考虑求 P[i] 的时候，如下图。

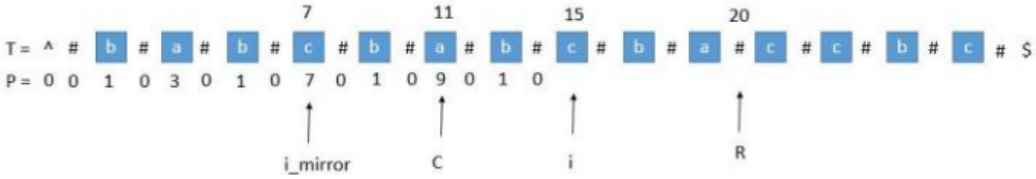
用 i_mirror 表示当前需要求的第 i 个字符关于 C 对应的下标。



我们现在要求 P [i]， 如果是用中心扩展法，那就向两边扩展比对就行了。但是我们其实可以利用回文串 C 的对称性。i 关于 C 的对称点是 i_mirror ，P [i_mirror] = 3，所以 P [i] 也等于 3 。

但是有三种情况将会造成直接赋值为 P [i_mirror] 是不正确的，下边一一讨论。

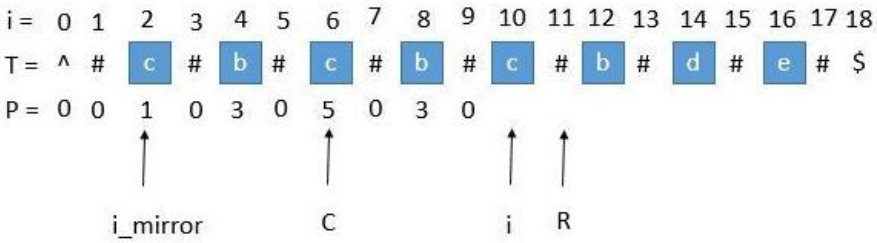
1. 超出了 R



当我们要求 P [i] 的时候，P [mirror] = 7，而此时 P [i] 并不等于 7 ，为什么呢，因为我们从 i 开始往后数 7 个，等于 22 ，已经超过了最右的 R ，此时不能利用对称性了，但我们一定可以扩展到 R 的，所以 P [i] 至少等于 R - i = 20 - 15 = 5，会不会更大呢，我们只需要比较 T [R+1] 和 T [R+1]关于 i 的对称点就行了，就像中心扩展法一样一个个扩展。

yes的练级攻略

2. P [i_mirror] 遇到了原字符串的左边界



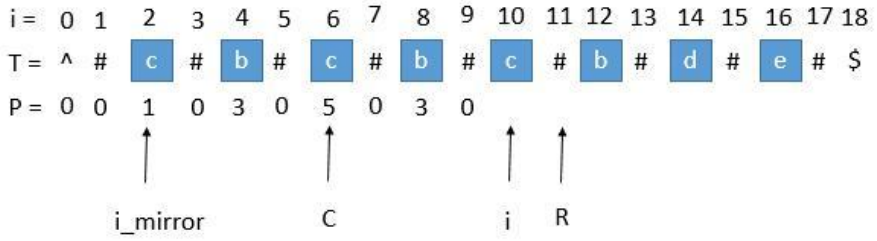
此时P [i_mirror] = 1，但是 P [i] 赋值成 1 是不正确的，出现这种情况的原因是 P [i_mirror] 在扩展的时候首先是 "#" == "#"，之后遇到了 "^"和另一个字符比较，也就是到了边界，才终止循环的。而 P [i] 并没有遇到边界，所以我们可以继续通过中心扩展法一步一步向两边扩展就行了。

3. i 等于了 R

此时我们先把 P [i] 赋值为 0，然后通过中心扩展法一步一步扩展就行了。

考虑 C 和 R 的更新

就这样一步一步的求出每个 P [i]，当求出的 P [i] 的右边界大于当前的 R 时，我们就需要更新 C 和 R 为当前的回文串了。因为我们必须保证 i 在 R 里面，所以一旦有更右边的 R 就要更新 R。



此时的 P [i] 求出来将会是 3，P [i] 对应的右边界将是 10 + 3 = 13，所以大于当前的 R，我们需要把 C 更新成 i 的值，也就是 10，R 更新成 13。继续下边的循环。

```
class Solution {
    public String longestPalindrome(String s) {
        String T = preProcess(s);
        int n = T.length();
        int[] P = new int[n];
// C 表示回文串的中心，用 R 表示回文串的右边半径，R = C + P[ i ]
```

```
        int C = 0, R = 0;
        for (int i = 1; i < n - 1; i++) {
            int i_mirror = 2 * C - i;
            if (R > i) {
                P[i] = Math.min(R - i, P[i_mirror]);// 防止超出 R
            } else {
                P[i] = 0;// 等于 R 的情况
            }
            // 碰到之前讲的三种情况时候，需要利用中心扩展法
            while (T.charAt(i + 1 + P[i]) == T.charAt(i - 1 - P[i])) {
                P[i]++;
            }
            // 判断是否需要更新 R
            if (i + P[i] > R) {
                C = i;
                R = i + P[i];
            }
        }
        // 找出 P 的最大值
        int maxLen = 0;
        int centerIndex = 0;
        for (int i = 1; i < n - 1; i++) {
            if (P[i] > maxLen) {
                maxLen = P[i];
                centerIndex = i;
            }
        }
        int start = (centerIndex - maxLen) / 2; //最开始讲的求原字符串下标
        return s.substring(start, start + maxLen);
    }
//preProcess 后的字符串形式：^#a#b#c#$
    public String preProcess(String s) {
        int n = s.length();
        if (n == 0) {
            return "^$";
        }
        String ret = "^";
        for (int i = 0; i < n; i++){
            ret += "#" + s.charAt(i);
        }
        ret += "#$";
        return ret;
    }
}
```

2、回文子串（647）【统计回文子串的数目（不删除字符）】

647. 回文子串

难度 中等 286 收藏 分享 切换为英文 关注 反馈

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被计为是不同的子串。

示例 1:

```
输入: "abc"
输出: 3
解释: 三个回文子串: "a", "b", "c".
```

示例 2:

```
输入: "aaa"
输出: 6
说明: 6个回文子串: "a", "a", "a", "aa", "aa", "aaa".
```

注意:

1. 输入的字符串长度不会超过1000。

https://leetcode-cn.com/problems/palindromic-substrings/solution/hui-wen-zi-chuan-by-leetcode/

方法一：从中心往两侧延伸【通过】

思路

在长度为 `N` 的字符串中，可能的回文串中心位置有 `2N-1` 个：字母，或两个字母中间。

从每一个回文串中心开始统计回文串数量。回文区间 `[a, b]` 表示 `S[a], S[a+1], ..., S[b]` 是回文串，根据回文串定义可知 `[a+1, b-1]` 也是回文区间。

算法

对于每个可能的回文串中心位置，尽可能扩大它的回文区间 `[left, right]`。当 `left >= 0 and right < N and S[left] == S[right]` 时，扩大区间。此时回文区间表示的回文串为 `S[left], S[left+1], ..., S[right]`。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 表示字符串 `S` 的长度，每次扩大回文区间的需要时间 $O(N)$。
- 空间复杂度：$O(1)$。

方法 1：中心扩散法（时间复杂度 o(n^2），空间复杂度 o(1)）

```
class Solution {
    public int countSubstrings(String s) {
        int n = s.length(), ans = 0;
//i 是中心位
```

```
            for (int i = 0; i <= 2 * n - 1; ++i) {
                int left = i / 2;
                //i % 2 可以用 i & 1 表示
                int right = left + (i & 1);
                while (left >= 0 && right < n && s.charAt(left) == s.charAt
(right)) {
                    ans++;
                    left--;
                    right++;
                }
            }
            return ans;
        }
}
```

方法二：马拉车算法【通过】

思路

马拉车算法可以在线性时间内找出以任何位置为中心的最大回文串。

算法

假设一个回文串中心为 `center`，该中心对应的最大回文串右边界为 `right`。存在一个 `i` 为当前回文串中心，满足 `i > center`，那么也存在一个 `j` 与 `i` 关于 `center` 对称，可以根据 `Z[i]` 快速计算出 `Z[j]`。

当 `i < right` 时，找出 `i` 关于 `center` 的对称点 `j = 2 * center - i`。此时以 `i` 为中心，半径为 `right - i` 的区间内存在的最大回文串的半径 `Z[i]` 等于 `Z[j]`。

例如，对于字符串 `A = '@#A#B#A#A#B#A#$'`，当 `center = 7, right = 13, i = 10` 时，`center` 为两个字母 `A` 中间的 `#`，最大回文串右边界为最后一个 `#`，`i` 是最后一个 `B`，`j` 是第一个 `B`。

在 `[center - (right - center), right]` 中，区间中心为 `center`，右边界为 `right`，`i` 和 `j` 关于 `center` 对称，且 `Z[j] = 3`，可以快速计算出 `Z[i] = min(right - i, Z[j]) = 3`。

在 while 循环中，只有当 `Z[i]` 超过 `right - i` 时，才需要逐个比较字符。这种情况下，`Z[i]` 每增加 `1`，`right` 也会增加 `1`，且最多能够增加 `2*N+2` 次。因此这个过程是线性的。

最后，对 `Z` 中每一项 `v` 计算 `(v+1) / 2`，然后求和。假设给定最大回文串中心为 `C`，半径为 `R`，那么以 `C` 为中心，半径为 `R-1, R-2, ..., 0` 的子串也都是回文串。例如 `abcdedcba` 是以 `e` 为中心，半径为 4 的回文串，那么 `e`，`ded`，`cdedc`，`bcdedcb` 和 `abcdedcba` 也都是回文串。

除以 2 是因为实际回文串的半径为 `v` 的一半。例如回文串 `a#b#c#d#e#d#c#b#a` 的半径为实际原回文串半径的 2 倍。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是 `S` 的长度。根据上面分析，复杂度是线性的。
- 空间复杂度：$O(N)$，数组 `A` 和 `Z` 的大小。

方法 2：马拉车（时间复杂度和空间复杂度都是 o(n)）

```
class Solution {
    public int countSubstrings(String s) {
        char[] A = new char[2 * s.length() + 3];
        A[0] = '^';
        A[1] = '#';
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        A[A.length - 1] = '$';
        int t = 2;
        for (char c : s.toCharArray()) {
            A[t++] = c;
            A[t++] = '#';
        }
        int[] Z = new int[A.length];
        int center = 0, right = 0;
        for (int i = 1; i < Z.length - 1; i++) {
            if (i < right) Z[i] = Math.min(right - i, Z[2 * center - i]
);
            while (A[i + Z[i] + 1] == A[i - Z[i] - 1]) Z[i]++;
            if (i + Z[i] > right) {
                center = i;
                right = i + Z[i];
            }
        }
        int ans = 0;
//((v + 1) >> 1)与 v/2 等效
        for (int v : Z) ans += ((v + 1) >> 1);
        return ans;
    }
}
```

3、最长回文子序列（516）

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

516. 最长回文子序列

难度 中等 276 收藏 分享 切换为英文 关注 反馈

给定一个字符串 `s` ，找到其中最长的回文子序列，并返回该序列的长度。可以假设 `s` 的最大长度为 `1000` 。

示例 1:
输入:

```
"bbbab"
```

输出:

```
4
```

一个可能的最长回文子序列为 "bbbb"。

示例 2:
输入:

```
"cbbd"
```

输出:

```
2
```

一个可能的最长回文子序列为 "bb"。

提示：

- `1 <= s.length <= 1000`
- `s` 只包含小写英文字母

https://leetcode-cn.com/problems/longest-palindromic-subsequence/solution/zi-xu-lie-wen-ti-tong-yong-si-lu-zui-chang-hui-wen/

子序列问题是常见的算法问题，而且并不好解决。

首先，子序列问题本身就相对子串、子数组更困难一些，因为前者是不连续的序列，而后两者是连续的，就算穷举你都不一定会，更别说求解相关的算法问题了。

而且，子序列问题很可能涉及到两个字符串，比如前文「最长公共子序列」，如果没有一定的处理经验，真的不容易想出来。所以本文就来扒一扒子序列问题的套路，其实就有两种模板，相关问题只要往这两种思路上想，十拿九稳。

一般来说，这类问题都是让你求一个**最长子序列**，因为最短子序列就是一个字符嘛，没啥可问的。一旦涉及到子序列和最值，那几乎可以肯定，**考察的是动态规划技巧，时间复杂度一般都是 O(n^2)**。

原因很简单，你想想一个字符串，它的子序列有多少种可能？起码是指数级的吧，这种情况下，不用动态规划技巧，还想怎么着？

既然要用动态规划，那就要定义 dp 数组，找状态转移关系。我们说的两种思路模板，就是 dp 数组的定义思路。不同的问题可能需要不同的 dp 数组定义来解决。

更多硬核内容就在公众号「yes 的练级攻略」

一、两种思路

1、第一种思路模板是一个一维的 dp 数组：

```
int n = array.length;
int[] dp = new int[n];

for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        dp[i] = 最值(dp[i], dp[j] + ...)
    }
}
```

举个我们写过的例子「最长递增子序列」，在这个思路中 dp 数组的定义是：

在子数组 `array[0..i]` 中，我们要求的子序列（最长递增子序列）的长度是 `dp[i]`。

为啥最长递增子序列需要这种思路呢？前文说得很清楚了，因为这样符合归纳法，可以找到状态转移的关系，这里就不具体展开了。

2、第二种思路模板是一个二维的 dp 数组：

```
int n = arr.length;
int[][] dp = new dp[n][n];

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (arr[i] == arr[j])
            dp[i][j] = dp[i][j] + ...
        else
            dp[i][j] = 最值(...)
    }
}
```

这种思路运用相对更多一些，尤其是涉及两个字符串/数组的子序列，比如前文讲的「最长公共子序列」。本思路中 dp 数组含义又分为「只涉及一个字符串」和「涉及两个字符串」两种情况。

2.1 涉及两个字符串/数组时（比如最长公共子序列），dp 数组的含义如下：

在子数组 `arr1[0..i]` 和子数组 `arr2[0..j]` 中，我们要求的子序列（最长公共子序列）长度为 `dp[i][j]`。

2.2 只涉及一个字符串/数组时（比如本文要讲的最长回文子序列），dp 数组的含义如下：

在子数组 `array[i..j]` 中，我们要求的子序列（最长回文子序列）的长度为 `dp[i][j]`。

第一种情况可以参考这两篇旧文：「编辑距离」「公共子序列」

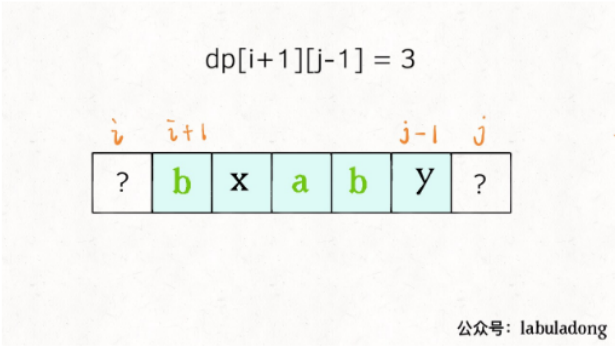
下面就借最长回文子序列这个问题，详解一下第二种情况下如何使用动态规划。

更多硬核内容就在公众号「yes 的练级攻略」

我们说这个问题对 dp 数组的定义是：**在子串 `s[i..j]` 中，最长回文子序列的长度为 `dp[i][j]`**。一定要记住这个定义才能理解算法。

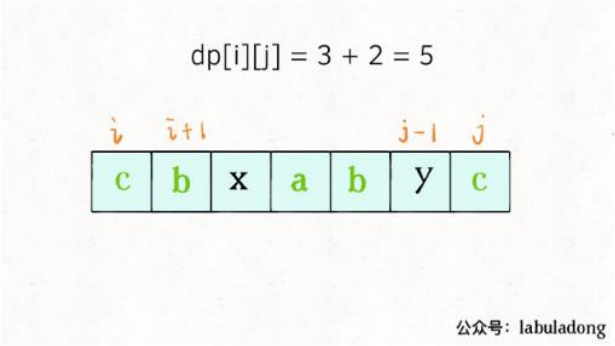
为啥这个问题要这样定义二维的 dp 数组呢？我们前文多次提到，**找状态转移需要归纳思维，说白了就是如何从已知的结果推出未知的部分**，这样定义容易归纳，容易发现状态转移关系。

具体来说，如果我们想求 `dp[i][j]`，假设你知道了子问题 `dp[i+1][j-1]` 的结果（`s[i+1..j-1]` 中最长回文子序列的长度），你是否能想办法算出 `dp[i][j]` 的值（`s[i..j]` 中，最长回文子序列的长度）呢？



可以！这取决于 `s[i]` 和 `s[j]` 的字符：

如果它俩相等，那么它俩加上 `s[i+1..j-1]` 中的最长回文子序列就是 `s[i..j]` 的最长回文子序列：



如果它俩不相等，说明它俩**不可能同时**出现在 `s[i..j]` 的最长回文子序列中，那么把它俩**分别**加入 `s[i+1..j-1]` 中，看看哪个子串产生的回文子序列更长即可：

无效的图片地址

以上两种情况写成代码就是这样：

```
if (s[i] == s[j])
    // 它俩一定在最长回文子序列中
    dp[i][j] = dp[i + 1][j - 1] + 2;
else
    // s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长？
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
```

至此，状态转移方程就写出来了，根据 dp 数组的定义，我们要求的就是 `dp[0][n - 1]`，也就是整个 `s` 的最长回文子序列的长度。

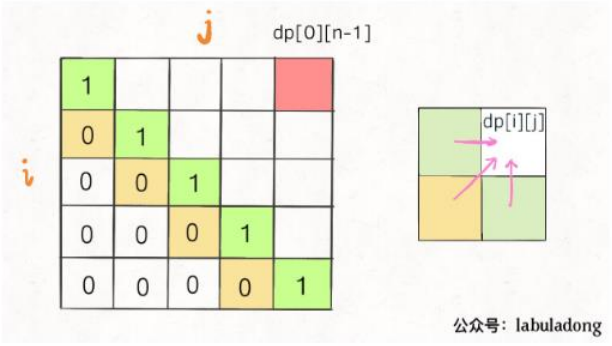
更多硬核内容就在公众号「yes 的练级攻略」

三、代码实现

首先明确一下 base case，如果只有一个字符，显然最长回文子序列长度是 1，也就是 `dp[i][j] = 1 (i == j)`。

因为 `i` 肯定小于等于 `j`，所以对于那些 `i > j` 的位置，根本不存在什么子序列，应该初始化为 0。

另外，看看刚才写的状态转移方程，想求 `dp[i][j]` 需要知道 `dp[i+1][j-1]`，`dp[i+1][j]`，`dp[i][j-1]` 这三个位置；再看看我们确定的 base case，填入 dp 数组之后是这样：



为了保证每次计算 `dp[i][j]`，左下右方向的位置已经被计算出来，只能斜着遍历或者反着遍历：

本题代码

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        int n = s.length();
        // dp 数组全部初始化为 0
        int[][] dp = new int[n][n];
        // base case
        for (int i = 0; i < n; i++)
            dp[i][i] = 1;
        // 反着遍历保证正确的状态转移
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                // 状态转移方程
                if (s.charAt(i) == s.charAt(j))
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                else
                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
        // 整个 s 的最长回文子串长度
        return dp[0][n - 1];
    }
}
```

打印一个最长回文子序列

```
class Solution {
    // 需要对去头、去尾，相等做一些标记，需要一个 path 数组
    public static int longestPalindromeSubseq(String s) {
        int n = s.length();
```

```
        int[][] dp = new int[n][n];
        int[][] path = new int[n][n];
        for (int i = 0; i < n; i++) {
            dp[i][i] = 1;
        }
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
                //如果是去头,path 上标记为 0，如果是去尾标记为 1，如果是相等，标
记为 2
                if (dp[i][j] == dp[i + 1][j]) {
                    path[i][j] = 0;
                } else if (dp[i][j] == dp[i][j - 1]) {
                    path[i][j] = 1;
                }
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = Math.max(dp[i][j], dp[i + 1][j - 1] + 2);
                    if (dp[i][j] == dp[i + 1][j - 1] + 2) {
                        path[i][j] = 2;
                    }
                }
            }
        }
        char[] res = new char[dp[0][n - 1]];
        //遍历 s 的两个指针
        int i = 0, j = n - 1;
        //填充 res 数组的两个指针
        int p = 0, q = dp[0][n - 1] - 1;
        while (i <= j) {
            if (i == j) {
                res[p] = s.charAt(i);
                break;
            }
            if (i + 1 == j) {
                res[p] = s.charAt(i);
                res[q] = s.charAt(j);
                break;
            }
            //其他情况，去头、去尾、相等
            if (path[i][j] == 0) {
                i++;
            } else if (path[i][j] == 1) {
                j--;
```

```
            } else {
                res[p++] = s.charAt(i++);
                res[q--] = s.charAt(j--);
            }
        }
        for (int k = 0; k < res.length; k++) {
            System.out.print(res[k] + " ");
        }
        return dp[0][n - 1];
    }
}
```

链接：
https://leetcode-cn.com/problems/longest-palindromic-subsequence/solution/mian-shi-guan-followupda-yin-yi-ge-zui-chang-hui-w/

4、统计不同回文子序列（730）【可以删除零至多个字符来获得子序列】

730. 统计不同回文子序列

难度 困难 64 收藏 分享 切换为英文 关注 反馈

给定一个字符串 S，找出 S 中不同的非空回文子序列个数，并返回该数字与 10^9 + 7 的模。

通过从 S 中删除 0 个或多个字符来获得子序列。

如果一个字符序列与它反转后的字符序列一致，那么它是回文字符序列。

如果对于某个 i，A_i != B_i，那么 A_1, A_2, ... 和 B_1, B_2, ... 这两个字符序列是不同的。

示例 1：

输入：
S = 'bccb'
输出： 6
解释：
6 个不同的非空回文子字符序列分别为：'b', 'c', 'bb', 'cc', 'bcb', 'bccb'。
注意：'bcb' 虽然出现两次但仅计数一次。

示例 2：

输入：
S = 'abcdabcdabcdabcdabcdabcdabcddcbadcbadcbadcbadcbadcbadcba'
输出： 104860361
解释：
共有 3104860382 个不同的非空回文子序列，对 10^9 + 7 取模为 104860361。

提示：

- 字符串 S 的长度将在 [1, 1000] 范围内。
- 每个字符 S[i] 将会是集合 {'a', 'b', 'c', 'd'} 中的某一个。

https://leetcode-cn.com/problems/count-different-palindromic-subsequences/solution/tong-ji-bu-tong-hui-wen-zi-zi-fu-chuan-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：动态规划（使用二维数组）

几乎每一个回文字符串都将采用这四种形式之一：`a_a`、`b_b`、`c_c` 或 `d_d`，其中 `_` 表示零个或多个字符的回文字符串。（其他回文字符串只有 `a`、`b`、`c`、`d` 和空字符串）

让我们试着数一数 `a_a` 形式的回文（其他形式是类似的）。我们应该取第一个和最后一个 `a`，然后计算所有可以在其间形成的回文字符串。

算法：

- 定义 `dp(i, j)` 是字符串 `T = S[i], S[i+1], ..., S[j]` 中的回文字符串个数（包括回文 `''`）。要在 `T` 中计算 `a_a` 形式的回文数，我们需要知道该字符串中 `'a'` 第一次和最后一次出现的位置。定义 `next[i][0]` 将是 `S[i:]` 中 `'a'` 的下一次出现的位置，`next[i][1]` 将是 `S[i:]` 中 `'b'` 下一次出现的位置，依此类推。
- 此外，我们还需要知道 `T` 中唯一字母的数目，才能计算出单个字母的回文数。我们可以用 `next` 得到的信息来推断它：如果 `next[i][0]` 在区间 `[i, j]` 中，那么 `'a'` 出现在 `T` 中，以此类推。
- 由于许多状态 `dp(i, j)` 不需要计算，最直接的方法是自顶向下变化的动态规划。

复杂度分析

- 时间复杂度：$O(N^2)$ 其中 N 是字符串 S 的长度。我们对 `prv` 和 `nxt` 的计算是在 $O(N)$ 时间内完成的，其中 `dp` 有最多 N^2 的状态，对每个状态的计算需要 $O(1)$ 的时间
- 空间复杂度：$O(N^2)$，`memo` 使用的空间。

具体动规代码

```
class Solution {
    int[][] memo, prv, nxt;
    byte[] A;
    int MOD = 1_000_000_007;
    public int countPalindromicSubsequences(String S) {
        int N = S.length();
        prv = new int[N][4];
        nxt = new int[N][4];
        memo = new int[N][N];
        for (int[] row: prv) Arrays.fill(row, -1);
        for (int[] row: nxt) Arrays.fill(row, -1);

        A = new byte[N];
        int ix = 0;
        for (char c: S.toCharArray()) {
            A[ix++] = (byte) (c - 'a');
        }

        int[] last = new int[4];
        Arrays.fill(last, -1);
        for (int i = 0; i < N; ++i) {
            last[A[i]] = i;
            for (int k = 0; k < 4; ++k)
                prv[i][k] = last[k];
        }
```

```
        Arrays.fill(last, -1);
        for (int i = N-1; i >= 0; --i) {
            last[A[i]] = i;
            for (int k = 0; k < 4; ++k)
                nxt[i][k] = last[k];
        }

        return dp(0, N-1) - 1;
    }

    public int dp(int i, int j) {
        if (memo[i][j] > 0) return memo[i][j];
        int ans = 1;
        if (i <= j) {
            for (int k = 0; k < 4; ++k) {
                int i0 = nxt[i][k];
                int j0 = prv[j][k];
                if (i <= i0 && i0 <= j) ans++;
                if (-1 < i0 && i0 < j0) ans += dp(i0 + 1, j0 - 1);
                if (ans >= MOD) ans -= MOD;
            }
        }
        memo[i][j] = ans;
        return ans;
    }
}
```

5、最长回文串（409）

409. 最长回文串

难度 简单 193 收藏 分享 切换为英文 关注 反馈

给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。

在构造过程中，请注意区分大小写。比如 "Aa" 不能当做一个回文字符串。

注意:
假设字符串的长度不会超过 1010。

示例 1:

输入:
"abccccdd"

输出:
7

解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。

ASCII 编码

更多硬核内容就在公众号「yes 的练级攻略」

ASCII 字符代码表 一

高四位 / 低四位		ASCII非打印控制字符 0000 0					ASCII非打印控制字符 0001 1					ASCII 打印字符 0010 2		ASCII 打印字符 0011 3		ASCII 打印字符 0100 4		ASCII 打印字符 0101 5		ASCII 打印字符 0110 6		ASCII 打印字符 0111 7		
		十进制	字符	ctrl	代码	字符解释	十进制	字符	ctrl	代码	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	ctrl
0000	0	0	BLANK NULL	^@	NUL	空	16	▶	^P	DLE	数据链路转意	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH	头标开始	17	◀	^Q	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX	正文开始	18	↕	^R	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX	正文结束	19	‼	^S	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT	传输结束	20	¶	^T	DC4	设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ	查询	21	§	^U	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK	确认	22	■	^V	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	震铃	23	↨	^W	ETB	传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	退格	24	↑	^X	CAN	取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	TAB	水平制表符	25	↓	^Y	EM	媒体结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	换行/新行	26	→	^Z	SUB	替换	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	竖直制表符	27	←	^[	ESC	转意	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	换页/新页	28	∟	^\	FS	文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	回车	29	↔	^]	GS	组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO	移出	30	▲	^6	RS	记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI	移入	31	▼	^-	US	单元分隔符	47	/	63	?	79	O	95	_	111	o	127	Δ	`Back space

注：表中的ASCII字符可以用:ALT + "小键盘上的数字键" 输入

参考链接：https://zhuanlan.zhihu.com/p/114535214

'z'-'A' = 122-65+1=58

用位运算代替取余，求余的优化因此我觉得 jvm 很难对模运算进行优化（当然也有可能优化，比如先判断一下是不是正数，但我觉得这么麻烦的话，可能并没有优化 ）总之，结论就是：能用 & 就不要取模。

4. 这里为啥不写成x - x % 2?

取模是一个消耗较大的操作，因此大多数语言的编译器比如C++都对模运算进行了优化，比如可以将 x%2 优化成 x&1，这种情况下怎么写都一样啦。但是Java比较特殊，Java中是不存在无符号整型的，数字是用补码来表示的（大家还记得补码吧...最高位是符号位，0表示正数，1表示负数），所以对于负数来说，%2 与 &1 两个操作是完全不同的（如下）：

```
int a = -3 % 2; // -1
int b = -3 & 1; // 1
```

因此我觉得jvm很难对模运算进行优化（当然也有可能优化，比如先判断一下是不是正数，但我觉得这么麻烦的话，可能并没有优化 ）总之，结论就是：**能用 & 就不要取模**。这里提一下一个常见的规律，对于模数是2的幂的情况下，都可以优化成&运算，如下：

```
int x = 87;
x % 2 = x & (2 - 1) = 1010111 & 1 = 1 = 1;
x % 4 = x & (4 - 1) = 1010111 & 11 = 11 = 3;
x % 8 = x & (8 - 1) = 1010111 & 111 = 111 = 7;
x % 16 = x & (16 - 1) = 1010111 & 1111 = 111 = 7;
x % 32 = x & (32 - 1) = 1010111 & 11111 = 10111 = 23;
```

方法 1

```
class Solution {
    public int longestPalindrome(String s) {
        //'z' - 'A' + 1= 122 - 65 + 1 = 58
// 1. counter 数组计数，统计每种字母出现的次数
        int[] count = new int[58];
        for (char c : s.toCharArray()) {
            count[c-'A']++;
        }
// 2. 构造回文串，每种字母使用偶数次
        int res = 0;
        for (int x :count) {
            //如果 x 是奇数，x & 1 的结果就是1，偶数就是0，实现了偶数不变、奇数减1的逻辑。当然这里还有另外一种位运算写法： (x >> 1) << 1 ，先右移一位去掉最末位的0或1，再左移一位，也实现了偶数不变、奇数减1的逻辑
            res += x - (x & 1);
        }
// 3. 如果最终的长度小于原字符串的长度，说明存在某个字符出现了奇数次，那可以在最中间再补一个字母。
        return res < s.length() ? res + 1 : res;
    }
}
```

方法 2：Java8 的流式写法【更慢】

```
class Solution {
    public int longestPalindrome(String s) {
        //s.chars()的返回值是一个 IntStream，就是Int 的流；
        //.boxed()会装箱返回Stream<Integer>；
        //.collect()是聚合的算子，Collectors.toMap 的三个参数分别是 keyMapper，valueMapper 和 mergeFunction。分别表示聚合出来的 Map 的 key 是什么，value 是什么，如果遇到key 相同的，怎么合并值。
        Map<Integer, Integer> map = s.chars().boxed()
            .collect(Collectors.toMap(k -> k, v -> 1, Integer :: sum));
        //.values() 返回的是map 值的集合Collection<Integer>，先用.stream()转成流以后，利用mapToInt 转成 IntStream，因为 IntStream 是支持 sum 算子的，通过sum 算子进行求和。
        int res = map.values().stream().mapToInt(i -> i - (i & 1)).sum(
);
        return res < s.length() ? res + 1 : res;
        //至于为啥这里流运算比传统循环慢，是因为对于小数据集无法发挥流的优势，这里只是为了介绍这种优雅的写法
    }
}
```

6、最短回文串（214）

更多硬核内容就在公众号「yes 的练级攻略」

214. 最短回文串

难度 困难 149 收藏 分享 切换为英文 关注 反馈

给定一个字符串 **s**，你可以通过在字符串前面添加字符将其转换为回文串。找到并返回可以用这种方式转换的最短回文串。

示例 1:

```
输入："aacecaaa"
输出："aaacecaaa"
```

示例 2:

```
输入："abcd"
输出："dcbabcd"
```

方法 1：马拉车算法（Manacher 算法）

```java
class Solution {
    public String shortestPalindrome(String s) {
        String T = preProcess(s);
        int len = T.length();
        int [] P = new int[len];
        int C = 0, R = 0;
        for (int i = 1; i < len-1; i++) {
            int i_mirror = 2 * C - i;
            if (R > i) {
                P[i] = Math.min(R-i, P[i_mirror]);// 防止超出 R
            } else {
                P[i] = 0; // 等于 R 的情况
            }
            // 碰到之前讲的三种情况时候，需要利用中心扩展法
            while (T.charAt(i+1+P[i]) == T.charAt(i-1-P[i])) {
                P[i]++;
            }
            // 判断是否需要更新 R
            if (i + P[i] > R) {
                C = i;
                R = i + P[i];
            }
        }
        //这里的话需要修改
        int maxLen = 0;
        int centerIndex = 0;
        for (int i = 1; i < len -1; i++){
            int start = (i-P[i])/2;
            //我们要判断当前回文串是不是开头是不是从 0 开始的
            if (start == 0) {
                maxLen = P[i] > maxLen ? P[i] :maxLen;
```

```
            }
        }
        return new StringBuilder(s.substring(maxLen)).reverse() + s;
    }
    public String preProcess(String s) {
        if (s.length() == 0) {
            return "^$";
        }
        String res = "^";
        for (int i = 0; i < s.length(); i++) {
            res += "#" + s.charAt(i);
        }
        res += "#$";
        return res;
    }
}
```

KMP 推导：

https://learnku.com/articles/10622/introduction-of-kmp-algorithm-and-derivation-of-next-array

字符串匹配算法，即 KMP 算法。推荐两个链接，大家可以先学习一下，我就不多说了。KMP 算法代码简单，但理解求 next 数组的话，确实有些麻烦。

公众号：
yes的练级攻略

如果熟悉了 KMP 算法，下边就简单了。

再回想一下解法三，倒置字符串的思路，依次比较对应子串。

```
abbacd

原s: abbacd，长度记为 n
逆r: dcabba，长度记为 n

我们把两个字符串写在一起
abbacd dcabba

判断 abbacd 和 dcabba 是否相等
判断 abbac 和 cabba 是否相等
判断 abba 和 abba 是否相等
```

如果我们把 `abbacd dcabba` 看成一个字符串，中间加上一个分隔符 `#`，`abbacd#dcabba`。

回味一下上边的三条判断，判断 XXX 和 XXX 是否相等，按列看一下。

左半部分 `abbacd`，`abbac`，`abba` 其实就是 `abbacd#dcabba` 的一些前缀。

右半部分 `dcabba`，`cabba`，`abba` 其实就是 `abbacd#dcabba` 的一些后缀。

寻找前缀和后缀相等。

想一想 `KMP` 算法，这不就是 `next` 数组做的事情吗。

而我们中间加了分隔符，也就保证了前缀和后缀相等时，前缀一定在 `abbacd` 中。

换句话说，我们如果求出了 `abbacd#dcabba` 的 `next` 数组，因为我们构造的字符串后缀就是原字符串的倒置，前缀后缀相等时，也就意味着当前前缀是一个回文串，而 `next` 数组是寻求最长的前缀，我们也就找到了开头开始的最长回文串。

因为 `next` 数组的含义并不统一，但 `KMP` 算法本质上都是一样的，所以下边的代码仅供参考。

我的 `next` 数组 `next[i]` 所考虑的对应字符串不包含 `s[i]`。

方法 2：KMP 算法（8ms）

```
class Solution {
public String shortestPalindrome(String s) {
    String ss = s + '#' + new StringBuilder(s).reverse();
    int max = getLastNext(ss);
    return new StringBuilder(s.substring(max)).reverse() + s;
}

//返回 next 数组的最后一个值
public int getLastNext(String s) {
    int n = s.length();
    char[] c = s.toCharArray();
    int[] next = new int[n + 1];
    next[0] = -1;
    next[1] = 0;
    int k = 0;
    int i = 2;
    while (i <= n) {
        if (k == -1 || c[i - 1] == c[k]) {
```

```
                next[i] = k + 1;
                k++;
                i++;
            } else {
                k = next[k];
            }
        }
        return next[n];
}
}
```

7、回文数（9）【验证数】

9. 回文数

难度 简单 1163 收藏 分享 切换为英文 关注 反馈

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

示例 1:

输入: 121
输出: true

示例 2:

输入: -121
输出: false
解释: 从左向右读，为 -121 。 从右向左读，为 121- 。因此它不是一个回文数。

示例 3:

输入: 10
输出: false
解释: 从右向左读，为 01 。因此它不是一个回文数。

进阶:

你能不将整数转为字符串来解决这个问题吗?

方法 1：Java 中将 int 转成字符串的方法 o(n)

1.) String s = String.valueOf(x);

2.) String s = Integer.toString(x);

3.) String s = “” + x;

```
class Solution {
    public boolean isPalindrome(int x) {
        //String s = String.valueOf(x); //12ms
        String s = Integer.toString(x); //11ms
        //String s = "" + x; //16ms
        if (s.length() < 2) return true;
        int left = 0, right = s.length() - 1;
        while (left < right) {
            if (s.charAt(left) == s.charAt(right)) {
                left++;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                right--;
            }else {
                return false;
            }
        }
        return true;
    }
}
```

https://leetcode-cn.com/problems/palindrome-number/solution/hui-wen-shu-by-leetcode-solution/

方法 2：反转一半的数字 o(logn)

方法一：反转一半数字

思路

映入脑海的第一个想法是将数字转换为字符串，并检查字符串是否为回文。但是，这需要额外的非常量空间来创建问题描述中所不允许的字符串。

第二个想法是将数字本身反转，然后将反转后的数字与原始数字进行比较，如果它们是相同的，那么这个数字就是回文。但是，如果反转后的数字大于 int.MAX，我们将遇到整数溢出问题。

按照第二个想法，为了避免数字反转可能导致的溢出问题，为什么不考虑只反转 int 数字的一半？毕竟，如果该数字是回文，其后半部分反转后应该与原始数字的前半部分相同。

例如，输入 `1221`，我们可以将数字 “1221” 的后半部分从 “21” 反转为 “12”，并将其与前半部分 “12” 进行比较，因为二者相同，我们得知数字 `1221` 是回文。

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

算法

首先，我们应该处理一些临界情况。所有负数都不可能是回文，例如：`-123` 不是回文，因为 `-` 不等于 `3`。所以我们可以对所有负数返回 `false`。除了 `0` 以外，所有个位是 `0` 的数字不可能是回文，因为最高位不等于 `0`。所以我们可以对所有大于 `0` 且个位是 `0` 的数字返回 `false`。

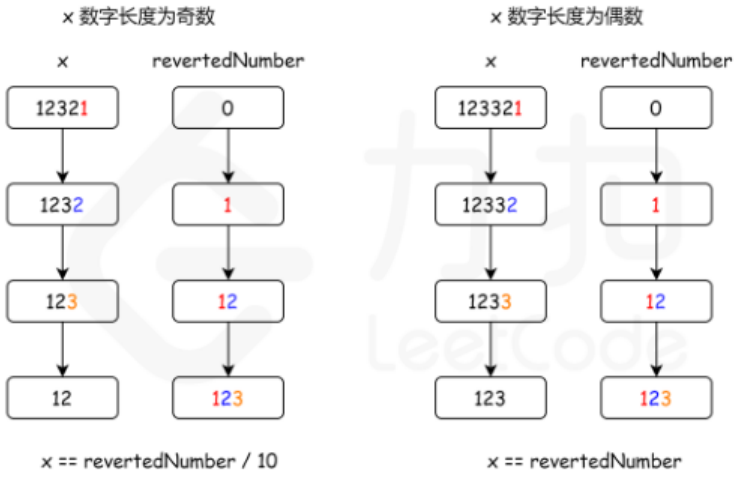
现在，让我们来考虑如何反转后半部分的数字。

对于数字 `1221`，如果执行 `1221 % 10`，我们将得到最后一位数字 `1`，要得到倒数第二位数字，我们可以先通过除以 `10` 把最后一位数字从 `1221` 中移除，`1221 / 10 = 122`，再求出上一步结果除以 `10` 的余数，`122 % 10 = 2`，就可以得到倒数第二位数字。如果我们把最后一位数字乘以 `10`，再加上倒数第二位数字，`1 * 10 + 2 = 12`，就得到了我们想要的反转后的数字。如果继续这个过程，我们将得到更多位数的反转数字。

现在的问题是，我们如何知道反转数字的位数已经达到原始数字位数的一半？

由于整个过程我们不断将原始数字除以 `10`，然后给反转后的数字乘上 `10`，所以，当原始数字小于或等于反转后的数字时，就意味着我们已经处理了一半位数的数字了。

☆ 收藏



复杂度分析

- 时间复杂度：$O(\log n)$，对于每次迭代，我们会将输入除以 10，因此时间复杂度为 $O(\log n)$。
- 空间复杂度：$O(1)$。我们只需要常数空间存放若干变量。

```
class Solution {
    public boolean isPalindrome(int x) {
        // 特殊情况：
        // 如上所述，当 x < 0 时，x 不是回文数。
        // 同样地，如果数字的最后一位是 0，为了使该数字为回文，
        // 则其第一位数字也应该是 0
        // 只有 0 满足这一属性
        if (x < 0 || (x % 10 == 0 && x != 0)) {
            return false;
        }

        int rev = 0;
        while (x > rev) {
            rev = rev * 10 + x % 10;
            x /= 10;
```

```
            }

            // 当数字长度为奇数时，我们可以通过 rev/10 去除处于中位的数字。
            // 例如，当输入为 12321 时，在 while 循环的末尾我们可以得
到 x = rev = 123，
            // 由于处于中位的数字不影响回文（它总是与自己相等），所以我们可以简单地
将其去除。
            return x == rev || x == rev / 10;
        }
}
```

8、验证回文串（125）【验证字符串】

125. 验证回文串

难度 简单 238 收藏 分享 切换为英文 关注 反馈

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

示例 1:

```
输入: "A man, a plan, a canal: Panama"
输出: true
```

示例 2:

```
输入: "race a car"
输出: false
```

字符串换成小写的方法：Character.toLowerCase()；判断是否字母或数字 Character.isLetterOrDigit()，是的话返回 true

https://leetcode-cn.com/problems/valid-palindrome/solution/javade-4chong-jie-ti-fang-shi-by-sdwwld/

```
class Solution {
    public boolean isPalindrome(String s) {
        if (s == null || s.length() == 0)
            return true;
        s = s.toLowerCase();
        for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
            while (i < j && !Character.isLetterOrDigit(s.charAt(i)))
                i++;
            while (i < j && !Character.isLetterOrDigit(s.charAt(j)))
                j--;
            if (s.charAt(i) != s.charAt(j))
                return false;
        }
        return true;
    }
}
```

9、验证回文字符串 2（680）

更多硬核内容就在公众号「yes 的练级攻略」

680. 验证回文字符串 II

难度 简单 261 收藏 分享 切换为英文 关注 反馈

给定一个非空字符串 `s`，**最多**删除一个字符。判断是否能成为回文字符串。

示例 1:

```
输入: "aba"
输出: True
```

示例 2:

```
输入: "abca"
输出: True
解释: 你可以删除c字符。
```

注意:

1. 字符串只包含从 a-z 的小写字母。字符串的最大长度是50000。

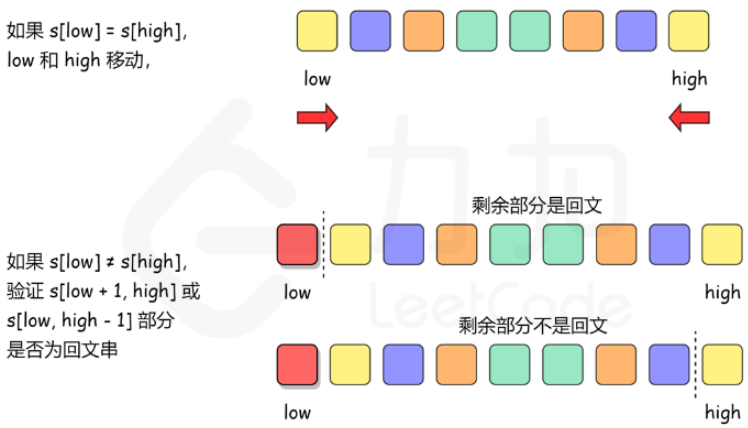
https://leetcode-cn.com/problems/valid-palindrome-ii/solution/yan-zheng-hui-wen-zi-fu-chuan-ii-by-leetcode-solut/

方法一：贪心算法

考虑最朴素的方法：首先判断原串是否是回文串，如果是，就返回 `true`；如果不是，则枚举每一个位置作为被删除的位置，再判断剩下的字符串是否是回文串。这种做法的渐进时间复杂度是 $O(n^2)$ 的，会超出时间限制。

我们换一种想法。首先考虑如果不允许删除字符，如何判断一个字符串是否是回文串。常见的做法是使用双指针。定义左右指针，初始时分别指向字符串的第一个字符和最后一个字符，每次判断左右指针指向的字符是否相同，如果不相同，则不是回文串；如果相同，则将左右指针都往中间移动一位，直到左右指针相遇，则字符串是回文串。

在允许最多删除一个字符的情况下，同样可以使用双指针，通过贪心算法实现。初始化两个指针 `low` 和 `high` 分别指向字符串的第一个字符和最后一个字符。每次判断两个指针指向的字符是否相同，如果相同，则更新指针，令 `low = low + 1` 和 `high = high - 1`，然后判断更新后的指针范围内的子串是否是回文字符串。如果两个指针指向的字符不同，则两个字符中必须有一个被删除，此时我们就分成两种情况：即删除左指针对应的字符，留下子串 `s[low + 1], s[low + 1], ..., s[high]`，或者删除右指针对应的字符，留下子串 `s[low], s[low + 1], ..., s[high - 1]`。当这两个子串中至少有一个是回文串时，就说明原始字符串删除一个字符之后就以成为回文串。



复杂度分析

- 时间复杂度：$O(n)$，其中 n 是字符串的长度。判断整个字符串是否是回文字符串的时间复杂度是 $O(n)$，遇到不同字符时，判断两个子串是否是回文字符串的时间复杂度也都是 $O(n)$。
- 空间复杂度：$O(1)$。只需要维护有限的常量空间。

```java
class Solution {
    public boolean validPalindrome(String s) {
        int low = 0, high = s.length() - 1;
        while (low < high) {
            char c1 = s.charAt(low), c2 = s.charAt(high);
            if (c1 == c2) {
                low++;
                high--;
            } else {
                boolean flag1 = true, flag2 = true;
                for (int i = low, j = high - 1; i < j; i++, j--) {
                    char c3 = s.charAt(i), c4 = s.charAt(j);
                    if (c3 != c4) {
                        flag1 = false;
                        break;
                    }
                }
                for (int i = low + 1, j = high; i < j; i++, j--) {
                    char c3 = s.charAt(i), c4 = s.charAt(j);
                    if (c3 != c4) {
                        flag2 = false;
                        break;
                    }
                }
                return flag1 || flag2;
            }
        }
        return true;
    }
}
```

10、验证回文字符串 3（1216）

更多硬核内容就在公众号「yes 的练级攻略」

1216. 验证回文字符串 III

难度 困难 21 收藏 分享 切换为英文 关注 反馈

给出一个字符串 `s` 和一个整数 `k`，请你帮忙判断这个字符串是不是一个「K 回文」。

所谓「K 回文」：如果可以通过从字符串中删去最多 `k` 个字符将其转换为回文，那么这个字符串就是一个「K 回文」。

示例:

```
输入: s = "abcdeca", k = 2
输出: true
解释: 删除字符 “b” 和 “e”。
```

提示:

- `1 <= s.length <= 1000`
- `s` 中只含有小写英文字母
- `1 <= k <= s.length`

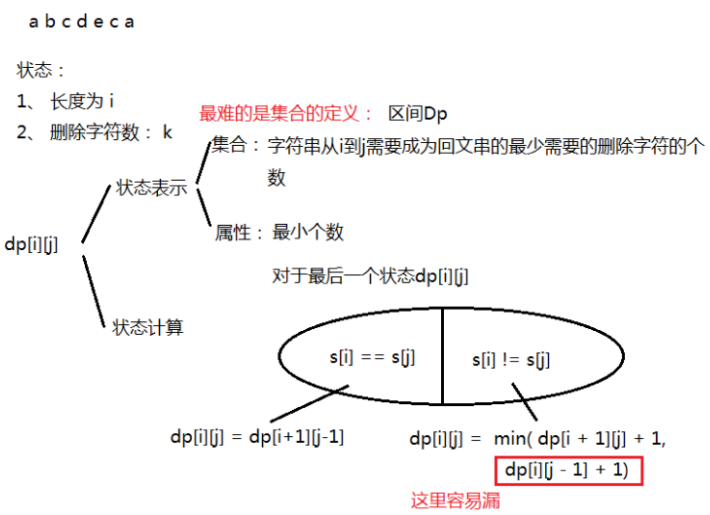
禁止转载

https://leetcode-cn.com/problems/valid-palindrome-iii/solution/qu-jian-dp-java-by-acw_weian/

题目：
1216. 验证回文字符串 III
给出一个字符串 s 和一个整数 k，请你帮忙判断这个字符串是不是一个「K 回文」。
所谓「K 回文」：如果可以通过从字符串中删去最多 k 个字符将其转换为回文，那么这个字符串就是一个「K 回文」。

示例：
输入：s = "abcdeca", k = 2
输出：true
解释：删除字符 “b” 和 “e”。

提示：
1 <= s.length <= 1000
s 中只含有小写英文字母
1 <= k <= s.length




```
class Solution {
```

```
    public boolean isValidPalindrome(String s, int k){
        int n = s.length(), INF = 0x3f3f3f3f;
        int[][] dp = new int[n][n];
        for(int i = 0; i < n; i++)
            Arrays.fill(dp[i], INF);
        char[] cs = s.toCharArray();
        for(int i = 0; i < n; i++) dp[i][i] = 0;
        for(int len = 2; len <= n; len++){
            for(int i = 0; i + len - 1 < n; i++){
                int j = i + len - 1;
                if(cs[i] == cs[j]) {
                    if(len == 2)   dp[i][j] = 0;
                    else dp[i][j] = dp[i + 1][j - 1];
                }else{
                    dp[i][j] = Math.min(dp[i + 1][j] + 1, dp[i][j - 1] + 1);
                }
            }
        }
        return dp[0][n - 1] <= k;
    }
}
```

11、回文对（336）

336. 回文对

难度 困难 69 收藏 分享 切换为英文 关注 反馈

给定一组唯一的单词， 找出所有不同的索引对 (i, j) ，使得列表中的两个单词， words[i] + words[j] ，可拼接成回文串。

示例 1:

```
输入: ["abcd","dcba","lls","s","sssll"]
输出: [[0,1],[1,0],[3,2],[2,4]]
解释: 可拼接成的回文串为 ["dcbaabcd","abcddcba","slls","llssssll"]
```

示例 2:

```
输入: ["bat","tab","cat"]
输出: [[0,1],[1,0]]
解释: 可拼接成的回文串为 ["battab","tabbat"]
```

https://leetcode-cn.com/problems/palindrome-pairs/solution/hui-wen-dui-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：枚举前缀和后缀

思路及算法

假设存在两个字符串 s_1 和 s_2，$s_1 + s_2$ 是一个回文串，记这两个字符串的长度分别为 len_1 和 len_2，我们分三种情况进行讨论：

1. $len_1 = len_2$，这种情况下 s_1 是 s_2 的翻转。
2. $len_1 > len_2$，这种情况下我们可以将 s_1 拆成左右两部分：t_1 和 t_2，其中 t_1 是 s_2 的翻转，t_2 是一个回文串。
3. $len_1 < len_2$，这种情况下我们可以将 s_2 拆成左右两部分：t_1 和 t_2，其中 t_2 是 s_1 的翻转，t_1 是一个回文串。

这样，对于每一个字符串，我们令其为 s_1 和 s_2 中较长的那一个，然后找到可能和它构成回文串的字符串即可。

具体地说，我们枚举每一个字符串 k，令其为 s_1 和 s_2 中较长的那一个，那么 k 可以被拆分为两部分，t_1 和 t_2。

1. 当 t_1 是回文串时，符合情况 3，我们只需要查询给定的字符串序列中是否包含 t_2 的翻转。
2. 当 t_2 是回文串时，符合情况 2，我们只需要查询给定的字符串序列中是否包含 t_1 的翻转。

也就是说，我们要枚举字符串 k 的每一个前缀和后缀，判断其是否为回文串。如果是回文串，我们就查询其剩余部分的翻转是否在给定的字符串序列中出现即可。

注意到空串也是回文串，所以我们可以将 k 拆解为 $k + \varnothing$ 或 $\varnothing + k$，这样我们就能将情况 1 也解释为特殊的情况 2 或情况 3。

而要实现这些操作，我们只需要设计一个能够在一系列字符串中查询「某个字符串的子串的翻转」是否存在的数据结构，有两种实现方法：

- 我们可以使用字典树存储所有的字符串。在进行查询时，我们将待查询串的子串逆序地在字典树上进行遍历，即可判断其是否存在。
- 我们可以使用哈希表存储所有字符串的翻转串。在进行查询时，我们判断带查询串的子串是否在哈希表中出现，就等价于判断了其翻转是否存在。

复杂度分析

- 时间复杂度：$O(n \times m^2)$，其中 n 是字符串的数量，m 是字符串的平均长度。对于每一个字符串，我们需要 $O(m^2)$ 地判断其所有前缀与后缀是否是回文串，并 $O(m^2)$ 地寻找其所有前缀与后缀是否在给定的字符串序列中出现。
- 空间复杂度：$O(n \times m)$，其中 n 是字符串的数量，m 是字符串的平均长度。为字典树的空间开销。

方法 1：HashMap

```
class Solution {
    List<String> wordsRev = new ArrayList<String>();
    Map<String, Integer> indices = new HashMap<String, Integer>();

    public List<List<Integer>> palindromePairs(String[] words) {
        int n = words.length;
        for (String word: words) {
            wordsRev.add(new StringBuffer(word).reverse().toString());
        }
        for (int i = 0; i < n; ++i) {
            indices.put(wordsRev.get(i), i);
        }

        List<List<Integer>> ret = new ArrayList<List<Integer>>();
        for (int i = 0; i < n; i++) {
            String word = words[i];
            int m = words[i].length();
```

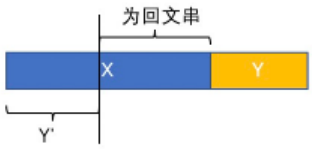
更多硬核内容就在公众号「yes 的练级攻略」

```
            if (m == 0) {
                continue;
            }
            for (int j = 0; j <= m; j++) {
                if (isPalindrome(word, j, m - 1)) {
                    int leftId = findWord(word, 0, j - 1);
                    if (leftId != -1 && leftId != i) {
                        ret.add(Arrays.asList(i, leftId));
                    }
                }
                if (j != 0 && isPalindrome(word, 0, j - 1)) {
                    int rightId = findWord(word, j, m - 1);
                    if (rightId != -1 && rightId != i) {
                        ret.add(Arrays.asList(rightId, i));
                    }
                }
            }
        }
        return ret;
    }

    public boolean isPalindrome(String s, int left, int right) {
        int len = right - left + 1;
        for (int i = 0; i < len / 2; i++) {
            if (s.charAt(left + i) != s.charAt(right - i)) {
                return false;
            }
        }
        return true;
    }

    public int findWord(String s, int left, int right) {
        return indices.getOrDefault(s.substring(left, right + 1), -1);
    }
}
```

方法二：字典树（Trie树）[通过]

分析：

- 根据回文字符串的性质，我们可以不用暴力枚举出所有字符串对
- 对于一个字符串对(x, y), 若想要字符串$x + y$是一个回文字符串，则必须满足以下条件之一
- 1. 当$x.length() \geq y.length()$时, 字符串$x$的$y.length()$长度的前缀与$y$的**逆序**相等，且字符串$x$去除长度为$y.length()$的前缀后，余下的部分也是一个回文字符串。
- 2. 当$x.length() < y.length()$时,与情况一正相反。



Y' 为Y的逆序字符串

- 对于字符串x时，我们依次遍历其每一个字母，假设当前遍历到的位置为i,若$[i, x.length()]$是一个回文对，且$[0, i]$的逆序存在于$word$列表中（设其下标为y），则(x, y)可以构成回文对，加入结果数组中。
- 当我们遍历完字符串x的每一个字符，我们还**需要考虑分析中第2种情况，即**$x.length() < y.length()$
- 为了加快查找速率，我们利用字典树辅助查找，其中重要的是字典树每一个 `Node` 里所保存的信息。
- 1. `List<Integer> words` 存储以当前节点为终止节点的所有单词（逆序之后的),来获取所有第1种情况下的匹配成功的字符串y
- 2. `List<Integer> suffixs` 保存所有到当前节点为止，其之后剩余字符可以构成回文对的单词。用来获取所有长度大于$x.length()$，且可以和其构成回文对的单词y（即针对第二种情况）。

代码实现：

- 用每个单词的逆序构建字典树
- 在每个节点上维护以该节点为终止节点的单词
- 在若在该节点之和的部分能构成回文对，则加入suffixs列表中。
- 详情见注释

方法 2：字典树

```
class Node {
    public Node[] children;
    public List<Integer> words;
    public List<Integer> suffixs;
    public Node() {
        this.children = new Node[26];
        this.words = new ArrayList<>();
        this.suffixs = new ArrayList<>();
    }
}
public class Solution {
    private Node root;
    public List<List<Integer>> palindromePairs(String[] words) {
        this.root = new Node();
        int n = words.length;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        // 字典树的插入，注意维护每个节点上的两个列表
        for (int i = 0; i < n; i++) {
            String rev = new StringBuilder(words[i]).reverse().toString
();
            Node cur = root;
            if (isPalindrome(rev.substring(0))) cur.suffixs.add(i);
            for (int j = 0; j < rev.length(); j++) {
                char ch = rev.charAt(j);
                if (cur.children[ch-'a']==null) cur.children[ch-'a'] =
new Node();
                cur = cur.children[ch-'a'];
                if (isPalindrome(rev.substring(j+1))) cur.suffixs.add(i
);
            }
            cur.words.add(i);
        }
        // 用以存放答案的列表
        List<List<Integer>> ans = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            String word = words[i];
            Node cur = root;
            int j = 0;
            for ( ;j < word.length(); j++) {
                // 到j 位置，后续字符串若是回文对，则在该节点位置上所有单词都
可以与words[i]构成回文对
                // 因为我们插入的时候是用每个单词的逆序插入的:)
                if(isPalindrome(word.substring(j)))
                    for (int k : cur.words)
                        if (k != i) ans.add(Arrays.asList(i,k));

                char ch = word.charAt(j);
                if (cur.children[ch-'a'] == null) break;
                cur = cur.children[ch-'a'];

            }
            // words[i]遍历完了，现在找所有大于words[i]长度且符合要求的单词，
suffixs 列表就派上用场了:)
            if (j == word.length())
                for (int k : cur.suffixs)
                    if (k != i) ans.add(Arrays.asList(i,k));

        }
        return ans;
```

```
    }
    //  判断一个字符串是否是回文字符串
    private boolean isPalindrome(String w) {
        int i = 0, j = w.length()-1;
        while (i < j) {
            if (w.charAt(i) != w.charAt(j)) return false;
            i++; j--;
        }
        return true;
    }
}
```

12、删除回文子数组（1246）【区间 DP】

1246. 删除回文子数组

难度 困难 34 收藏 分享 切换为英文 关注 反馈

给你一个整数数组 `arr`，每一次操作你都可以选择并删除它的一个 **回文** 子数组 `arr[i], arr[i+1], ..., arr[j]` （`i <= j`）。

注意，每当你删除掉一个子数组，右侧元素都会自行向前移动填补空位。

请你计算并返回从数组中删除所有数字所需的最少操作次数。

示例 1：

```
输入：arr = [1,2]
输出：2
```

示例 2：

```
输入：arr = [1,3,4,1,5]
输出：3
解释：先删除 [4]，然后删除 [1,3,1]，最后再删除 [5]。
```

提示：

- `1 <= arr.length <= 100`
- `1 <= arr[i] <= 20`

https://leetcode-cn.com/problems/palindrome-removal/solution/java-liang-chong-jie-fa-dp-and-backtrack-by-npe_tl/

思路

经典的区间dp，我们令$f(i,j)$代表删除区间$[i,j]$的最小值（即最小删除次数），那么可得以下递推公式：

$$f(i,j)=\begin{cases} min(f(i+1,j-1),f(i,k)+f(k+1,j)) & if\ a[i]=a[j] \\ min(f(i,j),f(i,k)+f(k+1,j)) & if\ a[i]\neq a[j] \end{cases} i\le k<j$$

分析一下，这里根据a[i]是否等于a[j]，分两种情况。因为当a[i] = a[j]的时候，有可能产生最小删除次数的方案是a[i]和a[j]等到最后一起被删。这里提供两种版本的解法，一种是动态规划、一种是回溯+记忆化。

```
class Solution {
 public int minimumMoves(int[] arr) {
        int len = arr.length;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        int[][] dp = new int[len][len];
        // 单个字符也是回文串，删除单个字符的最小删除次数就是 1
        for (int i = 0; i < len; i++) {
            dp[i][i] = 1;
        }
        for (int j = 1; j < len; j++) {
            for (int i = j-1; i >= 0; i--) {
                if (i == j - 1) {
                    // 就两个元素
                    dp[i][j] = arr[i] == arr[j] ? 1 : 2;
                    continue;
                }
                // 下面至少三个元素
                int min = Integer.MAX_VALUE;
                if (arr[i] == arr[j]) {
                    // 头尾相等，最小值有可能是出现在这对头尾最后被删的结果
                    min = dp[i+1][j-1];
                }

                for (int k = i; k < j; k++) {
                    min = Math.min(min, dp[i][k] + dp[k + 1][j]);
                }
                dp[i][j] = min;
            }
        }
        return dp[0][len-1];
    }
}
```

13、分割回文串（131）

131. 分割回文串

难度 中等 352 收藏 分享 切换为英文

给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。

返回 s 所有可能的分割方案。

示例:

输入: "aab"
输出:

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

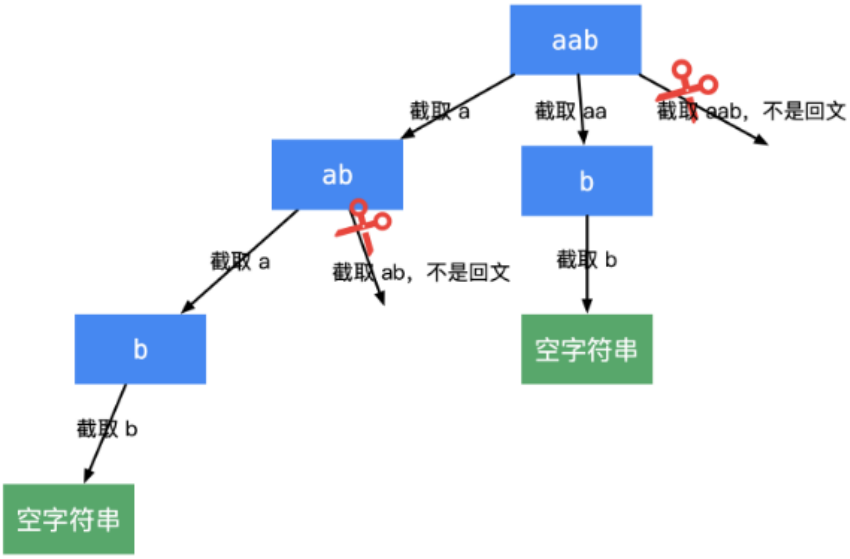
https://leetcode-cn.com/problems/palindrome-partitioning/solution/hui-su-you-hua-jia-liao-dong-tai-gui-hua-by-liweiw/

搜索问题主要使用回溯法。

回溯法思考的步骤：

1、画递归树；

2、根据自己画的递归树编码。



思考如何根据这棵递归树编码：

1、每一个结点表示剩余没有扫描到的字符串，产生分支是截取了剩余字符串的前缀；

2、产生前缀字符串的时候，判断前缀字符串是否是回文。

- 如果前缀字符串是回文，则可以产生分支和结点；
- 如果前缀字符串不是回文，则不产生分支和结点，这一步是剪枝操作。

3、在叶子结点是空字符串的时候结算，此时**从根结点到叶子结点的路径，就是结果集里的一个结果，使用深度优先遍历，记录下所有可能的结果。**

- 采用一个路径变量 `path` 搜索， `path` 全局使用一个（注意结算的时候，需要生成一个拷贝），因此在递归执行方法结束以后需要回溯，即将递归之前添加进来的元素拿出去；
- `path` 的操作只在列表的末端，因此合适的数据结构是栈。

方法二：回溯的优化（加了动态规划）

在上一步，验证回文串那里，每一次都得使用“两边夹”的方式验证子串是否是回文子串。于是“用空间换时间”，利用「力扣」第 5 题：最长回文子串 的思路，利用动态规划把结果先算出来，这样就可以以 $O(1)$ 的时间复杂度直接得到一个子串是否是回文。

```
class Solution {
    public List<List<String>> partition(String s) {
        int n = s.length();
        List<List<String>> res = new ArrayList<>();
```

```
        if (n == 0) return res;
        // 预处理
        // 状态：dp[i][j] 表示 s[i][j] 是否是回文
        boolean[][] dp = new boolean[n][n];
        // 状态转移方程：在 s[i] == s[j] 的时候，dp[i][j] 参考 dp[i + 1][j - 1]
        for (int j = 0; j < n; j++) {
            // 注意：i <= j 取等号表示 1 个字符的时候也需要判断
            for (int i = 0; i <= j; i++) {
                if (s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i + 1][j -
1])) {
                    dp[i][j] = true;
                }
            }
        }
         Deque<String> sta = new ArrayDeque<>();
         dfs(s, 0, n, dp, sta, res);
         return res;
    }
    public void dfs(String s, int start, int length, boolean[][] dp, Deque<Stri
ng> sta, List<List<String>> res) {
        if (start == length) {
            res.add(new ArrayList<>(sta));
            return;
        }
        for (int i = start; i < length; i++) {
            if (!dp[start][i]) continue;
            sta.addLast(s.substring(start, i + 1));
            dfs(s, i + 1, length, dp, sta, res);
            sta.removeLast();
        }
    }
}
```

14、分割回文串 2（132）

132. 分割回文串 II

难度 困难 186 收藏 分享 切换为英文 关注 反馈

给定一个字符串 *s*，将 *s* 分割成一些子串，使每个子串都是回文串。

返回符合要求的最少分割次数。

示例:

```
输入: "aab"
输出: 1
解释: 进行一次分割就可将 s 分割成 ["aa","b"] 这样两个回文子串。
```

https://leetcode-cn.com/problems/palindrome-partitioning-ii/solution/dong-tai-gui-hua-by-liweiwei1419-2/

方法一：动态规划

步骤 1：思考状态

状态就尝试定义成题目问的那样，看看状态转移方程是否容易得到。

`dp[i]`：表示前缀子串 `s[0:i]` 分割成若干个回文子串所需要最小分割次数。

步骤 2：思考状态转移方程

思考的方向是：大问题的最优解怎么由小问题的最优解得到。

即 `dp[i]` 如何与 `dp[i - 1]`、`dp[i - 2]`、…、`dp[0]` 建立联系。

比较容易想到的是：如果 `s[0:i]` 本身就是一个回文串，那么不用分割，即 `dp[i] = 0`，这是首先可以判断的，否则就需要去遍历；

接下来枚举可能分割的位置：即如果 `s[0:i]` 本身不是一个回文串，就尝试分割，枚举分割的边界 `j`。

如果 `s[j + 1, i]` 不是回文串，尝试下一个分割边界。

如果 `s[j + 1, i]` 是回文串，则 `dp[i]` 就是在 `dp[j]` 的基础上多一个分割。

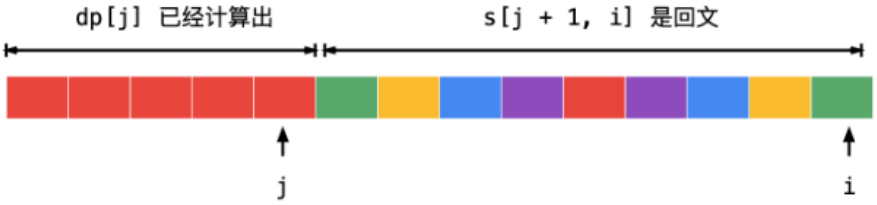
于是枚举 `j` 所有可能的位置，取所有 `dp[j]` 中最小的再加 1，就是 `dp[i]`。

得到状态转移方程如下：

```
dp[i] = min([dp[j] + 1 for j in range(i) if s[j + 1, i] 是回文])
```

状态转移方程：

```
dp[i] = min([dp[j] + 1 for j in range(i) if s[j + 1, i] 是回文])
```



步骤 3：思考初始状态

初始状态：单个字符一定是回文串，因此 `dp[0] = 0`。

步骤 4：思考输出

状态转移方程可以得到，并且状态就是题目问的，因此返回最后一个状态即可，即 `dp[len - 1]`。

步骤 5：思考是否可以优化空间

每一个状态值都与之前的状态值有关，因此不能优化空间。

参考代码 1： （Python 代码会超时）

方法二：动态规划（优化）

上面判断回文串的时候方法 `checkPalindrome()` 是线性的，时间复杂度为 $O(N)$。我们可以借助「力扣」第 5 题：最长回文子串 的做法，依然是使用动态规划的做法，得到一个预处理的动态规划数组，这样就可以通过 $O(1)$ 的时间复杂度，得到一个子串是否是回文的结果了。

```java
public class Solution {
    public int minCut(String s) {
        int len = s.length();
        // 特判
        if (len < 2) return 0;
        // 状态定义：dp[i]：前缀子串 s[0:i] （包括索引 i 处的字符）符合要求的最少分割次数
        // 状态转移方程：
        // dp[i] = min(dp[j] + 1 if s[j + 1: i] 是回文 for j in range(i))
        int[] dp = new int[len];
        // 2 个字符最多分割 1 次；
        // 3 个字符最多分割 2 次
        // 初始化的时候，设置成为这个最多分割次数
        for (int i = 0; i < len; i++) dp[i] = i;
        // 参考「力扣」第 5 题：最长回文子串 动态规划 的解法
        boolean[][] checkPalindrome = new boolean[len][len];
        for (int right = 0; right < len; right++) {
            // 注意：left <= right 取等号表示 1 个字符的时候也需要判断
            for (int left = 0; left <= right; left++) {
```

```
                if (s.charAt(left) == s.charAt(right) && (right - left <= 2 ||
checkPalindrome[left + 1][right - 1])) {
                    checkPalindrome[left][right] = true;
                }
            }
        }
        // 1 个字符的时候，不用判断，因此 i 从 1 开始
        for (int i = 1; i < len; i++) {
            if (checkPalindrome[0][i]){
                dp[i] = 0;
                continue;
            }
            // 注意：这里是严格，要保证 s[j + 1:i] 至少得有一个字符串
            for (int j = 0; j < i; j++) {
                if (checkPalindrome[j + 1][i]) {
                    dp[i] = Math.min(dp[i], dp[j] + 1);
                }
            }
        }
        return dp[len - 1];
    }
}
```

15、分割回文串 3（1278）

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

1278. 分割回文串 III

难度 困难 50 收藏 分享 切换为英文 关注 反馈

给你一个由小写字母组成的字符串 `s`，和一个整数 `k`。

请你按下面的要求分割字符串：

- 首先，你可以将 `s` 中的部分字符修改为其他的小写英文字母。
- 接着，你需要把 `s` 分割成 `k` 个非空且不相交的子串，并且每个子串都是回文串。

请返回以这种方式分割字符串所需修改的最少字符数。

示例 1：

```
输入：s = "abc", k = 2
输出：1
解释：你可以把字符串分割成 "ab" 和 "c"，并修改 "ab" 中的 1 个字符，将它变成回文串。
```

示例 2：

```
输入：s = "aabbc", k = 3
输出：0
解释：你可以把字符串分割成 "aa"、"bb" 和 "c"，它们都是回文串。
```

示例 3：

```
输入：s = "leetcode", k = 8
输出：0
```

提示：

- `1 <= k <= s.length <= 100`
- `s` 中只含有小写英文字母。

YES的练级攻略

https://leetcode-cn.com/problems/palindrome-partitioning-iii/solution/javadp-by-z446979478/

1.首先第一部分算出s的每一段substring 变成回文需要修改多少字符串
1.1这部分的状态转移方程是
pali[i][j] = pali[i+1][j-1]+1(如果i和j位置的字符串不等)
pali[i][j] = pali[i+1][j-1] (如果i和j位置的字符串相等)
注意 如果j-i<=1，即i到j之间有小于等于2个字符串的时候
pali[i][j] = 1(如果i和j位置的字符串不等)
pali[i][j] = 0(如果i和j位置的字符串相等)

2.计算分割成k个字符串后让每个字符串变成回文字符串的最少修改次数
用dp[i][j]表示，i表示切割成i个 j表示当前字符串的长度
状态转移方程是
dp[i][j] = min(dp[i-1][x] + pali[x+1][j])
其中x的取值范围是[i-1, j-1] (i-1表示前i-1个字符串，每一个字母一段，j-1表示最后一个字母一段)

```
class Solution {
    public int palindromePartition(String s, int k) {
        int[][] pali= new int[s.length() + 1][s.length() + 1];
        for(int i = s.length(); i >= 1; i--)
        {
            for(int j = i; j <= s.length(); j++)
            {
```

```
                if(j - i >= 2)
                    pali[i][j] = pali[i + 1][j - 1];
                if(s.charAt(i - 1) != s.charAt(j - 1))
                    pali[i][j]++;
            }
        }

        int[][] dp = new int[k + 1][s.length() + 1];
        for(int i = 1; i <= k; i++)
        {
            for(int j = i; j <= s.length(); j++)
            {
                if(i == 1)
                    dp[i][j] = pali[i][j];
                else
                {
                    dp[i][j] = dp[i - 1][i - 1] + pali[i][j];
                    for(int x = i; x < j; x++)
                        dp[i][j] = Math.min(dp[i][j], dp[i - 1][x] + pali[x + 1
][j]);
                }
            }
        }
        return dp[k][s.length()];
    }
}
```

16、回文排列（266）（面试题 01.04）

266. 回文排列

难度 简单 23 收藏 分享 切换为英文 关注 反馈

给定一个字符串，判断该字符串中是否可以通过重新排列组合，形成一个回文字符串。

示例 1：

```
输入: "code"
输出: false
```

示例 2：

```
输入: "aab"
输出: true
```

示例 3：

```
输入: "carerac"
输出: true
```

面试题 01.04. 回文排列

难度 简单 32 收藏 分享 切换为英文 关注 反馈

给定一个字符串，编写一个函数判定其是否为某个回文串的排列之一。

回文串是指正反两个方向都一样的单词或短语。排列是指字母的重新排列。

回文串不一定是字典当中的单词。

示例1：

输入： "tactcoa"
输出： true（排列有"tacocat"、"atcocta"，等等）

https://leetcode-cn.com/problems/palindrome-permutation-lcci/solution/shuang-bai-wei-tu-fang-fa-by-1ujin/

判断出现奇数次的字符最多只有一个（该字符将会出现在中点位置）。全体字符有128个，可以用高低两个 long 类型的位图表示128位。通过将 1L 左移，例如 'A' 的 ASCII 值为65，就向左移动65位，所以位图的从第0位向左数第65位表示 'A'，以此类推。0 ~ 63 放在低位，64 ~ 127 放在高位。如果是0与1异或是1：`0 ^ 1 = 1`，1再与1异或又变回0：`1 ^ 1 = 0`，所以0代表该位上得字符出现偶数次。最后利用`Long.bitCount()`统计一下1的数量，是否小于等于1，即是否最多只有1个字符出现奇数次。

```
class Solution {
    public boolean canPermutePalindrome(String s) {
        long highBmp = 0, lowBmp = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) >= 64) {
                highBmp ^= 1L << s.charAt(i) - 64;
            } else {
                lowBmp ^= 1L << s.charAt(i);
            }
        }
        return Long.bitCount(highBmp) + Long.bitCount(lowBmp) <= 1;
    }
}
```

17、回文排列 2（267）

更多硬核内容就在公众号「yes 的练级攻略」

267. 回文排列 II

难度 中等 34 收藏 分享 切换为英文 关注 反馈

给定一个字符串 `s` ，返回其通过重新排列组合后所有可能的回文字符串，并去除重复的组合。

如不能形成任何回文排列时，则返回一个空列表。

示例 1：

```
输入: "aabb"
输出: ["abba", "baab"]
```

示例 2：

```
输入: "abc"
输出: []
```

https://leetcode-cn.com/problems/palindrome-permutation-ii/solution/hui-wen-pai-lie-ii-by-leetcode/

方法二：回溯 [通过]

有可能字符串本身就没法构造出一个回文排列，因此需要首先判断该字符串是否存在回文排列，具体的方法可以参考回文排列这题中的题解部分。

当我们确定了字符串可以构造出回文排列之后，接下来仍然需要找出所有的回文排列。我们可以考虑直接枚举回文排列，即枚举一个排列的前半部分，将它的前半部分翻转即可得到一个回文排列。前半部分包含的每个字符的具体数量都是固定的，因此我们可以用与方法一中完全相同的方法，来枚举前半部分的所有排列。特别地，当字符串的长度为奇数时，那个唯一出现奇数次的字符需要被放在回文串的中间位置。

方法一中使用集合来进行去重操作，我们可以考虑在生成排列时添加约束，直接避免这些重复的回文串被生成。一个简单的做法是，当我们交换 `current_index` 和某个不同位置的两个字符时，如果这两个字符相同，就不进行交换操作（因为这与 `current_index` 和自己交换是等价的）。下面的幻灯片给出了一个具体的例子。

复杂度分析

- 时间复杂度：$O(|S| * (\frac{|S|}{2})!)$。在最坏情况下，会生成 $(\frac{S}{2})!$ 个排列。在生成每个排列时，翻转操作 `string.reverse()` 的时间复杂度为 $O(|S|)$，因此总的时间复杂度为 $O(|S| * (\frac{|S|}{2})!)$。
- 空间复杂度：$O(|S|)$。递归的层数最多为 $\frac{|S|}{2}$。

```
public class Solution {
    Set < String > set = new HashSet < > ();
    public List < String > generatePalindromes(String s) {
        int[] map = new int[128];
        char[] st = new char[s.length() / 2];
        if (!canPermutePalindrome(s, map))
            return new ArrayList < > ();
        char ch = 0;
        int k = 0;
        for (int i = 0; i < map.length; i++) {
            if (map[i] % 2 == 1)
                ch = (char) i;
            for (int j = 0; j < map[i] / 2; j++) {
                st[k++] = (char) i;
            }
        }
```

```
        permute(st, 0, ch);
        return new ArrayList < String > (set);
    }
    public boolean canPermutePalindrome(String s, int[] map) {
        int count = 0;
        for (int i = 0; i < s.length(); i++) {
            map[s.charAt(i)]++;
            if (map[s.charAt(i)] % 2 == 0)
                count--;
            else
                count++;
        }
        return count <= 1;
    }
    public void swap(char[] s, int i, int j) {
        char temp = s[i];
        s[i] = s[j];
        s[j] = temp;
    }
    void permute(char[] s, int l, char ch) {
        if (l == s.length) {
            set.add(new String(s) + (ch == 0 ? "" : ch) + new StringBuffer(new
String(s)).reverse());
        } else {
            for (int i = l; i < s.length; i++) {
                if (s[l] != s[i] || l == i) {
                    swap(s, l, i);
                    permute(s, l + 1, ch);
                    swap(s, l, i);
                }
            }
        }
    }
}
```

十五、链表

ArrayList 和 LinkedList 中常用的方法

ArrayList : List 的主要实现类

- void add(int index, Object ele)在指定的索引位置 index 添加元素
- boolean addAll(int index, Collection eles)在指定索引位置，加上一个集合
- Object get(int index)获取指定索引的元素
- Object remove(int index)删除指定索引位置的元素
- Object set(int index, Object ele)设置指定的索引的位置为 ele
- int indexOf(Object obj)返回 obj 在集合中首次出现的位置，没有则返回-1；
- int lastIndexOf(Object obj)返回 obj 在集合最后一次出现的位置，没有则返回-1；

- List subList(int fromIdx, int toIdx)返回从 fromIdx 到 toIdx 结束的子 list,左闭右开的区间

LinkedList

- void addFirst(Object obj)：在 0 的位置新增一个 obj
- void addLast(Object obj)：在最后的位新增一个 obj
- Object getFirst()：获取第一个元素
- Object getLast()：获取最后一个元素
- Object removeFirst()：删除第一个元素
- Object removeLast()：删除最后一个元素
- Collections.reverse(list);翻转字符串
- Collections.sort(list); 排序字符串

常见的 List 和数组之间的相互转换

- **int[] 转 List<Integer>**

```
List<Integer> list1 = Arrays.stream(data).boxed().collect(Collectors.toList());
```

说明：Arrays.stream(arr) 可以替换成 IntStream.of(arr)。

1.使用 Arrays.stream 将 int[]转换成 IntStream。

2.使用 IntStream 中的 boxed()装箱。将 IntStream 转换成 Stream<Integer>。

3.使用 Stream 的 collect()，将 Stream<T>转换成 List<T>，因此正是 List<Integer>。

- **int[] 转 Integer[]**

```
Integer[] integers1 = Arrays.stream(data).boxed().toArray(Integer[]::new);
```

说明：前两步同上，此时是 Stream<Integer>。然后使用 Stream 的 toArray，传入 IntFunction<A[]> generator。这样就可以返回 Integer 数组。不然默认是 Object[]。

- **List<Integer> 转 Integer[]**

```
Integer[] integers2 = list1.toArray(new Integer[0]);
```

说明：调用 toArray。传入参数 T[] a。这种用法是目前推荐的。

- **List<String>转 String[]**

```
String[] strs1 = list.toArray(new String[0]);
```

说明：如果指定的数组能容纳该 collection，则返回包含此 collection 元素的数组。否则，将根据指定数组的运行时类型和此 collection 的大小分配一个新数组。这里给的参数的数组长度是 0，因此就会返回包含此 collection 中所有元素的数组，并且返回数组的类型与指定数组的运行时类型相同。

- **List<Integer> 转 int[]**

```
int[] arr1 = list1.stream().mapToInt(Integer::valueOf).toArray();
```

说明：想要转换成 int[]类型，就得先转成 IntStream。这里就通过 mapToInt()把 Stream<Integer> 调用 Integer::valueOf 来转成 IntStream，而 IntStream 中默认 toArray()转成 int[]。

- **Integer[] 转 int[]**

```
int[] arr2 = Arrays.stream(integers1).mapToInt(Integer::valueOf).toArray();
```

说明：思路同上。先将 Integer[]转成 Stream<Integer>，再转成 IntStream。

- **Integer[] 转 List<Integer>**

```
List<Integer> list2 = Arrays.asList(integers1);
```

- **String[] 转 List<String>**

```
String[] strings1 = {"a", "b", "c"};
List<String> list3 = Arrays.asList(strings1);
```

【定义链表】

```
class ListNode{
```

禁止商用 公众号：(yes的练级攻略)

```
        int val;
        ListNode next;
        ListNode() {}
        ListNode(int val) { this.val = val; }
        ListNode(int val, ListNode next) { this.val = val; this.next = next;}
}
```

【生成链表】

主函数：

```
int[] arr = {1,2,3,4,5,6,7};
ListNode head = creat(arr);

public static ListNode creat(int[] nums){
    ListNode pre = new ListNode(0), cur = pre;
    for (int num : nums){
        cur.next = new ListNode(num);
        cur = cur.next;
    }
    return pre.next;
}
```

【输出链表】

方法 1：遍历输出

主函数只要：printList(head)

```
public static void printList(ListNode head){
    while (head != null) {
        if (head.next != null) System.out.print(head.val + "->");
        else System.out.print(head.val);
        head = head.next;
    }
    System.out.println();
}
```

方法 2：重写 toString 方法

主函数只要：System.out.println(ln[0].toString());

```
class ListNode{
        int val;
        ListNode next;
        ListNode(int val) { this.val = val; }
        @Override
        public String toString(){
            ListNode tmp = this;
            StringBuffer sb = new StringBuffer();
            while (tmp != null) {
                sb.append(tmp.val);
                if (tmp.next != null) {
                    sb.append("->");
```

禁止商用 公众号：yes的练级攻略

```
            }
            tmp = tmp.next;
        }
        return sb.toString();
    }
}
```

【计算链表的长度】

```
    public int length(ListNode head) {
        int cnt = 0;
        ListNode dummy = head;
        while (dummy != null){
            cnt++;
            dummy = dummy.next;
        }
        return cnt;
    }
```

1、两数相加（2）

2. 两数相加

难度 中等 4459 收藏 分享 切换为英文 关注 反馈

给出两个 **非空** 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 **逆序** 的方式存储的，并且它们的每个节点只能存储 **一位** 数字。

如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例：

```
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

https://leetcode-cn.com/problems/add-two-numbers/solution/liang-shu-xiang-jia-by-leetcode/

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
```

```
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode cur = dummy;
        int carry = 0;
        while (l1 != null || l2 != null) {
            int x = (l1 != null) ? l1.val : 0;
            int y = (l2 != null) ? l2.val : 0;
            int sum = carry + x + y;
            carry = sum/10;
            cur.next = new ListNode(sum % 10);
            cur = cur.next;
            l1 = l1 != null ? l1.next: l1;
            l2 = l2 != null ? l2.next: l2;
        }
        if (carry > 0) {
            cur.next = new ListNode(carry);
        }
        return dummy.next;
    }
}
```

2、链表求和（面试题 02.05）【与上一题相比，有很多边界条件】

面试题 02.05. 链表求和

难度 中等 28 收藏 分享 切换为英文 关注 反馈

给定两个用链表表示的整数，每个节点包含一个数位。

这些数位是反向存放的，也就是个位排在链表首部。

编写函数对这两个整数求和，并用链表形式返回结果。

示例：

输入： (7 -> 1 -> 6) + (5 -> 9 -> 2)，即617 + 295
输出： 2 -> 1 -> 9，即912

进阶： 假设这些数位是正向存放的，请再做一遍。

示例：

输入： (6 -> 1 -> 7) + (2 -> 9 -> 5)，即617 + 295
输出： 9 -> 1 -> 2，即912

https://leetcode-cn.com/problems/sum-lists-lcci/solution/java-ji-bai-shuang-bai-lian-biao-qiu-he-by-zdl-n/

本题代码

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        //初始进位为0
        int pre = 0;
        //操作数
        ListNode mid = new ListNode(0);
        //返回头节点
        ListNode anws = mid ;
        //当 l1 和 l2 都不为 null 时进入 while 循环
        while(l1!=null&&l2!=null){
            //操作数赋值
            mid.val = (l1.val+l2.val+pre)%10;
            //更新进位
            pre = (l1.val+l2.val+pre)/10;
            //更新头节点
            l1 = l1.next;
            l2 = l2.next;
            //头节点更新后判断是否为空
            if(l1==null){
                //如果 l1 头节点为空且进位为 0，则操作数的 next 直接为 l2 剩下的
                if(pre==0) {
                    mid.next = l2;
                    return anws;
                }else {
                    //如果有进位，则递归调用 addTwoNumbers 方法
                    mid.next = addTwoNumbers(l2,new ListNode(pre));
                    return anws;
                }
            }
            //同上
            if(l2 == null){
                if(pre==0) {
                    mid.next = l1;
                    return anws;
                }else {
                    mid.next = addTwoNumbers(l1,new ListNode(pre));
```

```
                return anws;
            }
        }
        //l1 l2 更新后都不为null，则设置操作数为0 进入下一次while 循环
        mid.next =new ListNode(0);
        mid = mid.next;
    }
    //l1 为null，直接不能进入上面while 循环的情况下，直接返回l2
    if(l1==null){
        return l2;
    }//同上
    else if(l2 ==null){
        return l1;
    }
    return anws;
  }
}
```

可运行代码（含反向存放，反向输出和正向存放和正向输出）

```
import java.util.*;
class ListNode{
    int val;
    ListNode next;
    ListNode(){}
    ListNode(int x){this.val = x;}
}
public class Main {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        String before1 = sc.nextLine();
        String before2 = sc.nextLine();
        String[] strs1 = before1.split("->");
        String[] strs2 = before2.split("->");
        int[] nums1 = new int[strs1.length];
        for (int i = 0; i < strs1.length; i++){
            nums1[i] = Integer.parseInt(strs1[i]);
        }
        int[] nums2 = new int[strs2.length];
        for (int i = 0; i < strs2.length; i++){
            nums2[i] = Integer.parseInt(strs2[i]);
        }
        ListNode l1 = creat(nums1), l2 = creat(nums2);
        System.out.println("反向存放，反向输出：");
        ListNode res1 = addTwoNumbers(l1, l2);
        printList(res1);
```

```
        System.out.println();
        System.out.println("正向存放，正向输出：");
        ListNode res2 = addTwoNumbers2(l1, l2);
        printList(res2);
        System.out.println();
    }
    public static ListNode creat(int[] nums){
        ListNode pre = new ListNode(0), cur = pre;
        for (int num : nums){
            cur.next = new ListNode(num);
            cur = cur.next;
        }
        return pre.next;
    }
    public static void printList(ListNode head){
        while (head != null){
            if (head.next != null) System.out.print(head.val+"->");
            else System.out.print(head.val);
            head = head.next;
        }
    }
    //逆序存放，逆序输出
    public static ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        int pre = 0;
        ListNode mid = new ListNode(0);
        ListNode anws = mid ;
        while(l1!=null&&l2!=null){
            mid.val = (l1.val + l2.val + pre) % 10;
            pre = (l1.val + l2.val + pre) / 10;
            l1 = l1.next;
            l2 = l2.next;
            if(l1 == null){
                if(pre == 0) {
                    mid.next = l2;
                    return anws;
                }else {
                    mid.next = addTwoNumbers(l2, new ListNode(pre));
                    return anws;
                }
            }
            if(l2 == null){
                if(pre == 0) {
                    mid.next = l1;
                    return anws;
```

```
                }else {
                    mid.next = addTwoNumbers(l1, new ListNode(pre));
                    return anws;
                }
            }
            mid.next =new ListNode(0);
            mid = mid.next;
        }
        if(l1 == null) return l2;
        if(l2 == null) return l1;
        return anws;
    }
    //正向存放，正向输出
    public static ListNode addTwoNumbers2(ListNode l1, ListNode l2) {
        Stack<Integer> stack1 = new Stack<>();
        Stack<Integer> stack2 = new Stack<>();
        while (l1 != null) {
            stack1.push(l1.val);
            l1 = l1.next;
        }
        while (l2 != null) {
            stack2.push(l2.val);
            l2 = l2.next;
        }
        int carry = 0;
        ListNode head = null;
        while (!stack1.isEmpty() || !stack2.isEmpty() || carry > 0) {
            int sum = carry;
            sum += stack1.isEmpty()? 0: stack1.pop();
            sum += stack2.isEmpty()? 0: stack2.pop();
            ListNode node = new ListNode(sum % 10);
            node.next = head;
            head = node;
            carry = sum / 10;
        }
        return head;
    }
}
```

3、两数相加 2（445）【正向存放，正向输出】

更多硬核内容就在公众号「yes 的练级攻略」

445. 两数相加 II

难度 中等 263 收藏 分享 切换为英文 关注 反馈

给你两个 **非空** 链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

进阶：

如果输入链表不能修改该如何处理？换句话说，你不能对列表中的节点进行翻转。

示例：

```
输入：(7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 8 -> 0 -> 7
```

https://leetcode-cn.com/problems/add-two-numbers-ii/solution/javakai-fa-by-sweetiee/

🙋‍♀️就喜欢这种短短的打卡题，短短的打个卡，短短的发个题解，哎又是美妙的夜晚！

update：统一回复下头插法的评论，不熟悉的可以用迭代法去做做 206. 反转链表，需要链表逆序的时候就用头插法。

用 stack 保存链表，再从 stack 中取出来，就是数字从低位到高位访问了。

复杂度分析

- 时间复杂度：$O(max(m,n))$，其中 m 与 n 分别为两个链表的长度。我们需要遍历每个链表。
- 空间复杂度：$O(m+n)$，其中 m 与 n 分别为两个链表的长度。这是我们把链表内容放入栈中所用的空间。

```java
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        Stack<Integer> stack1 = new Stack<>();
        Stack<Integer> stack2 = new Stack<>();
        while (l1 != null) {
            stack1.push(l1.val);
            l1 = l1.next;
        }
        while (l2 != null) {
            stack2.push(l2.val);
            l2 = l2.next;
        }
        int carry = 0;
        ListNode head = null;
        while (!stack1.isEmpty() || !stack2.isEmpty() || carry > 0) {
            int sum = carry;
            sum += stack1.isEmpty()? 0: stack1.pop();
            sum += stack2.isEmpty()? 0: stack2.pop();
            ListNode node = new ListNode(sum % 10);
            node.next = head;
            head = node;
            carry = sum / 10;
```

```
            }
            return head;
        }
    }
```

4、给单链表加一（369）

369. 给单链表加一

难度 中等 36 收藏 分享 切换为英文 关注 反

用一个 **非空** 单链表来表示一个非负整数，然后将这个整数加一。

你可以假设这个整数除了 0 本身，没有任何前导的 0。

这个整数的各个数位按照 **高位在链表头部、低位在链表尾部** 的顺序排列。

示例：

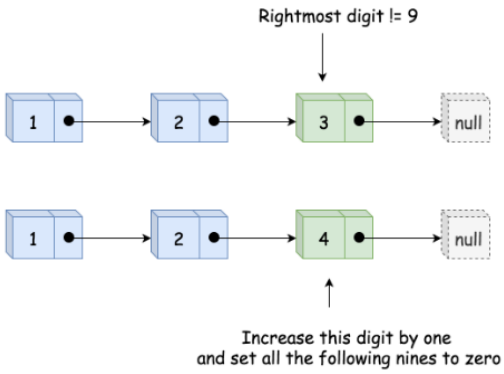
输入： [1,2,3]
输出： [1,2,4]

https://leetcode-cn.com/problems/plus-one-linked-list/solution/gei-dan-lian-biao-jia-yi-by-leetcode/

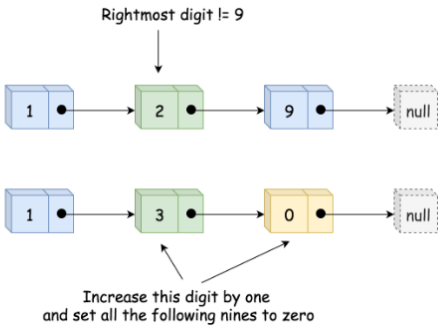
方法一：哨兵头节点
加法运算

找出最靠右的不是 9 的数字，将该数字加 1。然后将该数字之后所有的 9 改成 0。

下面是一个简单的例子，根据上述描述的方法可以得到正确答案。

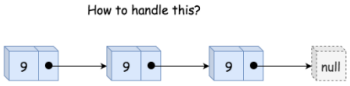


下面是一个相对复杂一点的例子，根据上述描述的方法也可以得到正确答案。



更多硬核内容就在公众号「yes 的练级攻略」

但下面这个例子就有点问题了。

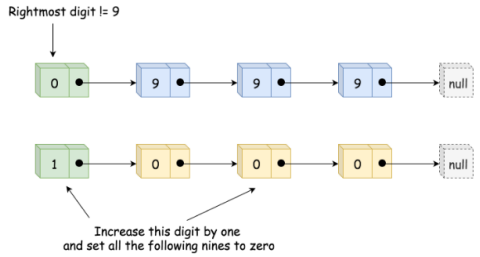


哨兵头结点

针对最后一个案列，需要用到 哨兵节点。树和链表问题中经常会用到哨兵节点，它们的主要目的是将边缘数据的处理常规化。

对于下面这个例子，在最左边增加一个数值为 0 的哨兵节点，这样就能保证一定有数值不为 9 的节点存在。

☆ 收藏



算法

- 初始化哨兵节点 `ListNode(0)` ，同时将它设为新的头节点： `sentinel.next = head` 。
- 找到最靠右的数值不为 9 的节点。
- 将该节点的数值加 1。
- 将该节点之后所有节点数值改为 0。
- 如果哨兵节点的数值为 1，直接返回哨兵节点，否则返回原始头节点 `sentinel.next` 。

复杂度分析

- 时间复杂度：$O(N)$，最多只需遍历两遍链表。
- 空间复杂度：$O(1)$。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode plusOne(ListNode head) {
        ListNode fast = head, slow = new ListNode(0);
        slow.next = head;
        while (fast != null) {
            if (fast.val != 9) slow = fast;
            fast = fast.next;
        }
        slow.val++;
        ListNode cur = slow.next;
        while (cur != null) {
            cur.val = 0;
            cur = cur.next;
        }
```

```
            return slow.next == head ? slow : head;
        }
    }
```

5、反转链表（206）（剑指 24）

206. 反转链表

难度 简单 1222 收藏 分享 切换为英文 关注 反馈

反转一个单链表。

示例:

```
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL
```

进阶:
你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

https://leetcode-cn.com/problems/reverse-linked-list/solution/fan-zhuan-lian-biao-by-leetcode/

方法一：迭代

假设存在链表 $1 \rightarrow 2 \rightarrow 3 \rightarrow \varnothing$，我们想要把它改成 $\varnothing \leftarrow 1 \leftarrow 2 \leftarrow 3$。

在遍历列表时，将当前节点的 `next` 指针改为指向前一个元素。由于节点没有引用其上一个节点，因此必须事先存储其前一个元素。在更改引用之前，还需要另一个指针来存储下一个节点。不要忘记在最后返回新的头引用！

复杂度分析

- 时间复杂度：$O(n)$，假设 n 是列表的长度，时间复杂度是 $O(n)$。
- 空间复杂度：$O(1)$。

方法 1：迭代法（0ms）：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null;
        ListNode cur = head;
        while (cur != null) {
            ListNode tmp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
```

}

方法二：递归

递归版本稍微复杂一些，其关键在于反向工作。假设列表的其余部分已经被反转，现在我该如何反转它前面的部分？

假设列表为：

$$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \rightarrow \ldots \rightarrow n_m \rightarrow \varnothing$$

若从节点 n_{k+1} 到 n_m 已经被反转，而我们正处于 n_k。

$$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \leftarrow \ldots \leftarrow n_m$$

我们希望 n_{k+1} 的下一个节点指向 n_k。

所以，n_k.next.next = n_k。

要小心的是 n_1 的下一个必须指向 Ø 。如果你忽略了这一点，你的链表中可能会产生循环。如果使用大小为 2 的链表测试代码，则可能会捕获此错误。

复杂度分析

- 时间复杂度：$O(n)$，假设 n 是列表的长度，那么时间复杂度为 $O(n)$。
- 空间复杂度：$O(n)$，由于使用递归，将会使用隐式栈空间。递归深度可能会达到 n 层。

方法 2：递归法：

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode p = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return p;
    }
}
```

可运行版本

```
import java.util.Scanner;

class ListNode{
    int val;
    ListNode next = null;
    ListNode(int x) {
        this.val = x;
    }
}

public class RverseList {
    public static ListNode reverseList(ListNode head) {
                ListNode pre = null;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            ListNode cur = head;
            while (cur != null) {
                ListNode tmp = cur.next;
                cur.next = pre;
                pre = cur;
                cur = tmp;
            }
            return pre;
    }

    public static void main(String[] args) {
        while (true) {
            System.out.println("请输入数据: ");
            Scanner sc = new Scanner(System.in);
            String str = sc.nextLine();
            String [] strs = str.split(",");
            ListNode head = new ListNode(0);
            ListNode tail = head;
            for (String s : strs) {
                ListNode newNode = new ListNode(Integer.valueOf(s));
                tail.next = newNode;
                tail = tail.next;
            }
            head = head.next;
            ListNode node = reverseList(head);
            while(node!=null){
                if (node.next == null) {
                    System.out.print(node.val);
                } else {
                    System.out.print(node.val + ",");
                }
                node = node.next;
            }
            System.out.println("");
        }
    }
}
```

6、反转链表 2（92）

禁止商用 公众号：yes的练级攻略

92. 反转链表 II

难度 中等 467 收藏 分享 切换为英

反转从位置 *m* 到 *n* 的链表。请使用一趟扫描完成反转。

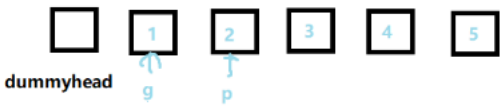
说明:
1 ≤ *m* ≤ *n* ≤ 链表长度。

示例:

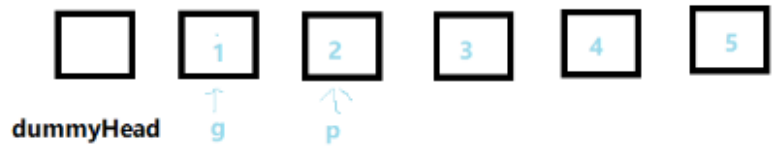
```
输入: 1->2->3->4->5->NULL, m = 2, n = 4
输出: 1->4->3->2->5->NULL
```

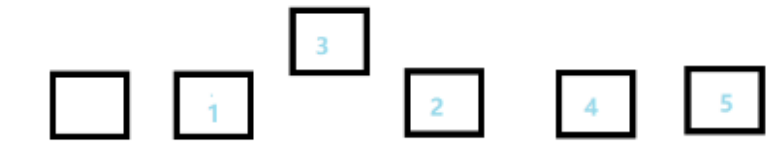
https://leetcode-cn.com/problems/reverse-linked-list-ii/solution/java-shuang-zhi-zhen-tou-cha-fa-by-mu-yi-cheng-zho/

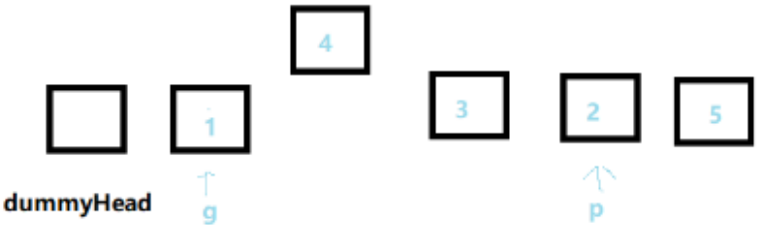
思路：1、我们定义两个指针，分别称之为g(guard 守卫)和p(point)。
我们首先根据方法的参数m确定g和p的位置。将g移动到第一个要反转的节点的前面，将p移动到第一个要反转的节点的位置上。我们以m=2，n=4为例。



2、将p后面的元素删除，然后添加到g的后面。也即头插法。







3、根据m和n重复步骤（2）
4、返回dummyHead.next

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode g = dummy, p = dummy.next;
        for (int i = 0; i < m - 1; i++) {
            g = g.next;
            p = p.next;
        }
        for (int i = 0; i < n - m; i++){
            ListNode removed = p.next;
            p.next = p.next.next;
            removed.next = g.next;
            g.next = removed;
        }
        return dummy.next;
    }
}
```

7、从尾到头打印链表（剑指 6）

剑指 Offer 06. 从尾到头打印链表

难度 简单 43 收藏 分享 切换为英文 关注 反馈

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

```
输入：head = [1,3,2]
输出：[2,3,1]
```

限制：

0 <= 链表长度 <= 10000

https://leetcode-cn.com/problems/cong-wei-dao-tou-da-yin-lian-biao-lcof/solution/mian-shi-ti-06-cong-wei-dao-tou-da-yin-lian-biao-d/

方法 1：辅助队列（7ms）

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] reversePrint(ListNode head) {
        ArrayList<Integer> list = new ArrayList<>();
        ListNode tmp = head;
        while (tmp != null) {
            list.add(0, tmp.val); //第一个位置为索引，第二个位置为值
            tmp = tmp.next;
        }
        int[] res = new int[list.size()];
        //第一种办法 ArrayList 转成 int[]的方法

        //res = list.stream().mapToInt(Integer::valueOf).toArray();
        //第二种 ArrayList 转成 int[]的方法
        for (int i = 0; i < list.size(); i++){
            res[i] = list.get(i);
        }
        return res;
    }
}
```

方法 2：反转链表以后输出（0ms）

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] reversePrint(ListNode head) {
        int cnt = length(head);
        ListNode rev = reverseList(head);
        int[] ans = new int[cnt];
```

```
        for (int i = 0; i < cnt; i++){
            ans[i] = rev.val;
            rev = rev.next;
        }
        return ans;
    }
    public ListNode reverseList(ListNode head) {
        ListNode pre = null, cur = head;
        while (cur != null){
            ListNode tmp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
    public int length(ListNode head) {
        int cnt = 0;
        ListNode dummy = head;
        while (dummy != null){
            cnt++;
            dummy = dummy.next;
        }
        return cnt;
    }
}
```

8、重排链表（143）

143. 重排链表

难度 中等 263 收藏 分享 切换为英文 关注

给定一个单链表 L：$L_0→L_1→…→L_{n-1}→L_n$，
将其重新排列后变为：$L_0→L_n→L_1→L_{n-1}→L_2→L_{n-2}→…$

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1:

```
给定链表 1->2->3->4, 重新排列为 1->4->2->3.
```

示例 2:

```
给定链表 1->2->3->4->5, 重新排列为 1->5->2->4->3.
```

https://leetcode-cn.com/problems/reorder-list/solution/3ge-bu-zou-jian-dan-yi-dong-by-wang_dong/

```
/**
```

```
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = n
ext; }
 * }
 */
```

方法 1：直接反转

```
class Solution {
    public void reorderList(ListNode head) {
        if (head == null || head.next == null) return;
        ListNode slow = head;
        ListNode fast = head;
        while (fast.next != null && fast.next.next != null){
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode pre = slow;
        ListNode cur = slow.next;
        while (cur.next != null){
            ListNode tmp = cur.next;
            cur.next = tmp.next;
            tmp.next = pre.next;
            pre.next = tmp;
        }
        ListNode p1 = head;
        ListNode p2 = slow.next;
        while (p1 != slow){
            slow.next = p2.next;
            p2.next = p1.next;
            p1.next = p2;
            p1 = p2.next;
            p2 = slow.next;
        }
    }
}
```

方法 2：单独写一个反转函数（推荐）

```
class Solution {
    public void reorderList(ListNode head) {
        if (head == null || head.next == null || head.next.next == null) {
            return;
```

```
        }
        //找中点，链表分成两个
        ListNode slow = head;
        ListNode fast = head;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        ListNode newHead = slow.next;
        slow.next = null;

        //第二个链表倒置
        newHead = reverseList(newHead);

        //链表节点依次连接
        while (newHead != null) {
            ListNode temp = newHead.next;
            newHead.next = head.next;
            head.next = newHead;
            head = newHead.next;
            newHead = temp;
        }

    }
    private ListNode reverseList(ListNode head) {
        if (head == null) {
            return null;
        }
        ListNode tail = head;
        head = head.next;

        tail.next = null;

        while (head != null) {
            ListNode temp = head.next;
            head.next = tail;
            tail = head;
            head = temp;
        }

        return tail;
    }
}
```

禁止商用
公众号：
yes的练级攻略

9、两两交换链表中的节点（24）

24. 两两交换链表中的节点

难度 中等 575 收藏 分享 切换为英文 关注

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例:

```
给定 1->2->3->4, 你应该返回 2->1->4->3.
```

https://leetcode-cn.com/problems/swap-nodes-in-pairs/solution/liang-liang-jiao-huan-lian-biao-zhong-de-jie-di-19/

方法一：递归

这个题目要求我们从第一个节点开始两两交换链表中的节点，且要真正的交换节点。

算法:

- 从链表的头节点 `head` 开始递归。
- 每次递归都负责交换一对节点。由 `firstNode` 和 `secondNode` 表示要交换的两个节点。
- 下一次递归则是传递的是下一对需要交换的节点。若链表中还有节点，则继续递归。
- 交换了两个节点以后，返回 `secondNode`，因为它是交换后的新头。
- 在所有节点交换完成以后，我们返回交换后的头，实际上是原始链表的第二个节点。

方法 1：递归

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode firstNode = head, secondNode = head.next;
        firstNode.next = swapPairs(secondNode.next);
        secondNode.next = firstNode;
        return secondNode;
    }
}
```

方法二：迭代

我们把链表分为两部分，即奇数节点为一部分，偶数节点为一部分，`A` 指的是交换节点中的前面的节点，`B` 指的是要交换节点中的后面的节点。在完成它们的交换，我们还得用 `prevNode` 记录 `A` 的前驱节点。

算法：

1. `firstNode（即 A）` 和 `secondNode（即 B）` 分别遍历偶数节点和奇数节点，即两步看作一步。
2. 交换两个节点：

```
firstNode.next = secondNode.next
secondNode.next = firstNode
```

3. 还需要更新 `prevNode.next` 指向交换后的头。

```
prevNode.next = secondNode
```

4. 迭代完成后得到最终的交换结果。

方法 2：迭代

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode preNode = dummy;
        while (head != null && head.next != null){
            ListNode firstNode = head, secondNode = head.next;
            preNode.next = secondNode;
            firstNode.next = secondNode.next;
            secondNode.next = firstNode;
            preNode = firstNode;
            head = firstNode.next;
        }
        return dummy.next;
    }
}
```

10、K 个一组翻转链表（25）

更多硬核内容就在公众号「yes 的练级攻略」

25. K 个一组翻转链表

难度 困难 667 收藏 分享 切换为英文 关注

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例：

给你这个链表：`1->2->3->4->5`

当 $k = 2$ 时，应当返回：`2->1->4->3->5`

当 $k = 3$ 时，应当返回：`3->2->1->4->5`

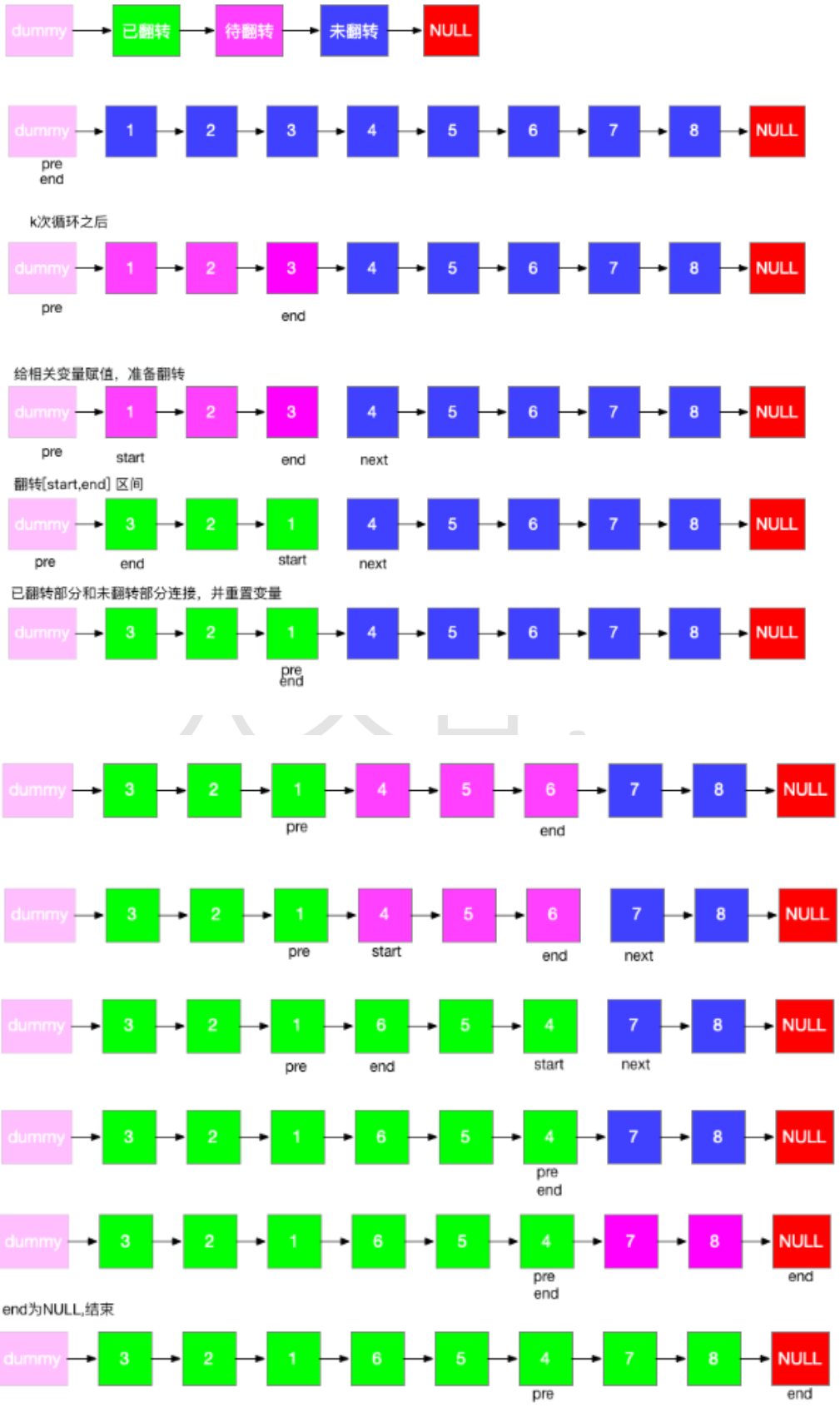
说明：

- 你的算法只能使用常数的额外空间。
- **你不能只是单纯的改变节点内部的值**，而是需要实际进行节点交换。

https://leetcode-cn.com/problems/reverse-nodes-in-k-group/solution/tu-jie-kge-yi-zu-fan-zhuan-lian-biao-by-user7208t/

步骤分解:

1. 链表分区为已翻转部分+待翻转部分+未翻转部分
2. 每次翻转前，要确定翻转链表的范围，这个必须通过 `k` 此循环来确定
3. 需记录翻转链表前驱和后继，方便翻转完成后把已翻转部分和未翻转部分连接起来
4. 初始需要两个变量 `pre` 和 `end`， `pre` 代表待翻转链表的前驱， `end` 代表待翻转链表的末尾
5. 经过k此循环， `end` 到达末尾，记录待翻转链表的后继 `next = end.next`
6. 翻转链表，然后将三部分链表连接起来，然后重置 `pre` 和 `end` 指针，然后进入下一次循环
7. 特殊情况，当翻转部分长度不足 `k` 时，在定位 `end` 完成后， `end==null`，已经到达末尾，说明题目已完成，直接返回即可
8. 时间复杂度为 $O(n * K)$ 最好的情况为 $O(n)$ 最差的情况未 $O(n^2)$
9. 空间复杂度为 $O(1)$ 除了几个必须的节点指针外，我们并没有占用其他空间




```
/**
 * Definition for singly-linked list.
```

```
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy, end = dummy;
        while (end.next != null){
            for (int i = 0; i < k && end != null; i++){
                end = end.next;
            }
            if (end == null) break;
            ListNode start = pre.next, next = end.next;
            end.next = null;
            pre.next = reverse(start);
            start.next = next;
            pre = start;
            end = pre;
        }
        return dummy.next;
    }
    private ListNode reverse(ListNode head){
        ListNode pre = null, cur = head;
        while (cur != null){
            ListNode tmp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
}
```

11、奇偶链表（328）

更多硬核内容就在公众号「yes 的练级攻略」

328. 奇偶链表

难度 中等 220 收藏 分享 切换为英文 关注 反馈

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nodes 为节点总数。

示例 1:

```
输入: 1->2->3->4->5->NULL
输出: 1->3->5->2->4->NULL
```

示例 2:

```
输入: 2->1->3->5->6->4->7->NULL
输出: 2->3->6->7->1->5->4->NULL
```

说明:

- 应当保持奇数节点和偶数节点的相对顺序。
- 链表的第一个节点视为奇数节点，第二个节点视为偶数节点，以此类推。

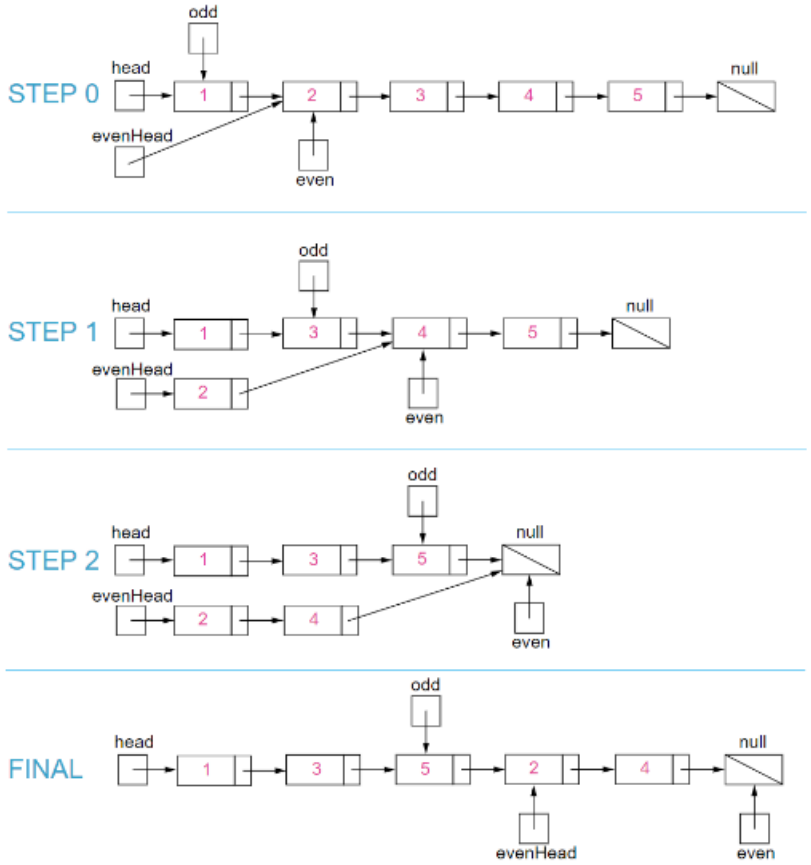
禁止商用
公众号：
yes的练级攻略

https://leetcode-cn.com/problems/odd-even-linked-list/solution/qi-ou-lian-biao-by-leetcode/

一个 `LinkedList` 需要一个头指针和一个尾指针来支持双端操作。我们用变量 `head` 和 `odd` 保存奇链表的头和尾指针。`evenHead` 和 `even` 保存偶链表的头和尾指针。算法会遍历原链表一次并把奇节点放到奇链表里去、偶节点放到偶链表里去。遍历整个链表我们至少需要一个指针作为迭代器。这里 `odd` 指针和 `even` 指针不仅仅是尾指针，也可以扮演原链表迭代器的角色。

解决链表问题最好的办法是在脑中或者纸上把链表画出来。比方说：

解决链表问题最好的办法是在脑中或者纸上把链表画出来。比方说：



图片 1. 奇偶链表的例子

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if (head == null) return head;
```

```
        ListNode odd = head, even = head.next, evenHead = even;
        while (even != null && even.next != null){
            odd.next = even.next;
            odd = odd.next;
            even.next = odd.next;
            even = even.next;
        }
        odd.next = evenHead;
        return head;
    }
}
```

12、旋转链表（61）

61. 旋转链表

难度 中等 303 收藏 分享 切换为英文 关注 反馈

给定一个链表，旋转链表，将链表每个节点向右移动 k 个位置，其中 k 是非负数。

示例 1:

```
输入: 1->2->3->4->5->NULL, k = 2
输出: 4->5->1->2->3->NULL
解释:
向右旋转 1 步: 5->1->2->3->4->NULL
向右旋转 2 步: 4->5->1->2->3->NULL
```

示例 2:

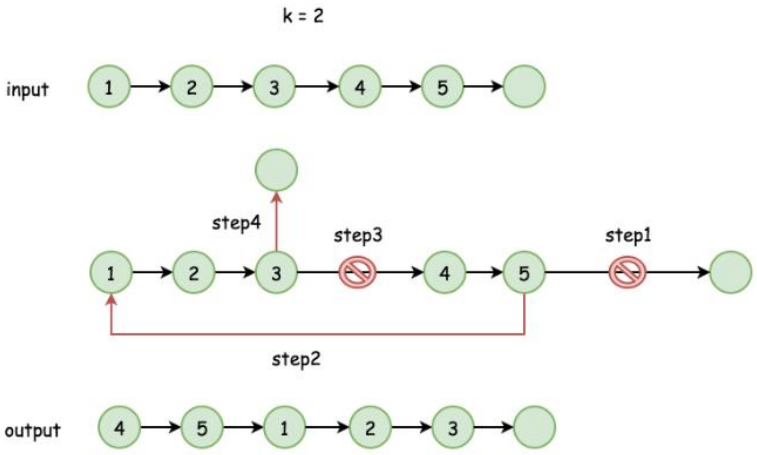
```
输入: 0->1->2->NULL, k = 4
输出: 2->0->1->NULL
解释:
向右旋转 1 步: 2->0->1->NULL
向右旋转 2 步: 1->2->0->NULL
向右旋转 3 步: 0->1->2->NULL
向右旋转 4 步: 2->0->1->NULL
```

https://leetcode-cn.com/problems/rotate-list/solution/xuan-zhuan-lian-biao-by-leetcode/

方法 1：
直觉

链表中的点已经相连，一次旋转操作意味着：

- 先将链表闭合成环
- 找到相应的位置断开这个环，确定新的链表头和链表尾



> 新的链表头在哪里？

在位置 `n-k` 处，其中 `n` 是链表中点的个数，新的链表尾就在头的前面，位于位置 `n-k-1`。

> 我们这里假设 `k < n`
>
> 如果 `k >= n` 怎么处理？

`k` 可以被写成 `k = (k // n) * n + k % n` 两者加和的形式，其中前面的部分不影响最终的结果，因此只需要考虑 `k%n` 的部分，这个值一定比 `n` 小。

算法

算法实现很直接：

- 找到旧的尾部并将其与链表头相连 `old_tail.next = head`，整个链表闭合成环，同时计算出链表的长度 `n`。
- 找到新的尾部，第 `(n - k % n - 1)` 个节点，新的链表头是第 `(n - k % n)` 个节点。
- 断开环 `new_tail.next = None`，并返回新的链表头 `new_head`。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是链表中的元素个数
- 空间复杂度：$O(1)$，因为只需要常数的空间

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || head.next == null) return head;
        // 1. 生成环
        ListNode old_tail = head;
        int n = 1;
```

```
        while (old_tail.next != null){
            old_tail = old_tail.next;
            n++;
        }
        old_tail.next = head;

        // find new tail : (n - k % n - 1)th node
        // and new head : (n - k % n)th node
        ListNode new_tail = head;
        for (int i = 0; i < n - k % n - 1; i++){
            new_tail = new_tail.next;
        }
        ListNode new_head = new_tail.next;

        // break the ring
        new_tail.next = null;

        return new_head;
    }
}
```

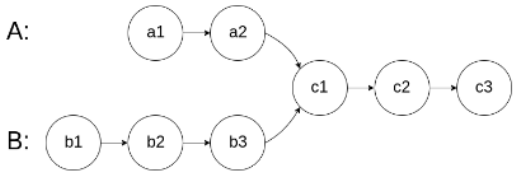
13、相交链表（160）【两个链表的第一个公共节点（剑指 52）】【链表相交（面试题 02.07）】

160. 相交链表

难度 简单 681 收藏 分享 切换为英文 关注 反馈

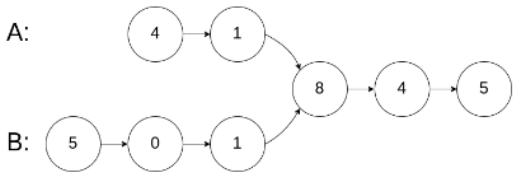
编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

示例 1：



输入： intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出： Reference of the node with value = 8
输入解释： 相交节点的值为 8 （注意，如果两个链表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

更多硬核内容就在公众号「yes 的练级攻略」

面试题 02.07. 链表相交

难度 简单 36 收藏 分享 切换为英文 关注 反馈

给定两个（单向）链表，判定它们是否相交并返回交点。请注意相交的定义基于节点的引用，而不是基于节点的值。换句话说，如果一个链表的第k个节点与另一个链表的第j个节点是同一节点（引用完全相同），则这两个链表相交。

示例 1：

输入： intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出： Reference of the node with value = 8
输入解释： 相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

示例 2：

输入： intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出： Reference of the node with value = 2
输入解释： 相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

示例 3：

输入： intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出： null
输入解释： 从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释： 这两个链表不相交，因此返回 null。

注意：

- 如果两个链表没有交点，返回 null 。
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(*n*) 时间复杂度，且仅用 O(*1*) 内存。

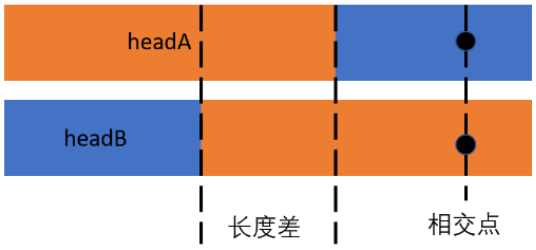
https://leetcode-cn.com/problems/intersection-of-two-linked-lists/solution/intersection-of-two-linked-lists-shuang-zhi-zhen-l/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

- 我们通常做这种题的思路是设定两个指针分别指向两个链表头部，一起向前走直到其中一个到达末端，另一个与末端距离则是两链表的 `长度差` 。再通过长链表指针先走的方式消除长度差，最终两链表即可同时走到相交点。
- **换个方式消除长度差：** 拼接两链表。
  设长-短链表为 `C` ，短-长链表为 `D` （分别代表长链表在前和短链表在前的拼接链表），则当 `C` 走到长短链表交接处时， `D` 走在长链表中，且与长链表头距离为 `长度差` ；

  以下图片帮助理解：当 `ha == hb` 时跳出，返回即可



一图胜千言，看图你就明白了

空间复杂度 $O(1)$ 时间复杂度为 $O(n)$

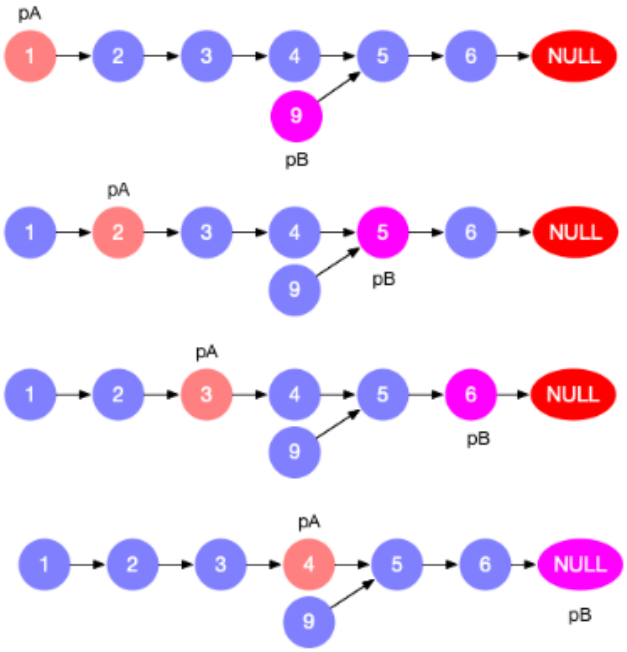
这里使用图解的方式，解释比较巧妙的一种实现。

根据题目意思
如果两个链表相交，那么相交点之后的长度是相同的

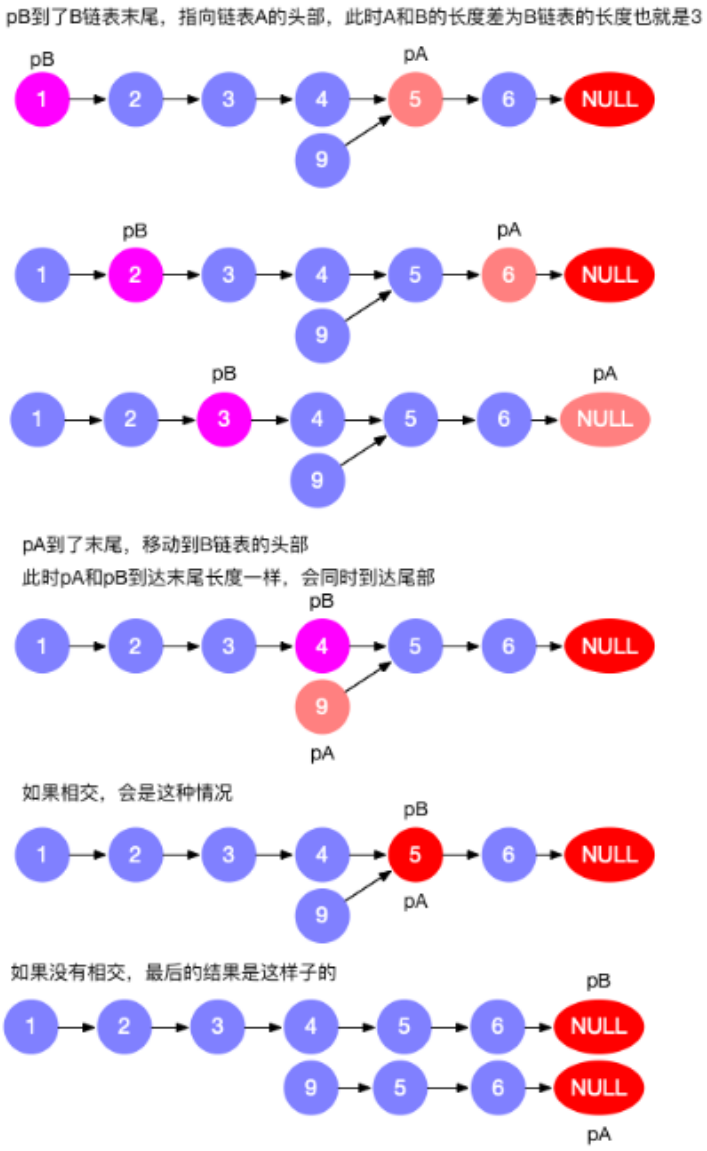
我们需要做的事情是，让两个链表从同距离末尾同等距离的位置开始遍历。这个位置只能是较短链表的头结点位置。
为此，我们必须消除两个链表的长度差

1. 指针 pA 指向 A 链表，指针 pB 指向 B 链表，依次往后遍历
2. 如果 pA 到了末尾，则 pA = headB 继续遍历
3. 如果 pB 到了末尾，则 pB = headA 继续遍历
4. 比较长的链表指针指向较短链表head时，长度差就消除了
5. 如此，只需要将最短链表遍历两次即可找到位置

听着可能有点绕，看图最直观，链表的题目最适合看图了



更多硬核内容就在公众号「yes 的练级攻略」



160 具体代码

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB)
 {
        if (headA == null || headB == null) {
            return null;
        }
        ListNode fast = headA, slow = headB;
        while (fast != slow) {
            fast = fast != null ? fast.next : headB;
            slow = slow != null ? slow.next : headA;
        }
        return fast;
    }
}
```

14、回文链表（234）（面试题 02.06）【验证是否为回文链表】

请判断一个链表是否为回文链表。

示例 1:

```
输入: 1->2
输出: false
```

示例 2:

```
输入: 1->2->2->1
输出: true
```

进阶:
你能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决此题?

禁止转载

https://leetcode-cn.com/problems/palindrome-linked-list/solution/hui-wen-lian-biao-by-leetcode/

方法三：

避免使用 $O(n)$ 额外空间的方法就是改变输入。

我们可以将链表的后半部分反转（修改链表结构），然后将前半部分和后半部分进行比较。比较完成后我们应该将链表恢复原样。虽然不需要恢复也能通过测试用例，因为使用该函数的人不希望链表结构被更改。

算法：

我们可以分为以下几个步骤：

1. 找到前半部分链表的尾节点。
2. 反转后半部分链表。
3. 判断是否为回文。
4. 恢复链表。
5. 返回结果。

执行步骤一，我们可以计算链表节点的数量，然后遍历链表找到前半部分的尾节点。

或者可以使用快慢指针在一次遍历中找到：慢指针一次走一步，快指针一次走两步，快慢指针同时出发。当快指针移动到链表的末尾时，慢指针到链表的中间。通过慢指针将链表分为两部分。

若链表有奇数个节点，则中间的节点应该看作是前半部分。

步骤二可以使用在反向链表问题中找到解决方法来反转链表的后半部分。

步骤三比较两个部分的值，当后半部分到达末尾则比较完成，可以忽略计数情况中的中间节点。

步骤四与步骤二使用的函数相同，再反转一次恢复链表本身。

234 具体代码

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null) {
            return true;
        }
        if (head.next == null) {
```

```
            return true;
        }
        ListNode firstHalfEnd = endOfFirstHalf(head);
        ListNode secondHalfStart = reverse(firstHalfEnd.next);
        ListNode p1 = head, p2 = secondHalfStart;
        while (p2 != null) {
            if (p1.val != p2.val) {
                return false;
            }
            p1 = p1.next;
            p2 = p2.next;
        }
        return true;
    }

    private ListNode reverse(ListNode head) {
        ListNode pre = null, cur = head;
        while (cur != null) {
            ListNode tmp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }

    private ListNode endOfFirstHalf (ListNode head) {
            ListNode fast = head, slow = head;
            while (fast.next != null && fast.next.next != null) {
                fast = fast.next.next;
                slow = slow.next;
            }
            return slow;
    }
}
```

15、环形链表（141）【判断有没有环】

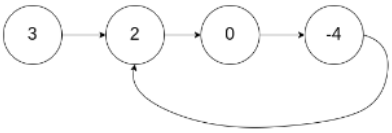
更多硬核内容就在公众号「yes 的练级攻略」

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

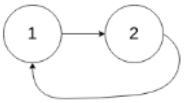
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：true
解释：链表中有一个环，其尾部连接到第一个节点。
```



https://leetcode-cn.com/problems/linked-list-cycle/solution/huan-xing-lian-biao-by-leetcode/

方法二：双指针

思路

想象一下，两名运动员以不同的速度在环形赛道上跑步会发生什么？

算法

通过使用具有 **不同速度** 的快、慢两个指针遍历链表，空间复杂度可以被降低至 $O(1)$。慢指针每次移动一步，而快指针每次移动两步。

如果列表中不存在环，最终快指针将会最先到达尾部，此时我们可以返回 `false`。

现在考虑一个环形链表，把慢指针和快指针想象成两个在环形赛道上跑步的运动员（分别称之为慢跑者与快跑者）。而快跑者最终一定会追上慢跑者。这是为什么呢？考虑下面这种情况（记作情况 A）- 假如快跑者只落后慢跑者一步，在下一次迭代中，它们就会分别跑了一步或两步并相遇。

其他情况又会怎样呢？例如，我们没有考虑快跑者在慢跑者之后两步或三步的情况。但其实不难想到，因为在下一次或者下下次迭代后，又会变成上面提到的情况 A。

复杂度分析

- 时间复杂度：$O(n)$，让我们将 n 设为链表中结点的总数。为了分析时间复杂度，我们分别考虑下面两种情况。
  - ***链表中不存在环：***
    快指针将会首先到达尾部，其时间取决于列表的长度，也就是 $O(n)$。
  - ***链表中存在环：***
    我们将慢指针的移动过程划分为两个阶段：非环部分与环形部分：
    1. 慢指针在走完非环部分阶段后将进入环形部分：此时，快指针已经进入环中 迭代次数 = 非环部分长度 = N
    2. 两个指针都在环形区域中：考虑两个在环形赛道上的运动员 - 快跑者每次移动两步而慢跑者每次只移动一步。其速度的差值为 1，因此需要经过 $\frac{\text{二者之间距离}}{\text{速度差值}}$ 次循环后，快跑者可以追上慢跑者。这个距离几乎就是 "环形部分长度 K" 且速度差值为 1，我们得出这样的结论 迭代次数 = 近似于 "环形部分长度 K".

  因此，在最糟糕的情形下，时间复杂度为 $O(N+K)$，也就是 $O(n)$。
- 空间复杂度：$O(1)$，我们只使用了慢指针和快指针两个结点，所以空间复杂度为 $O(1)$。

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }
        ListNode slow = head;
        ListNode fast = head.next;
        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false;
            }
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
    }
}
```

16、环形链表 2（142）【返回链表开始入环的第一个节点】【环路检测（面试题 02.08）】

更多硬核内容就在公众号「yes 的练级攻略」

142. 环形链表 II

难度 中等 561 收藏 分享 切换为英文 关注 反馈

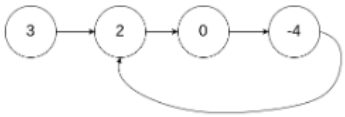
给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 `null`。

为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 `pos` 是 `-1`，则在该链表中没有环。

说明： 不允许修改给定的链表。

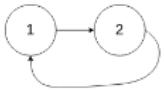
示例 1：

```
输入：head = [3,2,0,-4], pos = 1
输出：tail connects to node index 1
解释：链表中有一个环，其尾部连接到第二个节点。
```



示例 2：

```
输入：head = [1,2], pos = 0
输出：tail connects to node index 0
解释：链表中有一个环，其尾部连接到第一个节点。
```



示例 3：

```
输入：head = [1], pos = -1
输出：no cycle
解释：链表中没有环。
```

进阶：
你是否可以不用额外空间解决此题？

https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/linked-list-cycle-ii-kuai-man-zhi-zhen-shuang-zhi-/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

- 这类链表题目一般都是使用双指针法解决的，例如寻找距离尾部第K个节点、寻找环入口、寻找公共尾部入口等。

算法流程：

1. **双指针第一次相遇：** 设两指针 `fast` ， `slow` 指向链表头部 `head` ， `fast` 每轮走 2 步， `slow` 每轮走 1 步；
    1. **第一种结果：** `fast` 指针走过链表末端，说明链表无环，直接返回 `null` ；
        - TIPS: 若有环，两指针一定会相遇。因为每走 1 轮， `fast` 与 `slow` 的间距 $+1$， `fast` 终会追上 `slow` ；
    2. **第二种结果：** 当 `fast` $=$ `slow` 时， 两指针在环中 **第一次相遇** 。下面分析此时 `fast` 与 `slow` 走过的 **步数关系** ：
        - 设链表共有 $a+b$ 个节点，其中 **链表头部到链表入口** 有 a 个节点（不计链表入口节点）， **链表环** 有 b 个节点（这里需要注意，a 和 b 是未知数，例如图解上链表 $a=4$, $b=5$）；设两指针分别走了 f，s 步，则有：
            1. `fast` 走的步数是 `slow` 步数的 2 倍，即 $f=2s$；（**解析：** `fast` 每轮走 2 步）
            2. `fast` 比 `slow` 多走了 n 个环的长度，即 $f=s+nb$；（**解析：** 双指针都走过 a 步，然后在环内绕圈直到重合，重合时 `fast` 比 `slow` 多走 **环的长度整数倍** ）；
        - 以上两式相减得：$f=2nb$，$s=nb$，即 `fast` 和 `slow` 指针分别走了 $2n$，n 个 **环的周长** （注意： n 是未知数，不同链表的情况不同）。
2. **目前情况分析：**
    - 如果让指针从链表头部一直向前走并统计步数 `k` ，那么所有 **走到链表入口节点时的步数** 是： `k=a+nb` （先走 a 步到入口节点，之后每绕 1 圈环（ b 步）都会再次到入口节点）。
    - 而目前， `slow` 指针走过的步数为 nb 步。因此，我们只要想办法让 `slow` 再走 a 步停下来，就可以到环的入口。
    - 但是我们不知道 a 的值，该怎么办？依然是使用双指针法。我们构建一个指针，此指针需要有以下性质：此指针和 `slow` 一起向前走 `a` 步后，两者在入口节点重合。那么从哪里走到入口节点需要 a 步？答案是链表头部 `head` 。
3. **双指针第二次相遇：**
    - `slow` 指针 **位置不变** ，将 `fast` 指针重新 **指向链表头部节点** ； `slow` 和 `fast` 同时每轮向前走 1 步；
        - TIPS：此时 $f=0$，$s=nb$ ；
    - 当 `fast` 指针走到$f=a$ 步时， `slow` 指针走到步$s=a+nb$，此时 **两指针重合，并同时指向链表环入口** 。
4. **返回 `slow` 指针指向的节点。**

复杂度分析：

- **时间复杂度** $O(N)$ ：第二次相遇中，慢指针须走步数 $a<a+b$；第一次相遇中，慢指针须走步数 $a+b-x<a+b$，其中 x 为双指针重合点与环入口距离；因此总体为线性复杂度；
- **空间复杂度** $O(1)$ ：双指针使用常数大小的额外空间。

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
```

```
 * }
 */
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast = head, slow = head;
        while (true) {
            if (fast == null || fast.next == null) return null;
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        fast = head;
        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }
        return fast;
    }
}
```

17、删除链表的节点（剑指 18）

面试题18. 删除链表的节点

难度 简单 25 收藏 分享 切换为英文 关注 反馈

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

注意：此题对比原题有改动

示例 1:

```
输入: head = [4,5,1,9], val = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], val = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

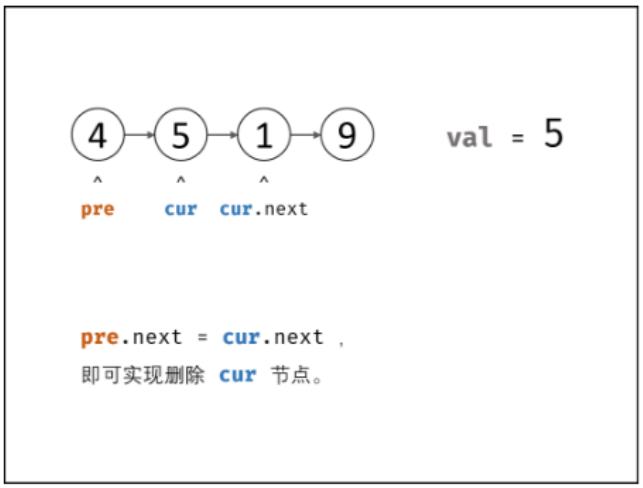
说明:

- 题目保证链表中节点的值互不相同
- 若使用 C 或 C++ 语言，你不需要 `free` 或 `delete` 被删除的节点

https://leetcode-cn.com/problems/shan-chu-lian-biao-de-jie-dian-lcof/solution/mian-shi-ti-18-shan-chu-lian-biao-de-jie-dian-sh-2/

解题思路：

> 删除值为 `val` 的节点可分为两步：定位节点、修改引用。
>
> 1. **定位节点：** 遍历链表，直到 `head.val == val` 时跳出，即可定位目标节点。
> 2. **修改引用：** 设节点 `cur` 的前驱节点为 `pre` ，后继节点为 `cur.next` ；则执行 `pre.next = cur.next` ，即可实现删除 `cur` 节点。



算法流程：

1. **特例处理：** 当应删除头节点 `head` 时，直接返回 `head.next` 即可。
2. **初始化：** `pre = head` ，`cur = head.next` 。
3. **定位节点：** 当 `cur` 为空 **或** `cur` 节点值等于 `val` 时跳出。
    1. 保存当前节点索引，即 `pre = cur` 。
    2. 遍历下一节点，即 `cur = cur.next` 。
4. **删除节点：** 若 `cur` 指向某节点，则执行 `pre.next = cur.next` 。（若 `cur` 指向 $null$ ，代表链表中不包含值为 `val` 的节点。
5. **返回值：** 返回链表头部节点 `head` 即可。

面试题 18 具体代码

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        if (head == null) {
            return null;
        }
        if (head.val == val) {
            return head.next;
        }
        ListNode pre = head, cur = head.next;
        while (cur != null && cur.val != val) {
```

```
            pre = cur;
            cur = cur.next;
        }
        if (cur.val == val) {
            pre.next = cur.next;
        }
        return head;
    }
}
```

第二种方法：结合下一题的方法【注意需要单独判断头尾节点】

```
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        if (head == null) {
            return null;
        }
//头结点删除
        if (head.val == val) {
            return head.next;
        }
        ListNode pre = head, cur = head.next;
        while (cur.val != val) {
            pre = cur;
            cur = cur.next;
        }
//中间节点删除
        if (cur.val == val && cur.next != null) {
            deleteOneNode(cur);
        }
//尾结点删除
        if (cur.val == val && cur.next == null) {
            pre.next = cur.next;
        }
        return head;
    }
    public void deleteOneNode(ListNode node) {
        node.val = node.next.val;
        node.next = node.next.next;
    }
}
```

18、删除链表中的节点（237）【删除中间节点（面试题 02.03）】

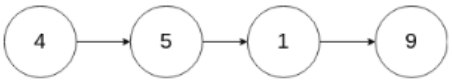
【非末尾节点，只给要求被删除的节点】

237. 删除链表中的节点

难度 简单 694 收藏 分享 切换为英文 关注 反馈

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:



示例 1:

```
输入: head = [4,5,1,9], node = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], node = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。

面试题 02.03. 删除中间节点

难度 简单 50 收藏 分享 切换为英文 关注 反馈

实现一种算法，删除单向链表中间的某个节点（即不是第一个或最后一个节点），假定你只能访问该节点。

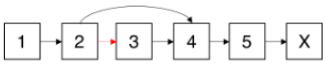
示例：

```
输入：单向链表a->b->c->d->e->f中的节点c
结果：不返回任何数据，但该链表变为a->b->d->e->f
```

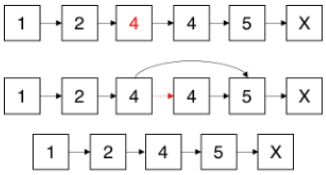
https://leetcode-cn.com/problems/delete-node-in-a-linked-list/solution/shan-chu-lian-biao-zhong-de-jie-dian-by-leetcode/

方法：与下一个节点交换

从链表里删除一个节点 `node` 的最常见方法是修改*之前*节点的 `next` 指针，使其指向*之后*的节点。



因为，我们无法访问我们想要删除的节点 *之前* 的节点，我们始终不能修改该节点的 `next` 指针。相反，我们必须将想要删除的节点的值替换为它后面节点中的值，然后删除它之后的节点。



因为我们知道要删除的节点不是列表的末尾，所以我们可以保证这种方法是可行的。

237 具体代码

```
    public void deleteNode(ListNode node) {
        node.val = node.next.val;
        node.next = node.next.next;
    }
```

19、删除链表的倒数第 N 个节点（19）

19. 删除链表的倒数第N个节点

难度 中等　931　收藏　分享　切换为英文　关注　反馈

给定一个链表，删除链表的倒数第 *n* 个节点，并且返回链表的头结点。

示例：

```
给定一个链表: 1->2->3->4->5, 和 n = 2.

当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 *n* 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/solution/hua-jie-suan-fa-19-shan-chu-lian-biao-de-dao-shu-d/

解题思路

- 标签：链表
- 整体思路是让前面的指针先移动 `n` 步，之后前后指针共同移动直到前面的指针到尾部为止
- 首先设立预先指针 pre，预先指针是一个小技巧，在第2题中进行了讲解
- 设预先指针 `pre` 的下一个节点指向 `head`，设前指针为 `start`，后指针为 `end`，二者都等于 `pre`
- `start` 先向前移动n步
- 之后 `start` 和 `end` 共同向前移动，此时二者的距离为 `n`，当 `start` 到尾部时，`end` 的位置恰好为倒数第 `n` 个节点
- 因为要删除该节点，所以要移动到该节点的前一个才能删除，所以循环结束条件为 `start.next != null`
- 删除后返回 `pre.next`，为什么不直接返回 `head` 呢，因为 `head` 有可能是被删掉的点
- 时间复杂度：$O(n)$

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode pre = new ListNode(0);
        pre.next = head;
        ListNode start = pre, end = pre;
        while(n != 0) {
            start = start.next;
            n--;
        }
        while(start.next != null) {
            start = start.next;
            end = end.next;
        }
        end.next = end.next.next;
        return pre.next;
    }
}
```

20、链表中倒数第 k 个节点（剑指 22）

剑指 Offer 22. 链表中倒数第k个节点

难度 简单 70 收藏 分享 切换为英文 关注 反馈

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。例如，一个链表有6个节点，从头节点开始，它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个节点是值为4的节点。

示例：

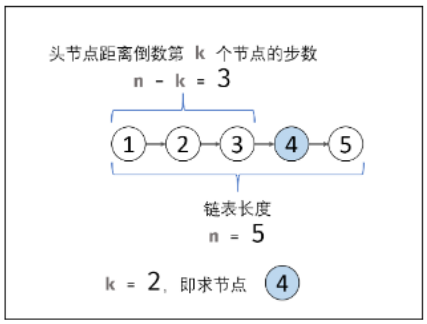
给定一个链表：1->2->3->4->5, 和 *k* = 2.

返回链表 4->5.

https://leetcode-cn.com/problems/lian-biao-zhong-dao-shu-di-kge-jie-dian-lcof/solution/mian-shi-ti-22-lian-biao-zhong-dao-shu-di-kge-j-11/

解题思路：

- 第一时间想到的解法：
  1. 先遍历统计链表长度，记为 n ；
  2. 设置一个指针走 $(n-k)$ 步，即可找到链表倒数第 k 个节点。
- 使用双指针则可以不用统计链表长度。



算法流程：

1. **初始化：** 前指针 `former` 、后指针 `latter` ，双指针都指向头节点 `head` 。
2. **构建双指针距离：** 前指针 `former` 先向前走 k 步（结束后，双指针 `former` 和 `latter` 间相距 k 步）。
3. **双指针共同移动：** 循环中，双指针 `former` 和 `latter` 每轮都向前走一步，直至 `former` 走过链表 **尾节点** 时跳出（跳出后， `latter` 与尾节点距离为 $k-1$，即 `latter` 指向倒数第 k 个节点）。
4. **返回值：** 返回 `latter` 即可。

复杂度分析：

- **时间复杂度 $O(N)$ ：** N 为链表长度；总体看， `former` 走了 N 步， `latter` 走了 $(N-k)$ 步。
- **空间复杂度 $O(1)$ ：** 双指针 `former` 、 `latter` 使用常数大小的额外空间。

代码：

> 本题没有 k 大于链表长度的 case ，因此不用考虑越界问题。
> 考虑越界问题的代码请见第三、四栏。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode former = head, latter = head;
        for(int i = 0; i < k; i++) {
            if(former == null) return null;
            former = former.next;
        }
        while(former != null) {
            former = former.next;
            latter = latter.next;
```

```
        }
        return latter;
    }
}
```

21、返回倒数第 k 个节点（面试题 02.02）【返回节点的值】

面试题 02.02. 返回倒数第 k 个节点

难度 简单 41 收藏 分享 切换为英文 关注

实现一种算法，找出单向链表中倒数第 k 个节点。返回该节点的值。

注意：本题相对原题稍作改动

示例：

```
输入： 1->2->3->4->5 和 k = 2
输出： 4
```

说明：

给定的 *k* 保证是有效的。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public int kthToLast(ListNode head, int k) {
        ListNode fast = head, slow = head;
        while (k-- > 0) {
            fast = fast.next;
        }
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow.val;
    }
}
```

22、寻找链表的中点

3、寻找链表的中点

类似上面的思路，我们还可以让快指针一次前进两步，慢指针一次前进一步，当快指针到达链表尽头时，慢指针就处于链表的中间位置。

```
while (fast != null && fast.next != null) {
    fast = fast.next.next;
    slow = slow.next;
}
// slow 就在中间位置
return slow;
```

当链表的长度是奇数时，slow 恰巧停在中点位置；如果长度是偶数，slow 最终的位置是中间偏右：

23、删除排序链表中的重复元素（83）

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

示例 1:

```
输入: 1->1->2
输出: 1->2
```

示例 2:

```
输入: 1->1->2->3->3
输出: 1->2->3
```

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/solution/hua-jie-suan-fa-83-shan-chu-pai-xu-lian-biao-zhong/

思路

- 标签：链表
- 指定 cur 指针指向头部 head
- 当 cur 和 cur.next 的存在为循环结束条件，当二者有一个不存在时说明链表没有去重复的必要了
- 当 cur.val 和 cur.next.val 相等时说明需要去重，则将 cur 的下一个指针指向下一个的下一个，这样就能达到去重复的效果
- 如果不相等则 cur 移动到下一个位置继续循环
- 时间复杂度：$O(n)$

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode cur = head;
        while(cur != null && cur.next != null) {
            if (cur.next.val == cur.val) {
                cur.next = cur.next.next;
```

```
            } else {
                cur = cur.next;
            }
        }
        return head;
    }
}
```

24、删除排序链表中的重复元素 2（82）【只保留原始链表中没有重复出现的数字】

82. 删除排序链表中的重复元素 II

难度 中等 366 收藏 分享 切换为英文 关注 反馈

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 *没有重复出现* 的数字。

示例 1:

```
输入: 1->2->3->3->4->4->5
输出: 1->2->5
```

示例 2:

```
输入: 1->1->1->2->3
输出: 2->3
```

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list-ii/solution/san-chong-jie-fa-duo-tu-zhan-shi-82-shan-chu-pai-x/

这里我们使用双指针的方式，定义 `a` ， `b` 两个指针。
考虑到一些边界条件，比如 `1->1->1->2` 这种情况，需要把开头的几个 `1` 给去掉，我们增加一个**哑结点**，方便边界处理。

初始的两个指针如下：

- 将 `a` 指针指向哑结点
- 将 `b` 指针指向 `head` (哑结点的下一个节点)

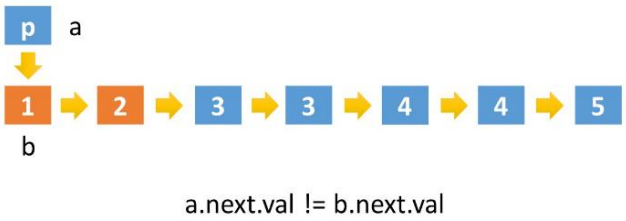
如果 `a` 指向的值**不等于** `b` 指向的值，则两个指针都前进一位
否则，就单独移动 `b` ， `b` 不断往前走，直到 `a` 指向的值**不等于** `b` 指向的值。

注意，这里不是直接比较 `a.val==b.val` ，这么比较不对，因为**初始**的时候， `a` 指向的是哑结点，所以比较逻辑应该是这样：

```
a.next.val == b.next.val
```

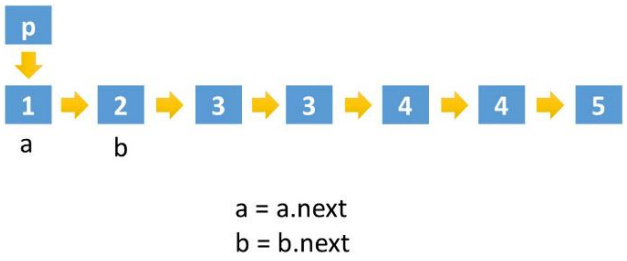
当两个指针指向的值相等时， `b` 不断往前移动，这里是通过一个 `while` 循环判断的，因为要过滤掉 `1->2->2->2->3` 重复的 `2` 。
那么整个逻辑就是两个 `while` ，但时间复杂度不是O(N^2)，而是O(N)，空间上也只是常数级别。

请点击下面幻灯片查看完整的执行过程：



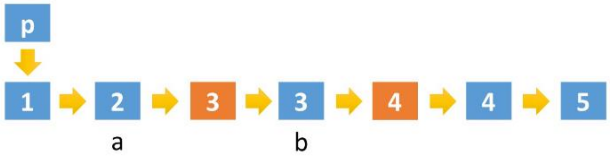
a.next.val != b.next.val

更多硬核内容就在公众号「yes 的练级攻略」

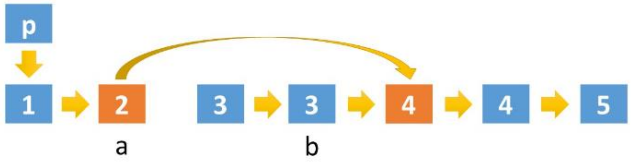


a = a.next
b = b.next

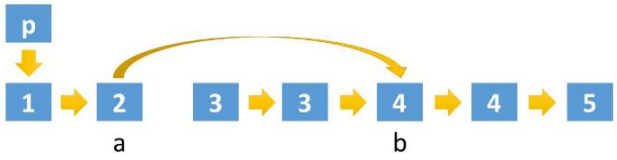
a.next.val == b.next.val



a.next.val != b.next.val



a.next = b.next



b = b.next

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null) {
            return head;
        }
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode pre = dummy;
```

```
            ListNode cur = head;
            while(cur != null && cur.next != null) {
                if (cur.next.val != pre.next.val) {
                    pre = pre.next;
                    cur = cur.next;
                } else {
                    while (cur != null && cur.next != null && cur.next.val
== pre.next.val) {
                        cur = cur.next;
                    }
                    pre.next = cur.next;
                    cur = cur.next;
                }
            }
            return dummy.next;
        }
}
```

25、移除重复节点（面试题 02.01）

面试题 02.01. 移除重复节点

难度 简单 65 收藏 分享 切换为英文 关注 反馈

编写代码，移除未排序链表中的重复节点。保留最开始出现的节点。

示例1:

```
输入：[1, 2, 3, 3, 2, 1]
输出：[1, 2, 3]
```

示例2:

```
输入：[1, 1, 1, 1, 2]
输出：[1, 2]
```

提示:

1. 链表长度在[0, 20000]范围内。
2. 链表元素在[0, 20000]范围内。

进阶:

如果不得使用临时缓冲区，该怎么解决?

https://leetcode-cn.com/problems/remove-duplicate-node-lcci/solution/yi-chu-zhong-fu-jie-dian-by-leetcode-solution/

方法一：哈希表

我们对给定的链表进行一次遍历，并用一个哈希集合（HashSet）来存储所有出现过的节点。由于在大部分语言中，对给定的链表元素直接进行「相等」比较，实际上是对两个链表元素的地址（而不是值）进行比较。因此，我们在哈希集合中存储链表元素的值，方便直接使用等号进行比较。

具体地，我们从链表的头节点 $head$ 开始进行遍历，遍历的指针记为 pos。由于头节点一定不会被删除，因此我们可以枚举待移除节点的前驱节点，减少编写代码的复杂度。

> 这样枚举有什么好处？试想一下，如果我们直接枚举待移除节点，那么在将它进行移除时，我们本质上是将它的前驱节点连向后继节点。而由于题目给定的链表结构中，我们无法直接访问一个节点的前驱节点。因此，我们不如直接枚举前驱节点 `u`，那么节点本身就是 `u.next`，后继节点就是 `u.next.next`。

在遍历完成后，我们就得到了最终的答案链表。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是给定链表中节点的数目。
- 空间复杂度：$O(N)$。在最坏情况下，给定链表中每个节点都不相同，哈希表中需要存储所有的 N 个值。

方法 1：hashSet（5ms）

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode removeDuplicateNodes(ListNode head) {
        if (head == null) {
            return head;
        }
        Set<Integer> occurred = new HashSet<Integer>();
        occurred.add(head.val);
        ListNode pos = head;
        // 枚举前驱节点
        while (pos.next != null) {
            // 当前待删除节点
            ListNode cur = pos.next;
            if (occurred.add(cur.val)) {
                pos = pos.next;
            } else {
                pos.next = pos.next.next;
            }
        }
        pos.next = null;
```

```
            return head;
        }
    }
```

方法二：两重循环

考虑题目描述中的「进阶」部分，是否存在一种不使用临时缓冲区（也就是方法一中的哈希表）的方法呢？

不幸的是，在保证方法一时间复杂度 $O(N)$ 的前提下，是不存在这样的方法的。因此我们需要增加时间复杂度，使得我们可以仅使用常数的空间来完成本题。一种简单的方法是，我们在给定的链表上使用两重循环，其中第一重循环从链表的头节点开始，枚举一个保留的节点，这是因为我们保留的是「最开始出现的节点」。第二重循环从枚举的保留节点开始，到链表的末尾结束，将所有与保留节点相同的节点全部移除。

方法二的细节部分与方法一类似。第一重循环枚举保留的节点本身，而为了编码方便，第二重循环可以枚举待移除节点的前驱节点，方便我们对节点进行移除。这样一来，我们使用「时间换空间」的方法，就可以不使用临时缓冲区解决本题了。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 是给定链表中节点的数目。
- 空间复杂度：$O(1)$。

方法 2：两重循环（395ms）

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode removeDuplicateNodes(ListNode head) {
        ListNode ob = head;
        while (ob != null) {
            ListNode oc = ob;
            while (oc.next != null) {
                if (oc.next.val == ob.val) {
                    oc.next = oc.next.next;
                } else {
                    oc = oc.next;
                }
            }
            ob = ob.next;
        }
        return head;
    }
}
```

方法 3：位运算（3ms）

```
/**
```

```
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode removeDuplicateNodes(ListNode head) {
        int[] bits = new int[20000 / 32 + 1];
        ListNode cur = head;
        while (cur != null && cur.next != null) {
            bits[cur.val / 32] |= 1 << (cur.val % 32);
            if ((bits[cur.next.val / 32] & (1 << (cur.next.val % 32)))
!= 0)
                cur.next = cur.next.next;
            else
                cur = cur.next;
        }
        return head;
    }
}
```

26、移除链表元素（203）

203. 移除链表元素

难度 简单 443 收藏 分享 切换为英文 关注

删除链表中等于给定值 **val** 的所有节点。

示例:

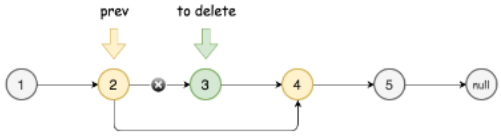
输入: 1->2->6->3->4->5->6, *val* = 6
输出: 1->2->3->4->5

https://leetcode-cn.com/problems/remove-linked-list-elements/solution/yi-chu-lian-biao-yuan-su-by-leetcode/

方法：哨兵节点

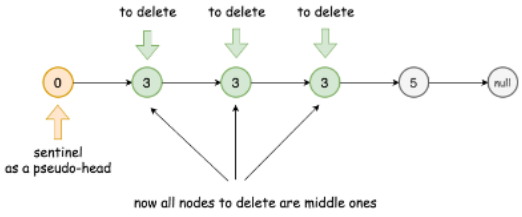
如果删除的节点是中间的节点，则问题似乎非常简单：

- 选择要删除节点的前一个结点 `prev`。
- 将 `prev` 的 `next` 设置为要删除结点的 `next`。



当要删除的一个或多个节点位于链表的头部时，事情会变得复杂。

可以通过哨兵节点去解决它，哨兵节点广泛应用于树和链表中，如伪头、伪尾、标记等，它们是纯功能的，通常不保存任何数据，其主要目的是使链表标准化，如使链表永不为空、永不无头、简化插入和删除。



在这里哨兵节点将被用于伪头。

算法：

- 初始化哨兵节点为 `ListNode(0)` 且设置 `sentinel.next = head`。
- 初始化两个指针 `curr` 和 `prev` 指向当前节点和前继节点。
- 当 `curr != nullptr`：
  - 比较当前节点和要删除的节点：
    - 若当前节点就是要删除的节点：则 `prev.next = curr.next`。
    - 否则设 `prve = curr`。
  - 遍历下一个元素：`curr = curr.next`。
- 返回 `sentinel.next`。

复杂度分析

- 时间复杂度：$\mathcal{O}(N)$，只遍历了一次。
- 空间复杂度：$\mathcal{O}(1)$。

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
```

```
 * }
 */
class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode sentinel = new ListNode(0);
        sentinel.next = head;

        ListNode prev = sentinel, curr = head;
        while (curr != null) {
        if (curr.val == val) prev.next = curr.next;
        else prev = curr;
        curr = curr.next;
        }
        return sentinel.next;
    }
}
```

删除链表的测试函数

```
import java.util.*;
public class Test {
    /*DeleteValEqualNode(面试题 18)
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ListNode head = new ListNode(4);
        ListNode node1 = new ListNode(5);
        ListNode node2 = new ListNode(1);
        ListNode node3 = new ListNode(9);
        head.next = node1;
        node1.next = node2;
        node2.next = node3;
        DeleteValEqualNode del_val = new DeleteValEqualNode();
        int val = 5;
        ListNode res1 = del_val.deleteNode(head,val);
        while (res1 != null) {
            System.out.println(res1.val);
            res1 = res1.next;
        }
    }
    */
    /*DeleteDuplicates(83)
    public static void main(String[] args) {
        DeleteDuplicates d = new DeleteDuplicates();
        ListNode head = new ListNode(1);
        ListNode node1 = new ListNode(1);
        ListNode node2 = new ListNode(2);
```

禁止商用 公众号：yes的练级攻略

```
    ListNode node3 = new ListNode(3);
    ListNode node4 = new ListNode(3);
    ListNode node5 = new ListNode(4);
    head.next = node1;
    node1.next = node2;
    node2.next = node3;
    node3.next = node4;
    node4.next = node5;
    ListNode res = d.deleteDuplicates(head);
    while (res != null) {
        System.out.println(res.val);
        res = res.next;
    }
}
*/
/*DeleteAllDuplicates(83)
public static void main(String[] args) {
    DeleteAllDuplicates d = new DeleteAllDuplicates();
    ListNode head = new ListNode(1);
    ListNode node1 = new ListNode(1);
    ListNode node2 = new ListNode(2);
    ListNode node3 = new ListNode(3);
    ListNode node4 = new ListNode(3);
    ListNode node5 = new ListNode(4);
    head.next = node1;
    node1.next = node2;
    node2.next = node3;
    node3.next = node4;
    node4.next = node5;
    ListNode res = d.deleteAllDuplicates(head);
    while (res != null) {
        System.out.println(res.val);
        res = res.next;
    }
}
*/
/*RemoveElements(203)
public static void main(String[] args) {
    RemoveElements re = new RemoveElements();
    ListNode head = new ListNode(1);
    ListNode node1 = new ListNode(2);
    ListNode node2 = new ListNode(6);
    ListNode node3 = new ListNode(3);
    ListNode node4 = new ListNode(4);
```

禁止商用 公众号：yes的练级攻略

```
        ListNode node5 = new ListNode(5);
        ListNode node6 = new ListNode(6);
        head.next = node1;
        node1.next = node2;
        node2.next = node3;
        node3.next = node4;
        node4.next = node5;
        node5.next = node6;
        int val = 6;
        ListNode res = re.removeElements(head,val);
        while (res != null) {
            System.out.println(res.val);
            res = res.next;
        }
    }
    */
}
class ListNode {
    public int val;
    public ListNode next;
    public ListNode(int x) {
        this.val = x;
        this.next = null;
    }
}
```

27、合并两个有序链表（21）

剑指 Offer 25. 合并两个排序的链表

难度 简单 39 收藏 分享 切换为英文 关注 反馈

输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。

示例1：

输入： 1->2->4, 1->3->4
输出： 1->1->2->3->4->4

限制：

0 <= 链表长度 <= 1000

https://leetcode-cn.com/problems/merge-two-sorted-lists/solution/he-bing-liang-ge-you-xu-lian-biao-by-leetcode-solu/

禁止商用 公众号：

方法二：迭代

思路

我们可以用迭代的方法来实现上述算法。当 `l1` 和 `l2` 都不是空链表时，判断 `l1` 和 `l2` 哪一个链表的头节点的值更小，将较小值的节点添加到结果里，当一个节点被添加到结果里之后，将对应链表中的节点向后移一位。

算法

首先，我们设定一个哨兵节点 `prehead` ，这可以在最后让我们比较容易地返回合并后的链表。我们维护一个 `prev` 指针，我们需要做的是调整它的 `next` 指针。然后，我们重复以下过程，直到 `l1` 或者 `l2` 指向了 `null` ：如果 `l1` 当前节点的值小于等于 `l2` ，我们就把 `l1` 当前的节点接在 `prev` 节点的后面同时将 `l1` 指针往后移一位。否则，我们对 `l2` 做同样的操作。不管我们将哪一个元素接在了后面，我们都需要把 `prev` 向后移一位。

在循环终止的时候， `l1` 和 `l2` 至多有一个是非空的。由于输入的两个链表都是有序的，所以不管哪个链表是非空的，它包含的所有元素都比前面已经合并链表中的所有元素都要大。这意味着我们只需要简单地将非空链表接在合并链表的后面，并返回合并链表即可。

下图展示了 `1->4->5` 和 `1->2->3->6` 两个链表迭代合并的过程：

复杂度分析

- 时间复杂度：$O(n+m)$，其中 n 和 m 分别为两个链表的长度。因为每次循环迭代中， `l1` 和 `l2` 只有一个元素会被放进合并链表中， 因此 `while` 循环的次数不会超过两个链表的长度之和。所有其他操作的时间复杂度都是常数级别的，因此总的时间复杂度为 $O(n+m)$。
- 空间复杂度：$O(1)$。我们只需要常数的空间存放若干变量。

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1);
        ListNode pre = dummy;
        while (l1 != null && l2 !=null) {
            if (l1.val < l2.val) {
                pre.next = l1;
                l1 = l1.next;
            } else {
                pre.next = l2;
                l2 = l2.next;
            }
            pre = pre.next;
        }
        pre.next = l1==null ? l2 : l1;
        return dummy.next;
    }
}
```

28、合并 K 个排序链表（23）

更多硬核内容就在公众号「yes 的练级攻略」

23. 合并K个排序链表

难度 困难 717 收藏 分享 切换为英文 关注 反馈

合并 *k* 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

示例:

```
输入:
[
  1->4->5,
  1->3->4,
  2->6
]
输出: 1->1->2->3->4->4->5->6
```

https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/4-chong-fang-fa-xiang-jie-bi-xu-miao-dong-by-sweet/

一、K 指针：K 个指针分别指向 K 条链表

1. 每次 $O(K)$ 比较 K个指针求 min, 时间复杂度：$O(NK)$

2. 使用小根堆对 1 进行优化，每次 $O(logK)$ 比较 K个指针求 min, 时间复杂度：$O(NlogK)$

二、 逐一合并两条链表

首先复习一下 「21. 合并两个有序链表」
下面分别贴出「merge2Lists」的「递归」 和 「迭代」两种实现：

合并两条有序链表 — 递归

合并两条有序链表 — 迭代

1. 逐一合并两条链表, 时间复杂度：$O(NK)$

2. 两两合并对 1 进行优化，时间复杂度：$O(NlogK)$

时间复杂度分析：K 条链表的总结点数是 N，平均每条链表有 N/K 个节点，因此合并两条链表的时间复杂度是 $O(N/K)$。从 K 条链表开始两两合并成 1 条链表，因此每条链表都会被合并 $logK$ 次，因此 K 条链表会被合并 $K*logK$ 次，因此总共的时间复杂度是 $K*logK*N/K$ 即 $O(NlogK)$。

下面分别贴出「两两合并」的「递归」 和 「迭代」两种实现：

两两合并 - 迭代

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    /*  两两合并对 1 进行优化-迭代法，算法复杂度 o(NlogK)
```

```
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists.length == 0) {
            return null;
        }
        int k = lists.length;
        while (k > 1) {
            int idx = 0;
            for (int i = 0; i < k; i += 2) {
                if (i == k-1) {
                    lists[idx++] = lists[i];
                } else {
                    lists[idx++] = merge2Lists(lists[i], lists[i + 1]);
                }
            }
            k = idx;
        }
        return lists[0];

    }
    private ListNode merge2Lists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(-1);
    ListNode pre = dummy;
    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            pre.next = l1;
            l1 = l1.next;
        } else {
            pre.next = l2;
            l2 = l2.next;
        }
        pre = pre.next;
    }
    pre.next = l1 == null? l2: l1;
    return dummy.next;
    }
    */
    //使用小根堆对 1 进行优化，每次 O(logK) 比较 K 个指针求 min，时间复杂度：
O(NlogK)
    public ListNode mergeKLists(ListNode[] lists) {
        Queue<ListNode> pq = new PriorityQueue<>((v1, v2) -> v1.val - v
2.val);
        for (ListNode node: lists) {
            if (node != null) {
                pq.offer(node);
```

```
            }
        }

        ListNode dummyHead = new ListNode(0);
        ListNode tail = dummyHead;
        while (!pq.isEmpty()) {
            ListNode minNode = pq.poll();
            tail.next = minNode;
            tail = minNode;
            if (minNode.next != null) {
                pq.offer(minNode.next);
            }
        }

        return dummyHead.next;
    }

}
```

29、分隔链表（86）【分割链表（面试题 02.04）】【所有小于 x 的节点都在大于或等于 x 的节点之前】

86. 分隔链表

难度 中等 240 收藏 分享 切换为英文 关注 反馈

给定一个链表和一个特定值 *x*，对链表进行分隔，使得所有小于 *x* 的节点都在大于或等于 *x* 的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

示例:

```
输入: head = 1->4->3->2->5->2, x = 3
输出: 1->2->2->4->3->5
```

面试题 02.04. 分割链表

难度 中等 26 收藏 分享 切换为英文 关注 反馈

编写程序以 x 为基准分割链表，使得所有小于 x 的节点排在大于或等于 x 的节点之前。如果链表中包含 x，x 只需出现在小于 x 的元素之后(如下所示)。分割元素 x 只需处于“右半部分”即可，其不需要被置于左右两部分之间。

示例:

```
输入: head = 3->5->8->5->10->2->1, x = 5
输出: 3->1->2->10->5->5->8
```

https://leetcode-cn.com/problems/partition-list/solution/liang-ge-zi-lian-biao-pin-jie-han-zhu-shi-ji-bai-1/

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode partition(ListNode head, int x) {
        //链表 less 记录小于 x 的元素
        ListNode less = new ListNode(0);
        ListNode ptr1 = less;
        //链表 greatOrEqual 记录大于等于 x 的元素
        ListNode greatOrEqual = new ListNode(0);
        ListNode ptr2 = greatOrEqual;
        while (head != null) {
            if (head.val < x) {
                ptr1.next = new ListNode(head.val);
                ptr1 = ptr1.next;
            } else {
                ptr2.next = new ListNode(head.val);
                ptr2 = ptr2.next;
            }
            head = head.next;
        }
        //拼接两段链表
        ptr1.next = greatOrEqual.next;
        return less.next;
    }
}
```

30、分隔链表（725）【分隔为 k 个连续部分】

更多硬核内容就在公众号「yes 的练级攻略」

725. 分隔链表

难度 中等 88 收藏 分享 切换为英文 关注 反馈

给定一个头结点为 `root` 的链表, 编写一个函数以将链表分隔为 `k` 个连续的部分。

每部分的长度应该尽可能的相等: 任意两部分的长度差距不能超过 1，也就是说可能有些部分为 null。

这k个部分应该按照在链表中出现的顺序进行输出，并且排在前面的部分的长度应该大于或等于后面的长度。

返回一个符合上述规则的链表的列表。

举例： 1->2->3->4, k = 5 // 5 结果 [[1], [2], [3], [4], null]

示例 1：

输入：
root = [1, 2, 3], k = 5
输出： [[1],[2],[3],[],[]]
解释：
输入输出各部分都应该是链表，而不是数组。
例如，输入的结点 root 的 val= 1, root.next.val = 2, \root.next.next.val = 3, 且 root.next.next.next = null。
第一个输出 output[0] 是 output[0].val = 1, output[0].next = null。
最后一个元素 output[4] 为 null，它代表了最后一个部分为空链表。

示例 2：

输入：
root = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], k = 3
输出： [[1, 2, 3, 4], [5, 6, 7], [8, 9, 10]]
解释：
输入被分成了几个连续的部分，并且每部分的长度相差不超过1.前面部分的长度大于等于后面部分的长度。

提示：

- `root` 的长度范围： `[0, 1000]`.
- 输入的每个节点的大小范围： `[0, 999]`.
- `k` 的取值范围： `[1, 50]`.

yes的练级攻略

https://leetcode-cn.com/problems/split-linked-list-in-parts/solution/jie-jue-lian-biao-fen-ge-wen-ti-javajie-fa-by-erne/

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode[] splitListToParts(ListNode root, int k) {
        int n = 0;
        ListNode cur = root;
        //确定链表共有多少个节点
        while (cur != null) {
            n++;
            cur = cur.next;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }
        //分割之后链表的节点个数，前 mod 个链表的节点个数为 size+1，之后的为 size
        //若 n < k,则前面每个链表为一个元素，后面的为空
        int mod = n % k, size = n / k;
        ListNode[] res = new ListNode[k];
        cur = root;
         //给 ListNode[]数组的每个链表元素循环赋值
         for (int i = 0; i < k && cur != null; i++) {
             res[i] = cur;
             int curSize = size;
             if (mod > 0) {
                 curSize++;
                 mod--;
             }
             for (int j = 0; j < curSize - 1; j++) {
                 cur = cur.next;
             }
            //ListNode[]数组里的每个链表元素结尾要指向 null;
             ListNode next = cur.next;
             cur.next = null;
             cur = next;
         }
         return res;
    }
}
```

31、排序链表（148）【要求 O(n log n) 时间复杂度和常数级空间复杂度】

148. 排序链表

难度 中等 601 收藏 分享 切换为英文 关注 反馈

在 $O(n \log n)$ 时间复杂度和常数级空间复杂度下，对链表进行排序。

示例 1:

```
输入: 4->2->1->3
输出: 1->2->3->4
```

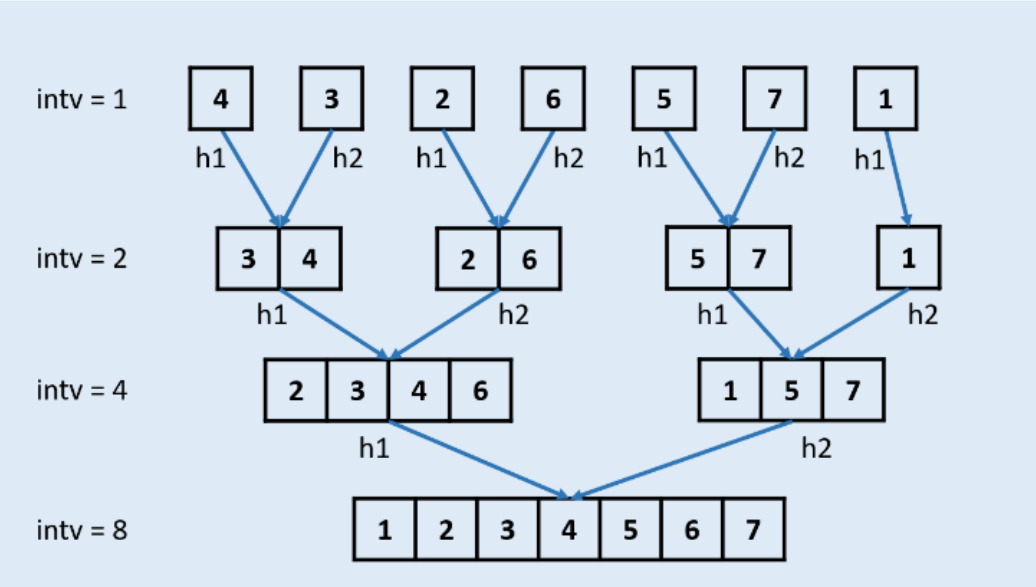
示例 2:

```
输入: -1->5->3->4->0
输出: -1->0->3->4->5
```

https://leetcode-cn.com/problems/sort-list/solution/sort-list-gui-bing-pai-xu-lian-biao-by-jyd/

解答二：归并排序（从底至顶直接合并）

- 对于非递归的归并排序，需要使用迭代的方式替换 `cut` 环节：
  - 我们知道， `cut` 环节本质上是通过二分法得到链表最小节点单元，再通过多轮合并得到排序结果。
  - 每一轮合并 `merge` 操作针对的单元都有固定长度 `intv` ，例如：
    - 第一轮合并时 `intv = 1` ，即将整个链表切分为多个长度为1的单元，并按顺序两两排序合并，合并完成的已排序单元长度为2。
    - 第二轮合并时 `intv = 2` ，即将整个链表切分为多个长度为2的单元，并按顺序两两排序合并，合并完成已排序单元长度为4。
    - 以此类推，直到单元长度 `intv >= 链表长度` ，代表已经排序完成。
  - 根据以上推论，我们可以仅根据 `intv` 计算每个单元边界，并完成链表的每轮排序合并，例如：
    - 当 `intv = 1` 时，将链表第 `1` 和第 `2` 节点排序合并，第 `3` 和第 `4` 节点排序合并，……。
    - 当 `intv = 2` 时，将链表第 `1-2` 和第 `3-4` 节点排序合并，第 `5-6` 和第 `7-8` 节点排序合并，……。
    - 当 `intv = 4` 时，将链表第 `1-4` 和第 `5-8` 节点排序合并，第 `9-12` 和第 `13-16` 节点排序合并，……。
- 此方法时间复杂度$O(nlogn)$，空间复杂度$O(1)$。



- 模拟上述的多轮排序合并：
  - 统计链表长度 `length` ，用于通过判断 `intv < length` 判定是否完成排序；
  - 额外声明一个节点 `res` ，作为头部后面接整个链表，用于：
    - `intv *= 2` 即切换到下一轮合并时，可通过 `res.next` 找到链表头部 `h` ；
    - 执行排序合并时，需要一个辅助节点作为头部，而 `res` 则作为链表头部排序合并时的辅助头部 `pre` ；后面的合并排序可以将上次合并排序的尾部 `tail` 用做辅助节点。
  - 在每轮 `intv` 下的合并流程：
    1. 根据 `intv` 找到合并单元1和单元2的头部 `h1` 、 `h2` 。由于链表长度可能不是 `2^n` ，需要考虑边界条件：
       - 在找 `h2` 过程中，如果链表剩余元素个数少于 `intv` ，则无需合并环节，直接 `break` ，执行下一轮合并；
       - 若 `h2` 存在，但以 `h2` 为头部的剩余元素个数少于 `intv` ，也执行合并环节， `h2` 单元的长度为 `c2 = intv - i` 。
    2. 合并长度为 `c1, c2` 的 `h1, h2` 链表，其中：
       - 合并完后，需要修改新的合并单元的尾部 `pre` 指针指向下一个合并单元头部 `h` 。（在寻找 `h1, h2` 环节中，h指针已经被移动到下一个单元头部）
       - 合并单元尾部同时也作为下次合并的辅助头部 `pre` 。
    3. 当 `h == None` ，代表此轮 `intv` 合并完成，跳出。
  - 每轮合并完成后将单元长度×2，切换到下轮合并： `intv *= 2` 。

方法 1：常数级空间复杂度一定要用非递归

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode sortList(ListNode head) {
        if(head == null || head.next == null) return head;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        int len = getListLen(head);
        int itrv = 1;
        while(itrv < len) {
            ListNode pre = dummy;
            ListNode h = dummy.next;
            // 找到合并链表的 h1 和 h2 头节点
            while(h!=null) {
                int i = itrv;
                ListNode h1 = h;
                for(; h != null && i > 0; i--) {
                    h = h.next;
                }
                // i>0 说明没有链表 2 直接返回
                if(i > 0) break;
                ListNode h2 = h;
                i = itrv;
                for(; h != null && i > 0; i--) {
                    h = h.next;
                }
                // 求出两个链表的长度
                int c1 = itrv;
                int c2 = itrv - i;

                //合并
                while(c1 > 0 && c2 > 0) {
                    if(h1.val < h2.val) {
                        pre.next = h1;
                        h1 = h1.next;
                        c1--;
                    }else {
```

```
                    pre.next = h2;
                    h2 = h2.next;
                    c2--;
                }
                pre = pre.next;
            }
            pre.next = c1 > 0 ? h1 : h2;
            while(c1 > 0 || c2 > 0) {
                pre = pre.next;
                c1--;
                c2--;
            }
            pre.next = h;
        }
        itrv*=2;
    }
    return dummy.next;
}

private int getListLen(ListNode head) {
    ListNode cur = head;
    int len = 0;
    while(cur != null) {
        len++;
        cur = cur.next;
    }
    return len;
}
}
```

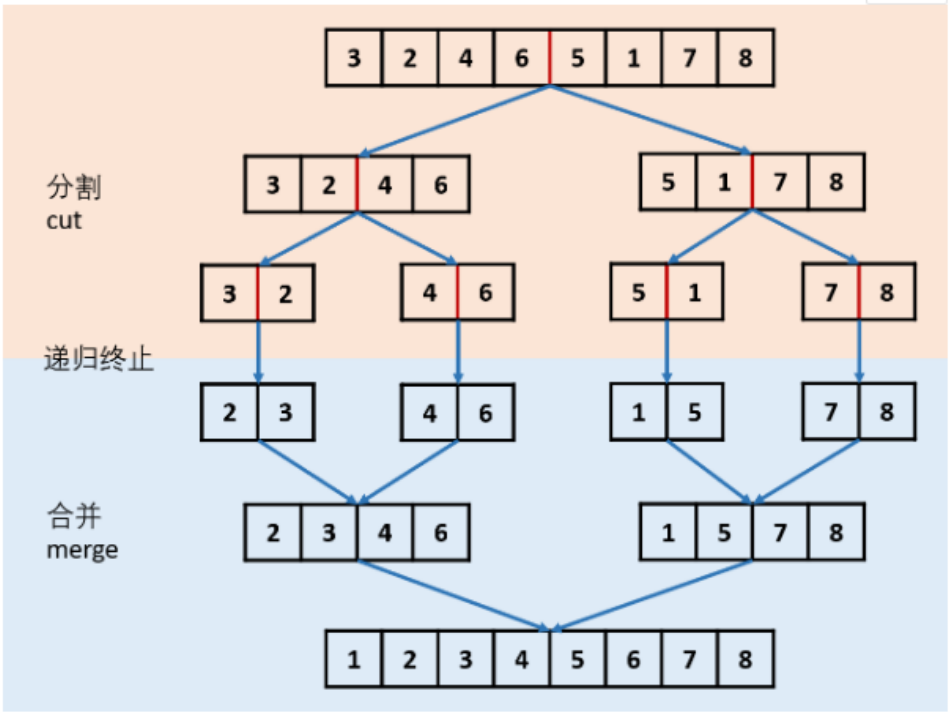
https://leetcode-cn.com/problems/sort-list/solution/sort-list-gui-bing-pai-xu-lian-biao-by-jyd/

解答一：归并排序（递归法）

- 题目要求时间空间复杂度分别为$O(nlogn)$和$O(1)$，根据时间复杂度我们自然想到二分法，从而联想到归并排序；
- 对数组做归并排序的空间复杂度为 $O(n)$，分别由新开辟数组$O(n)$和递归函数调用$O(logn)$组成，而根据链表特性：
  - 数组额外空间：链表可以通过修改引用来更改节点顺序，无需像数组一样开辟额外空间；
  - 递归额外空间：递归调用函数将带来$O(logn)$的空间复杂度，因此若希望达到$O(1)$空间复杂度，则不能使用递归。
- 通过递归实现链表归并排序，有以下两个环节：
  - **分割 cut 环节：** 找到当前链表 `中点` ，并从 `中点` 将链表断开（以便在下次递归 `cut` 时，链表片段拥有正确边界）；
    - 我们使用 `fast,slow` 快慢双指针法，奇数个节点找到中点，偶数个节点找到中心左边的节点。
    - 找到中点 `slow` 后，执行 `slow.next = None` 将链表切断。
    - 递归分割时，输入当前链表左端点 `head` 和中心节点 `slow` 的下一个节点 `tmp`(因为链表是从 `slow` 切断的)。
    - **cut 递归终止条件：** 当 `head.next == None` 时，说明只有一个节点了，直接返回此节点。
  - **合并 merge 环节：** 将两个排序链表合并，转化为一个排序链表。
    - 双指针法合并，建立辅助ListNode `h` 作为头部。
    - 设置两指针 `left` , `right` 分别指向两链表头部，比较两指针处节点值大小，由小到大加入合并链表头部，指针交替前进，直至添加完两个链表。
    - 返回辅助ListNode `h` 作为头部的下个节点 `h.next` 。
    - 时间复杂度 `O(l + r)` ， `l, r` 分别代表两个链表长度。
  - 当题目输入的 `head == None` 时，直接返回None。

- 当题目输入的 `head == None` 时，直接返回None。

☆收藏



方法 2：如果不要求常数级的空间复杂度，可以使用递归

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
```

```
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode fast = head.next, slow = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode tmp = slow.next;
        slow.next = null;
        ListNode left = sortList(head);
        ListNode right = sortList(tmp);
        ListNode h = new ListNode(0);
        ListNode res = h;
        while (left != null && right != null) {
            if (left.val < right.val) {
                h.next = left;
                left = left.next;
            } else {
                h.next = right;
                right = right.next;
            }
            h = h.next;
        }
        h.next = left != null ? left : right;
        return res.next;
    }
}
```

32、复制带随机指针的链表（138）【复杂链表的复制（剑指 35）】

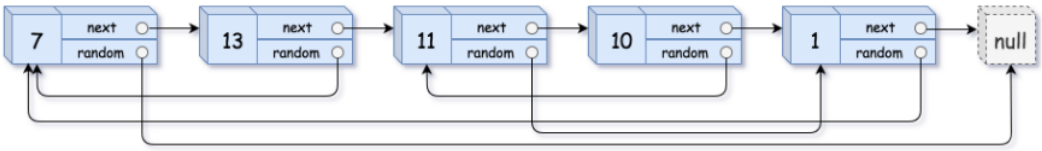
更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 35. 复杂链表的复制

难度 中等 78 收藏 分享 切换为英文 关注 反馈

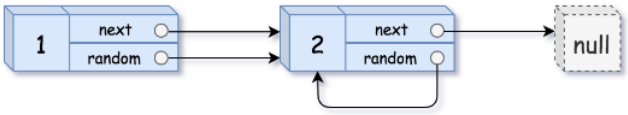
请实现 `copyRandomList` 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 `next` 指针指向下一个节点，还有一个 `random` 指针指向链表中的任意节点或者 `null`。

示例 1：



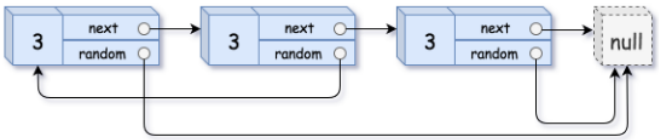

```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

示例 2：




```
输入：head = [[1,1],[2,1]]
输出：[[1,1],[2,1]]
```

示例 3：




```
输入：head = [[3,null],[3,0],[3,null]]
输出：[[3,null],[3,0],[3,null]]
```

示例 4：

```
输入：head = []
输出：[]
解释：给定的链表为空（空指针），因此返回 null。
```

提示：

- `-10000 <= Node.val <= 10000`
- `Node.random` 为空（null）或指向链表中的节点。
- 节点数目不超过 1000 。

注意：本题与主站 138 题相同：https://leetcode-cn.com/problems/copy-list-with-random-pointer/

https://leetcode-cn.com/problems/copy-list-with-random-pointer/solution/liang-chong-shi-xian-tu-jie-138-fu-zhi-dai-sui-ji-/

解法一

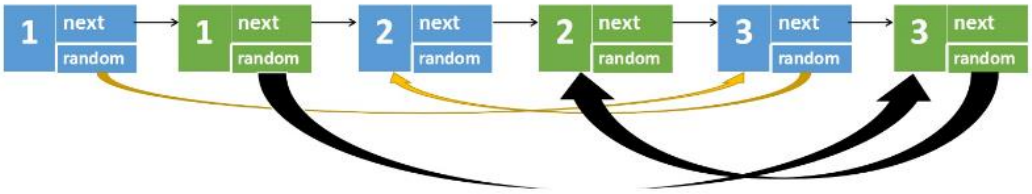
这题的最大难点就在于**复制随机指针**，比如下图中

- 节点1的随机指针指向节点3
- 节点3的随机指针指向节点2
- 节点2的随机指针指向空

我们可以用三步走来搞定这个问题

第一步，根据遍历到的原节点创建对应的新节点，每个新创建的节点是在原节点后面，比如下图中原节点1不再指向原原节点2，而是指向新节点1

第二步是最关键的一步，用来设置新链表的随机指针



上图中，我们可以观察到这么一个规律

- 原节点1的随机指针指向原节点3，新节点1的随机指针指向的是原节点3的next
- 原节点3的随机指针指向原节点2，新节点3的随机指针指向的是原节点2的next

也就是，原节点 `i` 的随机指针(如果有的话)，指向的是原节点 `j`
那么新节点 `i` 的随机指针，指向的是原节点 `j` 的**next**

第三步就简单了，只要将两个链表分离开，再返回新链表就可以了

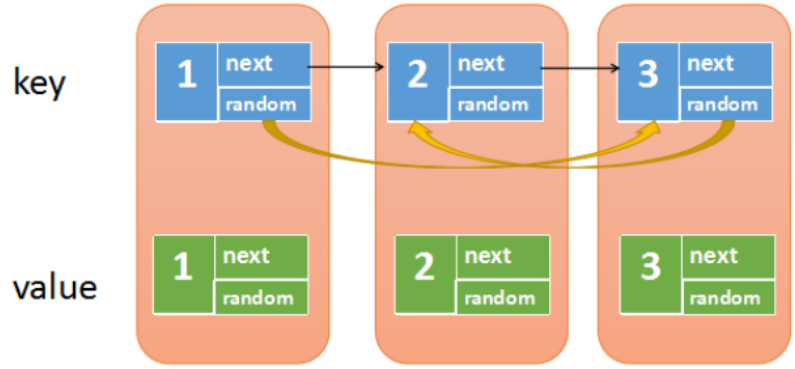
方法 1：分别拷贝链表、指定随机指针、重连链表

```
/*
// Definition for a Node.
class Node {
    int val;
    Node next;
    Node random;

    public Node(int val) {
        this.val = val;
```

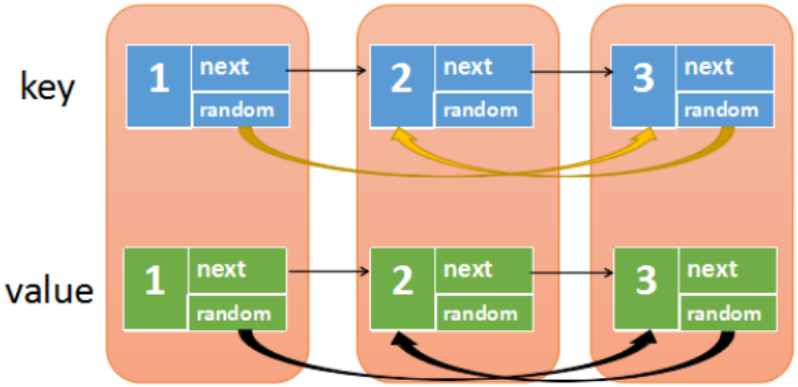
```
        this.next = null;
        this.random = null;
    }
}
*/
class Solution {
    public Node copyRandomList(Node head) {
        if(head==null) {
            return null;
        }
        Node p = head;
        //第一步，在每个原节点后面创建一个新节点
        //1->1'->2->2'->3->3'
        while(p!=null) {
            Node newNode = new Node(p.val);
            newNode.next = p.next;
            p.next = newNode;
            p = newNode.next;
        }
        p = head;
        //第二步，设置新节点的随机节点
        while(p!=null) {
            if(p.random!=null) {
                p.next.random = p.random.next;
            }
            p = p.next.next;
        }
        Node dummy = new Node(-1);
        p = head;
        Node cur = dummy;
        //第三步，将两个链表分离
        while(p!=null) {
            cur.next = p.next;
            cur = cur.next;
            p.next = cur.next;
            p = p.next;
        }
        return dummy.next;
    }
}
```

解法二

我们用哈希表来解决这个问题
首先创建一个哈希表，再遍历原链表，遍历的同时再不断创建新节点
我们将原节点作为**key**，新节点作为**value**放入哈希表中



第二步我们再遍历原链表，这次我们要将新链表的next和random指针给设置上



从上图中我们可以发现，原节点和新节点是一一对应的关系，所以

- map.get(原节点)，得到的就是对应的新节点
- map.get(原节点.next)，得到的就是对应的新节点.next
- map.get(原节点.random)，得到的就是对应的新节点.random

所以，我们只需要再次遍历原链表，然后设置：
新节点.next -> map.get(原节点.next)
新节点.random -> map.get(原节点.random)
这样新链表的next和random都被串联起来了
最后，我们然后 `map.get(head)` ，也就是对应的新链表的头节点，就可以解决此问题了。

方法 2：利用 HashMap

```
class Solution {
    public Node copyRandomList(Node head) {
        if(head==null) {
            return null;
        }
        //创建一个哈希表，key 是原节点，value 是新节点
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        Map<Node,Node> map = new HashMap<Node,Node>();
        Node p = head;
        //将原节点和新节点放入哈希表中
        while(p!=null) {
            Node newNode = new Node(p.val);
            map.put(p,newNode);
            p = p.next;
        }
        p = head;
        //遍历原链表，设置新节点的next 和random
        while(p!=null) {
            Node newNode = map.get(p);
            //p 是原节点，map.get(p)是对应的新节点，p.next 是原节点的下一个
            //map.get(p.next)是原节点下一个对应的新节点
            if(p.next!=null) {
                newNode.next = map.get(p.next);
            }
            //p.random 是原节点随机指向
            //map.get(p.random)是原节点随机指向  对应的新节点
            if(p.random!=null) {
                newNode.random = map.get(p.random);
            }
            p = p.next;
        }
        //返回头结点，即原节点对应的value(新节点)
        return map.get(head);
    }
}
```

33、设计跳表（1206）

更多硬核内容就在公众号「yes 的练级攻略」

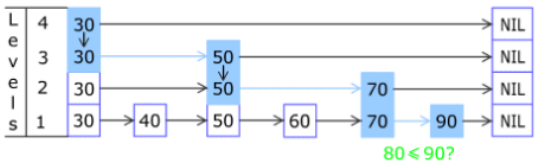
1206. 设计跳表

难度 困难 32 收藏 分享 切换为英文 关注 反馈

不使用任何库函数，设计一个跳表。

跳表是在 O(log(n)) 时间内完成增加、删除、搜索操作的数据结构。跳表相比于树堆与红黑树，其功能与性能相当，并且跳表的代码长度相较下更短，其设计思想与链表相似。

例如，一个跳表包含 [30, 40, 50, 60, 70, 90]，然后增加 80、45 到跳表中，以下图的方式操作：



Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

跳表中有很多层，每一层是一个短的链表。在第一层的作用下，增加、删除和搜索操作的时间复杂度不超过 O(n)。跳表的每一个操作的平均时间复杂度是 O(log(n))，空间复杂度是 O(n)。

在本题中，你的设计应该要包含这些函数：

- `bool search(int target)` : 返回target是否存在于跳表中。
- `void add(int num)`: 插入一个元素到跳表。
- `bool erase(int num)`: 在跳表中删除一个值，如果 `num` 不存在，直接返回false. 如果存在多个 `num` ，删除其中任意一个即可。

了解更多：https://en.wikipedia.org/wiki/Skip_list

注意，跳表中可能存在多个相同的值，你的代码需要处理这种情况。

样例：

```
Skiplist skiplist = new Skiplist();

skiplist.add(1);
skiplist.add(2);
skiplist.add(3);
skiplist.search(0);   // 返回 false
skiplist.add(4);
skiplist.search(1);   // 返回 true
skiplist.erase(0);    // 返回 false，0 不在跳表中
skiplist.erase(1);    // 返回 true
skiplist.search(1);   // 返回 false，1 已被擦除
```

约束条件：

- `0 <= num, target <= 20000`
- 最多调用 `50000` 次 `search`, `add`, 以及 `erase` 操作。

https://leetcode-cn.com/problems/design-skiplist/solution/ti-mu-fen-xi-java-shi-xian-by-pravez/

跳表其实就是加了几层索引的链表，一共有 N 层，以 0 ~ N-1 层表示，设第 0 层是原链表，抽取其中部分元素，在第 1 层形成新的链表，上下层的相同元素之间连起来；再抽取第 1 层部分元素，构成第 2 层，以此类推。
具体选哪些元素呢？题目给的是coinFlip，也就是每次插入元素的时候，都进行一次 50% 概率的判断，如果 true 则向上层添加一个索引，如果 false 就不加了。

【查找】

在查找的时候，从最上层开始找，因为最顶层元素最少，比较次数也少，如果在较上层找到了，直接返回 true；如果在较上层比较时发现了更大的元素，或者到头了，说明再往后是找不见的，直接向下层跳即可（所以这里需要记录访问的前一个元素是什么以进行向下的跳跃，可以多加一个元素在循环的时候记录前次访问的元素，或者使用双链表实现）。直到跳到最后一层还是没到，那就是找不到了，返回 false 。

【添加和删除】

添加和删除的时候，存在**数据重复**的问题，经过我的测试，发现题目要求的是，将重复的数据当做不同的值来对待，即存了一个元素 9，再存一个 9，此时表里有俩 9，相互独立，删除一个 9，表里应该还剩余有一个9。

由于添加和删除元素都是从底层开始，而在查找的时候是从顶层开始查找的，因此可以使用一个数组记录每层的"跳跃节点"的位置，就不必反复的从顶层开始查找每层的位置了。

在添加的时候，首先是查找到添加位置，过程与查找类似，首先找到表中 >= 插入元素 的最小节点位置，然后插入节点，50% 概率判别，如果需要添加索引，就添加索引，继续 50% 概率判别，直到结束。

在删除的时候，首先也是查找，找到表中所有 = 插入元素的节点位置（每层只找一个，找到直接跳层即可），然后挨个删除。

```
import java.util.ArrayList;
import java.util.List;
public class Skiplist {
    private List<Node> heads;
    // 使用 ArrayList 保存每层链表的表头，表头不存储元素，值为 -i ， i 为层数，从 1 开始。
    private class Node{
        int val;
        // 使用双链表方便操作
        Node left,right,down;

        public Node(int val) {
            this.val = val;
            this.left = null;
            this.down = null;
            this.right = null;
        }
    }
    public Skiplist() {
        Node head0 = new Node(-1);
        heads = new ArrayList<>();
        heads.add(head0);
    }
    public boolean search(int target){
        int level = heads.size()-1;
        Node searchingStart = heads.get(level);
        while(level != -1){
            boolean jump = false;
            while(searchingStart.right!=null){
                searchingStart = searchingStart.right;
                if(searchingStart.val==target)
                    return true;
                else if(searchingStart.val>target){
                    if(level == 0)
                        return false;
                    else {
                        level --;
                        searchingStart = searchingStart.left.down;
                        jump = true;
                        break;
                    }
                }
            }
            if(jump)
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                continue;
            level --;
            searchingStart = searchingStart.down;
        }
        return false;
    }
    public void add(int num){
        // 在底层添加元素
        Node n0 = new Node( num);
        Node[] jumpingNodes = searchNodes(num);
        Node lefter = jumpingNodes[0];
        n0.right = lefter.right;
        n0.left = lefter;
        if(lefter.right!=null)
            lefter.right.left=n0;
        lefter.right = n0;

        // 添加索引
        int levelToAddIndex = 1;
        while(throwCoin()){
            Node hi;
            if(jumpingNodes.length <= levelToAddIndex){
                hi= new Node(-levelToAddIndex-1);
                heads.add(hi);
                hi.down = heads.get(levelToAddIndex-1);
            }
            else
                hi = jumpingNodes[levelToAddIndex];
            Node ni = new Node(num);
            // 连接上下层相同的元素，用于跳
            ni.down = n0;
            ni.right = hi.right;
            ni.left = hi;
            if(hi.right!=null)
                hi.right.left = ni;
            hi.right = ni;
            n0 = ni;
            levelToAddIndex++;
        }
    }
    public boolean erase(int num) {
        Node[] jmpt = searchNodes(num);
        boolean exist = false;
        for (Node n:jmpt)
```

```
            if(n.val==num){
                exist = true;
                del(n);
            }
        return exist;
    }
    private Node[] searchNodes(int target){
        // 本函数也能实现 public boolean search(int target)的功能，但是由于是直到底层才停止的，效果不如在中间层找到就直接返回 true 的好。
        Node[] jumps = new Node[heads.size()];
        int level = heads.size()-1;
        Node searchingStart = heads.get(level);
        while(level!=-1){
            boolean jump = false;
            while(searchingStart.right!=null){
                searchingStart = searchingStart.right;
                if(searchingStart.val==target) {
                    while(level!=-1){
                        jumps[level] = searchingStart;
                        searchingStart=searchingStart.down;
                        level --;
                    }
                    return jumps;
                }
                else if(searchingStart.val>target){
                    jumps[level] = searchingStart.left;
                    level --;
                    searchingStart = searchingStart.left.down;
                    jump = true;
                    break;
                }
            }
            if(jump)
                continue;
            jumps[level] = searchingStart;
            level --;
            searchingStart = searchingStart.down;
        }
        return jumps;
    }
    private void del(Node n){
        n.left.right = n.right;
        if(n.right != null ) {
            n.right.left = n.left;
```

```
        }
    }
    private boolean throwCoin(){
        // 抛硬币， 50% 概率
        return Math.random()<0.5f;
    }

}
/**
 * Your Skiplist object will be instantiated and called as such:
 * Skiplist obj = new Skiplist();
 * boolean param_1 = obj.search(target);
 * obj.add(num);
 * boolean param_3 = obj.erase(num);
 */
```

34、特定深度节点链表（面试题 04.03）

面试题 04.03. 特定深度节点链表

难度 中等 20 收藏 分享 切换为英文 关注 反馈

给定一棵二叉树，设计一个算法，创建含有某一深度上所有节点的链表（比如，若一棵树的深度为 D，则会创建出 D 个链表）。返回一个包含所有深度的链表的数组。

示例：

输入： [1,2,3,4,5,null,7,8]

```
       1
      /  \
     2    3
    / \    \
   4   5    7
  /
 8
```

输出： [[1],[2,3],[4,5,7],[8]]

https://leetcode-cn.com/problems/list-of-depth-lcci/solution/java-bfsdfs-liang-chong-jie-fa-by-lixianfeng/

DFS

解题思路：

- 在前序遍历过程中，传入level，表示当前节点所在的层值
- 当level >= listNode.size()时，则需要创建一个链表头节点
- 定义一个tmp指针，找到每层链表的最后的一个节点，并往最后一个节点加入新的节点
- 我们在刚开始每一层链表创建了一个值为0的头节点，返回时需要获取每一层链表头节点的下一个节点

方法 1：DFS

```
/**
```

```java
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode[] listOfDepth(TreeNode tree) {
        List<ListNode> listNode = new ArrayList<>();
        preorder(tree, listNode, 0);
        ListNode[] res = new ListNode[listNode.size()];
        for(int i=0; i<listNode.size(); i++){
            res[i] = listNode.get(i).next;
        }
        return res;
    }
    public void preorder(TreeNode root, List<ListNode> listNode, int level){
        if(root != null){
            // 关键部分代码
            if(level >= listNode.size()){
                listNode.add(new ListNode(0));
            }
            ListNode tmp = listNode.get(level);
            while(tmp.next != null){
                tmp = tmp.next;
            }
            tmp.next = new ListNode(root.val);

            if(root.left != null){
                preorder(root.left, listNode, level+1);
            }
            if(root.right != null){
                preorder(root.right, listNode, level+1);
            }
```

```
        }
    }
}
```

BFS

解题思路：

- 层序遍历

方法 2：BFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode[] listOfDepth(TreeNode tree) {
        if(tree == null) return new ListNode[]{};
        List<ListNode> list = new ArrayList<>();
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(tree);
        while(!queue.isEmpty()){
            int size = queue.size();
            ListNode point = new ListNode(0);
            ListNode tmp = point;
            for(int i=0; i<size; i++){
                TreeNode node = queue.poll();
                tmp.next = new ListNode(node.val);
                tmp = tmp.next;
                if(node.left != null){
                    queue.add(node.left);
                }
                if(node.right != null){
```

```
                queue.add(node.right);
            }
        }
        list.add(point.next);
    }
    ListNode[] res = new ListNode[list.size()];
    for(int i=0; i<list.size(); i++){
        res[i] = list.get(i);
    }
    return res;
  }
}
```

十六、二叉树

【定义 TreeNode】

```
class TreeNode{
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(){}
    TreeNode(int x){
        this.val = x;
    }
    TreeNode(int x, TreeNode left, TreeNode right){
        this.val = x;
        this.left = left;
        this.right = right;
    }
}
```

【生成 TreeNode】将包含前后[]的 String 转成 TreeNode。

方法 1：

其中 main 函数获取字符串仅需要两行

```
Scanner sc = new Scanner(System.in);
String str = sc.nextLine();

public static TreeNode deserialize(String data) {
    if(data.length() < 3) return null;
    String[] strs = data.substring(1, data.length()-1).split(","); //因为头尾有括号
//String[] strs = data.split(",");//没有括号时候可以这么写
    TreeNode root = new TreeNode(Integer.valueOf(strs[0]));
    Deque<TreeNode> que = new LinkedList<>();
    que.add(root);
    TreeNode cur;
    int idx = 0;
    while(true){
```

```
            cur=que.remove();
            if(++idx >= strs.length) break;
            if(strs[idx].compareTo("null") != 0) {
                cur.left = new TreeNode(Integer.valueOf(strs[idx]));
                que.add(cur.left);
            }
            if(++idx >= strs.length) break;
            if(strs[idx].compareTo("null") != 0) {
                cur.right=new TreeNode(Integer.valueOf(strs[idx]));
                que.add(cur.right);
            }
        }
        return root;
    }
}
```

方法 2：

```
    public static TreeNode buildTree(Integer[] arr) {
        if (arr == null || arr.length == 0 || arr[0] == null) return null;
        TreeNode root = generateTreeNode(arr[0]);
        TreeNode cur = root;
        Queue<TreeNode> queue = new LinkedList<>();
        int index = 0;
        while (cur != null) {
            if (2 * index + 1 < arr.length) {
                cur.left = generateTreeNode(arr[2 * index + 1]);
                queue.add(cur.left);
            }
            if (2 * index + 2 < arr.length) {
                cur.right = generateTreeNode(arr[2 * index + 2]);
                queue.add(cur.right);
            }
            cur = null;
            while (!queue.isEmpty()) {
                cur = queue.poll();
                if (cur != null) {
                    break;
                }
            }
            index++;
        }
        return root;
    }
    public   static TreeNode generateTreeNode(Integer val) {
        if (val == null) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            return null;
        }
        return new TreeNode(val);
    }
```

【输出成 String】

需要对应的遍历+printList

```
public static String printList (List<Integer> list) {
        String res = "";
        int[] arr = list.stream().mapToInt(Integer::valueOf).toArray();
        for (int i = 0; i < arr.length; i++) {
            if (i != arr.length - 1) {
                res += arr[i];
                res += " ";
            }   else {
                res += arr[i];
            }
        }
        return res;
}
```

遍历二叉树详解：

https://leetcode-cn.com/problems/binary-tree-preorder-traversal/solution/leetcodesuan-fa-xiu-lian-dong-hua-yan-shi-xbian-2/

1、二叉树的前序遍历（144）

144. 二叉树的前序遍历

难度 中等 353 收藏 分享 切换为英文 关注

给定一个二叉树，返回它的 *前序* 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,2,3]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

方法 1：递归

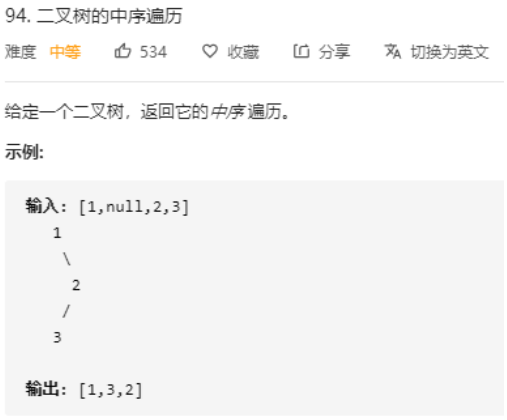
```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
```

```
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        dfs(res, root);
        return res;
    }
    public void dfs(List<Integer> res, TreeNode root) {
        if (root == null) return;
        res.add(root.val);
        dfs(res, root.left);
        dfs(res, root.right);
    }
}
```

方法 2：迭代

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> sta = new Stack<>();
        while (!sta.isEmpty() || root != null) {
            while (root != null) {
                res.add(root.val);
                sta.push(root);
                root = root.left;
            }
            if (!sta.isEmpty()) {
                root = sta.pop();
                root = root.right;
            }
        }
        return res;
    }
}
```

2、二叉树的中序遍历（94）



94. 二叉树的中序遍历

难度 中等 534 收藏 分享 切换为英文

给定一个二叉树，返回它的*中序*遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [1,3,2]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

145. 二叉树的后序遍历

难度 困难 383 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，返回它的 *后序* 遍历。

示例:

```
输入: [1,null,2,3]
   1
    \
     2
    /
   3

输出: [3,2,1]
```

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

方法 1：递归算法（最快最简单）：

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        dfs(res,root);
        return res;
    }
    void dfs(List<Integer> res, TreeNode root) {
        if (root == null) {
            return;
        }
        dfs(res, root.left);
        res.add(root.val);
        dfs(res, root.right);
    }
}
```

方法 2：迭代算法：

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
```

```
            while (stack.size() > 0 || root != null) {
                //不断往左子树方向走，每走一次就将当前节点保存到栈中
                //这是模拟递归的调用
                if(root!=null) {
                    stack.add(root);
                    root = root.left;
                //当前节点为空，说明左边走到头了，从栈中弹出节点并保存
                //然后转向右边节点，继续上面整个过程
                } else {
                    TreeNode tmp = stack.pop();
                    res.add(tmp.val);
                    root = tmp.right;
                }
            }
            return res;
        }
}
```

3、二叉树后序遍历（145）

145. 二叉树的后序遍历

难度 困难 383 收藏 分享 切换为英文 关注 反

给定一个二叉树，返回它的 *后序* 遍历。

示例:

输入: [1,null,2,3]

```
   1
    \
     2
    /
   3
```

输出: [3,2,1]

进阶: 递归算法很简单，你可以通过迭代算法完成吗?

方法 1：递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
```

```
        dfs(res,root);
        return res;
    }
    void dfs(List<Integer> res, TreeNode root) {
        if (root == null) return;
        dfs(res, root.left);
        dfs(res, root.right);
        res.add(root.val);
    }
}
```

方法 2：迭代

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        int left = 1, right = 2;
        Stack<TreeNode> sta1 = new Stack<TreeNode>();
        Stack<Integer> sta2 = new Stack<Integer>();//辅助栈,用来判断子节点返回父节点时处于左节点还是右节点。
        while (!sta1.isEmpty() || root != null) {
            while (root != null) {
                sta1.push(root);
                sta2.push(left);
                root = root.left;
            }
            while (!sta1.isEmpty() && sta2.peek() == right) {
                //如果是从右子节点返回父节点，则任务完成，将两个栈的栈顶弹出
                sta2.pop();
                res.add(sta1.pop().val);
            }
            while (!sta1.isEmpty() && sta2.peek() == left) {
                //如果是从左子节点返回父节点，则将标记改为右子节点
                sta2.pop();
                sta2.push(right);
                root = sta1.peek().right;
            }
```

```
        }
        return res;
    }
}
```

完整的所有遍历

```
import java.util.*;
//树的结构参考 LeetCode 94 题 中序遍历
class TreeNode{
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(){}
    TreeNode(int x){
        this.val = x;
    }
}
public class Traversal {
    public static void main(String[] args) {
        while (true) {
            System.out.println("请输入一个树，节点间用逗号隔开：");
            Scanner sc = new Scanner(System.in);
            String str = sc.nextLine();
            //第一种方法
//              String[] strs = str.split(",");
//              Integer[] arr = new Integer[strs.length];
//              String special = "null";
//              for (int i = 0; i < strs.length; i++) {
//                  if (strs[i].equals(special)) {
//                      arr[i] = null;
//                  } else {
//                      arr[i] = Integer.valueOf(strs[i]);
//                  }
//              }
//              //Integer[] arr = {3,1,4,null,null,2};
//              //Integer[] arr = {1,null,2,3};
//              TreeNode before = buildTree(arr);
            //第二种方法
            TreeNode before = deserialize(str);
            List<Integer> list1 = DLR_recursion(before);
            String res1 = printList(list1);
            System.out.println("前序遍历-递归法结果：" + res1);
            List<Integer> list2 = DLR_iteraytion(before);
            String res2 = printList(list2);
            System.out.println("前序遍历-迭代法结果：" + res2);
```

宗北尚用 公众号：yes的练级攻略

```
            List<Integer> list3 = LDR_recursion(before);
            String res3 = printList(list3);
            System.out.println("中序遍历-递归法结果：" + res3);
            List<Integer> list4 = LDR_iteraytion(before);
            String res4 = printList(list4);
            System.out.println("中序遍历-迭代法结果：" + res4);
            List<Integer> list5 = LRD_recursion(before);
            String res5 = printList(list5);
            System.out.println("后序遍历-递归法结果：" + res5);
            List<Integer> list6 = LRD_iteraytion(before);
            String res6 = printList(list6);
            System.out.println("后序遍历-迭代法结果：" + res6);
            System.out.println("");
        }
    }
    //前序遍历-方法 1：递归
    public static List<Integer> DLR_recursion(TreeNode before){
        List<Integer> res = new ArrayList<>();
        dfs_DLR(res, before);
        return res;
    }
    public static void dfs_DLR(List<Integer> res, TreeNode before) {
        if (before == null) return;
        res.add(before.val);
        dfs_DLR(res, before.left);
        dfs_DLR(res, before.right);
    }
    //前序遍历-方法 2：迭代
    public static List<Integer> DLR_iteraytion(TreeNode before){
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while (!stack.isEmpty() || before != null) {
            while (before != null) {
                res.add(before.val);
                stack.push(before);
                before = before.left;
            }
            if (!stack.isEmpty()) {
                before = stack.pop();
                before = before.right;
            }
        }
        return res;
    }
```

```
//中序遍历-方法 1：递归
public static List<Integer> LDR_recursion(TreeNode before){
    List<Integer> res = new ArrayList<>();
    dfs_LDR(res, before);
    return res;
}
public static void dfs_LDR(List<Integer> res, TreeNode before) {
    if (before == null) return;
    dfs_LDR(res, before.left);
    res.add(before.val);
    dfs_LDR(res, before.right);
}
//中序遍历-方法 2：迭代
public static List<Integer> LDR_iteraytion(TreeNode before){
    List<Integer> res = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    while (!stack.isEmpty() || before != null) {
        while (before != null) {
            stack.push(before);
            before = before.left;
        }
        if (!stack.isEmpty()) {
            before = stack.pop();
            res.add(before.val);
            before = before.right;
        }
    }
    return res;
}
//后序遍历-方法 1：递归
public static List<Integer> LRD_recursion(TreeNode before){
    List<Integer> res = new ArrayList<>();
    dfs_LRD(res, before);
    return res;
}
public static void dfs_LRD(List<Integer> res, TreeNode before) {
    if (before == null) return;
    dfs_LRD(res, before.left);
    dfs_LRD(res, before.right);
    res.add(before.val);
}
//后序遍历-方法 2：迭代
public static List<Integer> LRD_iteraytion(TreeNode before){
    List<Integer> res = new ArrayList<>();
```

```
        int left = 1, right = 2;
        Stack<TreeNode> stack1 = new Stack<TreeNode>();
        Stack<Integer> stack2 = new Stack<Integer>();//辅助栈，用来判断子节点返回父节点时处于左节点还是右节点。
        while (!stack1.isEmpty() || before != null) {
            while (before != null) {
                //将节点压入栈 1，并在栈 2 将节点标记为左节点
                stack1.push(before);
                stack2.push(left);
                before = before.left;
            }
            while (!stack1.isEmpty() && stack2.peek() == right) {
                //如果是从右子节点返回父节点，则任务完成，将两个栈的栈顶弹出
                stack2.pop();
                res.add(stack1.pop().val);
            }
            if (!stack1.isEmpty() && stack2.peek() == left) {
                //如果是从左子节点返回父节点，则将标记改为右子节点
                stack2.pop();
                stack2.push(right);
                before = stack1.peek().right;
            }
        }
        return res;
    }

    public static String printList (List<Integer> list) {
        String res = "";
        int[] arr = list.stream().mapToInt(Integer::valueOf).toArray();
        for (int i = 0; i < arr.length; i++) {
            if (i != arr.length - 1) {
                res += arr[i];
                res += " ";
            }  else {
                res += arr[i];
            }
        }
        return res;
    }

    static TreeNode buildTree(Integer[] arr) {
        if (arr == null || arr.length == 0 || arr[0] == null) return null;
        TreeNode root = generateTreeNode(arr[0]);
        TreeNode cur = root;
```

```
        Queue<TreeNode> queue = new LinkedList<>();
        int index = 0;
        while (cur != null) {
            if (2 * index + 1 < arr.length) {
                cur.left = generateTreeNode(arr[2 * index + 1]);
                queue.add(cur.left);
            }
            if (2 * index + 2 < arr.length) {
                cur.right = generateTreeNode(arr[2 * index + 2]);
                queue.add(cur.right);
            }
            cur = null;
            while (!queue.isEmpty()) {
                cur = queue.poll();
                if (cur != null) {
                    break;
                }
            }
            index++;
        }
        return root;
    }
    static TreeNode generateTreeNode(Integer val) {
        if (val == null) {
            return null;
        }
        return new TreeNode(val);
    }

}
```

4、从上到下打印二叉树（剑指 32-1）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 32 - I. 从上到下打印二叉树

难度 中等 33 收藏 分享 切换为英文 关注

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如：
给定二叉树：`[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回：

```
[3,9,20,15,7]
```

提示：

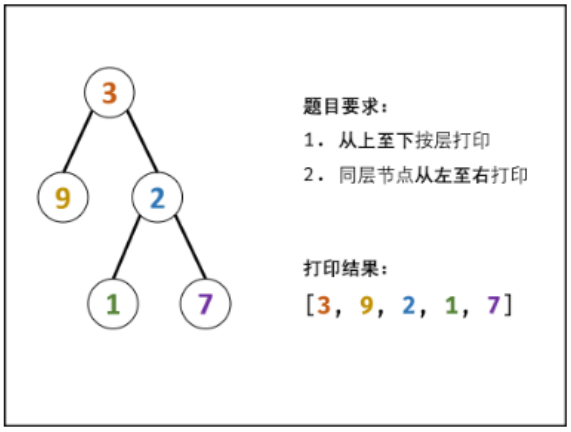
1. 节点总数 <= 1000

禁止转用

https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/solution/mian-shi-ti-32-i-cong-shang-dao-xia-da-yin-er-ch-4/

解题思路：

- 题目要求的二叉树的 **从上至下** 打印（即按层打印），又称为二叉树的 **广度优先搜索**（BFS）。
- BFS 通常借助 **队列** 的先入先出特性来实现。



算法流程：

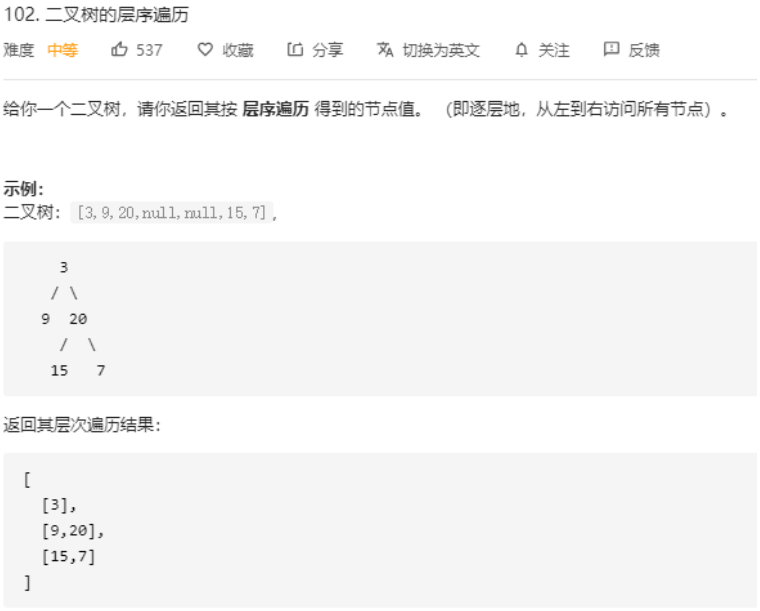
1. **特例处理：** 当树的根节点为空，则直接返回空列表 `[]` ；
2. **初始化：** 打印结果列表 `res = []` ，包含根节点的队列 `queue = [root]` ；
3. **BFS 循环：** 当队列 `queue` 为空时跳出；
    1. **出队：** 队首元素出队，记为 `node` ；
    2. **打印：** 将 `node.val` 添加至列表 `tmp` 尾部；
    3. **添加子节点：** 若 `node` 的左（右）子节点不为空，则将左（右）子节点加入队列 `queue` ；
4. **返回值：** 返回打印结果列表 `res` 即可。

复杂度分析：

- **时间复杂度 $O(N)$：** N 为二叉树的节点数量，即 BFS 需循环 N 次。
- **空间复杂度 $O(N)$：** 最差情况下，即当树为平衡二叉树时，最多有 $N/2$ 个树节点**同时**在 `queue` 中，使用 $O(N)$ 大小的额外空间。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int[] levelOrder(TreeNode root) {
        if(root == null) return new int[0];
        Queue<TreeNode> queue = new LinkedList<>(){{ add(root); }};
        ArrayList<Integer> ans = new ArrayList<>();
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            ans.add(node.val);
            if(node.left != null) queue.add(node.left);
            if(node.right != null) queue.add(node.right);
        }
        int[] res = new int[ans.size()];
        for(int i = 0; i < ans.size(); i++)
            res[i] = ans.get(i);
        return res;
    }
}
```

5、从上到下打印二叉树 2（剑指 32-2）【二叉树的层序遍历（102）】



链接：
https://leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/tao-mo-ban-bfs-he-dfs-du-ke-yi-jie-jue-by-fuxuemin/

BFS 使用队列，把每个还没有搜索到的点依次放入队列，然后再弹出队列的头部元素当做当前遍历点。BFS 总共有两个模板：

如果不需要确定当前遍历到了哪一层，BFS 模板如下。

```
while queue 不空：
    cur = queue.pop()
    for 节点 in cur 的所有相邻节点：
        if 该节点有效且未访问过：
            queue.push(该节点)
```

如果要确定当前遍历到了哪一层，BFS 模板如下。

这里增加了 level 表示当前遍历到二叉树中的哪一层了，也可以理解为在一个图中，现在已经走了多少步了。size 表示在当前遍历层有多少个元素，也就是队列中的元素数，我们把这些元素一次性遍历完，即把当前层的所有元素都向外走了一步。

```
level = 0
while queue 不空：
    size = queue.size()
    while (size --) {
        cur = queue.pop()
        for 节点 in cur 的所有相邻节点：
            if 该节点有效且未被访问过：
                queue.push(该节点)
    }
    level ++;
```

上面两个是通用模板，在任何题目中都可以用，是要记住的！

本题要求二叉树的层次遍历，所以同一层的节点应该放在一起，故使用模板二。

使用队列保存每层的所有节点，每次把队列里的原先所有节点进行出队列操作，再把每个元

禁止商用 公众号：yes 的练级攻略

素的非空左右子节点进入队列。因此即可得到每层的遍历。

方法 1：BFS（1ms）

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (! queue.isEmpty()) {
            int size = queue.size();
            List<Integer> level = new LinkedList<>();
            for (int i = 0; i < size; ++i) {
                TreeNode cur = queue.peek();
                queue.poll();
                if (cur == null){
                    continue;
                }
                level.add(cur.val);
                queue.offer(cur.left);
                queue.offer(cur.right);
            }
            if(! level.isEmpty()) {
                res.add(level);
            }
        }
        return res;
    }
}
```

方法 2：DFS（0ms）

DFS 不是按照层次遍历的。为了让递归的过程中同一层的节点放到同一个列表中，在递归时要记录每个节点的深度 level。递归到新节点要把该节点放入 level 对应列表的末尾。当遍历到一个新的深度 level，而最终结果 res 中还没有创建 level 对应的列表时，应该在 res 中新建一个列表用来保存该 level 的所有节点。

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res  = new ArrayList<>();
```

```
        if(root != null){
            dfs(res, root, 0);
        }
        return res;
    }

    private void dfs(List<List<Integer>> res, TreeNode node, int level)
{
        if(res.size() - 1 < level){
            res.add(new ArrayList<Integer>());
        }
        res.get(level).add(node.val);
        if(node.left!=null){
            dfs(res, node.left, level + 1);
        }
        if(node.right!=null){
            dfs(res, node.right, level + 1);
        }
    }
}
```

如果输出是 List<String>，可以看不同的二叉搜索树后面的整个项目源码

6、从上到下打印二叉树 3（剑指 32-3）

剑指 Offer 32 - III. 从上到下打印二叉树 III

难度 中等 38 收藏 分享 切换为英文 关注 反馈

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

例如：
给定二叉树：`[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层次遍历结果：

```
[
  [3],
  [20,9],
  [15,7]
]
```

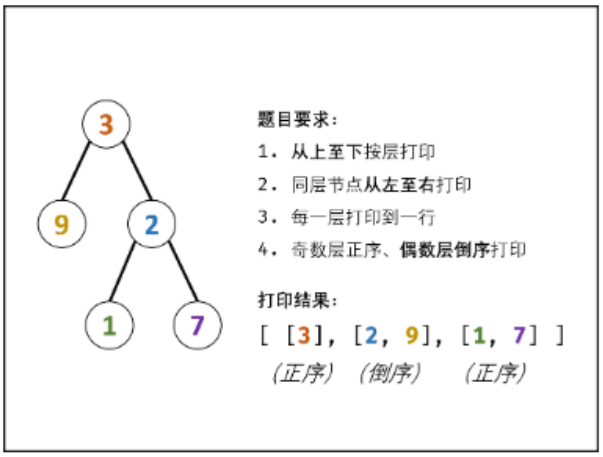
提示：

1. 节点总数 <= 1000

https://leetcode-cn.com/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/solution/mian-shi-ti-32-iii-cong-shang-dao-xia-da-yin-er--3/

更多硬核内容就在公众号「yes 的练级攻略」

面试题32 - I. 从上到下打印二叉树 主要考察 **树的按层打印**；
面试题32 - II. 从上到下打印二叉树 II 额外要求 **每一层打印到一行**；
本题额外要求 **打印顺序交替变化**（建议按顺序做此三道题）。



方法一：层序遍历 + 双端队列

- 利用双端队列的两端皆可添加元素的特性，设打印列表（双端队列） `tmp` ，并规定：
  - 奇数层 则添加至 `tmp` **尾部**，
  - 偶数层 则添加至 `tmp` **头部**。

算法流程：

1. **特例处理：** 当树的根节点为空，则直接返回空列表 `[]` ；
2. **初始化：** 打印结果空列表 `res` ，包含根节点的双端队列 `deque` ；
3. BFS **循环：** 当 `deque` 为空时跳出；
   1. 新建列表 `tmp` ，用于临时存储当前层打印结果；
   2. **当前层打印循环：** 循环次数为当前层节点数（即 `deque` 长度）；
      1. **出队：** 队首元素出队，记为 `node` ；
      2. **打印：** 若为奇数层，将 `node.val` 添加至 `tmp` 尾部；否则，添加至 `tmp` 头部；
      3. **添加子节点：** 若 `node` 的左（右）子节点不为空，则加入 `deque` ；
   3. 将当前层结果 `tmp` 转化为 list 并添加入 `res` ；
4. **返回值：** 返回打印结果列表 `res` 即可；

复杂度分析：

- **时间复杂度** $O(N)$ ： N 为二叉树的节点数量，即 BFS 需循环 N 次，占用 $O(N)$ ；双端队列的队首和队尾的添加和删除操作的时间复杂度均为 $O(1)$ 。
- **空间复杂度** $O(N)$ ： 最差情况下，即当树为满二叉树时，最多有 $N/2$ 个树节点 **同时** 在 `deque` 中，使用 $O(N)$ 大小的额外空间。

方法 1：层序遍历加双端队列

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
 */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()) {
            LinkedList<Integer> tmp = new LinkedList<>();
            for(int i = queue.size(); i > 0; i--) {
                TreeNode node = queue.poll();
                if(res.size() % 2 == 0) tmp.addLast(node.val); // 偶数层 -> 队列
头部
                else tmp.addFirst(node.val); // 奇数层 -> 队列尾部
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
            res.add(tmp);
        }
        return res;
    }
}
```

方法三：层序遍历 + 倒序

- 此方法的优点是只用列表即可，无需其他数据结构。
- **偶数层倒序：** 若 `res` 的长度为 **奇数** ，说明当前是偶数层，则对 `tmp` 执行 **倒序** 操作。

复杂度分析：

- **时间复杂度** $O(N)$ ： N 为二叉树的节点数量，即 BFS 需循环 N 次，占用 $O(N)$ 。**共完成** 少于 N 个节点的倒序操作，占用 $O(N)$ 。
- **空间复杂度** $O(N)$ ： 最差情况下，即当树为满二叉树时，最多有 $N/2$ 个树节点**同时**在 `queue` 中，使用 $O(N)$ 大小的额外空间。

☆ 收藏

方法 2：层序遍历加倒序

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()) {
            List<Integer> tmp = new ArrayList<>();
            for(int i = queue.size(); i > 0; i--) {
                TreeNode node = queue.poll();
                tmp.add(node.val);
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
```

```
            if(res.size() % 2 == 1) Collections.reverse(tmp);
            res.add(tmp);
        }
        return res;
    }
}
```

7、二叉树的层序遍历 2（107）

107. 二叉树的层次遍历 II

难度 简单 285 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，返回其节点值自底向上的层次遍历。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

例如：
给定二叉树 [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其自底向上的层次遍历为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

https://leetcode-cn.com/problems/binary-tree-level-order-traversal-ii/solution/san-chong-shi-xian-tu-jie-107er-cha-shu-de-ceng-ci/

方法 1：BFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        if(root==null) {
            return new ArrayList<List<Integer>>();
        }
        //改用链表实现，每次都插入到第一位
        LinkedList<List<Integer>> res = new LinkedList<List<Integer>>();
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        while(!queue.isEmpty()) {
            int size = queue.size();
            ArrayList<Integer> tmp = new ArrayList<Integer>();
```

```
            for(int i=0;i<size;++i) {
                TreeNode node = queue.poll();
                tmp.add(node.val);
                if(node.left!=null) {
                    queue.offer(node.left);
                }
                if(node.right!=null) {
                    queue.offer(node.right);
                }
            }
            //每次都保存到链表的第一位，这样自带了翻转的功能
            res.add(0,tmp);
        }
        return res;
    }
}
```

方法 2：DFS

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        if(root==null) {
            return new ArrayList<List<Integer>>();
        }
        //用来存放最终结果
        ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();
        dfs(root,res,1);
        Collections.reverse(res);
        return res;
    }

    private void dfs(TreeNode root,ArrayList<List<Integer>> res,int index) {
        if(root==null) {
            return;
        }
        //如果 index 大于 res 大小，说明这一层没有对应的集合，需要新创建
        if(index>res.size()) {
            res.add(new ArrayList<Integer>());
        }
        //将当前层的元素直接放到对应层的末尾即可
        res.get(index-1).add(root.val);
        //继续遍历左右节点，同时将层高+1
        dfs(root.left,res,index+1);
        dfs(root.right,res,index+1);
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

8、二叉树的锯齿形层次遍历（103）

103. 二叉树的锯齿形层次遍历

难度 中等 256 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：
给定二叉树 [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

返回锯齿形层次遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

禁止商用

https://leetcode-cn.com/problems/binary-tree-zigzag-level-order-traversal/solution/er-cha-shu-de-ju-chi-xing-ceng-ci-bian-li-by-leetc/

方法一：BFS（广度优先遍历）

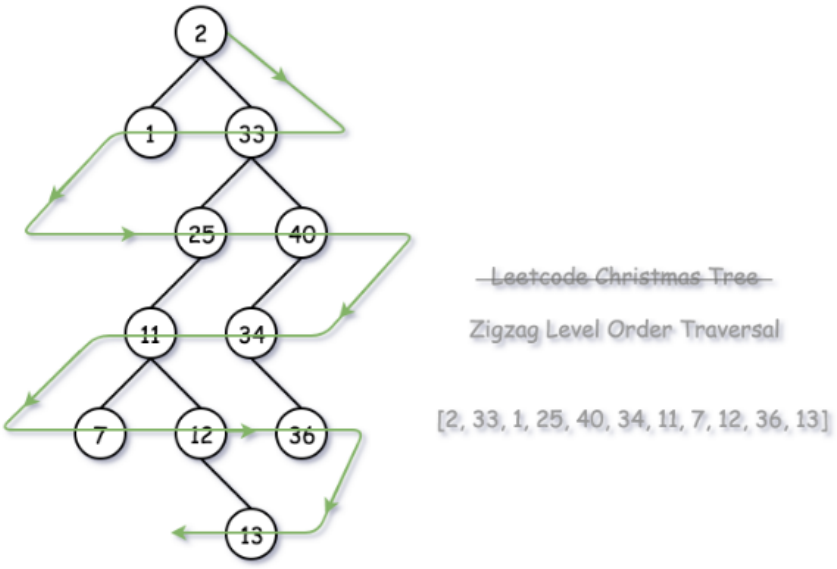
思路

最直观的方法是 BFS，逐层遍历树。

BFS 在每层的默认顺序是从左到右，因此需要调整 BFS 算法以生成锯齿序列。

> 最关键的是使用双端队列遍历，可以在队列的任一端插入元素。

如果需要 FIFO （先进先出）的顺序，则将新元素添加到队列尾部，后插入的元素就可以排在后面。如果需要 FILO （先进后出）的顺序，则将新元素添加到队列首部，后插入的元素就可以排在前面。

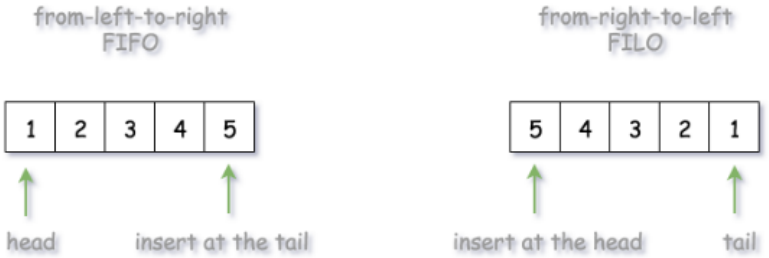


算法

实现 BFS 的几种算法。

- 使用两层嵌套循环。外层循环迭代树的层级，内层循环迭代每层上的节点。
- 也可以使用一层循环实现 BFS。将要访问的节点添加到队列中，使用 **分隔符**（例如：空节点）把不同层的节点分隔开。分隔符表示一层结束和新一层开始。

这里采用第二种方法。在此算法的基础上，借助双端队列实现锯齿形顺序。在每一层，使用一个空的双端队列保存该层所有的节点。根据每一层的访问顺序，即从左到右或从右到左，决定从双端队列的哪一端插入节点。



- 实现从左到右的遍历顺序（FIFO）。将元素添加到队列尾部，保证后添加的节点后被访问。从上图中可以看出，输入序列 `[1, 2, 3, 4, 5]`，按照 FIFO 顺序得到输出序列为 `[1, 2, 3, 4, 5]`。
- 实现从右到左的遍历顺序（FILO）。将元素添加到队列头部，保证后添加的节点先被访问。输入序列 `[1, 2, 3, 4, 5]`，按照 FILO 顺序得到输出序列为 `[5, 4, 3, 2, 1]`。

注意：一种替代做法是，实现标准的 BFS 算法，得到每层节点从左到右的遍历顺序。然后按照要求 **翻转** 某些层节点的顺序，得到锯齿形的遍历结果。

复杂度分析

- 时间复杂度：$\mathcal{O}(N)$，其中 N 是树中节点的数量。
  - 每个节点仅访问一次。
  - 双端队列的插入操作为常数时间。如果使用数组或 list，头部插入需要 $\mathcal{O}(K)$ 的时间，其中 K 是数组或 list 的长度。
- 空间复杂度：$\mathcal{O}(N)$，其中 N 是树中节点的数量。
  - 除了输出数组，主要的内存开销是双端队列。
  - 任何时刻，双端队列中最多只存储两层节点。因此双端队列的大小不超过 $2 \cdot L$，其中 L 是一层的最大节点数。包含最多节点的层可能是完全二叉树的叶节点层，大约有 $L = \frac{N}{2}$ 个节点。因此最坏情况下，空间复杂度为 $2 \cdot \frac{N}{2} = N$。

方法 1：BFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

更多硬核内容就在公众号「yes 的练级攻略」

```java
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) return res;
        LinkedList<Integer> path = new LinkedList<>();
        LinkedList<TreeNode> nodes = new LinkedList<>();
        nodes.addLast(root);
        nodes.addLast(null);
        boolean flag = true; //true 代表从左到右， false 代表从右到左
        while (nodes.size() > 0) {
            TreeNode cur = nodes.pollFirst();
            if (cur != null) {
                if (flag) path.addLast(cur.val);
                else path.addFirst(cur.val);
                if (cur.left != null) nodes.addLast(cur.left);
                if (cur.right != null) nodes.addLast(cur.right);
            } else {
                res.add(path);
                path = new LinkedList<>();
                if (nodes.size() > 0) nodes.addLast(null);
                flag = !flag;
            }
        }
        return res;
    }
}
```

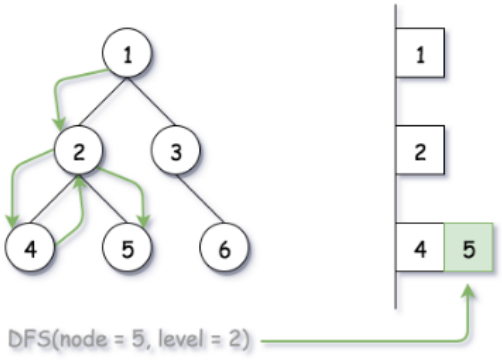
yes的练级攻略

方法二：DFS （深度优先遍历）

思路

也可以使用 *DFS* 实现 *BFS* 的遍历顺序。

> 在 DFS 遍历期间，将结果保存在按层数索引的全局数组中。即元素 `array[level]` 存储同一层的所有节点。然后在 DFS 的每一步更新全局数组。



与改进的 BFS 算法类似，使用双端队列保存同一层的所有节点，并交替插入方向（从首部插入或从尾部插入）得到需要的输出顺序。

算法

使用递归实现 DFS 算法。定义一个递归方法 `DFS(node, level)`，方法参数为当前节点 `node` 和指定层数 `level`。该方法共执行三个步骤：

- 如果是第一次访问该层的节点，即该层的双端队列不存在。那么创建一个双端队列，并添加该节点到队列中。
- 如果当前层的双端队列已存在，根据顺序，将当前节点插入队列头部或尾部。
- 最后，为每个节点调用该递归方法。

也可以通过 **迭代** 实现 DFS 遍历。

复杂度分析

- 时间复杂度：$\mathcal{O}(N)$，其中N 是树中节点的数量。
  - 与 BFS 相同，每个节点只访问一次。
- 空间复杂度：$\mathcal{O}(H)$，其中 H 是树的高度。例如：包含 N 个节点的树，高度大约为 $\log_2 N$。
  - 与 BFS 不同，在 DFS 中不需要维护双端队列。
  - 方法递归调用会产生额外的内存消耗。方法 `DFS(node, level)` 的调用堆栈大小刚好等于节点所在层数。因此 DFS 的空间复杂度为 $\mathcal{O}(\log_2 N)$，这比 BFS 好很多。

方法 2：DFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```java
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) return res;
        dfs(root, 0, res);
        return res;
    }
    public void dfs(TreeNode node, int level, List<List<Integer>> res) {
        if (level >= res.size()) {
            LinkedList<Integer> newLevel = new LinkedList<>();
            newLevel.add(node.val);
            res.add(newLevel);
        } else {
            if ((level & 1) == 0) res.get(level).add(node.val);
            else res.get(level).add(0, node.val);
        }
        if (node.left != null) dfs(node.left, level + 1, res);
        if (node.right != null) dfs(node.right, level + 1, res);
    }
}
```

9、二叉树的层平均值（637）

637. 二叉树的层平均值

难度 简单 156 收藏 分享 切换为英文 关注 反馈

给定一个非空二叉树, 返回一个由每层节点平均值组成的数组。

示例 1：

输入：

```
    3
   / \
  9  20
    /  \
   15   7
```

输出： [3, 14.5, 11]

解释：

第 0 层的平均值是 3 , 第1层是 14.5 , 第2层是 11 。因此返回 [3, 14.5, 11] 。

提示：

- 节点值的范围在32位有符号整数范围内。

https://leetcode-cn.com/problems/average-of-levels-in-binary-tree/solution/er-cha-shu-de-ceng-ping-jun-zhi-li-jie-er-cha-shu-/

解题思路

1. 既然是求层平均值，那么当然是层次遍历嘛！
2. 借助队列实现二叉树的层次遍历
    1. 初始化一个队列，将根节点入队；
    2. 根据队列的size实现遍历当前层；
    3. 每次出队一个节点，记录节点的值，并把该节点的左右节点入队；
    4. 循环执行2,3直至队列为空。
3. 注意：根据题目给定的数据范围，应该用double类型，int类型可能会溢出。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是树中的节点个数。
- 空间复杂度：$O(M)$，其中 M 是树中每一层节点个数的最大值，即为广度优先搜索中使用队列存储同一层节点需要的空间。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Double> averageOfLevels(TreeNode root) {
        List<Double> res = new LinkedList<>();
        Queue<TreeNode> que = new LinkedList<>();
        que.offer(root);
        while (!que.isEmpty()) {
            double sum = 0;
            int cnt = que.size();
            for (int i = que.size(); i > 0; i--) {
                TreeNode node = que.poll();
                sum += node.val;
                if (node.left != null) que.offer(node.left);
                if (node.right != null) que.offer(node.right);
            }
            res.add(sum / cnt);
        }
        return res;
    }
}
```

10、输出二叉树（655）

更多硬核内容就在公众号「yes 的练级攻略」

655. 输出二叉树

难度 中等 66 收藏 分享 切换为英文 关注 反馈

在一个 m*n 的二维字符串数组中输出二叉树，并遵守以下规则：

1. 行数 `m` 应当等于给定二叉树的高度。
2. 列数 `n` 应当总是奇数。
3. 根节点的值（以字符串格式给出）应当放在可放置的第一行正中间。根节点所在的行与列会将剩余空间划分为两部分（**左下部分和右下部分**）。你应该将左子树输出在左下部分，右子树输出在右下部分。左下和右下部分应当有相同的大小。即使一个子树为空而另一个非空，你不需要为空的子树输出任何东西，但仍需要为另一个子树留出足够的空间。然而，如果两个子树都为空则不需要为它们留出任何空间。
4. 每个未使用的空间应包含一个空的字符串`""`。
5. 使用相同的规则输出子树。

示例 1:

输入:

```
     1
    /
   2
```

输出:

```
[["", "1", ""],
 ["2", "", ""]]
```

示例 2:

输入:

```
     1
    / \
   2   3
    \
     4
```

输出:

```
[["", "", "", "1", "", "", ""],
 ["", "2", "", "", "", "3", ""],
 ["", "", "4", "", "", "", ""]]
```

示例 3:

输入:

```
      1
     / \
    2   5
   /
  3
 /
4
```

输出:

```
[["",  "",  "",  "",  "",  "",  "",  "1",  "",  "",  "",  "",  "",  "",  ""]
 ["",  "",  "",  "2",  "",  "",  "",  "",  "",  "",  "",  "5",  "",  "",  ""]
 ["",  "3",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  ""]
 ["4",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  "",  ""]]
```

注意: 二叉树的高度在范围 [1, 10] 中。

https://leetcode-cn.com/problems/print-binary-tree/solution/shu-chu-er-cha-shu-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：递归【通过】

创建一个长度为 $height \times (2^{height} - 1)$ 的数组 res，其中 $height$ 是树的高度。先使用空字符串 `""` 填充数组 res。然后递归调用 `fill(res, root, i, l, r)` 将节点的值输出到数组 res 中，其中 i 表示当前节点所在层数，l 和 r 表示当前子树在数组 res 中的左右边界，当前节点被输出到数组 res 第 i 行的第 l 列和第 r 列中间位置上。

在递归方法中，执行以下步骤：

1. 如果到达树的末尾，即 `root = null`，直接返回。
2. 确定当前节点所在的列 $j = (l + r)/2$。将当前节点输出到数组 res 的第 i 行第 j 列，即 $res[i][j]$。
3. 递归调用 `fill(res, root.left, i + 1, l, (l + r) / 2)`，输出 $root$ 的左子树。
4. 递归调用 `fill(res, root.right, i + 1, (l + r + 1) / 2, r)`，输出 $root$ 的右子树。

注意：在第三步和第四步的递归调用中需要更新行号，确保子节点可以输出的正确的位置。另外，也要根据 l 和 r 更新子树的左右边界。

另外，创建方法 `getHeight(root)`，用于计算 $root$ 为根节点的树高度 $height$。递归遍历树的所有分支，从中找出最深的一个分支作为树的高度。

最后，将数组 res 转换成题目要求格式。

复杂度分析

- 时间复杂度：$O(h * 2^h)$，其中 h 是树的高度，填充长度为 $h \times (2^h - 1)$ 的数组 res。
- 空间复杂度：$O(h * 2^h)$，数组 res 的长度为 $h \times (2^h - 1)$。

方法 1：递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<String>> printTree(TreeNode root) {
        int height = getHeight(root);
        String[][] res = new String[height][(1 << height) - 1];
        for (String[] arr : res) {
            Arrays.fill(arr, "");
        }
        List<List<String>> ans = new ArrayList<>();
        fill(res, root, 0, 0, res[0].length);
        for (String[] arr : res) {
            ans.add(Arrays.asList(arr));
        }
        return ans;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    }
    public int getHeight(TreeNode root) {
        if (root == null) return 0;
        return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
    }
    public void fill(String[][] res, TreeNode root, int i, int l, int r) {
        if (root == null) return;
        int mid = l + (r - l) / 2;
        res[i][mid] = "" + root.val;
        fill(res, root.left, i + 1, l, mid);
        fill(res, root.right, i + 1, mid + 1, r);
    }
}
```

方法二：使用队列（BFS）【通过】

算法

也可以使用广度优先搜索解决该问题。使用类 $Params$ 存储树的节点 $node$，类中包含该节点的值，所在层数 i，和以该节点为根的子树的左边界 l 和右边界 r。

初始化一个与 *方法一* 用途相同的数组 res，将根节点 $root$ 加入到队列 $queue$ 中，然后执行以下步骤。

1. 从队列中移除元素 p。
2. 将该元素的值输出到 $res[p.i][(p.l + p.r)/2]$，其中 i 表示当前节点的所在行，l 和 r 表示当前子树的左右边界。这些值都是在节点加入 $queue$ 之前就已经计算好的。
3. 如果节点 p 有左孩子，则将它的左孩子加入到 $queue$，同时计算左孩子的所在行，和以该节点为根的子树的左右边界。
4. 如果节点 p 有右孩子，则将它的右孩子加入到 $queue$，同时计算右孩子的所在行，和以该节点为根的子树的左右边界。
5. 回复步骤一到四，直到 $queue$ 为空。

最后，将数组 res 转换为题目要求的格式返回。

复杂度分析

- 时间复杂度：$O(h * 2^h)$，其中 h 是树的高度，填充长度为 $h \times (2^h - 1)$ 的数组 res。
- 空间复杂度：$O(h * 2^h)$，数组 res 的长度为 $h \times (2^h - 1)$。

方法 2：迭代

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
class Params {
    TreeNode root;
    int i, l, r;
    Params(TreeNode root, int i, int l, int r) {
        this.root = root;
        this.i = i;
        this.l = l;
        this.r = r;
    }
}
public List<List<String>> printTree(TreeNode root) {
    int height = getHeight(root);
    String[][] res = new String[height][(1 << height) - 1];
    for (String[] arr : res) {
        Arrays.fill(arr, "");
    }
    List<List<String>> ans = new ArrayList<>();
    fill(res, root, 0, 0, res[0].length);
    for (String[] arr : res) {
        ans.add(Arrays.asList(arr));
    }
    return ans;
}
public int getHeight(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> que = new LinkedList<>();
    que.offer(root);
    int res = 0;
    while (!que.isEmpty()){
        int size = que.size();
        while (size > 0) {
            TreeNode node = que.poll();
            if (node.left != null) que.offer(node.left);
            if (node.right != null) que.offer(node.right);
            size--;
        }
        res++;
    }
    return res;
}
public void fill(String[][] res, TreeNode root, int i, int l, int r) {
    if (root == null) return;
    Queue<Params> que = new LinkedList();
    que.add(new Params(root, 0, 0, res[0].length));
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            while (!que.isEmpty()){
                Params p = que.remove();
                int mid = p.l + (p.r - p.l) / 2;
                res[p.i][mid] = "" + p.root.val;
                if (p.root.left != null) que.add(new Params(p.root.left, p.i + 1, p
.l, mid));
                if (p.root.right != null) que.add(new Params(p.root.right, p.i + 1,
 mid + 1, p.r));
            }
        }
}
```

11、二叉树的堂兄弟节点（993）

993. 二叉树的堂兄弟节点

难度 简单 88 收藏 分享 切换为英文 关注 反馈

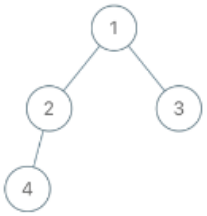
在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 k+1 处。

如果二叉树的两个节点深度相同，但**父节点不同**，则它们是一对*堂兄弟节点*。

我们给出了具有唯一值的二叉树的根节点 root，以及树中两个不同节点的值 x 和 y。

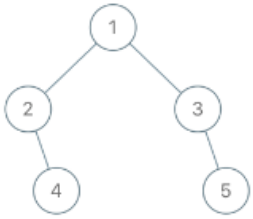
只有与值 x 和 y 对应的节点是堂兄弟节点时，才返回 true。否则，返回 false。

示例 1：



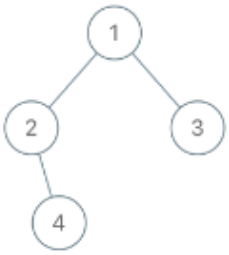
输入： root = [1,2,3,4], x = 4, y = 3
输出： false

示例 2：



输入： root = [1,2,3,null,4,null,5], x = 5, y = 4
输出： true

示例 3：



输入： root = [1,2,3,null,4], x = 2, y = 3
输出： false

提示：

1. 二叉树的节点数介于 2 到 100 之间。
2. 每个节点的值都是唯一的、范围为 1 到 100 的整数。

禁止商用

https://leetcode-cn.com/problems/cousins-in-binary-tree/solution/java-jian-ji-dai-ma-shi-xian-by-gary-liu-2/

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
        if (root == null) return false;
        // 初始化
        Queue<TreeNode> que = new LinkedList<>();
        que.offer(root);
        // 层序遍历
        while (!que.isEmpty()) {
            int size = que.size();
```

```
            TreeNode parent = null; // 用于存放堂兄弟节点的父亲节点
            // 以下遍历发生在一层，因此不用对深度做判断
            for (int i = 0; i < size; i++) {
                TreeNode node = que.poll();
                if (node.left != null) {
                    que.offer(node.left);
                    // 如果当前节点左孩子值为x 或y，说明当前节点为堂兄弟的父亲
                    if (node.left.val == x || node.left.val == y) {
                        // 如果上一个堂兄弟的父亲已找到并且与当前节点不同
                        if (parent != null && parent != node) {
                            return true;
                        }
                        parent = node;
                    }
                }
                if (node.right != null) {
                    que.offer(node.right);
                    // 如果当前节点右孩子值为x 或y，说明当前节点为堂兄弟的父亲
                    if (node.right.val == x || node.right.val == y) {
                        // 如果上一个堂兄弟的父亲已找到并且与当前节点不同
                        if (parent != null && parent != node) {
                            return true;
                        }
                        parent = node;
                    }
                }
            }
        }
        return false;
    }
}
```

12、二叉树的完全性检验（958）

更多硬核内容就在公众号「yes 的练级攻略」

958. 二叉树的完全性检验

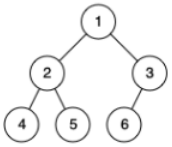
难度 中等 66 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，确定它是否是一个*完全二叉树*。

百度百科中对完全二叉树的定义如下：

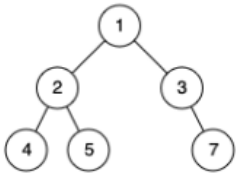
若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。（注：第 h 层可能包含 1～2^h 个节点。）

示例 1：



输入： [1,2,3,4,5,6]
输出： true
解释： 最后一层前的每一层都是满的（即，结点值为 {1} 和 {2,3} 的两层），且最后一层中的所有结点（{4,5,6}）都尽可能地向左。

示例 2：



输入： [1,2,3,4,5,null,7]
输出： false
解释： 值为 7 的结点没有尽可能靠向左侧。

提示：

1. 树中将会有 1 到 100 个结点。

https://leetcode-cn.com/problems/check-completeness-of-a-binary-tree/solution/hen-jian-dan-de-si-lu-dai-ma-hen-jian-ji-by-yuanyb/

层序遍历

思路：把一颗树的节点（包括空节点）按层序遍历排成一行，当且仅当存在两个相邻节点：前一个为null，后一个不为null时，才不是不是完全二叉树。
如：

```
      1
     / \
    2   3
   / \   \
  4   5  6
 / \   /
7   8 9
```

层序遍历序列为：[1, 2, 3, 4, 5, 6, 7, 8, null, null, 9]，其中 null 出现在了9前面，所以不合法

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
```

```
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isCompleteTree(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        TreeNode prev = root;
        queue.add(root);
        while (!queue.isEmpty()) {
            TreeNode node = queue.remove();
            if (prev == null && node != null)
                return false;
            if (node != null) {
                queue.add(node.left);
                queue.add(node.right);
            }
            prev = node;
        }
        return true;
    }
}
```

13、二叉树的序列化和反序列化（297）【序列化二叉树（剑指 37）】【序列化和反序列化二叉搜索树（449）】（lintcode 1235）

297. 二叉树的序列化与反序列化

难度 困难 323 收藏 分享 切换为英文 关注 反馈

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例:

你可以将以下二叉树：

```
    1
   / \
  2   3
     / \
    4   5
```

序列化为 "[1,2,3,null,null,4,5]"

提示: 这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明: 不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

更多硬核内容就在公众号「yes 的练级攻略」

1235. 序列化和反序列化二叉搜索树

中文 English

序列化是将数据结构或对象转换为一系列位的过程，以便它可以存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一个或另一个计算机环境中重建。

设计一个算法来序列化和反序列化**二叉搜索树**。 对序列化/反序列化算法的工作方式没有限制。 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。

编码的字符串应尽可能紧凑。

注意： 不要使用类成员/全局/静态变量来存储状态。 你的序列化和反序列化算法应该是无状态的。

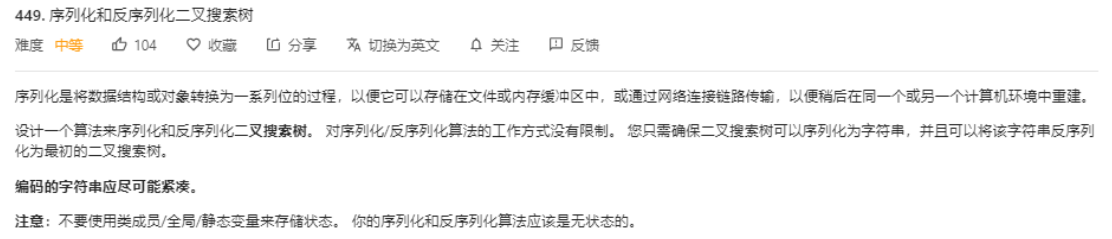
样例

样例 1:

```
输入：[2,1,3]
输出：[2,1,3]
解释：
    2
   / \
  1   3
```

样例 2:

```
输入：[1,#,2]
输出：[1,#,2]
解释：
  1
   \
    2
```



449. 序列化和反序列化二叉搜索树

难度 中等 104 收藏 分享 切换为英文 关注 反馈

序列化是将数据结构或对象转换为一系列位的过程，以便它可以存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一个或另一个计算机环境中重建。

设计一个算法来序列化和反序列化**二叉搜索树**。 对序列化/反序列化算法的工作方式没有限制。 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。

编码的字符串应尽可能紧凑。

注意：不要使用类成员/全局/静态变量来存储状态。 你的序列化和反序列化算法应该是无状态的。

https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/solution/er-cha-shu-de-xu-lie-hua-yu-fan-xu-lie-hua-by-le-2/

方法 1：DFS（163ms）

复杂度分析

- 时间复杂度：在序列化和反序列化函数中，我们只访问每个节点一次，因此时间复杂度为 $O(n)$，其中 n 是节点数，即树的大小。
- 空间复杂度：在序列化和反序列化函数中，我们递归会使用栈空间，故渐进空间复杂度为 $O(n)$。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
```

```
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        String s = reSerialize(root, "");
        String res = s.substring(0, s.length() - 1);//去最后一个逗号
        return res;
    }
    public String reSerialize(TreeNode root, String s){
        if (root == null) {
            s += "null,";
        } else {
            s += root.val + ",";
            s = reSerialize(root.left, s);
            s = reSerialize(root.right, s);
        }
        return s;
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        String[] nodes = data.split(",");
        List<String> nodeList = new LinkedList<>(Arrays.asList(nodes));
        return reDeserialize(nodeList);
    }

    public TreeNode reDeserialize(List<String> list){
        if (list.get(0).equals("null")){
            list.remove(0);
            return null;
        }
        TreeNode root = new TreeNode(Integer.valueOf(list.get(0)));
        list.remove(0);
        root.left = reDeserialize(list);
        root.right = reDeserialize(list);
        return root;
    }
}
// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));
```

方法 2：BFS（15ms）

https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/solution/die-dai-ban-ben-bfsxu-lie-hua-jie-wei-bu-e-wai-zen/

```
public class Codec {
    // Encodes a tree to a single string.
```

```
    public String serialize(TreeNode root) {
        if(root == null) return new String();
        StringBuilder res = new StringBuilder();
        res.append('[');
        Deque<TreeNode> que = new LinkedList<>();
        que.add(root);
        TreeNode cur;
        int num = 1;//此时队列非空结点为1
        while(num != 0){
            cur = que.remove();
            if(cur != null){
                num--;//非空结点-1
                res.append(cur.val);
                res.append(',');
                if(cur.left != null){
                    que.add(cur.left);
                    num++;//非空结点+1
                }
                else {
                    que.add(null);
                }
                if(cur.right != null){
                    que.add(cur.right);
                    num++;//非空结点+1
                }
                else {
                    que.add(null);
                }
            }
            else {
                res.append("null,");
            }
        }
        res.setLength(res.length() - 1);//去掉最后一个逗号
        res.append(']');
        return res.toString();
    }
    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data.length() < 3) return null;
        String[] strs = data.substring(1, data.length()-1).split(",");
//因为头尾有括号
        TreeNode root = new TreeNode(Integer.valueOf(strs[0]));
        Deque<TreeNode> que = new LinkedList<>();
```

```
            que.add(root);
            TreeNode cur;
            int idx = 0;
            while(true){
                cur=que.remove();
                if(++idx >= strs.length) break;
                if(strs[idx].compareTo("null") != 0) {
                    cur.left = new TreeNode(Integer.valueOf(strs[idx]));
                    que.add(cur.left);
                }
                if(++idx >= strs.length) break;
                if(strs[idx].compareTo("null") != 0) {
                    cur.right=new TreeNode(Integer.valueOf(strs[idx]));
                    que.add(cur.right);
                }
            }
            return root;
        }
}
```

以下是方法 3 的链接

https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/solution/er-cha-shu-de-xu-lie-hua-yu-fan-xu-lie-hua-by-le-2/

思路和算法

我们也可以这样表示一颗二叉树：

- 如果当前的树为空，则表示为 `X`
- 如果当前的树不为空，则表示为 `(<LEFT_SUB_TREE>)CUR_NUM(RIGHT_SUB_TREE)`，其中：
  - `<LEFT_SUB_TREE>` 是左子树序列化之后的结果
  - `<RIGHT_SUB_TREE>` 是右子树序列化之后的结果
  - `CUR_NUM` 是当前节点的值

根据这样的定义，我们很好写出序列化的过程，后序遍历这颗二叉树即可，那如何反序列化呢？根据定义，我们可以推导出这样的巴科斯范式（BNF）：

```
T -> (T) num (T) | X
```

它的意义是：用 `T` 代表一棵树序列化之后的结果，`|` 表示 `T` 的构成为 `(T) num (T)` 或者 `X`，`|` 左边是对 `T` 的递归定义，右边规定了递归终止的边界条件。

因为：

- `T` 的定义中，序列中的第一个字符要么是 `X`，要么是 `(`，所以这个定义是不含左递归的
- 当我们开始解析一个字符串的时候，如果开头是 `X`，我们就知道这一定是解析一个「空树」的结构，如果开头是 `(`，我们就知道需要解析 `(T) num (T)` 的结构，因此这里两种开头和两种解析方法一一对应，可以确定这是一个无二义性的文法
- 我们只需要通过开头的第一个字母是 `X` 还是 `(` 来判断使用哪一种解析方法

所以这个文法是 LL(1) 型文法，**如果你不知道什么是 LL(1) 型文法也没有关系，你只需要知道它定义了一种递归的方法来反序列化，也保证了这个方法的正确性——我们可以设计一个递归函数：**

- 这个递归函数传入两个参数，带解析的字符串和当前当解析的位置 `ptr`，`ptr` 之前的位置是已经解析的，`ptr` 和 `ptr` 后面的字符串是待解析的
- 如果当前位置为 `X` 说明解析到了一棵空树，直接返回
- 否则当前位置一定是 `(`，对括号内部按照 `(T) num (T)` 的模式解析

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：序列化时做了一次遍历，渐进时间复杂度为 $O(n)$。反序列化时，在解析字符串的时候 `ptr` 指针对字符串做了一次顺序遍历，字符串长度为 $O(n)$，故这里的渐进时间复杂度为 $O(n)$。
- 空间复杂度：考虑递归使用的栈空间的大小，这里栈空间的使用和递归深度有关，递归深度又和二叉树的深度有关，在最差情况下，二叉树退化成一条链，故这里的渐进空间复杂度为 $O(n)$。

方法 3：括号表示编码+递归下降解码（23ms）

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {
    public String serialize(TreeNode root) {
        if (root == null) {
            return "X";
        }
        String l = "(" + serialize(root.left) + ")";
        String r = "(" + serialize(root.right) + ")";
        return  l + root.val + r;
    }

    public TreeNode deserialize(String data) {
        int[] ptr = {0};
        return parse(data, ptr);
    }

    public TreeNode parse(String data, int[] ptr) {
        if (data.charAt(ptr[0]) == 'X') {
            ++ptr[0];
            return null;
        }
        TreeNode cur = new TreeNode(0);
        cur.left = parseSubtree(data, ptr);
        cur.val = parseInt(data, ptr);
        cur.right = parseSubtree(data, ptr);
        return cur;
    }

    public TreeNode parseSubtree(String data, int[] ptr) {
        ++ptr[0]; // 跳过左括号
```

```
            TreeNode subtree = parse(data, ptr);
            ++ptr[0]; // 跳过右括号
            return subtree;
        }

        public int parseInt(String data, int[] ptr) {
            int x = 0, sgn = 1;
            if (!Character.isDigit(data.charAt(ptr[0]))) {
                sgn = -1;
                ++ptr[0];
            }
            while (Character.isDigit(data.charAt(ptr[0]))) {
                x = x * 10 + data.charAt(ptr[0]++) - '0';
            }
            return x * sgn;
        }
}
```

14、序列化和反序列化 N 叉树（428）

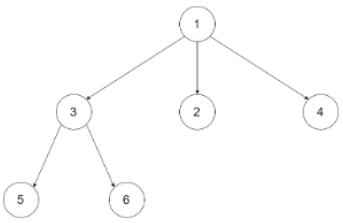
428. 序列化和反序列化 N 叉树

难度 困难 26 收藏 分享 切换为英文 关注 反馈

序列化是指将一个数据结构转化为位序列的过程，因此可以将其存储在文件中或内存缓冲区中，以便稍后在相同或不同的计算机环境中恢复结构。

设计一个序列化和反序列化 N 叉树的算法。一个 N 叉树是指每个节点都有不超过 N 个孩子节点的有根树。序列化 / 反序列化算法的算法实现没有限制。你只需要保证 N 叉树可以被序列化为一个字符串并且该字符串可以被反序列化成原树结构即可。

例如，你需要序列化下面的 3-叉 树。



为 [1 [3[5 6] 2 4]]。你不需要以这种形式完成，你可以自己创造和实现不同的方法。

注意：

1. N 的范围在 [1, 1000]
2. 不要使用类成员 / 全局变量 / 静态变量来存储状态。你的序列化和反序列化算法应是无状态的。

https://leetcode-cn.com/problems/serialize-and-deserialize-n-ary-tree/solution/shi-yong-stack-ji-lu-mei-yi-ge-fu-jie-dian-dfs-by-/

解题思路

根据题目提示，至少有两种 encode 形式，而我选择与 Json 字符串类似的、具有层次信息的一种：“1 [3 [5 6] 2 4]”。这样就有如下处理思路：

1. 使用栈每一个 “[” 前的节点，作为父节点（因为可能有很多层级，便要纪律很多个父节点）；
2. 将 “[” 与 “]” 之间的节点放到父节点的 “children” 列表中；
3. 遇到 “]” 时，将该父节点出栈，尝试处理与该父节点同一层级、或上一层级的节点。

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) {
        val = _val;
    }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Codec {
    // Encodes a tree to a single string.
    public String serialize(Node root) {
        StringBuilder sb = new StringBuilder();
        if (root == null) {
            return sb.toString();
        }
        encode(root, sb);
        return sb.toString();
    }
    private void encode(Node node, StringBuilder sb) {
        if (node == null) {
            return;
        }
        sb.append(node.val);
        sb.append(" ");

        boolean hasChildren = !node.children.isEmpty();

        // only append "[ ]" when the node has children
        if (hasChildren) {
            sb.append("[ ");
        }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            for (int i = 0; i < node.children.size(); i++) {
                Node children = node.children.get(i);
                encode(children, sb);
                if (i == node.children.size() - 1) {
                    sb.deleteCharAt(sb.length() - 1);
                }
            }
            if (hasChildren) {
                sb.append(" ] ");
            }
        }

        // Decodes your encoded data to tree.
        public Node deserialize(String data) {
            if (data.isEmpty()) {
                return null;
            }
            String[] strings = data.split(" ");
            Stack<Node> stack = new Stack<Node>();
            Node root = null;
            Node cur = null;

            for (String s : strings) {
                if (s.equals("[")) {
                    stack.push(cur);
                } else if (s.equals("]")) {
                    stack.pop();
                } else {
                    Node node = new Node(Integer.valueOf(s));
                    node.children = new LinkedList<Node>();
                    if (root == null) {
                        root = node;
                    } else {
                        Node parent = stack.peek();
                        parent.children.add(node);
                    }
                    cur = node;
                }
            }
            return root;
        }
    }
    // Your Codec object will be instantiated and called as such:
    // Codec codec = new Codec();
```

```
// codec.deserialize(codec.serialize(root));
```

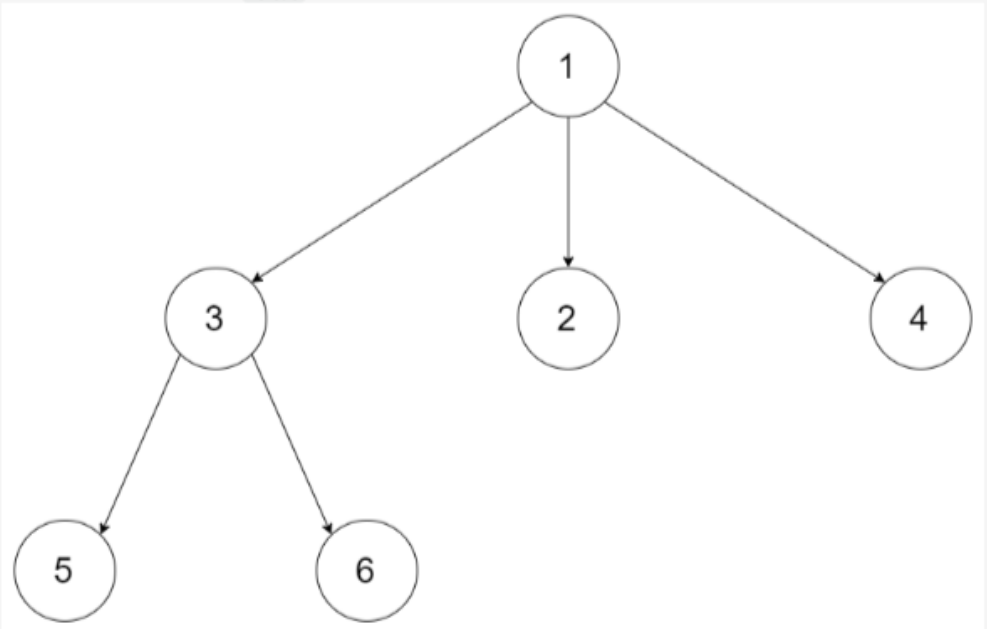
lintcode 版本 1532

1532. 序列化和反序列N叉树

序列化是将一个数据结构或对象转换成比特流的过程，以便将其存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一或另一计算机环境中重建。

设计一个算法来序列化和反序列化一个N叉树。一棵N叉树是一棵有根树，其中每个节点的子节点不超过N个。序列化/反序列化算法的实现方式没有限制。您只需要确保N叉树可以序列化为字符串，并且该字符串可以反序列化为原始树结构。

例如，你可以序列化如下的 3叉树



为 [1 [3[5 6] 2 4]]。你不一定要遵循这种格式，发挥创意，自己想出不同的方法。

样例

样例 1:

输入：{1,3,2,4#2#3,5,6#4#5#6}
输出：{1,3,2,4#2#3,5,6#4#5#6}
解释：如上图

样例 2:

输入：{1,3,2#2#3}
输出：{1,3,2#2#3}
解释：

```
     1
    / \
   3   2
```

注意事项

图模型说明:

https://www.lintcode.com/help/graph

攻略

https://www.jiuzhang.com/solution/serialize-and-deserialize-n-ary-tree/#tag-lang-java

```
/**
 * Definition for Directed graph.
```

```
 * class DirectedGraphNode {
 *         int label;
 *         ArrayList<DirectedGraphNode> neighbors;
 *                 DirectedGraphNode(int x) { label = x; neighbors = new
ArrayList<DirectedGraphNode>(); }
 * };
 */

public class Solution {
    public int pos = 1;
    public String dfs(DirectedGraphNode root) {
        String ans="";
        if(root == null)
            return ans;
        ans += "[";
        ans += String.valueOf(root.label);
        for(int i = 0; i < root.neighbors.size() ; i++) {
                ans += dfs(root.neighbors.get(i));
        }
        ans += "]";
        return ans;

    }
    public UndirectedGraphNode solve(String data) {
        int num = 0;
            while(data.charAt(pos) >= '0' && data.charAt(pos) <= '9') {
                num *= 10;
                num += data.charAt(pos) - '0';
                pos++;
            }
            UndirectedGraphNode node =  new UndirectedGraphNode(num);
            while(pos < data.length()) {
                if(data.charAt(pos) == '[' ) {
                    ++pos;
                    node.neighbors.add(solve(data));
                }
                else if(data.charAt(pos) == ']') {
                    pos++;
                    return node;
                }
            }
        return null;
    }
    public String serialize(ArrayList<DirectedGraphNode> nodes) {
```

禁止商用
公众号：
yes的练级攻略

```
            String ans="";
            if(nodes.size() == 0)
                return ans;
            return dfs(nodes.get(0));
        }
        public UndirectedGraphNode deserialize(String data) {
            if(data.length() == 0)
                return null;
            return solve(data);
        }
}
```

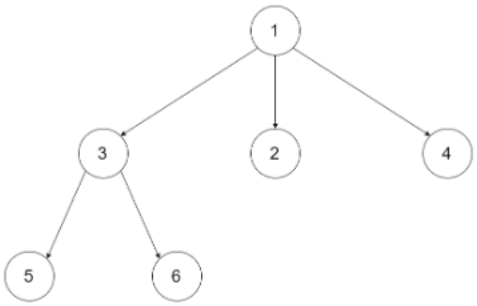
15、N 叉树的前序遍历（589）

589. N叉树的前序遍历

难度 简单 94 收藏 分享 切换为英文 关注

给定一个 N 叉树，返回其节点值的*前序遍历*。

例如，给定一个 3叉树：



返回其前序遍历: [1, 3, 5, 6, 2, 4]。

说明: 递归法很简单，你可以使用迭代法完成此题吗?

https://leetcode-cn.com/problems/n-ary-tree-preorder-traversal/solution/marveljian-dan-de-xue-xi-bi-ji-589-by-tyanyonecanc/

方法 1：递归

```
class Solution {
    List<Integer> res;
    public List<Integer> preorder(Node root) {
        res = new LinkedList<>();
        dfs(root);
        return res;
    }
    public void dfs(Node root) {
        if (root == null) return;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            res.add(root.val);
            for (Node cur : root.children) {
                dfs(cur);
            }
        }
    }
```

https://leetcode-cn.com/problems/n-ary-tree-preorder-traversal/solution/ncha-shu-de-qian-xu-bian-li-by-leetcode/

方法一：迭代

由于递归实现 `N` 叉树的前序遍历较为简单，因此我们只讲解如何使用迭代的方法得到 `N` 叉树的前序遍历。

我们使用一个栈来帮助我们得到前序遍历，需要保证栈顶的节点就是我们当前遍历到的节点。我们首先把根节点入栈，因为根节点是前序遍历中的第一个节点。随后每次我们从栈顶取出一个节点 `u`，它是我们当前遍历到的节点，并把 `u` 的所有子节点逆序推入栈中。例如 `u` 的子节点从左到右为 `v1, v2, v3`，那么推入栈的顺序应当为 `v3, v2, v1`，这样就保证了下一个遍历到的节点（即 `u` 的第一个子节点 `v1`）出现在栈顶的位置。

复杂度分析

- 时间复杂度：时间复杂度：$O(M)$，其中 M 是 `N` 叉树中的节点个数。每个节点只会入栈和出栈各一次。
- 空间复杂度：$O(M)$。在最坏的情况下，这棵 `N` 叉树只有 `2` 层，所有第 `2` 层的节点都是根节点的孩子。将根节点推出栈后，需要将这些节点都放入栈，共有 $M-1$ 个节点，因此栈的大小为 $O(M)$。

方法 2：迭代

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) {
        val = _val;
    }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
    public List<Integer> preorder(Node root) {
        List<Integer> res = new LinkedList<>();
        Stack<Node> sta = new Stack<>();
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (root == null) return res;
        sta.push(root);
        while (!sta.isEmpty()) {
            Node cur = sta.pop();
            res.add(cur.val);
            Collections.reverse(cur.children);
            for (Node i : cur.children) {
                sta.push(i);
            }
        }
        return res;
    }
}
```

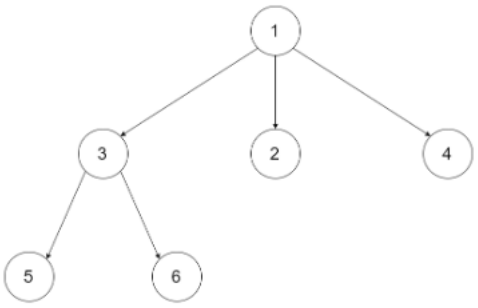
16、N 叉树的后序遍历（590）

590. N叉树的后序遍历

难度 简单 96 收藏 分享 切换为英文 关

给定一个 N 叉树，返回其节点值的*后序遍历*。

例如，给定一个 3叉树 :



返回其后序遍历: [5, 6, 3, 2, 4, 1].

说明: 递归法很简单，你可以使用迭代法完成此题吗?

https://leetcode-cn.com/problems/n-ary-tree-postorder-traversal/solution/ncha-shu-de-hou-xu-bian-li-javade-liang-chong-zuo-/

方法 1：递归

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) {
```

```
        val = _val;
    }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
    List<Integer> res;
    public List<Integer> postorder(Node root) {
        res = new LinkedList<>();
        dfs(root);
        return res;
    }
    public void dfs(Node root) {
        if (root == null) return;
        for (Node cur : root.children) {
            dfs(cur);
        }
        res.add(root.val);//与前序遍历的不同之处
    }
}
```

https://leetcode-cn.com/problems/n-ary-tree-postorder-traversal/solution/ncha-shu-de-hou-xu-bian-li-by-leetcode/

方法一：迭代

由于递归实现 `N` 叉树的后序遍历较为简单，因此我们只讲解如何使用迭代的方法得到 `N` 叉树的后序遍历。

在后序遍历中，我们会先遍历一个节点的所有子节点，再遍历这个节点本身。例如当前的节点为 `u`，它的子节点为 `v1, v2, v3` 时，那么后序遍历的结果为 `[children of v1], v1, [children of v2], v2, [children of v3], v3, u`，其中 `[children of vk]` 表示以 `vk` 为根节点的子树的后序遍历结果（不包括 `vk` 本身）。我们将这个结果反转，可以得到 `u, v3, [children of v3]', v2, [children of v2]', v1, [children of v1]'`，其中 `[a]'` 表示 `[a]` 的反转。此时我们发现，结果和前序遍历非常类似，只不过前序遍历中对子节点的遍历顺序是 `v1, v2, v3`，而这里是 `v3, v2, v1`。

因此我们可以使用和 N叉树的前序遍历 相同的方法，使用一个栈来得到后序遍历。我们首先把根节点入栈。当每次我们从栈顶取出一个节点 `u` 时，就把 `u` 的所有子节点顺序推入栈中。例如 `u` 的子节点从左到右为 `v1, v2, v3`，那么推入栈的顺序应当为 `v1, v2, v3`，这样就保证了下一个遍历到的节点（即 `u` 的第一个子节点 `v3`）出现在栈顶的位置。在遍历结束之后，我们把遍历结果反转，就可以得到后序遍历。

复杂度分析

- 时间复杂度：时间复杂度：$O(M)$，其中 M 是 `N` 叉树中的节点个数。每个节点只会入栈和出栈各一次。
- 空间复杂度：$O(M)$。在最坏的情况下，这棵 `N` 叉树只有 `2` 层，所有第 `2` 层的节点都是根节点的孩子。将根节点推出栈后，需要将这些节点都放入栈，共有 $M-1$ 个节点，因此栈的大小为 $O(M)$。

方法 2：迭代

```
/*
// Definition for a Node.
class Node {
```

```
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) {
        val = _val;
    }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
    public List<Integer> postorder(Node root) {
        LinkedList<Integer> res = new LinkedList<>(); //这边有个不同
        Stack<Node> sta = new Stack<>();
        if (root == null) return res;
        sta.push(root);
        while (!sta.isEmpty()) {
            Node cur = sta.pop();
            res.addFirst(cur.val); //这边有个不同，并且下面无需反转
            for (Node i : cur.children) {
                if (i != null) sta.push(i);
            }
        }
        return res;
    }
}
```

17、N 叉树的层序遍历（429）

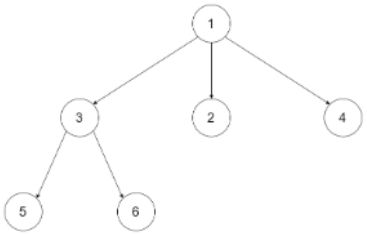
更多硬核内容就在公众号「yes 的练级攻略」

429. N叉树的层序遍历

难度 中等 106 收藏 分享 切换为英文 关注 反馈

给定一个 N 叉树，返回其节点值的*层序遍历*。(即从左到右，逐层遍历)。

例如，给定一个 3叉树 :



返回其层序遍历:

```
[
     [1],
     [3,2,4],
     [5,6]
]
```

说明:

1. 树的深度不会超过 1000。
2. 树的节点总数不会超过 5000。

https://leetcode-cn.com/problems/n-ary-tree-level-order-traversal/solution/ncha-shu-de-ceng-xu-bian-li-by-leetcode/

方法三：递归

算法:

我们可以使用递归来解决这个问题，通常我们不能使用递归进行广度优先搜索。这是因为广度优先搜索基于队列，而递归运行时使用堆栈，适合深度优先搜索。但是在本题中，我们可以以不同的顺序添加到最终列表中，只要我们知道节点在哪一层并确保在那一层的列表顺序正确就可以了。

Python | Java

```python
def levelOrder(self, root: 'Node') -> List[List[int]]:

    def traverse_node(node, level):
        if len(result) == level:
            result.append([])
        result[level].append(node.val)
        for child in node.children:
            traverse_node(child, level + 1)

    result = []

    if root is not None:
        traverse_node(root, 0)
    return result
```

复杂度分析

- 时间复杂度：$O(n)$。n 指的是节点的数量
- 空间复杂度：正常情况 $O(\log n)$，最坏情况 $O(n)$。运行时在堆栈上的空间。

方法 1：递归

更多硬核内容就在公众号「yes 的练级攻略」

```java
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) {
        val = _val;
    }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
    private List<List<Integer>> result = new ArrayList<>();
    public List<List<Integer>> levelOrder(Node root) {
        if (root != null) traverseNode(root, 0);
        return result;
    }
    private void traverseNode(Node node, int level) {
        if (result.size() <= level) {
            result.add(new ArrayList<>());
        }
        result.get(level).add(node.val);
        for (Node child : node.children) {
            traverseNode(child, level + 1);
        }
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

算法：
上面的基本算法在一定程度上帮助了我们解决这道题目，但是我们还需要保存每一层的列表，并且在根节点为空时正常工作。

再构造下一层的列表时，我们需要创建新的子列表，然后将该层的所有节点的值插入到列表中。一个很好的方法时在 `while` 循环体开始时记录队列的当前大小 `size` 。然后用另一个循环来处理 `size` 数量的节点。这样可以保证 `while` 循环在每一次迭代处理一层。

使用队列十分重要，如果使用 `Vector` ， `List` ， `Array` 的话，我们删除元素需要 $O(n)$ 的时间复杂度。而队列删除元素只需要 $O(1)$ 的时间。

Python | Java

```
// This code is a modified version of the code posted by
// #zzzliu on the discussion forums.
class Solution {

    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                Node node = queue.poll();
                level.add(node.val);
                queue.addAll(node.children);
            }
            result.add(level);
        }
        return result;
    }
}
```

复杂度分析

- 时间复杂度：$O(n)$。n 指的是节点的数量。
- 空间复杂度：$O(n)$。

方法 2：迭代

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) {
        val = _val;
    }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
```

更多硬核内容就在公众号「yes 的练级攻略」

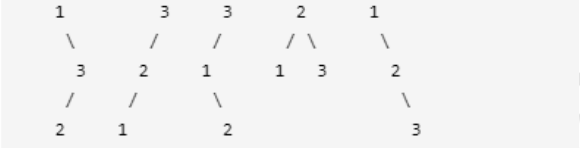
```
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            List<Integer> level = new ArrayList<>();
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                Node node = queue.poll();
                level.add(node.val);
                queue.addAll(node.children);
            }
            result.add(level);
        }
        return result;
    }
}
```

18、不同的二叉搜索树（96）

给定一个整数 *n*，求以 1 ... *n* 为节点组成的二叉搜索树有多少种?

示例:

输入: 3
输出: 5
解释:
给定 *n* = 3, 一共有 5 种不同结构的二叉搜索树:



https://leetcode-cn.com/problems/unique-binary-search-trees/solution/bu-tong-de-er-cha-sou-suo-shu-by-leetcode-solution/

方法一：动态规划

思路

给定一个有序序列 $1 \cdots n$，为了构建出一棵二叉搜索树，我们可以遍历每个数字 i，将该数字作为树根，将 $1 \cdots (i-1)$ 序列作为左子树，将 $(i+1) \cdots n$ 序列作为右子树。接着我们可以按照同样的方式递归构建左子树和右子树。

在上述构建的过程中，由于根的值不同，因此我们能保证每棵二叉搜索树是唯一的。

由此可见，原问题可以分解成规模较小的两个子问题，且子问题的解可以复用。因此，我们可以想到使用动态规划来求解本题。

算法

题目要求是计算不同二叉搜索树的个数。为此，我们可以定义两个函数：

1. $G(n)$: 长度为 n 的序列能构成的不同二叉搜索树的个数。
2. $F(i, n)$: 以 i 为根、序列长度为 n 的不同二叉搜索树个数 $(1 \leq i \leq n)$。

可见，$G(n)$ 是我们求解需要的函数。

更多硬核内容就在公众号「yes 的练级攻略」

稍后我们将看到，$G(n)$ 可以从 $F(i,n)$ 得到，而 $F(i,n)$ 又会递归地依赖于 $G(n)$。

首先，根据上一节中的思路，不同的二叉搜索树的总数 $G(n)$，是对遍历所有 i $(1 \le i \le n)$ 的 $F(i,n)$ 之和。换言之：

$$G(n) = \sum_{i=1}^{n} F(i,n) \qquad (1)$$

对于边界情况，当序列长度为 1（只有根）或为 0（空树）时，只有一种情况，即：

$$G(0) = 1, \qquad G(1) = 1$$

给定序列 $1 \cdots n$，我们选择数字 i 作为根，则根为 i 的所有二叉搜索树的集合是左子树集合和右子树集合的**笛卡尔积**，对于笛卡尔积中的每个元素，加上根节点之后形成完整的二叉搜索树，如下图所示：

举例而言，创建以 3 为根、长度为 7 的不同二叉搜索树，整个序列是 $[1,2,3,4,5,6,7]$，我们需要从左子序列 $[1,2]$ 构建左子树，从右子序列 $[4,5,6,7]$ 构建右子树，然后将它们组合（即笛卡尔积）。

对于这个例子，不同二叉搜索树的个数为 $F(3,7)$。我们将 $[1,2]$ 构建不同左子树的数量表示为 $G(2)$，从 $[4,5,6,7]$ 构建不同右子树的数量表示为 $G(4)$，注意到 $G(n)$ 和序列的内容无关，只和序列的长度有关。于是，$F(3,7) = G(2) \cdot G(4)$。因此，我们可以得到以下公式：

$$F(i,n) = G(i-1) \cdot G(n-i) \qquad (2)$$

将公式 (1)，(2) 结合，可以得到 $G(n)$ 的递归表达式：

$$G(n) = \sum_{i=1}^{n} G(i-1) \cdot G(n-i) \qquad (3)$$

至此，我们从小到大计算 G 函数即可，因为 $G(n)$ 的值依赖于 $G(0) \cdots G(n-1)$。

复杂度分析

- 时间复杂度：$O(n^2)$，其中 n 表示二叉搜索树的节点个数。$G(n)$ 函数一共有 n 个值需要求解，每次求解需要 $O(n)$ 的时间复杂度，因此总时间复杂度为 $O(n^2)$。
- 空间复杂度：$O(n)$。我们需要 $O(n)$ 的空间存储 G 数组。

方法 1：动态规划

```
class Solution {
    public int numTrees(int n) {
        if (n == 0) return 0;
        int[] G = new int[n+1];
        G[0] = 1;
        G[1] = 1;
        for (int i = 2; i <=n; i++) {
            for (int j = 1; j <= i; j++) {
                G[i] += G[j - 1] * G[i - j];
            }
        }
```

```
            return G[n];
        }
    }
```

方法二：数学

思路与算法

事实上我们在方法一中推导出的 $G(n)$ 函数的值在数学上被称为卡塔兰数 C_n。卡塔兰数更便于计算的定义如下：

$$C_0 = 1, \qquad C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

证明过程可以参考上述文献，此处不再赘述。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 表示二叉搜索树的节点个数。我们只需要循环遍历一次即可。
- 空间复杂度：$O(1)$。我们只需要常数空间存放若干变量。

方法 2：数学定理

```
class Solution {
    public int numTrees(int n) {
        // 提示：我们在这里需要用 long 类型防止计算过程中的溢出
        long C = 1;
        for (int i = 0; i < n; ++i) {
            C = C * 2 * (2 * i + 1) / (i + 2);
        }
        return (int) C;
    }
}
```

19、不同的二叉搜索树 2（95）

更多硬核内容就在公众号「yes 的练级攻略」

95. 不同的二叉搜索树 II

难度 中等 483 收藏 分享 切换为英文 关注 反馈

给定一个整数 *n*，生成所有由 1 ... *n* 为节点所组成的 **二叉搜索树** 。

示例：

```
输入：3
输出：
[
  [1,null,3,2],
  [3,2,null,1],
  [3,1,null,null,2],
  [2,1,3],
  [1,null,2,null,3]
]
解释：
以上的输出对应以下 5 种不同结构的二叉搜索树：

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

提示：

- `0 <= n <= 8`

https://leetcode-cn.com/problems/unique-binary-search-trees-ii/solution/dong-tai-gui-hua-on5jing-dian-qia-te-lan-shu-dpti-/

从*96.不同的二叉搜索树*我们知道了1...n的不同二叉搜索树的个数是卡特兰数，从本题我们就体会一下卡特兰数的递推方程的结构

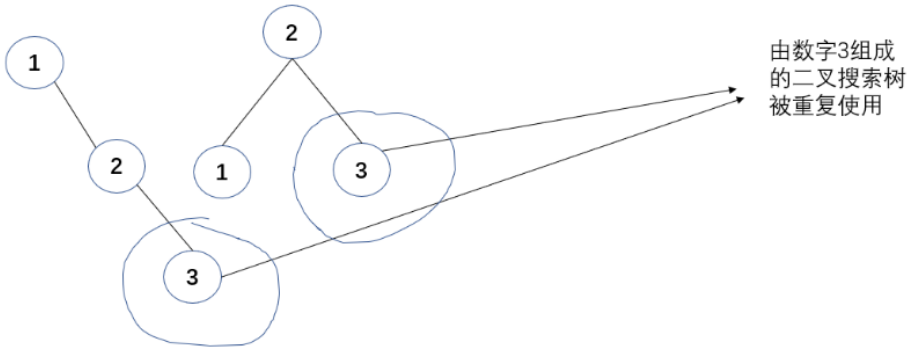
首先用递归的思想去生成1...n组成的不同的二叉搜索树：

一、当1为根节点时，左子树为空，右子树上有（n-1）个节点，这（n-1）个节点即2...n组成的不同的二叉搜索树

二、当i为根节点时（1<i<n),左子树是由1...i-1这(i-1)个节点组成的;右子树是由i+1...n这（n-i)个节点构成的

三、当n为根节点时，右子树为空，左子树上有（n-1）个节点，这（n-1）个节点是1...n-1组成的不同的二叉搜索树

我们可以注意到，在二叉搜索树生成的过程中，同一个子树被使用了多次。
如图所示：



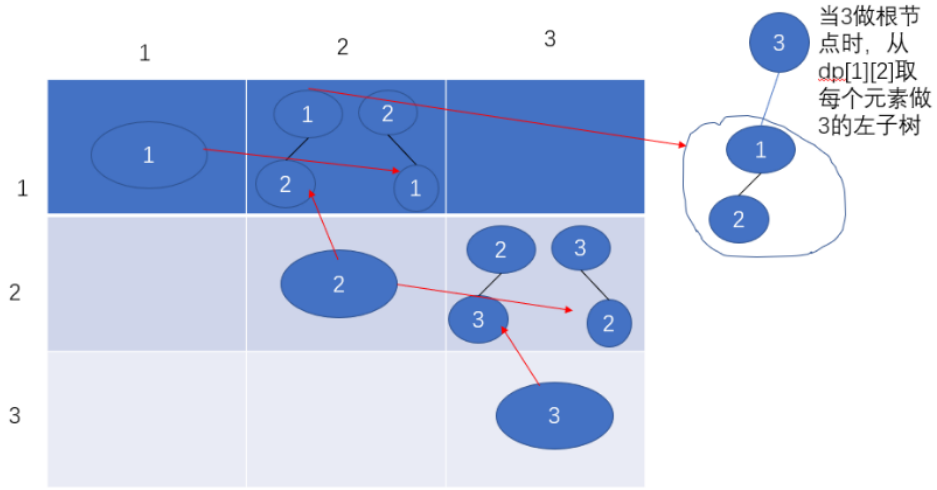
那么我们就可用DP的方法存储被重复计算的子问题的解：

设dp[i][j]存储i...j的不同的二叉搜索树，

当只有一个节点时，它自己就是二叉搜索树，所以初始化主对角线上的元素dp[i][i]=i;（这里用节点的数值代表节点本身）

当节点个数r>=2时，遍历每一个节点k作为当前树的根节点，从DP表中读取dp[i][k-1]中的每个元素作为它的左子树，读取dp[k+1][j]的每个元素作为他的右子树,表示成状态转移方程即：
dp[i][j]=sum(dp[i][k-1]+k+dp[k+1][j]),i<=k<=j

如下所示：



方法 1：卡特兰数

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
```

```
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public List<TreeNode> generateTrees(int n) {
        if (n == 0) {
            return new ArrayList<TreeNode>();
        }
        ArrayList<TreeNode> [][] dp=new ArrayList[n+1][n+1];//从下标 1 开始使用
        for(int i = 1; i <= n; i++) {
            dp[i][i] = new ArrayList<TreeNode>();
            dp[i][i].add(new TreeNode(i));
        }//1 个节点的情况
        //开始 dp:
        for(int r = 2;r <= n; r++){//r 个节点
            for(int i = 1;i <= n-r+1; i++){//从数字 i 开始
                int j = i + r - 1;//到数字 j
                 dp[i][j] = new ArrayList<TreeNode>();
                //生成 i...j 的二叉搜索树
                TreeNode node_root_i=new TreeNode(i);//以 i 为根节点,左子树为 null，只有右子树
                for (TreeNode node : dp[i+1][j]) {
                    node_root_i.left = null;
                    node_root_i.right = node;
                    dp[i][j].add(node_root_i);
                    node_root_i = new TreeNode(i);
                }
                for(int k = i+1; k <= j - 1; k++){//选择节点 k 作为根节点，i<k<j
                    TreeNode node_root_k = new TreeNode(k);//先创建根节点
                    //左子树为 i...k-1 组成的二叉搜索树
                    //右子树为 k+1...j 的二叉搜索树
                      for (TreeNode t_left : dp[i][k - 1]) { //左右子树的组合用了双重 for 循环，这里不知道能不能优化一下
                          for (TreeNode t_right : dp[k + 1][j]) {
                            node_root_k.left = t_left;
                            node_root_k.right = t_right;
                            dp[i][j].add(node_root_k);
```

```
                        node_root_k = new TreeNode(k);
                    }
                }
                TreeNode node_root_j = new TreeNode(j);//以 j 为根节点,右子树为 null，只有左子树
                for (TreeNode node : dp[i][j-1]) {
                    node_root_j.right = null;
                    node_root_j.left = node;
                    dp[i][j].add(node_root_j);
                    node_root_j = new TreeNode(j);
                }
            }
        }
        return dp[1][n];
    }
}
```

方法 2：递归

解题思路

本题依旧是树的题目，并要求定一个整数 *n*，生成所有由 1 ... *n* 为节点所组成的**二叉搜索树（二叉搜索树：树当中的任何子树的左子树的节点都小于根节点，任何子树的右子树的节点都大于根节点）**。根据搜索二叉树的定义，我们从1~n中取第i个数作为根节点，那么他的前i-1（**从1到i-1**）个数为它的左子树，后n-i（**从i+1到n**）个数为它的右子树，最后只需将找好的左子树，与 右子树根据排列的方式进行组合即可找出所有的情况。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
import java.util.LinkedList;
class Solution {
    public List<TreeNode> generateTrees(int n) {
        if(n == 0){
            return new LinkedList<TreeNode>();
```

```
        }

        return generate_trees(1,n);
    }

    public LinkedList<TreeNode> generate_trees(int start, int end){
        LinkedList<TreeNode> result = new LinkedList<>();
        //如果开始下标 start，大于结束下标 end，则表示已经没有点了
        if(start > end){
            result.add(null);
            return result;
        }
        for(int i = start; i <= end; i++){
            //找出（start，i-1）个数作为其根节点左子树(二叉搜索树)
            LinkedList<TreeNode> left = generate_trees(start,i-1);
            //找出（i+1，end）个数作为其根节点右子树(二叉搜索树)
            LinkedList<TreeNode> right = generate_trees(i+1,end);
            //将左子树与右子树分别与根节点相连
            for(TreeNode k : left){
                for(TreeNode x: right){
                    TreeNode node = new TreeNode(i);
                    node.left = k;
                    node.right = x;
                    result.add(node);
                }
            }
        }
        return result;
    }
}
```

本地中可以运行的代码

```
import java.util.*;
//树结构在 Traversal 里面，这题针对 LeetCode 95 和 96
class TreeNode{
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(){}
    TreeNode(int val){
        this.val = val;
    }
    TreeNode(int x, TreeNode left, TreeNode right){
        this.val = val;
        this.left = left;
```

```
        this.right = right;
    }
}
public class DifferentTree {

    public static void main(String[] args) {
        while (true) {
            System.out.println("请给定一个整数 n：");
            Scanner sc = new Scanner(System.in);
            int n = sc.nextInt();

            int num = numTrees(n);
            System.out.print("以 1...n 为节点组成的二叉树种类数为：");
            System.out.println(num);

            List<TreeNode> res = generateTrees(n);
            System.out.println("以 1...n 为节点组成的二叉树具体结构为：");
            printTreeNode(res);
        }
    }

    public  static  int  numTrees(int  n)  {
        if  (n  ==  0)  return  0;
        int[]  G  =  new  int[n+1];
        G[0]  =  1;
        G[1]  =  1;
        for  (int  i  =  2;  i  <=n;  i++)  {
            for  (int  j  =  1;  j  <=  i;  j++)  {
                G[i]  +=  G[j  -  1]  *  G[i  -  j];
            }
        }
        return  G[n];
    }

    public  static List<TreeNode>  generateTrees(int  n)  {
        if (n  ==  0)  return  new  ArrayList<TreeNode>();
        ArrayList<TreeNode>  [][]  dp=new  ArrayList[n+1][n+1];//从下标 1
开始使用
        for(int  i  =  1;  i  <=  n;  i++)  {
            dp[i][i]  =  new  ArrayList<TreeNode>();
            dp[i][i].add(new  TreeNode(i));
        }//1 个节点的情况
        //开始 dp:
        for(int  r  =  2;r  <=  n;  r++){//r 个节点
```

```
            for(int i = 1;i <= n-r+1; i++){//从数字 i 开始
                int j = i + r - 1;//到数字 j
                dp[i][j] = new ArrayList<TreeNode>();
                //生成 i...j 的二叉搜索树
                TreeNode node_root_i=new TreeNode(i);//以 i 为根节点,左子树为 null，只有右子树
                for (TreeNode node : dp[i+1][j]) {
                    node_root_i.left = null;
                    node_root_i.right = node;
                    dp[i][j].add(node_root_i);
                    node_root_i = new TreeNode(i);
                }
                for(int k = i+1; k <= j - 1; k++){//选择节点 k 作为根节点，i<k<j
                    TreeNode node_root_k = new TreeNode(k);//先创建根节点
                    //左子树为 i...k-1 组成的二叉搜索树
                    //右子树为 k+1...j 的二叉搜索树
                    for (TreeNode t_left : dp[i][k - 1]) { //左右子树的组合用了双重 for 循环，这里不知道能不能优化一下
                        for (TreeNode t_right : dp[k + 1][j]) {
                            node_root_k.left = t_left;
                            node_root_k.right = t_right;
                            dp[i][j].add(node_root_k);
                            node_root_k = new TreeNode(k);
                        }
                    }
                }
                TreeNode node_root_j = new TreeNode(j);//以 j 为根节点，右子树为 null，只有左子树
                for (TreeNode node : dp[i][j-1]) {
                    node_root_j.right = null;
                    node_root_j.left = node;
                    dp[i][j].add(node_root_j);
                    node_root_j = new TreeNode(j);
                }
            }
        }
        return dp[1][n];
    }
    public static void printTreeNode (List<TreeNode> list) {
        String[] res = new String[list.size()];
        for (int i = 0; i < list.size(); i++){
            TreeNode tmp = list.get(i);
```

```
            List<String> arrList = levelOrder (tmp);
            res[i] = printList(arrList);
            System.out.println(res[i]);
        }
    }
//层序遍历
    public static List<String> levelOrder(TreeNode root) {
        List<String> res = new ArrayList<>();
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while (! queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; ++i) {
                TreeNode cur = queue.peek();
                res.add(cur == null ? "null" : String.valueOf(cur.val));
                queue.poll();
                if (cur == null){
                    continue;
                }
                queue.offer(cur.left);
                queue.offer(cur.right);
            }
        }
        return res;
    }
    public static String printList (List<String> list) {
        StringBuilder res = new StringBuilder();
        for (int i = list.size() - 1; i >= 0; i--) {
            if ("null".equals(list.get(i))) {
                continue;
            }
            for (int j = 0; j < i + 1; j++) {
                res.append(list.get(j)).append(" ");
            }
            break;
        }
        return res.toString().trim();
    }
}
```

20、后继者（面试题 04.06）【找到二叉搜索树中指定节点的下一个节点】

更多硬核内容就在公众号「yes 的练级攻略」

面试题 04.06. 后继者

难度 中等 27 收藏 分享 切换为英文 关注 反馈

设计一个算法，找出二叉搜索树中指定节点的“下一个”节点（也即中序后继）。

如果指定节点没有对应的“下一个”节点，则返回`null`。

示例 1:

```
输入: root = [2,1,3], p = 1

  2
 / \
1   3

输出: 2
```

示例 2:

```
输入: root = [5,3,6,2,4,null,null,1], p = 6

      5
     / \
    3   6
   / \
  2   4
 /
1

输出: null
```

https://leetcode-cn.com/problems/successor-lcci/solution/er-cha-sou-suo-shu-xing-zhi-zhao-shun-xu-hou-ji-by/

1.p存在右子树，那么p的后继就是p.right子树的最左节点
2.p不存在右子树，那么p的后继就是p所在子树的最近左孩子的父节点

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        // p 存在右子树，直接后继就是右子树的最左节点
        if (p.right != null){
            p = p.right;
            while (p.left != null){
                p = p.left;
```

```
            }
            return p;
        }
        // p 不存在右子树
        TreeNode node = root;
        TreeNode res = null;
        while (p != node){
            if (p.val < node.val){
                // node 比 p 大，表示 node 在 p 的后继路径上
                res = node;// 左孩子的父节点
                node = node.left;
            }else {
                // node 比 p 小，表示 node 在 p 的前驱路径上
                node = node.right;
            }
        }
        return res;
    }
}
```

21、验证二叉树（1361）

1361. 验证二叉树

难度 中等 31 收藏 分享 切换为英文 关注 反馈

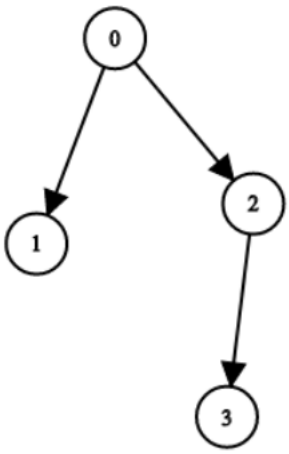
二叉树上有 `n` 个节点，按从 `0` 到 `n - 1` 编号，其中节点 `i` 的两个子节点分别是 `leftChild[i]` 和 `rightChild[i]`。

只有 **所有** 节点能够形成且 **只** 形成 **一颗** 有效的二叉树时，返回 `true`；否则返回 `false`。

如果节点 `i` 没有左子节点，那么 `leftChild[i]` 就等于 `-1`。右子节点也符合该规则。

注意：节点没有值，本问题中仅仅使用节点编号。

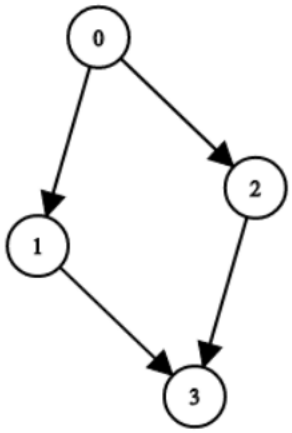
示例 1：



更多硬核内容就在公众号「yes 的练级攻略」

```
输入：n = 4, leftChild = [1,-1,3,-1], rightChild = [2,-1,-1,-1]
输出：true
```

示例 2：

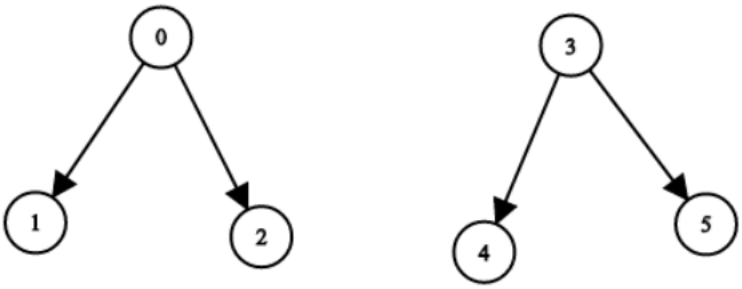



```
输入：n = 4, leftChild = [1,-1,3,-1], rightChild = [2,3,-1,-1]
输出：false
```

示例 3：

```
输入：n = 2, leftChild = [1,0], rightChild = [-1,-1]
输出：false
```

示例 4：




```
输入：n = 6, leftChild = [1,-1,-1,4,-1,-1], rightChild = [2,-1,-1,5,-1,-1]
输出：false
```

提示：

- `1 <= n <= 10^4`
- `leftChild.length == rightChild.length == n`
- `-1 <= leftChild[i], rightChild[i] <= n - 1`

https://leetcode-cn.com/problems/validate-binary-tree-nodes/solution/judgedai-ma-ji-caseyou-cuo-wu-ti-gong-lia-zheng-qu/

二、正解

所以，正确的解法还需要判断一下连通域的数目，本文提供两个可行方案。

1、DFS/BFS

1. 先通过统计各个节点的入度数，排除不满足上述入度数判断的情况。（目前Judge代码和case只做到了这里，所以只能判断出一部分多连通域是错的）
2. 再从root开始扫一遍（下面我用的是dfs）之后判断是否所有的节点都被访问过。

```
class Solution {
    int[] leftChild, rightChild;
    boolean[] visited;
    public boolean validateBinaryTreeNodes(int n, int[] leftChild, int[
] rightChild) {
        // 统计每个结点的父亲数：1. 若父亲数大于 1 则为 false  2.若父亲数等于 0
的不是 1 个则为 false
        int[] indegreeCnt = new int[n];
        for (int i = 0; i < n; i++) {
            if((leftChild[i] != -1 && ++indegreeCnt[leftChild[i]] > 1)
             || (rightChild[i] != -1 && ++indegreeCnt[rightChild[i]] >
1)) {
                return false;
            }
        }
        int root = -1;
        for (int i = 0; i < n; i++){
            if (indegreeCnt[i] == 0) {
                if (root != -1) {
                    return false;
                }
                root = i;
            }
        }
        // 上面的判断可以解决单个连通域的树判断、及部分多连通域情况
        // 例如下面的情况无法判断，需要再判断连通域的个数。从 root 一遍 dfs 之后，
若存在结点未被访问说明多个连通域。
        // 1 <--- 0 <--->  2,  4 --->3
        this.leftChild = leftChild;
        this.rightChild = rightChild;
```

```
            visited = new boolean[n];
            dfs(root);
            for (boolean v: visited) {
                if (!v) {
                    return false;
                }
            }
            return true;
        }
        private void dfs(int v) {
            if (v == -1) return;
            visited[v] = true;
            dfs(leftChild[v]);
            dfs(rightChild[v]);
        }
}
```

22、验证二叉搜索树（98）【合法二叉搜索树（面试题 04.05）】

98. 验证二叉搜索树

难度 中等 623 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含**小于**当前节点的数。
- 节点的右子树只包含**大于**当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

```
输入:
    2
   / \
  1   3
输出: true
```

示例 2:

```
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

https://leetcode-cn.com/problems/validate-binary-search-tree/solution/zhong-xu-bian-li-qing-song-na-xia-bi-xu-miao-dong-/

题目

验证二叉搜索树

思路

中序遍历时，判断当前节点是否大于中序遍历的前一个节点，如果大于，说明满足 BST，继续遍历；否则直接返回 false。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    long pre = Long.MIN_VALUE;
    public boolean isValidBST(TreeNode root) {
        if (root == null) {
            return true;
        }
        if ( ! isValidBST(root.left)) {
            return false;
        }
        if (root.val <= pre) {
            return false;
        }
        pre = root.val;
        return isValidBST(root.right);
    }
}
```

23、验证前序遍历序列二叉搜索树（255）

更多硬核内容就在公众号「yes 的练级攻略」

255. 验证前序遍历序列二叉搜索树

难度 中等 53 收藏 分享 切换为英文 关注 反馈

给定一个整数数组，你需要验证它是否是一个二叉搜索树正确的先序遍历序列。

你可以假定该序列中的数都是不相同的。

参考以下这颗二叉搜索树：

```
     5
    / \
   2   6
  / \
 1   3
```

示例 1：

```
输入：[5,2,6,1,3]
输出：false
```

示例 2：

```
输入：[5,2,1,3,6]
输出：true
```

进阶挑战：

您能否使用恒定的空间复杂度来完成此题？

https://leetcode-cn.com/problems/verify-preorder-sequence-in-binary-search-tree/solution/java-liang-chong-jie-fa-by-npe_tle/

方法二：单调栈

维护一个单调递减栈（从栈底到栈顶），如某一状态下栈元素为$[5,4,3]$。若碰到一个6的时候，说明从左子树（或者没有左子树）到达了右子树，此时将小于6的元素都pop掉，栈变成$[6]$,并且记录一个最小值为5，由于6是右子树，因此6右侧的元素都必须大于5，否则不合法。

复杂度

时间复杂度：$O(n)$

空间复杂度：$O(1)$

说明，这里有人会有疑问，代码里头不是有两重循环吗？时间复杂度不是应该是$O(n^2)$？其实不是的虽然有两重循环，但是这里每个元素最多只会进栈一次和出栈一次，也就是复杂度应该是$O(2*n)$，去掉系数也就是$O(n)$，所有单调栈的题目都是类似的。

```
class Solution {
    // 用单调栈的方式，递减栈，当碰到一个数比栈顶元素大的时候，说明从左子树到了右子树。
    // 此时要删掉左子树的所有节点，并且保留子树的根为最小值，此时遍历的所有右子树的节点都必须大于这个根，否则非法
    public boolean verifyPreorder(int[] preorder) {
        int len = preorder.length;
        int[] stack = new int[len];
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            int top = -1;
            int min = Integer.MIN_VALUE;
            for (int value : preorder) {
                if (value < min) return false;
                while (top > -1 && value > stack[top]) {
                    min = stack[top];
                    top--;
                }
                stack[++top] = value;
            }
            return true;
        }
}
```

24、恢复二叉搜索树（99）

99. 恢复二叉搜索树

难度 困难 252 收藏 分享 切换为英文

二叉搜索树中的两个节点被错误地交换。

请在不改变其结构的情况下，恢复这棵树。

示例 1:

输入: [1,3,null,null,2]

```
   1
  /
 3
  \
   2
```

输出: [3,1,null,null,2]

```
   3
  /
 1
  \
   2
```

面试通用
公众号：
练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

示例 2:

```
输入: [3,1,4,null,null,2]

  3
 / \
1   4
   /
  2

输出: [2,1,4,null,null,3]

  2
 / \
1   4
   /
  3
```

进阶:

- 使用 O(*n*) 空间复杂度的解法很容易实现。
- 你能想出一个只使用常数空间的解决方案吗?

https://leetcode-cn.com/problems/recover-binary-search-tree/solution/hui-fu-er-cha-sou-suo-shu-by-leetcode-solution/

方法二：迭代中序遍历

算法：

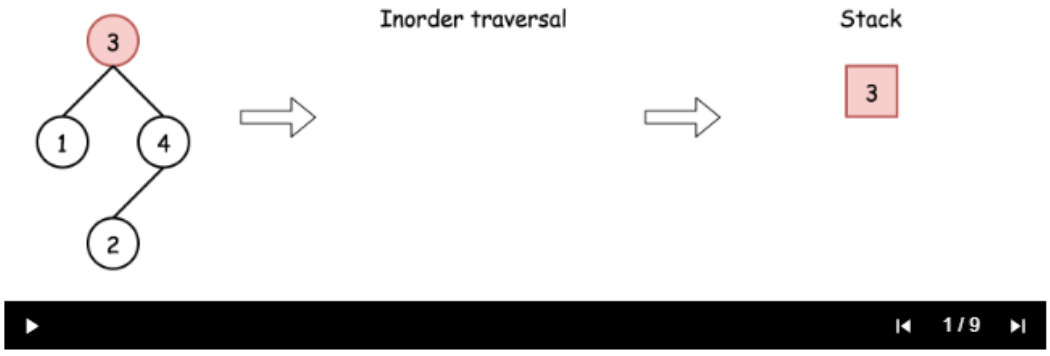
在这里，我们通过迭代构造中序遍历，并在一次遍历中找到交换的节点。

迭代顺序很简单：尽可能的向左走，然后向右走一步，重复一直到结束。

若要找到交换的节点，就记录中序遍历中的最后一个节点 `pred` （即当前节点的前置节点），并与当前节点的值进行比较。如果当前节点的值小于前置节点 `pred` 的值，说明该节点是交换节点之一。

交换的节点只有两个，因此在确定了第二个交换节点以后，可以终止遍历。

这样，就可以直接获取节点（而不仅仅是它们的值），从而实现 $\mathcal{O}(1)$ 的交换时间，大大减少了步骤 3 所需的时间。



在 java 中用 `ArrayDeque` 代替栈。

迭代中序遍历（需要用到 ArrayDeque）3ms

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public void swap(TreeNode a, TreeNode b) {
        int tmp = a.val;
        a.val = b.val;
        b.val = tmp;
    }

    public void recoverTree(TreeNode root) {
        Deque<TreeNode> ad = new ArrayDeque();
        TreeNode x = null, y = null, pred = null;

        while (!ad.isEmpty() || root != null) {
        while (root != null) {
            ad.add(root);
            root = root.left;
        }
        root = ad.removeLast();
        if (pred != null && root.val < pred.val) {
            y = root;
            if (x == null) x = pred;
            else break;
        }
        pred = root;
        root = root.right;
        }

        swap(x, y);
    }
```

}

25、修建二叉搜索树（669）

669. 修剪二叉搜索树

难度 简单 268 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树，同时给定最小边界 L 和最大边界 R。通过修剪二叉搜索树，使得所有节点的值在 [L, R] 中 (R>=L)。你可能需要改变树的根节点，所以结果应当返回修剪好的二叉搜索树的新的根节点。

示例 1:

```
输入:
    1
   / \
  0   2

  L = 1
  R = 2

输出:
    1
     \
      2
```

示例 2:

```
输入:
    3
   / \
  0   4
   \
    2
   /
  1

  L = 1
  R = 3

输出:
      3
     /
   2
  /
 1
```

yes的练级攻略

https://leetcode-cn.com/problems/trim-a-binary-search-tree/solution/marveljian-dan-de-xue-xi-bi-ji-669-by-tyanyonecanc/

递归

题目要求返回BST被修剪后的根结点，那么我们从根结点开始修剪。
如果根结点太小，根结点的左子树的所有结点只会更小，说明根结点及其左子树都应该剪掉，因此直接返回右子树的修剪结果。
如果根结点太大，根结点的右子树的所有结点只会更大，说明根结点及其右子树都应该剪掉，因此直接返回左子树的修剪结果。
如果根结点没问题，则递归地修剪左子结点和右子结点。
如果结点为空，说明无需修剪，直接返回空即可。
左右子结点都修剪完后，返回自身。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是给定的树的全部节点。我们最多访问每个节点一次。
- 空间复杂度：$O(N)$，即使我们没有明确使用任何额外的内存，在最糟糕的情况下，我们递归调用的栈可能与节点数一样大。

```
/**
```

```
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode trimBST(TreeNode root, int L, int R) {
        if (root == null) return root;
        if (root.val > R) return trimBST(root.left, L, R);
        if (root.val < L) return trimBST(root.right, L, R);

        root.left = trimBST(root.left, L, R);
        root.right = trimBST(root.right, L, R);
        return root;
    }
}
```

26、二叉搜索树序列（面试题 04.09）

面试题 04.09. 二叉搜索树序列

难度 困难 36 收藏 分享 切换为英文 关注 反馈

从左向右遍历一个数组，通过不断将其中的元素插入树中可以逐步地生成一棵二叉搜索树。给定一个由不同节点组成的二叉搜索树，输出所有可能生成此树的数组。

示例:
给定如下二叉树

```
        2
       / \
      1   3
```

返回:

```
[
   [2,1,3],
   [2,3,1]
]
```

https://leetcode-cn.com/problems/bst-sequences-lcci/solution/5ms-hui-shuo-bian-li-si-lu-jian-dan-by-yu-da-xian-/

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
```

```
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
 //每一次递归循环当前所有可以的解，等 free 元素使用完，组成一组答案
class Solution {
    public static List<List<Integer>> BSTSequences(TreeNode root) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();
        if(root != null) {
            List<Integer> single = new ArrayList<Integer>();
            List<TreeNode> free = new LinkedList<TreeNode>();
            single.add(root.val);
            addList(root, single, free, res);
        }
        if(res.size() == 0) res.add(new ArrayList<Integer>());
        return res;
    }
    private static  void addList(TreeNode root, List<Integer> single, L
ist<TreeNode> free,  List<List<Integer>> res) {
        if(root == null) return;
        if(root.left != null) free.add(root.left);
        if(root.right != null) free.add(root.right);
        if(free.size() == 0) {
            List<Integer> one = new ArrayList<Integer>();
            for (int i = 0; i < single.size(); i ++) one.add(single.get
(i));
            res.add(one);
        }
        for (int i = 0; i < free.size(); i ++) {
            single.add(free.get(i).val);
            TreeNode remove = free.remove(i);
            addList(remove, single, free, res);
            free.add(i, remove);
            single.remove(single.size() - 1);
        }
        if(root.right != null) free.remove(root.right);
        if(root.left != null) free.remove(root.left);
    }
}
```

27、二叉搜索树迭代器（173）

更多硬核内容就在公众号「yes 的练级攻略」

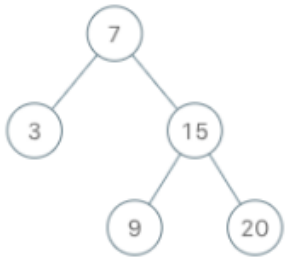
173. 二叉搜索树迭代器

难度 中等 236 收藏 分享 切换为英文 关注 反馈

实现一个二叉搜索树迭代器。你将使用二叉搜索树的根节点初始化迭代器。

调用 `next()` 将返回二叉搜索树中的下一个最小的数。

示例：




```
BSTIterator iterator = new BSTIterator(root);
iterator.next();    // 返回 3
iterator.next();    // 返回 7
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 9
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 15
iterator.hasNext(); // 返回 true
iterator.next();    // 返回 20
iterator.hasNext(); // 返回 false
```

提示：

- `next()` 和 `hasNext()` 操作的时间复杂度是 O(1)，并使用 O(*h*) 内存，其中 *h* 是树的高度。
- 你可以假设 `next()` 调用总是有效的，也就是说，当调用 `next()` 时，BST 中至少存在一个下一个最小的数。

https://leetcode-cn.com/problems/binary-search-tree-iterator/solution/er-cha-sou-suo-shu-die-dai-qi-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：受控递归

我们前面使用的方法在空间上与二叉搜索树中的节点数呈线性关系。然而，我们不得不使用这种方法的原因是我们可以在数组上迭代，且我们不能够暂停递归，然后在某个时候启动它。

但是，如果我们可以模拟中序遍历的受控递归，那么除了堆栈用于模拟递归的空间之外，实际上不需要使用任何其他空间。

因此，这种方法的本质上是使用自定义的栈来模拟中序遍历。也就是说，我们将采用迭代的方式来模拟中序遍历，而不是采用递归的方法；这样做的过程中，我们能够轻松的实现这两个函数的调用，而不是用其他额外的空间。

然而，就这两个函数的复杂性而言，会有点复杂，我们需要花一些时间来理解这种方法是否符合问题所说的渐近复杂性的要求。让我们更加具体的看一下这个算法。

算法：

1. 初始化一个空栈 `S`，用于模拟二叉搜索树的中序遍历。中序遍历我们采用与之前相同的方法，只是我们现在使用的是自己的栈而不是系统的堆栈。由于我们使用自定义的数据结构，因此可以随时暂停和恢复递归。
2. 我们还要实现一个帮助函数，在实现中将一次又一次的调用它。这个函数叫 `_inorder_left`，它将给定节点中的所有左子节点添加到栈中，直到节点没有左子节点为止。如下：

```
def inorder_left(root):
  while (root):
    S.append(root)
    root = root.left
```

3. 第一次调用 `next()` 函数时，必须返回二叉搜索树的最小元素，然后我们模拟递归必须向前移动一步，即移动到二叉搜索树的下一个最小元素上。栈的顶部始终包含 `next()` 函数返回的元素。`hasNext()` 很容易实现，因为我们只需要检查栈是否为空。
4. 首先，给定二叉搜索树的根节点，我们调用函数 `_inorder_left`，这确保了栈顶部始终包含了 `next()` 函数返回的元素。
5. 假设我们调用 `next()`，我们需要返回二叉搜索树中的下一个最小元素，即栈的顶部元素。有两种可能性：
   - 一个是栈顶部的节点是一个叶节点。这是最好的情况，因为我们什么都不用做，只需将节点从栈中弹出并返回其值。所以这是个常数时间的操作。
   - 另一个情况是栈顶部的节点拥有右节点。我们不需要检查左节点，因为左节点已经添加到栈中了。栈顶元素要么没有左节点，要么左节点已经被处理了。如果栈顶部拥有右节点，那么我们需要对右节点上调用帮助函数。该时间复杂度取决于树的结构。
6. `next()` 函数调用中，获取下一个最小的元素不需要花太多时间，但是要保证栈顶元素是 `next()` 函数返回的元素方面花费了一些时间。

复杂度分析

- 时间复杂度：
  - `hasNext()`：若栈中还有元素，则返回 `true`，反之返回 `false`。所以这是一个 $O(1)$ 的操作。
  - `next()`：包含了两个主要步骤。一个是从栈中弹出一个元素，它是下一个最小的元素。这是一个 $O(1)$ 的操作。然而，随后我们要调用帮助函数 `_inorder_left`，它需要递归的，将左节点添加到栈上，是线性时间的操作，最坏的情况下为 $O(N)$。但是我们只对含有右节点的节点进行调用，它也不会总是处理 N 个节点。只有当我们有一个倾斜的树，才会有 N 个节点。因此该操作的平均时间复杂度仍然是 $O(1)$，符合问题中所要求的。
- 空间复杂度：$O(h)$，使用了一个栈来模拟递归。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
```

```
 */
class BSTIterator {
    Stack<TreeNode> sta;
    public BSTIterator(TreeNode root) {
        this.sta = new Stack<TreeNode>();
        this.inorder(root);
    }
    public void inorder(TreeNode root){
        while (root != null) {
            this.sta.push(root);
            root = root.left;
        }
    }
    /** @return the next smallest number */
    public int next() {
        TreeNode top = this.sta.pop();
        if (top.right != null) {
            this.inorder(top.right);
        }
        return top.val;
    }
    /** @return whether we have a next smallest number */
    public boolean hasNext() {
        return this.sta.size() > 0;
    }
}
/**
 * Your BSTIterator object will be instantiated and called as such:
 * BSTIterator obj = new BSTIterator(root);
 * int param_1 = obj.next();
 * boolean param_2 = obj.hasNext();
 */
```

28、将二叉搜索树变平衡（1382）

1382. 将二叉搜索树变平衡

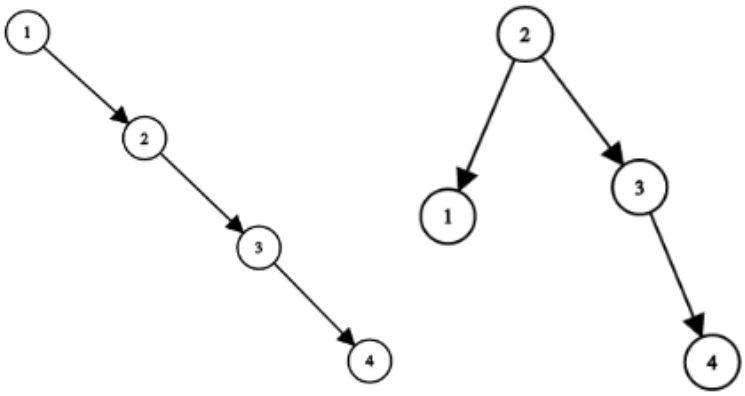
难度 中等 36 收藏 分享 切换为英文 关注 反馈

给你一棵二叉搜索树，请你返回一棵 **平衡后** 的二叉搜索树，新生成的树应该与原来的树有着相同的节点值。

如果一棵二叉搜索树中，每个节点的两棵子树高度差不超过 1 ，我们就称这棵二叉搜索树是 **平衡的** 。

如果有多种构造方法，请你返回任意一种。

示例：



输入： `root = [1,null,2,null,3,null,4,null,null]`
输出： `[2,1,3,null,null,null,4]`
解释： 这不是唯一的正确答案，`[3,1,4,null,2,null,null]` 也是一个可行的构造方案。

提示：

- 树节点的数目在 `1` 到 `10^4` 之间。
- 树节点的值互不相同，且在 `1` 到 `10^5` 之间。

https://leetcode-cn.com/problems/balance-a-binary-search-tree/solution/5179java-er-cha-sou-suo-shu-zhong-xu-bian-li-by-us/

思路分析：

- 本题的最终目标是构建一棵平衡二叉搜索树，原材料是一棵不平衡的BST。由一个有序数组来构建二叉搜索树比较简单，而二叉搜索树有一个重要性质，就是其中序遍历可以得到一个有序的数组。所以问题转化为两个子问题，见下面两个链接中的具体解答。
- 有序数组转化为平衡二叉树。 二叉树的中序遍历
- 由于不知道原BST中有几个结点，所以用 `List<Integer> vals = new ArrayList<>();` 来存放结点，作为中序遍历的递归函数的第二个实参。
- 时间复杂度，不论是中序遍历还是构建平衡BST，都是对结点遍历，所以时间复杂度为$O(n)$。空间复杂度，额外使用了一个List$O(n)$，以及构建平衡BST递归调用的函数栈$O(log(n))$，中序遍历的函数调用栈则正比于树高。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
```

```
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode balanceBST(TreeNode root) {
        if(root == null)
            return null;
        List<Integer> vals = new ArrayList<>();
        traversal(root, vals);
        return build(vals, 0, vals.size() - 1);
    }

    private void traversal(TreeNode x, List<Integer> vals){
        if(x == null) return;
        traversal(x.left, vals);
        vals.add(x.val);
        traversal(x.right, vals);
    }

    private TreeNode build(List<Integer> vals, int lo, int hi){
        if(lo > hi) return null;
        if(lo == hi) return new TreeNode(vals.get(hi));
        int mid = (hi - lo) / 2 + lo;
        TreeNode x = new TreeNode(vals.get(mid));
        x.left = build(vals, lo, mid - 1);
        x.right = build(vals, mid + 1, hi);
        return x;
    }
}
```

29、把二叉搜索树转换为累加树（538）【1038 从二叉搜索树到更大和树】

更多硬核内容就在公众号「yes 的练级攻略」

538. 把二叉搜索树转换为累加树

难度 简单 292 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树（Binary Search Tree），把它转换成为累加树（Greater Tree），使得每个节点的值是原来的节点值加上所有大于它的节点值之和。

例如：

输入：原始二叉搜索树：

```
              5
            /   \
           2     13
```

输出：转换为累加树：

```
             18
            /   \
          20     13
```

注意：本题和 1038: https://leetcode-cn.com/problems/binary-search-tree-to-greater-sum-tree/ 相同

https://leetcode-cn.com/problems/convert-bst-to-greater-tree/solution/ba-er-cha-sou-suo-shu-zhuan-huan-wei-lei-jia-shu-3/

方法 1：回溯 [Accepted]

想法

一个反序中序遍历的方法是通过递归实现。通过调用栈回到之前的节点，我们可以轻松地反序遍历所有节点。

算法

在递归方法中，我们维护一些递归调用过程中可以访问和修改的全局变量。首先我们判断当前访问的节点是否存在，如果存在就递归右子树，递归回来的时候更新总和和当前点的值，然后递归左子树。如果我们分别正确地递归 `root.right` 和 `root.left` ，那么我们就能正确地用大于某个节点的值去更新此节点，然后才遍历比它小的值。

复杂度分析

- 时间复杂度：$\mathcal{O}(n)$

  一个二叉树是没有环的，所以 `convertBST` 对于每个节点来说不会被调用超过 1 次。除去递归调用以外，`convertBST` 做的工作是常数时间的，所以线性次调用 `convertBST` 的运行时间是线性时间的。

- 空间复杂度：$\mathcal{O}(n)$

  使用之前的结论 `convertBST` 会被调用线性次，我们可以知道整个算法的空间复杂度也是线性的。考虑最坏情况，一棵树只有右子树（或者只有左子树），调用栈会一直增长直到到达叶子节点，也就是包含 n 个节点。

方法 1：回溯 双 o(n)

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int sum = 0;
    public TreeNode convertBST(TreeNode root) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (root != null) {
            convertBST(root.right);
            sum += root.val;
            root.val = sum;
            convertBST(root.left);
        }
        return root;
    }
}
```

方法 3：反序中序 Morris 遍历 [Accepted]

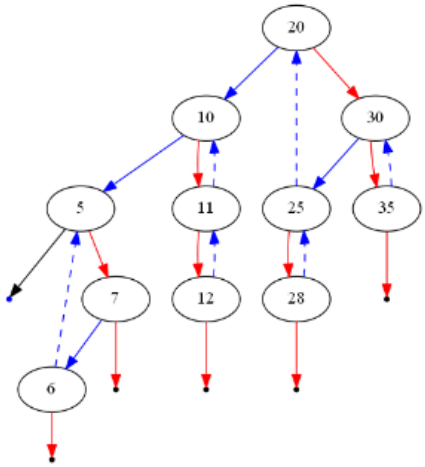
想法

有一种很机智的办法只用线性时间和常数空间来实现中序遍历，这种方法由 J. H. Morris 在他 1979 年的论文 "Traversing Binary Trees Simply and Cheaply" 中首次提出。总的来说，递归和迭代栈的方法在遍历左子树后返回当前节点，都牺牲了线性的空间。 Morris 遍历使用指针 `null` 指针在左子树以外创建了一个临时指针，使得遍历只需要使用常数额外空间。为了应用于此题，我们只需要交换所有的"左""右"引用，就可以反转整个遍历过程。

算法

首先，我们初始化 `node` ，它指向根。然后，在 `node` 指向 `null` 之前（具体指树中最小值节点的左孩子 `null` ），我们重复如下过程：先考虑当前节点是否有右子树，如果没有右子树那么所有大于当前节点的值都已遍历，我们就可以更新当前节点并进入左子树。如果还有右子树，那么至少有一个未访问过的更大值，我们必须先访问右子树。我们通过辅助函数 `getSuccessor` ，得到一个引用来记录当前节点中序遍历的下一个节点（最小大于当前值的节点），这个节点是恰好先于我们访问当前节点之前的节点，所以根据定义这一下一节点有一个为 `null` 的左孩子指针（否则它不是当前节点的下一个节点）。所以当我们首次找到一个节点的下一节点时，我们将该下一节点的左指针指向当前节点。这样，当我们遍历完当前节点的右子树时，右子树中最左边节点的左指针会是我们的临时指针，它指向当前节点，我们就可以快速回到当前节点中来。如果我们发现从下一节点的左指针回到了当前节点，说明右子树已经遍历完了，我们可以删除这个临时指针并继续递归左子树。

yes的练级攻略



上图说明了 Morris 遍历过程中一棵修改了的树。左指针被标为蓝色，右指针标为红色。虚线边表明遍历过程中的临时指针（在遍历结束之后会被删除）。注意到蓝色的边也可以标为虚线边，就像我们使用的"下一节点"的左空指针一样。除此以外，我们注意到每一个有右子树的节点都有一个"下一节点"的边指向自己。

复杂度分析

- 时间复杂度：$\mathcal{O}(n)$

  尽管 Morris遍历比起其他方法来说额外多做了一些工作，但只是一个常数级别的。具体来说，如果我们可以证明一条边被遍历了不超过常数 k 次，那么这个算法就是一个渐进线性的复杂度。首先注意到 `getSuccessor` 每个节点最多会调用 2 次，在第一次调用时，会有一个临时指针指向当前节点，第二次调用时，临时指针被删除。然后这个算法会进入左子树且不再会回到该节点。所以每条边最多被遍历 3 次：一次是移动 `node` 指针，另外两次遍历是调用 `getSuccessor` 时。

- 空间复杂度：$\mathcal{O}(1)$

  由于我们只操作已经存在的指针， Morris 只需要常数的额外空间。

方法 2：反序中序 `Morris` 遍历 空间复杂度降为 `o(1)`

```
class Solution {
    /* Get the node with the smallest value greater than this one. */
    private TreeNode getSuccessor(TreeNode node) {
        TreeNode succ = node.right;
        while (succ.left != null && succ.left != node) {
            succ = succ.left;
        }
        return succ;
    }

    public TreeNode convertBST(TreeNode root) {
        int sum = 0;
        TreeNode node = root;

        while (node != null) {
            /*
              * If there is no right subtree, then we can visit this node and
              * continue traversing left.
```

```
                */
                if (node.right == null) {
                    sum += node.val;
                    node.val = sum;
                    node = node.left;
                }
                /*
                 * If there is a right subtree, then there is at least one node that
                 * has a greater value than the current one. therefore, we must
                 * traverse that subtree first.
                 */
                else {
                    TreeNode succ = getSuccessor(node);
                    /*
                     * If the left subtree is null, then we have never been here before.
                     */
                    if (succ.left == null) {
                        succ.left = node;
                        node = node.right;
                    }
                    /*
                     * If there is a left subtree, it is a link that we created on a
                     * previous pass, so we should unlink it and visit this node.
                     */
                    else {
                        succ.left = null;
                        sum += node.val;
                        node.val = sum;
                        node = node.left;
                    }
                }
            }

            return root;
        }
    }
```

30、查找两棵二叉搜索树之和（1214）

更多硬核内容就在公众号「yes 的练级攻略」

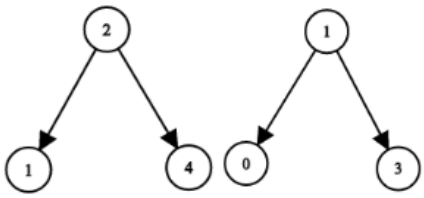
1214. 查找两棵二叉搜索树之和

难度 中等 19 收藏 分享 切换为英文 关注 反馈

给出两棵二叉搜索树，请你从两棵树中各找出一个节点，使得这两个节点的值之和等于目标值 `Target`。

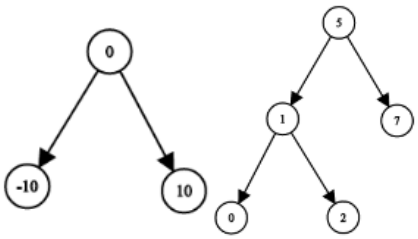
如果可以找到返回 `True`，否则返回 `False`。

示例 1：




```
输入：root1 = [2,1,4], root2 = [1,0,3], target = 5
输出：true
解释：2 加 3 和为 5 。
```

示例 2：




```
输入：root1 = [0,-10,10], root2 = [5,1,7,0,2], target = 18
输出：false
```

提示：

1. 每棵树上最多有 `5000` 个节点。
2. `-10^9 <= target, node.val <= 10^9`

https://leetcode-cn.com/problems/two-sum-bsts/solution/onlogn-suan-fa-ji-yu-er-cha-sou-suo-shu-cha-zhao-b/

思路：

1. 遍历其中一棵BST1,遍历到每个结点的时候，取出该结点值node.val，求得(target-node.val)的值拿到BST2去查找，找到就返回true。
2. 若同步步骤1没找到，则递归遍历BST1的左子树和BST1的右子树，不断执行步骤1，直到找到或者所有节点遍历完也未找到为止。

O(n*logn) 算法，基于二叉搜索树查找

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private boolean find(TreeNode root, int value) {
        if (root == null) return false;
        if (root.val == value) return true;
        else if (root.val < value) return find(root.right, value);
        else return find(root.left, value);
    }
    public boolean twoSumBSTs(TreeNode root1, TreeNode root2, int target) {
        if (root1 == null) return false;
        // 使用或运算进行短路操作，找到就终止
        return find(root2, target - root1.val) || twoSumBSTs(root1.left, root2,
 target) ||
                twoSumBSTs(root1.right, root2, target);
    }
}
```

31、二叉搜索子树的最大键值和（1373）

1373. 二叉搜索子树的最大键值和

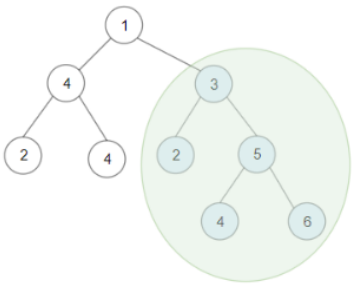
难度 困难　33　收藏　分享　切换为英文　关注　反馈

给你一棵以 `root` 为根的 **二叉树** ，请你返回 **任意** 二叉搜索子树的最大键值和。

二叉搜索树的定义如下：

- 任意节点的左子树中的键值都 **小于** 此节点的键值。
- 任意节点的右子树中的键值都 **大于** 此节点的键值。
- 任意节点的左子树和右子树都是二叉搜索树。

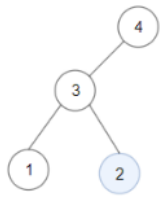
示例 1：



输入： root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]
输出： 20
解释： 键值为 3 的子树是和最大的二叉搜索树。

更多硬核内容就在公众号「yes 的练级攻略」

示例 2：




```
输入：root = [4,3,null,1,2]
输出：2
解释：键值为 2 的单节点子树是和最大的二叉搜索树。
```

示例 3：

```
输入：root = [-4,-2,-5]
输出：0
解释：所有节点键值都为负数，和最大的二叉搜索树为空。
```

示例 4：

```
输入：root = [2,1,3]
输出：6
```

示例 5：

```
输入：root = [5,4,8,3,null,6,3]
输出：7
```

提示：

- 每棵树最多有 `40000` 个节点。
- 每个节点的键值在 `[-4 * 10^4 , 4 * 10^4]` 之间。

https://leetcode-cn.com/problems/maximum-sum-bst-in-binary-tree/solution/shen-du-you-xian-sou-suo-zui-you-jie-yi-ding-zai-b/

解题思路

由root开始的树，是一颗**二叉树**，题目要求**二叉搜索树**的最大键值和。显然，我们要先找到**二叉搜索树节点**。

如何判断一棵树是否BST呢?

左右子树满足BST性质，同时$max(root.left) < root.val < min(root.right)$。只要满足这个条件的，就是BST。

这里为什么不需要再判断 `root.left` ， `root` ， `root.right` 的大小了呢?

因为左右子树已经是BST了，自然他们就满足BST的性质，满足$left.val < max(left)$，$right.val > min(right)$，所以，我们只要比较左子树最大，右子树最小和root节点的大小关系即可。

由于二叉搜索树的左右子树也是二叉搜索树。那么，只要找到**每条路径最上层的二叉树搜索树节点**，然后以这些BST节点为根，对其节点值求和，就能得到所有的BST节点和的结果集。

每个BST的节点和，都满足$sum(node) = node.val + sum(node.left) + sum(node.right)$，那么在获取当前节点和的时候，必然已经知道了子节点的节点之和。故，最大值必然在求解的过程中。

递归进入左右子树的值没有返回的必要，因为最终结果都存在res[]中，局部结果集，不需要递归给上层。

画外音：部分小伙伴可能会疑惑，为什么我没有用全局变量在存储值，反而用一个局部数组来呢？因为 `鲁迅先生说过：以局部变量为荣，以全局变量为耻。`

更多硬核内容就在公众号「yes 的练级攻略」

在工业开发过程中，全局变量能避免，最好避免。为什么呢？全局变量（或者说成员变量），在这个类里是共享的。如果这个类有两个指针指向它，就会出现成员变量被修改的安全问题。比较常见的就是单例模式。（当然了，如果这个成员变量，只读不写，那是不会有线程安全问题的）

比如说线程1的一个指针把成员变量p的值设置成了x，那么自然，我希望下一次取这个成员变量p的时候它的值依然是x。可是如果线程2指针也来操作这个成员变量p，把他的值设置成了y。那么在这时候线程1来获取p的值的时候，就会拿到y的值，这就和预期的x不一样了，这个就是安全问题。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int maxSumBST(TreeNode root) {
        int[] res = {0};
        maxSumBST(res, root);
        return res[0];
    }
    /**
     * 判断root为根节点的树是否是二叉搜索树
     *
     * @param root root 节点
     * @param min  最小值
     * @param max  最大值
     * @return 是否是二叉搜索树
     */
    private boolean isBST(TreeNode root, int min, int max) {
        if (root == null) return true;
        return min < root.val && root.val < max && isBST(root.left, min, root.v
al) && isBST(root.right, root.val, max);
    }

    /**
     * 求节点和的最大值
     *
     * @param res res
     * @param node node
     * @return 较大节点和
     */
    private void maxSumBST(int[] res, TreeNode node) {
        // 这个节点是BST，直接求和了
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            if (isBST(node, Integer.MIN_VALUE, Integer.MAX_VALUE)) {
                // 是二叉搜索树，求节点和，节点和的最优解，肯定在子节点求和的过程中
                sumNodeValue(res, node);
                return;
            }
            // 不是BST，递归进入左右子树
            maxSumBST(res, node.left);
            maxSumBST(res, node.right);
        }
        /**
         * 以node为root的节点求和
         *
         * @param res  结果集
         * @param node node
         * @return 节点之和
         */
        private int sumNodeValue(int[] res, TreeNode node) {
            if (node == null) return 0;
            // 二叉搜索树节点和的最大值，一定在子节点中会出现
            int sum = node.val + sumNodeValue(res, node.left) + sumNodeValue(res, n
ode.right);
            // 这里记录结果的最大值
            res[0] = Math.max(res[0], sum);
            return sum;
        }
}
```

32、二叉搜索树中的众数（501）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

501. 二叉搜索树中的众数

难度 简单 139 收藏 分享 切换为英文 关注 反馈

给定一个有相同值的二叉搜索树（BST），找出 BST 中的所有众数（出现频率最高的元素）。

假定 BST 有如下定义：

- 结点左子树中所含结点的值小于等于当前结点的值
- 结点右子树中所含结点的值大于等于当前结点的值
- 左子树和右子树都是二叉搜索树

例如：
给定 BST `[1, null, 2, 2]`,

```
   1
    \
     2
    /
   2
```

返回`[2]`.

提示：如果众数超过1个，不需考虑输出顺序

进阶：你可以不使用额外的空间吗？（假设由递归产生的隐式调用栈的开销不被计算在内）

https://leetcode-cn.com/problems/find-mode-in-binary-search-tree/solution/zhen-zheng-fu-he-ti-mu-yao-qiu-de-o1kong-jian-fu-z/

我看了题解区和评论区的很多生成 $O(1)$ 的高赞解法，但实际上这些解法都不是真正 $O(1)$ 的。考虑这样一种最坏情况：`[1, 2, 3, 4, ..., n-2, n-1, n-1]`，返回的答案应该只有一个，但中间开辟的额外空间会达到 $n-2$ 个，即在这种最坏情况下，空间复杂度是 $O(n)$，这显然是不符合题目的进阶要求的。

怎么才能做到真正 $O(1)$ 的空间复杂度呢？我认为需要两点：

1. 使用中序遍历。BST 的中序遍历结果是有序的。
2. 遍历两次。第一次确定结果集的大小，第二次得到结果集。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private TreeNode pre = null;
    private int[] ret;
    private int retCount = 0;
    private int maxCount = 0;
```

```
    private int currCount = 0;

    public int[] findMode(TreeNode root) {
        inOrder(root);
        pre = null;
        ret = new int[retCount];
        retCount = 0;
        currCount = 0;
        inOrder(root);
        return ret;
    }

    private void inOrder(TreeNode root) {
        if (root == null)
            return;
        inOrder(root.left);
        if (pre != null && pre.val == root.val)
            currCount++;
        else
            currCount = 1;
        if (currCount > maxCount) {
            maxCount = currCount;
            retCount = 1;
        }
        else if (currCount == maxCount) {
            if (ret != null)
                ret[retCount] = root.val;
            retCount++;
        }
        pre = root;
        inOrder(root.right);
    }
}
```

33、二叉搜索树的最小绝对差（530）【二叉搜索树节点最小距离（783）】

更多硬核内容就在公众号「yes 的练级攻略」

530. 二叉搜索树的最小绝对差

难度 简单 133 收藏 分享 切换为英文 关注 反馈

给你一棵所有节点为非负值的二叉搜索树，请你计算树中任意两节点的差的绝对值的最小值。

示例：

输入：

```
   1
    \
     3
    /
   2
```

输出：
1

解释：
最小绝对差为 1，其中 2 和 1 的差的绝对值为 1（或者 2 和 3）。

提示：

- 树中至少有 2 个节点。
- 本题与 783 https://leetcode-cn.com/problems/minimum-distance-between-bst-nodes/ 相同

783. 二叉搜索树节点最小距离

难度 简单 77 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树的根节点 `root`，返回树中任意两节点的差的最小值。

示例：

输入： root = [4,2,6,1,3,null,null]
输出： 1
解释：
注意，root是树节点对象(TreeNode object)，而不是数组。

给定的树 [4,2,6,1,3,null,null] 可表示为下图：

```
          4
        /   \
      2      6
     / \
    1   3
```

最小的差值是 1，它是节点1和节点2的差值，也是节点3和节点2的差值。

注意：

1. 二叉树的大小范围在 2 到 100。
2. 二叉树总是有效的，每个节点的值都是整数，且不重复。

https://leetcode-cn.com/problems/minimum-absolute-difference-in-bst/solution/marveljian-dan-de-xue-xi-bi-ji-530-by-tyanyonecanc/

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
```

```
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int res = Integer.MAX_VALUE;
    TreeNode pre;
    public int getMinimumDifference(TreeNode root) {
        dfs(root);
        return res;
    }
    public void dfs(TreeNode root){
        if (root == null) return;
        dfs(root.left);
        if (pre != null) res = Math.min(res, root.val - pre.val);
        pre = root;
        dfs(root.right);
    }
}
```

34、二叉搜索树中第 K 小的元素（230）

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

230. 二叉搜索树中第K小的元素

难度 中等 250 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树，编写一个函数 `kthSmallest` 来查找其中第 **k** 个最小的元素。

说明：
你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。

示例 1:

```
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 1
```

示例 2:

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 3
```

进阶：
如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化 `kthSmallest` 函数？

https://leetcode-cn.com/problems/kth-smallest-element-in-a-bst/solution/er-cha-sou-suo-shu-zhong-di-kxiao-de-yuan-su-by-le/

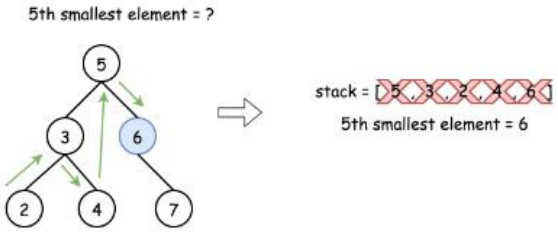
方法 1：递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
```

```
    int num = 0;
    int res;

    public int kthSmallest(TreeNode root, int k) {
        inorderTraversal(root, k);
        return res;
    }

    private void inorderTraversal(TreeNode node, int k) {
        if (node == null) {
            return;
        }
        inorderTraversal(node.left, k);
        num++;
        if (num == k) {
            res = node.val;
            return;
        }
        inorderTraversal(node.right, k);
    }
}
```

算法：

在栈的帮助下，可以将方法一的递归转换为迭代，这样可以加快速度，因为这样可以不用遍历整个树，可以在找到答案后停止。



复杂度分析

- 时间复杂度：$\mathcal{O}(H+k)$，其中 H 指的是树的高度，由于我们开始遍历之前，要先向下达到叶，当树是一个平衡树时：复杂度为 $\mathcal{O}(\log N+k)$。当树是一个不平衡树时：复杂度为 $\mathcal{O}(N+k)$，此时所有的节点都在左子树。
- 空间复杂度：$\mathcal{O}(H+k)$。当树是一个平衡树时：$\mathcal{O}(\log N+k)$。当树是一个非平衡树时：$\mathcal{O}(N+k)$。

方法 2：迭代（更短）

```
class Solution {
    public int kthSmallest(TreeNode root, int k) {
        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();

        while (true) {
        while (root != null) {
```

```
                stack.add(root);
                root = root.left;
            }
            root = stack.removeLast();
            if (--k == 0) return root.val;
            root = root.right;
            }
        }
}
```

35、二叉搜索树的第 k 大节点（剑指 54）【跟上一题的区别就是遍历时候右节点先遍历】

剑指 Offer 54. 二叉搜索树的第k大节点

难度 简单 50 收藏 分享 切换为英文 关注 反馈

给定一棵二叉搜索树，请找出其中第k大的节点。

示例 1:

```
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 4
```

示例 2:

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 4
```

限制:

1 ≤ k ≤ 二叉搜索树元素个数

https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-di-kda-jie-dian-lcof/solution/mian-shi-ti-54-er-cha-sou-suo-shu-de-di-k-da-jie-d/

方法 1：先反转链表，再找第 k 小的元素

```
class Solution {
    public int kthLargest(TreeNode root, int k) {
```

```
        TreeNode rot = mirrorTree(root);
        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
        while (true) {
        while (rot != null) {
            stack.add(rot);
            rot = rot.left;
        }
        rot = stack.removeLast();
        if (--k == 0) return rot.val;
        rot = rot.right;
        }
    }
    public TreeNode mirrorTree(TreeNode root) {
        if (root == null) return root;
        TreeNode left = mirrorTree(root.left);
        TreeNode right = mirrorTree(root.right);
        root.left = right;
        root.right = left;
        return root;
    }
}
```

方法 2：递归

```
class Solution {
    int res, k;
    public int kthLargest(TreeNode root, int k) {
        this.k = k;
        dfs(root);
        return res;
    }
    void dfs(TreeNode root) {
        if(root == null) return;
        dfs(root.right);
        if(k == 0) return;
        if(--k == 0) res = root.val;
        dfs(root.left);
    }
}
```

方法 3：迭代

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
```

```
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int kthLargest(TreeNode root, int k) {
        LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
        while (true) {
            while (root != null) {
                stack.add(root);
                root = root.right;
            }
            root = stack.removeLast();
            if (--k == 0) return root.val;
            root = root.left;
        }
    }
}
```

36、删除二叉搜索树中的节点（450）

450. 删除二叉搜索树中的节点

难度 中等 274 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树的根节点 **root** 和一个值 **key**，删除二叉搜索树中的 **key** 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

1. 首先找到需要删除的节点；
2. 如果找到了，删除它。

说明： 要求算法时间复杂度为 O(h)，h 为树的高度。

示例:

```
root = [5,3,6,2,4,null,7]
key = 3

    5
   / \
  3   6
 / \   \
2   4   7
```

给定需要删除的节点值是 `3`，所以我们首先找到 `3` 这个节点，然后删除它。

一个正确的答案是 `[5,4,6,2,null,null,7]`，如下图所示。

```
    5
   / \
  4   6
 /     \
2       7
```

另一个正确答案是 `[5,2,6,null,4,null,7]`。

```
    5
   / \
  2   6
   \   \
    4   7
```

https://leetcode-cn.com/problems/delete-node-in-a-bst/solution/shan-chu-er-cha-sou-suo-shu-zhong-de-jie-dian-by-l/

二叉搜索树的三个特性:

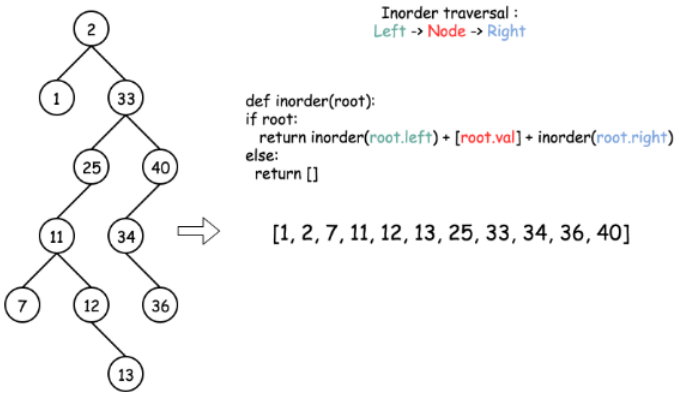
这些性质最好在面试之前了解清楚:

- 二叉搜索树的中序遍历的序列是递增排序的序列。中序遍历的遍历次序: `Left -> Node -> Right`。

Java | Python

```java
public LinkedList<Integer> inorder(TreeNode root, LinkedList<Integer> arr) {
  if (root == null) return arr;
  inorder(root.left, arr);
  arr.add(root.val);
  inorder(root.right, arr);
  return arr;
}
```

更多硬核内容就在公众号「yes 的练级攻略」



- `Successor` 代表的是中序遍历序列的下一个节点。即比当前节点大的最小节点，简称后继节点。 先取当前节点的右节点，然后一直取该节点的左节点，直到左节点为空，则最后指向的节点为后继节点。

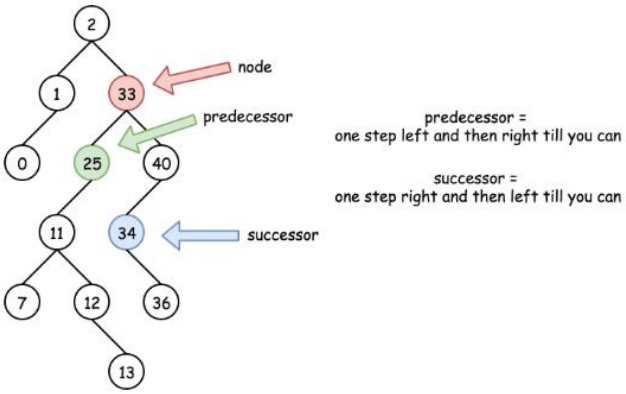
Java | Python

```java
public int successor(TreeNode root) {
  root = root.right;
  while (root.left != null) root = root.left;
  return root;
}
```

- `Predecessor` 代表的是中序遍历序列的前一个节点。即比当前节点小的最大节点，简称前驱节点。先取当前节点的左节点，然后取该节点的右节点，直到右节点为空，则最后指向的节点为前驱节点。

Java | Python

```java
public int predecessor(TreeNode root) {
  root = root.left;
  while (root.right != null) root = root.right;
  return root;
}
```

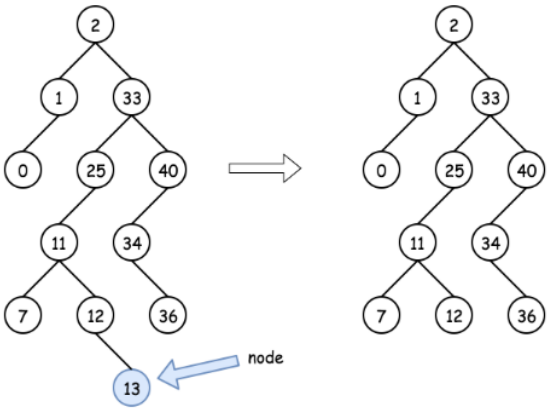


更多硬核内容就在公众号「yes 的练级攻略」

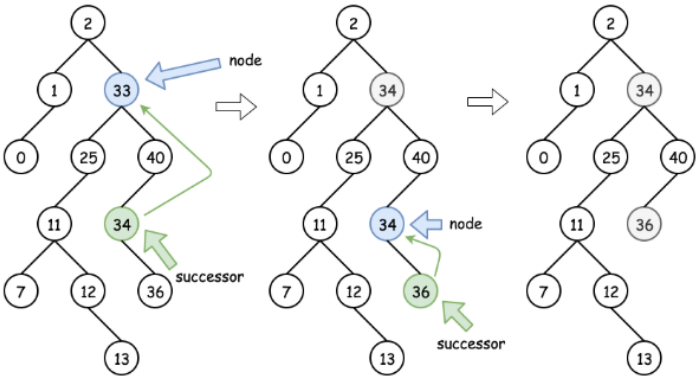
方法：递归

这里有三种可能的情况：

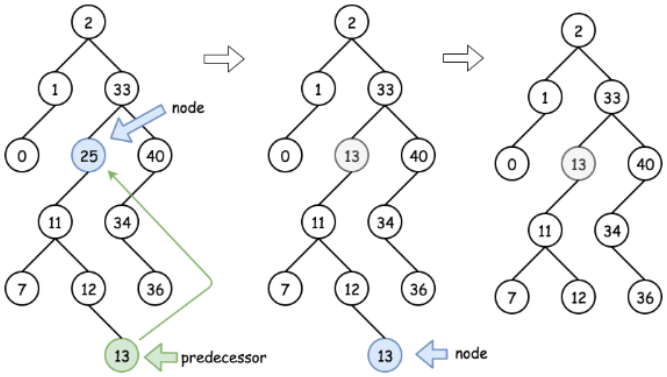
- 要删除的节点为叶子节点，可以直接删除。



- 要删除的几点不是叶子节点且拥有右节点，则该节点可以由该节点的后继节点进行替代，该后继节点位于右子树中较低的位置。然后可以从后继节点的位置递归向下操作以删除后继节点。



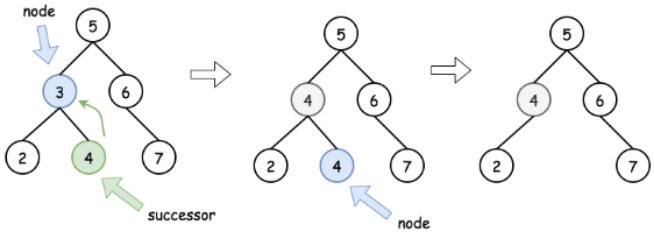
- 要删除的节点不是叶子节点，且没有右节点但是有左节点。这意味着它的后继节点在它的上面，但是我们并不想返回。我们可以使用它的前驱节点进行替代，然后再递归的向下删除前驱节点。



更多硬核内容就在公众号「yes 的练级攻略」

算法：

- 如果 `key > root.val`，说明要删除的节点在右子树，`root.right = deleteNode(root.right, key)`。
- 如果 `key < root.val`，说明要删除的节点在左子树，`root.left = deleteNode(root.left, key)`。
- 如果 `key == root.val`，则该节点就是我们要删除的节点，则：
  - 如果该节点是叶子节点，则直接删除它：`root = null`。
  - 如果该节点不是叶子节点且有右节点，则用它的后继节点的值替代 `root.val = successor.val`，然后删除后继节点。
  - 如果该节点不是叶子节点且只有左节点，则用它的前驱节点的值替代 `root.val = predecessor.val`，然后删除前驱节点。
- 返回 `root`。



复杂度分析

- 时间复杂度：$\mathcal{O}(\log N)$。在算法的执行过程中，我们一直在树上向左或向右移动。首先先用 $\mathcal{O}(H_1)$ 的时间找到要删除的节点，H_1 值得是从根节点到要删除节点的高度。然后删除节点需要 $\mathcal{O}(H_2)$ 的时间，H_2 指的是从要删除节点到替换节点的高度。由于 $\mathcal{O}(H_1 + H_2) = \mathcal{O}(H)$，$H$ 值得是树的高度，若树是一个平衡树则 $H = \log N$。
- 空间复杂度：$\mathcal{O}(H)$，递归时堆栈使用的空间，H 是树的高度。

方法 1：递归

```
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) return root;  // 递归到 root 为 null ，说明找不到要删的节点
        if (key < root.val) {           // 要删的节点在左子树，删完保持 BST 特性
            root.left = deleteNode(root.left, key);
        } else if (key > root.val) {    // 要删的节点在右子树，删完保持 BST 特性
            root.right = deleteNode(root.right, key);
        } else {                        // 找到了要删除的节点
            if (root.left == null && root.right == null) {  // 如果是叶子节点，删了返回 null 给其父节点即可
                return null;
            } else if (root.left != null && root.right == null) {   // 如果左不空而右空，父节点继承它的左子树即可
                return root.left;
            } else if (root.left == null && root.right != null) {   // 如果右不空而左空，父节点继承它的右子树即可
                return root.right;
```

```
            } else {                                    // 左右都不空，在右子树寻找一个节点继承左子树
                TreeNode last = null;                   // 在右子树继承 root.left 子树的节点
                TreeNode currentRoot = root.right;      // 进入右子树
                while (currentRoot != null) {
                    last = currentRoot;
                    currentRoot = currentRoot.left;     // 一直往左走，走到空
                }
                last.left = root.left;      // 继承
                return root.right;          // 父节点继承修改后得到新子树
            }
        }
        return root;
    }
}
```

https://leetcode-cn.com/problems/delete-node-in-a-bst/solution/shan-chu-er-cha-sou-suo-shu-zhong-de-jie-dian-di-4/

迭代思路

1. 先寻找要删除的节点，并记录它的父节点，因为父节点要继承它的子树。
2. 继承关系与递归思路中的一致。
3. 比较容易理解，但由于个人水平有限，写的代码臭又长。

算法分析

设 `n` 为 `BST` 中的节点数

1. 由于 `BST特性` ，若在 `BST` 中能找到要删除的节点，则需要$O(log(n))$的时间；找到后删除需要$O(log(n))$的时间，因此时间复杂度为$O(log(n))$
2. 只使用了几个变量，空间复杂度为$O(1)$

方法 2：迭代

```
class Solution {
    TreeNode keyFather = null;  // 要删除节点的父节点
    TreeNode keyNode = null;    // 要删除的节点
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) {     // 空的 BST
            return null;
        }
        findKey(root, key);         // 查找
        if (keyNode == null) {      // 找不到要删的节点
            return root;
```

```
        }
        if (keyFather == null) {    // 删的是根节点
            if (keyNode.left == null && keyNode.right == null) {
    // 没有可替代的选择
                return null;
            } else if (keyNode.left != null && keyNode.right == null) {
    // 左子树不空，右子树空
                return keyNode.left;
            } else if (keyNode.left == null && keyNode.right != null){
    // 右子树不空，左子树空
                return keyNode.right;
            } else {        // 左右子树都不为空
                TreeNode last = null;
                keyNode = keyNode.right;
                while (keyNode != null) {
                    last = keyNode;
                    keyNode = keyNode.left;
                }
                last.left = root.left;
                return root.right;
            }
        } else {    // 删的不是根节点
            if (keyNode.left == null && keyNode.right == null) {    //
删的是叶子节点
                if (keyFather.left == keyNode) {
                    keyFather.left = null;
                } else {
                    keyFather.right = null;
                }
            } else if (keyNode.left != null && keyNode.right == null) {
    // 删除节点的左子树不空，右子树为空
                if (keyNode == keyFather.left) {    // 删除节点是父节点的
左子节点
                    keyFather.left = keyNode.left;
                } else {                            // 删除节点是父节点的
右子节点
                    keyFather.right = keyNode.left;
                }
            } else if (keyNode.left == null && keyNode.right != null) {
    // 删除节点的右子树不空，左子树为空
                if (keyNode == keyFather.left) {    // 删除节点是父节点的
左子节点
                    keyFather.left = keyNode.right;
```

```
                } else {                                // 删除节点是父节点的
右子节点
                    keyFather.right = keyNode.right;
                }
            } else {    // 删除节点的左右子树都不为空

                TreeNode last = null;
                TreeNode currentRoot = keyNode.right;
                while (currentRoot != null) {
                    last = currentRoot;
                    currentRoot = currentRoot.left;
                }
                last.left = keyNode.left;
                // 继承
                if (keyNode == keyFather.left) {
                    keyFather.left = keyNode.right;
                } else {
                    keyFather.right = keyNode.right;
                }
            }
        }
        return root;
    }

    // 利用 BST 特性查找要删除的节点
    public void findKey(TreeNode root, int key) {
        TreeNode father = null;
        TreeNode currentRoot = root;
        while (currentRoot != null) {
            if (key == currentRoot.val) {
                break;
            } else {
                father = currentRoot;
                if (key < currentRoot.val) {
                    currentRoot = currentRoot.left;
                } else {
                    currentRoot = currentRoot.right;
                }
            }
        }
        // 记录
        keyFather = father;
        keyNode = currentRoot;
    }
```

}

37、二叉搜索树的后续遍历序列（剑指 33）

剑指 Offer 33. 二叉搜索树的后序遍历序列

难度 中等 93 收藏 分享 切换为英文 关注 反馈

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 `true`，否则返回 `false`。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```
     5
    / \
   2   6
  / \
 1   3
```

示例 1：

```
输入：[1,6,3,2,5]
输出：false
```

示例 2：

```
输入：[1,3,2,6,5]
输出：true
```

提示：

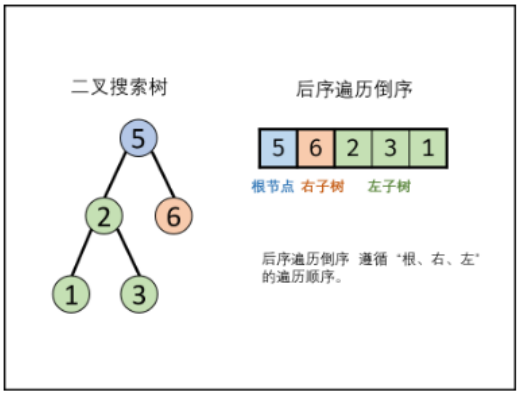
1. 数组长度 <= 1000

https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-hou-xu-bian-li-xu-lie-lcof/solution/mian-shi-ti-33-er-cha-sou-suo-shu-de-hou-xu-bian-6/

方法二：辅助单调栈

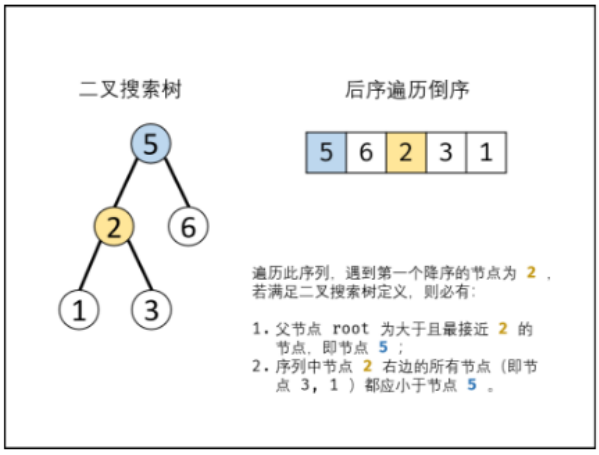
> 此方法参考于大佬 @失火的夏天 的解析。

- **后序遍历倒序：** `[ 根节点 | 右子树 | 左子树 ]` 。类似 **先序遍历的镜像** ，即先序遍历为 “根、左、右” 的顺序，而后序遍历的倒序为 “根、右、左” 顺序。



- 设后序遍历倒序列表为 $[r_n, r_{n-1}, ..., r_1]$，遍历此列表，设索引为 i，若为 **二叉搜索树** ，则有：
  - **当节点值** $r_i > r_{i+1}$ **时：** 节点 r_i 一定是节点 r_{i+1} 的右子节点。
  - **当节点值** $r_i < r_{i+1}$ **时：** 节点 r_i 一定是某节点 $root$ 的左子节点，且 $root$ 为节点 $r_{i+1}, r_{i+2}, ..., r_n$ 中值大于且最接近 r_i 的节点（$\because root$ **直接连接** 左子节点 r_i ）。
- 当遍历时遇到递减节点 $r_i < r_{i+1}$ ，若为二叉搜索树，则对于后序遍历中节点 r_i 右边的任意节点 $r_x \in [r_{i-1}, r_{i-2}, ..., r_1]$ ，必有节点值 $r_x < root$ 。

> 节点 r_x 只可能为以下两种情况：① r_x 为 r_i 的左、右子树的各节点；② r_x 为 $root$ 的父节点或更高层父节点的左子树的各节点。在二叉搜索树中，以上节点都应小于 $root$ 。



- 遍历 “后序遍历的倒序” 会多次遇到递减节点 r_i ，若所有的递减节点 r_i 对应的父节点 $root$ 都满足以上条件，则可判定为二叉搜索树。
- 根据以上特点，考虑借助 **单调栈** 实现：
  1. 借助一个单调栈 $stack$ 存储值递增的节点；
  2. 每当遇到值递减的节点 r_i ，则通过出栈来更新节点 r_i 的父节点 $root$ ；
  3. 每轮判断 r_i 和 $root$ 的值关系：
     1. 若 $r_i > root$ 则说明不满足二叉搜索树定义，直接返回 $false$ 。
     2. 若 $r_i < root$ 则说明满足二叉搜索树定义，则继续遍历。

算法流程：

1. **初始化：** 单调栈 $stack$ ，父节点值 $root = +\infty$ （初始值为正无穷大，可把树的根节点看为此无穷大节点的左孩子）；
2. **倒序遍历** $postorder$ ：记每个节点为 r_i ；
   1. **判断：** 若 $r_i > root$ ，说明此后序遍历序列不满足二叉搜索树定义，直接返回 $false$ ；
   2. **更新父节点** $root$ ： 当栈不为空 **且** $r_i < stack.peek()$ 时，循环执行出栈，并将出栈节点赋给 $root$ 。
   3. **入栈：** 将当前节点 r_i 入栈；
3. 若遍历完成，则说明后序遍历满足二叉搜索树定义，返回 $true$ 。

复杂度分析：

- **时间复杂度** $O(N)$ ： 遍历 $postorder$ 所有节点，各节点均入栈 / 出栈一次，使用 $O(N)$ 时间。
- **空间复杂度** $O(N)$ ： 最差情况下，单调栈 $stack$ 存储所有节点，使用 $O(N)$ 额外空间。

```
class Solution {
    public boolean verifyPostorder(int[] postorder) {
        Stack<Integer> sta = new Stack<>();
        int root = Integer.MAX_VALUE;
        for (int i = postorder.length - 1; i >= 0; i--){
            if (postorder[i] > root) return false;
            while (!sta.isEmpty() && sta.peek() > postorder[i]) {
                root = sta.pop();
            }
            sta.add(postorder[i]);
        }
        return true;
    }
}
```

38、将二叉搜索树转化为排序的双向链表（426）【二叉搜索树和双向链表（剑指 36）】

更多硬核内容就在公众号「yes 的练级攻略」

426. 将二叉搜索树转化为排序的双向链表

难度 中等 51 收藏 分享 切换为英文 关注 反馈

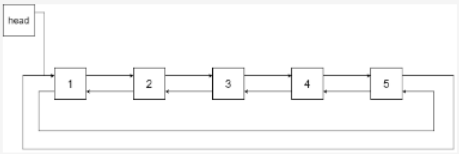
将一个 **二叉搜索树** 就地转化为一个 **已排序的双向循环链表** 。

对于双向循环列表，你可以将左右孩子指针作为双向循环链表的前驱和后继指针，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

特别地，我们希望可以 **就地** 完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中最小元素的指针。

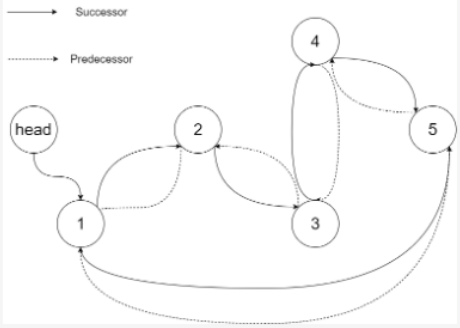
示例 1：

输入： root = [4,2,5,1,3]



输出： [1,2,3,4,5]

解释： 下图显示了转化后的二叉搜索树，实线表示后继关系，虚线表示前驱关系。



示例 2：

输入： root = [2,1,3]
输出： [1,2,3]

示例 3：

输入： root = []
输出： []
解释： 输入是空树，所以输出也是空链表。

示例 4：

输入： root = [1]
输出： [1]

提示：

- -1000 <= Node.val <= 1000
- Node.left.val < Node.val < Node.right.val
- Node.val 的所有值都是独一无二的
- 0 <= Number of Nodes <= 2000

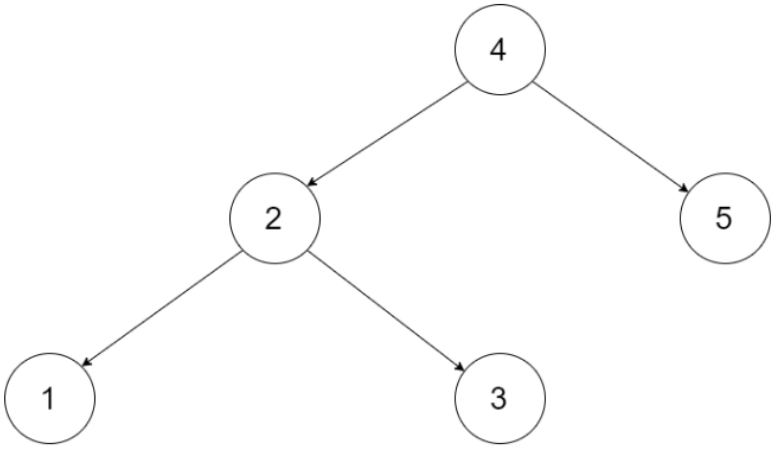
更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 36. 二叉搜索树与双向链表

难度 中等 86 收藏 分享 切换为英文 关注 反馈

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

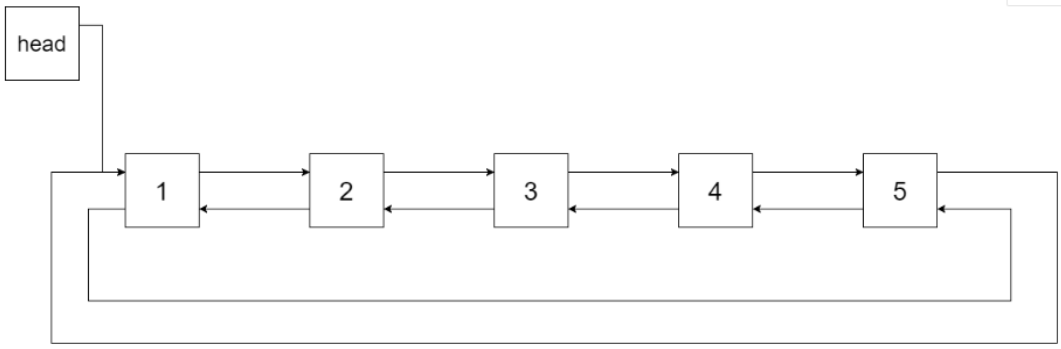
为了让您更好地理解问题，以下面的二叉搜索树为例：



我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。

☆ 收藏



特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

注意：本题与主站 426 题相同：https://leetcode-cn.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/

注意：此题对比原题有改动。

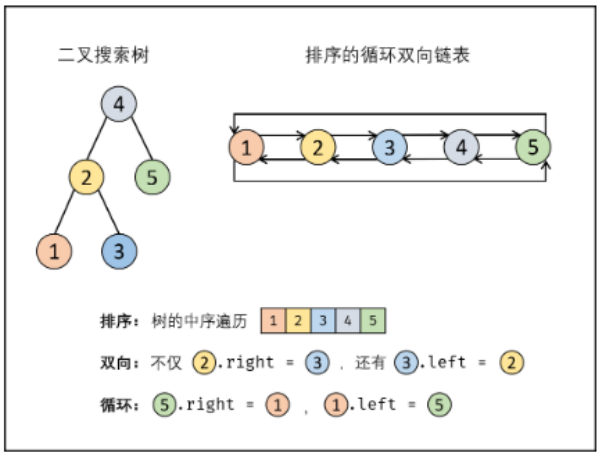
https://leetcode-cn.com/problems/er-cha-sou-suo-shu-yu-shuang-xiang-lian-biao-lcof/solution/mian-shi-ti-36-er-cha-sou-suo-shu-yu-shuang-xian-5/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

本文解法基于性质：二叉搜索树的中序遍历为 **递增序列** 。
将 二叉搜索树 转换成一个 “排序的循环双向链表” ，其中包含三个要素：

1. **排序链表：** 节点应从小到大排序，因此应使用 **中序遍历** “从小到大”访问树的节点；
2. **双向链表：** 在构建相邻节点（设前驱节点 pre ，当前节点 cur ）关系时，不仅应 $pre.right = cur$ ，也应 $cur.left = pre$ 。
3. **循环链表：** 设链表头节点 $head$ 和尾节点 $tail$ ，则应构建 $head.left = tail$ 和 $tail.right = head$ 。



中序遍历 为 “左、根、右” 顺序，递归实现代码如下：

python java

```
// 打印中序遍历
void dfs(TreeNode root) {
    if(root == null) return;
    dfs(root.left); // 左
    System.out.println(root.val); // 根
    dfs(root.right); // 右
}
```

根据以上分析，考虑使用中序遍历访问树的各节点 cur ；并在访问每个节点时构建 cur 和前驱节点 pre 的引用指向；中序遍历完成后，最后构建头节点和尾节点的引用指向即可。

算法流程：

`dfs(cur)`：递归法中序遍历；

1. **终止条件：** 当节点 cur 为空，代表越过叶节点，直接返回；
2. 递归左子树，即 `dfs(cur.left)` ；
3. **构建链表：**
    1. **当 pre 为空时：** 代表正在访问链表头节点，记为 $head$ 。
    2. **当 pre 不为空时：** 修改双向节点引用，即 $pre.right = cur$ ， $cur.left = pre$ ；
    3. **保存 cur ：** 更新 $pre = cur$ ，即节点 cur 是后继节点的 pre ；
4. 递归右子树，即 `dfs(cur.left)` ；

`treeToDoublyList(root)`：

1. **特例处理：** 若节点 $root$ 为空，则直接返回；
2. **初始化：** 空节点 pre ；
3. **转化为双向链表：** 调用 `dfs(root)` ；
4. **构建循环链表：** 中序遍历完成后，$head$ 指向头节点， pre 指向尾节点，因此修改 $head$ 和 pre 的双向节点引用即可。
5. **返回值：** 返回链表的头节点 $head$ 即可。

复杂度分析：

- **时间复杂度 $O(N)$：** N 为二叉树的节点数，中序遍历需要访问所有节点。
- **空间复杂度 $O(N)$：** 最差情况下，即树退化为链表时，递归深度达到 N，系统使用 $O(N)$ 栈空间。

426 具体代码

```
/*
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val,Node _left,Node _right) {
        val = _val;
        left = _left;
        right = _right;
    }
};
*/
class Solution {
    Node pre, head;
    public Node treeToDoublyList(Node root) {
        if (root == null) return root;
        dfs(root);
        head.left = pre;
        pre.right = head;
        return head;
    }
    public void dfs(Node cur){
        if (cur == null) return;
        dfs(cur.left);
        if (pre != null) pre.right = cur;
        else head = cur;
        cur.left = pre;
        pre = cur;
        dfs(cur.right);
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

39、对称二叉树（101）（剑指 28）

101. 对称二叉树

难度 简单 863 收藏 分享 切换

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

进阶:

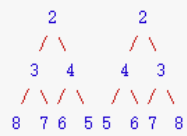
你可以运用递归和迭代两种方法解决这个问题吗?

用
:
攻略

更多硬核内容就在公众号「yes 的练级攻略」

递归实现

乍一看无从下手，但用递归其实很好解决。
根据题目的描述，镜像对称，就是左右两边相等，也就是左子树和右子树是相当的。
注意这句话，左子树和右子树相等，也就是说要递归的比较左子树和右子树。
我们将根节点的左子树记做 `left`，右子树记做 `right`。比较 `left` 是否等于 `right`，不等的话直接返回就可以了。
如果相当，比较 `left` 的左节点和 `right` 的右节点，再比较 `left` 的右节点和 `right` 的左节点
比如看下面这两个子树(他们分别是根节点的左子树和右子树)，能观察到这么一个规律：
左子树 2 的左孩子 == 右子树 2 的右孩子
左子树 2 的右孩子 == 右子树 2 的左孩子



根据上面信息可以总结出递归函数的两个条件：
终止条件：

1. `left` 和 `right` 不等，或者 `left` 和 `right` 都为空
2. 递归的比较 `left`，`left` 和 `right.right`，递归比较 `left`，`right` 和 `right.left`

算法的时间复杂度是 $O(n)$，因为要遍历 n 个节点
空间复杂度是 $O(n)$，空间复杂度是递归的深度，也就是跟树高度有关，最坏情况下树变成一个链表结构，高度是n。

参考链接：
https://leetcode-cn.com/problems/symmetric-tree/solution/dong-hua-yan-shi-101-dui-cheng-er-cha-shu-by-user7/

方法 1：递归算法（1ms）

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) {
            return true;
        }
        return dfs(root.left,root.right);
    }
    private boolean dfs(TreeNode left, TreeNode right) {
        //递归的终止条件是两个节点都为空
        //或者两个节点中有一个为空
        //或者两个节点的值不相等
        if(left==null && right==null) {
            return true;
        }
        if(left==null || right==null) {
            return false;
        }
        if(left.val!=right.val) {
            return false;
        }
        //再递归的比较 左节点的左孩子 和 右节点的右孩子
```

```
            //以及比较  左节点的右孩子 和 右节点的左孩子
            return dfs(left.left,right.right) && dfs(left.right,right.left)
;
        }
    }
```

方法 2：递归算法（0ms）简写版本

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null ) {
            return true;
        }
        return isMirror(root.left,root.right);
    }
    public boolean isMirror(TreeNode l, TreeNode r) {
        if (l == null && r == null) return true;
        if (l == null || r == null) return false;
        return (l.val == r.val) && isMirror(l.left,r.right) && isMirror
(l.right, r.left);
    }
}
```

队列实现

回想下递归的实现：
当两个子树的根节点相等时，就比较：
左子树的 `left` 和 右子树的 `right` ，这个比较是用递归实现的。
现在我们改用队列来实现，思路如下：
首先从队列中拿出两个节点(`left` 和 `right`)比较
将 `left` 的 `left` 节点和 `right` 的 `right` 节点放入队列
将 `left` 的 `right` 节点和 `right` 的 `left` 节点放入队列
时间复杂度是 $O(n)$，空间复杂度是 $O(n)$

方法 3：迭代算法（1ms）

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null || (root.left == null && root.right == null))
{
            return true;
        }
        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(root.left);
        queue.add(root.right);
        while (queue.size() > 0) {
            //从队列中取出两个节点，再比较这两个节点
```

```
            TreeNode left = queue.removeFirst();
            TreeNode right = queue.removeFirst();
            //如果两个节点都为空就继续循环，两者有一个为空就返回false
            if (left == null && right == null) {
                continue;
            }
            if (left == null || right == null) {
                return false;
            }
            if (left.val != right.val) {
                return false;
            }
            //将左节点的左孩子， 右节点的右孩子放入队列
            queue.add(left.left);
            queue.add(right.right);
            //将左节点的右孩子， 右节点的左孩子放入队列
            queue.add(left.right);
            queue.add(right.left);
        }
        return true;
    }
}
```

40、二叉树的镜像（剑指 27）【与下一题翻转二叉树一样】

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 27. 二叉树的镜像

难度 简单 45 收藏 分享 切换为英文 关注 反馈

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

例如输入：

镜像输出：

示例 1：

输入： root = [4,2,7,1,3,6,9]
输出： [4,7,2,9,6,3,1]

限制：

0 <= 节点个数 <= 1000

注意：本题与主站 226 题相同：https://leetcode-cn.com/problems/invert-binary-tree/

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if (root == null) return root;
        TreeNode left = mirrorTree(root.left);
        TreeNode right = mirrorTree(root.right);
        root.left = right;
```

```
            root.right = left;
            return root;
        }
    }
```

41、翻转二叉树（226）【与牛客网 18 题一致】

226. 翻转二叉树

难度 简单 474 收藏 分享 切换为英文 关注 反馈

翻转一棵二叉树。

示例:

输入:

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

输出:

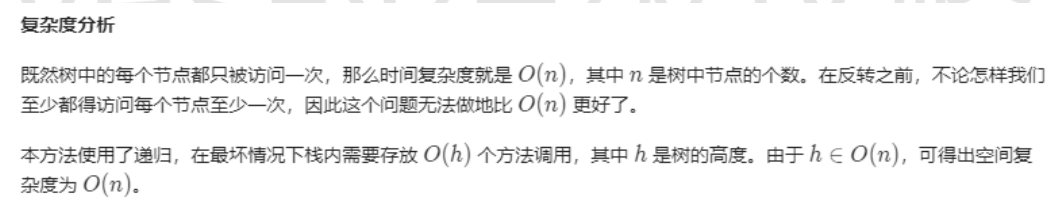
```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

备注:
这个问题是受到 Max Howell 的 原问题 启发的：

谷歌：我们90%的工程师使用您编写的软件(Homebrew)，但是您却无法在面试时在白板上写出翻转二叉树这道题，这太糟糕了。

https://leetcode-cn.com/problems/invert-binary-tree/solution/fan-zhuan-er-cha-shu-by-leetcode/

一定要注意树的结构和输入输出



复杂度分析

既然树中的每个节点都只被访问一次，那么时间复杂度就是 $O(n)$，其中 n 是树中节点的个数。在反转之前，不论怎样我们至少都得访问每个节点至少一次，因此这个问题无法做地比 $O(n)$ 更好了。

本方法使用了递归，在最坏情况下栈内需要存放 $O(h)$ 个方法调用，其中 h 是树的高度。由于 $h \in O(n)$，可得出空间复杂度为 $O(n)$。

方法 1：递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    public TreeNode invertTree(TreeNode root) {
        if (root == null) return null;
        TreeNode left = invertTree(root.left);
        TreeNode right = invertTree(root.right);
        root.left = right;
        root.right = left;
        return root;
    }
}
```

我们也可以用迭代方法来解决这个问题，这种做法和深度优先搜索（Breadth-fist Search, BFS）很接近。

算法

这个方法的思路就是，我们需要交换树中所有节点的左孩子和右孩子。因此可以创一个队列来存储所有左孩子和右孩子还没有被交换过的节点。开始的时候，只有根节点在这个队列里面。只要这个队列不空，就一直从队列中出队节点，然后互换这个节点的左右孩子节点，接着再把孩子节点入队到队列，对于其中的空节点不需要加入队列。最终队列一定会空，这时候所有节点的孩子节点都被互换过了，直接返回最初的根节点就可以了。

复杂度分析

既然树中的每个节点都只被访问/入队一次，时间复杂度就是 $O(n)$，其中 n 是树中节点的个数。

空间复杂度是 $O(n)$，即使在最坏的情况下，也就是队列里包含了树中所有的节点。对于一颗完整二叉树来说，叶子节点那一层拥有 $\lceil \frac{n}{2} \rceil = O(n)$ 个节点。

方法 2：迭代

```
class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) return null;
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        while (!queue.isEmpty()) {
            TreeNode current = queue.poll();
            TreeNode temp = current.left;
            current.left = current.right;
            current.right = temp;
            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }
        return root;
    }
}
```

42、上下翻转二叉树（156）

更多硬核内容就在公众号「yes 的练级攻略」

156. 上下翻转二叉树

难度 中等 44 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，其中所有的右节点要么是具有兄弟节点（拥有相同父节点的左节点）的叶节点，要么为空，将此二叉树上下翻转并将它变成一棵树， 原来的右节点将转换成左叶节点。返回新的根。

例子:

输入： [1,2,3,4,5]

```
    1
   / \
  2   3
 / \
4   5
```

输出： 返回二叉树的根 [4,5,2,#,#,3,1]

```
   4
  / \
 5   2
    / \
   3   1
```

说明:

对 [4,5,2,#,#,3,1] 感到困惑? 下面详细介绍请查看 二叉树是如何被序列化的。

二叉树的序列化遵循层次遍历规则，当没有节点存在时，'#' 表示路径终止符。

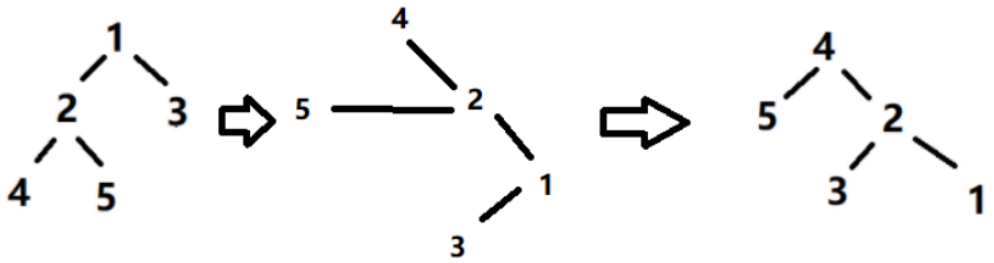
这里有一个例子:

```
   1
  / \
 2   3
    /
   4
    \
     5
```

上面的二叉树则被序列化为 [1,2,3,#,#,4,#,#,5].

https://leetcode-cn.com/problems/binary-tree-upside-down/solution/156-shang-xia-fan-zhuan-er-cha-shu-by-klb/

光看题目还是懵懵懂懂，不如我把一个二叉树倒着过来看：



我们可以看出一个规律：对于任意一个节点，它的右结点变为左子节点的左子节点，它自己变为它左子节点的右子结点。

我们知道，对于一个节点来说，关心的是它的左右子结点，至于它的父节点是谁可以递归求解。

因此，我们需要对每一个节点的左子节点进行重新设置左右节点。

定义一个 head 作为新的头结点。

定义 DFS 函数，功能如下：

1、node 为叶子节点且 head 为空，则 node 的左子节点成为新的头结点，head = node.left;

2、node 为 null，return;

3、node 不是叶子节点也不为 null，则把 node 的右子节点作为 node 的左子节点的左子节点，node 自己作为它左子节点的右子节点，最后 node 的左右子结点置空，即不再是原来左右子结点的父节点了。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    // 处理之后的根节点
    TreeNode head;
    public  TreeNode upsideDownBinaryTree(TreeNode root) {
        if (root == null) return null;
        dfs(root);  // 对 root 进行上下翻转
        return head;
    }

    public  TreeNode dfs(TreeNode node) {
```

```
        if (node == null) return null;
        if (node.left == null && node.right == null) {
            if (head == null) head = node; // 最左边的节点即为实际的根节点
            return node;
        }
        TreeNode left = dfs(node.left);
        TreeNode right = dfs(node.right);
        if (left != null) {
            // 左孩子的左子树为当前的右节点
            left.left = right;
            // 左孩子的右子树为当前父节点
            left.right = node;
        }
        // 清空的当前父节点的左右子树
        node.right = null;
        node.left = null;
        return node;
    }
}
```

43、二叉树的最大深度（104）（剑指 55-1）

104. 二叉树的最大深度

难度 简单 568 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例:
给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/solution/er-cha-shu-de-zui-da-shen-du-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：递归

思路与算法

如果我们知道了左子树和右子树的最大深度 l 和 r，那么该二叉树的最大深度即为

$$\max(l, r) + 1$$

而左子树和右子树的最大深度又可以以同样的方式进行计算。因此我们在计算当前二叉树的最大深度时，可以先递归计算出其左子树和右子树的最大深度，然后在 $O(1)$ 时间内计算出当前二叉树的最大深度。递归在访问到空节点时退出。

☆ 收藏

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为二叉树节点的个数。每个节点在递归中只被遍历一次。
- 空间复杂度：$O(height)$，其中 $height$ 表示二叉树的高度。递归函数需要栈空间，而栈空间取决于递归的深度，因此空间复杂度等价于二叉树的高度。

方法 1：递归（0ms）

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;
        int depthLeft = maxDepth (root.left);
        int depthRight = maxDepth (root.right);
        return Math.max(depthLeft,depthRight) + 1;
    }
}
```

方法二：广度优先搜索

思路与算法

我们也可以用「广度优先搜索」的方法来解决这道题目，但我们需要对其进行一些修改，此时我们广度优先搜索的队列里存放的是「当前层的所有节点」。每次拓展下一层的时候，不同于广度优先搜索的每次只从队列里拿出一个节点，我们需要将队列里的所有节点都拿出来进行拓展，这样能保证每次拓展完的时候队列里存放的是当前层的所有节点，即我们是一层一层地进行拓展，最后我们用一个变量 ans 来维护拓展的次数，该二叉树的最大深度即为 ans。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为二叉树的节点个数。与方法一同样的分析，每个节点只会被访问一次。
- 空间复杂度：此方法空间的消耗取决于队列存储的元素数量，其在最坏情况下会达到 $O(n)$。

方法 2：BFS 迭代

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int maxDepth(TreeNode root) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (root == null) return 0;
        Queue<TreeNode> que = new LinkedList<>();
        que.offer(root);
        int res = 0;
        while (!que.isEmpty()){
            int size = que.size();
            while (size > 0) {
                TreeNode node = que.poll();
                if (node.left != null) que.offer(node.left);
                if (node.right != null) que.offer(node.right);
                size--;
            }
            res++;
        }
        return res;
    }
}
```

44、N 叉树的最大深度（559）

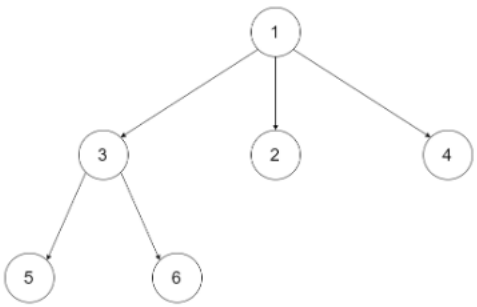
559. N叉树的最大深度

难度 简单 112 收藏 分享 切换为英文 关注

给定一个 N 叉树，找到其最大深度。

最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

例如，给定一个 3叉树 ：



我们应返回其最大深度，3。

说明:

1. 树的深度不会超过 1000 。
2. 树的节点总不会超过 5000 。

https://leetcode-cn.com/problems/maximum-depth-of-n-ary-tree/solution/ncha-shu-de-zui-da-shen-du-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：每个节点遍历一次，所以时间复杂度是 $O(N)$，其中 N 为节点数。
- 空间复杂度：最坏情况下，树完全非平衡，
  例如 每个节点有且仅有一个孩子节点，递归调用会发生 N 次（等于树的深度），所以存储调用栈需要 $O(N)$。
  但是在最好情况下（树完全平衡），树的高度为 $\log(N)$。
  所以在此情况下空间复杂度为 $O(\log(N))$。

方法 1：递归

```
/*
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }

    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
    public int maxDepth(Node root) {
        if (root == null) return 0;
        if (root.children.isEmpty()) return 1;
        List<Integer> list = new ArrayList<>();
        for (Node cur : root.children) {
            list.add(maxDepth(cur));
        }
        return Collections.max(list) + 1;
    }
}
```

方法二: 迭代

我们还可以在堆栈的帮助下将上面的递归转换为迭代。

思路是是使用深度优先搜索策略访问每个节点，同时更新每次访问时的最大深度。

所以可以从包含根节点的、对应深度为 1 的栈开始。
然后继续迭代，从栈中弹出当前节点并将子节点压入栈中，每次都更新对应深度。

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：$O(N)$。

方法 2：迭代

```
class Solution {
//迭代：层序遍历
    public int maxDepth(Node root) {
        if(root == null) return 0;
        Queue<Node> queue = new LinkedList<>();
        queue.add(root);
        int maxDepth = 0;
        while(!queue.isEmpty()){
            int count = queue.size();
            maxDepth++;
            while(count > 0){
                count--;
                Node cur = queue.poll();
                for(Node node : cur.children){
                    if(node != null)
                        queue.add(node);
                }
            }
        }
        return maxDepth;
    }
}
```

45、二叉树的最小深度（111）

更多硬核内容就在公众号「yes 的练级攻略」

111. 二叉树的最小深度

难度 简单 300 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最小深度 2.

https://leetcode-cn.com/problems/minimum-depth-of-binary-tree/solution/er-cha-shu-de-zui-xiao-shen-du-by-leetcode/

复杂度分析

- 时间复杂度：我们访问每个节点一次，时间复杂度为 $O(N)$，其中 N 是节点个数。
- 空间复杂度：最坏情况下，整棵树是非平衡的，例如每个节点都只有一个孩子，递归会调用 N（树的高度）次，因此栈的空间开销是 $O(N)$。但在最好情况下，树是完全平衡的，高度只有 $\log(N)$，因此在这种情况下空间复杂度只有 $O(\log(N))$。

1、递归（0ms）

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int minDepth(TreeNode root) {
        if (root == null) return 0;
        if (root.left == null && root.right == null) return 1;
        int leftDep = Integer.MAX_VALUE, rightDep = Integer.MAX_VALUE;
        if (root.left != null) leftDep = Math.min(leftDep, minDepth(roo
t.left));
```

```
        if (root.right != null) rightDep = Math.min(rightDep, minDepth(
root.right));
        return Math.min(leftDep, rightDep) + 1;
    }
}
```

方法二：广度优先搜索迭代

深度优先搜索方法的缺陷是所有节点都必须访问到，以保证能够找到最小深度。因此复杂度是 $O(N)$ 。

一个优化的方法是利用广度优先搜索，我们按照树的层去迭代，第一个访问到的叶子就是最小深度的节点，这样就不用遍历所有的节点了。

复杂度分析

- 时间复杂度：最坏情况下，这是一棵平衡树，我们需要按照树的层次一层一层的访问完所有节点，除去最后一层的节点。这样访问了 $N/2$ 个节点，因此复杂度是 $O(N)$。
- 空间复杂度：和时间复杂度相同，也是 $O(N)$。

2、迭代 BFS

```
class Solution {
  public int minDepth(TreeNode root) {
    LinkedList<Pair<TreeNode, Integer>> stack = new LinkedList<>();
    if (root == null) {
      return 0;
    }
    else {
      stack.add(new Pair(root, 1));
    }

    int current_depth = 0;
    while (!stack.isEmpty()) {
      Pair<TreeNode, Integer> current = stack.poll();
      root = current.getKey();
      current_depth = current.getValue();
      if ((root.left == null) && (root.right == null)) {
        break;
      }
      if (root.left != null) {
        stack.add(new Pair(root.left, current_depth + 1));
      }
      if (root.right != null) {
        stack.add(new Pair(root.right, current_depth + 1));
      }
    }
    return current_depth;
  }
}
```

46、平衡二叉树（110）（剑指 55-2）【检查平衡性（面试题 04.04）】

110. 平衡二叉树

难度 简单 394 收藏 分享 切换为英文

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

一个二叉树*每个节点*的左右两个子树的高度差的绝对值不超过1。

示例 1:

给定二叉树 `[3,9,20,null,null,15,7]`

```
    3
   / \
  9  20
    /  \
   15   7
```

返回 `true` 。

示例 2:

给定二叉树 `[1,2,2,3,3,null,null,4,4]`

```
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
```

返回 `false` 。

https://leetcode-cn.com/problems/balanced-binary-tree/solution/balanced-binary-tree-di-gui-fang-fa-by-jin40789108/

从底至顶（提前阻断）

> 此方法为本题的最优解法，但"从底至顶"的思路不易第一时间想到。

思路是对二叉树做先序遍历，从底至顶返回子树最大高度，若判定某子树不是平衡树则 "剪枝"，直接向上返回。

算法流程：

`recur(root)`：

- **递归返回值：**
  1. 当节点 `root` 左 / 右子树的高度差 < 2：则返回以节点 `root` 为根节点的子树的最大高度，即节点 `root` 的左右子树中最大高度加 1（`max(left, right) + 1`）；
  2. 当节点 `root` 左 / 右子树的高度差 ≥ 2：则返回 -1，代表 **此子树不是平衡树** 。
- **递归终止条件：**
  1. 当越过叶子节点时，返回高度 0；
  2. 当左（右）子树高度 `left== -1` 时，代表此子树的 **左（右）子树** 不是平衡树，因此直接返回 -1；

`isBalanced(root)`：

- **返回值：** 若 `recur(root) != 1`，则说明此树平衡，返回 $true$；否则返回 $false$。

复杂度分析：

- **时间复杂度 $O(N)$：** N 为树的节点数；最差情况下，需要递归遍历树的所有节点。
- **空间复杂度 $O(N)$：** 最差情况下（树退化为链表时），系统递归需要使用 $O(N)$ 的栈空间。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isBalanced(TreeNode root) {
        return recur(root) != -1;
    }
    private int recur(TreeNode root) {
        if (root == null) return 0;
        int left = recur(root.left);
        if(left == -1) return -1;
        int right = recur(root.right);
        if(right == -1) return -1;
        return Math.abs(left - right) < 2 ? Math.max(left, right) + 1 : -1;
    }
}
```

47、二叉树的直径（543）

543. 二叉树的直径

难度 简单 420 收藏 分享 切换为英文 关注 反馈

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例：
给定二叉树

```
          1
         / \
        2   3
       / \
      4   5
```

返回 **3**, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

https://leetcode-cn.com/problems/diameter-of-binary-tree/solution/java-shen-du-you-xian-bian-li-dfs-by-sugar-31/

任意两个节点之间的边数都可能是最大直径

最大的直径不一定包括根节点

这道题很容易有的误区就是：从根节点出发，找到左边数的最大深度 `leftDepth`，再找到右边树的最大深度 `rigthDepth`

然后 `return leftDepth+rigthDepth + 1` （如果二叉树的根节点深度为0的话）

从上面的图可以看出，情况并不是这样，因为最大值不一定要包含根节点。

解题思路

从上面的分析可知，最大值不一定包含根节点，但是一定是：经过一个节点，该节点左右子树的最大深度之和 +1（二叉树的根节点深度为 0）
于是，可以使用 DFS，找出所有节点的最大直径，在取出最大值 `res`；

定义一个全局变量 `res`，用来记录最大直径
使用 `dfs(root)` 遍历所有的节点，**`dfs(root)` 的作用是：找出以 `root` 为根节点的二叉树的最大深度**

将根节点的深度定义为 1（和上面分析的深度定义不一样）
`root` 为跟节点的最大深度为 `Math.max(leftDepth,rigthDepth) + 1`
`res` 取值为以经过 `root`，左右子树的最大深度之和 `leftDepth + rigthDepth`（不用加 1，是因为根节点的深度是 1）
通过递归，找到 `res` 的最大值

543 具体代码

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int max = 0;
    public int diameterOfBinaryTree(TreeNode root){
        depth(root);
        return max;
    }
    public int depth(TreeNode node) {
        if(node == null) return 0;
        int LD = depth(node.left);
        int RD = depth(node.right);
        max = Math.max(LD + RD, max);
        return Math.max(LD, RD) + 1;
    }
}
```

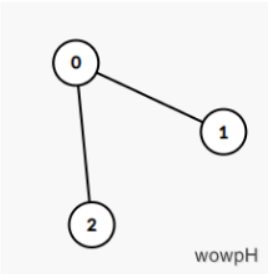
48、树的直径（1245）

给你这棵「无向树」，请你测算并返回它的「直径」：这棵树上最长简单路径的 **边数**。

我们用一个由所有「边」组成的数组 `edges` 来表示一棵无向树，其中 `edges[i] = [u, v]` 表示节点 `u` 和 `v` 之间的双向边。

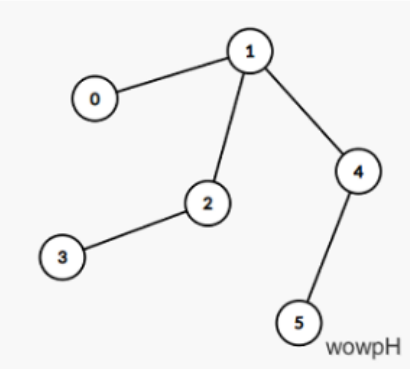
树上的节点都已经用 `{0, 1, ..., edges.length}` 中的数做了标记，每个节点上的标记都是独一无二的。

示例 1：



输入：edges = [[0,1],[0,2]]
输出：2
解释：这棵树上最长的路径是 1 - 0 - 2，边数为 2。

示例 2：




```
输入：edges = [[0,1],[1,2],[2,3],[1,4],[4,5]]
输出：4
解释：这棵树上最长的路径是 3 - 2 - 1 - 4 - 5，边数为 4。
```

提示：

- `0 <= edges.length < 10^4`
- `edges[i][0] != edges[i][1]`
- `0 <= edges[i][j] <= edges.length`
- `edges` 会形成一棵无向树

https://blog.csdn.net/pfdvnah/article/details/102922383

题解

采用 **邻接表** 的存储结构保存树的信息。

深度优先搜索所有端点。最长的就是最终的结果。

```
class Solution {
    public int treeDiameter(int[][] edges) {
        int n = edges.length + 1;// 结点数
        // 用邻接表的存储结构保存树的信息
        List<List<Integer>> tree = new ArrayList<>();
        for (int i = 0; i < n; ++i) {
            tree.add(new ArrayList<>());
        }

        // 根据边的信息生成邻接表
        for (int[] edge : edges) {
            tree.get(edge[0]).add(edge[1]);
```

```
            tree.get(edge[1]).add(edge[0]);
        }

        int ret = 0;
        for (int i = 0; i < n; ++i) {
            // 最长的路径的端点的度一定为 1，这样能去掉不必要的 DFS
            if (tree.get(i).size() == 1) {
                ret = Math.max(ret, dfs(tree, -1, i));
            }
        }
        return ret - 1;// 题目问的是直径（边数），所以要减 1
    }

    // 返回最大结点数。@wowpH
    private int dfs(List<List<Integer>> tree,
                    int pre,// 前驱点
                    int cur) {// 当前点
        int ret = 0;
        List<Integer> list = tree.get(cur);// 与当前结点相连的结点
        for (int i = 0; i < list.size(); ++i) {
            int next = list.get(i);
            if (next != pre) {// 不是前驱结点，下一层 DFS
                ret = Math.max(ret, dfs(tree, cur, next));
            }
        }
        return ret + 1;// 加上当前结点
    }
}
```

49、二叉树的所有路径（257）

禁止商用 公众号：yes的练级攻略

257. 二叉树的所有路径

难度 简单 312 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，返回所有从根节点到叶子节点的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

输入:

```
   1
 /   \
2     3
 \
  5
```

输出: ["1->2->5", "1->3"]

解释: 所有根节点到叶子节点的路径为: 1->2->5, 1->3

https://leetcode-cn.com/problems/binary-tree-paths/solution/er-cha-shu-de-suo-you-lu-jing-by-leetcode-solution/

方法一：深度优先搜索

思路与算法

最直观的方法是使用深度优先搜索。在深度优先搜索遍历二叉树时，我们需要考虑当前的节点以及它的孩子节点。

- 如果当前节点**不是叶子节点**，则在当前的路径末尾添加该节点，并继续递归遍历该节点的每一个孩子节点。
- 如果当前节点**是叶子节点**，则在当前路径末尾添加该节点后我们就得到了一条从根节点到叶子节点的路径，将该路径加入到答案即可。

如此，当遍历完整棵二叉树以后我们就得到了所有从根节点到叶子节点的路径。当然，深度优先搜索也可以使用非递归的方式实现，这里不再赘述。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 表示节点数目。在深度优先搜索中每个节点会被访问一次且只会被访问一次，每一次会对 `path` 变量进行拷贝构造，时间代价为 $O(N)$，故时间复杂度为 $O(N^2)$。
- 空间复杂度：$O(N^2)$，其中 N 表示节点数目。除答案数组外我们需要考虑递归调用的栈空间。在最坏情况下，当二叉树中每个节点只有一个孩子节点时，即整棵二叉树呈一个链状，此时递归的层数为 N，此时每一层的 `path` 变量的空间代价的总和为 $O(\sum_{i=1}^{N} i) = O(N^2)$ 空间复杂度为 $O(N^2)$。最好情况下，当二叉树为平衡二叉树时，它的高度为 $\log N$，此时空间复杂度为 $O((\log N)^2)$。

方法 1：递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
```

```
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<String> binaryTreePaths(TreeNode root) {
        List<String> res = new LinkedList<>();
        dfs(root, "", res);
        return res;
    }
    public void dfs(TreeNode root, String path, List<String> res) {
        if (root == null) return;
        path += String.valueOf(root.val);
        if ((root.left == null) && (root.right == null)) res.add(path);
        else {
            path += "->";
            dfs(root.left, path, res);
            dfs(root.right, path, res);
        }
    }
}
```

方法二：广度优先搜索

思路与算法

我们也可以用广度优先搜索来实现。我们维护一个队列，存储节点以及根到该节点的路径。一开始这个队列里只有根节点。在每一步迭代中，我们取出队列中的首节点，如果它**是叶子节点**，则将它对应的路径加入到答案中。如果它**不是叶子节点**，则将它的所有孩子节点加入到队列的末尾。当队列为空时广度优先搜索结束，我们即能得到答案。

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 表示节点数目。分析同方法一。
- 空间复杂度：$O(N^2)$，其中 N 表示节点数目。在最坏情况下，队列中会存在 N 个节点，保存字符串的队列中每个节点的最大长度为 N，故空间复杂度为 $O(N^2)$。

方法 2：迭代

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
```

```
    public List<String> binaryTreePaths(TreeNode root) {
        LinkedList<String> paths = new LinkedList();
        if (root == null)
            return paths;

        LinkedList<TreeNode> node_stack = new LinkedList();
        LinkedList<String> path_stack = new LinkedList();
        node_stack.add(root);
        path_stack.add(String.valueOf(root.val));
        TreeNode node;
        String path;
        while (!node_stack.isEmpty()) {
            node = node_stack.pollLast();
            path = path_stack.pollLast();
            if ((node.left == null) && (node.right == null))
                paths.add(path);
            if (node.left != null) {
                node_stack.add(node.left);
                path_stack.add(path + "->" + String.valueOf(node.left.val));
            }
            if (node.right != null) {
                node_stack.add(node.right);
                path_stack.add(path + "->" + String.valueOf(node.right.val));
            }
        }
        return paths;
    }
}
```

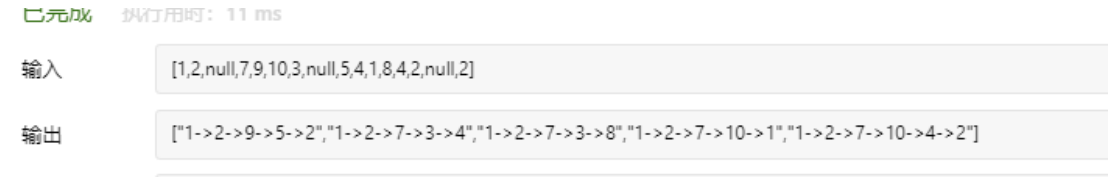
50、二叉树中所有路径中最大路径和的路径（从根到叶子）

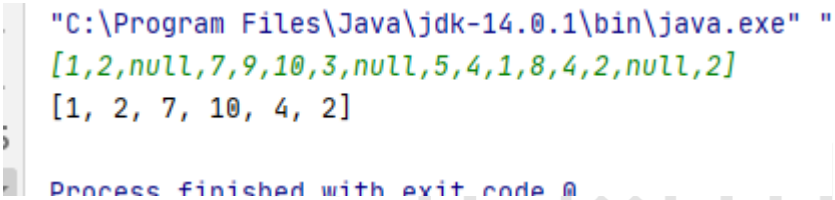
```
import java.util.*;
class TreeNode{
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x){
        this.val = x;
    }
}
public class Main {
    private static int tolMax = 0;
    private static Deque<Integer> res;
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        String data = sc.nextLine();
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        TreeNode root = deserialize(data);
        Deque<Integer> res = maximumPathSum(root);
        System.out.println(res);
    }
    //生成 TreeNode
    public static TreeNode deserialize(String data) {
        if(data.length() < 3) return null;
        String[] strs = data.substring(1, data.length()-1).split(","); //因为头尾有括号
        TreeNode root = new TreeNode(Integer.valueOf(strs[0]));
        Deque<TreeNode> que = new LinkedList<>();
        que.add(root);
        TreeNode cur;
        int idx = 0;
        while(true){
            cur=que.remove();
            if(++idx >= strs.length) break;
            if(strs[idx].compareTo("null") != 0) {
                cur.left = new TreeNode(Integer.valueOf(strs[idx]));
                que.add(cur.left);
            }
            if(++idx >= strs.length) break;
            if(strs[idx].compareTo("null") != 0) {
                cur.right=new TreeNode(Integer.valueOf(strs[idx]));
                que.add(cur.right);
            }
        }
        return root;
    }
    //计算最大的路径
    public static Deque<Integer> maximumPathSum(TreeNode root) {
        if (root == null) return res;
        Deque<Integer> path = new ArrayDeque<>();
        int curMax = 0, tolMax = 0;
        dfs(root, path, curMax);
        return res;
    }
    //递归
    public static void dfs(TreeNode node, Deque<Integer> path, int curMax) {
        if (node == null) return;
        curMax += node.val;
        path.addLast(node.val);
        if (curMax > tolMax && node.left == null && node.right == null) {
            res = new ArrayDeque<>(path);
            tolMax = curMax;
```

```
            path.removeLast();
            return;
        }
        dfs(node.left, path, curMax);
        dfs(node.right, path, curMax);
        path.removeLast();
    }
}
```





51、二叉树中的最大路径和（124）

124. 二叉树中的最大路径和

难度 困难 523 收藏 分享 切换为英文 关注 反馈

给定一个**非空**二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径**至少包含一个**节点，且不一定经过根节点。

示例 1:

输入: [1,2,3]

```
       1
      / \
     2   3
```

输出: 6

示例 2:

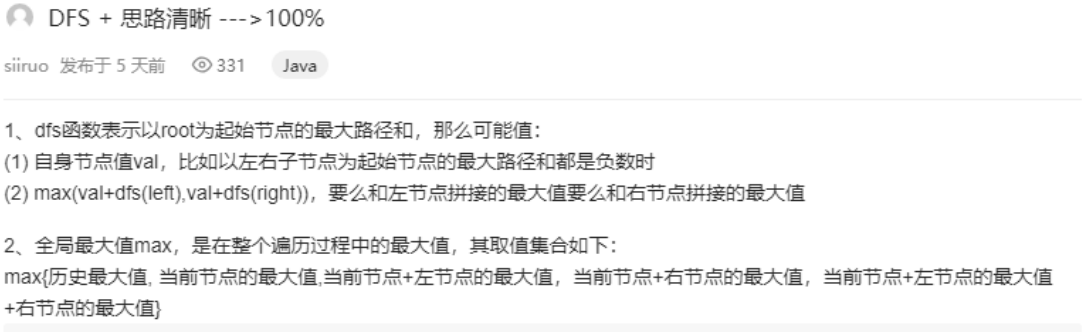
输入: [-10,9,20,null,null,15,7]

```
   -10
   / \
  9  20
    /  \
   15   7
```

输出: 42

https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/solution/dfs-si-lu-qing-xi-100-by-siiruo/

更多硬核内容就在公众号「yes 的练级攻略」



方法 1：DFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    int max = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        dfs(root);
        return max;
    }
    public int dfs (TreeNode root) {
        if (root == null) return 0;
        int leftMax = dfs(root.left);
        int rightMax = dfs(root.right);
        int curMax = Math.max(root.val + leftMax + rightMax, Math.max(r
oot.val + leftMax, root.val + rightMax));
        max = Math.max(max, Math.max(curMax, root.val));
        return Math.max(root.val, Math.max(root.val + leftMax, root.val
 + rightMax));
    }
}
```

https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/solution/jian-dan-de-jie-fa-ji-bai-9965de-yong-hu-by-xu-kon/

更多硬核内容就在公众号「yes 的练级攻略」

简单的解法，击败99.65%的用户

大魔导师 发布于 2020-05-03 1.3k Java

解题思路

改题的题意比较难以理解，很容易跑题而导致解答错误。
该题说明的是，能够找到1个节点到另外一个节点的最大路径和，因此思路如下
定义一个递归方法，该方法的功能是：得到当前树的最大的路径和，而计算当前的最大路径和是通过递归计算左右子树的最大路径好得到的。最终当前树的最大路径和为Math.max(current.val, Math.max(左子树最大路径和，右子树最大路径和));

方法 2：DFS 更简短的

```
class Solution {
    int max = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        dfs(root);
        return max;
    }
    public int dfs (TreeNode root) {
        if (root == null) return 0;
        int leftMax = Math.max(dfs(root.left),0);
        int rightMax = Math.max(dfs(root.right),0);
        max = Math.max(max, leftMax + rightMax + root.val);
        return root.val + Math.max(leftMax,rightMax);
    }
}
```

52、路径总和（112）

112. 路径总和

难度 简单 347 收藏 分享 切换为英文 关注 反馈

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

说明: 叶子节点是指没有子节点的节点。

示例:
给定如下二叉树，以及目标和 `sum = 22`，

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \      \
        7    2      1
```

返回 `true`，因为存在目标和为 22 的根节点到叶子节点的路径 `5->4->11->2`。

https://leetcode-cn.com/problems/path-sum/solution/jian-dan-di-gui-bi-xu-miao-dong-by-sweetiee-2/

方法 1：递归代码

```
class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
```

```
        if (root == null) {
            return false;
        }
        if (root.left == null && root.right == null) {
            return sum == root.val;
        }
        return hasPathSum(root.left, sum - root.val) || hasPathSum(root
.right, sum - root.val);
    }
}
```

https://leetcode-cn.com/problems/path-sum/solution/lu-jing-zong-he-by-leetcode-solution/

方法 2：BFS 代码

```
class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if (root == null) {
            return false;
        }
        Queue<TreeNode> queNode = new LinkedList<TreeNode>();
        Queue<Integer> queVal = new LinkedList<Integer>();
        queNode.offer(root);
        queVal.offer(root.val);
        while (!queNode.isEmpty()) {
            TreeNode now = queNode.poll();
            int temp = queVal.poll();
            if (now.left == null && now.right == null) {
                if (temp == sum) {
                    return true;
                }
                continue;
            }
            if (now.left != null) {
                queNode.offer(now.left);
                queVal.offer(now.left.val + temp);
            }
            if (now.right != null) {
                queNode.offer(now.right);
                queVal.offer(now.right.val + temp);
            }
        }
        return false;
    }
}
```

53、路径总和 2（113）【二叉树中和为某一值的路径（剑指 34）】

更多硬核内容就在公众号「yes 的练级攻略」

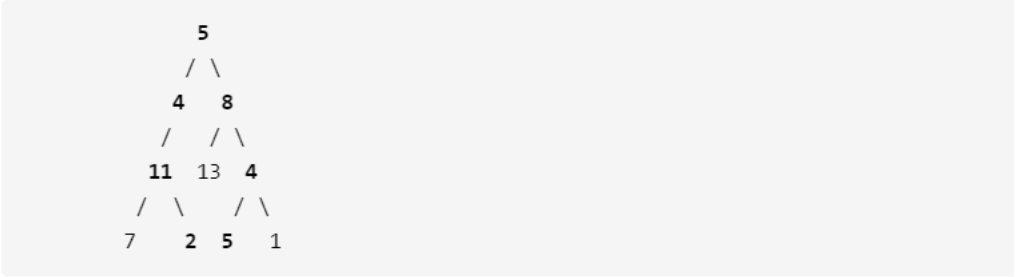
113. 路径总和 II

难度 中等 280 收藏 分享 切换为英文 关注 反馈

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

说明: 叶子节点是指没有子节点的节点。

示例:
给定如下二叉树，以及目标和 `sum = 22`，



返回:

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

https://leetcode-cn.com/problems/path-sum-ii/solution/hui-su-suan-fa-shen-du-you-xian-bian-li-zhuang-tai/

这道题写法比较多，但是核心思想还**是「深度优先遍历」（回溯算法）**，以考虑清楚「回溯算法」是「深度优先遍历」的一些细节问题。

读者可以思考：

- 在 `res.add(new ArrayList<>(path));` 这段代码为什么要套一层 `new ArrayList<>(path)`，不这么做是否可以；
- 「深度优先遍历」不同的「状态」之间差别很小，因此「状态重置」很方便。「广度优先遍历」就没有这种特点，因此「深度优先遍历」可以成为强大的「回溯搜索」算法；
- 这里的「状态」是指完成一件事情进行到哪一个阶段，在下面的代码中：`path` 、`sum` 都是状态变量。

113 代码：深度优先加状态重置

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.List;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
```

```
    }
}
public class Solution {

    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) {
            return res;
        }

        // Java 文档中 Stack 类建议使用 Deque 代替 Stack，注意：只使用栈的相关接口
        Deque<Integer> path = new ArrayDeque<>();
        pathSum(root, sum, path, res);
        return res;
    }

    public void pathSum(TreeNode node, int sum, Deque<Integer> path, List<List<Integer>> res)
{
        // 打开注释调试
        // System.out.println(path);

        // 递归终止条件
        if (node == null) {
            return;
        }

        // 沿途结点必须选择，这个时候要做两件事：1、sum 减去这个结点的值；2、添
加到 path 里
        sum -= node.val;
        path.addLast(node.val);

        if (sum == 0 && node.left == null && node.right == null) {
            // path 全局只有一份，必须做拷贝
            res.add(new ArrayList<>(path));
            // 注意：这里 return 之前必须重置
            path.removeLast();
            return;
        }

        pathSum(node.left, sum, path, res);
        pathSum(node.right, sum, path, res);
        // 递归完成以后，必须重置变量
        path.removeLast();
    }
```

禁止商用 公众号：yes的练级攻略

```
    public static void main(String[] args) {
        TreeNode treeNode5 = new TreeNode(5);
        TreeNode treeNode4 = new TreeNode(4);
        TreeNode treeNode8 = new TreeNode(8);
        TreeNode treeNode11 = new TreeNode(11);
        TreeNode treeNode13 = new TreeNode(13);
        TreeNode treeNode4_ = new TreeNode(4);
        TreeNode treeNode7 = new TreeNode(7);
        TreeNode treeNode2 = new TreeNode(2);
        TreeNode treeNode5_ = new TreeNode(5);
        TreeNode treeNode1 = new TreeNode(1);

        treeNode5.left = treeNode4;
        treeNode5.right = treeNode8;
        treeNode4.left = treeNode11;
        treeNode11.left = treeNode7;
        treeNode11.right = treeNode2;
        treeNode8.left = treeNode13;
        treeNode8.right = treeNode4_;
        treeNode4_.left = treeNode5_;
        treeNode4_.right = treeNode1;

        Solution solution = new Solution();
        List<List<Integer>> res = solution.pathSum(treeNode5, 22);
        System.out.println(res);
    }
}
```

54、路径总和 3（437）【求和路径（面试题 04.12）】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

437. 路径总和 III

难度 中等 507 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有：

1.  5 -> 3
2.  5 -> 2 -> 1
3.  -3 -> 11
```

https://leetcode-cn.com/problems/path-sum-iii/solution/javajie-fa-shi-jian-100-kong-jian-93-by-xiao-chao-/

1. 遍历每个节点。 关键点：递归
2. 计算以当前节点为路径终点的所有路径和。 关键点：用一个数组保存从根节点到当前节点路径

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public int pathSum(TreeNode root, int sum) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        return pathSum(root, sum, new int[1000], 0);
    }

    public int pathSum(TreeNode root, int sum, int[] array/*保存路径
*/, int p/*指向路径终点*/) {
        if (root == null) {
            return 0;
        }
        int tmp = root.val;
        int n = root.val == sum ? 1 : 0;
        for (int i = p - 1; i >= 0; i--) {
            tmp += array[i];
            if (tmp == sum) {
                n++;
            }
        }
        array[p] = root.val;
        int n1 = pathSum(root.left, sum, array, p + 1);
        int n2 = pathSum(root.right, sum, array, p + 1);
        return n + n1 + n2;
    }
}
```

55、路径和 4（666）

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

666. 路径和 IV

难度 中等 19 收藏 分享 切换为英文 关注 反馈

对于一棵深度小于 5 的树，可以用一组三位十进制整数来表示。

对于每个整数：

1. 百位上的数字表示这个节点的深度 D，1 <= D <= 4。
2. 十位上的数字表示这个节点在当前层所在的位置 P，1 <= P <= 8。位置编号与一棵满二叉树的位置编号相同。
3. 个位上的数字表示这个节点的权值 V，0 <= V <= 9。

给定一个包含三位整数的 升序 数组，表示一棵深度小于 5 的二叉树，请你返回从根到所有叶子结点的路径之和。

样例 1:

```
输入: [113, 215, 221]
输出: 12
解释:
这棵树形状如下：
    3
   / \
  5   1

路径和 = (3 + 5) + (3 + 1) = 12.
```

样例 2:

```
输入: [113, 221]
输出: 4
解释:
这棵树形状如下：
    3
     \
      1

路径和 = (3 + 1) = 4.
```

YOSHI练级攻略

https://leetcode-cn.com/problems/path-sum-iv/solution/lu-jing-he-iv-by-leetcode/

方法一：转换成树 [Accepted]

将数组给定的数据构造一颗树。然后计算从根节点到每个叶子结点的路径之和，就得到答案。

算法：

- 分为两个步骤，构造树和遍历树。
- 在树的构造过程中，我们有深度、位置和权值这些信息，我们可以根据条件 `pos - 1 < 2*(depth - 2)` 来判断结点在右边还是左边。例如，当 `depth = 4` 时，`pos` 可取 `1, 2, 3, 4, 5, 6, 7, 8`，当 `pos<=4` 时，结点的位置在左边。
- 在遍历过程中，我们执行深度优先搜索的策略遍历树，并沿着我们所走过的路径记录当前和。每当我们到达一个叶结点 `(node.left == null && node.right == null)` 时，将该路径的和添加到答案中。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(N)$。其中 N 是 `nums` 的长度。
- 空间复杂度：$O(N)$，深度优先搜索中隐式调用堆栈的大小。

方法二：直接遍历 [Accepted]

算法：

在方法 1 中，我们将在树上进行深度优先搜索。一个省时的想法是，我们根据等式 `root = num / 10 = 10 * depth + pos` 作为根节点的唯一标识符。则左子结点的标识符是 `left = 10 * (depth + 1) + 2 * pos - 1`，而右子节点则是 `right = left + 1`。

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：$O(N)$，分析与方法 1 相同。

```
class Solution {
    int ans = 0;
    Map<Integer, Integer> values;
    public int pathSum(int[] nums) {
        values = new HashMap();
        for (int num: nums)
            values.put(num / 10, num % 10);

        dfs(nums[0] / 10, 0);
        return ans;
    }
    public void dfs(int node, int sum) {
        if (!values.containsKey(node)) return;
        sum += values.get(node);
        int depth = node / 10, pos = node % 10;
        int left = (depth + 1) * 10 + 2 * pos - 1;
        int right = left + 1;

        if (!values.containsKey(left) && !values.containsKey(right)) {
            ans += sum;
        } else {
            dfs(left, sum);
            dfs(right, sum);
        }
    }
}
```

56、同长同值路径（687）

更多硬核内容就在公众号「yes 的练级攻略」

687. 最长同值路径

难度 简单 342 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，找到最长的路径，这个路径中的每个节点具有相同值。 这条路径可以经过也可以不经过根节点。

注意：两个节点之间的路径长度由它们之间的边数表示。

示例 1:

输入:

```
              5
             / \
            4   5
           / \   \
          1   1   5
```

输出:

```
2
```

示例 2:

输入:

```
              1
             / \
            4   5
           / \   \
          4   4   5
```

输出:

```
2
```

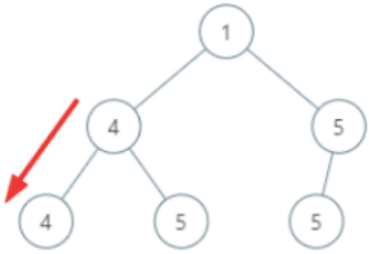
注意: 给定的二叉树不超过10000个结点。 树的高度不超过1000。

https://leetcode-cn.com/problems/longest-univalue-path/solution/guan-yu-di-gui-si-lu-de-chao-xiang-xi-ge-ren-jian-/

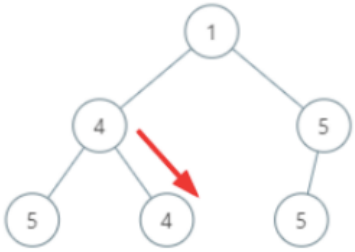
更多硬核内容就在公众号「yes 的练级攻略」

我看了官方给的答案，但是官方给的答案说实话真的不好懂，给我的感觉就是奇淫巧技，没看懂这个递归函数的功能是什么。写递归程序，重要的是四点，输入参数、输出结果、函数的功能、终止条件。把这四点写明白，递归程序就写成了。其实**递归本身就是完成一种功能的形式化描述**。再往大点说，我们写的程序也是形式化描述。计算机又不懂你这背后的逻辑是什么，它只是按照指令的形式去执行而已。所以把递归函数的**功能**搞明白，就成功了一大半。

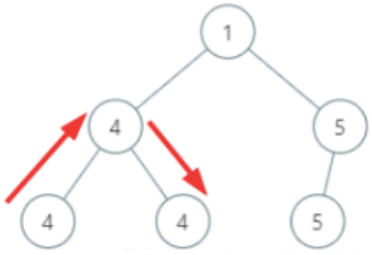
下面我们就来说说，这道题如何解决。 对于二叉树本身就具有递归这种性质的数据结构，我们需要把它看成"左子树-根节点-右子树"这种结构，就不要再囿于左子节点 / 右子节点这个思维里了。这道题其实可以看成是解决以root为路径起始点的最长路径，这其实就只有两种情况：(1)第一种是root和左子树（同值）；



(2) 第二种是root和右子树（同值）。



但是还有种特殊情况，大家应该都能想到，就是左子树-root-右子树（同值），但是这种特殊情况不影响我们递归函数的**功能**的设计。



代码说明

我一直强调的是"其实**递归本身就是完成一种功能的形式化描述**"，那么我们的递归函数的功能是什么？看过代码后，应该知道了8。我们递归函数的功能其实就是从左子树和右子树中返回一个最大的同值路径，这也就是为什么我们return的是 `maxLorRres`，即node左子树或右子树的最长同值路径。

很感谢这篇博客LeetCode 最长同值路径给了我灵感，不然看官答真的理解不了QAQ.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
```

```
    int res;
    public int longestUnivaluePath(TreeNode root) {
        res = 0;
        dfs(root);
        return res;
    }
    //递归函数功能：搜寻以 node 为起点的最长同值路径:要么是以 node 为起点的左子树，要么是以 node 为起点的右子树
    public int dfs(TreeNode root) {
        if (root == null) return 0;
        int max = 0;
        int left = dfs(root.left);
        int right = dfs(root.right);
         //这种情况对于寻找最长同值路径长有帮助，对于搜索以 root 为路径起始点的最长路径没有帮助
         if (root.left != null && root.left.val == root.val && root.right != null && root.right.val == root.val) {
             res = Math.max(res, left + right + 2);
         }
         //从左右子树中选择最长的同值路径
         if (root.left != null && root.left.val == root.val) max = left + 1;
         if (root.right != null && root.right.val == root.val) max = Math.max(max, right + 1);
         //从 ans 与 maxLorRres 中更新最大值
         res = Math.max(res, max);
         return max;
    }
}
```

57、求根到叶子节点数字总和（129）

129. 求根到叶子节点数字之和

难度 中等 173 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，它的每个结点都存放一个 0-9 的数字，每条从根到叶子节点的路径都代表一个数字。

例如，从根到叶子节点路径 1->2->3 代表数字 123。

计算从根到叶子节点生成的所有数字之和。

说明: 叶子节点是指没有子节点的节点。

示例 1:

```
输入: [1,2,3]
    1
   / \
  2   3
输出: 25
解释:
从根到叶子节点路径 1->2 代表数字 12.
从根到叶子节点路径 1->3 代表数字 13.
因此，数字总和 = 12 + 13 = 25.
```

示例 2:

```
输入: [4,9,0,5,1]
    4
   / \
  9   0
 / \
5   1
输出: 1026
解释:
从根到叶子节点路径 4->9->5 代表数字 495.
从根到叶子节点路径 4->9->1 代表数字 491.
从根到叶子节点路径 4->0 代表数字 40.
因此，数字总和 = 495 + 491 + 40 = 1026.
```

https://leetcode-cn.com/problems/sum-root-to-leaf-numbers/solution/javabfsdfsliang-chong-jie-fa-by-lixianfeng/

方法 1：BFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
```

```
 */
class Solution {
    public int sumNumbers(TreeNode root) {
        if(root == null){
            return 0;
        }
        int res = 0;
        Queue<TreeNode> nodeQueue = new LinkedList<>();
        Queue<Integer> numQueue = new LinkedList<>();
        nodeQueue.add(root);
        numQueue.add(root.val);
        while(!nodeQueue.isEmpty()){
            // 同时维护两个队列
            TreeNode node = nodeQueue.poll();
            int num = numQueue.poll();
            // 如果该节点是叶子节点，加到res 中
            if(node.left==null && node.right==null){
                res += num;
            } else {
                // 左节点不为空时，左节点进入队列，左节点对应的值是当前节点
num*10+node.left.val
                if(node.left != null){
                    nodeQueue.add(node.left);
                    numQueue.add(num*10 + node.left.val);
                }
                if(node.right != null){
                    nodeQueue.add(node.right);
                    numQueue.add(num*10 + node.right.val);
                }
            }
        }
        return res;
    }
}
```

方法 2：DFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```java
class Solution {
    int res = 0;
    public int sumNumbers(TreeNode root){
        preOrder(root,0);
        return res;
    }
    public void preOrder(TreeNode root, int val){
        if(root != null){
            // 值先乘 10，后相加
            int tmp = val * 10 + root.val;

            // 当前节点是叶子节点
            if(root.left == null && root.right == null){
                res += tmp;
            } else {
                // 当前节点的左子节点不为空，继续递归，val 的值是父节点的值，也就是 tmp
                if(root.left != null){
                    preOrder(root.left, tmp);
                }
                if(root.right != null){
                    preOrder(root.right, tmp);
                }
            }
        }
    }
}
```

58、左叶子之和（404）

404. 左叶子之和

难度 简单 186 收藏 分享 切换为英文 关注 反馈

计算给定二叉树的所有左叶子之和。

示例：

```
    3
   / \
  9  20
    /  \
   15   7
```

在这个二叉树中，有两个左叶子，分别是 9 和 15，所以返回 24

https://leetcode-cn.com/problems/sum-of-left-leaves/solution/javadi-gui-shi-yong-biao-zhi-wei-qiu-he-by-ha-ha-h/

本题要求，求所有树的左叶子节点，这里抓关键字，一定要是左孩子并且是叶子节点
如何判断叶子节点很简单 `root.left == null && root.right == null`
那么如何判断是左孩子呢？
这里我使用一个**标志位**判断叶子节点

具体是这样的写的
当你写递归函数的时候是不是递归左孩子 `sumOfLeftLeaves(root.left);`
递归右孩子 `sumOfLeftLeaves(root.right);`
这里就非常明显了，递归左孩子是不是说明这个节点一定是左孩子
我们就可以将代码这样改写 `sumOfLeftLeaves(root.left, true);` , `sumOfLeftLeaves(root.right, false);`

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int sumOfLeftLeaves(TreeNode root) {
        return sumOfLeftLeaves(root, false);
    }
    public int sumOfLeftLeaves(TreeNode root, boolean flag){
        if(root == null){
            return 0;
        }
        if(flag && root.left == null && root.right == null){
            return root.val;
        }
        int leftSum = sumOfLeftLeaves(root.left, true);
        int rightSum = sumOfLeftLeaves(root.right, false);
        return leftSum + rightSum;
    }
}
```

59、二叉树中所有距离为 K 的结点（863）

863. 二叉树中所有距离为 K 的结点

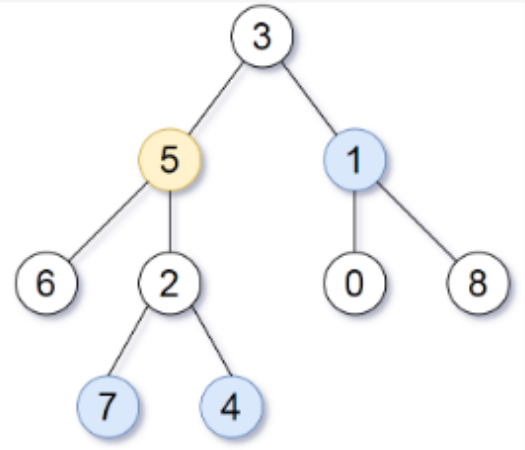
难度 中等 155 收藏 分享 切换为英文 关注 反馈

给定一个二叉树（具有根结点 `root`），一个目标结点 `target`，和一个整数值 `K`。

返回到目标结点 `target` 距离为 `K` 的所有结点的值的列表。 答案可以以任何顺序返回。

示例 1：

输入： root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2
输出： [7,4,1]
解释：
所求结点为与目标结点（值为 5）距离为 2 的结点，
值分别为 7，4，以及 1



注意，输入的 "root" 和 "target" 实际上是树上的结点。
上面的输入仅仅是对这些对象进行了序列化描述。

提示：

1. 给定的树是非空的。
2. 树上的每个结点都具有唯一的值 `0 <= node.val <= 500`。
3. 目标结点 `target` 是树上的结点。
4. `0 <= K <= 1000`.

https://leetcode-cn.com/problems/all-nodes-distance-k-in-binary-tree/solution/er-cha-shu-zhong-suo-you-ju-chi-wei-k-de-jie-dian-/

更多硬核内容就在公众号「yes 的练级攻略」

方法一： 深度优先搜索

思路

如果节点有指向父节点的引用，也就知道了距离该节点 `1` 距离的所有节点。之后就可以从 `target` 节点开始进行深度优先搜索了。

算法

对所有节点添加一个指向父节点的引用，之后做深度优先搜索，找到所有距离 `target` 节点 `K` 距离的节点。

复杂度分析

- 时间复杂度： $O(N)$，其中 N 是树中节点个数。
- 空间复杂度： $O(N)$。

方法二： 计算节点之间距离

思路

如果 `target` 节点在 `root` 节点的左子树中，且 `target` 节点深度为 `3`，那所有 `root` 节点右子树中深度为 `K - 3` 的节点到 `target` 的距离就都是 `K`。

算法

深度优先遍历所有节点。定义方法 `dfs(node)`，这个函数会返回 `node` 到 `target` 的距离。在 `dfs(node)` 中处理下面四种情况：

- 如果 `node == target`，把子树中距离 `target` 节点距离为 `K` 的所有节点加入答案。
- 如果 `target` 在 `node` 左子树中，假设 `target` 距离 `node` 的距离为 `L+1`，找出右子树中距离 `target` 节点 `K - L - 1` 距离的所有节点加入答案。
- 如果 `target` 在 `node` 右子树中，跟在左子树中一样的处理方法。
- 如果 `target` 不在节点的子树中，不用处理。

实现的算法中，还会用到一个辅助方法 `subtree_add(node, dist)`，这个方法会将子树中距离节点 `node` `K - dist` 距离的节点加入答案。

yes的练级攻略

复杂度分析

- 时间复杂度： $O(N)$，其中 N 树的大小。
- 空间复杂度： $O(N)$。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    List<Integer> ans;
```

```
TreeNode target;
int K;
public List<Integer> distanceK(TreeNode root, TreeNode target, int K) {
    ans = new LinkedList();
    this.target = target;
    this.K = K;
    dfs(root);
    return ans;
}

// Return vertex distance from node to target if exists, else -1
// Vertex distance: the number of vertices on the path from node to target
public int dfs(TreeNode node) {
    if (node == null)
        return -1;
    else if (node == target) {
        subtree_add(node, 0);
        return 1;
    } else {
        int L = dfs(node.left), R = dfs(node.right);
        if (L != -1) {
            if (L == K) ans.add(node.val);
            subtree_add(node.right, L + 1);
            return L + 1;
        } else if (R != -1) {
            if (R == K) ans.add(node.val);
            subtree_add(node.left, R + 1);
            return R + 1;
        } else {
            return -1;
        }
    }
}

// Add all nodes 'K - dist' from the node to answer.
public void subtree_add(TreeNode node, int dist) {
    if (node == null) return;
    if (dist == K)
        ans.add(node.val);
    else {
        subtree_add(node.left, dist + 1);
        subtree_add(node.right, dist + 1);
    }
}
```

}

60、从前序与中序遍历序列构造二叉树（105）【重建二叉树（剑指 7）】

105. 从前序与中序遍历序列构造二叉树

难度 中等 538 收藏 分享 切换

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树：

```
    3
   / \
  9  20
    /  \
   15   7
```

通过次数 87,784 | 提交次数 130,559

https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/solution/cong-qian-xu-yu-zhong-xu-bian-li-xu-lie-gou-zao-9/

二叉树前序遍历的顺序为：

- 先遍历根节点；
- 随后递归地遍历左子树；
- 最后递归地遍历右子树。

二叉树中序遍历的顺序为：

- 先递归地遍历左子树；
- 随后遍历根节点；
- 最后递归地遍历右子树。

在「递归」地遍历某个子树的过程中，我们也是将这颗子树看成一颗全新的树，按照上述的顺序进行遍历。挖掘「前序遍历」和「中序遍历」的性质，我们就可以得出本题的做法。

更多硬核内容就在公众号「yes 的练级攻略」

方法一：递归

思路

对于任意一颗树而言，前序遍历的形式总是

```
[ 根节点，[左子树的前序遍历结果]，[右子树的前序遍历结果] ]
```

即根节点总是前序遍历中的第一个节点。而中序遍历的形式总是

```
[ [左子树的中序遍历结果]，根节点，[右子树的中序遍历结果] ]
```

只要我们在中序遍历中**定位**到根节点，那么我们就可以分别知道左子树和右子树中的节点数目。由于同一颗子树的前序遍历和中序遍历的长度显然是相同的，因此我们就可以对应到前序遍历的结果中，对上述形式中的所有**左右括号**进行定位。

这样以来，我们就知道了左子树的前序遍历和中序遍历结果，以及右子树的前序遍历和中序遍历结果，我们就可以递归地对构造出左子树和右子树，再将这两颗子树接到根节点的左右位置。

细节

在中序遍历中对根节点进行定位时，一种简单的方法是直接扫描整个中序遍历的结果并找出根节点，但这样做的时间复杂度较高。我们可以考虑使用哈希映射（HashMap）来帮助我们快速地定位根节点。对于哈希映射中的每个键值对，键表示一个元素（节点的值），值表示其在中序遍历中的出现位置。在构造二叉树的过程之前，我们可以对中序遍历的列表进行一遍扫描，就可以构造出这个哈希映射。在此后构造二叉树的过程中，我们就只需要 $O(1)$ 的时间对根节点进行定位了。

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是树中的节点个数。
- 空间复杂度：$O(n)$，除去返回的答案需要的 $O(n)$ 空间之外，我们还需要使用 $O(n)$ 的空间存储哈希映射，以及 $O(h)$（其中 h 是树的高度）的空间表示递归时栈空间。这里 $h < n$，所以总空间复杂度为 $O(n)$。

方法 1：递归做法（3ms）

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private Map<Integer, Integer> indexMap;
```

```
    public TreeNode myBuildTree(int[] preorder, int[] inorder, int preorder_lef
t, int preorder_right, int inorder_left, int inorder_right) {
        if (preorder_left > preorder_right) return null;
        // 前序遍历中的第一个节点就是根节点
        int preorder_root = preorder_left;
        // 在中序遍历中定位根节点
        int inorder_root = indexMap.get(preorder[preorder_root]);
        // 先把根节点建立出来
        TreeNode root = new TreeNode(preorder[preorder_root]);
        // 得到左子树中的节点数目
        int size_left_subtree = inorder_root - inorder_left;
        // 递归地构造左子树，并连接到根节点
        // 先序遍历中「从 左边界+1 开始的 size_left_subtree」个元素就对应了中序遍历中
「从 左边界 开始到 根节点定位-1」的元素
        root.left = myBuildTree(preorder, inorder, preorder_left + 1, preorder_
left + size_left_subtree, inorder_left, inorder_root - 1);
        // 递归地构造右子树，并连接到根节点
        // 先序遍历中「从 左边界+1+左子树节点数目 开始到 右边界」的元素就对应了中序遍
历中「从 根节点定位+1 到 右边界」的元素
        root.right = myBuildTree(preorder, inorder, preorder_left + size_left_s
ubtree + 1, preorder_right, inorder_root + 1, inorder_right);
        return root;
    }
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int n = preorder.length;
        // 构造哈希映射，帮助我们快速定位根节点
        indexMap = new HashMap<Integer, Integer>();
        for (int i = 0; i < n; i++) {
            indexMap.put(inorder[i], i);
        }
        return myBuildTree(preorder, inorder, 0, n - 1, 0, n - 1);
    }
}
```

https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/solution/ding-wei-chu-gen-jie-dian-de-wei-zhi-hua-fen-zuo-y/

方法 2：最优解：

贴出国外 StefanPochmann 大佬的解答

- Simple O(n) without map
- 他没有每次都在 inorder 数组中找根节点，也没有用 哈希表 存 inorder 每一项。

思路

- 变量 pre 保存当前要构造的树的 root
- 变量 in 保存 inorder 数组中可以成为 root 的数字们的开头那个
- 对于当前要构造的树，有一个停止点 stop ，inorder 数组中第 in 项到第 stop 项是要构造的树的节点值们
- 每次递归调用，都会确定出一个停止点，它告诉了子调用在哪里停止，把自己的根节点值作为左子树调用的停止点，自己的（父调用给下来的）停止点作为右子树的停止点

```
class Solution {
    private int in = 0;
    private int pre = 0;
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return build(preorder, inorder, Integer.MIN_VALUE);
    }
    private TreeNode build(int[] preorder, int[] inorder, int stop) {
        if (pre == preorder.length) return null; // pre 走到 preorder 末尾
        if (inorder[in] == stop) { // in 指针走到了停止点
            in++; // stop 点废弃了，in 推进一位
            return null;
        }
        TreeNode node = new TreeNode(preorder[pre++]);
        node.left = build(preorder, inorder, node.val); // 左子树的停止点是当前的根节点的值
        node.right = build(preorder, inorder, stop); // 右子树的停止点是当前树的停止点
        return node;
    }
}
```

参考：https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/solution/ding-wei-chu-gen-jie-dian-de-wei-zhi-hua-fen-zuo-y/

如果有重复元素，可以用方法 2，完整版代码如下：

```
import java.util.*;
class TreeNode{
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x){
        this.val = x;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

```
public class Main {
    //重建树，含有重复元素
    static private int in = 0;
    static private int pre = 0;
    public static TreeNode buildTree(int[] preorder, int[] inorder) {
        return build(preorder, inorder, Integer.MIN_VALUE);
    }
    private static TreeNode build(int[] preorder, int[] inorder, int stop) {
        if (pre == preorder.length) return null; // pre 走到 preorder 末尾
        if (inorder[in] == stop) { // in 指针走到了停止点
            in++; // stop 点废弃了，in 推进一位
            return null;
        }
        TreeNode node = new TreeNode(preorder[pre++]);
        node.left = build(preorder, inorder, node.val); // 左子树的停止点是当前的根节点的
值
        node.right = build(preorder, inorder, stop); // 右子树的停止点是当前树的停止点
        return node;
    }
    public static void main(String[] args){
        int[] preorder = {1,1,2,3,1};
        int[] inorder = {2,1,3,1,1};
        TreeNode root = buildTree(preorder, inorder);
        List<Integer> list2 = DLR_iteraytion(root);
        String res2 = printList(list2);
        System.out.println("前序遍历-迭代法结果：" + res2);
        List<Integer> list4 = LDR_iteraytion(root);
        String res4 = printList(list4);
        System.out.println("中序遍历-迭代法结果：" + res4);
        List<List<Integer>> lev = levelOrder(root);
        for (int i = 0; i < lev.size(); i++) {
            List<Integer> tmp = lev.get(i);
            String res = printList(tmp);
            System.out.println(res);
        }
    }
    //前序遍历-迭代
    public static List<Integer> DLR_iteraytion(TreeNode before){
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while (!stack.isEmpty() || before != null) {
            while (before != null) {
                res.add(before.val);
                stack.push(before);
```

```
                before = before.left;
            }
            if (!stack.isEmpty()) {
                before = stack.pop();
                before = before.right;
            }
        }
        return res;
    }
    //中序遍历-迭代
    public static List<Integer> LDR_iteraytion(TreeNode before){
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while (!stack.isEmpty() || before != null) {
            while (before != null) {
                stack.push(before);
                before = before.left;
            }
            if (!stack.isEmpty()) {
                before = stack.pop();
                res.add(before.val);
                before = before.right;
            }
        }
        return res;
    }
    //打印输出树
    public static String printList (List<Integer> list) {
        String res = "";
        int[] arr = list.stream().mapToInt(Integer::valueOf).toArray();
        for (int i = 0; i < arr.length; i++) {
            if (i != arr.length - 1) {
                res += arr[i];
                res += " ";
            }   else {
                res += arr[i];
            }
        }
        return res;
    }
    //层序遍历
    public static List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
```

```
            if(root != null) queue.add(root);
            while(!queue.isEmpty()) {
                LinkedList<Integer> tmp = new LinkedList<>();
                for(int i = queue.size(); i > 0; i--) {
                    TreeNode node = queue.poll();
                    if(res.size() % 2 == 0) tmp.addLast(node.val); // 偶数层 -> 队列头部
                    else tmp.addFirst(node.val); // 奇数层 -> 队列尾部
                    if(node.left != null) queue.add(node.left);
                    if(node.right != null) queue.add(node.right);
                }
                res.add(tmp);
            }
            return res;
        }
}
```

61、从中序与后序遍历序列构造二叉树（106）

106. 从中序与后序遍历序列构造二叉树

难度 中等 264 收藏 分享 切换为英文 关注 反馈

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:
你可以假设树中没有重复的元素。

例如，给出

```
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
```

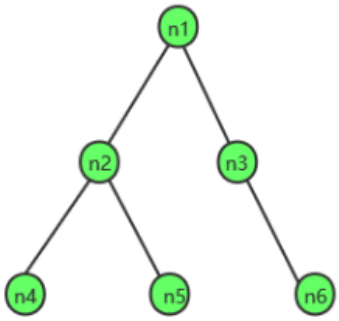
返回如下的二叉树：

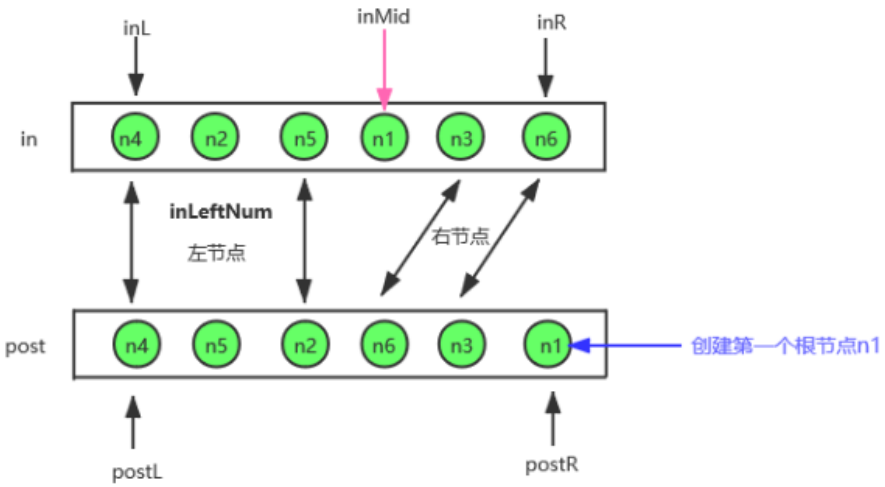
```
    3
   / \
  9  20
    /  \
   15   7
```

https://leetcode-cn.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/solution/106-zhong-xu-hou-xu-by-wulin-v/

变量说明：

- in 中序遍历的数组
- post 后序遍历的数组
- inL 中序遍历的第一个节点
- inR 中序遍历的最后一个节点
- postL 后序遍历的第一个节点
- postR 后序遍历的最后一个节点






```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
    class Solution{
    Map<Integer,Integer> map = new HashMap<>();
```

```
        public TreeNode buildTree(int[] inorder, int[] postorder) {
        //为了和105、889 题目结构一致都利用HashMap 方便查找节点位置（索引）
        //我们利用中序来区分左右子数
        for(int i=0;i<inorder.length;i++)
            map.put(inorder[i],i);
        return helper(inorder,postorder,0,inorder.length-1,0,postorder.length-1
);
    }
    private TreeNode helper(int[] in, int[] post, int inL, int inR, int postL,
int postR){
        if (inL>inR)  //说明数组无效
            return null;

        TreeNode root=new TreeNode(post[postR]);//利用后序的最后一个节点创建新的树
的根节点
        int postRootVal=post[postR];     //后序遍历的最后一个节点post_last_node
        int inMid=map.get(postRootVal); //后序遍历节点post_last_node 在中序遍历中
的位置（利用map 直接以O(1)的复杂度取出索引）
        int inLeftNum=inMid-inL;         //中序遍历的左子树的个数
        /*
            中序左子树的范围说明:
                从中序第一个节点开始，到后序遍历的最后一个节点在中序遍历位置的前一个节
点结束 inL,inMid-1
            中序右子树的范围说明:
                从后序遍历的最后一个节点在中序遍历位置的后一个节点开始，到中序的最后一
个位置结束 inMid+1,inR
            后序左子树的范围说明:
                从后序的第一个的节点开始，到第一个节点+左子树的个数-1 结
束 postL,postL+inLeftNum-1
            后序右子树的范围说明:
                从后序的第一个节点+左子树的个数开始，到到后序的最后一个位置-1 结
束 postL+inLeftNum,postR-1

            **注意**中序可以确定左子树的个数，后序遍历左子树的个数应该和中序遍历的结果
相同，
                所以是从后序遍历的第一个加上左子树个数，再减去第一个
            **注意**后序的右节点是从去除左节点开始的，所以是从后序的第一个节点+左子树
的个数开始
            **注意**千万不要被后序遍历的顺序给骗了
            利用后序遍历的顺序（右 左 根)做题（你会被坑的连你爸都不认识你。。信不信由
你)
            后序遍历顺序是  右 左 根是对的。但是，但是，她也是**从最左边子树的最左边的
子节点开始遍历**
```

```
        (请认真思考，这意味着什么)，这**意味着后序遍历的开始打印出来的都是树的左边的值**。

        中序左子树的范围[inL,inMid-1] 后序左子树的范围[postL,postL+inLeftNum-1]
        中序右子树的范围[inMid+1,inR] 后序右子树的范围[postL+inLeftNum,postR-1]
        */

        root.left=helper(in,post,inL,inMid-1,postL,postL+inLeftNum-1);//递归构建左子树
        root.right=helper(in,post,inMid+1,inR,postL+inLeftNum,postR-1);//递归构建右子树
        return root;
    }
}
```

62、根据前序和后序遍历构造二叉树（889）

889. 根据前序和后序遍历构造二叉树

难度 中等 102 收藏 分享 切换为英文 关注 反馈

返回与给定的前序和后序遍历匹配的任何二叉树。

`pre` 和 `post` 遍历中的值是不同的正整数。

示例：

输入： `pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]`
输出： `[1,2,3,4,5,6,7]`

提示：

- `1 <= pre.length == post.length <= 30`
- `pre[]` 和 `post[]` 都是 `1, 2, ..., pre.length` 的排列
- 每个输入保证至少有一个答案。如果有多个答案，可以返回其中一个。

https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-postorder-traversal/solution/889qian-xu-hou-xu-zhu-shi-xiang-jie-by-wulin-v/

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
```

```
 *     TreeNode(int x) { val = x; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
import java.util.ArrayList;
import java.util.List;

public class Solution {
        Map<Integer,Integer> map = new HashMap<>();
        public TreeNode constructFromPrePost(int[] pre, int[] post) {
            //为了和105、106 题目结构一致都利用HashMap 方便查找节点位置（索引）
            //利用后序将树分成左右子树
            for(int i=0;i<post.length;i++)
                map.put(post[i],i);
            return helper(pre,post,0,pre.length-1,0,post.length-1);
        }
        private TreeNode helper(int[] pre, int[] post, int preL, int preR, int
postL, int postR){
            if (preL>preR)
                return null;
            if(preL==preR)   //要提前结束，不然后面pre[preL+1]数组会越界，（注意105
题、106 题）
                return new TreeNode(pre[preL]);

            TreeNode head=new TreeNode(pre[preL]);
            int preRootVal=pre[preL+1]; //前序遍历的第二个节点就是左子树的开始节点
pre_second_node
            int postMid=map.get(preRootVal); //后序遍历节点pre_second_node 出现位
置（在数组post 中的索引）
            int postLeftNum=postMid-postL; //后序遍历数组的左子树的节点的个数
            /*
                前序左子树的范围说明:
                    从前序第一个节点是根节点所以我们从第二个开始，到第二个节点+后序确
定的左子树个数结束 preL+1,preL+1+postLeftNum
                前序右子树的范围说明:
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                从前序第二个节点+后序确定的左子树个数+1 开始，到前序的最后一个位置结束 preL+1+postLeftNum+1,preR
                后序左子树的范围说明:
                从后序第一个节点开始，到前序第二个的节点在后序中出现的位置结束 postL,postMid
                后序右子树的范围说明:
                从前序第二个的节点在后序中出现的位置+1 开始，到后序的最后一个位置-1 结束 postMid+1,postR-1

                前序左子树的范围[preL+1 , preL+1+postLeftNum] 后序左子树的范围[postL , postMid]
                前序右子树的范围[preL+1+postLeftNum+1 , preR] 后序右子树的范围[postMid+1 , postR-1]
            */
            head.left=helper(pre,post,preL+1,preL+1+postLeftNum,postL,postMid);//递归构建左子树
            head.right=helper(pre,post,preL+1+postLeftNum+1,preR,postMid+1,postR-1);//递归构建右子树
            return head;
        }
}
```

63、二叉树展开为链表（114）

公众号：
yes的练级攻略

114. 二叉树展开为链表

难度 中等 382 收藏 分享 切换为英文 关注 反馈

给定一个二叉树，原地将它展开为一个单链表。

例如，给定二叉树

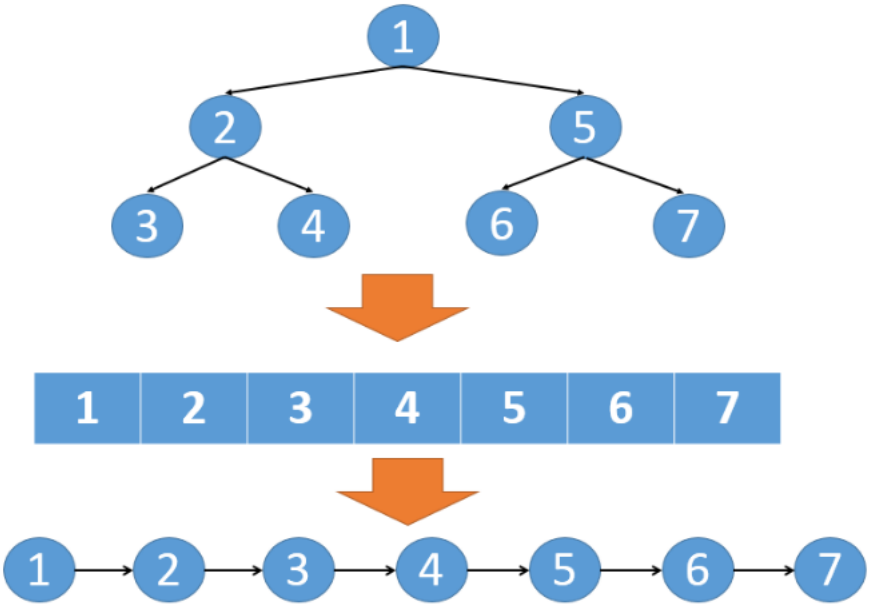
```
    1
   / \
  2   5
 / \   \
3   4   6
```

将其展开为：

```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

https://leetcode-cn.com/problems/flatten-binary-tree-to-linked-list/solution/dong-hua-yan-shi-si-chong-jie-fa-114-er-cha-shu-zh/

展开后的链表是 1->2->3->4->5->6 ，这个顺序就是二叉树前序遍历的顺序，我们用前序遍历的方式遍历这棵树，将结果保存到一个数组中，再把这个数组中的每个元素前后串联起来就可以了。



时间复杂度:O(N)
空间复杂度:O(N)

解法 1：利用前序遍历的递归（1ms，不是最佳）

```
/* public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    LinkedList<TreeNode> list = new LinkedList<>();
    public void flatten(TreeNode root) {
        if (root == null) return;
        addNode(root);
        TreeNode head = list.removeFirst();
        head.left = null;
        while(list.size() > 0) {
            TreeNode tmp = list.removeFirst();
            tmp.left = null;
            head.right = tmp;
```

```
            head = head.right;
        }
    }
    public void addNode(TreeNode cur) {
        if (cur == null) return;
        list.add(cur);
        addNode(cur.left);
        addNode(cur.right);
    }
}
```

解法3

我们采用后序遍历的方式，也就是 左节点-右节点-打印根节点 这个顺序遍历二叉树。
当遍历到根节点后，我们对根节点的 左右子树 做一些调整。

```
  1
 / \
2   3
```

将 右节点 挂到 左节点的最右边

```
  1
 /
2
 \
  3
```

再将整个 左子树 挂到根节点的 右边 ，这样就可以将整棵树变成链表结构了。

```
1
 \
  2
   \
    3
```

解法 2：采用后序遍历

```
class Solution {
    public void flatten(TreeNode root) {
        helper(root);
    }

    void helper(TreeNode root) {
        if(root==null) {
            return;
        }
        helper(root.left);
        helper(root.right);
        //将右子树挂到 左子树的最右边
        //再将整个左子树挂到根节点的右边
        if(root.left!=null) {
            TreeNode pre = root.left;
```

```
            while(pre.right!=null) {
                pre = pre.right;
            }
            pre.right = root.right;
            root.right = root.left;
            root.left = null;
        }
    }
}
```

64、BiNode（面试题 17.12）

面试题 17.12. BiNode

难度 简单 42 收藏 分享 切换为英文 关注 反馈

二叉树数据结构 TreeNode 可用来表示单向链表（其中 left 置空，right 为下一个链表节点）。实现一个方法，把二叉搜索树转换为单向链表，要求依然符合二叉搜索树的性质，转换操作应是原址的，也就是在原始的二叉搜索树上直接修改。

返回转换后的单向链表的头节点。

注意：本题相对原题稍作改动

示例：

```
输入：[4,2,5,1,3,null,6,0]
输出：[0,null,1,null,2,null,3,null,4,null,5,null,6]
```

提示：

- 节点数量不会超过 100000。

https://leetcode-cn.com/problems/binode-lcci/solution/binode100jian-dan-yi-dong-by-zui-weng-jiu-xian/

解题思路

1. 要求是原址修改，即**要直接利用这些已经存在的节点空间**，**修改**这些节点的**指向**来得到单向链表。
2. 要求单向链表依然符合二叉搜索树的性质，说是单向链表，其实还是一棵特殊的二叉搜索树。
3. 众所周知，对二叉搜索树采用**中序遍历**就能得到一个升序序列。 那么如果我们在遍历过程中，修改每一个根节点的左右指向，不就实现了原址修改了吗。

算法分析

设n为二叉搜索树中节点的个数。

1. 中序遍历所有节点仅访问一次，所以时间复杂度为$O(n)$.
2. 递归使用辅助栈空间$O(n)$，几个临时变量$O(1)$，因此总的空间复杂度为$O(n)$.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```
class Solution {
    TreeNode head = new TreeNode(-1);   // 为了返回单向链表的头节点而多设的一个节点
    TreeNode pre = null;                // 指向当前节点的前一个节点
    public TreeNode convertBiNode(TreeNode root) {
        dfs(root);
        return head.right;
    }
    public void dfs(TreeNode root) {
        if (root == null) return;
        dfs(root.left);
        if (pre == null) { // 第一个节点
            pre = root; // 记录第一个节点
            head.right = root; // 记录它，它将作为单链表的表头
        } else {
            pre.right = root; // 前一个节点的右指针指向当前节点
            pre = root; // 更新 per 指向
        }
        root.left = null; // 当前节点的左指针设为 null
        dfs(root.right);
    }
}
```

65、将有序数组转换为二叉搜索树（108）【最小高度树（面试题 04.02）】

108. 将有序数组转换为二叉搜索树

难度 简单 554 收藏 分享 切换为英文 关注 反馈

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。

示例:

给定有序数组: [-10,-3,0,5,9],

一个可能的答案是：[0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树：

```
      0
     / \
   -3   9
   /   /
 -10  5
```

更多硬核内容就在公众号「yes 的练级攻略」

面试题 04.02. 最小高度树

难度 简单 50 收藏 分享 切换为英文 关注 反馈

给定一个有序整数数组，元素各不相同且按升序排列，编写一个算法，创建一棵高度最小的二叉搜索树。

示例:

给定有序数组: [-10,-3,0,5,9],

一个可能的答案是: [0,-3,9,-10,null,5]，它可以表示下面这个高度平衡二叉搜索树:

```
          0
         / \
       -3   9
       /   /
     -10  5
```

https://leetcode-cn.com/problems/convert-sorted-array-to-binary-search-tree/solution/jian-dan-di-gui-bi-xu-miao-dong-by-sweetiee/

一、题目分析

题意：根据升序数组，恢复一棵高度平衡的BST🌲。

分析：BST的中序遍历是升序的，因此本题等同于根据中序遍历的序列恢复二叉搜索树。因此我们可以以升序序列中的任一个元素作为根节点，以该元素左边的升序序列构建左子树，以该元素右边的升序序列构建右子树，这样得到的树就是一棵二叉搜索树啦～ 又因为本题要求高度平衡，因此我们需要选择升序序列的中间元素作为根节点奥～

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return dfs(nums, 0, nums.length - 1);
    }

    private TreeNode dfs(int[] nums, int left, int right) {
        if (left > right) {
            return null;
        }
        // 以升序数组的中间元素作为根节点 root。
        int mid = left + (right - left) / 2;
        TreeNode root = new TreeNode(nums[mid]);
```

```
            // 递归的构建 root 的左子树与右子树。
            root.left = dfs(nums, left, mid - 1);
            root.right = dfs(nums, mid + 1, right);
            return root;
        }
}
```

66、有序链表转换二叉搜索树（109）

109. 有序链表转换二叉搜索树

难度 中等 313 收藏 分享 切换为英文 关注 反馈

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树*每个节点*的左右两个子树的高度差的绝对值不超过 1。

示例:

```
给定的有序链表： [-10, -3, 0, 5, 9],

一个可能的答案是：[0, -3, 9, -10, null, 5], 它可以表示下面这个高度平衡二叉搜索树：

      0
     / \
   -3   9
   /   /
 -10  5
```

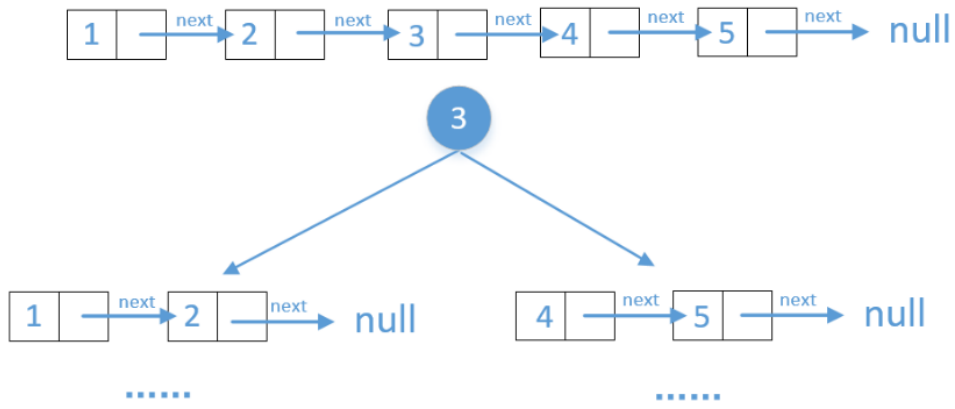
https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/solution/kuai-man-zhi-zhen-jie-jue-ji-bai-liao-100de-yong-h/

1，快慢指针

题中说了，是按照升序排列的单链表，我们只需要找到他的**中间结点，让他成为树的根节点，中间结点前面的就是根节点左子树的所有节点，中间节点后面的就是根节点右子树的所有节点**，然后使用递归的方式再分别对左右子树进行相同的操作.....

这里就以链表 1→2→3→4→5 为例来画个图看一下

☆ 收藏



复杂度分析

- 时间复杂度：$O(n \log n)$，其中 n 是链表的长度。

  设长度为 n 的链表构造二叉搜索树的时间为 $T(n)$，递推式为 $T(n) = 2 \cdot T(n/2) + O(n)$，根据主定理，$T(n) = O(n \log n)$。

- 空间复杂度：$O(\log n)$，这里只计算除了返回答案之外的空间。平衡二叉树的高度为 $O(\log n)$，即为递归过程中栈的最大深度，也就是需要的空间。

方法 1：快慢指针找中点（时间复杂度 o(nlogn)）

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = n
ext; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public TreeNode sortedListToBST(ListNode head) {
        //边界条件的判断
        if (head == null)
            return null;
        if (head.next == null)
            return new TreeNode(head.val);
        //这里通过快慢指针找到链表的中间结点 slow，pre 就是中间
        //结点 slow 的前一个结点
        ListNode slow = head, fast = head, pre = null;
```

```
        while (fast != null && fast.next != null) {
            pre = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        //链表断开为两部分，一部分是 node 的左子节点，一部分是 node
        //的右子节点
        pre.next = null;
        //node 就是当前节点
        TreeNode node = new TreeNode(slow.val);
        //从 head 节点到 pre 节点是 node 左子树的节点
        node.left = sortedListToBST(head);
        //从 slow.next 到链表的末尾是 node 的右子树的结点
        node.right = sortedListToBST(slow.next);
        return node;
    }
}
```

https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/solution/you-xu-lian-biao-zhuan-huan-er-cha-sou-suo-shu-1-3/

方法一的时间复杂度的瓶颈在于寻找中位数节点。由于**构造出的二叉搜索树的中序遍历结果就是链表本身**，因此我们可以将分治和中序遍历结合起来，减少时间复杂度。

具体地，设当前链表的左端点**编号**为 $left$，右端点**编号**为 $right$，包含关系为「双闭」，即 $left$ 和 $right$ 均包含在链表中。链表节点的编号为 $[0, n)$。中序遍历的顺序是「左子树 - 根节点 - 右子树」，那么在分治的过程中，我们不用急着找出链表的中位数节点，而是使用一个占位节点，等到中序遍历到该节点时，再填充它的值。

我们可以通过计算编号范围来进行中序遍历：

- 中位数节点对应的编号为 $mid = (left + right + 1)/2$；

  > 根据方法一中对于中位数的定义，编号为 $(left + right)/2$ 的节点同样也可以是中位数节点。

- 左右子树对应的编号范围分别为 $[left, mid - 1]$ 和 $[mid + 1, right]$。

如果 $left > right$，那么遍历到的位置对应着一个空节点，否则对应着二叉搜索树中的一个节点。

这样一来，我们其实已经知道了这棵二叉搜索树的结构，并且题目给定了它的中序遍历结果，那么我们只要对其进行中序遍历，就可以还原出整棵二叉搜索树了。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(n)$，其中 n 是链表的长度。

  设长度为 n 的链表构造二叉搜索树的时间为 $T(n)$，递推式为 $T(n) = 2 \cdot T(n/2) + O(1)$，根据主定理，$T(n) = O(n)$。

- 空间复杂度：$O(\log n)$，这里只计算除了返回答案之外的空间。平衡二叉树的高度为 $O(\log n)$，即为递归过程中栈的最大深度，也就是需要的空间。

正确性证明

我们需要证明的是：对于任意的有序链表，用「前言」部分的构造方法得到的二叉搜索树一定是平衡的。显然，如果二叉搜索树的左右子树都是平衡的，并且它们的高度差不超过 1，那么该二叉搜索树就是平衡的。

我们使用数学归纳法，设链表的长度为 n，对应的二叉搜索树的高度为 $H(n)$：

> 这里规定只包含根节点的二叉搜索树的高度为 1。

- 当 $n = 1, 2$ 时，对应的二叉搜索树都是平衡的，并且 $H(1) = 1$，$H(2) = 2$；
- 当 $n = 3$ 时，对应的二叉搜索树包含一个根节点和左右子树各一个节点，它是平衡的，并且 $H(3) = 2$；
- 假设当 $n < n_0$ 时，对应的二叉搜索树是平衡的，并且对于任意的 $1 \leq n < n_0 - 1$，都有 $H(n+1) - H(n) \leq 1$，那么当 $n = n_0$ 时：
  - 如果 n 为奇数，设 $n = 2k + 3$，那么二叉搜索树的左右子树的大小均为 $k + 1$，高度相等，因此二叉搜索树是平衡的，并且有 $H(n) = H(k+1) + 1$。而 $n - 1 = 2k + 2$，对应的二叉搜索树的左右子树的大小为 k 和 $k + 1$，即 $H(n-1) = H(k+1) + 1$，因此 $H(n) = H(n-1)$；
  - 如果 n 为偶数，设 $n = 2k + 2$，那么二叉搜索树的左右子树的大小为 k 和 $k + 1$，根据假设，高度差 $H(k+1) - H(k) \leq 1$，因此二叉搜索树是平衡的，并且有 $H(n) = H(k+1) + 1$。而 $n - 1 = 2k + 1$，对应的二叉搜索树的左右子树的大小均为 k，即 $H(n-1) = H(k) + 1$，因此 $H(n) - H(n-1) = H(k+1) - H(k) \leq 1$。

这样我们就证明了任意长度的链表对应的二叉搜索树是平衡的。

方法 2：分治加中序（时间复杂度 o(n)）

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = n
ext; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
```

```
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    ListNode globalHead;
    public TreeNode sortedListToBST(ListNode head) {
        globalHead = head;
        int len = getLength(head);
        return buildTree(0, len - 1);
    }
    public int getLength(ListNode head) {
        int res = 0;
        while (head != null) {
            ++res;
            head = head.next;
        }
        return res;
    }
    public TreeNode buildTree(int left, int right){
        if (left > right) return null;
        int mid = (left + right + 1) / 2;
        TreeNode root = new TreeNode();
        root.left = buildTree(left, mid - 1);
        root.val = globalHead.val;
        globalHead = globalHead.next;
        root.right = buildTree(mid + 1, right);
        return root;
    }
}
```

67、相同的树（100）

100. 相同的树

难度 简单 431 收藏 分享 切换为英文 关注 反馈

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1:

```
输入:       1         1
          / \       / \
         2   3     2   3

        [1,2,3],   [1,2,3]

输出: true
```

示例 2:

```
输入:      1          1
          /            \
         2              2

        [1,2],     [1,null,2]

输出: false
```

示例 3:

```
输入:       1         1
          / \       / \
         2   1     1   2

        [1,2,1],   [1,1,2]

输出: false
```

通过次数 122,230 | 提交次数 205,138

https://leetcode-cn.com/problems/same-tree/solution/100java-di-gui-die-dai-xiang-jie-by-ustcyyw/

思路分析：

- 两棵树相同意味着，树的结构一模一样且对应位置的结点的值相等。所以，根节点值相同，且左右子树相同才能判断一颗树相同。这就是递归的关系。
- 递归结束的条件，当两颗树均是 `null` 时他们相等，当两颗树其中之一为 `null` 另外一颗非 `null` 时他们不相等。
- 其它情况下，先判断当前节点的值是否相等，如果相等再判断当前节点的左右子树是否均相同。
- 时间复杂度：$O(n)$，其中 n 是树的结点数，因为每个结点都访问一次。空间复杂度：最优情况（完全平衡二叉树）时为 $O(log(n))$，最坏情况下（完全不平衡二叉树）时为 $O(n)$，用于维护递归栈。

方法 1：递归

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;
        if (p == null || q == null) return false;
        if (p.val != q.val) return false;
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

思路分析：

- 树的相同，意味着结点位置一一对应，且对应位置的结点的值是相等的。所以只需要遍历每个结点，一旦有结点值不同就可以返回 `false`。
- 使用DFS或者BFS均可以，只要每次一一对应地将结点放入队列（栈），再一一对应地取出比较。当树的结构不相同时，两颗树每次都入队（栈）相同数量的结点，取出相同数量的结点，必然导致值不同。当树结构相同，但值不同时，也能比较值得到结果。所以这个思路正确。
- 迭代和递归等价，所以时间复杂度与空间复杂度同方法一。

代码解释：

- 4~5两行，根结点为 `null` 也可以放入队列中，并且这种情况会在while循环的9-10行得到处理，如果均为 `null` 直接循环结束，由17行返回结果为 `true`。有其一为 `null` 即答案为 `false`，第10行直接返回。
- 11行，比较当前结点的值。
- 7-8行及12-15行，即为每次一一对应地将结点放入队列（栈），再一一对应地取出比较。

方法 2：迭代

```java
class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        Queue<TreeNode> queue1 = new LinkedList<>();
        Queue<TreeNode> queue2 = new LinkedList<>();
        queue1.add(q);
        queue2.add(p);
        while (!queue1.isEmpty()){
            TreeNode tempQ = queue1.remove();
```

```
            TreeNode tempP = queue2.remove();
            if (tempQ == null && tempP == null) continue;
            if (tempQ == null || tempP == null) return false;
            if(tempP.val != tempQ.val) return false;
            queue1.add(tempQ.left);
            queue1.add(tempQ.right);
            queue2.add(tempP.left);
            queue2.add(tempP.right);
        }
        return true;
    }
}
```

68、合并二叉树（617）

617. 合并二叉树

难度 简单 399 收藏 分享 切换为英文 关注 反馈

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则**不为** NULL 的节点将直接作为新二叉树的节点。

示例 1:

输入:

```
Tree 1                     Tree 2
          1                         2
         / \                       / \
        3   2                     1   3
       /                           \   \
      5                             4   7
```

输出:

合并后的树:

```
         3
        / \
       4   5
      / \   \
     5   4   7
```

注意: 合并必须从两个树的根节点开始。

https://leetcode-cn.com/problems/merge-two-binary-trees/solution/he-bing-er-cha-shu-by-leetcode/

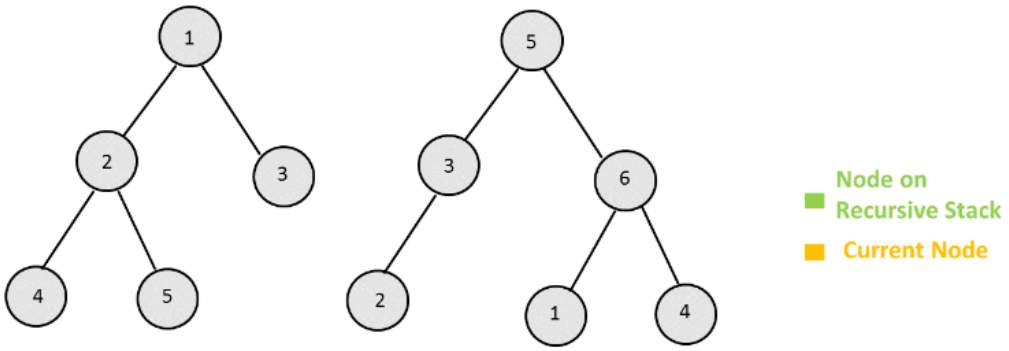
递归要注意输入输出

更多硬核内容就在公众号「yes 的练级攻略」

方法一：递归

我们可以对这两棵树同时进行前序遍历，并将对应的节点进行合并。在遍历时，如果两棵树的当前节点均不为空，我们就将它们的值进行相加，并对它们的左孩子和右孩子进行递归合并；如果其中有一棵树为空，那么我们返回另一颗树作为结果；如果两棵树均为空，此时返回任意一棵树均可（因为都是空）。

☆ 收藏



复杂度分析

- 时间复杂度：$O(N)$，其中 N 是两棵树中节点个数的较小值。
- 空间复杂度：$O(N)$，在最坏情况下，会递归 N 层，需要 $O(N)$ 的栈空间。

方法 1：递归

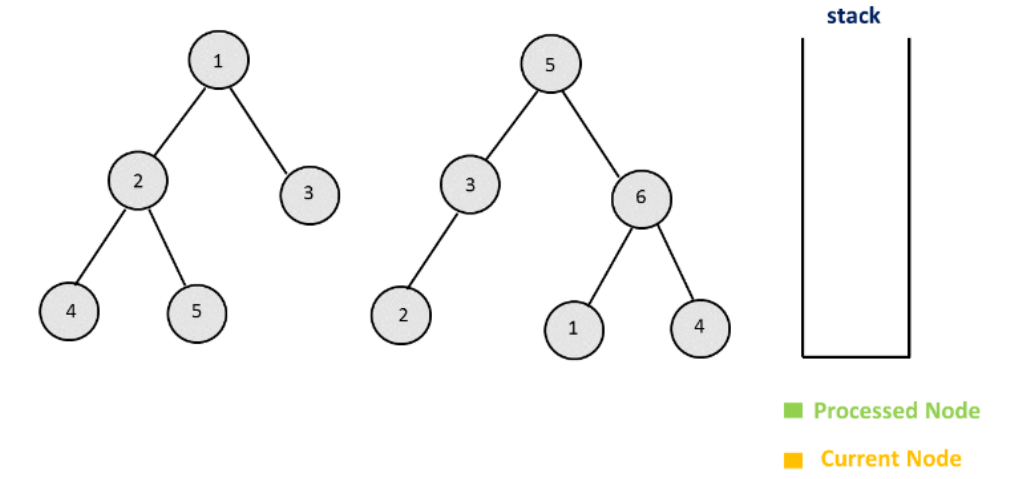
```
class Solution {
    public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
        TreeNode res = t1;
        if (t1 == null && t2 == null) return res;
        if (t1 == null || t2 == null) {
            return res = t1==null ? t2 : t1;
        }
        res.val = t1.val + t2.val;
        res.left = mergeTrees (t1.left,t2.left);
        res.right = mergeTrees (t1.right,t2.right);
        return res;
    }
}
```

方法二：迭代

我们也可以用迭代的方法解决这个问题。

我们首先把两棵树的根节点入栈，栈中的每个元素都会存放两个根节点，并且栈顶的元素表示当前需要处理的节点。在迭代的每一步中，我们取出栈顶的元素并把它移出栈，并将它们的值相加。随后我们分别考虑这两个节点的左孩子和右孩子，如果两个节点都有左孩子，那么就将左孩子入栈；如果只有一个节点有左孩子，那么将其作为第一个节点的左孩子；如果都没有左孩子，那么不用做任何事情。对于右孩子同理。

最后我们返回第一棵树的根节点作为答案。



复杂度分析

- 时间复杂度：$O(N)$，其中 N 是两棵树中节点个数的较小值。
- 空间复杂度：$O(N)$，在最坏情况下，栈中会存放 N 个节点。

方法 2：迭代

```
public class Solution {
    public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {
        if (t1 == null)
            return t2;
        Stack < TreeNode[] > stack = new Stack < > ();
        stack.push(new TreeNode[] {t1, t2});
        while (!stack.isEmpty()) {
            TreeNode[] t = stack.pop();
            if (t[0] == null || t[1] == null) {
                continue;
            }
            t[0].val += t[1].val;
            if (t[0].left == null) {
                t[0].left = t[1].left;
```

```
            } else {
                stack.push(new TreeNode[] {t[0].left, t[1].left});
            }
            if (t[0].right == null) {
                t[0].right = t[1].right;
            } else {
                stack.push(new TreeNode[] {t[0].right, t[1].right});
            }
        }
        return t1;
    }
}
```

69、完全二叉树的节点个数（222）

222. 完全二叉树的节点个数

难度 中等 208 收藏 分享 切换为英文 关注 反馈

给出一个**完全二叉树**，求出该树的节点个数。

说明：

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~ 2^h 个节点。

示例：

输入：

```
    1
   / \
  2   3
 / \  /
4  5 6
```

输出： 6

https://leetcode-cn.com/problems/count-complete-tree-nodes/solution/ologn2de-zuo-fa-by-jin-ai-yi/

首先知道公式：高度为 h 的满二叉树的节点个数为 2^h-1

当前函数是求一个完全二叉树的节点个数的。

对于一个高度为 h 的完全二叉树，如果他的右子树高度为 h-1（到最后一层了），说明右子树是完全二叉树，左子树是满二叉树，那么左子树直接套公式，右子树递归调用当前函数；如果右子树高度为 h-2(没有到达最后一层),说明左子树是完全二叉树，右子树是满二叉树，左子树递归，右子树套公式。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int countNodes(TreeNode root) {
```

```
        if (root == null) return 0;
        if (root.left == null && root.right == null) return 1;
        int depth = CountDepth(root, 0); //先求出整个完全二叉树的最深深度
        int rightDepth = CountDepth(root.right, 1); //再求右子树的最深深度
        int res = 0;
        if (rightDepth == depth) { //左子树一定是满二叉树，且高度为(h-1);右子树是完全二叉树
            res += Math.pow(2, (depth - 1)) - 1; //左子树节点个数
            res += countNodes(root.right); //递归地去求右子树的节点个数
            res++;//当前的head
            return res;
        } else {//右子树一定是满二叉树，且高度为(h-2)，因为是从右子节点开始，且只到了倒数第二层；左子树是完全二叉树
            res += Math.pow(2, (depth - 2)) - 1; //右子树节点个数
            res += countNodes(root.left); //递归地去求左子树的节点个数
            res++;
            return res;
        }
    }
    //获得一棵完全二叉树的最深深度，一直往左走
    public int CountDepth(TreeNode root, int d){
        if (root == null) return d;
        return CountDepth(root.left, d + 1);
    }
}
```

70、计算右侧小于当前元素的个数（315）

315. 计算右侧小于当前元素的个数

难度 困难 286 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 *nums*，按要求返回一个新数组 *counts*。数组 *counts* 有该性质： `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例:

```
输入: [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1).
2 的右侧仅有 1 个更小的元素 (1).
6 的右侧有 1 个更小的元素 (1).
1 的右侧有 0 个更小的元素.
```

https://leetcode-cn.com/problems/count-of-smaller-numbers-after-self/solution/cong-you-wang-zuo-cha-ru-pai-xu-by-utopiahiker/

二叉搜索树

使用二叉搜索树也可以完成插入并统计的功能，从右往左构建二叉树。
递归实现：
当走到右节点时，（1）统计根节点和左节点的个数，（2）继续插入并统计右边是否还有较小值；
当走到左节点或者根节点时，（1）计数器加一，（2）继续插入并统计左边是否还有较小值。

遍历一遍就可以完成搜索，时间复杂度O(nlogn)。

```
class Solution {
    public List<Integer> countSmaller(int[] nums) {
        //初始化
        Integer[] res = new Integer[nums.length];
        Arrays.fill(res, 0);
        List<Integer> list = new ArrayList<>();
        //反向构造二叉树，统计右边的较小数
        TreeNode root = null;
        for (int i = nums.length - 1; i >= 0; i--){
            root = addAndCount(root, new TreeNode(nums[i]), res, i);
        }
        return Arrays.asList(res);
    }

    public TreeNode addAndCount(TreeNode root, TreeNode node, Integer[]
 res, int i){
        if(root == null){
            root = node;
            return root;
        }
        //根节点的左边保存不大于根节点的元素
        if(root.val >= node.val){
            //统计左节点的元素个数
            root.count++;
            root.left = addAndCount(root.left, node, res, i);
        }else{
            //走到右边获取该元素左边的个数（根节点 1 + 左节点 root.count）
            res[i] += 1 + root.count;
            //统计右边是否还有更小的元素
            root.right = addAndCount(root.right, node, res, i);
        }
        return root;
    }
}

class TreeNode{
```

```
    int val;
    int count;
    TreeNode left, right;

    public TreeNode(int val){
        this.val = val;
        this.count = 0;
        left = null;
        right = null;
    }
}
```

71、最大树（654）（lintcode126）【单调栈（数据量大）】【递归（数据量小）】

654. 最大二叉树

难度 中等 171 收藏 分享 切换为英文 关注 反馈

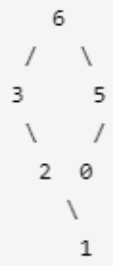
给定一个不含重复元素的整数数组。一个以此数组构建的最大二叉树定义如下：

1. 二叉树的根是数组中的最大元素。
2. 左子树是通过数组中最大值左边部分构造出的最大二叉树。
3. 右子树是通过数组中最大值右边部分构造出的最大二叉树。

通过给定的数组构建最大二叉树，并且输出这个树的根节点。

示例：

输入： [3,2,1,6,0,5]
输出： 返回下面这棵树的根节点：



提示：

1. 给定的数组的大小在 [1, 1000] 之间。

更多硬核内容就在公众号「yes 的练级攻略」

126. 最大树

给出一个没有重复的整数数组，在此数组上建立最大树的定义如下：

- 根是数组中最大的数
- 左子树和右子树元素分别是被父节点元素切分开的子数组中的最大值

利用给定的数组构造最大树。

样例

样例 1:

```
输入: [2, 5, 6, 0, 3, 1]
输出: {6,5,3,2,#,0,1}
解释:
此数组构造的最大树是:
    6
   / \
  5   3
 /   / \
2   0   1
```

样例 2:

```
输入: [6,4,20]
输出: {20,6,#,#,4}
解释:
     20
     /
    6
     \
      4
```

挑战

要求时间复杂度和空间复杂度均为O(n)

https://www.jiuzhang.com/solution/max-tree/#tag-highlight-lang-java

更多硬核内容就在公众号「yes 的练级攻略」

算法：单调栈

思路

- 单调栈可以在O(n)的时间里跑出以每个元素作为最大/小值的最左/右端点下标。
- 基本过程：
- 单调栈的操作：以一个递减的单调栈为例，若栈为空或者栈顶元素大于当前元素则压入，否则弹出栈内比当前元素小的所有元素。
- 以序列{7， 2， 5， 3， 11， 9}为例，模拟一下实现过程。

1. 第一步：栈为空，压入7。 此时栈内：`[7]`
2. 第二步：7比2大，压入2。 此时栈内：`[7, 2]`
3. 第三步：2比5小，弹出2，压入5。 此时栈内：`[7, 5]`
4. 第四步：5比3大，压入3。 此时栈内：`[7, 5, 3]`
5. 第五步：3、5、7 比11小，弹出3、弹出5、弹出7，压入11。 此时栈内：`[11]`
6. 第六步：11比9大，压入9 。 此时栈内：`[11, 9]`

- 在这个过程中可以发现，栈底元素是栈内元素中的最大值，而最后留在栈内的最底部的元素就是整个数组中的最大元素root。一个元素作为节点时的左儿子也就是左半边最大的值，是这个元素被压入栈时，栈弹出的最后一个元素；而右儿子也就是右半边最大的值，是这个元素的上面一个元素，因为递减的单调栈是从底部往顶部依次减小的。
- 具体实现：

从前往后依次遍历数组元素，对每个元素：

1. 如果栈内有元素且元素小于当前元素，那么就需要依次弹出栈内比当前元素小的元素。在这个过程中记录最后一个被弹出的元素，这个元素就是当前元素的左儿子。
2. 如果栈内有元素且栈顶元素大于当前元素，那么当前元素就是栈顶元素的右儿子。
3. 最后压入当前元素。

- 所以我们很容易在这个过程中得到答案，也就是{11,7,9,#,5,#,#,2,3}。

复杂度

- 空间复杂度O(n)
- 时间复杂度O(n)

方法 1：单调栈（不是很快，但是适用于数据比较大的时候）

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
     * @param A: Given an integer array with no duplicates.
     * @return: The root of max tree.
     */
    public TreeNode maxTree(int[] A) {
        int n = A.length;
        int[] stack = new int[n + 2];
        int cnt = -1;
        List<TreeNode> pNodes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            TreeNode tn = new TreeNode(A[i]);
```

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                pNodes.add(tn);
                if (cnt != -1 && A[stack[cnt]] < A[i]){
                    // 如果栈中有元素且元素小于当前元素，则弹出栈内元素
                    while (cnt > -1 &&    A[stack[cnt]] < A[i]){
                        cnt--;
                    }
                    // 弹出的最后一个元素就是当前元素的左儿子
                    pNodes.get(i).left = pNodes.get(stack[cnt+1]);
                }
                if (cnt != -1 && A[stack[cnt]] > A[i]){
                    // 如果栈内有元素且栈顶元素大于当前元素
                    // 那么当前元素是栈顶元素的右儿子
                    pNodes.get(stack[cnt]).right = pNodes.get(i);
                }
                // 压入当前元素
                stack[++cnt] = i;
            }
            return pNodes.get(stack[0]);
        }
}
```

禁止商用

https://leetcode-cn.com/problems/maximum-binary-tree/solution/zui-da-er-cha-shu-by-leetcode/

方法一：递归

本方法非常简单。创建方法 `construct(nums, l, r)`，用于找出在数组 $nums$ 中从 l 到 r 索引（不包含第 r 个位置）中最大二叉树的根节点。

算法步骤如下：

1. 首先调用 `construct(nums, 0, n)`，其中 n 是数组 $nums$ 的长度。
2. 在索引范围 $(l : r-1)$ 内找到最大值的索引，将 $nums[max_i]$ 作为根节点。
3. 调用 `construct(nums, l, max_i)` 创建根节点的左孩子。递归执行此操作，创建根节点的整个左子树。
4. 类似的，调用 `construct(nums, max_i + 1, r)` 创建根节点的右子树。
5. 方法 `construct(nums, 0, n)` 返回最大二叉树的根节点。

复杂度分析

- 时间复杂度：$O(n^2)$。方法 `construct` 一共被调用 n 次。每次递归寻找根节点时，需要遍历当前索引范围内所有元素找出最大值。一般情况下，每次遍历的复杂度为 $O(\log n)$，总复杂度为 $O(n \log n)$。最坏的情况下，数组 $nums$ 有序，总的复杂度为 $O(n^2)$。
- 空间复杂度：$O(n)$。递归调用深度为 n。平均情况下，长度为 n 的数组递归调用深度为 $O(\log n)$。

方法 2：递归（适用于数字比较小的时候）

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public TreeNode constructMaximumBinaryTree(int[] nums) {
        return construct(nums, 0, nums.length);
    }
    public TreeNode construct(int[] nums, int l, int r) {
        if (l == r)
            return null;
        int max_i = max(nums, l, r);
        TreeNode root = new TreeNode(nums[max_i]);
        root.left = construct(nums, l, max_i);
        root.right = construct(nums, max_i + 1, r);
        return root;
    }
    public int max(int[] nums, int l, int r) {
        int max_i = l;
        for (int i = l; i < r; i++) {
            if (nums[max_i] < nums[i])
                max_i = i;
        }
        return max_i;
    }
}
```

72、二叉搜索树的最近公共祖先（235）（剑指 68-1）

yes的练级攻略

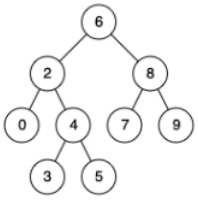
235. 二叉搜索树的最近公共祖先

难度 简单 346 收藏 分享 切换为英文 关注 反馈

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。

示例 2:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

方法 1：直接用下一题优化后的代码（7ms）
二叉搜索树是二叉树的一种

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, Tre
eNode q) {
        if (root == null || root == p || root == q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left == null) return right;
        if (right == null) return left;
        return root;
    }
```

}

https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-search-tree/solution/er-cha-sou-suo-shu-de-zui-jin-gong-gong-zu-xian--2/

算法

1. 从根节点开始遍历树
2. 如果节点 p 和节点 q 都在右子树上，那么以右孩子为根节点继续 1 的操作
3. 如果节点 p 和节点 q 都在左子树上，那么以左孩子为根节点继续 1 的操作
4. 如果条件 2 和条件 3 都不成立，这就意味着我们已经找到节 p 和节点 q 的 LCA 了

复杂度分析

- 时间复杂度：$O(N)$
  其中 N 为 BST 中节点的个数，在最坏的情况下我们可能需要访问 BST 中所有的节点。
- 空间复杂度：$O(N)$
  所需开辟的额外空间主要是递归栈产生的，之所以是 N 是因为 BST 的高度为 N。

方法 2：（递归）利用二叉搜索树的特性（6ms）

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root.val > p.val && root.val > q.val) return lowestCommonAncestor(root.left, p, q);
        if(root.val < p.val && root.val < q.val) return lowestCommonAncestor(root.right, p, q);
        return root;
    }

}
```

算法

这个方法跟方法一很接近。唯一的不同是，我们用迭代的方式替代了递归来遍历整棵树。由于我们不需要回溯来找到 LCA 节点，所以我们是完全可以不利用栈或者是递归的。实际上这个问题本身就是可以迭代的，我们只需要找到分割点就可以了。这个分割点就是能让节点 p 和节点 q 不能在同一颗子树上的那个节点，或者是节点 p 和节点 q 中的一个，这种情况下其中一个节点是另一个节点的父亲节点。

复杂度分析

- 时间复杂度：$O(N)$
  其中 N 为 BST 中节点的个数，在最坏的情况下我们可能需要遍历 BST 中所有的节点。
- 空间复杂度：$O(1)$

方法 3：（迭代）利用二叉搜索树的特性（6ms）

```
class Solution {
```

```
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        while (root != null) {
            if (p.val > root.val && q.val > root.val) {
                root = root.right;
            } else if (p.val < root.val && q.val < root.val) {
                root = root.left;
            } else {
                break;
            }
        }
        return root;
    }
}
```

73、二叉树的最近公共祖先（236）（剑指 68-2）【首个共同祖先（面试题 04.08）】

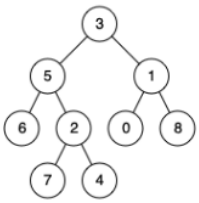
236. 二叉树的最近公共祖先

难度 中等 674 收藏 分享 切换为英文 关注 反馈

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

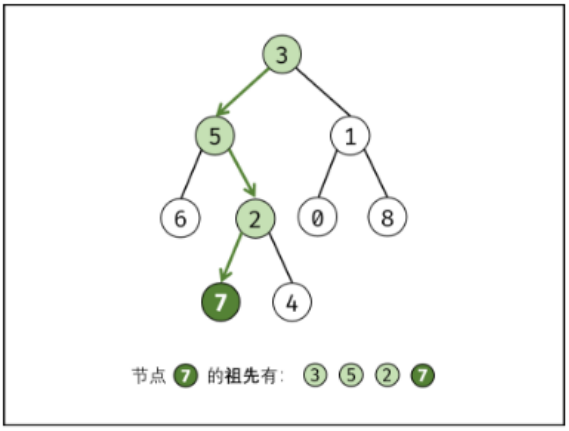
说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/236-er-cha-shu-de-zui-jin-gong-gong-zu-xian-hou-xu/

解题思路：

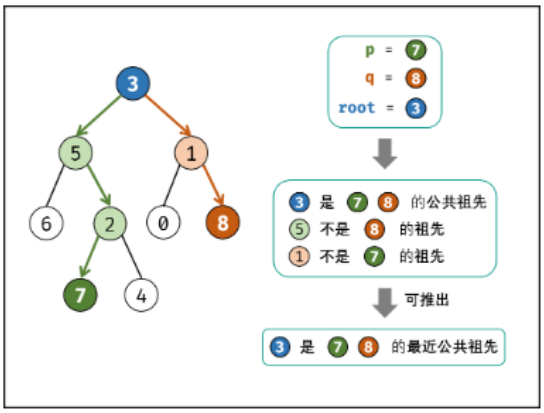
祖先的定义： 若节点 p 在节点 $root$ 的左（右）子树中，或 $p = root$，则称 $root$ 是 p 的祖先。



最近公共祖先的定义： 设节点 $root$ 为节点 p, q 的某公共祖先，若其左子节点 $root.left$ 和右子节点 $root.right$ 都不是 p, q 的公共祖先，则称 $root$ 是 “最近的公共祖先” 。

根据以上定义，若 $root$ 是 p, q 的 **最近公共祖先** ，则只可能为以下情况之一：

1. p 和 q 在 $root$ 的子树中，且分列 $root$ 的 **异侧**（即分别在左、右子树中）；
2. $p = root$ ，且 q 在 $root$ 的左或右子树中；
3. $q = root$ ，且 p 在 $root$ 的左或右子树中；



考虑通过递归对二叉树进行后序遍历，当遇到节点 p 或 q 时返回。从底至顶回溯，当节点 p, q 在节点 $root$ 的异侧时，节点 $root$ 即为最近公共祖先，则向上返回 $root$ 。

更多硬核内容就在公众号「yes 的练级攻略」

递归解析：

1. **终止条件：**
   1. 当越过叶节点，则直接返回 $null$ ；
   2. 当 $root$ 等于 p, q ，则直接返回 $root$ ；
2. **递推工作：**
   1. 开启递归左子节点，返回值记为 $left$ ；
   2. 开启递归右子节点，返回值记为 $right$ ；
3. **返回值：** 根据 $left$ 和 $right$ ，可展开为四种情况；
   1. 当 $left$ 和 $right$ **同时为空** ：说明 $root$ 的左 / 右子树中都不包含 p, q ，返回 $null$ ；
   2. 当 $left$ 和 $right$ **同时不为空** ：说明 p, q 分列在 $root$ 的 **异侧** （分别在 左 / 右子树），因此 $root$ 为最近公共祖先，返回 $root$ ；
   3. 当 $left$ **为空** ，$right$ **不为空** ：p, q 都不在 $root$ 的左子树中，直接返回 $right$ 。具体可分为两种情况：
      1. p, q 其中一个在 $root$ 的 **右子树** 中，此时 $right$ 指向 p（假设为 p ）；
      2. p, q 两节点都在 $root$ 的 **右子树** 中，此时的 $right$ 指向 **最近公共祖先节点** ；
   4. 当 $left$ **不为空** ， $right$ **为空** ：与情况 `3.` 同理；

复杂度分析：

- **时间复杂度 $O(N)$ ：** 其中 N 为二叉树节点数；最差情况下，需要递归遍历树的所有节点。
- **空间复杂度 $O(N)$ ：** 最差情况下，递归深度达到 N ，系统使用 $O(N)$ 大小的额外空间。

方法 1：后序遍历 DFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    //后序遍历
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, Tre
eNode q) {
        if(root == null || root == p || root == q) return root; //当越过
叶节点或者root 等于p、q
        //递归左子节点
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        //递归右子节点
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left == null && right == null) return null; // 两节点同时为空，
意味着左右子树都不包含p 和q
        if(left == null) return right; // 当左节点为空，说明p 和q 在右边
        if(right == null) return left; // 当右节点为空，说明p 和q 在左边
        return root; // 同时不为空，p 和q 的公共祖先是root
    }
```

```
}
```

方法 2：优化以后【因为 p、q 为不同节点且均存在于给定的二叉树中，删除都为空的判断】

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, Tre
eNode q) {
        if(root == null || root == p || root == q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left == null) return right;
        if(right == null) return left;
        return root;
    }
}
```

74、树节点的第 K 个祖先（1483）

1483. 树节点的第 K 个祖先

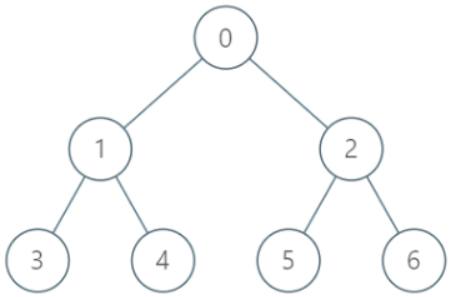
难度 困难　46　收藏　分享　切换为英文　关注　反馈

给你一棵树，树上有 `n` 个节点，按从 `0` 到 `n-1` 编号。树以父节点数组的形式给出，其中 `parent[i]` 是节点 `i` 的父节点。树的根节点是编号为 `0` 的节点。

请你设计并实现 `getKthAncestor(int node, int k)` 函数，函数返回节点 `node` 的第 `k` 个祖先节点。如果不存在这样的祖先节点，返回 `-1` 。

树节点的第 *k* 个祖先节点是从该节点到根节点路径上的第 `k` 个节点。

示例：



输入：
```
["TreeAncestor","getKthAncestor","getKthAncestor","getKthAncestor"]
[[7,[-1,0,0,1,1,2,2]],[3,1],[5,2],[6,3]]
```

输出：
```
[null,1,0,-1]
```

解释：
```
TreeAncestor treeAncestor = new TreeAncestor(7, [-1, 0, 0, 1, 1, 2, 2]);

treeAncestor.getKthAncestor(3, 1);  // 返回 1 ，它是 3 的父节点
treeAncestor.getKthAncestor(5, 2);  // 返回 0 ，它是 5 的祖父节点
treeAncestor.getKthAncestor(6, 3);  // 返回 -1 因为不存在满足要求的祖先节点
```

提示：

- `1 <= k <= n <= 5*10^4`
- `parent[0] == -1` 表示编号为 `0` 的节点是根节点。
- 对于所有的 `0 < i < n` ，`0 <= parent[i] < n` 总成立
- `0 <= node < n`
- 至多查询 `5*10^4` 次

https://leetcode-cn.com/problems/kth-ancestor-of-a-tree-node/solution/java-bei-zeng-fa-you-zhu-shi-by-gitkid/

dp[i][j] 表示节点i的第2^j代祖先。

因为第7代祖先，可以表示为$2^0 + 2^1 + 2^2$代祖先，同理，第k代祖先也可以这样表示。

所以有dp[i][j] = dp[dp[i][j-1]][j-1];

其他就是细节部分要注意。

```java
class TreeAncestor {
    int[][] dp;
    public TreeAncestor(int n, int[] parent) {
        // dp[i][j] 表示节点 i 的第 2^j 代祖先。
        // j = 0，就是父亲。
        dp = new int[n+5][20];
        // dp[0][0] = 0;
        for(int i = 1; i <= n; i++){
            // 为什么要+1？因为否则会出现-1 的索引，那样数组索引就会报错。
            // 注意 dp[i]存储的是第 i-1 的祖宗节点
            dp[i][0] = parent[i -1] + 1;
        }
        for(int j = 1; j < 20; j++){
            for(int i = 0; i <= n; i++){
                dp[i][j] = dp[dp[i][j-1]][j-1];
            }
        }
    }
    public int getKthAncestor(int node, int k) {
        int res = node+1;
        for(int i = 0; i < 20; i++){
```

```
            // 测试了两种方法，都可以
            // if(((k >> i) & 1) > 0){
            //     res = dp[res][i];

            // }
            if(((1 << i) & k) > 0){
                res = dp[res][i];

            }
        }
        return res - 1;
    }
}
/**
 * Your TreeAncestor object will be instantiated and called as such:
 * TreeAncestor obj = new TreeAncestor(n, parent);
 * int param_1 = obj.getKthAncestor(node,k);
 */
```

75、节点与其祖先之间的最大差值（1026）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

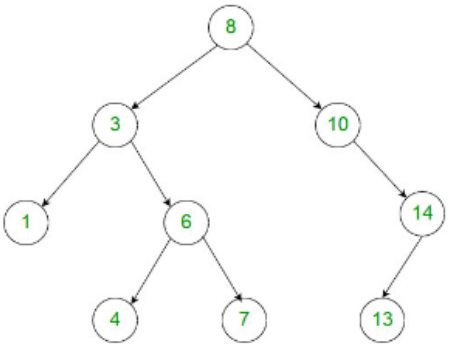
1026. 节点与其祖先之间的最大差值

难度 中等 56 收藏 分享 切换为英文 关注 反馈

给定二叉树的根节点 `root`，找出存在于不同节点 `A` 和 `B` 之间的最大值 `V`，其中 `V = |A.val - B.val|`，且 `A` 是 `B` 的祖先。

（如果 A 的任何子节点之一为 B，或者 A 的任何子节点是 B 的祖先，那么我们认为 A 是 B 的祖先）

示例：



输入： [8,3,10,1,6,null,14,null,null,4,7,13]
输出： 7
解释：
我们有大量的节点与其祖先的差值，其中一些如下：
|8 - 3| = 5
|3 - 7| = 4
|8 - 1| = 7
|10 - 13| = 3
在所有可能的差值中，最大值 7 由 |8 - 1| = 7 得出。

提示：

1. 树中的节点数在 2 到 5000 之间。
2. 每个节点的值介于 0 到 100000 之间。

https://leetcode-cn.com/problems/maximum-difference-between-node-and-ancestor/solution/dfs-guo-cheng-zhong-geng-xin-liang-ge-bian-liang-z/

解题思路

本题意在求出节点与其祖先节点之间的最大差值，所以我们首先想到的是遍历树，然后在遍历的过程中进行一定的处理。
我们这里需要用到的是前序遍历，先访问该节点，再访问左右节点，在每次访问一个节点的时候，会传来它的祖先节点中的最大val和最小val，然后访问该节点的时候就能通过比较得到目前的最大差值，顺便更新它的子节点的祖先节点的最大val和最小val。
这样，通过前序遍历完整棵树，我们可以得到全局的最大差值。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
```

```
 */
class Solution {
    public int res = 0; //保存节点和祖先节点之间的最大差值
    public int maxAncestorDiff(TreeNode root) {
        dfs(root, root.val, root.val);
        return res;
    }
    private void dfs(TreeNode root, int maxV, int minV){
        if(root==null)
            return;
        maxV = Math.max(maxV, root.val);//更新该节点子节点的 祖先节点的最大val 和最小val
        minV = Math.min(minV, root.val);
        int res1 = Math.max(Math.abs(root.val-maxV), Math.abs(root.val-minV));
        res = Math.max(res, res1);
        dfs(root.left, maxV, minV);
        dfs(root.right, maxV, minV);
    }
}
```

76、树的子结构（剑指 26）【子结构不是子树】

公众号：
yes的练级攻略

剑指 Offer 26. 树的子结构

难度 中等 92 收藏 分享 切换为英文 关注 反馈

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构， 即 A中有出现和B相同的结构和节点值。

例如:
给定的树 A:

```
     3
    / \
   4   5
  / \
 1   2
```

给定的树 B:

```
   4
  /
 1
```

返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。

示例 1:

```
输入: A = [1,2,3], B = [3,1]
输出: false
```

示例 2:

```
输入: A = [3,4,5,1,2], B = [4,1]
输出: true
```

限制:

`0 <= 节点个数 <= 10000`

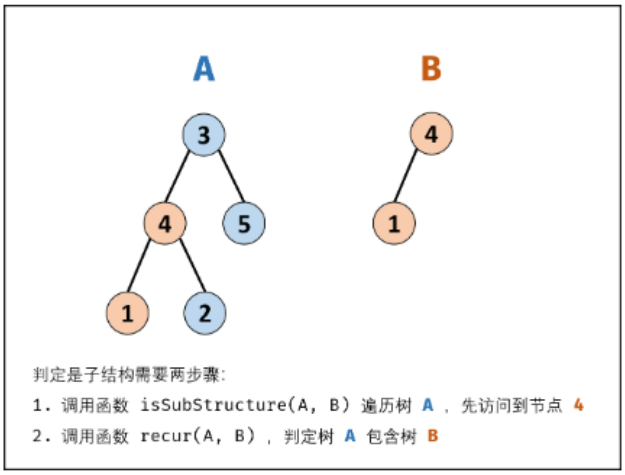
https://leetcode-cn.com/problems/shu-de-zi-jie-gou-lcof/solution/mian-shi-ti-26-shu-de-zi-jie-gou-xian-xu-bian-li-p/

更多硬核内容就在公众号「yes 的练级攻略」

解题思路：

若树 B 是树 A 的子结构，则子结构的根节点可能为树 A 的任意一个节点。因此，判断树 B 是否是树 A 的子结构，需完成以下两步工作：

1. 先序遍历树 A 中的每个节点 n_A ； （对应函数 `isSubStructure(A, B)` ）
2. 判断树 A 中 **以 n_A 为根节点的子树** 是否包含树 B 。 （对应函数 `recur(A, B)` ）



算法流程：

> 名词规定：树 A 的根节点记作 **节点 A** ，树 B 的根节点称为 **节点 B** 。

`recur(A, B)` **函数：**

1. **终止条件：**
    1. 当节点 B 为空：说明树 B 已匹配完成（越过叶子节点），因此返回 $true$ ；
    2. 当节点 A 为空：说明已经越过树 A 叶子节点，即匹配失败，返回 $false$ ；
    3. 当节点 A 和 B 的值不同：说明匹配失败，返回 $false$ ；
2. **返回值：**
    1. 判断 A 和 B 的**左**子节点是否相等，即 `recur(A.left, B.left)` ；
    2. 判断 A 和 B 的**右**子节点是否相等，即 `recur(A.right, B.right)` ；

`isSubStructure(A, B)` **函数：**

1. **特例处理：** 当 树 A 为空 **或** 树 B 为空 时，直接返回 $false$ ；
2. **返回值：** 若树 B 是树 A 的子结构，则必满足以下三种情况之一，因此用或 `||` 连接；
    1. 以 **节点 A 为根节点的子树** 包含树 B ，对应 `recur(A, B)` ；
    2. 树 B 是 **树 A 左子树** 的子结构，对应 `isSubStructure(A.left, B)` ；
    3. 树 B 是 **树 A 右子树** 的子结构，对应 `isSubStructure(A.right, B)` ；

> 以上 `2.` `3.` 实质上是在对树 A 做 **先序遍历** 。

复杂度分析：

- **时间复杂度 $O(MN)$ ：** 其中 M, N 分别为树 A 和 树 B 的节点数量；先序遍历树 A 占用 $O(M)$ ，每次调用 `recur(A, B)` 判断占用 $O(N)$ 。
- **空间复杂度 $O(M)$ ：** 当树 A 和树 B 都退化为链表时，递归调用深度最大。当 $M \leq N$ 时，遍历树 A 与递归判断的总递归深度为 M ；当 $M > N$ 时，最差情况为遍历至树 A 叶子节点，此时总递归深度为 M。

```
/**
```

```
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        return (A != null && B != null) && (recur(A, B) || isSubStructure(A.lef
t, B) || isSubStructure(A.right, B));
    }
    boolean recur(TreeNode A, TreeNode B) {
        if(B == null) return true;
        if(A == null || A.val != B.val) return false;
        return recur(A.left, B.left) && recur(A.right, B.right);
    }
}
```

77、另一个树的子树（572）【子树是节点和所有子孙】【检查子树（面试题 04.10）】

572. 另一个树的子树

难度 简单 341 收藏 分享 切换为英文 关注 反馈

给定两个非空二叉树 **s** 和 **t**，检验 **s** 中是否包含和 **t** 具有相同结构和节点值的子树。**s** 的一个子树包括 **s** 的一个节点和这个节点的所有子孙。**s** 也可以看做它自身的一棵子树。

示例 1:
给定的树 s:

```
     3
    / \
   4   5
  / \
 1   2
```

给定的树 t:

```
    4
   / \
  1   2
```

返回 **true**，因为 t 与 s 的一个子树拥有相同的结构和节点值。

示例 2:
给定的树 s:

```
     3
    / \
   4   5
  / \
 1   2
    /
   0
```

给定的树 t:

```
   4
  / \
 1   2
```

返回 **false**。

https://leetcode-cn.com/problems/subtree-of-another-tree/solution/java-di-gui-ban-by-kelly2018/

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (t == null) return true;   // t 为 null 一定都是 true
        if (s == null) return false;  // 这里 t 一定不为 null，只
要 s 为 null，肯定是 false
        return isSubtree(s.left, t) || isSubtree(s.right, t) || isSameT
ree(s,t);
    }
    /**
     * 判断两棵树是否相同
     */
    public boolean isSameTree(TreeNode s, TreeNode t){
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (s == null && t == null) return true;
        if (s == null || t == null) return false;
        if (s.val != t.val) return false;
        return isSameTree(s.left, t.left) && isSameTree(s.right, t.righ
t);
    }
}
```

78、最大 BST 子树（333）

333. 最大 BST 子树

难度 中等 44

给定一个二叉树，找到其中最大的二叉搜索树（BST）子树，其中最大指的是子树节点数最多的。

注意:
子树必须包含其所有后代。

示例:

```
输入：[10,5,15,1,8,null,7]

   10
   / \
  5  15
 / \   \
1   8   7

输出：3
解释：高亮部分为最大的 BST 子树。
     返回值 3 在这个样例中为子树大小。
```

进阶:
你能想出用 O(n) 的时间复杂度解决这个问题吗？

https://blog.csdn.net/ruochen82155551

思路

利用递归函数设计一个二叉树后序遍历的过程：
（1）先遍历左子树收集信息
（2）遍历右子树收集信息
（3）在根节点做信息整合

以节点root为根节点的子树中，最大BST有以下3种情况：
（1）以节点root为根节点的子树中，最大BST就是root的左子树中的最大BST
（2）以节点root为根节点的子树中，最大BST就是root的右子树中的最大BST
（3）如果root左子树的最大BST是左子树全体，root右子树的最大BST是右子树全体，并且root的值大于左子树所有节点最大值，root的值小于右子树所有节点的最小值。则root为跟姐的最大BST子树就是root全体子树；即用root将左右子树链接起来

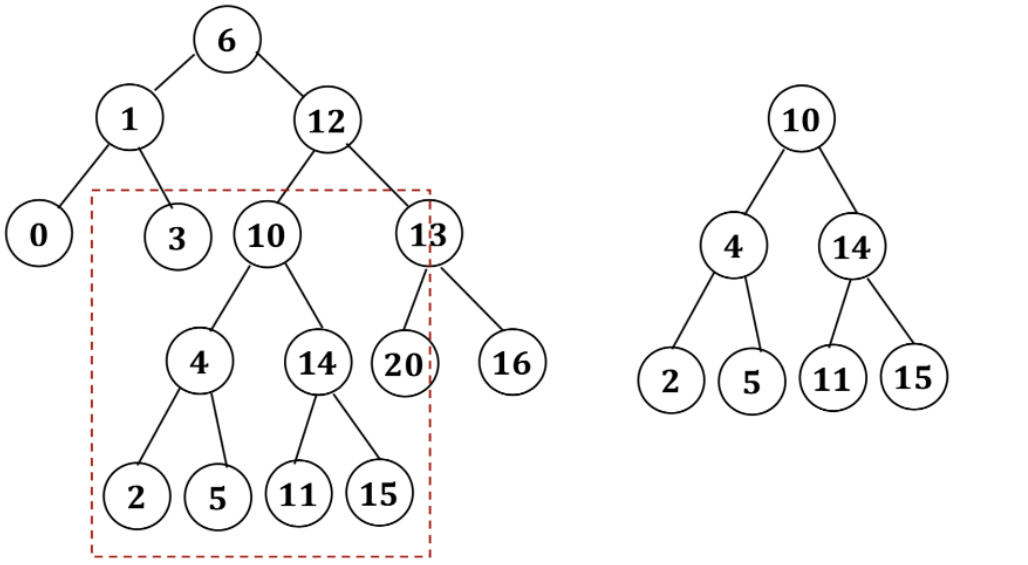
因为是后序遍历，时间复杂度为O（n）

https://www.baidu.com/link?url=la7ilRHFf5W-ug4j1CRKDCndLVb9pdo8QJxNXZ4QkdBjPs6-aWlIlB0-WQuEy8CHmRz1gon1-nQk4CKvjq-nmvmPjFpHC0gWeKWW-7RkRnS&wd=&eqid=da75a145004879a7000000035f261e03

更多硬核内容就在公众号「yes 的练级攻略」

题目

给定一棵二叉树的头节点head，已知其中所有节点的值都不一样，找到含有节点最多的搜索二叉子树，并返回这棵子树的头节点。例如，下图中，右树就是左树的最大搜索子树。



解题思路—**后序遍历**：**后序遍历二叉树**，若当前结点的左右子树都符合搜索二叉树，就返回当前结点；若有左子树（右子树）不符合，则返回右子树根节点（左子树根节点）。**需要设置全局变量来保存当前子树的最大值、最小值以及节点数。**

代码不是特别好理解，所以给出了main，里面已经建立了一棵与上图中一样的二叉树，结合debug更易理解。

```
public static void main(String[] args) {
    // 下面已经建立了一棵与示例图中相同的二叉树
    TreeNode root = new TreeNode(6);
    root.left = new TreeNode(1);
    root.right = new TreeNode(12);
    root.left.left = new TreeNode(0);
    root.left.right = new TreeNode(3);
    root.right.left = new TreeNode(10);
    root.right.left.left = new TreeNode(4);
    root.right.left.left.left = new TreeNode(2);
    root.right.left.left.right = new TreeNode(5);
    root.right.left.right = new TreeNode(14);
    root.right.left.right.left = new TreeNode(11);
    root.right.left.right.right = new TreeNode(15);
    root.right.right = new TreeNode(13);
    root.right.right.left = new TreeNode(20);
    root.right.right.right = new TreeNode(16);


    TreeNode sub = biggestSubBST(root);
}
```

公众号：yes的练级攻略

```
public static TreeNode biggestSubBST(TreeNode head) {
    // 记录节点数，当前子树的最大值，最小值
    int[] record = new int[3];
    return posOrder(head, record);
}

public static TreeNode posOrder(TreeNode head, int[] record) {
    if (head == null) {
        // 当遍历到叶子节点时
        record[0] = 0;
        record[1] = Integer.MAX_VALUE;
        record[2] = Integer.MIN_VALUE;
        return null;
    }
    int value = head.val;// 头节点的值
    TreeNode left = head.left;// 左子树的头节点
    TreeNode right = head.right;// 右子树的头节点
    TreeNode lBST = posOrder(left, record);
    int lSize = record[0];
    int lMin = record[1];
    int lMax = record[2];
    TreeNode rBST = posOrder(right, record);
    int rSize = record[0];
    int rMin = record[1];
    int rMax = record[2];
    // 保存当前子树中的最大值与最小值，不是只保存左子树的最大值与右子树的最小值
    // 因为在递归的过程中，不知道当前是左子树还是右子树
    record[1] = Math.min(lMin, value);
    record[2] = Math.max(rMax, value);
    // left == lBST && right == rBST 这个判断条件特别重要
    // 只有当左右两个子树的根节点是当前节点的左右子节点，当前节点才能作为根节点
加入
    if (left == lBST && right == rBST && lMax < value && value < rMin) {
        record[0] = lSize + rSize + 1;
        return head;
    }
    record[0] = Math.max(lSize, rSize);
    // 若左右子树不能合并，返回当前最长的那个子树的根节点
    return lSize > rSize ? lBST : rBST;
}
```

79、二叉树的右视图（199）

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

199. 二叉树的右视图

难度 中等 293 收藏 分享 切换为英文 关注 反馈

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

输入: [1,2,3,null,5,null,4]
输出: [1, 3, 4]
解释:

```
   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

https://leetcode-cn.com/problems/binary-tree-right-side-view/solution/jian-dan-bfsdfs-bi-xu-miao-dong-by-sweetiee/

禁止商用

一、BFS

思路： 利用 BFS 进行层次遍历，记录下每层的最后一个元素。
时间复杂度： $O(N)$，每个节点都入队出队了 1 次。
空间复杂度： $O(N)$，使用了额外的队列空间。

方法 1：BFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> que = new LinkedList<>();
        que.offer(root);
        while (!que.isEmpty()){
            int size = que.size();
            for (int i = 0; i < size; i++){
                TreeNode node = que.poll();
                if (node.left != null) que.offer(node.left);
                if (node.right != null) que.offer(node.right);
```

```
                if(i == size - 1) res.add(node.val);
            }
        }
        return res;
    }
}
```

二、DFS （时间100%）

思路： 我们按照 「根结点 -> 右子树 -> 左子树」 的顺序访问，就可以保证每层都是最先访问最右边的节点的。

（与先序遍历 「根结点 -> 左子树 -> 右子树」 正好相反，先序遍历每层最先访问的是最左边的节点）

时间复杂度： $O(N)$，每个节点都访问了 1 次。
空间复杂度： $O(N)$，因为这不是一棵平衡二叉树，二叉树的深度最少是 $logN$，最坏的情况下会退化成一条链表，深度就是 N，因此递归时使用的栈空间是 $O(N)$ 的。

方法 2：DFS

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    List<Integer> res = new ArrayList<>();
    public List<Integer> rightSideView(TreeNode root) {
        dfs(root, 0); // 从根节点开始访问，根节点深度是0
        return res;
    }
    public void dfs(TreeNode root, int depth){
        if (root == null) return;
        // 先访问 当前节点，再递归地访问 右子树 和 左子树。
        // 如果当前节点所在深度还没有出现在res 里，说明在该深度下当前节点是第一个被访问的节点，因此将当前节点加入 res 中。
        if (depth == res.size()) res.add(root.val);
        depth++;
        dfs(root.right, depth);
        dfs(root.left, depth);
    }
}
```

80、二叉树中的最长交错路径（1372）

1372. 二叉树中的最长交错路径

难度 中等 28 收藏 分享 切换为英文 关注 反馈

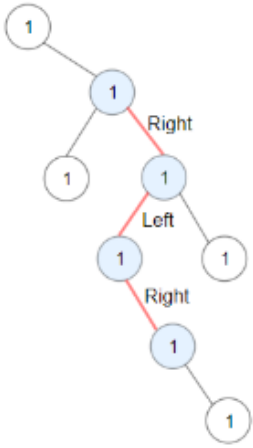
给你一棵以 `root` 为根的二叉树，二叉树中的交错路径定义如下：

- 选择二叉树中 **任意** 节点和一个方向（左或者右）。
- 如果前进方向为右，那么移动到当前节点的的右子节点，否则移动到它的左子节点。
- 改变前进方向：左变右或者右变左。
- 重复第二步和第三步，直到你在树中无法继续移动。

交错路径的长度定义为：**访问过的节点数目 - 1**（单个节点的路径长度为 0 ）。

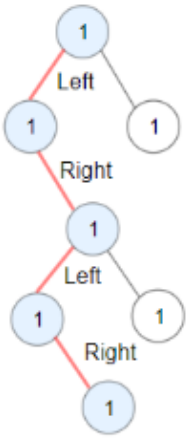
请你返回给定树中最长 **交错路径** 的长度。

示例 1：



输入： root = [1,null,1,1,1,null,null,1,1,null,1,null,null,null,1,null,1]
输出： 3
解释： 蓝色节点为树中最长交错路径（右 -> 左 -> 右）。

示例 2：




```
输入：root = [1,1,1,null,1,null,null,1,1,null,1]
输出：4
解释：蓝色节点为树中最长交错路径（左 -> 右 -> 左 -> 右）。
```

示例 3：

```
输入：root = [1]
输出：0
```

提示：

- 每棵树最多有 `50000` 个节点。
- 每个节点的值在 `[1, 100]` 之间。

方法 1：递归

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int res;
    public int longestZigZag(TreeNode root) {
        dfs(root.left, -1, 1);
        dfs(root.right, 1, 1);
        return res;
    }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    //last = -1 代表上一步是往左走的，last = 1 代表上一步是往右走的
    private void dfs(TreeNode root, int last, int len) {
        if (root == null) return;
        res = Math.max(res, len);
        if (last == -1) {
            dfs(root.left, -1, 1);
            dfs(root.right, 1, len + 1);
        } else {
            dfs(root.left, -1, len + 1);
            dfs(root.right, 1, 1);
        }
    }
}
```

https://leetcode-cn.com/problems/longest-zigzag-path-in-a-binary-tree/solution/javazi-di-xiang-shang-shu-xing-dp-by-jackie-tien/

解题思路

res[0]表示当前节点下一步向左走带来的最大收益，res[1]表示当前节点下一步向右走带来的最大收益
res[0]=1+left[1] 当前节点下一步向左走带来的最大收益等于左子节点向右走的最大收益+1
res[1]=1+right[0] 当前节点下一步向右走带来的最大收益等于右子节点向左走的最大收益+1

维护一个全局变量maxPath，每次遍历某一节点时，更新它

方法 2：树形 DP

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    private int max;
    public int longestZigZag(TreeNode root) {
        dfs(root);
        return max;
    }
    //last = -1 代表上一步是往左走的，last = 1 代表上一步是往右走的
    private int[] dfs(TreeNode root) {
        int[] res = new int[2];
        if (root == null) {
            res[0] = -1;
            res[1] = -1;
```

```
            return res;
        }
        int[] left = dfs(root.left);
        int[] right = dfs(root.right);
        res[0] = left[1] + 1;
        res[1] = right[0] + 1;
        max = Math.max(max, Math.max(res[0], res[1]));
        return res;
    }
}
```

81、二叉树的边界（545）

545. 二叉树的边界

难度 中等 32 收藏 分享 切换为英文 关注 反馈

给定一棵二叉树，以**逆时针**顺序从根开始返回其边界。边界按顺序包括左边界、叶子结点和右边界而不包括重复的**结点**。(结点的值可能重复)

左边界的定义是从根到**最左侧**结点的路径。**右边界**的定义是从根到**最右侧**结点的路径。若根没有左子树或右子树，则根自身就是左边界或右边界。注意该定义只对输入的二叉树有效，而对子树无效。

最左侧结点的定义是：在左子树存在时总是优先访问，如果不存在左子树则访问右子树。重复以上操作，首先抵达的结点就是最左侧结点。

最右侧结点的定义方式相同，只是将左替换成右。

示例 1

输入:

```
  1
   \
    2
   / \
  3   4
```

输出:
`[1, 3, 4, 2]`

解析:
根不存在左子树，故根自身即为左边界。
叶子结点是3和4。
右边界是1，2，4。注意逆时针顺序输出需要你输出时调整右边界顺序。
以逆时针顺序无重复地排列边界，得到答案`[1,3,4,2]`。

示例 2

输入:

```
     ____1_____
    /          \
   2            3
  / \          /
 4   5        6
    / \      / \
   7   8    9  10
```

输出:
`[1,2,4,7,8,9,10,6,3]`

解析:
左边界是结点`1,2,4`。(根据定义，4是最左侧结点)
叶子结点是结点`4,7,8,9,10`。
右边界是结点`1,3,6,10`。(10是最右侧结点)
以逆时针顺序无重复地排列边界，得到答案 `[1,2,4,7,8,9,10,6,3]`。

https://leetcode-cn.com/problems/boundary-of-binary-tree/solution/545-er-cha-shu-de-bian-jie-by-klb/

解题思路

把任务拆解：寻找上边界、寻找左边界、寻找下边界、寻找右边界。

定义一个 list 存储结果。

1、寻找上边界

根节点就是上边界，值存入 list 中。

2、寻找左边界

如果 root 有左子树，就存在左边界：

1、从 root.left 开始，一直往下遍历左子树；

2、当走到某一个 node 发现没有左子树，但是有右子树，那么就用 node.right 继续往下遍历；

3、当遇到 node 没有左右子结点，说明已经到下边界了，这个 node 不加入到左边界中。

把遍历到的每一个节点的值依次加入到 list 中。

如果 root 没有左子树，则没有左边界，进入下一步寻找下边界。

3、寻找下边界

下边界即 root 的所有叶子节点，把叶子节点全部加入到 list 中。

4、寻找右边界

右边界的和左边界的操作一样，只不过把左和右进行互换，把值存入到 stack 中，最后 stack 倒出来实现逆序，加入到总 list 中。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<Integer> boundaryOfBinaryTree(TreeNode root) {
        ArrayList<Integer> res = new ArrayList<>();
        if (root == null) return res;
        if (!isLeaf(root)) res.add(root.val);   // 根节点就是上边界
        TreeNode t = root.left;
        // 如果有左子树，那么就存在左边界，否则没有左边界
        while (t != null) {
            // 不是叶子节点，即为左边界
```

```
            if (!isLeaf(t)) res.add(t.val);
            if (t.left != null) {
                t = t.left;
            } else { // 当走到某一层的左边界节点没有左子节点，则拿它的右子节点继续遍历
                t = t.right;
            }
        }
        // 添加下边界
        addLeaves(res, root);
        // 添加右边界，定义一个栈来最后调转顺序
        Stack<Integer> s = new Stack<>();
        t = root.right;
        while (t != null) {
            if (!isLeaf(t)) {
                s.push(t.val);
            }
            if (t.right != null) {
                t = t.right;
            } else {
                t = t.left;
            }
        }
        while (!s.empty()) {
            res.add(s.pop());
        }
        return res;
    }
    public boolean isLeaf(TreeNode t) {
        return t.left == null && t.right == null;
    }
    public void addLeaves(List<Integer> res, TreeNode root) {
        if (isLeaf(root)) {
            res.add(root.val);
        } else {
            // 左子树的叶子节点加入 res
            if (root.left != null) addLeaves(res, root.left);
            // 右子树的叶子节点加入 res
            if (root.right != null) addLeaves(res, root.right);
        }
    }
}
```

十七、线段树

线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。

使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为 O(logN)。而未优化的空间复杂度为 2N，实际应用时一般还要开 4N 的数组以免越界，因此有时需要离散化让空间压缩。

对于线段树中的每一个非叶子节点[a,b]，它的左儿子表示的区间为[a,(a+b)/2]，右儿子表示的区间为[(a+b)/2+1,b]。因此线段树是平衡二叉树，最后的子节点数目为 N，即整个线段区间的长度。

使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为 O(logN)。而未优化的空间复杂度为 2N，因此有时需要离散化让空间压缩。

1、线段树的构造（lintcode 201）

201. 线段树的构造 中文

线段树是一棵二叉树，他的每个节点包含了两个额外的属性 `start` 和 `end` 用于表示该节点所代表的区间。start和end都是整数，并按照如下的方式赋值:

- 根节点的 *start* 和 *end* 由 `build` 方法所给出。
- 对于节点 A 的左儿子，有 `start=A.start, end=(A.start + A.end) / 2`。
- 对于节点 A 的右儿子，有 `start=(A.start + A.end) / 2 + 1, end=A.end`。
- 如果 *start* 等于 *end*, 那么该节点是叶子节点，不再有左右儿子。

实现一个 `build` 方法，接受 *start* 和 *end* 作为参数, 然后构造一个代表区间 `[start, end]` 的线段树，返回这棵线段树的根。

样例

样例 1:

```
输入: [1,4]
输出: "[1,4][1,2][3,4][1,1][2,2][3,3][4,4]"
解释:
                    [1,  4]
                   /       \
             [1,  2]         [3, 4]
             /     \         /     \
          [1, 1]  [2, 2]    [3, 3]  [4, 4]
```

样例 2:

```
输入: [1,6]
输出: "[1,6][1,3][4,6][1,2][3,3][4,5][6,6][1,1][2,2][4,4][5,5]"
解释:
              [1,  6]
             /       \
      [1,  3]         [4,  6]
      /     \         /     \
   [1, 2]  [3,3]    [4, 5]   [6,6]
   /    \          /     \
[1,1]   [2,2]    [4,4]   [5,5]
```

说明

线段树(又称区间树), 是一种高级数据结构，他可以支持这样的一些操作:

- 查找给定的点包含在了哪些区间内
- 查找给定的区间包含了哪些点

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end) {
 *         this.start = start, this.end = end;
 *         this.left = this.right = null;
 *     }
```

```
 * }
 */
public class Solution {
    /*
     * @param start: start value.
     * @param end: end value.
     * @return: The root of Segment Tree.
     */
    public SegmentTreeNode build(int start, int end) {
        if (start > end) return null;
        SegmentTreeNode root = new SegmentTreeNode(start, end);
        if (start != end) {
            int mid = start + (end - start) / 2;
            root.left = build(start, mid);
            root.right = build(mid + 1, end);
        }
        return root;
    }
}
```

禁止商用

2、线段树的构造 2（lintcode 439）

439. 线段树的构造 II　　中文

线段树是一棵二叉树，他的每个节点包含了两个额外的属性 `start` 和 `end` 用于表示该节点所代表的区间。 `start` 和 `end` 都是整数，并按照如下的方式赋值:

- 根节点的 `start` 和 `end` 由 `build` 方法所给出。
- 对于节点 A 的左儿子，有 `start=A.left, end=(A.left + A.right) / 2`。
- 对于节点 A 的右儿子，有 `start=(A.left + A.right) / 2 + 1, end=A.right`。
- 如果 `start` 等于 `end`，那么该节点是叶子节点，不再有左右儿子。

对于给定数组实现 `build` 方法，线段树的每个节点储存区间最大值，返回根节点。

样例
输入：[3,2,1,4]
解释：
这颗线段树将会是

```
          [0,3](max=4)
          /          \
       [0,1]         [2,3]
     (max=3)       (max=4)
     /   \          /    \
  [0,0]  [1,1]    [2,2]  [3,3]
 (max=3)(max=2)  (max=1)(max=4)
```

说明
线段树(又称区间树)是一种高级数据结构，支持一系列区间查询/修改操作。

维基百科: https://zh.wikipedia.org/zh-hans/線段樹

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end, max;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end, int max) {
```

```
 *              this.start = start;
 *              this.end = end;
 *              this.max = max
 *              this.left = this.right = null;
 *      }
 * }
 */

public class Solution {
    /**
     * @param A: a list of integer
     * @return: The root of Segment Tree
     */
    public SegmentTreeNode build(int[] A) {
        if (A == null || A.length == 0) return null;
        return buildTree(0, A.length - 1, A);
    }
    public SegmentTreeNode buildTree(int start, int end, int[] A){
        if (start == end) return new SegmentTreeNode(start, end, A[start]);
        SegmentTreeNode node = new SegmentTreeNode(start, end, Integer.MIN_VALUE);
        int mid = start + (end - start) / 2;
        node.left = buildTree(start, mid, A);
        node.right = buildTree(mid+1, end, A);
        node.max = Math.max(node.left.max, node.right.max);
        return node;
    }
}
```

3、线段树的查询（lintcode 202）

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

202. 线段树的查询 中文 English

对于一个有n个数的整数数组，在对应的线段树中, 根节点所代表的区间为0-n-1, 每个节点有一个额外的属性 `max`，值为该节点所代表的数组区间start到end内的最大值。

为SegmentTree设计一个 `query` 的方法，接受3个参数 `root`，`start` 和 `end`，线段树root所代表的数组中子区间[start, end]内的最大值。

样例

样例 1:

```
输入: "[0,3,max=4][0,1,max=4][2,3,max=3][0,0,max=1][1,1,max=4][2,2,max=2][3,3,max=3]",1,2
输出: 4
解释:
对于数组 [1, 4, 2, 3], 对应的线段树为 :

                        [0, 3, max=4]
                       /             \
                [0,1,max=4]        [2,3,max=3]
                /         \        /         \
          [0,0,max=1] [1,1,max=4] [2,2,max=2], [3,3,max=3]
[1,2]区间最大值为4
```

样例 2:

```
输入: "[0,3,max=4][0,1,max=4][2,3,max=3][0,0,max=1][1,1,max=4][2,2,max=2][3,3,max=3]",2,3
输出: 3
解释:
对于数组 [1, 4, 2, 3], 对应的线段树为 :

                        [0, 3, max=4]
                       /             \
                [0,1,max=4]        [2,3,max=3]
                /         \        /         \
          [0,0,max=1] [1,1,max=4] [2,2,max=2], [3,3,max=3]
[2,3]区间最大值为3
```

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end, max;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end, int max) {
 *         this.start = start;
 *         this.end = end;
 *         this.max = max
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
     * @param root: The root of segment tree.
     * @param start: start value.
     * @param end: end value.
     * @return: The maximum number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        if (start > end) return 0;
        if (root.start == root.end) return root.max;
```

公众号：yes的练级攻略

```
            int mid = root.start + (root.end - root.start) / 2;
            int left = query(root.left, start, Math.min(mid, end));
            int right = query(root.right, Math.max(mid+1, start), end);
            return Math.max(left, right);
        }
}
```

一个逻辑比reference更清晰的解法：

首先大的前提条件是:root.start<=start && end<=root.end 在这个大的前提条件下，按以下关系递归下去的每一个node一定满足:node.start<=start && end<=node.end

所以node, start, end一共有以下几种情况： 1. node.start==start && end==node.end: return node.max 2. node.start<start || end<node.end(包含以下几种子情况): (1)end<=mid(只在node左边): return (node.left, start, end) (2)mid+1<=start(只在node右边): return (node.right, start, end) (3)node左右都有: return Math.max((node.left, start, mid), (node.right, mid+1, end))

```
/**
* 本参考程序由九章算法用户提供。版权所有，转发请注明出处。
* - 九章算法致力于帮助更多中国人找到好的工作，授课老师均来自硅谷和国内的一线大公司在职工程师。
* - 现有的求职课程包括：九章算法班 2020升级版，算法强化班，算法基础班，北美算法面试高频题班，Java 高级工程师 P6+ 小班课，面试软技能指导
- BQ / Resume / Project 2020版
* - Design类课程包括：系统设计 System Design，面向对象设计 OOD
* - 专题及项目类课程包括：动态规划专题班，Big Data - Spark 项目实战，Django 开发项目课
* - 更多详情请见官方网站：http://www.jiuzhang.com/?utm_source=code
*/
public int query(SegmentTreeNode node, int start, int end) {
        if(node.start==start && node.end==end) {
            return node.max;
        }

        int nodeMid = (node.start+node.end)/2;

        if(end<=nodeMid) {
            return query(node.left, start, end);
        }
        else if(start>=nodeMid+1) {
            return query(node.right, start, end);
        }
        else {
            int leftMax = query(node.left, start, nodeMid);
            int rightMax = query(node.right, nodeMid+1, end);
            return Math.max(leftMax, rightMax);
        }
    }
```

4、线段树的查询 2（lintcode 247）

247. 线段树查询 II　　中文 English

对于一个数组，我们可以对其建立一棵 线段树，每个结点存储一个额外的值 count 来代表这个结点所指代的数组区间内的元素个数. (数组中并不一定每个位置上都有元素)

实现一个 query 的方法，该方法接受三个参数 root, start 和 end，分别代表线段树的根节点和需要查询的区间，找到数组中在区间 [start, end]内的元素个数。

样例

样例 1:

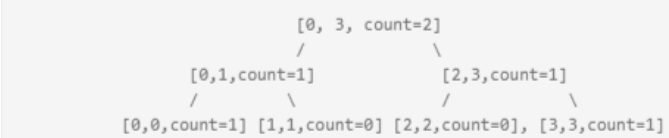
输入："[0,3,count=3][0,1,count=1][2,3,count=2][0,0,count=1][1,1,count=0][2,2,count=1][3,3,count=1]",[[1, 1], [1, 2], [2, 3], [0, 2]]
输出：[0,1,2,2]
解释：
对应的线段树为：

```
                          [0, 3, count=3]
                          /             \
                 [0,1,count=1]             [2,3,count=2]
                 /         \               /           \
          [0,0,count=1] [1,1,count=0] [2,2,count=1], [3,3,count=1]
```

```
Input : query(1,1), Output: 0

Input : query(1,2), Output: 1

Input : query(2,3), Output: 2

Input : query(0,2), Output: 2
```

样例 2:

输入: "[0,3,count=3][0,1,count=1][2,3,count=2][0,0,count=1][1,1,count=0][2,2,count=0][3,3,count=1]",[[1, 1], [1, 2], [2, 3], [0, 2]]
输出: [0,0,1,1]
解释:
对应的线段树为:




```
Input : query(1,1), Output: 0

Input : query(1,2), Output: 0

Input : query(2,3), Output: 1

Input : query(0,2), Output: 1
```

与 I 相比，这里的节点值是 count，最小为 0，就不需要和 mid 比较了。如果与 start，end 区间没有交集，或者为 null 就是 0. 其余递归累加左右子树，当子节点区间在查找区间内，返回其 count。

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end, count;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end, int count) {
 *         this.start = start;
 *         this.end = end;
 *         this.count = count;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /*
     * @param root: The root of segment tree.
     * @param start: start value.
     * @param end: end value.
     * @return: The count number in the interval [start, end]
     */
    public int query(SegmentTreeNode root, int start, int end) {
        if (root == null || root.start > end || root.end < start) return 0;
        if (start <= root.start && root.end <= end) return root.count;
        return query(root.left, start, end) + query(root.right, start, end);
    }
}
```

5、线段树的修改（lintcode 203）

禁止商用
公众号：
yes的练级攻略

203. 线段树的修改 中文

对于一棵 **最大线段树**, 每个节点包含一个额外的 `max` 属性，用于存储该节点所代表区间的最大值。

设计一个 `modify` 的方法，接受三个参数 `root`、 `index` 和 `value`。该方法将 *root* 为根的线段树中 [*start*, *end*] = [*index*, *index*] 的节点修改为了新的 *value* ，并确保在修改后，线段树的每个节点的 *max* 属性仍然具有正确的值。

样例
样例 1:

输入: "[1,4,max=3][1,2,max=2][3,4,max=3][1,1,max=2][2,2,max=1][3,3,max=0][4,4,max=3]",2,4
输出: "[1,4,max=4][1,2,max=4][3,4,max=3][1,1,max=2][2,2,max=4][3,3,max=0][4,4,max=3]"
解释:
线段树:

```
                          [1, 4, max=3]
                         /              \
                [1, 2, max=2]            [3, 4, max=3]
               /              \          /             \
       [1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=3]
```

如何调用modify(root, 2, 4)，可以得到:

```
                          [1, 4, max=4]
                         /              \
                [1, 2, max=4]            [3, 4, max=3]
               /              \          /             \
       [1, 1, max=2], [2, 2, max=4], [3, 3, max=0], [4, 4, max=3]
```

样例 2:

输入: "[1,4,max=3][1,2,max=2][3,4,max=3][1,1,max=2][2,2,max=1][3,3,max=0][4,4,max=3]",4,0
输出: "[1,4,max=4][1,2,max=4][3,4,max=0][1,1,max=2][2,2,max=4][3,3,max=0][4,4,max=0]"
解释:
线段树:

```
                          [1, 4, max=3]
                         /              \
                [1, 2, max=2]            [3, 4, max=3]
               /              \          /             \
       [1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=3]
```

如果调用modify(root, 4, 0)，可以得到:

```
                          [1, 4, max=2]
                         /              \
                [1, 2, max=2]            [3, 4, max=0]
               /              \          /             \
       [1, 1, max=2], [2, 2, max=1], [3, 3, max=0], [4, 4, max=0]
```

挑战
时间复杂度 `O(h)` , h 是线段树的高度

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end, max;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end, int max) {
 *         this.start = start;
 *         this.end = end;
 *         this.max = max
 *         this.left = this.right = null;
 *     }
```

```
 * }
 */
public class Solution {
    /**
     * @param root: The root of segment tree.
     * @param index: index.
     * @param value: value
     * @return: nothing
     */
    public void modify(SegmentTreeNode root, int index, int value) {
        if (index < root.start || index > root.end) return;
        if (root.start == root.end) {
            if (index == root.start) root.max = value;
            return;
        }
        modify(index < root.right.start ? root.left : root.right, index, value);
        root.max = Math.max(root.left.max, root.right.max);
    }
}
```

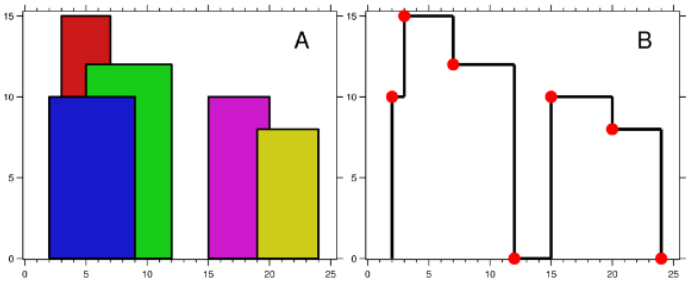
禁止商用

6、天际线问题（218）

218. 天际线问题

难度 困难 252 收藏 分享 切换为英文 关注 反馈

城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。现在，假设您获得了城市风光照片（图A）上**显示的所有建筑物的位置和高度**，请编写一个程序以输出由这些建筑物**形成的天际线**（图B）。



每个建筑物的几何信息用三元组 `[Li，Ri，Hi]` 表示，其中 `Li` 和 `Ri` 分别是第 i 座建筑物左右边缘的 x 坐标，`Hi` 是其高度。可以保证 `0 ≤ Li, Ri ≤ INT_MAX`，`0 < Hi ≤ INT_MAX` 和 `Ri - Li > 0`。您可以假设所有建筑物都是在绝对平坦且高度为 0 的表面上的完美矩形。

例如，图A中所有建筑物的尺寸记录为：`[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]` 。

输出是以 `[ [x1,y1], [x2, y2], [x3, y3], ... ]` 格式的"**关键点**"（图B中的红点）的列表，它们唯一地定义了天际线。**关键点是水平线段的左端点**。请注意，最右侧建筑物的最后一个关键点仅用于标记天际线的终点，并始终为零高度。此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。

例如，图B中的天际线应该表示为：`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`。

说明:

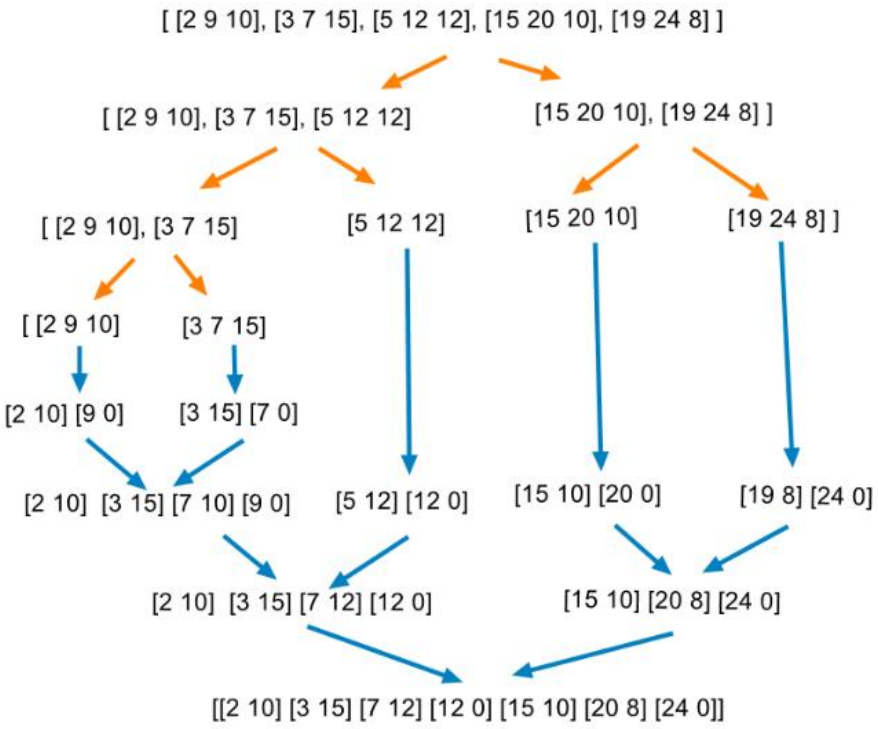
- 任何输入列表中的建筑物数量保证在 `[0, 10000]` 范围内。
- 输入列表已经按左 `x` 坐标 `Li` 进行升序排列。
- 输出列表必须按 x 位排序。
- 输出天际线中不得有连续的相同高度的水平线。例如 `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` 是不正确的答案；三条高度为 5 的线应该在最终输出中合并为一个：`[...[2 3], [4 5], [12 7], ...]`

https://leetcode-cn.com/problems/the-skyline-problem/solution/xian-duan-shu-he-sao-miao-xian-suan-fa-jie-jue-ci-/

更多硬核内容就在公众号「yes 的练级攻略」

解题

光看题目描述似乎觉得很难，但是学习过线段树就会觉得这道题变得很容易。不信，来试试看下面的图：



我们可以把输入列表作为一个顶点，按照输入列表的长度选取中间的值，建议使用这个方式： `mid := l + (r-l)/2` 选择中间值，然后进行分治算法。

直到当前输入列表的长度为1，说明不能再分了，在这个地方作为结束条件，然后返回到另外路径划分其它的输入列表。

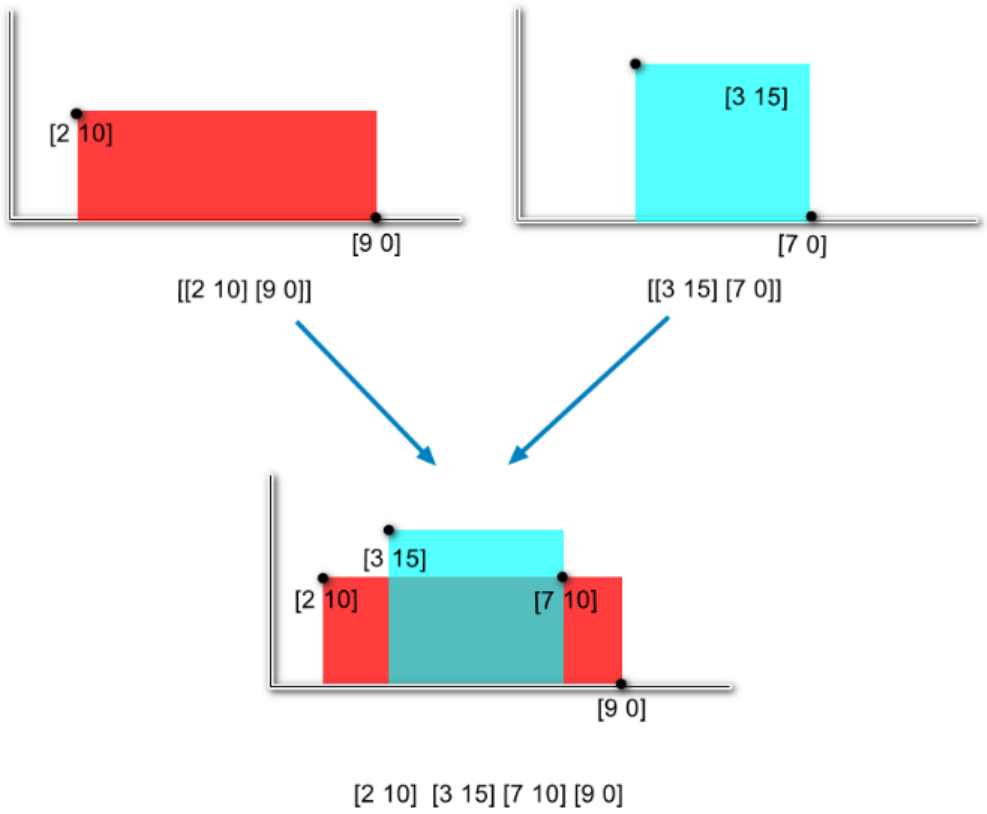
例如我们划分到 [[2 9 10]] 的时候，当前输入列表的长度为1，不能再进行分治了。

[[2 9 10]]表示的是一个建筑物，分别是左右边缘的横坐标和高度。题目已经将天际线定义为水平线左端点的集合，如[[2 9 10]]关键点集合为[[2 10] [9 0]]，分别是一个建筑物上的左上端点和右下端点。

同理，[[3 7 15]]的关键点集合为[[3 15] [7 0]]。

关键的一点来了，我们得到了[[2 9 10]] 和 [[3 7 15]] 两个集合之后，要求在满足题目天际线情况下，怎么把这两个集合进行合并呢？意思是合并之后的集合，也是满足天际线的，如下面合并的过程：

更多硬核内容就在公众号「yes 的练级攻略」

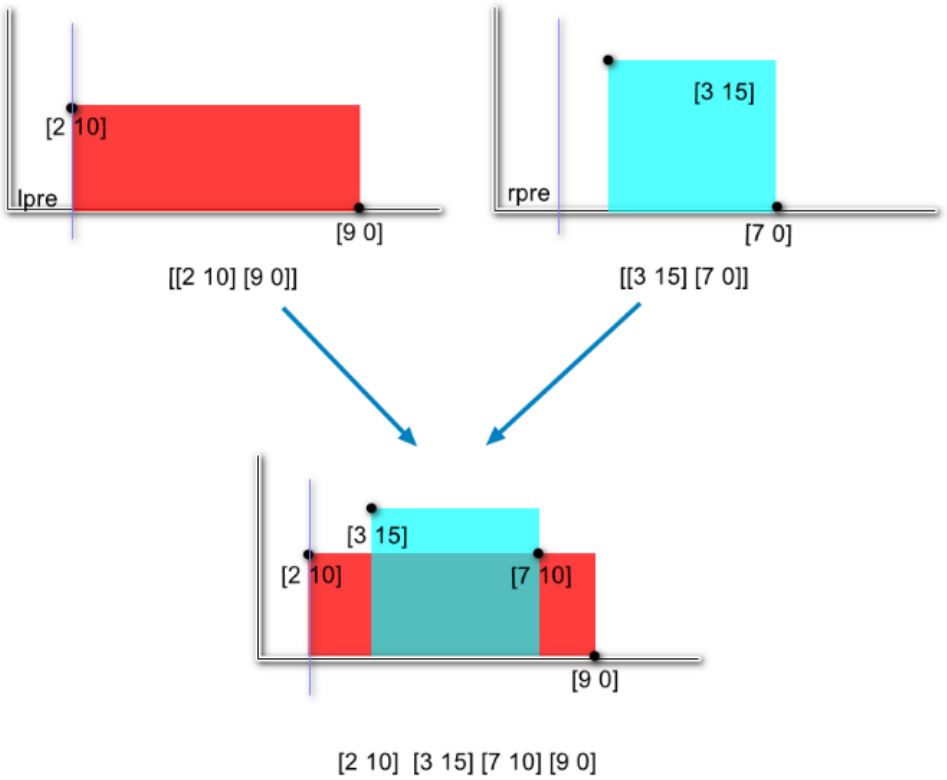


公众号：
yes的练级攻略

其实我们在题目标签看到了Line Sweep，[线扫描或扫描线]，扫描线可以想象成一条向右扫过平面的竖直线，也是一个算法，一般是玩图形学的。

接着上面的步骤，可以通过扫描线算法将两个关键点集合进行合并。

如下图，扫描线从两个集合的起始点，同时向右移动，接触到第一个关键点，则判断这一个关键点是不是满足天际线的，如果是，则将这个关键点添加到"父"集合中；如果不是，则继续同时移动到下一个关键点。



但如何判断是否是属于"父"集合中的关键点呢？可以创建两个集合（"子"）的目前高度，然后多方角度找到满足关键点的条件。

扫描线移到[2 10]关键点时，10要大于rpre的，可以满足；

扫描线移到[3 15]关键点时，lpre此时目前的高度为10，而15要大于10的，可以满足；

扫描线移到[7 10]关键点时，rpre大于lpre可以满足，反之就不满足；

接着有一个集合已经遍历完了，剩下的集合的关键点肯定是满足的，因为没有其它的集合可以阻挡到这个集合，所以直接就是满足。

```
import java.util.*;
class Solution {
    // 线段树
    public List<List<Integer>> getSkyline(int[][] buildings) {
        int len = buildings.length;
        if (len == 0) return new ArrayList<>();
        return segment(buildings, 0, len - 1);
    }
    private List<List<Integer>> segment(int[][] buildings, int l, int r) {
        // 创建返回值
```

```
        List<List<Integer>> res = new ArrayList<>();
        // 找到树底下的结束条件 -> 一个建筑物
        if (l == r) {
            res.add(Arrays.asList(buildings[l][0], buildings[l][2])); // 左上端
坐标
            res.add(Arrays.asList(buildings[l][1], 0)); // 右下端坐标
            return res;
        }
        int mid = l + (r - l) / 2; // 取中间值
        // 左边递归
        List<List<Integer>> left = segment(buildings, l, mid);
        // 右边递归
        List<List<Integer>> right = segment(buildings, mid + 1, r);
        // 左右合并
        // 创建 left 和 right 的索引位置
        int m = 0, n = 0;
        // 创建 left 和 right 目前的高度
        int lpreH = 0, rpreH = 0;
        // 两个坐标
        int leftX, leftY, rightX, rightY;
        while (m < left.size() || n < right.size()) {
            // 当有一边完全加入到 res 时，则加入剩余的那部分
            if (m >= left.size()) res.add(right.get(n++));
            else if (n >= right.size()) res.add(left.get(m++));
            else { // 开始判断 left 和 right
                leftX = left.get(m).get(0); // 不会出现 null，可以直接用 int 类型
                leftY = left.get(m).get(1);
                rightX = right.get(n).get(0);
                rightY = right.get(n).get(1);
                if (leftX < rightX) {
                  if (leftY > rpreH) res.add(left.get(m));
                  else if (lpreH > rpreH) res.add(Arrays.asList(leftX, rpreH));
                   lpreH = leftY;
                   m++;
                } else if (leftX > rightX) {
                  if (rightY > lpreH) res.add(right.get(n));
                 else if (rpreH > lpreH) res.add(Arrays.asList(rightX, lpreH));
                    rpreH = rightY;
                    n++;
                } else { // left 和 right 的横坐标相等
                    if (leftY >= rightY && leftY != (lpreH > rpreH ? lpreH : rp
reH))
                        res.add(left.get(m));
```

```
                    else if (leftY <= rightY && rightY != (lpreH > rpreH ? lpre
H : rpreH))
                        res.add(right.get(n));
                    lpreH = leftY;
                    rpreH = rightY;
                    m++;
                    n++;
                }
            }
        }
        return res;
    }
}
```

十八、栈

1、用栈实现队列（232）【用两个栈实现队列（剑指 9）】【化栈为队（面试题 03.04）】

232. 用栈实现队列

难度 简单 210 收藏 分享 切换为英文 关注 反馈

使用栈实现队列的下列操作：

- push(x) -- 将一个元素放入队列的尾部。
- pop() -- 从队列首部移除元素。
- peek() -- 返回队列首部的元素。
- empty() -- 返回队列是否为空。

示例:

```
MyQueue queue = new MyQueue();

queue.push(1);
queue.push(2);
queue.peek();  // 返回 1
queue.pop();   // 返回 1
queue.empty(); // 返回 false
```

说明:

- 你只能使用标准的栈操作 -- 也就是只有 `push to top`, `peek/pop from top`, `size`, 和 `is empty` 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。
- 假设所有操作都是有效的 （例如，一个空的队列不会调用 pop 或者 peek 操作）。

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 09. 用两个栈实现队列

难度 简单 102 收藏 分享 切换为英文 关注 反馈

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 `appendTail` 和 `deleteHead` ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素， `deleteHead` 操作返回 -1)

示例 1：

```
输入：
["CQueue","appendTail","deleteHead","deleteHead"]
[[],[3],[],[]]
输出：[null,null,3,-1]
```

示例 2：

```
输入：
["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"]
[[],[],[5],[2],[],[]]
输出：[null,-1,null,null,5,2]
```

提示：

- `1 <= values <= 10000`
- 最多会对 `appendTail、deleteHead` 进行 `10000` 次调用

https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/solution/mian-shi-ti-09-yong-liang-ge-zhan-shi-xian-dui-l-3/

方法一：双栈

思路和算法

维护两个栈，第一个栈支持插入操作，第二个栈支持删除操作。

根据栈先进后出的特性，我们每次往第一个栈里插入元素后，第一个栈的底部元素是最后插入的元素，第一个栈的顶部元素是下一个待删除的元素。为了维护队列先进先出的特性，我们引入第二个栈，用第二个栈维护待删除的元素，在执行删除操作的时候我们首先看下第二个栈是否为空。如果为空，我们将第一个栈里的元素一个个弹出插入到第二个栈里，这样第二个栈里元素的顺序就是待删除的元素的顺序，要执行删除操作的时候我们直接弹出第二个栈的元素返回即可。

成员变量

- 维护两个栈 `stack1` 和 `stack2` ，其中 `stack1` 支持插入操作， `stack2` 支持删除操作

构造方法

- 初始化 `stack1` 和 `stack2` 为空

插入元素

插入元素对应方法 `appendTail`

- `stack1` 直接插入元素

删除元素

删除元素对应方法 `deleteHead`

- 如果 `stack2` 为空，则将 `stack1` 里的所有元素弹出插入到 `stack2` 里
- 如果 `stack2` 仍为空，则返回 `-1` ，否则从 `stack2` 弹出一个元素并返回

复杂度分析

- 时间复杂度：对于插入和删除操作，时间复杂度均为 $O(1)$。插入不多说，对于删除操作，虽然看起来是 $O(n)$ 的时间复杂度，但是仔细考虑下每个元素只会「至多被插入和弹出 `stack2` 一次」，因此均摊下来每个元素被删除的时间复杂度仍为 $O(1)$。
- 空间复杂度：$O(n)$。需要使用两个栈存储已有的元素。

232 可运行版本

```
import java.util.Stack;
public class MyQueue {
    Stack<Integer> s1, s2;
    /** Initialize your data structure here. */
    public MyQueue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }
    public void push(int x) {
        s1.push(x);
    }
    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        if (s2.isEmpty()) {
            while (!s1.isEmpty()){
                s2.push(s1.pop());
            }
        }
        if (!s2.isEmpty()) return s2.pop();
        throw new RuntimeException("There are no elements in the queue!");
    }
    /** Get the front element. */
    public int peek() {
        if (s2.isEmpty()) {
            while (!s1.isEmpty()){
                s2.push(s1.pop());
            }
        }
        if (!s2.isEmpty()) return s2.peek();
        throw new RuntimeException("There are no elements in the queue!");
    }
    /** Returns whether the queue is empty. */
    public boolean empty() {
        return s1.isEmpty() && s2.isEmpty();
    }
    public static void main(String[] args){
        MyQueue queue = new MyQueue();
        queue.push(1);
        queue.push(2);
        System.out.println(queue.peek());  // 返回 1
        System.out.println(queue.pop());   // 返回 1
        System.out.println(queue.empty()); // 返回 false
```

```
    }
}
```

剑指 9 版本

```
class CQueue {
    Deque<Integer> stack1;
    Deque<Integer> stack2;
    public CQueue() {
        stack1 = new LinkedList<Integer>();
        stack2 = new LinkedList<Integer>();
    }
    public void appendTail(int value) {
        stack1.push(value);
    }
    public int deleteHead() {
        // 如果第二个栈为空
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        if (stack2.isEmpty()) return -1;
        else {
            int deleteItem = stack2.pop();
            return deleteItem;
        }
    }
}
/**
 * Your CQueue object will be instantiated and called as such:
 * CQueue obj = new CQueue();
 * obj.appendTail(value);
 * int param_2 = obj.deleteHead();
 */
```

2、用队列实现栈（225）

225. 用队列实现栈

难度 简单 209 收藏 分享 切换为英文 关注 反馈

使用队列实现栈的下列操作：

- push(x) -- 元素 x 入栈
- pop() -- 移除栈顶元素
- top() -- 获取栈顶元素
- empty() -- 返回栈是否为空

注意：

- 你只能使用队列的基本操作-- 也就是 `push to back`, `peek/pop from front`, `size`, 和 `is empty` 这些操作是合法的。
- 你所使用的语言也许不支持队列。 你可以使用 list 或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。
- 你可以假设所有操作都是有效的（例如, 对一个空的栈不会调用 pop 或者 top 操作）。

https://leetcode-cn.com/problems/implement-stack-using-queues/solution/liang-ge-dui-lie-shi-xian-zhan-by-georgedc/

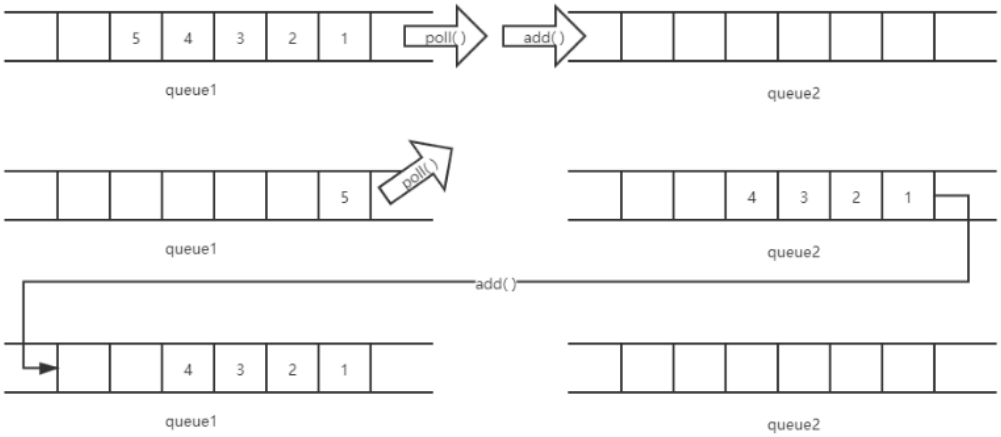
解题思路

队列（FIFO）先进先出
栈（LIFO）后进先出

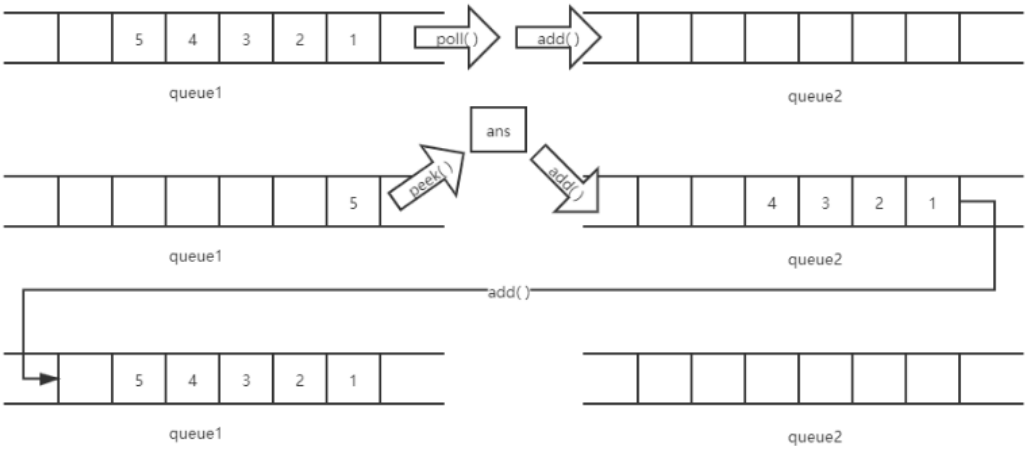
队列(queue)的poll()操作弹出队尾元素，与栈的pop()操作类似

执行push()操作时，将所有元素都加入到queue1中

执行pop()操作时，将queue1中的n-1个元素(n=queue1.size())poll()到queue2中,此时poll()queue1中的元素实现pop()操作，操作结束后再将queue2中的元素重新添加到queue1中



执行top()操作与pop()操作类似，记录下queue1中的最后一个元素再将这个元素poll()到queue2中，同样在结束后将queue2中的元素重新添加到queue1中



empty()操作只需判断queue1是否为空

```
class MyStack {
    private Queue<Integer> queue1;
    private Queue<Integer> queue2;
    /** Initialize your data structure here. */
    public MyStack() {
        queue1 = new LinkedList<>();
```

```
            queue2 = new LinkedList<>();
        }
        /** Push element x onto stack. */
        public void push(int x) {
            queue1.add(x);
        }
        /** Removes the element on top of the stack and returns that element. */
        public int pop() {
            while (queue1.size() > 1)
                queue2.add(queue1.poll());
            int ans = queue1.poll();
            while (queue2.size() > 0)
                queue1.add(queue2.poll());
            return ans;
        }
        /** Get the top element. */
        public int top() {
            while (queue1.size() > 1)
                queue2.add(queue1.poll());
            int ans = queue1.peek();
            queue2.add(queue1.poll());
            while (queue2.size() > 0)
                queue1.add(queue2.poll());
            return ans;
        }
        /** Returns whether the stack is empty. */
        public boolean empty() {
            return queue1.isEmpty();
        }
}
/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * boolean param_4 = obj.empty();
 */
```

3、动物收容所（面试题 03.06）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 03.06. 动物收容所

难度 简单 8 收藏 分享 切换为英文 关注 反馈

动物收容所。有家动物收容所只收容狗与猫，且严格遵守"先进先出"的原则。在收养该收容所的动物时，收养人只能收养所有动物中"最老"（由其进入收容所的时间长短而定）的动物，或者可以挑选猫或狗（同时必须收养此类动物中"最老"的）。换言之，收养人不能自由挑选想收养的对象。请创建适用于这个系统的数据结构，实现各种操作方法，比如 `enqueue`、`dequeueAny`、`dequeueDog` 和 `dequeueCat`。允许使用Java内置的LinkedList数据结构。

`enqueue` 方法有一个 `animal` 参数，`animal[0]` 代表动物编号，`animal[1]` 代表动物种类，其中 0 代表猫，1 代表狗。

`dequeue*` 方法返回一个列表 `[动物编号, 动物种类]`，若没有可以收养的动物，则返回 `[-1,-1]`。

示例1:

```
 输入：
["AnimalShelf", "enqueue", "enqueue", "dequeueCat", "dequeueDog", "dequeueAny"]
[[], [[0, 0]], [[1, 0]], [], [], []]
 输出：
[null,null,null,[0,0],[-1,-1],[1,0]]
```

示例2:

```
 输入：
["AnimalShelf", "enqueue", "enqueue", "enqueue", "dequeueDog", "dequeueCat", "dequeueAny"]
[[], [[0, 0]], [[1, 0]], [[2, 1]], [], [], []]
 输出：
[null,null,null,null,[2,1],[0,0],[1,0]]
```

说明:

1. 收纳所的最大容量为20000

https://leetcode-cn.com/problems/animal-shelter-lcci/solution/javayong-1ge-queueshi-xian-dong-wu-shou-rong-suo-b/

解题思路

用1个队列存放猫和狗，收养任意动物时先判空再返回，收养指定动物则遍历找到第一个出现的指定动物，从队列中删除后返回该动物

```
class AnimalShelf {
    Queue<int[]> queue;
    public AnimalShelf() {
        queue = new LinkedList<>();
    }
    public void enqueue(int[] animal) {
        queue.offer(animal);
    }
    public int[] dequeueAny() {
        if(queue.isEmpty())
            return new int[]{-1,-1};
        return queue.poll();
    }
    public int[] dequeueDog() {
```

```
        for(int[] animal : queue)
            if(animal[1] == 1){
                queue.remove(animal);
                return animal;
            }
        return new int[]{-1, -1};
    }
    public int[] dequeueCat() {
        for(int[] animal : queue)
            if(animal[1] == 0){
                queue.remove(animal);
                return animal;
            }
        return new int[]{-1, -1};
    }
}
/**
 * Your AnimalShelf object will be instantiated and called as such:
 * AnimalShelf obj = new AnimalShelf();
 * obj.enqueue(animal);
 * int[] param_2 = obj.dequeueAny();
 * int[] param_3 = obj.dequeueDog();
 * int[] param_4 = obj.dequeueCat();
 */
```

4、最小栈（155）【包含 min 函数的栈（剑指 30）】【栈的最小值（面试题 03.02）】

yes的练级攻略

155. 最小栈

难度 简单 566 收藏 分享 切换为英文 关注 反馈

设计一个支持 `push` ， `pop` ， `top` 操作，并能在常数时间内检索到最小元素的栈。

- `push(x)` —— 将元素 x 推入栈中。
- `pop()` —— 删除栈顶的元素。
- `top()` —— 获取栈顶元素。
- `getMin()` —— 检索栈中的最小元素。

示例:

输入：
```
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]
```

输出：
```
[null,null,null,null,-3,null,0,-2]
```

解释：
```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

提示:

- `pop` 、 `top` 和 `getMin` 操作总是在 **非空栈** 上调用。

更多硬核内容就在公众号「yes 的练级攻略」

面试题 03.02. 栈的最小值

难度 简单 27 收藏 分享 切换为英文 关注 反馈

请设计一个栈，除了常规栈支持的pop与push函数以外，还支持min函数，该函数返回栈元素中的最小值。执行push、pop和min操作的时间复杂度必须为O(1)。

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

甜姨总结：

https://leetcode-cn.com/problems/min-stack/solution/3-chong-fang-fa-shi-xian-bi-xu-miao-dong-by-sweeti/

思路 1，用辅助栈（7ms）

首先定义一个 「数据栈」来支持常规的 push、pop、top 操作；

再定义一个「辅助栈」，其栈顶一直保持为当前的最小值，以支持常数时间复杂度的 getMin 操作。

```
class MinStack {
    private Stack<Integer> dataStack;
    private Stack<Integer> minStack;
    /** initialize your data structure here. */
    public MinStack() {
        dataStack = new Stack<>();
        minStack = new Stack<>();
    }
    public void push(int x) {
        dataStack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek() ) {
            minStack.push(x);
        }
    }
    public void pop() {
        int x = dataStack.pop();
        if (x == minStack.peek()) {
            minStack.pop();
        }
    }
    public int top() {
        return dataStack.peek();
```

```
    }
    public int getMin() {
        return minStack.peek();
    }
}
/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.getMin();
 */
```

思路 2：不利用辅助栈，自建 Node（5ms）

```
class MinStack {
    private Node head;
    public MinStack() {
    }
    public void push(int x) {
        // 新建节点插入链表头部，作为新的头节点，存储当前的元素值和最小值，并且指向之前的头节点。
        if (head == null) {
            head = new Node(x, x);
        } else {
            head = new Node(x, Math.min(x, head.min), head);
        }
    }
    public void pop() {
        // 删除链表头节点
        head = head.next;
    }
    public int top() {
        // 返回头节点中存储的元素值
        return head.val;
    }
    public int getMin() {
        // 返回头节点中存储的最小值
        return head.min;
    }
    private static class Node {
        int val; // 元素值
        int min; // 最小值
        Node next; // 后继节点
        public Node(int val, int min) {
```

```
            this(val, min, null);
        }
        public Node(int val, int min, Node next) {
            this.val = val;
            this.min = min;
            this.next = next;
        }
    }
}
```

5、最大栈（716）

716. 最大栈

难度 简单 37 收藏 分享 切换为英文 关注 反馈

设计一个最大栈，支持 push、pop、top、peekMax 和 popMax 操作。

1. push(x) -- 将元素 x 压入栈中。
2. pop() -- 移除栈顶元素并返回这个值。
3. top() -- 返回栈顶元素。
4. peekMax() -- 返回栈中最大元素。
5. popMax() -- 返回栈中最大的元素，并将其删除。如果有多个最大元素，只要删除最靠近栈顶的那个。

样例 1:

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

注释:

1. -1e7 <= x <= 1e7
2. 操作次数不会超过 10000。
3. 当栈为空的时候不会出现后四个操作。

https://leetcode-cn.com/problems/max-stack/solution/max-stack-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：双向链表 + 平衡树

分析

使用线性的数据结构（例如数组和栈）无法在较短的时间复杂度内完成 `popMax()` 操作，因此我们可以考虑使用双向链表 + 平衡树，其中双向链表用来表示栈，平衡树中的每一个节点存储一个键值对，其中"键"表示某个在栈中出现的值，"值"为一个列表，表示这个值在双向链表中出现的位置，存储方式为指针。平衡树的插入，删除和查找操作都是 $O(\log n)$ 的，而通过平衡树定位到双向链表中的某个节点后，对该节点进行删除也是 $O(1)$ 的，因此所有操作的时间复杂度都不会超过 $O(\log n)$。

算法

我们使用双向链表存储栈，使用带键值对的平衡树（Java 中的 `TreeMap`）存储栈中出现的值以及这个值在双向链表中出现的位置。

- `push(x)` 操作：在双向链表的末尾添加一个节点，并且在平衡树上找到 x，给它的列表中添加一个位置，指向新的节点。
- `pop()` 操作：在双向链表的末尾删除一个节点，它的值为 val，随后在平衡树上找到 val，删除它的列表中的最后一个位置。
- `top()` 操作：返回双向链表中最后一个节点的值。
- `peekMax()` 操作：返回平衡树上的最大值。
- `popMax()` 操作：在平衡树上找到最大值和它对应的列表，得到列表中的最后一个位置，并将它在双向链表中和平衡树上分别删除。

复杂度分析

- 时间复杂度：$O(\log n)$。`top()` 操作的时间复杂度为 $O(1)$，其余操作的时间复杂度为 $O(\log n)$，因此总的时间复杂度为 $O(\log n)$。
- 空间复杂度：$O(n)$。

```
class MaxStack {
    TreeMap<Integer, List<Node>> map;
    DoubleLinkedList dll;
    public MaxStack() {
        map = new TreeMap();
        dll = new DoubleLinkedList();
    }
    public void push(int x) {
        Node node = dll.add(x);
        if(!map.containsKey(x))
            map.put(x, new ArrayList<Node>());
        map.get(x).add(node);
    }
    public int pop() {
        int val = dll.pop();
        List<Node> L = map.get(val);
        L.remove(L.size() - 1);
        if (L.isEmpty()) map.remove(val);
        return val;
    }
```

```
    public int top() {
        return dll.peek();
    }
    public int peekMax() {
        return map.lastKey();
    }
    public int popMax() {
        int max = peekMax();
        List<Node> L = map.get(max);
        Node node = L.remove(L.size() - 1);
        dll.unlink(node);
        if (L.isEmpty()) map.remove(max);
        return max;
    }
}
class DoubleLinkedList {
    Node head, tail;
    public DoubleLinkedList() {
        head = new Node(0);
        tail = new Node(0);
        head.next = tail;
        tail.prev = head;
    }
    public Node add(int val) {
        Node x = new Node(val);
        x.next = tail;
        x.prev = tail.prev;
        tail.prev = tail.prev.next = x;
        return x;
    }
    public int pop() {
        return unlink(tail.prev).val;
    }
    public int peek() {
        return tail.prev.val;
    }
    public Node unlink(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        return node;
    }
}
class Node {
    int val;
```

```
    Node prev, next;
    public Node(int v) {val = v;}
}
/**
 * Your MaxStack object will be instantiated and called as such:
 * MaxStack obj = new MaxStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.peekMax();
 * int param_5 = obj.popMax();
 */
```

6、扁平化嵌套列表迭代器（341）

341. 扁平化嵌套列表迭代器

难度 中等 139 收藏 分享 切换为英文 关注 反馈

给你一个嵌套的整型列表。请你设计一个迭代器，使其能够遍历这个整型列表中的所有整数。

列表中的每一项或者为一个整数，或者是另一个列表。其中列表的元素也可能是整数或是其他列表。

示例 1:

输入: [[1,1],2,[1,1]]
输出: [1,1,2,1,1]
解释: 通过重复调用 *next* 直到 *hasNext* 返回 false，*next* 返回的元素的顺序应该是: [1,1,2,1,1]。

示例 2:

输入: [1,[4,[6]]]
输出: [1,4,6]
解释: 通过重复调用 *next* 直到 *hasNext* 返回 false，*next* 返回的元素的顺序应该是: [1,4,6]。

https://leetcode-cn.com/problems/flatten-nested-list-iterator/solution/javazhan-die-dai-qi-by-ffo rkboat/

```
/**
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * public interface NestedInteger {
 *
 *     // @return true if this NestedInteger holds a single integer, rather tha
n a nested list.
 *     public boolean isInteger();
 *
 *     // @return the single integer that this NestedInteger holds, if it holds
 a single integer
 *     // Return null if this NestedInteger holds a nested list
```

```
 *     public Integer getInteger();
 *
 *     // @return the nested list that this NestedInteger holds, if it holds a
nested list
 *     // Return null if this NestedInteger holds a single integer
 *     public List<NestedInteger> getList();
 * }
 */
public class NestedIterator implements Iterator<Integer> {
    // 使用栈来保存List<NestedInteger>的迭代器ListIterator<NestedInteger>
    private LinkedList<ListIterator<NestedInteger>> stack = new LinkedList<>();

    public NestedIterator(List<NestedInteger> nestedList) {
        stack.push(nestedList.listIterator());
    }
    @Override
    public Integer next() {
        return stack.peek().next().getInteger();
    }
    @Override
    public boolean hasNext() {
        stopBeforeNextInt();
        return stack.size() != 0;
    }
    // 移动栈顶的迭代器，使得它的下一个元素是数字，移动的过程中遇到新的
List<NestedInteger>的话，就把它的迭代器入栈
    private void stopBeforeNextInt() {
        out:while (stack.size() != 0) {
            ListIterator<NestedInteger> iter = stack.peek();
            if (!iter.hasNext()) {
                stack.pop();
            }
            while (iter.hasNext()) {
                NestedInteger ni = iter.next();
                // 找到了数字，将迭代器回退一步，直接break 外层循环
                if (ni.isInteger()) {
                    iter.previous();
                    break out;
                }
                // 找到了新的List<NestedInteger>，将它入栈，先处理它，所以需要break
内存循环
                else {
                    stack.push(ni.getList().listIterator());
                    break;
```

```
                }
            }
        }
    }
}
/**
 * Your NestedIterator object will be instantiated and called as such:
 * NestedIterator i = new NestedIterator(nestedList);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

7、验证栈序列（946）【栈的压入、弹出序列（剑指 31）】

946. 验证栈序列

难度 中等 112 收藏 分享 切换为英文 关注 反馈

给定 pushed 和 popped 两个序列，每个序列中的 **值都不重复**，只有当它们可能是在最初空栈上进行的推入 push 和弹出 pop 操作序列的结果时，返回 true；否则，返回 false 。

示例 1：

输入： pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出： true
解释： 我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1

示例 2：

输入： pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出： false
解释： 1 不能在 2 之前弹出。

提示：

1. 0 <= pushed.length == popped.length <= 1000
2. 0 <= pushed[i], popped[i] < 1000
3. pushed 是 popped 的排列。

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 31. 栈的压入、弹出序列

难度 中等 62 收藏 分享 切换为英文 关注 反馈

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列 {1,2,3,4,5} 是某栈的压栈序列，序列 {4,5,3,2,1} 是该压栈序列对应的一个弹出序列，但 {4,3,5,1,2} 就不可能是该压栈序列的弹出序列。

示例 1：

```
输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

示例 2：

```
输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。
```

提示：

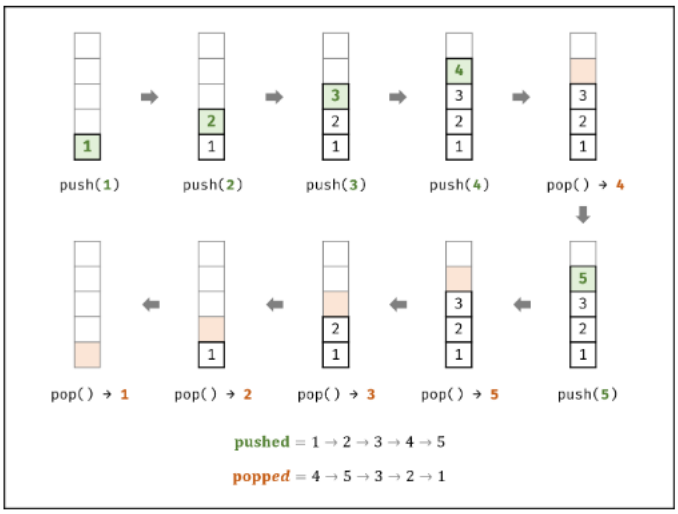
1. `0 <= pushed.length == popped.length <= 1000`
2. `0 <= pushed[i], popped[i] < 1000`
3. `pushed` 是 `popped` 的排列。

注意：本题与主站 946 题相同：https://leetcode-cn.com/problems/validate-stack-sequences/

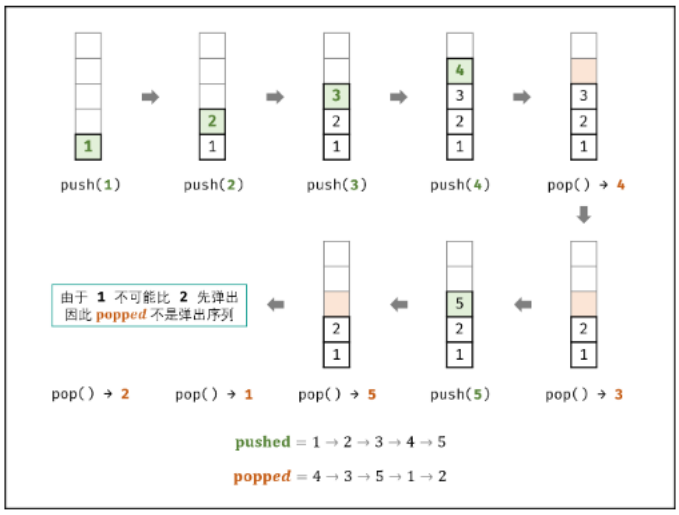
https://leetcode-cn.com/problems/zhan-de-ya-ru-dan-chu-xu-lie-lcof/solution/mian-shi-ti-31-zhan-de-ya-ru-dan-chu-xu-lie-mo-n-2/

解题思路：

如下图所示，给定一个压入序列 *pushed* 和弹出序列 *popped* ，则压入 / 弹出操作的顺序（即排列）是 **唯一确定** 的。



如下图所示，栈的数据操作具有 **先入后出** 的特性，因此某些弹出序列是无法实现的。



考虑借用一个辅助栈 $stack$ ，**模拟** 压入 / 弹出操作的排列。根据是否模拟成功，即可得到结果。

- **入栈操作：** 按照压栈序列的顺序执行。
- **出栈操作：** 每次入栈后，循环判断 “栈顶元素 = 弹出序列的当前元素” 是否成立，将符合弹出序列顺序的栈顶元素全部弹出。

> 由于题目规定 `栈的所有数字均不相等` ，因此在循环入栈中，每个元素出栈的位置的可能性是唯一的（若有重复数字，则具有多个可出栈的位置）。因而，在遇到 “栈顶元素 = 弹出序列的当前元素” 就应立即执行出栈。

算法流程：

1. **初始化：** 辅助栈 $stack$ ，弹出序列的索引 i ；
2. **遍历压栈序列：** 各元素记为 num ；
    1. 元素 num 入栈；
    2. 循环出栈：若 $stack$ 的栈顶元素 = 弹出序列元素 $popped[i]$ ，则执行出栈与 $i++$ ；
3. **返回值：** 若 $stack$ 为空，则此弹出序列合法。

复杂度分析：

- **时间复杂度 $O(N)$ ：** 其中 N 为列表 $pushed$ 的长度；每个元素最多入栈与出栈一次，即最多共 $2N$ 次出入栈操作。
- **空间复杂度 $O(N)$ ：** 辅助栈 $stack$ 最多同时存储 N 个元素。

```
class Solution {
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        Stack<Integer> sta = new Stack<>();
        int i = 0;
        for (int num : pushed) {
            sta.push(num);
            while (!sta.isEmpty() && sta.peek() == popped[i]){
                sta.pop();
                i++;
            }
        }
        return sta.isEmpty();
    }
}
```

8、栈排序（面试题 03.05）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 03.05. 栈排序

难度 中等 21 收藏 分享 切换为英文 关注 反馈

栈排序。 编写程序，对栈进行排序使最小元素位于栈顶。最多只能使用一个其他的临时栈存放数据，但不得将元素复制到别的数据结构（如数组）中。该栈支持如下操作：`push`、`pop`、`peek` 和 `isEmpty`。当栈为空时，`peek` 返回 -1。

示例1:

```
 输入:
["SortedStack", "push", "push", "peek", "pop", "peek"]
[[], [1], [2], [], [], []]
 输出:
[null,null,null,1,null,2]
```

示例2:

```
 输入:
["SortedStack", "pop", "pop", "push", "pop", "isEmpty"]
[[], [], [], [1], [], []]
 输出:
[null,null,null,null,null,true]
```

说明:

1. 栈中的元素数目在[0, 5000]范围内。

禁止转用

https://leetcode-cn.com/problems/sort-of-stacks-lcci/solution/fei-you-hua-yu-you-hua-de-qu-bie-by-sa1024/

非优化思路：
采用直接插入的思想，假设主栈已经有序，那么push一个元素，需要插到对应的位置上，
1.这个位置上面的所有元素都暂时放到一个辅助栈里面
2.将该元素插入正确位置
3.将之前的元素在移动回来即可。
优化思路：
优化：既然有辅助栈，那么充分利用辅助栈，考虑到主栈栈顶元素>=辅助栈的栈顶元素，如果处在中间的元素，我们直接放入辅助栈中，而不用来回倒了
逻辑：
1.设新加入的元素为x
2. x与主栈栈顶y、辅栈栈顶元素z比较只可能出现三种情况，x>=y>=z,y>=x>=z,y>=z>=x,根据这三种情况分别讨论即可
3. 如果主栈空，x加入主栈
4.如果主栈不空并且x大于栈顶的元素，那么需要将主栈中所有小于x的元素送到辅栈,将x加入主栈
5.如果主栈不空并且x小于栈顶的元素，那么就需要将x与辅栈作比较
6.如果辅栈为空或者辅栈的栈顶元素小于等于x，那么直接加入辅栈
7.如果辅栈不空并且辅栈栈顶元素大于x，那么就需要将辅栈中所有大于x的元素移到主栈，将x加入辅栈
8.pop和peek时候需要将辅栈中的所有元素移到主栈

```
class SortedStack {
    Stack<Integer> stack;
    Stack<Integer> helpStack;
    public SortedStack() {
        stack = new Stack<>();
        helpStack = new Stack<>();
    }
    public void push(int val) {
        if (stack.empty()) {
            stack.push(val);
        } else if (stack.peek() < val) {
```

```
            while (!stack.empty() && stack.peek() < val) {
                helpStack.push(stack.pop());
            }
            stack.push(val);
        } else if (helpStack.empty() || helpStack.peek() <= val) {
            helpStack.push(val);
        }else{
            while (!helpStack.empty() && helpStack.peek() > val) {
                stack.push(helpStack.pop());
            }
            helpStack.push(val);
        }
    }
    public void pop() {
        while (!helpStack.empty()) {
            stack.push(helpStack.pop());
        }
        if (!isEmpty()) {
            stack.pop();
        }
    }
    public int peek() {
        while (!helpStack.empty()) {
            stack.push(helpStack.pop());
        }
        if (isEmpty()) {
            return -1;
        }
        return stack.peek();
    }
    public boolean isEmpty() {
        return stack.isEmpty();
    }
}
/**
 * Your SortedStack object will be instantiated and called as such:
 * SortedStack obj = new SortedStack();
 * obj.push(val);
 * obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.isEmpty();
 */
```

9、堆盘子（面试题 03.03）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 03.03. 堆盘子

难度 中等 6 收藏 分享 切换为英文 关注 反馈

堆盘子。设想有一堆盘子，堆太高可能会倒下来。因此，在现实生活中，盘子堆到一定高度时，我们就会另外堆一堆盘子。请实现数据结构 SetOfStacks，模拟这种行为。 SetOfStacks 应该由多个栈组成，并且在前一个栈填满时新建一个栈。此外，SetOfStacks.push() 和 SetOfStacks.pop() 应该与普通栈的操作方法相同（也就是说，pop()返回的值，应该跟只有一个栈时的情况一样）。 进阶：实现一个 popAt(int index) 方法，根据指定的子栈，执行pop操作。

当某个栈为空时，应当删除该栈。当栈中没有元素或不存在该栈时，pop，popAt 应返回 -1.

示例1:

```
 输入：
["StackOfPlates", "push", "push", "popAt", "pop", "pop"]
[[1], [1], [2], [1], [], []]
 输出：
[null, null, null, 2, 1, -1]
```

示例2:

```
 输入：
["StackOfPlates", "push", "push", "push", "popAt", "popAt", "popAt"]
[[2], [1], [2], [3], [0], [0], [0]]
 输出：
[null, null, null, null, 2, 1, 3]
```

https://leetcode-cn.com/problems/stack-of-plates-lcci/solution/zhi-xing-yong-shi-ji-bai-liao-8485-de-yong-hu-nei-/

```
class StackOfPlates {
    int maxcaps;//每个栈的最大cap 数
    LinkedList list = new LinkedList();//存储栈的linkedlist,因为linkedlist 增删快
    public StackOfPlates(int cap) {
        this.maxcaps = cap;
    }
    public void push(int val) {
        if(maxcaps<=0)
        return;
        if(list.isEmpty()){
            ArrayList arr = new ArrayList();//用来当做栈，存储元素的，arraylist 查询较快
            arr.add(val);
            list.add(arr);
        }
        else{
            ArrayList oldarr =(ArrayList) list.get(list.size()-1);
            if(oldarr.size()==maxcaps){
                ArrayList arr = new ArrayList();
                arr.add(val);
                list.add(arr);
            }
            else{
                oldarr.add(val);//最后一个没有满
```

```
            }
        }
    }
    public int pop() {
        if(list.size()==0)
        return -1;
        ArrayList oldarr =(ArrayList) list.get(list.size()-1);
        int re = (int)oldarr.get(oldarr.size()-1);
        oldarr.remove(oldarr.size()-1);//删掉 pop 的元素
        if(oldarr.size()==0){
            list.remove(oldarr);//删掉空的栈
        }
        return re;
    }
    public int popAt(int index) {
        if(list.size()==0 || index>list.size()-1)
        return -1;
        ArrayList oldarr =(ArrayList) list.get(index);
        int re = (int)oldarr.get(oldarr.size()-1);
        oldarr.remove(oldarr.size()-1);//删掉 pop 的元素
        if(oldarr.size()==0){
            list.remove(oldarr);//删掉空的栈
        }
        return re;

    }
}
/**
 * Your StackOfPlates object will be instantiated and called as such:
 * StackOfPlates obj = new StackOfPlates(cap);
 * obj.push(val);
 * int param_2 = obj.pop();
 * int param_3 = obj.popAt(index);
 */
```

10、三合一（面试题 03.01）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 03.01. 三合一

难度 简单 17 收藏 分享 切换为英文 关注 反馈

三合一。描述如何只用一个数组来实现三个栈。

你应该实现 push(stackNum, value) 、 pop(stackNum) 、 isEmpty(stackNum) 、 peek(stackNum) 方法。 stackNum 表示栈下标， value 表示压入的值。

构造函数会传入一个 stackSize 参数，代表每个栈的大小。

示例1:

```
 输入：
["TripleInOne", "push", "push", "pop", "pop", "pop", "isEmpty"]
[[1], [0, 1], [0, 2], [0], [0], [0], [0]]
 输出：
[null, null, null, 1, -1, -1, true]
说明：当栈为空时`pop, peek`返回-1，当栈满时`push`不压入元素。
```

示例2:

```
 输入：
["TripleInOne", "push", "push", "push", "pop", "pop", "pop", "peek"]
[[2], [0, 1], [0, 2], [0, 3], [0], [0], [0], [0]]
 输出：
[null, null, null, null, 2, 1, -1, -1]
```

https://leetcode-cn.com/problems/three-in-one-lcci/solution/zhu-shi-jie-fa-by-xing-kong-chuan-hen/

```
class TripleInOne {
    private int[] data; //真实的数据
    private int stackSize; //每个栈的最大数
    private int[] stackSizeArray; //每个栈内已经存放的数据个数
    public TripleInOne(int stackSize) {
        this.stackSize = stackSize;
        data = new int[stackSize * 3];
        stackSizeArray = new int[3];
    }
    public void push(int stackNum, int value) {
        //判断当前数组内是否已经放满
        if(stackSizeArray[stackNum] < stackSize){
            //设置对应位置的数字
            data[getStackStart(stackNum) + stackSizeArray[stackNum]] = value ;
            //对应栈的数量加上1
            stackSizeArray[stackNum] += 1;
        }
    }
    public int pop(int stackNum) {
        if(stackSizeArray[stackNum] <= 0){
            return -1;
        }else{
            //取开始位置加当前数量-1 即为要获取的数据
```

```
            int val = data[getStackStart(stackNum) + (stackSizeArray[stackNum]
- 1)];
            data[getStackStart(stackNum) + (stackSizeArray[stackNum] - 1)] = -1
 ;
            stackSizeArray[stackNum] -= 1;
            return val;
        }
    }

    public int peek(int stackNum) {
        if(stackSizeArray[stackNum] <= 0){
            return -1;
        }else{
            return  data[getStackStart(stackNum) + (stackSizeArray[stackNum] -
1)];
        }
    }
    public boolean isEmpty(int stackNum) {
        return stackSizeArray[stackNum] <= 0;
    }
    private int getStackStart(int stackNum){
        return  stackNum * stackSize ;
    }
}
/**
 * Your TripleInOne object will be instantiated and called as such:
 * TripleInOne obj = new TripleInOne(stackSize);
 * obj.push(stackNum,value);
 * int param_2 = obj.pop(stackNum);
 * int param_3 = obj.peek(stackNum);
 * boolean param_4 = obj.isEmpty(stackNum);
 */
```

11、最终优惠价（leedcode1852 题）【单调栈】

https://www.jiuzhang.com/solution/final-discounted-price/#tag-highlight-lang-java

更多硬核内容就在公众号「yes 的练级攻略」

1852. 最终优惠价

中文 English

一位店主需要完成一项销售任务，他将要出售的物品排成一排。
从左侧开始，店主以其全价减去位于该物品右侧的第一个价格较低或价格相同的商品的价格。
如果右侧没有价格低于或等于当前商品价格的商品，则以全价出售当前商品。
你需要返回每一个物品实际售出价格。

样例
示例 1:
输入:
Prices = [2, 3, 1, 2, 4, 2]
输出: [1, 2, 1, 0, 2, 2]
解释: 第0个和第1个物品右边第一个更低的价格都是1，所以实际售价需要在全价上减去1， 第3个物品右边第一个更低的价格是2，所以实际售价要在全价上面减去2。
示例 2:
输入:
Prices = [1, 2, 3, 4, 5]
输出: [1, 2, 3, 4, 5]
解释: 每一个物品都保持原价，他们的右边都没有等于或者更低价格的物品

注意事项
- 数组 Prices 的长度范围是: [1, 100000]
- Prices[i] 的整数范围是: [1, 1000000]

```
public class Solution {
    /**
     * @param prices: a list of integer
     * @return: return the actual prices
     */
    public int[] FinalDiscountedPrice(int[] prices) {
        // write your code here
        int[] res = new int[prices.length];
        Stack<Integer> s = new Stack<>();
        for (int i = 0; i < prices.length; i++) {
            res[i] = prices[i];
        }
        for (int i = 0; i < prices.length; i++) {
            while (! s.isEmpty() && prices[s.peek()] >= prices[i] ) {
                int index = s.pop();
                res[index] = prices[index] - prices[i];
            }
            s.push(i);
        }
        return res;
    }
}
```

单调栈

禁止商用 公众号：yes的练级攻略

题目存在两个“变量”

固定最小值，假设选 A[i] 为最小值的情况下，子数组应该从 A[i] 出发向两边延伸尽可能的多
因此想到找到 A[i] 左边第一个比他小的，找到 A[i] 右边第一个比他小的
这就是单调栈的典型应用场景

12、最大区间（lintcode1339）【单调栈】【区间中最小值*区间所有数的和】

描述 控制台 笔记

1339. 最大区间 中文 Engli

给定一个非负数的整数数组，需要选出一个区间，使得该区间是所有区间中值最大的一个，我们定义一个区间的值为：区间中的最小数 * 区间所有数的和。

样例
样例 1:

输入：[6, 2, 1]
输出：36
解释：所有区间及其对应的值为：

[6] = 6 * 6 = 36;

[2] = 2 * 2 = 4;

[1] = 1 * 1 = 1;

[6,2] = 2 * 8 = 16;

[2,1] = 1 * 3 = 3;

[6, 2, 1] = 1 * 9 = 9;

因此，最大的值为36。
挑战
最直观的算法复杂度是 $O(n^2)$，请在此基础上优化你的算法。

注意事项
数据范围：1 ≤ len(nums) ≤ 1e6; 0 ≤ nums[n] ≤ 100;

https://www.jiuzhang.com/solution/maximum-range/#tag-highlight-lang-java

张同学
更新于 2020/6/9 下午7:04:00

- 使用单调栈解决这个问题。维护单调栈（非递减栈） dataStack 和同步的前缀和栈 preSum 。
- 遍历 nums 数组， 如果 nums[i] 大于等于栈尾元素，那么 nums[i] 入 dataStack 栈， 它的前缀和 nums[i] 前面比它大的连续数组和 currSum 入 preSum 栈。 如果 nums[i] 小于栈尾元素，此时我们找一下以栈尾为最小值的区间的值，具体来说，令 leftmost 为 dataStack 中pop出的元素，同时令 currSum+= leftmost + 从 preSum中pop的元素 ，那么 res = max(res, currSum * leftmost); 即为该区间的值。
- 复杂度分析：时间复杂度为O(n)，n为数组长度。一次遍历，对 dataStack 来说，每个元素只会入栈和出栈一次，对 preSum 来说同理。空间复杂度为O(n)，最差情况下，栈会占用O(n)空间。

```
public class Solution {
    /**
     * @param nums:
     * @return: Find the maxmum range value. A range value is defined as the minimum value times the sum of all values in a range.
     */
    public int maxRange(int[] nums) {
        // write your code here
        Stack<Integer> dataStack = new Stack<>();
        Stack<Integer> preSum = new Stack<>();
        int currSum = 0, res = 0, i = 0;
```

```
            while (i < nums.length || !dataStack.isEmpty()) {
                if (i < nums.length && (dataStack.isEmpty() || nums[i] >= dataStack.peek())){
                    dataStack.push(nums[i]);
                    preSum.push(currSum);
                    currSum = 0;
                    i ++;
                }
                else {
                    int leftmost = dataStack.peek();
                    dataStack.pop();
                    currSum += (leftmost + preSum.peek());
                    preSum.pop();
                    res = Math.max(res, currSum * leftmost);
                }
            }
            return res;
        }
}
```

禁止商用

13、逛街（腾讯 2020 校园招聘-后台第 2 题）【单调栈】

小Q在周末的时候和他的小伙伴来到大城市逛街，一条步行街上有很多高楼，共有n座高楼排成一行。
小Q从第一栋一直走到了最后一栋，小Q从来都没有见到这么多的楼，所以他想知道他在每栋楼的位置处能看到多少栋楼呢？（当前面的楼的高度大于等于后面的楼时，后面的楼将被挡住）

输入描述:

输入第一行将包含一个数字n，代表楼的栋数，接下来的一行将包含n个数字w_i(1<=i<=n)，代表每一栋楼的高度。
1<=n<=100000;
1<=w_i<=100000;

输出描述:

输出一行，包含空格分割的n个数字v_i，分别代表小Q在第i栋楼时能看到的楼的数量。

输入例子1:

```
6
5 3 8 3 2 5
```

输出例子1:

```
3 3 5 4 4 4
```

例子说明1:

当小Q处于位置3时，他可以向前看到位置2,1处的楼，向后看到位置4,6处的楼，加上第3栋楼，共可看到5栋楼。当小Q处于位置4时，他可以向前看到位置3处的楼，向后看到位置5,6处的楼，加上第4栋楼，共可看到4栋楼。

```
import java.util.*;
public class Main{
    public static int[] count(int n, int[] w){
        int[] v = new int[n];
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        Stack<Integer> stack = new Stack<>();
        //for (int i = 0; i < n; i++){
        for (int i = n - 1; i >= 0; i--){
            v[i] = stack.size();
            while ((! stack.isEmpty()) && (w[i] >= w[stack.peek()])) {
                stack.pop();
            }
            stack.push(i);
        }
        stack.clear();
        //与原来相反的遍历，保持递减
        //for (int i = n - 1; i >= 0; i--){
        for (int i = 0; i < n; i++){
            v[i] = v[i] + 1 + stack.size();
            while ((! stack.isEmpty()) && (w[i] >= w[stack.peek()])) {
                stack.pop();
            }
            stack.push(i);
        }
        return v;
    }

    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] w = new int[n];
        for (int i = 0; i < n; i++){
            w[i] = sc.nextInt();
        }
        //Main m = new Main();
        //int[] v = m.count(n, w);
        int[] v = count(n, w); //把函数加上 static 以后可以直接调用
        for (int i = 0; i < n; i++){
            System.out.print(v[i] + " ");
        }
    }
}
```

14、柱状图中最大矩形（84）；直方图最大矩形覆盖（lintcode122）【单调栈】

禁止商用
公众号：
yes的练级攻略

122. 直方图最大矩形覆盖

给出的n个非负整数表示每个直方图的高度，每个直方图的宽均为1，在直方图中找到最大的矩形面积。

样例
样例 1:

输入: [2,1,5,6,2,3]
输出: 10
解释:
第三个和第四个直方图截取矩形面积为2*5=10。
样例 2:

输入: [1,1]
输出: 2
解释:
第一个和第二个直方图截取矩形面积为2*1=2。

https://www.jiuzhang.com/solution/largest-rectangle-in-histogram/

- 单调栈

题解： 用九章算法强化班中讲过的单调栈(stack)。维护一个单调递增栈，逐个将元素 push 到栈里。push 进去之前先把 >= 自己的元素 pop 出来。 每次从栈中 pop 出一个数的时候，就找到了往左数比它小的第一个数（当前栈顶）和往右数比它小的第一个数（即将入栈的数）， 从而可以计算出这两个数中间的部分宽度 * 被pop出的数，就是以这个被pop出来的数为最低的那个直方向两边展开的最大矩阵面积。 因为要计算两个数中间的宽度，因此放在 stack 里的是每个数的下标。

复杂度

- 假设数组大小为n;
- 空间复杂度O(n);
- 时间复杂度O(n)。

方法 1：九章

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        if (heights == null || heights.length == 0) return 0;
        Stack<Integer> sta = new Stack<>();
        int max = 0;
        for (int i = 0; i <= heights.length; i++){
            int top = (i == heights.length) ? -1 : heights[i];
            while (!sta.isEmpty() && top <= heights[sta.peek()]){
                int h = heights[sta.pop()];
                int w = sta.isEmpty() ? i : i - sta.peek() - 1;
                max = Math.max(max, h * w);
            }
            sta.push(i);
        }
        return max;
    }
}
```

https://leetcode-cn.com/problems/largest-rectangle-in-histogram/solution/xiang-jie-dan-diao-zhan-bi-xu-miao-dong-by-sweetie/

方法二：单调栈求解

以上暴力写法 Java 可以通过，但我们不妨想一下这里的双重循环是否可以优化？

我们每遍历到当前柱体 i 时：

- 上述写法中，我们需要再嵌套一层 while 循环来向左找到第一个比柱体 i 高度小的柱体，这个过程是 $O(N)$ 的；
- 那么我们可以 $O(1)$ 的获取柱体 i 左边第一个比它小的柱体吗？答案就是单调增栈，因为对于栈中的柱体来说，栈中下一个柱体就是左边第一个高度小于自身的柱体。

因此做法就很简单了，我们遍历每个柱体，若当前的柱体高度大于等于栈顶柱体的高度，就直接将当前柱体入栈，否则若当前的柱体高度小于栈顶柱体的高度，说明当前栈顶柱体找到了右边的第一个小于自身的柱体，那么就可以将栈顶柱体出栈来计算以其为高的矩形的面积了。

方法 2：甜姨

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        // 这里为了代码简便，在柱体数组的头和尾加了两个高度为 0 的柱体。
        int[] tmp = new int[heights.length + 2];
//public static native void arraycopy(Object src, int srcPos, Object dest, int destPos, int length);
        System.arraycopy(heights, 0, tmp, 1, heights.length);
        Deque<Integer> stack = new ArrayDeque<>();
        int area = 0;
        for (int i = 0; i < tmp.length; i++) {
            // 对栈中柱体来说，栈中的下一个柱体就是其「左边第一个小于自身的柱体」；
            // 若当前柱体 i 的高度小于栈顶柱体的高度，说明 i 是栈顶柱体的「右边第一个小于栈顶柱体的柱体」。
            // 因此以栈顶柱体为高的矩形的左右宽度边界就确定了，可以计算面积
            while (!stack.isEmpty() && tmp[i] < tmp[stack.peek()]) {
                int h = tmp[stack.pop()];
                area = Math.max(area, (i - 1 - stack.peek()) * h);
            }
            stack.push(i);
        }
        return area;
    }
}
```

15、最大矩形（85）（lintcode510）【单调栈+DP】

510. 最大矩形

给你一个二维矩阵，权值为 `False` 和 `True`，找到一个最大的矩形，使得里面的值全部为 `True`，输出它的面积

样例
样例1

输入：
```
[
  [1, 1, 0, 0, 1],
  [0, 1, 0, 0, 1],
  [0, 0, 1, 1, 1],
  [0, 0, 1, 1, 1],
  [0, 0, 0, 0, 1]
]
```
输出：6
样例2

输入：
```
[
    [0,0],
    [0,0]
]
```
输出：0

算法：单调栈+DP

这题和 122. 直方图最大矩形覆盖 很相似，只需要求出以每一行作为底最大的矩形是多少，每一行都有一个 `height` 数组，利用单调栈，每次更新 `height` 数组，`height` 数组代表的是这一列上面有多少个连续的 `1`，即矩形的高度，以每一行作为底(直方图最下面)时最大矩形面积，然后记录最大值即可。

- 初始化 `dp` 数组，用dp数组记录当前位置上方有多少个连续 `1`
- 对于每一行作为底，利用单调栈求高度，寻找最大的底乘高
  - 注意这个栈是 从栈底到栈顶依次是从小到大的
  - 如果栈中的数比当前的数大(或者等于)就要处理栈顶的(记录左右两边的比它小的第一个数)
  - 然后如果遍历完之后，单独处理栈，此时所有元素右边都不存在比它小的
  - `height[j]` 表示目前的底上(第 `i` 行)，`j` 位置往上(包括 `j` 位置)有多少连续的 `1`
- 不断更新最大面积

复杂度分析

- 时间复杂度 `O(m*n)`
  - dp的O(n*m)和单调栈的O(n)
- 空间复杂度 `O(n*m)`
  - dp的大小

Lintcode510 代码

```
public class Solution {
    /**
     * @param matrix: a boolean 2D matrix
     * @return: an integer
     */
    public int maximalRectangle(boolean[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
        int ans = 0;
        int row = matrix.length, col = matrix[0].length;
        int[][] dp = new int[row][col+1];
        for (int i = 0; i < row; i++) {//每个位置上方有多少连续的 1
            for (int j = 0; j < col; j++) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                if (i == 0 && matrix[i][j]) {
                    dp[i][j] = 1;
                    continue;
                }
                if (matrix[i][j]) {
                    dp[i][j] = dp[i - 1][j] + 1;
                }
            }
        }
        for (int i = 0; i < row; i++) { //把每一行作为底找最大矩形
            ans = Math.max(ans, largestRectangleArea(dp[i]));
        }
        return ans;
    }
    private int largestRectangleArea(int[] height) {
        int[] tmp = new int[height.length + 2];
        System.arraycopy(height, 0, tmp, 1, height.length);
        Deque<Integer> sta = new ArrayDeque<>();
        int area = 0;
        for (int i = 0; i < tmp.length; i++){
            while (!sta.isEmpty() && tmp[i] < tmp[sta.peek()]){
                int h = tmp[sta.pop()];
                area = Math.max(area, (i - sta.peek() - 1) * h);
            }
            sta.push(i);
        }
        return area;
    }
}
```

禁止商用
公众号：
yes的练级攻略

与在 leetcode 中的不同用加粗表示。以下是 leetcode85 代码

```
class Solution {
    public int maximalRectangle(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length ==
 0) return 0;
        int res = 0;
        int row = matrix.length, col = matrix[0].length;
        int[][] dp = new int[row][col+1];
        for (int i = 0; i < row; ++i){
            for (int j = 0; j < col; ++j){
                if (i == 0 && matrix[i][j] == '1'){
                    dp[i][j] = 1;
                    continue;
                }
                if (matrix[i][j] == '1'){
```

```
                    dp[i][j] = dp[i-1][j] + 1;
                }
            }
        }
        for (int i = 0; i < row; ++i){
            res = Math.max(res, largeArea(dp[i]));
        }
        return res;
    }
    public int largeArea(int[] height){
        int[] tmp = new int[height.length + 2];
        System.arraycopy(height, 0, tmp, 1, height.length);
        Deque<Integer> sta = new ArrayDeque<>();
        int area = 0;
        for (int i = 0; i < tmp.length; i++){
            while (!sta.isEmpty() && tmp[i] < tmp[sta.peek()]){
                int h = tmp[sta.pop()];
                area = Math.max(area, (i - sta.peek() - 1) * h);
            }
            sta.push(i);
        }
        return area;
    }
}
```

16、最大正方形（221）（lintcode436）【单调的+DP，上一题改造了两行】【只 DP】

221. 最大正方形

难度 中等 489 收藏 分享 切换为英文 关注 反馈

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

示例:

输入:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

输出: 4

https://leetcode-cn.com/problems/maximal-square/solution/zui-da-ju-xing-zheng-fang-xing-de-dan-diao-zhan-ji/

方法 1：单调栈+DP

更多硬核内容就在公众号「yes 的练级攻略」

```
class Solution {
    public int maximalSquare(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length ==
 0) return 0;
        int res = 0;
        int row = matrix.length, col = matrix[0].length;
        int[][] dp = new int[row][col+1];
        for (int i = 0; i < row; ++i){
            for (int j = 0; j < col; ++j){
                if (i == 0 && matrix[i][j] == '1'){
                    dp[i][j] = 1;
                    continue;
                }
                if (matrix[i][j] == '1'){
                    dp[i][j] = dp[i-1][j] + 1;
                }
            }
        }
        for (int i = 0; i < row; ++i){
            res = Math.max(res, largeArea(dp[i]));
        }
        return res;
    }
    public int largeArea(int[] height){
        int[] tmp = new int[height.length + 2];
        System.arraycopy(height, 0, tmp, 1, height.length);
        Deque<Integer> sta = new ArrayDeque<>();
        int area = 0;
        for (int i = 0; i < tmp.length; i++){
            while (!sta.isEmpty() && tmp[i] < tmp[sta.peek()]){
                //这边找到高 h，宽 w 里面最小的作为正方形的边长
                int h = tmp[sta.pop()];
                int w = i - sta.peek() - 1;
                int min = Math.min(h, w);
                area = Math.max(area, min * min);
            }
            sta.push(i);
        }
        return area;
    }
}
```

https://leetcode-cn.com/problems/maximal-square/solution/zui-da-zheng-fang-xing-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：动态规划

方法一虽然直观，但是时间复杂度太高，有没有办法降低时间复杂度呢？

可以使用动态规划降低时间复杂度。我们用 $dp(i,j)$ 表示以 (i,j) 为右下角，且只包含 1 的正方形的边长最大值。如果我们能计算出所有 $dp(i,j)$ 的值，那么其中的最大值即为矩阵中只包含 1 的正方形的边长最大值，其平方即为最大正方形的面积。

那么如何计算 dp 中的每个元素值呢？对于每个位置 (i,j)，检查在矩阵中该位置的值：

- 如果该位置的值是 0，则 $dp(i,j)=0$，因为当前位置不可能在由 1 组成的正方形中；
- 如果该位置的值是 1，则 $dp(i,j)$ 的值由其上方、左方和左上方的三个相邻位置的 dp 值决定。具体而言，当前位置的元素值等于三个相邻位置的元素中的最小值加 1，状态转移方程如下：

$$dp(i,j)=min(dp(i-1,j),dp(i-1,j-1),dp(i,j-1))+1$$

如果读者对这个状态转移方程感到不解，可以参考 1277. 统计全为 1 的正方形子矩阵的官方题解，其中给出了详细的证明。

此外，还需要考虑边界条件。如果 i 和 j 中至少有一个为 0，则以位置 (i,j) 为右下角的最大正方形的边长只能是 1，因此 $dp(i,j)=1$。

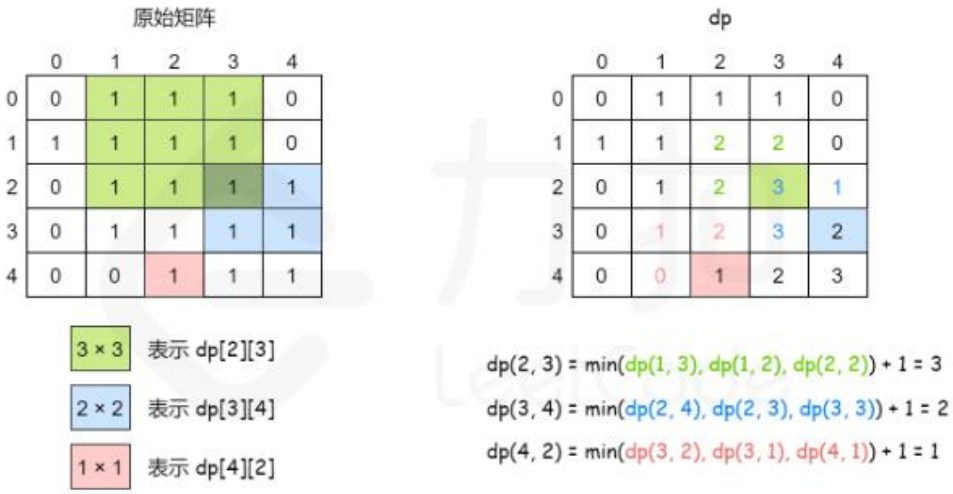
以下用一个例子具体说明。原始矩阵如下。

```
0 1 1 1 0
1 1 1 1 0
0 1 1 1 1
0 1 1 1 1
0 0 1 1 1
```

对应的 dp 值如下。

```
0 1 1 1 0
1 1 2 2 0
0 1 2 3 1
0 1 2 3 2
0 0 1 2 3
```

下图也给出了计算 dp 值的过程。



复杂度分析

- 时间复杂度：$O(mn)$，其中 m 和 n 是矩阵的行数和列数。需要遍历原始矩阵中的每个元素计算 `dp` 的值。
- 空间复杂度：$O(mn)$，其中 m 和 n 是矩阵的行数和列数。创建了一个和原始矩阵大小相同的矩阵 `dp`。由于状态转移方程中的 $dp(i,j)$ 由其上方、左方和左上方的三个相邻位置的 dp 值决定，因此可以使用两个一维数组进行状态转移，空间复杂度优化至 $O(n)$。

方法 2：只用动规

```
class Solution {
    public int maximalSquare(char[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length ==
 0) return 0;
        int res = 0, len = 0;
        int row = matrix.length, col = matrix[0].length;
        int[][] dp = new int[row][col];
        for (int i = 0; i < row; ++i){
            for (int j = 0; j < col; ++j){
                if (matrix[i][j] == '1'){
                    if (i == 0 || j == 0){
                        dp[i][j] = 1;
                    } else {
                        dp[i][j] = Math.min(Math.min(dp[i-1][j], dp[i][
j-1]), dp[i-1][j-1]) + 1;
                    }
                    len = Math.max(len, dp[i][j]);
                }
            }
        }
        res = len * len;
        return res;
    }
}
```

17、每日温度（739）（lintcode1060）【方法 2 是单调栈】

739. 每日温度

难度 中等　434　收藏　分享　切换为英文　关注　反馈

请根据每日 `气温` 列表，重新生成一个列表。对应位置的输出为：要想观测到更高的气温，至少需要等待的天数。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

例如，给定一个列表 `temperatures = [73, 74, 75, 71, 69, 72, 76, 73]`，你的输出应该是 `[1, 1, 4, 2, 1, 1, 0, 0]`。

提示： `气温` 列表长度的范围是 `[1, 30000]`。每个气温的值的均为华氏度，都是在 `[30, 100]` 范围内的整数。

通过次数 86,970　提交次数 136,021

https://leetcode-cn.com/problems/daily-temperatures/solution/java-by-sdwwld/

方法 1：从后面开始找，效率比较高（4ms）

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
```

```
        int[] res = new int[T.length];
        //从后面开始查找
        for (int i = res.length - 1; i >= 0; i--) {
            int j = i + 1;
            while (j < res.length) {
                if (T[j] > T[i]) {
                    //如果找到就停止 while 循环
                    res[i] = j - i;
                    break;
                } else if (res[j] == 0) {
                    //如果没找到，并且 res[j]==0。说明第 j 个元素后面没有
                    //比第 j 个元素大的值，因为这一步是第 i 个元素大于第 j 个元
素的值，
                    //那么很明显这后面就更没有大于第 i 个元素的值。直接终止
while 循环。
                    break;
                } else {
                    //如果没找到，并且 res[j]！=0 说明第 j 个元素后面有比第 j
个元素大的值，
                    //然后我们让 j 往后挪 res[j]个单位，找到那个值，再和第 i 个
元素比较
                    j += res[j];
                }
            }
        }
        return res;
    }
}
```

方法 2：单调栈（17ms）

```
class Solution {
    public int[] dailyTemperatures(int[] T) {
        int len = T.length;
        Deque<Integer> stack = new ArrayDeque<>();
        int[] res = new int[len];
        for (int i = 0; i < len; i++){
            int h = T[i];
            if (stack.isEmpty() || h <= T[stack.peek()]){
                stack.push(i);
            } else {
                int top = stack.pop();
                res[top] = i - top;
                i--;
            }
        }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            return res;
        }
}
```

18、股票价格跨度（901）（lintcode1740）【单调栈】

901. 股票价格跨度

难度 中等 55 收藏 分享 切换为英文 关注 反馈

编写一个 StockSpanner 类，它收集某些股票的每日报价，并返回该股票当日价格的跨度。

今天股票价格的跨度被定义为股票价格小于或等于今天价格的最大连续日数（从今天开始往回数，包括今天）。

例如，如果未来7天股票的价格是 [100, 80, 60, 70, 60, 75, 85]，那么股票跨度将是 [1, 1, 1, 2, 1, 4, 6]。

示例：

输入：["StockSpanner","next","next","next","next","next","next","next"], [[],[100],[80],[60],[70],[60],[75],[85]]
输出：[null,1,1,1,2,1,4,6]
解释：
首先，初始化 S = StockSpanner()，然后：
S.next(100) 被调用并返回 1，
S.next(80) 被调用并返回 1，
S.next(60) 被调用并返回 1，
S.next(70) 被调用并返回 2，
S.next(60) 被调用并返回 1，
S.next(75) 被调用并返回 4，
S.next(85) 被调用并返回 6。

注意（例如）S.next(75) 返回 4，因为截至今天的最后 4 个价格
（包括今天的价格 75）小于或等于今天的价格。

提示：

1. 调用 StockSpanner.next(int price) 时，将有 1 <= price <= 10^5。
2. 每个测试用例最多可以调用 10000 次 StockSpanner.next。
3. 在所有测试用例中，最多调用 150000 次 StockSpanner.next。
4. 此问题的总时间限制减少了 50%。

单调栈问题 题目中提到股票价格小于或等于今天价格的最大连续日数。由于这是一个在线问题，所以我们必然是要将输入的 price 给存储起来，而且同时我们也需要保留这是第几次输入的信息。 需要注意的是边界问题，当我们输入第一个 price 的时候，此时 stack 空。 所以这里拿出来判断特殊处理一下。

https://leetcode-cn.com/problems/online-stock-span/solution/dan-diao-zhan-by-huan-shi-da-ai-xin-lan-2/

```
class StockSpanner {
    public StockSpanner() {
    }
    private Stack<int[]> stack = new Stack<>();
    public int next(int price) {
        int res = 1;
        while (!stack.isEmpty() && stack.peek()[0] <= price)
            res += stack.pop()[1];
        stack.push(new int[]{price, res});
        return res;
    }
}
```

```
/**
 * Your StockSpanner object will be instantiated and called as such:
 * StockSpanner obj = new StockSpanner();
 * int param_1 = obj.next(price);
 */
```

19、下一个更大元素 1（496）【单调栈】

496. 下一个更大元素 I

难度 简单 245 收藏 分享 切换为英文 关注 反馈

给定两个 **没有重复元素** 的数组 `nums1` 和 `nums2` ，其中 `nums1` 是 `nums2` 的子集。找到 `nums1` 中每个元素在 `nums2` 中的下一个比其大的值。

`nums1` 中数字 **x** 的下一个更大元素是指 **x** 在 `nums2` 中对应位置的右边的第一个比 **x** 大的元素。如果不存在，对应位置输出 -1 。

示例 1:

```
输入: nums1 = [4,1,2], nums2 = [1,3,4,2].
输出: [-1,3,-1]
解释:
    对于num1中的数字4，你无法在第二个数组中找到下一个更大的数字，因此输出 -1。
    对于num1中的数字1，第二个数组中数字1右边的下一个较大数字是 3。
    对于num1中的数字2，第二个数组中没有下一个更大的数字，因此输出 -1。
```

示例 2:

```
输入: nums1 = [2,4], nums2 = [1,2,3,4].
输出: [3,-1]
解释:
    对于 num1 中的数字 2 ，第二个数组中的下一个较大数字是 3 。
    对于 num1 中的数字 4 ，第二个数组中没有下一个更大的数字，因此输出 -1 。
```

提示：

1. `nums1` 和 `nums2` 中所有元素是唯一的。
2. `nums1` 和 `nums2` 的数组大小都不超过1000。

YES的练级攻略

https://leetcode-cn.com/problems/next-greater-element-i/solution/xia-yi-ge-geng-da-yuan-su-i-by-leetcode/

方法一：单调栈

我们可以忽略数组 `nums1` ，先对将 `nums2` 中的每一个元素，求出其下一个更大的元素。随后对于将这些答案放入哈希映射（HashMap）中，再遍历数组 `nums1` ，并直接找出答案。对于 `nums2` ，我们可以使用单调栈来解决这个问题。

我们首先把第一个元素 `nums2[1]` 放入栈，随后对于第二个元素 `nums2[2]` ，如果 `nums2[2] > nums2[1]` ，那么我们就找到了 `nums2[1]` 的下一个更大元素 `nums2[2]` ，此时就可以把 `nums2[1]` 出栈并把 `nums2[2]` 入栈；如果 `nums2[2] <= nums2[1]` ，我们就仅把 `nums2[2]` 入栈。对于第三个元素 `nums2[3]` ，此时栈中有若干个元素，那么所有比 `nums2[3]` 小的元素都找到了下一个更大元素（即 `nums2[3]` ），因此可以出栈，在这之后，我们将 `nums2[3]` 入栈，以此类推。

可以发现，我们维护了一个单调栈，栈中的元素从栈顶到栈底是单调不降的。当我们遇到一个新的元素 `nums2[i]` 时，我们判断栈顶元素是否小于 `nums2[i]` ，如果是，那么栈顶元素的下一个更大元素即为 `nums2[i]` ，我们将栈顶元素出栈。重复这一操作，直到栈为空或者栈顶元素大于 `nums2[i]` 。此时我们将 `nums2[i]` 入栈，保持栈的单调性，并对接下来的 `nums2[i + 1]`, `nums2[i + 2]` ... 执行同样的操作。

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Deque<Integer> stack = new ArrayDeque<>();
        HashMap<Integer, Integer> map =  new HashMap<>();
```

```
        int[] res = new int[nums1.length];
        for (int i = 0; i < nums2.length; i++){
            while (!stack.isEmpty() && nums2[i] > stack.peek()){
                map.put(stack.pop(), nums2[i]);
            }
            stack.push(nums2[i]);
        }
        while (!stack.isEmpty()){
            map.put(stack.pop(), -1);
        }
        for (int i = 0; i < nums1.length; i++){
            res[i] = map.get(nums1[i]);
        }
        return res;
    }
}
```

20、去除重复字母（316）【单调栈】

316. 去除重复字母

难度 困难 190 收藏 分享 切换为英文 关注 反馈

给你一个仅包含小写字母的字符串，请你去除字符串中重复的字母，使得每个字母只出现一次。需保证返回结果的字典序最小（要求不能打乱其他字符的相对位置）。

示例 1:

```
输入："bcabc"
输出："abc"
```

示例 2:

```
输入："cbacdcbc"
输出："acdb"
```

注意：该题与 1081 https://leetcode-cn.com/problems/smallest-subsequence-of-distinct-characters 相同

https://leetcode-cn.com/problems/remove-duplicate-letters/solution/zhan-by-liweiwei1419/

```
class Solution {
    public String removeDuplicateLetters(String s) {
        if (s == null || s.length() < 2) return s;
        Deque<Character> stack = new ArrayDeque<>();
        char[] ch = s.toCharArray();
        StringBuilder res = new StringBuilder();
        for (int i = 0; i < ch.length; i++){
            if (!stack.isEmpty() && stack.contains(ch[i])) continue;
            while (!stack.isEmpty() && ch[i] < stack.peekLast() && s.in
dexOf(stack.peekLast(), i) > 0){
```

```
                stack.removeLast();
            }
            stack.addLast(ch[i]);
        }

        for (Character cur : stack){
            res.append(cur);
        }
        return res.toString();
    }
}
```

21、不同字符的最小子序列（1081）【单调栈，与上一题的意思一样】

1081. 不同字符的最小子序列

难度 中等 39 收藏 分享 切换为英文 关注 反馈

返回字符串 `text` 中按字典序排列最小的子序列，该子序列包含 `text` 中所有不同字符一次。

示例 1：

输入： "cdadabcc"
输出： "adbc"

示例 2：

输入： "abcd"
输出： "abcd"

示例 3：

输入： "ecbacba"
输出： "eacb"

示例 4：

输入： "leetcode"
输出： "letcod"

提示：

1. `1 <= text.length <= 1000`
2. `text` 由小写英文字母组成

注意： 本题目与 316 https://leetcode-cn.com/problems/remove-duplicate-letters/ 相同

https://leetcode-cn.com/problems/smallest-subsequence-of-distinct-characters/solution/tan-xin-suan-fa-zhan-wei-yan-ma-python-dai-ma-java/

```
class Solution {
    public String smallestSubsequence(String text) {
        if (text == null || text.length() < 2) return text;
        Deque<Character> stack = new ArrayDeque<>();
        char[] ch = text.toCharArray();
```

```
        StringBuilder res = new StringBuilder();
        for (int i = 0; i < ch.length; i++){
            if (!stack.isEmpty() && stack.contains(ch[i])) continue;
            while (!stack.isEmpty() && ch[i] < stack.peekLast() && text
.indexOf(stack.peekLast(), i) > 0){
                stack.removeLast();
            }
            stack.addLast(ch[i]);
        }

        for (Character cur : stack){
            res.append(cur);
        }
        return res.toString();
    }
}
```

22、移掉 k 位数字（402）【单调栈】【双指针】

> 402. 移掉K位数字
>
> 难度 中等 264 收藏 分享 切换为英文 关注 反馈
>
> 给定一个以字符串表示的非负整数 *num*，移除这个数中的 *k* 位数字，使得剩下的数字最小。
>
> **注意:**
>
> - *num* 的长度小于 10002 且 ≥ *k*。
> - *num* 不会包含任何前导零。
>
> **示例 1 :**
>
> ```
> 输入: num = "1432219", k = 3
> 输出: "1219"
> 解释: 移除掉三个数字 4, 3, 和 2 形成一个新的最小的数字 1219。
> ```
>
> **示例 2 :**
>
> ```
> 输入: num = "10200", k = 1
> 输出: "200"
> 解释: 移掉首位的 1 剩下的数字为 200. 注意输出不能有任何前导零。
> ```
>
> 示例 **3 :**
>
> ```
> 输入: num = "10", k = 2
> 输出: "0"
> 解释: 从原数字移除所有的数字，剩余为空就是0。
> ```

https://leetcode-cn.com/problems/remove-k-digits/solution/java-dan-diao-dui-lie-shuang-zhi-zhen-by-lzhlyle/

更多硬核内容就在公众号「yes 的练级攻略」

- 总体思路：比较相邻两者
  - 若前小，则保留
  - 若前大，则移除
- 实现思路：单调栈（一般可用 栈、队列、双端队列、双指针 实现）
- 实现步骤
  - 定义单调栈结构（保证单调递增，非严格），用于存储最终结果
  - 遍历字符串，逐个入栈
    - 若新来的比栈顶小，则栈顶出栈 (有人比你年轻，你老得走了)
    - 出栈相当于移除动作，需次数并与 `k` 比较，别删多了
  - `k` 还有余
    - 怎么会还有剩呢？因为上文步骤只是把大的数移走，相等的数都入栈了
    - 从后面的删起
  - 移除头部零（双端队列香不香）有 `'0'`
  - 全空，`return "0";`
  - 构建字符串，顺序返回整个单调栈元素（双端队列好香）
- 复杂度分析
  - 时间复杂度 $O(n)$
  - 空间复杂度 $O(n)$

方法 1：单调栈

```
class Solution {
    public String removeKdigits(String num, int k) {
        Deque<Character> stack = new ArrayDeque<>();
        char[] ch = num.toCharArray();
        // 遍历，逐个入队
        for (char c : ch) {
            while (!stack.isEmpty() && k > 0 && stack.getLast() > c){
                k--;
                stack.removeLast();
            }
            stack.addLast(c);
        }
        // k 还有余，后面大的删起
        while (k > 0 && !stack.isEmpty()){
            k--;
            stack.removeLast();
        }
        //移除头部零
        while (!stack.isEmpty() && stack.getFirst() == '0'){
            stack.removeFirst();
        }
        if (stack.isEmpty()) return "0";
        // 构建，返回
        StringBuilder res = new StringBuilder();
        for (Character tmp : stack){
            res.append(tmp);
        }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        return res.toString();
    }
}
```

方法 2：双指针

```
class Solution {
    public String removeKdigits(String num, int k) {
        int r = 0, n = num.length();
        char[] res = new char[n];

        // 遍历，逐个入数组
        for (char c : num.toCharArray()) {
            while (r > 0 && k > 0 && res[r - 1] > c) {
                k--;
                r--;
            }
            res[r++] = c;
        }

        // k 还有余，后面大的删起
        while (k-- > 0 && r > 0) r--;

        // 移除头部零
        int l = 0; // 此时才需要 left 指针
        while (l < r && res[l] == '0') l++;
        if (l == r) return "0";

        // 构建，返回
        return new String(res, l, r - l);
    }
}
```

23、子数组的最小值之和（907）（lintcode1734）【单调栈】

更多硬核内容就在公众号「yes 的练级攻略」

907. 子数组的最小值之和

难度 中等 140 收藏 分享 切换为英文 关注 反馈

给定一个整数数组 A，找到 min(B) 的总和，其中 B 的范围为 A 的每个（连续）子数组。

由于答案可能很大，因此**返回答案模** `10^9 + 7`。

示例：

```
输入：[3,1,2,4]
输出：17
解释：
子数组为 [3]，[1]，[2]，[4]，[3,1]，[1,2]，[2,4]，[3,1,2]，[1,2,4]，[3,1,2,4]。
最小值为 3，1，2，4，1，1，2，1，1，1，和为 17。
```

提示：

1. `1 <= A <= 30000`
2. `1 <= A[i] <= 30000`

https://leetcode-cn.com/problems/sum-of-subarray-minimums/solution/zi-shu-zu-de-zui-xiao-zhi-zhi-he-by-leetcode/

方法 2：维护最小值栈

想法

对于每个 `j`，考虑所有子序列 `[i, j]` 的最小值。想法是每当我们增加 `j`，这些最小值可能会有关联，事实上，`min(A[i:j+1]) = min(A[i:j], A[j+1])`。

模拟数组 `A = [1, 7, 5, 2, 4, 3, 9]`，当 `j = 6` 事所有子序列 `[i, j]` 的最小值为 `B = [1, 2, 2, 2, 3, 3, 9]`，可以发现重要点是 `i = 0, i = 3, i = 5, i = 6`，分别是从 `j` 开始向左移动遇到的最小值的位置。

算法

维护关于重要点的编码 `B`，对于上面提到的 `(A, j)` 维护 `stack = [(val=1, count=1), (val=2, count=3), (val=3, count=2), (val=9, count=1)]`，这表示最小值的编码为 `B = [1, 2, 2, 2, 3, 3, 9]`。对于每个 `j` 我们需要计算 `sum(B)`。

当我们增加 `j`，我们用最新的元素 `(val=x, count=1)` 更新栈。弹出所有值 `>= x` 的元素，因为当前子序列 `[i, j]` 的最小值将是 `A[j]` 而不是之前的值。

最后，结果是栈元素的点积 $\sum_{e \in \text{stack}} e.\text{val} * e.\text{count}$，我们同时会用变量 `dot` 来维护。

复杂度分析

- 时间复杂度：$O(N)$，其中 N 是 `A` 的长度。
- 空间复杂度：$O(N)$。

```
class Solution {
    public int sumSubarrayMins(int[] A) {
        int MOD = 1_000_000_007;
        Stack<RepInteger> stack = new Stack();
        int ans = 0, dot = 0;
        for (int j = 0; j < A.length; ++j) {
            // Add all answers for subarrays [i, j], i <= j
            int count = 1;
            while (!stack.isEmpty() && stack.peek().val >= A[j]) {
```

```
                RepInteger node = stack.pop();
                count += node.count;
                dot -= node.val * node.count;
            }
            stack.push(new RepInteger(A[j], count));
            dot += A[j] * count;
            ans += dot;
            ans %= MOD;
        }
        return ans;
    }
}
class RepInteger {
    int val, count;
    RepInteger(int v, int c) {
        val = v;
        count = c;
    }
}
```

24、K 个空花盆（683）

683. K 个空花盆

难度 困难 30 收藏 分享 切换为英文 关注 反馈

花园里有 N 个花盆，每个花盆里都有一朵花。这 N 朵花会在 N 天内依次开放，每天有且仅有一朵花会开放并且会一直盛开下去。

给定一个数组 flowers 包含从 1 到 N 的数字，每个数字表示在那一天开放的花所在的花盆编号。

例如，flowers[i] = x 表示在第 i+1 天盛开的花在第 x 个花盆中，i 和 x 都在 1 到 N 的范围内。

给你一个整数 k，请你输出在哪一天恰好有两朵盛开的花，他们中间间隔了 k 朵花并且都没有开放。

如果不存在，输出 -1。

样例 1:

输入：
flowers: [1,3,2]
k: 1
输出： 2
解释： 在第二天，第一朵和第三朵花都盛开了。

样例 2:

```
输入:
flowers: [1,2,3]
k: 1
输出: -1
```

注释:

1. 给定的数组范围是 [1, 20000]。

https://leetcode-cn.com/problems/k-empty-slots/solution/dan-diao-zhan-by-sghk6zextz/

方法 1：单调栈（111ms）

```
class Solution {
    //对于一个位置的花盆，找到其左右距离它自身最近的更小值，使用单调栈找最近小值，若有多个符合条件，输出最小天数。时间复杂度：O(n)
    public int kEmptySlots(int[] bulbs, int K) {
        Stack<Integer> stack=new Stack<Integer>();
        Stack<Integer> stack2=new Stack<Integer>();
        int[] bulbs2=new int[bulbs.length+1];
        for(int i=0;i<bulbs2.length-1;i++){
            bulbs2[bulbs[i]]=i+1;
        }
        //stack.push(0);
        int answer=Integer.MAX_VALUE;;
        for(int i=1;i<bulbs.length+1;i++){
            while(!stack.empty()&&bulbs2[stack.peek()]>bulbs2[i]){
                stack.pop();
            }
            if(!stack.empty()){
                if(i-stack.peek()-1==K)
                    answer=Math.min(bulbs2[i],answer);
            }
            stack.push(i);

        }
        for(int i=bulbs.length;i>=1;i--){
            while(!stack2.empty()&&bulbs2[stack2.peek()]>bulbs2[i]){
                stack2.pop();
            }
            if(!stack2.empty()){
                if(stack2.peek()-1-i==K)
                    answer=Math.min(bulbs2[i],answer);
            }
```

```
                stack2.push(i);

            }
            return answer==Integer.MAX_VALUE?-1:answer;
        }
}
```

https://leetcode-cn.com/problems/k-empty-slots/solution/zhi-zhen-hua-dong-by-aidenmum/

方法 2：双指针（5ms）

```
class Solution {
 public int kEmptySlots(int[] bulbs, int K) {
        //bulbs[i]=x:第 i 天，花盆 X 的花会开放
        //重点：把数据结构重新转换一下
        //day[i]=x:第 i 盆花会在第 x 天开放
        int[] days = new int[bulbs.length];
        for (int i = 0; i < days.length; i++) {
            days[bulbs[i] - 1] = i + 1;
        }
        //在 day[i]上使用指针滑动
        //恰好有两朵盛开的花，他们中间间隔了 k 朵花并且都没有开放.
        //假设区间 [i,i+k].
        //只要 day[i]<= day[i+1]~day[i+k-1] && day[i+k]<=day[i],那么他们在第 N 天时，（i~N）就能同时开放
        //如果不符合，start=start+k+1
        int start = 0;
        int end = start + K + 1;
        int result = Integer.MAX_VALUE;
        for (int i = 0; end < days.length; i++) {
            //不符合条件
            if (days[i] <= days[start] || days[i] <= days[end]) {
                if (i == end) {
                    //取满足条件的结果中最小的一个就是题目答案
                    result = Math.min(result, Math.max(days[start], days[end]));

                }
                //继续寻找下一个目标，start 从 i 开始
                start = i;
                end = start + K + 1;
            }
        }
        return result == Integer.MAX_VALUE ? -1 : result;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/k-empty-slots/solution/hong-hei-shu-mei-tian-xin-kai-de-hua-pen-pan-duan-/

方法 3：红黑树（81ms）

```
class Solution {
    public int kEmptySlots(int[] bulbs, int K) {
        // 间隔 K 个花盆，则两者的差要是 K+1
        // 开花的花盆编号
        TreeSet<Integer> slotSet = new TreeSet<>();
        for (int i = 0; i < bulbs.length; i++) {
            // 编号小于当前花盆的开花花盆
            Integer low = slotSet.lower(bulbs[i]);
            if (low != null && bulbs[i] - low == K + 1){
                return i + 1;// 满足条件的天
            }
            // 编号大于当前花盆开花的花盆
            Integer high = slotSet.higher(bulbs[i]);
            if (high != null && high - bulbs[i] == K + 1){
                return i + 1;// 满足条件的天
            }
            slotSet.add(bulbs[i]);// 当前花盆加入
        }
        return -1;
    }
}
```

十九、有限自动机

https://chengzhaoxi.xyz/52244.html

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

$1 词法分析和有限自动机的逻辑链条

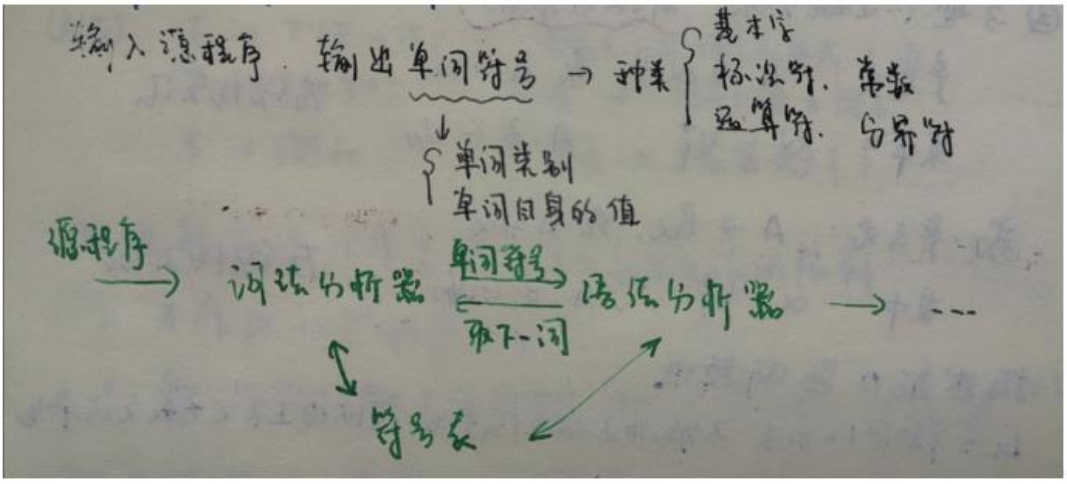
在编译原理中，一个程序语言的**单词表**

```
基本字: for, while, int, return, const, ...
标识符: tmp, next, left, ...
常数: 3, 1.7, 'o', "abc", 1e9, 2.5e-7, ...
运算符: +, -, *, /, ^, ...
分界符: 空格，空行
```

是词法分析的对象，单词表里的所有单词构成了一个字符串的集合，这种集合称为**正则集**。词法分析要研究的是正则集。

正则集可以用**正则表达式**表示，它们是一一对应的。

词法分析器的功能就是输入源串，输出单词符号，这里的输出单词符号包含两部分：单词类别和单词自身的值，类别和值由单词表给定。



可以以**状态转换图**为工具手工设计词法分析器。

确定有限自动机 DFA 是状态转换图的形式化。
DFA 是一种自动机，可以用于描述状态转换图，容易做程序实现，且任何一个 DFA 的程序实现框架是一样的，差别仅在于**状态转移矩阵**(相当于邻接矩阵)。
伪代码如下：

更多硬核内容就在公众号「yes 的练级攻略」

```
// cur_state = 初态
// 获取下一字符 ch
string str = "";
while(statetrans[cur_state][ch] 有定义)
{
    str += ch;
    cur_state = statetrans[cur_state][ch];
    if(cur_state 是终态)
        return str;
    // 获取下一字符
}
```

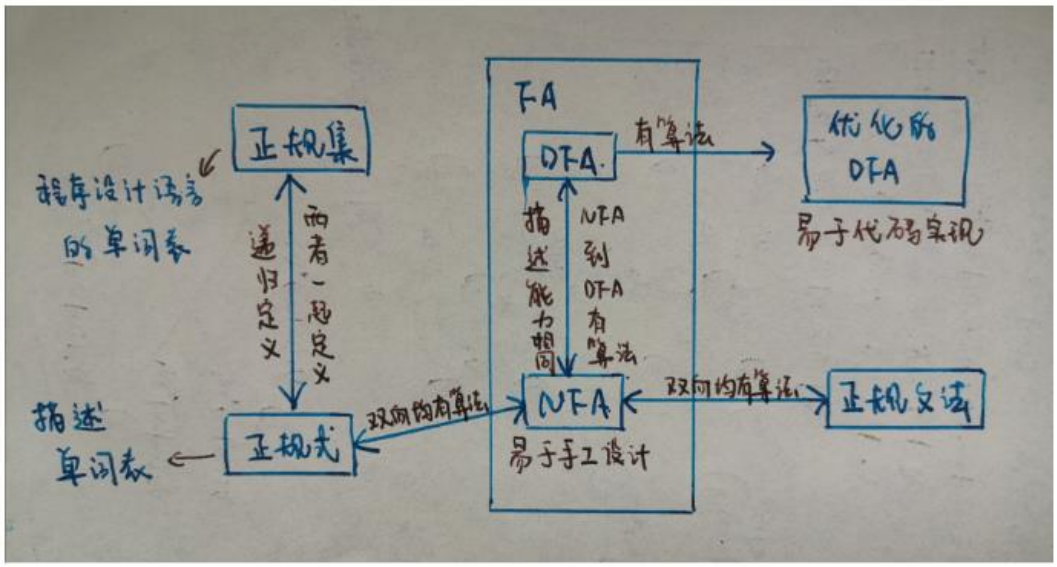
但是 DFA 不容易做设计，给定一个语言的单词表试图画一个 DFA 识别这种语言的话，很困难。

但是幸运的是设计**非确定有限自动机 NFA** 识别某种语言相对比较容易，NFA 和 DFA 是等价的。DFA 是 NFA 的特例，**任何 NFA 都有算法可以转化为 DFA，即子集法。**

给定单词表，用 NFA 做设计还是嫌有些难，但是幸运的是 **NFA 和正则表达式之间可以互相转化。**而拿到语言的单词表后，写正则表达式是相对容易的。

通过 `语言单词表 -> 正则表达式 -> NFA -> DFA` 这条路线得到 DFA，之后，还**有成熟算法可以将 DFA 化简，使得状态数量最少。**每个步骤的是有算法的(算法本身就是等价性的构造性证明)，因此可以自动化地实现。

词法规则还可以用文法描述，正则文法和 NFA 之间可以互相转化。



因此最终结论：如果要研究一个语言的词法分析，拿到单词表后有两种选择

1. 用正则表达式描述词法规则
2. 用正则文法描述词法规则

这两种方法都可以得到 NFA，进而得到 DFA，进而优化 DFA 使得状态数最少。

以上就是词法分析的整体逻辑，涉及到文法，正则表达式，有限自动机，概念之间的关系如上图，其中每个箭头都有证明，而证明过程是一种构造性证明，**这种构造性证明就是概念间进行转化的算法。**
证明过程和算法的细节需要参考编译原理词法分析部分。

力扣上有几道经典的有限自动机的题目，涉及到的方法是上述词法分析逻辑链条中 以状态转换图为工具手工设计词法分析器 这一块，
而 DFA 是状态转换图的形式化，因此就是**拿到较简单的单词表后手工设计对应的 DFA，进而画出状态转换图，写出状态转移矩阵。**

1、字符串转换整数 atoi（8）

8. 字符串转换整数 (atoi)

难度 中等 771 收藏 分享 切换为英文 关注 反馈

请你来实现一个 `atoi` 函数，使其能将字符串转换成整数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。接下来的转化规则如下：

- 如果第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字字符组合起来，形成一个有符号整数。
- 假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成一个整数。
- 该字符串在有效的整数部分之后也可能会存在多余的字符，那么这些字符可以被忽略，它们对函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换，即无法进行有效转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0 。

提示：

- 本题中的空白字符只包括空格字符 `' '` 。
- 假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 $[-2^{31}, 2^{31} - 1]$。如果数值超过这个范围，请返回 INT_MAX ($2^{31} - 1$) 或 INT_MIN (-2^{31}) 。

示例 1:

```
输入："42"
输出：42
```

示例 2:

```
输入："   -42"
输出：-42
解释：第一个非空白字符为 '-'，它是一个负号。
      我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。
```

示例 3:

```
输入: "4193 with words"
输出: 4193
解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。
```

示例 4:

```
输入: "words and 987"
输出: 0
解释: 第一个非空字符是 'w'，但它不是数字或正、负号。
     因此无法执行有效的转换。
```

示例 5:

```
输入: "-91283472332"
输出: -2147483648
解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
     因此返回 INT_MIN (-2^31) 。
```

方法2：有限自动机

记 DFA $M = (S, \Sigma, f, S_0, F)$

首先看状态集合 S

-1: 非法状态
0: 起始的空格
1: 起始的+-
2: 整数，例如(9, -11, +191)

再看字母表 Σ

> *空格*
> *+-*
> *0123456789*
> *非法字符*

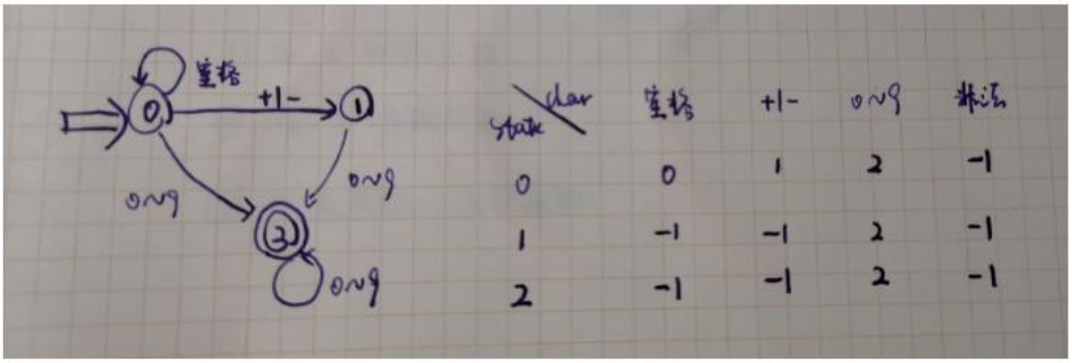
初始状态 $S_0 = 0$

终结状态 $F = \{2, 3\}$

DFA 从状态 0 开始逐字符读字符串，一边读一边更新结果。
在某字符处状态转移后，若所得状态非法，直接返回此时的结果，否则继续读下一字符，直至字符串耗尽。

当字符串耗尽，返回此时的结果。

状态转移函数 f 以状态转换图表示。



方法 1：有限自动机

```
public class Solution {
    private int state = 0, sigh = 1;
    private int[][] table = {{0, 1, 2, -1}, {-1, -1, 2, -1}, {-1, -1, 2, -1}};
    private long res = 0;
    public int myAtoi(String str) {
        if (str == null || str.length() == 0) return 0;
        int n = str.length();
        for (int i = 0; i < n; i++){
            state = table[state][make(str.charAt(i))];
            if (state < 0) return  sigh * (int) res;
            if (state == 1 && str.charAt(i) == '-') sigh = -1;
            if (state == 2) {
                res *= 10;
                res += str.charAt(i) - '0';
            }
            if (res * sigh <= Integer.MIN_VALUE) return Integer.MIN_VALUE;
            if (res * sigh >= Integer.MAX_VALUE) return Integer.MAX_VALUE;
        }
        return sigh * (int) res;
    }
    private int make(char c){
        switch (c){
            case ' ': return 0;
            case '+': return 1;
            case '-': return 1;
            default: return number(c);
        }
    }
    private int number(char c){
        if (c >= '0' && c <= '9') return 2;
        else return 3;
    }
}
```

方法 2：简单模拟

```
public class Solution {
    public int myAtoi(String str) {
        char[] chars = str.toCharArray();
        int n = chars.length;
        int idx = 0, res = 0;
        // 去掉前导空格
        while (idx < n && chars[idx] == ' ') idx++;
        if (idx == n) return 0;
        // 判断符号位
        boolean negative = false;
        if (chars[idx] == '-') {
            //遇到负号
            negative = true;
            idx++;
        } else if (chars[idx] == '+') {
            // 遇到正号
            idx++;
        } else if (!isDigit(chars[idx])) {
            // 其他符号
            return 0;
        }
        //去除首部0
        while (idx < n && chars[idx] == '0') idx++;
        //判断其他位置的数字
    //isDigit(chr c)与Character.isDigit(chr c)相同
        while (idx < n && isDigit(chars[idx])) {
            int digit = chars[idx] - '0';
            if (res > (Integer.MAX_VALUE - digit) / 10) {
                // 本来应该是 res * 10 + digit > Integer.MAX_VALUE
                // 但是 *10 和 + digit 都有可能越界，所有都移动到右边去就可
以了。
                return negative? Integer.MIN_VALUE : Integer.MAX_VALUE;
            }
            res = res * 10 + digit;
            idx++;
        }
        return negative? -res : res;
    }
    //与Character.isDigit(chr c)相同
    public boolean isDigit(char c) {
        return c >= '0' && c <= '9';
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

2、正则表达式匹配（10）（剑指 19）

> 10. 正则表达式匹配
>
> 难度 困难 1241
>
> 给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。
>
> ```
> '.' 匹配任意单个字符
> '*' 匹配零个或多个前面的那一个元素
> ```
>
> 所谓匹配，是要涵盖 **整个** 字符串 `s` 的，而不是部分字符串。
>
> **说明:**
>
> - `s` 可能为空，且只包含从 `a-z` 的小写字母。
> - `p` 可能为空，且只包含从 `a-z` 的小写字母，以及字符 `.` 和 `*`。

动态规划参考链接：

https://leetcode-cn.com/problems/regular-expression-matching/solution/dong-tai-gui-hua-zen-yao-cong-0kai-shi-si-kao-da-b/

禁止商用

状态

首先状态 dp 一定能自己想出来。
`dp[i][j]` 表示 `s` 的前 i 个是否能被 `p` 的前 j 个匹配

转移方程

怎么想转移方程？首先想的时候从已经求出了 `dp[i-1][j-1]` 入手，再加上已知 `s[i]`、`p[j]`，要想的问题就是怎么去求 `dp[i][j]`。

已知 `dp[i-1][j-1]` 意思就是前面子串都匹配上了，不知道新的一位的情况。
那就分情况考虑，所以对于新的一位 `p[j]` `s[i]` 的值不同，要分情况讨论：

1. 考虑最简单的 `p[j] == s[i] : dp[i][j] = dp[i-1][j-1]`

然后从 `p[j]` 可能的情况来考虑，让 `p[j]=各种能等于的东西`。

2. `p[j] == ".” : dp[i][j] = dp[i-1][j-1]`
3. `p[j] ==" * ":`

更多硬核内容就在公众号「yes 的练级攻略」

第一个难想出来的点：怎么区分 * 的两种讨论情况

首先给了 `*`，明白 `*` 的含义是 匹配零个或多个前面的那一个元素，**所以要考虑他前面的元素** `p[j-1]`。`*` 跟着他前一个字符走，前一个能匹配上 `s[i]`，`*` 才能有用，前一个都不能匹配上 `s[i]`，`*` 也无能为力，只能让前一个字符消失，也就是匹配 0 次前一个字符。
所以按照 `p[j-1]` 和 `s[i]` 是否相等，我们分为两种情况：

3.1 `p[j-1] != s[i] : dp[i][j] = dp[i][j-2]`

- 这就是刚才说的那种前一个字符匹配不上的情况。
- 比如 `(ab, abc * )`。遇到 `*` 往前看两个，发现前面 `s[i]` 的 `ab` 对 `p[j-2]` 的 `ab` 能匹配，虽然后面是 `c*`，但是可以看做匹配 0 次 `c`，相当于直接去掉 `c *`，所以也是 `True`。注意 `(ab, abc**)` 是 `False`。

3.2 `p[j-1] == s[i] or p[j-1] == ".":`

- `*` 前面那个字符，能匹配 `s[i]`，或者 `*` 前面那个字符是万能的 `.`
- 因为 `. *` 就相当于 `. .`，那就只要看前面可不可以匹配就行。
- 比如 `(##b , ###b *)`，或者 `( ##b , ### . * )` 只看 `###` 后面一定是能够匹配上的。
- 所以要看 `b` 和 `b *` 前面那部分 `##` 的地方匹不匹配。

第二个难想出来的点：怎么判断前面是否匹配

```
dp[i][j] = dp[i-1][j] // 多个字符匹配的情况
or dp[i][j] = dp[i][j-1] // 单个字符匹配的情况
or dp[i][j] = dp[i][j-2] // 没有匹配的情况
```

看 `###` 匹不匹配，不是直接只看 `###` 匹不匹配，要综合后面的 `b b*` 来分析
这三种情况是 *or* 的关系，满足任意一种都可以匹配上，同时是最难以理解的地方：

`dp[i-1][j]` 就是看 `s` 里 `b` 多不多，`###` 和 `###b *` 是否匹配，一旦匹配，`s` 后面再添个 `b` 也不影响，因为有 `*` 在，也就是 `###b` 和 `###b *` 也会匹配。

`dp[i][j-1]` 就是去掉 `*` 的那部分，`###b` 和 `###b` 是否匹配，比如 `qqb qqb`

`dp[i][j-2]` 就是 去掉多余的 `b *`，`p` 本身之前的能否匹配，`###b` 和 `###` 是否匹配，比如 `qqb qqbb*` 之前的 `qqb qqb` 就可以匹配，那多了的 `b *` 也无所谓，因为 `b *` 可以是匹配 0 次 `b`，相当于 `b *` 可以直接去掉了。

三种满足一种就能匹配上。

为什么没有 `dp[i-1][j-2]` 的情况？ 就是 `###` 和 `###` 是否匹配？因为这种情况已经是 `dp[i][j-1]` 的子问题。也就是 `s[i]==p[j-1]`，则 `dp[i-1][j-2]=dp[i][j-1]`。

最后来个归纳：

1. 如果 `p.charAt(j) == s.charAt(i) : dp[i][j] = dp[i-1][j-1]`；
2. 如果 `p.charAt(j) == '.' : dp[i][j] = dp[i-1][j-1]`；
3. 如果 `p.charAt(j) == '*'`：
    1. 如果 `p.charAt(j-1) != s.charAt(i) : dp[i][j] = dp[i][j-2] //in this case, a* only counts as empty`
    2. 如果 `p.charAt(i-1) == s.charAt(i) or p.charAt(i-1) == '.'`：
        - `dp[i][j] = dp[i-1][j] //in this case, a* counts as multiple a`
        - `or dp[i][j] = dp[i][j-1] // in this case, a* counts as single a`
        - `or dp[i][j] = dp[i][j-2] // in this case, a* counts as empty`

方法 1：动态规划

```
class Solution {
    public boolean isMatch(String s, String p) {
        if (s == null || p == null) return false;
        int m = s.length(), n = p.length();
        boolean[][] dp = new boolean[m + 1][n + 1];
        dp[0][0] = true;
        //"" 和 p 的匹配关系初始化，a*a*a*a*a*这种能够匹配空串，其他的是都是
false。
        //  奇数位不管什么字符都是 false，偶数位为* 时
则: dp[0][i] = dp[0][i - 2]
        for (int i = 2; i <= n; i+= 2) {
            if (p.charAt(i - 1) == '*') {
                dp[0][i] = dp[0][i - 2];
            }
        }
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                char sc = s.charAt(i - 1);
                char pc = p.charAt(j - 1);
                if (sc == pc || pc == '.') {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (pc == '*') {
                    if (dp[i][j - 2]) {
                        dp[i][j] = true;
                    } else if (sc == p.charAt(j - 2) || p.charAt(j - 2)
 == '.') {
                        dp[i][j] = dp[i - 1][j];
                    }
                }
            }
        }
        return dp[m][n];
    }
}
```

方法 2：有限自动机（时间复杂度 o(MN)）

```
class Solution {
    public boolean isMatch(String s, String p) {
        // S 串当前下标位置，刚开始为0
        int index = 0;
        // 下一个状态集合
        // 这里状态用 p 串的下标表示，代表下一次可以从 p 的这些下标开始匹配
        Set<Integer> nextState = new HashSet<>();
        // 由于是刚开始匹配，这时候 p 下一个下标是 0
```

```
            nextMatch(p, 0, nextState);
            // nextState 不为空时，表示还有合法的下一个状态，匹配继续
            while (!nextState.isEmpty()){
                // 当前状态就是上一次的 nextState
                Set<Integer> nowState = nextState;
                // 创建新的 nextState
                nextState = new HashSet<>();
                // 测试 s[index] 和集合里的状态是否有匹配
                for (int state : nowState){
                    // 如果同时到达 s 和 p 串末尾，匹配成功
                    if (state >= p.length() && index >= s.length()){
                        return true;
                    }
                    // 仅仅 p 到达末尾还不行
                    else if (state >= p.length()){
                        continue;
                    }
                    // s 和 p 都未到达末尾
                    else if (index < s.length()){
                        // 这里是匹配上的情况
                        if (p.charAt(state) == '.' || s.charAt(index) == p.charAt(s
tate)){
                            // 如果 p 串的下一个字符是'*'，当前状态可以匹配任意多次，所以
下一个状态还是当前
                            if (state+1 < p.length() && p.charAt(state+1) == '*'){
                                nextMatch(p, state, nextState);
                            }
                            // 否则，下一个状态就是 state+1
                            else {
                                nextMatch(p, state+1, nextState);
                            }
                        }
                    }
                }
                index++;
            }
            // 此时，nextState 为空，代表没有合法的下一个状态了，匹配失败
            return false;
        }

        // p:正则表达式
        // state:下一个状态
        // nextState:下一个状态集合，无重复
        private void nextMatch(String p, int state, Set<Integer> nextState){
```

```
            // 首先加上下一个状态到状态集中
            nextState.add(state);
            // 这里是判断下一个字符是'*'的情况，由于此时匹配次数可以是0，所以state+2 也是合法的下一个状态
            if (state+1 < p.length() && p.charAt(state+1) == '*'){
                nextMatch(p, state+2, nextState);
            }
        }
    }
```

3、有效数字（65）【表示数值的字符串（剑指 20）】

65. 有效数字

难度 困难 134 收藏 分享 切换为英文 关注 反馈

验证给定的字符串是否可以解释为十进制数字。

例如:

"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
" -90e3 " => true
" 1e" => false
"e3" => false
" 6e-1" => true
" 99e2.5 " => false
"53.5e93" => true
" --6 " => false
"-+3" => false
"95a54e53" => false

说明: 我们有意将问题陈述地比较模糊。在实现代码之前，你应当事先思考所有可能的情况。这里给出一份可能存在于有效十进制数字中的字符列表:

- 数字 0-9
- 指数 - "e"
- 正/负号 - "+"/"-"
- 小数点 - "."

当然，在输入中，这些字符的上下文也很重要。

更新于 2015-02-10:
C++ 函数的形式已经更新了。如果你仍然看见你的函数接收 `const char *` 类型的参数，请点击重载按钮重置你的代码。

方法 1：正则表达式（22ms）

//boolean java.util.regex.Pattern.matches(String regex, CharSequence input)

^表示正则表达式的开始；

$表示正常表达式的结尾；

使用?来表示非贪婪匹配；

\\d+表示 1 个或多个 0 到 9 的数字,是整数部分(至少是一位整数的整数部分)

\\d*表示 0 个或多个 0 到 9 的数字,是小数部分,0 个时表示这个数没有小数部分,只有整数部分.

\\.表示一个点

```
import java.util.regex.Pattern;
class Solution {
    public boolean isNumber(String s) {
        String p = "(^[+-]?\\d*\\.\\d+([eE][+-]?\\d+)?$)|(^[+-]?\\d+(\\.\\d*)?([eE][+-]?\\d+)?$)";
```

```
        return Pattern.matches(p, s.trim());
    }
}
```

方法 2：逐个匹配（3ms）

```
class Solution {
    public boolean isNumber(String s) {
        if(s ==null || s.length() == 0) return false;
        boolean numSeen=false; //是否出现数字
        boolean dotSeen=false; //是否出现点
        boolean eSeen=false; //是否出现 E 或 e
        char arr[] = s.trim().toCharArray(); //trim()可以删除字符串的头尾空白符
        for(int i = 0; i < arr.length; i++){
            //1. 判断是否属于数字 0-9 区间
            if(arr[i] >= '0' && arr[i] <= '9'){
                numSeen = true;
            }else if(arr[i] == '.'){
                //2. 遇到点的时候，判断前面是否有点或者 E，都需要 return false
                if(dotSeen || eSeen){
                    return false;
                }
                dotSeen = true;
            }else if(arr[i] == 'E' || arr[i] == 'e'){
                //3. 遇到 E 的时候，判断前面数字是否合理，是否有 E，并把 numSeen 置为 false，防止 E 后无数字
                if(eSeen || !numSeen){
                    return false;
                }
                eSeen = true;
                numSeen = false;
            }else if(arr[i] == '+' || arr[i] == '-'){
                //4. 遇到-+的时候，判断是否是第一个，如果不是第一个判断是否在 E 后面，都不满足则 return false
                if(i != 0 && arr[i-1] != 'e' && arr[i-1] != 'E'){
                    return false;
                }
            }else{
                //5. 其他情况
                return false;
            }
        }
        return numSeen;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

方法2: 手工设计 DFA

因状态数比较少，可以手工设计 DFA：

记 DFA $M = (S, \Sigma, f, S_0, F)$

首先看状态集合 S

> -1: 非法状态
> 0: 开始的空格,如" "
> 1: 到e前的正负号，如"+"
> 2: 到e前左侧无数字的小数点，如"+."
> 3: 到e前未遇到小数点的整数，如"+1"
> 4: 到e前的小数，如"+1.e"
> 5: 到e，如"+1.1e"
> 6: 到e后的正负号，如"+1.1e"
> 7: 到e后的整数，如"+1.1e1"
> 8: 到最后的空格，如"+1.1e1 "

再看字母表 Σ

> 空格
> +-
> 0123456789
> .
> e
> 非法字符

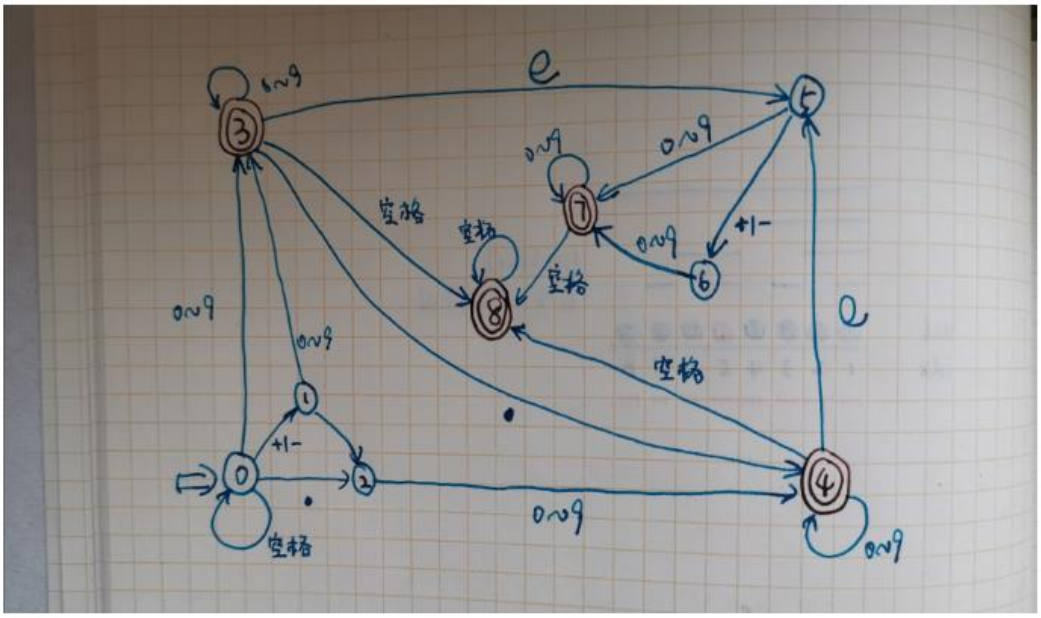
初始状态 $S_0 = 0$

终结状态 $F = \{3, 4, 7, 8\}$

DFA 从状态 0 开始逐字符读字符串，在某字符处状态转移后，若所得状态非法，直接返回 false，否则继续读下一字符，直至字符串耗尽。

当字符串耗尽，若到达终止状态，则返回 true，否则返回 false

状态转移函数 f 以状态转换图表示。



根据装谈转换图，列出状态转移矩阵

state	空格	+-	0~9	.	e	非法
0	0	1	3	2	-1	-1
1	-1	-1	3	2	-1	-1
2	-1	-1	4	-1	-1	-1
3	8	-1	3	4	5	-1
4	8	-1	4	-1	5	-1
5	-1	6	7	-1	-1	-1
6	-1	-1	7	-1	-1	-1
7	8	-1	7	-1	-1	-1
8	8	-1	-1	-1	-1	-1

方法 3：有限自动机（3ms）

```
import java.util.regex.Pattern;

class Solution {
    public boolean isNumber(String s) {
        if (s == null || s.length() == 0) return false;
        int n = s.length();
        int state = 0;
        boolean[] finals = {false, false, false, true, true, false, false, true
, true}; //合法的终止状态
        int[][] table = {
            {0,  1,  3,  2,  -1, -1},
            {-1, -1, 3,  2,  -1, -1},
            {-1, -1, 4,  -1, -1, -1},
            {8,  -1, 3,  4,  5,  -1},
            {8,  -1, 4,  -1, 5,  -1},
            {-1, 6,  7,  -1, -1, -1},
            {-1, -1, 7,  -1, -1, -1},
            {8,  -1, 7,  -1, -1, -1},
            {8,  -1, -1, -1, -1, -1},
        };
        for (int i = 0; i < n; i++){
            state = table[state][make(s.charAt(i))];
            if (state < 0) return false;
        }
        return finals[state];
    }
    private int make(char c){
        switch (c){
            case ' ': return 0;
            case '+': return 1;
            case '-': return 1;
            case '.': return 3;
            case 'e': return 4;
            default: return number(c);
        }
    }
    private int number(char c){
        if (c >= '0' && c <= '9') return 2;
        else return 5;
    }
}
```

二十、数学推导题

1、机器人跳跃（lintcode1426）

1426. 机器人跳跃

中文 English

机器人正在玩一个古老的基于DOS的游戏。游戏中有N+1座建筑——从0到N编号，从左到右排列。编号为0的建筑高度为0个单位，编号为i的建筑的高度为H(i)个单位。
起初， 机器人在编号为0的建筑处。每一步，它跳到下一个（右边）建筑。假设机器人在第k个建筑，且它现在的能量值是E, 下一步它将跳到第个k+1建筑。它将会得到或者失去正比于与H(k+1)与E之差的能量。如果 H(k+1) > E 那么机器人就失去 H(k+1) - E 的能量值，否则它将得到 E - H(k+1) 的能量值。
游戏目标是到达第个N建筑，在这个过程中，能量值不能为负数个单位。现在的问题是机器人以多少能量值开始游戏，才可以保证成功完成游戏?

样例
例 1:

输入:
height:[3,4,3,2,4]
输出: 4
解释:
初始能量为4才能从0走到4
注意事项
- $1 \leq height.size() \leq 10^5$
- $1 \leq height[i] \leq 10^5$

逆递推 假设跳跃前能力为 E,要跳的高度为 H，那么跳跃后的能量就是 2E-H， 那么跳跃后的能量加上高度就是跳跃前的两倍，然后从后往前逆推可以获得答案

```
public class Solution {
    /**
     * @param height: the Building height
     * @return: The minimum unit of initial energy required to complete the game
     */
    public int LeastEnergy(List<Integer> height) {
        int E = 0;
        for (int i = height.size() - 1; i >= 0; i--) {
            E = (E + height.get(i) + 1) >> 1;
        }
        return E;
    }
}
```

2、质因数统计（lintcode1295）【枚举】

描述 控制台 笔记

1295. 质因数统计

题目给定一个正整数N，你需要统计(1,N]之间所有整数质数分解后，所有质数的总个数。

样例
输入: 6
输出: 7
解释: 2=2, 3=3, 4=2*2, 5=5, 6=2*3, 个数和为1+1+2+1+2=7
注意事项
1<N<=100000

```
public class Solution {
    /**
     * @param N: a number
```

禁止商用 公众号：yes的练级攻略

```
     * @return: the number of prime numbers.
     */
    public int Count_PrimeNum(int N) {
        //
        int[] sum = new int[N + 1];
        sum[0] = 0;
        sum[1] = 0;
        for (int n = 2; n <= N; n++){
            int tmp = n;
            int count = 0;
            for (int i = 2; i * i <= tmp; i++){
                while (tmp % i == 0) {
                    count++;
                    tmp /= i;
                }
            }
            if (tmp > 1) count++;
            sum[n] = sum[n-1] + count;
        }
        return sum[N];
    }

}
```

3、丑数（263）【判断一个数是不是丑数】

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

263. 丑数

难度 简单 147 收藏 分享 切换为英文 关注 反馈

编写一个程序判断给定的数是否为丑数。

丑数就是只包含质因数 `2, 3, 5` 的**正整数**。

示例 1:

```
输入: 6
输出: true
解释: 6 = 2 × 3
```

示例 2:

```
输入: 8
输出: true
解释: 8 = 2 × 2 × 2
```

示例 3:

```
输入: 14
输出: false
解释: 14 不是丑数，因为它包含了另外一个质因数 7。
```

说明:

1. `1` 是丑数。
2. 输入不会超过 32 位有符号整数的范围: $[-2^{31}, 2^{31} - 1]$。

https://leetcode-cn.com/problems/ugly-number/solution/gen-ju-cheng-fa-jiao-huan-lu-bu-duan-chu-yi-san-ge/

解题思路

根据定义， 任何一个丑叔都可以写成 n = 2i * 3j * 5k （i,j,k都是幂，输入法不太方便写成指数，三个数都为0的时候，为丑数1的情况）
于是我们按照2,3,5 的顺序依次循环除，当除到不是当前因数的倍数时，进行下一个因素的整数，这样，最后剩下的数为1则为丑数

```
class Solution {
    public boolean isUgly(int num) {
        if(num <= 0) return false;
        int[] factors = new int[]{2, 3, 5};
        for(int f : factors){
            while(num % f == 0) num /= f;
        }
        return num == 1;
```

```
    }
}
```

4、丑数 2（264）【丑数（剑指 49）】【2,3,5 用动规】

> 264. 丑数 II
>
> 难度 中等 357 收藏 分享 切换为英文 关注 反馈
>
> 编写一个程序，找出第 `n` 个丑数。
>
> 丑数就是质因数只包含 `2, 3, 5` 的**正整数**。
>
> **示例:**
>
> ```
> 输入: n = 10
> 输出: 12
> 解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
> ```
>
> **说明:**
>
> 1. `1` 是丑数。
> 2. `n` **不超过**1690。

https://leetcode-cn.com/problems/ugly-number-ii/solution/chou-shu-ii-by-leetcode/

```
class Solution {
    public int nthUglyNumber(int n) {
        if (n <= 0) return 0;
        int[] arr = new int[n];
        arr[0] = 1;
        int n2 = 0, n3 = 0, n5 = 0;
        for (int i = 1; i < n; i++){
           int min = Math.min(arr[n2] * 2, Math.min(arr[n3] * 3, arr[n5] * 5));
            arr[i] = min;
            if (arr[n2] * 2 == min) n2++;
            if (arr[n3] * 3 == min) n3++;
            if (arr[n5] * 5 == min) n5++;
        }
        return arr[n-1];
    }
}
```

5、超级丑数（313）

313. 超级丑数

难度 中等 104 收藏 分享 切换为英文 关注 反馈

编写一段程序来查找第 `n` 个超级丑数。

超级丑数是指其所有质因数都是长度为 `k` 的质数列表 `primes` 中的正整数。

示例:

```
输入: n = 12, primes = [2,7,13,19]
输出: 32
解释: 给定长度为 4 的质数列表 primes = [2,7,13,19]，前 12 个超级丑数序列为: [1,2,4,7,8,13,14,16,19,26,28,32] 。
```

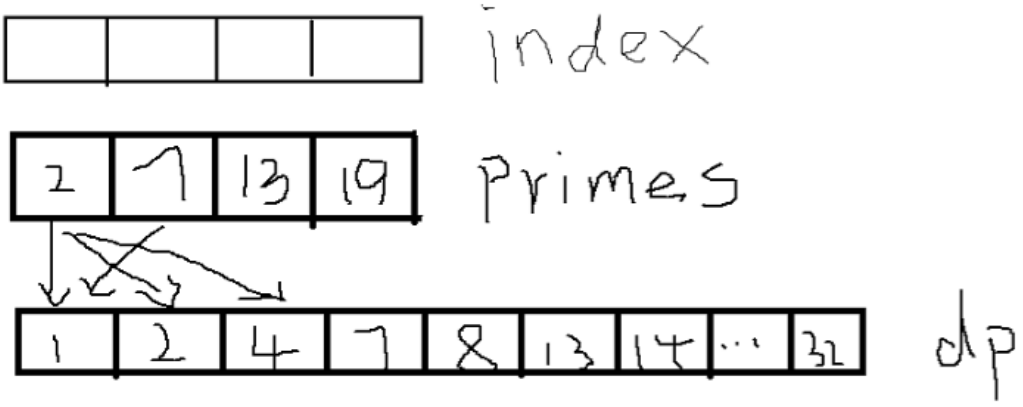
说明:

- `1` 是任何给定 `primes` 的超级丑数。
- 给定 `primes` 中的数字以升序排列。
- $0 < k \le 100, 0 < n \le 10^6, 0 < primes[i] < 1000$ 。
- 第 `n` 个超级丑数确保在 32 位有符整数范围内。

https://leetcode-cn.com/problems/super-ugly-number/solution/313-chao-ji-chou-shu-dui-by-akian/

解题思路:

主要解法: 动态规划。



-使用dp来保存我们的丑数序列。

-每个index元素对应一个primes元素。index滑动选取已有dp元素，dp[index[j]] * primes[j]作为备选丑数。

-将备选丑数与当前min比较，取小。完成一次内循环，选出一个丑数，并更新相应位置的index。

数据结构:

要实现对数据的操作，我们要先明确存储数据的数据结构。

该题的数据结构的作用:

1.整型数组保存丑数序列，索引序列。

算法:

既然明确了我们的数据结构，我们就可以开始我们的算法分析了。

1.第一步，初始化工作，构建整数数组。

2.第二步，循环n-1次，获取丑数序列。

选取最小元素为丑数，并更新索引。

3.第三步，返回丑数结果

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        int[] dp = new int[n];
        dp[0] = 1;
        int len = primes.length;
```

更多硬核内容就在公众号「yes 的练级攻略」

```
            int[] idx = new int[len];
            for (int i = 1; i < n; i++){
                int min = Integer.MAX_VALUE;
                for (int j = 0; j < len; j++){
                    if (min > dp[idx[j]] * primes[j]) min = dp[idx[j]] * primes[j];
                }
                dp[i] = min;
                for (int j = 0; j < len; j++) {
                    if (min == dp[idx[j]] * primes[j]) idx[j]++;
                }
            }
            return dp[n-1];
        }
    }
```

6、第 k 个数（面试题 17.09）【3,5,7 用动规】

面试题 17.09. 第 k 个数

难度 中等　26　收藏　分享　切换为英文　关注　反馈

有些数的素因子只有 3，5，7，请设计一个算法找出第 k 个数。注意，不是必须有这些素因子，而是必须不包含其他的素因子。例如，前几个数按顺序应该是 1，3，5，7，9，15，21。

示例 1:

输入： k = 5

输出： 9

https://leetcode-cn.com/problems/get-kth-magic-number-lcci/solution/chao-jian-dan-jie-fa-yi-kan-jiu-dong-by-yuanninesu/

yes的练级攻略

【思路】

- 定义三个指针 p3, p5, p7， p3 指向的数字永远乘3， p5 指向的数字永远乘5， p7 指向的数字永远乘7
- 初始化所有指针都指向第一个丑数，即1
- 我们从 dp[p3]*3, dp[p5]*5, dp[p7]*7 选取最小的一个数字，作为新的丑数。这边新的丑数就是 3*dp[p3]=3*1=3，然后 p3++
- 此时 p5和p7 指向第1个丑数， p3 指向第2个丑数。然后重复上一步
- 这里基于的一个事实是，丑数数列是递增的，当 p5 指针在当前位置时，后面的数乘以5必然比前面的数乘以5大，所以下一个丑数必然是先考虑前面的数乘以5。 p3,p7 同理，所以才可以使用指针

```
class Solution {
    public int getKthMagicNumber(int k) {
        if (k <= 0) return 0;
        int[] arr = new int[k];
        arr[0] = 1;
        int n3 = 0, n5 = 0, n7 = 0;
        for (int i = 1; i < k; i++){
            int min = Math.min(arr[n3] * 3, Math.min(arr[n5] * 5, arr[n
7] * 7));
            arr[i] = min;
            if (arr[n3] * 3 == min) n3++;
            if (arr[n5] * 5 == min) n5++;
            if (arr[n7] * 7 == min) n7++;
        }
        return arr[k-1];
    }
}
```

7、连续整数求和（829）（lintcode1439）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

829. 连续整数求和

难度 困难 72 收藏 分享 切换为英文 关注 反馈

给定一个正整数 `N`，试求有多少组连续正整数满足所有数字之和为 `N`？

示例 1:

```
输入：5
输出：2
解释：5 = 5 = 2 + 3，共有两组连续整数([5],[2,3])求和后为 5。
```

示例 2:

```
输入：9
输出：3
解释：9 = 9 = 4 + 5 = 2 + 3 + 4
```

示例 3:

```
输入：15
输出：4
解释：15 = 15 = 8 + 7 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5
```

说明: `1 <= N <= 10 ^ 9`

https://leetcode-cn.com/problems/consecutive-numbers-sum/solution/lian-xu-zheng-shu-qiu-he-by-leetcode/

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法三：进阶数学

在 $2N = k(2x + k + 1)$ 中，我们可以发现 $k < 2x + k + 1$，因此有 $k < \sqrt{2N}$，即我们只需要枚举 $1 \le k \le \lfloor \sqrt{2N} \rfloor$ 即可，此时通过枚举可以通过本题。

我们还可以继续挖掘一些性质。由于 k 和 $2x + k + 1$ 的奇偶性不同，此时将 $2N$ 写成 $2^{\alpha} * M$ 的形式，其中 α 为 $2N$ 中因子 2 的个数，M 为一个奇数。对于 M 的一种拆分 $M = a * b, a \le b$，可以将 $2N$ 分成奇数 a 和偶数 $2^{\alpha} * b$ 或者奇数 b 和偶数 $2^{\alpha} * a$，每一种分配方法中，将小的那个数给 k，大的那个数给 $2x + k + 1$，就对应了一组解，那么一种拆分方法对应了两组解。如果不限制 $a \le b$，那么可以看作一种拆分方法对应了一组解。有一种特殊情况是 $a = b$，此时这种拆分方法只对应了一组解，但仍然和之前的对应（一对一）相同。因此，我们只需要求出 M 的拆分方法即可，其中 M 为 N 的最大奇因子。M 的拆分方法等价于 M 的因子个数。

方法 1：进阶数学

```
class Solution {
    public int consecutiveNumbersSum(int N) {
        while ((N & 1) == 0) N >>= 1;
        int ans = 1, d = 3;

        while (d * d <= N) {
            int e = 0;
            while (N % d == 0) {
                N /= d;
                e++;
            }
            ans *= e + 1;
            d += 2;
        }

        if (N > 1) ans <<= 1;
        return ans;
    }
}
```

https://leetcode-cn.com/problems/consecutive-numbers-sum/solution/java-shu-xue-by-masterliheng/

方法 2：数学

解题思路

简单的求和思路，假设开始元素为 $a_1 (a_1 >= 0)$, 项数为n, 则这n项的连续和为 $na_1 + \frac{n(n-1)}{2} = N$,
即 $n(2a_1 + n + 1) = 2N$, 由于 $a_1 >= 0$，所以 $n * n <= 2N$, 所以我们可以枚举n的值，然后求满足条件的 a_1 的个数。

```
class Solution {
    public int consecutiveNumbersSum(int N) {
        int res = 0;
        int upper = (int)Math.sqrt(2*N);
        for (int n = 1; n <= upper; ++n)
```

禁止商用 公众号：yes的练级攻略

```
            if (2 * N % n == 0) {
                int y = 2 * N / n - n - 1;
                if (y % 2 == 0 && y >= 0)
                    res++;
            }
        return res;
    }
}
```

8、除数博弈（1025）

1025. 除数博弈

难度 简单 150 收藏 分享 切换为英文 关注 反馈

爱丽丝和鲍勃一起玩游戏，他们轮流行动。爱丽丝先手开局。

最初，黑板上有一个数字 `N` 。在每个玩家的回合，玩家需要执行以下操作：

- 选出任一 `x`，满足 `0 < x < N` 且 `N % x == 0` 。
- 用 `N - x` 替换黑板上的数字 `N` 。

如果玩家无法执行这些操作，就会输掉游戏。

只有在爱丽丝在游戏中取得胜利时才返回 `True`，否则返回 `false`。假设两个玩家都以最佳状态参与游戏。

示例 1：

```
输入：2
输出：true
解释：爱丽丝选择 1，鲍勃无法进行操作。
```

示例 2：

```
输入：3
输出：false
解释：爱丽丝选择 1，鲍勃也选择 1，然后爱丽丝无法进行操作。
```

https://leetcode-cn.com/problems/divisor-game/solution/chu-shu-bo-yi-by-leetcode-solution/

方法一：找规律

思路与算法

博弈类的问题常常让我们摸不着头脑。当我们没有解题思路的时候，不妨试着写几项试试：

- $N = 1$ 的时候，区间 $(0, 1)$ 中没有整数是 n 的因数，所以此时 Alice 败。
- $N = 2$ 的时候，Alice 只能拿 1，N 变成 1，Bob 无法继续操作，故 Alice 胜。
- $N = 3$ 的时候，Alice 只能拿 1，N 变成 2，根据 $N = 2$ 的结论，我们知道此时 Bob 会获胜，Alice 败。
- $N = 4$ 的时候，Alice 能拿 1 或 2，如果 Alice 拿 1，根据 $N = 3$ 的结论，Bob 会失败，Alice 会获胜。
- $N = 5$ 的时候，Alice 只能拿 1，根据 $N = 4$ 的结论，Alice 会失败。
- .....

写到这里，也许你有了一些猜想。没关系，请大胆地猜想，在这种情况下大胆地猜想是 AC 的第一步。也许你会发现这样一个现象：**N 为奇数的时候 Alice（先手）必败，N 为偶数的时候 Alice 必胜。** 这个猜想是否正确呢？下面我们来想办法证明它。

更多硬核内容就在公众号「yes 的练级攻略」

证明

1. $N = 1$ 和 $N = 2$ 时结论成立。
2. $N > 2$ 时，假设 $N \leq k$ 时该结论成立，则 $N = k + 1$ 时：
   - 如果 k 为偶数，则 $k + 1$ 为奇数，x 是 $k + 1$ 的因数，只可能是奇数，而奇数减去奇数等于偶数，且 $k + 1 - x \leq k$，故轮到 Bob 的时候都是偶数。而根据我们的猜想假设 $N \leq k$ 的时候偶数的时候先手必胜，故此时无论 Alice 拿走什么，Bob 都会处于必胜态，所以 Alice 处于必败态。
   - 如果 k 为奇数，则 $k + 1$ 为偶数，x 可以是奇数也可以是偶数，若 Alice 减去一个奇数，那么 $k + 1 - x$ 是一个小于等于 k 的奇数，此时 Bob 占有它，处于必败态，则 Alice 处于必胜态。

综上所述，这个猜想是正确的。

方法 1：找规律

```
class Solution {
    public boolean divisorGame(int N) {
        return N % 2 == 0;
    }
}
```

https://leetcode-cn.com/problems/divisor-game/solution/dong-tai-gui-hua-he-shu-xue-si-wei-by-charles-gao/

动态规划

dp[]代表一个长度为n+1的数组，如果 dp[n-x]为false,则Alice会减去x,即 Bob==dp[n-x]==false, Alice胜。否则Alice输，因为不管x是多少，dp[n-x]为true, 则dp[n]==Alice==false.

方法 2：动规

```
class Solution {
    public boolean divisorGame(int N) {
        if (N == 1) return false;
        if (N == 2) return true;
        boolean[] dp = new boolean[N+1];
        dp[1] = false;
        dp[2] = true;
        for(int i = 3; i <= N; i++){
            dp[i] = false;
            for(int j = 1; j < i; j++){
                if(i % j == 0 && !dp[i - j]){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[N];
    }
}
```

9、Nim 游戏（292）

更多硬核内容就在公众号「yes 的练级攻略」

292. Nim 游戏

难度 简单 347 收藏 分享 切换为英文 关注 反馈

你和你的朋友，两个人一起玩 Nim 游戏：桌子上有一堆石头，每次你们轮流拿掉 1 - 3 块石头。 拿掉最后一块石头的人就是获胜者。你作为先手。

你们是聪明人，每一步都是最优解。 编写一个函数，来判断你是否可以在给定石头数量的情况下赢得游戏。

示例:

```
输入：4
输出：false
解释：如果堆中有 4 块石头，那么你永远不会赢得比赛；
     因为无论你拿走 1 块、2 块 还是 3 块石头，最后一块石头总是会被你的朋友拿走。
```

https://leetcode-cn.com/problems/nim-game/solution/nimyou-xi-by-leetcode/

推理

让我们考虑一些小例子。显而易见的是，如果石头堆中只有一块、两块、或是三块石头，那么在你的回合，你就可以把全部石子拿走，从而在游戏中取胜。而如果就像题目描述那样，堆中恰好有四块石头，你就会失败。因为在这种情况下不管你取走多少石头，总会为你的对手留下几块，使得他可以在游戏中打败你。因此，要想获胜，在你的回合中，必须避免石头堆中的石子数为 4 的情况。

同样地，如果有五块、六块、或是七块石头，你可以控制自己拿取的石头数，总是恰好给你的对手留下四块石头，使他输掉这场比赛。但是如果石头堆里有八块石头，你就不可避免地会输掉，因为不管你从一堆石头中挑出一块、两块还是三块，你的对手都可以选择三块、两块或一块，以确保在再一次轮到你的时候，你会面对四块石头。

显然，它以相同的模式不断重复 $n = 4, 8, 12, 16, \ldots$，基本可以看出是 4 的倍数。

复杂度分析

- 时间复杂度：$O(1)$，只进行了一次检查。
- 空间复杂度：$O(1)$，没有使用额外的空间。

```java
class Solution {
    public boolean canWinNim(int n) {
        return n % 4 != 0;
    }
}
```

10、Pow(x,n)（50）【数值的整数次方（剑指 16）】

更多硬核内容就在公众号「yes 的练级攻略」

50. Pow(x, n)

难度 中等 457 收藏 分享 切换为英文 关注 反馈

实现 pow(*x*, *n*)，即计算 x 的 n 次幂函数。

示例 1:

```
输入: 2.00000, 10
输出: 1024.00000
```

示例 2:

```
输入: 2.10000, 3
输出: 9.26100
```

示例 3:

```
输入: 2.00000, -2
输出: 0.25000
解释: 2^-2 = 1/2^2 = 1/4 = 0.25
```

说明:

- -100.0 < *x* < 100.0
- *n* 是 32 位有符号整数，其数值范围是 $[-2^{31}, 2^{31} - 1]$。

https://leetcode-cn.com/problems/powx-n/solution/powx-n-by-leetcode-solution/

方法一：快速幂 + 递归

「快速幂算法」的本质是分治算法。举个例子，如果我们要计算 x^{64}，我们可以按照：

$$x \to x^2 \to x^4 \to x^8 \to x^{16} \to x^{32} \to x^{64}$$

的顺序，从 x 开始，每次直接把上一次的结果进行平方，计算 6 次就可以得到 x^{64} 的值，而不需要对 x 乘 63 次 x。

再举一个例子，如果我们要计算 x^{77}，我们可以按照：

$$x \to x^2 \to x^4 \to x^9 \to x^{19} \to x^{38} \to x^{77}$$

的顺序，在 $x \to x^2$，$x^2 \to x^4$，$x^{19} \to x^{38}$ 这些步骤中，我们直接把上一次的结果进行平方，而在 $x^4 \to x^9$，$x^9 \to x^{19}$，$x^{38} \to x^{77}$ 这些步骤中，我们把上一次的结果进行平方后，还要额外乘一个 x。

直接从左到右进行推导看上去很困难，因为在每一步中，我们不知道在将上一次的结果平方之后，还需不需要额外乘 x。但如果我们从右往左看，分治的思想就十分明显了：

- 当我们要计算 x^n 时，我们可以先递归地计算出 $y = x^{\lfloor n/2 \rfloor}$，其中 $\lfloor a \rfloor$ 表示对 a 进行下取整；
- 根据递归计算的结果，如果 n 为偶数，那么 $x^n = y^2$；如果 n 为奇数，那么 $x^n = y^2 * x$；
- 递归的边界为 $n = 0$，任意数的 0 次方均为 1。

由于每次递归都会使得指数减少一半，因此递归的层数为 $O(\log n)$，算法可以在很快的时间内得到结果。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(\log n)$，即为递归的层数。
- 空间复杂度：$O(\log n)$，即为递归的层数。这是由于递归的函数调用会使用栈空间。

方法 1：快速幂+递归

```
class Solution {
    public double myPow(double x, int n) {
        long N = n;
        return N >= 0 ? quickMul(x, N) : 1.0 / quickMul(x, -N);
    }
    public double quickMul(double x, long N){
        if (N == 0) return 1.0;
        double y = quickMul(x, N/2);
        return (N & 1) == 0 ? y * y : y * y * x;
    }
}
```

方法二：快速幂 + 迭代

由于递归需要使用额外的栈空间，我们试着将递归转写为迭代。在方法一中，我们也提到过，从左到右进行推导是不容易的，因为我们不知道是否需要额外乘 x。但我们不妨找一找规律，看看哪些地方额外乘了 x，并且它们对答案产生了什么影响。

我们还是以 x^{77} 作为例子：

$$x \to x^2 \to x^4 \to^+ x^9 \to^+ x^{19} \to x^{38} \to^+ x^{77}$$

并且把需要额外乘 x 的步骤打上了 $+$ 标记。可以发现：

- $x^{38} \to^+ x^{77}$ 中额外乘的 x 在 x^{77} 中贡献了 x；
- $x^9 \to^+ x^{19}$ 中额外乘的 x 在之后被平方了 2 次，因此在 x^{77} 中贡献了 $x^{2^2} = x^4$；
- $x^4 \to^+ x^9$ 中额外乘的 x 在之后被平方了 3 次，因此在 x^{77} 中贡献了 $x^{2^3} = x^8$；
- 最初的 x 在之后被平方了 6 次，因此在 x^{77} 中贡献了 $x^{2^6} = x^{64}$。

我们把这些贡献相乘，$x * x^4 * x^8 * x^{64}$ 恰好等于 x^{77}。而这些贡献的指数部分又是什么呢？它们都是 2 的幂次，这是因为每个额外乘的 x 在之后都会被平方若干次。而这些指数 1，4，8 和 64，**恰好就对应了 77 的二进制表示 $(1001101)_2$ 中的每个 1！**

因此我们借助整数的二进制拆分，就可以得到迭代计算的方法，一般地，如果整数 n 的二进制拆分为

$$n = 2^{i0} + 2^{i1} + \cdots + 2^{ik}$$

那么

$$x^n = x^{2^{i0}} * x^{2^{i1}} * \cdots * x^{2^{ik}}$$

这样以来，我们从 x 开始不断地进行平方，得到 $x^2, x^4, x^8, x^{16}, \cdots$，如果 n 的第 k 个（从右往左，从 0 开始计数）二进制位为 1，那么我们就将对应的贡献 x^{2^k} 计入答案。

复杂度分析

- 时间复杂度：$O(\log n)$，即为对 n 进行二进制拆分的时间复杂度。
- 空间复杂度：$O(1)$。

更多硬核内容就在公众号「yes 的练级攻略」

方法 2：快速幂+迭代

```
class Solution {
    double quickMul(double x, long N) {
        double ans = 1.0;
        // 贡献的初始值为 x
        double x_contribute = x;
        // 在对 N 进行二进制拆分的同时计算答案
        while (N > 0) {
            if (N % 2 == 1) {
                // 如果 N 二进制表示的最低位为 1，那么需要计入贡献
                ans *= x_contribute;
            }
            // 将贡献不断地平方
            x_contribute *= x_contribute;
            // 舍弃 N 二进制表示的最低位，这样我们每次只要判断最低位即可
            N /= 2;
        }
        return ans;
    }

    public double myPow(double x, int n) {
        long N = n;
        return N >= 0 ? quickMul(x, N) : 1.0 / quickMul(x, -N);
    }
}
```

11、1~n 整数中 1 出现的次数（面试题 43.1）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 43. 1～n整数中1出现的次数

难度 中等 70 收藏 分享 切换为英文 关注 反馈

输入一个整数 `n` ，求1～n这n个整数的十进制表示中1出现的次数。

例如，输入12，1～12这些整数中包含1 的数字有1、10、11和12，1一共出现了5次。

示例 1：

```
输入：n = 12
输出：5
```

示例 2：

```
输入：n = 13
输出：6
```

限制：

- `1 <= n < 2^31`

注意：本题与主站 233 题相同：https://leetcode-cn.com/problems/number-of-digit-one/

https://leetcode-cn.com/problems/1nzheng-shu-zhong-1chu-xian-de-ci-shu-lcof/solution/mian-shi-ti-43-1n-zheng-shu-zhong-1-chu-xian-de-2/

解题思路：

将 $1\sim n$ 的个位、十位、百位、...的 1 出现次数相加，即为 1 出现的总次数。

设数字 n 是个 x 位数，记 n 的第 i 位为 n_i ，则可将 n 写为 $n_xn_{x-1}\cdots n_2n_1$ ：

- 称 "n_i" 为 **当前位** ，记为 cur ，
- 将 "$n_{i-1}n_{i-2}\cdots n_2n_1$" 称为 **低位** ，记为 low ；
- 将 "$n_xn_{x-1}\cdots n_{i+2}n_{i+1}$" 称为 **高位** ，记为 $high$ 。
- 将 10^i 称为 **位因子** ，记为 $digit$ 。

某位中 1 出现次数的计算方法：

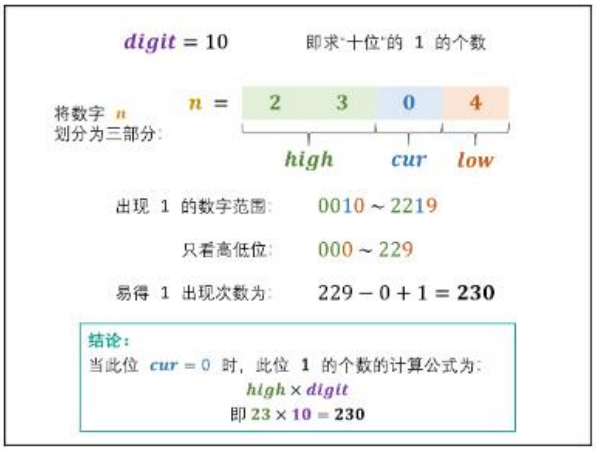
更多硬核内容就在公众号「yes 的练级攻略」

根据当前位 cur 值的不同，分为以下三种情况：

1. 当 $cur = 0$ **时：** 此位 1 的出现次数只由高位 $high$ 决定，计算公式为：

$$high \times digit$$

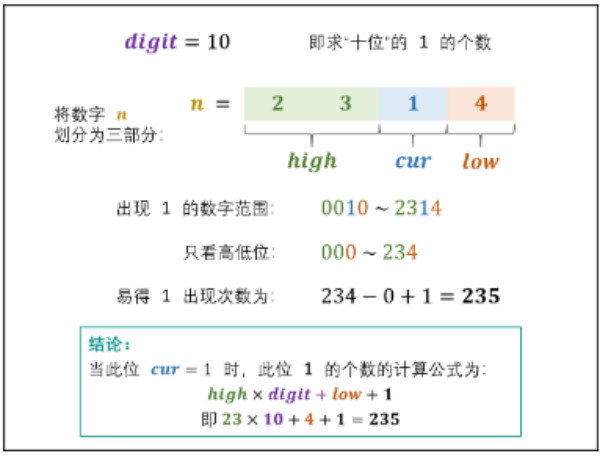
> 如下图所示，以 $n = 2304$ 为例，求 $digit = 10$（即十位）的 1 出现次数。



2. 当 $cur = 1$ **时：** 此位 1 的出现次数由高位 $high$ 和低位 low 决定，计算公式为：

$$high \times digit + low + 1$$

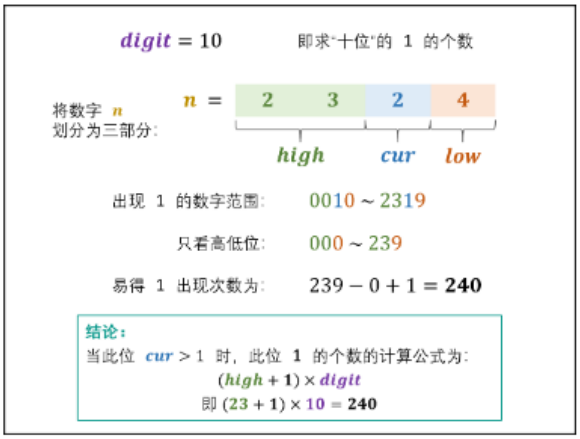
> 如下图所示，以 $n = 2314$ 为例，求 $digit = 10$（即十位）的 1 出现次数。



3. 当 $cur = 2, 3, \cdots, 9$ 时：此位 1 的出现次数只由高位 $high$ 决定，计算公式为：

$$(high + 1) \times digit$$

> 如下图所示，以 $n = 2324$ 为例，求 $digit = 10$（即十位）的 1 出现次数。



变量递推公式：

设计按照 "个位、十位、..." 的顺序计算，则 $high/cur/low/digit$ 应初始化为：

```
high = n // 10
cur = n % 10
low = 0
digit = 1 # 个位
```

因此，从个位到最高位的变量递推公式为：

```
while high != 0 or cur != 0: # 当 high 和 cur 同时为 0 时，说明已经越过最高位，因此跳出
    low += cur * digit # 将 cur 加入 low ，组成下轮 low
    cur = high % 10 # 下轮 cur 是本轮 high 的最低位
    high //= 10 # 将本轮 high 最低位删除，得到下轮 high
    digit *= 10 # 位因子每轮 × 10
```

复杂度分析：

- **时间复杂度** $O(\log n)$：循环内的计算操作使用 $O(1)$ 时间；循环次数为数字 n 的位数，即 $\log_{10} n$，因此循环使用 $O(\log n)$ 时间。
- **空间复杂度** $O(1)$：几个变量使用常数大小的额外空间。

```java
class Solution {
    public int countDigitOne(int n) {
        int digit = 1, res = 0;
        int high = n / 10, cur = n % 10, low = 0;
        while(high != 0 || cur != 0) {
            if(cur == 0) res += high * digit;
            else if(cur == 1) res += high * digit + low + 1;
            else res += (high + 1) * digit;
            low += cur * digit;
            cur = high % 10;
            high /= 10;
            digit *= 10;
```

```
        }
        return res;
    }
}
```

12、数字 1 的个数（233）

233. 数字 1 的个数

难度 困难 148 收藏 分享 切换为英文 关注 反馈

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例:

```
输入: 13
输出: 6
解释: 数字 1 出现在以下数字中: 1, 10, 11, 12, 13 。
```

【这题与上一题的区别在于 n 可能是负数】

```
class Solution {
    public int countDigitOne(int n) {
        if (n <= 0) return 0; //加这一句就行了
        int digit = 1, res = 0;
        int high = n / 10, cur = n % 10, low = 0;
        while(high != 0 || cur != 0) {
            if(cur == 0) res += high * digit;
            else if(cur == 1) res += high * digit + low + 1;
            else res += (high + 1) * digit;
            low += cur * digit;
            cur = high % 10;
            high /= 10;
            digit *= 10;
        }
        return res;
    }
}
```

13、可被 K 整除的最小整数（1015）

更多硬核内容就在公众号「yes 的练级攻略」

1015. 可被 K 整除的最小整数

难度 中等 24 收藏 分享 切换为英文 关注 反馈

给定正整数 `K`，你需要找出可以被 K 整除的、仅包含数字 **1** 的最小正整数 N。

返回 `N` 的长度。如果不存在这样的 `N`，就返回 `-1`。

示例 1：

```
输入：1
输出：1
解释：最小的答案是 N = 1，其长度为 1。
```

示例 2：

```
输入：2
输出：-1
解释：不存在可被 2 整除的正整数 N 。
```

示例 3：

```
输入：3
输出：3
解释：最小的答案是 N = 111，其长度为 3。
```

https://leetcode-cn.com/problems/smallest-integer-divisible-by-k/solution/javajie-fa-yi-ji-zheng-ming-de-si-lu-by-jiangzk/

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

正常思路

通过以下代码，我们可以计算出的答案。首先我们检查 `x` 是否是 `K` 的倍数，如果不是，我们将其乘10加1。

```
while (x % K != 0) {
    x = x * 10 + 1;
}
```

优化方案

容易想到，反复乘10加1很快就会超出 `int` 能表示的范围。

对于任意整数$x > 0$，存在非负整数p和q，使得$x = pK + q$。例如，$x = 2$，$K = 5$时，可以令$p = 0$，$q = 2$。又例如，$x = 29$，$K = 7$时，可以令$p = 4$，$q = 1$。

我们在计算答案时，使用了以下公式：

$$x_{i+1} = 10x_i + 1$$

请注意代码与公式的不同。代码中，我们把x_{i+1}的值又保存在了x_i的位置，覆盖了其原来的值。为了使得证明过程更加清晰明了，我们在公式中体现这种区别。

我们将前一条公式代入，因此有了：

$$10x_i + 1 = 10(pK + q) + 1 = 10pK + 10q + 1$$

由于等号两边相等，那么在两边分别对K取余，其结果也应该相等。

$$(10x_i + 1)\%K = (10pK + 10q + 1)\%K$$

观察等式右侧，有一项$10pK$，由于它是K的倍数，因此无论p为何值，这一项都不会影响最终结果，因此将它去掉，得到：

$$(10x_i + 1)\%K = (10q + 1)\%K$$

由于$x = pK + q$，我们能够推出$x\%K = q$，带入上式：

$$(10x_i + 1)\%K = (10(x_i\%K) + 1)\%K$$

再看一眼上一节中的代码：

```
while (x % K != 0) {
    x = x * 10 + 1;
}
```

最后的公式告诉我们，`x * 10 + 1` 和 `(x % K) * 10 + 1` 在后续判断 `x % K != 0` 时是没有任何区别的。因此我们可以将代码改成：

```
while (x % K != 0) {
    x = x % K;
    x = x * 10 + 1;
}
```

这样就可以避免 `x` 超范围了。

```
class Solution {
    public static int smallestRepunitDivByK(int K) {
        if (K % 2 == 0 || K % 5 == 0) {
            return -1;
        }
        int temp = 1;
        int len = 1;
        while (temp % K != 0) {
```

```
            temp = temp % K;
            temp = temp * 10 + 1;
            len += 1;
        }
        return len;
    }
}
```

14、灯泡开关（319）

319. 灯泡开关

难度 中等 123 收藏 分享 切换为英文 关注 反馈

初始时有 *n* 个灯泡关闭。 第 1 轮，你打开所有的灯泡。 第 2 轮，每两个灯泡你关闭一次。 第 3 轮，每三个灯泡切换一次开关（如果关闭则开启，如果开启则关闭）。第 *i* 轮，每 *i* 个灯泡切换一次开关。 对于第 *n* 轮，你只切换最后一个灯泡的开关。 找出 *n* 轮后有多少个亮着的灯泡。

示例:

输入： 3
输出： 1
解释：
初始时，灯泡状态 [**关闭，关闭，关闭**].
第一轮后，灯泡状态 [**开启，开启，开启**].
第二轮后，灯泡状态 [**开启，关闭，开启**].
第三轮后，灯泡状态 [**开启，关闭，关闭**].

你应该返回 1，因为只有一个灯泡还亮着。

https://leetcode-cn.com/problems/bulb-switcher/solution/qiu-ping-fang-gen-by-feng-ji-wei-wang/

我刚开始是使用数组做的，超出了内存的限制，后来看了deen大佬的题解，这里只是做一个总结
对于这道题，首先分析对于第i个灯泡，只有它的因子轮的操作可以改变它的状态，例如4号灯泡，就只会在1,2,4这些轮改变，5号灯泡，就只会在1,5这两轮改变，因为初始的所有的灯泡的状态都为关闭，所以如果一个灯泡含有奇数个因子，那么在这些操作之后他应该是开启的，相对的，如果是偶数个因子，那就是关闭的，所以问题此时转化为了看1-n的所有的灯泡有几个因子
然后去分析每个数的因子，看它们因子的个数，但是这里有一个可以优化的点就是对于一个数例如 a=b*c,b和c一定是在根号a的两端或者就等于根号a，那么统计它们的因子的个数只需要让j从1到根号a，如果灯泡序号可以整除这个j，那么因子数加2,当然如果两个因子数一样（也就是这个数字可以开平方）就加1，由此可以总结出来的一个规律是对于能开平方的数字，它的因子数一定是一个奇数，而对于不能开平方数字，它的因子数为一个偶数，所以又将问题转化为找出1-n中可以开平方的数
对于1-n中的可以开平方的数字，例如36，它里面可以开平方的数字1*1，2*2...6*6，对于100，它里面可以开平方的数字，1*1,...10*10,可以发现，对于一个数，它里面可以开平方的数字的个数就是它自己的开平方数，所以这道题的最终的解法就变为了直接求n的开平方数就是最后的结果

```
class Solution {
    public int bulbSwitch(int n) {
        return (int) Math.sqrt(n);
    }
}
```

15、月份天数（lintcode1141）

1141. 月份天数

给定年份和月份，返回这个月的天数。

样例
样例 1:

输入：
2020
2
输出：
29
样例 2:

输入：
2020
3
输出：
31
注意事项
1 ≤ *year* ≤ 10000
1 ≤ *month* ≤ 12

```
public class Solution {
    /**
     * @param year: a number year
     * @param month: a number month
     * @return: Given the year and the month, return the number of days of the month.
     */
    public int getTheMonthDays(int year, int month) {
        // write your code here
        int[] month_run = {31,29,31,30,31,30,31,31,30,31,30,31};
        int[] month_nor = {31,28,31,30,31,30,31,31,30,31,30,31};
        if (month == 2) {
            return (year % 4 == 0 && year %100 != 0) || year % 400 == 0 ? month_run[month-1] : month_nor[month-1];
        } else {
            return month_nor[month-1];
        }
    }
}
```

16、圆圈中最后剩下的数字（剑指 62）【约瑟夫环问题】

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 62. 圆圈中最后剩下的数字

难度 简单 193 收藏 分享 切换为英文 关注 反馈

0,1,,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1：

```
输入：n = 5, m = 3
输出：3
```

示例 2：

```
输入：n = 10, m = 17
输出：2
```

限制：

- `1 <= n <= 10^5`
- `1 <= m <= 10^6`

https://leetcode-cn.com/problems/yuan-quan-zhong-zui-hou-sheng-xia-de-shu-zi-lcof/solution/javajie-jue-yue-se-fu-huan-wen-ti-gao-su-ni-wei-sh/

解题思路：

> 这个问题是以弗拉维奥·约瑟夫命名的，他是1世纪的一名犹太历史学家。他在自己的日记中写道，他和他的40个战友被罗马军队包围在洞中。他们讨论是自杀还是被俘，最终决定自杀，并以抽签的方式决定谁杀掉谁。约瑟夫斯和另外一个人是最后两个留下的人。约瑟夫斯说服了那个人，他们将向罗马军队投降，不再自杀。约瑟夫斯把他的存活归因于运气或天意，他不知道是哪一个。 —— 【约瑟夫问题】维基百科

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

如果单纯用链表模拟的话，时间复杂度是 $O(nm)$ 的，可以看下题目的数据范围，肯定是不能这么做的。关于运行时间的预估，经验是如果 $n < 10^5$ ，那么 $O(n^2)$ 的解法耗时大概是几秒左右（当然时间复杂度会忽略常数，而且也有可能由于执行程序的机器性能的不同，$O(n^2)$ 的实际耗时也有可能一秒多，也有可能十几秒）。本题由于 $1 <= m <= 10^6$ ，所以 $O(nm)$ 肯定是超时的。

可以分析下纯暴力的做法，每次找到删除的那个数字，需要 $O(m)$ 的时间复杂度，然后删除了 $n-1$ 次。**但实际上我们可以直接找到下一个要删除的位置的！**

假设当前删除的位置是 idx，下一个删除的数字的位置是 $idx+m$。但是，由于把当前位置的数字删除了，后面的数字会前移一位，所以实际的下一个位置是 $idx+m-1$。由于数到末尾会从头继续数，所以最后取模一下，就是 $(idx+m-1) \pmod n$。

至于这种思路的代码实现，我尝试了下 `LinkedList` 会超时，我猜是因为 `LinkedList` 虽然删除指定节点的时间复杂度是 $O(1)$ 的，但是在 `remove` 时间复杂度仍然是 $O(n)$ 的，**因为需要从头遍历到需要删除的位置。**那 `ArrayList` 呢？索引到需要删除的位置，时间复杂度是 $O(1)$，删除元素时间复杂度是 $O(n)$（因为后续元素需要向前移位），`remove` 整体时间复杂度是 $O(n)$ 的。看起来 `LinkedList` 和 `ArrayList` 单次删除操作的时间复杂度是一样的？所累哇多卡纳！`ArrayList` 的 `remove` 操作在后续移位的时候，**其实是内存连续空间的拷贝的！所以相比于 `LinkedList` 大量非连续性地址访问，`ArrayList` 的性能是很 OK 的！**

这是 `ArrayList` 的 `remove` 源码：

```
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, srcPos: index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

每次删除的时间复杂度是 $O(n)$，删除了 $n-1$ 次，所以整体时间复杂度是 $O(n^2)$。leetcode 上该方法勉强可以通过，大概是 1s 多一点。

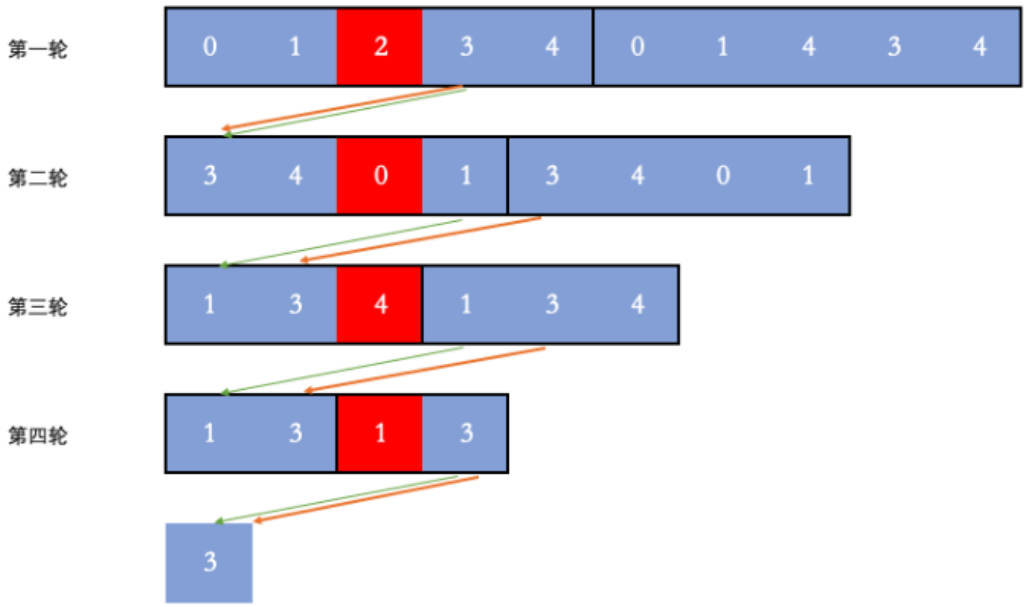
方法 1：模拟链表（1074ms）

```
class Solution {
    public int lastRemaining(int n, int m) {
        ArrayList<Integer> list = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            list.add(i);
        }
        int idx = 0;
        while (n > 1) {
            idx = (idx + m - 1) % n;
            list.remove(idx);
            n--;
        }
        return list.get(0);
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

数学解法，$O(n)$

这么著名的约瑟夫环问题，是有数学解法的！
因为数据是放在数组里，所以我在数组后面加上了数组的复制，以体现是环状的。我们先忽略图片里的箭头：
【第一轮后面的数字应该是 `[0, 1, 2 ,3 ,4]`，手误打错了。。抱歉】



很明显我们每次删除的是第 m 个数字，我都标红了。

第一轮是 `[0, 1, 2, 3, 4]`，所以是 `[0, 1, 2, 3, 4]` 这个数组的多个复制。这一轮 `2` 删除了。

第二轮开始时，从 `3` 开始，所以是 `[3, 4, 0, 1]` 这个数组的多个复制。这一轮 `0` 删除了。

第三轮开始时，从 `1` 开始，所以是 `[1, 3, 4]` 这个数组的多个复制。这一轮 `4` 删除了。

第四轮开始时，还是从 `1` 开始，所以是 `[1, 3]` 这个数组的多个复制。这一轮 `1` 删除了。

最后剩下的数字是 `3`。

图中的绿色的线指的是新的一轮的开头是怎么指定的，每次都是固定地向前移位 m 个位置。

然后我们从最后剩下的 `3` 倒着看，我们可以反向推出这个数字在之前每个轮次的位置。

最后剩下的 `3` 的下标是 `0`。

第四轮反推，补上 m 个位置，然后模上当时的数组大小 2，位置是 `(0 + 3) % 2 = 1`。

第三轮反推，补上 m 个位置，然后模上当时的数组大小 3，位置是 `(1 + 3) % 3 = 1`。

第二轮反推，补上 m 个位置，然后模上当时的数组大小 4，位置是 `(1 + 3) % 4 = 0`。

第一轮反推，补上 m 个位置，然后模上当时的数组大小 5，位置是 `(0 + 3) % 5 = 3`。

所以最终剩下的数字的下标就是 `3`。因为数组是从0开始的，所以最终的答案就是 `3`。

总结一下反推的过程，就是 `(当前index + m) % 上一轮剩余数字的个数`。

代码就很简单了。

方法 2：数学（7ms）

```
class Solution {
    public int lastRemaining(int n, int m) {
        if (n < 1 || m < 1) return -1;
        int last = 0;
        for (int i = 2; i <= n; i++){
            last = (last + m) % i;
        }
        return last;
    }
}
```

17、第 N 个数字（400）【数字序列中某一位的数字（剑指 44）】

https://leetcode-cn.com/problems/shu-zi-xu-lie-zhong-mou-yi-wei-de-shu-zi-lcof/solution/mian-shi-ti-44-shu-zi-xu-lie-zhong-mou-yi-wei-de-6/

解题思路：

1. 将 $101112\cdots$ 中的每一位称为 **数位** ，记为 n ；
2. 将 $10, 11, 12, \cdots$ 称为 **数字** ，记为 num ；
3. 数字 10 是一个两位数，称此数字的 **位数** 为 2 ，记为 $digit$ ；
4. 每 $digit$ 位数的起始数字（即：$1, 10, 100, \cdots$），记为 $start$ 。

数字范围	位数	数字数量	数位数量
1~9	1	9	9
10~99	2	90	180
100~999	3	900	2700
...	...	...	...
start~*end*	*digit*	$9 \times start$	$9 \times start \times digit$

位数递推公式 $digit = digit + 1$

起始数字递推公式 $start = start \times 10$

数位数量计算公式 $count = 9 \times start \times digit$

观察上表，可推出各 $digit$ 下的数位数量 $count$ 的计算公式：

$$count = 9 \times start \times digit$$

根据以上分析，可将求解分为三步：

1. 确定 n 所在 **数字** 的 **位数** ，记为 $digit$ ；
2. 确定 n 所在的 **数字** ，记为 num ；
3. 确定 n 是 num 中的哪一数位，并返回结果。

更多硬核内容就在公众号「yes 的练级攻略」

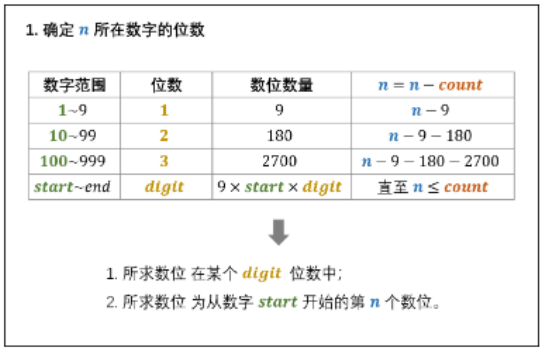
1. 确定所求数位的所在数字的位数

如下图所示，循环执行 n 减去 一位数、两位数、... 的数位数量 $count$ ，直至 $n \leq count$ 时跳出。

由于 n 已经减去了一位数、两位数、...、$(digit-1)$ 位数的 **数位数量** $count$ ，因而此时的 n 是从起始数字 $start$ 开始计数的。

```
digit, start, count = 1, 1, 9
while n > count:
    n -= count
    start *= 10 # 1,  10,  100, ...
    digit += 1  # 1,   2,   3, ...
    count = 9 * start * digit # 9, 180, 2700, ...
```

结论： 所求数位 ① 在某个 $digit$ 位数中； ② 为从数字 $start$ 开始的第 n 个数位。

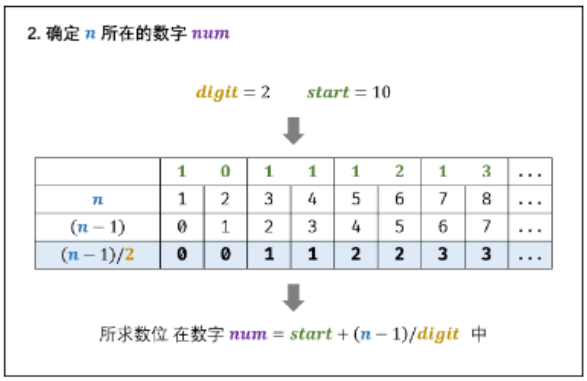


2. 确定所求数位所在的数字

如下图所示，所求数位 在从数字 $start$ 开始的第 $[(n-1)/digit]$ 个 **数字** 中（ $start$ 为第 0 个数字）。

```
num = start + (n - 1) // digit
```

结论： 所求数位在数字 num 中。

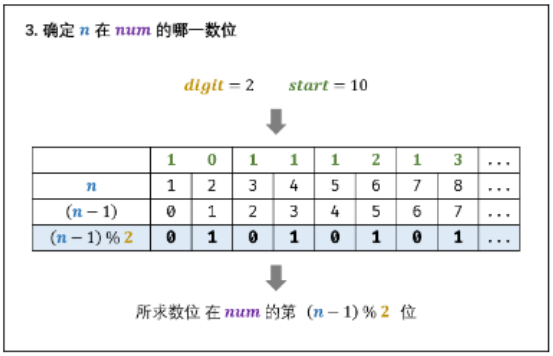


3. 确定所求数位在 num 的哪一数位

如下图所示，所求数位为数字 num 的第 $(n-1)\%digit$ 位（数字的首个数位为第 0 位）。

```
s = str(num) # 转化为 string
res = int(s[(n - 1) % digit]) # 获得 num 的 第 (n - 1) % digit 个数位，并转化为 int
```

结论： 所求数位是 res 。



复杂度分析：

- **时间复杂度** $O(\log n)$ ： 所求数位 n 对应数字 num 的位数 $digit$ 最大为 $O(\log n)$ ；第一步最多循环 $O(\log n)$ 次；第三步中将 num 转化为字符串使用 $O(\log n)$ 时间；因此总体为 $O(\log n)$ 。
- **空间复杂度** $O(\log n)$ ： 将数字 num 转化为字符串 `str(num)` ，占用 $O(\log n)$ 的额外空间。

```java
class Solution {
    public int findNthDigit(int n) {
        int digit = 1;
        long start = 1, cnt = 9;
        while (n > cnt) {
            n -= cnt;
            digit++;
            start *= 10;
            cnt = digit * start * 9;
        }
        long num = start + (n - 1) / digit;
        return Long.toString(num).charAt((n-1) % digit) - '0';
    }
}
```

18、求 1+2+...+n（剑指 64）

更多硬核内容就在公众号「yes 的练级攻略」

剑指 Offer 64. 求1+2+...+n

难度 中等 181

求 1+2+...+n ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

示例 1:

```
输入: n = 3
输出: 6
```

示例 2:

```
输入: n = 9
输出: 45
```

限制:

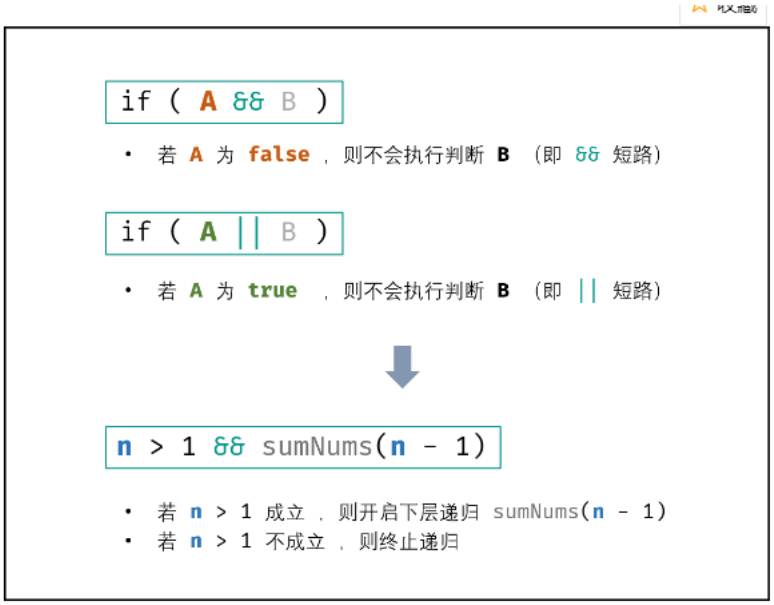
- 1 <= n <= 10000

https://leetcode-cn.com/problems/qiu-12n-lcof/solution/mian-shi-ti-64-qiu-1-2-nluo-ji-fu-duan-lu-qing-xi-/

方法三： 递归
问题： 终止条件需要使用 if ，因此本方法不可取。
思考： 除了 if 和 $switch$ 等判断语句外，是否有其他方法可用来终止递归?

java | python

```java
public int sumNums(int n) {
    if(n == 1) return 1;
    n += sumNums(n - 1);
    return n;
}
```



逻辑运算符的短路效应：

常见的逻辑运算符有三种，即“与 &&”，“或 ||”，“非 !”；而其有重要的短路效应，如下所示：

```
if(A && B)  // 若 A 为 false ，则 B 的判断不会执行（即短路），直接判定 A && B 为 false

if(A || B) // 若 A 为 true ，则 B 的判断不会执行（即短路），直接判定 A || B 为 true
```

本题需要实现“当 $n = 1$ 时终止递归”的需求，可通过短路效应实现。

```
n > 1 && sumNums(n - 1) // 当 n = 1 时 n > 1 不成立 ，此时 “短路” ，终止后续递归
```

复杂度分析：

- **时间复杂度 $O(n)$：** 计算 $n + (n-1) + ... + 2 + 1$ 需要开启 n 个递归函数。
- **空间复杂度 $O(n)$：** 递归深度达到 n，系统使用 $O(n)$ 大小的额外空间。

代码：

1. Java 中，为构成语句，需加一个辅助布尔量 x，否则会报错；
2. Java 中，开启递归函数需改写为 `sumNums(n - 1) > 0`，此整体作为一个布尔量输出，否则会报错；
3. 初始化变量 res 记录结果。（Java 可使用第二栏的简洁写法，不用借助变量 res）。

```
class Solution {
    public int sumNums(int n) {
        boolean x = n > 1 && (n += sumNums(n - 1)) > 0;
        return n;
    }
}
```

19、交点（面试题 16.03）【两个浮点型大小的判断】

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.03. 交点

难度 困难 55 收藏 分享 切换为英文 关注 反馈

给定两条线段（表示为起点 `start = {X1, Y1}` 和终点 `end = {X2, Y2}`），如果它们有交点，请计算其交点，没有交点则返回空值。

要求浮点型误差不超过 `10^-6`。若有多个交点（线段重叠）则返回 X 值最小的点，X 坐标相同则返回 Y 值最小的点。

示例 1：

```
输入：
line1 = {0, 0}, {1, 0}
line2 = {1, 1}, {0, -1}
输出： {0.5, 0}
```

示例 2：

```
输入：
line1 = {0, 0}, {3, 3}
line2 = {1, 1}, {2, 2}
输出： {1, 1}
```

禁止商用

https://leetcode-cn.com/problems/intersection-lcci/solution/shi-yong-zhi-xian-xie-jie-shi-fang-cheng-zai-pan-d/

思路：先把线段看成直线，求出交点的坐标，再检测交点是否在两条线段之外。

说明：这里用「斜截式」参数比较少，公式的计算量也比较少。**缺点是要做斜率不存在的情况的讨论**，书上没有写，这部分代码是我加上去的，仅供参考。

总结：我也不懂这个题来自「计算几何」这门学科，做这个问题我感觉比较锻炼编码的能力，像是在写工程代码一样。如果封装得比较好的话（当然这是书上的做法），代码的逻辑还是会比较清晰的。

下面是具体步骤：

- 第 1 步：计算斜率和截距

根据两个端点的坐标，计算线段所在直线的表达式，这里使用「斜截式」 $y = kx + b$（斜率为 0 的情况单独做讨论），假设线段的两个端点的坐标分别为 (x_1, y_1) 和(x_2, y_2)，则 $k = \frac{y_2 - y_1}{x_2 - x_1}$，然后将 (x_1, y_1) 或 (x_2, y_2) 代入 $y = kx + b$ 可以得到直线在 y 轴上的截距 b；

- 第 2 步：计算两条「直线」的交点坐标

假设两条直线的方程分别为：$y = k_1x + b_1$、$y = k_2x + b_2$，交点的计算公式为：$x = -\frac{b_2 - b_1}{k_2 - k_1}$，把 x 代入其中一条直线的表达式，得到 y。

- 第 3 步：检测交点的横、纵坐标是否在两条线段的横纵坐标范围之内。

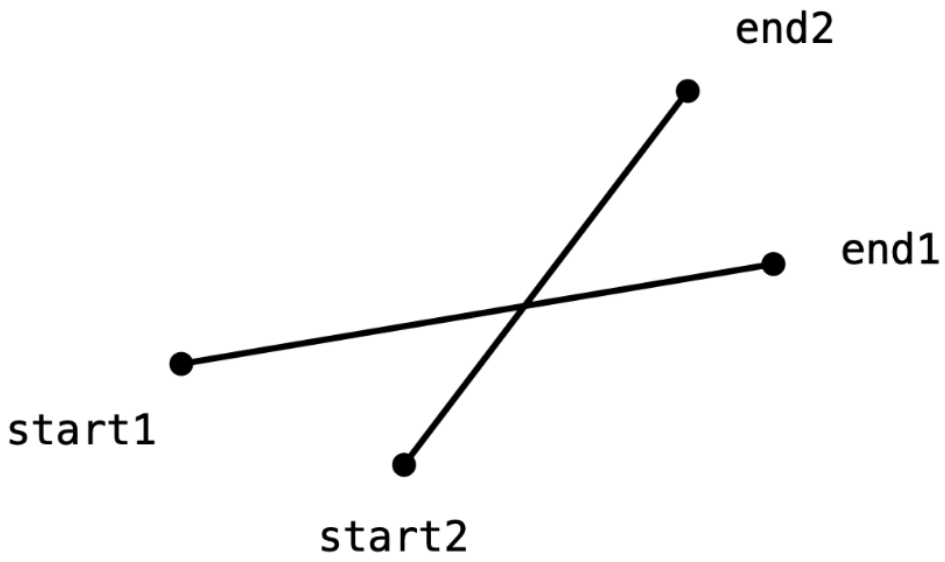
细节：

1、假设四个点的坐标大小关系是固定的，即：

（1）对于单条线段而言，起点坐标总是横坐标较小的那一个；
（2）对于两条线段而言，线段 1 的横坐标小于等于线段 2 的横坐标；

收藏

更多硬核内容就在公众号「yes 的练级攻略」



这里可以认为是参数校验或者说规定了一种顺序，方便后面检测交点是否在两条线段的横纵坐标范围之内。

2、将点封装成 `Point` 类，由于输入是 int 类型，计算得用浮点类型，因此在封装成 `Point` 类的同时就做了类型转换；

3、将两点所在的直线封装成 `Line` 类，以便计算直线的斜率和截距，和后面计算交点时直接使用；

4、讨论特殊情况：

（1）斜率为正无穷，即横坐标相等的线段，斜率记为正无穷，截距在这里记录成为 `a`， `a` 就是这个相同的横坐标。

两条线段只要其中之一斜率为正无穷，交点的情况都比较方便计算；

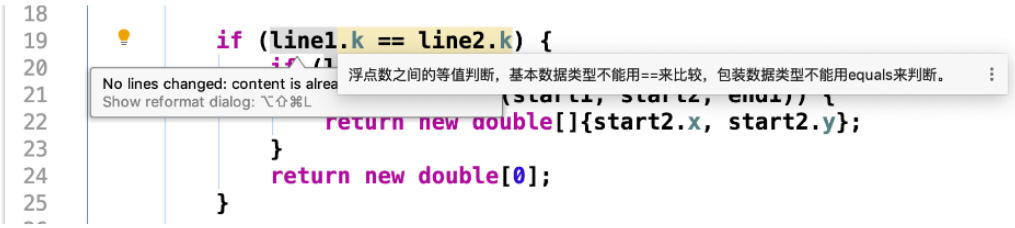
（2）斜率相等的时候，如果截距也相等，就说明两条线段在一条直线上，由于我们在第 1 点假设了输入数据之间的相对关系，这里应该将 `(start2, end2)` 设计为交点，以便后续检查交点是否在有效的范围内；

（3）斜率都不是正无穷，并且截距还不相等的情况，就是最一般的情况；

5、判断交点是否在两条线段横纵坐标的有效范围内，如果不在，就说明没有交点；

6、浮点数之间的等值判断，基本数据类型不能用 `==` 来比较，包装数据类型不能用 `equals` 来判断。

下面是 IDEA 这个工具给的提示。



浮点数之间的等值判断，基本数据类型不能用==来比较，包装数据类型不能用equals来判断。

Inspection info:

浮点数之间的等值判断，基本数据类型不能用==来比较，包装数据类型不能用equals来判断。
浮点数采用“尾数+阶码”的编码方式，类似于科学计数法的“有效数字+指数”的表示方式。二进制无法精确表示大部分的十进制小数，具
改进方式：
1）指定一个误差范围，两个浮点数的差值在此范围之内，则认为是相等的

```
    float a = 1.0f - 0.9f;
    float b = 0.9f - 0.8f;
    float diff = 1e-6f;

    if (Math.abs(a - b) < diff) {
        System.out.println("true");
    }
}
```

2）使用BigDecimal来定义值，再进行浮点数的运算操作

```
    BigDecimal a = new BigDecimal("1.0");
    BigDecimal b = new BigDecimal("0.9");
    BigDecimal c = new BigDecimal("0.8");

    BigDecimal x = a.subtract(b);
    BigDecimal y = b.subtract(c);

    if (x.equals(y)) {
        System.out.println("true");
    }
```

Negative example:

```
        float g = 0.7f-0.6f;
        float h = 0.8f-0.7f;
        if (g == h) {
            System.out.println("true");
        }
```

Positive example:

```
        double dis = 1e-6;
        double d1 = 0.0000001d;
        double d2 = 0d;
        System.out.println(Math.abs(d1 - d2) < dis);
```

```
import java.util.Arrays;
public class Solution {
    public double[] intersection(int[] start1_, int[] end1_, int[] star
t2_, int[] end2_) {
        // 为了避免给后面使用造成不便，修改了入参声明
        Point[] points = checkAndConvertIntoPoint(start1_, end1_, start
2_, end2_);
        Point start1 = points[0];
        Point end1 = points[1];
        Point start2 = points[2];
        Point end2 = points[3];
```

```
        // 封装成直线类，以便计算斜率和截距
        Line line1 = new Line(start1, end1);
        Line line2 = new Line(start2, end2);
        // 误差精度
        double epslion = 1e-6f;
        // 两条直线的交点（如果存在的话）
        Point intersection;
        // 情况 1：（特殊情况）两条直线有一条斜率为正无穷
        if (line1.k == Integer.MAX_VALUE || line2.k == Integer.MAX_VALU
E) {
            // 两条直线斜率都不存在时
            if (line1.k == Integer.MAX_VALUE && line2.k == Integer.MAX_
VALUE) {
                // 这里讨论两条直线重合的情况， b 不是截距的意思，而是表
示 x = a 这条线段
                if (Math.abs(line1.b - line2.b) <= epslion && (isBetwee
n(start1, start2, end1)) || isBetween(start2, start1, end2)) {
                    if (isBetween(start1, start2, end1)) {
                        return new double[]{start2.x, start2.y};
                    } else {
                        return new double[]{start1.x, start1.y};
                    }
                }
            }
            // 其中一条直线斜率不存在
            if (line1.k == Integer.MAX_VALUE) {
                intersection = new Point(line1.b, line1.b * line2.k + l
ine2.b);
            } else {
                intersection = new Point(line2.b, line2.b * line1.k + l
ine1.b);
            }
        } else if (Math.abs(line1.k - line2.k) <= epslion) {
            // 情况 2：（特殊情况）斜率相等的情况下，如果在 y 轴上的截距相等，
就表示两条直线重合
            if (Math.abs(line1.b - line2.b) <= epslion && isBetween(sta
rt1, start2, end1)) {
                return new double[]{start2.x, start2.y};
            }
            return new double[0];
        } else {
            // 情况 3：（一般情况）使用公式计算交点的坐标
            double x = (line2.b - line1.b) / (line1.k - line2.k);
            double y = x * line1.k + line1.b;
```

```
                intersection = new Point(x, y);
            }
            // 检测所在直线的交点是否在两条线段的横纵坐标范围之内
            if (isBetween(start1, intersection, end1) && isBetween(start2,
intersection, end2)) {
                return new double[]{intersection.x, intersection.y};
            }
            return new double[0];
        }
        private Point[] checkAndConvertIntoPoint(int[] start1_, int[] end1_
, int[] start2_, int[] end2_) {
            // 封装成 Point 类，将 int 转换成 double 类型，以便于计算
            Point start1 = new Point(start1_[0], start1_[1]);
            Point end1 = new Point(end1_[0], end1_[1]);
            Point start2 = new Point(start2_[0], start2_[1]);
            Point end2 = new Point(end2_[0], end2_[1]);
            // 参数校验：保证横坐标符合约定
            // 对于单条线段而言，起点坐标总是横坐标较小的那一个
            if (start1.x > end1.x) {
                swap(start1, end1);
            }
            if (start2.x > end2.x) {
                swap(start2, end2);
            }
            // 对于两条线段而言，线段 1 的横坐标小于等于线段 2 的横坐标
            if (start1.x > start2.x) {
                // 两条线段交换
                swap(start1, start2);
                swap(end1, end2);
            }
            return new Point[]{start1, end1, start2, end2};
        }
        private boolean isBetween(double start, double middle, double end)
{
            if (start > end) {
                // 逆序
                return end <= middle && middle <= start;
            } else {
                // 顺序
                return start <= middle && middle <= end;
            }
        }

        private boolean isBetween(Point start, Point middle, Point end) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        return isBetween(start.x, middle.x, end.x) && isBetween(start.y
, middle.y, end.y);
    }
    /**
     * 交换两个点坐标的数值
     *
     * @param point1
     * @param point2
     */
    private void swap(Point point1, Point point2) {
        double tempX = point1.x;
        double tempY = point1.y;
        point1.x = point2.x;
        point1.y = point2.y;
        point2.x = tempX;
        point2.y = tempY;
    }
    private class Line {
        /**
         * 斜率
         */
        private double k;

        /**
         * 在 y 轴上的截距
         */
        private double b;

        public Line(Point start, Point end) {
            double deltaY = end.y - start.y;
            double deltaX = end.x - start.x;

            // 注意：当 deltaX = 0 的时候，设置斜率为正无穷
            if (deltaX == 0) {
                k = Integer.MAX_VALUE;
                // 此时截距为直线在 x 轴上的截距，这里是特殊的做法
                b = end.x;
            } else {
                k = deltaY / deltaX;
                b = end.y - k * end.x;
            }
        }
    }
    /**
```

```
     * 将输入封装成 Point，以便把 int 类型转换成 double 类型，便于计算
     */
    private class Point {
        private double x;
        private double y;
        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
    }
    public static void main(String[] args) {

//        int[] start1 = new int[]{0, 0};
//        int[] end1 = new int[]{1, 0};
//
//        int[] start2 = new int[]{1, 1};
//        int[] end2 = new int[]{0, -1};

//        int[] start1 = new int[]{0, 0};
//        int[] end1 = new int[]{0, 1};
//
//        int[] start2 = new int[]{0, 0};
//        int[] end2 = new int[]{0, -1};
        int[] start1 = new int[]{0, 0};
        int[] end1 = new int[]{0, 6};
        int[] start2 = new int[]{0, 1};
        int[] end2 = new int[]{0, 5};
        Solution solution = new Solution();
        double[] res = solution.intersection(start1, end1, start2, end2
);
        System.out.println(Arrays.toString(res));
    }
}
```

20、平分正方形（面试题 16.13）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 16.13. 平分正方形

难度 中等 2 收藏 分享 切换为英文 关注 反馈

给定两个正方形及一个二维平面。请找出将这两个正方形分割成两半的一条直线。假设正方形顶边和底边与 x 轴平行。

每个正方形的数据 `square` 包含3个数值，正方形的左下顶点坐标 `[X, Y] = [square[0], square[1]]`，以及正方形的边长 `square[2]`。所求直线穿过两个正方形会形成4个交点，请返回4个交点形成线段的两端点坐标（两个端点即为4个交点中距离最远的2个点，这2个点所连成的线段一定会穿过另外2个交点）。2个端点坐标 $[X_1, Y_1]$ 和 $[X_2, Y_2]$ 的返回格式为 $\{X_1, Y_1, X_2, Y_2\}$，要求若 X_1 != X_2，需保证 X_1 < X_2，否则需保证 Y_1 <= Y_2。

若同时有多条直线满足要求，则选择斜率最大的一条计算并返回（与Y轴平行的直线视为斜率无穷大）。

示例：

输入：
square1 = {-1, -1, 2}
square2 = {0, -1, 2}
输出： {-1,0,2,0}
解释： 直线 y = 0 能将两个正方形同时分为等面积的两部分，返回的两线段端点为[-1,0]和[2,0]

提示：

- `square.length == 3`
- `square[2] > 0`

https://leetcode-cn.com/problems/bisect-squares-lcci/solution/chun-shu-xue-ti-zhu-yao-fen-san-chong-qing-kuang-t/

思路

两个正方形的中心点确定的那条直线就是我们所要求的直线。主要可以分为三种情况：

1. 两个中心点重合，或者直线的斜率无穷大（即平行于y轴）
2. 直线的斜率绝对值小于1
3. 直线的斜率绝对值大于等于1

其实题目所说的如果有多条直线满足要求，只会出现在情况1的两个中心点重合，这个时候我们取直线平行于y轴即可。

问题来了，为什么要把斜率绝对值大于1小于1分开讨论呢?
其实就是为了简化我们的情况，因为按照正常情况下，我们要计算边界点，必须要将这条直线与8条边相交，得到8个点（其中有4个是无效点），然后从剩下的4个点中找出边界点。如果我们图形不是正方形，而是普通的矩形的话，我们将不得不这么做。
但是，题目给的是正方形，我们知道，中心点到题目给出的正方形（题目给的正方形的边是平行于x或y轴的）的各个顶点的斜率的绝对值都是等于1的，这就意味着：

1. 斜率小于1 只会与正方形的左右两条边相交
2. 斜率大于1 只会与正方形的上下两条边相交
   由这点性质，我们就可以只用与4条边相交即可得出答案。

```
class Solution {
    public double[] cutSquares(int[] square1, int[] square2) {
        //计算两个正方形的中心点
        double[] circle1 = {square1[0] + square1[2]/2.0,
                            square1[1] + square1[2]/2.0};
        double[] circle2 = {square2[0] + square2[2]/2.0,
                            square2[1] + square2[2]/2.0};
        double[] ans = new double[4];
```

```
        if(circle1[0] == circle2[0]) {
            //垂直的直线
            ans[0] = ans[2] = circle1[0];
            ans[1] = Math.min(square1[1], square2[1]);
            ans[3] = Math.max(square1[1] + square1[2], square2[1] + squ
are2[2]);
            return ans;
        }
        //计算斜率
        double k = (circle2[1] - circle1[1]) / (circle2[0] - circle1[0]
);
        if(Math.abs(k) < 1) {
            //和垂直的边相交
            double leftX = Math.min(square1[0], square2[0]);
            double rightX = Math.max(square1[0] + square1[2], square2[0
] + square2[2]);
            double leftY = k * (leftX - circle1[0]) + circle1[1];
            double rightY = k * (rightX - circle1[0]) + circle1[1];
            return new double[]{leftX, leftY, rightX, rightY};
        } else {
            //和水平的边相交
            double bottonY = Math.min(square1[1], square2[1]);
            double topY = Math.max(square1[1] + square1[2], square2[1]
+ square2[2]);
            double bottonX = (bottonY - circle1[1]) / k + circle1[0];
            double topX = (topY - circle1[1]) / k + circle1[0];
            return bottonX < topX ? new double[]{bottonX, bottonY, topX
, topY}
                                  : new double[]{topX, topY, bottonX,
 bottonY};
        }
    }
}
```

21、最小公倍数和最大公因数

https://www.cnblogs.com/echoing/p/7878954.html

更多硬核内容就在公众号「yes 的练级攻略」

1. 最大公约数(Greatest Common Divisor(GCD))

1.1 基本概念

最大公因数，也称最大公约数、最大公因子，指两个或多个整数共有约数中最大的一个。a，b的最大公约数记为（a，b），同样的，a，b，c的最大公约数记为（a，b，c），多个整数的最大公约数也有同样的记号。求最大公约数有多种方法，常见的有质因数分解法、短除法、辗转相除法、更相减损法。与最大公约数相对应的概念是最小公倍数，a，b的最小公倍数记为[a，b]。

1.2 算法

辗转相除法

辗转相除法：辗转相除法是求两个自然数的最大公约数的一种方法，也叫欧几里德算法。

例如，求（319，377）：

∵ 319÷377=0（余319）

∴（319，377）=（377，319）；

∵ 377÷319=1（余58）

∴（377，319）=（319，58）；

∵ 319÷58=5（余29）

∴（319，58）=（58，29）；

∵ 58÷29=2（余0）

∴（58，29）= 29；

∴（319，377）=29。

可以写成右边的格式。

用辗转相除法求几个数的最大公约数，可以先求出其中任意两个数的最大公约数，再求这个最大公约数与第三个数的最大公约数，依次求下去，直到最后一个数为止。最后所得的那个最大公约数，就是所有这些数的最大公约数。

2. 最小公倍数(Least Common Multiple(LCM))

2.1 基本概念

两个或多个整数公有的倍数叫做它们的公倍数，其中除0以外最小的一个公倍数就叫做这几个整数的最小公倍数。整数a，b的最小公倍数记为[a，b]，同样的，a，b，c的最小公倍数记为[a，b，c]，多个整数的最小公倍数也有同样的记号。

与最小公倍数相对应的概念是最大公约数，a，b的最大公约数记为（a，b）。关于最小公倍数与最大公约数，我们有这样的定理：(a,b)[a,b]=ab(a,b均为整数)

2.2 算法

公式法

由于两个数的乘积等于这两个数的最大公约数与最小公倍数的积。即（a，b）×[a，b]=a×b。所以，求两个数的最小公倍数，就可以先求出它们的最大公约数，然后用上述公式求出它们的最小公倍数。

Java语言实现求最大公约数(GCD)和最小公倍数(LCM)

更多硬核内容就在公众号「yes 的练级攻略」

我猜你看到的是对辗转相除法最坏情况下复杂度的一个估计，见拉梅定理下面敲了一遍证明过程：

求 $gcd\,(a,b)$ ，设 $r_0=a$， $r_1=b$ ，不妨令 $a\geq b$ ，不断进行除数运算，得到一系列整数序列 q_i ， r_i ，和下面的方程：

$$r_0=q_1r_1+r_2,(r_1>r_2)$$

$$r_1=q_2r_2+r_3,(r_2>r_3)$$

.....

$$r_{n-3}=q_{n-2}r_{n-2}+r_{n-1},(r_{n-2}>r_{n-1})$$

$$r_{n-2}=q_{n-1}r_{n-1}+r_n,(r_{n-1}>r_n)$$

$$r_{n-1}=q_nr_n$$

每进行一次辗转相除，被除数就减少，到最后一步 r_n 就是我们所求的最大公约数。

由上面的方程组可以得到一个严格递减的数列：

$$r_1>r_2\ldots r_{n-1}>r_n$$

所有的 $q_i\geq 1$ 且有最后一项 $q_n\geq 2$ ，反证法若 $q_n\leq 1$ ，那么数列与严格递减矛盾。

下面考虑斐波那契数列：

0、1、1、2、3、$5\ldots F_{n-1}$、F_n，$F_0=0$

其中，$r_n\geq 1=F_2,r_{n-1}\geq 1=q_nr_n\geq 2=F_3$ 由归纳法，

对 $j\geq 0$，$r_{n-j}\geq F_{j+2}$

$$r_{n-j-1}=q_{n-j}r_{n-j}+r_{n-j+1}\geq r_{n-j}+r_{n-j+1}\geq F_{j+2}+F_{j+1}=F_{j+3}$$

所以证明了 $j\geq 0$，$r_{n-j}\geq F_{j+2}$ ，左右下标和为 $n+2$ ，进一步得到：

$b=r_q\geq F_{n+1}$ ，得到了一个含 n 的不等式，再根据斐波那契数列的黄金比的性质：

$$F_{n+1}>(\frac{1+\sqrt{5}}{2})^{n-1}$$

更多硬核内容就在公众号「yes 的练级攻略」

所以，$b \geq F_{n+1} > (\frac{1+\sqrt{5}}{2})^{n-1}$，两边取对数得到

$$log(b) > (n-1)log\left(\frac{1+\sqrt{5}}{2}\right) > \frac{n-1}{5}$$

$$n < 5log(b) + 1$$

...至此，得到了题主想要的5，实际上 $log(b)+1$ 就是数字的位数个数，又可以得到，

$$n < 5log(b) + 1 < 5(log(b) + 1) = 5d(b), d(b) \text{为整数} b \text{的位数}$$

代码 1：迭代式辗转相除法（时间复杂度 o(logn)，在《离散数学》上给出了一个证明（P77），可以证明，gcd 的运行次数不会超过 2*log2（n+1）n 为较小数。）

```
import java.util.Scanner;
public class GCDLCM {
    // 最大公约数迭代方法 1
    public static int get_gcd(int n1, int n2) {
        int gcd = 0;
        if (n1 < n2) {// 交换 n1、n2 的值
            n1 = n1 + n2;
            n2 = n1 - n2;
            n1 = n1 - n2;
        }
        if (n1 % n2 == 0) {
            gcd = n2;
        }
        while (n1 % n2 > 0) {
            n1 = n1 % n2;
            if (n1 < n2) {
                n1 = n1 + n2;
                n2 = n1 - n2;
                n1 = n1 - n2;
            }
            if (n1 % n2 == 0) {
                gcd = n2;
            }
        }
        return gcd;
    }
    //最大公约数迭代方法 2
    public int gcd(int x,int y){
        if(x < y){
            x = x ^ y;
            y = x ^ y;
```

禁止商用 公众号：yes的练级攻略

```
            x = x ^ y;
        }
        int tmp = 1;
        while(tmp != 0){
            tmp = x % y;
            x = y;
            y = tmp;
        }
        return x;
    }
    // 最小公倍数
    public static int get_lcm(int n1, int n2) {
        return n1 * n2 / get_gcd(n1, n2);
    }
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("请输入第一个整数：");
        int n1 = input.nextInt();
        System.out.print("请输入第二个整数：");
        int n2 = input.nextInt();
        System.out.println("(" + n1 + "," + n2 + ")" + "=" + get_gcd(n1, n2));
        System.out.println("[" + n1 + "," + n2 + "]" + "=" + get_lcm(n1, n2));
    }
}
```

代码 2：递归式辗转相除法

```
import java.util.Scanner;
public class GCDLCM {
    // 最大公约数递归方式 1
    public static int get_gcd(int a, int b) {
        int max, min;
        max = (a > b) ? a : b;
        min = (a < b) ? a : b;
        if (max % min != 0) {
            return get_gcd(min, max % min);
        } else
            return min;
    }
    // 最大公约数递归方式 2
    public int gcd(int x, int y) {
        return x == 0 ? y : gcd(y % x, x);
    }
    // 最小公倍数
    public static int get_lcm(int a, int b) {
        return a * b / get_gcd(a, b);
```

禁止商用 公众号：yes的练级攻略

```
    }
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int n1 = input.nextInt();
        int n2 = input.nextInt();
        System.out.println("(" + n1 + "," + n2 + ")" + "=" + get_gcd(n1, n2));
        System.out.println("[" + n1 + "," + n2 + "]" + "=" + get_lcm(n1, n2));
    }
}
```

22、丑数 3（1201）

1201. 丑数 III

难度 中等 49 收藏 分享 切换为英文 关注 反馈

请你帮忙设计一个程序，用来找出第 `n` 个丑数。

丑数是可以被 `a` 或 `b` 或 `c` 整除的 **正整数**。

示例 1：

输入： `n = 3, a = 2, b = 3, c = 5`
输出： `4`
解释： 丑数序列为 2, 3, 4, 5, 6, 8, 9, 10... 其中第 3 个是 4。

示例 2：

输入： `n = 4, a = 2, b = 3, c = 4`
输出： `6`
解释： 丑数序列为 2, 3, 4, 6, 8, 9, 10, 12... 其中第 4 个是 6。

示例 3：

输入： `n = 5, a = 2, b = 11, c = 13`
输出： `10`
解释： 丑数序列为 2, 4, 6, 8, 10, 11, 12, 13... 其中第 5 个是 10。

示例 4：

输入： `n = 1000000000, a = 2, b = 217983653, c = 336916467`
输出： `1999999984`

提示：

- `1 <= n, a, b, c <= 10^9`
- `1 <= a * b * c <= 10^18`
- 本题结果在 `[1, 2 * 10^9]` 的范围内

更多硬核内容就在公众号「yes 的练级攻略」

https://leetcode-cn.com/problems/ugly-number-iii/solution/ji-he-yun-suan-er-fen-cha-zhao-chao-guo-100-by-msh/

由题目描述"丑数是可以被 a 或 b 或 c 整除的 正整数。"可以想到使用集合并集操作来解决。
丑数=【被a整除的正整数】∪【被b整除的正整数】∪【被b整除的正整数】

集合公式为：

$$|A_1 \cup A_2 \cup \cdots \cup A_m| =$$

$$\sum_{1\le i\le m} |A_i| - \sum_{1\le i<j\le m} |A_i \cap A_j| + \sum_{1\le i<j<k\le m} |A_i \cap A_j \cap A_k| - \cdots + (-1)^{m-1} |A_1 \cap A_2 \cap \cdots \cap A_m|$$

特别的，当集合数量为三个时，公式为：
|A∪B∪C| = |A|+|B|+|C| - |A∩B| - |B∩C| - |C∩A| + |A∩B∩C|

假设x是满足条件的数，
【被a整除的正整数】可以用：x/a表示;
同理
【被b整除的正整数】可以用：x/b表示;
【被c整除的正整数】可以用：x/c表示;

【被a整除的正整数】∩【被b整除的正整数】，可以用：x/(a,b的最小公倍数)表示;
同理
【被a整除的正整数】∩【被c整除的正整数】，可以用：x/(a,c的最小公倍数)表示;
【被b整除的正整数】∩【被c整除的正整数】，可以用：x/(b,c的最小公倍数)表示;
【被a整除的正整数】∩【被b整除的正整数】∩【被c整除的正整数】，可以用：x/(a,b,c的最小公倍数)表示;

至此，集合容斥公式的就能用x表示出来了。

因为，x的大小可定在[n,Integer.maxValue]之间，因此可以使用二分查找法寻找可能的x。

```
class Solution {
    public int nthUglyNumber(int n, int a, int b, int c) {
        int lcm_ab = (int) Math.min((long) a * b / gcd(a, b), Integer.MAX_VALUE
);
        int lcm_ac = (int) Math.min((long) a * c / gcd(a, c), Integer.MAX_VALUE
);
        int lcm_bc = (int) Math.min((long) b * c / gcd(b, c), Integer.MAX_VALUE
);
        int lcm_abc = (int) Math.min((long) lcm_ab * c / gcd(lcm_ab, c), Intege
r.MAX_VALUE);
        int left = n, right = Integer.MAX_VALUE, mid = 0;
        while(left <= right){
            mid = left + (right - left) / 2;
            int val = mid / a + mid / b + mid / c - mid /lcm_ab - mid / lcm_ac
- mid / lcm_bc + mid / lcm_abc;
            if(val == n) break;
            else if(val>n) right = mid - 1;
            else left = mid + 1;
        }
        if( mid % a == 0 || mid % b == 0 || mid % c == 0){
```

```
            return mid;
        }else{
            return mid - Math.min(Math.min(mid % a, mid % b), mid % c);
        }
    }
    //最大公约数
    public int gcd(int x,int y){
        if(x < y){
            x = x ^ y;
            y = x ^ y;
            x = x ^ y;
        }
        int tmp = 1;
        while(tmp != 0){
            tmp = x % y;
            x = y;
            y = tmp;
        }
        return x;
    }
}
```

23、水壶问题（365）

365. 水壶问题

难度 中等 245 收藏 分享 切换为英文 关注 反馈

有两个容量分别为 *x*升 和 *y*升 的水壶以及无限多的水。请判断能否通过使用这两个水壶，从而可以得到恰好 *z*升 的水？

如果可以，最后请用以上水壶中的一或两个来盛放取得的 *z*升 水。

你允许：

- 装满任意一个水壶
- 清空任意一个水壶
- 从一个水壶向另外一个水壶倒水，直到装满或者倒空

示例 1: (From the famous *"Die Hard"* example)

```
输入: x = 3, y = 5, z = 4
输出: True
```

示例 2:

```
输入: x = 2, y = 6, z = 5
输出: False
```

https://leetcode-cn.com/problems/water-and-jug-problem/solution/javade-bfsxie-fa-by-sweetiee/

更多硬核内容就在公众号「yes 的练级攻略」

—

这道题目一眼看过去可以用BFS。Java没有原生的Pair，所以我用Map.Entry替代了。leetcode上需要额外加上import语句。
注意到这道题操作水壶的时候，**两个水壶不可能同时都是半满的。**如果某个水壶是半满的，另外一个肯定是满的或者空的。而且如果某个水壶是半满的（此时另外一个就是空的或者满的），就不能直接把这个水壶填满，也不能把这个半满的水倒掉，因为这会回到初始状态，这么做没有意义。

小Tips：如果某天你写了一个BFS版本的搜索，如何最快的速度再写一份DFS版本发的呢？**只需要把queue改为stack就可以了~**

还有，visited这个set的更新，**一定要在入queue的时候**，而不是在queue取出元素的时候，否则队列里会塞很多很多已经遍历过的状态~（某个状态可能是由多个之前的状态转移过来的，会导致该状态的被多次加到queue里去），这点非常重要！！在某些题目的情况下BFS没写好，性能差距是很大的！！

我去看了官方题解，发现还有数学的方法🤦‍♀️，蒟蒻本弱的我学习到了新知识~官方题解说的很好，大家可以去看下~

方法 1：BFS

```
import java.util.*;
class Solution {
    public boolean canMeasureWater(int x, int y, int z) {
        if (z == 0) return true;
        if (x + y < z) return false;
        Queue<Map.Entry<Integer, Integer>> que = new ArrayDeque<>();
        AbstractMap.SimpleEntry<Integer, Integer> start = new AbstractMap.Simpl
eEntry<>(0, 0);
        que.add(start);
        Set<Map.Entry<Integer, Integer>> vis = new HashSet<>();
        vis.add(start);
        while (!que.isEmpty()) {
            Map.Entry<Integer, Integer> entry = que.poll();
            int nx = entry.getKey();
            int ny = entry.getValue();
            if (nx == z || ny == z || nx + ny == z) return true;
            if (nx == 0) addIntQueue(que, vis, new  AbstractMap.SimpleEntry<>(x
, ny)); // 把第一个桶填满
            if (ny == 0) addIntQueue(que, vis, new  AbstractMap.SimpleEntry<>(n
x, y)); // 把第二个桶填满
            if (ny < y) addIntQueue(que, vis, new  AbstractMap.SimpleEntry<>(0,
 ny)); // 把第一个桶倒空
            if (nx < x) addIntQueue(que, vis, new  AbstractMap.SimpleEntry<>(nx
, 0)); // 把第二个桶倒空
            // y - ny 是第二个桶还可以再加的水的升数，但是最多只能加 nx 升水
            int move = Math.min(nx, y - ny);
            // 把第一个桶里的 nx 升水倒到第二个桶里去。
            addIntQueue(que, vis, new  AbstractMap.SimpleEntry<>(nx - move, ny
+ move));
            // 反过来同理，x - nx 是第一个桶还可以再加的升数，但是最多只能加 ny 升水
            move = Math.min(x - nx, ny);
            // 把第二个桶里的 ny 升水倒到第一个桶里去。
```

```
            addIntQueue(que, vis, new  AbstractMap.SimpleEntry<>(nx + move, ny
- move));
        }
        return false;
    }
    private void addIntQueue(Queue<Map.Entry<Integer, Integer>> que, Set<Map.En
try<Integer, Integer>> vis, AbstractMap.SimpleEntry<Integer, Integer> newEntry)
 {
        if (!vis.contains(newEntry)) {
            vis.add(newEntry);
            que.add(newEntry);
        }
    }
}
```

https://leetcode-cn.com/problems/water-and-jug-problem/solution/shui-hu-wen-ti-by-leetcode-solution/

思路及算法

预备知识：贝祖定理

我们认为，每次操作只会让桶里的水总量增加 `x`，增加 `y`，减少 `x`，或者减少 `y`。

你可能认为这有问题：如果往一个不满的桶里放水，或者把它排空呢？那变化量不就不是 `x` 或者 `y` 了吗？接下来我们来解释这一点：

- 首先要清楚，在题目所给的操作下，两个桶不可能同时有水且不满。因为观察所有题目中的操作，操作的结果都至少有一个桶是空的或者满的；
- 其次，对一个不满的桶加水是没有意义的。因为如果另一个桶是空的，那么这个操作的结果等价于直接从初始状态给这个桶加满水；而如果另一个桶是满的，那么这个操作的结果等价于从初始状态分别给两个桶加满；
- 再次，把一个不满的桶里面的水倒掉是没有意义的。因为如果另一个桶是空的，那么这个操作的结果等价于回到初始状态；而如果另一个桶是满的，那么这个操作的结果等价于从初始状态直接给另一个桶倒满。

因此，我们可以认为每次操作只会给水的总量带来 `x` 或者 `y` 的变化量。因此我们的目标可以改写成：找到一对整数 a, b，使得

$$ax + by = z$$

而只要满足 $z \leq x + y$，且这样的 a, b 存在，那么我们的目标就是可以达成的。这是因为：

- 若 $a \geq 0, b \geq 0$，那么显然可以达成目标。
- 若 $a < 0$，那么可以进行以下操作：

  1. 往 `y` 壶倒水；
  2. 把 `y` 壶的水倒入 `x` 壶；
  3. 如果 `y` 壶不为空，那么 `x` 壶肯定是满的，把 `x` 壶倒空，然后再把 `y` 壶的水倒入 `x` 壶。

  重复以上操作直至某一步时 `x` 壶进行了 a 次倒空操作，`y` 壶进行了 b 次倒水操作。
- 若 $b < 0$，方法同上，`x` 与 `y` 互换。

而贝祖定理告诉我们，$ax + by = z$ 有解当且仅当 z 是 x, y 的最大公约数的倍数。因此我们只需要找到 x, y 的最大公约数并判断 z 是否是它的倍数即可。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(\log(\min(x,y)))$，取决于计算最大公约数所使用的辗转相除法。
- 空间复杂度：$O(1)$，只需要常数个变量。

裴蜀定理（或贝祖定理）得名于法国数学家艾蒂安·裴蜀，说明了对任何整数a、b和它们的最大公约数d，关于未知数x和y的线性不定方程（称为裴蜀等式）：若a,b是整数,且gcd(a,b)=d，那么对于任意的整数x,y,ax+by都一定是d的倍数，特别地，一定存在整数x,y，使ax+by=d成立。

它的一个重要推论是：a,b互质的充要条件是存在整数x,y使ax+by=1.

$$\left\{\left(\frac{m}{d}\left(x_0+\frac{kb}{d}\right),\ \frac{m}{d}\left(y_0-\frac{ka}{d}\right)\right)\mid k\in\mathbb{Z}\right\}$$

证法一：

设$d=gcd(a,b)$，则$d|a,d|b$。由整除的性质，$\forall x,y\in Z$，有d|(ax+by)。

设s为ax+by最小正值，令$q=\lfloor\frac{a}{s}\rfloor$，则,$r=amods=a-q(ax+by)=a(1-qx)+b(-qy)$。

可见r 也为a,b 的线性组合。

由于r为$amods$ 所得,所以$0\leq r<s$。

由于s为a,b线性组合的最小正值,可知$r=0$。
因此有$s|a$,同理$s|b$ ，因此，s是a与b的公约数，所以d≥ s.....①。

因为d|a，d|b，且s是a与b的一个线性组合，所以由整除性质知d|s。

但由于d|s和s＞ 0，因此d≤ s.....②。

由①②得d=s，命题得证

证法二：

⑴若b=0，则（a,b)=a.这时定理显然成立。

⑵若a,b不等于0.

记d = (a, b), 对ax + by = d，两边同时除以d，可得(a1)x + (b1)y = 1，其中(a1,b1) = 1。
转证(a1)x + (b1)y = 1。由带余除法：
① (a1) = (q1)(b1) + (r1), 其中0 < r1 < b1
② (b1) = (q2)(r1) + (r2), 其中0 < r2 < r1
③ (r1) = (q3)(r2) + (r3), 其中0 < r3 < r2

④ (rn-4) = (qn-2)(rn-3) + (rn-2)
⑤ (rn-3) = (qn-1)(rn-2) + (rn-1)
⑥ (rn-2) = (qn)(rn-1) + (rn)
⑦ (rn-1) = (qn+1)(rn) + 1
故，由⑦和⑥推出(rn-2)An-2 + (rn-1)Bn-1 = 1
再结合⑤推出(rn-3)An-3 + (rn-2)Bn-2 = 1
再结合④推出(rn-4)An-4 + (rn-3)Bn-3 = 1
.....
再结合③推出(r1)A1 + (r2)B2 = 1
再结合②推出(b1)A0 + (r1)B0 = 1
再结合①推出(a1)x + (b1)y = 1
得证。

不定方程 例题： 求不定方程 $11x + 15y = 7$ 的整数解.

解： $x_0 = 2, y_0 = -1$ 是方程的一组整数解.

$\therefore x = 2 + 15t, y = -1 = 11\ (t \in Z)$

定理

编辑

在数论中，裴蜀定理是一个关于最大公约数（或最大公约式）的定理。裴蜀定理得名于法国数学家艾蒂安·裴蜀，说明了对任何整数a、b和它们的最大公约数d，关于未知数x和y的线性丢番图方程（称为裴蜀等式）：

ax + by = m

有解当且仅当m是d的倍数。裴蜀等式有解时必然有无穷多个整数解，每组解x、y都称为裴蜀数，可用辗转相除法求得。

例如，12和42的最大公因子是6，则方程12x + 42y = 6有解。事实上有（-3）×12 + 1×42 = 6及4×12 + (-1）×42 = 6。

特别来说，方程 ax + by = 1 有解当且仅当整数a和b互素。

裴蜀等式也可以用来给最大公约数定义：d其实就是最小的可以写成ax + by形式的正整数。这个定义的本质是整环中"理想"的概念。因此对于多项式整环也有相应的裴蜀定理。

方法 2：数学【贝祖定理】

```java
import java.util.*;
class Solution {
    public boolean canMeasureWater(int x, int y, int z) {
        if (x + y < z) return false;
        if (x == 0 || y == 0) return z == 0 || x + y == z;
        return z % gcd(x, y) == 0;
    }
    public int gcd (int x, int y) {
        if (x < y) {
            x = x ^ y;
            y = x ^ y;
            x = x ^ y;
        }
        int tmp = 1;
        while (tmp != 0) {
            tmp = x % y;
            x = y;
            y = tmp;
        }
        return x;
    }
```

}

24、卡牌分组（914）

914. 卡牌分组

难度 简单 181 收藏 分享 切换为英文 关注 反馈

给定一副牌，每张牌上都写着一个整数。

此时，你需要选定一个数字 `X`，使我们可以将整副牌按下述规则分成 1 组或更多组：

- 每组都有 `X` 张牌。
- 组内所有的牌上都写着相同的整数。

仅当你可选的 `X >= 2` 时返回 `true`。

示例 1：

```
输入：[1,2,3,4,4,3,2,1]
输出：true
解释：可行的分组是 [1,1]，[2,2]，[3,3]，[4,4]
```

示例 2：

```
输入：[1,1,1,2,2,2,3,3]
输出：false
解释：没有满足要求的分组。
```

示例 3：

```
输入：[1]
输出：false
解释：没有满足要求的分组。
```

示例 4：

```
输入：[1,1]
输出：true
解释：可行的分组是 [1,1]
```

示例 5：

```
输入：[1,1,2,2,2,2]
输出：true
解释：可行的分组是 [1,1]，[2,2]，[2,2]
```

提示：

1. `1 <= deck.length <= 10000`
2. `0 <= deck[i] < 10000`

https://leetcode-cn.com/problems/x-of-a-kind-in-a-deck-of-cards/solution/qia-pai-fen-zu-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：最大公约数

思路和算法

由于方法一已经提及 `X` 一定为 $count_i$ 的约数，这个条件是对所有牌中存在的数字 `i` 成立的，所以我们可以推出，只有当 `X` 为所有 $count_i$ 的约数，即所有 $count_i$ 的最大公约数的约数时，才存在可能的分组。公式化来说，我们假设牌中存在的数字集合为 `a, b, c, d, e`，那么只有当 `X` 为

$$gcd(count_a, count_b, count_c, count_d, count_e)$$

的约数时才能满足要求。

因此我们只要求出所有 $count_i$ 最大公约数 g，判断 g 是否大于等于 2 即可，如果大于等于 2，则满足条件，否则不满足。

复杂度分析

- 时间复杂度：$O(N \log C)$，其中 N 是卡牌的个数，C 是数组 `deck` 中数的范围，在本题中 C 的值为 10000。求两个数最大公约数的复杂度是 $O(\log C)$，需要求最多 $N-1$ 次。
- 空间复杂度：$O(N+C)$ 或 $O(N)$。

```java
public class Solution {
    public boolean hasGroupsSizeX(int[] deck) {
        int len = deck.length;
        if (len < 2) {
            return false;
        }
        // 计数数组，10000 是根据题目给出的数值范围定的
        int[] cnt = new int[10000];
        for (int num : deck) {
            cnt[num]++;
        }
        // 先得到第 1 个数的个数，以避免在循环中赋值
        int x = cnt[deck[0]];
        for (int i = 0; i < 10000; i++) {
            if (cnt[i] == 1) return false;
            if (cnt[i] > 1) {
                x = gcd(x, cnt[i]);
                // 这里做判断可以提前终止运行，也可以等到最后再做，各有优劣，任选其一
                if (x == 1) return false;
            }
        }
        return true;
    }
    private int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
```

```
        // int[] deck = new int[]{1, 2, 3, 4, 4, 3, 2, 1};
        // int[] deck = new int[]{1, 1, 1, 2, 2, 2, 3, 3};
        // int[] deck = new int[]{1};
        // int[] deck = new int[]{1, 1};
        int[] deck = new int[]{0, 0, 1, 1, 1, 1, 2, 2, 3, 4};
        // boolean res = solution.hasGroupsSizeX(deck);
        // System.out.println(res);
        System.out.println(solution.gcd(2, 8));
    }
}
```

25、分数加减运算（592）

592. 分数加减运算

难度 中等 30 收藏 分享 切换为英文 关注 反馈

给定一个表示分数加减运算表达式的字符串，你需要返回一个字符串形式的计算结果。 这个结果应该是不可约分的分数，即最简分数。 如果最终结果是一个整数，例如 2，你需要将它转换成分数形式，其分母为 1。所以在上述例子中, 2 应该被转换为 2/1。

示例 1:

```
输入:"-1/2+1/2"
输出: "0/1"
```

示例 2:

```
输入:"-1/2+1/2+1/3"
输出: "1/3"
```

示例 3:

```
输入:"1/3-1/2"
输出: "-1/6"
```

示例 4:

```
输入:"5/3+1/3"
输出: "2/1"
```

说明:

1. 输入和输出字符串只包含 '0' 到 '9' 的数字，以及 '/', '+' 和 '-'。
2. 输入和输出分数格式均为 ±分子/分母。如果输入的第一个分数或者输出的分数是正数，则 '+' 会被省略掉。
3. 输入只包含合法的**最简分数**，每个分数的**分子**与**分母**的范围是 [1,10]。 如果分母是1，意味着这个分数实际上是一个整数。
4. 输入的分数个数范围是 [1,10]。
5. **最终结果**的分子与分母保证是 32 位整数范围内的有效整数。

https://leetcode-cn.com/problems/fraction-addition-and-subtraction/solution/fen-shu-jia-jian-yun-suan-by-a-long-tai-zi/

```
ublic class Solution {
    public String fractionAddition(String expression) {
        List<Integer> list1=new ArrayList<>();  //List1 用来存储分子
        List<Integer> list2=new ArrayList<>();  //List2 用来存储分母
        int tmp=0;
        for(int i=1;i<expression.length();i++){
            if(expression.charAt(i)=='+'){
                String sub=expression.substring(tmp,i);
                list1.add(Integer.parseInt(sub.substring(0,sub.indexOf("/"))));
                list2.add(Integer.parseInt(sub.substring(sub.indexOf("/")+1)));
                tmp=i+1;
            }else if(expression.charAt(i)=='-'){
                String sub=expression.substring(tmp,i);
                list1.add(Integer.parseInt(sub.substring(0,sub.indexOf("/"))));
                list2.add(Integer.parseInt(sub.substring(sub.indexOf("/")+1)));
                tmp=i;  //这样保证下一次变换成整数时把负号加进去，因为减法可以看做加
上一个负数
            }
        }
        //别忘了，最后还有一个数后面没有相应符号
        String sub=expression.substring(tmp);
        list1.add(Integer.parseInt(sub.substring(0,sub.indexOf("/"))));
        list2.add(Integer.parseInt(sub.substring(sub.indexOf("/")+1)));
        int times=list1.size()-1;  //其实需要加法的次数是总的长度减一
        for(int i=0;i<times;i++){
            int [] tmp1=plus(list1.get(0),list2.get(0),list1.get(1),list2.get(1
));
            list1.remove(0);list1.remove(0);list2.remove(0);list2.remove(0);
            list1.add(0,tmp1[0]);
            list2.add(0,tmp1[1]);
        }
        return String.valueOf(list1.get(0))+"/"+String.valueOf(list2.get(0));
    }
    public int [] plus(int a,int b,int c,int d){
        //计算（a/b)+(c/d)
        int tmp1=a*d+b*c;
        int tmp2=b*d;
        int x=gcd(tmp1,tmp2);
        if(tmp1/x>0&&tmp2/x<0){
            tmp1=(-1)*tmp1;
            tmp2=(-1)*tmp2;
        }
        return new int[]{tmp1/x,tmp2/x};
    }
```

```
    public int gcd(int x, int y) {
        return x == 0 ? y : gcd (y % x, x);
    }
}
```

26、直线上最多的点数（149）

149. 直线上最多的点数

难度 困难 178 收藏 分享 切换为英文 关注

给定一个二维平面，平面上有 *n* 个点，求最多有多少个点在同一条直线上。

示例 1:

```
输入: [[1,1],[2,2],[3,3]]
输出: 3
解释:
^
|
|        o
|     o
|  o
+-------------->
0  1  2  3  4
```

示例 2:

```
输入: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出: 4
解释:
^
|
|  o
|     o        o
|        o
|  o        o
+------------------->
0  1  2  3  4  5  6
```

https://leetcode-cn.com/problems/max-points-on-a-line/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by--35/

更多硬核内容就在公众号「yes 的练级攻略」

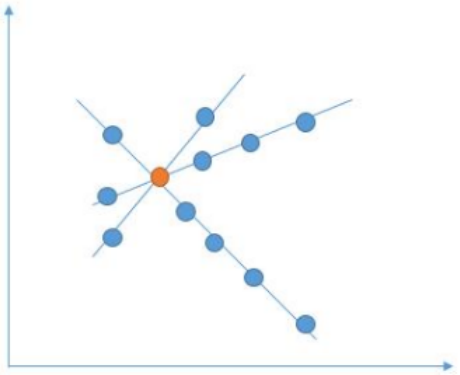
解法三

解法二中，我们相当于是对直线的分类，一条直线一条直线的考虑，去求直线上的点。

这里 看到另一种想法，分享一下。

灵感应该来自于直线方程的另一种表示方式，「点斜式」，换句话，一个点加一个斜率即可唯一的确定一条直线。

所以我们可以对「点」进行分类然后去求，问题转换成，经过某个点的直线，哪条直线上的点最多。



当确定一个点后，平面上的其他点都和这个点可以求出一个斜率，斜率相同的点就意味着在同一条直线上。

所以我们可以用 `HashMap` 去计数，斜率作为 `key`，然后遍历平面上的其他点，相同的 `key` 意味着在同一条直线上。

上边的思想解决了「经过某个点的直线，哪条直线上的点最多」的问题。接下来只需要换一个点，然后用同样的方法考虑完所有的点即可。

当然还有一个问题就是斜率是小数，怎么办。

之前提到过了，我们用分数去表示，求分子分母的最大公约数，然后约分，最后将 「分子 + "@" + "分母"」作为 `key` 即可。

最后还有一个细节就是，当确定某个点的时候，平面内如果有和这个重叠的点，如果按照正常的算法约分的话，会出现除 `0` 的情况，所以我们需要单独用一个变量记录重复点的个数，而重复点一定是过当前点的直线的。

```
class Solution {
    public int maxPoints(int[][] points) {
        if (points.length < 3) {
            return points.length;
        }
        int res = 0;
        //遍历每个点
        for (int i = 0; i < points.length; i++) {
            int duplicate = 0;
            int max = 0;//保存经过当前点的直线中，最多的点
            HashMap<String, Integer> map = new HashMap<>();
            for (int j = i + 1; j < points.length; j++) {
                //求出分子分母
                int x = points[j][0] - points[i][0];
                int y = points[j][1] - points[i][1];
                if (x == 0 && y == 0) {
```

```
                duplicate++;
                continue;

            }
            //进行约分
            int gcd = gcd(x, y);
            x = x / gcd;
            y = y / gcd;
            String key = x + "@" + y;
            map.put(key, map.getOrDefault(key, 0) + 1);
            max = Math.max(max, map.get(key));
        }
        //1 代表当前考虑的点，duplicate 代表和当前的点重复的点
        res = Math.max(res, max + duplicate + 1);
    }
    return res;
}

private int gcd(int a, int b) {
    return a == 0 ? b : gcd(b % a, a);
}
}
```

27、最佳直线（面试题 16.14）

面试题 16.14. 最佳直线

难度 中等 7 收藏 分享 切换为英文 关注 反馈

给定一个二维平面及平面上的 N 个点列表 Points，其中第 i 个点的坐标为 Points[i]=[X_i, Y_i]。请找出一条直线，其通过的点的数目最多。

设穿过最多点的直线所穿过的全部点编号从小到大排序的列表为 S，你仅需返回 [S[0], S[1]] 作为答案，若有多条直线穿过了相同数量的点，则选择 S[0] 值较小的直线返回，S[0] 相同则选择 S[1] 值较小的直线返回。

示例：

```
输入：  [[0,0],[1,1],[1,0],[2,0]]
输出：  [0,2]
解释：  所求直线穿过的3个点的编号为[0,2,3]
```

提示：

- 2 <= len(Points) <= 300
- len(Points[i]) = 2

https://leetcode-cn.com/problems/best-line-lcci/solution/shu-xue-ti-zhi-xian-de-yi-ban-shi-fang-cheng-by-tu/

直线的一般式方程:Ax + By + C = 0。

直线一般式方程适用于所有的二维空间直线。而其他直线方程，如点斜式、两点式，要么不能支持所有情况下的直线（比如跟坐标轴垂直或者平行），要么不能支持所有情况下的点（比如x坐标相等，或者y坐标相等）。所以一般式方程在用计算机处理二维图形数据时特别有用。

注意:
1）这里，要唯一定位一条直线。只需要知道三个数A、B、C即可。
不过，由于4x + 6y + 2 = 0、2x + 3y + 1 = 0其实是表示同一条直线。所以，要对A、B、C求最大公约数。
2）对于任意两点(X1,Y1)、(X2,Y2)，都有 A = Y2 - Y1 B = X1 - X2 C = X2*Y1* - *X1*Y2

方法 1：数学（直线的一般方程）

```
class Solution {
    public int[] bestLine(int[][] points) {
        int max = 0;
        int maxHash = 0;
        int[] res = new int[2];
        HashMap<Integer,Integer> map = new HashMap<>();
        HashMap<Integer,int[]> record = new HashMap<>();
        for(int i = 0;i < points.length - 1;i++){
            for(int j = i + 1;j < points.length;j++){
                long A = points[j][1] - points[i][1];
                long B = points[i][0] - points[j][0];
                long C = ((long)points[j][0]) * points[i][1] - ((long)p
oints[i][0]) * points[j][1];
                long gcd = gcd(gcd(A,B),C);
                A /= gcd;
                B /= gcd;
                C /= gcd;
                int hash = hash((int)A,(int)B,(int)C);
                int count = map.getOrDefault(hash,0) + 1;
                map.put(hash,count);
                if(count == 1) record.put(hash,new int[]{i,j});
                if(count > max){
                    max = count;
                    maxHash = hash;
                    res = record.get(hash);
                }else if(count == max){
                    int[] t1 = record.get(maxHash);
                    int[] t2 = record.get(hash);
                    if(t1[0] > t2[0] || t1[0] == t2[0] && t1[1] > t2[1]
){
                        maxHash = hash;
                        res = t2;
                    }
                }
            }
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        }
        return res;
    }

    private long gcd(long a,long b){
        return b == 0? a:gcd(b,a % b);
    }

    //随便写的 hash 方法。反正数据量也不大。冲突可能性小
    private int hash(int a,int b,int c){
        a = (a ^ (a >>> 16) & 0x0000ffff) << 20;
        b = (b ^ (b >>> 16) & 0x0000ffff) << 10;
        c = c ^ (c >>> 16) & 0x00000ffff;
        return a | b | c;
    }
}
```

https://leetcode-cn.com/problems/best-line-lcci/solution/linkedhashmap-by-silentteller/

方法 2：LinkedHashMap

```
//45 ms
class Solution {
    public int[] bestLine(int[][] points) {
        map = new LinkedHashMap<>();
        List<Integer> res = new ArrayList<>();
        for(int i = 0; i < points.length-1; ++i){
            for(int j = i+1; j < points.length; ++j){
                int diffx = points[j][0] - points[i][0];
                int diffy = points[j][1] - points[i][1];
                double k = compute(diffx, diffy);
                List<Integer> list;
                if(map.get(k) == null){
                    list = new ArrayList<>();
                    list.add(i);
                    list.add(j);
                    map.put(k, list);
                }else{
                    list = map.get(k);
                    list.add(j);
                    map.put(k, list);
                }
                if(list.size() > res.size()){
                    res = list;
                }
            }
```

```
            map = new LinkedHashMap<>();
        }
        return new int[]{res.get(0), res.get(1)};
    }
    private HashMap<Double, List<Integer>> map;
    private double compute(int diffx, int diffy){
        if(diffx == 0){
            return -12345.0;
        }else if(diffy == 0){
            return 0.0;
        }else{
            return (double)diffy / (double)diffx;
        }
    }
}
```

28、字符串的最大公因子（1071）

1071. 字符串的最大公因子

难度 简单 164 收藏 分享 切换为英文 关注 反馈

对于字符串 `S` 和 `T`，只有在 `S = T + ... + T`（`T` 与自身连接 1 次或多次）时，我们才认定“`T` 能除尽 `S`”。

返回最长字符串 `X`，要求满足 `X` 能除尽 `str1` 且 `X` 能除尽 `str2`。

示例 1：

```
输入：str1 = "ABCABC", str2 = "ABC"
输出："ABC"
```

示例 2：

```
输入：str1 = "ABABAB", str2 = "ABAB"
输出："AB"
```

示例 3：

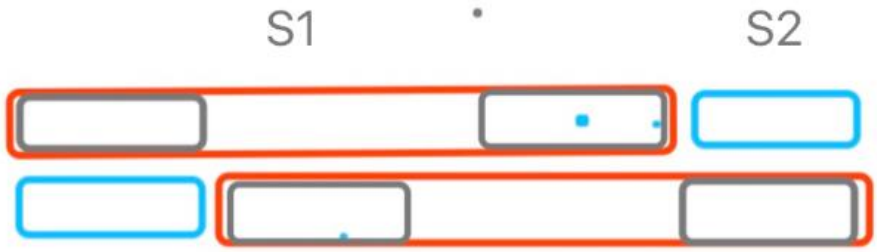
```
输入：str1 = "LEET", str2 = "CODE"
输出：""
```

提示：

1. `1 <= str1.length <= 1000`
2. `1 <= str2.length <= 1000`
3. `str1[i]` 和 `str2[i]` 为大写英文字母

https://leetcode-cn.com/problems/greatest-common-divisor-of-strings/solution/java-hen-jian-ji-yi-yan-jiu-neng-kan-ming-bai-by-s/

看见评论里有人在讨论，简单证明了一下：
1、若s1和s2存在相同的除数X，即s1=XXX, s2=XX, 可见s1+s2=s2+s1
2、现在的问题是：若s1和s2不存在相同的除数，那么是否存在s1+s2=s2+s1的情况呢?
我们假设s1和s2不存在相同的除数，且满足s1+s2=s2+s1，如下图所示（红色的是s1，蓝色的是s2，其中s1长于s2）



那由上图可知s2既是s1的前缀，又是s2的后缀（灰色表示）。

再看上图的黄色部分可知s3+s2=s2+s3；其中s3=s1-s2。
以上，我们由s1+s2=s2+s1推出了s3+s2=s2+s3， 其中s3 = s1 - s2。
继而，s3 + s4 = s4 + s3,其中s4 = s3 - s2。
然后可以递归下去，每次用长的减去短的。
因为我们的假设条件是等式s1+s2=s2+s1成立，即上图示成立，所以每次递归时的两个字符串，短串总是长串的前缀和后缀，所以两串总是可减的，
所以我们总是能减到两个串相等的时候，相等时即是最大公共除数，与假设的没有公共除数矛盾。
因此结论是若s1和s2不存在公共除数，则s1+s2!=s2+s1(反之亦然)。

```
class Solution {
    public String gcdOfStrings(String str1, String str2) {
        // 假设 str1 是 N 个 x，str2 是 M 个 x，那么 str1+str2 肯定是等于 str2+str1 的。
        if (!(str1 + str2).equals(str2 + str1)) return "";
        // 辗转相除法求 gcd。
        return str1.substring(0, gcd(str1.length(), str2.length()));
    }
    private int gcd(int a, int b) {
        return a == 0 ? b : gcd(b % a, a);
    }
}
```

29、第 N 个神奇数字（878）

878. 第 N 个神奇数字

难度 困难 54 收藏 分享 切换为英文 关注 反馈

如果正整数可以被 A 或 B 整除，那么它是神奇的。

返回第 N 个神奇数字。由于答案可能非常大，**返回它模** `10^9 + 7` **的结果。**

示例 1：

```
输入：N = 1, A = 2, B = 3
输出：2
```

示例 2：

```
输入：N = 4, A = 2, B = 3
输出：6
```

示例 3：

```
输入：N = 5, A = 2, B = 4
输出：10
```

示例 4：

```
输入：N = 3, A = 6, B = 4
输出：8
```

提示：

1. `1 <= N <= 10^9`
2. `2 <= A <= 40000`
3. `2 <= B <= 40000`

YESHI练级攻略

https://leetcode-cn.com/problems/nth-magical-number/solution/di-n-ge-shen-qi-shu-zi-by-leetcode/

方法一：数学方法

思路和算法

用数学方法找出第 N 个神奇数字。

神奇数字是有规律。设 L 为 A，B 的最小公倍数，如果 $X \leq L$ 是神奇数字，那么 $X+L$ 也是神奇数字，因为 $A \| X$，$A \| L$ 可以推出 $A \| (X+L)$，对于 B 也是如此。

有 $M = \frac{L}{A} + \frac{L}{B} - 1$ 个神奇数字小于等于 L：其中 $\frac{L}{A}$ 个是能被 A 整除的，$\frac{L}{B}$ 个能被 B 整除，1 个能同时被 A，B 整除。

设 $N = M * q + r$ $(r < M)$，前 $L * q$ 个数字有 $M * q$ 个神奇数字，$(L*q+1, L*q+2, \cdots)$ 之间有 r 个神奇数字。可以暴力搜 r，下一个神奇数字要么是 $L*q+A$ 要么是 $L*q+B$，依此类推。

方法二： 二分搜索

思路

小于等于 x 的神奇数字的个数是一个单调递增函数，可以用二分搜索来做这道题。

算法

设 $L = lcm(A, B)$，为 A，B 的**最小公倍数**，$L = \frac{A*B}{gcd(A,B)}$。

$f(x)$ 为小于等于 x 的神奇数字的个数。$f(x) = \lfloor \frac{x}{A} \rfloor + \lfloor \frac{x}{B} \rfloor - \lfloor \frac{x}{L} \rfloor$。有 $\lfloor \frac{x}{A} \rfloor$ 个数字能被 A 整除的，$\lfloor \frac{x}{B} \rfloor$ 个数字能被 B 整除，同时需要减去 $\lfloor \frac{x}{L} \rfloor$ 个能被 A，B 整除的数。

复杂度分析

- 时间复杂度：$O(\log(N * \max(A, B)))$。
- 空间复杂度：$O(1)$。

```
class Solution {
    public int nthMagicalNumber(int N, int A, int B) {
        int mod = 1000000007;
        int L = A / gcd(A, B) * B;
        long left = 0, right = (long) 1e15;
        while (left < right) {
            long mid = left + (right - left) / 2;
            if (mid / A + mid /B - mid /L < N) left = mid + 1;
            else right = mid;
        }
        return (int) (left % mod);
    }
    public int gcd(int a, int b) {
        int max, min;
        max = (a > b) ? a : b;
        min = (a < b) ? a : b;
        if (max % min != 0) return gcd(min, max % min);
        else return min;
    }
}
```

30、Rand7 求 rand10（470）

更多硬核内容就在公众号「yes 的练级攻略」

470. 用 Rand7() 实现 Rand10()

难度 中等 92 收藏 分享 切换为英文 关注 反馈

已有方法 `rand7` 可生成 1 到 7 范围内的均匀随机整数，试写一个方法 `rand10` 生成 1 到 10 范围内的均匀随机整数。

不要使用系统的 `Math.random()` 方法。

示例 1:

```
输入：1
输出：[7]
```

示例 2:

```
输入：2
输出：[8,4]
```

示例 3:

```
输入：3
输出：[8,1,10]
```

提示:

1. `rand7` 已定义。
2. 传入参数: `n` 表示 `rand10` 的调用次数。

进阶:

1. `rand7()` 调用次数的 期望值 是多少？
2. 你能否尽量少调用 `rand7()` ？

https://leetcode-cn.com/problems/implement-rand10-using-rand7/solution/cong-zui-ji-chu-de-jiang-qi-ru-he-zuo-dao-jun-yun-/

已知 rand_N() 可以等概率的生成[1, N]范围的随机数，(rand_X() - 1) × Y + rand_Y() ==> 可以等概率的生成[1, X * Y]范围的随机数即实现了 rand_XY()

如果是较大的数变成较小的数，rand_M() = rand_N() % N + 1；M 必须是 N 的倍数，否则不是等概率的。

方法 1：拒绝采样

```
/**
 * The rand7() API is already defined in the parent class SolBase.
 * public int rand7();
 * @return a random integer in the range 1 to 7
 */
class Solution extends SolBase {
    public int rand10() {
        while (true) {
            int num = (rand7() - 1) * 7 + rand7();
            if (num <= 40) return num % 10 + 1;
        }
    }
```

}

31、快乐数（202）

202. 快乐数

难度 简单 422 收藏 分享 切换为英文 关注 反馈

编写一个算法来判断一个数 `n` 是不是快乐数。

「快乐数」定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是 **无限循环** 但始终变不到 1。如果 **可以变为** 1，那么这个数就是快乐数。

如果 `n` 是快乐数就返回 `True` ；不是，则返回 `False` 。

示例：

输入： 19
输出： true
解释：
$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

禁止商用

https://leetcode-cn.com/problems/happy-number/solution/kuai-le-de-zhi-shi-dian-zeng-jia-liao-by-sweetiee/

链表找环

每个数字都会根据 *各位平方和* 指向另一个数字，所以从任意数字开始进行 *各位平方和* 的迭代操作，就相当于在链表上游走。如果 **无限循环** 但始终变不到 1，那说明肯定是链表游走到了环。

为什么呢？ ~~（此处应该有图）~~

为什么无限循环就一定是走到了环了呢？而不是链表上的数字越变越大？这里我来给一个证明。

设 $n = \overline{a_1 a_2 ... a_m}$，$a_i$ 表示每一位上的数字，并且 $0 <= a_i <= 9$。我们有 $squareSum(n) = \sum_1^m a_i^2$

我们需要证明：

$$\lim_{n \to \infty} \frac{squareSum(n)}{n} < 1$$

以下是证明：

$$\lim_{n\to\infty}\frac{squareSum(n)}{n}$$
$$=\lim_{n\to\infty,m\to\infty}\frac{\sum_1^m a_i^2}{\overline{a_1a_2...a_m}}$$
$$\le\lim_{m\to\infty}\frac{m*9^2}{10^{m-1}}$$
$$=0$$

注意最后一步放缩，分子向上放（$a_i <= 9$），分母向下缩（m 位数肯定是大于等于 m 位数最小的那个数）。

好了，这就证明了不会有一个一直增长的链表，而且在 n 非常大的时候，通过 $squareSum(n)$ 操作会急剧变小。

链表找环 是一个很经典的问题了，使用快慢指针可破解。

```
public class Solution {
    public int squareSum(int n) {
        int sum = 0;
        while(n > 0){
            int digit = n % 10;
            sum += digit * digit;
            n /= 10;
        }
        return sum;
    }

    public boolean isHappy(int n) {
        int slow = n, fast = squareSum(n);
        while (slow != fast){
            slow = squareSum(slow);
            fast = squareSum(squareSum(fast));
        };
        return slow == 1;
    }
}
```

32、服务中心的最佳位置（1515）

更多硬核内容就在公众号「yes 的练级攻略」

1515. 服务中心的最佳位置

难度 困难 16 收藏 分享 切换为英文 关注 反馈

一家快递公司希望在新城市建立新的服务中心。公司统计了该城市所有客户在二维地图上的坐标，并希望能够以此为依据为新的服务中心选址：使服务中心 **到所有客户的欧几里得距离的总和最小**。

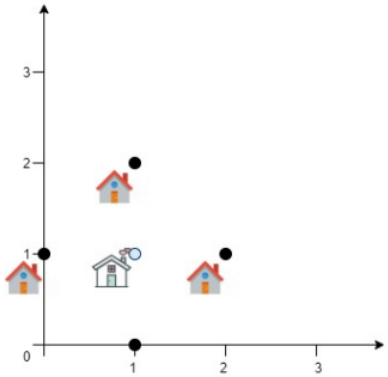
给你一个数组 `positions` ，其中 `positions[i] = [xi, yi]` 表示第 `i` 个客户在二维地图上的位置，返回到所有客户的 **欧几里得距离的最小总和** 。

换句话说，请你为服务中心选址，该位置的坐标 $[x_{centre}, y_{centre}]$ 需要使下面的公式取到最小值：

$$\sum_{i=0}^{n-1} \sqrt{(x_{centre} - x_i)^2 + (y_{centre} - y_i)^2}$$

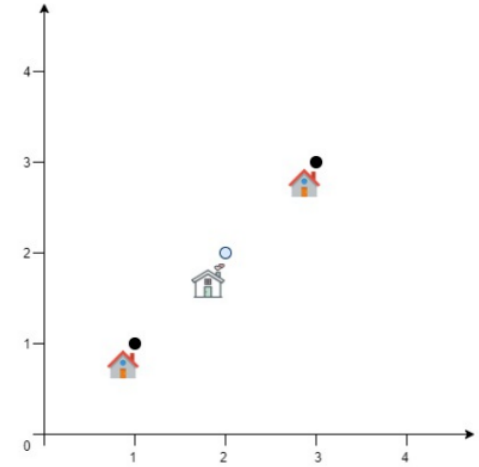
与真实值误差在 `10^-5` 之内的答案将被视作正确答案。

示例 1：



输入： positions = [[0,1],[1,0],[1,2],[2,1]]
输出： 4.00000
解释： 如图所示，你可以选 $[x_{centre}, y_{centre}] = [1, 1]$ 作为新中心的位置，这样一来到每个客户的距离就都是 1，所有距离之和为 4 ，这也是可以找到的最小值。

示例 2：



输入： positions = [[1,1],[3,3]]
输出： 2.82843
解释： 欧几里得距离可能的最小总和为 sqrt(2) + sqrt(2) = 2.82843

示例 3：

输入： positions = [[1,1]]
输出： 0.00000

更多硬核内容就在公众号「yes 的练级攻略」

示例 4：

```
输入：positions = [[1,1],[0,0],[2,0]]
输出：2.73205
解释：乍一看，你可能会将中心定在 [1, 0] 并期待能够得到最小总和，但是如果选址在 [1, 0] 距离总和为 3
如果将位置选在 [1.0, 0.5773502711] ，距离总和将会变为 2.73205
当心精度问题！
```

示例 5：

```
输入：positions = [[0,1],[3,2],[4,5],[7,6],[8,9],[11,1],[2,12]]
输出：32.94036
解释：你可以用 [4.3460852395, 4.9813795505] 作为新中心的位置
```

提示：

- `1 <= positions.length <= 50`
- `positions[i].length == 2`
- `0 <= positions[i][0], positions[i][1] <= 100`

禁止商用

https://leetcode-cn.com/problems/best-position-for-a-service-centre/solution/fu-wu-zhong-xin-de-zui-jia-wei-zhi-by-leetcode-sol/

前言

比起普通的算法题，本题更像是机器学习岗位面试中的一道数学题。题目中给定的函数是一个凸函数（convex function），因此这是一个凸优化问题，「局部最小值」就等于「全局最小值」。

需要注意的是，最终的结果 (x_c, y_c) 是给定的 $\{x_i\}$ 和 $\{y_i\}$ 的「几何中位数」。几何中位数并没有解析解，我们无法直接求出「全局最小值」。但我们可以使用求解「局部最小值」的方法，得到的结果就等于「全局最小值」。

> 为了叙述方便，本题解用 (x_c, y_c) 表示题目描述中的 (x_{center}, y_{center})。

本题解会简单科普一些常用的求解「局部最小值」的算法。由于大部分算法为迭代法，因此不给出复杂度分析。凸函数的证明过程可参考文末的「凸函数证明」部分。

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法一：梯度下降法

「梯度下降法」是机器学习中常用的一种求解局部最小值的算法。对于给定的点 (x, y)，它的梯度方向是函数值上升最快的方向，因此梯度反向就是函数值下降最快的方向。本题中需要优化的函数为：

$$f(x_c, y_c) = \sum_{i=0}^{n-1} \sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}$$

它的导数为：

$$\begin{cases} \dfrac{\partial f}{\partial x} = \displaystyle\sum_{i=0}^{n-1} \dfrac{x_c - x_i}{\sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}} \\ \dfrac{\partial f}{\partial y} = \displaystyle\sum_{i=0}^{n-1} \dfrac{y_c - y_i}{\sqrt{(x_c - x_i)^2 + (y_c - y_i)^2}} \end{cases}$$

那么梯度反向 $-\nabla f = (-\dfrac{\partial f}{\partial x}, -\dfrac{\partial f}{\partial y})$。我们从一个初始点 (x_{start}, y_{start}) 开始进行迭代，每次令

$$\begin{cases} x'_{start} = x_{start} - \alpha \cdot \dfrac{\partial f}{\partial x} \\ y'_{start} = y_{start} - \alpha \cdot \dfrac{\partial f}{\partial y} \end{cases}$$

得到一个新的点 (x'_{start}, y'_{start})，其中 α 为学习率（learning rate）。当迭代了一定次数之后，当前的点会非常接近真正的最小值点，如果我们的学习速率保持不变，迭代的结果将会在最小值点的周围来回「震荡」，无法继续接近最小值点。因此，我们需要设置学习率衰减（learning rate decay），在迭代的过程中逐步减小学习率，向最小值点逼近。

我们也可以使用机器学习中的一些技巧，例如「小批量梯度下降法」等，每次只对一批 (x_i, y_i) 进行求导并更新答案。在下面的代码中，我们令：

- 初始点 $(x_{start}, y_{start}) = \left(\dfrac{\sum_{i=0}^{n-1} x_i}{n}, \dfrac{\sum_{i=0}^{n-1} y_i}{n}\right)$，即算术平均值；
- 学习率 $\alpha = 1$；
- 学习率衰减 $\eta = 10^{-3}$；
- 当 (x_{start}, y_{start}) 与 (x'_{start}, y'_{start}) 的距离小于 10^{-7} 时结束迭代。

对于下面的代码，当批大小 $\mathrm{batchSize} = n$，为「批量梯度下降法」，可以通过本题；当 $n = 8/16$ 等常用值时，为「小批量梯度下降法」，由于具有随机性，可能得到的解并不精确，但有较高概率可以通过本题。

更多硬核内容就在公众号「yes 的练级攻略」

凸函数证明

给定 $x_0,\cdots,x_{n-1}\in\mathbb{R}$ 以及 $y_0,\cdots,y_{n-1}\in\mathbb{R}$，证明

$$f(x,y)=\sum_{i=0}^{n-1}\sqrt{(x-x_i)^2+(y-y_i)^2}$$

是 $\mathbb{R}^2$ 上的凸函数（convex function）。

证明

记 $f_i(x,y)=\sqrt{(x-x_i)^2+(y-y_i)^2}$，显然 $f_i(x,y)$ 连续可导。一阶导数为：

$$\begin{cases}\dfrac{\partial f_i}{\partial x}=\dfrac{x-x_i}{\sqrt{(x-x_i)^2+(y-y_i)^2}}\\[2ex]\dfrac{\partial f_i}{\partial y}=\dfrac{y-y_i}{\sqrt{(x-x_i)^2+(y-y_i)^2}}\end{cases}$$

二阶导数为：

$$\begin{cases}\dfrac{\partial^2 f_i}{\partial x^2}=\dfrac{(y-y_i)^2}{\left((x-x_i)^2+(y-y_i)^2\right)^{\frac{3}{2}}}\\[2ex]\dfrac{\partial^2 f_i}{\partial y^2}=\dfrac{(x-x_i)^2}{\left((x-x_i)^2+(y-y_i)^2\right)^{\frac{3}{2}}}\\[2ex]\dfrac{\partial^2 f_i}{\partial x\partial y}=\dfrac{\partial^2 f_i}{\partial y\partial x}=-\dfrac{(x-x_i)(y-y_i)}{\left((x-x_i)^2+(y-y_i)^2\right)^{\frac{3}{2}}}\end{cases}$$

对应的 Hessian 矩阵

$$H(f_i)=\begin{bmatrix}\dfrac{\partial^2 f_i}{\partial x^2} & \dfrac{\partial^2 f_i}{\partial x\partial y}\\[2ex]\dfrac{\partial^2 f_i}{\partial y\partial x} & \dfrac{\partial^2 f_i}{\partial y^2}\end{bmatrix}$$

的主子式

$$\begin{cases}\dfrac{\partial^2 f_i}{\partial x^2}\ge 0\\[2ex]\dfrac{\partial^2 f_i}{\partial y^2}\ge 0\\[2ex]\dfrac{\partial^2 f_i}{\partial x^2}\dfrac{\partial^2 f_i}{\partial y^2}-\dfrac{\partial^2 f_i}{\partial x\partial y}\dfrac{\partial^2 f_i}{\partial y\partial x}=0\end{cases}$$

均大于等于零。因此 Hessian 矩阵为半正定矩阵，即 $f_i(x,y)$ 是凸函数。

因此 $f=\sum_{i=0}^{n-1}f_i$ 也是凸函数。

方法 1：梯度下降法（97ms）

```
class Solution {
    public double getMinDistSum(int[][] positions) {
        double eps = 1e-7;
        double alpha = 1;
```

```
        double decay = 1e-3;
        int n = positions.length;
        // 调整批大小
        int batchSize = n;
        double x = 0.0, y = 0.0;
        for (int[] pos : positions) {
            x += pos[0];
            y += pos[1];
        }
        x /= n;
        y /= n;
        while (true) {
            // 将数据随机打乱
            shuffle(positions);
            double xPrev = x;
            double yPrev = y;
            for (int i = 0; i < n; i += batchSize) {
                int j = Math.min(i + batchSize, n);
                double dx = 0.0, dy = 0.0;
                // 计算导数，注意处理分母为零的情况
                for (int k = i; k < j; ++k) {
                    int[] pos = positions[k];
                    dx += (x - pos[0]) / (Math.sqrt((x - pos[0]) * (x - pos[0])
 + (y - pos[1]) * (y - pos[1])) + eps);
                    dy += (y - pos[1]) / (Math.sqrt((x - pos[0]) * (x - pos[0])
 + (y - pos[1]) * (y - pos[1])) + eps);
                }
                x -= alpha * dx;
                y -= alpha * dy;
                // 每一轮迭代后，将学习率进行衰减
                alpha *= (1.0 - decay);
            }
            // 判断是否结束迭代
            if (Math.sqrt((x - xPrev) * (x - xPrev) + (y - yPrev) * (y - yPrev)
) < eps) {
                break;
            }
        }
        return getDist(x, y, positions);
    }
    public void shuffle(int[][] positions) {
        Random rand = new Random();
        int n = positions.length;
        for (int i = 0; i < n; i++) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                int x = positions[i][0], y = positions[i][1];
                int randIndex = rand.nextInt(n);
                positions[i][0] = positions[randIndex][0];
                positions[i][1] = positions[randIndex][1];
                positions[randIndex][0] = x;
                positions[randIndex][1] = y;
            }
        }
        // 计算服务中心 (xc, yc) 到客户的欧几里得距离之和
        public double getDist(double xc, double yc, int[][] positions) {
            double ans = 0;
            for (int[] pos : positions) {
                ans += Math.sqrt((pos[0] - xc) * (pos[0] - xc) + (pos[1] - yc) * (p
os[1] - yc));
            }
            return ans;
        }
}
```

方法二：爬山法

如果给定的凸函数很难进行求导（或者读者懒得求导）怎么办？注意到梯度反向 $-\nabla f = (-\frac{\partial f}{\partial x}, -\frac{\partial f}{\partial y})$ 实际上可以拆分成：

$$
\begin{aligned}
(-\frac{\partial f}{\partial x}, -\frac{\partial f}{\partial y}) &= (-\frac{\partial f}{\partial x}, 0) + (0, -\frac{\partial f}{\partial y}) \\
&= -\frac{\partial f}{\partial x} \cdot (1, 0) - \frac{\partial f}{\partial y} \cdot (0, 1)
\end{aligned}
$$

即我们「横向」移动若干个单位长度，「纵向」移动若干个单位长度，也可以得到和梯度下降一样的结果。因此我们只需要考虑四个方向，即单位向量 $(1, 0)$，$(-1, 0)$，$(0, 1)$ 和 $(0, -1)$。

初始时，我们选择一个「步长」step，表示每次移动的距离。如果我们当前在位置 (x, y)，我们就依次枚举四个方向 (d_x, d_y)，并判断 $(x + \text{step} \cdot d_x, y + \text{step} \cdot d_y)$ 对应的函数值是否更小。如果找到一个满足要求的方向，我们就进行移动；否则说明我们当前的「步长」较大，直接越过了最值点，因此调整步长为原来的一半，直到步长小于给定的阈值 ϵ。

在下面的代码中，我们令：

- 初始点 $(x_{start}, y_{start}) = \left(\frac{\sum_{i=0}^{n-1} x_i}{n}, \frac{\sum_{i=0}^{n-1} y_i}{n} \right)$，即算术平均值；
- 步长 $\text{step} = 1$；
- 步长阈值 $\epsilon = 10^{-7}$；

就可以通过本题。

这种方法叫做「爬山法」：如果我们将一个凸函数倒过来看，它会形成一个类似「山峰」的形状。在我们攀登顶峰（找到最大值点）的过程中，我们可能并不知道顶峰在哪里。但如果我们每一步都环顾四周，走向更高的地方（检查四个方向的函数值是否更大），那么我们最终总能到达山峰。

方法 2：爬山法（10ms）

```
class Solution {
    private static int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    public double getMinDistSum(int[][] positions) {
```

```
        double eps = 1e-7;
        double step = 1;
        double decay = 0.5;
        int n = positions.length;
        double x = 0.0, y = 0.0;
        for (int[] pos : positions) {
            x += pos[0];
            y += pos[1];
        }
        x /= n;
        y /= n;
        while (step > eps) {
            boolean modified = false;
            for (int i = 0; i < 4; ++i) {
                double xNext = x + step * dirs[i][0];
                double yNext = y + step * dirs[i][1];
                if (getDist(xNext, yNext, positions) < getDist(x, y, positions)
) {
                    x = xNext;
                    y = yNext;
                    modified = true;
                    break;
                }
            }
            if (!modified) {
                step *= (1.0 - decay);
            }
        }
        return getDist(x, y, positions);
    }
    // 计算服务中心 (xc, yc) 到客户的欧几里得距离之和
    public double getDist(double xc, double yc, int[][] positions) {
        double ans = 0;
        for (int[] pos : positions) {
            ans += Math.sqrt((pos[0] - xc) * (pos[0] - xc) + (pos[1] - yc) * (p
os[1] - yc));
        }
        return ans;
    }
}
```

更多硬核内容就在公众号「yes 的练级攻略」

方法三：三分查找

除了寻找下降的方向之外，我们也可以使用「三分查找」，逐步缩小范围，找到最小值点。

考虑一个 $\mathbb{R}$ 上的凸函数 $y = f(x)$，它的定义域为 $[L, R]$，我们可以用三分查找的方式将不可能包含最小值点的范围排除：

- 我们取两个点 x_1, x_2，满足 $L < x_1 < x_2 < R$，记 $y_1 = f(x_1)$，$y_2 = f(x_2)$；
- 如果 $y_1 > y_2$，那么最小值点一定不在 $[L, x_1]$ 内；如果 $y_1 < y_2$，那么最小值点一定不在 $[x_2, R]$ 内。

证明也很容易，只需要用到凸函数的定义。假设最小值点在 $[L, x_1]$ 内，记为 x_m。由于 $y_1 > y_2$，那么最小值点显然不为 x_1，即 $x_1 \neq x_m$。因此有 $x_m < x_1 < x_2$ 且 $f(x_m) < f(x_1)$，$f(x_2) < f(x_1)$。由于 f 是凸函数，使用定比分点可以得到：

$$\frac{x_2 - x_1}{x_2 - x_m} f(x_m) + \frac{x_1 - x_m}{x_2 - x_m} f(x_2) \geq f\left(\frac{x_2 - x_1}{x_2 - x_m} \cdot x_m + \frac{x_1 - x_m}{x_2 - x_m} \cdot x_2\right) = f(x_1)$$

然而：

$$\frac{x_2 - x_1}{x_2 - x_m} f(x_m) + \frac{x_1 - x_m}{x_2 - x_m} f(x_2) < \frac{x_2 - x_1}{x_2 - x_m} f(x_1) + \frac{x_1 - x_m}{x_2 - x_m} f(x_1) = f(x_1)$$

产生了矛盾！因此最小值点一定不在 $[L, x_1]$ 内。对于 $y_1 < y_2$ 的情况同理。

我们可以取 x_1, x_2 为 $[L, R]$ 的三等分点，即：

$$\begin{cases} x_1 = \dfrac{2L + R}{3} \\ x_2 = \dfrac{L + 2R}{3} \end{cases}$$

这样我们每次就可以排除三分之一的定义域。当定义域的宽度 $R - L$ 收敛至给定的阈值 ϵ 时，就可以结束三分查找。此时 L 和 R 足够接近，可以将 $[L, R]$ 内的所有值都视作极值点。

题目中的函数 $f(x_c, y_c)$ 是二元函数，我们可以使用「三分查找」套「三分查找」的方法。外层的三分查找用来确定 x_c 的范围，每次三分查找时，取定义域 $[x_L, x_R]$ 中的两个三等分点 x_1, x_2，并分别固定 x_c 的值为 x_1 和 x_2，对 y_c 进行三分查找。将得到的最优解进行比较，以此选择忽略 $[x_L, x_1]$ 或 $[x_2, x_R]$。

每一层的三分查找都会在定义域的宽度小于给定的阈值 ϵ 时结束。在下面的代码中，我们令：

- 区间宽度阈值 $\epsilon = 10^{-7}$；

yes的练级攻略

方法 3：三分查找（246ms）

```
class Solution {
    public double getMinDistSum(int[][] positions) {
        double eps = 1e-7;
        double xLeft = 0.0, xRight = 100.0;
        while (xRight - xLeft > eps) {
            // 左 1/3 点
            double xFirst = (xLeft + xLeft + xRight) / 3;
            // 右 1/3 点
            double xSecond = (xLeft + xRight + xRight) / 3;
            if (checkOptimal(xFirst, positions, eps) < checkOptimal(xSecond, po
sitions, eps)) {
                xRight = xSecond;
            } else {
                xLeft = xFirst;
            }
```

```
        }
        return checkOptimal(xLeft, positions, eps);
    }
    // 计算服务中心 (xc, yc) 到客户的欧几里得距离之和
    public double getDist(double xc, double yc, int[][] positions) {
        double ans = 0;
        for (int[] pos : positions) {
            ans += Math.sqrt((pos[0] - xc) * (pos[0] - xc) + (pos[1] - yc) * (p
os[1] - yc));
        }
        return ans;
    }
    // 固定 xc，使用三分法找出最优的 yc
    public double checkOptimal(double xc, int[][] positions, double eps) {
        double yLeft = 0.0, yRight = 100.0;
        while (yRight - yLeft > eps) {
            double yFirst = (yLeft + yLeft + yRight) / 3;
            double ySecond = (yLeft + yRight + yRight) / 3;
            if (getDist(xc, yFirst, positions) < getDist(xc, ySecond, positions
)) {
                yRight = ySecond;
            } else {
                yLeft = yFirst;
            }
        }
        return getDist(xc, yLeft, positions);
    }
}
```

33、分数到小数（166）

更多硬核内容就在公众号「yes 的练级攻略」

166. 分数到小数

难度 中等 156 收藏 分享 切换为英文 关注 反馈

给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

如果小数部分为循环小数，则将循环的部分括在括号内。

示例 1:

```
输入: numerator = 1, denominator = 2
输出: "0.5"
```

示例 2:

```
输入: numerator = 2, denominator = 1
输出: "2"
```

示例 3:

```
输入: numerator = 2, denominator = 3
输出: "0.(6)"
```

https://leetcode-cn.com/problems/fraction-to-recurring-decimal/solution/marvelzhong-deng-de-xue-xi-bi-ji-166-by-tyanyoneca/

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

执行用时1ms

本题要考虑的细节比较多，不妨先处理几个。

- 如果分子是0，直接返回0作为答案。
- 为了方便处理，若商为负，先把负号添加到答案中，然后将分子分母全部转为正数来处理。
- 为了防止溢出，全部用long类型来处理。

不妨以4/700为例，我们模拟手算的过程。
用4/700，商0余4，把商和小数点添加进答案，然后我们会把被除数更新为40。
用40/700，商0余40，我们在答案后添加一个0。被除数更新为400。
用400/700，商0余400，我们再次在答案后添加一个0。被除数更新为4000。
用**4000/700，商5余500**，添加5到答案中，被除数更新为5000。
用5000/700，商7余100，添加7，被除数更新为1000。
用1000/700，商1余300，添加1，被除数更新为3000。
用3000/700，商4余200，添加4，被除数更新为2000。
用2000/700，商2余600，添加2，被除数更新为6000。
用6000/700，商8余400，添加8，被除数更新为4000。
用**4000/700，商5余500**，此时一个完整的循环节已经出来了：(571428)。
不难发现，当某个商（本例中是5）第二次出现时，说明它第一次出现至第二次出现的前一个商为一个完整的循环节。

接下来的问题在于如何添加括号。
很容易想到，我们需要用哈希表记录所有得到过的商，以及他们在答案字符串中的位置，以便找到后添加左括号。
但是要考虑到，如果只是记录重复的商，那本例会出现重复的前导零，而它们不属于循环节。
所以我们要换一个对象作为哈希表的键。
可以注意到，当同样的商第二次出现时（本例中为5），其实同样的被除数也出现了第二次（本例为4000），见上述粗体字部分。如果用被除数代替商作为哈希表的键，既可以起到检查循环节的作用，也不用担心重复前导零的问题。

这样一来，对于小数点后面的情况，我们每做一次除法前，先检查被除数是否重复出现过，若没有，则把商添加进答案，同时把这个被除数作为键值对的键，商在答案字符串中的下标作为键值对的值，将键值对添加进哈希表。
若找到重复出现的被除数，说明完整的循环节已找到，添加右括号，然后查询哈希表，得到该被除数第一次出现时的商在字符串中的下标（即循环节的第一个数字），在该下标位置处添加左括号即可。

```
class Solution {
    public String fractionToDecimal(int numerator, int denominator) {
        if(numerator == 0)    return String.valueOf(0);
        StringBuilder res = new StringBuilder();
        if(numerator < 0 != denominator < 0)
            res.append("-");
       long num = Math.abs((long)numerator), den = Math.abs((long)denominator);
        res.append(num / den);
        if(num % den == 0)    return res.toString();
        res.append(".");
        Map<Long, Integer> ht = new HashMap<>(); //记录被除数
        while(num % den != 0) {
            num = num % den * 10;
            if(ht.containsKey(num)) {
                res.insert(ht.get(num), "(").append(")");
                break;
            }
            ht.put(num, res.length());
            res.append(num / den);
        }
```

```
            return res.toString();
        }
}
```

34、阶乘后的零（172）【阶乘尾数（面试题 16.05）】

172. 阶乘后的零

难度 简单 345 收藏 分享 切换为英文 关注

给定一个整数 *n*，返回 *n*! 结果尾数中零的数量。

示例 1:

```
输入: 3
输出: 0
解释: 3! = 6, 尾数中没有零。
```

示例 2:

```
输入: 5
输出: 1
解释: 5! = 120, 尾数中有 1 个零.
```

说明: 你算法的时间复杂度应为 *O*(log *n*) 。

https://leetcode-cn.com/problems/factorial-trailing-zeroes/solution/jie-cheng-hou-de-ling-by-leetcode/

公众号·yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法三：高效的计算因子 5

在方法 2 中，我们找到了一种计算阶乘后的零的个数的方法，而不需要实际计算阶乘。这是通过在 5 的每个倍数上循环，从 5 到 n，并计算 5 的每个倍数中有多少个因子。我们把所有这些数字加起来得到最终结果。

然而，无论是实际上还是对问题的要求来说，方法 2 仍然太慢。为了得出一个足够快的算法，我们需要做进一步改进，这个改进能使我们在对数时间内计算出我们的答案。

思考我们之前简化的算法（但不正确），它不正确是因为对于有多个因子 5 时会计算出错，例如 25。

```
fives = 0
for i from 1 to n inclusive:
    if i is divisible by 5:
        fives += 1
```

你会发现这是执行 $\frac{n}{5}$ 的低效方法。我们只对 5 的倍数感兴趣，不是 5 的倍数可以忽略,因此可以简化成:

```
fives = n / 5
tens = fives
```

那么，如何解决有多重因子的数字呢？所有包含两个及以上的因子 5 的数字都是 25 的倍数。所以我们可以简单的除以 25 来计算 25 的倍数是多少。另外，我们在 $\frac{n}{5}$ 已经计算了 25 一次，所以我们只需要再计算额外因子一次 $\frac{n}{25}$ （而不是 $2 \cdot \frac{n}{25}$ ）

所以结合起来我们得到:

```
fives = n / 5 + n / 25
tens = fives
```

但是如果有三个因子 5 呢！为了得到最终的结果，我们需要将所有的 $\frac{n}{5}$、$\frac{n}{25}$、$\frac{n}{125}$、$\frac{n}{625}$ 等相加。得到:

$$fives = \frac{n}{5} + \frac{n}{25} + \frac{n}{125} + \frac{n}{625} + \frac{n}{3125} + \cdots$$

这样看起来会一直计算下去，但是并非如此！我们使用整数除法，最终，分母将大于 n，因此当项等于 0 时，就可以停止我们的计算。

例如，如果 $n = 12345$，我们得到:

$$fives = \frac{12345}{5} + \frac{12345}{25} + \frac{12345}{125} + \frac{12345}{625} + \frac{12345}{3125} + \frac{12345}{16075} + \frac{12345}{80375} + \cdots$$

等于:

$$fives = 2469 + 493 + 98 + 19 + 3 + 0 + 0 + \cdots = 3082$$

在代码中，我们可以通过循环 5 的幂来计算。

```
fives = 0
power_of_5 = 5
while n >= power_of_5:
    fives += n / power_of_5
    power_of_5 *= 5

tens = fives
```

更多硬核内容就在公众号「yes 的练级攻略」

算法：

Python | Java

```
public int trailingZeroes(int n) {
    int zeroCount = 0;
    // We need to use long because currentMultiple can potentially become
    // larger than an int.
    long currentMultiple = 5;
    while (n >= currentMultiple) {
        zeroCount += (n / currentMultiple);
        currentMultiple *= 5;
    }
    return zeroCount;
}
```

编写此算法的另一种方法是，我们不必每次尝试 5 的幂，而是每次将 n 本身除以 5。这也是一样的，因为我们最终得到的序列是：

$$fives = \frac{n}{5} + \frac{(\frac{n}{5})}{5} + \frac{(\frac{(\frac{n}{5})}{5})}{5} + \cdots$$

注意，在第二步中，我们有 $\frac{(\frac{n}{5})}{5}$。这是因为前一步将 n 本身除以 5。等等。

如果你熟悉分数规则，你会发现 $\frac{(\frac{n}{5})}{5}$ 和 $\frac{n}{5 \cdot 5} = \frac{n}{25}$ 是一样的。这意味着序列与：

$$\frac{n}{5} + \frac{n}{25} + \frac{n}{125} + \cdots$$

因此，这种编写算法的替代方法是等价的。

因此，这种编写算法的替代方法是等价的。

Python | Java

```
public int trailingZeroes(int n) {
    int zeroCount = 0;
    long currentMultiple = 5;
    while (n > 0) {
        n /= 5;
        zeroCount += n;
    }
    return zeroCount;
}
```

复杂度分析

- 时间复杂度：$O(\log n)$。在这种方法中，我们将 n 除以 5 的每个幂。根据定义，5 的 $\log_5 n$ 幂小于或等于 n。由于乘法和除法在 32 位整数范围内，我们将这些计算视为 $O(1)$。因此，我们正在执行 $\log_5 n \cdot O(1) = \log n$ 操作
- 空间复杂度：$O(1)$，只是用了常数空间。

```
class Solution {
    public int trailingZeroes(int n) {
        int zeroCount = 0;
        long currentMultiple = 5;
        while (n > 0) {
            n /= 5;
            zeroCount += n;
        }
```

```
            return zeroCount;
        }
    }
```

35、阶乘函数后 k 个零（793）

793. 阶乘函数后K个零

难度 困难 46 收藏 分享 切换为英文 关注 反馈

f(x) 是 x! 末尾是0的数量。（回想一下 x! = 1 * 2 * 3 * ... * x，且0! = 1）

例如，f(3) = 0 ，因为3! = 6的末尾没有0；而 f(11) = 2 ，因为11!= 39916800末端有2个0。给定 K，找出多少个非负整数x，有 f(x) = K 的性质。

示例 1:
输入:K = 0
输出:5
解释: 0!, 1!, 2!, 3!, and 4! 均符合 K = 0 的条件。

示例 2:
输入:K = 5
输出:0
解释:没有匹配到这样的 x!，符合K = 5 的条件。

注意:

- K是范围在 [0, 10^9] 的整数。

https://leetcode-cn.com/problems/preimage-size-of-factorial-zeroes-function/solution/jie-cheng-han-shu-hou-kge-ling-by-leetcode/

方法一：二分查找【通过】

思路和算法

令 zeta(x) 为 x! 末尾零的个数。如果 x! 可以分解为素数的乘积，如 $(2^a * 5^b * \cdots)$ 的形式，那么 x! 末尾零的个数为 min(a, b) = b 。

zeta(x) 就是 x 除以 5 的次数之和，即 zeta(x) 等于 $\lfloor \frac{x}{5^1} \rfloor + \lfloor \frac{x}{5^2} \rfloor + \lfloor \frac{x}{5^3} \rfloor + \lfloor \frac{x}{5^4} \rfloor + \cdots$。

可以看出，zeta(x) 是一个单调递增函数，因此可以使用二分查找求解。

使用二分查找找出满足 zeta(x) = K 的最大 x 和最小 x 。由于一定存在 zeta(5a-1) < zeta(5a) = zeta(5a+1) = zeta(5a+2) = zeta(5a+3) = zeta(5a+4) < zeta(5a+5) ，即如果存在某个 x 使得 zeta(x) = K ，那么一定存在连续 5 个数的阶乘末尾零的个数都为 K ；如果不存在这样的 x ，那么阶乘末尾零的个数为 K 的数字只有 0 个。

复杂度分析

- 时间复杂度：$O(\log^2 K)$，二分查找的复杂度为 $O(\log K)$，其中每一步计算 zeta 的复杂度也为 $O(\log K)$。
- 空间复杂度：$O(\log K)$，zeta 递归调用栈的大小。

```
class Solution {
    public int preimageSizeFZF(long K) {
        long lo = K, hi = 10*K + 1;
        while (lo < hi) {
            long mi = lo + (hi - lo) / 2;
            long zmi = zeta(mi);
            if (zmi == K) return 5;
            else if (zmi < K) lo = mi + 1;
```

```
            else hi = mi;
        }
        return 0;
    }
    public long zeta(long x) {
        if (x == 0) return 0;
        return x/5 + zeta(x/5);
    }
}
```

36、计算质数（204）

204. 计数质数

难度 简单 425 收藏 分享 切换为英文 关注

统计所有小于非负整数 *n* 的质数的数量。

示例:

输入: 10
输出: 4
解释: 小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7 。

https://leetcode-cn.com/problems/count-primes/solution/java8msai-la-tuo-si-te-ni-shai-fa-by-kevin_hwang/

解题思路

首先本解法从@cang-lang-a同学的代码（https://leetcode-cn.com/problems/count-primes/solution/ting-shuo-zhe-jiao-e-la-duo-sai-shai-fa-zhi-xing-y/ ）优化而来，采用的算法是埃拉托斯特尼筛法（百度百科https://baike.baidu.com/item/%E5%9F%83%E6%8B%89%E6%89%98%E6%96%AF%E7%89%B9%E5%B0%BC%E7%AD%9B%E6%B3%95/374984）。

埃拉托斯特尼筛法的具体内容就是：
要得到自然数n以内的全部素数，必须把不大于根号n的所有素数的倍数剔除，剩下的就是素数。
同时考虑到大于2的偶数都不是素数，所以可以进一步优化成：

要得到自然数n以内的全部素数，必须把不大于根号n的所有素数的奇数倍剔除，剩下的奇数就是素数。

那么思路就很清晰了：
1、判断n是否大于2，如果大于2了那么至少有一个小于n的质数2。
2、遍历大于1小于n的所有奇数i，如果i是质数，并且i不大于根号n，那么将i的所有奇数倍都标记成合数，如果i是质数但是i大于根号n那么直接转步骤3。
3、标记完成后质数个数+1。
4、循环结束后返回质数个数。

```
class Solution {
    public int countPrimes(int n) {
        int result = 0, sqrt_n = (int) Math.sqrt(n);
        boolean[] b = new boolean[n];   // 初始化默认值都为 false，为质数标记
        if (2 < n) result++; // 如果大于 2 则一定拥有 2 这个质数
```

```
        for (int i = 3; i < n; i += 2) {  // 从 3 开始遍历，且只遍历奇数
            if (!b[i]) {  // 是质数
                if (i <= sqrt_n)//不大于根号n
                    for (int j = i; i * j < n; j += 2)
                        b[i * j] = true;    // 将当前质数的奇数倍都设置成非质数标记 true
                result++;   // 质数个数 +1
            }
        }
        return result;
    }
}
```

37、质数排列（1175）

1175. 质数排列

难度 简单 21 收藏 分享 切换为英文 关注 反馈

请你帮忙给从 1 到 n 的数设计排列方案，使得所有的「质数」都应该被放在「质数索引」（索引从 1 开始）上；你需要返回可能的方案总数。

让我们一起来回顾一下「质数」：质数一定是大于 1 的，并且不能用两个小于它的正整数的乘积来表示。

由于答案可能会很大，所以请你返回答案 模 mod 10^9 + 7 之后的结果即可。

示例 1：

输入： n = 5
输出： 12
解释： 举个例子，[1,2,5,4,3] 是一个有效的排列，但 [5,2,3,4,1] 不是，因为在第二种情况里质数 5 被错误地放在索引为 1 的位置上。

示例 2：

输入： n = 100
输出： 682289015

提示：

- 1 <= n <= 100

https://leetcode-cn.com/problems/prime-arrangements/solution/javashuang-bai-by-xun-lu-1/

解题思路

直接撸出100以内的25个质数，计算n以内质数的个数和合数的个数
然后质数全排列乘以合数全排列，在计算时要及时取模防止溢出

```
class Solution {
    int mod = 1000000007;
    public int numPrimeArrangements(int n) {
        int[] zhishu={
            2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,
            83,89,97
```

```
        };
        //计算质数个数
        int nzhishu=0;
        for(int c : zhishu){
            if(n >= c){
                nzhishu++;
            }
        }
        //计算非质数的个数
        int nfeizhishu = n - nzhishu;
        long ans = 1;
        for(int i = nfeizhishu; i > 1; i--){
            if(ans < mod / i){
                ans *= i;
            }
            else{
                ans = ans * i % mod;
            }
        }
        for(int i = nzhishu; i > 1; i--){
            if(ans < mod / i){
                ans *= i;
            }
            else{
                ans = ans * i % mod;
            }
        }
        return (int) ans % mod;
    }
}
```

38、递归乘法（面试题 08.05）

面试题 08.05. 递归乘法

难度 中等 22 收藏 分享 切换为英文 关注 反馈

递归乘法。 写一个递归函数，不使用 * 运算符， 实现两个正整数的相乘。可以使用加号、减号、位移，但要吝啬一些。

示例1:

```
输入: A = 1, B = 10
输出: 10
```

示例2:

```
输入: A = 3, B = 4
输出: 12
```

提示:

1. 保证乘法范围不会溢出

方法 1：递归

```
class Solution {
    public int multiply(int A, int B) {
        return B == 0 ? 0 : (A > B ? A + multiply(A, B-1) : multiply(B, A));
    }
}
```

https://leetcode-cn.com/problems/recursive-mulitply-lcci/solution/java-jiu-shi-bu-yong-di-gui-by-npe_tle/

思路

1. 首先，求得 `A` 和 `B` 的最小值和最大值;
2. 然后，可以对其中的最小值当做乘数（为什么选最小值，因为选最小值当乘数，可以算的少），将其拆分成 `2` 的幂的和，即$min = a_0 * 2^0 + a_1 * 2^1 + ... + a_i * 2^i + ...$，其中$a_i$取 `0` 或者 `1` 。其实就是用二进制的视角去看待 `min` ，比如 `12` 用二进制表示就是 `1100` ，即 `1000+0100` 。举个例子， `13 * 12 = 13 * (8 + 4) = 13 * 8 + 13 * 4 = (13 << 3) + (13 << 2)` ;具体详见如下代码:

复杂度分析

- 时间复杂度：$O(logn)$，n不会超过65536。
- 空间复杂度：$O(1)$。

方法 2：位运算

```
class Solution {
    public int multiply(int A, int B) {
        int min = Math.min(A, B);
        int max = Math.max(A, B);
        int ans = 0;
        for (int i = 0; min != 0; i++) {
            if ((min & 1) == 1) {
                ans += max << i;
            }
            min >>= 1;
        }
        return ans;
```

```
    }
}
```

39、运算（面试题 16.09）

面试题 16.09. 运算

难度 中等 7 收藏 分享 切换为英文 关注 反馈

请实现整数数字的乘法、减法和除法运算，运算结果均为整数数字，程序中只允许使用加法运算符和逻辑运算符，允许程序中出现正负常数，不允许使用位运算。

你的实现应该支持如下操作：

- `Operations()` 构造函数
- `minus(a, b)` 减法，返回 `a - b`
- `multiply(a, b)` 乘法，返回 `a * b`
- `divide(a, b)` 除法，返回 `a / b`

示例：

```
Operations operations = new Operations();
operations.minus(1, 2); //返回-1
operations.multiply(3, 4); //返回12
operations.divide(5, -2); //返回-2
```

提示：

- 你可以假设函数输入一定是有效的，例如不会出现除法分母为0的情况
- 单个用例的函数调用次数不会超过1000次

https://leetcode-cn.com/problems/operations-lcci/solution/mian-shi-ti-1609-yun-suan-zhi-yong-liao-jia-fa-yu-/

```
class Operations {
    // 用来获取-1
    int ne = Integer.MAX_VALUE + Integer.MAX_VALUE + 1;
    long[] neCache = new long[32];// 放置 -1,-2,-4,-8...
    long[] poCache = new long[32];// 放置 1,2,4,8...
    long[] cache = new long[32];// 存放乘数或除数的倍数，1*a,2*a,4*a,8*a...主要用于快速计算，不然容易超时
    long[] cache1 = new long[32];// 存放乘数或除数的倍数 负数 -1*a,-2*a,-4*a,-8*a
    public Operations() {
        neCache[0] = ne;
        poCache[0] = 1;
        for (int i = 1; i < 32; ++i) {
            neCache[i] = neCache[i + ne] + neCache[i + ne];
            poCache[i] = poCache[i + ne] + poCache[i + ne];
        }
    }
    public int minus(int a, int b) {
        if (a == b) return 0;
        int index = 31;// 从最大值开始比较
        while (b != 0) {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        if (b > 0) {
            if (b >= poCache[index]) { // 如果 b 大于 2 的 index 次方，
                b += neCache[index];// a 与 b 同时减
                a += neCache[index];
            } else {
                index += ne;
            }
        } else { // b 小于 0 时同理
            if (b <= neCache[index]) {
                b += poCache[index];
                a += poCache[index];
            } else {
                index += ne;
            }
        }
    }
    return a;
}

public int multiply(int a, int b) {
    if (a == 0 || b == 0) return 0;
    if (a == 1) return b;
    if (b == 1) return a;
    if (a == ne) return minus(0, b);
    if (b == ne) return minus(0, a);
    int sign = (a > 0 && b > 0) || (a < 0 && b < 0) ? 1 : ne;
    // 把 b 变成正数
    if (b < 0) {
        b = minus(0, b);
    }

    cache[0] = a;
    for (int i = 1; i < 32; i++) {
        cache[i] = cache[i + ne] + cache[i + ne];
    }
    int index = 30; // 从 31 开始应该也是可以的
    int ret = 0;
    int retSign = a > 0 ? 1 : ne; // 记录返回值的符号
    while (b > 0) {
        if (b >= poCache[index]) {
            b += neCache[index];
            ret += cache[index];
            retSign = ret > 0 ? 1 : ne;// 记录返回值的符号
        } else {
```

```
                index += ne;
            }
        }
        // 根据初始值改变返回值的符号
        if ((sign < 0 && ret > 0) || (sign > 0 && ret < 0)) {
            ret = minus(0, ret);
        }
        // 结果溢出，返回值的符号会变成相反的
        if (retSign != (a > 0 ? 1 : ne)) {
            ret = minus(0, ret);
        }
        return ret;
    }

    public int divide(int a, int b) {
        if (a == 0) return 0;
        if (b == 1) return a;
        if (b == ne) return minus(0, a);
        int ret = 0;
        int sign = (a > 0 && b > 0) || (a < 0 && b < 0) ? 1 : ne;
        long nb = b;
        long pb = b;
        if (b < 0) {
            b = minus(0, b);
        } else {
            nb = minus(0, b);
        }
        if (a < 0) {
            a = minus(0, a);
        }
        cache[0] = b;
        cache1[0] = nb;
        int index = 1;
        for (; index < 32; ++index) {
            cache[index] = cache[index + ne] + cache[index + ne];
            cache1[index] = cache1[index + ne] + cache1[index + ne];
            if (cache1[index] >= a) {
                break; // 找到最大值就可以返回了，不用计算完
            }
        }
        if (index >= 32) index = 31;
        while (a >= b) {
            if (a >= cache[index]) {
                ret += poCache[index];// 注意这里是 2 的 index 次方的值
```

```
                a += cache1[index];
            } else {
                index += ne;
            }
        }
        if (sign < 0) {
            ret = minus(0, ret);
        }
        return ret;
    }
}
/**
 * Your Operations object will be instantiated and called as such:
 * Operations obj = new Operations();
 * int param_1 = obj.minus(a,b);
 * int param_2 = obj.multiply(a,b);
 * int param_3 = obj.divide(a,b);
 */
```

40、最大数值（面试题 16.07）

面试题 16.07. 最大数值

难度 简单 51 收藏 分享 切换为英文 关注 反馈

编写一个方法，找出两个数字 a 和 b 中最大的那一个。不得使用if-else或其他比较运算符。

示例：

```
输入： a = 1, b = 2
输出： 2
```

https://blog.csdn.net/weixin_44353507/article/details/107369013

一种非常巧妙的位运算方法解决该题 由于有些复杂 单独写下来

首先我们要考虑两种情况 a 和 b之间的加减为不会造成溢出

所以分为溢出 不溢出

而溢出和不溢出要分为a b同号 a b异号的情况

还有就是两者同号 都为正 都为负的情况

不溢出

当 先通过int k = b - a >>> 31; 无符号右移 来算出最高位 其中负数返回1 正数返回0

①当b为正数 a为负数时

此时k=0
int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=1　bSign=0

接着通过异或运算　int diff = aSign ^ bSign; 这是为了区分同号还是异号 同号为0 异号为1

得到diff为1

k = k & (diff ^ 1) | bSign & diff=0 & (1^1)|0&1=0　这里有一点要提 就是符号运算的优先级问题 可以看下表

运算符	优先级
postfix	`expr++ expr--`
unary	`++ expr -- expr + expr - expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

此时我们知道 与 异或 或 是按这个顺序来的 千万不要以为是从左往右的顺序哦!

最后return a * k + b * (k ^ 1)=a*0+b*1=b　最终返回b　这是第一种情况

②当a为正数 b为负数时

int k = b - a >>> 31 b-a为负

此时k=1

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=0 bSign=1

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为1

k = k & (diff ^ 1) | bSign & diff=1 & (1^1)|1&1=1

最后return a * k + b * (k ^ 1)=a*1+b*0=a 最终返回a

③当ab为正 且a>b时

int k = b - a >>> 31 b-a为负

此时k=1

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=0 bSign=0

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为0

k = k & (diff ^ 1) | bSign & diff=1 & (1^0) | 0 & 0=1

最后return a * k + b * (k ^ 1)=a*1+b*0=a 最终返回a

④当ab为正 且a<b时

int k = b - a >>> 31 b-a为正

此时k=0

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=0 bSign=0

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为0

k = k & (diff ^ 1) | bSign & diff=0 & (1^0) | 0 & 0=0

最后return a * k + b * (k ^ 1)=a*0+b*1=a 最终返回b

更多硬核内容就在公众号「yes 的练级攻略」

⑤当ab为负 且a<b时

int k = b - a >>> 31 b-a为正

此时k=0

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=1 bSign=1

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为0

k = k & (diff ^ 1) | bSign & diff=0 & (0^1) | 1 & 0=0

最后return a * k + b * (k ^ 1)=a*0+b*1=a 最终返回b

⑥当ab为负 且a>b时

int k = b - a >>> 31 b-a为负

此时k=1

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=1 bSign=1

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为0

k = k & (diff ^ 1) | bSign & diff=1 & (0^1) | 1 & 0=1

最后return a * k + b * (k ^ 1)=a*1+b*0=a 最终返回a

溢出时

① 负溢出的情况

也就是b为负数 a为正数

k=b-a>>>31 这样不要担心会报错 当负溢出右移31位是返回0 正溢出返回1

所以k=0

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=0 bSign=1

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为1

k = k & (diff ^ 1) | bSign & diff=0 & (1^1) | 1 & 1=1

最后return a * k + b * (k ^ 1)=a*1+b*0=a 最终返回a

②正溢出的情况

也就是a为正数 b为负数 k=1

int aSign = a >>> 31, bSign = b >>> 31;

算出aSign=1 bSign=0

接着通过异或运算 int diff = aSign ^ bSign;

得到diff为1

k = k & (diff ^ 1) | bSign & diff=1 & (1^1) | 0 & 1=0

最后return a * k + b * (k ^ 1)=a*0+b*1=a 最终返回b

以上就是所有的情况了 这个算法是真的牛逼！！！ 关键是没有用到if else 所以要将那么多情况考虑进去真的很难

```
class Solution {
    public int maximum(int a, int b) {
        // 先考虑没有溢出时的情况，计算 b - a 的最高位，依照题目所给提
示 k = 1 时 a > b，即 b - a 为负
        int k = b - a >>> 31;
        // 再考虑 a b 异号的情况，此时无脑选是正号的数字
        int aSign = a >>> 31, bSign = b >>> 31;
        // diff = 0 时同号，diff = 1 时异号
        int diff = aSign ^ bSign;
        // 在异号，即 diff = 1 时，使之前算出的 k 无效，只考虑两个数字的正负关系
        k = k & (diff ^ 1) | bSign & diff;
        return a * k + b * (k ^ 1);
    }
}
```

41、范围内的数字技术（1067）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

1067. 范围内的数字计数

难度 困难 14 收藏 分享 切换为英文 关注 反馈

给定一个在 0 到 9 之间的整数 d，和两个正整数 low 和 high 分别作为上下界。返回 d 在 low 和 high 之间的整数中出现的次数，包括边界 low 和 high。

示例 1：

输入： d = 1, low = 1, high = 13
输出： 6
解释：
数字 d=1 在 1,10,11,12,13 中出现 6 次。注意 d=1 在数字 11 中出现两次。

示例 2：

输入： d = 3, low = 100, high = 250
输出： 35
解释：
数字 d=3 在 103,113,123,130,131,...,238,239,243 出现 35 次。

提示：

1. 0 <= d <= 9
2. 1 <= low <= high <= 2×10^8

https://leetcode-cn.com/problems/digit-count-in-range/solution/shu-xue-gui-na-fa-zhao-gui-lu-chun-shu-xue-by-jiac/

公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

Analysis

> 给定范围计算显然不是简单的数学模型
>
> 划归到计算 `[0, N]` 中 `d` 出现的次数会容易很多，找规律即可
>
> 接下来的难点在于 `0` 不会出现在最高位，需要进行单独的判断。

利用数学公式可以直接计算出最终的结果，该方法是依次求出数字 X 在个位、十位、百位等等出现的次数，再相加得到最终结果。这里的 $X \in [1,9]$，因为 $X = 0$ 不符合下列规律，需要单独计算。

首先要知道以下的规律，可由数学归纳法证明：

- N ∈ $[1,10]$，在它们的个位数中，任意的 X 都出现了 1 次。
- N ∈ $[1,100]$，在它们的十位数中，任意的 X 都出现了 10 次。
- N ∈ $[1,1000]$，在它们的千位数中，任意的 X 都出现了 100 次。
- ...
- N ∈ $[1,10^i]$，在它们的左数第二位（右数第 i 位）中，任意的 X 都出现了 10^{i-1} 次。

伪代码描述如下：

当计算右数第 i 位包含的 X 的个数时：

当 X != 0 时：

1. 取第 i 位左边（高位）的数字，乘以 10^{i-1}，得到 **基础值** a。
2. 取第 i 位数字，计算修正值：
   - 如果大于 X，则结果为 $a + 10^{i-1}$。
   - 如果小于 X，则结果为 a。
   - 如果等 X，则取第 i 位右边（低位）数字，设为 b，最后结果为 $a + b + 1$。

当 $X = 0$ 时，由于最高位中永远是不会包含 0 的，因此：

- 从个位累加到左起第二位就要结束
- 第 i 位的基础值是高位数字乘以 $10^{i-1} - 1$

时间复杂度：$O(\log_{10} n)$

Solution【数学】

> 执行用时：1 ms，在 Digit Count in Range 的 Java 提交中击败了 100.00% 的用户
>
> 内存消耗：33.6 MB，在 Digit Count in Range 的 Java 提交中击败了 100.00% 的用户

复杂度分析

时间：$O(log_{10} N)$

空间：$O(1)$

```java
class Solution {
    public int digitsCount(int d, int low, int high) {
        return count(high, d) - count(low - 1, d);
    }
    /* 计算数字 d 在 1-n 中出现的次数。 */
    public int count(int n, int d) {
        int cnt = 0, k;
        for (int i = 1;(k = n / i) != 0;i *= 10) {
```

```
                // 高位的数字。
                int high = k / 10;
                if (d == 0) {
                    if (high != 0) {
                        high--;
                    } else {
                        break;
                    }
                }
                cnt += high * i;
                // 当前位的数字。
                int cur = k % 10;
                if (cur > d) {
                    cnt += i;
                } else if (cur == d) {
                    // n - k * i 为低位的数字。
                    cnt += n - k * i + 1;
                }
            }
            return cnt;
        }
    }
```

二十一、大数据

1、推荐结果打散（lintcode1166）

1166. 推荐结果打散

某个推荐系统会推荐出一个由视频和图片元素组成的列表。**x** 表示元素编号，视频用 **V_x** 表示，图片用 **P_x** 表示。
现在需要对这些元素进行打散，打散规则如下：

1. 第一个[图片P]出现的位置不变；
2. 从第一个[图片P]后开始，每 **n** 个元素里面恰好出现1个[图片P]；
3. 图片之间的相对顺序不变；

不能满足打散规则的元素需要舍去。给出元素列表和 **n** 的值，请返回打散后的元素列表。

样例
样例1：
输入样例：
elements=["V_0", "V_1", "V_2", "P_3", "P_4", "P_5", "V_6", "P_7", "V_8", "V_9"] , n=3
输出样例：
["V_0", "V_1", "V_2", "P_3", "V_6", "V_8", "P_4", "V_9"]
样例解释：
P_3 需出现在从0开始的第3位，由于每3个元素中恰好一个 P，P_5 和 P_7 满足不了打散要求，被删除。

样例2：
输入样例：
elements=["V_0", "P_1", "V_2", "V_3", "V_4"] , n=2
输出样例：
["V_0", "P_1", "V_2"]
样例解释：
P_1 需出现在从0开始的第1位，由于此后每2个元素中恰好一个 P，V_3 和 V_4 满足不了打散要求，被删除。

注意事项
1 ≤元素个数 ≤ 10000
2 ≤ n ≤ 元素个数
保证元素中至少有 1 个 [图片P]

更多硬核内容就在公众号「yes 的练级攻略」

本题可以利用双队列进行模拟。利用队列分别存储 P, V元素，按要求每 n 个元素需有 1 个 P 元素，因此对于每 n 个元素，弹出 1 个 P 元素，弹出 n-1 个 V 元素即可。 需要注意第一个 P 元素的特殊处理。

时间复杂度：O(N), N 为元素个数 空间复杂度：O(N), N 为元素个数

```
public class Solution {
    /**
     * @param elements: A list of recommended elements.
     * @param n:              [picture P] can appear at most 1 in every n
     * @return: Return the scattered result.
     */
    public List<String> scatter(List<String> elements, int n) {
        // write your code here
        // 定义双队列，分别存储图片元素 P, 视频元素 V
        Queue<String> queueP = new LinkedList<String>();
        Queue<String> queueV = new LinkedList<String>();

        // 第 1 个图片 P 出现的位置
        int firstP = -1;
        for (int i = 0; i < elements.size(); i++) {
            if (elements.get(i).charAt(0) == 'P') {
                // 第 1 个图片 P
                if (firstP == -1) {
                    firstP = i;
                }
                // 该元素加入 P 队列
                queueP.offer(elements.get(i));
            } else {
                // 该元素加入 V 队列
                queueV.offer(elements.get(i));
            }
        }
        // 定义打散后的结果序列
        List<String> result = new ArrayList<>();

        // 将出现第 1 个 P 前的 V 元素存入结果序列
        while (firstP > 0) {
            firstP--;
            result.add(queueV.element());
            queueV.poll();
        }

        // 辅助变量
        int step = 0;
        while (!queueP.isEmpty()) {
            // 每 n 个元素中需要恰好有 1 个 P
```

禁止商用 公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

```
                result.add(queueP.element());
                queueP.poll();
                step = n;

                // 将 n-1 个 V 元素存入结果序列
                while (!queueV.isEmpty() && step > 1) {
                    result.add(queueV.element());
                    queueV.poll();
                    step--;
                }

                // 若当前已经无法满足打散要求，直接结束
                if (step > 1) {
                    break;
                }
            }
            return result;
        }
}
```

禁止商用

2、关联查询（lintcode1145）

1145. 关联查询 中文

给定一张员工表datalist1，存储员工ID，员工姓名。
给定一张员工工作时长表datalist2，存储员工ID，月份，工时。
计算每个员工1-3月每月工时及总工时。

输入描述：
[[员工ID,员工姓名],[员工ID,员工姓名],...]
[[员工ID,月份,工时,月份,工时,月份,工时],[员工ID,月份,工时,月份,工时,月份,工时],...]

输出描述：
[[员工姓名,一月份工时，二月份工时，三月份工时，总工时],[员工姓名,一月份工时，二月份工时，三月份工时，总工时],...]

样例
输入：
[["1","zhangwei01"]]
[["1","01","200","02","150","03","196"]]
输出：
[["zhangwei01","200","150","196","546"]]
注意事项
两张表均按员工ID从小到大依次给出相关数据，返回的表中的数据也应按员工ID从小到大依次排列。

```
public class Solution {
    /**
     * @param datalist1: a list represents the employee table
     * @param datalist2: a list represents the employee hours table
     * @return: Returns a list of strings represents the datalist3
     */
    public List<List<String>> getList(List<List<String>> datalist1, List<List<String>> datalist2) {
        // 员工人数
        int n = datalist1.size();
```

```
            List<List<String>> datalist3 = new ArrayList();
            for (int i = 0; i < n; i++) {
                datalist3.add(new ArrayList());
                datalist3.get(i).add(datalist1.get(i).get(1));
                // sum 存三个月份的工时总和
                int sum = 0;
                for (int j = 2; j < 7; j += 2) {//偶数位置是工时
                    datalist3.get(i).add(datalist2.get(i).get(j));
                    sum += Integer.parseInt(datalist2.get(i).get(j)); //把()里的内容转换成整数
                }
                datalist3.get(i).add("" + sum);
            }
            return datalist3;
        }
}
```

二十二、多线程

1、哲学家进程（1226）

1226. 哲学家进餐

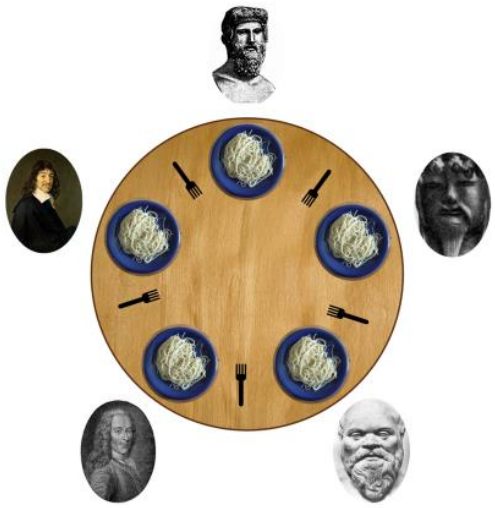
难度 中等 37 收藏 分享 切换为英文 关注 反馈

5 个沉默寡言的哲学家围坐在圆桌前，每人面前一盘意面。叉子放在哲学家之间的桌面上。（5 个哲学家，5 根叉子）

所有的哲学家都只会在思考和进餐两种行为间交替。哲学家只有同时拿到左边和右边的叉子才能吃到面，而同一根叉子在同一时间只能被一个哲学家使用。每个哲学家吃完面后都需要把叉子放回桌面以供其他哲学家吃面。只要条件允许，哲学家可以拿起左边或者右边的叉子，但在没有同时拿到左右叉子时不能进食。

假设面的数量没有限制，哲学家也能随便吃，不需要考虑吃不吃得下。

设计一个进餐规则（并行算法）使得每个哲学家都不会挨饿；也就是说，在没有人知道别人什么时候想吃东西或思考的情况下，每个哲学家都可以在吃饭和思考之间一直交替下去。



问题描述和图片来自维基百科 wikipedia.org

更多硬核内容就在公众号「yes 的练级攻略」

哲学家从 0 到 4 按 **顺时针** 编号。请实现函数 `void wantsToEat(philosopher, pickLeftFork, pickRightFork, eat, putLeftFork, putRightFork)`：

- `philosopher` 哲学家的编号。
- `pickLeftFork` 和 `pickRightFork` 表示拿起左边或右边的叉子。
- `eat` 表示吃面。
- `putLeftFork` 和 `putRightFork` 表示放下左边或右边的叉子。
- 由于哲学家不是在吃面就是在想着啥时候吃面，所以思考这个方法没有对应的回调。

给你 5 个线程，每个都代表一个哲学家，请你使用类的同一个对象来模拟这个过程。在最后一次调用结束之前，可能会为同一个哲学家多次调用该函数。

示例：

```
输入：n = 1
输出：[[4,2,1],[4,1,1],[0,1,1],[2,2,1],[2,1,1],[2,0,3],[2,1,2],[2,2,2],[4,0,3],[4,1,2],[0,2,1],[4,2,2],[3,2,1],[3,1,1],
[0,0,3],[0,1,2],[0,2,2],[1,2,1],[1,1,1],[3,0,3],[3,1,2],[3,2,2],[1,0,3],[1,1,2],[1,2,2]]
解释：
n 表示每个哲学家需要进餐的次数。
输出数组描述了叉子的控制和进餐的调用，它的格式如下：
output[i] = [a, b, c] (3个整数)
- a 哲学家编号。
- b 指定叉子：{1 : 左边, 2 : 右边}.
- c 指定行为：{1 : 拿起, 2 : 放下, 3 : 吃面}。
如 [4,2,1] 表示 4 号哲学家拿起了右边的叉子。
```

提示：

- `1 <= n <= 60`

东北树州

https://leetcode-cn.com/problems/the-dining-philosophers/solution/1ge-semaphore-1ge-reentrantlockshu-zu-by-gfu/

```
class DiningPhilosophers {
    //1 个Fork 视为1 个ReentrantLock，5 个叉子即5 个ReentrantLock，将其都放入数组中
    private final ReentrantLock[] lockList = {
            new ReentrantLock(),
            new ReentrantLock(),
            new ReentrantLock(),
            new ReentrantLock(),
            new ReentrantLock()};
    //构造函数
    public DiningPhilosophers() {

    }
    // call the run() method of any runnable to execute its code
    public void wantsToEat(int philosopher,
                           Runnable pickLeftFork,
                           Runnable pickRightFork,
                           Runnable eat,
                           Runnable putLeftFork,
                           Runnable putRightFork) throws InterruptedExc
eption {

        int leftFork = (philosopher + 1) % 5;     //左边的叉子 的编号
        int rightFork = philosopher;     //右边的叉子 的编号
```

更多硬核内容就在公众号「yes 的练级攻略」

```
        //编号为偶数的哲学家，优先拿起左边的叉子，再拿起右边的叉子
        if (philosopher % 2 == 0) {
            lockList[leftFork].lock();     //拿起左边的叉子
            lockList[rightFork].lock();     //拿起右边的叉子
        }
        //编号为奇数的哲学家，优先拿起右边的叉子，再拿起左边的叉子
        else {
            lockList[rightFork].lock();     //拿起右边的叉子
            lockList[leftFork].lock();     //拿起左边的叉子
        }

        pickLeftFork.run();     //拿起左边的叉子 的具体执行
        pickRightFork.run();     //拿起右边的叉子 的具体执行

        eat.run();     //吃意大利面 的具体执行

        putLeftFork.run();     //放下左边的叉子 的具体执行
        putRightFork.run();     //放下右边的叉子 的具体执行

        lockList[leftFork].unlock();     //放下左边的叉子
        lockList[rightFork].unlock();     //放下右边的叉子
    }
}
```

2、按序打印（1114）

yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

1114. 按序打印

难度 简单 168 收藏 分享 切换为英文 关注 反馈

我们提供了一个类：

```
public class Foo {
  public void first() { print("first"); }
  public void second() { print("second"); }
  public void third() { print("third"); }
}
```

三个不同的线程将会共用一个 `Foo` 实例。

- 线程 A 将会调用 `first()` 方法
- 线程 B 将会调用 `second()` 方法
- 线程 C 将会调用 `third()` 方法

请设计修改程序，以确保 `second()` 方法在 `first()` 方法之后被执行，`third()` 方法在 `second()` 方法之后被执行。

示例 1:

```
输入: [1,2,3]
输出: "firstsecondthird"
解释:
有三个线程会被异步启动。
输入 [1,2,3] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 second() 方法，线程 C 将会调用 third() 方法。
正确的输出是 "firstsecondthird"。
```

示例 2:

```
输入: [1,3,2]
输出: "firstsecondthird"
解释:
输入 [1,3,2] 表示线程 A 将会调用 first() 方法，线程 B 将会调用 third() 方法，线程 C 将会调用 second() 方法。
正确的输出是 "firstsecondthird"。
```

提示:

- 尽管输入中的数字似乎暗示了顺序，但是我们并不保证线程在操作系统中的调度顺序。
- 你看到的输入格式主要是为了确保测试的全面性。

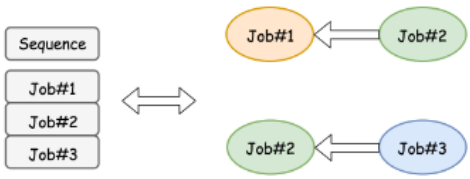
https://leetcode-cn.com/problems/print-in-order/solution/an-xu-da-yin-by-leetcode/

方法一：使用 synchronization

思路

题目要求按顺序依次执行三个方法，且每个方法都在单独的线程中运行。为了保证线程的执行顺序，可以在方法之间创建一些依赖关系，即第二个方法必须在第一个方法之后执行，第三个方法必须在第二个方法之后执行。

> 方法对之间的依赖关系形成了所有方法的特定的执行顺序。例如 `A < B`，`B < C`，则所有方法的执行顺序为 `A < B < C`。

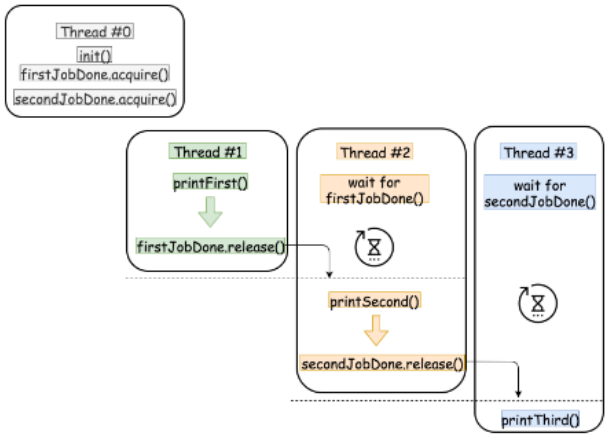


依赖关系可以通过并发机制实现。使用一个共享变量 `firstJobDone` 协调第一个方法与第二个方法的执行顺序，使用另一个共享变量 `secondJobDone` 协调第二个方法与第三个方法的执行顺序。

更多硬核内容就在公众号「yes 的练级攻略」

算法

- 首先初始化共享变量 `firstJobDone` 和 `secondJobDone`，初始值表示所有方法未执行。
- 方法 `first()` 没有依赖关系，可以直接执行。在方法最后更新变量 `firstJobDone` 表示该方法执行完成。
- 方法 `second()` 中，检查 `firstJobDone` 的状态。如果未更新则进入等待状态，否则执行方法 `second()`。在方法末尾，更新变量 `secondJobDone` 表示方法 `second()` 执行完成。
- 方法 `third()` 中，检查 `secondJobDone` 的状态。与方法 `second()` 类似，执行 `third()` 之前，需要先等待 `secondJobDone` 的状态。



实现

上述算法的实现在很大程度上取决于选择的编程语言。尽管在 Java，C++ 和 Python 中都存在互斥与信号量，但不同语言对并发机制有不同实现。

```
import java.util.concurrent.atomic.AtomicInteger;
class Foo {
    private AtomicInteger firstJobDone = new AtomicInteger(0);
    private AtomicInteger secondJobDone = new AtomicInteger(0);
    public Foo() {}
    public void first(Runnable printFirst) throws InterruptedException {
        // printFirst.run() outputs "first".
        printFirst.run();
        // mark the first job as done, by increasing its count.
        firstJobDone.incrementAndGet();
    }
    public void second(Runnable printSecond) throws InterruptedException {
        while (firstJobDone.get() != 1) {
            // waiting for the first job to be done.
        }
        // printSecond.run() outputs "second".
        printSecond.run();
        // mark the second as done, by increasing its count.
        secondJobDone.incrementAndGet();
    }
    public void third(Runnable printThird) throws InterruptedException {
        while (secondJobDone.get() != 1) {
```

```
                // waiting for the second job to be done.
            }
            // printThird.run() outputs "third".
            printThird.run();
        }
}
```

测试类

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Foo f = new Foo();
        Thread t1 = new Thread(()->{
            try {
                f.first(()->{
                    System.out.print("first");
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(()->{
            try {
                f.second(()->{
                    System.out.print("second");
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t3 = new Thread(()->{
            try {
                f.third(()->{
                    System.out.print("third");
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t1.start();
        t2.start();
        t3.start();
    }
}
```

3、交替打印 FooBar（1115）

更多硬核内容就在公众号「yes 的练级攻略」

1115. 交替打印FooBar

难度 中等 71 收藏 分享 切换为英文 关注 反馈

我们提供一个类：

```
class FooBar {
  public void foo() {
    for (int i = 0; i < n; i++) {
      print("foo");
    }
  }

  public void bar() {
    for (int i = 0; i < n; i++) {
      print("bar");
    }
  }
}
```

两个不同的线程将会共用一个 `FooBar` 实例。其中一个线程将会调用 `foo()` 方法，另一个线程将会调用 `bar()` 方法。

请设计修改程序，以确保 "foobar" 被输出 n 次。

示例 1:

```
输入: n = 1
输出: "foobar"
解释: 这里有两个线程被异步启动。其中一个调用 foo() 方法，另一个调用 bar() 方法，"foobar" 将被输出一次。
```

示例 2:

```
输入: n = 2
输出: "foobarfoobar"
解释: "foobar" 将被输出两次。
```

yes的练级攻略

https://leetcode-cn.com/problems/print-foobar-alternately/solution/javayong-synchronizedguan-jian-zi-xie-he-yong-bool/

方法 1：synchronized

```
import java.util.concurrent.atomic.AtomicInteger;
class FooBar {
    private int n;
    private boolean fooTurn = true;
    private Object lock = new Object();
    public FooBar(int n) {
        this.n = n;
    }
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            synchronized(lock) {
                if (!fooTurn) lock.wait();
                fooTurn = false;
                // printFoo.run() outputs "foo". Do not change or remove this line.
                printFoo.run();
```

更多硬核内容就在公众号「yes 的练级攻略」

```
                    lock.notifyAll();
                }
            }
        }
        public void bar(Runnable printBar) throws InterruptedException {
            for (int i = 0; i < n; i++) {
                synchronized(lock) {
                    if (fooTurn) lock.wait();
                    fooTurn = true;
                    // printBar.run() outputs "bar". Do not change or remove this line.
                    printBar.run();
                    lock.notifyAll();
                }
            }
        }
    }
```

测试函数

```
    public class Test{
        public static void main(String[] args) {
            FooBar p = new FooBar(10);
            new Thread(){
                public void run(){
                    try {
                        p.foo(new printFoo());
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }.start();
            new Thread(){
                public void run(){
                    try {
                        p.bar(new printBar());
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }.start();
        }
        static class printFoo implements Runnable{
            @Override
            public void run() {
                System.out.print("Foo");
            }
```

```
    }
    static class printBar implements Runnable{
        @Override
        public void run() {
            System.out.print("Bar");
        }
    }
}
```

https://leetcode-cn.com/problems/print-foobar-alternately/solution/liang-ge-xin-hao-liang-jiao-ti-kong-zhi-by-atizose/

方法 2：两个 Semaphore 交替运行

```
import java.util.concurrent.Semaphore;
public class FooBar {
    private int n;
    private Semaphore foo = new Semaphore(1);
    private Semaphore bar = new Semaphore(0);
    public FooBar(int n) {
        this.n = n;
    }
    public void foo(Runnable printFoo) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            foo.acquire();
            // printFoo.run() outputs "foo". Do not change or remove this line.
            printFoo.run();
            bar.release();
        }
    }
    public void bar(Runnable printBar) throws InterruptedException {
        for (int i = 0; i < n; i++) {
            bar.acquire();
            // printBar.run() outputs "bar". Do not change or remove this line.
            printBar.run();
            foo.release();
        }
    }
}
```

测试类

```
public class Test{
    public static void main(String[] args) throws InterruptedException {
        FooBar fooBar = new FooBar(3);
        Thread t1 = new Thread(()->{
            try {
                fooBar.foo(()->{
                    System.out.print("foo");
```

```
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(()->{
            try {
                fooBar.bar(()->{
                    System.out.print("bar");
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t2.start();
        t1.start();
    }
}
```

4、打印零与奇偶数（1116）

1116. 打印零与奇偶数

难度 中等 68 收藏 分享 切换为英文 关注 反馈

假设有这么一个类：

```
class ZeroEvenOdd {
  public ZeroEvenOdd(int n) { ... }      // 构造函数
  public void zero(printNumber) { ... }  // 仅打印出 0
  public void even(printNumber) { ... }  // 仅打印出 偶数
  public void odd(printNumber) { ... }   // 仅打印出 奇数
}
```

相同的一个 `ZeroEvenOdd` 类实例将会传递给三个不同的线程：

1. 线程 A 将调用 `zero()`，它只输出 0 。
2. 线程 B 将调用 `even()`，它只输出偶数。
3. 线程 C 将调用 `odd()`，它只输出奇数。

每个线程都有一个 `printNumber` 方法来输出一个整数。请修改给出的代码以输出整数序列 `010203040506` ...，其中序列的长度必须为 $2n$。

示例 1：

```
输入：n = 2
输出："0102"
说明：三条线程异步执行，其中一个调用 zero()，另一个线程调用 even()，最后一个线程调用odd()。正确的输出为 "0102"。
```

示例 2：

```
输入：n = 5
输出："0102030405"
```

通过次数 11,160 | 提交次数 22,746

https://leetcode-cn.com/problems/print-zero-even-odd/solution/java-sychronizedjie-fa-by-charlotte-28/

方法 1：synchronized

```
import java.util.function.IntConsumer;
class ZeroEvenOdd {
    private int n;
    private int i = 1;
    public ZeroEvenOdd(int n) {
        this.n = n;
    }
    // printNumber.accept(x) outputs "x", where x is an integer.
    public void zero(IntConsumer printNumber) throws InterruptedException {
        synchronized(this){
            while(true){
                while (i <= 2 *n && i%2 == 0) wait();
                if (i > 2*n) break;
                printNumber.accept(0);
                i++;
                notifyAll();
            }
        }
    }
    public void even(IntConsumer printNumber) throws InterruptedException {
        synchronized(this){
            while(true){
                while (i <= 2*n && (i%2 == 1 || (i%2 == 0 && i/2%2 == 1))) wait();
                if(i > 2*n) break;
                printNumber.accept(i/2);
                i++;
                notifyAll();
            }
        }
    }
    public void odd(IntConsumer printNumber) throws InterruptedException {
        synchronized(this){
            while(true){
                while (i <= 2*n &&(i%2 == 1 || (i%2 == 0 && i/2%2 == 0))) wait();
                if(i > 2*n) break;
                printNumber.accept(i/2);
                i++;
                notifyAll();
            }
        }
    }
}
```

https://leetcode-cn.com/problems/print-zero-even-odd/solution/xin-hao-liang-jie-jue-by-dan-xie-6/

方法 2：三个 Semaphore 交替运行

```
import java.util.concurrent.Semaphore;
import java.util.function.IntConsumer;
class ZeroEvenOdd {
    private int n;
    private Semaphore numzero = new Semaphore(1);
    private Semaphore numeven = new Semaphore(0);
    private Semaphore numodd = new Semaphore(0);
    public ZeroEvenOdd(int n) {
        this.n = n;
    }
    // printNumber.accept(x) outputs "x", where x is an integer.
    public void zero(IntConsumer printNumber) throws InterruptedException {
        for (int i = 1; i <= n; i++) {
            numzero.acquire();
            printNumber.accept(0);
            if ((i & 1) == 0) {
                numeven.release();
            } else {
                numodd.release();
            }
        }
    }
    public void even(IntConsumer printNumber) throws InterruptedException {
        for (int i = 2; i <= n; i += 2) {
            numeven.acquire();
            printNumber.accept(i);
            numzero.release();
        }
    }
    public void odd(IntConsumer printNumber) throws InterruptedException {
        for (int i = 1; i <= n; i += 2) {
            numodd.acquire();
            printNumber.accept(i);
            numzero.release();
        }
    }
}
```

两个都能用的测试类

```
import java.util.function.IntConsumer;
public class Test {
    public static void main(String[] args) throws InterruptedException{
```

```
        ZeroEvenOdd zeo = new ZeroEvenOdd(5);
        IntConsumer consumer = new IntConsumer() {
            @Override
            public void accept(int value) {
                System.out.print(value);
            }
        };
        Thread t1 = new Thread(() -> {
            try{
                zeo.zero(consumer);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(() -> {
            try{
                zeo.even(consumer);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t3 = new Thread(() -> {
            try{
                zeo.odd(consumer);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t1.start();
        t2.start();
        t3.start();
    }
}
```

5、Fizz Buzz（412）

更多硬核内容就在公众号「yes 的练级攻略」

412. Fizz Buzz

难度 简单 71 收藏 分享 切换为英文 关注 反

写一个程序，输出从 1 到 *n* 数字的字符串表示。

1. 如果 *n* 是3的倍数，输出“Fizz”；

2. 如果 *n* 是5的倍数，输出“Buzz”；

3.如果 *n* 同时是3和5的倍数，输出 “FizzBuzz”。

示例：

```
n = 15,

返回:
[
    "1",
    "2",
    "Fizz",
    "4",
    "Buzz",
    "Fizz",
    "7",
    "8",
    "Fizz",
    "Buzz",
    "11",
    "Fizz",
    "13",
    "14",
    "FizzBuzz"
]
```

https://leetcode-cn.com/problems/fizz-buzz/solution/java-jian-shao-qu-yu-ci-shu-jia-kuai-yun-xing-xiao/

而 唯一一点能够**加快运行效率**的是：

> 将和3、5取余的结果保存下来
> 这样，我们就不用再次对15取余，只需要将上一步的俩个结果相加即可

```java
class Solution {
    public List<String> fizzBuzz(int n) {
        if (n <= 0) return null;
        List<String> res = new ArrayList<>();
        for (int i = 1; i <= n; i++) {
            if (i % 15 == 0) {
```

```
                res.add("FizzBuzz");
            } else if (i % 3 == 0) {
                res.add("Fizz");
            } else if (i % 5 == 0) {
                res.add("Buzz");
            } else {
                res.add(String.valueOf(i));
            }
        }
        return res;
    }
}
```

6、交替打印字符串（1195）

1195. 交替打印字符串

难度 中等 29 收藏 分享 切换为英文 关注 反馈

编写一个可以从 1 到 n 输出代表这个数字的字符串的程序，但是：

- 如果这个数字可以被 3 整除，输出 "fizz"。
- 如果这个数字可以被 5 整除，输出 "buzz"。
- 如果这个数字可以同时被 3 和 5 整除，输出 "fizzbuzz"。

例如，当 `n = 15`，输出：`1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz`。

假设有这么一个类：

```
class FizzBuzz {
  public FizzBuzz(int n) { ... }               // constructor
  public void fizz(printFizz) { ... }          // only output "fizz"
  public void buzz(printBuzz) { ... }          // only output "buzz"
  public void fizzbuzz(printFizzBuzz) { ... }  // only output "fizzbuzz"
  public void number(printNumber) { ... }      // only output the numbers
}
```

请你实现一个有四个线程的多线程版 `FizzBuzz`， 同一个 `FizzBuzz` 实例会被如下四个线程使用：

1. 线程A将调用 `fizz()` 来判断是否能被 3 整除，如果可以，则输出 `fizz`。
2. 线程B将调用 `buzz()` 来判断是否能被 5 整除，如果可以，则输出 `buzz`。
3. 线程C将调用 `fizzbuzz()` 来判断是否同时能被 3 和 5 整除，如果可以，则输出 `fizzbuzz`。
4. 线程D将调用 `number()` 来实现输出既不能被 3 整除也不能被 5 整除的数字。

https://leetcode-cn.com/problems/fizz-buzz-multithreaded/solution/99shi-jian-suan-fa-by-weitongbai/

方法 1：synchronized

```
import java.util.function.IntConsumer;
class FizzBuzz {
    private int n;
    private int i = 1;
    public FizzBuzz(int n) {
        this.n = n;
```

```
    }
    // printFizz.run() outputs "fizz".
    public void fizz(Runnable printFizz) throws InterruptedException {
        synchronized (this) {
            while (i <= n) {
                if(!(i % 5 != 0 && i % 3 == 0)) wait();
                if(!(i % 5 != 0 && i % 3 == 0)) continue;
                if (i > n) break;
                printFizz.run();
                i++;
                notifyAll();
            }
        }
    }
    // printBuzz.run() outputs "buzz".
    public void buzz(Runnable printBuzz) throws InterruptedException {
        synchronized (this) {
            while (i <= n) {
                if (!(i % 5 == 0 && i % 3 != 0)) wait();
                if (!(i % 5 == 0 && i % 3 != 0)) continue;
                if (i > n) break;
                printBuzz.run();
                i++;
                notifyAll();
            }
        }
    }
    // printFizzBuzz.run() outputs "fizzbuzz".
    public void fizzbuzz(Runnable printFizzBuzz) throws InterruptedException {
        synchronized (this) {
            while (i <= n) {
                if (!(i % 5 == 0 && i % 3 == 0)) wait();
                if (!(i % 5 == 0 && i % 3 == 0)) continue;
                if (i > n) break;
                printFizzBuzz.run();
                i++;
                notifyAll();
            }
        }
    }
    // printNumber.accept(x) outputs "x", where x is an integer.
    public void number(IntConsumer printNumber) throws InterruptedException {
        synchronized (this) {
            while (i <= n) {
```

```
                if (i % 3 == 0 || i % 5 == 0) wait();
                if (i % 3 == 0 || i % 5 == 0) continue;
                if (i > n) break;
                printNumber.accept(i);
                i++;
                notifyAll();
            }
        }
    }
}
```

https://leetcode-cn.com/problems/fizz-buzz-multithreaded/solution/liang-chong-javajie-jue-fang-an-shi-yong-semaphore/

方法 2：Semaphore

```
import java.util.concurrent.Semaphore;
import java.util.function.IntConsumer;
class FizzBuzz {
    // Initialize the permit, limit to one.
    private Semaphore sema = new Semaphore(1);
    // The current number.
    private int curNum = 1;
    private int n;
    public FizzBuzz(int n) {
        this.n = n;
    }
    // printFizz.run() outputs "fizz".
    public void fizz(Runnable printFizz) throws InterruptedException {
        for(;;) {
            // Acquire the permit, try to run the logic exclusively.
            this.sema.acquire(1);
            try {
                if (this.curNum > n) {
                    return;
                }
                if ((this.curNum % 3 == 0) && (this.curNum % 5 != 0)) {
                    printFizz.run();
                    this.curNum++;
                }
            } finally {
                // Release the permit anyway.
                this.sema.release(1);
            }
        }
    }
    // printBuzz.run() outputs "buzz".
```

```
public void buzz(Runnable printBuzz) throws InterruptedException {
    for(;;) {
        this.sema.acquire(1);

        try {
            if (this.curNum > n) {
                return;
            }

            if ((this.curNum % 3 != 0) && (this.curNum % 5 == 0)) {
                printBuzz.run();
                this.curNum++;
            }
        } finally {
            this.sema.release(1);
        }
    }
}
// printFizzBuzz.run() outputs "fizzbuzz".
public void fizzbuzz(Runnable printFizzBuzz) throws InterruptedException {
    for(;;) {
        this.sema.acquire(1);

        try {
            if (this.curNum > n) {
                return;
            }

            if ((this.curNum % 3 == 0) && (this.curNum % 5 == 0)) {
                printFizzBuzz.run();
                this.curNum++;
            }
        } finally {
            this.sema.release(1);
        }
    }
}
// printNumber.accept(x) outputs "x", where x is an integer.
public void number(IntConsumer printNumber) throws InterruptedException {
    for(;;) {
        this.sema.acquire(1);

        try {
            if (this.curNum > n) {
```

```
                    return;
                }

                if ((this.curNum % 3 != 0) && (this.curNum % 5 != 0)) {
                    printNumber.accept(this.curNum);
                    this.curNum++;
                }
            } finally {
                this.sema.release(1);
            }
        }
    }
}
```

两个都能用的测试类

```
import java.util.function.IntConsumer;
public class Test {
    public static void main(String[] args) throws InterruptedException{
        FizzBuzz fb = new FizzBuzz(15);
        IntConsumer consumer = new IntConsumer() {
            @Override
            public void accept(int value) {
                System.out.print(value+ " ");
            }
        };
        Thread t1 = new Thread(() -> {
            try {
                fb.fizz(new printFizz());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t2 = new Thread(() -> {
            try {
                fb.buzz(new printBuzz());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        Thread t3 = new Thread(() -> {
            try {
                fb.fizzbuzz(new printFizzBuzz());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            });
            Thread t4 = new Thread(() -> {
                try {
                    fb.number(consumer);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
            t1.start();
            t2.start();
            t3.start();
            t4.start();
        }
        static class printFizz implements Runnable{
            @Override
            public void run() {
                System.out.print("fizz ");
            }
        }
        static class printBuzz implements Runnable{
            @Override
            public void run() {
                System.out.print("buzz ");
            }
        }
        static class printFizzBuzz implements Runnable{
            @Override
            public void run() {
                System.out.print("fizzbuzz ");
            }
        }
}
```

二十三、位运算

【位异或(^)的性质】

1、交换律

所有的程序教科书都会向初学者指出，要交换两个变量的值，必须要引入一个中间变量。但如果使用异或，就可以节约一个变量的存储空间： 设有 A,B 两个变量，存储的值分别为 a，b，则以下三行表达式将互换他们的值 表达式 （值） ：

```
int a = 10, b = 5；
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

2、结合律（即(a^b)^c == a^(b^c)）

3、对于任何数 x，都有 x^x=0，x^0=x，x^1 = ~x（和 1 异或结果为原操作数取反）

4、自反性 A^B^B = A^0 = A

问题 1：1-1000 放在含有 1001 个元素的数组中，只有唯一的一个元素值重复，其它均只出现一次。每个数组元素只能访问一次，设计一个算法，将它找出来；不用辅助存储空间，能否设计一个算法实现？

将所有的数全部异或，得到的结果与 1^2^3^...^1000 的结果进行异或，得到的结果就是重复数。

但是这个算法虽然很简单，但证明起来并不是一件容易的事情。这与异或运算的几个特性有关系。首先是异或运算满足交换律、结合律。

1^2^...^n^...^n^...^1000，无论这两个 n 出现在什么位置，都可以转换成为 1^2^...^1000^(n^n)的形式。

其次，对于任何数 x，都有 x^x=0，x^0=x。

1^2^...^n^...^n^...^1000 = 1^2^...^1000^(n^n)= 1^2^...^1000^0 = 1^2^...^1000（即序列中除了 n 的所有数的异或）。

令，1^2^...^1000（序列中不包含 n）的结果为 T

则 1^2^...^1000（序列中包含 n）的结果就是 T^n。

T^(T^n)=n。

所以，将所有的数全部异或，得到的结果与 1^2^3^...^1000 的结果进行异或，得到的结果就是重复数。

```
int result = 0;
for(int i = 1; i< length; i++){
    result ^= i;
}
for(int j = 0; j< length; j++){
    result ^= arr[j];
}
```

问题 2：大房子中有很多关闭的电灯，随机操作电灯开关，实时显示房间中电灯的状态，并允许一键关闭所有电灯。【只允许使用按位异或】

分析：

1、连续两次操作电灯开关，电灯将处于操作前状态。

2、关闭所有开关。任何数和自己异或结果为零。

实现：

1、定义“大房子”类。

lightFlags 标识房间全部电灯实时状态。

封装电灯操作方法以及一键关闭所有电灯的方法。

```
/**
 * 1、按位运算操作应用。
 * 2、走廊灯、厕所灯、厨房灯、主卧灯、次卧灯分别对应一个字段位
 *  走廊灯=0000 0001
 *  厕所灯=0000 0010
 *  厨房灯=0000 0100
 *  主卧灯=0000 1000
 *  次卧灯=0001 0000
 * @author DrodYoung
 */
```

禁止商用 公众号：yes的练级攻略

```
static class 大房子{
  private static final int 走廊灯 = (1 << 0); // 1{0000 0001}
  private static final int 厕所灯 = (1 << 1); // 2{0000 0001}
  private static final int 厨房灯 = (1 << 2); // 4{0000 0001}
  private static final int 主卧灯 = (1 << 3); // 8{0000 0001}
  private static final int 次卧灯 = (1 << 4); //16{0000 0001}
  private static final int LIGHT_MASK = 走廊灯 |
        厕所灯 |
        厨房灯 |
        主卧灯 |
        次卧灯;
  //lightFlags = 电灯状态标记字段。
  private int lightFlags = 0;

  private void 操作电灯开关(int flag){
     lightFlags ^= flag;
  }
  //任何数和自己异或结果为零。
  private void 让所有的灯关闭(){
  操作电灯开关(lightFlags);
  }
  @Override
  public String toString() {
  return "房子中电灯状况：\n"
  +"走廊灯="+(((lightFlags&走廊灯)!=0)?"开":"关")+"\n"
  +"厕所灯="+(((lightFlags&厕所灯)!=0)?"开":"关")+"\n"
  +"厨房灯="+(((lightFlags&厨房灯)!=0)?"开":"关")+"\n"
  +"主卧灯="+(((lightFlags&主卧灯)!=0)?"开":"关")+"\n"
  +"次卧灯="+(((lightFlags&次卧灯)!=0)?"开":"关")+"\n";
  }
}
```

测试

```
//1、初始化大房子对象
大房子 hourse = new 大房子();
System.out.println(hourse);
//2、操作厨房灯开关
hourse.操作电灯开关(大房子.厨房灯);
System.out.println(hourse);//厨房灯为【开】
//3、再次操作厨房灯开关
hourse.操作电灯开关(大房子.厨房灯);
System.out.println(hourse);//厨房灯为【关】
//4、依次操作主卧灯、次卧灯、走廊灯开关
hourse.操作电灯开关(大房子.主卧灯);
hourse.操作电灯开关(大房子.次卧灯);
```

禁止商用 公众号：yes的练级攻略

hourse.操作电灯开关(大房子.走廊灯);
//5、一键关闭所有灯
hourse.让所有的灯关闭();
System.out.println(hourse);
【位与(&)　的性质】
(a & 1) == 0　相当于　a % 2 == 0
【位非(~)　的性质】
取反
【位或(|) 的性质】
只要有一个为 1 就是 1
【移位】
左移(<<) ：向左移动指定的位数，低位用 0 补
右移(>>) ：向右移动指定的位数，正数右移，高位用 0 补，负数右移，高位用 1 补
无符号右移(>>>)：正数右移，高位用 0 补，负数右移，高位用 1 补，当负数使用无符号右移时，用 0 进行补位(自然而然的，就由负数变成了正数了)
注意：在这里说的是右移，高位补位的情况。正数或者负数左移，低位都是用 0 补
1.　交换数字（面试题 16.01）

面试题 16.01. 交换数字

难度 中等　24　收藏　分享　切换为英文　关注　反馈

编写一个函数，不用临时变量，直接交换 `numbers = [a, b]` 中 `a` 与 `b` 的值。

示例：

```
输入: numbers = [1,2]
输出: [2,1]
```

提示：

- `numbers.length == 2`

https://leetcode-cn.com/problems/swap-numbers-lcci/solution/javabu-shi-yong-lin-shi-bian-liang-de-3chong-fang-/
方法 1：位运算

```
class Solution {
    public int[] swapNumbers(int[] numbers) {
        numbers[0] = numbers[0] ^ numbers[1];
        numbers[1] = numbers[0] ^ numbers[1];
        numbers[0] = numbers[0] ^ numbers[1];
        return numbers;
    }
}
```

方法 2：加法（+--）

```
class Solution {
    public int[] swapNumbers(int[] numbers) {
```

```
        numbers[0] = numbers[0] + numbers[1];
        numbers[1] = numbers[0] - numbers[1];
        numbers[0] = numbers[0] - numbers[1];
        return numbers;
    }
}
```

方法 3：减法（-+-）

```
class Solution {
    public int[] swapNumbers(int[] numbers) {
        numbers[0] = numbers[0] - numbers[1];
        numbers[1] = numbers[0] + numbers[1];
        numbers[0] = numbers[1] - numbers[0];
        return numbers;
    }
}
```

2. 汉明距离（461）

461. 汉明距离

难度 简单 300 收藏 分享 切换为英文 关注 反馈

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给出两个整数 `x` 和 `y`，计算它们之间的汉明距离。

注意：
$0 \le x, y < 2^{31}$.

示例：

输入： `x = 1, y = 4`

输出： `2`

解释：

```
1   (0 0 0 1)
4   (0 1 0 0)
      ↑   ↑
```

上面的箭头指出了对应二进制位不同的位置。

https://leetcode-cn.com/problems/hamming-distance/solution/yi-ming-ju-chi-by-leetcode/

更多硬核内容就在公众号「yes 的练级攻略」

方法一：内置位计数功能

思路

大多数编程语言中，都存在各种内置计算等于 1 的位数函数。如果这是一个项目中的问题，应该直接使用内置函数，而不是重复造轮子。

但这是一个力扣问题，有人会认为使用内置函数就好像使用 *使用 LinkedList 实现 LinkedList*。对此，我们完全赞同。因此后面会有手工实现的位计数算法。

Python | Java

```
class Solution {
    public int hammingDistance(int x, int y) {
        return Integer.bitCount(x ^ y);
    }
}
```

复杂度分析

- 时间复杂度：$\mathcal{O}(1)$。
  - 该算法有两个操作。计算 XOR 花费恒定时间。
  - 调用内置的 bitCount 函数。最坏情况下，该函数复杂度为 $\mathcal{O}(k)$，其中 k 是整数的位数。在 Python 和 Java 中 Integer 是固定长度的。因此该算法复杂度恒定，与输入大小无关。
- 空间复杂度：$\mathcal{O}(1)$，使用恒定大小的空间保存 XOR 的结果。
  - 假设内置函数也使用恒定空间。

方法 1：内置函数

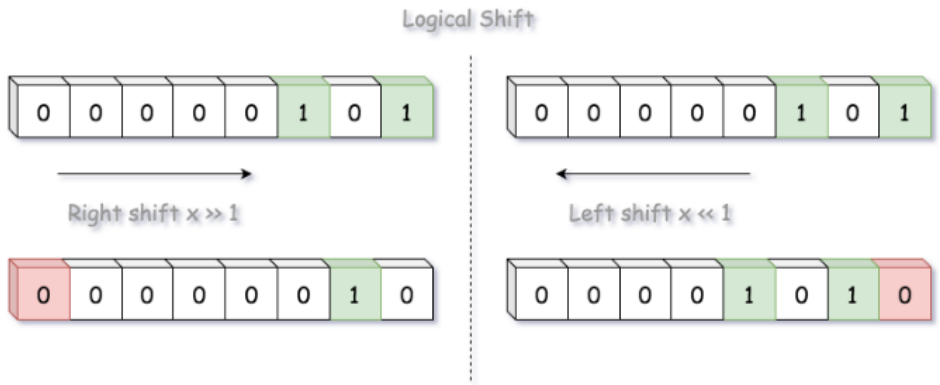
```
class Solution {
    public int hammingDistance(int x, int y) {
        return Integer.bitCount(x ^ y);
    }
}
```

方法二：移位

思路

为了计算等于 `1` 的位数，可以将每个位移动到最左侧或最右侧，然后检查该位是否为 `1`。

更准确的说，应该进行逻辑移位，移入零替换丢弃的位。



这里采用右移位，每个位置都会被移动到最右边。移位后检查最右位的位是否为 `1` 即可。检查最右位是否为 `1`，可以使用取模运算（`i % 2`）或者 AND 操作（`i & 1`），这两个操作都会屏蔽最右位以外的其他位。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$\mathcal{O}(1)$，在 Python 和 Java 中 Integer 的大小是固定的，处理时间也是固定的。 32 位整数需要 32 次迭代。
- 空间复杂度：$\mathcal{O}(1)$，使用恒定大小的空间。

方法 2：移位

```
class Solution {
    public int hammingDistance(int x, int y) {
        int xor = x ^ y;
        int distance = 0;
        while (xor != 0) {
        if (xor % 2 == 1)
            distance += 1;
        xor = xor >> 1;
        }
        return distance;
    }
}
```

方法三：布赖恩·克尼根算法

思路

方法二是逐位移动，逐位比较边缘位置是否为 1。寻找一种更快的方法找出等于 1 的位数。

> 是否可以像人类直观的计数比特为 1 的位数，跳过两个 1 之间的 0。例如：`10001000`。

上面例子中，遇到最右边的 1 后，如果可以跳过中间的 0，直接跳到下一个 1，效率会高很多。

这是布赖恩·克尼根位计数算法的基本思想。该算法使用特定比特位和算术运算移除等于 1 的最右比特位。

> 当我们在 `number` 和 `number-1` 上做 AND 位运算时，原数字 `number` 的最右边等于 1 的比特会被移除。

基于以上思路，通过 2 次迭代就可以知道 `10001000` 中 1 的位数，而不需要 8 次。

注意：该算法发布在 1988 年《C 语言编程第二版》的练习中（由 **Brian W. Kernighan** 和 Dennis M. Ritchie 编写），但是 Donald Knuth 在 2006 年 4 月 19 日指出，该方法第一次是由 Peter Wegner 在 1960 年的 CACM3 上出版。顺便说一句，可以在上述书籍中找到更多位操作的技巧。

复杂度分析

- 时间复杂度：$\mathcal{O}(1)$。
  - 与移位方法相似，由于整数的位数恒定，因此具有恒定的时间复杂度。
  - 但是该方法需要的迭代操作更少。
- 空间复杂度：$\mathcal{O}(1)$，与输入无关，使用恒定大小的空间。

方法 3：布赖恩 • 克尼根（Brian Kernighan）算法

```
class Solution {
    public int hammingDistance(int x, int y) {
        int xor = x ^ y;
        int distance = 0;
        while (xor != 0) {
        distance += 1;
        // remove the rightmost bit of '1'
        xor = xor & (xor - 1);
        }
        return distance;
    }
}
```

3. 整数转换（面试题 05.06）

面试题 05.06. 整数转换

难度 简单　9　收藏　分享　切换为英文　关注　反馈

整数转换。编写一个函数，确定需要改变几个位才能将整数A转成整数B。

示例1:

```
输入：A = 29 （或者0b11101），B = 15（或者0b01111）
输出：2
```

示例2:

```
输入：A = 1，B = 2
输出：2
```

提示:

1. A，B范围在[-2147483648, 2147483647]之间

https://leetcode-cn.com/problems/convert-integer-lcci/solution/mian-shi-ti-0506-zheng-shu-zhuan-huan-n-n-1-by-lia/

解题思路：
1，先通过异或得到相差几位的数字；
2，然后根据 n &= (n - 1) 能够将n最右侧为1的位变0 统计改数1的个数，即为需要的结果。

```
class Solution {
    public int convertInteger(int A, int B) {
        return hammingWeight(A ^ B);
    }

    private int hammingWeight(int n) {
        int sum = 0;
        while(n != 0){
            sum++;
            n &= (n - 1);
        }
        return sum;
    }
}
```

4. 数字范围按位与（201）

201. 数字范围按位与

难度 中等 159 收藏 分享 切换为英文 关注 反馈

给定范围 [m, n]，其中 0 <= m <= n <= 2147483647，返回此范围内所有数字的按位与（包含 m, n 两端点）。

示例 1:

输入: [5,7]
输出: 4

示例 2:

输入: [0,1]
输出: 0

https://leetcode-cn.com/problems/bitwise-and-of-numbers-range/solution/shu-zi-fan-wei-an-wei-yu-by-leetcode-solution/

更多硬核内容就在公众号「yes 的练级攻略」

方法二：Brian Kernighan 算法

思路与算法

还有一个位移相关的算法叫做「Brian Kernighan 算法」，它用于清除二进制串中最右边的 1。

Brian Kernighan 算法的关键在于我们每次对 $number$ 和 $number-1$ 之间进行按位与运算后，$number$ 中最右边的 1 会被抹去变成 0。

基于上述技巧，我们可以用它来计算两个二进制字符串的公共前缀。

其思想是，对于给定的范围 $[m,n]$（$m<n$），我们可以对数字 n 迭代地应用上述技巧，清除最右边的 1，直到它小于或等于 m，此时非公共前缀部分的 1 均被消去。因此最后我们返回 n 即可。

在上图所示的示例（$m=9,n=12$）中，公共前缀是 00001。在对数字 n 应用 Brian Kernighan 算法后，后面三位都将变为零，最后我们返回 n 即可。

更多硬核内容就在公众号「yes 的练级攻略」

复杂度分析

- 时间复杂度：$O(\log n)$。和位移方法类似，算法的时间复杂度取决于 m 和 n 二进制展开的位数。尽管和位移方法具有相同的渐近复杂度，但 Brian Kernighan 的算法需要的迭代次数会更少，因为它跳过了两个数字之间的所有零位。
- 空间复杂度：$O(1)$。我们只需要常数空间存放若干变量。

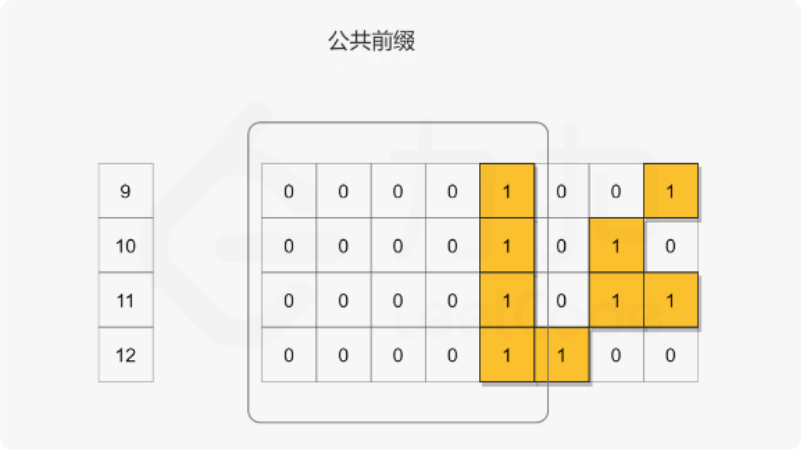
方法 1：布赖恩 • 克尼根（Brian Kernighan）算法

```
class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        while (m < n) {
            n &= (n-1);
        }
        return n;
    }
}
```

最直观的解决方案就是迭代范围内的每个数字，依次执行按位与运算，得到最终的结果，但此方法在 $[m, n]$ 范围较大的测试用例中会因超出时间限制而无法通过，因此我们需要另寻他路。

我们观察按位与运算的性质。对于一系列的位，例如 $[1, 1, 0, 1, 1]$，只要有一个零值的位，那么这一系列位的按位与运算结果都将为零。

回到本题，首先我们可以对范围内的每个数字用二进制的字符串表示，例如 $9 = 00001001_{(2)}$，然后我们将每个二进制字符串的位置对齐。



在上图的例子中，我们可以发现，对所有数字执行按位与运算的结果是所有对应二进制字符串的公共前缀再用零补上后面的剩余位。

那么这个规律是否正确呢？我们可以进行简单的证明。假设对于所有这些二进制串，前 i 位均相同，第 $i+1$ 位开始不同，由于 $[m, n]$ 连续，所以第 $i+1$ 位在 $[m, n]$ 的数字范围从小到大列举出来一定是前面全部是 0，后面全部是 1，在上图中对应 $[9, 11]$ 均为 0，$[12, 12]$ 均为 1。并且一定存在连续的两个数 x 和 $x+1$，满足 x 的第 $i+1$ 位为 0，后面全为 1，$x+1$ 的第 $i+1$ 位为 1，后面全为 0，对应上图中的例子即为 11 和 12。这种形如 $0111\ldots$ 和 $1000\ldots$ 的二进制串的按位与的结果一定为 $0000\ldots$，因此第 $i+1$ 位开始的剩余位均为 0，前 i 位由于均相同，因此按位与结果不变。最后的答案即为二进制字符串的公共前缀再用零补上后面的剩余位。

进一步来说，所有这些二进制字符串的公共前缀也即指定范围的起始和结束数字 m 和 n 的公共前缀（即在上面的示例中分别为 9 和 12）。

因此，最终我们可以将问题重新表述为：给定两个整数，我们要找到它们对应的二进制字符串的公共前缀。

更多硬核内容就在公众号「yes 的练级攻略」

方法一：位移

思路

鉴于上述问题的陈述，我们的目的是求出两个给定数字的二进制字符串的公共前缀，这里给出的第一个方法是采用位移操作。

我们的想法是将两个数字不断向右移动，直到数字相等，即数字被缩减为它们的公共前缀。然后，通过将公共前缀向左移动，将零添加到公共前缀的右边以获得最终结果。

算法

如上述所说，算法由两个步骤组成：

- 我们通过右移，将两个数字压缩为它们的公共前缀。在迭代过程中，我们计算执行的右移操作数。
- 将得到的公共前缀左移相同的操作数得到结果。

复杂度分析

- 时间复杂度：$O(\log n)$。算法的时间复杂度取决于 m 和 n 的二进制位数，由于 $m \leq n$，因此时间复杂度取决于 n 的二进制位数。
- 空间复杂度：$O(1)$。我们只需要常数空间存放若干变量。

方法 2：位移

```
class Solution {
    public int rangeBitwiseAnd(int m, int n) {
        int shift = 0;
        while (m < n) {
            m >>= 1;
            n >>= 1;
            ++shift;
        }
        return m << shift;
```

```
    }
}
```

5. 颠倒二进制位（190）

190. 颠倒二进制位

难度 简单 194 收藏 分享 切换为英文 关注 反馈

颠倒给定的 32 位无符号整数的二进制位。

示例 1：

输入： 00000010100101000001111010011100
输出： 00111001011110000010100101000000
解释： 输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596，
因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。

示例 2：

输入： 11111111111111111111111111111101
输出： 10111111111111111111111111111111
解释： 输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293，
因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111 。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 **示例 2** 中，输入表示有符号整数 -3，输出表示有符号整数 -1073741825。

进阶：
如果多次调用这个函数，你将如何优化你的算法？

https://leetcode-cn.com/problems/reverse-bits/solution/yi-dong-zui-hou-yi-wei-dao-di-yi-wei-4-xing-dai-ma/

解题思路

1. 取当前 n 的最后一位：`n & 1`
2. 将最后一位移动到对应位置，第一次为 31 位，第二次是 30 位，即：31、30、29... 1、0，写作代码 `bit << 31`；
3. 退出条件，二进制反转时，如果剩余位数全位 0，则可以不用再反转。

相似练习：

这题的移位操作与以下两题的一致，可同时练习：位 1 的个数、汉明距离。

方法 1：o(n)时间复杂度

```
public class Solution {
    public int reverseBits(int n) {
        int res = 0;
        for (int i = 31; n != 0; i--){
            res += (n & 1) << i;
            n >>>= 1;
        }
        return res;
    }
}
```

https://leetcode-cn.com/problems/reverse-bits/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-4-9/

更多硬核内容就在公众号「yes 的练级攻略」

另一种想法，参考 这里。

如果是两位数字怎么逆序呢？比如 `2 4`，我们只需要**交换**两个数字的位置，变成 `4 2`。

如果是四位数字怎么逆序呢？比如 `1 2 3 4`，同样的我们只需要交换两部分 `1 2` 和 `3 4` 的数字，变成 `3 4 1 2`，接下来只需要分别将两部分 `3 4` 和 `1 2` 分别逆序，两位数的逆序已经讨论过。

如果是八位数字怎么逆序呢？比如 `1 2 3 4 5 6 7 8`，同样的我们只需要交换两部分 `1 2 3 4` 和 `5 6 7 8` 的数字，变成 `5 6 7 8 1 2 3 4`，接下来只需要分别将两部分 `5 6 7 8` 和 `1 2 3 4` 分别逆序，四位数的逆序已经讨论过。

这道题也可以用这个思想去解决，`32` 位的数字左半部分何右半部分交换，得到两个 `16` 位的数字，然后两部分再交换，得到两个 `8` 位的数字...

在二进制中交换两部分，可以用一个技巧，举个例子，对于 `x = 1101` 交换两部分，我们只需要

`(1100) & x >>> 2 | (0011) & x <<< 2 = (0011)|(0100)= 0111`，然后就完成了 `11` 和 `01` 的交换。

```
public int reverseBits(int n) {
    n = ((n & 0xffff0000) >>> 16) | ((n & 0x0000ffff) << 16);
    n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8);
    n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4);
    n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2);
    n = ((n & 0xaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1);
    return n;
}
```

上边的写成 `16` 进制可能一下子不能理解，写成 `2` 进制就明白了。

比如 `n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2);` 也就是之前讨论的两位和两位交换。

我们需要分别和 `1100` 和 `0011` 进行与操作，写成 `16` 进制就是 `c` 和 `3`，因为我们要同时对 `32` 位的数字操作，每四个算一组，也就是八组，也就是 `8` 和 `c` 和 `8` 个 `3` 了。

另外需要注意的是一定要是逻辑右移，也就是三个大于号 `>>>`，不能去考虑符号位，具体就是涉及到 补码 的知识了。

方法 2：o(1)时间复杂度

```
public class Solution {
    public int reverseBits(int n) {
        n = ((n & 0xffff0000) >>> 16) | ((n & 0x0000ffff) << 16);
        n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8);
        n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4);
        n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2);
        n = ((n & 0xaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1);
        return n;
    }
}
```

6. 位 1 的个数（191）【二进制中 1 的个数（剑指 15）】

更多硬核内容就在公众号「yes 的练级攻略」

191. 位1的个数

难度 简单 192 收藏 分享 切换为英文 关注 反馈

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

示例 1：

```
输入：00000000000000000000000000001011
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
```

示例 2：

```
输入：00000000000000000000000010000000
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。
```

示例 3：

```
输入：11111111111111111111111111111101
输出：31
解释：输入的二进制串 11111111111111111111111111111101 中，共有 31 位为 '1'。
```

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 **示例 3** 中，输入表示有符号整数 `-3`。

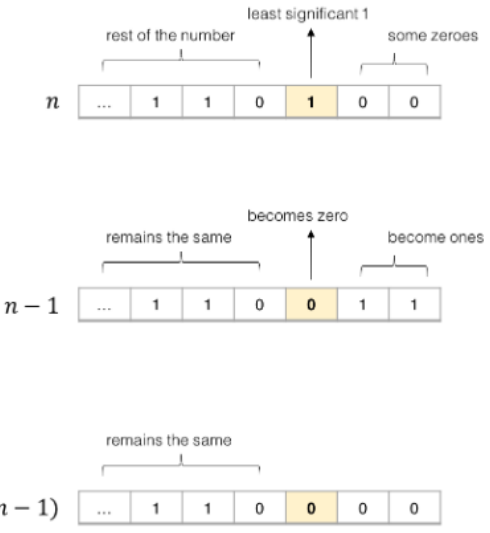
进阶：
如果多次调用这个函数，你将如何优化你的算法？

https://leetcode-cn.com/problems/number-of-1-bits/solution/wei-1de-ge-shu-by-leetcode/

算法

我们可以把前面的算法进行优化。我们不再检查数字的每一个位，而是不断把数字最后一个 1 反转，并把答案加一。当数字变成 0 的时候偶，我们就知道它没有 1 的位了，此时返回答案。

这里关键的想法是对于任意数字 n，将 n 和 $n-1$ 做与运算，会把最后一个 1 的位变成 0。为什么？考虑 n 和 $n-1$ 的二进制表示。



图片 1. 将 n 和 $n-1$ 做与运算会将最低位的 1 变成 0

在二进制表示中，数字 n 中最低位的 1 总是对应 $n-1$ 中的 0。因此，将 n 和 $n-1$ 与运算总是能把 n 中最低位的 1 变成 0，并保持其他位不变。

使用这个小技巧，代码变得非常简单。

复杂度分析

- 时间复杂度：$O(1)$。运行时间与 n 中位为 1 的有关。在最坏情况下，n 中所有位都是 1。对于 32 位整数，运行时间是 $O(1)$ 的。
- 空间复杂度：$O(1)$。没有使用额外空间。

```
public class Solution {
    public int hammingWeight(int n) {
        int sum = 0;
        while (n != 0) {
            sum++;
            n &= (n - 1);
        }
        return sum;
    }
}
```

7. 两数相除（29）

29. 两数相除

难度 中等 388 收藏 分享 切换为英文 关注 反馈

给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用乘法、除法和 mod 运算符。

返回被除数 dividend 除以除数 divisor 得到的商。

整数除法的结果应当截去（truncate）其小数部分，例如：truncate(8.345) = 8 以及 truncate(-2.7335) = -2

示例 1:

```
输入: dividend = 10, divisor = 3
输出: 3
解释: 10/3 = truncate(3.33333..) = truncate(3) = 3
```

示例 2:

```
输入: dividend = 7, divisor = -3
输出: -2
解释: 7/-3 = truncate(-2.33333..) = -2
```

提示:

- 被除数和除数均为 32 位有符号整数。
- 除数不为 0。
- 假设我们的环境只能存储 32 位有符号整数，其数值范围是 $[-2^{31}, 2^{31} - 1]$。本题中，如果除法结果溢出，则返回 $2^{31} - 1$。

https://leetcode-cn.com/problems/divide-two-integers/solution/zhi-xing-shi-jian-1msji-bai-100-by-luca-zhao/

改用负数计算

```
class Solution {
    public int divide(int dividend, int divisor) {
        boolean sign = (dividend > 0) ^ (divisor > 0);
        int result = 0;
```

```
        if(dividend>0) {
            dividend = -dividend;
        }
        if(divisor>0) divisor = -divisor;
        while(dividend <= divisor) {
            int temp_result = -1;
            int temp_divisor = divisor;
            while(dividend <= (temp_divisor << 1)) {
                if(temp_divisor <= (Integer.MIN_VALUE >> 1))break;
                temp_result = temp_result << 1;
                temp_divisor = temp_divisor << 1;
            }
            dividend = dividend - temp_divisor;
            result += temp_result;
        }
        if(!sign) {
            if(result <= Integer.MIN_VALUE) return Integer.MAX_VALUE;
            result = - result;
        }
        return result;
    }
}
```

8. 两整数之和（371）【不用加减乘除做加法（剑指 65）】【不用加号的加法（面试题 17.01）】

371. 两整数之和

难度 简单 290

不使用运算符 + 和 - ，计算两整数 a 、b 之和。

示例 1:

```
输入: a = 1, b = 2
输出: 3
```

示例 2:

```
输入: a = -2, b = 3
输出: 1
```

https://leetcode-cn.com/problems/sum-of-two-integers/solution/li-yong-wei-cao-zuo-shi-xian-liang-shu-qiu-he-by-p/

更多硬核内容就在公众号「yes 的练级攻略」

利用位操作实现加法

首先看十进制是如何做的： 5+7=12，三步走

- 第一步：相加各位的值，不算进位，得到2。
- 第二步：计算进位值，得到10. 如果这一步的进位值为0，那么第一步得到的值就是最终结果。
- 第三步：重复上述两步，只是相加的值变成上述两步的得到的结果2和10，得到12。

同样我们可以用三步走的方式计算二进制值相加： 5---101，7---111

- 第一步：相加各位的值，不算进位，得到010，**二进制每位相加就相当于各位做异或操作**，101^111。
- 第二步：计算进位值，得到1010，**相当于各位进行与操作得到101，再向左移一位得到1010**，(101&111)<<1。
- 第三步重复上述两步，各位相加 010^1010=1000，进位值为100=(010 & 1010)<<1。
- 继续重复上述两步：1000^100 = 1100，进位值为0，跳出循环，1100为最终结果。
- 结束条件：**进位为0，即a为最终的求和结果。**

```
class Solution {
    public int getSum(int a, int b) {
        while (b != 0){
            int tmp = a ^ b;
            b = (a & b) << 1;
            a = tmp;
        }
        return a;
        // 第二种方法：直接调用 Math 中的函数
        //return Math.addExact(a, b);
    }
}
```

9. 格雷编码（89）【移位和异或】

禁止商用
公众号：
yes的练级攻略

89. 格雷编码

难度 中等 199 收藏 分享 切换为英文 关注 反馈

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 *n*，打印其格雷编码序列。即使有多个不同答案，你也只需要返回其中一种。

格雷编码序列必须以 0 开头。

示例 1:

```
输入: 2
输出: [0,1,3,2]
解释:
00 - 0
01 - 1
11 - 3
10 - 2

对于给定的 n，其格雷编码序列并不唯一。
例如，[0,2,3,1] 也是一个有效的格雷编码序列。

00 - 0
10 - 2
11 - 3
01 - 1
```

示例 2:

```
输入: 0
输出: [0]
解释: 我们定义格雷编码序列必须以 0 开头。
     给定编码总位数为 n 的格雷编码序列，其长度为 2^n。当 n = 0 时，长度为 2^0 = 1。
     因此，当 n = 0 时，其格雷编码序列为 [0]。
```

https://leetcode-cn.com/problems/gray-code/solution/javazui-jian-dan-de-ban-ben-jian-dan-fen-xi-xiang-/

更多硬核内容就在公众号「yes 的练级攻略」

Java最简单的版本 简单分析，详细注解

简单分析：

1. 我们举几个例子来看一下。
2. 当n=1时，为[0,1]
   当n=2时，为[00,01,11,10]
   当n=3时，为[000,001,011,010,110,111,101,100]
   如果把前面的0都去掉，就变成：
   当n=1时，为[0,1]
   当n=2时，为[0,1,11,10]
   当n=3时，为[0,1,11,10,110,111,101,100]
   有没有发现什么规律呢？
3. 原来，第n层的数是建立在n-1层的基础上，第n层的前半部分跟n-1层的全部元素一样。后半部分，就是在前半部分的基础上最高位加个1.
4. 这样以来，怎么样在最高位加1呢。那就利用异或运算。将当前的数和它的二分之一（num>>1）做异或，这样，因为当前的数比它的二分之一在最高位上多了个1，所以，这就是要添加的下一个数。
5. OK，可以写代码了。

```
class Solution {
    public List<Integer> grayCode(int n) {
        List<Integer> res = new ArrayList<>();
        int num = 1 << n;
        for (int i = 0; i < num; i++){
            res.add(i >> 1 ^ i);
        }
        return res;
    }
}
```

10. 生命游戏（289）

公众号：yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

289. 生命游戏

难度 中等 241 收藏 分享 切换为英文 关注 反馈

根据 百度百科，生命游戏，简称为生命，是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机。

给定一个包含 m × n 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞都具有一个初始状态：1 即为活细胞（live），或 0 即为死细胞（dead）。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

1. 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；
2. 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；
3. 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；
4. 如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

根据当前状态，写一个函数来计算面板上所有细胞的下一个（一次更新后的）状态。下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。

示例：

输入：
```
[
  [0,1,0],
  [0,0,1],
  [1,1,1],
  [0,0,0]
]
```
输出：
```
[
  [0,0,0],
  [1,0,1],
  [0,1,1],
  [0,1,0]
]
```

进阶：

- 你可以使用原地算法解决本题吗？请注意，面板上所有格子需要同时被更新：你不能先更新某些格子，然后使用它们的更新后的值再更新其他格子。
- 本题中，我们使用二维数组来表示面板。原则上，面板是无限的，但当活细胞侵占了面板边界时会造成问题。你将如何解决这些问题？

https://leetcode-cn.com/problems/game-of-life/solution/si-lu-jian-dan-yi-dong-zhu-xing-jie-shi-by-jerry_n/

解题思路

分析：1 代表细胞活的， 0 代表细胞死的，那么这个位置就四种状态，用【下一个状态，当前状态】表示，最后需要用右移操作更新结果

- 状态 0： 00 ，死的，下一轮还是死的
- 状态 1： 01，活的，下一轮死了
- 状态 2： 10，死的，下一轮活了
- 状态 3： 11，活的，下一轮继续活着

进一步：下一轮活的可能有两种，也就是要把单元格变为 1

1. 这个活细胞周围八个位置有两个或三个活细胞，下一轮继续活，属于 11
2. 这个细胞本来死的，周围有三个活着的，下一轮复活了，属于 10

那遍历下每个格子看他周围细胞有多少个活细胞就行了，然后更改为状态，那么对于第一种可能，把 $board[i][j]$ 设置为 3 ，对于第二种可能状态设置为 2，设置个高位flag，遍历后面的格子，拿到与他相邻的格子中有多少个 $alive$ 的，和 1 与一下即可，最后我们把 $board[i][j]$ 右移 1位，更新结果

```
class Solution {
    int[] dx = {-1, 1, 0, 0, -1, -1, 1, 1};
    int[] dy = {0, 0, -1, 1, -1, 1, -1, 1};
    int[][] board;
    int m, n;
    //状态 0： 00 ，死的，下一轮还是死的
```

```
    //状态 1： 01，活的，下一轮死了
    //状态 2： 10，死的，下一轮活了
    //状态 3： 11，活的，下一轮继续活着
    public void gameOfLife(int[][] board) {
        this.board = board;
        if (board == null || board.length == 0 || board[0] == null || b
oard[0].length == 0) return;
        this.m = board.length;
        this.n = board[0].length;
        for (int i = 0; i < m; i++){
            for (int j = 0; j < n; j++){
                // 拿到当前位置周围活细胞数量
               int cnt = count(i, j);
                // 1. 活细胞周围八个位置有两个或三个活细胞，下一轮继续活
               if (board[i][j] == 1 && (cnt == 2 || cnt == 3)) board[i]
[j] = 3;
                // 2. 死细胞周围有三个活细胞，下一轮复活了
               if (board[i][j] == 0 && cnt == 3) board[i][j] = 2;
            }
        }
        // 更新结果
        for (int i = 0; i < m; i++){
            for (int j = 0; j < n; j++){
                board[i][j] >>= 1;
            }
        }
    }
    private int count(int x, int y){
        int cnt = 0;
        for (int k = 0; k < 8; k++){
            int nx = x + dx[k];
            int ny = y + dy[k];
            if (nx < 0 || nx >= m || ny < 0 || ny >= n) continue;
         // 如果这个位置为 0，代表当前轮是死的，不需要算进去
         // 如果这个位置为 1，代表当前轮是活得，需要算进去
         // 如果这个位置为 2，代表当前轮是死的（状态10，下一轮是活的），不需要
算进去
         // 如果这个位置为 3，代表是当前轮是活的（状态11，下一轮也是活的），需
要算进去
            cnt += (board[nx][ny] & 1);
        }
        return cnt;
    }
}
```

11. 二进制求和（67）

67. 二进制求和

难度 简单 460 收藏 分享 切换为英文 关注 反馈

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 **非空** 字符串且只包含数字 `1` 和 `0`。

示例 1:

```
输入: a = "11", b = "1"
输出: "100"
```

示例 2:

```
输入: a = "1010", b = "1011"
输出: "10101"
```

提示:

- 每个字符串仅由字符 `'0'` 或 `'1'` 组成。
- `1 <= a.length, b.length <= 10^4`
- 字符串如果不是 `"0"` ，就都不含前导零。

https://leetcode-cn.com/problems/add-binary/solution/er-jin-zhi-qiu-he-by-leetcode-solution/

方法一：模拟

思路和算法

我们可以借鉴「列竖式」的方法，末尾对齐，逐位相加。在十进制的计算中「逢十进一」，二进制中我们需要「逢二进一」。

具体的，我们可以取 $n = \max\{|a|, |b|\}$，循环 n 次，从最低位开始遍历。我们使用一个变量 carry 表示上一个位置的进位，初始值为 0。记当前位置对其的两个位为 a_i 和 b_i，则每一位的答案为 $(\text{carry} + a_i + b_i) \bmod 2$，下一位的进位为 $\lfloor \frac{\text{carry}+a_i+b_i}{2} \rfloor$。重复上述步骤，直到数字 a 和 b 的每一位计算完毕。最后如果 carry 的最高位不为 0，则将最高位添加到计算结果的末尾。

注意，为了让各个位置对齐，你可以先反转这个代表二进制数字的字符串，然后低下标对应低位，高下标对应高位。当然你也可以直接把 a 和 b 中短的那一个补 0 直到和长的那个一样长，然后从高位向低位遍历，对应位置的答案按照顺序存入答案字符串内，最终将答案串反转。这里的代码给出第一种的实现。

代码

复杂度分析

假设 $n = \max\{|a|, |b|\}$。

- 时间复杂度：$O(n)$，这里的时间复杂度来源于顺序遍历 a 和 b。
- 空间复杂度：$O(1)$，除去答案所占用的空间，这里使用了常数个临时变量。

方法 1：模拟，类似链表求和

```
class Solution {
```

更多硬核内容就在公众号「yes 的练级攻略」

```
    public String addBinary(String a, String b) {
        int i = a.length() - 1, j = b.length() - 1;
        int carry = 0;
        StringBuilder res = new StringBuilder();
        while (i >= 0 || j >= 0 || carry > 0){
            int x = (i >= 0) ? a.charAt(i) - '0' : 0;
            int y = (j >= 0) ? b.charAt(j) - '0' : 0;
            int sum = x + y + carry;
            int curr = sum % 2;
            carry = sum / 2;
            res.append(curr);
            i--;
            j--;
        }
        return res.reverse().toString();
    }
}
```

禁止商用
公众号：
yes的练级攻略

更多硬核内容就在公众号「yes 的练级攻略」

方法二：位运算

思路和算法

如果不允许使用加减乘除，则可以使用位运算替代上述运算中的一些加减乘除的操作。

如果不了解位运算，可以先了解位运算并尝试练习以下题目：

- 只出现一次的数字 II
- 只出现一次的数字 III
- 数组中两个数的最大异或值
- 重复的DNA序列
- 最大单词长度乘积

我们可以设计这样的算法来计算：

- 把 a 和 b 转换成整型数字 x 和 y，在接下来的过程中，x 保存结果，y 保存进位。
- 当进位不为 0 时
  - 计算当前 x 和 y 的无进位相加结果：`answer = x ^ y`
  - 计算当前 x 和 y 的进位：`carry = (x & y) << 1`
  - 完成本次循环，更新 `x = answer`，`y = carry`
- 返回 x 的二进制形式

为什么这个方法是可行的呢？在第一轮计算中，`answer` 的最后一位是 x 和 y 相加之后的结果，`carry` 的倒数第二位是 x 和 y 最后一位相加的进位。接着每一轮中，由于 `carry` 是由 x 和 y 按位与并且左移得到的，那么最后会补零，所以在下面计算的过程中后面的数位不受影响，而每一轮都可以得到一个低 i 位的答案和它向低 $i+1$ 位的进位，也就模拟了加法的过程。

复杂度分析

- 时间复杂度：$O(|a|+|b|+X\cdot\max(|a|+|b|))$，字符串转化成数字需要的时间代价为 $O(|a|+|b|)$，计算的时间代价为 $O(\max\{|a|,|b|\})$，X 为位运算所需的时间，因为这里用到了高精度计算，所以位运算的时间不一定为 $O(1)$。
- 空间复杂度：这里使用了 x 和 y 来保存 a 和 b 的整数形式，如果用 Python 实现，这里用到了 Python 的高精度功能，实际的空间代价是 $O(|a|+|b|)$。

方法 2：位运算

```
class Solution {
    public String addBinary(String a, String b) {
        int index1 = a.length() - 1;
        int index2 = b.length() - 1;
        StringBuilder builder = new StringBuilder();
        int temp = 0;
        int a1 ;
        int b1 ;
        while (index1 >= 0 || index2 >= 0){
            //a 上的字符
            a1 = index1 >= 0 ? a.charAt(index1--) - '0' : 0;
            //b 上的字符
            b1 = index2 >= 0 ? b.charAt(index2--) - '0' : 0;
            //a1 ^ b1 ^ temp 求出当前位置的结果，逢 2 进 1
            builder.append(a1 ^ b1 ^ temp);
```

```
            //进位
            temp = (a1 + b1 + temp) >> 1;
        }
        if(temp > 0){
            builder.append(temp);
        }
        return builder.reverse().toString();
    }
}
```

12. 配对交换（面试题 05.07）

面试题 05.07. 配对交换

难度 简单 27 收藏 分享 切换为英文 关注 反馈

配对交换。编写程序，交换某个整数的奇数位和偶数位，尽量使用较少的指令（也就是说，位0与位1交换，位2与位3交换，以此类推）。

示例1:

```
输入：num = 2（或者0b10）
输出 1 （或者 0b01）
```

示例2:

```
输入：num = 3
输出：3
```

提示:

1. num 的范围在[0, 2^30 - 1]之间，不会发生整数溢出。

https://leetcode-cn.com/problems/exchange-lcci/solution/wei-yun-suan-jie-jue-by-releng-xing/

思路的话就是分别取出奇数位和偶数位，移动后做或运算。
题目规定 num 是int范围的数
0x55555555 = 0b0101_0101_0101_0101_0101_0101_0101_0101
0xaaaaaaaa = 0b1010_1010_1010_1010_1010_1010_1010_1010

用这两个数做与运算，就可以把奇数位和偶数位取出来，
然后位左移奇数位，右移偶数位，
再把 奇数位和偶数位做或运算。

```
class Solution {
    public int exchangeBits(int num) {
        //奇数
        int odd = num & 0x55555555;
        //偶数
        int even = num & 0xaaaaaaaa;
        odd = odd << 1;
        even = even >>> 1;
        return odd | even;
    }
}
```

13. 插入（面试题 05.01）

面试题 05.01. 插入

难度 简单 15 收藏 分享 切换为英文 关注 反馈

插入。给定两个32位的整数 N 与 M，以及表示比特位置的 i 与 j。编写一种方法，将 M 插入 N，使得M从N的第j位开始，到第 i 位结束。假定从 j 位到 i 位足以容纳 M，也即若M = 10 011，那么j和i之间至少可容纳5个位。例如，不可能出现j = 3和i = 2的情况，因为第3位和第2位之间放不下M。

示例1:

```
输入：N = 1024(10000000000), M = 19(10011), i = 2, j = 6
输出：N = 1100(10001001100)
```

示例2:

```
输入： N = 0, M = 31(11111), i = 0, j = 4
输出：N = 31(11111)
```

https://leetcode-cn.com/problems/insert-into-bits-lcci/solution/shuang-bai-zhi-xing-jie-guo-ji-xing-dai-ma-jian-da/

解题思路

首先结合需要插入的起始位i和终止位j，构造辅助二进制数n，例如i=2,j=6;那么n=1111 ... 1111 0000 0011;
然后N=N&n；目的是使N的第i—j位全部为0，便于下一步的插入。
接着把M进行左移i位操作。
最后把M和N进行或运算，得出结果。

```
class Solution {
    public int insertBits(int N, int M, int i, int j) {
        int n = ~((-1 >>> (31 - j)) & (-1 << i));
        N = N & n;
        M = M << i;
        N = M | N;
        return N;
    }
}
```

14. 绘制直线（面试题 05.08）

更多硬核内容就在公众号「yes 的练级攻略」

面试题 05.08. 绘制直线

难度 中等 4 收藏 分享 切换为英文 关注 反馈

绘制直线。有个单色屏幕存储在一个一维数组中，使得32个连续像素可以存放在一个 int 里。屏幕宽度为 `w`，且 `w` 可被32整除（即一个 int 不会分布在两行上），屏幕高度可由数组长度及屏幕宽度推算得出。请实现一个函数，绘制从点 (x1, y) 到点 (x2, y) 的水平线。

给出数组的长度 `length`，宽度 `w`（以比特为单位）、直线开始位置 `x1`（比特为单位）、直线结束位置 `x2`（比特为单位）、直线所在行数 `y`。返回绘制过后的数组。

示例1:

```
输入：length = 1, w = 32, x1 = 30, x2 = 31, y = 0
输出：[3]
说明：在第0行的第30位到第31为画一条直线，屏幕表示为[0b00000000000000000000000000000011]
```

示例2:

```
输入：length = 3, w = 96, x1 = 0, x2 = 95, y = 0
输出：[-1, -1, -1]
```

https://leetcode-cn.com/problems/draw-line-lcci/solution/javawei-yun-suan-by-loser-11/

审题

首先要审题，说实话这题目乍一看我并没有看懂，还是画个图容易理解。

00000000 00000000 00000000 00000000，00000000 00000000 00000000 00000000，.....00000000 00000000 00000000 00000000
00000000 01111111 11111111 11111111，11111111 11111111 11000000 00000000，.....00000000 00000000 00000000 00000000
.....
00000000 00000000 00000000 00000000，00000000 00000000 00000000 00000000，.....00000000 00000000 00000000 00000000

length：表示int的总数，一个int32位，换句话说就是总共有length*32位二进制数
w: 表示一行二进制位数，题目说了w能被32整除，也就是一行有w/32个int
y: 表示在哪一行画直线，像我上面的图就是行1（从0开始）
x1：表示从行y开始下标为x1的二进制开始画，上图就是9（也就是第10个）
x2：表示画到行y下标为x2的二进制，上图就是49（也就是第50个）

yes的练级攻略

解题思路

每一位二进制表示一个像素，默认都为0，画线也就是从开始位置到结束位置将0变为1。画线的位置可能是跨多个int的，那么我们先将相关的int全部置为-1，为什么是-1，因为-1的补码为32位1，也就是将相关的int由32位全0变为全1，如下图（接上图的那个例子）。

00000000 00000000 00000000 00000000，00000000 00000000 00000000 00000000，.....00000000 00000000 00000000 00000000
11111111 11111111 11111111 11111111，11111111 11111111 11111111 11111111，.....00000000 00000000 00000000 00000000
.....
00000000 00000000 00000000 00000000，00000000 00000000 00000000 00000000，.....00000000 00000000 00000000 00000000

那么首先的问题就是怎么求出哪些int是相关的，由于给出了行数y，x1和x2，行y下标x1对应的二进制从第一行第一个二进制开始数应该为y*w+x1个，除以32就是对应的int，x2也是同理。将相关的int全部置为1后，并不是要求的结果，因为我们将最前面和最后面的也置为1了。那接下来的问题就是将多余的1置为0。

下面就是位运算的关键了。首先回顾一下java位运算，>>表示有符号数的右移，如果为负数（高位为1），高位补1，如果为正数（高位为0），高位补0；<<表示有符号数的左移，直接低位补0；>>>表示不管有符号数还是无符号数的右移，高位直接补0。至于为什么没有<<<，因为左移低位只可能补0，那不就跟<<一样了。

接着看代码，我们现在将前面的0也置为1了，要恢复为0，将对应的int>>>x1%32就行。最后就要将最后面的1变为0，也就是对应int的低32-x2%32置为0，这好像是一个很常见的位运算，我竟然一时间没看出来，只要与一个数作与运算就行，这个数前x2%32全为1，后面全为0，这样与下来低位全为0，高位不变。

```
  11111111 11111111 11111111 11111111
& 11111111 11111111 11000000 00000000
-----------------------------------
  11111111 11111111 11000000 00000000
```

那这个要与的数怎么得到呢，只要将Integer.MIN_VALUE，也就是1000...000(31个0)左移x2%32位，这样高x2%32全为1，低位全为0。注意移位运算的优先级高于&运算。

```
class Solution {
    public int[] drawLine(int length, int w, int x1, int x2, int y) {
        int[] ans=new int[length];
        int low=(y*w+x1)/32;
        int high=(y*w+x2)/32;
        for(int i=low;i<=high;i++){
            ans[i]=-1;
        }
        ans[low]=ans[low]>>>x1%32;
        ans[high]=ans[high]&Integer.MIN_VALUE>> x2 % 32;
        return ans;

    }
}
```