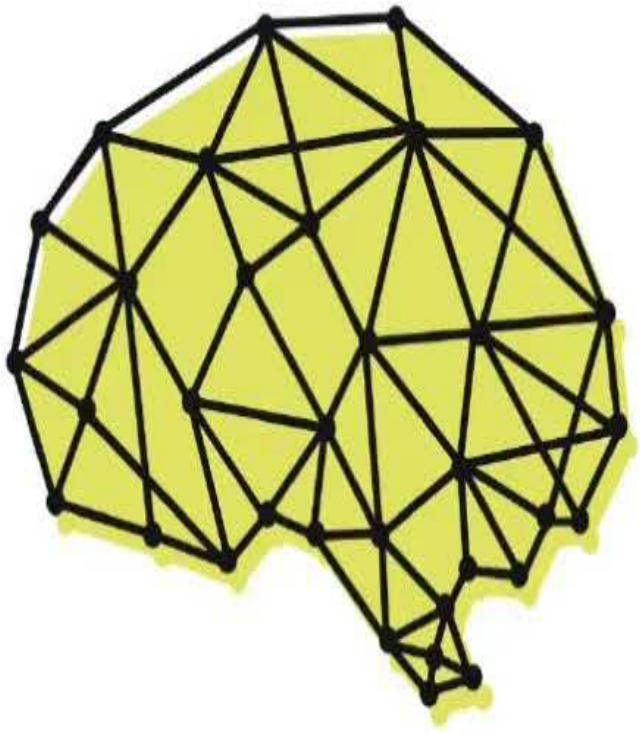


力扣加加

努力做西湖区最好的算法题解

目录

Introduction 1.1

第一章 - 算法专题 1.2

- 数据结构 1.2.1
- 链表专题 1.2.2
- 树专题 1.2.3
- 堆专题（上） 1.2.4
- 堆专题（下） 1.2.5
- 二分专题（上） 1.2.6
- 二分专题（下） 1.2.7
- 动态规划(重置版) 1.2.8
- 二叉树的遍历 1.2.9
- 哈夫曼编码和游程编码 1.2.10
- 布隆过滤器 1.2.11
- 前缀树 1.2.12
- 回溯 1.2.13
- 滑动窗口（思路 + 模板） 1.2.14
- 位运算 1.2.15
- 小岛问题 1.2.16
- 最大公约数 1.2.17
- 并查集 1.2.18
- 平衡二叉树专题 1.2.19
- 蓄水池抽样 1.2.20
- 单调栈 1.2.21

第二章 - 91 天学算法 1.3

- 91 天学算法第三期视频会议总结 1.3.1
- 第一期讲义-二分法 1.3.2
- 第一期讲义-双指针 1.3.3
- 第三期正在火热进行中 1.3.4

第三章 - 精选题解 1.4

- 字典序列删除 1.4.1
- 西法的刷题秘籍】一次搞定前缀和 1.4.2
- 字节跳动的算法面试题是什么难度? 1.4.3

字节跳动的算法面试题是什么难度？（第二弹） 1.4.4
《我是你的妈妈呀》 * 第一期 1.4.5
一文带你看懂二叉树的序列化 1.4.6
穿上衣服我就不认识你了？来聊聊最长上升子序列 1.4.7
你的衣服我扒了 * 《最长公共子序列》 1.4.8
一文看懂《最大子序列和问题》 1.4.9
第四章 - 高频考题（简单） 1.5
面试题 17.12. BiNode 1.5.1
0001. 两数之和 1.5.2
0020. 有效的括号 1.5.3
0021. 合并两个有序链表 1.5.4
0026. 删除排序数组中的重复项 1.5.5
0053. 最大子序和 1.5.6
0160. 相交链表 1.5.7
0066. 加一 1.5.8
0088. 合并两个有序数组 1.5.9
0101. 对称二叉树 1.5.10
0104. 二叉树的最大深度 1.5.11
0108. 将有序数组转换为二叉搜索树 1.5.12
0121. 买卖股票的最佳时机 1.5.13
0122. 买卖股票的最佳时机 II 1.5.14
0125. 验证回文串 1.5.15
0136. 只出现一次的数字 1.5.16
0155. 最小栈 1.5.17
0167. 两数之和 II 输入有序数组 1.5.18
0169. 多数元素 1.5.19
0172. 阶乘后的零 1.5.20
0190. 颠倒二进制位 1.5.21
0191. 位 1 的个数 1.5.22
0198. 打家劫舍 1.5.23
0203. 移除链表元素 1.5.24
0206. 反转链表 1.5.25
0219. 存在重复元素 II 1.5.26
0226. 翻转二叉树 1.5.27
0232. 用栈实现队列 1.5.28

0263. 丑数 1.5.29
0283. 移动零 1.5.30
0342. 4 的幂 1.5.31
0349. 两个数组的交集 1.5.32
0371. 两整数之和 1.5.33
401. 二进制手表 1.5.34
0437. 路径总和 III 1.5.35
0455. 分发饼干 1.5.36
0504. 七进制数 1.5.37
0575. 分糖果 1.5.38
0665. 非递减数列 1.5.39
821. 字符的最短距离 1.5.40
0874. 模拟行走机器人 1.5.41
1128. 等价多米诺骨牌对的数量 1.5.42
1260. 二维网格迁移 1.5.43
1332. 删除回文子序列 1.5.44
第五章 - 高频考题（中等） 1.6
面试题 17.09. 第 k 个数 1.6.1
面试题 17.23. 最大黑方阵 1.6.2
Increasing Digits 1.6.3
Longest Contiguously Strictly Increasing Sublist After Deletion 1.6.4
Consecutive Wins 1.6.5
Number of Substrings with Single Character Difference 1.6.6
Bus Fare 1.6.7
Minimum Dropping Path Sum 1.6.8
0002. 两数相加 1.6.9
0003. 无重复字符的最长子串 1.6.10
0005. 最长回文子串 1.6.11
0011. 盛最多水的容器 1.6.12
0015. 三数之和 1.6.13
0017. 电话号码的字母组合 1.6.14
0019. 删除链表的倒数第 N 个节点 1.6.15
0022. 括号生成 1.6.16
0024. 两两交换链表中的节点 1.6.17
0029. 两数相除 1.6.18

0031. 下一个排列	1.6.19
0033. 搜索旋转排序数组	1.6.20
0039. 组合总和	1.6.21
0040. 组合总和 II	1.6.22
0046. 全排列	1.6.23
0047. 全排列 II	1.6.24
0048. 旋转图像	1.6.25
0049. 字母异位词分组	1.6.26
0050. Pow(x, n)	1.6.27
0055. 跳跃游戏	1.6.28
0056. 合并区间	1.6.29
0060. 第 k 个排列	1.6.30
0061. 旋转链表	1.6.31
0062. 不同路径	1.6.32
0073. 矩阵置零	1.6.33
0075. 颜色分类	1.6.34
0078. 子集	1.6.35
0079. 单词搜索	1.6.36
0080. 删除排序数组中的重复项 II	1.6.37
0086. 分隔链表	1.6.38
0090. 子集 II	1.6.39
0091. 解码方法	1.6.40
0092. 反转链表 II	1.6.41
0094. 二叉树的中序遍历	1.6.42
0095. 不同的二叉搜索树 II	1.6.43
0096. 不同的二叉搜索树	1.6.44
0098. 验证二叉搜索树	1.6.45
0102. 二叉树的层序遍历	1.6.46
0103. 二叉树的锯齿形层次遍历	1.6.47
0113. 路径总和 II	1.6.48
0129. 求根到叶子节点数字之和	1.6.49
0130. 被围绕的区域	1.6.50
0131. 分割回文串	1.6.51
0139. 单词拆分	1.6.52
0144. 二叉树的前序遍历	1.6.53

0147. 对链表进行插入排序	1.6.54
0150. 逆波兰表达式求值	1.6.55
0152. 乘积最大子数组	1.6.56
0153. 寻找旋转排序数组中的最小值	1.6.57
0199. 二叉树的右视图	1.6.58
0200. 岛屿数量	1.6.59
0201. 数字范围按位与	1.6.60
0208. 实现 Trie (前缀树)	1.6.61
0209. 长度最小的子数组	1.6.62
0211. 添加与搜索单词 * 数据结构设计	1.6.63
0215. 数组中的第 K 个最大元素	1.6.64
0220. 存在重复元素 III	1.6.65
0221. 最大正方形	1.6.66
0227. 基本计算器 II	1.6.67
0229. 求众数 II	1.6.68
0230. 二叉搜索树中第 K 小的元素	1.6.69
0236. 二叉树的最近公共祖先	1.6.70
0238. 除自身以外数组的乘积	1.6.71
0240. 搜索二维矩阵 II	1.6.72
0279. 完全平方数	1.6.73
0309. 最佳买卖股票时机含冷冻期	1.6.74
0322. 零钱兑换	1.6.75
0328. 奇偶链表	1.6.76
0331. 验证二叉树的前序序列化	1.6.77
0334. 递增的三元子序列	1.6.78
0337. 打家劫舍 III	1.6.79
0343. 整数拆分	1.6.80
0365. 水壶问题	1.6.81
0378. 有序矩阵中第 K 小的元素	1.6.82
0380. 常数时间插入、删除和获取随机元素	1.6.83
0394. 字符串解码	1.6.84
0416. 分割等和子集	1.6.85
0424. 替换后的最长重复字符	1.6.86
0445. 两数相加 II	1.6.87
0454. 四数相加 II	1.6.88

0456. 132 模式	1.6.89
0464. 我能赢么	1.6.90
0494. 目标和	1.6.91
0516. 最长回文子序列	1.6.92
0513. 找树左下角的值	1.6.93
0518. 零钱兑换 II	1.6.94
0547. 朋友圈	1.6.95
0560. 和为 K 的子数组	1.6.96
0609. 在系统中查找重复文件	1.6.97
0611. 有效三角形的个数	1.6.98
0673. 最长递增子序列的个数	1.6.99
0686. 重复叠加字符串匹配	1.6.100
0714. 买卖股票的最佳时机含手续费	1.6.101
0718. 最长重复子数组	1.6.102
0754. 到达终点数字	1.6.103
0785. 判断二分图	1.6.104
0790. 多米诺和托米诺平铺	1.6.105
0799. 香槟塔	1.6.106
0801. 使序列递增的最小交换次数	1.6.107
0816. 模糊坐标	1.6.108
0820. 单词的压缩编码	1.6.109
0873. 最长的斐波那契子序列的长度	1.6.110
0875. 爱吃香蕉的珂珂	1.6.111
0877. 石子游戏	1.6.112
0886. 可能的二分法	1.6.113
0898. 子数组按位或操作	1.6.114
0900. RLE 迭代器	1.6.115
0911. 在线选举	1.6.116
0912. 排序数组	1.6.117
0932. 漂亮数组	1.6.118
0935. 骑士拨号器	1.6.119
0947. 移除最多的同行或同列石头	1.6.120
0959. 由斜杠划分区域	1.6.121
0978. 最长湍流子数组	1.6.122
0987. 二叉树的垂序遍历	1.6.123

1004. 最大连续 1 的个数 III 1.6.124
1011. 在 D 天内送达包裹的能力 1.6.125
1014. 最佳观光组合 1.6.126
1015. 可被 K 整除的最小整数 1.6.127
1019. 链表中的下一个更大节点 1.6.128
1020. 飞地的数量 1.6.129
1023. 驼峰式匹配 1.6.130
1031. 两个非重叠子数组的最大和 1.6.131
1043. 分隔数组以得到最大和 1.6.132
1104. 二叉树寻路 1.6.133
1131.绝对值表达式的最大值 1.6.134
1186. 删除一次得到子数组最大和 1.6.135
1218. 最长定差子序列 1.6.136
1227. 飞机座位分配概率 1.6.137
1261. 在受污染的二叉树中查找元素 1.6.138
1262. 可被三整除的最大和 1.6.139
1297. 子串的最大出现次数 1.6.140
1310. 子数组异或查询 1.6.141
1334. 阈值距离内邻居最少的城市 1.6.142
1371.每个元音包含偶数次的最长子字符串 1.6.143
1381. 设计一个支持增量操作的栈 1.6.144
1438. 绝对差不超过限制的最长连续子数组 1.6.145
1558. 得到目标数组的最少函数调用次数 1.6.146
1574. 删除最短的子数组使剩余数组有序 1.6.147
1631. 最小体力消耗路径 1.6.148
1658. 将 x 减到 0 的最小操作数 1.6.149
1697. 检查边长度限制的路径是否存在 1.6.150
1737. 满足三条件之一需改变的最少字符数 1.6.151
第六章 - 高频考题（困难） 1.7
LCP 20. 快速公交 1.7.1
LCP 21. 追逐游戏 1.7.2
Number Stream to Intervals 1.7.3
Triple-Inversion 1.7.4
Kth-Pair-Distance 1.7.5
Minimum-Light-Radius 1.7.6

Largest Equivalent Set of Pairs	1.7.7
0004. 寻找两个正序数组的中位数	1.7.8
0023. 合并 K 个升序链表	1.7.9
0025. K 个一组翻转链表	1.7.10
0030. 串联所有单词的子串	1.7.11
0032. 最长有效括号	1.7.12
0042. 接雨水	1.7.13
0052. N 皇后 II	1.7.14
0057. 插入区间	1.7.15
0084. 柱状图中最大的矩形	1.7.16
0085. 最大矩形	1.7.17
0087. 扰乱字符串	1.7.18
0124. 二叉树中的最大路径和	1.7.19
0128. 最长连续序列	1.7.20
0132. 分割回文串 II	1.7.21
0140. 单词拆分 II	1.7.22
0145. 二叉树的后序遍历	1.7.23
0146. LRU 缓存机制	1.7.24
0154. 寻找旋转排序数组中的最小值 II	1.7.25
0212. 单词搜索 II	1.7.26
0239. 滑动窗口最大值	1.7.27
0295. 数据流的中位数	1.7.28
0297. 二叉树的序列化与反序列化	1.7.29
0301. 删除无效的括号	1.7.30
0312. 戳气球	1.7.31
330. 按要求补齐数组	1.7.32
0335. 路径交叉	1.7.33
0460. LFU 缓存	1.7.34
0472. 连接词	1.7.35
0480. 滑动窗口中位数	1.7.36
0483. 最小好进制	1.7.37
0488. 祖玛游戏	1.7.38
0493. 翻转对	1.7.39
0679. 24 点游戏	1.7.40
0715. Range 模块	1.7.41

0768. 最多能完成排序的块 II 1.7.42
0805. 数组的均值分割 1.7.43
0839. 相似字符串组 1.7.44
0887. 鸡蛋掉落 1.7.45
0895. 最大频率栈 1.7.46
0975. 奇偶跳 1.7.47
0995. K 连续位的最小翻转次数 1.7.48
1032. 字符流 1.7.49
1168. 水资源分配优化 1.7.50
1178. 猜字谜 1.7.51
1203. 项目管理 1.7.52
1255. 得分最高的单词集合 1.7.53
1345. 跳跃游戏 IV 1.7.54
1449. 数位成本和为目标值的最大数字 1.7.55
1494. 并行课程 II 1.7.56
1521. 找到最接近目标值的函数值 1.7.57
1526. 形成目标数组的子数组最少增加次数 1.7.58
1649. 通过指令创建有序数组 1.7.59
1671. 得到山形数组的最少删除次数 1.7.60
1707. 与数组中元素的最大异或值 1.7.61
后序 1.8

西法的刷题秘籍

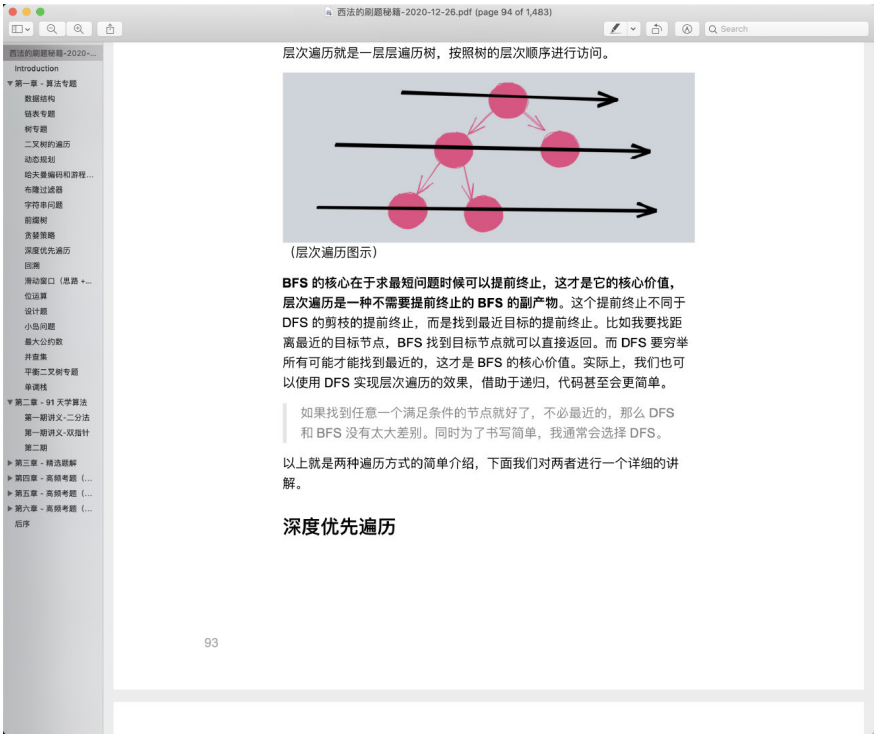
简体中文 | English

我们的 slogon 是： **只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。**

电子书

在线阅读

这是我将我的所有公开的算法资料整理的一个电子书，全部题目信息中文化，以前会有一些英文描述，感谢 @CYL 的中文整理。



限时免费下载！后期随时可能收费

有些动图，在做成电子书（比如 pdf）的时候自然就变没了，如果需要看动图的， 可以去我的公众号《力扣加加》或者我的 leetcode 题解仓库看。

> epub 还是有动图的

另外有些内容只在公众号发布，因此大家觉得内容不错的话，可以关注一下。如果再给 ➕ 个星标就更棒啦！

> 大家也可以用 Github 提供的 RSS 来订阅我的仓库更新。

刷题群

组队刷题活动，关注上面的公众号《力扣加加》回复 leetcode 即可获取进群方式，从此刷题不再孤单。

另外春招已经开始了。你是不是已经开始准备了呢？为了帮助大家获得更好的 offer，lucifer 开辟了「春招冲冲冲」栏目。

第一期我们的猎物是「虾皮」。来看看虾皮的算法题难度几何吧！

- 春招冲冲冲

图片加载不出来如何解决?

https://github.com/fe-lucifer/fanqiang

仓库介绍

leetcode 题解，记录自己的 leetcode 解题之路。

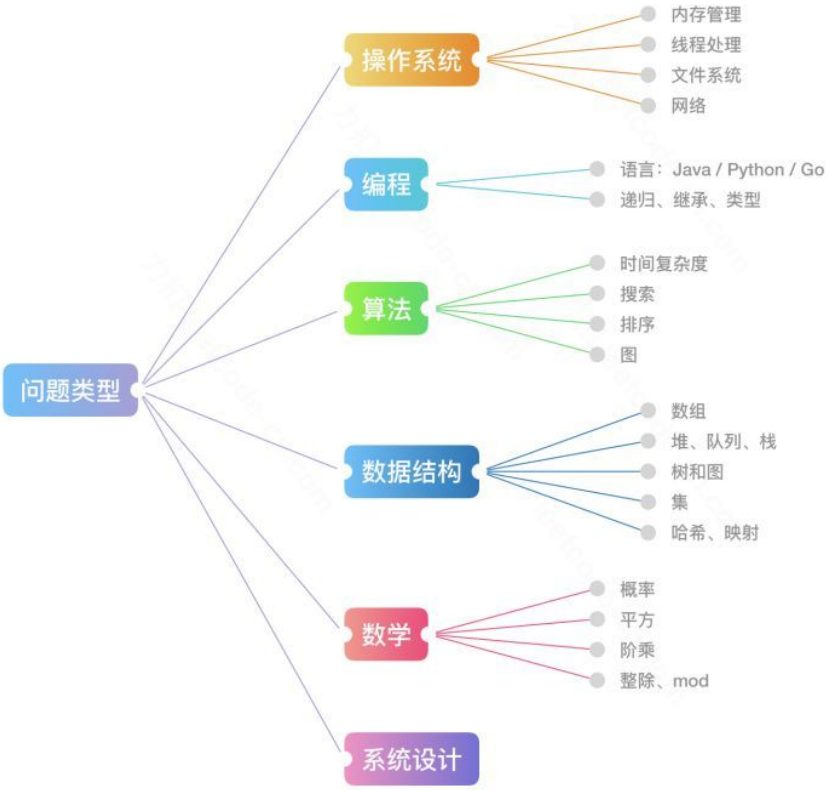
本仓库目前分为**五个**部分：

- 第一个部分是 leetcode 经典题目的解析，包括思路，关键点和具体的代码实现。
- 第二部分是对于数据结构与算法的总结

- 第三部分是 anki 卡片， 将 leetcode 题目按照一定的方式记录在 anki 中，方便大家记忆。
- 第四部分是每日一题，每日一题是在交流群（包括微信和 qq）里进行的一种活动，大家一起 解一道题，这样讨论问题更加集中，会得到更多的反馈。而且 这些题目可以被记录下来，日后会进行筛选添加到仓库的题解模块。
- 第五部分是计划， 这里会记录将来要加入到以上三个部分内容

仓库食用指南

- 对于最近添加的部分， 后面会有 🆕 标注
- 对于最近更新的部分， 后面会有 🖌️ 标注
- 这里有一张互联网公司面试中经常考察的问题类型总结的思维导图，我们可以结合图片中的信息分析一下。



（图片来自 leetcode)

其中算法，主要是以下几种：

- 基础技巧：分治、二分、贪心
- 排序算法：快速排序、归并排序、计数排序
- 搜索算法：回溯、递归、深度优先遍历，广度优先遍历，二叉搜索树等

- 图论：最短路径、最小生成树
- 动态规划：背包问题、最长子序列

数据结构，主要有如下几种：

- 数组与链表：单 / 双向链表
- 栈与队列
- 哈希表
- 堆：最大堆 / 最小堆
- 树与图：最近公共祖先、并查集
- 字符串：前缀树（字典树） / 后缀树

数据结构与算法的总结(25 篇)

- 数据结构总览
- 链表专题 🆕
- 树专题 🆕
- 堆专题（上）
- 堆专题（下） 🆕
- 二叉树的遍历
- 动态规划
- 回溯
- 哈夫曼编码和游程编码
- 布隆过滤器🖋
- 字符串问题
- 前缀树🖋
- 《日程安排》专题
- 《构造二叉树》专题
- 《贪婪策略》专题
- 深度优先遍历
- 滑动窗口（思路 + 模板）
- 位运算
- 设计题
- 小岛问题🖋
- 最大公约数
- 并查集
- 平衡二叉树专题
- 蓄水池抽样 🆕
- 单调栈 🆕

精选题解（9 篇）

- 字典序列删除
- 一次搞定前缀和

- 字节跳动的算法面试题是什么难度?
- 字节跳动的算法面试题是什么难度？（第二弹）
- 《我是你的妈妈呀》 - 第一期
- 一文带你看懂二叉树的序列化
- 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 你的衣服我扒了 - 《最长公共子序列》
- 一文看懂《最大子序列和问题》

插件

或许是一个可以改变你刷题效率的浏览器扩展插件。

插件地址：https://chrome.google.com/webstore/detail/leetcode-cheatsheet/fniccleejlofifaakbgppmbbcdfjonle?hl=en-US。

> 不能访问谷歌商店的朋友可以去我的公众号回复插件获取离线版。强烈推荐大家使用谷歌商店安装， 这样如果有更新可以自动安装，毕竟咱们的插件更新还是蛮快的。

怎么刷 LeetCode?

- 我是如何刷 LeetCode 的
- 算法小白如何高效、快速刷 leetcode?
- 刷题效率低？或许你就差这么一个插件
- 力扣刷题插件

《91 天学算法》限时活动

很多教育机构宣传的 7 天，一个月搞定算法面试的，我大概都了解了下，不怎么靠谱。学习算法这东西，还是要考积累，没有量变是不可能有质变的。还有的人选择看书，这是一个不错的选择。但是很多人选了过时的或者质量差的书，又或者不会去写书中给的练习题，导致效果很差。

基于这几个原因，我组织了一个 91 天刷题活动，通过一个相对比较长的时间（91 天）给出最新的学习路径，并强制大家打卡这种高强度练习来让大家**在 91 天后遇见更好的自己**。详细活动介绍可以点下方链接查看。另外往期的讲义也在下面了，大家可以看看合不合你的口味。

最后送给大家一句话： **坚持下去，会有突然间成长的一天**。

- 91 天学算法第三期视频会议总结
- 第一期讲义-二分法
- 第一期讲义-双指针
- 第三期正在火热进行中

leetcode 经典题目的解析（200 多道）

> 这里仅列举具有**代表性题目**，并不是全部题目

目前更新了 200 多道题解，加上专题涉及的题目，差不多有 **300 道**。

简单难度题目合集

这里的题目难度比较小， 大多是模拟题，或者是很容易看出解法的题目，另外简单题目一般使用暴力法都是可以解决的。 这个时候只有看一下数据范围，思考下你的算法复杂度就行了。

当然也不排除很多 hard 题目也可以暴力模拟，大家平时多注意数据范围即可。

以下是我列举的经典题目（带 91 字样的表示出自 **91 天学算法**活动）：

- 面试题 17.12. BiNode
- 0001. 两数之和 👍
- 0020. 有效的括号 👍
- 0021. 合并两个有序链表
- 0026. 删除排序数组中的重复项
- 0053. 最大子序和
- 0066. 加一 91
- 0088. 合并两个有序数组
- 0101. 对称二叉树
- 0104. 二叉树的最大深度
- 0108. 将有序数组转换为二叉搜索树
- 0121. 买卖股票的最佳时机
- 0122. 买卖股票的最佳时机 II
- 0125. 验证回文串 👍
- 0136. 只出现一次的数字
- 0155. 最小栈
- 0160. 相交链表 91
- 0167. 两数之和 II 输入有序数组
- 0169. 多数元素
- 0172. 阶乘后的零
- 0190. 颠倒二进制位
- 0191. 位 1 的个数 👍
- 0198. 打家劫舍
- 0203. 移除链表元素
- 0206. 反转链表
- 0219. 存在重复元素 II
- 0226. 翻转二叉树
- 0232. 用栈实现队列 91
- 0263. 丑数

- 0283. 移动零
- 0342. 4 的幂
- 0349. 两个数组的交集
- 0371. 两整数之和
- 401. 二进制手表
- 0437. 路径总和 III
- 0455. 分发饼干
- 0504. 七进制数 🆕
- 0575. 分糖果
- 0665. 非递减数列 🆕
- 821. 字符的最短距离 91
- 0874. 模拟行走机器人
- 1128. 等价多米诺骨牌对的数量 🆕
- 1260. 二维网格迁移
- 1332. 删除回文子序列

中等难度题目合集

中等题目是力扣比例最大的部分，因此这部分我的题解也是最多的。 大家不要太过追求难题，先把中等难度题目做熟了再说。

这部分的题目要不需要我们挖掘题目的内含信息， 将其抽象成简单题目。 要么是一些写起来比较麻烦的题目， 一些人编码能力不行就挂了。因此大家一定要自己做， 即使看了题解”会了“，也要自己码一遍。自己不亲自写一遍，里面的细节永远不知道。

以下是我列举的经典题目（带 91 字样的表示出自 **91 天学算法**活动）：

- 面试题 17.09. 第 k 个数
- 面试题 17.23. 最大黑方阵 🆕
- Increasing Digits 👍 🆕
- Longest Contiguously Strictly Increasing Sublist After Deletion 👍 🆕
- 0002. 两数相加 👍
- 0003. 无重复字符的最长子串
- 0005. 最长回文子串
- 0011. 盛最多水的容器
- 0015. 三数之和 👍
- 0017. 电话号码的字母组合
- 0019. 删除链表的倒数第 N 个节点
- 0022. 括号生成
- 0024. 两两交换链表中的节点
- 0029. 两数相除
- 0031. 下一个排列

- 0033. 搜索旋转排序数组
- 0039. 组合总和
- 0040. 组合总和 II
- 0046. 全排列 👍
- 0047. 全排列 II
- 0048. 旋转图像
- 0049. 字母异位词分组
- 0050. Pow(x, n)
- 0055. 跳跃游戏
- 0056. 合并区间
- 0060. 第 k 个排列
- 0061. 旋转链表 91
- 0062. 不同路径
- 0073. 矩阵置零
- 0075. 颜色分类
- 0078. 子集
- 0079. 单词搜索
- 0080. 删除排序数组中的重复项 II
- 0086. 分隔链表
- 0090. 子集 II
- 0091. 解码方法
- 0092. 反转链表 II
- 0094. 二叉树的中序遍历 👍
- 0095. 不同的二叉搜索树 II
- 0096. 不同的二叉搜索树
- 0098. 验证二叉搜索树
- 0102. 二叉树的层序遍历
- 0103. 二叉树的锯齿形层次遍历
- 0113. 路径总和 II
- 0129. 求根到叶子节点数字之和
- 0130. 被围绕的区域
- 0131. 分割回文串
- 0139. 单词拆分
- 0144. 二叉树的前序遍历
- 0147. 对链表进行插入排序 🆕
- 0150. 逆波兰表达式求值
- 0152. 乘积最大子数组
- 0199. 二叉树的右视图 👍
- 0200. 岛屿数量
- 0201. 数字范围按位与
- 0208. 实现 Trie (前缀树)
- 0209. 长度最小的子数组
- 0211. 添加与搜索单词 * 数据结构设计
- 0215. 数组中的第 K 个最大元素

- 0220. 存在重复元素 III 🆕
- 0221. 最大正方形
- 0227. 基本计算器 II 🆕
- 0229. 求众数 II
- 0230. 二叉搜索树中第 K 小的元素
- 0236. 二叉树的最近公共祖先
- 0238. 除自身以外数组的乘积
- 0240. 搜索二维矩阵 II
- 0279. 完全平方数
- 0309. 最佳买卖股票时机含冷冻期
- 0322. 零钱兑换 👍
- 0328. 奇偶链表
- 0331. 验证二叉树的前序序列化 🆕
- 0334. 递增的三元子序列
- 0337. 打家劫舍 III
- 0343. 整数拆分
- 0365. 水壶问题
- 0378. 有序矩阵中第 K 小的元素
- 0380. 常数时间插入、删除和获取随机元素
- 0394. 字符串解码 91
- 0416. 分割等和子集
- 0424. 替换后的最长重复字符 🆕
- 0445. 两数相加 II
- 0454. 四数相加 II
- 0464. 我能赢么
- 0494. 目标和
- 0516. 最长回文子序列
- 0513. 找树左下角的值 91
- 0518. 零钱兑换 II
- 0547. 朋友圈
- 0560. 和为 K 的子数组
- 0609. 在系统中查找重复文件
- 0611. 有效三角形的个数
- 0673. 最长递增子序列的个数
- 0686. 重复叠加字符串匹配
- 0718. 最长重复子数组
- 0714. 买卖股票的最佳时机含手续费 🆕
- 0754. 到达终点数字
- 0785. 判断二分图
- 0790. 多米诺和托米诺平铺
- 0799. 香槟塔 🆕
- 0801. 使序列递增的最小交换次数 🆕
- 0816. 模糊坐标
- 0820. 单词的压缩编码

- 0875. 爱吃香蕉的珂珂
- 0877. 石子游戏
- 0886. 可能的二分法
- 0898. 子数组按位或操作
- 0900. RLE 迭代器
- 0911. 在线选举
- 0912. 排序数组
- 0932. 漂亮数组 🆕
- 0935. 骑士拨号器
- 0947. 移除最多的同行或同列石头
- 0959. 由斜杠划分区域 🆕
- 0978. 最长湍流子数组
- 0987. 二叉树的垂序遍历 91
- 1004. 最大连续 1 的个数 III 🆕
- 1011. 在 D 天内送达包裹的能力
- 1014. 最佳观光组合
- 1015. 可被 K 整除的最小整数
- 1019. 链表中的下一个更大节点
- 1020. 飞地的数量
- 1023. 驼峰式匹配
- 1031. 两个非重叠子数组的最大和
- 1043. 分隔数组以得到最大和 🆕
- 1104. 二叉树寻路
- 1131.绝对值表达式的最大值
- 1186. 删除一次得到子数组最大和
- 1218. 最长定差子序列
- 1227. 飞机座位分配概率 👍
- 1261. 在受污染的二叉树中查找元素
- 1262. 可被三整除的最大和
- 1297. 子串的最大出现次数
- 1310. 子数组异或查询
- 1334. 阈值距离内邻居最少的城市
- 1371.每个元音包含偶数次的最长子字符串
- 1381. 设计一个支持增量操作的栈 91
- 1438. 绝对差不超过限制的最长连续子数组 🆕
- 1558. 得到目标数组的最少函数调用次数 🆕
- 1574. 删除最短的子数组使剩余数组有序 🆕
- 1631. 最小体力消耗路径
- 1658. 将 x 减到 0 的最小操作数 🆕
- 1697. 检查边长度限制的路径是否存在 🆕
- 1737. 满足三条件之一需改变的最少字符数 🆕

困难难度题目合集

困难难度题目从类型上说多是：

- 图
- 设计题
- 游戏场景题目
- 中等题目的 follow up

从解法上来说，多是：

- 图算法
- 动态规划
- 二分法
- DFS & BFS
- 状态压缩
- 剪枝

从逻辑上说， 要么就是非常难想到，要么就是非常难写代码。 这里我总结了几个技巧：

1. 看题目的数据范围， 看能否暴力模拟
2. 暴力枚举所有可能的算法往上套，比如图的题目。
3. 总结和记忆解题模板，减少解题压力

以下是我列举的经典题目（带 91 字样的表示出自 **91 天学算法**活动）：

- LCP 20. 快速公交 🆕
- Number Stream to Intervals 🆕
- Triple-Inversion 91
- Kth-Pair-Distance 91
- Minimum-Light-Radius 91

- 0004. 寻找两个正序数组的中位数 👍

- 0023. 合并 K 个升序链表
- 0025. K 个一组翻转链表 👍
- 0030. 串联所有单词的子串
- 0032. 最长有效括号
- 0042. 接雨水
- 0052. N 皇后 II
- 0057. 插入区间
- 0084. 柱状图中最大的矩形
- 0085. 最大矩形
- 0124. 二叉树中的最大路径和
- 0128. 最长连续序列
- 0132. 分割回文串 II 👍
- 0140. 单词拆分 II
- 0145. 二叉树的后序遍历
- 0146. LRU 缓存机制

- 0212. 单词搜索 II
- 0239. 滑动窗口最大值 👍
- 0295. 数据流的中位数
- 0297. 二叉树的序列化与反序列化 91
- 0301. 删除无效的括号
- 0312. 戳气球 👍
- 330. 按要求补齐数组
- 0335. 路径交叉
- 0460. LFU 缓存
- 0472. 连接词
- 0480. 滑动窗口中位数 🆕
- 0483. 最小好进制 🆕
- 0488. 祖玛游戏
- 0493. 翻转对
- 0679. 24 点游戏 🆕
- 0715. Range 模块 🆕
- 0768. 最多能完成排序的块 II 91
- 0805. 数组的均值分割
- 0839. 相似字符串组
- 0887. 鸡蛋掉落
- 0895. 最大频率栈
- 0975. 奇偶跳 🆕
- 0995. K 连续位的最小翻转次数 🆕
- 1032. 字符流
- 1168. 水资源分配优化
- 1178. 猜字谜 🆕
- 1203. 项目管理 🆕
- 1255. 得分最高的单词集合
- 1345. 跳跃游戏 IV
- 1449. 数位成本和为目标值的最大数字
- 1494. 并行课程 II 🆕
- 1521. 找到最接近目标值的函数值
- 1526. 形成目标数组的子数组最少增加次数
- 1649. 通过指令创建有序数组
- 1671. 得到山形数组的最少删除次数
- 1707. 与数组中元素的最大异或值

anki 卡片

Anki 主要分为两个部分：一部分是关键点到题目的映射，另一部分是题目到思路，关键点，代码的映射。

全部卡片都在 anki-card

使用方法：

anki - 文件 - 导入 - 下拉格式选择“打包的 anki 集合”，然后选中你下载好的文件，确定即可。

更多关于 anki 使用方法的请查看 anki 官网

目前已更新卡片一览（仅列举正面）：

- 二分法解决问题的关键点是什么，相关问题有哪些?
- 如何用栈的特点来简化操作， 涉及到的题目有哪些?
- 双指针问题的思路以及相关题目有哪些?
- 滑动窗口问题的思路以及相关题目有哪些?
- 回溯法解题的思路以及相关题目有哪些?
- 数论解决问题的关键点是什么，相关问题有哪些?
- 位运算解决问题的关键点是什么，相关问题有哪些?

> 已加入的题目有：#2 #3 #11

大事件

- 2019-07-10： 纪念项目 Star 突破 1W 的一个短文， 记录了项目的"兴起"之路，大家有兴趣可以看一下，如果对这个项目感兴趣，请**点击一下 Star**， 项目会**持续更新**，感谢大家的支持。
- 2019-10-08: 纪念 LeetCode 项目 Star 突破 2W，并且 Github 搜索“LeetCode”，排名第一。
- 2020-04-12: 项目突破三万 Star。
- 2020-04-14//leetcode-solution.cn/

- 2021-02-23: star 破四万

贡献

- 如果有想法和创意，请提 issue 或者进群提
- 如果想贡献增加题解或者翻译， 可以参考 贡献指南
  > 关于如何提交题解，我写了一份 指南

- 如果需要修改项目中图片，这里 存放了项目中绘制图的源代码， 大家可以用 draw.io 打开进行编辑。

鸣谢

感谢为这个项目作出贡献的所有 小伙伴

License

CC BY-NC-ND 4.0

算法专题

以下是一些我总结的类型题目，提前搞懂这些东西对之后的做题很有帮助，强烈建议先掌握。另外我的 91 天学算法也对专题进行了更细粒度的整理，具体参见91 天学算法

首先基础的数据结构大家是必须掌握的，其次就是暴力法。暴力法也是算法，只不过我们追求的肯定是性能更好的算法。因此了解暴力法的算法瓶颈以及各种数据结构的特点就很重要， 这样你就可以根据这些知识去一步步逼近最优解。

再之后就是必须掌握的算法。比如搜索算法是必须掌握的，搜索算法的范围很广，但是核心就是搜索的，不同的算法在于搜索的方式不同，典型的就是 BFS 和 DFS，当然二分法本质上也是一种搜索算法。

还有就是暴力优化法也是必须掌握的，和搜索一样，范围很广。 有剪枝， 空间换时间等。 其中空间换时间又有很多，比如哈希表， 前缀树等等。

围绕这个思想去学习， 就不会差太多，其他我就不多说，大家慢慢体会。

- 数据结构总览
- 链表专题
- 树专题
- 堆专题（上）
- 堆专题（下）
- 二分专题（上）
- 二分专题（下） 🆕
- 二叉树的遍历
- 动态规划
- 回溯
- 哈夫曼编码和游程编码
- 布隆过滤器🖋
- 前缀树🖋
- 《日程安排》专题
- 《构造二叉树》专题
- 滑动窗口（思路 + 模板）
- 位运算
- 小岛问题🖋
- 最大公约数
- 并查集
- 平衡二叉树专题
- 蓄水池抽样 🆕
- 单调栈 🆕

基础的数据结构（总览）

这篇文章不是讲解数据结构的文章，而是结合现实的场景帮助大家 `理解和复习` 数据结构与算法，如果你的数据结构基础很差，建议先去看一些基础教程，再转过来看。

本篇文章的定位是侧重于前端的，通过学习前端中实际场景的数据结构，从而加深大家对数据结构的理解和认识。

线性结构

数据结构我们可以从逻辑上分为线性结构和非线性结构。线性结构有数组，栈，链表等， 非线性结构有树，图等。

需要注意的是，线性和非线性不代表存储结构是线性的还是非线性的，这两者没有任何关系，它只是一种逻辑上的划分。比如我们可以用数组去存储二叉树。一般而言，有前驱和后继的就是线性数据结构。比如数组和链表。

数组

其实后面的数据结构很多都有数组的影子。数组是最简单的数据结构了，很多地方都用到它。 比如用一个数据列表存储一些用户的 id，就可以用数组进行存储。

我们之后要讲的栈和队列其实都可以看成是一种 `受限` 的数组，怎么个受限法呢？我们后面讨论。

接下来通过几个有趣的例子来加深大家对数组这种数据结构的理解。

React Hooks（非前端党慎入）

Hooks 的本质就是一个数组， 伪代码：

```
let hooks = null;

export function useHook() {
  hooks.push(hookData);
}

function reactsInternalRenderAComponentMethod(component) {
  hooks = [];
  component();
  let hooksForThisComponent = hooks;
  hooks = null;
}
```

那么为什么 hooks 要用数组？ 我们可以换个角度来解释，如果不用数组会怎么样?

```
function Form() {
  // 1. Use the name state variable
  const [name, setName] = useState("Mary");

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem("formData", name);
  });

  // 3. Use the surname state variable
  const [surname, setSurname] = useState("Poppins");

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = name + " " + surname;
  });

  // ...
}
```

基于数组的方式，Form 的 hooks 就是 [hook1, hook2, hook3, hook4]。

进而我们可以得出这样的关系， hook1 就是 [name, setName] 这一对，hook2 就是 persistForm 这个。

如果不用数组实现，比如对象，Form 的 hooks 就是

```
{
  'key1': hook1,
  'key2': hook2,
  'key3': hook3,
  'key4': hook4,
}
```

那么问题是 key1，key2，key3，key4 怎么取呢？这就是个问题了。更多关于 React hooks 的本质研究，请查看 React hooks: not magic, just arrays

不过使用数组也有一个问题， 那就是 React 将 `如何确保组件内部 hooks 保存的状态之间的对应关系` 这个工作交给了开发人员去保证，即你必须保证 HOOKS 的顺序严格一致，具体可以看 React 官网关于 Hooks Rule 部分。

队列

队列是一种**受限**的序列。受限在哪呢？受限就受限在它只能够操作队尾和队首，并且只能只能在队尾添加元素，在队首删除元素。而数组就没有这个限制。

队列作为一种最常见的数据结构同样有着非常广泛的应用， 比如消息队列

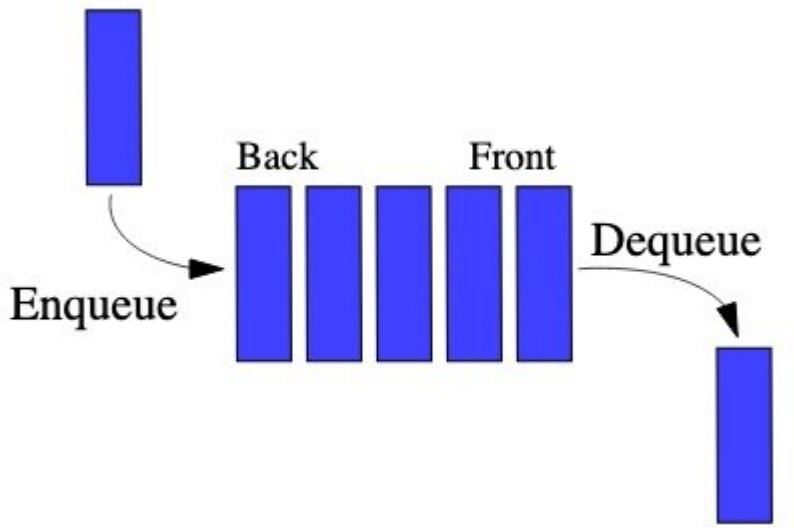
> "队列"这个名称，可类比为现实生活中排队（不插队的那种）

在计算机科学中，一个 队列 (queue) 是一种特殊类型的抽象数据类型或集合，集合中的实体按顺序保存。

队列基本操作有两种：

- 向队列的后端位置添加实体，称为入队
- 从队列的前端位置移除实体，称为出队。

队列中元素先进先出 FIFO (first in, first out) 的示意：



（图片来自 https://github.com/trekhleb/javascript-algorithms/blob/master/src/data-structures/queue/README.zh-CN.md）

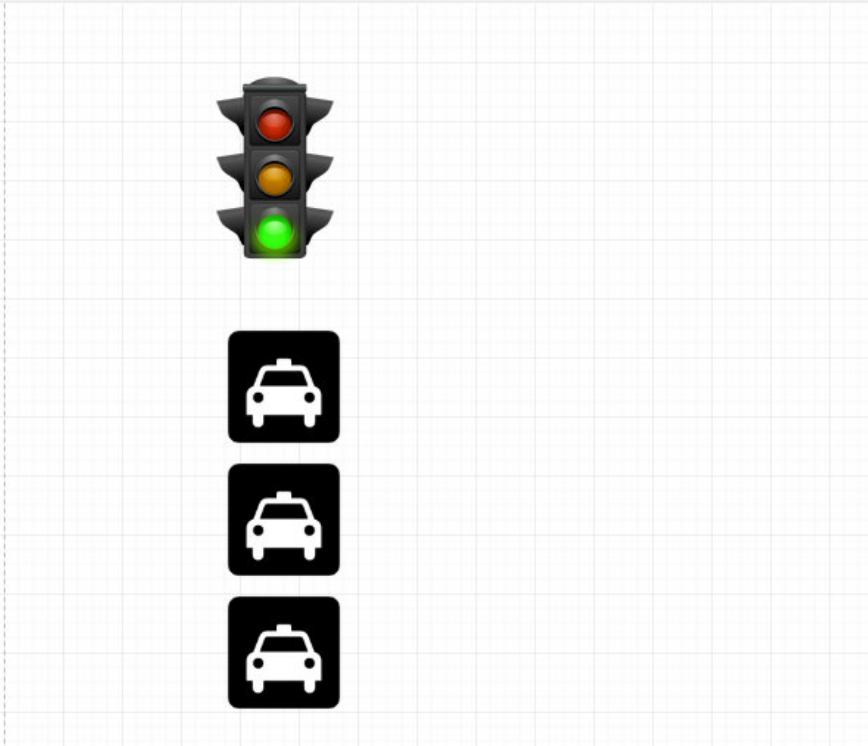
队列的实际使用

我们在做性能优化的时候，很多时候会提到的一点就是“HTTP 1.1 的队头阻塞问题”，具体来说就是 HTTP2 解决了 HTTP1.1 中的队头阻塞问题，但是为什么 HTTP1.1 有队头阻塞问题，HTTP2 究竟怎么解决的这个问题，很多人都不清楚。

其实 `队头阻塞` 是一个专有名词，不仅仅在 HTTP 有，交换器等其他地方也都涉及到了这个问题。实际上引起这个问题的根本原因是使用了 `队列` 这种数据结构。

协议规定， 对于同一个 tcp 连接，所有的 http1.0 请求放入队列中，只有前一个 `请求的响应` 收到了，才能发送下一个请求，这个时候就发生了阻塞，并且这个阻塞主要发生在客户端。

这就好像我们在等红绿灯，即使旁边绿灯亮了，你的这个车道是红灯，你还是不能走，还是要等着。



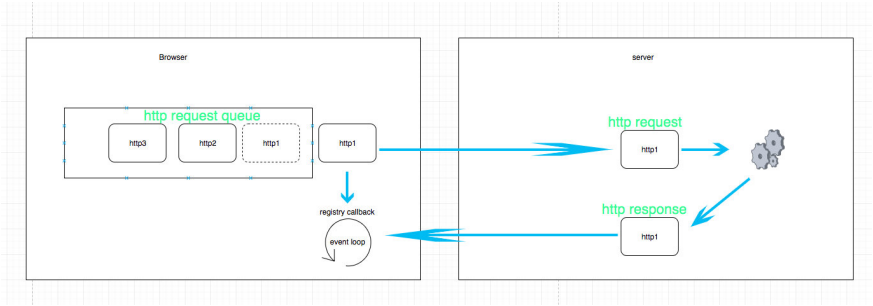
`HTTP/1.0` 和 `HTTP/1.1`：

在 `HTTP/1.0` 中每一次请求都需要建立一个 TCP 连接，请求结束后立即断开连接。

在 `HTTP/1.1` 中，每一个连接都默认是长连接 (persistent connection)。对于同一个 tcp 连接，允许一次发送多个 http1.1 请求，也就是说，不必等前一个响应收到，就可以发送下一个请求。这样就解决了 http1.0 的客户端的队头阻塞，而这也就是 `HTTP/1.1` 中 `管道（Pipeline）` 的概念了。

但是， `http1.1` 规定，服务器端的响应的发送要根据请求被接收的顺序排队 ，也就是说，先接收到的请求的响应也要先发送。这样造成的问题是，如果最先收到的请求的处理时间长的话，响应生成也慢，就会阻塞已经生成了的响应的发送，这也会造成队头阻塞。可见，http1.1 的队首阻塞是发生在服务器端。

如果用图来表示的话，过程大概是：



`HTTP/2` 和 `HTTP/1.1`：

为了解决 `HTTP/1.1` 中的服务端队首阻塞，`HTTP/2` 采用了 `二进制分帧` 和 `多路复用` 等方法。

帧是 `HTTP/2` 数据通信的最小单位。在 `HTTP/1.1` 中数据包是文本格式，而 `HTTP/2` 的数据包是二进制格式的，也就是二进制帧。

采用帧的传输方式可以将请求和响应的数据分割得更小，且二进制协议可以被高效解析。`HTTP/2` 中，同域名下所有通信都在单个连接上完成，该连接可以承载任意数量的双向数据流。每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装。

`多路复用` 用以替代原来的序列和拥塞机制。在 `HTTP/1.1` 中，并发多个请求需要多个 TCP 链接，且单个域名有 6-8 个 TCP 链接请求限制（这个限制是浏览器限制的，不同的浏览器也不一定一样）。在 `HTTP/2` 中，同一域名下的所有通信在单个链接完成，仅占用一个 TCP 链接，且在这一个链接上可以并行请求和响应，互不干扰。

> 此网站 可以直观感受 `HTTP/1.1` 和 `HTTP/2` 的性能对比。

栈

栈也是一种**受限**的序列，它受限就受限在只能够操作栈顶，不管入栈还是出栈，都是在栈顶操作。同样地，数组就没有这个限制。

在计算机科学中，一个栈 (stack) 是一种抽象数据类型，用作表示元素的集合，具有两种主要操作：

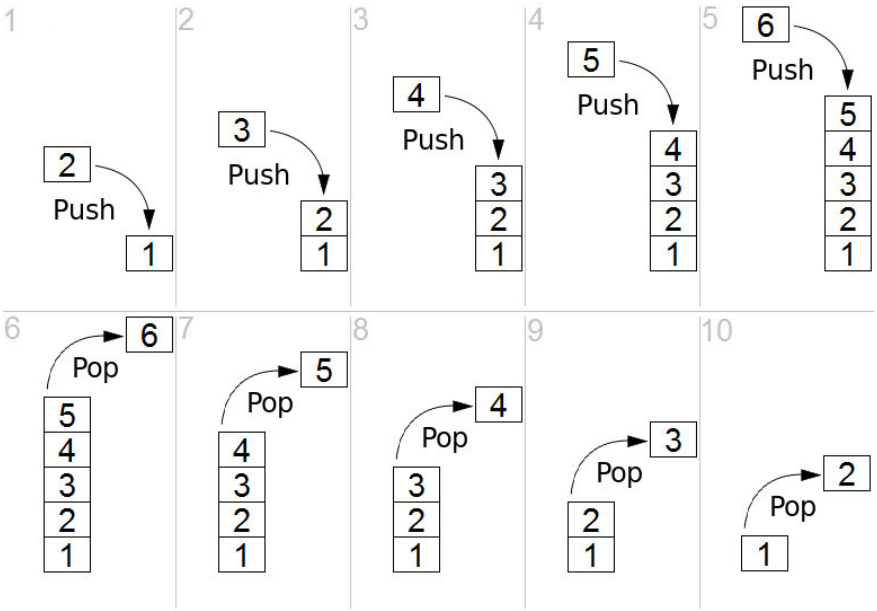
- push, 添加元素到栈的顶端（末尾）
- pop, 移除栈最顶端（末尾）的元素

以上两种操作可以简单概括为**后进先出 (LIFO = last in, first out)**。

此外，应有一个 peek 操作用于访问栈当前顶端（末尾）的元素。（只返回不弹出）

> "栈"这个名称，可类比于一组物体的堆叠（一摞书，一摞盘子之类的）。

栈的 push 和 pop 操作的示意：



（图片来自 https://github.com/trekhleb/javascript-algorithms/blob/master/src/data-structures/stack/README.zh-CN.md）

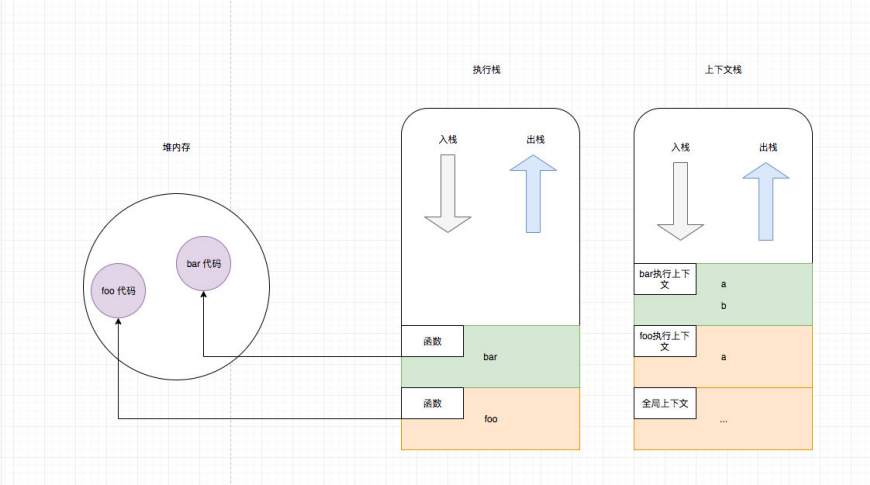
栈的应用（非前端慎入）

栈在很多地方都有着应用，比如大家熟悉的浏览器就有很多栈，其实浏览器的执行栈就是一个基本的栈结构，从数据结构上说，它就是一个栈。这也就解释了，我们用递归的解法和用循环+栈的解法本质上是差不多的。

比如如下 JS 代码：

```
function bar() {
  const a = 1;
  const b = 2;
  console.log(a, b);
}
function foo() {
  const a = 1;
  bar();
}

foo();
```

真正执行的时候，内部大概是这样的：



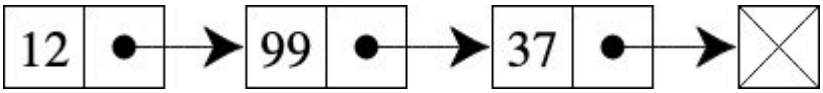
> 我画的图没有画出执行上下文中其他部分（this 和 scope 等），这部分是闭包的关键，而我这里不是讲闭包的，是为了讲解栈的。
>
> 社区中有很多“执行上下文中的 scope 指的是执行栈中父级声明的变量”说法，这是完全错误的，JS 是词法作用域，scope 指的是函数定义时候的父级，和执行没关系

栈常见的应用有进制转换，括号匹配，栈混洗，中缀表达式（用的很少），后缀表达式（逆波兰表达式）等。

合法的栈混洗操作也是一个经典的题目，这其实和合法的括号匹配表达式之间存在着一一对应的关系，也就是说 n 个元素的栈混洗有多少种，n 对括号的合法表达式就有多少种。感兴趣的可以查找相关资料。

链表

链表是一种最基本数据结构，熟练掌握链表的结构和常见操作是基础中的基础。

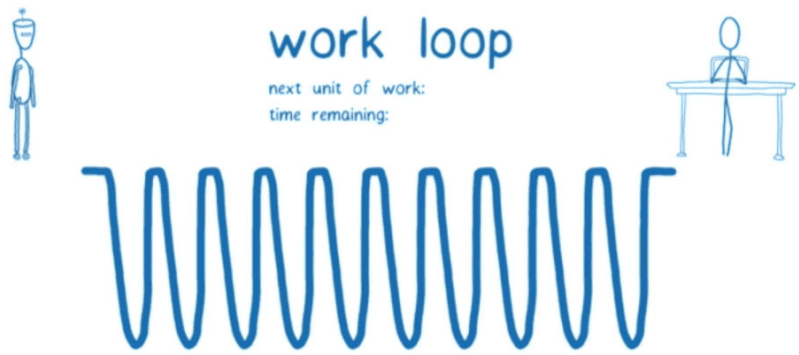


（图片来自：https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/linked-list/traversal）

React Fiber（非前端慎入）

很多人都说 fiber 是基于链表实现的，但是为什么要基于链表呢，可能很多人并没有答案，那么我觉得可以把这两个点（fiber 和链表）放到一起来讲下。

fiber 出现的目的其实是为了解决 react 在执行的时候是无法停下来的，需要一口气执行完的问题的。



> 图片来自 Lin Clark 在 ReactConf 2017 分享

上面已经指出了引入 fiber 之前的问题，就是 react 会阻止优先级高的代码（比如用户输入）执行。因此他们打算自己自建一个 `虚拟执行栈` 来解决这个问题，这个虚拟执行栈的底层实现就是链表。

Fiber 的基本原理是将协调过程分成小块，一次执行一块，然后将运算结果保存起来，并判断是否有时间继续执行下一块（react 自己实现了一个类似 requestIdleCallback 的功能）。如果有时间，则继续。 否则跳出，让浏览器主线程歇一会，执行别的优先级高的代码。

当协调过程完成（所有的小块都运算完毕）， 那么就会进入提交阶段，执行真正的进行副作用（side effect）操作，比如更新 DOM，这个过程是没有办法取消的，原因就是这部分有副作用。

问题的关键就是将协调的过程划分为一块块的，最后还可以合并到一起，有点像 Map / Reduce。

React 必须重新实现遍历树的算法，从依赖于 `内置堆栈的同步递归模型` ，变为 `具有链表和指针的异步模型` 。

> Andrew 是这么说的： 如果你只依赖于 [内置] 调用堆栈，它将继续工作直到堆栈为空。

如果我们可以随意中断调用堆栈并手动操作堆栈帧，那不是很好吗？ 这就是 React Fiber 的目的。 `Fiber 是堆栈的重新实现，专门用于 React 组件` 。 你可以将单个 Fiber 视为一个 `虚拟堆栈帧` 。

react fiber 大概是这样的：

```
let fiber = {
  tag: HOST_COMPONENT,
  type: "div",
  return: parentFiber,
  children: childFiber,
  sibling: childFiber,
  alternate: currentFiber,
  stateNode: document.createElement("div"),
  props: { children: [], className: "foo" },
  partialState: null,
  effectTag: PLACEMENT,
  effects: [],
};
```

从这里可以看出 fiber 本质上是个对象，使用 parent，child，sibling 属性去构建 fiber 树来表示组件的结构树， return, children, sibling 也都是一个 fiber，因此 fiber 看起来就是一个链表。

> 细心的朋友可能已经发现了， alternate 也是一个 fiber， 那么它是用来做什么的呢？ 它其实原理有点像 git， 可以用来执行 git revert ,git commit 等操作，这部分挺有意思，我会在我的《从零开发 git》中讲解

想要了解更多的朋友可以看 这个文章

如果可以翻墙， 可以看 英文原文

这篇文章 也是早期讲述 fiber 架构的优秀文章

我目前也在写关于《从零开发 react 系列教程》中关于 fiber 架构的部分，如果你对具体实现感兴趣，欢迎关注。

非线性结构

那么有了线性结构，我们为什么还需要非线性结构呢？ 答案是为了高效地兼顾静态操作和动态操作，**我们一般使用树去管理需要大量动态操作的数据**。大家可以对照各种数据结构的各种操作的复杂度来直观感受一下。

树

树的应用同样非常广泛，小到文件系统，大到因特网，组织架构等都可以表示为树结构，而在我们前端眼中比较熟悉的 DOM 树也是一种树结构，而 HTML 作为一种 DSL 去描述这种树结构的具体表现形式。如果你接触过 AST，那么 AST 也是一种树，XML 也是树结构。树的应用远比大多数人想象的要多得多。

树其实是一种特殊的 `图` ，是一种无环连通图，是一种极大无环图，也是一种极小连通图。

从另一个角度看，树是一种递归的数据结构。而且树的不同表示方法，比如不常用的 `长子 + 兄弟` 法，对于 你理解树这种数据结构有着很大用处， 说是一种对树的本质的更深刻的理解也不为过。

树的基本算法有前中后序遍历和层次遍历，有的同学对前中后这三个分别具体表现的访问顺序比较模糊，其实当初我也是一样的，后面我学到了一点，你只需要记住： `所谓的前中后指的是根节点的位置，其他位置按照先左后右排列即可` 。比如前序遍历就是 `根左右` ，中序就是 `左根右` ，后序就是 `左右根` ， 很简单吧?

我刚才提到了树是一种递归的数据结构，因此树的遍历算法使用递归去完成非常简单，幸运的是树的算法基本上都要依赖于树的遍历。

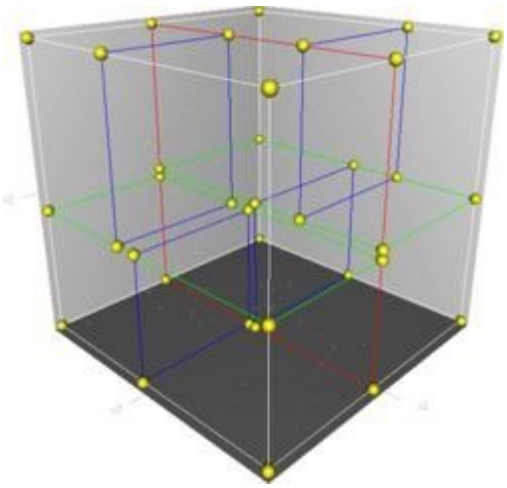
但是递归在计算机中的性能一直都有问题，因此掌握不那么容易理解的"命令式地迭代"遍历算法在某些情况下是有用的。如果你使用迭代式方式去遍历的话，可以借助上面提到的 `栈` 来进行，可以极大减少代码量。

> 如果使用栈来简化运算，由于栈是 FILO 的，因此一定要注意左右子树的推入顺序。

树的重要性质：

- 如果树有 n 个顶点，那么其就有 n - 1 条边，这说明了树的顶点数和边数是同阶的。
- 任何一个节点到根节点存在 `唯一` 路径，路径的长度为节点所处的深度

实际使用的树有可能会更复杂，比如使用在游戏中的碰撞检测可能会用到四叉树或者八叉树。以及 k 维的树结构 `k-d 树` 等。



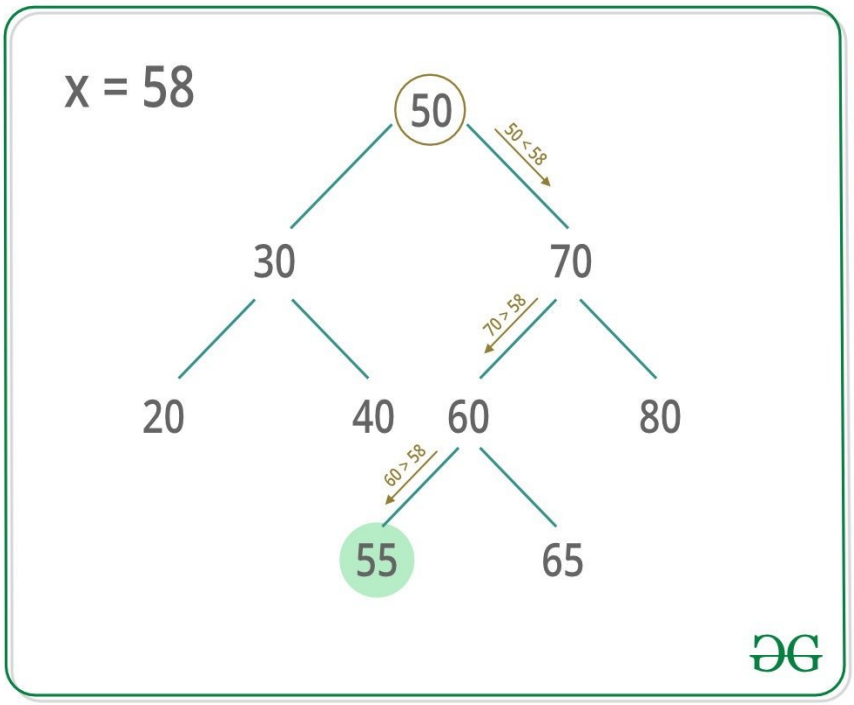
（图片来自 https://zh.wikipedia.org/wiki/K-d%E6%A0%91）

二叉树

二叉树是节点度数不超过二的树，是树的一种特殊子集，有趣的是二叉树这种被限制的树结构却能够表示和实现所有的树， 它背后的原理正是 `长子 + 兄弟` 法，用邓老师的话说就是 `二叉树是多叉树的特例，但在有根且有序时，其描述能力却足以覆盖后者` 。

> 实际上， 在你使用 `长子 + 兄弟` 法表示树的同时，进行 45 度角旋转即可。

一个典型的二叉树：



（图片来自 https://github.com/trekhleb/javascript-algorithms/blob/master/src/data-structures/tree/README.zh-CN.md）

对于一般的树，我们通常会去遍历，这里又会有很多变种。

下面我列举一些二叉树遍历的相关算法：

- 94.binary-tree-inorder-traversal
- 102.binary-tree-level-order-traversal
- 103.binary-tree-zigzag-level-order-traversal
- 144.binary-tree-preorder-traversal
- 145.binary-tree-postorder-traversal
- 199.binary-tree-right-side-view

相关概念：

- 真二叉树 （所有节点的度数只能是偶数，即只能为 0 或者 2）

另外我也专门开设了 二叉树的遍历 章节，具体细节和算法可以去那里查看。

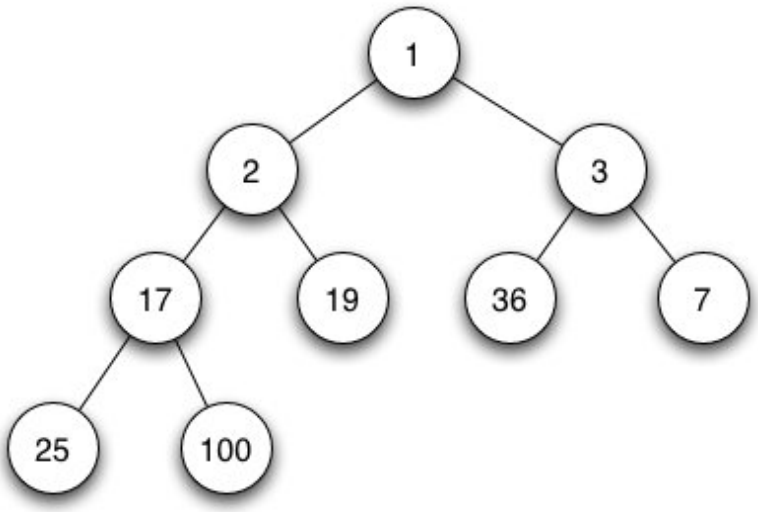
堆

堆其实是一种优先级队列，在很多语言都有对应的内置数据结构，很遗憾 javascript 没有这种原生的数据结构。不过这对我们理解和运用不会有影响。

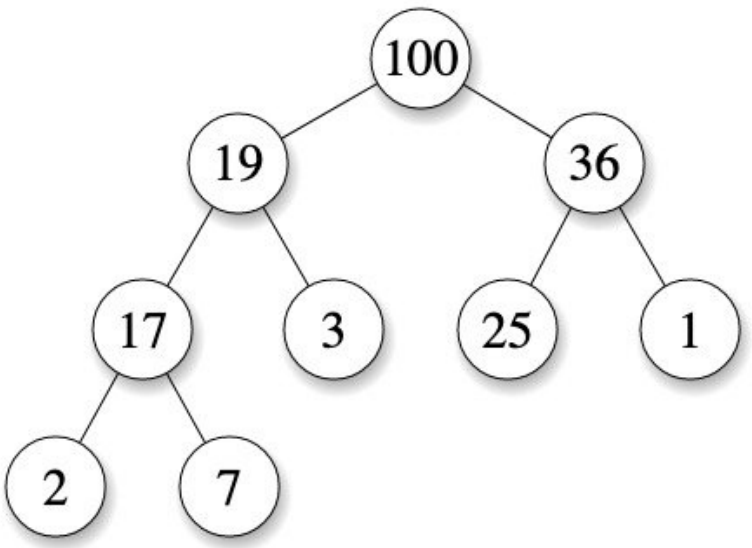
堆的一种典型的实现就是二叉堆。

二叉堆的特点：

- 在一个 最小堆 (min heap) 中，如果 P 是 C 的一个父级节点，那么 P 的 key（或 value) 应小于或等于 C 的对应值。 正因为此，堆顶元素一定是最小的，我们会利用这个特点求最小值或者第 k 小的值。



- 在一个 最大堆 (max heap) 中，P 的 key（或 value) 大于或等于 C 的对应值。



需要注意的是优先队列不仅有堆一种，还有更复杂的，但是通常来说，我们会把两者做等价。

相关算法：

- 295.find-median-from-data-stream

二叉查找树

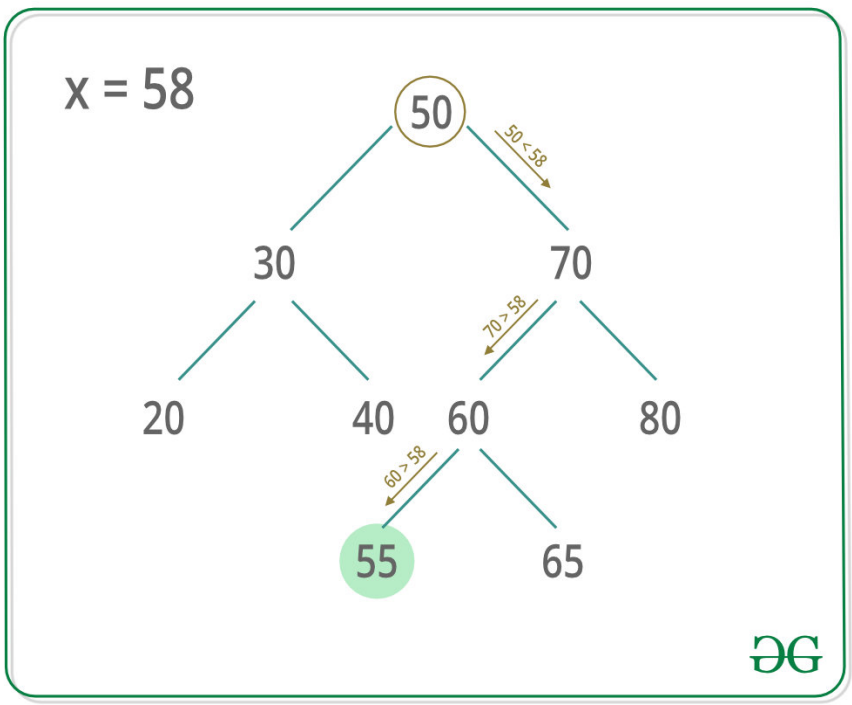
二叉排序树（Binary Sort Tree），又称二叉查找树（Binary Search Tree），亦称二叉搜索树。

二叉查找树具有下列性质的二叉树：

- 若左子树不空，则左子树上所有节点的值均小于它的根节点的值；
- 若右子树不空，则右子树上所有节点的值均大于它的根节点的值；
- 左、右子树也分别为二叉排序树；
- 没有键值相等的节点。

对于一个二叉查找树，常规操作有插入，查找，删除，找父节点，求最大值，求最小值。

二叉查找树，**之所以叫查找树就是因为其非常适合查找**。举个例子，如下一颗二叉查找树，我们想找节点值小于且最接近 58 的节点，搜索的流程如图所示：



（图片来自 https://www.geeksforgeeks.org/floor-in-binary-search-tree-bst/）

另外我们二叉查找树有一个性质是：`其中序遍历的结果是一个有序数组`。有时候我们可以利用到这个性质。

相关题目：

- 98.validate-binary-search-tree

二叉平衡树

平衡树是计算机科学中的一类数据结构，是一种改进的二叉查找树。一般的二叉查找树的查询复杂度取决于目标结点到树根的距离（即深度），因此当结点的深度普遍较大时，查询的均摊复杂度会上升。为了实现更高效的查询，产生了平衡树。

在这里，平衡指所有叶子的深度趋于平衡，更广义的是指在树上所有可能查找的均摊复杂度偏低。

一些数据库引擎内部就是用的这种数据结构，其目标也是将查询的操作降低到 logn（树的深度），可以简单理解为`树在数据结构层面构造了二分查找算法`。

基本操作：

- 旋转
- 插入
- 删除
- 查询前驱
- 查询后继

AVL

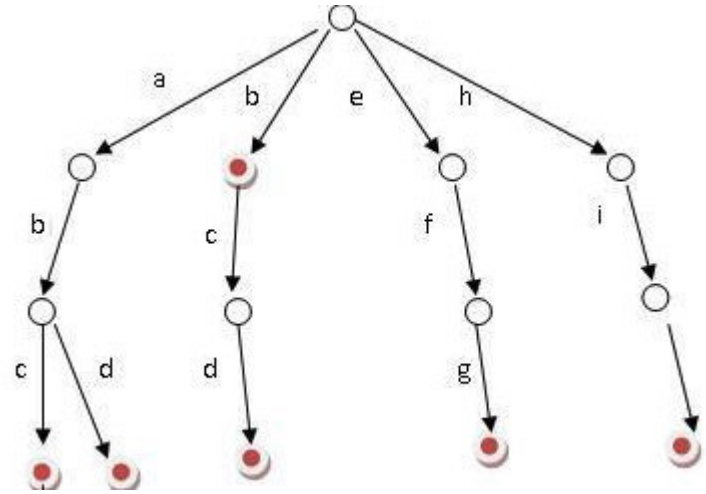
是最早被发明的自平衡二叉查找树。在 AVL 树中，任一节点对应的两棵子树的最大高度差为 1，因此它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下的时间复杂度都是 O(logn)。增加和删除元素的操作则可能需要借由一次或多次树旋转，以实现树的重新平衡。AVL 树得名于它的发明者 G. M. Adelson-Velsky 和 Evgenii Landis，他们在 1962 年的论文 An algorithm for the organization of information 中公开了这一数据结构。 节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。带有平衡因子 1、0 或 -1 的节点被认为是平衡的。带有平衡因子 -2 或 2 的节点被认为是不平衡的，并需要重新平衡这个树。平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。

红黑树

在 1972 年由鲁道夫·贝尔发明，被称为"对称二叉 B 树"，它现代的名字源于 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在 O(logn) 时间内完成查找，插入和删除，这里的 n 是树中元素的数目

字典树（前缀树）

又称 Trie 树，是一种树形结构。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。



（图来自 https://baike.baidu.com/item/%E5%AD%97%E5%85%B8%E6%A0%91/9825209?fr=aladdin) 它有 3 个基本性质：

- 根节点不包含字符，除根节点外每一个节点都只包含一个字符；
- 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串；
- 每个节点的所有子节点包含的字符都不相同。

immutable 与 字典树（非前端慎入）

`immutableJS` 的底层就是 `share + tree` . 这样看的话，其实和字典树是一致的。

关于这点，我写过一篇文章 immutablejs 是如何优化我们的代码的？，强烈建议前端开发阅读。

相关算法：

- 208.implement-trie-prefix-tree
- 211.add-and-search-word-data-structure-design
- 212.word-search-ii

图

前面讲的数据结构都可以看成是图的特例。 前面提到了二叉树完全可以实现其他树结构，其实有向图也完全可以实现无向图和混合图，因此有向图的研究一直是重点考察对象。

图论〔Graph Theory〕是数学的一个分支。它以图为研究对象。图论中的图是由若干给定的点及连接两点的线所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系，用点代表事物，用连接两点的线表示相应两个事物间具有这种关系。

基本概念

- 无向图 & 有向图
- 有权图 & 无权图
- 入度 & 出度
- 路径 & 环
- 连通图 & 强连通图

在无向图中，若任意两个顶点 i 与 j 都有路径相通，则称该无向图为连通图。

在有向图中，若任意两个顶点 i 与 j 都有路径相通，则称该有向图为强连通图。

- 生成树

一个连通图的生成树是指一个连通子图，它含有图中全部 n 个顶点，但只有足以构成一棵树的 n-1 条边。一颗有 n 个顶点的生成树有且仅有 n-1 条边，如果生成树中再添加一条边，则必定成环。在连通网的所有生成树中，所有边的**代价和最小**的生成树，称为最小生成树，其中**代价和**指的是所有边的权重和。

图的建立

一般图的题目都不会给你一个现成的图结构。当你知道这是一个图的题目的时候，解题的第一步通常就是建图。这里我简单介绍两种常见的建图方式。

邻接矩阵（常见）

使用一个 n * n 的矩阵来描述图 graph，其就是一个二维的矩阵，其中 graph[i][j] 描述边的关系。

一般而言，我都用 graph[i][j] = 1 来表示 顶点 i 和顶点 j 之间有一条边，并且边的指向是从 i 到 j。用 graph[i][j] = 0 来表示 顶点 i 和顶点 j 之间不存在一条边。 对于有权图来说，我们可以存储其他数字，表示的是权重。

这种存储方式的空间复杂度为 O(n ^ 2)，其中 n 为顶点个数。如果是稀疏图（图的边的数目远小于顶点的数目），那么会很浪费空间。并且如果图是无向图，始终至少会有 50 % 的空间浪费。下面的图也直观地反应了这一点。

邻接矩阵的优点主要有：

1. 直观，简单。
2. 判断两个顶点是否连接，获取入度和出度以及更新度数，时间复杂度都是 O(1)

由于使用起来比较简单， 因此我的所有的需要建图的题目基本都用这种方式。

比如力扣 743. 网络延迟时间。 题目描述：

```
有 N 个网络节点，标记为 1 到 N。

给定一个列表 times，表示信号经过有向边的传递时间。 times[i] = (u,

现在，我们从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号?


示例：

输入：times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2
输出：2


注意：

N 的范围在 [1, 100] 之间。
K 的范围在 [1, N] 之间。
times 的长度在 [1, 6000] 之间。
所有的边 times[i] = (u, v, w) 都有 1 <= u, v <= N 且 0 <= w <
```

这是一个典型的图的题目，对于这道题，我们如何用邻接矩阵建图呢?

一个典型的建图代码：

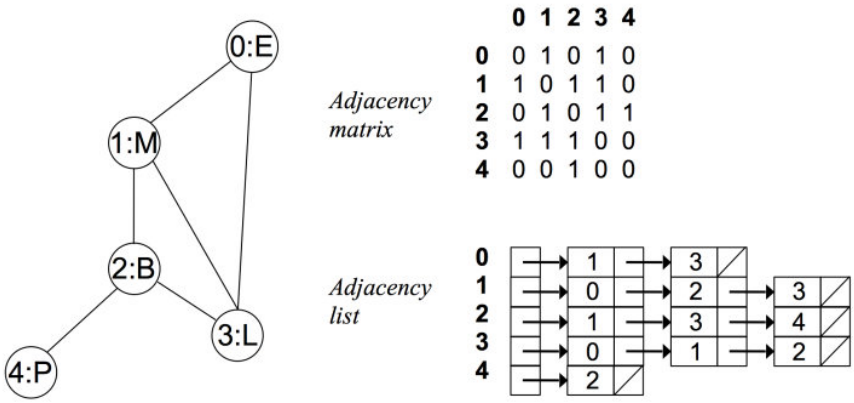
```
    graph = collections.defaultdict(list)
    for fr, to, w in times:
        graph[fr - 1].append((to - 1, w))
```

这就构造了一个临界矩阵，之后我们基于这个邻接矩阵遍历图即可。

邻接表

对于每个点，存储着一个链表，用来指向所有与该点直接相连的点。对于有权图来说，链表中元素值对应着权重。

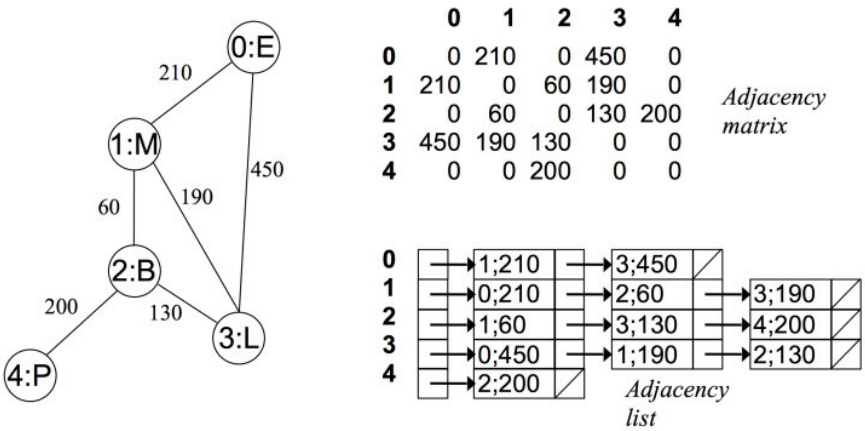
例如在无向无权图中：



（图片来自 https://zhuanlan.zhihu.com/p/25498681）

可以看出在无向图中，邻接矩阵关于对角线对称，而邻接链表总有两条对称的边。

而在有向无权图中：



（图片来自 https://zhuanlan.zhihu.com/p/25498681）

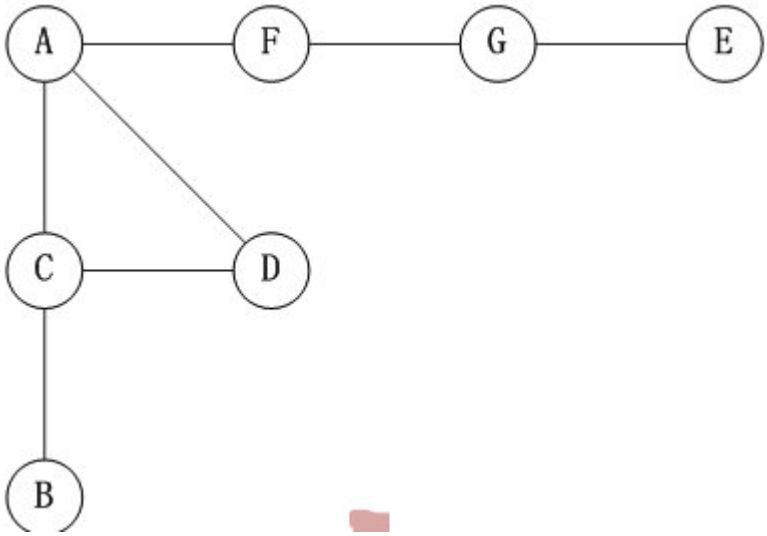
图的遍历

图建立好了，接下来就是要遍历。不管你是什么算法，肯定都要遍历的，一般有以下两种方法（其他奇葩的遍历方式实际意义不大，没有必要学习）。不管是哪一种遍历， 如果图有环，就一定要记录节点的访问情况，防止死循环。当然你可能不需要真正地使用一个集合记录节点的访问情况，比如使用一个数据范围外的数据原地标记，这样的空间复杂度会是$O(1)$。

这里以有向图为例， 有向图也是类似，这里不再赘述。

深度优先遍历：(Depth First Search, DFS)

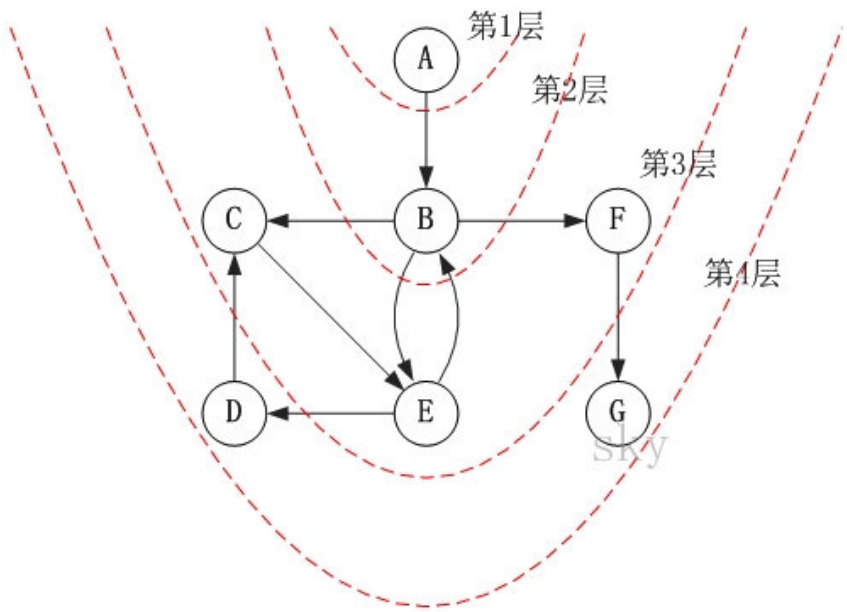
深度优先遍历图的方法是，从图中某顶点 v 出发， 不断访问邻居， 邻居的邻居直到访问完毕。



如上图， 如果我们使用 DFS，并且从 A 节点开始的话， **一个可能的**的访问顺序是： **A -> C -> B -> D -> F -> G -> E**，当然也可能是 **A -> D -> C -> B -> F -> G -> E** 等，具体取决于你的代码，但他们都是深度优先的。

广度优先搜索：(Breadth First Search, BFS)

广度优先搜索，可以被形象地描述为 "浅尝辄止"，它也需要一个队列以保持遍历过的顶点顺序，以便按出队的顺序再去访问这些顶点的邻接顶点。



如上图， 如果我们使用 BFS，并且从 A 节点开始的话， **一个可能的**的访问顺序是： **A -> B -> C -> F -> E -> G -> D**，当然也可能是 **A -> B -> F -> E -> C -> G -> D** 等，具体取决于你的代码，但他们都是广度优先的。

需要注意的是 DFS 和 BFS 只是一种算法思想，不是一种具体的算法。因此其有着很强的适应性，而不是局限于特点的数据结构的，本文讲的图可以用，前面讲的树也可以用。实际上， 只要是**非线性的数据结构都可以用**。

常见算法

图的题目的算法比较适合套模板。题目类型主要有：

- dijkstra
- floyd_warshall
- 最小生成树（Kruskal & Prim）
- A 星寻路算法
- 二分图（染色法）
- 拓扑排序

下面列举常见算法的模板，以下所有的模板都是基于邻接矩阵。

最短距离，最短路径

dijkstra 算法

DIJKSTRA 算法主要解决的是图中任意两点的最短距离。

算法的基本思想是贪心，每次都遍历所有邻居，并从中找到距离最小的，本质上是一种广度优先遍历。这里我们借助堆这种数据结构，使得可以在 $logN$ 的时间内找到 cost 最小的点。

代码模板：

```
import heapq


def dijkstra(graph, start, end):
    # 堆里的数据都是 (cost, i) 的二元祖，其含义是“从 start 走到 i
    heap = [(0, start)]
    visited = set()
    while heap:
        (cost, u) = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        if u == end:
            return cost
        for v, c in graph[u]:
            if v in visited:
                continue
            next = cost + c
            heapq.heappush(heap, (next, v))
    return -1
```

比如一个图是这样的：

```
E -- 1 --> B -- 1 --> C -- 1 --> D -- 1 --> F
 \                                          /\
  \                                         ||
    -------- 2 ---------> G ------- 1 ------
```

我们使用邻接矩阵来构造：

```
G = {
    "B": [["C", 1]],
    "C": [["D", 1]],
    "D": [["F", 1]],
    "E": [["B", 1], ["G", 2]],
    "F": [],
    "G": [["F", 1]],
}

shortDistance = dijkstra(G, "E", "C")
print(shortDistance)  # E -- 3 --> F -- 3 --> C == 6
```

会了这个算法模板， 你就可以去 AC 743. 网络延迟时间 了。

完整代码：

```
class Solution:
    def dijkstra(self, graph, start, end):

        heap = [(0, start)]
        visited = set()
        while heap:
            (cost, u) = heapq.heappop(heap)
            if u in visited:
                continue
            visited.add(u)
            if u == end:
                return cost
            for v, c in graph[u]:
                if v in visited:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v))
        return -1
    def networkDelayTime(self, times: List[List[int]], N: 
        graph = collections.defaultdict(list)
        for fr, to, w in times:
            graph[fr - 1].append((to - 1, w))
        ans = -1
        for to in range(N):
            dist = self.dijkstra(graph, K - 1, to)
            if dist == -1: return -1
            ans = max(ans, dist)
        return ans
```

你学会了么?

floyd_warshall 算法

floyd_warshall 也是解决两个点距离的算法，只不过由于其计算过程会把中间运算结果保存起来防止重复计算，因此其特别适合**求图中任意两点的距离**，比如力扣的 1462. 课程安排 IV。除了这个优点，还有一个非常重要的点是 floyd_warshall 算法由于使用了动态规划的思想而不是贪心，因此其**可以处理负权重**的情况。

floyd_warshall 的基本思想是动态规划。该算法的时间复杂度是 $O(N^3)$，空间复杂度是 $O(N^2)$，其中 N 为顶点个数。

算法也不难理解，简单来说就是： **i 到 j 的最短路径 = i 到 k 的最短路径 + k 到 j 的最短路径**的最小值。

算法的正确性不言而喻，因为从 i 到 j，要么直接到，要么经过图中的另外一个点 k。直接到的情况就是我们算法的临界值，而经过中间点的情况取出最小的，自然就是 i 到 j 的最短距离。。

代码模板：

```
# graph 是邻接矩阵，v 是顶点个数
def floyd_warshall(graph, v):
    dist = [[float("inf") for _ in range(v)] for _ in range

    for i in range(v):
        for j in range(v):
            dist[i][j] = graph[i][j]

    # check vertex k against all other vertices (i, j)
    for k in range(v):
        # looping through rows of graph array
        for i in range(v):
            # looping through columns of graph array
            for j in range(v):
                if (
                    dist[i][k] != float("inf")
                    and dist[k][j] != float("inf")
                    and dist[i][k] + dist[k][j] < dist[i][
                ):
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist, v
```

我们回过头来看下如何套模板解决 力扣的 1462. 课程安排 IV，题目描述：

你总共需要上 n 门课，课程编号依次为 0 到 n-1 。

有的课会有直接的先修课程，比如如果想上课程 0 ，你必须先上课程 1 ，那么

给你课程总数 n 和一个直接先修课程数对列表 prerequisite 和一个查询对

对于每个查询对 queries[i] ，请判断 queries[i][0] 是否是 queries

请返回一个布尔值列表，列表中每个元素依次分别对应 queries 每个查询对的

注意：如果课程 a 是课程 b 的先修课程且课程 b 是课程 c 的先修课程，那

示例 1：

输入：n = 2, prerequisites = [[1,0]], queries = [[0,1],[1,0]
输出：[false,true]
解释：课程 0 不是课程 1 的先修课程，但课程 1 是课程 0 的先修课程。
示例 2：

输入：n = 2, prerequisites = [], queries = [[1,0],[0,1]]
输出：[false,false]
解释：没有先修课程对，所以每门课程之间是独立的。
示例 3：

输入：n = 3, prerequisites = [[1,2],[1,0],[2,0]], queries =
输出：[true,true]
示例 4：

输入：n = 3, prerequisites = [[1,0],[2,0]], queries = [[0,1]
输出：[false,true]
示例 5：

输入：n = 5, prerequisites = [[0,1],[1,2],[2,3],[3,4]], quer
输出：[true,false,true,false]

提示：

2 <= n <= 100
0 <= prerequisite.length <= (n * (n - 1) / 2)
0 <= prerequisite[i][0], prerequisite[i][1] < n
prerequisite[i][0] != prerequisite[i][1]

```
先修课程图中没有环。
先修课程图中没有重复的边。
1 <= queries.length <= 10^4
queries[i][0] != queries[i][1]
```

这道题也可以使用 floyd_warshall 来做。 你可以这么想， 如果从 i 到 j 的距离大于 0，那不就是先修课么。而这道题数据范围 queries 大概是 10 ^ 4 ， 用上面的 dijkstra 算法肯定超时，，因此 floyd_warshall 算法是明智的选择。

我这里直接套模板，稍微改下就过了。完整代码：

```
class Solution:
    def floyd_warshall(self, dist, v):
        for k in range(v):
            for i in range(v):
                for j in range(v):
                    dist[i][j] = dist[i][j] or (dist[i][k]

        return dist

    def checkIfPrerequisite(self, n: int, prerequisites: Li
        graph = [[False] * n for _ in range(n)]
        ans = []

        for to, fr in prerequisites:
            graph[fr][to] = True
        dist = self.floyd_warshall(graph, n)
        for to, fr in queries:
            ans.append(bool(dist[fr][to]))
        return ans
```

A 星寻路算法

A 星寻路解决的问题是在一个二维的表格中找出任意两点的最短距离或者最短路径。常用于游戏中的 NPC 的移动计算，是一种常用启发式算法。一般这种题目都会有障碍物。除了障碍物，力扣的题目还会增加一些限制，使得题目难度增加。

这种题目一般都是力扣的困难难度。理解起来不难， 但但是完整没有 bug 地写出来却不那么容易。

在该算法中，我们从起点开始，检查其相邻的四个方格并尝试扩展，直至找到目标。A 星寻路算法的寻路方式不止一种，感兴趣的可以自行了解一下。

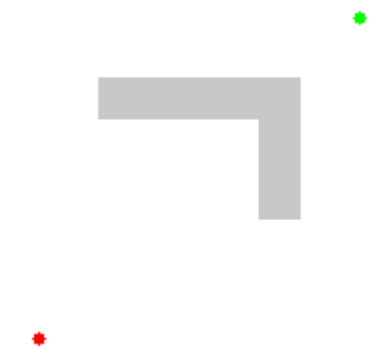
公式表示为： f(n)=g(n)+h(n)。

其中：

- f(n) 是从初始状态经由状态 n 到目标状态的估计代价，
- g(n) 是在状态空间中从初始状态到状态 n 的实际代价，
- h(n) 是从状态 n 到目标状态的最佳路径的估计代价。

如果 g(n)为 0，即只计算任意顶点 n 到目标的评估函数 h(n)，而不计算起点到顶点 n 的距离，则算法转化为使用贪心策略的最良优先搜索，速度最快，但可能得不出最优解； 如果 h(n)不大于顶点 n 到目标顶点的实际距离，则一定可以求出最优解，而且 h(n)越小，需要计算的节点越多，算法效率越低，常见的评估函数有——欧几里得距离、曼哈顿距离、切比雪夫距离； 如果 h(n)为 0，即只需求出起点到任意顶点 n 的最短路径 g(n)，而不计算任何评估函数 h(n)，则转化为单源最短路径问题，即 Dijkstra 算法，此时需要计算最多的顶点；

这里有一个重要的概念是**估价算法**，一般我们使用 **曼哈顿距离**来进行估价，即 `H(n) = D * (abs ( n.x – goal.x ) + abs ( n.y – goal.y ) )` 。



（图来自维基百科 https://zh.wikipedia.org/wiki/A*%E6%90%9C%E5%B0%8B%E6%BC%94%E7%AE%97%E6%B3%95 ）

一个完整的代码模板：

```
grid = [
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],  # 0 are free path whereas 1's are
    [0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 0],
]

"""
heuristic = [[9, 8, 7, 6, 5, 4],
             [8, 7, 6, 5, 4, 3],
             [7, 6, 5, 4, 3, 2],
             [6, 5, 4, 3, 2, 1],
             [5, 4, 3, 2, 1, 0]]"""

init = [0, 0]
goal = [len(grid) - 1, len(grid[0]) - 1]  # all coordinates
cost = 1

# the cost map which pushes the path closer to the goal
heuristic = [[0 for row in range(len(grid[0]))] for col in
for i in range(len(grid)):
    for j in range(len(grid[0])):
        heuristic[i][j] = abs(i - goal[0]) + abs(j - goal[1
        if grid[i][j] == 1:
            heuristic[i][j] = 99  # added extra penalty in


# the actions we can take
delta = [[-1, 0], [0, -1], [1, 0], [0, 1]]  # go up  # go


# function to search the path
def search(grid, init, goal, cost, heuristic):

    closed = [
        [0 for col in range(len(grid[0]))] for row in range
    ]  # the reference grid
    closed[init[0]][init[1]] = 1
    action = [
        [0 for col in range(len(grid[0]))] for row in range
    ]  # the action grid

    x = init[0]
    y = init[1]
    g = 0
    f = g + heuristic[init[0]][init[0]]
    cell = [[f, g, x, y]]
```

```
    found = False  # flag that is set when search is comple
    resign = False  # flag set if we can't find expand

    while not found and not resign:
        if len(cell) == 0:
            return "FAIL"
        else:  # to choose the least costliest action so as
            cell.sort()
            cell.reverse()
            next = cell.pop()
            x = next[2]
            y = next[3]
            g = next[1]

            if x == goal[0] and y == goal[1]:
                found = True
            else:
                for i in range(len(delta)):  # to try out d
                    x2 = x + delta[i][0]
                    y2 = y + delta[i][1]
                    if x2 >= 0 and x2 < len(grid) and y2 >=
                        if closed[x2][y2] == 0 and grid[x2]
                            g2 = g + cost
                            f2 = g2 + heuristic[x2][y2]
                            cell.append([f2, g2, x2, y2])
                            closed[x2][y2] = 1
                            action[x2][y2] = i
    invpath = []
    x = goal[0]
    y = goal[1]
    invpath.append([x, y])  # we get the reverse path from
    while x != init[0] or y != init[1]:
        x2 = x - delta[action[x][y]][0]
        y2 = y - delta[action[x][y]][1]
        x = x2
        y = y2
        invpath.append([x, y])

    path = []
    for i in range(len(invpath)):
        path.append(invpath[len(invpath) - 1 - i])
    print("ACTION MAP")
    for i in range(len(action)):
        print(action[i])

    return path
```

```
a = search(grid, init, goal, cost, heuristic)
for i in range(len(a)):
    print(a[i])
```

典型题目1263. 推箱子

拓扑排序

在计算机科学领域，有向图的拓扑排序是对其顶点的一种线性排序，使得对于从顶点 u 到顶点 v 的每个有向边 uv， u 在排序中都在之前。当且仅当图中没有定向环时（即有向无环图），才有可能进行拓扑排序。

典型的题目就是给你一堆课程，课程之间有先修关系，让你给出一种可行的学习路径方式，要求先修的课程要先学。任何有向无环图至少有一个拓扑排序。已知有算法可以在线性时间内，构建任何有向无环图的拓扑排序。

Kahn 算法

简单来说，假设 L 是存放结果的列表，先找到那些入度为零的节点，把这些节点放到 L 中，因为这些节点没有任何的父节点。然后把与这些节点相连的边从图中去掉，再寻找图中的入度为零的节点。对于新找到的这些入度为零的节点来说，他们的父节点已经都在 L 中了，所以也可以放入 L。重复上述操作，直到找不到入度为零的节点。如果此时 L 中的元素个数和节点总数相同，说明排序完成；如果 L 中的元素个数和节点总数不同，说明原图中存在环，无法进行拓扑排序。

```
def topologicalSort(graph):
    """
    Kahn's Algorithm is used to find Topological ordering o
    using BFS
    """
    indegree = [0] * len(graph)
    queue = []
    topo = []
    cnt = 0

    for key, values in graph.items():
        for i in values:
            indegree[i] += 1

    for i in range(len(indegree)):
        if indegree[i] == 0:
            queue.append(i)

    while queue:
        vertex = queue.pop(0)
        cnt += 1
        topo.append(vertex)
        for x in graph[vertex]:
            indegree[x] -= 1
            if indegree[x] == 0:
                queue.append(x)

    if cnt != len(graph):
        print("Cycle exists")
    else:
        print(topo)


# Adjacency List of Graph
graph = {0: [1, 2], 1: [3], 2: [3], 3: [4, 5], 4: [], 5: []
topologicalSort(graph)
```

最小生成树

Kruskal 和 Prim 这两个算法暂时先不写了，先留个模板给大家。

Kruskal

```
from typing import List, Tuple


def kruskal(num_nodes: int, num_edges: int, edges: List[Tu
    """
    >>> kruskal(4, 3, [(0, 1, 3), (1, 2, 5), (2, 3, 1)])
    [(2, 3, 1), (0, 1, 3), (1, 2, 5)]

    >>> kruskal(4, 5, [(0, 1, 3), (1, 2, 5), (2, 3, 1), (0,
    [(2, 3, 1), (0, 2, 1), (0, 1, 3)]

    >>> kruskal(4, 6, [(0, 1, 3), (1, 2, 5), (2, 3, 1), (0,
    ... (2, 1, 1)])
    [(2, 3, 1), (0, 2, 1), (2, 1, 1)]
    """
    edges = sorted(edges, key=lambda edge: edge[2])

    parent = list(range(num_nodes))

    def find_parent(i):
        if i != parent[i]:
            parent[i] = find_parent(parent[i])
        return parent[i]

    minimum_spanning_tree_cost = 0
    minimum_spanning_tree = []

    for edge in edges:
        parent_a = find_parent(edge[0])
        parent_b = find_parent(edge[1])
        if parent_a != parent_b:
            minimum_spanning_tree_cost += edge[2]
            minimum_spanning_tree.append(edge)
            parent[parent_a] = parent_b

    return minimum_spanning_tree


if __name__ == "__main__":  # pragma: no cover
    num_nodes, num_edges = list(map(int, input().strip().sp
    edges = []

    for _ in range(num_edges):
        node1, node2, cost = [int(x) for x in input().strip
        edges.append((node1, node2, cost))

    kruskal(num_nodes, num_edges, edges)
```

Prim

```
import sys
from collections import defaultdict


def PrimsAlgorithm(l):  # noqa: E741

    nodePosition = []

    def get_position(vertex):
        return nodePosition[vertex]

    def set_position(vertex, pos):
        nodePosition[vertex] = pos

    def top_to_bottom(heap, start, size, positions):
        if start > size // 2 - 1:
            return
        else:
            if 2 * start + 2 >= size:
                m = 2 * start + 1
            else:
                if heap[2 * start + 1] < heap[2 * start + 2
                    m = 2 * start + 1
                else:
                    m = 2 * start + 2
            if heap[m] < heap[start]:
                temp, temp1 = heap[m], positions[m]
                heap[m], positions[m] = heap[start], positi
                heap[start], positions[start] = temp, temp1

                temp = get_position(positions[m])
                set_position(positions[m], get_position(pos
                set_position(positions[start], temp)

                top_to_bottom(heap, m, size, positions)

    # Update function if value of any node in min-heap deci
    def bottom_to_top(val, index, heap, position):
        temp = position[index]

        while index != 0:
            if index % 2 == 0:
                parent = int((index - 2) / 2)
            else:
                parent = int((index - 1) / 2)

            if val < heap[parent]:
                heap[index] = heap[parent]
```

```
                position[index] = position[parent]
                set_position(position[parent], index)
            else:
                heap[index] = val
                position[index] = temp
                set_position(temp, index)
                break
            index = parent
        else:
            heap[0] = val
            position[0] = temp
            set_position(temp, 0)

    def heapify(heap, positions):
        start = len(heap) // 2 - 1
        for i in range(start, -1, -1):
            top_to_bottom(heap, i, len(heap), positions)

    def deleteMinimum(heap, positions):
        temp = positions[0]
        heap[0] = sys.maxsize
        top_to_bottom(heap, 0, len(heap), positions)
        return temp

    visited = [0 for i in range(len(l))]
    Nbr_TV = [-1 for i in range(len(l))]  # Neighboring Tre
    # Minimum Distance of explored vertex with neighboring
    # formed in graph
    Distance_TV = []  # Heap of Distance of vertices from t
    Positions = []

    for x in range(len(l)):
        p = sys.maxsize
        Distance_TV.append(p)
        Positions.append(x)
        nodePosition.append(x)

    TreeEdges = []
    visited[0] = 1
    Distance_TV[0] = sys.maxsize
    for x in l[0]:
        Nbr_TV[x[0]] = 0
        Distance_TV[x[0]] = x[1]
    heapify(Distance_TV, Positions)

    for i in range(1, len(l)):
        vertex = deleteMinimum(Distance_TV, Positions)
        if visited[vertex] == 0:
```

```
            TreeEdges.append((Nbr_TV[vertex], vertex))
            visited[vertex] = 1
            for v in l[vertex]:
                if visited[v[0]] == 0 and v[1] < Distance_T
                    Distance_TV[get_position(v[0])] = v[1]
                    bottom_to_top(v[1], get_position(v[0]),
                    Nbr_TV[v[0]] = vertex
    return TreeEdges


if __name__ == "__main__":  # pragma: no cover
    # < --------- Prims Algorithm --------- >
    n = int(input("Enter number of vertices: ").strip())
    e = int(input("Enter number of edges: ").strip())
    adjlist = defaultdict(list)
    for x in range(e):
        l = [int(x) for x in input().strip().split()]  # no
        adjlist[l[0]].append([l[1], l[2]])
        adjlist[l[1]].append([l[0], l[2]])
    print(PrimsAlgorithm(adjlist))
```

二分图

二分图我在这两道题中讲过了，大家看一下之后把这两道题做一下就行了。其实这两道题和一道题没啥区别。

- 0886. 可能的二分法
- 0785. 判断二分图

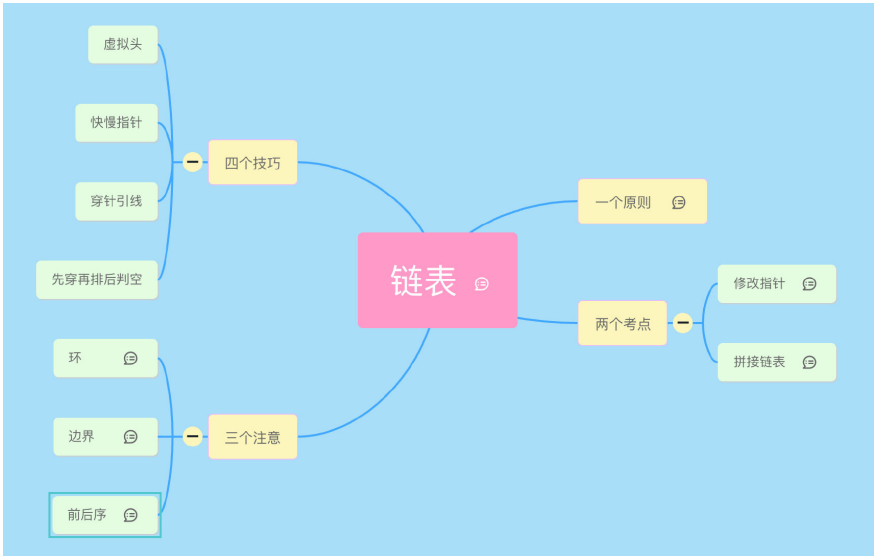
推荐顺序为： 先看 886 再看 785。

总结

理解图的常见概念，我们就算入门了。接下来，我们就可以做题了，一般的图题目第一步都是建图，第二步都是基于第一步的图进行遍历以寻找可行解。

图的题目相对而言比较难，尤其是代码书写层面。但是就面试题目而言，图的题目类型却不多，而且很多题目都是套模板就可以解决。因此建议大家多练习模板，并自己多手敲，确保可以自己敲出来。

几乎刷完了力扣所有的链表题，我发现了这些东西。。。



先上下本文的提纲，这个是我用 mindmap 画的一个脑图，之后我后继续完善，将其他专题逐步完善起来。

> 大家也可以使用 vscode blink-mind 打开源文件查看，里面有一些笔记可以点开查看。源文件可以去我的公众号《力扣加加》回复脑图获取，以后脑图也会持续更新更多内容。vscode 插件地址：https://marketplace.visualstudio.com/items?itemName=awehook.vscode-blink-mind

大家好，我是 lucifer。今天给大家带来的专题是《链表》。很多人觉得链表是一个很难的专题。实际上，只要你掌握了诀窍，它并没那么难。接下来，我们展开说说。

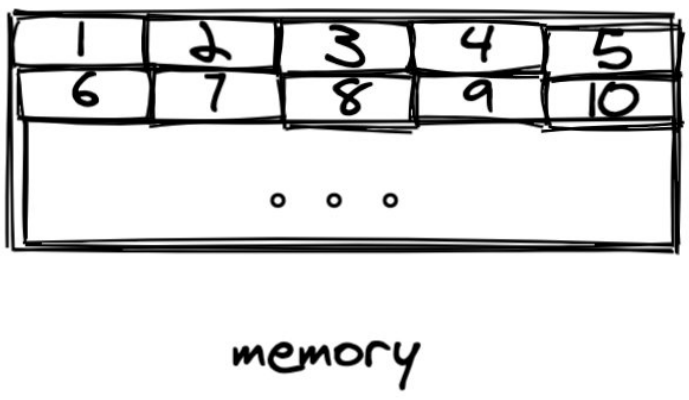
链表标签在 leetcode 一共有 **54 道题**。 为了准备这个专题，我花了几天时间将 leetcode 几乎所有的链表题目都刷了一遍。

可以看出，除了六个上锁的，其他我都刷了一遍。而实际上，这六个上锁的也没有什么难度，甚至和其他 48 道题差不多。

通过集中刷这些题，我发现了一些有趣的信息，今天就分享给大家。

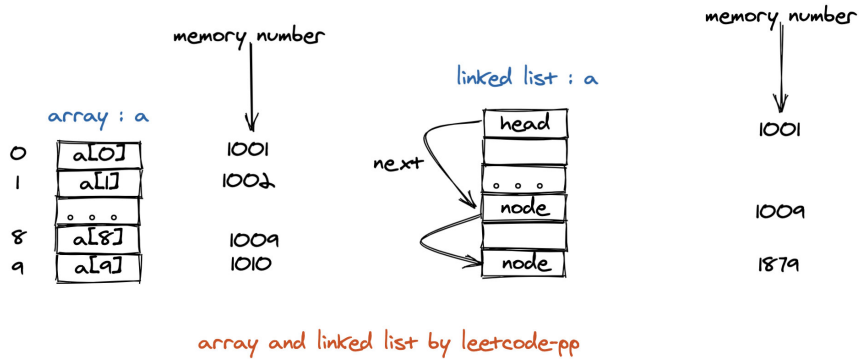
简介

各种数据结构，不管是队列，栈等线性数据结构还是树，图的等非线性数据结构，从根本上底层都是数组和链表。不管你用的是数组还是链表，用的都是计算机内存，物理内存是一个个大小相同的内存单元构成的，如图：



（图 1. 物理内存）

而数组和链表虽然用的都是物理内存，都是两者在对物理的使用上是非常不一样的，如图：



（图 2. 数组和链表的物理存储图）

不难看出，数组和链表只是使用物理内存的两种方式。

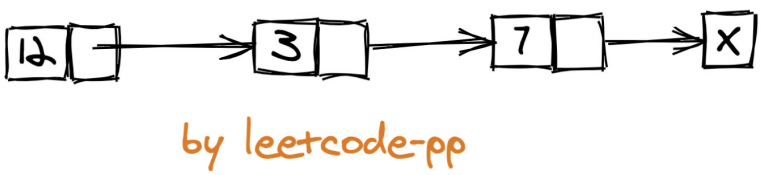
数组是连续的内存空间，通常每一个单位的大小也是固定的，因此可以按下标随机访问。而链表则不一定连续，因此其查找只能依靠别的方式，一般我们是通过一个叫 next 指针来遍历查找。链表其实就是一个结构体。比如一个可能的单链表的定义可以是：

```
interface ListNode<T> {
  data: T;
  next: ListNode<T>;
}
```

data 是数据域，存放数据，next 是一个指向下一个节点的指针。

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。

从上面的物理结构图可以看出数组是一块连续的空间，数组的每一项都是紧密相连的，因此如果要执行插入和删除操作就很麻烦。对数组头部的插入和删除时间复杂度都是$O(N)$，而平均复杂度也是$O(N)$，只有对尾部的插入和删除才是$O(1)$。简单来说”数组对查询特别友好，对删除和添加不友好“。为了解决这个问题，就有了链表这种数据结构。链表适合在数据需要有一定顺序，但是又需要进行频繁增删除的场景，具体内容参考后面的《链表的基本操作》小节。



（图 3. 一个典型的链表逻辑表示图）

> 后面所有的图都是基于逻辑结构，而不是物理结构

链表只有一个后驱节点 next，如果是双向链表还会有一个前驱节点 pre。

> 有没有想过为啥只有二叉树，而没有一叉树。实际上链表就是特殊的树，即一叉树。

链表的基本操作

要想写出链表的题目， 熟悉链表的各种基本操作和复杂度是必须的。

插入

插入只需要考虑要插入位置前驱节点和后继节点（双向链表的情况下需要更新后继节点）即可，其他节点不受影响，因此在给定指针的情况下插入的操作时间复杂度为 `O(1)` 。这里给定指针中的指针指的是插入位置的前驱节点。

伪代码：

```
temp = 待插入位置的前驱节点.next
待插入位置的前驱节点.next = 待插入指针
待插入指针.next = temp
```

如果没有给定指针，我们需要先遍历找到节点，因此最坏情况下时间复杂度为 `O(N)` 。

> 提示 1: 考虑头尾指针的情况。
>
> 提示 2: 新手推荐先画图，再写代码。等熟练之后，自然就不需要画图了。

删除

只需要将需要删除的节点的前驱指针的 next 指针修正为其下下个节点即可，注意考虑**边界条件**。

伪代码：

```
待删除位置的前驱节点.next = 待删除位置的前驱节点.next.next
```

> 提示 1: 考虑头尾指针的情况。
>
> 提示 2: 新手推荐先画图，再写代码。等熟练之后，自然就不需要画图了。

遍历

遍历比较简单，直接上伪代码。

迭代伪代码：

```
当前指针 =  头指针
while 当前节点不为空 {
   print(当前节点)
   当前指针 = 当前指针.next
}
```

一个前序遍历的递归的伪代码：

```
dfs(cur) {
    if 当前节点为空 return
    print(cur.val)
    return dfs(cur.next)
}
```

链表和数组到底有多大的差异?

熟悉我的小伙伴应该经常听到我说过一句话，那就是**数组和链表同样作为线性的数组结构，二者在很多方便都是相同的，只在细微的操作和使用场景上有差异而已**。而使用场景，很难在题目中直接考察。

> 实际上，使用场景是可以死记硬背的。

因此，对于我们做题来说，**二者的差异通常就只是细微的操作差异**。这么说大家可能感受不够强烈，我给大家举几个例子。

数组的遍历：

```
for(int i = 0; i < arr.size();i++) {
    print(arr[i])
}
```

链表的遍历：

```
for (ListNode cur = head; cur != null; cur = cur.next) {
    print(cur.val)
}
```

是不是很像?

可以看出二者逻辑是一致的，只不过细微操作不一样。比如：

- 数组是索引 ++
- 链表是 cur = cur.next

如果我们需要逆序遍历呢?

```
for(int i = arr.size() - 1; i > - 1;i--) {
    print(arr[i])
}
```

如果是链表，通常需要借助于双向链表。而双向链表在力扣的题目很少，因此大多数你没有办法拿到前驱节点，这也是为啥很多时候会自己记录一个前驱节点 pre 的原因。

```
for (ListNode cur = tail; cur != null; cur = cur.pre) {
    print(cur.val)
}
```

如果往数组末尾添加一个元素就是：

```
arr.push(1)
```

链表的话，很多语言没有内置的数组类型。比如力扣通常使用如下的类来模拟。

```
 public class ListNode {
      int val;
      ListNode next;
      ListNode() {}
      ListNode(int val) { this.val = val; }
      ListNode(int val, ListNode next) { this.val = val; th
 }
```

我们是不能直接调用 push 方法的。想一下，如果让你实现这个，你怎么做？你可以先自己想一下，再往下看。

3...2...1

ok，其实很简单。

```
// 假设 tail 是链表的尾部节点
tail.next = new ListNode('lucifer')
tail = tail.next
```

经过上面两行代码之后， tail 仍然指向尾部节点。是不是很简单，你学会了么?

这有什么用？比如有的题目需要你复制一个新的链表， 你是不是需要开辟一个新的链表头，然后不断拼接（push）复制的节点？这就用上了。

对于数组的底层也是类似的，一个可能的数组 push 底层实现：

```
arr.length += 1
arr[arr.length - 1] = 'lucifer'
```

总结一下， 数组和链表逻辑上二者有很多相似之处，不同的只是一些使用场景和操作细节，对于做题来说，我们通常更关注的是操作细节。关于细节，接下来给大家介绍，这一小节主要让大家知道二者在思想和逻辑的**神相似**。

有些小伙伴做链表题先把链表换成数组，然后用数组做，本人不推荐这种做法，这等于是否认了链表存在的价值，小朋友不要模仿。

链表题难度几何？

链表题真的不难。说链表不难是有证据的。就拿 LeetCode 平台来说，处于困难难度的题目只有两个。

其中 第 23 题基本没有什么链表操作，一个常规的“归并排序”即可搞定，而合并两个有序链表是一个简单题。如果你懂得数组的归并排序和合并两个有序链表，应该轻松拿下这道题。

> 合并两个有序数组也是一个简单题目，二者难度几乎一样。

而对于第 25 题， 相信你看完本节的内容，也可以做出来。

不过，话虽这么说，但是还是有很多小朋友给我说 ”指针绕来绕去就绕晕了“， ”老是死循环“ 。。。。。。链表题目真的那么难么？我们又该如何破解？ lucifer 给大家准备了一个口诀 **一个原则， 两种题型，三个注意，四个技巧**，让你轻松搞定链表题，再也不怕手撕链表。 我们依次来看下这个口诀的内容。

一个原则

一个原则就是 **画图**，尤其是对于新手来说。不管是简单题还是难题一定要画图，这是贯穿链表题目的一个准则。

画图可以减少我们的认知负担，这其实和打草稿，备忘录道理是一样的，将存在脑子里的东西放到纸上。举一个不太恰当的例子就是你的脑子就是CPU，脑子的记忆就是寄存器。寄存器的容量有限，我们需要把不那么频繁使用的东西放到内存，把寄存器用在真正该用的地方，这个内存就是纸或者电脑平板等一切你可以画图的东西。

画的好看不好看都不重要，能看清就行了。用笔随便勾画一下， 能看出关系就够了。

两个考点

我把力扣的链表做了个遍。发现一个有趣的现象，那就是链表的考点很单一。除了设计类题目，其考点无法就两点：

- 指针的修改
- 链表的拼接

指针的修改

其中指针修改最典型的就是链表反转。其实链表反转不就是修改指针么？

对于数组这种支持随机访问的数据结构来说， 反转很容易， 只需要头尾不断交换即可。

```
function reverseArray(arr) {
  let left = 0;
  let right = arr.length - 1;
  while (left < right) {
    const temp = arr[left];
    arr[left++] = arr[right];
    arr[right--] = temp;
  }
  return arr;
}
```

而对于链表来说，就没那么容易了。力扣关于反转链表的题简直不要太多了。

今天我给大家写了一个最完整的链表反转，以后碰到可以直接用。当然，前提是大家要先理解再去套。

接下来，我要实现的一个反转**任意一段链表**

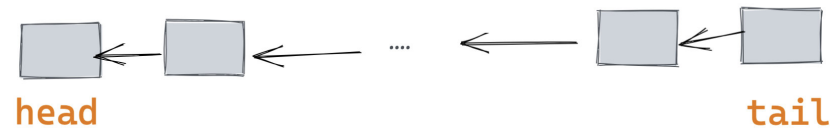
```
reverse(self, head: ListNode, tail: ListNode)。
```

其中 head 指的是需要反转的头节点，tail 是需要反转的尾节点。 不难看出，如果 head 是整个链表的头，tail 是整个链表的尾，那就是反转整个链表，否则就是反转局部链表。接下来，我们就来实现它。

首先，我们要做的就是画图。这个我在**一个原则**部分讲过了。

如下图，是我们需要反转的部分链表：

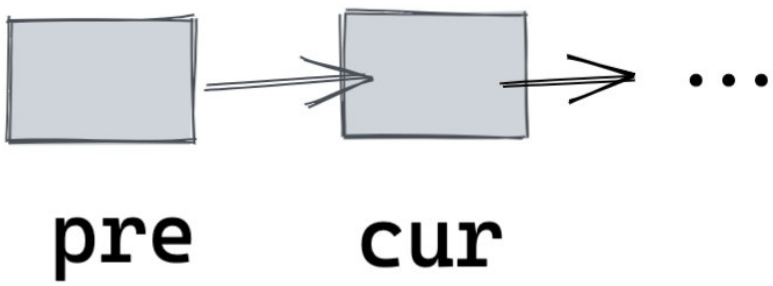
而我们期望反转之后的长这个样子：



不难看出， **最终返回 tail 即可**。

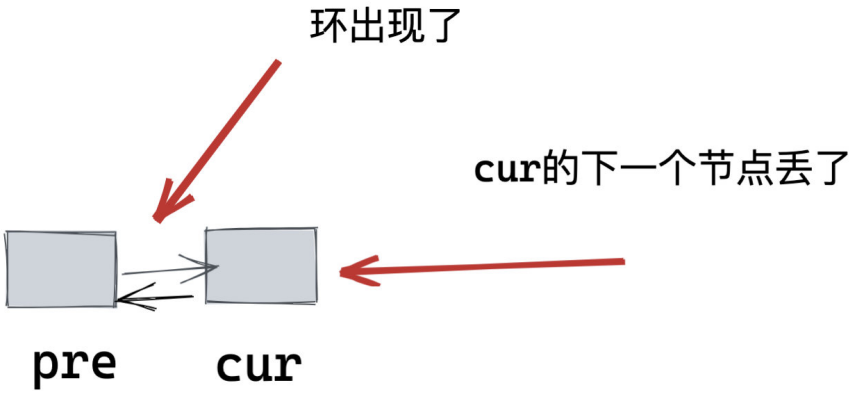
由于链表的递归性，实际上，我们只要反转其中相邻的两个，剩下的采用同样的方法完成即可。

> 链表是一种递归的数据结构，因此采用递归的思想去考虑往往事半功倍，关于递归思考链表将在后面《三个注意》部分展开。



对于两个节点来说，我们只需要下修改一次指针即可，这好像不难。

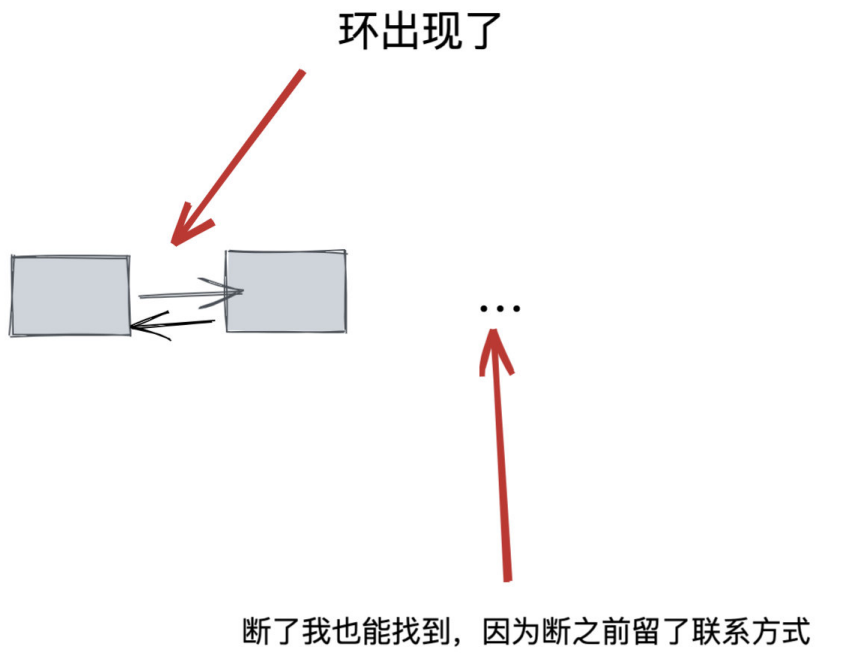
```
cur.next = pre
```



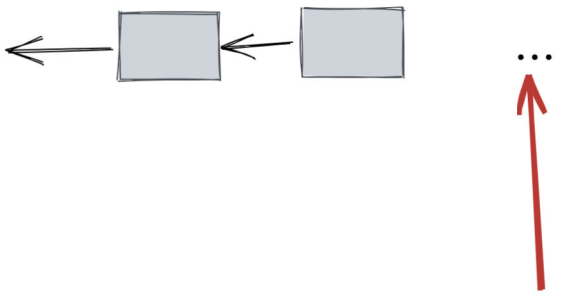
就是这一个操作，不仅硬生生有了环，让你死循环。还让不应该一刀两断的它们分道扬镳。

关于分道扬镳这个不难解决， 我们只需要反转前，记录一下下一个节点即可：

```
next = cur.next
cur.next = pre

cur = next
```



那么环呢？ 实际上， 环不用解决。因为如果我们是从前往后遍历，那么实际上，前面的链表已经被反转了，因此上面我的图是错的。正确的图应该是：



断了我也能找到，因为断之前留了联系方式

至此为止，我们可以写出如下代码：

```
    # 翻转一个子链表，并返回新的头与尾
    def reverse(self, head: ListNode, tail: ListNode):
        cur = head
        pre = None
        while cur != tail:
            # 留下联系方式
            next = cur.next
            # 修改指针
            cur.next = pre
            # 继续往下走
            pre = cur
            cur = next
        # 反转后的新的头尾节点返回出去
        return tail, head
```

如果你仔细观察，会发现，我们的 tail 实际上是没有被反转的。解决方法很简单，将 tail 后面的节点作为参数传进来呗。

```
class Solution:
    # 翻转一个子链表，并且返回新的头与尾
    def reverse(self, head: ListNode, tail: ListNode, termi
        cur = head
        pre = None
        while cur != terminal:
            # 留下联系方式
            next = cur.next
            # 修改指针
            cur.next = pre

            # 继续往下走
            pre = cur
            cur = next
         # 反转后的新的头尾节点返回出去
        return tail, head
```

相信你对反转链表已经有了一定的了解。后面我们还会对这个问题做更详细的讲解，大家先留个印象就好。

链表的拼接

大家有没有发现链表总喜欢穿来穿去（拼接）的？比如反转链表 II，再比如合并有序链表等。

为啥链表总喜欢穿来穿去呢？实际上，这就是链表存在的价值，这就是设计它的初衷呀！

链表的价值就在于其**不必要求物理内存的连续性，以及对插入和删除的友好**。这在文章开头的链表和数组的物理结构图就能看出来。

因此链表的题目很多拼接的操作。如果上面我讲的链表基本操作你会了，我相信这难不倒你。除了环，边界 等 。。。 ^_^。 这几个问题我们后面再看。

三个注意

链表最容易出错的地方就是我们应该注意的地方。链表最容易出的错 90 % 集中在以下三种情况：

- 出现了环，造成死循环。
- 分不清边界，导致边界条件出错。
- 搞不懂递归怎么做

接下来，我们一一来看。

环

环的考点有两个：

- 题目就有可能环，让你判断是否有环，以及环的位置。
- 题目链表没环，但是被你操作指针整出环了。

这里我们只讨论第二种，而第一种可以用我们后面提到的**快慢指针算法**。

避免出现环最简单有效的措施就是画图，如果两个或者几个链表节点构成了环，通过图是很容易看出来的。因此一个简单的**实操技巧就是先画图，然后对指针的操作都反应在图中**。

但是链表那么长，我不可能全部画出来呀。其实完全不用，上面提到了链表是递归的数据结构， 很多链表问题天生具有递归性，比如反转链表，因此**仅仅画出一个子结构就可以了**。这个知识，我们放在后面的**前后序**部分讲解。

边界

很多人错的是没有考虑边界。一个考虑边界的技巧就是看题目信息。

- 如果题目的头节点可能被移除，那么考虑使用虚拟节点，这样**头节点就变成了中间节点**，就不需要为头节点做特殊判断了。
- 题目让你返回的不是原本的头节点，而是尾部节点或者其他中间节点，这个时候要注意指针的变化。

以上两者部分的具体内容，我们在稍后讲到的虚拟头部分讲解。老规矩，大家留个印象即可。

前后序

ok，是时候填坑了。上面提到了链表结构天生具有递归性，那么使用递归的解法或者递归的思维都会对我们解题有帮助。

在 二叉树遍历 部分，我讲了二叉树的三种流行的遍历方法，分别是前序遍历，中序遍历和后序遍历。

前中后序实际上是指的当前节点相对子节点的处理顺序。如果先处理当前节点再处理子节点，那么就是前序。如果先处理左节点，再处理当前节点，最后处理右节点，就是中序遍历。后序遍历自然是最后处理当前节点了。

实际过程中，我们不会这么扣的这么死。比如：

```
def traverse(root):
    print('pre')
    traverse(root.left)
    traverse(root.righ)
    print('post')
```

如上代码，我们既在**进入左右节点前**有逻辑， 又在**退出左右节点之后**有逻辑。这算什么遍历方式呢？一般意义上，我习惯只看主逻辑的位置，如果你的主逻辑是在后面就是后序遍历，主逻辑在前面就是前序遍历。 这个不是重点，对我们解题帮助不大，对我们解题帮助大的是接下来要讲的内容。

> 绝大多数的题目都是单链表，而单链表只有一个后继指针。因此只有前序和后序，没有中序遍历。

还是以上面讲的经典的反转链表来说。 如果是前序遍历，我们的代码是这样的：

```
def dfs(head, pre):
    if not head: return pre
    next = head.next
    # # 主逻辑（改变指针）在后面
    head.next = pre
    dfs(next, head)

dfs(head, None)
```

后续遍历的代码是这样的：

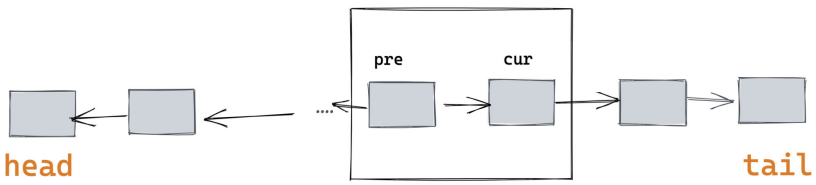
```
def dfs(head):
    if not head or not head.next: return head
    res = dfs(head.next)
    # 主逻辑（改变指针）在进入后面的节点的后面，也就是递归返回的过程会
    head.next.next = head
    head.next = None

    return res
```

可以看出，这两种写法不管是边界，入参，还是代码都不太一样。为什么会有这样的差异呢?

回答这个问题也不难，大家只要记住一个很简单的一句话就好了，那就是**如果是前序遍历，那么你可以想象前面的链表都处理好了，怎么处理的不用管**。相应地**如果是后序遍历，那么你可以想象后面的链表都处理好了，怎么处理的不用管**。这句话的正确性也是毋庸置疑。

如下图，是前序遍历的时候，我们应该画的图。大家把注意力集中在中间的框（子结构）就行了，同时注意两点。

1. 前面的已经处理好了
2. 后面的还没处理好



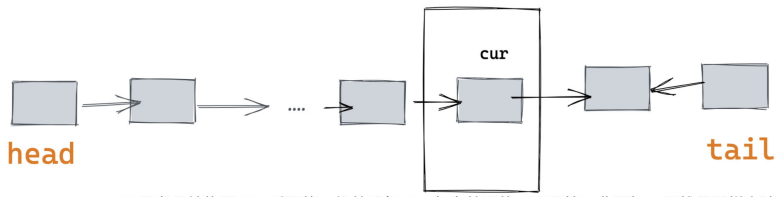
只思考子结构即可，前面的已经处理好了，怎么处理的，不用管。非要问，那就是同样方法

只思考子结构即可，后面的不需考虑如何处理。非要问，那就是用同样方法

据此，我们不难写出以下递归代码，代码注释很详细，大家看注释就好了。

```
def dfs(head, pre):
    if not head: return pre
    # 留下联系方式（由于后面的都没处理，因此可以通过 head.next 定位
    next = head.next
    # 主逻辑（改变指针）在进入后面节点的前面（由于前面的都已经处理好了
    head.next = pre
    dfs(next, head)

dfs(head, None)
```

如果是后序遍历呢？老规矩，秉承我们的一个原则，**先画图**。



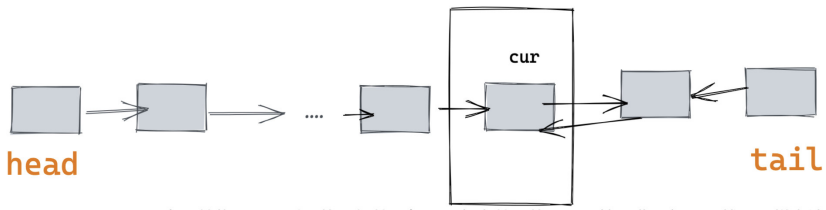
只思考子结构即可，后面的已经处理好了，怎么处理的，不用管。非要问，那就是同样方法

只思考子结构即可，前面的不需考虑如何处理。非要问，那就是用同样方法

不难看出，我们可以通过 head.next 拿到下一个元素，然后将下一个元素的 next 指向自身来完成反转。

用代码表示就是:

```
head.next.next = head
```

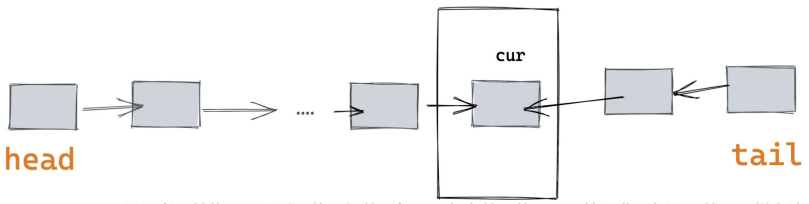


只思考子结构即可，后面的已经处理好了，怎么处理的，不用管。非要问，那就是同样方法

只思考子结构即可，前面的不需考虑如何处理。非要问，那就是用同样方法

画出图之后，是不是很容易看出图中有一个环？ 现在知道画图的好处了吧？就是这么直观，当你很熟练了，就不需要画了，但是在此之前，请不要偷懒。

因此我们需要将 head.next 改为不会造成环的一个值，比如置空。



只思考子结构即可，后面的已经处理好了，怎么处理的，不用管。非要问，那就是同样方法

只思考子结构即可，前面的不需考虑如何处理。非要问，那就是用同样方法

```
def dfs(head):
    if not head or not head.next: return head
    # 不需要留联系方式了，因为我们后面已经走过了，不需走了，现在我们要
    res = dfs(head.next)
    # 主逻辑（改变指针）在进入后面的节点的后面，也就是递归返回的过程会
    head.next.next = head
    # 置空，防止环的产生
    head.next = None

    return res
```

值得注意的是，**前序遍历很容易改造成迭代，因此推荐大家使用前序遍历**。我拿上面的迭代和这里的前序遍历给大家对比一下。

```
cur = head
pre = None

while cur ≠ tail:
    # 留下联系方式
    next = cur.next
    # 修改指针
    cur.next = pre
    # 继续往下走
    pre = cur
    cur = next
```

```
def dfs(head, pre):
    # 相当于迭代的 while cur ≠ tail
    if not head: return pre
    # 留下联系方式
    next = head.next
    # 修改指针
    head.next = pre
    # 继续往下走
    dfs(next, head)
```

那么为什么**前序遍历很容易改造成迭代**呢？实际上，这句话我说的不准确，准确地说应该是**前序遍历容易改成不需要栈的递归，而后续遍历需要借助栈来完成**。这也不难理解，由于后续遍历的主逻辑在函数调用栈的弹出过程，而前序遍历则不需要。

> 这里给大家插播一个写递归的技巧，那就是想象我们已经处理好了一部分数据，并把他们用手挡起来，但是还有一部分等待处理，接下来思考”如何根据已经处理的数据和当前的数据来推导还没有处理的数据“就行了。

四个技巧

针对上面的考点和注意点，我总结了四个技巧来应对，这都是在平时做题中非常实用的技巧。

虚拟头

来了解虚拟头的意义之前，先给大家做几个小测验。

Q1: 如下代码 ans.next 指向什么?

```
ans = ListNode(1)
ans.next = head
head = head.next
head = head.next
```

A1: 最开始的 head。

Q2：如下代码 ans.next 指向什么?

```
ans = ListNode(1)
head = ans
head.next = ListNode(3)
head.next = ListNode(4)
```

A2: ListNode(4)

似乎也不难，我们继续看一道题。

Q3: 如下代码 ans.next 指向什么?

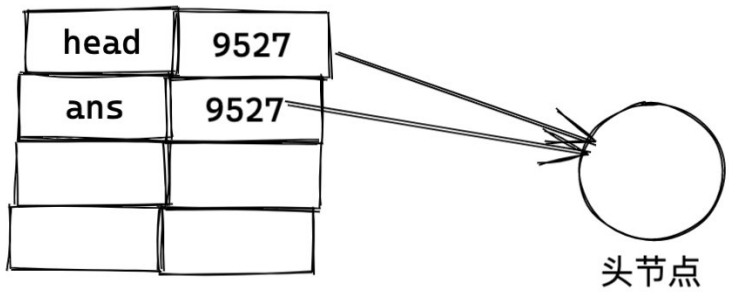
```
ans = ListNode(1)
head = ans
head.next = ListNode(3)
head = ListNode(2)
head.next = ListNode(4)
```

A3: ListNode(3)

如果三道题你都答对了，那么恭喜你，这一部分可以跳过。

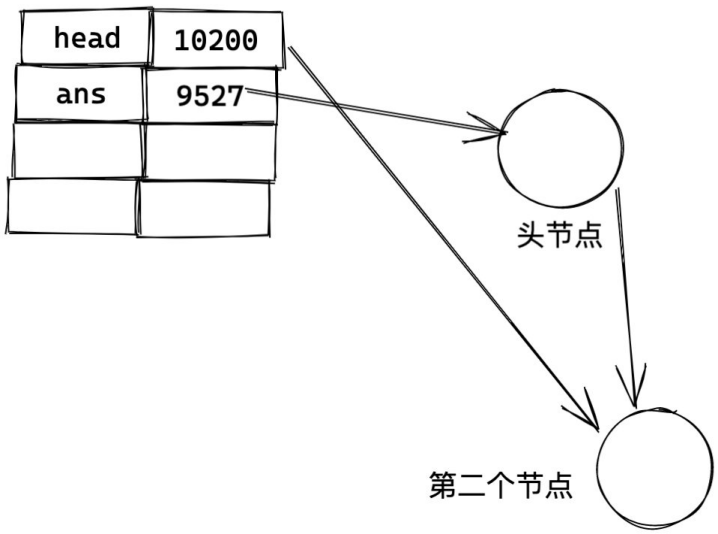
如果你没有懂也没关系，我这里简单解释一下你就懂了。

ans.next 指向什么取决于最后切断 ans.next 指向的地方在哪。比如 Q1，ans.next 指向的是 head，我们假设其指向的内存编号为 `9527` 。



之后执行 `head = head.next` （ans 和 head 被切断联系了），此时的内存图：

> 我们假设头节点的 next 指针指向的节点的内存地址为 10200

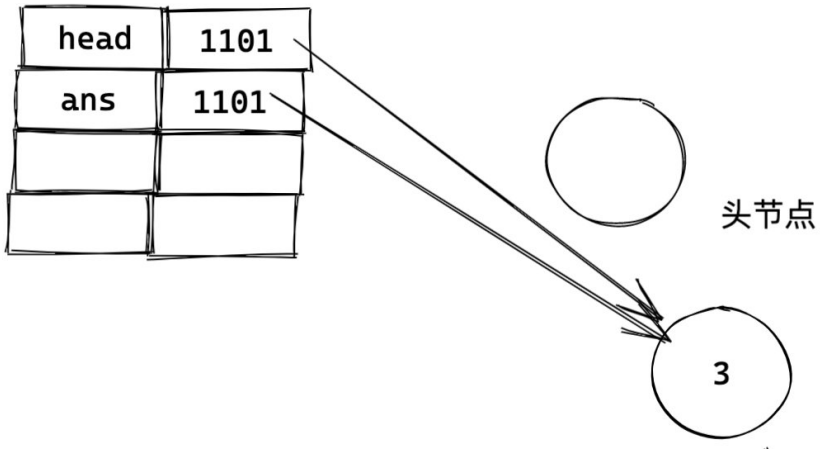


不难看出，ans 没变。

对于第二个例子。一开始和上面例子一样，都是指向 9527。而后执行了：

```
head.next = ListNode(3)
head.next = ListNode(4)
```

ans 和 head 又同时指向 ListNode(3) 了。如图：



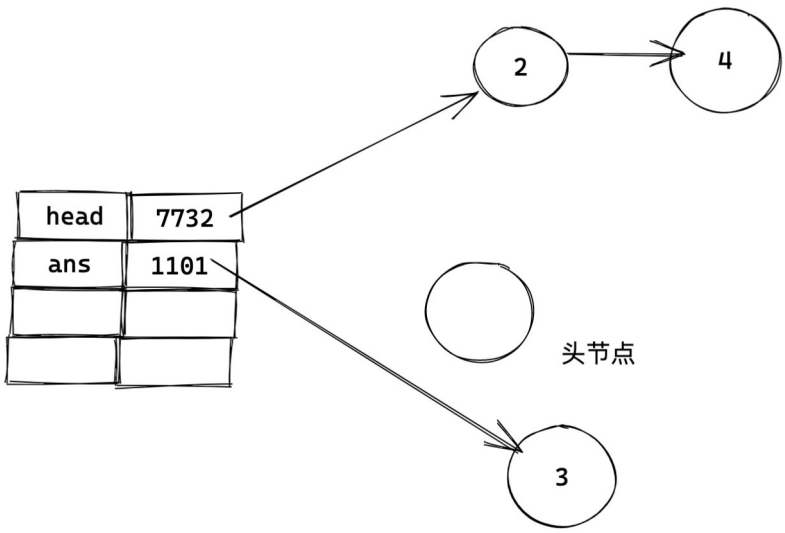
`head.next = ListNode(4)` 也是同理。因此最终的指向 ans.next 是 ListNode(4)。

我们来看最后一个。前半部分和 Q2 是一样的。

```
ans = ListNode(1)
head = ans
head.next = ListNode(3)
```

按照上面的分析，此时 head 和 ans 的 next 都指向 ListNode(3)。关键是下面两行：

```
head = ListNode(2)
head.next = ListNode(4)
```



指向了 `head = ListNode(2)` 之后， head 和 ans 的关系就被切断了，**当前以及之后所有的 head 操作都不会影响到 ans**，因此 ans 还指向被切断前的节点，因此 ans.next 输出的是 ListNode(3)。

花了这么大的篇幅讲这个东西的原因就是，指针操作是链表的核心，如果这些基础不懂， 那么就很难做。接下来，我们介绍主角 - 虚拟头。

相信做过链表的小伙伴都听过这么个名字。为什么它这么好用？它的作用无非就两个：

- 将头节点变成中间节点，简化判断。
- 通过在合适的时候断开链接，返回链表的中间节点。

我上面提到了链表的三个注意，有一个是边界。头节点是最常见的边界，那如果**我们用一个虚拟头指向头节点，虚拟头就是新的头节点了，而虚拟头不是题目给的节点，不参与运算，因此不需要特殊判断**，虚拟头就是这个作用。

如果题目需要返回链表中间的某个节点呢？实际上也可借助虚拟节点。由于我上面提到的指针的操作，实际上，你可以新建一个虚拟头，然后让虚拟头在恰当的时候（刚好指向需要返回的节点）断开连接，这样我们就可以返回虚拟头的 next 就 ok 了。25. K 个一组翻转链表 就用到了这个技巧。

不仅仅是链表， 二叉树等也经常用到这个技巧。 比如我让你返回二叉树的最左下方的节点怎么做？我们也可以利用上面提到的技巧。新建一个虚拟节点，虚拟节点 next 指向当前节点，并跟着一起走，在递归到最左下的时候断开链接，最后返回 虚拟节点的 next 指针即可。

快慢指针

判断链表是否有环，以及环的入口都是使用快慢指针即可解决。这种题就是不知道不会，知道了就不容易忘。不多说了，大家可以参考我之前的题解 https://github.com/azl397985856/leetcode/issues/274#issuecomment-573985706 。

除了这个，求链表的交点也是快慢指针，算法也是类似的。不这都属于不知道就难，知道了就容易。且下次写不容易想不到或者出错。

这部分大家参考我上面的题解理一下， 写一道题就可以掌握。接下来，我们来看下**穿针引线**大法。

另外由于链表不支持随机访问，因此如果想要获取数组中间项和倒数第几项等特定元素就需要一些特殊的手段，而这个手段就是快慢指针。比如要找链表中间项就**搞两个指针，一个大步走（一次走两步），一个小步走（一次走一步）**，这样快指针走到头，慢指针刚好在中间。 如果要求链表倒数第 2 个，那就**让快指针先走一步，慢指针再走**，这样快指针走到

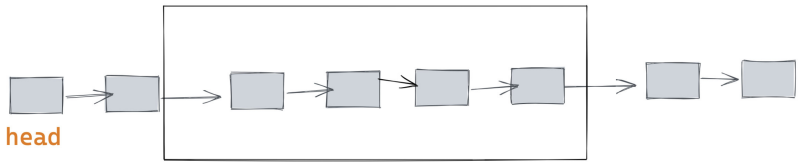
头，慢指针刚好在倒数第二个。这个原理不难理解吧？这种技巧属于**会了就容易，且不容易忘。不会就很难想出的类型**，因此大家学会了拿几道题练一下就可以放下了。

穿针引线

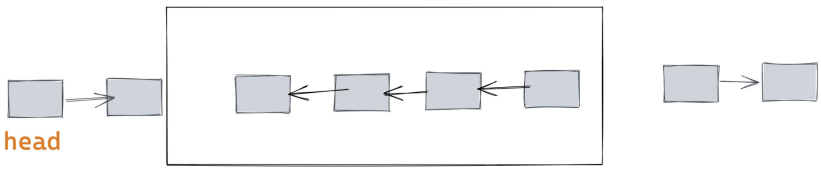
这是链表的第二个考点 - **拼接链表**。我在 25. K 个一组翻转链表，61. 旋转链表 和 92. 反转链表 II 都用了这个方法。穿针引线是我自己起的一个名字，起名字的好处就是方便记忆。

这个方法通常不是最优解，但是好理解，方便书写，不易出错，推荐新手用。

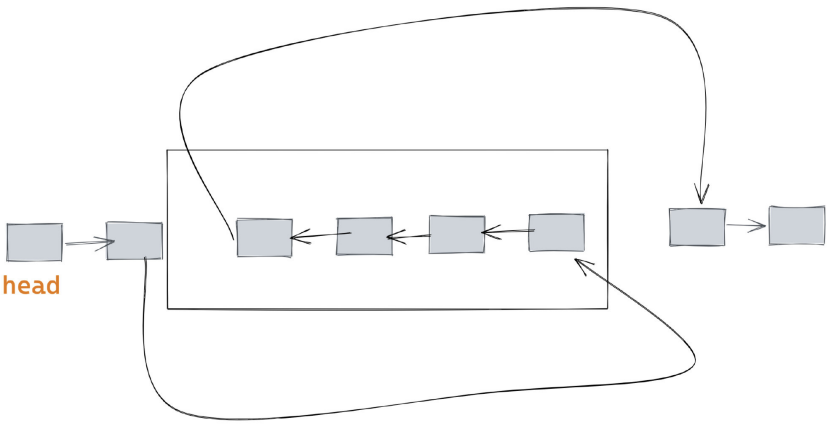
还是以反转链表为例，只不过这次是 `反转链表的中间一部分` ，那我们该怎么做？



反转前面我们已经讲过了，于是我假设链表已经反转好了，那么如何将反转好的链表拼后去呢？



我们想要的效果是这样的：



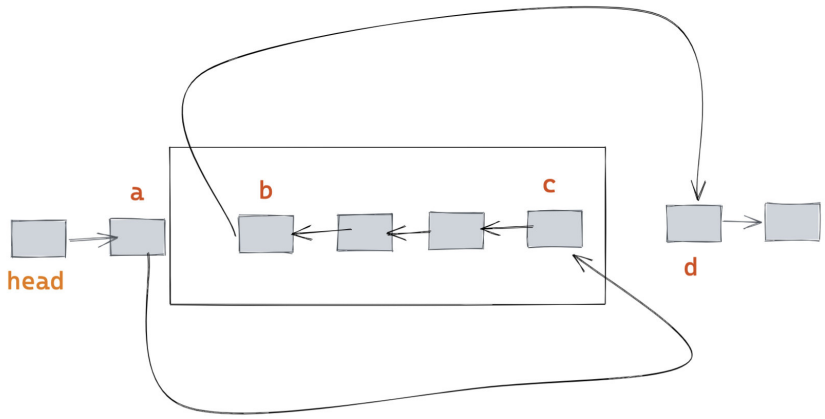
那怎么达到图上的效果呢？我的做法是从做到右给断点编号。如图有两个断点，共涉及到四个节点。于是我给它们依次编号为 a，b，c，d。

其实 a，d 分别是需要反转的链表部分的前驱和后继（不参与反转），而 b 和 c 是需要反转的部分的头和尾（参与反转）。

因此除了 cur， 多用两个指针 pre 和 next 即可找到 a，b，c，d。

找到后就简单了，直接**穿针引线**。

```
a.next = c
b.next = d
```



这不就好了么？我记得的就有 25 题，61 题 和 92 题都是这么做的，清晰不混乱。

先穿再排后判空

这是四个技巧的最后一个技巧了。虽然是最后讲，但并不意味着它不重要。相反，它的实操价值很大。

继续回到上面讲的链表反转题。

```
cur = head
pre = None
while cur != tail:
    # 留下联系方式
    next = cur.next
    # 修改指针
    cur.next = pre
    # 继续往下走
    pre = cur
    cur = next
# 反转后的新的头尾节点返回出去
```

什么时候需要判断 next 是否存在，上面两行代码先写哪个呢？

是这样？

```
    next = cur.next
    cur.next = pre
```

还是这样?

```
    cur.next = pre
    next = cur.next
```

先穿

我给你的建议是：先穿。这里的穿是修改指针，包括反转链表的修改指针和穿针引线的修改指针。**先别管顺序，先穿**。

再排

穿完之后，代码的总数已经确定了，无非就是排列组合让代码没有 bug。

因此第二步考虑顺序，那上面的两行代码哪个在前？应该是先 next = cur.next ，原因在于后一条语句执行后 cur.next 就变了。由于上面代码的作用是反转，那么其实经过 cur.next = pre 之后链表就断开了，后面的都访问不到了，也就是说此时你**只能返回头节点这一个节点**。

实际上，有假如有十行**穿**的代码，我们很多时候没有必要全考虑。我们**需要考虑的仅仅是被改变 next 指针的部分**。比如 cur.next = pre 的 cur 被改了 next。因此下面用到了 cur.next 的地方就要考虑放哪。其他代码不需要考虑。

后判空

和上面的原则类似，穿完之后，代码的总数已经确定了，无非就是看看哪行代码会空指针异常。

和上面的技巧一样，我们很多时候没有必要全考虑。我们**需要考虑的仅仅是被改变 next 指针的部分**。

比如这样的代码

```
while cur:
    cur = cur.next
```

我们考虑 cur 是否为空呢？ 很明显不可能，因为 while 条件保证了，因此不需判空。

那如何是这样的代码呢?

```
while cur:
    next = cur.next
    n_next = next.next
```

如上代码有两个 next，第一个不用判空，上面已经讲了。而第二个是需要的，因为 next 可能是 null。如果 next 是 null ，就会引发空指针异常。因此需要修改为类似这样的代码：

```
while cur:
    next = cur.next
    if not next: break
    n_next = next.next
```

以上就是我们给大家的四个技巧了。相信有了这四个技巧，写链表题就没那么艰难啦~ ^_^

题目推荐

最后推荐几道题给大家，用今天学到的知识解决它们吧~

- 21. 合并两个有序链表
- 82. 删除排序链表中的重复元素 II
- 83. 删除排序链表中的重复元素
- 86. 分隔链表
- 92. 反转链表 II
- 138. 复制带随机指针的链表
- 141. 环形链表
- 142. 环形链表 II
- 143. 重排链表
- 148. 排序链表
- 206. 反转链表
- 234. 回文链表

总结

数组和栈从逻辑上没有大的区别，你看基本操作都是差不多的。如果是单链表，我们无法在 $O(1)$ 的时间拿到前驱节点，这也是为什么我们遍历的时候老是维护一个前驱节点的原因。但是本质原因其实是链表的增删操作都依赖前驱节点。这是链表的基本操作，是链表的特性天生决定的。

可能有的同学有这样的疑问”考点你只讲了指针的修改和链表拼接，难道说链表就只会这些就够了？那我做的题怎么还需要我会前缀和啥的呢？你是不是坑我呢？“

我前面说了，所有的数据结构底层都是数组和链表中的一种或两种。而我们这里讲的链表指的是考察链表的基本操作的题目。因此如果题目中需要你使用归并排序去合并链表，那其实归并排序这部分已经不再本文的讨论范围了。

实际上，你去力扣或者其他 OJ 翻链表题会发现他们的链表题大都指的是入参是链表，且你需要对链表进行一些操作的题目。再比如树的题目大多数是入参是树，你需要在树上进行搜索的题目。也就是说需要操作树（比如修改树的指针）的题目很少，比如有一道题让你给树增加一个 right 指针，指向同级的右侧指针，如果已经是最右侧了，则指向空。

链表的基本操作就是增删查，牢记链表的基本操作和复杂度是解决问题的基本。有了这些基本还不够，大家要牢记我的口诀”一个原则，两个考点，三个注意，四个技巧“。

做链表的题，要想入门，无它，唯画图尔。能画出图，并根据图进行操作你就入门了，甭管你写的代码有没有 bug 。

而链表的题目核心的考察点只有两个，一个是指针操作，典型的就是反转。另外一个是链表的拼接。这两个既是链表的精髓，也是主要考点。

知道了考点肯定不够，我们写代码哪些地方容易犯错？要注意什么？ 这里我列举了三个容易犯错的地方，分别是环，边界和前后序。

其中环指的是节点之间的相互引用，环的题目如果题目本身就有环， 90 % 双指针可以解决，如果本身没有环，那么环就是我们操作指针的时候留下的。如何解决出现环的问题？那就是**画图，然后聚焦子结构，忽略其他信息。**

除了环，另外一个容易犯错的地方往往是边界的条件， 而边界这块链表头的判断又是一个大头。克服这点，我们需要认真读题，看题目的要求以及返回值，另外一个很有用的技巧是虚拟节点。

如果大家用递归去解链表的题， 一定要注意自己写的是前序还是后序。

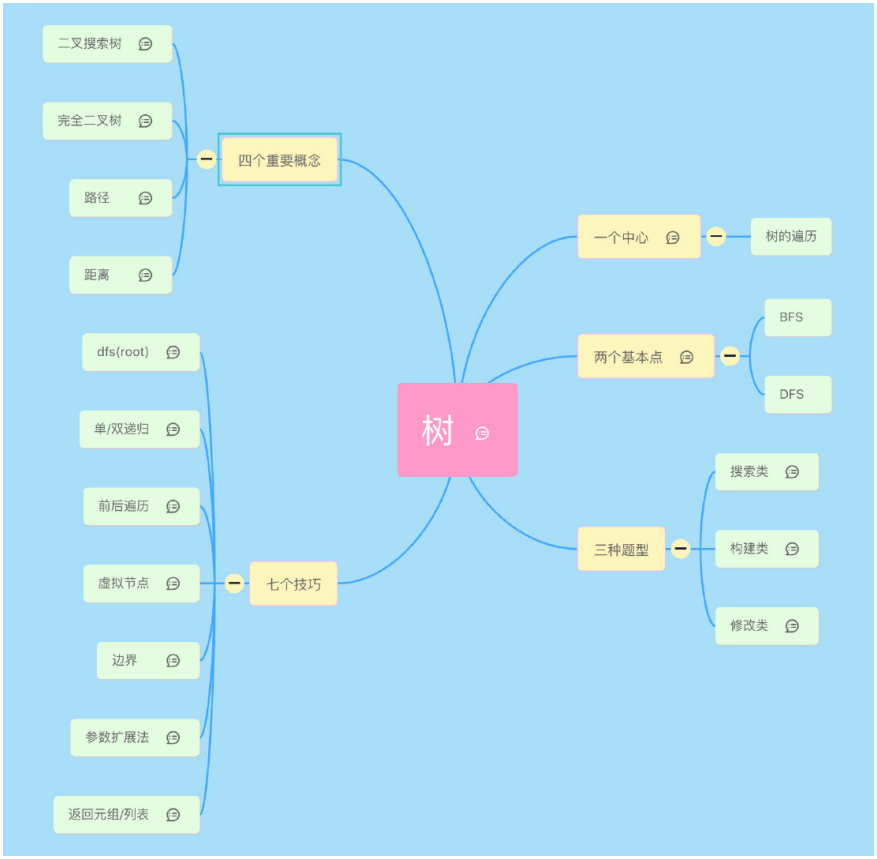
- 如果是前序，那么**只思考子结构即可，前面的已经处理好了，怎么处理的，不用管。非要问，那就是同样方法。后面的也不需考虑如何处理，非要问，那就是用同样方法**
- 如果是后续，那么**只思考子结构即可，后面的已经处理好了，怎么处理的，不用管。非要问，那就是同样方法。前面的不需考虑如何处理。非要问，那就是用同样方法**

如果你想递归和迭代都写， 我推荐你用前序遍历。因为前序遍历容易改成不用栈的递归。

以上就是链表专题的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

我整理的 1000 多页的电子书已经开发下载了，大家可以去我的公众号《力扣加加》后台回复电子书获取。

几乎刷完了力扣所有的树题，我发现了这些东西。。。



先上下本文的提纲，这个是我用 mindmap 画的一个脑图，之后我会继续完善，将其他专题逐步完善起来。

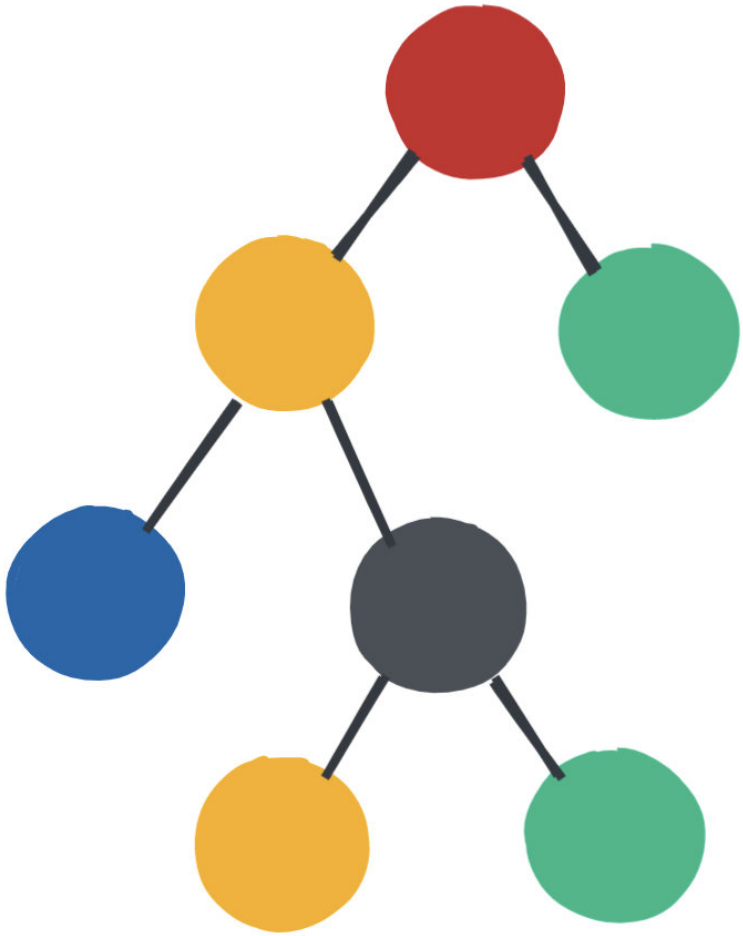
> 大家也可以使用 vscode blink-mind 打开源文件查看，里面有一些笔记可以点开查看。源文件可以去我的公众号《力扣加加》回复脑图获取，以后脑图也会持续更新更多内容。vscode 插件地址：https://marketplace.visualstudio.com/items?itemName=awehook.vscode-blink-mind

本系列包含以下专题：

- 几乎刷完了力扣所有的链表题，我发现了这些东西。。。
- 几乎刷完了力扣所有的树题，我发现了这些东西。。。(就是本文)

一点絮叨

首先亮一下本文的主角 - 树（我的化妆技术还行吧^_^）：



树标签在 leetcode 一共有 **175 道题**。 为了准备这个专题，我花了几天时间将 leetcode 几乎所有的树题目都刷了一遍。

除了 35 个上锁的，1 个不能做的题（1628 题不知道为啥做不了），4 个标着树的标签但却是图的题目，其他我都刷了一遍。通过集中刷这些题，我发现了一些有趣的信息，今天就分享给大家。

食用指南

大家好，我是 lucifer。今天给大家带来的是《树》专题。另外为了保持章节的聚焦性和实用性，省去了一些内容，比如哈夫曼树，前缀树，平衡二叉树（红黑树等），二叉堆。这些内容相对来说实用性没有那么强，如果大家对这些内容也感兴趣，可以关注下我的仓库 leetcode 算法题解，大家有想看的内容也可以留言告诉我哦~

另外要提前告知大家的是本文所讲的很多内容都很依赖于递归。关于递归的练习我推荐大家把递归过程画到纸上，手动代入几次。等大脑熟悉了递归之后就不用这么辛苦了。 实在懒得画图的同学也可以找一个可视化递归的网站，比如 https://recursion.now.sh/。 等你对递归有了一定的理解之后就仔细研究一下树的各种遍历方法，再把本文看完，最后把文章末尾的题目做一做，搞定个递归问题不大。

> 文章的后面《两个基本点 - 深度优先遍历》部分，对于如何练习树的遍历的递归思维我也提出了一种方法

最后要强调的是，本文只是帮助你搞定树题目的常见套路，但不是说树的所有题目涉及的考点都讲。比如树状 DP 这种不在本文的讨论范围，因为这种题更侧重的是 DP，如果你不懂 DP 多半是做不出来的，你需要的是学完树和 DP 之后再去学树状 DP。如果你对这些内容感兴趣，可以期待我的后续专题。

前言

提到树大家更熟悉的是现实中的树，而现实中的树是这样的：

而计算机中的树其实是现实中的树的倒影。

计算机的数据结构是对现实世界物体间关系的一种抽象。比如家族的族谱，公司架构中的人员组织关系，电脑中的文件夹结构，html 渲染的 dom 结构等等，这些有层次关系的结构在计算机领域都叫做树。

首先明确一下，树其实是一种逻辑结构。比如笔者平时写复杂递归的时候，尽管笔者做的题目不是树，也会画一个递归树帮助自己理解。

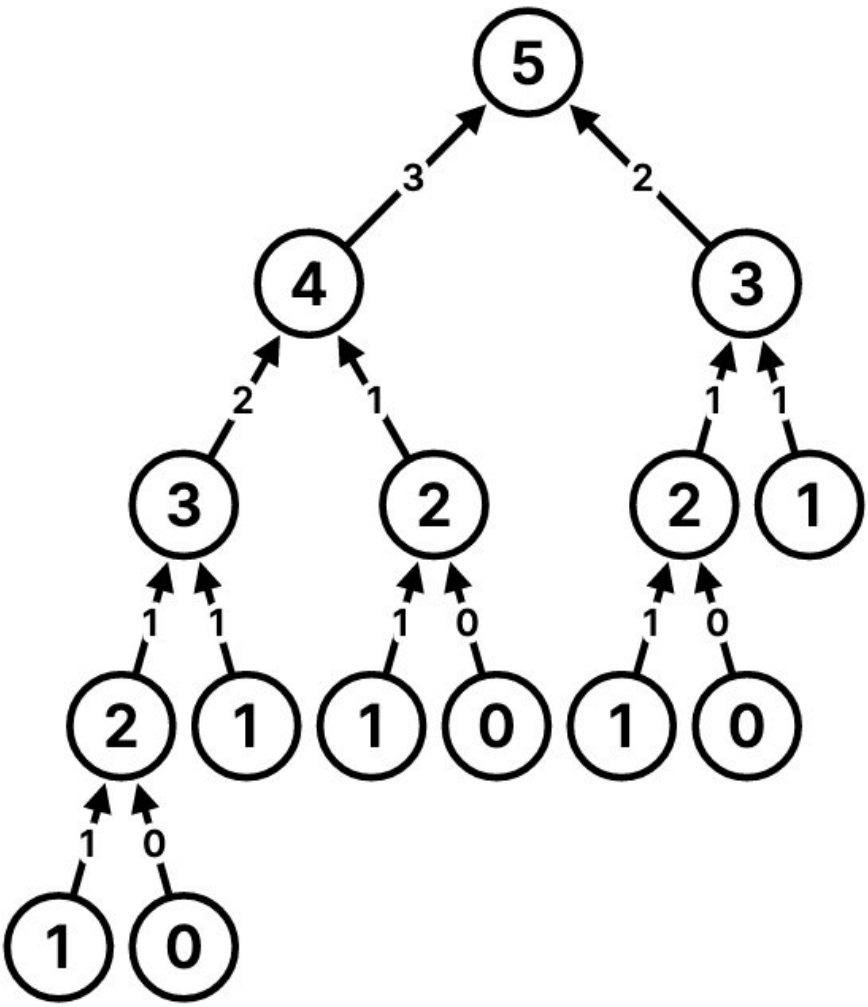
> 树是一种重要的思维工具

以最简单的计算 fibonacci 数列为例：

```
function fn(n) {
  if (n == 0 || n == 1) return n;

  return fn(n - 1) + fn(n - 2);
}
```

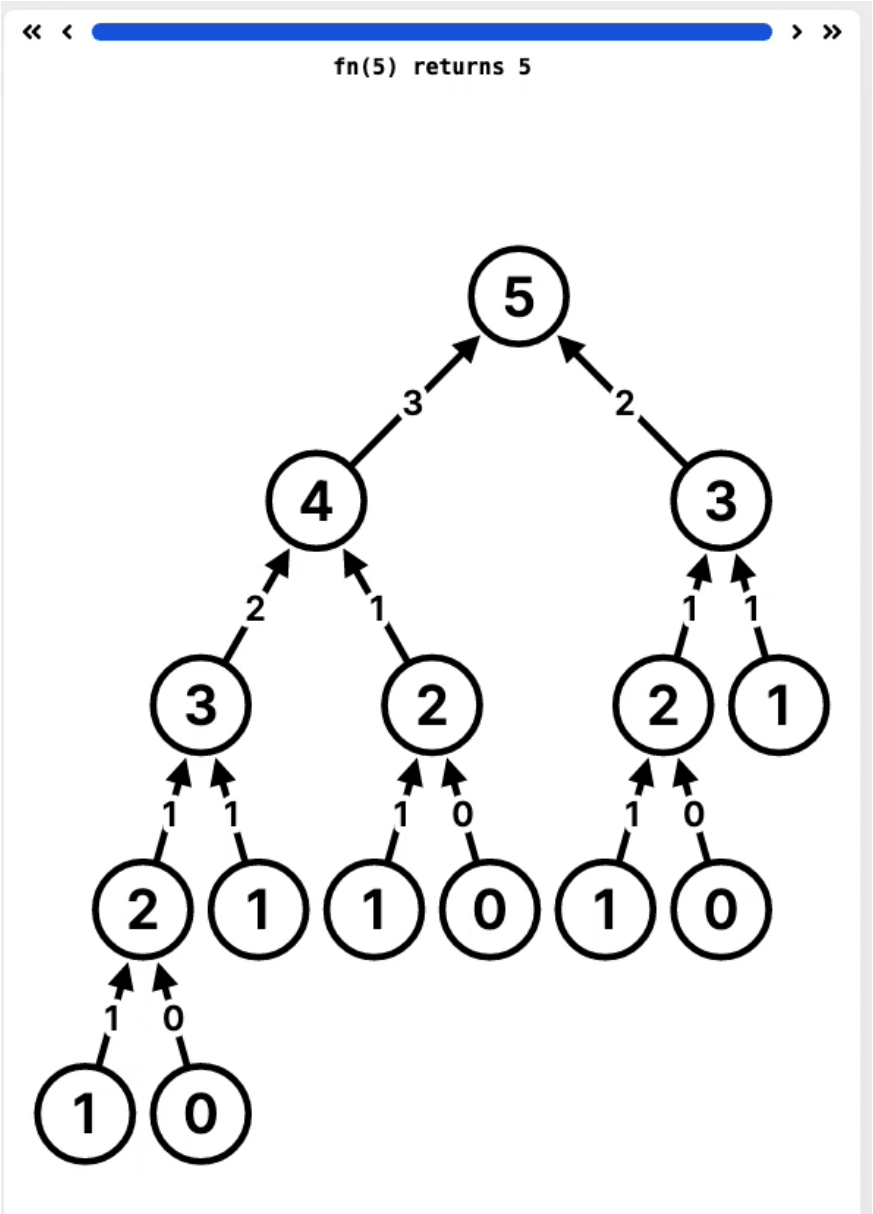
很明显它的入参和返回值都不是树，但是却不影响我们用树的思维去思考。

继续回到上面的代码，根据上面的代码可以画出如下的递归树。



其中树的边表示的是返回值，树节点表示的是需要计算的值，即 fn(n) 。

以计算 5 的 fibbonacci 为例，过程大概是这样的（动图演示）：



这其实就是一个树的后序遍历，你说树（逻辑上的树）是不是很重要？关于后序遍历咱们后面再讲，现在大家知道是这么回事就行。

大家也可以去 这个网站 查看上面算法的单步执行效果。当然这个网站还有更多的算法的动画演示。

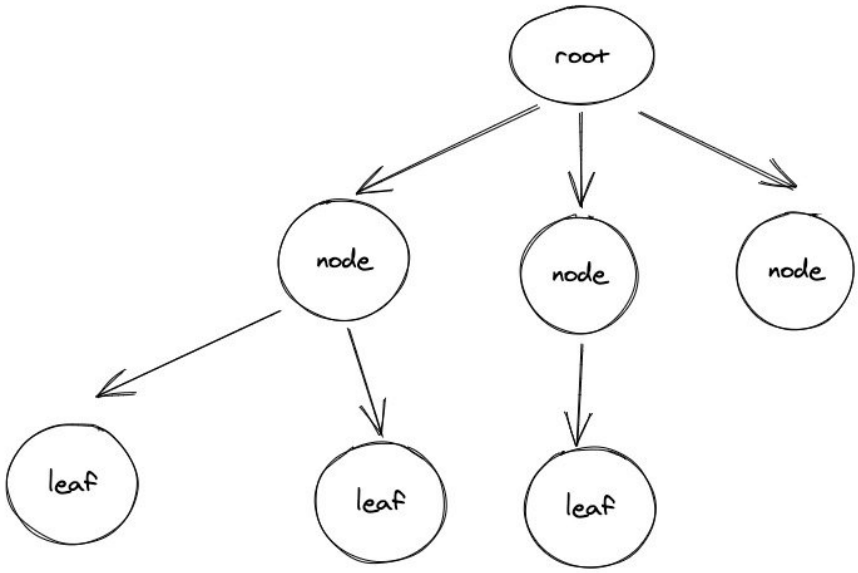
> 上面的图箭头方向是为了方便大家理解。其实箭头方向变成向下的才是真的树结构。

广义的树真的很有用，但是它范围太大了。 本文所讲的树的题目是比较狭隘的树，指的是输入（参数）或者输出（返回值）是树结构的题目。

基本概念

> 树的基本概念难度都不大，为了节省篇幅，我这里简单过一下。对于你不熟悉的点，大家自行去查找一下相关资料。我相信大家也不是来看这些的，大家应该想看一些不一样的东西，比如说一些做题的套路。

树是一种非线性数据结构。树结构的基本单位是节点。节点之间的链接，称为分支（branch）。节点与分支形成树状，结构的开端，称为根（root），或根结点。根节点之外的节点，称为子节点（child）。没有链接到其他子节点的节点，称为叶节点（leaf）。如下图是一个典型的树结构：



每个节点可以用以下数据结构来表示：

```
Node {
    value: any; // 当前节点的值
    children: Array<Node>; // 指向其儿子
}
```

其他重要概念：

- 树的高度：节点到叶子节点的最大值就是其高度。
- 树的深度：高度和深度是相反的，高度是从下往上数，深度是从上往下。因此根节点的深度和叶子节点的高度是 0。
- 树的层：根开始定义，根为第一层，根的孩子为第二层。
- 二叉树，三叉树，。。。 N 叉树，由其子节点最多可以有几个决定，最多有 N 个就是 N 叉树。

二叉树

二叉树是树结构的一种，两个叉就是说每个节点**最多**只有两个子节点，我们习惯称之为左节点和右节点。

> 注意这个只是名字而已，并不是实际位置上的左右

二叉树也是我们做算法题最常见的一种树，因此我们花大篇幅介绍它，大家也要花大量时间重点掌握。

二叉树可以用以下数据结构表示：

```
Node {
    value: any; // 当前节点的值
    left: Node | null; // 左儿子
    right: Node | null; // 右儿子
}
```

二叉树分类

- 完全二叉树
- 满二叉树
- 二叉搜索树
- 平衡二叉树
- 红黑树
- 。。。

二叉树的表示

- 链表存储
- 数组存储。非常适合完全二叉树

树题难度几何?

很多人觉得树是一个很难的专题。实际上，只要你掌握了诀窍，它并没那么难。

从官方的难度标签来看，树的题目处于困难难度的一共是 14 道， 这其中还有 1 个标着树的标签但是却是图的题目，因此困难率是 13 / 175 ，也就是 7.4 % 左右。如果排除上锁的 5 道，困难的只有 9 道。大多数困难题，相信你看完本节的内容，也可以做出来。

从通过率来看，只有**不到三分之一**的题目平均通过率在 50% 以下，其他（绝大多数的题目）通过率都是 50%以上。50% 是一个什么概念呢？这其实很高了。举个例子来说， BFS 的平均通过率差不多在 50%。 而大家认为比较难的二分法和动态规划的平均通过率差不多 40%。

大家不要对树有压力， 树和链表一样是相对容易的专题，今天 lucifer 给大家带来了一个口诀**一个中心，两个基本点，三种题型，四个重要概念，七个技巧**，帮助你克服树这个难关。

一个中心

一个中心指的是**树的遍历**。整个树的专题只有一个中心点，那就是树的遍历，大家务必牢牢记住。

不管是什么题目，核心就是树的遍历，这是一切的基础，不会树的遍历后面讲的都是白搭。

其实树的遍历的本质就是去把树里边儿的每个元素都访问一遍（任何数据结构的遍历不都是如此么？）。但怎么访问的？我不能直接访问叶子节点啊，我必须得从根节点开始访问，然后根据子节点指针访问子节点，但是子节点有多个（二叉树最多两个）方向，所以又有了先访问哪个的问题，这造成了不同的遍历方式。

> 左右子节点的访问顺序通常不重要，极个别情况下会有一些微妙区别。比如说我们想要访问一棵树的最左下角节点，那么顺序就会产生影响，但这种题目会比较少一点。

而遍历不是目的，遍历是为了更好地做处理，这里的处理包括搜索，修改树等。树虽然只能从根开始访问，但是我们可以**选择**在访问完毕回来的时候做处理，还是在访问回来之前做处理，这两种不同的方式就是**后序遍历**和**先序遍历**。

> 关于具体的遍历，后面会给大家详细讲，现在只要知道这些遍历是怎么来的就行了。

而树的遍历又可以分为两个基本类型，分别是深度优先遍历和广度优先遍历。这两种遍历方式并不是树特有的，但却伴随树的所有题目。值得注意的是，这两种遍历方式只是一种逻辑而已，因此理论可以应用于任何数据结构，比如 365. 水壶问题 中，就可以对水壶的状态使用广度优先遍历，而水壶的状态可以用**一个二元组**来表示。

> 遗憾的是这道题的广度优先遍历解法在 LeetCode 上提交会超时

树的遍历迭代写法

很多小朋友表示二叉树前中后序的递归写法没问题，但是迭代就写不出来，问我有什么好的方法没有。

这里就给大家介绍一种写迭代遍历树的实操技巧，统一三种树的遍历方式，包你不会错，这个方法叫做双色标记法。 如果你会了这个技巧，那么你平时练习大可**只用递归**。然后面试的时候，真的要求用迭代或者是对性能有特别要求的那种题目，那你就用我的方法套就行了，下面我来详细讲一下这种方法。

我们知道垃圾回收算法中，有一种算法叫三色标记法。 即：

- 用白色表示尚未访问

- 灰色表示尚未完全访问子节点
- 黑色表示子节点全部访问

那么我们可以模仿其思想，使用双色标记法来统一三种遍历。

其核心思想如下：

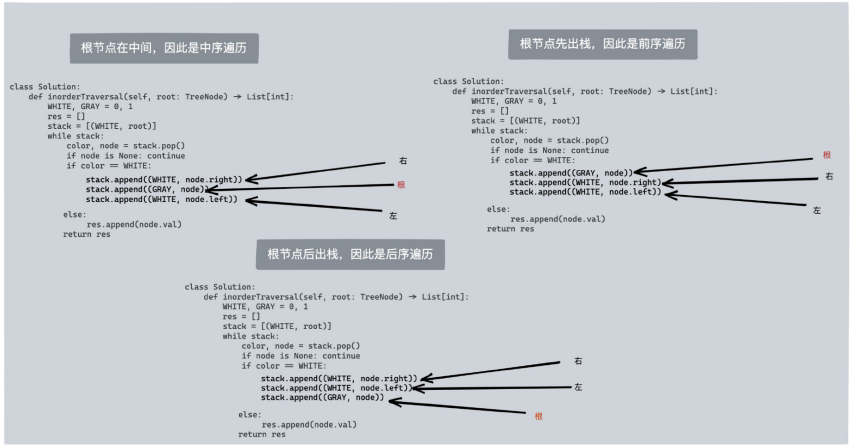
- 使用颜色标记节点的状态，新节点为白色，已访问的节点为灰色。
- 如果遇到的节点为白色，则将其标记为灰色，然后将其右子节点、自身、左子节点依次入栈。
- 如果遇到的节点为灰色，则将节点的值输出。

使用这种方法实现的中序遍历如下：

```
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]
        WHITE, GRAY = 0, 1
        res = []
        stack = [(WHITE, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == WHITE:
                stack.append((WHITE, node.right))
                stack.append((GRAY, node))
                stack.append((WHITE, node.left))
            else:
                res.append(node.val)
        return res
```

可以看出，实现上 WHITE 就表示的是递归中的第一次进入过程，Gray 则表示递归中的从叶子节点返回的过程。 因此这种迭代的写法更接近递归写法的本质。

如要实现前序、后序遍历，也只需要调整左右子节点的入栈顺序即可，其他部分是无需做任何变化。



二色标记法 by lucifer@力扣加加

（前中后序遍历只需要调整这三句话的位置即可）

可以看出使用三色标记法，其写法类似递归的形式，因此便于记忆和书写。

有的同学可能会说，这里的每一个节点都会入栈出栈两次，相比普通的迭代入栈和出栈次数整整加了一倍，这性能可以接受么？我要说的是这种时间和空间的增加仅仅是常数项的增加，大多数情况并不会都程序造成太大的影响。 除了有时候比赛会比较恶心人，会**卡常**（卡常是指通过计算机原理相关的、与理论复杂度无关的方法对代码运行速度进行优化)。反过来看，大家写的代码大多数是递归，要知道递归由于内存栈的开销，性能通常比这里的二色标记法更差才对， 那为啥不用一次入栈的迭代呢？更极端一点，为啥大家不都用 morris 遍历 呢?

> morris 遍历 是可以在常数的空间复杂度完成树的遍历的一种算法。

我认为在大多数情况下，大家对这种细小的差异可以不用太关注。另外如果这种遍历方式完全掌握了，再根据递归的思想去写一次入栈的迭代也不是难事。 无非就是调用函数的时候入栈，函数 return 时候出栈罢了。更多二叉树遍历的内容，大家也可以访问我之前写的专题《二叉树的遍历》。

小结

简单总结一下，树的题目一个中心就是树的遍历。树的遍历分为两种，分别是深度优先遍历和广度优先遍历。关于树的不同深度优先遍历（前序，中序和后序遍历）的迭代写法是大多数人容易犯错的地方，因此我介绍了一种统一三种遍历的方法 - 二色标记法，这样大家以后写迭代的树的前中后序遍历就再也不用怕了。如果大家彻底熟悉了这种写法，再去记忆和练习一次入栈甚至是 Morris 遍历即可。

其实用一次入栈和出栈的迭代实现递归也很简单，无非就是还是用递归思想，只不过你把递归体放到循环里边而已。大家可以在熟悉递归之后再回头看看就容易理解了。树的深度遍历的递归技巧，我们会在后面的《两个

基本点》部分讲解。

两个基本点

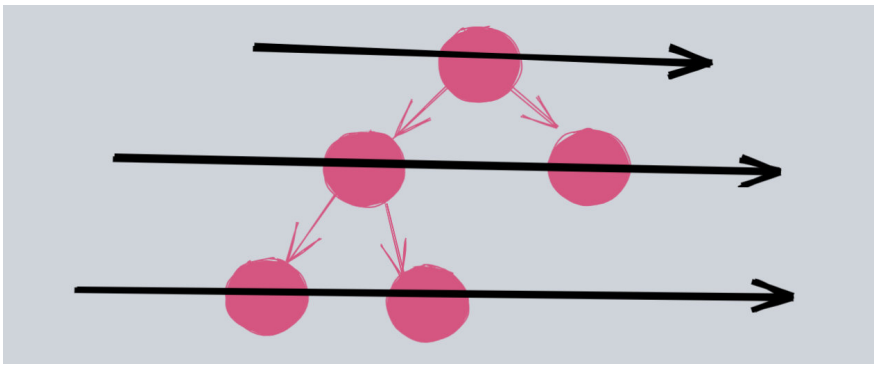
上面提到了树的遍历有两种基本方式，分别是**深度优先遍历（以下简称 DFS）和广度优先遍历（以下简称 BFS），这就是两个基本点**。这两种遍历方式下面又会细分几种方式。比如 **DFS 细分为前中后序遍历， BFS 细分为带层的和不带层的。**

DFS 适合做一些暴力枚举的题目，DFS 如果借助函数调用栈，则可以轻松地使用递归来实现。

BFS 不是 层次遍历

而 BFS 适合求最短距离，这个和层次遍历是不一样的，很多人搞混。这里强调一下，层次遍历和 BFS 是**完全不一样**的东西。

层次遍历就是一层层遍历树，按照树的层次顺序进行访问。



（层次遍历图示）

BFS 的核心在于求最短问题时候可以提前终止，这才是它的核心价值，层次遍历是一种不需要提前终止的 BFS 的副产物。这个提前终止不同于 DFS 的剪枝的提前终止，而是找到最近目标的提前终止。比如我要找距离最近的目标节点，BFS 找到目标节点就可以直接返回。而 DFS 要穷举所有可能才能找到最近的，这才是 BFS 的核心价值。实际上，我们也可以使用 DFS 实现层次遍历的效果，借助于递归，代码甚至会更简单。

> 如果找到任意一个满足条件的节点就好了，不必最近的，那么 DFS 和 BFS 没有太大差别。同时为了书写简单，我通常会选择 DFS。

以上就是两种遍历方式的简单介绍，下面我们对两者进行一个详细的讲解。

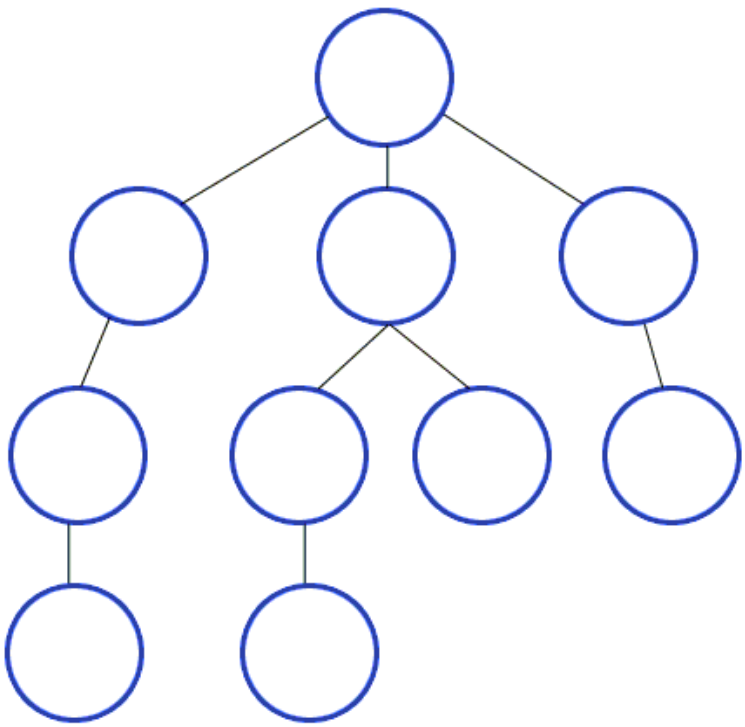
深度优先遍历

深度优先搜索算法（英语：Depth-First-Search，DFS）是一种用于遍历树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点 v 的所在边都己被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止，属于**盲目搜索**。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。因发明「深度优先搜索算法」，约翰·霍普克洛夫特与罗伯特·塔扬在 1986 年共同获得计算机领域的最高奖：图灵奖。

截止目前（2020-02-21），深度优先遍历在 LeetCode 中的题目是 129 道。在 LeetCode 中的题型绝对是超级大户了。而对于树的题目，我们基本上都可以使用 DFS 来解决，甚至我们可以基于 DFS 来做层次遍历，而且由于 DFS 可以基于递归去做，因此算法会更简洁。 在对性能有很高要求的场合，我建议你使用迭代，否则尽量使用递归，不仅写起来简单快速，还不容易出错。

DFS 图解：



(图片来自 https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/depth-first-search)

算法流程

1. 首先将根节点放入**stack**中。
2. 从*stack*中取出第一个节点，并检验它是否为目标。如果找到所有的节点，则结束搜寻并回传结果。否则将它某一个尚未检验过的直接子节点加入**stack**中。
3. 重复步骤 2。
4. 如果不存在未检测过的直接子节点。将上一级节点加入**stack**中。 重复步骤 2。
5. 重复步骤 4。
6. 若**stack**为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。

这里的 stack 可以理解为自己实现的栈，也可以理解为调用栈。如果是调用栈的时候就是递归，如果是自己实现的栈的话就是迭代。

算法模板

一个典型的通用的 DFS 模板可能是这样的：

```
const visited = {}
function dfs(i) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }

    visited[i] = true // 将当前状态标为已搜索
    for (根据i能到达的下个状态j) {
        if (!visited[j]) { // 如果状态j没有被搜索过
            dfs(j)
        }
    }
}
```

上面的 visited 是为了防止由于环的存在造成的死循环的。 而我们知道树是不存在环的，因此树的题目大多数不需要 visited，除非你对树的结构做了修改，比如就左子树的 left 指针指向自身，此时会有环。再比如 138. 复制带随机指针的链表 这道题需要记录已经复制的节点，这些需要记录 visited 信息的树的题目**少之又少**。

因此一个树的 DFS 更多是：

```
function dfs(root) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }
    for (const child of root.children) {
        dfs(child)
    }
}
```

而几乎所有的题目几乎都是二叉树，因此下面这个模板更常见。

```
function dfs(root) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }
    dfs(root.left)
    dfs(root.right)
}
```

而我们不同的题目除了 if (满足特定条件部分不同之外)，还会写一些特有的逻辑，这些逻辑写的位置不同，效果也截然不同。那么位置不同会有什么影响，什么时候应该写哪里呢？接下来，我们就聊聊两种常见的 DFS 方式。

两种常见分类

前序遍历和后序遍历是最常见的两种 DFS 方式。而另外一种遍历方式（中序遍历）一般用于平衡二叉树，这个我们后面的**四个重要概念**部分再讲。

前序遍历

如果你的代码大概是这么写的（注意主要逻辑的位置）：

```
function dfs(root) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }
    // 主要逻辑
    dfs(root.left)
    dfs(root.right)
}
```

那么此时我们称为前序遍历。

后续遍历

而如果你的代码大概是这么写的（注意主要逻辑的位置）：

```
function dfs(root) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }
    dfs(root.left)
    dfs(root.right)
    // 主要逻辑
}
```

那么此时我们称为后序遍历。

值得注意的是， 我们有时也会会写出这样的代码：

```
function dfs(root) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }
    // 做一些事
    dfs(root.left)
    dfs(root.right)
    // 做另外的事
}
```

如上代码，我们在进入和退出左右子树的时候分别执行了一些代码。那么这个时候，是前序遍历还是后续遍历呢？实际上，这属于混合遍历了。不过我们这里只考虑**主逻辑**的位置，关键词是**主逻辑**。

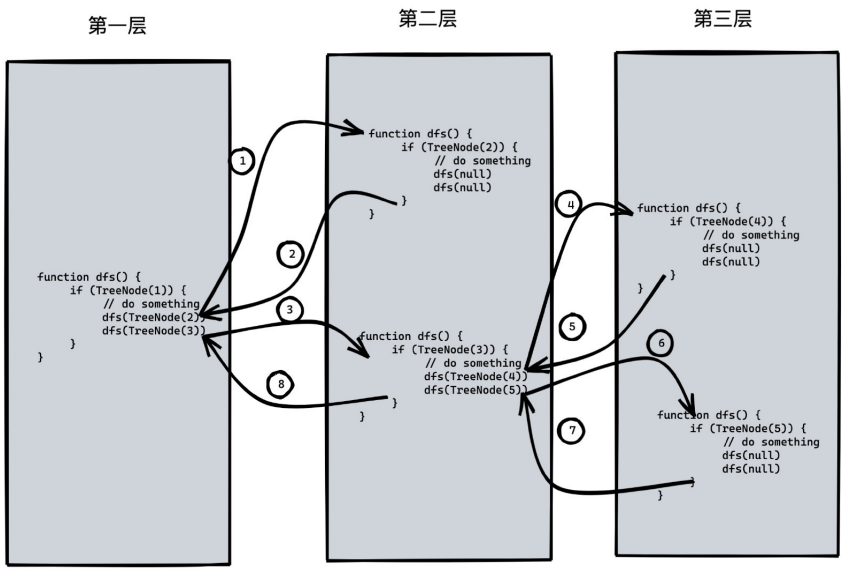
如果代码主逻辑在左右子树之前执行，那么就是前序遍历。如果代码主逻辑在左右子树之后执行，那么就是后序遍历。关于更详细的内容， 我会在**七个技巧** 中的**前后遍历**部分讲解，大家先留个印象，知道有着两种方式就好。

递归遍历的学习技巧

上面的《一个中心》部分，给大家介绍了一种干货技巧《双色遍历》统一三种遍历的迭代写法。 而树的遍历的递归的写法其实大多数人都没问题。为什么递归写的没问题，用栈写迭代就有问题呢？本质上其实还是对递归的理解不够。那 lucifer 今天给大家介绍一种练习递归的技巧。其实文章开头也提到了，那就是画图 + 手动代入。有的同学不知道怎么画，这里我抛砖引玉分享一下我学习递归的画法。

比如我们要前序遍历一棵这样的树：

```
    1
   / \
  2   3
     / \
    4   5
```



图画的还算比较清楚， 就不多解释了。大家遇到题目多画几次这样的递归图，慢慢就对递归有感觉了。

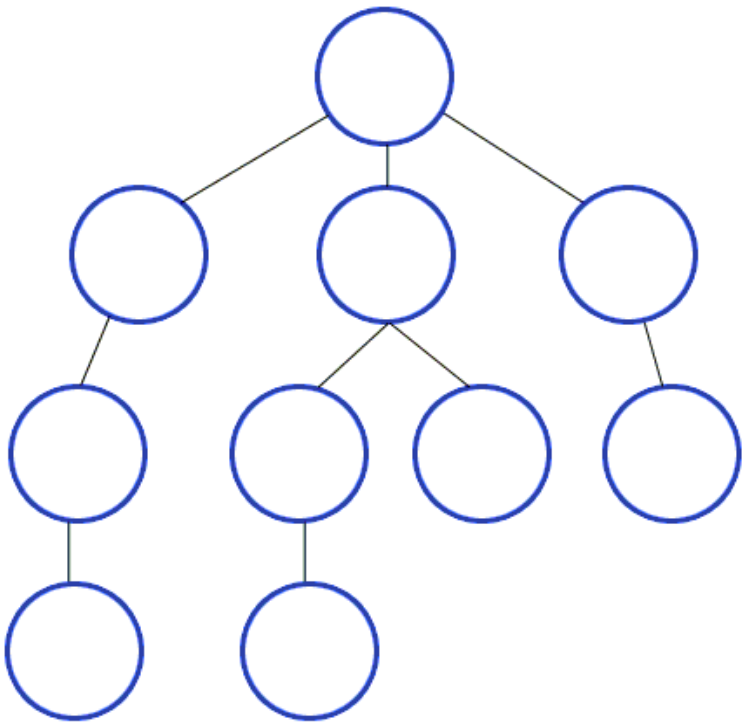
广度优先遍历

树的遍历的两种方式分别是 DFS 和 BFS，刚才的 DFS 我们简单过了一下前序和后序遍历，对它们有了一个简单印象。这一小节，我们来看下树的另外一种遍历方式 - BFS。

BFS 也是图论中算法的一种，不同于 DFS， BFS 采用横向搜索的方式，在数据结构上通常采用队列结构。 注意，DFS 我们借助的是栈来完成，而这里借助的是队列。

BFS 比较适合找**最短距离/路径**和**某一个距离的目标**。比如 `给定一个二叉树，在树的最后一行找到最左边的值。` ，此题是力扣 513 的原题。这不就是求距离根节点**最远距离**的目标么？ 一个 BFS 模板就解决了。

BFS 图解：



(图片来自 https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/breadth-first-search)

算法流程

1. 首先将根节点放入队列中。
2. 从队列中取出第一个节点，并检验它是否为目标。
    - 如果找到目标，则结束搜索并回传结果。
    - 否则将它所有尚未检验过的直接子节点加入队列中。
3. 若队列为空，表示整张图都检查过了——亦即图中没有欲搜索的目标。结束搜索并回传“找不到目标”。
4. 重复步骤 2。

算法模板

```
const visited = {}
function bfs() {
    let q = new Queue()
    q.push(初始状态)
    while(q.length) {
        let i = q.pop()
        if (visited[i]) continue
        if (i 是我们要找的目标) return 结果
        for (i的可抵达状态j) {
            if (j 合法) {
                q.push(j)
            }
        }
    }
    return 没找到
}
```

两种常见分类

BFS 我目前使用的模板就两种，这两个模板可以解决所有的树的 BFS 问题。

前面我提到了“BFS 比较适合找**最短距离/路径**和**某一个距离的目标**”。 如果我需要求的是最短距离/路径，我是不关心我走到第几步的，这个时候可是用不标记层的目标。而如果我需要求距离某个节点距离等于 k 的所有节点，这个时候第几步这个信息就值得被记录了。

> 小于 k 或者 大于 k 也是同理。

标记层

一个常见的 BFS 模板，代入题目只需要根据题目微调即可。

```
class Solution:
    def bfs(k):
        # 使用双端队列，而不是数组。因为数组从头部删除元素的时间复杂
        queue = collections.deque([root])
        # 记录层数
        steps = 0
        # 需要返回的节点
        ans = []
        # 队列不空，生命不止！
        while queue:
            size = len(queue)
            # 遍历当前层的所有节点
            for _ in range(size):
                node = queue.popleft()
                if (step == k) ans.append(node)
                if node.right:
                    queue.append(node.right)
                if node.left:
                    queue.append(node.left)
            # 遍历完当前层所有的节点后 steps + 1
            steps += 1
        return ans
```

不标记层

不带层的模板更简单，因此大家其实只需要掌握带层信息的目标就够了。

一个常见的 BFS 模板，代入题目只需要根据题目微调即可。

```
class Solution:
    def bfs(k):
        # 使用双端队列，而不是数组。因为数组从头部删除元素的时间复杂
        queue = collections.deque([root])
        # 队列不空，生命不止！
        while queue:
            node = queue.popleft()
            # 由于没有记录 steps，因此我们肯定是不需要根据层的信息
            if (node 是我们要找到的) return node
            if node.right:
                queue.append(node.right)
            if node.left:
                queue.append(node.left)
        return -1
```

以上就是 BFS 的两种基本方式，即带层和不带层，具体使用哪种看题目是否需要根据层信息做判断即可。

小结

树的遍历是后面所有内容的基础，而树的遍历的两种方式 DFS 和 BFS 到这里就简单告一段落，现在大家只要知道 DFS 和 BFS 分别有两种常见的方式就够了，后面我会给大家详细补充。

三种题型

树的题目就三种类型，分别是：**搜索类，构建类和修改类，而这三类题型的比例也是逐渐降低的**，即搜索类的题目最多，其次是构建类，最后是修改类。这一点和链表有很大的不同，链表更多的是修改类。

接下来，lucifer 给大家逐一讲解这三种题型。

搜索类

搜索类的题目是树的题目的绝对大头。而搜索类只有两种解法，那就是 DFS 和 BFS，下面分别介绍。

几乎所有的搜索类题目都可以方便地使用递归来实现，关于递归的技巧会在**七个技巧中的单/双递归**部分讲解。还有一小部分使用递归不好实现，我们可以使用 BFS，借助队列轻松实现，比如最经典的是求二叉树任意两点的距离，树的距离其实就是最短距离，因此可以用 BFS 模板解决。这也是为啥我说**DFS 和 BFS**是树的题目的两个基本点的原因。

所有搜索类的题目只要把握三个核心点，即**开始点**，**结束点** 和 **目标**即可。

DFS 搜索

DFS 搜索类的基本套路就是从入口开始做 dfs，然后在 dfs 内部判断是否是结束点，这个结束点通常是**叶子节点**或**空节点**，关于结束这个话题我们放在**七个技巧中的边界**部分介绍，如果目标是一个基本值（比如数字）直接返回或者使用一个全局变量记录即可，如果是一个数组，则可以通过扩展参数的技巧来完成，关于扩展参数，会在**七个技巧中的参数扩展**部分介绍。 这基本就是搜索问题的全部了，当你读完后面的七个技巧，回头再回来看这个会更清晰。

套路模板：

```
# 其中 path 是树的路径， 如果需要就带上，不需要就不带
def dfs(root, path):
    # 空节点
    if not root: return
    # 叶子节点
    if not root.left and not root.right: return
    path.append(root)
    # 逻辑可以写这里，此时是前序遍历
    dfs(root.left)
    dfs(root.right)
    # 需要弹出，不然会错误计算。
    # 比如对于如下树：
    """
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
    """
    # 如果不 pop，那么 5 -> 4 -> 11 -> 2 这条路径会变成 5 -> 4

    path.pop()
    # 逻辑也可以写这里，此时是后序遍历

    return 你想返回的数据
```

比如剑指 Offer 34. 二叉树中和为某一值的路径 这道题，题目是：输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。 这不就是从根节点开始，到叶子节点结束的所有路径**搜索出来**，挑选出和为目标值的路径么？这里的开始点是根节点， 结束点是叶子节点，目标就是路径。

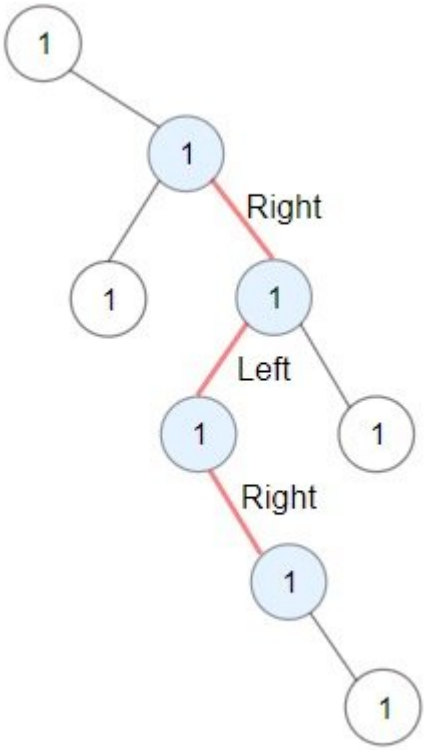
对于求这种满足**特定和**的题目，我们都可以方便地使用**前序遍历 + 参数扩展的形式**，关于这个，我会在**七个技巧中的前后序部分**展开。

> 由于需要找到所有的路径，而不仅仅是一条，因此这里适合使用回溯暴力枚举。关于回溯，可以参考我的 回溯专题

```
class Solution:
    def pathSum(self, root: TreeNode, target: int) -> List
        def backtrack(nodes, path, cur, remain):
            # 空节点
            if not cur: return
            # 叶子节点
            if cur and not cur.left and not cur.right:
                if remain == cur.val:
                    res.append((path + [cur.val]).copy())
                return
            # 选择
            path.append(cur.val)
            # 递归左右子树
            backtrack(nodes, path, cur.left, remain - cur.v
            backtrack(nodes, path, cur.right, remain - cur.
            # 撤销选择
            path.pop(-1)
        ans = []
        # 入口，路径，目标值全部传进去，其中路径和path都是扩展的参数
        backtrack(ans, [], root, target)
        return ans
```

再比如：1372. 二叉树中的最长交错路径，题目描述：

```
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
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。

比如：
```




```
此时需要返回 3
解释：蓝色节点为树中最长交错路径（右 -> 左 -> 右）。
```

这不就是从任意节点**开始**，到任意节点**结束**的所有交错**路径**全部**搜索出来**，挑选出最长的么？这里的开始点是树中的任意节点，结束点也是任意节点，目标就是最长的交错路径。

对于入口是任意节点的题目，我们都可以方便地使用**双递归**来完成，关于这个，我会在**七个技巧中的单/双递归部分**展开。

对于这种交错类的题目，一个好用的技巧是使用 -1 和 1 来记录方向，这样我们就可以通过乘以 -1 得到另外一个方向。

> 886. 可能的二分法 和 785. 判断二分图 都用了这个技巧。

用代码表示就是：

```
next_direction = cur_direction * -1
```

这里我们使用双递归即可解决。 如果题目限定了只从根节点开始，那就可以用单递归解决了。值得注意的是，这里内部递归需要 cache 一下，不然容易因为重复计算导致超时。

> 我的代码是 Python，这里的 lru_cache 就是一个缓存，大家可以使用自己语言的字典模拟实现。

```
class Solution:
    @lru_cache(None)
    def dfs(self, root, dir):
        if not root:
            return 0
        if dir == -1:
            return int(root.left != None) + self.dfs(root.
        return int(root.right != None) + self.dfs(root.righ

    def longestZigZag(self, root: TreeNode) -> int:
        if not root:
            return 0
        return max(self.dfs(root, 1), self.dfs(root, -1), s
```

这个代码不懂没关系，大家只有知道搜索类题目的大方向即可，具体做法我们后面会介绍，大家留个印象就行。更多的题目以及这些技巧的详细使用方式放在**七个技巧部分**展开。

BFS 搜索

这种类型相比 DFS，题目数量明显降低，套路也少很多。题目大多是求距离，套用我上面的两种 BFS 模板基本都可以轻松解决，这个不多介绍了。

构建类

除了搜索类，另外一个大头是构建类。构建类又分为两种：普通二叉树的构建和二叉搜索树的构建。

普通二叉树的构建

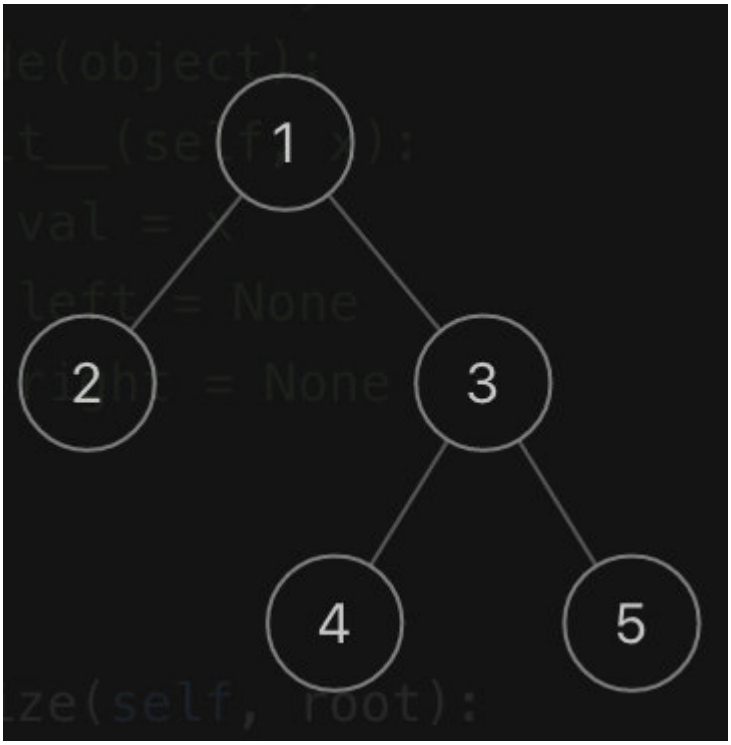
而普通二叉树的构建又分为三种：

1. 给你两种 DFS 的遍历的结果数组，让你构建出原始的树结构。比如根据先序遍历和后序遍历的数组，构造原始二叉树。这种题我在构造二叉树系列 系列里讲的很清楚了，大家可以去看看。

> 这种题目假设输入的遍历的序列中都不含重复的数字，想想这是为什么。

1. 给你一个 BFS 的遍历的结果数组，让你构建出原始的树结构。

最经典的就是 剑指 Offer 37. 序列化二叉树。我们知道力扣的所有的树表示都是使用数字来表示的，而这个数组就是一棵树的层次遍历结果，部分叶子节点的子节点（空节点）也会被打印。比如：[1,2,3,null,null,4,5]，就表示的是如下的一颗二叉树：



我们是如何根据这样的一个层次遍历结果构造出原始二叉树的呢？这其实就属于构造二叉树的内容，这个类型目前力扣就这一道题。这道题如果你彻底理解 BFS，那么就难不倒你。

1. 还有一种是给你描述一种场景，让你构造一个符合条件的二叉树。这种题和上面的没啥区别，套路简直不要太像，比如 654. 最大二叉树，我就不多说了，大家通过这道题练习一下就知道了。

除了这种静态构建，还有一种很很罕见的动态构建二叉树的，比如 894. 所有可能的满二叉树 ,对于这个题，直接 BFS 就好了。由于这种题很少，因此不做多的介绍。大家只要把最核心的掌握了，这种东西自然水到渠成。

二叉搜索树的构建

普通二叉树无法根据一种序列重构的原因是只知道根节点，无法区分左右子树。如果是二叉搜索树，那么就有可能根据**一种遍历序列**构造出来。原因就在于二叉搜索树的根节点的值大于所有的左子树的值，且小于所有的右子树的值。因此我们可以根据这一特性去确定左右子树的位置，经过这样的转换就和上面的普通二叉树没有啥区别了。比如 1008. 前序遍历构造二叉搜索树

修改类

上面介绍了两种常见的题型：搜索类和构建类。还有一种比例相对比较小的题目类型是修改类。

> 当然修改类的题目也是要基于搜索算法的，不找到目标怎么删呢?

修改类的题目有两种基本类型。

题目要求的修改

一种是题目让你增加，删除节点，或者是修改节点的值或者指向。

修改指针的题目一般不难，比如 116. 填充每个节点的下一个右侧节点指针，这不就是 BFS 的时候顺便记录一下上一次访问的同层节点，然后增加一个指针不就行了么？关于 BFS ，套用我的**带层的 BFS 模板**就搞定了。

增加和删除的题目一般稍微复杂，比如 450. 删除二叉搜索树中的节点 和 669. 修剪二叉搜索树。西法我教你两个套路，面对这种问题就不带怕的。那就是**后续遍历 + 虚拟节点**，这两个技巧同样放在后面的七个技巧部分讲解。是不是对七个技巧很期待？^_^

> 实际工程中，我们也可以不删除节点，而是给节点做一个标记，表示已经被删除了，这叫做软删除。

算法需要，自己修改

另外一种是为了方便计算，自己加了一个指针。

比如 863. 二叉树中所有距离为 K 的结点 通过修改树的节点类，增加一个指向父节点的引用 parent，问题就转化为距离目标节点一定距离的问题了，此时可是用我上面讲的**带层的 BFS 模板**解决。

动态语言可以直接加属性（比如上面的 parent），而静态语言是不允许的，因此你需要增加一个新的类定义。不过你也可以使用字典来实现，key 是 node 引用， value 是你想记录的东西，比如这里的 parent 节点。

比如对于 Java 来说，我们可以：

```
class Solution {
    Map<TreeNode, TreeNode> parent;
    public void dfs(TreeNode node, TreeNode parent) {
        if (node != null) {
            parent.put(node, parent);
            dfs(node.left, node);
            dfs(node.right, node);
        }
    }
}
```

简单回顾一下这一小节的知识。

接下来是做树的题目不得不知的四个重要概念。

四个重要概念

二叉搜索树

二叉搜索树（Binary Search Tree），亦称二叉查找树。

二叉搜索树具有下列性质的二叉树：

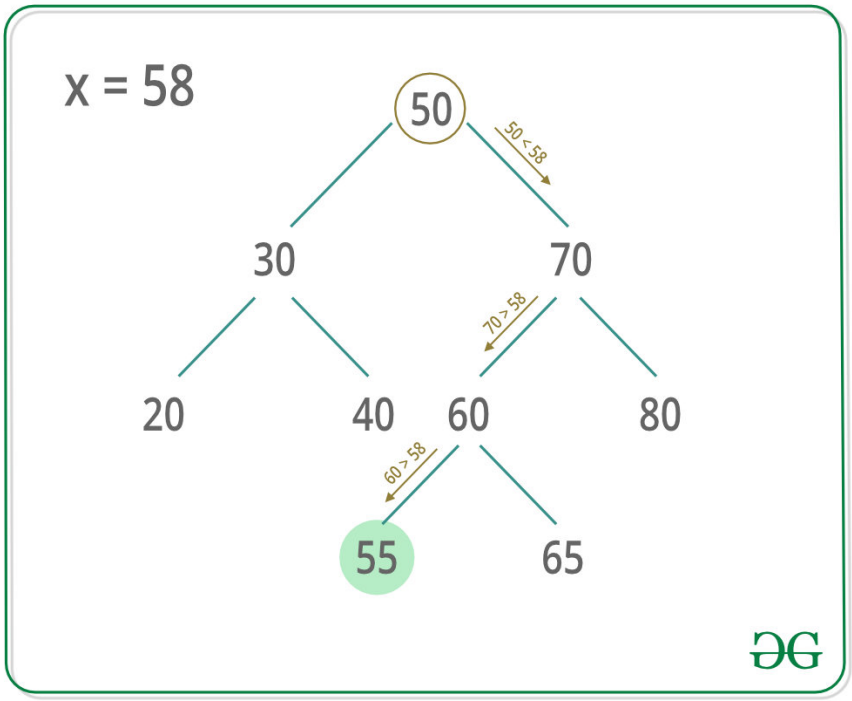
- 若左子树不空，则左子树上所有节点的值均小于它的根节点的值；
- 若右子树不空，则右子树上所有节点的值均大于它的根节点的值；
- 左、右子树也分别为二叉排序树；
- 没有键值相等的节点。

对于一个二叉查找树，常规操作有 `插入，查找，删除，找父节点，求最大值，求最小值。`

天生适合查找

二叉查找树，之所以叫查找树就是因为其非常适合查找。

举个例子，如下一颗二叉查找树，我们想找节点值小于且最接近 58 的节点，搜索的流程如图所示：



（图片来自 https://www.geeksforgeeks.org/floor-in-binary-search-tree-bst/）

可以看出每次向下走，都会排除了一个分支，如果一颗二叉搜索树同时也是一颗二叉平衡树的话，那么其搜索过程时间复杂度就是 $O(logN)$。实际上，**平衡二叉搜索树的查找和有序数组的二分查找本质都是一样的，只是数据的存储方式不同罢了**。那为什么有了有序数组二分，还需要二叉搜索树呢？原因在于树的结构对于动态数据比较友好，比如数据是频繁变动的，比如经常添加和删除，那么就可以使用二叉搜索树。理论上添加和删除的时间复杂度都是 $O(h)$，其中 h 为树的高度，如果是一颗平衡二叉搜索树，那么时间复杂度就是 $O(logN)$。而数组的添加和删除的时间复杂度为 $O(N)$，其中 N 为数组长度。

方便搜索，是二叉搜索树核心的设计初衷。不让查找算法时间复杂度退化到线性是平衡二叉树的初衷。

我们平时说的二分很多是数组的二分，因为数组可以随机访问嘛。不过这种二分实在太狭义了，二分的本质是将问题规模缩小到一半，因此二分和数据结构没有本质关系，但是不同的数据结构却给二分赋予了不同的色彩。比如跳表就是链表的二分，二叉搜索树就是树的二分等。随着大家对算法和数据结构的了解的加深，会发现更多有意思的东西^_^

中序遍历是有序的

另外二叉查找树有一个性质，这个性质对于做题很多帮助，那就是： **二叉搜索树的中序遍历的结果是一个有序数组**。 比如 98. 验证二叉搜索树 就可以直接中序遍历，并**一边遍历一边判断遍历结果是否是单调递增的**，如果不是则提前返回 False 即可。

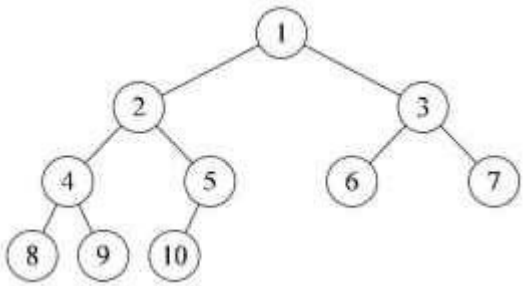
再比如 99. 恢复二叉搜索树，官方难度为困难。题目大意是 `给你二叉搜索树的根节点 root ，该树中的两个节点被错误地交换。请在不改变其结构的情况下，恢复这棵树。` 我们可以先中序遍历发现不是递增的节点，他们就是被错误交换的节点，然后交换恢复即可。这道题难点就在于一点，即错误交换可能错误交换了中序遍历的相邻节点或者中序遍历的非相邻节点，这是两种 case，需要分别讨论。

类似的题目很多，不再赘述。大家如果**碰到二叉搜索树的搜索类题目，一定先想下能不能利用这个性质来做。**

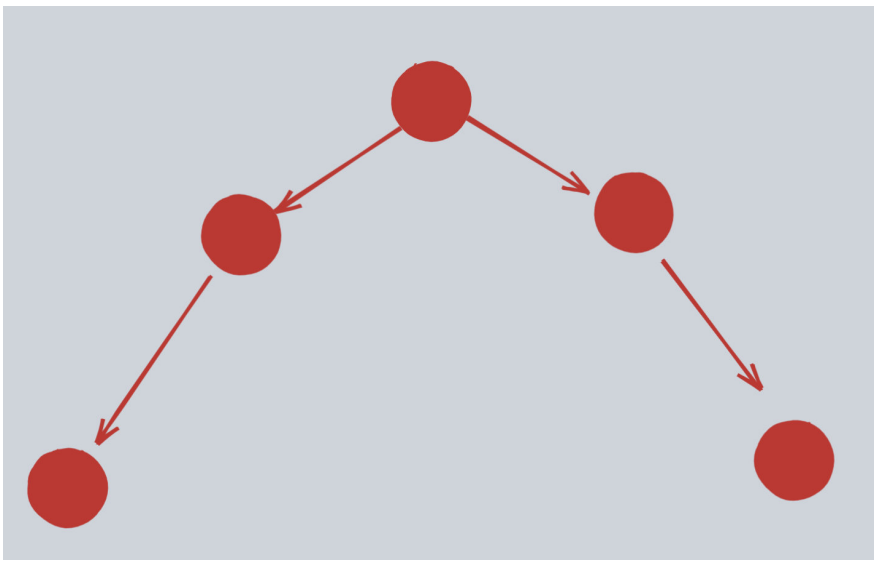
完全二叉树

一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i（1≤i≤n）的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。

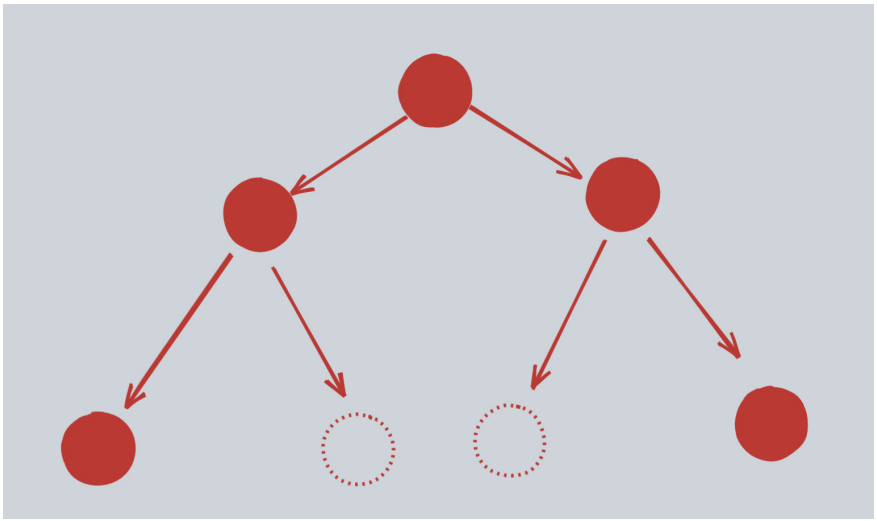
如下就是一颗完全二叉树：



直接考察完全二叉树的题目虽然不多，貌似只有一道 222. 完全二叉树的节点个数（二分可解），但是理解完全二叉树对你做题其实帮助很大。



如上图，是一颗普通的二叉树。如果我将其中的空节点补充完全，那么它就是一颗完全二叉树了。



这有什么用呢？这很有用！我总结了两个用处：

1. 我们可以给完全二叉树编号，这样父子之间就可以通过编号轻松求出。比如我给所有节点从左到右从上到下依次从 1 开始编号。那么已知一个节点的编号是 i，那么其左子节点就是 2 *i*，*右子节点就是* 2 i + 1，父节点就是 (i + 1) / 2。

熟悉二叉堆的同学可能发现了，这就是用数组实现的二叉堆，其实**二叉堆就是完全二叉树的一个应用**。

有的同学会说，”但是很多题目都不是完全二叉树呀，那不是用不上了么？“其实不然，我们只要想象它存在即可，我们将空节点脑补上去不就可以了？比如 662. 二叉树最大宽度。题目描述：

```
给定一个二叉树，编写一个函数来获取这个树的最大宽度。树的宽度是所有层中
每一层的宽度被定义为两个端点（该层最左和最右的非空节点，两端点间的null
示例 1:
输入:
           1
         /   \
        3     2
       / \     \
      5   3     9
输出: 4
解释: 最大值出现在树的第 3 层，宽度为 4 (5,3,null,9)。
```

很简单，一个带层的 BFS 模板即可搞定，简直就是默写题。不过这里需要注意两点：

- 入队的时候除了要将普通节点入队，还要空节点入队。
- 出队的时候除了入队节点本身，还要将节点的位置信息入队，即下方代码的 pos。

参考代码：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def widthOfBinaryTree(self, root: TreeNode) -> int:
        q = collections.deque([(root, 0)])
        steps = 0
        cur_depth = leftmost = ans = 0

        while q:
            for _ in range(len(q)):
                node, pos = q.popleft()
                if node:
                    # 节点编号关关系是不是用上了?
                    q.append((node.left, pos * 2))
                    q.append((node.right, pos * 2 + 1))
                    # 逻辑开始
                    if cur_depth != steps:
                        cur_depth = steps
                        leftmost = pos
                    ans = max(ans, pos - leftmost + 1)
                    # 逻辑结束
            steps += 1
        return ans
```

再比如剑指 Offer 37. 序列化二叉树。如果我将一个二叉树的完全二叉树形式序列化，然后通过 BFS 反序列化，这不就是力扣官方序列化树的方式么？比如：

```
    1
  / \
 2   3
    / \
   4   5
```

序列化为 "[1,2,3,null,null,4,5]"。 这不就是我刚刚画的完全二叉树么？就是将一个普通的二叉树硬生生当成完全二叉树用了。

> 其实这并不是序列化成了完全二叉树，下面会纠正。

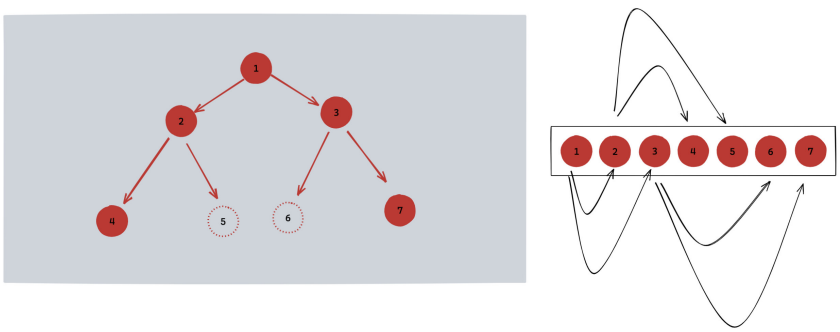
将一颗普通树序列化为完全二叉树很简单，只要将空节点当成普通节点入队处理即可。代码：

```
class Codec:

    def serialize(self, root):
        q = collections.deque([root])
        ans = ''
        while q:
            cur = q.popleft()
            if cur:
                ans += str(cur.val) + ','
                q.append(cur.left)
                q.append(cur.right)
            else:
                # 除了这里不一样，其他和普通的不记录层的 BFS 没区别
                ans += 'null,'
        # 末尾会多一个逗号，我们去掉它。
        return ans[:-1]
```

细心的同学可能会发现，我上面的代码其实并不是将树序列化成了完全二叉树，这个我们稍后就会讲到。另外后面多余的空节点也一并序列化了。这其实是可以优化的，优化的方式也很简单，那就是去除末尾的 null 即可。

你只要彻底理解我刚才讲的 `我们可以给完全二叉树编号，这样父子之间就可以通过编号轻松求出。比如我给所有节点从左到右从上到下依次从 1 开始编号。那么已知一个节点的编号是 i，那么其左子节点就是 2 * i，右子节点就是 2 * i + 1，父节点就是 (i + 1) / 2。` 这句话，那么反序列化对你就不是难事。

如果我用一个箭头表示节点的父子关系，箭头指向节点的两个子节点，那么大概是这样的：



我们刚才提到了：

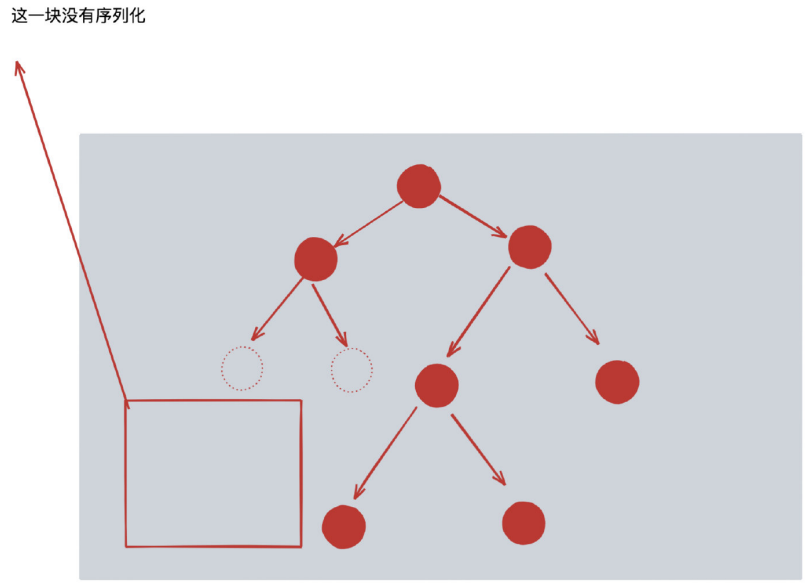
- 1 号节点的两个子节点的 2 号 和 3 号。

- 2 号节点的两个子节点的 4 号 和 5 号。
- 。。。
- i 号节点的两个子节点的 `2 * i` 号 和 `2 * 1 + 1` 号。

此时你可能会写出类似这样的代码：

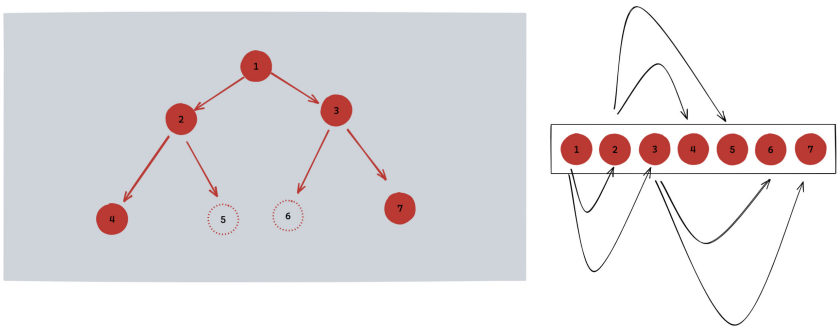
```
    def deserialize(self, data):
        if data == 'null': return None
        nodes = data.split(',')
        root = TreeNode(nodes[0])
        # 从一号开始编号，编号信息一起入队
        q = collections.deque([(root, 1)])
        while q:
            cur, i = q.popleft()
            # 2 * i 是左节点，而 2 * i 编号对应的其实是索引为 2 
            if 2 * i - 1 < len(nodes): lv = nodes[2 * i - 1
            if 2 * i < len(nodes): rv = nodes[2 * i]
            if lv != 'null':
                l = TreeNode(lv)
                # 将左节点和 它的编号 2 * i 入队
                q.append((l, 2 * i))
                cur.left = l
            if rv != 'null':
                r = TreeNode(rv)
                # 将右节点和 它的编号 2 * i + 1 入队
                q.append((r, 2 * i + 1))
                cur.right = r

        return root
```

但是上面的代码是不对的，因为我们序列化的时候其实不是完全二叉树，这也是上面我埋下的伏笔。因此遇到类似这样的 case 就会挂：



这也是我前面说”上面代码的序列化并不是一颗完全二叉树“的原因。

其实这个很好解决， 核心还是上面我画的那种图：



其实我们可以：

- 用三个指针分别指向数组第一项，第二项和第三项（如果存在的话），这里用 p1，p2，p3 来标记，分别表示当前处理的节点，当前处理的节点的左子节点和当前处理的节点的右子节点。
- p1 每次移动一位，p2 和 p3 每次移动两位。
- p1.left = p2; p1.right = p3。
- 持续上面的步骤直到 p1 移动到最后。

因此代码就不难写出了。反序列化代码如下：

```
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    q = collections.deque([root])
    i = 0
    while q and i < len(nodes) - 2:
        cur = q.popleft()
        lv = nodes[i + 1]
        rv = nodes[i + 2]
        i += 2
        if lv != 'null':
            l = TreeNode(lv)
            q.append(l)
            cur.left = l
        if rv != 'null':
            r = TreeNode(rv)
            q.append(r)
            cur.right = r

    return root
```

这个题目虽然并不是完全二叉树的题目，但是却和完全二叉树很像，有借鉴完全二叉树的地方。

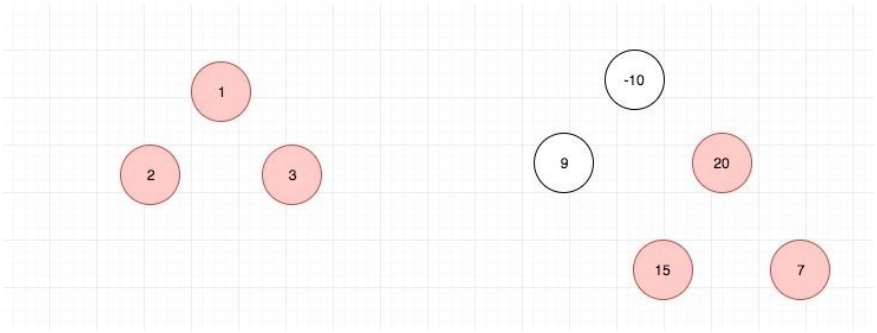
路径

关于路径这个概念，leetcode 真的挺喜欢考察的，不信你自己去 leetcode 官网搜索一下路径，看有多少题。树的路径这种题目的变种很多，算是一种经典的考点了。

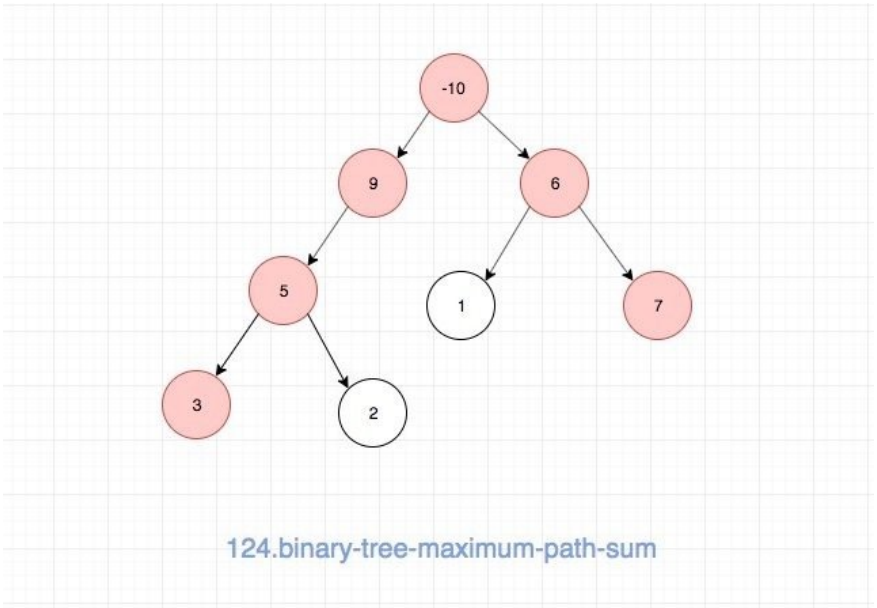
要明白路径的概念，以及如何解决这种题，只需要看一个题目就好了 124. 二叉树中的最大路径和，虽然是困难难度，但是搞清楚概念的话，和简单难度没啥区别。 接下来，我们就以这道题讲解一下。

这道题的题目是 `给定一个非空二叉树，返回其最大路径和` 。路径的概念是： `一条从树中任意节点出发，沿父节点-子节点连接，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。` 这听起来真的不容易理解，力扣给的 demo 我也没搞懂，这里我自己画了几个图来给大家解释一下这个概念。

首先是官网给的两个例子：



接着是我自己画的一个例子：



如图红色的部分是最大路径上的节点。

可以看出：

- 路径可以由一个节点做成，可以由两个节点组成，也可以由三个节点组成等等，但是必须连续。
- 路径必须是”直来直去“的，不能有分叉。 比如上图的路径的左下角是 3，当然也可以是 2，但是 2 比较小。但是不可以 2 和 3 同时选。

我们继续回到 124 题。题目说是 ”从任意节点出发.....“ 看完这个描述我会想到大概率是要么全局记录最大值，要么双递归。

- 如果使用双递归，那么复杂度就是 $O(N^2)$，实际上，子树的路径和计算出来了，可以推导出父节点的最大路径和，因此如果使用双递归会有重复计算。一个可行的方式是记忆化递归。
- 如果使用全局记录最大值，只需要在递归的时候 return 当前的一条边（上面提了不能拐），并在函数内部计算以当前节点出发的最大路径和，并更新全局最大值即可。 这里的核心其实是 return 较大的一条边，因为较小的边不可能是答案。

这里我选择使用第二种方法。

代码：

```
class Solution:
    ans = float('-inf')
    def maxPathSum(self, root: TreeNode) -> int:
        def dfs(node):
            if not node: return 0
            l = dfs(node.left)
            r = dfs(node.right)
            # 选择当前的节点，并选择左右两边，当然左右两边也可以不选
            self.ans = max(self.ans, max(l,0) + max(r, 0) +
            # 只返回一边，因此我们挑大的返回。当然左右两边也可以不选
            return max(l, r, 0) + node.val
        dfs(root)
        return self.ans
```

> 类似题目 113. 路径总和 I

距离

和路径类似，距离也是一个相似且频繁出现的一个考点，并且二者都是搜索类题目的考点。原因就在于最短路径就是距离，而树的最短路径就是边的数目。

这两个题练习一下，碰到距离的题目基本就稳了。

- 834.树中距离之和
- 863.二叉树中所有距离为 K 的结点

七个技巧

上面数次提到了七个技巧，相信大家已经迫不及待想要看看这七个技巧了吧。那就让我拿出本章压箱底的内容吧~

> 注意，这七个技巧全部是基于 dfs 的，bfs 掌握了模板就行，基本没有什么技巧可言。

认真学习的小伙伴可以发现了， 上面的内容只有**二叉树的迭代写法（双色标记法）** 和 **两个 BFS 模板** 具有实操性，其他大多是战略思想上的。算法思想固然重要，但是要结合具体实践落地才能有实践价值，才能让我们把知识消化成自己的。而这一节满满的全是实用干货ヽ(￣ω￣(￣ω￣〃)ゝ。

dfs(root)

第一个技巧，也是最容易掌握的一个技巧。我们写力扣的树题目的时候，函数的入参全都是叫 root。而这个技巧是说，我们在写 dfs 函数的时候，要将函数中表示当前节点的形参**也**写成 root。即：

```
def dfs(root):
    # your code
```

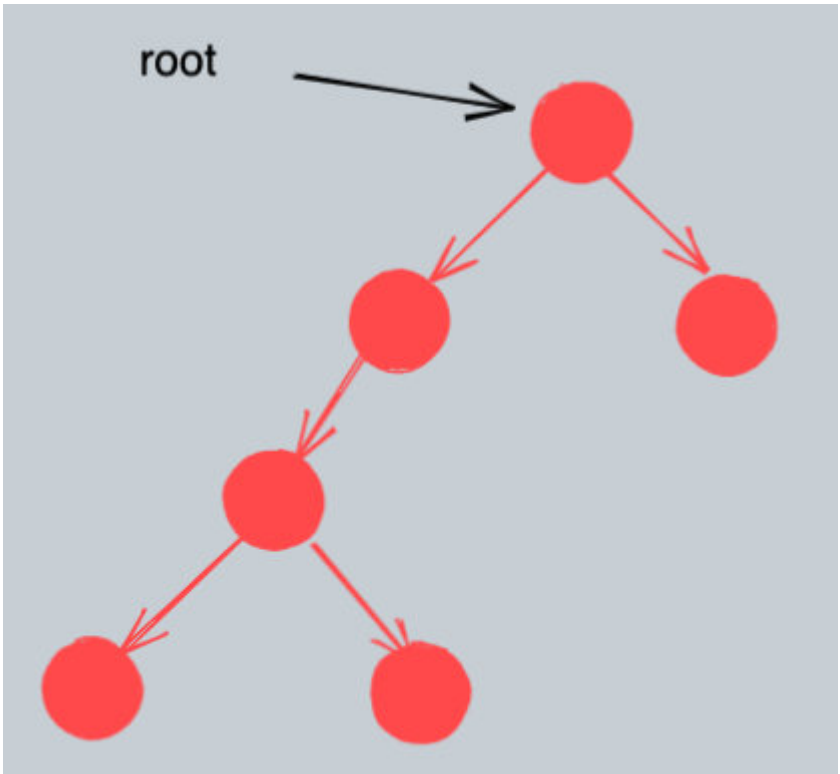
而之前我一直习惯写成 node，即：

```
def dfs(node):
    # your code
```

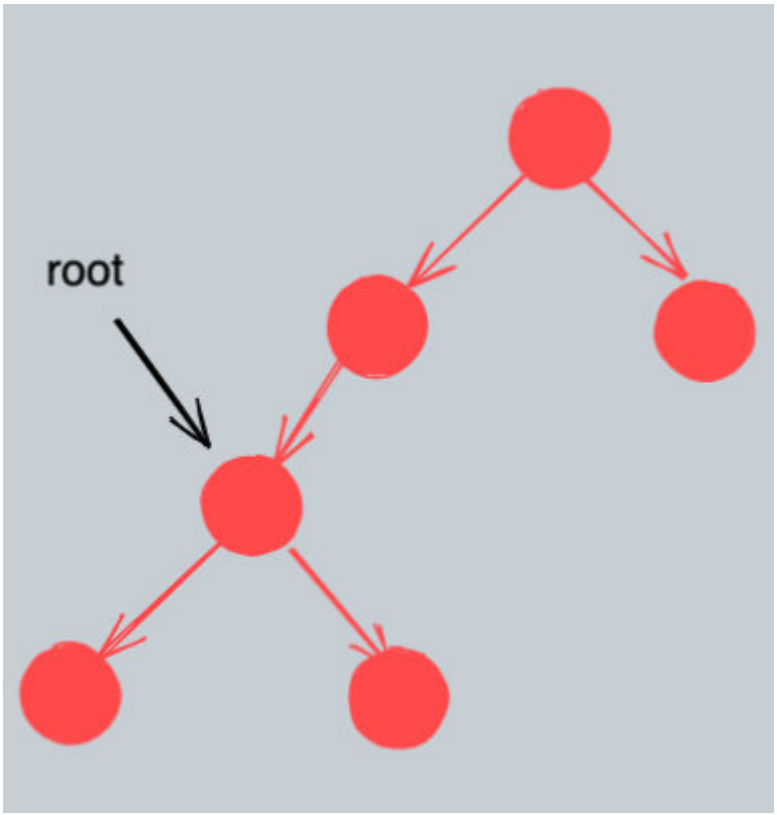
可能有的同学想问：”这有什么关系么？“。我总结了两个原因。

第一个原因是：以前 dfs 的形参写的是 node， 而我经常误写成 root，导致出错（这个错误并不会抛错，因此不是特别容易发现）。自从换成了 root 就没有发生这样的问题了。

第二个原因是：这样写相当于把 root 当成是 current 指针来用了。最开始 current 指针指向 root，然后不断修改指向树的其它节点。这样就概念就简化了，只有一个当前指针的概念。如果使用 node，就是当前指针 + root 指针两个概念了。



（一开始 current 就是 root）



（后面 current 不断改变。具体如何改变，取决于你的搜索算法，是 dfs 还是 bfs 等）

单/双递归

上面的技巧稍显简单，但是却有用。这里介绍一个稍微难一点的技巧，也更加有用。

我们知道递归是一个很有用的编程技巧，灵活使用递归，可以使自己的代码更加简洁，简洁意味着代码不容易出错，即使出错了，也能及时发现问题并修复。

树的题目大多数都可以用递归轻松地解决。**如果一个递归不行，那么来两个。（至今没见过三递归或更多递归）**

单递归大家写的比较多了，其实本篇文章的大部分递归都是单递归。 那什么时候需要两个递归呢？其实我上面已经提到了，那就是**如果题目有类似，任意节点开始 xxxx 或者所有 xxx**这样的说法，就可以考虑使用双递归。但是如果递归中有重复计算，则可以使用双递归 + 记忆化 或者直接单递归。

比如 面试题 04.12. 求和路径，再比如 563.二叉树的坡度 这两道题的题目说法都可以考虑使用双递归求解。

双递归的基本套路就是一个主递归函数和一个内部递归函数。主递归函数负责计算以某一个节点开始的 xxxx，内部递归函数负责计算 xxxx，这样就实现了以**所有节点开始的 xxxx**。

> 其中 xxx 可以替换成任何题目描述，比如路径和等

一个典型的加法双递归是这样的：

```
def dfs_inner(root):
    # 这里写你的逻辑，就是前序遍历
    dfs_inner(root.left)
    dfs_inner(root.right)
    # 或者在这里写你的逻辑，那就是后序遍历
def dfs_main(root):
    return dfs_inner(root) + dfs_main(root.left) + dfs_mai
```

大家可以用我的模板去套一下上面两道题试试。

前后遍历

前面我的链表专题也提到了前后序遍历。由于链表只有一个 next 指针，因此只有两种遍历。而二叉树有两个指针，因此常见的遍历有三个，除了前后序，还有一个中序。而中序除了二叉搜索树，其他地方用的并不多。

和链表一样， 要掌握树的前后序，也只需要记住一句话就好了。那就是**如果是前序遍历，那么你可以想象上面的节点都处理好了，怎么处理的不用管**。相应地**如果是后序遍历，那么你可以想象下面的树都处理好了，怎么处理的不用管**。这句话的正确性也是毋庸置疑。

前后序对链表来说比较直观。对于树来说，其实更形象地说应该是自顶向下或者自底向上。自顶向下和自底向上在算法上是不同的，不同的写法有时候对应不同的书写难度。比如 https://leetcode-cn.com/problems/sum-root-to-leaf-numbers/，这种题目就适合通过参数扩展 + 前序来完成。

> 关于参数扩展的技巧，我们在后面展开。

- **自顶向下**就是在每个递归层级，首先访问节点来计算一些值，并在递归调用函数时将这些值传递到子节点，一般是**通过参数传到子树**中。
- **自底向上**是另一种常见的递归方法，首先对所有子节点递归地调用函数，然后根据**返回值**和**根节点本身**的值得到答案。

关于前后序的思维技巧，可以参考我的这个文章 的**前后序部分**。

总结下我的经验：

- 大多数树的题使用后序遍历比较简单，并且大多需要依赖左右子树的返回值。比如 1448. 统计二叉树中好节点的数目

- 不多的问题需要前序遍历，而前序遍历通常要结合参数扩展技巧。比如 1022. 从根到叶的二进制数之和
- 如果你能使用参数和节点本身的值来决定什么应该是传递给它子节点的参数，那就用前序遍历。
- 如果对于树中的任意一个节点，如果你知道它子节点的答案，你能计算出当前节点的答案，那就用后序遍历。
- 如果遇到二叉搜索树则考虑中序遍历

虚拟节点

是的！不仅仅链表有虚拟节点的技巧，树也是一样。关于这点大家可能比较容易忽视。

回忆一下链表的虚拟指针的技巧，我们通常在什么时候才会使用?

- 其中一种情况是 `链表的头会被修改` 。这个时候通常需要一个虚拟指针来做新的头指针，这样就不需要考虑第一个指针的问题了（因为此时第一个指针变成了我们的虚拟指针，而虚拟指针是不用参与题目运算的）。树也是一样，当你需要对树的头节点（在树中我们称之为根节点）进行修改的时候， 就可以考虑使用虚拟指针的技巧了。
- 另外一种是题目需要返回树中间的某个节点（不是返回根节点）。实际上也可借助虚拟节点。由于我上面提到的指针的操作，实际上，你可以新建一个虚拟头，然后让虚拟头在恰当的时候（刚好指向需要返回的节点）断开连接，这样我们就可以返回虚拟头的 next 就 ok 了。

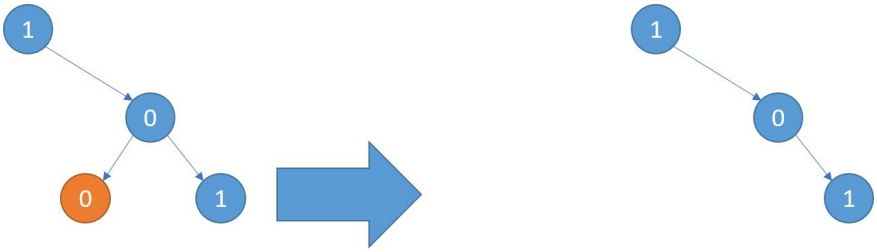
更多关于虚拟指针的技巧可以参考这个文章 的**虚拟头部分**。

下面就力扣中的两道题来看一下。

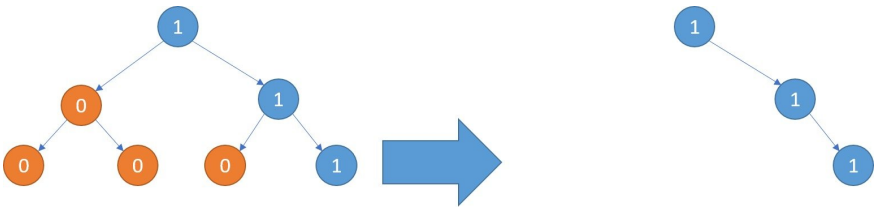
【题目一】814. 二叉树剪枝

题目描述：

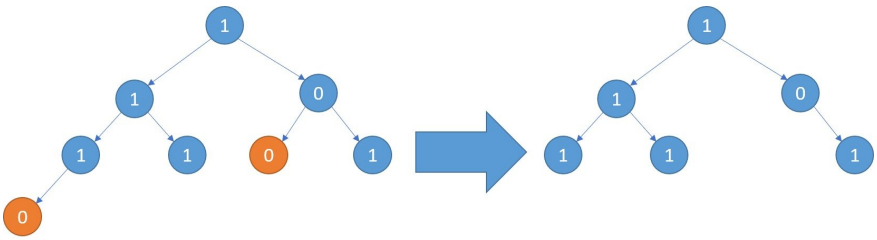
```
给定二叉树根结点 root ，此外树的每个结点的值要么是 0，要么是 1。

返回移除了所有不包含 1 的子树的原二叉树。

( 节点 X 的子树为 X 本身，以及所有 X 的后代。)

示例1:
输入: [1,null,0,0,1]
输出: [1,null,0,null,1]

解释:
只有红色节点满足条件“所有不包含 1 的子树”。
右图为返回的答案。
```



示例2:
输入: [1,0,1,0,0,0,1]
输出: [1,null,1,null,1]



示例3:
输入: [1,1,0,1,1,0,1,0]
输出: [1,1,0,1,1,null,1]



说明:

给定的二叉树最多有 100 个节点。
每个节点的值只会为 0 或 1 。

根据题目描述不难看出， 我们的根节点可能会被整个移除掉。这就是我上面说的 根节点被修改 的情况。 这个时候，我们只要新建一个虚拟节点当做新的根节点，就不需要考虑这个问题了。

此时的代码是这样的:

```
var pruneTree = function (root) {
  function dfs(root) {
    // do something
  }
  ans = new TreeNode(-1);
  ans.left = root;
  dfs(ans);
  return ans.left;
};
```

接下来，只需要完善 dfs 框架即可。 dfs 框架也很容易，我们只需要将子树和为 0 的节点移除即可，而计算子树和是一个难度为 easy 的题目，只需要后序遍历一次并收集值即可。

计算子树和的代码如下:

```
function dfs(root) {
  if (!root) return 0;
  const l = dfs(root.left);
  const r = dfs(root.right);
  return root.val + l + r;
}
```

有了上面的铺垫，最终代码就不难写出了。

完整代码(JS):

```
var pruneTree = function (root) {
  function dfs(root) {
    if (!root) return 0;
    const l = dfs(root.left);
    const r = dfs(root.right);
    if (l == 0) root.left = null;
    if (r == 0) root.right = null;
    return root.val + l + r;
  }
  ans = new TreeNode(-1);
  ans.left = root;
  dfs(ans);
  return ans.left;
};
```

【题目一】1325. 删除给定值的叶子节点

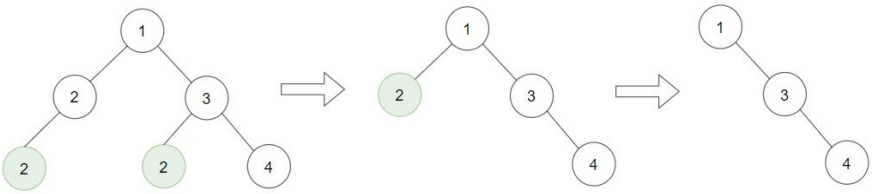
题目描述:

给你一棵以 root 为根的二叉树和一个整数 target ，请你删除所有值为 ta

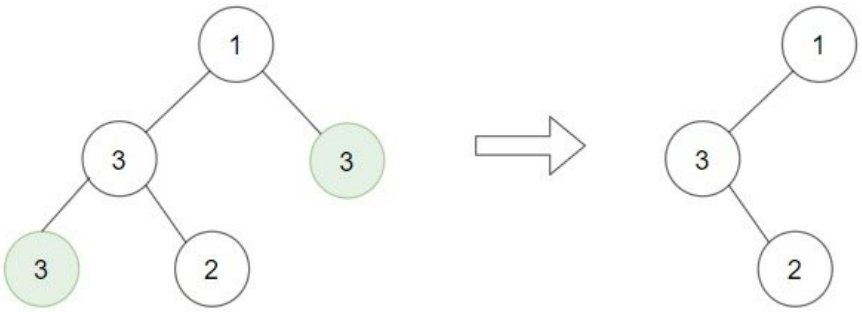
注意，一旦删除值为 target 的叶子节点，它的父节点就可能变成叶子节点；如

也就是说，你需要重复此过程直到不能继续删除。

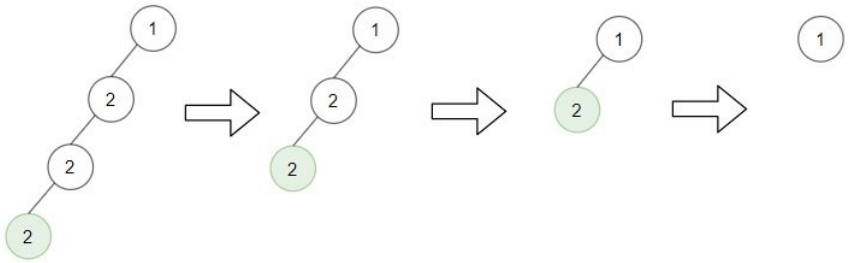
示例 1：



输入：root = [1,2,3,2,null,2,4], target = 2
输出：[1,null,3,null,4]
解释：
上面左边的图中，绿色节点为叶子节点，且它们的值与 target 相同（同为 2
有一个新的节点变成了叶子节点且它的值与 target 相同，所以将再次进行删除
示例 2：



输入：root = [1,3,3,3,2], target = 3
输出：[1,3,null,null,2]
示例 3：




```
输入：root = [1,2,null,2,null,2], target = 2
输出：[1]
解释：每一步都删除一个绿色的叶子节点（值为 2）。
示例 4：

输入：root = [1,1,1], target = 1
输出：[]
示例 5：

输入：root = [1,2,3], target = 1
输出：[1,2,3]


提示：

1 <= target <= 1000
每一棵树最多有 3000 个节点。
每一个节点值的范围是 [1, 1000] 。
```

和上面题目类似，这道题的根节点也可能被删除，因此这里我们采取和上面题目类似的技巧。

由于题目说明了**一旦删除值为 target 的叶子节点，它的父节点就可能变成叶子节点；如果新叶子节点的值恰好也是 target ，那么这个节点也应该被删除。也就是说，你需要重复此过程直到不能继续删除。** 因此这里使用后序遍历会比较容易，因为形象地看上面的描述过程你会发现这是一个自底向上的过程，而自底向上通常用后序遍历。

上面的题目，我们可以根据子节点的返回值决定是否删除子节点。而这道题是根据左右子树是否为空，删除**自己**，关键字是自己。而树的删除和链表删除类似，树的删除需要父节点，因此这里的技巧和链表类似，记录一下当前节点的父节点即可，并通过**参数扩展**向下传递。至此，我们的代码大概是：

```
class Solution:
    def removeLeafNodes(self, root: TreeNode, target: int)
        # 单链表只有一个 next 指针，而二叉树有两个指针 left 和 ri
        def dfs(node, parent, is_left=True):
            # do something
        ans = TreeNode(-1)
        ans.left = root
        dfs(root, ans)
        return ans.left
```

有了上面的铺垫，最终代码就不难写出了。

完整代码（Python）：

```
class Solution:
    def removeLeafNodes(self, root: TreeNode, target: int)
        def dfs(node, parent, is_left=True):
            if not node: return
            dfs(node.left, node, True)
            dfs(node.right, node, False)
            if node.val == target and parent and not node.
                if is_left: parent.left = None
                else: parent.right = None
        ans = TreeNode(-1)
        ans.left = root
        dfs(root, ans)
        return ans.left
```

边界

发现自己老是边界考虑不到，首先要知道这是正常的，人类的本能。 大家要克服这种本能， 只有多做，慢慢就能克服。 就像改一个坏习惯一样，除了坚持，一个有用的技巧是奖励和惩罚，我也用过这个技巧。

上面我介绍了树的三种题型。对于不同的题型其实边界考虑的侧重点也是不一样的，下面我们展开聊聊。

搜索类

搜索类的题目，树的边界其实比较简单。 90% 以上的题目边界就两种情况。

> 树的题目绝大多树又是搜索类，你想想掌握这两种情况多重要。

1. 空节点

伪代码：

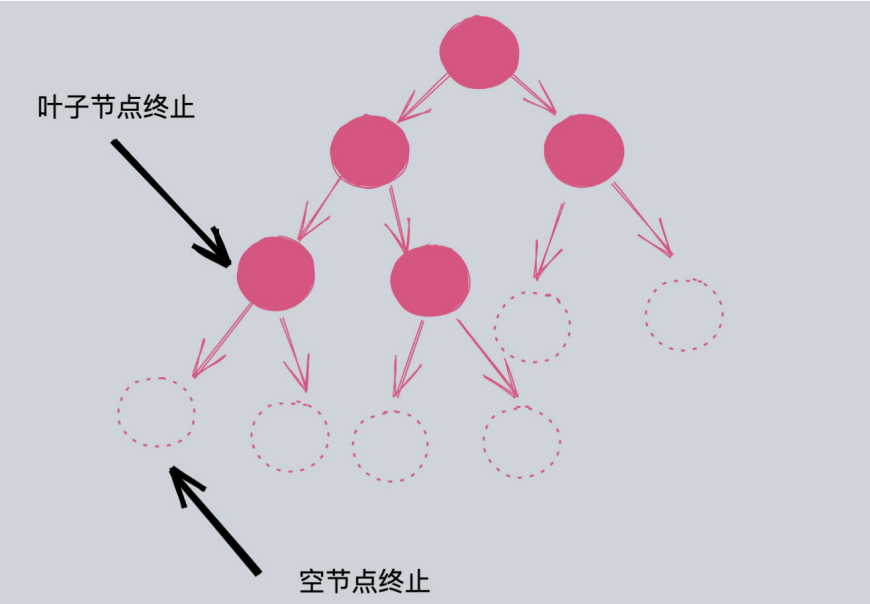
```
def dfs(root):
    if not root: print('是空节点，你需要返回合适的值')
    # your code here`
```

1. 叶子节点

伪代码：

```
def dfs(root):
    if not root: print('是空节点，你需要返回合适的值')
    if not root.left and not root.right: print('是叶子节点，
# your code here`
```

一张图总结一下：



经过这样的处理，后面的代码基本都不需要判空了。

构建类

相比于搜索类， 构建就比较麻烦了。我总结了两个常见的边界。

1. 参数扩展的边界

比如 1008 题， 根据前序遍历构造二叉搜索树。我就少考虑的边界。

```
def bstFromPreorder(self, preorder: List[int]) -> TreeNode:
    def dfs(start, end):
        if start > end:
            return None
        if start == end:
            return TreeNode(preorder[start])
        root = TreeNode(preorder[start])
        mid = -1
        for i in range(start + 1, end + 1):
            if preorder[i] > preorder[start]:
                mid = i
                break
        if mid == -1:
            root.left = dfs(start + 1, end)
        else:
            root.left = dfs(start + 1, mid - 1)
            root.right = dfs(mid, end)
        return root

    return dfs(0, len(preorder) - 1)
```

注意上面的代码没有判断 start == end 的情况，加下面这个判断就好了。

```
if start == end: return TreeNode(preorder[start])
```

1. 虚拟节点

除了搜索类的技巧可以用于构建类外，也可以考虑用我上面的讲的虚拟节点。

参数扩展大法

参数扩展这个技巧非常好用，一旦掌握你会爱不释手。

如果不考虑参数扩展， 一个最简单的 dfs 通常是下面这样：

```
def dfs(root):
    # do something
```

而有时候，我们需要 dfs 携带更多的有用信息。典型的有以下三种情况：

1. 携带父亲或者爷爷的信息。

```
def dfs(root, parent):
    if not root: return
    dfs(root.left, root)
    dfs(root.right, root)
```

1. 携带路径信息，可以是路径和或者具体的路径数组等。

路径和：

```
def dfs(root, path_sum):
    if not root:
        # 这里可以拿到根到叶子的路径和
        return path_sum
    dfs(root.left, path_sum + root.val)
    dfs(root.right, path_sum + root.val)
```

路径：

```
def dfs(root, path):
    if not root:
        # 这里可以拿到根到叶子的路径
        return path
    path.append(root.val)
    dfs(root.left, path)
    dfs(root.right, path)
    # 撤销
    path.pop()
```

学会了这个技巧，大家可以用 面试题 04.12. 求和路径 来练练手。

以上几个模板都很常见，类似的场景还有很多。总之当你需要传递额外信息给子节点（关键字是子节点）的时候，请务必掌握这种技巧。这也解释了为啥参数扩展经常用于前序遍历。

1. 二叉搜索树的搜索题大多数都需要扩展参考，甚至怎么扩展都是固定的。

二叉搜索树的搜索总是将最大值和最小值通过参数传递到左右子树，类似 `dfs(root, lower, upper)` ，然后在递归过程更新最大和最小值即可。这里需要注意的是 (lower, upper) 是的一个左右都开放的区间。

比如有一个题783. 二叉搜索树节点最小距离是求二叉搜索树的最小差值的绝对值。当然这道题也可以用我们前面提到的**二叉搜索树的中序遍历的结果是一个有序数组**这个性质来做。只需要一次遍历，最小差一定出现在相邻的两个节点之间。

这里我用另外一种方法，该方法就是扩展参数大法中的 左右边界法。

```
class Solution:
def minDiffInBST(self, root):
    def dfs(node, lower, upper):
        if not node:
            return upper - lower
        left = dfs(node.left, lower, node.val)
        right = dfs(node.right, node.val, upper)
        # 要么在左，要么在右，不可能横跨（因为是 BST）
        return min(left, right)
    return dfs(root, float('-inf'), float('inf')
```

其实这个技巧不仅适用二叉搜索树，也可是适用在别的树，比如 1026. 节点与其祖先之间的最大差值,题目大意是：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。

使用类似上面的套路轻松求解。

```
class Solution:
def maxAncestorDiff(self, root: TreeNode) -> int:
    def dfs(root, lower, upper):
        if not root:
            return upper - lower
        # 要么在左，要么在右，要么横跨。
        return max(dfs(root.left, min(root.val, lower), ma
    return dfs(root, float('inf'), float('-inf'))
```

返回元组/列表

通常，我们的 dfs 函数的返回值是一个单值。而有时候为了方便计算，我们会返回一个数组或者元祖。

> 对于个数固定情况，我们一般使用元组，当然返回数组也是一样的。

这个技巧和参数扩展有异曲同工之妙，只不过一个作用于函数参数，一个作用于函数返回值。

返回元祖

返回元组的情况还算比较常见。比如 865. 具有所有最深节点的最小子树，一个简单的想法是 dfs 返回深度，我们通过比较左右子树的深度来定位答案（最深的节点位置）。

代码：

```
class Solution:
    def subtreeWithAllDeepest(self, root: TreeNode) -> int:
        def dfs(node, d):
            if not node: return d
            l_d = dfs(node.left, d + 1)
            r_d = dfs(node.right, d + 1)
            if l_d >= r_d: return l_d
            return r_d
        return dfs(root, -1)
```

但是题目要求返回的是树节点的引用啊，这个时候应该考虑返回元祖，即**除了返回深度，也要把节点给返回**。

```
class Solution:
    def subtreeWithAllDeepest(self, root: TreeNode) -> Tree:
        def dfs(node, d):
            if not node: return (node, d)
            l, l_d = dfs(node.left, d + 1)
            r, r_d = dfs(node.right, d + 1)
            if l_d == r_d: return (node, l_d)
            if l_d > r_d: return (l, l_d)
            return (r, r_d)
        return dfs(root, -1)[0]
```

返回数组

dfs 返回数组比较少见。即使题目要求返回数组，我们也通常是声明一个数组，在 dfs 过程不断 push，最终返回这个数组。而不会选择返回一个数组。绝大多数情况下，返回数组是用于计算笛卡尔积。因此你需要用到笛卡尔积的时候，考虑使用返回数组的方式。

> 一般来说，如果需要使用笛卡尔积的情况还是比较容易看出的。另外一个不太准确的技巧是，如果题目有”所有可能“，”所有情况“，可以考虑使用此技巧。

一个典型的题目是 1530.好叶子节点对的数量

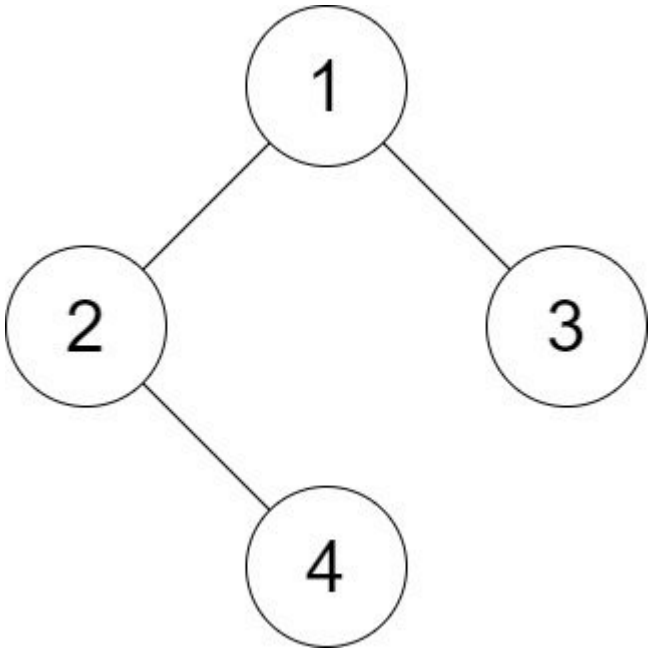
题目描述：

给你二叉树的根节点 root 和一个整数 distance 。

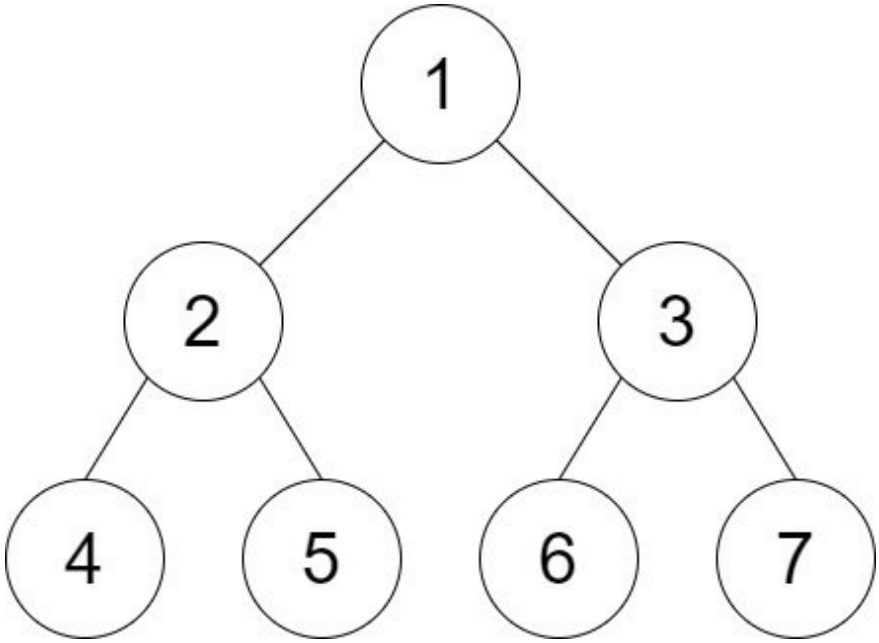
如果二叉树中两个叶节点之间的 最短路径长度 小于或者等于 distance ，那

返回树中 好叶子节点对的数量 。

示例 1：



输入：root = [1,2,3,null,4], distance = 3
输出：1
解释：树的叶节点是 3 和 4 ，它们之间的最短路径的长度是 3 。这是唯一的
示例 2：




```
输入：root = [1,2,3,4,5,6,7], distance = 3
输出：2
解释：好叶子节点对为 [4,5] 和 [6,7] ，最短路径长度都是 2 。但是叶子
示例 3：

输入：root = [7,1,4,6,null,5,3,null,null,null,null,null,2],
输出：1
解释：唯一的好叶子节点对是 [2,5] 。
示例 4：

输入：root = [100], distance = 1
输出：0
示例 5：

输入：root = [1,1,1], distance = 2
输出：1


提示：

tree 的节点数在 [1, 2^10] 范围内。
每个节点的值都在 [1, 100] 之间。
1 <= distance <= 10
```

上面我们学习了路径的概念，在这道题又用上了。

其实两个叶子节点的最短路径（距离）可以用其最近的公共祖先来辅助计算。即 `两个叶子节点的最短路径 = 其中一个叶子节点到最近公共祖先的距离 + 另外一个叶子节点到最近公共祖先的距离` 。

因此我们可以定义 dfs(root)，其功能是计算以 root 作为出发点，到其各个叶子节点的距离。 如果其子节点有 8 个叶子节点，那么就返回一个长度为 8 的数组， 数组每一项的值就是其到对应叶子节点的距离。

如果子树的结果计算出来了，那么父节点只需要把子树的每一项加 1 即可。这点不难理解，因为**父到各个叶子节点的距离就是父节点到子节点的距离（1） + 子节点到各个叶子节点的距离**。

由上面的推导可知需要先计算子树的信息，因此我们选择前序遍历。

完整代码（Python）：

```
class Solution:
    def countPairs(self, root: TreeNode, distance: int) ->
        self.ans = 0

        def dfs(root):
            if not root:
                return []
            if not root.left and not root.right:
                return [0]
            ls = [l + 1 for l in dfs(root.left)]
            rs = [r + 1 for r in dfs(root.right)]
            # 笛卡尔积
            for l in ls:
                for r in rs:
                    if l + r <= distance:
                        self.ans += 1
            return ls + rs
        dfs(root)
        return self.ans
```

894. 所有可能的满二叉树 也是一样的套路，大家用上面的知识练下手吧~

经典题目

推荐大家先把本文提到的题目都做一遍，然后用本文学到的知识做一下下面十道练习题，检验一下自己的学习成果吧！

- 剑指 Offer 55 - I. 二叉树的深度
- 剑指 Offer 34. 二叉树中和为某一值的路径
- 101. 对称二叉树
- 226. 翻转二叉树
- 543. 二叉树的直径
- 662. 二叉树最大宽度
- 971. 翻转二叉树以匹配先序遍历
- 987. 二叉树的垂序遍历

- 863. 二叉树中所有距离为 K 的结点
- 面试题 04.06. 后继者

总结

树的题目一种中心点就是**遍历**，这是搜索问题和修改问题的基础。

而遍历从大的方向分为**广度优先遍历和深度优先遍历**，这就是我们的**两个基本点**。两个基本点可以进一步细分，比如广度优先遍历有带层信息的和不带层信息的（其实只要会带层信息的就够了）。深度优先遍历常见的是前序和后序，中序多用于二叉搜索树，因为二叉搜索树的中序遍历是严格递增的数组。

树的题目从大的方向上来看就三种，一种是搜索类，这类题目最多，这种题目牢牢把握**开始点，结束点 和 目标即可**。构建类型的题目我之前的专题以及讲过了，一句话概括就是**根据一种遍历结果确定根节点位置，根据另外一种遍历结果（如果是二叉搜索树就不需要了）确定左右子树**。修改类题目不多，这种问题边界需要特殊考虑，这是和搜索问题的本质区别，可以使用虚拟节点技巧。另外搜索问题，如果返回值不是根节点也可以考虑虚拟节点。

树有四个比较重要的对做题帮助很大的概念，分别是完全二叉树，二叉搜索树，路径和距离，这里面相关的题目推荐大家好好做一下，都很经典。

最后我给大家介绍了七种干货技巧，很多技巧都说明了在什么情况下可以使用。好不好用你自己去找几个题目试试就知道了。

以上就是树专题的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

我整理的 1000 多页的电子书已经开发下载了，大家可以去我的公众号《力扣加加》后台回复电子书获取。

力扣加加

努力做西湖区最好的算法题解

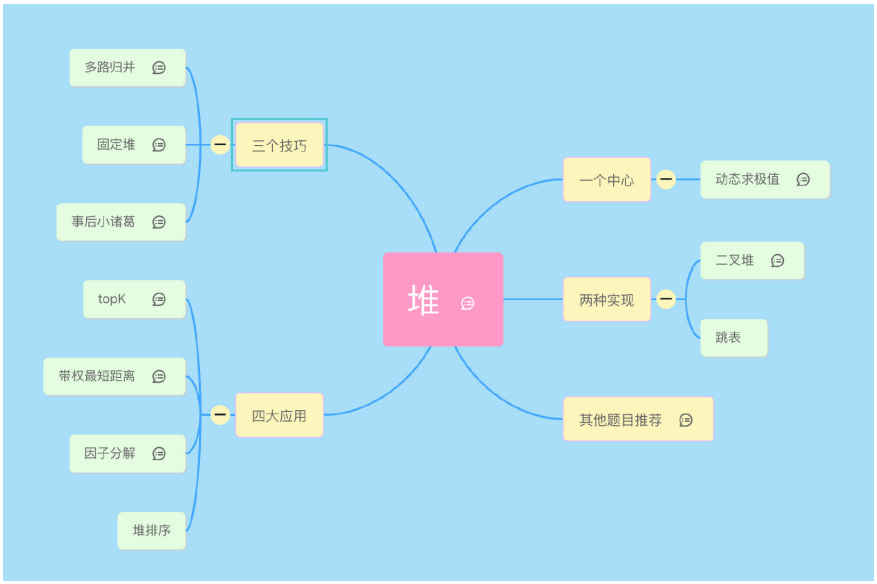
Back to page 1,291

目录

Introduction 1.1
第一章 – 算法专题 1.2
数据结构 1.2.1
基础算法 1.2.2
二叉树的遍历 1.2.3
动态规划 1.2.4
哈夫曼编码和游程编码 1.2.5
布隆过滤器 1.2.6
字符串问题 1.2.7
前缀树专题 1.2.8
《贪婪策略》专题 1.2.9
《深度优先遍历》专题 1.2.10
滑动窗口（思路 + 模板） 1.2.11
位运算 1.2.12
设计题 1.2.13
小岛问题 1.2.14
最大公约数 1.2.15
并查集 1.2.16
前缀和 1.2.17
平衡二叉树专题 1.2.18
第二章 – 91 天学算法 2.1
第一期讲义-二分法 2.1.1
第一期讲义-双指针 2.1.2
第二期 2.1.3
第三章 – 精选题解 3.1
《日程安排》专题 3.1.1
《构造二叉树》专题 3.1.2
字典序列删除 3.1.3
百度的算法面试题 * 祖玛游戏 3.1.4
西法带你学算法】一次搞定前缀和

Back to page 1,291

堆专题



大家好，我是 lucifer。今天给大家带来的是《堆》专题。先上下本文的提纲，这个是我用 mindmap 画的一个脑图，之后我会继续完善，将其他专题逐步完善起来。

> 大家也可以使用 vscode blink-mind 打开源文件查看，里面有一些笔记可以点开查看。源文件可以去我的公众号《力扣加加》回复脑图获取，以后脑图也会持续更新更多内容。vscode 插件地址：https://marketplace.visualstudio.com/items?itemName=awehook.vscode-blink-mind

本系列包含以下专题：

- 几乎刷完了力扣所有的链表题，我发现了这些东西。。。
- 几乎刷完了力扣所有的树题，我发现了这些东西。。。
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（就是本文）

一点絮叨

堆标签在 leetcode 一共有 **42 道题**。 为了准备这个专题，我将 leetcode 几乎所有的堆题目都刷了一遍。

可以看出，除了 3 个上锁的，其他我都刷了一遍。通过集中刷这些题，我发现了一些有趣的信息，今天就分享给大家。

需要注意的是，本文不对堆和优先队列进行区分。因此本文提到的堆和优先队列大家可以认为是同一个东西。如果大家对两者的学术区别感兴趣，可以去查阅相关资料。

> 如果不做特殊说明，本文的堆均指的是小顶堆。

堆的题难度几何?

堆确实是一个难度不低的专题。从官方的难度标签来看，堆的题目一共才 42 道，困难度将近 50%。没有对比就没有伤害，树专题困难度只有不到 10%。

从通过率来看，**一半以上**的题目平均通过率在 50% 以下。作为对比， 树的题目通过率在 50% 以下的只有**不到三分之一**。

不过大家不要太有压力。lucifer 给大家带来了一个口诀**一个中心，两种实现，三个技巧，四大应用**，我们不仅讲实现和原理，更讲问题的**背景以及套路和模板**。

> 文章里涉及的模板大家随时都可以从我的力扣刷题插件 leetcode-cheatsheet 中获取。

堆的使用场景分析

堆其实就是一种数据结构，数据结构是为了算法服务的，那堆这种数据结构是为哪种算法服务的？它的适用场景是什么？ 这是每一个学习堆的人**第一个**需要解决的问题。 在什么情况下我们会使用堆呢？堆的原理是什么？如何实现一个堆？别急，本文将一一为你揭秘。

在进入正文之前，给大家一个学习建议 - **先不要纠结堆怎么实现的，咱先了解堆解决了什么问题**。当你了解了使用背景和解决的问题之后，然后**当一个调包侠**，直接用现成的堆的 api 解决问题。等你理解得差不多了，再

去看堆的原理和实现。我就是这样学习堆的，因此这里就将这个学习经验分享给你。

为了对堆的使用场景进行说明，这里我虚拟了一个场景。

下面这个例子很重要， 后面会反复和这个例子进行对比。

一个挂号系统

问题描述

假如你是一个排队挂号系统的技术负责人。该系统需要给每一个前来排队的人发放一个排队码（入队），并根据**先来后到**的原则进行叫号（出队）。

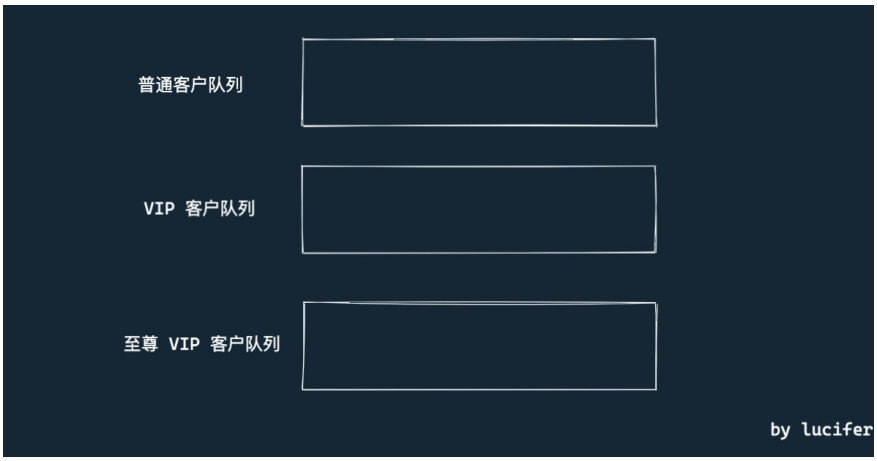
除此之外，我们还可以区分了几种客户类型， 分别是普通客户， VIP 客户 和 至尊 VIP 客户。

- 如果不同的客户使用不同的窗口的话，我该如何设计实现我的系统？（大家获得的服务不一样，比如 VIP 客户是专家级医生，普通客户是普通医生）
- 如果不同的客户都使用一个窗口的话，我该如何设计实现我的系统？（大家获得的服务都一样，但是优先级不一样。比如其他条件相同情况下(比如他们都是同时来挂号的)，VIP 客户 优先级高于普通客户）

我该如何设计我的系统才能满足需求，并获得较好的扩展性？

初步的解决方案

如果不同的客户使用不同的窗口。那么我们可以设计三个队列，分别存放正在排队的三种人。这种设计满足了题目要求，也足够简单。



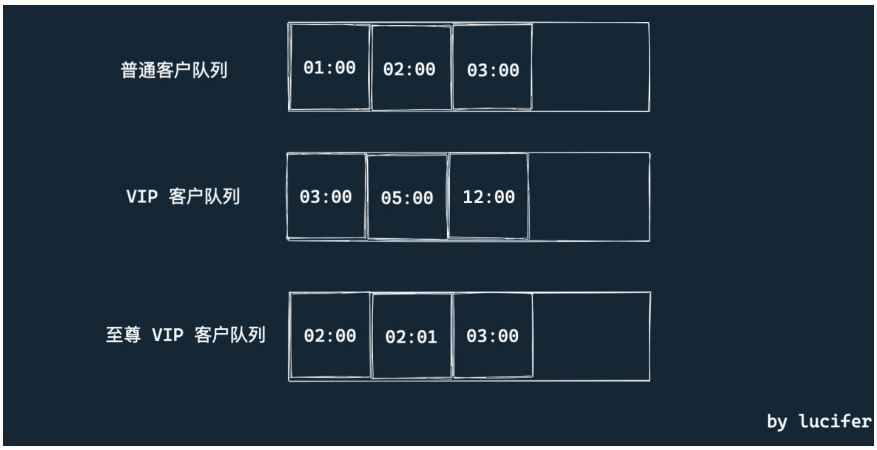
如果我们**只有一个窗口**，所有的病人需要使用同一个队列，并且同样的客户类型按照上面讲的**先到先服务原则**，但是不同客户类型之间可能会插队。

简单起见，我引入了**虚拟时间**这个概念。具体来说：

- 普通客户的虚拟时间就是真实时间。
- VIP 客户的虚拟时间按照实际到来时间减去一个小时。比如一个 VIP 客户是 14:00 到达的，我认为他是 13:00 到的。
- 至尊 VIP 客户的虚拟时间按照实际到来时间减去两个小时。比如一个至尊 VIP 客户是 14:00 到达的，我认为他是 12:00 到的。

这样，我们只需要按照上面的”虚拟到达时间“进行**先到先服务**即可。

因此我们就可以继续使用刚才的三个队列的方式，只不过队列存储的不是真实时间，而是虚拟时间。每次开始叫号的时候，我们使用虚拟时间比较，虚拟时间较小的先服务即可。



> 不难看出，队列内部的时间都是有序。

而这里的虚拟时间，其实就是优先队列中的优先权重，虚拟时间越小，权重越大。

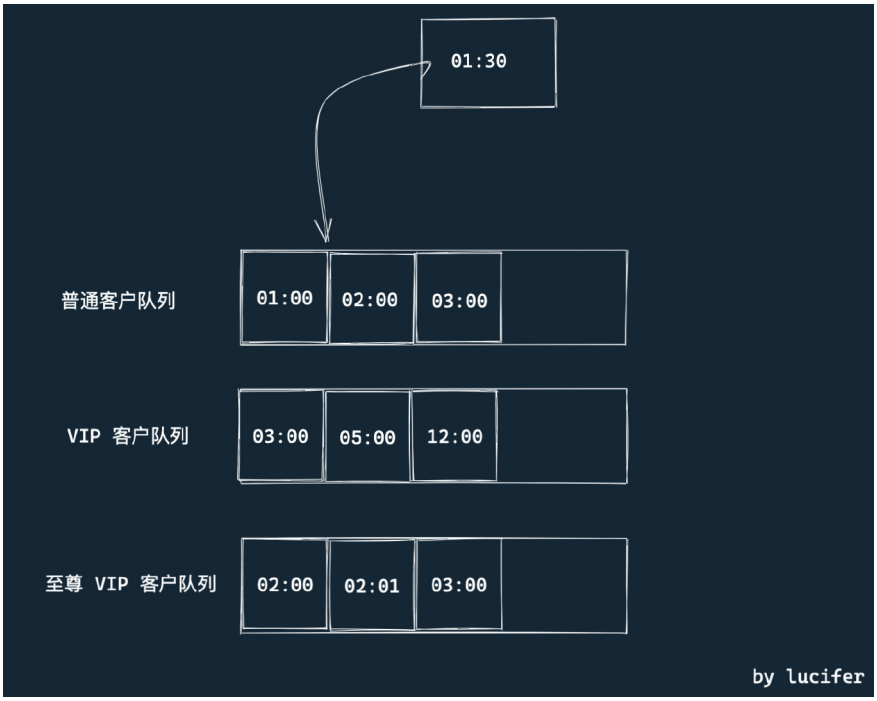
可以插队怎么办?

这种算法很好地完成了我们的需求，复杂度相当不错。不过事情还没有完结，这一次我们又碰到新的产品需求：

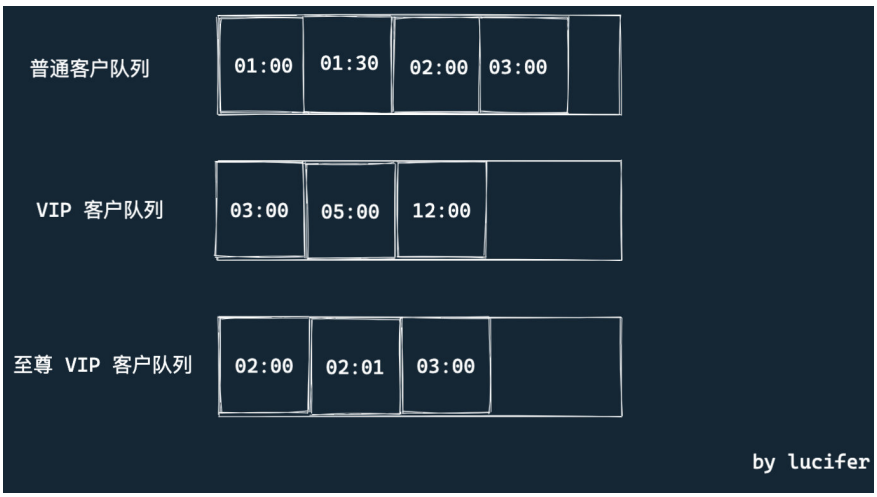
- 如果有别的门诊的病人转院到我们的诊所，则按照他之前的排队信息算，比如 ta 是 12:00 在别的院挂的号，那么转到本院仍然是按照 12:00 挂号算。
- 如果被叫到号三分钟没有应答，将其作废。但是如果后面病人重新来了，则认为他是当前时间减去一个小时的虚拟时间再次排队。比如 ta 是 13:00 被叫号，没有应答，13：30 又回来，则认为他是 12:30 排队的，重新进队列。

这样就有了”插队“的情况了。该怎么办呢？一个简单的做法是，将其插入到正确位置，并**重新调整后面所有人的排队位置**。

如下图是插入一个 1:30 开始排队的普通客户的情况。



（查找插入位置）

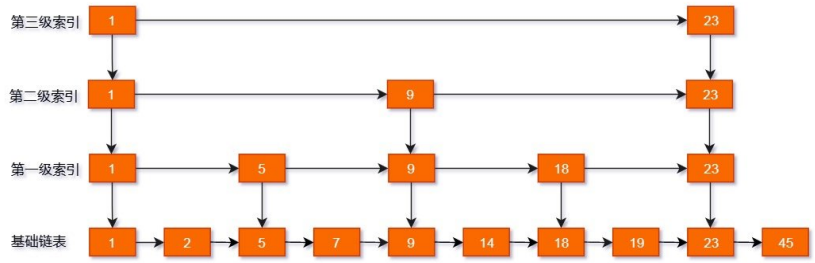


（将其插入）

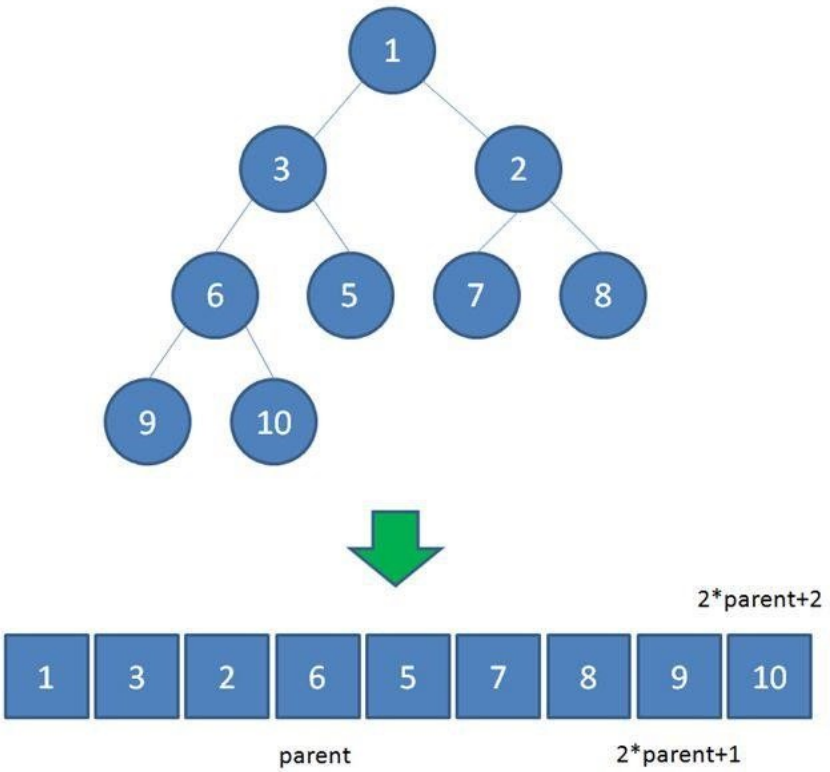
如果队列使用数组实现， 上面插队过程的时间复杂度为 $O(N)$，其中 N 为被插队的队伍长度。如果队伍很长，那么调整的次数明显增加。

不过我们发现，本质上我们就是在维护一个**有序列表**，而使用数组方式去维护有序列表的好处是可以随机访问，但是很明显这个需求并不需要这个特性。如果使用链表去实现，那么时间复杂度理论上是 $O(1)$，但是如何定位到需要插入的位置呢？朴素的思维是遍历查找，但是这样的时间复杂度又退化到了 $O(N)$。有没有时间复杂度更好的做法呢？答案就是本文的主角**优先队列**。

上面说了链表的实现核心在于查找也需要 $O(N)$，我们可以优化这个过程吗？实际上这就是优先级队列的链表实现，由于是有序的，我们可以用跳表加速查找，时间复杂度可以优化到 $O(logN)$。



其实算法界有很多类似的问题。比如建立数据库索引的算法，如果给某一个有序的列添加索引，不能每次插入一条数据都去调整所有的数据吧（上面的数组实现）？因此我们可以用平衡树来实现，这样每次插入可以最多调整 $(O(logN))$。优先队列的另外一种实现 - 二叉堆就是这个思想，时间复杂度也可以优化到 $O(logN)$



本文只讲解常见的二叉堆实现，对于跳表和红黑树不再这里讲。 关于优先队列的二叉堆实现，我们会在后面给大家详细介绍。这里大家只有明白优先队列解决的问题是什么就可以了。

使用堆解决问题

堆的两个核心 API 是 push 和 pop。

大家先不考虑它怎么实现的，你可以暂时把 ta 想象成一个黑盒，提供了两个 api：

- `push` ： 推入一个数据，内部怎么组织我不管。对应我上面场景里面的**排队**和**插队**。
- `pop` ： 弹出一个数据，该数据一定是最小的，内部怎么实现我不管。对应我上面场景里面的**叫号**。

> 这里的例子其实是小顶堆。而如果弹出的数据一定是最大的，那么对应的实现为大顶堆。

借助这两个 api 就可以实现上面的需求。

```
# 12:00 来了一个普通的顾客（push）
heapq.heappush(normal_pq, '12:00')
# 12:30 来了一个普通顾客（push）
heapq.heappush(normal_pq, '12:30')
# 13:00 来了一个普通顾客（push）
heapq.heappush(normal_pq, '13:00')
# 插队（push）。时间复杂度可以达到 O(logN)。如何做到先不管，我们先会
heapq.heappush(normal_pq, '12: 20')
# 叫号（pop）。12:00 来的先被叫到。需要注意的是这里的弹出时间复杂度也
heapq.heappop(normal_pq)
```

小结

上面这个场景单纯使用数组和链表都可以满足需求，但是使用其他数据结构在应对”插队“的情况表现地会更好。

具体来说：

- 如果永远都维护一个有序数组的方式取极值很容易，但是插队麻烦。
- 如果永远都维护一个有序链表的方式取极值也容易。 不过要想查找足够快，而不是线性扫描，就需要借助索引，这种实现对应的就是优先级队列的**跳表实现**。
- 如果永远都维护一个树的方式取极值也可以实现，比如根节点就是极值，这样 O(1) 也可以取到极值，但是调整过程需要 $O(logN)$。这种实现对应的就是优先级队列的**二叉堆实现**。

简单总结下就是，**堆就是动态帮你求极值的**。当你需要动态求最大或最小值就就用它。而具体怎么实现，复杂度的分析我们之后讲，现在你只要记住使用场景，堆是如何解决这些问题的以及堆的 api 就够了。

队列 VS 优先队列

上面通过一个例子带大家了解了一下优先队列。那么在接下来讲具体实现之前，我觉得有必要回答下一个大家普遍关心的问题，那就是**优先队列是队列么**？

很多人觉得队列和优先队列是完全不同的东西，就好像 Java 和 JavaScript 一样，我看了很多文章都是这么说的。

而我不这么认为。实际上，普通的队列也可以看成是一个特殊的**优先级队列**， 这和网上大多数的说法**优先级队列和队列没什么关系**有所不同。我认为**队列无非就是以时间这一变量作为优先级的优先队列**，时间越早，优先级越高，优先级越高越先出队。

大家平时写 BFS 的时候都会用到队列来帮你处理节点的访问顺序。那使用优先队列行不行？当然可以了！我举个例子：

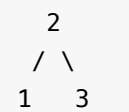
例题 - 513. 找树左下角的值

题目描述

定一个二叉树，在树的最后一行找到最左边的值。

示例 1:

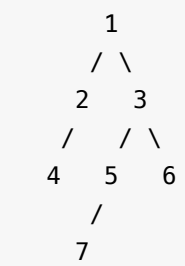
输入:



输出:
1

示例 2:

输入:



输出:
7

注意: 您可以假设树（即给定的根节点）不为 NULL。

思路

我们可以使用 BFS 来做一次层次遍历，并且每一层我们都从右向左遍历，这样层次遍历的最后一个节点就是**树左下角的节点**。

常规的做法是使用双端队列（就是队列）来实现，由于队列的先进先出原则很方便地就能实现**层次遍历**的效果。

代码

对于代码看不懂的同学，可以先不要着急。等完整读完本文之后再回过头看会容易很多。下同，不再赘述。

Python Code:

```
class Solution:
    def findBottomLeftValue(self, root: TreeNode) -> int:
        if root is None:
            return None
        queue = collections.deque([root])
        ans = None
        while queue:
            size = len(queue)
            for _ in range(size):
                ans = node = queue.popleft()
                if node.right:
                    queue.append(node.right)
                if node.left:
                    queue.append(node.left)
        return ans.val
```

实际上， 我们也可以使用优先队列的方式，思路和代码也几乎和上面完全一样。

```
class Solution:
    def findBottomLeftValue(self, root: TreeNode) -> int:
        if root is None:
            return None
        queue = []
        # 堆存储三元组(a,b,c)，a 表示层级，b 表示节点编号（以完全二
        heapq.heappush(queue, (1, 1, root))
        ans = None
        while queue:
            size = len(queue)
            for _ in range(size):
                level, i, node = heapq.heappop(queue)
                ans = node
                if node.right:
                    heapq.heappush(queue, (level + 1, 2 * i
                if node.left:
                    heapq.heappush(queue, (level + 1, 2 * i
        return ans.val
```

小结

所有使用队列的地方，都可以使用优先队列来完成，反之却不一定。

既然优先队列这么厉害，那平时都用优先队列不就行了？为啥使用队列的地方没见过别人用堆呢？最核心的原因是时间复杂度更差了。

比如上面的例子，本来入队和出队都可是很容易地在 $O(1)$ 的时间完成。而现在呢？入队和出队的复杂度都是 $O(logN)$，其中 N 为当前队列的大小。因此在没有必要的地方使用堆，会大大提高算法的时间复杂度，这当然不合适。说的粗俗一点就是脱了裤子放屁。

不过 BFS 真的就没人用优先队列实现么？当然不是！比如带权图的最短路径问题，如果用队列做 BFS 那就需要优先队列才可以，因为路径之间是有**权重的差异**的，这不就是优先队列的设计初衷么。**使用优先队列的 BFS 实现典型的就是 dijkstra 算法**。

这再一次应征了我的那句话**队列就是一种特殊的优先队列而已**。特殊到大家的权重就是按照到来的顺序定，谁先来谁的优先级越高。在这种特殊情况下，我们没必须去维护堆来完成，进而获得更好的时间复杂度。

一个中心

堆的问题核心点就一个，那就是**动态求极值**。动态和极值二者缺一不可。

求极值比较好理解，无非就是求最大值或者最小值，而动态却不然。比如要你求一个数组的第 k 小的数，这是动态么？这其实完全看你怎么理解。而在我们这里，这种情况就是动态的。

如何理解上面的例子是动态呢?

你可以这么想。由于堆只能求极值。比如能求最小值，但不能直接求第 k 小的值。

那我们是不是先求最小的值，然后将其出队（对应上面例子的叫号）。然后继续求最小的值，这个时候求的就是第 2 小了。如果要求第 k 小，那就如此反复 k 次即可。

这个过程，你会发现数据是在**动态变化的**，对应的就是堆的大小在变化。

接下来，我们通过几个例子来进行说明。

例一 - 1046. 最后一块石头的重量

题目描述

```
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

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
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



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


提示：

1 <= stones.length <= 30
1 <= stones[i] <= 1000
```

思路

题目比较简单，直接模拟即可。需要注意的是，每次选择两个最重的两个石头进行粉碎之后，最重的石头的重量便发生了变化。这会**影响到下次取最重的石头**。简单来说就是最重的石头在模拟过程中是**动态变化**的。

这种**动态取极值**的场景使用堆就非常适合。

> 当然看下这个数据范围 `1 <= stones.length <= 30` 且 `1 <= stones[i] <= 1000`，使用计数的方式应该也是可以的。

代码

Java Code:

```
import java.util.PriorityQueue;

public class Solution {

    public int lastStoneWeight(int[] stones) {
        int n = stones.length;
        PriorityQueue<Integer> maxHeap = new PriorityQueue<
        for (int stone : stones) {
            maxHeap.add(stone);
        }

        while (maxHeap.size() >= 2) {
            Integer head1 = maxHeap.poll();
            Integer head2 = maxHeap.poll();
            if (head1.equals(head2)) {
                continue;
            }
            maxHeap.offer(head1 - head2);
        }

        if (maxHeap.isEmpty()) {
            return 0;
        }
        return maxHeap.poll();
    }
}
```

例二 - 313. 超级丑数

题目描述

```
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
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 ≤ 100, 0 < n ≤ 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。
```

思路

这道题看似和动态求极值没关系。其实不然，让我们来分析一下这个题目。

我们可以实现生成超级多的丑数，比如先从小到大生成 N 个丑数，然后直接取第 N 个么?

拿这道题来说， 题目有一个数据范围限制 `0 < n ≤ 10^6` ，那我们是不是预先生成一个大小为 10^6 的超级丑数数组，这样我们就可通过 $O(1)$ 的时间获取到第 N 个超级丑数了。

首先第一个问题就是时间和空间浪费。我们其实没有必要每次都计算所有的超级丑数，这样的预处理空间和时间都很差。

第二个问题是，我们如何生成 10^6 以为的超级丑数呢?

通过丑数的定义，我们能知道超级丑数一定可以写出如下形式。

```
if primes = [a,b,c,....]
then f(ugly) = a * x1 * b * x2 * c * x3 ...
其中 x1，x2，x3 均为正整数。
```

不妨将问题先做一下简化处理。考虑题目给的例子：[2,7,13,19]。

我们可以使用四个指针来处理。直接看下代码吧：

```
public class Solution {
    public int solve(int n) {
        int ans[]=new int[n+5];
        ans[0]=1;
        int p1=0,p2=0,p3=0,p4=0;
        for(int i=1;i<n;i++){
            ans[i]=Math.min(ans[p1]*2,Math.min(ans[p2]*7,Ma
            if(ans[i]==ans[p1]*2) p1++;
            if(ans[i]==ans[p2]*7) p2++;
            if(ans[i]==ans[p3]*13) p3++;
            if(ans[i]==ans[p3]*19) p4++;
        }
        return ans[n-1];
    }
}
```

这个技巧我自己称之为**多路归并**（实现想不到什么好的名字），我也会在后面的**三个技巧**也会对此方法使用堆来优化。

由于这里的指针是动态的，指针的数量其实和题目给的 primes 数组长度一致。因此实际上，我们可以使用记忆化递归的形式来完成，**递归体和递归栈分别维护一个迭代变量即可**。而这道题其实可以看出是一个状态机，因此使用动态规划来解决是符合直觉的。而这里，介绍一种堆的解法，相比于动态规划，个人认为更简单和符合直觉。

> 关于状态机，我这里有一篇文章原来状态机也可以用来刷 LeetCode? ，大家可以参考一下哦。

实际上，我们可以**动态**维护一个当前最小的超级丑数。找到第一个， 我们将其移除，再找**下一个当前最小的超级丑数**（也就是全局第二小的超级丑数）。这样经过 n 轮，我们就得到了第 n 小的超级丑数。这种动态维护极值的场景正是堆的用武之地。

> 有没有觉得和上面石头的题目很像?

以题目给的例子 [2,7,13,19] 来说。

1. 将 [2,7,13,19] 依次入堆。
2. 出堆一个数字，也就是 2。这时取到了**第一个**超级丑数。
3. 接着将 2 和 [2,7,13,19] 的乘积，也就是 [4,14,26,38] 依次入堆。
4. 如此反复直到取到第 n 个超级丑数。

上面的正确性是毋庸置疑的，由于每次堆都可以取到最小的，每次我们也会将最小的从堆中移除。因此取 n 次自然就是第 n 大的超级丑数了。

堆的解法没有太大难度，唯一需要注意的是去重。比如 2 * 13 = 26，而 13 * 2 也是 26。我们不能将 26 入两次堆。解决的方法也很简单：

- 要么使用哈希表记录全部已经取出的数，对于已经取出的数字不再取即可。
- 另一种方法是记录上一次取出的数，由于取出的数字是按照**数字大小不严格递增**的，这样只需要拿上次取出的数和本次取出的数比较一下就知道了。

用哪种方法不用多说了吧?

代码

Java Code:

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        PriorityQueue<Long> queue=new PriorityQueue<>();
        int count = 0;
        long ans = 1;
        queue.add(ans);
        while (count < n) {
            ans=queue.poll();
            while (!queue.isEmpty() && ans == queue.peek()
                queue.poll();
            }
            count++;
            for (int i = 0; i < primes.length ; i++) {
                queue.offer(ans * primes[i]);
            }
        }
        return (int)ans;
    }
}
```

> ans 初始化为 1 的作用相当于虚拟头，仅仅起到了简化操作的作用

小结

堆的中心就一个，那就是**动态求极值**。

而求极值无非就是最大值或者最小值，这不难看出。如果求最大值，我们可以使用大顶堆，如果求最小值，可以用最小堆。

而实际上，如果没有动态两个字，很多情况下没有必要使用堆。比如可以直接一次遍历找出最大的即可。而动态这个点不容易看出来，这正是题目的难点。这需要你先对问题进行分析， 分析出这道题**其实就是动态求极值**，那么使用堆来优化就应该被想到。类似的例子有很多，我也会在后面的小节给大家做更多的讲解。

两种实现

上面简单提到了堆的几种实现。这里介绍两种常见的实现，一种是基于链表的实现- 跳表，另一种是基于数组的实现 - 二叉堆。

使用跳表的实现，如果你的算法没有经过精雕细琢，性能会比较不稳定，且在数据量大的情况下内存占用会明显增加。 因此我们仅详细讲述二叉堆的实现，而对于跳表的实现，仅讲述它的基本原理，对于代码实现等更详细的内容由于比较偏就不在这里讲了。

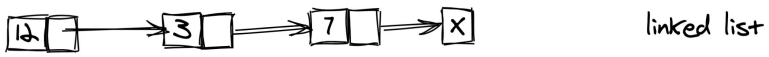
跳表

跳表也是一种数据结构，因此 ta 其实也是服务于某种算法的。

跳表虽然在面试中出现的频率不大，但是在工业中，跳表会经常被用到。力扣中关于跳表的题目只有一个。但是跳表的设计思路值得我们去学习和思考。 其中有很多算法和数据结构技巧值得我们学习。比如空间换时间的思想，比如效率的取舍问题等。

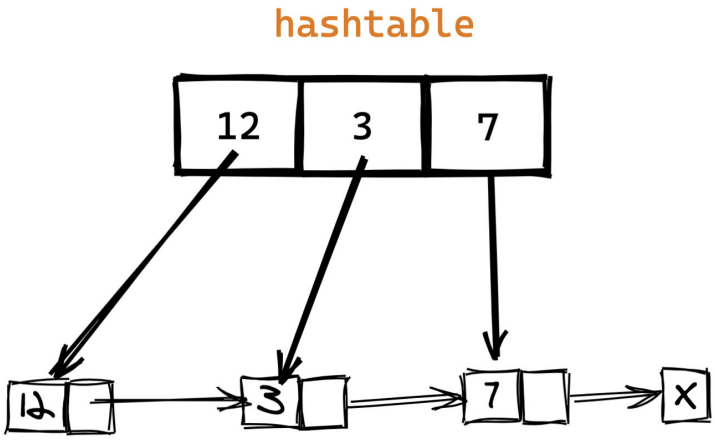
上面提到了应付插队问题是设计**堆**应该考虑的首要问题。堆的跳表实现是如何解决这个问题的呢?

我们知道，不借助额外空间的情况下，在链表中查找一个值，需要按照顺序一个个查找，时间复杂度为 $O(N)$，其中 N 为链表长度。



（单链表）

当链表长度很大的时候， 这种时间是很难接受的。 一种常见的的优化方式是**建立哈希表，将所有节点都放到哈希表中，以空间换时间的方式减少时间复杂度**，这种做法时间复杂度为 $O(1)$，但是空间复杂度为 $O(N)$。



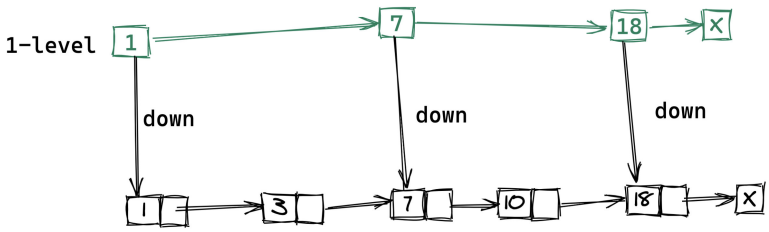
（单链表 + 哈希表）

为了防止链表中出现重复节点带来的问题，我们需要序列化节点，再建立哈希表，这种空间占用会更高，虽然只是系数级别的增加，但是这种开销也是不小的 。更重要的是，哈希表不能解决查找极值的问题，其仅适合根据 key 来获取内容。

为了解决上面的问题，跳表应运而生。

如下图所示，我们从链表中每两个元素抽出来，加一级索引，一级索引指向了原始链表，即：通过一级索引 7 的 down 指针可以找到原始链表的 7 。那怎么查找 10 呢?

> 注意这个算法要求链表是有序的。

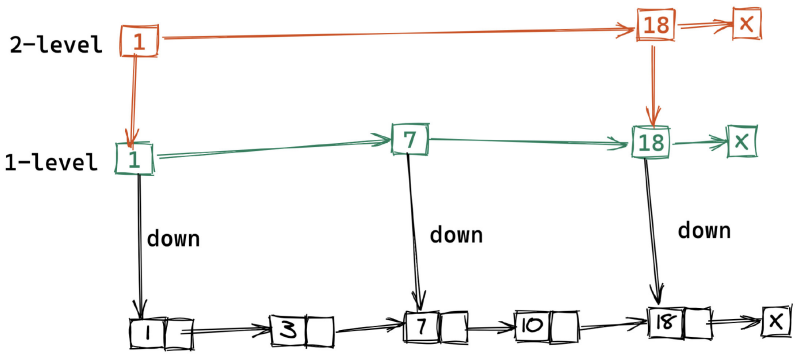


（建立一级索引）

我们可以：

- 通过现在一级跳表中搜索到 7，发现下一个 18 大于 10 ，也就是说我们要找的 10 在这两者之间。
- 通过 down 指针回到原始链表，通过原始链表的 next 指针我们找到了 10。

这个例子看不出性能提升。但是如果元素继续增大， 继续增加索引的层数，建立二级，三级。。。索引，使得链表能够实现二分查找，从而获得更好的效率。但是相应地，我们需要付出额外空间的代价。



（增加索引层数）

理解了上面的点，你可以形象地将跳表想象为玩游戏的**存档**。

一个游戏有 10 关。如果我想要玩第 5 关的某一个地方，那么我可以直接从第五关开始，这样要比从第一关开始快。我们甚至可以在每一关同时设置很多的存档。这样我如果想玩第 5 关的某一个地方，也可以不用从第 5 关的开头开始，而是直接选择**离你想玩的地方更近的存档**，这就相当于跳表的二级索引。

跳表的时间复杂度和空间复杂度不是很好分析。由于时间复杂度 = 索引的高度 * 平均每层索引遍历元素的个数，而高度大概为 $logn$，并且每层遍历的元素是常数，因此时间复杂度为 $logn$，和二分查找的空间复杂度是一样的。

空间复杂度就等同于索引节点的个数，以每两个节点建立一个索引为例，大概是 n/2 + n/4 + n/8 + … + 8 + 4 + 2 ，因此空间复杂度是 $O(n)$。当然你如果每三个建立一个索引节点的话，空间会更省，但是复杂度不变。

理解了上面的内容，使用跳表实现堆就不难了。

- 入堆操作，只需要根据索引插到链表中，并更新索引（可选）。
- 出堆操作，只需要删除头部（或者尾部），并更新索引（可选）。

大家如果想检测自己的实现是否有问题，可以去力扣的1206. 设计跳表 检测。

接下来，我们看下一种更加常见的实现 - 二叉堆。

二叉堆

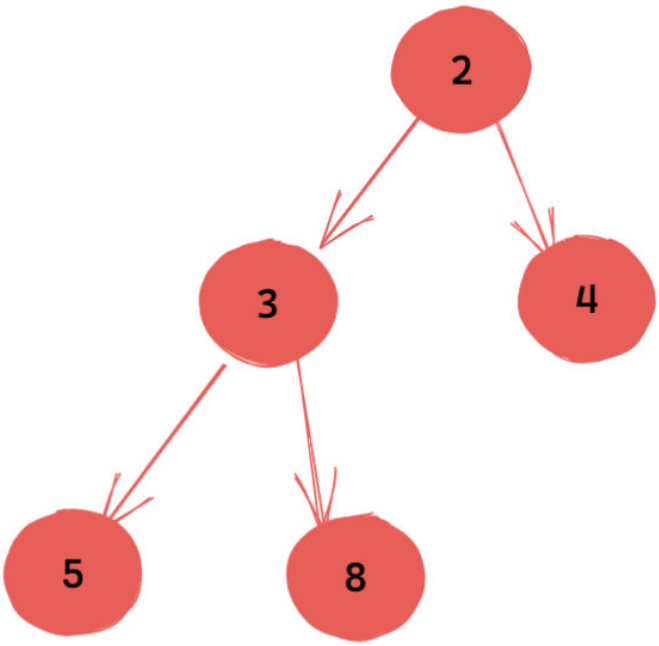
二叉堆的实现，我们仅讲解最核心的两个操作： heappop（出堆） 和 heappush（入堆）。对于其他操作不再讲解，不过我相信你会了这两个核心操作，其他的应该不是难事。

实现之后的使用效果大概是这样的：

```
h = min_heap()
h.build_heap([5, 6, 2, 3])

h.heappush(1)
h.heappop() # 1
h.heappop() # 2
h.heappush(1)
h.heappop() # 1
h.heappop() # 3
```

基本原理

本质上来说，二叉堆就是一颗特殊的完全二叉树。它的特殊性只体现在一点，那就是**父节点的权值不大于儿子的权值（小顶堆）**。



（一个小顶堆）

上面这句话需要大家记住，一切的一切都源于上面这句话。

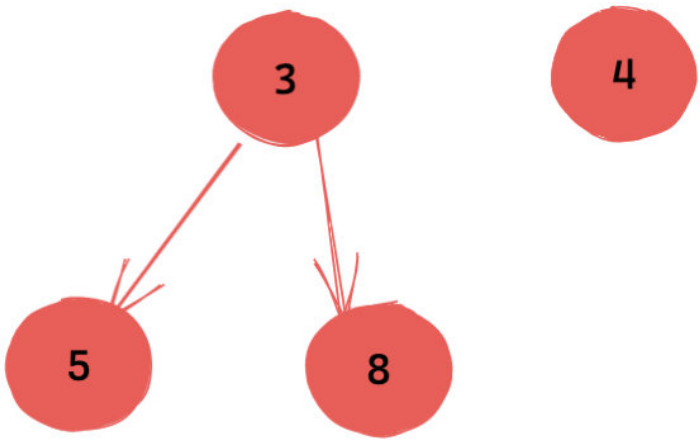
由于**父节点的权值不大于儿子的权值（小顶堆）**，那么很自然能推导出树的根节点就是最小值。这就起到了堆的**取极值**的作用了。

那动态性呢？二叉堆是怎么做到的呢？

出堆

假如，我将树的根节点出堆，那么根节点不就空缺了么？我应该将第二小的顶替上去。怎么顶替上去呢？一切的一切还是那句话**父节点的权值不大于儿子的权值（小顶堆）**。

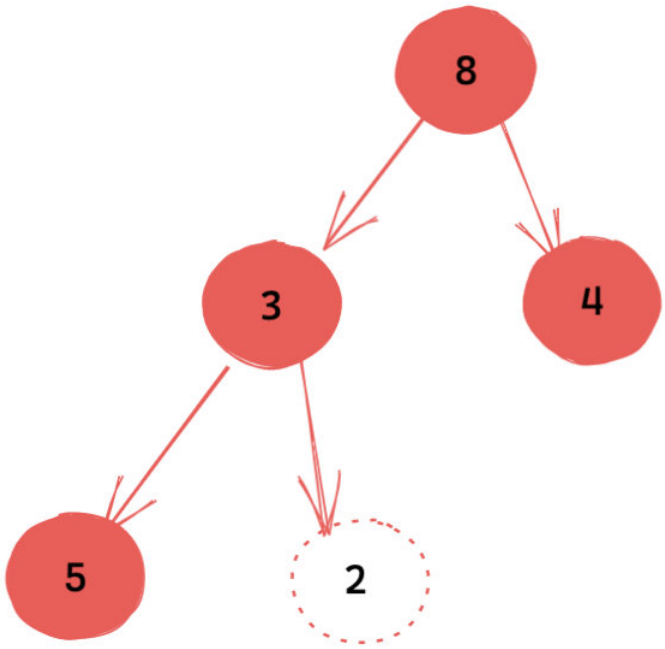
如果仅仅是删除，那么一个堆就会变成两个堆了，问题变复杂了。



（上图出堆之后会生成两个新的堆）

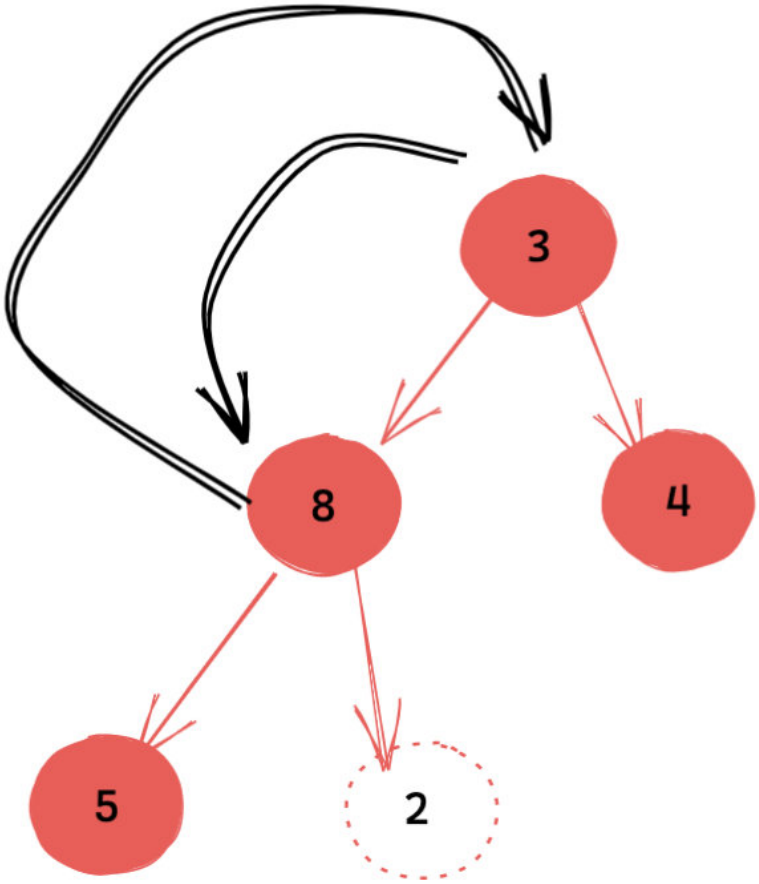
一个常见的操作是，把根结点和最后一个结点交换。但是新的根结点可能不满足 **父节点的权值不大于儿子的权值（小顶堆）** 。

如下图，我们将根节点的 2 和尾部的数字进行交换后，这个时候是不满足堆性质的。

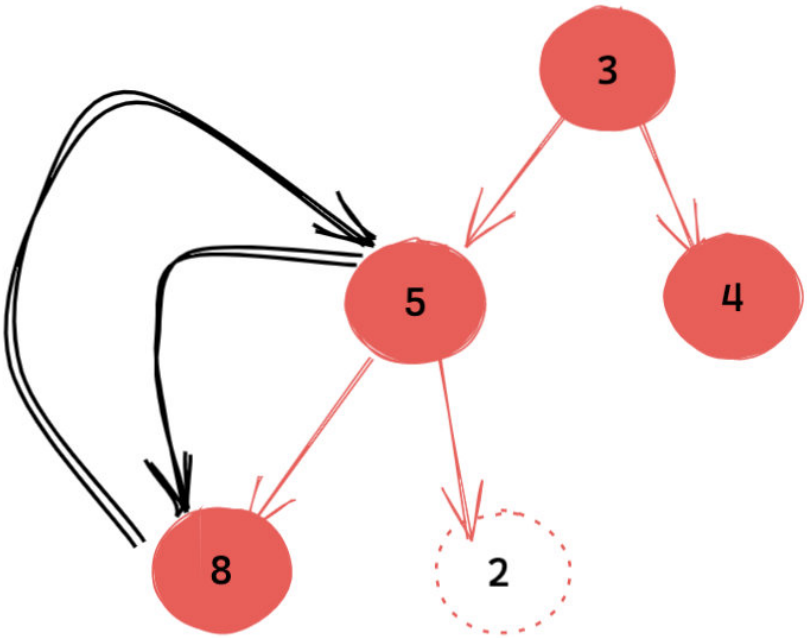


这个时候，其实只需要将新的根节点下沉到正确位置即可。这里的**正确位置**，指的还是那句话**父节点的权值不大于儿子的权值（小顶堆）** 。如果不满足这一点，我们就继续下沉，直到满足。

我们知道根节点往下下沉的过程，其实有两个方向可供选择，是下沉到左子节点？还是下沉到右子节点？以小顶堆来说，答案应该是下沉到较小的子节点处，否则会错失正确答案。以上面的堆为例，如果下沉到右子节点 4，那么就无法得到正确的堆顶 3。因此我们需要下沉到左子节点。



下沉到如图位置，还是不满足 **父节点的权值不大于儿子的权值（小顶堆）**，于是我们继续执行同样的操作。



有的同学可能有疑问。弹出根节点前堆满足堆的性质，但是弹出之后经过你上面讲的下沉操作，一定还满足么？

答案是肯定的。这个也不难理解。由于最后的叶子节点被提到了根节点，它其实最终在哪是不确定的，但是经过上面的操作，我们可以看出：

- 其下沉路径上的节点一定都满足堆的性质。
- 不在下沉路径上的节点都保持了堆之前的相对关系，因此也满足堆的性质。

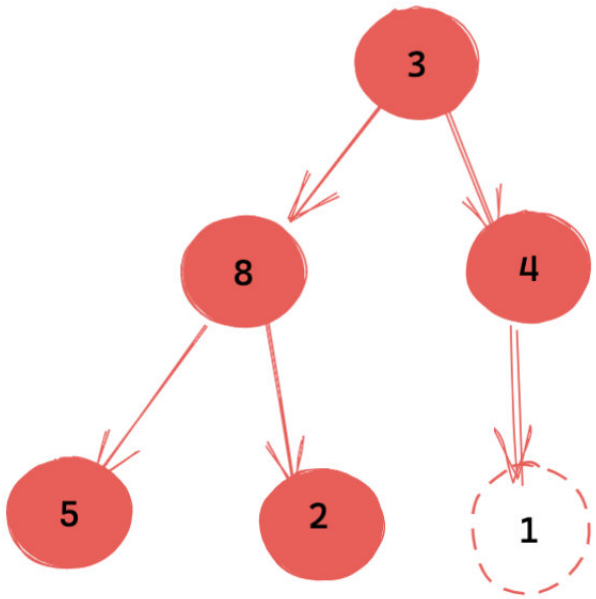
因此**弹出根节点后，经过上面的下沉操作一定仍然满足堆的性质**。

时间复杂度方面可以证明，下沉和树的高度成正相关，因此时间复杂度为 $logh$，其中 h 为树高。而由于二叉堆是一颗完全二叉树，因此树高大约是 $logN$，其中 N 为树中的节点个数。

入堆

入堆和出堆类似。我们可以直接往树的最后插入一个节点。和上面类似，这样的操作同样可能会破坏堆的性质。

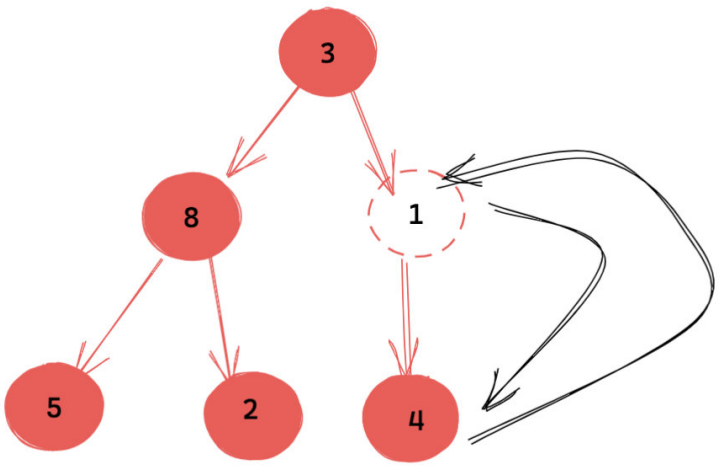
> 之所以这么做的其中一个原因是时间复杂度更低，因为我们是用数组进行模拟的，而在数组尾部添加元素的时间复杂度为 $O(1)$。



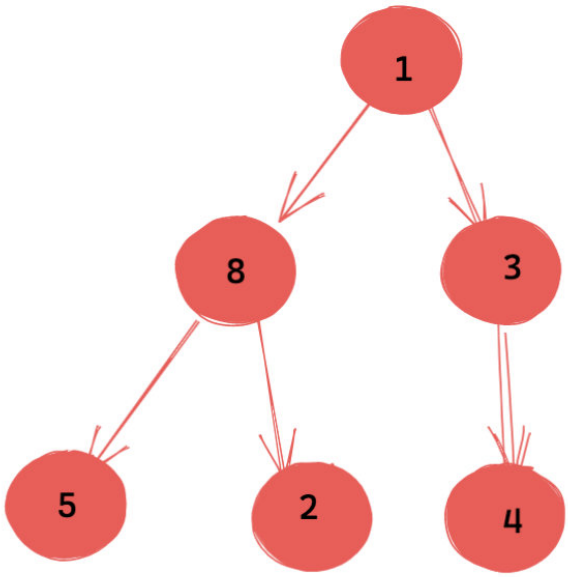
这次我们发现，不满足堆的节点目前是刚刚被插入节点的尾部节点，因此不能进行下沉操作了。这一次我们需要执行**上浮操作**。

> 叶子节点是只能上浮的（根节点只能下沉，其他节点既可以下沉，又可以上浮）

和上面基本类似，如果不满足堆的性质，我们将其和父节点交换（上浮），继续这个过程，直到满足堆的性质。



(第一次上浮，仍然不满足堆特性，继续上浮)



（满足了堆特性，上浮过程完毕）

经过这样的操作，其还是一个满足堆性质的堆。证明过程和上面类似，不再赘述。

需要注意的是，由于上浮**只需要拿当前节点和父节点进行比对就可以了，**由于省去了判断左右子节点哪个更小的过程，因此更加简单。

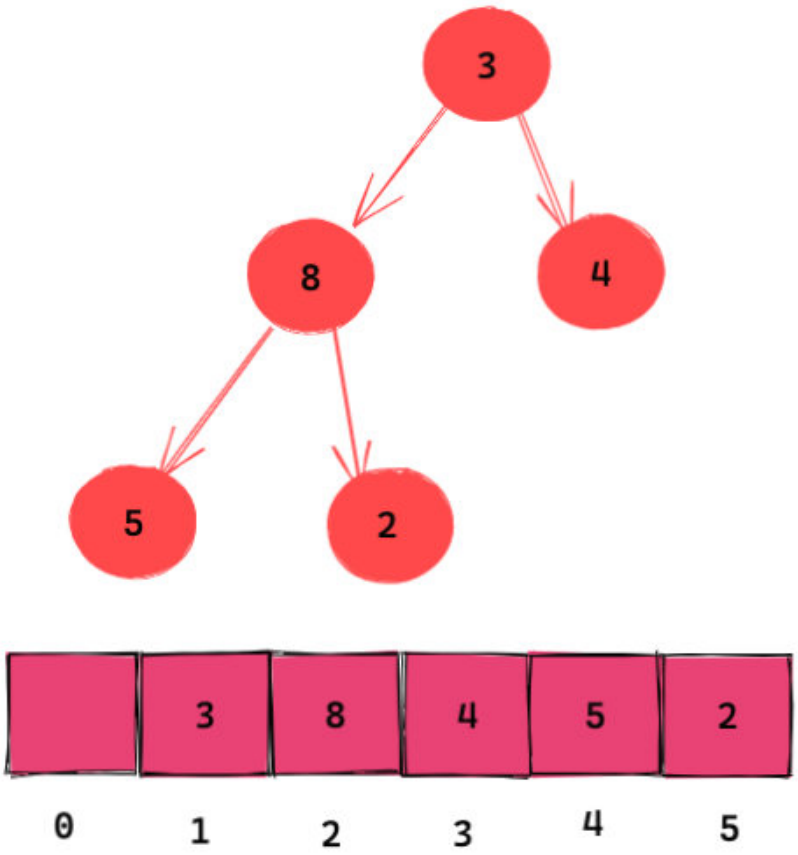
实现

对于完全二叉树来说使用数组实现非常方便。因为：

- 如果节点在数组中的下标为 i，那么其左子节点下标为 $2 \times i$，右节点为 $2 \times i$+1。
- 如果节点在数组中的下标为 i，那么父节点下标为 i//2（地板除）。

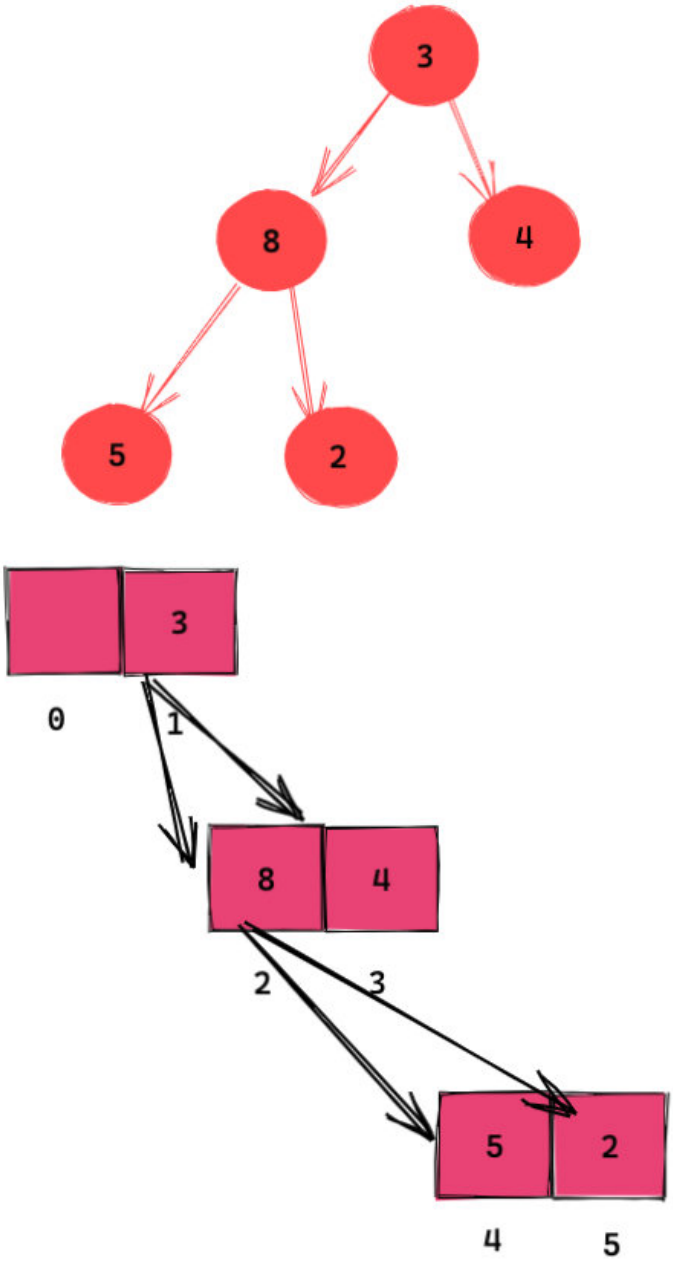
当然这要求你的**数组从 1 开始存储数据**。如果不是，上面的公式其实微调一下也可以达到同样的效果。不过这是一种业界习惯，我们还是和业界保持一致比较好。从 1 开始存储的另外一个好处是，我们可以将索引为 0 的位置空出来存储诸如**堆大小**的信息，这是一些大学教材里的做法，大家作为了解即可。

如图所示是一个完全二叉树和树的数组表示法。



（注意数组索引的对应关系）

形象点来看，我们可以可以画出如下的对应关系图：



这样一来，是不是和上面的树差不多一致了？有没有容易理解一点呢?

上面已经讲了上浮和下沉的过程。刚才也讲了父子节点坐标的关系。那么代码就呼之欲出了。我们来下最核心的**上浮**和**下沉**的代码实现吧。

伪代码：

```
// x 是要上浮的元素，从树的底部开始上浮
private void shift_up(int x) {
  while (x > 1 && h[x] > h[x / 2]) {
    //  swqp 就是交换数组两个位置的值
    swap(h[x], h[x / 2]);
    x /= 2;
  }
}
// x 是要下沉的元素，从树的顶部开始下沉
private void shift_down(int x) {
  while (x * 2 <= n) {
    // minChild 是获取更小的子节点的索引并返回
    mc = minChild(x);
    if (h[mc] <= h[x]) break;
    swap(h[x], h[mc]);
    x = mc;
  }
}
```

这里 Java 语言为例，讲述一下代码的编写。其他语言的二叉堆实现可以去我的**刷题插件 leetcode-cheatsheet** 中获取。插件的获取方式在公众号**力扣加加**里，回复插件即可。

```
import java.util.Arrays;
import java.util.Comparator;

/**
 * 用完全二叉树来构建 堆
 * 前置条件 起点为 1
 * 那么 子节点为  i <<1 和 i<<1 + 1
 * 核心方法为
 * shiftdown 交换下沉
 * shiftup 交换上浮
 * <p>
 * build 构建堆
 */

public class Heap {

    int size = 0;
    int queue[];

    public Heap(int initialCapacity) {
        if (initialCapacity < 1)
            throw new IllegalArgumentException();
        this.queue = new int[initialCapacity];
    }

    public Heap(int[] arr) {
        size = arr.length;
        queue = new int[arr.length + 1];
        int i = 1;
        for (int val : arr) {
            queue[i++] = val;
        }
    }

    public void shiftDown(int i) {

        int temp = queue[i];

        while ((i << 1) <= size) {
            int child = i << 1;
            // child!=size 判断当前元素是否包含右节点
            if (child != size && queue[child + 1] < queue[
                child++;
            }
            if (temp > queue[child]) {
                queue[i] = queue[child];
                i = child;
            } else {
```

```
                break;
            }
        }
        queue[i] = temp;
    }


    public void shiftUp(int i) {
        int temp = queue[i];
        while ((i >> 1) > 0) {
            if (temp < queue[i >> 1]) {
                queue[i] = queue[i >> 1];
                i >>= 1;
            } else {
                break;
            }
        }
        queue[i] = temp;
    }

    public int peek() {

        int res = queue[1];
        return res;
    }

    public int pop() {

        int res = queue[1];

        queue[1] = queue[size--];
        shiftDown(1);
        return res;
    }

    public void push(int val) {
        if (size == queue.length - 1) {
            queue = Arrays.copyOf(queue, size << 1+1);
        }
        queue[++size] = val;
        shiftUp(size);
    }

    public void buildHeap() {
        for (int i = size >> 1; i > 0; i--) {
            shiftDown(i);
        }
    }
```

```
    public static void main(String[] args) {

        int arr[] = new int[]{2,7,4,1,8,1};
        Heap heap = new Heap(arr);
        heap.buildHeap();
        System.out.println(heap.peek());
        heap.push(5);
        while (heap.size > 0) {
            int num = heap.pop();
            System.out.printf(num + "");
        }
    }
}
```

小结

堆的实现有很多。比如基于链表的跳表，基于数组的二叉堆和基于红黑树的实现等。这里我们详细地讲述了二叉堆的实现，不仅是其实现简单，而且其在很多情况下表现都不错，推荐大家重点掌握二叉堆实现。

对于二叉堆的实现，核心点就一点，那就是始终维护堆的性质不变，具体是什么性质呢？那就是 **父节点的权值不大于儿子的权值（小顶堆）** 。为了达到这个目的，我们需要在入堆和出堆的时候，使用上浮和下沉操作，并恰当地完成元素交换。具体来说就是上浮过程和比它大的父节点进行交换，下沉过程和两个子节点中较小的进行交换，当然前提是它有子节点且子节点比它小。

关于堆化我们并没有做详细分析。不过如果你理解了本文的入堆操作，这其实很容易。因此堆化本身就是一个不断入堆的过程，只不过**将时间上的离散的操作变成了一次性操作**而已。

预告

本文预计分两个部分发布。这是第一部分，后面的内容更加干货，分别是**三个技巧**和**四大应用**。

- 三个技巧
- 多路归并
- 固定堆
- 事后小诸葛
- 四大应用
- topK

- 带权最短距离
- 因子分解
- 堆排序

这两个主题是专门教你怎么解题的。掌握了它，力扣中的大多数堆的题目都不在话下（当然我指的仅仅是题目中涉及到堆的部分）。

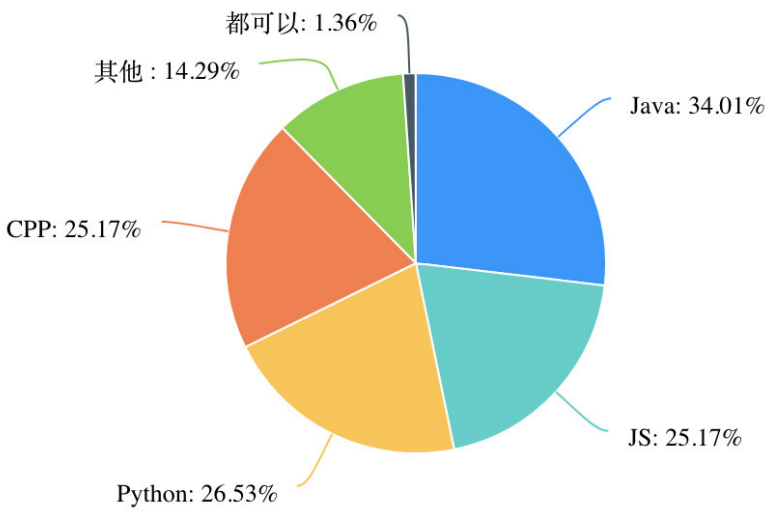
大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

几乎刷完了力扣所有的堆题，我发现了这些东西。。。（第二弹）

一点题外话

上次在我的公众号给大家做了一个小调查《投出你想要的题解编程语言吧~》。以下是调查的结果：



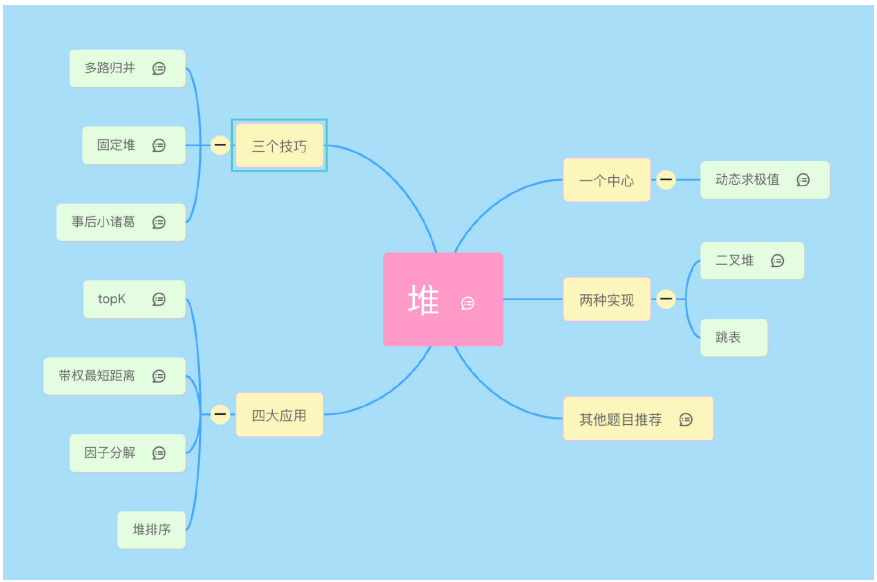
而关于其他，则大多数是 Go 语言。

序号	提交答卷时间	答案文本	查看答卷
2	1月12日 18:02	golang	查看答卷
3	1月12日 18:02	go	查看答卷
4	1月12日 18:03	zig	查看答卷
8	1月12日 18:03	go	查看答卷
13	1月12日 18:05	kotlin	查看答卷
15	1月12日 18:06	JavaScript	查看答卷
17	1月12日 18:06	一般大一都会教C++，来个C++大家都看得懂把	查看答卷
27	1月12日 18:11	go	查看答卷
28	1月12日 18:11	go	查看答卷
29	1月12日 18:12	lisp	查看答卷

由于 Java 和 Python 所占比例已经超过了 60%，这次我尝试一下 Java 和 Python 双语言来写，感谢 @CaptainZ 提供的 Java 代码。同时为了**不让文章又臭又长，我将 Java 本文所有代码（Java 和 Python）都放到了力扣加加官网上**，网站地址：https://leetcode-solution.cn/solution-code

> 如果不科学上网的话，可能打开会很慢。

正文



大家好，我是 lucifer。今天给大家带来的是《堆》专题。先上下本文的提纲，这个是我用 mindmap 画的一个脑图，之后我会继续完善，将其他专题逐步完善起来。

> 大家也可以使用 vscode blink-mind 打开源文件查看，里面有一些笔记可以点开查看。源文件可以去我的公众号《力扣加加》回复脑图获取，以后脑图也会持续更新更多内容。vscode 插件地址：https://marketplace.visualstudio.com/items?itemName=awehook.vscode-blink-mind

本系列包含以下专题：

- 几乎刷完了力扣所有的链表题，我发现了这些东西。。。
- 几乎刷完了力扣所有的树题，我发现了这些东西。。。
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（第一弹）

本次是下篇，没有看过上篇的同学强烈建议先阅读上篇几乎刷完了力扣所有的堆题，我发现了这些东西。。。(第一弹)

这是第二部分，后面的内容更加干货，分别是**三个技巧**和**四大应用**。这两个主题是专门教你怎么解题的。掌握了它，力扣中的大多数堆的题目都不在话下（当然我指的仅仅是题目中涉及到堆的部分）。

警告： 本章的题目基本都是力扣 hard 难度，这是因为堆的题目很多标记难度都不小，关于这点在前面也介绍过了。

一点说明

在上主菜之前，先给大家来个开胃菜。

这里给大家介绍两个概念，分别是**元组**和**模拟大顶堆** 。之所以进行这些说明就是防止大家后面看不懂。

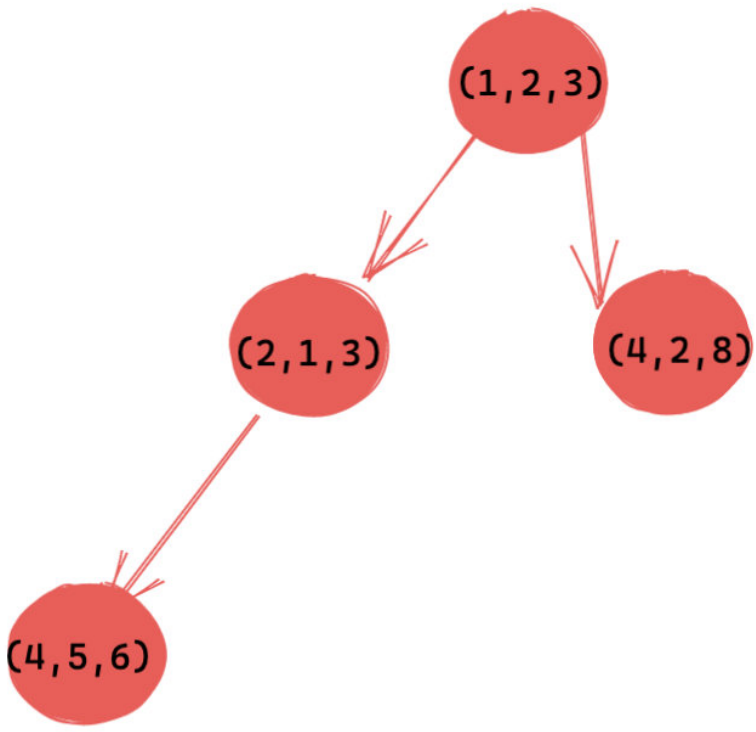
元组

使用堆不仅仅可以存储单一值，比如 [1,2,3,4] 的 1，2，3，4 分别都是单一值。除了单一值，也可以存储复合值，比如对象或者元组等。

这里我们介绍一种存储元组的方式，这个技巧会在后面被广泛使用，请务必掌握。比如 [(1,2,3), (4,5,6), (2,1,3),(4,2,8)]。

```
h = [(1,2,3), (4,5,6), (2,1,3),(4,2,8)]
heapq.heappify(h) # 堆化（小顶堆）

heapq.heappop() # 弹出（1,2,3）
heapq.heappop() # 弹出（2,1,3）
heapq.heappop() # 弹出（4,2,8）
heapq.heappop() # 弹出（4,5,6）
```

用图来表示堆结构就是下面这样：



使用元组的小顶堆

简单解释一下上面代码的执行结果。

使用元组的方式，默认将元组第一个值当做键来比较。如果第一个相同，继续比较第二个。比如上面的 (4,5,6) 和 (4,2,8)，由于第一个值相同，因此继续比较后一个，又由于 5 比 2 大，因此 (4,2,8)先出堆。

使用这个技巧有两个作用：

1. 携带一些额外的信息。 比如我想求二维矩阵中第 k 小数，当然是以值作为键。但是处理过程又需要用到其行和列信息，那么使用元组就很合适，比如 (val, row, col)这样的形式。
2. 想根据两个键进行排序，一个主键一个副键。这里面又有两种典型的用法，

   2.1 一种是两个都是同样的顺序，比如都是顺序或者都是逆序。

   2.2 另一种是两个不同顺序排序，即一个是逆序一个是顺序。

由于篇幅原因，具体就不再这里展开了，大家在平时做题过程中留意可以一下，有机会我会单独开一篇文章讲解。

> 如果你所使用的编程语言没有堆或者堆的实现不支持元组，那么也可以通过简单的改造使其支持，主要就是自定义比较逻辑即可。

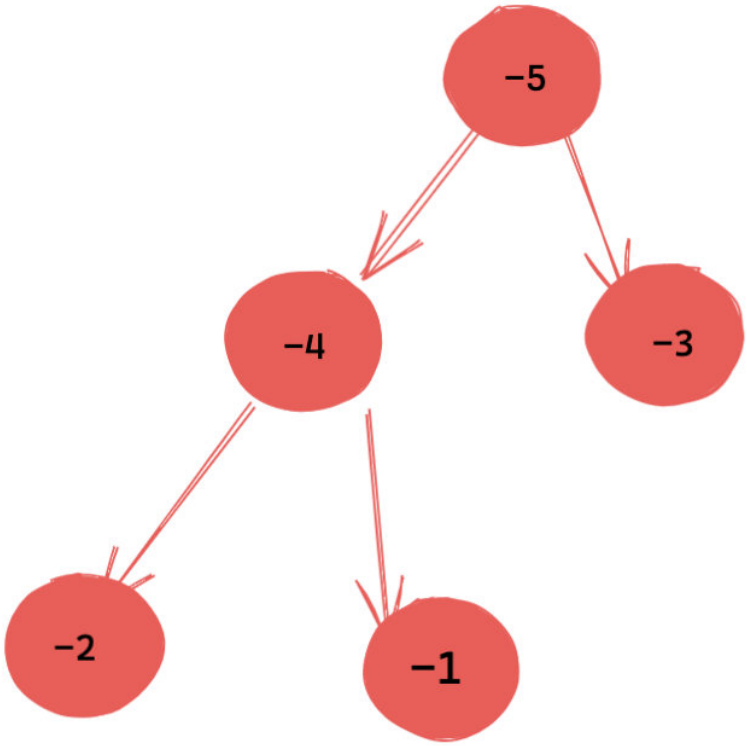
模拟大顶堆

由于 Python 没有大顶堆。因此我这里使用了小顶堆进行模拟实现。即将原有的数全部取相反数，比如原数字是 5，就将 -5 入堆。经过这样的处理，小顶堆就可以当成大顶堆用了。不过需要注意的是，当你 pop 出来的时候， **记得也要取反，将其还原回来**哦。

代码示例：

```
h = []
A = [1,2,3,4,5]
for a in A:
    heapq.heappush(h, -a)
-1 * heapq.heappop(h) # 5
-1 * heapq.heappop(h) # 4
-1 * heapq.heappop(h) # 3
-1 * heapq.heappop(h) # 2
-1 * heapq.heappop(h) # 1
```

用图来表示就是下面这样：



小顶堆模拟大顶堆

铺垫就到这里，接下来进入正题。

三个技巧

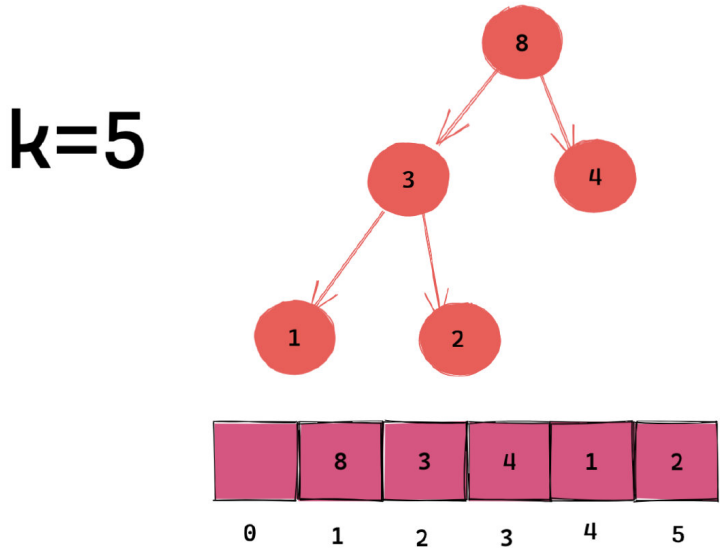
技巧一 - 固定堆

这个技巧指的是固定堆的大小 k 不变，代码上可通过**每 pop 出去一个就 push 进来一个**来实现。而由于初始堆可能是 0，我们刚开始需要一个一个 push 进堆以达到堆的大小为 k，因此严格来说应该是**维持堆的大小不大于 k**。

固定堆一个典型的应用就是求第 k 小的数。其实求第 k 小的数最简单的思路是建立小顶堆，将所有的数**先全部入堆，然后逐个出堆，一共出堆 k 次**。最后一次出堆的就是第 k 小的数。

然而，我们也可不先全部入堆，而是建立**大顶堆**（注意不是上面的小顶堆），并维持堆的大小为 k 个。如果新的数入堆之后堆的大小大于 k，则需要将堆顶的数和新的数进行比较，**并将较大的移除**。这样可以保证**堆中的数是全体数字中最小的 k 个**，而这最小的 k 个中最大的（即堆顶）不就是第 k 小的么？这也就是选择建立大顶堆，而不是小顶堆的原因。

堆是最小的 k 个数， 堆顶又是最大的，因此堆顶就是第 k 小的



简单一句话总结就是**固定一个大小为 k 的大顶堆可以快速求第 k 小的数，反之固定一个大小为 k 的小顶堆可以快速求第 k 大的数**。比如力扣 2020-02-24 的周赛第三题5663. 找出第 K 大的异或坐标值就可以用固定小顶堆技巧来实现（这道题让你求第 k 大的数）。

这么说可能你的感受并不强烈，接下来我给大家举两个例子来帮助大家加深印象。

295. 数据流的中位数

题目描述

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

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:

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?
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?

思路

这道题实际上可看出是求第 k 小的数的特例了。

- 如果列表长度是奇数，那么 k 就是 (n + 1) / 2，中位数就是第 k 个数，。比如 n 是 5， k 就是 (5 + 1)/ 2 = 3。
- 如果列表长度是偶数，那么 k 就是 (n + 1) / 2 和 (n + 1) / 2 + 1，中位数则是这两个数的平均值。比如 n 是 6， k 就是 (6 + 1)/ 2 = 3 和 (6 + 1) / 2 + 1 = 4。

因此我们的可以维护两个固定堆，固定堆的大小为 $(n + 1) \div 2$ 和 $n - (n + 1)\div2$，也就是两个堆的大小**最多**相差 1，更具体的就是 $ 0 <= (n + 1) \div 2 - (n - (n + 1) \div 2) <= 1$。

基于上面提到的知识，我们可以：

- 建立一个大顶堆，并存放最小的 $(n + 1) \div 2$ 个数，这样堆顶的数就是第 $(n + 1) \div 2$ 小的数，也就是奇数情况的中位数。
- 建立一个小顶堆，并存放最大的 n - $(n + 1) \div 2$ 个数，这样堆顶的数就是第 n - $(n + 1) \div 2$ 大的数，结合上面的大顶堆，可求出偶数情况的中位数。

有了这样一个知识，剩下的只是如何维护两个堆的大小了。

- 如果大顶堆的个数比小顶堆少，那么就将小顶堆中最小的转移到大顶堆。而由于小顶堆维护的是最大的 k 个数，大顶堆维护的是最小的 k 个数，因此小顶堆堆顶一定大于等于大顶堆堆顶，并且这两个堆顶是**此时**的中位数。
- 如果大顶堆的个数比小顶堆的个数多 2，那么就将大顶堆中最大的转移到小顶堆，理由同上。

至此，可能你已经明白了为什么分别建立两个堆，并且需要一个大顶堆一个小顶堆。这其中的原因正如上面所描述的那样。

固定堆的应用常见还不止于此，我们继续看一道题。

代码

```
class MedianFinder:
    def __init__(self):
        self.min_heap = []
        self.max_heap = []
    def addNum(self, num: int) -> None:
        if not self.max_heap or num < -self.max_heap[0]:
            heapq.heappush(self.max_heap, -num)
        else:
            heapq.heappush(self.min_heap, num)
        if len(self.max_heap) > len(self.min_heap) + 1:
            heappush(self.min_heap, -heappop(self.max_heap)
        elif len(self.min_heap) > len(self.max_heap):
            heappush(self.max_heap, -heappop(self.min_heap)
    def findMedian(self) -> float:
        if len(self.min_heap) == len(self.max_heap): retur
        return -self.max_heap[0]
```

(代码 1.3.1)

857. 雇佣 K 名工人的最低成本

题目描述

```
有 N 名工人。 第 i 名工人的工作质量为 quality[i] ，其最低期望工资为

现在我们想雇佣 K 名工人组成一个工资组。在雇佣 一组 K 名工人时，我们必

对工资组中的每名工人，应当按其工作质量与同组其他工人的工作质量的比例来
工资组中的每名工人至少应当得到他们的最低期望工资。
返回组成一个满足上述条件的工资组至少需要多少钱。



示例 1：

输入： quality = [10,20,5], wage = [70,50,30], K = 2
输出： 105.00000
解释： 我们向 0 号工人支付 70，向 2 号工人支付 35。
示例 2：

输入： quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3
输出： 30.66667
解释： 我们向 0 号工人支付 4，向 2 号和 3 号分别支付 13.33333。


提示：

1 <= K <= N <= 10000，其中 N = quality.length = wage.length
1 <= quality[i] <= 10000
1 <= wage[i] <= 10000
与正确答案误差在 10^-5 之内的答案将被视为正确的。
```

思路

题目要求我们选择 k 个人，按其工作质量与同组其他工人的工作质量的比例来支付工资，并且工资组中的每名工人至少应当得到他们的最低期望工资。

换句话说，同一组的 k 个人他们的工作质量和工资比是一个固定值才能使支付的工资最少。请先理解这句话，后面的内容都是基于这个前提产生的。

我们不妨定一个指标**工作效率**，其值等于 q / w。前面说了这 k 个人的 q / w 是相同的才能保证工资最少，并且这个 q / w 一定是这 k 个人最低的（短板），否则一定会有人得不到最低期望工资。

于是我们可以写出下面的代码：

```python
class Solution:
    def mincostToHireWorkers(self, quality: List[int], wage
        eff = [(q / w, q, w) for a, b in zip(quality, wage)
        eff.sort(key=lambda a: -a[0])
        ans = float('inf')
        for i in range(K-1, len(eff)):
            h = []
            k = K - 1
            rate, _, total = eff[i]
            # 找出工作效率比它高的 k 个人，这 k 个人的工资尽可能低
            # 由于已经工作效率倒序排了，因此前面的都是比它高的，然后
            for j in range(i):
                heapq.heappush(h, eff[j][1] / rate)
            while k > 0:
                total += heapq.heappop(h)
                k -= 1
            ans = min(ans, total)
        return ans
```

(代码 1.3.2)

这种做法每次都 push 很多数，并 pop k 次，并没有很好地利用堆的**动态**特性，而只利用了其**求极值**的特性。

一个更好的做法是使用**固定堆技巧**。

这道题可以换个角度思考。其实这道题不就是让我们选 k 个人，工作效率比取他们中最低的，并按照这个最低的工作效率计算总工资，找出最低的总工资么？ 因此这道题可以固定一个大小为 k 的大顶堆，通过一定操作保证堆顶的就是第 k 小的（操作和前面的题类似）。

并且前面的解法中堆使用了三元组 (q / w, q, w)，实际上这也没有必要。因为已知其中两个，可推导出另外一个，因此存储两个就行了，而又由于我们需要**根据工作效率比做堆的键**，因此任意选一个 q 或者 w 即可，这里我选择了 q，即存 (q/2, q) 二元组。

具体来说就是：以 rate 为最低工作效率比的 k 个人的总工资 = $\displaystyle \sum*{n=1}^{k}{q}{n}/rate$，这里的 rate 就是当前的 q / w，同时也是 k 个人的 q / w 的最小值。

代码

```
class Solution:
    def mincostToHireWorkers(self, quality: List[int], wage
        effs = [(q / w, q) for q, w in zip(quality, wage)]
        effs.sort(key=lambda a: -a[0])
        ans = float('inf')
        h = []
        total = 0
        for rate, q in effs:
            heapq.heappush(h, -q)
            total += q
            if len(h) > K:
                total += heapq.heappop(h)
            if len(h) == K:
                ans = min(ans, total / rate)
        return ans
```

(代码 1.3.3)

技巧二 - 多路归并

这个技巧其实在前面讲**超级丑数**的时候已经提到了，只是没有给这种类型的题目一个**名字**。

其实这个技巧，叫做多指针优化可能会更合适，只不过这个名字实在太过朴素且容易和双指针什么的混淆，因此我给 ta 起了个别致的名字 - **多路归并**。

- 多路体现在：有多条候选路线。代码上，我们可使用多指针来表示。
- 归并体现在：结果可能是多个候选路线中最长的或者最短，也可能是第 k 个 等。因此我们需要对多条路线的结果进行比较，并根据题目描述舍弃或者选取某一个或多个路线。

这样描述比较抽象，接下来通过几个例子来加深一下大家的理解。

这里我给大家精心准备了**四道难度为 hard** 的题目。 掌握了这个套路就可以去快乐地 AC 这四道题啦。

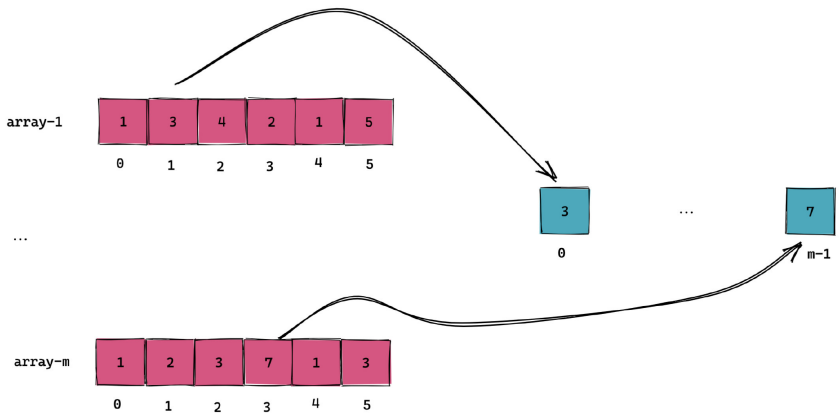
1439. 有序矩阵中的第 k 个最小数组和

题目描述

```
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

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



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,
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
```

思路

其实这道题就是给你 m 个长度均相同的一维数组，让我们从这 m 个数组中分别选出一个数，即一共选取 m 个数，求这 m 个数的和是**所有选取可能性**中和第 k 小的。



一个朴素的想法是使用多指针来解。对于这道题来说就是使用 m 个指针，分别指向 m 个一维数组，指针的位置表示当前选取的是该一维数组中第几个。

以题目中的 `mat = [[1,3,11],[2,4,6]], k = 5` 为例。

- 先初始化两个指针 p1,p2，分别指向两个一维数组的开头，代码表示就是全部初始化为 0。
- 此时两个指针指向的数字和为 1 + 2 = 3，这就是第 1 小的和。
- 接下来，我们移动其中一个指针。此时我们可以移动 p1，也可以移动 p2。
- 那么第 2 小的一定是移动 p1 和 移动 p2 这两种情况的较小值。而这里移动 p1 和 p2 实际上都会得到 5，也就是说第 2 和第 3 小的和都是 5。

到这里已经分叉了，出现了两种情况(注意看粗体的位置，粗体表示的是指针的位置)：

1. [1,**3**,11],[**2**,4,6] 和为 5
2. [**1**,3,11],[2,**4**,6] 和为 5

接下来，这两种情况应该**齐头并进，共同进行下去**。

对于情况 1 来说，接下来移动又有两种情况。

1. [1,3,**11**],[**2**,4,6] 和为 13
2. [1,**3**,11],[2,**4**,6] 和为 7

对于情况 2 来说，接下来移动也有两种情况。

1. [1,**3**,11],[2,**4**,6] 和为 7
2. [**1**,3,11],[2,4,**6**] 和为 7

我们通过比较这四种情况，得出结论： 第 4，5，6 小的数都是 7。但第 7 小的数并不一定是 13。原因和上面类似，可能第 7 小的就隐藏在前面的 7 分裂之后的新情况中，实际上确实如此。因此我们需要继续执行上述逻辑。

进一步，我们可以将上面的思路拓展到一般情况。

上面提到了题目需要求的其实是第 k 小的和，而最小的我们是容易知道的，即所有的一维数组首项和。我们又发现，根据最小的，我们可以推导出第 2 小，推导的方式就是移动其中一个指针，这就一共分裂出了 n 种情况了，其中 n 为一维数组长度，第 2 小的就在这分裂中的 n 种情况中，而筛选的方式是这 n 种情况和**最小**的，后面的情况也是类似。不难看出每次分裂之后极值也发生了变化，因此这是一个明显的求动态求极值的信号，使用堆是一个不错的选择。

那代码该如何书写呢?

上面说了，我们先要初始化 m 个指针，并赋值为 0。对应伪代码：

```
# 初始化堆
h = []
# sum(vec[0] for vec in mat) 是 m 个一维数组的首项和
# [0] * m 就是初始化了一个长度为 m 且全部填充为 0 的数组。
# 我们将上面的两个信息组装成元祖 cur 方便使用
cur = (sum(vec[0] for vec in mat), [0] * m)
# 将其入堆
heapq.heappush(h, cur)
```

接下来，我们每次都移动一个指针，从而形成分叉出一条新的分支。每次从堆中弹出一个最小的，弹出 k 次就是第 k 小的了。伪代码：

```
for 1 to K:
    # acc 当前的和，  pointers 是指针情况。
    acc, pointers = heapq.heappop(h)
    # 每次都粗暴地移动指针数组中的一个指针。每移动一个指针就分叉一次,
    for i, pointer in enumerate(pointers):
        # 如果 pointer == len(mat[0]) - 1 说明到头了，不能移动
        if pointer != len(mat[0]) - 1:
            # 下面两句话的含义是修改 pointers[i] 的指针 为 poin
            new_pointers = pointers.copy()
            new_pointers[i] += 1
            # 将更新后的 acc 和指针数组重新入堆
            heapq.heappush(h, (acc + mat[i][pointer + 1] -
```

这是**多路归并**问题的核心代码，请务必记住。

> 代码看起来很多，其实去掉注释一共才七行而已。

上面的伪代码有一个问题。比如有两个一维数组，指针都初始化为 0。第一次移动第一个一维数组的指针，第二次移动第二个数组的指针，此时指针数组为 [1, 1]，即全部指针均指向下标为 1 的元素。而如果第一次移动

第二个一维数组的指针，第二次移动第一个数组的指针，此时指针数组仍然为 [1, 1]。这实际上是一种情况，如果不加控制会被计算两次导致出错。

一个可能的解决方案是使用 hashset 记录所有的指针情况，这样就避免了同样的指针被计算多次的问题。为了做到这一点，我们需要对指针数组的使用做一些微调，即使用元组代替数组。原因在于数组是无法直接哈希化的。具体内容请参考代码区。

多路归并的题目，思路和代码都比较类似。为了后面的题目能够更高地理解，请务必搞定这道题，后面我们将不会这么详细地进行分析。

代码

```
class Solution:
    def kthSmallest(self, mat, k: int) -> int:
        h = []
        cur = (sum(vec[0] for vec in mat), tuple([0] * len(
        heapq.heappush(h, cur)
        seen = set(cur)

        for _ in range(k):
            acc, pointers = heapq.heappop(h)
            for i, pointer in enumerate(pointers):
                if pointer != len(mat[0]) - 1:
                    t = list(pointers)
                    t[i] = pointer + 1
                    tt = tuple(t)
                    if tt not in seen:
                        seen.add(tt)
                        heapq.heappush(h, (acc + mat[i][poi
        return acc
```

(代码 1.3.4)

719. 找出第 k 小的距离对

题目描述

```
给定一个整数数组，返回所有数对之间的第 k 个最小距离。一对 (A, B) 的距

示例 1:

输入：
nums = [1,3,1]
k = 1
输出：0
解释：
所有数对如下：
(1,3) -> 2
(1,1) -> 0
(3,1) -> 2
因此第 1 个最小距离的数对是 (1,1)，它们之间的距离为 0。
提示:

2 <= len(nums) <= 10000.
0 <= nums[i] < 1000000.
1 <= k <= len(nums) * (len(nums) - 1) / 2.
```

思路

不难看出所有的数对可能共 C_n^2 个，也就是 $n\times(n-1)\div2$。

因此我们可以使用两次循环找出所有的数对，并升序排序，之后取第 k 个。

实际上，我们可使用固定堆技巧，维护一个大小为 k 的大顶堆，这样堆顶的元素就是第 k 小的，这在前面的固定堆中已经讲过，不再赘述。

```python
class Solution:
    def smallestDistancePair(self, nums: List[int], k: int)
        h = []
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                a, b = nums[i], nums[j]
                # 维持堆大小不超过 k
                if len(h) == k and -abs(a - b) > h[0]:
                    heapq.heappop(h)
                if len(h) < k:
                    heapq.heappush(h, -abs(a - b))

        return -h[0]
```

(代码 1.3.5)

不过这种优化意义不大，因为算法的瓶颈在于 N^2 部分的枚举，我们应当设法优化这一点。

如果我们将数对进行排序，那么最小的数对距离一定在 nums[i] - nums[i - 1] 中，其中 i 为从 1 到 n 的整数，究竟是哪个取决于谁更小。接下来就可以使用上面多路归并的思路来解决了。

如果 nums[i] - nums[i - 1] 的差是最小的，那么第 2 小的一定是剩下的 n - 1 种情况和 nums[i] - nums[i - 1] 分裂的新情况。关于如何分裂，和上面类似，我们只需要移动其中 i 的指针为 i + 1 即可。这里的指针数组长度固定为 2，而不是上面题目中的 m。这里我将两个指针分别命名为 fr 和 to，分别代表 from 和 to。

代码

```
class Solution(object):
    def smallestDistancePair(self, nums, k):
        nums.sort()
        # n 种候选答案
        h = [(nums[i+1] - nums[i], i, i+1) for i in range(
        heapq.heapify(h)

        for _ in range(k):
            diff, fr, to = heapq.heappop(h)
            if to + 1 < len(nums):
                heapq.heappush((nums[to + 1] - nums[fr], fr

        return diff
```

(代码 1.3.6)

由于时间复杂度和 k 有关，而 k 最多可能达到 N^2 的量级，因此此方法实际上也会超时。**不过这证明了这种思路的正确性，如果题目稍加改变说不定就能用上。**

这道题可通过二分法来解决，由于和堆主题有偏差，因此这里简单讲一下。

求第 k 小的数比较容易想到的就是堆和二分法。二分的原因在于求第 k 小，本质就是求不大于其本身的有 k - 1 个的那个数。而这个问题很多时候满足单调性，因此就可使用二分来解决。

以这道题来说，最大的数对差就是数组的最大值 - 最小值，不妨记为 max_diff。我们可以这样发问：

- 数对差小于 max_diff 的有几个?
- 数对差小于 max_diff - 1 的有几个?
- 数对差小于 max_diff - 2 的有几个?

- 数对差小于 max_diff - 3 的有几个?
- 数对差小于 max_diff - 4 的有几个?
- 。。。

而我们知道，发问的答案也是不严格递减的，因此使用二分就应该被想到。我们不断发问直到问到**小于 x 的有 k - 1 个**即可。然而这样的发问也有问题。原因有两个：

1. 小于 x 的有 k - 1 个的数可能不止一个
2. 我们无法确定小于 x 的有 k - 1 个的数一定存在。 比如数对差分别为 [1,1,1,1,2]，让你求第 3 大的，那么小于 x 有两个的数根本就不存在。

我们的思路可调整为求**小于等于 x** 有 k 个的，接下来我们使用二分法的最左模板即可解决。关于最左模板可参考我的二分查找专题

代码：

```
class Solution:
    def smallestDistancePair(self, A: List[int], K: int) ->
        A.sort()
        l, r = 0, A[-1] - A[0]

        def count_ngt(mid):
            slow = 0
            ans = 0
            for fast in range(len(A)):
                while A[fast] - A[slow] > mid:
                    slow += 1
                ans += fast - slow
            return ans

        while l <= r:
            mid = (l + r) // 2
            if count_ngt(mid) >= K:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

(代码 1.3.7)

632. 最小区间

题目描述

```
你有 k 个 非递减排列 的整数列表。找到一个 最小 区间，使得 k 个列表中
```

```
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]
```

```
示例 1：

输入：nums = [[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。
示例 2：

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

输入：nums = [[10,10],[11,11]]
输出：[10,11]
示例 4：

输入：nums = [[10],[11]]
输出：[10,11]
示例 5：

输入：nums = [[1],[2],[3],[4],[5],[6],[7]]
输出：[1,7]


提示：

nums.length == k
1 <= k <= 3500
1 <= nums[i].length <= 50
-105 <= nums[i][j] <= 105
nums[i] 按非递减顺序排列
```

思路

这道题本质上就是**在 m 个一维数组中各取出一个数字，重新组成新的数组 A，使得新的数组 A 中最大值和最小值的差值（diff）最小。**

这道题和上面的题目有点类似，又略有不同。这道题是一个矩阵，上面一道题是一维数组。不过我们可以将二维矩阵看出一维数组，这样我们就可以沿用上面的思路了。

上面的思路 diff 最小的一定产生于排序之后相邻的元素之间。而这道题我们无法直接对二维数组进行排序，而且即使进行排序，也不好确定排序的原则。

我们其实可以继续使用前面两道题的思路。具体来说就是使用**小顶堆获取堆中最小值**，进而通过**一个变量记录堆中的最大值**，这样就知道了 diff，每次更新指针都会产生一个新的 diff，不断重复这个过程并维护全局最小 diff 即可。

这种算法的成立的前提是 k 个列表都是升序排列的，这里需要数组升序原理和上面题目是一样的，有序之后就可以对每个列表维护一个指针，进而使用上面的思路解决。

以题目中的 nums = [[1,2,3],[1,2,3],[1,2,3]] 为例：

- [1,2,3]
- [1,2,3]
- [1,2,3]

我们先选取所有行的最小值，也就是 [1,1,1]，这时的 diff 为 0，全局最大值为 1，最小值也为 1。接下来，继续寻找备胎，看有没有更好的备胎供我们选择。

接下来的备胎可能产生于情况 1：

- [**1**,2,3]
- [**1**,2,3]
- [1,**2**,3] 移动了这行的指针，将其从原来的 0 移动一个单位到达 1。

或者情况 2：

- [**1**,2,3]
- [1,**2**,3]移动了这行的指针，将其从原来的 0 移动一个单位到达 1。
- [**1**,2,3]

。。。

这几种情况又继续分裂更多的情况，这个就和上面的题目一样了，不再赘述。

代码

```
class Solution:
    def smallestRange(self, martrix: List[List[int]]) -> Li
        l, r = -10**9, 10**9
        # 将每一行最小的都放到堆中，同时记录其所在的行号和列号，一共
        h = [(row[0], i, 0) for i, row in enumerate(martrix
        heapq.heapify(h)
        # 维护最大值
        max_v = max(row[0] for row in martrix)

        while True:
            min_v, row, col = heapq.heappop(h)
            # max_v - min_v 是当前的最大最小差值， r - l 为全局
            if max_v - min_v < r - l:
                l, r = min_v, max_v
            if col == len(martrix[row]) - 1: return [l, r]
            # 更新指针，继续往后移动一位
            heapq.heappush(h, (martrix[row][col + 1], row,
            max_v = max(max_v, martrix[row][col + 1])
```

(代码 1.3.8)

1675. 数组的最小偏移量

题目描述

给你一个由 n 个正整数组成的数组 nums 。

你可以对数组的任意元素执行任意次数的两类操作：

如果元素是 偶数 ，除以 2
例如，如果数组是 [1,2,3,4] ，那么你可以对最后一个元素执行此操作，使其
如果元素是 奇数 ，乘上 2
例如，如果数组是 [1,2,3,4] ，那么你可以对第一个元素执行此操作，使其变
数组的 偏移量 是数组中任意两个元素之间的 最大差值 。

返回数组在执行某些操作之后可以拥有的 最小偏移量 。

示例 1：

输入：nums = [1,2,3,4]
输出：1
解释：你可以将数组转换为 [1,2,3,2]，然后转换成 [2,2,3,2]，偏移量是
示例 2：

输入：nums = [4,1,5,20,3]
输出：3
解释：两次操作后，你可以将数组转换为 [4,2,5,5,3]，偏移量是 5 - 2 =
示例 3：

输入：nums = [2,10,8]
输出：3

提示：

n == nums.length
2 <= n <= 105
1 <= nums[i] <= 109

思路

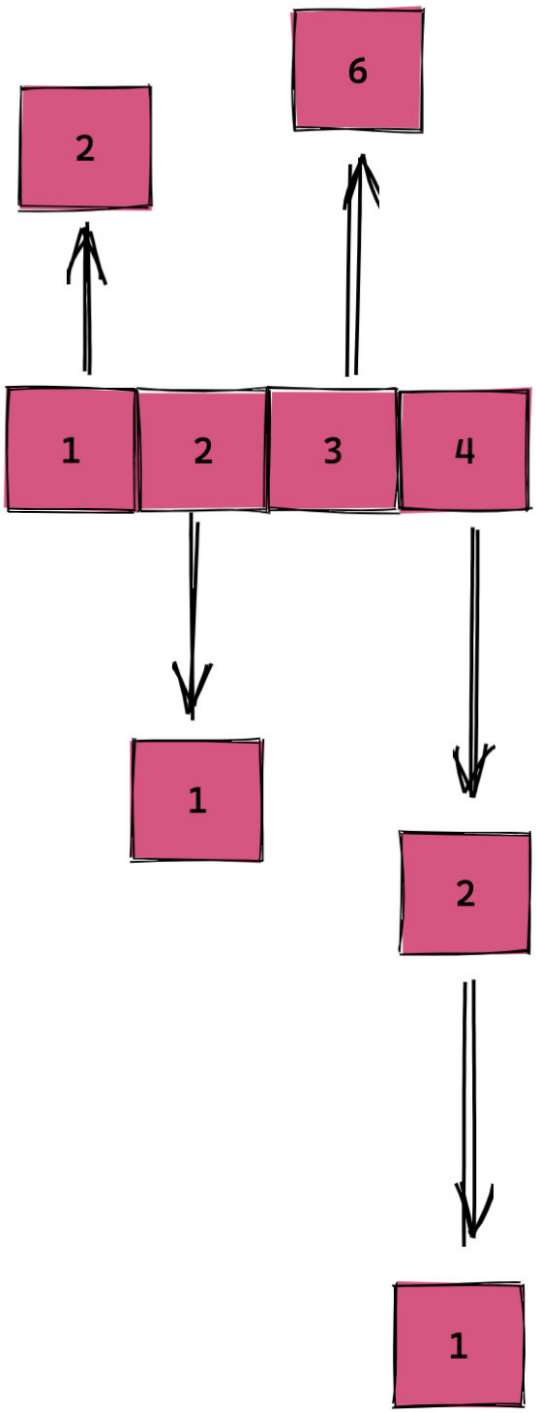
题目说可对数组中每一项都执行任意次操作，但其实操作是有限的。

- 我们只能对奇数进行一次 2 倍操作，因为 2 倍之后其就变成了偶数了。
- 我们可以对偶数进行若干次除 2 操作，直到等于一个奇数，不难看出这也是一个有限次的操作。

以题目中的 [1,2,3,4] 来说。我们可以：

- 将 1 变成 2（也可以不变）
- 将 2 变成 1（也可以不变）
- 将 3 变成 6（也可以不变）
- 将 4 变成 2 或 1（也可以不变）

用图来表示就是下面这样的：



这不就相当于: 从 [[1,2], [1,2], [3,6], [1,2,4]] 这样的一个二维数组中的每一行分别选取一个数，并使得其差最小么？这难道不是和上面的题目一模一样么？

这里我直接将上面的题目解法封装成了一个 api 调用了，具体看代码。

代码

```
class Solution:
    def smallestRange(self, martrix: List[List[int]]) -> Li
        l, r = -10**9, 10**9
        # 将每一行最小的都放到堆中，同时记录其所在的行号和列号，一共
        h = [(row[0], i, 0) for i, row in enumerate(martrix
        heapq.heapify(h)
        # 维护最大值
        max_v = max(row[0] for row in martrix)

        while True:
            min_v, row, col = heapq.heappop(h)
            # max_v - min_v 是当前的最大最小差值， r - l 为全局
            if max_v - min_v < r - l:
                l, r = min_v, max_v
            if col == len(martrix[row]) - 1: return [l, r]
            # 更新指针，继续往后移动一位
            heapq.heappush(h, (martrix[row][col + 1], row,
            max_v = max(max_v, martrix[row][col + 1])
    def minimumDeviation(self, nums: List[int]) -> int:
        matrix = [[] for _ in range(len(nums))]
        for i, num in enumerate(nums):
            if num & 1 == 1:
                matrix[i] += [num, num * 2]
            else:
                temp = []
                while num and num & 1 == 0:
                    temp += [num]
                    num //= 2
                temp += [num]
                matrix[i] += temp[::-1]
        a, b = self.smallestRange(matrix)
        return b - a
```

(代码 1.3.9)

技巧三 - 事后小诸葛

这个技巧指的是：当从左到右遍历的时候，我们是不知道右边是什么的，需要等到你到了右边之后才知道。

如果想知道右边是什么，一种简单的方式是遍历两次，第一次遍历将数据记录下来，当第二次遍历的时候，用上次遍历记录的数据。这是我们使用最多的方式。不过有时候，我们也可以在遍历到指定元素后，往前回溯，这样就可以边遍历边存储，使用一次遍历即可。具体来说就是将从左到右的数据全部收集起来，等到需要用的时候，从里面挑一个用。如果我们都要取最大值或者最小值且极值会发生变动， 就可**使用堆加速**。直观上就是使用了时光机回到之前，达到了事后诸葛亮的目的。

这样说**你肯定不明白啥意思**。没关系，我们通过几个例子来讲一下。当你看完这些例子之后，再回头看这句话。

871. 最低加油次数

题目描述

汽车从起点出发驶向目的地，该目的地位于出发位置东面 target 英里处。

沿途有加油站，每个 station[i] 代表一个加油站，它位于出发位置东面 sta

假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。它每行驶 1

当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。

为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则

注意：如果汽车到达加油站时剩余燃料为 0，它仍然可以在那里加油。如果汽车

示例 1:

```
输入：target = 1, startFuel = 1, stations = []
输出：0
解释：我们可以在不加油的情况下到达目的地。
```

示例 2:

```
输入：target = 100, startFuel = 1, stations = [[10,100]]
输出：-1
解释：我们无法抵达目的地，甚至无法到达第一个加油站。
```

示例 3:

```
输入：target = 100, startFuel = 10, stations = [[10,60],[20,
输出：2
解释：
我们出发时有 10 升燃料。
我们开车来到距起点 10 英里处的加油站，消耗 10 升燃料。将汽油从 0 升加
然后，我们从 10 英里处的加油站开到 60 英里处的加油站（消耗 50 升燃料
并将汽油从 10 升加到 50 升。然后我们开车抵达目的地。
我们沿途在1两个加油站停靠，所以返回 2 。
```

提示:

```
1 <= target, startFuel, stations[i][1] <= 10^9
0 <= stations.length <= 500
0 < stations[0][0] < stations[1][0] < ... < stations[stati
```

思路

为了能够获得**最低加油次数**，我们肯定希望能不加油就不加油。那什么时候必须加油呢？答案应该是**如果你不加油，就无法到达下一个目的地的时候**。

伪代码描述就是：

```
cur = startFuel # 刚开始有 startFuel 升汽油
last = 0 # 上一次的位置
for i, fuel in stations:
    cur -= i - last # 走过两个 staton 的耗油为两个 station 的距
    if cur < 0:
        # 我们必须在前面就加油，否则到不了这里
        # 但是在前面的哪个 station 加油呢?
        # 直觉告诉我们应该贪心地选择可以加汽油最多的站 i，如果加上 i
```

上面说了要选择可以加汽油最多的站 i，如果加了油还不行，继续选择第二多的站。这种动态求极值的场景非常适合使用 heap。

具体来说就是：

- 每经过一个站，就将其油量加到堆。
- 尽可能往前开，油只要不小于 0 就继续开。
- 如果油量小于 0 ，就从堆中取最大的加到油箱中去，如果油量还是小于 0 继续重复取堆中的最大油量。
- 如果加完油之后油量大于 0 ，继续开，重复上面的步骤。否则返回 -1，表示无法到达目的地。

那这个算法是如何体现**事后小诸葛**的呢？你可以把自己代入到题目中进行模拟。 把自己想象成正在开车，你的目标就是题目中的要求：**最少加油次数**。当你开到一个站的时候，你是不知道你的油量够不够支撑到下个站的，并且就算撑不到下个站，其实也许在上个站加油会更好。所以**现实中**你无论如何都**无法知道在当前站，我是应该加油还是不加油的**，因为信息太少了。

那我会怎么做呢？如果是我在开车的话，我只能每次都加油，这样都无法到达目的地，那肯定就无法到达目的地了。但如果这样可以到达目的地，我就可以说**如果我们在那个站加油，这个站选择不加就可以最少加油次数**

到达目的地了。你怎么不早说呢？ 这不就是事后诸葛亮么？

这个事后诸葛亮体现在**我们是等到没油了才去想应该在之前的某个站加油**。

所以这个事后诸葛亮本质上解决的是，基于当前信息无法获取最优解，我们必须掌握全部信息之后回溯。以这道题来说，我们可以先遍历一边 station，然后将每个 station 的油量记录到一个数组中，每次我们“预见“到无法到达下个站的时候，就从这个数组中取最大的。。。。 基于此，我们可以考虑使用堆优化取极值的过程，而不是使用数组的方式。

代码

```
class Solution:
    def minRefuelStops(self, target: int, startFuel: int, s
        stations += [(target, 0)]
        cur = startFuel
        ans = 0

        h = []
        last = 0
        for i, fuel in stations:
            cur -= i - last
            while cur < 0 and h:
                cur -= heapq.heappop(h)
                ans += 1
            if cur < 0:
                return -1
            heappush(h, -fuel)

            last = i
        return ans
```

(代码 1.3.10)

1488. 避免洪水泛滥

题目描述

你的国家有无数个湖泊，所有湖泊一开始都是空的。当第 n 个湖泊下雨的时候，

给你一个整数数组 rains ，其中：

rains[i] > 0 表示第 i 天时，第 rains[i] 个湖泊会下雨。
rains[i] == 0 表示第 i 天没有湖泊会下雨，你可以选择 一个 湖泊并 抽
请返回一个数组 ans ，满足：

ans.length == rains.length
如果 rains[i] > 0 ，那么ans[i] == -1 。
如果 rains[i] == 0 ，ans[i] 是你第 i 天选择抽干的湖泊。
如果有多种可行解，请返回它们中的 任意一个 。如果没办法阻止洪水，请返回

请注意，如果你选择抽干一个装满水的湖泊，它会变成一个空的湖泊。但如果你

示例 1：

输入：rains = [1,2,3,4]
输出：[-1,-1,-1,-1]
解释：第一天后，装满水的湖泊包括 [1]
第二天后，装满水的湖泊包括 [1,2]
第三天后，装满水的湖泊包括 [1,2,3]
第四天后，装满水的湖泊包括 [1,2,3,4]
没有哪一天你可以抽干任何湖泊的水，也没有湖泊会发生洪水。
示例 2：

输入：rains = [1,2,0,0,2,1]
输出：[-1,-1,2,1,-1,-1]
解释：第一天后，装满水的湖泊包括 [1]
第二天后，装满水的湖泊包括 [1,2]
第三天后，我们抽干湖泊 2 。所以剩下装满水的湖泊包括 [1]
第四天后，我们抽干湖泊 1 。所以暂时没有装满水的湖泊了。
第五天后，装满水的湖泊包括 [2]。
第六天后，装满水的湖泊包括 [1,2]。
可以看出，这个方案下不会有洪水发生。同时， [-1,-1,1,2,-1,-1] 也是另
示例 3：

输入：rains = [1,2,0,1,2]
输出：[]
解释：第二天后，装满水的湖泊包括 [1,2]。我们可以在第三天抽干一个湖泊的
但第三天后，湖泊 1 和 2 都会再次下雨，所以不管我们第三天抽干哪个湖泊的
示例 4：

输入：rains = [69,0,0,0,69]
输出：[-1,69,1,1,-1]
解释：任何形如 [-1,69,x,y,-1], [-1,x,69,y,-1] 或者 [-1,x,y,69,

```
示例 5：

输入：rains = [10,20,20]
输出：[]
解释：由于湖泊 20 会连续下 2 天的雨，所以没有没有办法阻止洪水。


提示：

1 <= rains.length <= 10^5
0 <= rains[i] <= 10^9
```

思路

如果上面的题用**事后诸葛亮**描述比较牵强的话，那后面这两个题可以说很适合了。

题目说明了我们可以在不下雨的时候抽干一个湖泊，如果有多个下满雨的湖泊，我们该抽干哪个湖呢？显然应该是抽干最近即将被洪水淹没的湖。但是现实中无论如何我们都不可能知道未来哪天哪个湖泊会下雨的，即使有天气预报也不行，因此它也不 100% 可靠。

但是代码可以啊。我们可以先遍历一遍 rain 数组就知道第几天哪个湖泊下雨了。有了这个信息，我们就可以事后诸葛亮了。

“今天天气很好，我开了天眼，明天湖泊 2 会被洪水淹没，我们今天就先抽干它，否则就洪水泛滥了。”。

和上面的题目一样，我们也可以不先遍历 rain 数组，再模拟每天的变化，而是直接模拟，即使当前是晴天我们也不抽干任何湖泊。接着在模拟的过程**记录晴天的情况**，等到洪水发生的时候，我们再考虑前面**哪一个晴天**应该抽干哪个湖泊。因此这个事后诸葛亮体现在**我们是等到洪水泛滥了才去想应该在之前的某天采取什么手段**。

算法：

- 遍历 rain， 模拟每天的变化
- 如果 rain 当前是 0 表示当前是晴天，我们不抽干任何湖泊。但是我们将当前天记录到 sunny 数组。
- 如果 rain 大于 0，说明有一个湖泊下雨了，我们去看下下雨的这个湖泊是否发生了洪水泛滥。其实就是看下下雨前是否已经有水了。这提示我们用一个数据结构 lakes 记录每个湖泊的情况，我们可以用 0 表示没有水，1 表示有水。这样当湖泊 i 下雨的时候且 lakes[i] = 1 就会发生洪水泛滥。
- 如果当前湖泊发生了洪水泛滥，那么就去 sunny 数组找一个晴天去抽干它，这样它就不会洪水泛滥，接下来只需要保持 lakes[i] = 1 即可。

这道题没有使用到堆，我是故意的。之所以这么做，是让大家明白**事后诸葛亮**这个技巧并不是堆特有的，实际上这就是一种普通的算法思想，就好像从后往前遍历一样。只不过，很多时候，我们**事后诸葛亮**的场景，需要动态取最大最小值， 这个时候就应该考虑使用堆了，这其实又回到文章开头的**一个中心**了，所以大家一定要灵活使用这些技巧，不可生搬硬套。

下一道题是一个不折不扣的**事后诸葛亮 + 堆优化**的题目。

代码

```
class Solution:
    def avoidFlood(self, rains: List[int]) -> List[int]:
        ans = [1] * len(rains)
        lakes = collections.defaultdict(int)
        sunny = []

        for i, rain in enumerate(rains):
            if rain > 0:
                ans[i] = -1
                if lakes[rain - 1] == 1:
                    if 0 == len(sunny):
                        return []
                    ans[sunny.pop()] = rain
                lakes[rain - 1] = 1
            else:
                sunny.append(i)
        return ans
```

(代码 1.3.11)

2021-04-06 fixed: 上面的代码有问题。错误的原因在于上述算法**如果当前湖泊发生了洪水泛滥，那么就去 sunny 数组找一个晴天去抽干它，这样它就不会洪水泛滥**部分的实现不对。sunny 数组找一个晴天去抽干它的根

本前提是 **出现晴天的时候湖泊里面要有水才能抽**，如果晴天的时候，湖泊里面没有水也不行。这提示我们的 lakes 不存储 0 和 1 ，而是存储发生洪水是第几天。这样问题就变为**在 sunny 中找一个日期大于 lakes[rain-1]** 的项，并将其移除 sunny 数组。由于 sunny 数组是有序的，因此我们可以使用二分来进行查找。

> 由于我们需要删除 sunny 数组的项，因此时间复杂度不会因为使用了二分而降低。

正确的代码应该为：

```
class Solution:
    def avoidFlood(self, rains: List[int]) -> List[int]:
        ans = [1] * len(rains)
        lakes = {}
        sunny = []

        for i, rain in enumerate(rains):
            if rain > 0:
                ans[i] = -1
                if rain - 1 in lakes:
                    j = bisect.bisect_left(sunny, lakes[ra
                    if j == len(sunny):
                        return []
                    ans[sunny.pop(j)] = rain
                lakes[rain - 1] = i
            else:
                sunny.append(i)
        return ans
```

1642. 可以到达的最远建筑

题目描述

```
给你一个整数数组 heights ，表示建筑物的高度。另有一些砖块 bricks 和

你从建筑物 0 开始旅程，不断向后面的建筑物移动，期间可能会用到砖块或梯子

当从建筑物 i 移动到建筑物 i+1（下标 从 0 开始 ）时：

如果当前建筑物的高度 大于或等于 下一建筑物的高度，则不需要梯子或砖块
如果当前建筑的高度 小于 下一个建筑的高度，您可以使用 一架梯子 或 (h[i
如果以最佳方式使用给定的梯子和砖块，返回你可以到达的最远建筑物的下标（
```

```
示例 1:


输入: heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1
输出: 4
解释: 从建筑物 0 出发，你可以按此方案完成旅程:
- 不使用砖块或梯子到达建筑物 1 ，因为 4 >= 2
- 使用 5 个砖块到达建筑物 2 。你必须使用砖块或梯子，因为 2 < 7
- 不使用砖块或梯子到达建筑物 3 ，因为 7 >= 6
- 使用唯一的梯子到达建筑物 4 。你必须使用砖块或梯子，因为 6 < 9
无法越过建筑物 4 ，因为没有更多砖块或梯子。
示例 2:

输入: heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladder
输出: 7
示例 3:

输入: heights = [14,3,19,3], bricks = 17, ladders = 0
输出: 3


提示:

1 <= heights.length <= 105
1 <= heights[i] <= 106
0 <= bricks <= 109
0 <= ladders <= heights.length
```

思路

我们可以将梯子看出是无限的砖块，只不过只能使用一次，我们当然希望能将好梯用在刀刃上。和上面一样，如果是现实生活，我们是无法知道啥时候用梯子好，啥时候用砖头好的。

没关系，我们继续使用事后诸葛亮法，一次遍历就可完成。和前面的思路类似，那就是我无脑用梯子，等梯子不够用了，我们就要开始事后诸葛亮了，**要是前面用砖头就好了**。那什么时候用砖头就好了呢？很明显就是当初用梯子的时候高度差，比现在的高度差小。

直白点就是当初我用梯子爬了个 5 米的墙，现在这里有个十米的墙，我没梯子了，只能用 10 个砖头了。要是之前用 5 个砖头，现在不就可以用一个梯子，从而省下 5 个砖头了吗？

这提示我们将用前面用梯子跨越的建筑物高度差存起来，等到后面梯子用完了，我们将前面被用的梯子“兑换”成砖头继续用。以上面的例子来说，我们就可以先兑换 10 个砖头，然后将 5 个砖头用掉，也就是相当于增加

了 5 个砖头。

如果前面多次使用了梯子，我们优先“兑换”哪次呢？显然是优先兑换**高度差**大的，这样兑换的砖头才最多。这提示每次都从之前存储的高度差中选最大的，并在“兑换”之后将其移除。这种**动态求极值**的场景用什么数据结构合适？当然是堆啦。

代码

```
class Solution:
    def furthestBuilding(self, heights: List[int], bricks:
        h = []
        for i in range(1, len(heights)):
            diff = heights[i] - heights[i - 1]
            if diff <= 0:
                continue
            if bricks < diff and ladders > 0:
                ladders -= 1
                if h and -h[0] > diff:
                    bricks -= heapq.heappop(h)
                else:
                    continue
            bricks -= diff
            if bricks < 0:
                return i - 1
            heapq.heappush(h, -diff)
        return len(heights) - 1
```

(代码 1.3.12)

四大应用

接下来是本文的最后一个部分《四大应用》，目的是通过这几个例子来帮助大家巩固前面的知识。

1. topK

求解 topK 是堆的一个很重要的功能。这个其实已经在前面的**固定堆**部分给大家介绍过了。

这里直接引用前面的话：

“其实求第 k 小的数最简单的思路是建立小顶堆，将所有的数先全部入堆，然后逐个出堆，一共出堆 k 次。最后一次出堆的就是第 k 小的数。然而，我们也可不先全部入堆，而是建立大顶堆（注意不是上面的小顶堆），并维持堆的大小为 k 个。如果新的数入堆之后堆的大小大于 k，则

需要将堆顶的数和新的数进行比较，并将较大的移除。这样可以保证堆中的数是全体数字中最小的 k 个，而这最小的 k 个中最大的（即堆顶）不就是第 k 小的么？这也就是选择建立大顶堆，而不是小顶堆的原因。”

其实除了第 k 小的数，我们也可以将中间的数全部收集起来，这就可以求出最小的 **k 个数**。和上面第 k 小的数唯一不同的点在于需要收集 popp 出来的所有的数。

需要注意的是，有时候权重并不是原本数组值本身的大小，也可以是距离，出现频率等。

相关题目：

- 面试题 17.14. 最小 K 个数
- 347. 前 K 个高频元素
- 973. 最接近原点的 K 个点

力扣中有关第 k 的题目很多都是堆。除了堆之外，第 k 的题目其实还会有一些**找规律**的题目，对于这种题目则可以通过**分治+递归**的方式来解决，具体就不再这里展开了，感兴趣的可以和我留言讨论。

2. 带权最短距离

关于这点，其实我在前面部分也提到过了，只不过当时只是一带而过。原话是“不过 BFS 真的就没人用优先队列实现么？当然不是！比如带权图的最短路径问题，如果用队列做 BFS 那就需要优先队列才可以，因为路径之间是有**权重的差异**的，这不就是优先队列的设计初衷么。**使用优先队列的 BFS 实现典型的就是 dijkstra 算法**。”

DIJKSTRA 算法主要解决的是图中任意两点的最短距离。

算法的基本思想是贪心，每次都遍历所有邻居，并从中找到距离最小的，本质上是一种广度优先遍历。这里我们借助堆这种数据结构，使得可以在 $logN$ 的时间内找到 cost 最小的点，其中 N 为 堆的大小。

代码模板：

```
def dijkstra(graph, start, end):
    # 堆里的数据都是 (cost, i) 的二元祖，其含义是“从 start 走到 i
    heap = [(0, start)]
    visited = set()
    while heap:
        (cost, u) = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        if u == end:
            return cost
        for v, c in graph[u]:
            if v in visited:
                continue
            next = cost + c
            heapq.heappush(heap, (next, v))
    return -1
```

(代码 1.4.1)

> 可以看出代码模板和 BFS 基本是类似的。如果你自己将堆的 key 设定为 steps 也可模拟实现 BFS，这个在前面已经讲过了，这里不再赘述。

比如一个图是这样的：

```
E -- 1 --> B -- 1 --> C -- 1 --> D -- 1 --> F
 \                                          /\
  \                                         ||
    -------- 2 ---------> G ------- 1 ------
```

我们使用邻接矩阵来构造：

```
G = {
    "B": [["C", 1]],
    "C": [["D", 1]],
    "D": [["F", 1]],
    "E": [["B", 1], ["G", 2]],
    "F": [],
    "G": [["F", 1]],
}

shortDistance = dijkstra(G, "E", "C")
print(shortDistance)  # E -- 3 --> F -- 3 --> C == 6
```

会了这个算法模板， 你就可以去 AC 743. 网络延迟时间 了。

完整代码：

```
class Solution:
    def dijkstra(self, graph, start, end):
        heap = [(0, start)]
        visited = set()
        while heap:
            (cost, u) = heapq.heappop(heap)
            if u in visited:
                continue
            visited.add(u)
            if u == end:
                return cost
            for v, c in graph[u]:
                if v in visited:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v))
        return -1
    def networkDelayTime(self, times: List[List[int]], N: 
        graph = collections.defaultdict(list)
        for fr, to, w in times:
            graph[fr - 1].append((to - 1, w))
        ans = -1
        for to in range(N):
            # 调用封装好的 dijkstra 方法
            dist = self.dijkstra(graph, K - 1, to)
            if dist == -1: return -1
            ans = max(ans, dist)
        return ans
```

(代码 1.4.2)

你学会了么？

上面的算法并不是最优解，我只是为了体现**将 dijkstra 封装为 api 调用**的思想。一个更好的做法是一次遍历记录所有的距离信息，而不是每次都重复计算。时间复杂度会大大降低。这在计算一个点到图中所有点的距离时有很大的意义。 为了实现这个目的，我们的算法会有什么样的调整？

> 提示：你可以使用一个 dist 哈希表记录开始点到每个点的最短距离来完成。想出来的话，可以用力扣 882 题去验证一下哦~

其实只需要做一个小的调整就可以了，由于调整很小，直接看代码会比较好。

代码：

```
class Solution:
    def dijkstra(self, graph, start, end):
        heap = [(0, start)]  # cost from start node,end noc
        dist = {}
        while heap:
            (cost, u) = heapq.heappop(heap)
            if u in dist:
                continue
            dist[u] = cost
            for v, c in graph[u]:
                if v in dist:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v))
        return dist
    def networkDelayTime(self, times: List[List[int]], N: 
        graph = collections.defaultdict(list)
        for fr, to, w in times:
            graph[fr - 1].append((to - 1, w))
        ans = -1
        dist = self.dijkstra(graph, K - 1, to)
        return -1 if len(dist) != N else max(dist.values())
```

(代码 1.4.3)

可以看出我们只是将 visitd 替换成了 dist，其他不变。另外 dist 其实只是带了 key 的 visited，它这里也起到了 visitd 的作用。

如果你需要计算一个节点到其他所有节点的最短路径，可以使用一个 dist（一个 hashmap）来记录出发点到所有点的最短路径信息，而不是使用 visited （一个 hashset）。

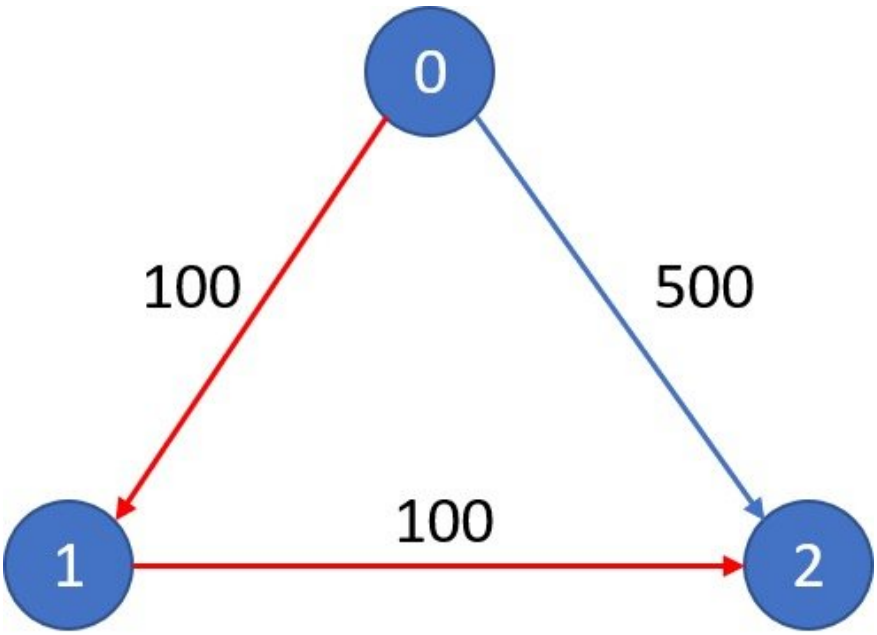
类似的题目也不少， 我再举一个给大家 787. K 站中转内最便宜的航班。题目描述：

有 n 个城市通过 m 个航班连接。每个航班都从城市 u 开始，以价格 w 抵达

现在给定所有的城市和航班，以及出发城市 src 和目的地 dst，你的任务是找

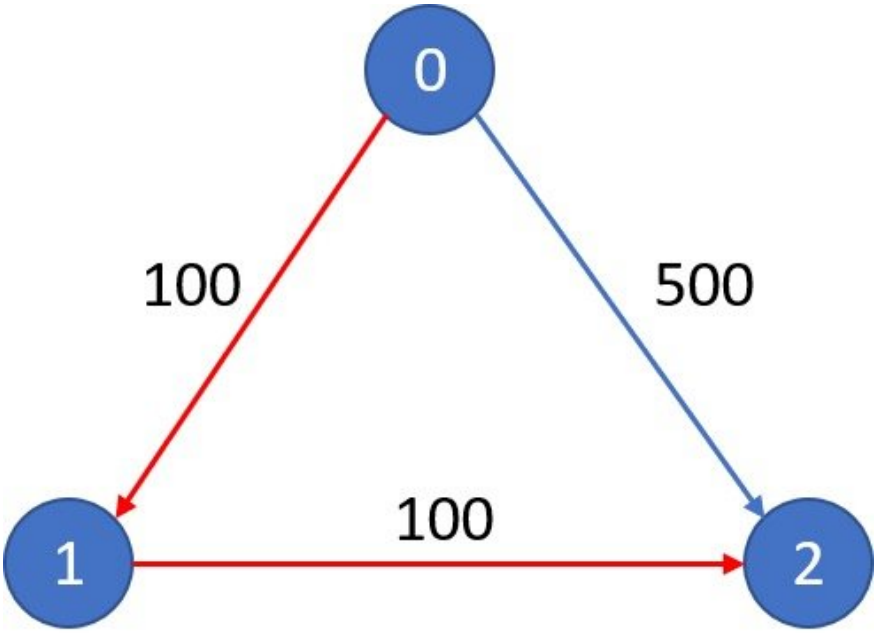
示例 1:

输入:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 1
输出: 200
解释:
城市航班图如下



从城市 0 到城市 2 在 1 站中转以内的最便宜价格是 200，如图中红色所示。
示例 2:

输入:
n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]
src = 0, dst = 2, k = 0
输出: 500
解释:
城市航班图如下




```
从城市 0 到城市 2 在 0 站中转以内的最便宜价格是 500，如图中蓝色所示。


提示：

n 范围是 [1, 100]，城市标签从 0 到 n - 1
航班数量范围是 [0, n * (n - 1) / 2]
每个航班的格式 (src, dst, price)
每个航班的价格范围是 [1, 10000]
k 范围是 [0, n - 1]
航班没有重复，且不存在自环
```

这道题和上面的没有本质不同， 我仍然将其封装成 API 来使用，具体看代码就行。

这道题唯一特别的点在于如果中转次数大于 k，也认为无法到达。这个其实很容易，我们只需要在堆中用元组来**多携带一个 steps**即可，这个 steps 就是 不带权 BFS 中的距离。如果 pop 出来 steps 大于 K，则认为非法，我们跳过继续处理即可。

```
class Solution:
    # 改造一下，增加参数 K，堆多携带一个 steps 即可
    def dijkstra(self, graph, start, end, K):
        heap = [(0, start, 0)]
        visited = set()
        while heap:
            (cost, u, steps) = heapq.heappop(heap)
            if u in visited:
                continue
            visited.add((u, steps))
            if steps > K: continue
            if u == end:
                return cost
            for v, c in graph[u]:
                if (v, steps) in visited:
                    continue
                next = cost + c
                heapq.heappush(heap, (next, v, steps + 1))
        return -1
    def findCheapestPrice(self, n: int, flights: List[List
        graph = collections.defaultdict(list)
        for fr, to, price in flights:
            graph[fr].append((to, price))
         # 调用封装好的 dijkstra 方法
        return self.dijkstra(graph, src, dst, K + 1)
```

(代码 1.4.4)

3. 因子分解

和上面两个应用一下，这个我在前面 《313. 超级丑数》部分也提到了。

回顾一下丑数的定义： **丑数就是质因数只包含 2, 3, 5 的正整数。** 因此丑数本质就是一个数经过**因子分解**之后只剩下 2，3，5 的整数，而不携带别的因子了。

关于丑数的题目有很多，大多数也可以从堆的角度考虑来解。只不过有时候因子个数有限，不使用堆也容易解决。比如：264. 丑数 II 就可以使用三个指针来记录即可，这个技巧在前面也讲过了，不再赘述。

一些题目并不是丑数，但是却明确提到了类似**因子**的信息，并让你求第 k 大的 xx，这个时候优先考虑使用堆来解决。如果题目中夹杂一些其他信息，比如**有序**，则也可考虑二分法。具体使用哪种方法，要具体问题具体分析，不过在此之前大家要对这两种方法都足够熟悉才行。

4. 堆排序

前面的三种应用或多或少在前面都提到过。而**堆排序**却未曾在前面提到。

直接考察堆排序的题目几乎没有。但是面试却有可能会考察，另外学习堆排序对你理解分治等重要算法思维都有重要意义。个人感觉，堆排序，构造二叉树，构造线段树等算法都有很大的相似性，掌握一种，其他都可以触类旁通。

实际上，经过前面的堆的学习，我们可以封装一个堆排序，方法非常简单。

这里我放一个使用堆的 api 实现堆排序的简单的示例代码：

```
h = [9,5,2,7]
heapq.heapify(h)
ans = []

while h:
    ans.append(heapq.heappop(h))
print(ans) # 2,5,7,9
```

明白了示例， 那封装成**通用堆排序**就不难了。

```
def heap_sort(h):
    heapq.heapify(h)
    ans = []
    while h:
        ans.append(heapq.heappop(h))
    return ans
```

这个方法足够简单，如果你明白了前面堆的原理，让你手撸一个堆排序也不难。可是这种方法有个弊端，它不是**原位算法**，也就是说你必须使用额外的空间承接结果，空间复杂度为 $O(N)$。但是其实调用完堆排序的方法后，原有的数组内存可以被释放了，因此理论上来说空间也没浪费，只不过我们计算空间复杂度的时候取的是使用内存最多的时刻，因此使用原地算法毫无疑问更优秀。如果你实在觉得不爽这个实现，也可以采用原地的修改的方式。这倒也不难，只不过稍微改造一下前面的堆的实现即可，由于篇幅的限制，这里不多讲了。

总结

堆和队列有千丝万缕的联系。 很多题目我都是先思考使用堆来完成。然后发现每次入堆都是 + 1，而不会跳着更新，比如下一个是 + 2，+3 等等，因此使用队列来完成性能更好。 比如 649. Dota2 参议院 和 1654. 到家的最少跳跃次数 等。

堆的中心就一个，那就是**动态求极值**。

而求极值无非就是最大值或者最小值，这不难看出。如果求最大值，我们可以使用大顶堆，如果求最小值，可以用最小堆。而实际上，如果没有动态两个字，很多情况下没有必要使用堆。比如可以直接一次遍历找出最大的即可。而动态这个点不容易看出来，这正是题目的难点。这需要你先对问题进行分析， 分析出这道题**其实就是动态求极值**，那么使用堆来优化就应该被想到。

堆的实现有很多。比如基于链表的跳表，基于数组的二叉堆和基于红黑树的实现等。这里我们介绍了**两种主要实现** 并详细地讲述了二叉堆的实现，不仅是其实现简单，而且其在很多情况下表现都不错，推荐大家重点掌握二叉堆实现。

对于二叉堆的实现，**核心点就一点**，那就是始终维护堆的性质不变，具体是什么性质呢？那就是 **父节点的权值不大于儿子的权值（小顶堆）**。为了达到这个目的，我们需要在入堆和出堆的时候，使用上浮和下沉操作，并恰当地完成元素交换。具体来说就是上浮过程和比它大的父节点进行交换，下沉过程和两个子节点中较小的进行交换，当然前提是它有子节点且子节点比它小。

关于堆化我们并没有做详细分析。不过如果你理解了本文的入堆操作，这其实很容易。因此堆化本身就是一个不断入堆的过程，只不过**将时间上的离散的操作变成了一次性操作**而已。

另外我给大家介绍了三个堆的做题技巧，分别是：

- 固定堆，不仅可以解决第 k 问题，还可有效利用已经计算的结果，避免重复计算。
- 多路归并，本质就是一个暴力解法，和暴力递归没有本质区别。如果你将其转化为递归，也是一种不能记忆化的递归。因此更像是**回溯算法**。
- 事后小诸葛。有些信息，我们在当前没有办法获取，就可用一种数据结构存起来，方便之后”东窗事发“的时候查。这种数据解决可以是很多，常见的有哈希表和堆。你也可以将这个技巧看成是**事后后悔**，有的人比较能接受这种叫法，不过不管叫法如何，指的都是这个含义。

最后给大家介绍了四种应用，这四种应用除了堆排序，其他在前面或多或少都讲过，它们分别是：

- topK
- 带权最短路径
- 因子分解
- 堆排序

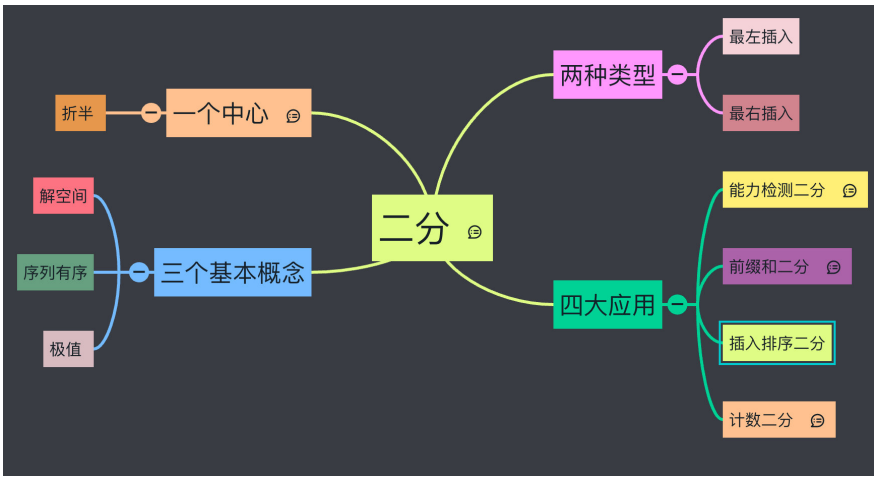
这四种应用实际上还是围绕了堆的一个中心**动态取极值**，这四种应用只不过是灵活使用了这个特点罢了。因此大家在做题的时候只要死记**动态求极值**即可。如果你能够分析出这道题和动态取极值有关，那么请务必考虑堆。接下来我们就要在脑子中过一下复杂度，对照一下题目数据范围就大概可以估算出是否可行啦。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 39K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

几乎刷完了力扣所有的二分题，我发现了这些东西。。。（上）

前言



大家好，我是 lucifer。今天给大家带来的是《二分》专题。先上下本文的提纲，这个是我用 mindmap 画的一个脑图，之后我会继续完善，将其他专题逐步完善起来。

> 大家也可以使用 vscode blink-mind 打开源文件查看，里面有一些笔记可以点开查看。源文件可以去我的公众号《力扣加加》回复脑图获取，以后脑图也会持续更新更多内容。vscode 插件地址：https://marketplace.visualstudio.com/items?itemName=awehook.vscode-blink-mind

本系列包含以下专题：

- 几乎刷完了力扣所有的链表题，我发现了这些东西。。。
- 几乎刷完了力扣所有的树题，我发现了这些东西。。。
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（上）
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（下）

本专题预计分两部分两进行。第一部分主要讲述**基本概念** 和 **一个中心**。有了这些基础知识之后，第二部分我们继续学习**两种二分类型** 和**四大应用**。

本文内容已经同步到我的刷题插件的 RoadMap 中，结合刷题插件食用味道更佳哦~ 插件的获取方式可以在我的公众号力扣加加中回复插件查看。

> 如果觉得文章有用，请点赞留言转发一下，让我有动力继续做下去。

前言

为了准备这个专题，我不仅肝完了力扣的所有二分题目，还肝完了另外一个 OJ 网站 - Binary Search 的所有二分题目，一共**100 多道**。大家看完如果觉得有用，可以通过点赞转发的方式告诉我，如果喜欢的人多，我继续尽快出下篇哦~

二分查找又称 `折半搜索算法` 。 狭义地来讲，二分查找是一种在有序数组查找某一特定元素的搜索算法。这同时也是大多数人所知道的一种说法。实际上， 广义的二分查找是将问题的规模缩小到原有的一半。类似的，三分法就是将问题规模缩小为原来的 1/3。

本文给大家带来的内容则是 `狭义地二分查找` ，如果想了解其他广义上的二分查找可以查看我之前写的一篇博文 从老鼠试毒问题来看二分法

> 尽管二分查找的基本思想相对简单，但细节可以令人难以招架 ... — 高德纳

当乔恩·本特利将二分搜索问题布置给专业编程课的学生时，百分之 90 的学生在花费数小时后还是无法给出正确的解答，主要因为这些错误程序在面对边界值的时候无法运行，或返回错误结果。1988 年开展的一项研究显示，20 本教科书里只有 5 本正确实现了二分搜索。不仅如此，本特利

自己 1986 年出版的《编程珠玑》一书中的二分搜索算法存在整数溢出的问题，二十多年来无人发现。Java 语言的库所实现的二分搜索算法中同样的溢出问题存在了九年多才被修复。

可见二分查找并不简单， 本文就试图带你走近 ta，明白 ta 的底层逻辑，并提供模板帮助大家写书 bug free 的二分查找代码。看完讲义后建议大家结合 LeetCode Book 二分查找 练习一下。

基本概念

首先我们要知道几个基本概念。这些概念对学习二分有着很重要的作用，之后遇到这些概念就不再讲述了，默认大家已经掌握。

解空间

解空间指的是**题目所有可能的解构成的集合**。比如一个题目所有解的可能是 1,2,3,4,5，但具体在某一种情况只能是其中某一个数（即可能是 1，2，3，4，5 中的**一个数**）。那么这里的解空间就是 1,2,3,4,5 构成的集合，在某一个具体的情况下可能是其中任意一个值，**我们的目标就是在某个具体的情况判断其具体是哪个**。如果线性枚举所有的可能，就枚举这部分来说时间复杂度就是 $O(n)$。

举了例子：

如果让你在一个数组 nums 中查找 target，如果存在则返回对应索引，如果不存在则返回 -1。那么对于这道题来说其解空间是什么？

很明显解空间是区间 [-1, n-1]，其中 n 为 nums 的长度。

需要注意的是上面题目的解空间只可能是区间 [-1,n-1] 之间的整数。而诸如 1.2 这样的小数是不可能存在的。这其实也是大多数二分的情况。 但也有少部分题目解空间包括小数的。如果解空间包括小数，就可能会涉及到精度问题，这一点大家需要注意。

比如让你求一个数 x 的平方根，答案误差在 10^-6 次方都认为正确。这里容易知道其解空间大小可定义为 [1,x]（当然可以定义地更精确，之后我们再讨论这个问题），其中解空间应该包括所有区间的实数，不仅仅是整数而已。这个时候解题思路和代码都没有太大变化，唯二需要变化的是：

1. 更新答案的步长。 比如之前的更新是 `l = mid + 1`，现在**可能**就不行了，因此这样**可能**会错过正确解，比如正确解恰好就在区间 [mid,mid+1] 内的某一个小数。
2. 判断条件时候需要考虑误差。由于精度的问题，判断的结束条件可能就要变成 **与答案的误差在某一个范围内**。

对于**搜索类题目**，解空间一定是有限的，不然问题不可解。对于搜索类问题，第一步就是需要明确解空间，这样你才能够在解空间内进行搜索。这个技巧不仅适用于二分法，只要是搜索问题都可以使用，比如 DFS，BFS 以及回溯等。只不过对于二分法来说，**明确解空间显得更为重要**。如果现在还不理解这句话也没关系，看完本文或许你就理解了。

定义解空间的时候的一个原则是： 可以大但不可以小。因为如果解空间偏大（只要不是无限大）无非就是多做几次运算，而如果解空间过小则可能**错失正确解**，导致结果错误。比如前面我提到的求 x 的平方根，我们当然可以将解空间定义的更小，比如定义为 [1,x/2]，这样可以减少运算的次数。但如果设置地太小，则可能会错过正确解。这是新手容易犯错的点之一。

有的同学可能会说我看不出来怎么办呀。我觉得如果你实在拿不准也完全没有关系，比如求 x 的平方根，就可以甚至为 [1,x]，就让它多做几次运算嘛。我建议你**给上下界设置一个宽泛的范围**。等你对二分逐步了解之后可以**卡地更死一点**。

序列有序

我这里说的是序列，并不是数组，链表等。也就是说二分法通常要求的序列有序，不一定是数组，链表，也有可能是其他数据结构。另外有的**序列有序**题目直接讲出来了，会比较容易。而有些则隐藏在题目信息之中。乍一看，题目并没有**有序**关键字，而有序其实就隐藏在字里行间。比如题目给了数组 nums，并且没有限定 nums 有序，但限定了 nums 为非负。这样如果给 nums 做前缀和或者前缀或（位运算或），就可以得到一个有序的序列啦。

> 更多技巧在四个应用部分展开哦。

虽然二分法不意味着需要序列有序，但大多数二分题目都有**有序**这个显著特征。只不过：

- 有的是题目直接限定了有序。这种题目通常难度不高，也容易让人想到用二分。
- 有的是需要你**自己构造有序序列**。这种类型的题目通常难度不低，需要大家有一定的观察能力。

比如Triple Inversion。题目描述如下：

```
Given a list of integers nums, return the number of pairs i

Constraints:  n ≤ 100,000 where n is the length of nums
Example 1
Input:
nums = [7, 1, 2]
Output:
2
Explanation:
We have the pairs (7, 1) and (7, 2)
```

这道题并没有限定数组 nums 是有序的，但是我们可以构造一个有序序列 d，进而在 d 上做二分。代码：

```python
class Solution:
    def solve(self, A):
        d = []
        ans = 0

        for a in A:
            i = bisect.bisect_right(d, a * 3)
            ans += len(d) - i
            bisect.insort(d, a)
        return ans
```

如果暂时不理解代码也没关系，大家先留个印象，知道有这么一种类型题即可，大家可以看完本章的所有内容（上下两篇）之后再回头做这道题。

极值

类似我在堆专题 提到的极值。只不过这里的极值是**静态的**，而不是动态的。这里的极值通常指的是**求第 k 大（或者第 k 小）的数。**

堆的一种很重要的用法是求第 k 大的数，而二分法也可以求第 k 大的数，只不过**二者的思路完全不同**。使用堆求第 k 大的思路我已经在前面提到的堆专题里详细解释了。那么二分呢？这里我们通过一个例子来感受一下：这道题是 Kth Pair Distance，题目描述如下：

```
Given a list of integers nums and an integer k, return the

Constraints:n ≤ 100,000 where n is the length of nums
Example 1
Input:
nums = [1, 5, 3, 2]
k = 3
Output:
2
Explanation:

Here are all the pair distances:

abs(1 - 5) = 4
abs(1 - 3) = 2
abs(1 - 2) = 1
abs(5 - 3) = 2
abs(5 - 2) = 3
abs(3 - 2) = 1

Sorted in ascending order we have [1, 1, 2, 2, 3, 4].
```

简单来说，题目就是给的一个数组 nums，让你求 nums 第 k 大的**任意两个数的差的绝对值**。当然，我们可以使用堆来做，只不过使用堆的时间复杂度会很高，导致无法通过所有的测试用例。这道题我们可以使用二分法来降维打击。

对于这道题来说，解空间就是从 0 到数组 nums 中最大最小值的差，用区间表示就是 [0, max(nums) - min(nums)]。明确了解空间之后，我们就需要对解空间进行二分。对于这道题来说，可以选当前解空间的中间值 mid，然后计算小于等于这个中间值的**任意两个数的差的绝对值**有几个，我们不妨令这个数字为 x。

- 如果 x 大于 k，那么解空间中大于等于 mid 的数都不可能是答案，可以将其舍弃。
- 如果 x 小于 k，那么解空间中小于等于 mid 的数都不可能是答案，可以将其舍弃。
- 如果 x 等于 k，那么 mid 就是答案。

基于此，我们可使用二分来解决。这种题型，我总结为**计数二分**。我会在后面的四大应用部分重点讲解。

代码：

```
class Solution:
    def solve(self, A, k):
        A.sort()
        def count_not_greater(diff):
            i = ans = 0
            for j in range(1, len(A)):
                while A[j] - A[i] > diff:
                    i += 1
                ans += j - i
            return ans
        l, r = 0, A[-1] - A[0]

        while l <= r:
            mid = (l + r) // 2
            if count_not_greater(mid) > k:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

如果暂时不理解代码也没关系，大家先留个印象，知道有这么一种类型题即可，大家可以看完本章的所有内容（上下两篇）之后再回头做这道题。

一个中心

二分法的一个中心大家一定牢牢记住。其他（比如序列有序，左右双指针）都是二分法的手和脚，都是表象，并不是本质，而**折半才是二分法的灵魂**。

前面已经给大家明确了解空间的概念。而这里的折半其实就是解空间的折半。

比如刚开始解空间是 [1, n]（n 为一个大于 n 的整数）。通过**某种方式**，我们确定 [1, m] 区间都**不可能是答案**。那么解空间就变成了 (m,n]，持续此过程知道解空间变成平凡（直接可解）。

> 注意区间 (m,n] 左侧是开放的，表示 m 不可能取到。

显然折半的难点是**根据什么条件舍弃哪一步部分**。这里有两个关键字:

1. 什么条件
2. 舍弃哪部分

几乎所有的二分的难点都在这两个点上。如果明确了这两点，几乎所有的二分问题都可以迎刃而解。幸运的是，关于这两个问题的答案通常都是有限的，题目考察的往往就是那几种。这其实就是所谓的做题套路。关于这

些套路，我会在之后的四个应用部分给大家做详细介绍。

二分法上篇小结

上篇主要就是带大家了解几个概念，这些概念对做题极为重要，请务必掌握。接下来讲解了二分法的中心 - 折半，这个中心需要大家做任何二分都要放到脑子中。

如果让我用一句话总结二分法，我会说**二分法是一种让未知世界无机可乘的算法**。即二分法无论如何我们都可以舍弃一半解，也就是无论如何都可以将解空间砍半。难点就是上面提到的两点：**什么条件** 和 **舍弃哪部分**。这是二分法核心要解决的问题。

以上就是《二分专题（上篇）》的所有内容。如果觉得文章有用，请点赞留言转发一下，让我有动力继续出下集。

下集预告

上集介绍的是基本概念。下一集我们介绍两种二分的类型以及四种二分的应用。

下集目录：

- 两种类型
    - 最左插入
    - 最右插入
- 四大应用
    - 能力检测二分
    - 前缀和二分
    - 插入排序二分（不是你理解的插入排序哦）
    - 计数二分

其中两种类型（最左和最右插入）主要解决的的是：**解空间已经明确出来了，如何用代码找出具体的解**。

而四大应用主要解决的是：**如何构造解空间**。更多的情况则是如何构建有序序列。

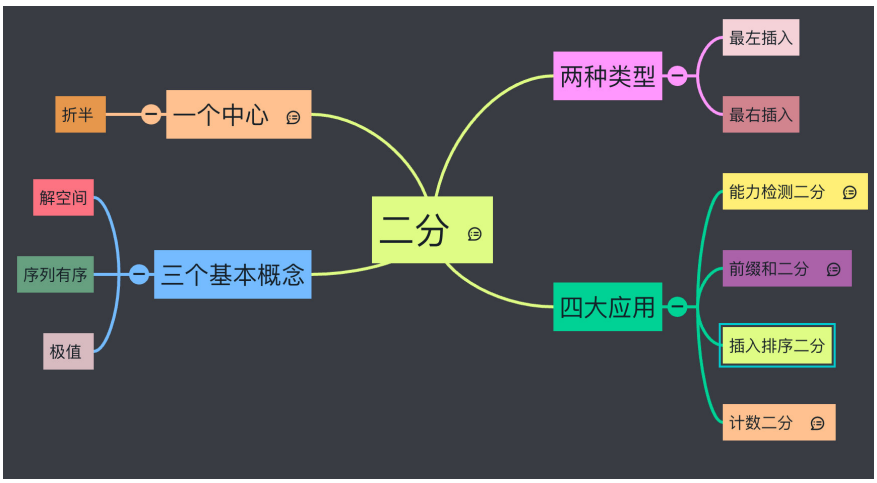
这两部分都是实操性很强的内容，在理解这两部分内容的同时，请大家务必牢记一个中心**折半**。那我们下篇见喽~

几乎刷完了力扣所有的二分题，我发现了这些东西。。。（下）

前言

大家好，我是 lucifer。今天给大家带来的是《二分》专题。先上下本文的提纲，这个是我用 mindmap 画的一个脑图，之后我会继续完善，将其他专题逐步完善起来。

> 大家也可以使用 vscode blink-mind 打开源文件查看，里面有一些笔记可以点开查看。源文件可以去我的公众号《力扣加加》回复脑图获取，以后脑图也会持续更新更多内容。vscode 插件地址：https://marketplace.visualstudio.com/items?itemName=awehook.vscode-blink-mind



本系列包含以下专题：

- 几乎刷完了力扣所有的链表题，我发现了这些东西。。。
- 几乎刷完了力扣所有的树题，我发现了这些东西。。。
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（上）
- 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（下）
- 几乎刷完了力扣所有的二分题，我发现了这些东西。。。（上）

本专题预计分两部分两进行。上一节主要讲述**基本概念** 和 **一个中心**。这一节我们继续学习**两种二分类型** 和**四大应用**。没有看过上篇的建议先看一下上篇，地址在上面。

> 如果觉得文章有用，请点赞留言转发一下，让我有动力继续做下去。

上篇回顾

上篇主要就是带大家了解几个概念，这些概念对做题极为重要，请务必掌握。接下来讲解了二分法的中心 - 折半，这个中心需要大家做任何二分都要放到脑子中。

二分法的精髓正如开篇提到的**二分法是一种让未知世界无机可乘的算法**。二分法无论如何我们都可以舍弃一半解，也就是无论如何都可以将解空间砍半。难点就是上面提到的两点：**什么条件** 和 **舍弃哪部分**。

接下来，我们继续下篇。下篇注主要内容是两种类型和四大应用。

其中两种类型主要解决的的是：这道题我的解空间以及明确出来了，如何用代码找出具体的值。而四大应用主要解决的是：如何构造解空间（更多的情况则是如何构建有序序列）以及一些变体。

这两部分都是实操性很强的内容。这里我提醒大家，在理解这两部分内容的同时，请大家务必牢记一个中心**折半**。

两种类型

问题定义

> 这里的问题定义是一个狭义的问题。而如果你理解了这个问题之后，可以将这个具体的问题进行推广以适应更复杂的问题。关于推广，我们之后再谈。

给定一个由数字组成的有序数组 nums，并给你一个数字 target。问 nums 中是否存在 target。如果存在， 则返回其在 nums 中的索引。如果不存在，则返回 - 1。

这是二分查找中最简单的一种形式。当然二分查找也有**很多的变形**，这也是二分查找容易出错，难以掌握的原因。

常见变体有：

- 如果存在多个满足条件的元素，返回最左边满足条件的索引。
- 如果存在多个满足条件的元素，返回最右边满足条件的索引。
- 数组不是整体有序的。 比如先升序再降序，或者先降序再升序。
- 将一维数组变成二维数组。
- 。。。

接下来，我们逐个进行查看。

前提

- 数组是有序的（如果无序，我们也可以考虑排序，不过要注意排序的复杂度）

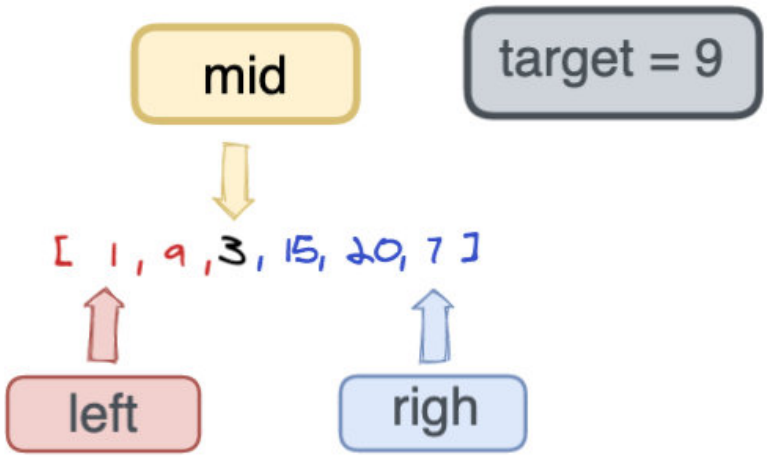
> 这个有序的数组可能是题目直接给的，也可能是你自己构造的。比如求数组的逆序数就可以在自己构造的有序序列上做二分。

术语

为了后面描述问题方便，有必要引入一些约定和术语。

二分查找中使用的术语：

- target —— 要查找的值
- index —— 当前位置
- l 和 r —— 左右指针
- mid —— 左右指针的中点，用来确定我们应该向左查找还是向右查找的索引（其实就是收缩解空间）



值得注意的是，除了 target 是固定不变的，其他都是动态变化的。其中 l 和 r 指的是解空间的上下界，mid 是上下界的中间值， index 是遍历指针，用于控制遍历过程。

查找一个数

前面我们已经对问题进行了定义。接下来，我们需要对定义的问题进行**分析和求解**。

为了更好理解接下来的内容，我们解决最简单的类型 - **查找某一个具体值**。

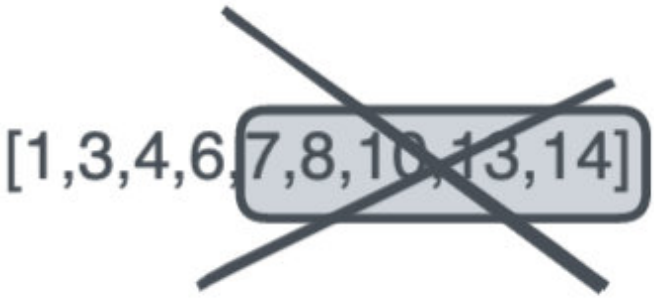
算法描述：

- 先从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；
- 如果目标元素大于中间元素，那么数组中小于中间元素的值都可以排除（由于数组有序，那么相当于是可以排除数组左侧的所有值），解空间可以收缩为 [mid+1, r]。

- 如果目标元素小于中间元素，那么数组中大于中间元素的值都可以排除（由于数组有序，那么相当于是可以排除数组右侧的所有值），解空间可以收缩为 [l, mid - 1]。
- 如果在某一步骤解空间为空，则代表找不到。

举一个具体的例子方便大家增加代入感。假设 nums 为 `[1,3,4,6,7,8,10,13,14]`， target 为 4·。

- 刚开始数组中间的元素为 7
- 7 > 4 ，由于 7 右边的数字都大于 7 ，因此不可能是答案。我们将范围缩写到了 7 的左侧。



- 解空间变成了 [1,3,4,6]，此时中间元素为 3。
- 3 < 4，由于 3 左边的数字都小于 3 ，因此不可能是答案。我们将范围缩写到了 3 的右侧。

- 解空间变成了 [4,6]，此时中间元素为 4，正好是我们要找的，返回其索引 2 即可。

复杂度分析

由于这种搜索算法每一次比较都使搜索范围缩小一半，是典型的二分查找。

- 平均时间复杂度： $O(logN)$
- 最坏时间复杂度： $O(logN)$
- 空间复杂度
  - 迭代: $O(1)$
  - 递归： $O(logN)$（无尾调用消除）

> 后面的复杂度也是类似的，不再赘述。

思维框架

如何将上面的算法转换为容易理解的可执行代码呢?

大家不要小看这样的一个算法。就算是这样一个简简单单，朴实无华的二分查找， 不同的人写出来的差别也是很大的。 如果没有一个**思维框架指导你，不同的时间你可能会写出差异很大的代码。这样的话，犯错的几率会大大增加。这里给大家介绍一个我经常使用的思维框架和代码模板。**

首先定义解空间为 [left, right]，注意是左右都闭合，之后会用到这个点

> 你可以定义别的解空间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的解空间。

- 由于定义的解空间为 [left, right]，因此当 left <= right 的时候，解空间都不为空，此时我们都需要继续搜索。 也就是说终止搜索条件应该为 left <= right。

> 举个例子容易明白一点。 比如对于区间 [4,4]，其包含了一个元素 4，因此解空间不为空，需要继续搜索（试想 4 恰好是我们要找的 target，如果不继续搜索， 会错过正确答案）。而当解空间为 [left, right) 的时候，同样对于 [4,4]，这个时候解空间却是空的，因为这样的一个区间不存在任何数字·。

- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则提前返回 mid（只需要找到一个满足条件的即可）
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候解空间可缩小为 [mid + 1, right] （mid 以及 mid 左侧的数字被我们排除在外）
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候解空间可缩小为 [left, mid - 1] （mid 以及 mid 右侧的数字被我们排除在外）
- 循环结束都没有找到，则说明找不到，返回 -1 表示未找到。

代码模板

Java

```java
public int binarySearch(int[] nums, int target) {
    // 左右都闭合的区间 [l, r]
    int left = 0;
    int right = nums.length - 1;

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        if (nums[mid] < target)
                  // 解空间变为 [mid+1, right]
            left = mid + 1;
        if (nums[mid] > target)
            // 解空间变为 [left, mid - 1]
            right = mid - 1;
    }
    return -1;
}
```

Python

```python
def binarySearch(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (left + right) >> 1
        if nums[mid] == target: return mid
        # 解空间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 解空间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    return -1
```

JavaScript

```
function binarySearch(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target) return mid;
    if (nums[mid] < target)
      // 解空间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 解空间变为 [left, mid - 1]
      right = mid - 1;
  }
  return -1;
}
```

C++

```
int binarySearch(vector<int>& nums, int target){
  if(nums.size() == 0)
    return -1;

  int left = 0, right = nums.size() - 1;
  while(left <= right){
    int mid = left + ((right - left) >> 1);
    if(nums[mid] == target){ return mid; }
    // 解空间变为 [mid+1, right]
    else if(nums[mid] < target)
    left = mid + 1;
    // 解空间变为 [left, mid - 1]
    else
    right = mid - 1;
  }
  return -1;
}
```

寻找最左插入位置

上面我们讲了 `寻找满足条件的值` 。如果找不到，就返回 -1。那如果不是返回 -1，而是返回应该插入的位置，使得插入之后列表仍然有序呢?

比如一个数组 nums: [1,3,4]，target 是 2。我们应该将其插入（注意不是真的插入）的位置是索引 1 的位置，即 [1,**2**,3,4]。因此 `寻找最左插入位置` 应该返回 1，而 `寻找满足条件的位置` 应该返回-1。

另外如果有多个满足条件的值，我们返回最左侧的。 比如一个数组 nums: [1,2,2,2,3,4]，target 是 2，我们应该插入的位置是 1。

思维框架

具体算法：

- 首先定义解空间为 [left, right]，注意是左右都闭合，之后会用到这个点。

> 你可以定义别的解空间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的解空间。

- 由于我们定义的解空间为 [left, right]，因此当 left <= right 的时候，解空间都不为空。 也就是说我们的终止搜索条件为 left <= right。
- 当 A[mid] >= x，说明找到一个备胎，我们令 r = mid - 1 将 mid 从解空间排除，继续看看有没有更好的备胎。
- 当 A[mid] < x，说明 mid 根本就不是答案，直接更新 l = mid + 1，从而将 mid 从解空间排除。
- 最后解空间的 l 就是最好的备胎，备胎转正。

代码模板

Python

```python
def bisect_left(nums, x):
    # 内置 api
    bisect.bisect_left(nums, x)
    # 手写
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if A[mid] >= x: r = mid - 1
        else: l = mid + 1
    return l
```

寻找最右插入位置

思维框架

具体算法：

- 首先定义解空间为 [left, right]，注意是左右都闭合，之后会用到这个点。

> 你可以定义别的解空间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的解空间。

- 由于我们定义的解空间为 [left, right]，因此当 left <= right 的时候，解空间都不为空。 也就是说我们的终止搜索条件为 left <= right。
- 当 A[mid] > x，说明找到一个备胎，我们令 r = mid - 1 将 mid 从解空间排除，继续看看有没有更好的备胎。
- 当 A[mid] <= x，说明 mid 根本就不是答案，直接更新 l = mid + 1，从而将 mid 从解空间排除。
- 最后解空间的 l 就是最好的备胎，备胎转正。

代码模板

Python

```
def bisect_right(nums, x):
    # 内置 api
    bisect.bisect_right(nums, x)
    # 手写
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if A[mid] <= x: l = mid + 1
        else: r = mid - 1
    return l
```

以上就是两种二分的基本形式了。而在实际的写代码过程中，我不会使用**寻找满足条件的值**模板，而是直接使用**最左** 或者 **最右** 插入模板。为什么呢？因为后者包含了前者，并还有前者实现不了的功能。比如我要实现**寻找满足条件的值**，就可直接使用**最左插入**模板找到插入索引 i，只不过最后判断一下 nums[i] 是否等于 target 即可，如果不等于则返回 -1，否则返回 i。这也是为什么我**将二分分为两种类型，而不是三种甚至四种的原因**。

另外最左插入和最右插入可以结合使用从而求出**有序序列**中和 target 相等的数的个数，这在有些时候会是一个考点。代码表示：

```
nums = [1,2,2,2,3,4]
i = bisect.bisect_left(nums, 2) # get 1
j = bisect.bisect_right(nums, 2) # get 4
# j - i 就是 nums 中 2 的个数
```

为了描述方便，以后所有的最左插入二分我都会简称**最左二分**，代码上直接用 bisect.bisect_left 表示，而最右插入二分我都会简称**最右二分**，代码上用 bisect.bisect_right 或者 bisect.bisect 表示。

小结

对于二分题目首先要明确解空间，然后根据一定条件（通常是和中间值比较），舍弃其中一半的解。大家可以先从查找满足条件的值的二分入手，进而学习最左和最右二分。同时大家只需要掌握最左和最右二分即可，因为后者功能大于前者。

对于最左和最右二分，简单用两句话总结一下：

1. 最左二分不断收缩右边界，最终返回左边界
2. 最右二分不断收缩左边界，最终返回右边界

四大应用

基础知识铺垫了差不多了。接下来，我们开始干货技巧。

接下来要讲的：

- 能力检测和计数二分本质差不多，都是**普通二分** 的泛化。
- 前缀和二分和插入排序二分，本质都是在**构建有序序列**。

那让我们开始吧。

能力检测二分

能力检测二分一般是：定义函数 possible， 参数是 mid，返回值是布尔值。外层根据返回值调整"解空间"。

示例代码（以最左二分为例）：

```
def ability_test_bs(nums):
  def possible(mid):
    pass
  l, r = 0, len(A) - 1
  while l <= r:
      mid = (l + r) // 2
      # 只有这里和最左二分不一样
      if possible(mid): l = mid + 1
      else: r = mid - 1
  return l
```

和最左最右二分这两种最最基本的类型相比，能力检测二分**只是将 while 内部的 if 语句调整为了一个函数罢了**。因此能力检测二分也分最左和最右两种基本类型。

基本上大家都可以用这个模式来套。明确了解题的框架，我们最后来看下能力检测二分可以解决哪些问题。这里通过三道题目带大家感受一下，类似的题目还有很多，大家课后自行体会。

875. 爱吃香蕉的珂珂（中等）

题目地址

https://leetcode-cn.com/problems/koko-eating-bananas/description/

题目描述

```
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

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

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
```

前置知识

- 二分查找

公司

- 字节

思路

题目是让我们求**H 小时内吃掉所有香蕉的最小速度**。

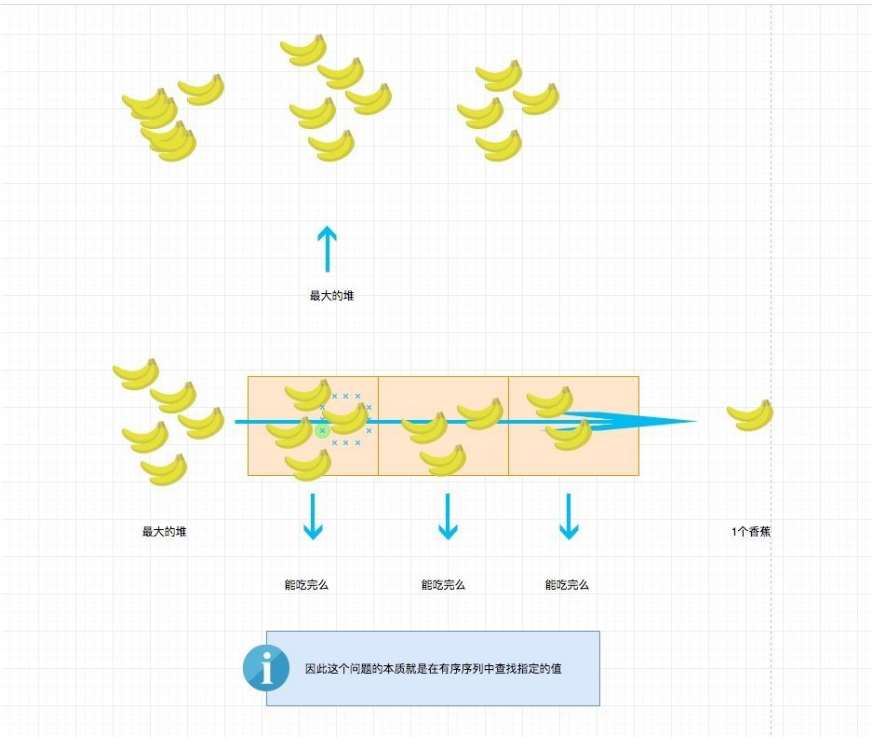
符合直觉的做法是枚举所有可能的速度，找出所有的可以吃完香蕉的速度，接下来选择最小的速度即可。由于需要返回最小的速度，因此选择从小到大枚举会比较好，因为可以提前退出。 这种解法的时间复杂度比较高，为 $O(N * M)$，其中 N 为 piles 长度， M 为 Piles 中最大的数（也就是解空间的最大值）。

观察到需要检测的解空间是个**有序序列**，应该想到可能能够使用二分来解决，而不是线性枚举。可以使用二分解决的关键和前面我们简化的二分问题并无二致，关键点在于**如果速度 k 吃不完所有香蕉，那么所有小于等于 k 的解都可以被排除。**

二分解决的关键在于：

- 明确解空间。 对于这道题来说， 解空间就是 [1,max(piles)]。
- 如何收缩解空间。关键点在于**如果速度 k 吃不完所有香蕉，那么所有小于等于 k 的解都可以被排除。**

综上，我们可以使用最左二分，即不断收缩右边界。



> 香蕉堆的香蕉个数上限是 10^9， 珂珂这也太能吃了吧?

关键点解析

- 二分查找模板

代码

代码支持：Python，JavaScript

Python Code:

```python
class Solution:
    def solve(self, piles, k):
        def possible(mid):
            t = 0
            for pile in piles:
                t += (pile + mid - 1) // mid
            return t <= k

        l, r = 1, max(piles)

        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l
```

JavaScript Code:

```
function canEatAllBananas(piles, H, mid) {
  let h = 0;
  for (let pile of piles) {
    h += Math.ceil(pile / mid);
  }

  return h <= H;
}
/**
 * @param {number[]} piles
 * @param {number} H
 * @return {number}
 */
var minEatingSpeed = function (piles, H) {
  let lo = 1,
    hi = Math.max(...piles);
  // [l, r) ,  左闭右开的好处是如果能找到，那么返回 l 和 r 都是一样
  while (lo <= hi) {
    let mid = lo + ((hi - lo) >> 1);
    if (canEatAllBananas(piles, H, mid)) {
      hi = mid - 1;
    } else {
      lo = mid + 1;
    }
  }

  return lo; //  不能选择hi
};
```

复杂度分析

- 时间复杂度：$O(max(N, N * logM))$，其中 N 为 piles 长度， M 为 Piles 中最大的数。
- 空间复杂度：$O(1)$

最小灯半径（困难）

题目描述

```
You are given a list of integers nums representing coordina

Constraints

n ≤ 100,000 where n is the length of nums
Example 1
Input
nums = [3, 4, 5, 6]
Output
0.5
Explanation
If we place the lamps on 3.5, 4.5 and 5.5 then with r = 0.5
```

前置知识

- 排序
- 二分法

二分法

思路

本题和力扣 475. 供暖器 类似。

这道题的意思是给你一个数组 nums，让你在 [min(nums),max(nums)] 范围内放置 3 个灯，每个灯覆盖半径都是 r，让你求最小的 r。

之所以不选择小于 min(nums) 的位置和大于 max(nums) 的位置是因为没有必要。比如选取了小于 min(nums) 的位置 pos，那么选取 pos **一定不比选择 min(nums) 位置结果更优**。

这道题的核心点还是一样的思维模型，即：

- 确定解空间。这里的解空间其实就是 r。不难看出 r 的下界是 0， 上界是 max(nums) - min(nums)。

> 没必要十分精准，只要不错过正确解即可，这个我们在前面讲过，这里再次强调一下。

- 对于上下界之间的所有可能 x 进行枚举（不妨从小到大枚举），检查半径为 x 是否可以覆盖所有，返回第一个可以覆盖所有的 x 即可。

注意到我们是在一个有序序列进行枚举，因此使用二分就应该想到。可使用二分的核心点在于：如果 x 不行，那么小于 x 的所有半径都必然不行。

接下来的问题就是给定一个半径 x，判断其是否可覆盖所有的房子。

判断其是否可覆盖就是所谓的能力检测，我定义的函数 possible 就是能力检测。

首先**对 nums 进行排序**，这在后面会用到。 然后从左开始模拟放置灯。先在 nums[0] + r 处放置一个灯，其可以覆盖 [0, 2 *r]。由于 nums 已经排好序了，那么这个等可以覆盖到的房间其实就是 nums 中坐标小于等于 2* \ r 所有房间，使用二分查找即可。对于 nums 右侧的所有的房间我们需要继续放置灯，采用同样的方式即可。

能力检测核心代码：

```
def possible(diameter):
    start = nums[0]
    end = start + diameter
    for i in range(LIGHTS):
        idx = bisect_right(nums, end)
        if idx >= N:
            return True
        start = nums[idx]
        end = start + diameter
    return False
```

由于我们想要找到满足条件的最小值，因此可直接套用**最左二分模板**。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, nums):
        nums.sort()
        N = len(nums)
        if N <= 3:
            return 0
        LIGHTS = 3
        # 这里使用的是直径，因此最终返回需要除以 2
        def possible(diameter):
            start = nums[0]
            end = start + diameter
            for i in range(LIGHTS):
                idx = bisect_right(nums, end)
                if idx >= N:
                    return True
                start = nums[idx]
                end = start + diameter
            return False

        l, r = 0, nums[-1] - nums[0]
        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l / 2
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：由于进行了排序， 因此时间复杂度大约是 $O(nlogn)$
- 空间复杂度：取决于排序的空间消耗

778. 水位上升的泳池中游泳（困难）

题目地址

https://leetcode-cn.com/problems/swim-in-rising-water

题目描述

```
在一个 N x N 的坐标方格  grid 中，每一个方格的值 grid[i][j] 表示在

现在开始下雨了。当时间为  t  时，此时雨水导致水池中任意位置的水位为

你从坐标方格的左上平台 (0，0) 出发。最少耗时多久你才能到达坐标方格的右

示例 1:

输入: [[0,2],[1,3]]
输出: 3
解释:
时间为 0 时，你位于坐标方格的位置为 (0, 0)。
此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时

等时间到达 3 时，你才可以游向平台 (1, 1). 因为此时的水位是 3，坐标方
示例 2:

输入: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,
输出: 16
解释:
0 1 2 3 4
24 23 22 21 5
12 13 14 15 16
11 17 18 19 20
10 9 8 7 6

最终的路线用加粗进行了标记。
我们必须等到时间为 16，此时才能保证平台 (0, 0) 和 (4, 4) 是连通的

提示:

2 <= N <= 50.
grid[i][j] 位于区间 [0, ..., N*N - 1] 内。
```

前置知识

- DFS
- 二分

思路

首先明确一下解空间。不难得出，解空间是[0, max(grid)]，其中 max(grid) 表示 grid 中的最大值。

因此一个简单的思路是一个个试。

- 试试 a 可以不
- 试试 a+1 可以不

- 。。。

试试 x 是否可行就是能力检测。

实际上，如果 x 不可以，那么小于 x 的所有值都是不可以的，这正是本题的突破口。基于此，我们同样可使用讲义中的**最左二分**模板解决。

伪代码:

```
def test(x):
    pass
while l <= r:
    mid = (l + r) // 2
    if test(mid, 0, 0):
        r = mid - 1
    else:
        l = mid + 1
return l
```

这个模板会在很多二分中使用。比如典型的计数型二分，典型的就是计算小于等于 x 的有多少，然后根据答案更新解空间。

明确了这点，剩下要做的就是完成能力检测部分 （test 函数） 了。其实这个就是一个普通的二维网格 dfs，我们从 (0,0) 开始在一个二维网格中搜索，直到无法继续或达到 (N-1,N-1)，如果可以达到 (N-1,N-1)，我们返回 true，否则返回 False 即可。对二维网格的 DFS 不熟悉的同学可以看下我之前写的小岛专题

代码

```
class Solution:
    def swimInWater(self, grid: List[List[int]]) -> int:
        l, r = 0, max([max(vec) for vec in grid])
        seen = set()

        def test(mid, x, y):
            if x > len(grid) - 1 or x < 0 or y > len(grid[0
                return False
            if grid[x][y] > mid:
                return False
            if (x, y) == (len(grid) - 1, len(grid[0]) - 1):
                return True
            if (x, y) in seen:
                return False
            seen.add((x, y))
            ans = test(mid, x + 1, y) or test(mid, x - 1,
                                              y) or test(mi
            return ans
        while l <= r:
            mid = (l + r) // 2
            if test(mid, 0, 0):
                r = mid - 1
            else:
                l = mid + 1
            seen = set()
        return l
```

复杂度分析

- 时间复杂度：$O(NlogM)$，其中 M 为 grid 中的最大值， N 为 grid 的总大小。
- 空间复杂度：$O(N)$，其中 N 为 grid 的总大小。

计数二分

计数二分和上面的思路已经代码都基本一致。 直接看代码会清楚一点：

```
def count_bs(nums, k):
  def count_not_greater(mid):
    pass
  l, r = 0, len(A) - 1
  while l <= r:
      mid = (l + r) // 2
      # 只有这里和最左二分不一样
      if count_not_greater(mid) > k: r = mid - 1
      else: l = mid + 1
  return l
```

可以看出只是将 `possible` 变成了 `count_not_greater`，返回值变成了数字而已。

实际上，我们可以将上面的代码稍微改造一下，使得两者更像：

```
def count_bs(nums, k):
  def possible(mid, k):
    # xxx
    return cnt > k
  l, r = 0, len(A) - 1
  while l <= r:
      mid = (l + r) // 2
      if possible(mid, k): r = mid - 1
      else: l = mid + 1
  return l
```

是不是基本一致了？

由于和上面基本一致， 因此这里直接推荐一个题目，大家用我的思路练习一下，看看我的技巧灵不灵。

- 第 k 小的距离对

前缀和二分

前面说了：如果数组全是正的，那么其前缀和就是一个严格递增的数组，基于这个特性，我们可以在其之上做二分。类似的有单调栈/队列。这种题目类型很多，为了节省篇幅就不举例说明了。提出前缀和二分的核心的点在于让大家保持对**有序序列**的敏感度。

插入排序二分

除了上面的前缀和之外，我们还可以自行维护有序序列。一般有两种方式：

- 直接对序列排序。

代码表示：

```
nums.sort()
bisect.bisect_left(nums, x) # 最左二分
bisect.bisect_right(nums, x) # 最右二分
```

- 遍历过程维护一个新的有序序列，有序序列的内容为**已经遍历过的值的集合**。

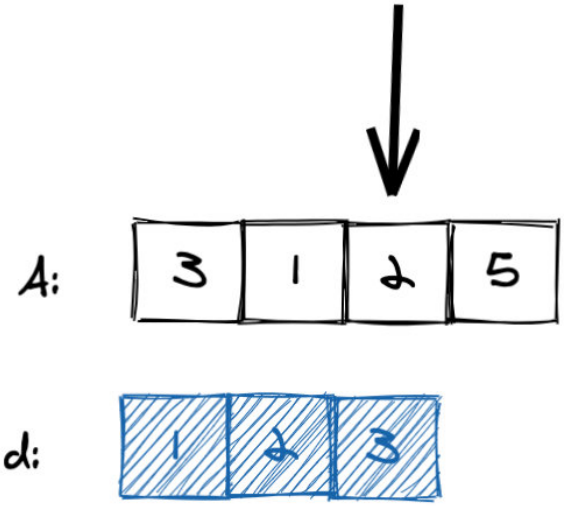
比如无序数组 [3,2,10,5]，遍历到索引为 2 的项（也就是值为 10 的项）时，我们构建的有序序列为 [2,3,10]。

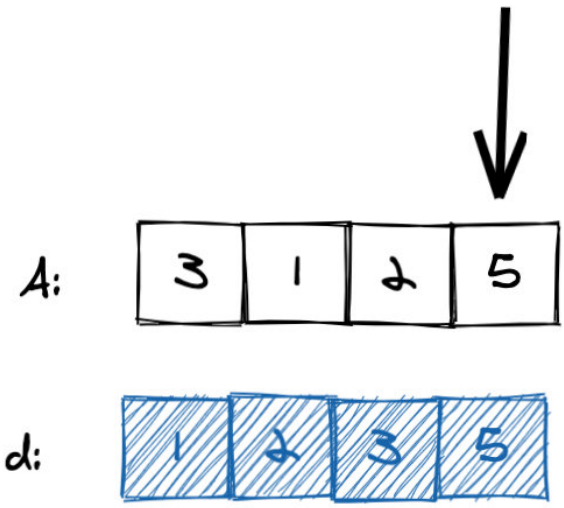
> 注意我描述的是有序序列，并不是指数组，链表等具体的数据结构。而实际上，这个有序序列很多情况下是平衡二叉树。后面题目会体现这一点。

代码表示：

```
d = SortedList()
for a in A:
    d.add(a) # 将 a 添加到 d，并维持 d 中数据有序
```

上面代码的 d 就是有序序列。





理论知识到此为止，接下来通过一个例子来说明。

327. 区间和的个数(困难)

题目地址

https://leetcode-cn.com/problems/count-of-range-sum

题目描述

```
给定一个整数数组 nums 。区间和 S(i, j) 表示在 nums 中，位置从 i 到

请你以下标 i （0 <= i <= nums.length ）为起点，元素个数逐次递增，

当元素和落在范围 [lower, upper] （包含 lower 和 upper）之内时，记

求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的



注意：
最直观的算法复杂度是 O(n2) ，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：
下标 i = 0 时，子数组 [-2]、[-2,5]、[-2,5,-1]，对应元素和分别为 -
下标 i = 1 时，子数组 [5]、[5,-1] ，元素和 5、4 ；没有满足题意的有
下标 i = 2 时，子数组 [-1] ，元素和 -1 ；记录有效区间和 S(2,2) 。
故，共有 3 个有效区间和。


提示：

0 <= nums.length <= 10^4
```

思路

题目很好理解。

由前缀和的性质知道：区间 i 到 j（包含）的和 sum(i,j) = pre[j] - pre[i-1]，其中 pre[i] 为数组前 i 项的和 0 <= i < n。

但是题目中的数字可能是负数，前缀和不一定是单调的啊？这如何是好呢？答案是手动维护前缀和的有序性。

比如 [-2,5,-1] 的前缀和 为 [-2,3,2]，但是我们可以将求手动维护为 [-2,2,3]，这样就有序了。但是这丧失了索引信息，因此这个技巧仅适用于**无需考虑索引，也就是不需要求具体的子序列，只需要知道有这么一个子序列就行了，具体是哪个，我们不关心**。

比如当前的前缀和是 cur，那么前缀和小于等于 cur - lower 有多少个，就说明以当前结尾的区间和大于等于 lower 的有多少个。类似地，前缀和小于等于 cur - upper 有多少个，就说明以当前结尾的区间和大于等于 upper

的有多少个。

基于这个想法，我们可使用二分在 $logn$ 的时间快速求出这两个数字，使用平衡二叉树代替数组可使得插入的时间复杂度降低到 $O(logn)$。Python 可使用 SortedList 来实现， Java 可用 TreeMap 代替。

代码

```
from sortedcontainers import SortedList
class Solution:
    def countRangeSum(self, A: List[int], lower: int, upper
        ans, pre, cur = 0, [0], 0
        for a in A:
            cur += a
            ans += pre.bisect_right(cur - lower) - pre.bise
            pre.add(cur)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(nlogn)$

493. 翻转对（困难）

题目地址

https://leetcode-cn.com/problems/reverse-pairs/

题目描述

```
给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将

你需要返回给定数组中的重要翻转对的数量。

示例 1:

输入: [1,3,2,3,1]
输出: 2


示例 2:

输入: [2,4,3,5,1]
输出: 3


注意:

给定数组的长度不会超过50000。
输入数组中的所有数字都在32位整数的表示范围内。
```

前置知识

- 二分

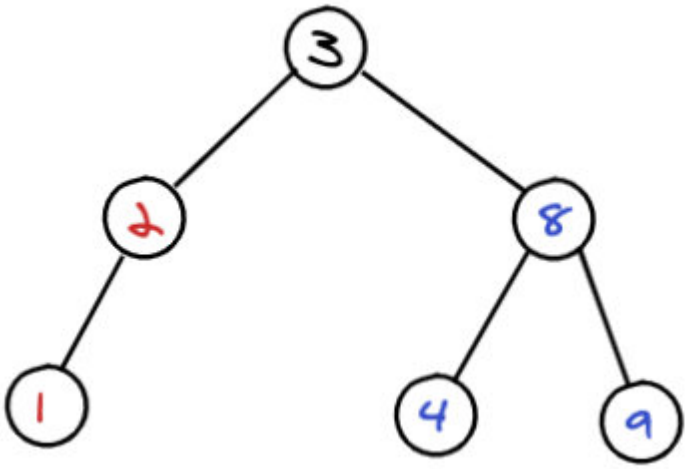
公司

- 暂无

思路

我们可以一边遍历一边维护一个有序序列 d，其中 d 为**已经遍历过的值的集合**。对于每一个位置 0 <= i < n，我们统计 d 中大于 2 * A[i] 的个数，这个个数就是题目要求的翻转对。这里的关键在于 d 中的值是比当前索引小的**全部**值。

我们当然可以线性遍历 d，求出个数。一个更好的方法是在遍历的同时维持 d 是**有序的**，这样我们就可以用二分了。和上面题目一样，使用平衡二叉树代替数组可使得插入的时间复杂度降低到 $O(logn)$。



关键点

- 插入排序二分

代码

- 语言支持：Python3

Python3 Code:

```python
from sortedcontainers import SortedList
class Solution:
    def reversePairs(self, A: List[int]) -> int:
        d = SortedList()
        ans = 0
        for a in A:
            ans += len(d) - d.bisect_right(2*a)
            d.add(a)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(n)$

小结

四个应用讲了两种构造有序序列的方式，分别是前缀和，插入排序，插入排序的部分其实也可以看下我之前写的最长上升子序列系列，那里面的贪心解法就是**自己构造有序序列再二分**的。 另外理论上单调栈/队列也是有

序的，也可是用来做二分，但是相关题目太少了，因此大家只要保持对**有序序列**的敏感度即可。

能力检测二分很常见，不过其仅仅是将普通二分的 if 部分改造成了函数而已。而对于计数二分，其实就是能力检测二分的特例，只不过其太常见了，就将其单独提取出来了。

另外，有时候有序序列也会给你稍微变化一种形式。比如二叉搜索树，大家都知道可以在 $logn$ 的时间完成查找，这个查找过程本质也是二分。二叉查找树有**有序序列**么？有的！二叉查找树的中序遍历恰好就是一个有序序列。因此如果一个数比当前节点值小，一定在左子树（也就是有序序列的左侧），如果一个数比当前节点值大，一定在右子树（也就是有序序列的右侧）。

总结

本文主要讲了两种二分类型：最左和最右，模板已经给大家了，大家只需要根据题目调整解空间和判断条件即可。关于四种应用更多的还是让大家理解二分的核心**折半**。表面上来看，二分就是对有序序列的查找。其实不然，只不过有序序列很容易做二分罢了。因此战术上大家保持对有序序列的敏感度，战略上要明确二分的本质是折半，核心在于什么时候将哪一半折半。

一个问题能否用二分解决的关键在于检测一个值的时候是否可以排除解空间中的一半元素。比如我前面反复提到的**如果 x 不行，那么解空间中所有小于等于 x 的值都不行**。

对于简单题目，通常就是给你一个有序序列，让你在上面找满足条件的位置。顶多变化一点，比如数组局部有序，一维变成二维等。对于这部分可以看下我写的91 算法 - 二分查找讲义

中等题目可能需要让你自己构造有序序列。

困难题则可能是二分和其他专题的结合，比如上面的 778. 水位上升的泳池中游泳（困难），就是二分和搜索（我用的是 DFS）的结合。

以上就是本文的全部内容了， 大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。我是 lucifer，维护西湖区最好的算法题解，Github 超 40K star 。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

另外我整理的 1000 多页的电子书已限时免费下载，大家可以去我的公众号《力扣加加》后台回复电子书获取。

动态规划到底有多难?

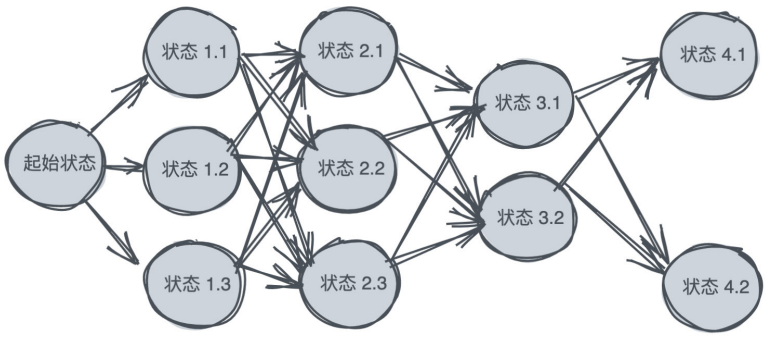
动态规划是一个从其他行业借鉴过来的词语。

它的大概意思先将一件事情分成**若干阶段**，然后通过阶段之间的**转移**达到目标。由于转移的方向通常是多个，因此这个时候就需要**决策**选择具体哪一个转移方向。

动态规划所要解决的事情通常是完成一个具体的目标，而这个目标往往是最优解。并且：

1. 阶段之间可以进行转移，这叫做动态。
2. 达到一个**可行解(目标阶段)** 需要不断地转移，那如何转移才能达到**最优解**？这叫规划。

每个阶段抽象为状态（用圆圈来表示），状态之间可能会发生转化（用箭头表示）。可以画出类似如下的图：



那我们应该做出如何的**决策序列**才能使得结果最优？换句话说就是每一个状态应该如何选择到下一个具体状态，并最终到达目标状态。这就是动态规划研究的问题。

每次决策实际上**不会考虑之后的决策，而只会考虑之前的状态。** 形象点来说，其实是走一步看一步这种短视思维。为什么这种短视可以来求解最优解呢？那是因为：

1. 我们将**所有可能的转移全部模拟了一遍**，最后挑了一个最优解。
2. 无后向性（这个我们后面再说，先卖个关子）

> 而如果你没有模拟所有可能，而直接走了一条最优解，那就是贪心算法了。

没错，动态规划刚开始就是来求最优解的。只不过有的时候顺便可以求总的方案数等其他东西，这其实是**动态规划的副产物**。

好了，我们把动态规划拆成两部分分别进行解释，或许你大概知道了动态规划是一个什么样的东西。但是这对你做题并没有帮助。那算法上的动态规划究竟是个啥呢?

在算法上，动态规划和**查表的递归（也称记忆化递归）**有很多相似的地方。我建议大家先从记忆化递归开始学习。本文也先从记忆化递归开始，逐步讲解到动态规划。

记忆化递归

那么什么是递归？什么是查表（记忆化）？让我们慢慢来看。

什么是递归?

递归是指在函数中**调用函数自身**的方法。

有意义的递归通常会把问题分解成**规模缩小的同类子问题**，当子问题缩写到寻常的时候，我们可以直接知道它的解。然后通过建立递归函数之间的联系（转移）即可解决原问题。

> 是不是和分治有点像? 分治指的是将问题一分为多，然后将多个解合并为一。而这里并不是这个意思。

一个问题要使用递归来解决必须有递归终止条件（算法的有穷性），也就是说递归会逐步缩小规模到寻常。

虽然以下代码也是递归，但由于其无法结束，因此不是一个有效的算法:

```
def f(x):
  return x + f(x - 1)
```

上面的代码除非外界干预，否则会永远执行下去，不会停止。

因此更多的情况应该是:

```
def f(n):
  if n == 1: return 1
  return n + f(n - 1)
```

使用递归通常可以使代码短小，有时候也更可读。算法中使用递归可以**很简单地**完成一些用循环不太容易实现的功能，比如二叉树的左中右序遍历。

递归在算法中有非常广泛的使用，包括现在日趋流行的函数式编程。

> 递归在函数式编程中地位很高。 纯粹的函数式编程中没有循环，只有递归。

实际上，除了在编码上通过函数调用自身实现递归。我们也可以定义递归的数据结构。比如大家所熟知的树，链表等都是递归的数据结构。

```
Node {
    value: any; // 当前节点的值
    children: Array<Node>; // 指向其儿子
}
```

如上代码就是一个多叉树的定义形式，可以看出 children 就是 Node 的集合类，这就是一种**递归的数据结构**。

不仅仅是普通的递归函数

本文中所提到的记忆化递归中的递归函数实际上**指的是特殊的递归函数**，即在普通的递归函数上满足以下几个条件：

1. 递归函数不依赖外部变量
2. 递归函数不改变外部变量

> 满足这两个条件有什么用呢？这是因为我们需要函数给定参数，其返回值也是确定的。这样我们才能记忆化。关于记忆化，我们后面再讲。

如果大家了解函数式编程，实际上这里的递归其实严格来说是**函数式编程中的函数**。如果不了解也没关系，这里的递归函数其实就是**数学中的函数**。

我们来回顾一下数学中的函数：

```
在一个变化过程中，假设有两个变量 x、y，如果对于任意一个 x 都有唯一确定
```

而本文所讲的所有递归都是指的这种数学中的函数。

比如上面的递归函数：

```python
def f(x):
  if x == 1: return 1
  return x + f(x - 1)
```

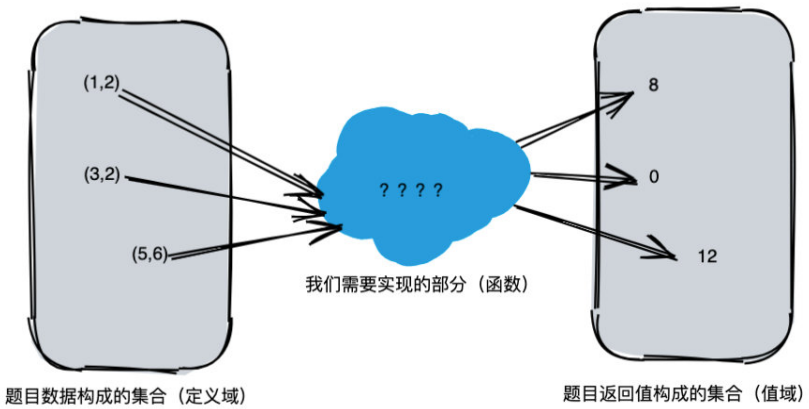
- x 就是自变量，x 的所有可能的返回值构成的集合就是定义域。
- f(x) 就是函数。
- f(x) 的所有可能的返回值构成的集合就是值域。

自变量也可以有多个，对应递归函数的参数可以有多个，比如 f(x1, x2, x3)。

通过函数来描述问题，并通过函数的调用关系来描述问题间的关系就是记忆化递归的核心内容。

每一个动态规划问题，实际上都可以抽象为一个数学上的函数。这个函数的自变量集合就是题目的所有取值，值域就是题目要求的答案的所有可能。我们的目标其实就是填充这个函数的内容，使得给定自变量 x，能够唯一映射到一个值 y。（当然自变量可能有多个，对应递归函数参数可能有多个）

解决动态规划问题可以看成是填充函数这个黑盒，使得定义域中的数并正确地映射到值域。



递归并不是算法，它是和迭代对应的一种编程方法。只不过，我们通常借助递归去分解问题而已。比如我们定义一个递归函数 f(n)，用 f(n) 来描述问题。就和使用普通动态规划 f[n] 描述问题是一样的，这里的 f 是 dp 数组。

什么是记忆化?

为了大家能够更好地对本节内容进行理解，我们通过一个例子来切入：

一个人爬楼梯，每次只能爬 1 个或 2 个台阶，假设有 n 个台阶，那么这个人有多少种不同的爬楼梯方法?

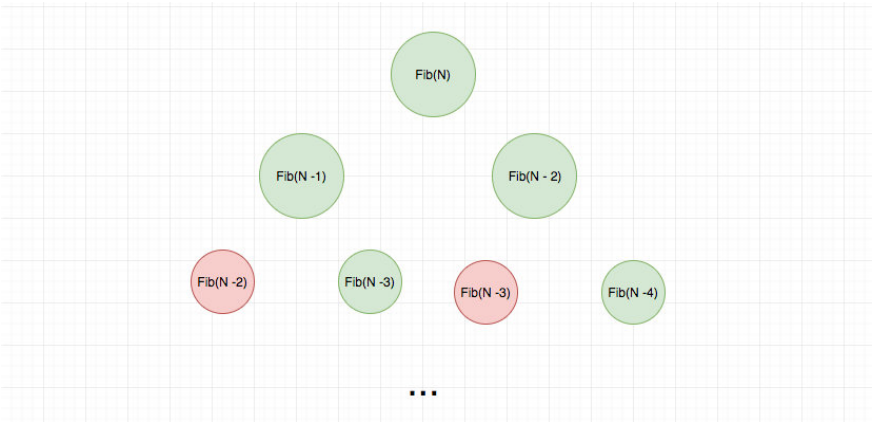
思路：

由于**第 n 级台阶一定是从 n - 1 级台阶或者 n - 2 级台阶来的**，因此到第 n 级台阶的数目就是 `到第 n − 1 级台阶的数目加上到第 n − 1 级台阶的数目` 。

递归代码：

```
function climbStairs(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;
  return climbStairs(n - 1) + climbStairs(n - 2);
}
```

我们用一个递归树来直观感受以下（每一个圆圈表示一个子问题）：



红色表示重复的计算。即 Fib(N-2) 和 Fib(N-3) 都被计算了两次，实际上计算一次就够了。比如第一次计算出了 Fib(N-2) 的值，那么下次再次需要计算 Fib(N-2)的时候，可以直接将上次计算的结果返回。之所以可以这么做的原因正是前文提到的**我们的递归函数是数学中的函数，也就是说参数一定，那么返回值也一定不会变**，因此下次如果碰到相同的参数，我们就可以**将上次计算过的值直接返回，而不必重新计算**。这样节省的时间就等价于重叠子问题的个数。

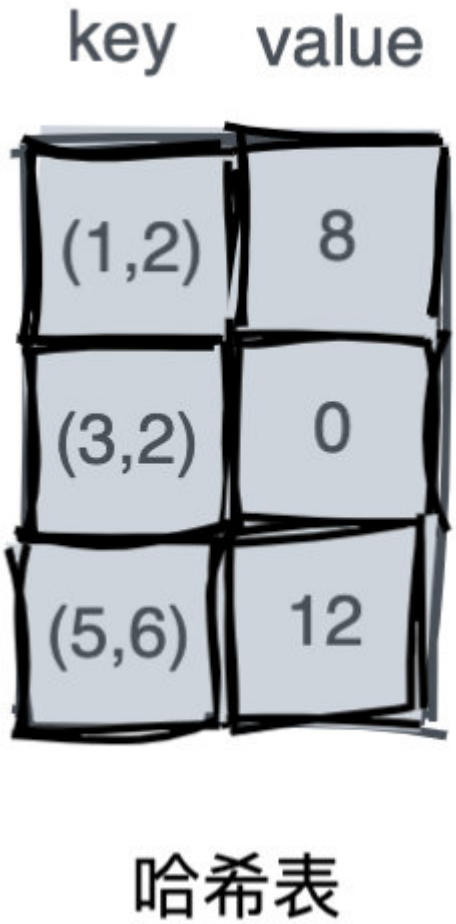
以这道题来说，本来需要计算 2^n 次，而如果使用了记忆化，只需要计算 n 次，就是这么神奇。

代码上，我们可以使用一个 hashtable 去缓存中间计算结果，从而省去不必要的计算。

我们使用记忆化来改造上面的代码：

```
memo = {}
def climbStairs(n):
  if n == 1:return 1
  if n == 2: return 2
  if n in memo: return memo[n]
  ans = func(n - 1) + func(n-2)
  memo[n] = ans
  return ans
climbStairs(10)
```

这里我使用了一个名为 **memo 的哈希表来存储递归函数的返回值，其中 key 为参数，value 为递归函数的返回值。**



> key 的形式为 (x, y)，表示的是一个元祖。通常动态规划的参数有多个，我们就可以使用元祖的方式来记忆化。或者也可采取多维数组的形式。对于上图来说，就可使用二维数组来表示。

大家可以通过删除和添加代码中的 memo 来感受一下**记忆化**的作用。

小结

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。这里我列举了几道算法题目，这几道算法题目都可以用递归轻松写出来：

- 递归实现 sum
- 二叉树的遍历
- 走楼梯问题

- 汉诺塔问题
- 杨辉三角

递归中**如果**存在重复计算（我们称重叠子问题，下文会讲到），那就是使用记忆化递归（或动态规划）解题的强有力信号之一。可以看出动态规划的核心就是使用记忆化的手段消除重复子问题的计算，如果这种重复子问题的规模是指数或者更高规模，那么记忆化递归（或动态规划）带来的收益会非常大。

为了消除这种重复计算，我们可使用查表的方式。即一边递归一边使用“记录表”（比如哈希表或者数组）记录我们已经计算过的情况，当下次再次碰到的时候，如果之前已经计算了，那么直接返回即可，这样就避免了重复计算。下文要讲的**动态规划中 DP 数组其实和这里“记录表”的作用是一样的**。

如果你刚开始接触递归， 建议大家先去练习一下递归再往后看。一个简单练习递归的方式是将你写的迭代全部改成递归形式。比如你写了一个程序，功能是“将一个字符串逆序输出”，那么使用迭代将其写出来会非常容易，那么你是否可以使用递归写出来呢？通过这样的练习，可以让你逐步适应使用递归来写程序。

当你已经适应了递归的时候，那就让我们继续学习动态规划吧！

动态规划

讲了这么多递归和记忆化，终于到了我们的主角登场了。

动态规划的基本概念

我们先来学习动态规划最重要的两个概念：最优子结构和无后效性。

其中：

- 无后效性决定了是否可使用动态规划来解决。
- 最优子结构决定了具体如何解决。

最优子结构

动态规划常常适用于有重叠子问题和最优子结构性质的问题。前面讲了重叠子问题，那么最优子结构是什么？这是我从维基百科找的定义：

> 如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子

举个例子：如果考试中的分数定义为 f，那么这个问题就可以被分解为语文，数学，英语等子问题。显然子问题最优的时候，总分这个大的问题的解也是最优的。

再比如 01 背包问题：定义 f(weights, values, capicity)。如果我们想要求 f([1,2,3], [2,2,4], 10) 的最优解。我们可以将其划分为如下子问题：

- `将第三件物品装进背包`，也就是 f([1,2], [2,2], 10)
- 和 `不将第三件物品装进背包`，也就是 f([1,2,3], [2,2,4], 9)

> 显然这两个问题还是复杂，我们需要进一步拆解。不过，这里不是讲如何拆解的。

原问题 f([1,2,3], [2,2,4], 10) 等于以上两个子问题的最大值。只有两个子问题都是**最优的**时候整体才是最优的，这是因为子问题之间不会相互影响。

无后效性

即子问题的解一旦确定，就不再改变，不受在这之后、包含它的更大的问题的求解决策影响。

继续以上面两个例子来说。

- 数学考得高不能影响英语（现实其实可能影响，比如时间一定，投入英语多，其他科目就少了）。
- 背包问题中 f([1,2,3], [2,2,4], 10) 选择是否拿第三件物品，不应该影响是否拿前面的物品。比如题目规定了拿了第三件物品之后，第二件物品的价值就会变低或变高）。这种情况就不满足无后向性。

动态规划三要素

状态定义

动态规划的中心点是什么？如果让我说的话，那就是**定义状态**。

动态规划解题的第一步就是定义状态。定义好了状态，就可以画出递归树，聚焦最优子结构写转移方程就好了，因此我才说状态定义是动态规划的核心，动态规划问题的状态确实不容易看出。

但是一旦你能把状态定义好了，那就可以顺藤摸瓜画出递归树，画出递归树之后就聚焦最优子结构就行了。但是能够画出递归树的前提是：对问题进行划分，专业点来说就是定义状态。那怎么才能定义出状态呢?

好在状态的定义都有特点的套路。 比如一个字符串的状态，通常是 dp[i] 表示字符串 s 以 i 结尾的 ....。 比如两个字符串的状态，通常是 dp[i][j] 表示字符串 s1 以 i 结尾，s2 以 j 结尾的 ....。

也就是说状态的定义通常有不同的套路，大家可以在做题的过程中进行学习和总结。但是这种套路非常多，那怎么搞定呢?

说实话，只能多练习，在练习的过程中总结套路。具体的套路参考后面的**动态规划的题型** 部分内容。之后大家就可以针对不同的题型，去思考大概的状态定义方向。

两个例子

关于状态定义，真的非常重要，以至于我将其列为动态规划的核心。因此我觉得有必要举几个例子来进行说明。我直接从力扣的动态规划专题中抽取前两道给大家讲讲。

第一道题：《5. 最长回文子串》难度中等

给你一个字符串 s，找到 s 中最长的回文子串。

示例 1：

输入：s = "babad"
输出："bab"
解释："aba" 同样是符合题意的答案。
示例 2：

输入：s = "cbbd"
输出："bb"
示例 3：

输入：s = "a"
输出："a"
示例 4：

输入：s = "ac"
输出："a"

提示：

1 <= s.length <= 1000
s 仅由数字和英文字母（大写和/或小写）组成

这道题入参是一个字符串，那我们要将其转化为规模更小的子问题，那无疑就是字符串变得更短的问题，临界条件也应该是空字符串或者一个字符这样。

因此：

- 一种定义状态的方式就是 f(s1)，含义是字符串 s1 的最长回文子串，其中 s1 就是题目中的字符串 s 的子串，那么答案就是 f(s)。
- 由于规模更小指的是字符串变得更短，而描述字符串我们也可以用两个变量来描述，这样实际上还省去了开辟字符串的开销。两个变量可以是**起点索引 + 子串长度**，也可以是**终点索引 + 子串长度**，也可以是**起点坐标 + 终点坐标**。随你喜欢，这里我就用**起点坐标 + 终点坐标**。那么状态定义就是 f(start, end)，含义是子串 s[start:end+1]的最长回文子串，那么答案就是 f(0, len(s) - 1)

> s[start:end+1] 指的是包含 s[start]，而不包含 s[end+1] 的连续子串。

这无疑是一种定义状态的方式，但是一旦我们这样去定义就会发现：状态转移方程会变得难以确定（实际上很多动态规划都有这个问题，比如最长上升子序列问题）。那究竟如何定义状态呢？我会在稍后的状态转移方程继续完成这道题。我们先来看下一道题。

第二道题：《10. 正则表达式匹配》难度困难

```
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

'.' 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。


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 = "a*"
输出：true
解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元
示例 3：

输入：s = "ab" p = ".*"
输出：true
解释：".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
示例 4：

输入：s = "aab" p = "c*a*b"
输出：true
解释：因为 '*' 表示零个或多个，这里 'c' 为 0 个，'a' 被重复一次。因
示例 5：

输入：s = "mississippi" p = "mis*is*p*."
输出：false


提示：

0 <= s.length <= 20
0 <= p.length <= 30
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 *。
保证每次出现字符 * 时，前面都匹配到有效的字符
```

这道题入参有两个， 一个是 s，一个是 p。沿用上面的思路，我们有两种定义状态的方式。

- 一种定义状态的方式就是 f(s1, p1)，含义是 p1 是否可匹配字符串 s1，其中 s1 就是题目中的字符串 s 的子串，p1 就是题目中的字符串 p 的子串,那么答案就是 f(s, p)。
- 另一种是 f(s_start, s_end, p_start, p_end)，含义是子串 p1[p_start:p_end+1] 是否可以匹配字符串 s[s_start:s_end+1]，那么答案就是 f(0, len(s) - 1, 0, len(p) - 1)

而这道题实际上我们也可采用更简单的状态定义方式，不过基本思路都是差不多的。我仍旧卖个关子，后面讲转移方程再揭晓。

搞定了状态定义，你会发现时间空间复杂度都变得很明显了。这也是为啥我反复强调状态定义是动态规划的核心。

时间空间复杂度怎么个明显法了呢?

首先空间复杂度，我刚才说了动态规划其实就是查表的暴力法，因此动态规划的空间复杂度打底就是表的大小。再直白一点就是上面的哈希表 memo 的大小。而 **memo**的大小基本就是状态的个数。状态个数是多少呢? 这不就取决你状态怎么定义了么？比如上面的 f(s1, p1) 。状态的多少是多少呢？很明显就是每个参数的取值范围大小的笛卡尔积。s1 的所有可能取值有 len(s) 种，p1 的所有可能有 len(p)种，那么总的状态大小就是 len(s) * len(p)。那空间复杂度是 $O(m * n)$，其中 m 和 n 分别为 s 和 p 的大小。

> 我说空间复杂度打底是状态个数， 这里暂时先不考虑状态压缩的情况。

其次是时间复杂度。时间复杂度就比较难说了。但是由于我们**无论如何都要枚举所有状态**，因此**时间复杂度打底就是状态总数**。以上面的状态定义方式，时间复杂度打底就是$O(m * n)$。

如果你枚举每一个状态都需要和 s 的每一个字符计算一下，那时间复杂度就是 $O(m^2 * n)$。

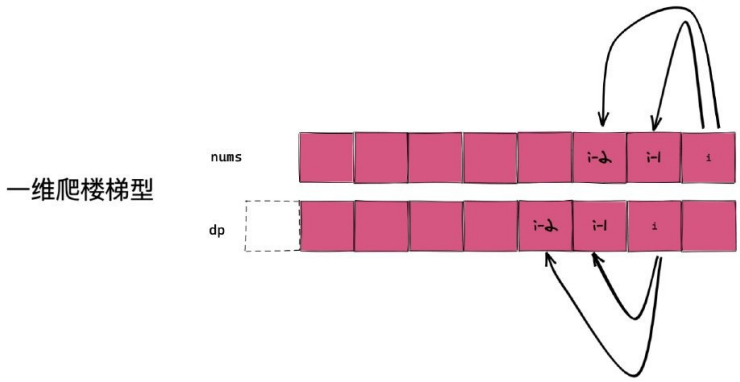
以上面的爬楼梯的例子来说，我们定义状态 f(n) 表示到达第 n 级台阶的方法数，那么状态总数就是 n，空间复杂度和时间复杂度打底就是 n 了。（仍然不考虑滚动数组优化）

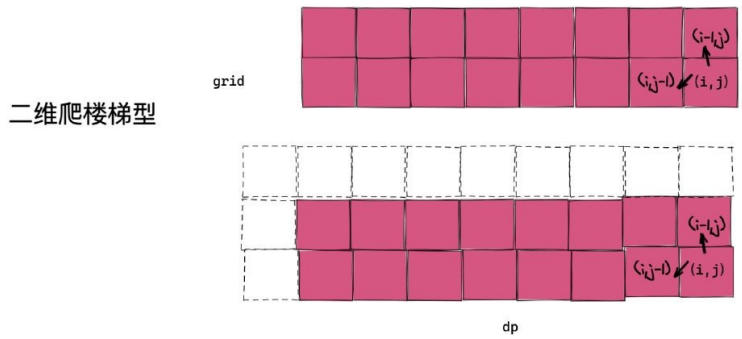
再举个例子：62. 不同路径

```
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

问总共有多少条不同的路径?
```

这道题是和上面的爬楼梯很像，只不过从一维变成了二维，我把它叫做**二维爬楼梯**，类似的换皮题还很多，大家慢慢体会。

这道题我定义状态为 f(i, j) 表示机器人到达点 (i,j) 的总的路径数。那么状态总数就是 i 和 j 的取值的笛卡尔积，也就是 m * n 。





总的来说，动态规划的空间和时间复杂度**打底就是状态的个数**，而状态的个数通常是参数的笛卡尔积，这是由动态规划的无后向性决定的。

临界条件是比较最容易的

当你定义好了状态，剩下就三件事了：

1. 临界条件
2. 状态转移方程
3. 枚举状态

在上面讲解的爬楼梯问题中，如果我们用 f(n) 表示爬 n 级台阶有多少种方法的话，那么：

```
f(1) 与 f(2) 就是【边界】
f(n) = f(n-1) + f(n-2) 就是【状态转移公式】
```

我用动态规划的形式表示一下：

```
dp[0] 与 dp[1] 就是【边界】
dp[n] = dp[n - 1] + dp[n - 2] 就是【状态转移方程】
```

可以看出记忆化递归和动态规划是多么的相似。

实际上临界条件相对简单，大家只有多刷几道题，里面就有感觉。困难的是找到状态转移方程和枚举状态。这两个核心点的都建立在**已经抽象好了状态**的基础上。比如爬楼梯的问题，如果我们用 f(n) 表示爬 n 级台阶有多少种方法的话，那么 f(1), f(2), ... 就是各个**独立的状态**。

搞定了状态的定义，那么我们来看下状态转移方程。

状态转移方程

动态规划中当前阶段的状态往往是上一阶段状态和上一阶段决策的结果。这里有两个关键字，分别是：

- 上一阶段状态
- 上一阶段决策

也就是说，如果给定了第 k 阶段的状态 s[k] 以及决策 choice(s[k])，则第 k+1 阶段的状态 s[k+1] 也就完全确定，用公式表示就是：s[k] + choice(s[k]) -> s[k+1]， 这就是状态转移方程。需要注意的是 choice 可能有多个，因此每个阶段的状态 s[k+1]也会有多个。

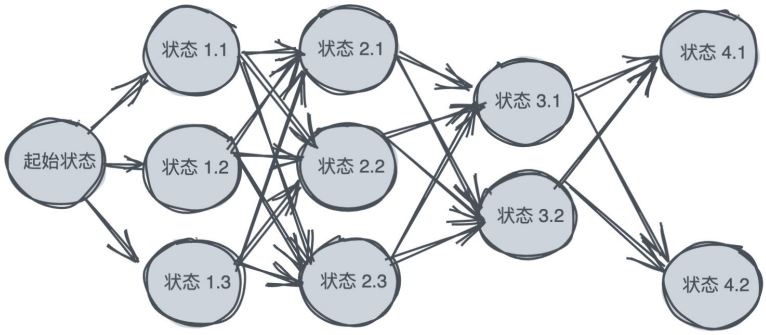
继续以上面的爬楼梯问题来说，爬楼梯问题由于上第 n 级台阶一定是从 n - 1 或者 n - 2 来的，因此 上第 n 级台阶的数目就是 `上 n - 1 级台阶的数目加上 n - 1 级台阶的数目` 。

上面的这个理解是核心， 它就是我们的状态转移方程，用代码表示就是 `f(n) = f(n - 1) + f(n - 2)` 。

实际操作的过程，有可能题目和爬楼梯一样直观，我们不难想到。也可能隐藏很深或者维度过高。 如果你实在想不到，可以尝试画图打开思路，这也是我刚学习动态规划时候的方法。当你做题量上去了，你的题感就会来，那个时候就可以不用画图了。

比如我们定义了状态方程，据此我们定义初始状态和目标状态。然后聚焦最优子结构，**思考每一个状态究竟如何进行扩展使得离目标状态越来越近**。

如下图所示：



理论差不多先这样，接下来来几个实战消化一下。

ok，接下来是解密环节。上面两道题我们都没有讲转移方程，我们在这里补上。

第一道题：《5. 最长回文子串》难度中等。上面我们的两种状态定义都不好，而我可以在上面的基础上**稍微变动一点**就可以使得转移方程变得非常好写。这个技巧在很多动态题目都有体现，比如最长上升子序列等，**需要大家掌握**。

以上面提到的 f(start, end) 来说，含义是子串 s[start:end+1]的最长回文子串。表示方式我们不变，只是将含义变成子串 s[start:end+1]的最长回文子串，**且必须包含 start 和 end**。经过这样的定义，实际上我们也没有必要定义 f(start, end)的返回值是长度了，而仅仅是布尔值就行了。如果返回 true， 则最长回文子串就是 end - start + 1，否则就是 0。

这样转移方程就可以写为：

```
f(i,j)=f(i+1,j-1) and s[i] == s[j]
```

第二道题：《10. 正则表达式匹配》难度困难。

以我们分析的 f(s_start, s_end, p_start, p_end) 来说，含义是子串 p1[p_start:p_end+1] 是否可以匹配字符串 s[s_start:s_end+1]。

实际上，我们可以定义更简单的方式，那就是 f(s_end, p_end)，含义是子串 p1[:p_end+1] 是否可以匹配字符串 s[:s_end+1]。也就是说固定起点为索引 0，这同样也是一个**很常见的技巧，请务必掌握。**

这样转移方程就可以写为：

1. if p[j] 是小写字母，是否匹配取决于 s[i] 是否等于 p[j]:

$$f(i,j) = \begin{cases} f(i-1,j-1) & s[i] == p[j] \\ false & s[i] != p[j] \end{cases}$$

1. if p[j] == '.'，一定可匹配:

```
f(i,j)=f(i-1,j-1)
```

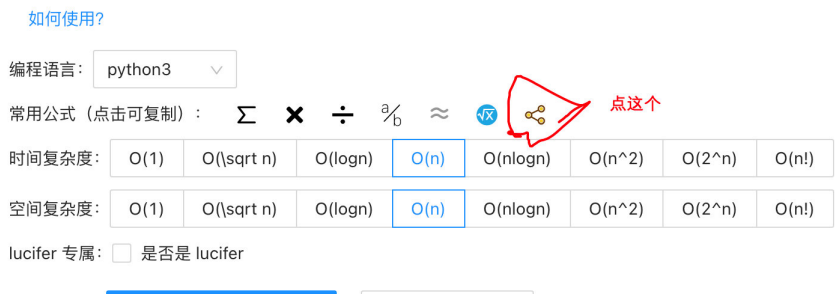
1. if p[j] == '*'，表示 p 可以匹配 s 第 j-1 个字符匹配任意次:

$$f(i,j) = \begin{cases} f(i-1,j) & match \quad 1+ \quad times \\ f(i,j-2) & match \quad 0 \quad time \end{cases}$$

相信你能分析到这里，写出代码就不是难事了。具体代码可参考我的力扣题解仓库，咱就不在这里讲了。

注意到了么？所有的状态转移方程我都使用了上述的数学公式来描述。没错，所有的转移方程都可以这样描述。我建议大家**做每一道动态规划题目都写出这样的公式**，起初你可能觉得很烦麻烦。不过相信我，你坚持下去，会发现自己慢慢变强大。就好像我强烈建议你每一道题都分析好复杂度一样。动态规划不仅要搞懂转移方程，还要自己像我那样完整地用数学公式写出来。

是不是觉得状态转移方程写起来麻烦？这里我给大家介绍一个小技巧，那就是使用 latex，latex 语法可以方便地写出这样的公式。另外西法还贴心地写了**一键生成动态规划转移方程公式**的功能，帮助大家以最快速度生成公诉处。 插件地址：https://leetcode-pp.github.io/leetcode-cheat/?tab=solution-template



状态转移方程实在是没有什么灵丹妙药，不同的题目有不同的解法。状态转移方程同时也是解决动态规划问题中最最困难和关键的点，大家一定要多多练习，提高题感。接下来，我们来看下不那么困难，但是新手疑问比较多的问题 - **如何枚举状态**。

当然状态转移方程可能不止一个， 不同的转移方程对应的效率也可能大相径庭，这个就是比较玄学的话题了，需要大家在做题的过程中领悟。

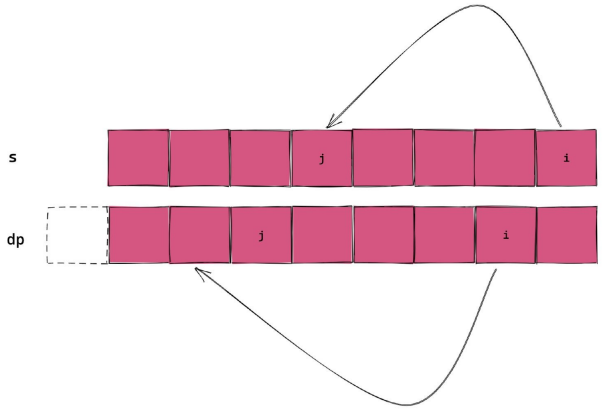
如何枚举状态

前面说了如何枚举状态，才能不重不漏是枚举状态的关键所在。

- 如果是一维状态，那么我们使用一层循环可以搞定。

```
for i in range(1, n + 1):
  pass
```

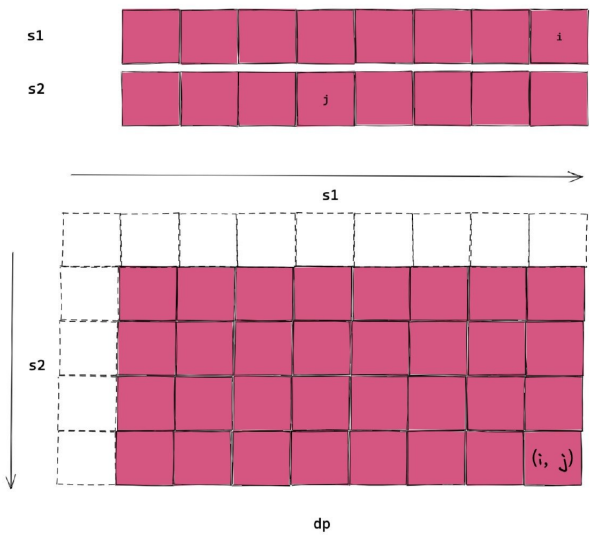
单字符串型



- 如果是两维状态，那么我们使用两层循环可以搞定。

```
for i in range(1, m + 1):
  for j in range(1, n + 1):
    pass
```

双字符串型



- 。。。

但是实际操作的过程有很多细节比如:

- 一维状态我是先枚举左边的还是右边的？（从左到右遍历还是从右到左遍历）
- 二维状态我是先枚举左上边的还是右上的，还是左下的还是右下的？
- 里层循环和外层循环的位置关系（可以互换么）
- 。。。

其实这个东西和很多因素有关，很难总结出一个规律，而且我认为也完全没有必要去总结规律。

不过这里我还是总结了一个关键点，那就是：

- **如果你没有使用滚动数组的技巧**，那么遍历顺序取决于状态转移方程。比如:

```
for i in range(1, n + 1):
  dp[i] = dp[i - 1] + 1
```

那么我们就需要从左到右遍历，原因很简单，因为 dp[i] 依赖于 dp[i - 1]，因此计算 dp[i] 的时候， dp[i - 1] 需要已经计算好了。

> 二维的也是一样的，大家可以试试。

- **如果你使用了滚动数组的技巧**，则怎么遍历都可以，但是不同的遍历意义通常不不同的。比如我将二维的压缩到了一维：

```
for i in range(1, n + 1):
  for j in range(1, n + 1):
    dp[j] = dp[j - 1] + 1;
```

这样是可以的。 dp[j - 1] 实际上指的是压缩前的 dp[i][j - 1]

而：

```
for i in range(1, n + 1):
  #  倒着遍历
  for j in range(n, 0, -1):
    dp[j] = dp[j - 1] + 1;
```

这样也是可以的。 但是 dp[j - 1] 实际上指的是压缩前的 dp[i - 1][j - 1]。因此实际中采用怎么样的遍历手段取决于题目。我特意写了一个 【完全背包问题】套路题（1449. 数位成本和为目标值的最大数字 文章，通过一个具体的例子告诉大家不同的遍历有什么实际不同，强烈建议大家看看，并顺手给个三连。

- 关于里外循环的问题，其实和上面原理类似。

这个比较微妙，大家可以参考这篇文章理解一下 0518.coin-change-2。

小结

关于如何确定临界条件通常是比较简单的，多做几个题就可以快速掌握。

关于如何确定状态转移方程，这个其实比较困难。 不过所幸的是，这些套路性比较强， 比如一个字符串的状态，通常是 dp[i] 表示字符串 s 以 i 结尾的 ....。 比如两个字符串的状态，通常是 dp[i][j] 表示字符串 s1 以 i 结尾，s2 以 j 结尾的 ....。 这样遇到新的题目可以往上套， 实在套不出那就先老实画图，不断观察，提高题感。

关于如何枚举状态，如果没有滚动数组， 那么根据转移方程决定如何枚举即可。 如果用了滚动数组，那么要注意压缩后和压缩前的 dp 对应关系即可。

动态规划 VS 记忆化递归

上面我们用记忆化递归的问题巧妙地解决了爬楼梯问题。 那么动态规划是怎么解决这个问题呢?

答案也是“查表”，我们平常写的 dp table 就是表，其实这个 dp table 和上面的 memo 没啥差别。

而一般我们写的 dp table，**数组的索引通常对应记忆化递归的函数参数，值对应递归函数的返回值。**

看起来两者似乎**没任何思想上的差异，区别的仅仅是写法**？？ 没错。不过这种写法上的差异还会带来一些别的相关差异，这点我们之后再讲。

如果上面的爬楼梯问题，使用动态规划，代码是怎么样的呢？我们来看下：

```
function climbStairs(n) {
  if (n == 1) return 1;
  const dp = new Array(n);
  dp[0] = 1;
  dp[1] = 2;

  for (let i = 2; i < n; i++) {
    dp[i] = dp[i - 1] + dp[i - 2];
  }
  return dp[dp.length - 1];
}
```

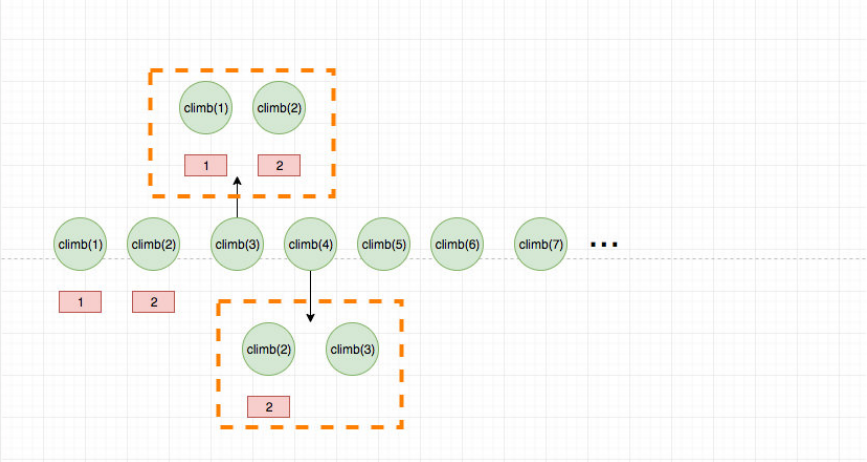
大家现在不会也没关系，我们将**前文的递归的代码稍微改造一下**。其实就是将函数的名字改一下：

```
function dp(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;
  return dp(n - 1) + dp(n - 2);
}
```

经过这样的变化。我们将 dp[n] 和 dp(n) 对比看，这样是不是有点理解了呢? 其实他们的区别只不过是**递归用调用栈枚举状态， 而动态规划使用迭代枚举状态。**

> 如果需要多个维度枚举，那么记忆化递归内部也可以使用迭代进行枚举，比如最长上升子序列问题。

动态规划的查表过程如果画成图，就是这样的：



> 虚线代表的是查表过程

滚动数组优化

爬楼梯我们并没有必要使用一维数组，而是借助两个变量来实现的，空间复杂度是 O(1)。代码：

```
function climbStairs(n) {
  if (n === 1) return 1;
  if (n === 2) return 2;

  let a = 1;
  let b = 2;
  let temp;

  for (let i = 3; i <= n; i++) {
    temp = a + b;
    a = b;
    b = temp;
  }

  return temp;
}
```

之所以能这么做，是因为爬楼梯问题的状态转移方程中**当前状态只和前两个状态有关**，因此只需要存储这两个即可。 动态规划问题有很多这种讨巧的方式，这个技巧叫做滚动数组。

这道题目是动态规划中最简单的问题了，因为仅涉及到单个因素的变化，如果涉及到多个因素，就比较复杂了，比如著名的背包问题，挖金矿问题等。

对于单个因素的，我们最多只需要一个一维数组即可，对于如背包问题我们需要二维数组等更高纬度。

回答上面的问题：记忆化递归和动态规划除了一个用递归一个用迭代，其他没差别。那两者有啥区别呢？我觉得最大的区别就是记忆化递归无法使用滚动数组优化（不信你用上面的爬楼梯试一下），记忆化调用栈的开销比较大（复杂度不变，你可以认为空间复杂度常数项更大），不过几乎不至于 TLE 或者 MLE。**因此我的建议就是没空间优化需求直接就记忆化，否则用迭代 dp**。

再次强调一下：

- 如果说递归是从问题的结果倒推，直到问题的规模缩小到寻常。 那么动态规划就是从寻常入手， 逐步扩大规模到最优子结构。
- 记忆化递归和动态规划没有本质不同。都是枚举状态，并根据状态直接的联系逐步推导求解。
- 动态规划性能通常更好。 一方面是递归的栈开销，一方面是滚动数组的技巧。

动态规划的基本类型

- 背包 DP（这个我们专门开了一个专题讲）
- 区间 DP

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。令状态 $f(i,j)$ 表示将下标位置 i 到 j 的所有元素合并能获得的价值的最大值，那么 $f(i,j)=\max{f(i,k)+f(k+1,j)+cost}$，$cost$ 为将这两组元素合并起来的代价。

区间 DP 的特点：

合并：即将两个或多个部分进行整合，当然也可以反过来；

特征：能将问题分解为能两两合并的形式；

求解：对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

推荐两道题：

- 877. 石子游戏

- 312. 戳气球
- 状压 DP

关于状压 DP 可以参考下我之前写过的一篇文章： 状压 DP 是什么？这篇题解带你入门

- 数位 DP

数位 DP 通常是这：给定一个闭区间 ，让你求这个区间中满足**某种条件**的数的总数。

推荐一道题 Increasing-Digits

- 计数 DP 和 概率 DP

这两个我就不多说。因为没啥规律。

之所以列举计数 DP 是因为两个原因：

1. 让大家知道确实有这个题型。
2. 计数是动态规划的副产物。

概率 DP 比较特殊，概率 DP 的状态转移公式一般是说一个状态**有多大的概率从某一个状态转移过来**，更像是期望的计算，因此也叫期望 DP。

推荐一道题: 837. 新 21 点

更多题目类型以及推荐题目见刷题插件的学习路线。插件获取方式：公众号力扣加加回复插件。

什么时候用记忆化递归?

- 从数组两端同时进行遍历的时候使用记忆化递归方便，其实也就是区间 DP（range dp）。比如石子游戏，再比如这道题 https://binarysearch.com/problems/Make-a-Palindrome-by-Inserting-Characters

如果区间 dp 你的遍历方式大概需要这样：

```
class Solution:
    def solve(self, s):
        n = len(s)
        dp = [[0] * n for _ in range(n)]
        # 右边界倒序遍历
        for i in range(n - 1, -1, -1):
            # 左边界正序遍历
            for j in range(i + 1, n):
                # do something
        return  dp[0][m-1] # 一般都是使用这个区间作为答案
```

如果使用记忆化递归则不需考虑遍历方式的问题。

代码：

```
class Solution:
    def solve(self, s):
        @lru_cache(None)
        def helper(l, r):
            if l >= r:
                return 0

            if s[l] == s[r]:
                return helper(l + 1, r - 1)

            return 1 + min(helper(l + 1, r), helper(l, r -

        return helper(0, len(s) - 1)
```

- **选择** 比较离散的时候，使用记忆化递归更好。比如马走棋盘。

那什么时候不用记忆化递归呢？答案是其他情况都不用。因为普通的 dp table 有一个重要的功能，这个功能记忆化递归是无法代替的，那就是**滚动数组优化**。如果你需要对空间进行优化，那一定要用 dp table。

热身开始

理论知识已经差不多了，我们拿一道题来试试手。

我们以一个非常经典的背包问题来练一下手。

题目：322. 零钱兑换

```
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

你可以认为每种硬币的数量是无限的。



示例 1:

输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

这道题的参数有两个，一个是 coins，一个是 amount。

我们可以定义状态为 f(i, j) 表示用 coins 的前 i 项找 j 元需要的最少硬币数。那么答案就是 f(len(coins) - 1, amount)。

由组合原理，coins 的所有选择状态是 2^n。状态总数就是 i 和 j 的取值的笛卡尔积，也就是 2^len(coins) * (amount + 1)。

> 减 1 是因为存在 0 元的情况。

明确了这些，我们需要考虑的就是状态如何转移，也就是如何从寻常转移到 f(len(coins) - 1, amount)。

如何确定状态转移方程？我们需要:

- 聚焦最优子结构
- 做选择，在选择中取最优解（如果是计数 dp 则进行计数）

对于这道题来说，我们的选择有两种：

- 选择 coins[i]
- 不选择 coins[i]

这无疑是完备的。只不过仅仅是对 coins 中的每一项进行**选择与不选择**，这样的状态数就已经是 2^n 了，其中 n 为 coins 长度。

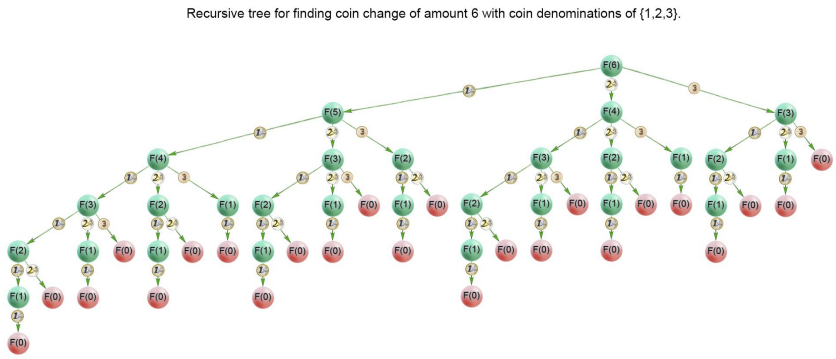
如果仅仅是这样枚举肯定会超时，因为状态数已经是指数级别了。

而这道题的核心在于 coins[i] 选择与否其实没有那么重要，**重要的其实是选择的 coins 一共有多少钱**。

因此我们可以定义 f(i, j) 表示选择了 coins 的前 i 项（怎么选的不关心），且组成 j 元需要的最少硬币数。

举个例子来说，比如 coins = [1,2,3] 。那么选择 [1,2] 和 选择 [3] 虽然是不一样的状态，但是我们压根不关心。因为这两者没有区别，我们还是谁对结果贡献大就 pick 谁。

以 coins = [1,2,3], amount = 6 来说，我们可以画出如下的递归树。



（图片来自https://leetcode.com/problems/coin-change/solution/）

因此转移方程就是 `min(dp[i][j], dp[i-1][j - coins[j]] + 1)`，含义就是： min(不选择 coins[j], 选择 coins[j]) 所需最少的硬币数。

用公式表示就是：

$$dp[i] = \begin{cases} min(dp[i][j], dp[i-1][j-coins[j]] + 1) & j >= coins[j] \\ amount + 1 & j < coins[j] \end{cases}$$

> amount 表示无解。因为硬币的面额都是正整数，不可能存在一种需要 amount + 1 枚硬币的方案。

代码

记忆化递归：

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        @lru_cache(None)
        def dfs(amount):
            if amount < 0: return float('inf')
            if amount == 0: return 0
            ans = float('inf')
            for coin in coins:
                ans = min(ans, 1 + dfs(amount - coin))
            return ans
        ans = dfs(amount)
        return -1 if ans == float('inf') else ans
```

二维 dp:

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        if amount < 0:
            return - 1
        dp = [[amount + 1 for _ in range(len(coins) + 1)]
              for _ in range(amount + 1)]
        # 初始化第一行为0，其他为最大值（也就是amount + 1）

        for j in range(len(coins) + 1):
            dp[0][j] = 0

        for i in range(1, amount + 1):
            for j in range(1, len(coins) + 1):
                if i - coins[j - 1] >= 0:
                    dp[i][j] = min(
                        dp[i][j - 1], dp[i - coins[j - 1]]
                else:
                    dp[i][j] = dp[i][j - 1]

        return -1 if dp[-1][-1] == amount + 1 else dp[-1][-
```

dp[i][j] 依赖于 `dp[i][j - 1]` 和 `dp[i - coins[j - 1]][j] + 1)` 这是一个优化的信号，我们可以将其优化到一维。

一维 dp（滚动数组优化）：

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1

        return -1 if dp[-1] == amount + 1 else dp[-1]
```

推荐练习题目

最后推荐几道题目给大家，建议大家分别使用记忆化递归和动态规划来解决。如果使用动态规划，则尽可能使用滚动数组优化空间。

- 0091.decode-ways

- 0139.word-break
- 0198.house-robber
- 0309.best-time-to-buy-and-sell-stock-with-cooldown
- 0322.coin-change
- 0416.partition-equal-subset-sum
- 0518.coin-change-2

总结

本篇文章总结了算法中比较常用的两个方法 - 递归和动态规划。递归的话可以拿树的题目练手，动态规划的话则将我上面推荐的刷完，再考虑去刷力扣的动态规划标签即可。

大家前期学习动态规划的时候，可以先尝试使用记忆化递归解决。然后将其改造为动态规划，这样多练习几次就会有感觉。之后大家可以练习一下滚动数组，这个技巧很有用，并且相对来说比较简单。

动态规划的核心在于定义状态，定义好了状态其他都是水到渠成。

动态规划的难点在于**枚举所有状态（不重不漏）** 和 **寻找状态转移方程**。

参考

- oi-wiki - dp 这个资料推荐大家学习，非常全面。只不过更适合有一定基础的人，大家可以配合本讲义食用哦。

另外，大家可以去 LeetCode 探索中的 递归 I 中进行互动式学习。

二叉树的遍历算法

概述

二叉树作为一个基础的数据结构，遍历算法作为一个基础的算法，两者结合当然是经典的组合了。很多题目都会有 ta 的身影，有直接问二叉树的遍历的，有间接问的。比如要你找到树中满足条件的节点，就是间接考察树的遍历，因为你要找到树中满足条件的点，就需要进行遍历。

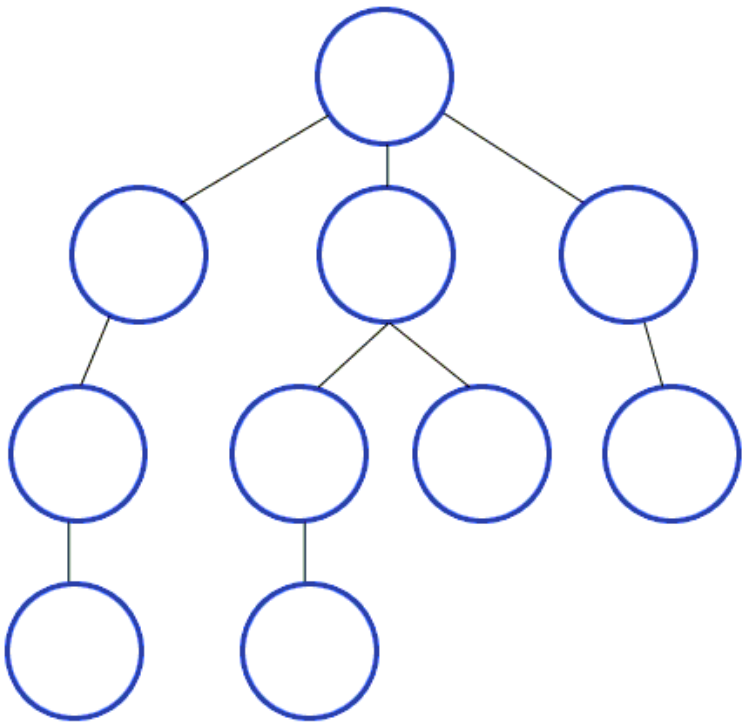
> 你如果掌握了二叉树的遍历，那么也许其他复杂的树对于你来说也并不遥远了

二叉数的遍历主要有前中后遍历和层次遍历。 前中后属于 DFS，层次遍历则可以使用 BFS 或者 DFS 来实现。只不过使用 BFS 来实现层次遍历会容易些，因为层次遍历就是 BFS 的副产物啊，你可以将层次遍历看成没有提前终止的 BFS

DFS 和 BFS 都有着自己的应用，比如 leetcode 301 号问题和 609 号问题。

DFS 都可以使用栈来简化操作，并且其实树本身是一种递归的数据结构，因此递归和栈对于 DFS 来说是两个关键点。

DFS 图解：



(图片来自 https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/depth-first-search)

BFS 的关键点在于如何记录每一层次是否遍历完成， 我们可以用一个标识位来表式当前层的结束。

对于前中后序遍历来说。首先不管是前中还是后序遍历，变的只是根节点的位置， 左右节点的顺序永远是先左后右。 比如前序遍历就是根在前面，即根左右。中序就是根在中间，即左根右。后序就是根在后面，即左右根。

下面我们依次讲解：

前序遍历

相关问题144.binary-tree-preorder-traversal

前序遍历的顺序是 `根-左-右`

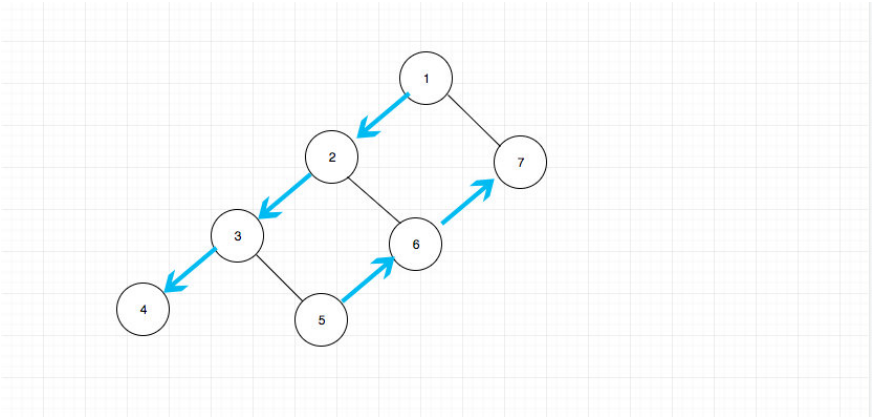
思路是：

1. 先将根结点入栈
2. 出栈一个元素，将右节点和左节点依次入栈
3. 重复 2 的步骤

总结： 典型的递归数据结构，典型的用栈来简化操作的算法。

其实从宏观上表现为：`自顶向下依次访问左侧链，然后自底向上依次访问右侧链`，如果从这个角度出发去写的话，算法就不一样了。从上向下我们可以直接递归访问即可，从下向上我们只需要借助栈也可以轻易做到。

整个过程大概是这样：



这种思路有一个好处就是可以`统一三种遍历的思路`. 这个很重要，如果不了解的朋友，希望能够记住这一点。

中序遍历

相关问题94.binary-tree-inorder-traversal

中序遍历的顺序是`左-根-右`，根节点不是先输出，这就有一点点复杂了。

1. 根节点入栈
2. 判断有没有左节点，如果有，则入栈，直到叶子节点

> 此时栈中保存的就是所有的左节点和根节点。

1. 出栈，判断有没有右节点，有则入栈，继续执行 2

值得注意的是，中序遍历一个二叉查找树（BST）的结果是一个有序数组，利用这个性质有些题目可以得到简化， 比如230.kth-smallest-element-in-a-bst， 以及98.validate-binary-search-tree

后序遍历

相关问题145.binary-tree-postorder-traversal

后序遍历的顺序是`左-右-根`

这个就有点难度了，要不也不会是 leetcode 困难的 难度啊。

其实这个也是属于根节点先不输出，并且根节点是最后输出。 这里可以采用一种讨巧的做法， 就是记录当前节点状态，如果：

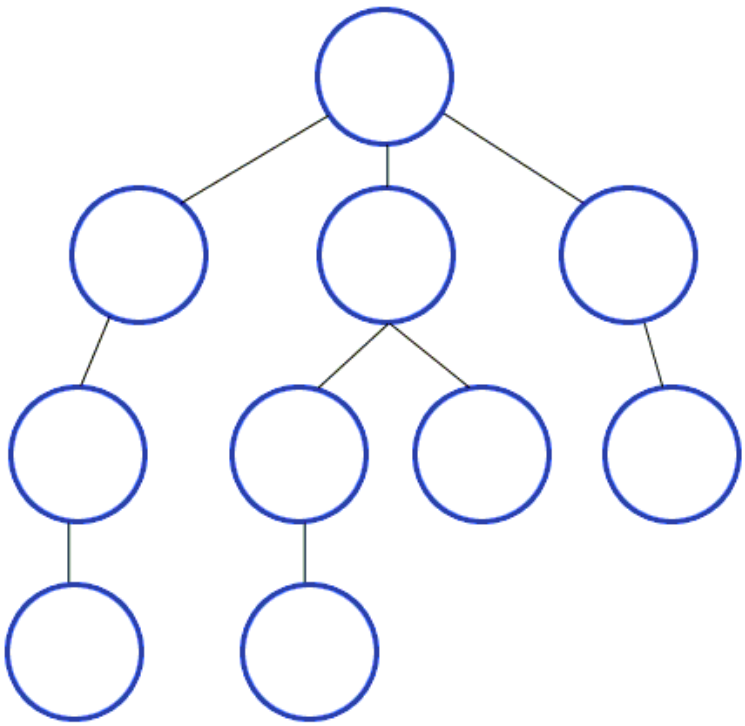
1. 当前节点是叶子节点或者
2. 当前节点的左右子树都已经遍历过了，那么就可以出栈了。

对于 `1. 当前节点是叶子节点` ，这个比较好判断，只要判断 left 和 rigt 是否同时为 null 就好。

对于 `2. 当前节点的左右子树都已经遍历过了` ， 只需要用一个变量记录即可。最坏的情况，我们记录每一个节点的访问状况就好了，空间复杂度 O(n) 但是仔细想一下，我们使用了栈的结构，从叶子节点开始输出，我们记录一个当前出栈的元素就好了，空间复杂度 O(1)， 具体请查看上方链接。

层次遍历

层次遍历的关键点在于如何记录每一层次是否遍历完成， 我们可以用一个标识位来表式当前层的结束。



(图片来自 https://github.com/trekhleb/javascript-algorithms/tree/master/src/algorithms/tree/breadth-first-search)

具体做法：

1. 根节点入队列， 并入队列一个特殊的标识位，此处是 null
2. 出队列
3. 判断是不是 null， 如果是则代表本层已经结束。我们再次判断是否当前队列为空，如果不为空继续入队一个 null，否则说明遍历已经完成，我们什么都不不用做
4. 如果不为 null，说明这一层还没完，则将其左右子树依次入队列。

相关问题：

- 102.binary-tree-level-order-traversal
- 117. 填充每个节点的下一个右侧节点指针 II

双色标记法

我们知道垃圾回收算法中，有一种算法叫三色标记法。 即：

- 用白色表示尚未访问
- 灰色表示尚未完全访问子节点
- 黑色表示子节点全部访问

那么我们可以模仿其思想，使用双色标记法来统一三种遍历。

其核心思想如下：

- 使用颜色标记节点的状态，新节点为白色，已访问的节点为灰色。
- 如果遇到的节点为白色，则将其标记为灰色，然后将其右子节点、自身、左子节点依次入栈。
- 如果遇到的节点为灰色，则将节点的值输出。

使用这种方法实现的中序遍历如下：

```
class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]
        WHITE, GRAY = 0, 1
        res = []
        stack = [(WHITE, root)]
        while stack:
            color, node = stack.pop()
            if node is None: continue
            if color == WHITE:
                stack.append((WHITE, node.right))
                stack.append((GRAY, node))
                stack.append((WHITE, node.left))
            else:
                res.append(node.val)
        return res
```

可以看出，实现上 WHITE 就表示的是递归中的第一次进入过程，Gray 则表示递归中的从叶子节点返回的过程。 因此这种迭代的写法更接近递归写法的本质。

如要实现前序、后序遍历，只需要调整左右子节点的入栈顺序即可。可以看出使用三色标记法， 其写法类似递归的形式，因此便于记忆和书写，缺点是使用了额外的内存空间。不过这个额外的空间是线性的，影响倒是不大。

> 虽然递归也是额外的线性时间，但是递归的栈开销还是比一个 0，1 变量开销大的。换句话说就是空间复杂度的常数项是不同的，这在一些情况下的差异还是蛮明显的。

划重点：双色迭代法是一种可以用迭代模拟递归的写法，其写法和递归非常相似，要比普通迭代简单地多。

Morris 遍历

我们可以使用一种叫做 Morris 遍历的方法，既不使用递归也不借助于栈。从而在 $O(1)$ 空间完成这个过程。

```
def MorrisTraversal(root):
    curr = root

    while curr:
        # If left child is null, print the
        # current node data. And, update
        # the current pointer to right child.
        if curr.left is None:
            print(curr.data, end= " ")
            curr = curr.right

        else:
            # Find the inorder predecessor
            prev = curr.left

            while prev.right is not None and prev.right is
                prev = prev.right

            # If the right child of inorder
            # predecessor already points to
            # the current node, update the
            # current with it's right child
            if prev.right is curr:
                prev.right = None
                curr = curr.right

            # else If right child doesn't point
            # to the current node, then print this
            # node's data and update the right child
            # pointer with the current node and update
            # the current with it's left child
            else:
                print (curr.data, end=" ")
                prev.right = curr
                curr = curr.left
```

参考： what-is-morris-traversal

划重点：Morris 是一种可以在 $O(1)$ 空间遍历二叉树的算法。*

相关题目

- lowest-common-ancestor-of-a-binary-tree
- binary-tree-level-order-traversal
- binary-tree-zigzag-level-order-traversal
- validate-binary-search-tree

- maximum-depth-of-binary-tree
- balanced-binary-tree
- binary-tree-level-order-traversal-ii
- binary-tree-maximum-path-sum
- insert-into-a-binary-search-tree

游程编码和哈夫曼编码

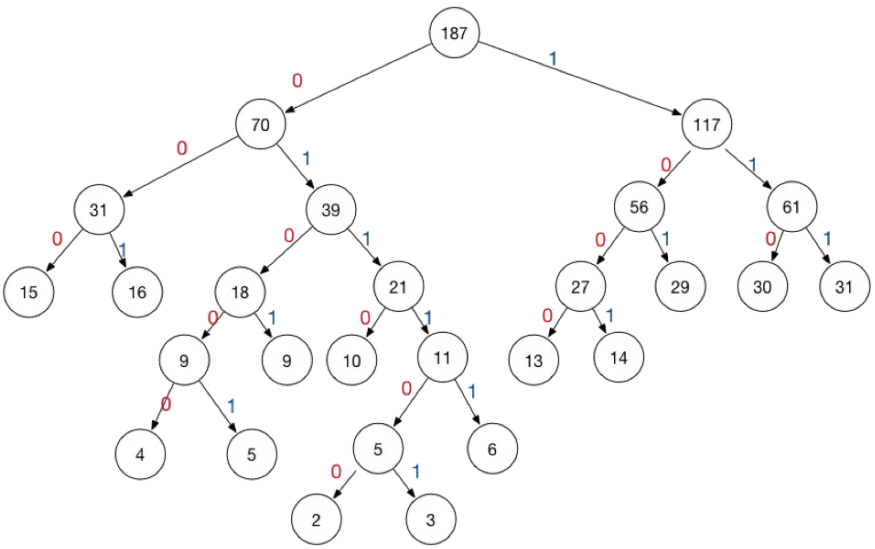
Huffman encode(哈夫曼编码)

Huffman 编码的基本思想就是用短的编码表示出现频率高的字符，用长的编码来表示出现频率低的字符，这使得编码之后的字符串的平均长度、长度的期望值降低，从而实现压缩的目的。 因此 Huffman 编码被广泛地应用于无损压缩领域。 可以看出， huffman 编码是**一种可变编码**，而不是固定长度编码。

Huffman 编码的过程包含两个主要部分：

- 根据输入字符构建 Huffman 树
- 遍历 Huffman 树，并将树的节点分配给字符

上面提到了他的基本原理就是 `用短的编码表示出现频率高的字符，用长的编码来表示出现频率低的字符` ，因此首先要做的就是统计字符的出现频率，然后根据统计的频率来构建 Huffman 树（又叫最优二叉树）。



如图，**huffman 树以一颗二叉树**。 其中节点的左子节点路径用 0 表示，右子节点用 1 表示，节点的值表示的是其权重，权重越大深度越小。深度表示的其实就是编码的长度。通常我们使用字符出现的频率作为权重。真正执行编码的时候，类似字典树，节点不用来编码，节点的路径用来编码.

> 如果计算机使用三进制，而不是二进制，那么 huffman 树就应该是一个三叉树。

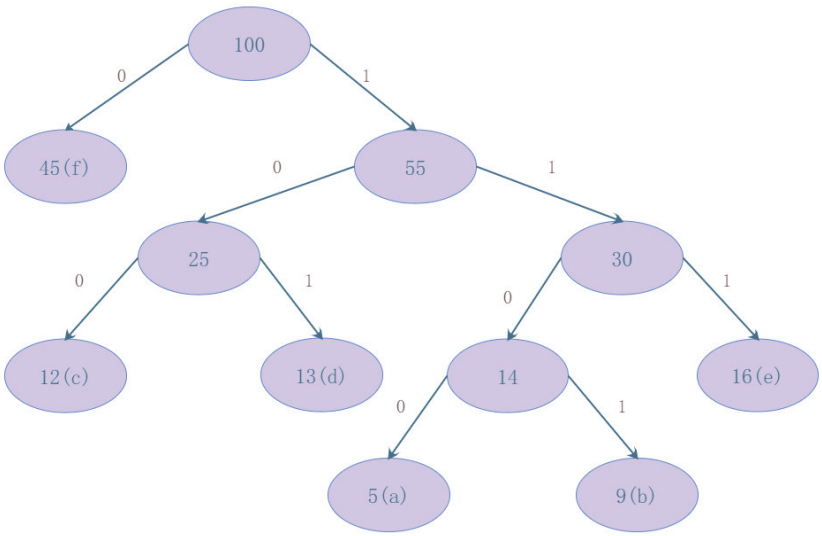
实例

比如我们对一个字符串的频率统计的结果如下：

character	frequency
a	5
b	9
c	12
d	13
e	16
f	45

- 将每个元素构造成一个节点，即只有一个元素的树。并构建一个最小堆，包含所有的节点，该算法用了最小堆来作为优先队列。
- `选取两个权值最小的节点`，并添加一个权值为 5+9=14 的节点，作为他们的父节点。并 `更新最小堆`，现在最小堆包含 5 个节点，其中 4 个树还是原来的节点，权值为 5 和 9 的节点合并为一个。

结果是这样的：



character	frequency	encoding
a	5	1100
b	9	1101
c	12	100
d	13	101
e	16	111
f	45	0

run-length encode(游程编码)

游程编码是一种比较简单的压缩算法，其基本思想是将重复且连续出现多次的字符使用（连续出现次数，某个字符）来描述。

比如一个字符串：

```
AAAAABBBBCCC
```

使用游程编码可以将其描述为：

```
5A4B3C
```

5A 表示这个地方有 5 个连续的 A，同理 4B 表示有 4 个连续的 B，3C 表示有 3 个连续的 C，其它情况以此类推。

但是实际上情况可能会非常复杂，我们可以对单个字符进行编码，也可以对多个字符进行编码。 因此如何提取子序列就是一个问题。这并没有看的那么简单。还是以上面的例子来说，我们有也可以把 `AAAAABBBBCCC` 整体看成一个子序列，这样编码的长度就有所编码。究竟使用哪种方法，取决于压缩的时间和压缩的比例等。 更复杂的情况还有很多，这里不做扩展。

对文件进行压缩比较适合的情况是文件内的二进制有大量的连续重复，一个经典的例子就是具有大面积色块的 BMP 图像，BMP 因为没有压缩，所以看到的是什么样子存储的时候二进制就是什么样子

> 这也是我们图片倾向于纯色的时候，压缩会有很好的效果

思考一个问题， 如果我们在 CDN 上存储两个图片，这两个图片几乎完全一样，我们是否可以进行优化呢？这虽然是 CDN 厂商更应该关心的问题，但是这个问题对我们影响依然很大，值得思考。

总结

游程编码和 Huffman 都是无损压缩算法，即解压缩过程不会损失原数据任何内容。 实际情况，我们先用游程编码一遍，然后再用 Huffman 再次编码一次。几乎所有的无损压缩格式都用到了它们，比如 PNG，GIF，PDF，ZIP 等。

对于有损压缩，通常是去除了人类无法识别的颜色，听力频率范围等。也就是说损失了原来的数据。 但由于人类无法识别这部分信息，因此很多情况下都是值得的。这种删除了人类无法感知内容的编码，我们称之为“感知编码”（也许是一个自创的新名词），比如 JPEG，MP3 等。关于有损压缩不是本文的讨论范围，感兴趣的可以搜素相关资料。

实际上，视频压缩的原理也是类似，只不过视频压缩会用到一些额外的算法，比如“时间冗余”，即仅存储变化的部分，对于不变的部分，存储一次就够了。

相关题目

900.rle-iterator

布隆过滤器

场景

假设你现在要处理这样一个问题，你有一个网站并且拥有 `很多` 访客，每当有用户访问时，你想知道这个 ip 是不是第一次访问你的网站。

hashtable 可以么

一个显而易见的答案是将所有的 IP 用 hashtable 存起来，每次访问都去 hashtable 中取，然后判断即可。但是题目说了网站有 `很多` 访客，假如有 10 亿个用户访问过，假设 IP 是 IPV4， 那么每个 IP 的长度是 4 byte，那么你一共需要 4 * 1000000000 = 4000000000Bytes = 4G 。

如果是判断 URL 黑名单，由于每个 URL 会更长（可能远大于上面 IPV4 地址的 4 byte），那么需要的空间可能会远远大于你的期望。

bit

另一个稍微难想到的解法是 bit， 我们知道 bit 有 0 和 1 两种状态，那么用来表示**存在**与**不存在**再合适不过了。

假如有 10 亿个 IP，就可以用 10 亿个 bit 来存储，那么你一共需要 1 * 1000000000 = (4000000000 / 8) Bytes = 128M, 变为原来的 1/32, 如果是存储 URL 这种更长的字符串，效率会更高。 问题是，我们怎么把 IPV4 和 bit 的位置关联上呢?

比如 `192.168.1.1` 应该是用第几位表示， `10.18.1.1` 应该是用第几位表示呢？ 答案是使用哈希函数。

基于这种想法，我们只需要两个操作，set(ip) 和 has(ip)，以及一个内置函数 hash(ip) 用于将 IP 映射到 bit 表。

这样做有两个非常致命的缺点：

1. 当样本分布极度不均匀的时候，会造成很大空间上的浪费

> 我们可以通过优化散列函数来解决

1. 当元素不是整型（比如 URL）的时候，BitSet 就不适用了

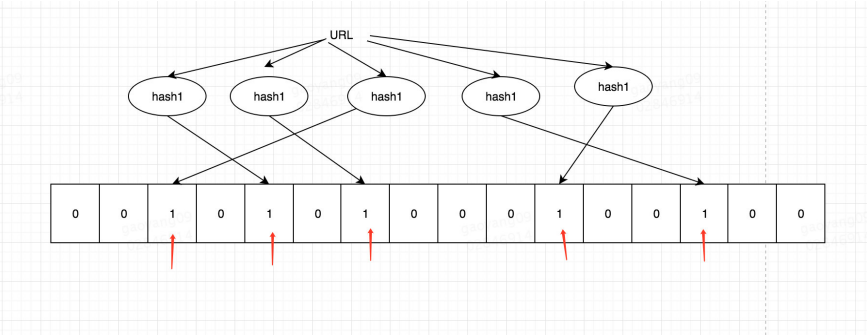
> 我们还是可以使用散列函数来解决， 甚至可以多 hash 几次

布隆过滤器

布隆过滤器其实就是 `bit + 多个散列函数` 。k 次 hash(ip) 会生成多个索引，并将其 k 个索引位置的二进制置为 1。

- 如果经过 k 个索引位置的值都为 1，那么认为其**可能存在**(因为有冲突的可能)。
- 如果有一个不为 1，那么**一定不存在**（一个值经过散列函数得到的值一定是唯一的），这也是布隆过滤器的一个重要特点。

也就是说布隆过滤器回答了：**可能存在** 和 **一定不存在** 的问题。



从上图可以看出， 布隆过滤器本质上是由**一个很长的二进制向量**和**多个哈希函数**组成。

由于没有 hashtable 的 100% 可靠性，因此这本质上是一种**可靠性换取空间的做法**。除了可靠性，布隆过滤器删除起来也比较麻烦。

误报

上面提到了布隆过滤器回答了：**可能存在** 和 **一定不存在** 的问题。 因此当回答是**可能存在**的时候你该怎么做？一般而言， 为了宁可错杀一千，也不放过一个，我们认为他存在。 这个时候就产生了**误报**。

误报率和二进制向量的长度成反比。

布隆过滤器的应用

1. 网络爬虫

判断某个 URL 是否已经被爬取过

1. K-V 数据库 判断某个 key 是否存在

比如 Hbase 的每个 Region 中都包含一个 BloomFilter，用于在查询时快速判断某个 key 在该 region 中是否存在。

1. 钓鱼网站识别

浏览器有时候会警告用户，访问的网站很可能是钓鱼网站，用的就是这种技术

> 从这个算法大家可以对 tradeoff(取舍) 有更入的理解。

1. 恶意网站识别

总之， 如果你需要判断**一个项目是否在一个集合中出现过，并且需要100% 确定没有出现过，或者可能出现过**，就可以考虑使用布隆过滤器。

代码

```
public   class  MyBloomFilter {
     private static final int DEFAULT_SIZE =  2 << 31 ;
     private static final int[] seeds = new int [] {3,5,7,1
     private  BitSet  bits = new BitSet(DEFAULT_SIZE);
     private  SimpleHash[] func  = new  SimpleHash[seeds.le

     public   static   void  main(String[] args) {
        //使用
        String value = "www.xxxxx.com" ;
        MyBloomFilter filter = new MyBloomFilter();
        System.out.println(filter.contains(value));
        filter.add(value);
        System.out.println(filter.contains(value));
    }
    //构造函数
     public  MyBloomFilter() {
         for  ( int  i  =   0 ; i  <  seeds.length; i ++ )
            func[i]  =   new  SimpleHash(DEFAULT_SIZE, seed
        }
    }
     //添加网站
     public   void  add(String value) {
         for  (SimpleHash f : func) {
            bits.set(f.hash(value),  true );
        }
    }
     //判断可疑网站是否存在
     public   boolean  contains(String value) {
         if  (value  ==   null ) {
             return   false ;
        }
         boolean  ret  =   true ;
         for  (SimpleHash f : func) {
            //核心就是通过“与”的操作
            ret  =  ret  &&  bits.get(f.hash(value));
        }
         return  ret;
    }
}
```

总结

布隆过滤器回答了：**可能存在** 和 **一定不存在** 的问题。本质是一种空间和准确率的一个取舍。实际使用可能会有误报的情况， 如果你的业务可以接受误报，那么使用布隆过滤器进行优化是一个不错的选择。

Trie（来自公众号力扣加加的活动《91天学算法》的讲义）

简介

字典树也叫前缀树、Trie。它本身就是一个树型结构，也就是一颗多叉树，学过树的朋友应该非常容易理解，它的核心操作是插入，查找。删除很少使用，因此这个讲义不包含删除操作。

截止目前（2020-02-04） 前缀树（字典树） 在 LeetCode 一共有 17 道题目。其中 2 道简单，8 个中等，7 个困难。

前缀树的特点

简单来说， 前缀树就是一个树。前缀树一般是将一系列的单词记录到树上， 如果这些单词没有公共前缀，则和直接用数组存没有任何区别。而如果有公共前缀， 则公共前缀仅会被存储一次。可以想象，如果一系列单词的公共前缀很多， 则会有效减少空间消耗。

而前缀树的意义实际上是空间换时间，这和哈希表，动态规划等的初衷是一样的。

其原理也很简单，正如我前面所言，其公共前缀仅会被存储一次，因此如果我想在一堆单词中找某个单词或者某个前缀是否出现，我无需进行完整遍历，而是遍历前缀树即可。本质上，使用前缀树和不使用前缀树减少的时间就是公共前缀的数目。也就是说，一堆单词没有公共前缀，使用前缀树没有任何意义。

知道了前缀树的特点，接下来我们自己实现一个前缀树。关于实现可以参考 0208.implement-trie-prefix-tree

应用场景及分析

正如上面所说，前缀树的核心思想是用空间换时间，利用字符串的公共前缀来降低查询的时间开销。

比如给你一个字符串 query，问你这个**字符串**是否在**字符串集合**中出现过，这样我们就可以将字符串集合建树，建好之后来匹配 query 是否出现，那有的朋友肯定会问，之前讲过的 hashmap 岂不是更好?

我们想一下用百度搜索时候，打个“一语”，搜索栏中会给出“一语道破”，“一语成谶(四声的 chen)”等推荐文本，这种叫模糊匹配，也就是给出一个模糊的 query，希望给出一个相关推荐列表，很明显，hashmap 并不容易做到模糊匹配，而 Trie 可以实现基于前缀的模糊搜索。

> 注意这里的模糊搜索也仅仅是基于前缀的。比如还是上面的例子，搜索“道破”就不会匹配到“一语道破”，而只能匹配“道破 xx”

因此，这里我的理解是：上述精确查找只是模糊查找一个特例，模糊查找 hashmap 显然做不到，并且如果在精确查找问题中，hashmap 出现过多冲突，效率还不一定比 Trie 高，有兴趣的朋友可以做一下测试，看看哪个快。

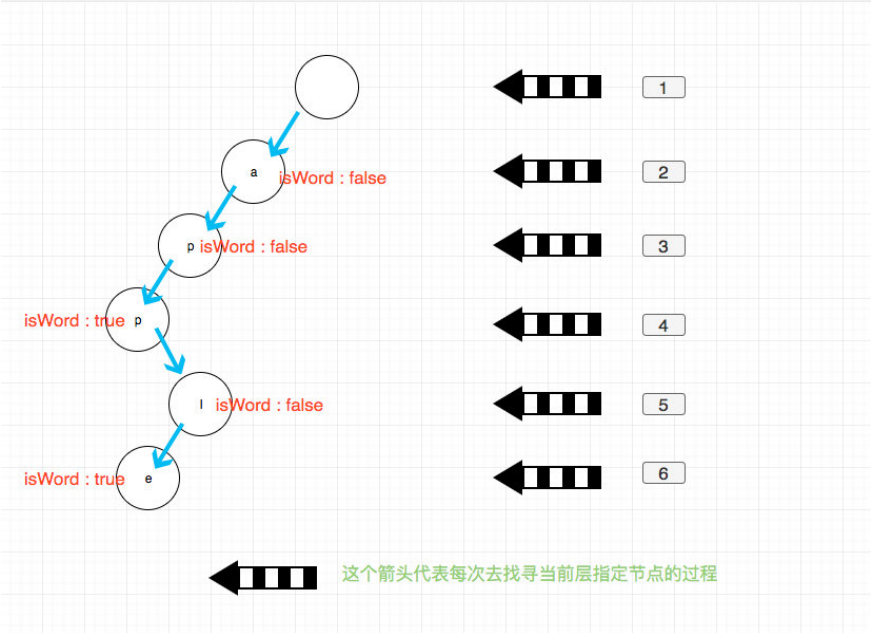
再比如给你一个长句和一堆敏感词，找出长句中所有敏感词出现的所有位置（想下，有时候我们口吐芬芳，结果发送出去却变成了****，懂了吧）

> 小提示：实际上 AC 自动机就利用了 trie 的性质来实现敏感词的匹配，性能非常好。以至于很多编辑器都是用的 AC 自动机的算法。

还有些其他场景，这里不过多讨论，有兴趣的可以 google 一下。

基本概念

一个前缀树大概是这个样子：



如图每一个节点存储一个字符，然后外加一个控制信息表示是否是单词结尾，实际使用过程可能会有细微差别，不过变化不大。

接下来，我们看下 Trie 里面的概念 - 节点。

- 根结点无实际意义
- 每一个节点代表一个字符
- 每个节点中的数据结构可以自定义，如 isWord(是否是单词)，count(该前缀出现的次数)等，需实际问题实际分析需要什么。

API

自己实现前缀树，首先要知道它的 api 有哪些，以及具体功能是什么。

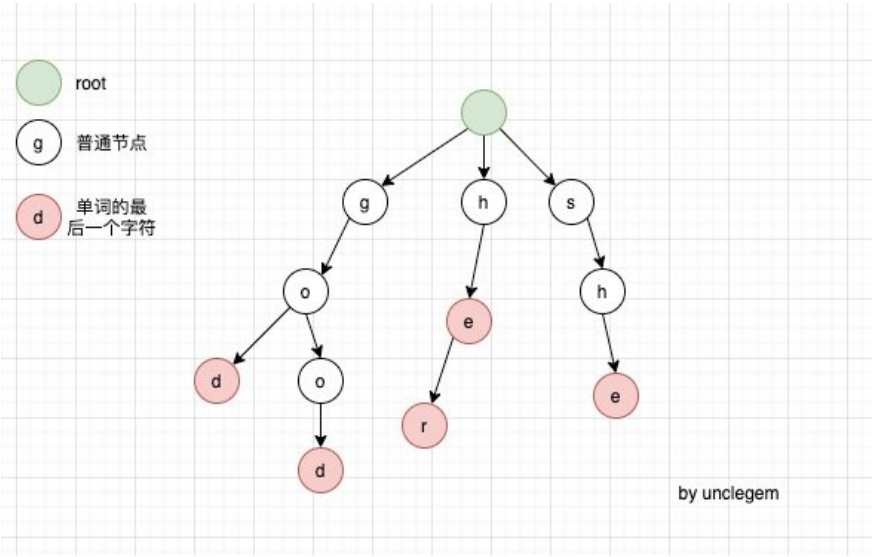
前缀树的 api 主要有以下几个：

- `insert(word)`：插入一个单词
- `search(word)`：查找一个单词是否存在
- `startWith(word)`：查找是否存在以 word 为前缀的单词

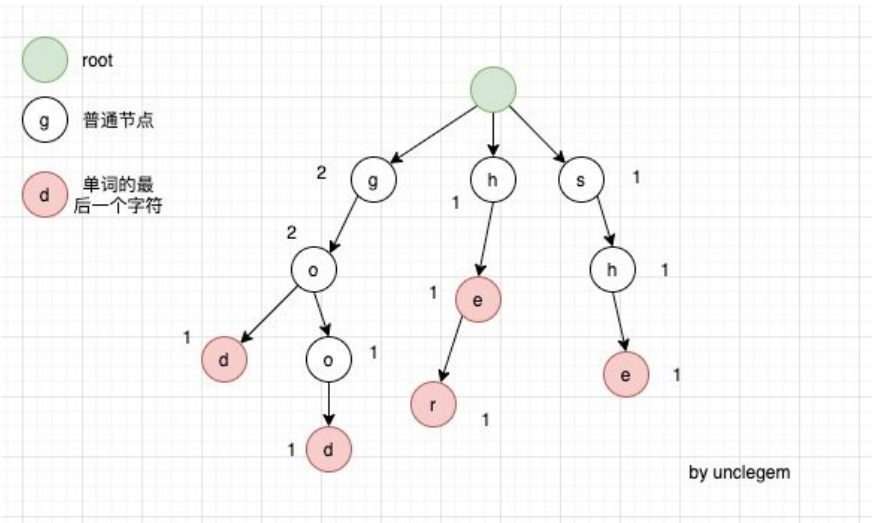
其中 startWith 是前缀树最核心的用法，其名称前缀树就从这里而来。大家可以先拿 208 题开始，熟悉一下前缀树，然后再尝试别的题目。

Trie 的插入

- 假定给出几个单词如[she,he,her,good,god]构造出一个 Trie 如下图：



- 也就是说从根结点出发到某一粉色节点所经过的字符组成的单词，在单词列表中出现过，当然我们也可以给树的每个节点加个 count 属性，代表根结点到该节点所构成的字符串前缀出现的次数



可以看出树的构造非常简单，插入新单词的时候就从根结点出发一个字符一个字符插入，有对应的字符节点就更新对应的属性，没有就创建一个！

Trie 的查询

查询更简单了，给定一个 Trie 和一个单词，和插入的过程类似，一个字符一个字符找

- 若中途有个字符没有对应节点 →Trie 不含该单词
- 若字符串遍历完了，都有对应节点，但最后一个字符对应的节点并不是粉色的，也就不是一个单词 →Trie 不含该单词

Trie 模版

了解了 Trie 的使用场景以及基本的 API， 那么最后就是用代码来实现了。

这里我提供了 Python 和 Java 两种语言的代码。

Java:

```
class Trie {

    TrieNode root;

    public Trie() {

        root = new TrieNode();
    }

    public void insert(String word) {

        TrieNode node = root;

        for (int i = 0; i < word.length(); i++) {

            if (node.children[word.charAt(i) - 'a'] == nul
                node.children[word.charAt(i) - 'a'] = new T

            node = node.children[word.charAt(i) - 'a'];
            node.preCount++;
        }

        node.count++;
    }

    public boolean search(String word) {

        TrieNode node = root;

        for (int i = 0; i < word.length(); i++) {

            if (node.children[word.charAt(i) - 'a'] == nul
                return false;

            node = node.children[word.charAt(i) - 'a'];
        }

        return node.count > 0;
    }

    public boolean startsWith(String prefix) {

        TrieNode node = root;

        for (int i = 0; i < prefix.length(); i++) {

            if (node.children[prefix.charAt(i) - 'a'] == nu
                return false;
```

```
            node = node.children[prefix.charAt(i) - 'a'];
        }

        return node.preCount > 0;
    }

    private class TrieNode {

        int count; //表示以该处节点构成的串的个数
        int preCount; //表示以该处节点构成的前缀的字串的个数
        TrieNode[] children;

        TrieNode() {

            children = new TrieNode[26];
            count = 0;
            preCount = 0;
        }
    }
}
```

Python:

```
class TrieNode:
    def __init__(self):
        self.count = 0
        self.preCount = 0
        self.children = {}

class Trie:

    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
            node.preCount += 1
        node.count += 1

    def search(self, word):
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.count > 0

    def startsWith(self, prefix):
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.preCount > 0
```

复杂度分析

- 插入和查询的时间复杂度自然是$O(len(key))$，key 是待插入(查找)的字串。
- 建树的最坏空间复杂度是$O(m^{n})$, m 是字符集中字符个数，n 是字符串长度。

题目推荐

以下是本专题的六道题目的题解，内容会持续更新，感谢你的关注～

- 0208.实现 Trie (前缀树)
- 0211.添加与搜索单词 - 数据结构设计
- 0212.单词搜索 II
- 0472.连接词
- 648. 单词替换
- 0820.单词的压缩编码
- 1032.字符流

总结

前缀树的核心思想是用空间换时间，利用字符串的公共前缀来降低查询的时间开销。因此如果题目中公共前缀比较多，就可以考虑使用前缀树来优化。

前缀树的基本操作就是插入和查询，其中查询可以完整查询，也可以前缀查询，其中基于前缀查询才是前缀树的灵魂，也是其名字的来源。

最后给大家提供了两种语言的前缀树模板，大家如果需要用，直接将其封装成标准 API 调用即可。

基于前缀树的题目变化通常不大， 使用模板就可以解决。如何知道该使用前缀树优化是一个难点，不过大家只要牢牢记一点即可，那就是**算法的复杂度瓶颈在字符串查找，并且字符串有很多公共前缀，就可以用前缀树优化**。

回溯

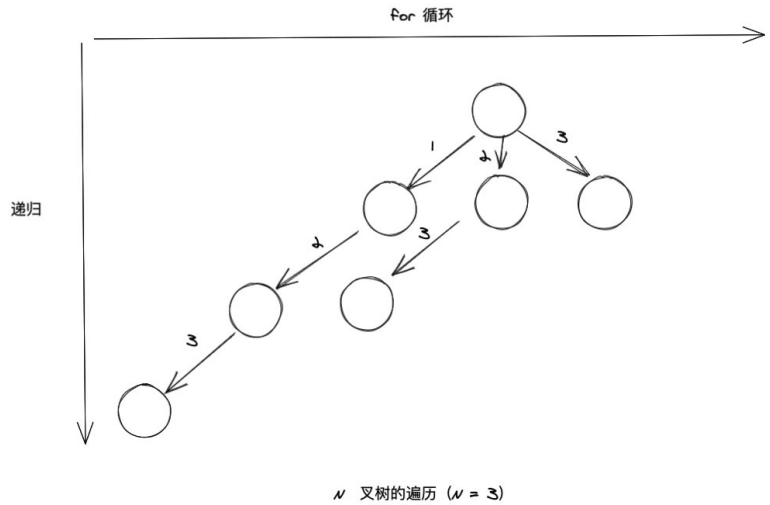
回溯是 DFS 中的一种技巧。回溯法采用 试错 的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将**取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案**。

通俗上讲，回溯是一种走不通就回头的算法。

回溯的本质是穷举所有可能，尽管有时候可以通过剪枝去除一些根本不可能是答案的分支， 但是从本质上讲，仍然是一种暴力枚举算法。

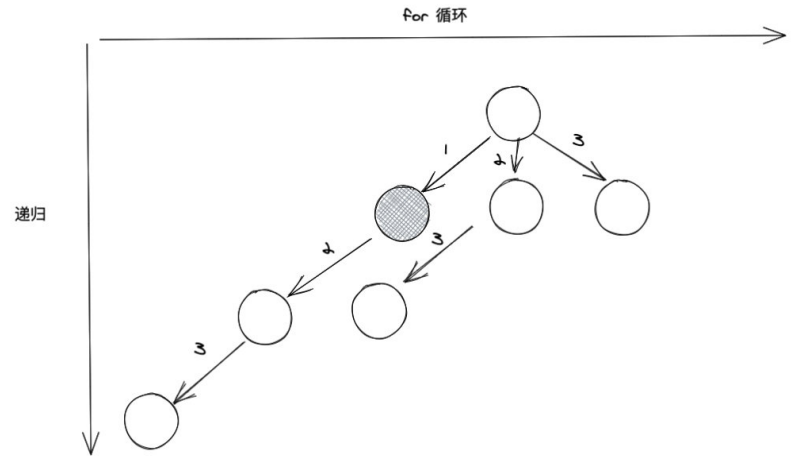
回溯法可以抽象为树形结构，并且是是一颗高度有限的树（N 叉树）。回溯法解决的都是在集合中查找子集，集合的大小就是树的叉树，递归的深度，构成树的高度。

以求数组 [1,2,3] 的子集为例：



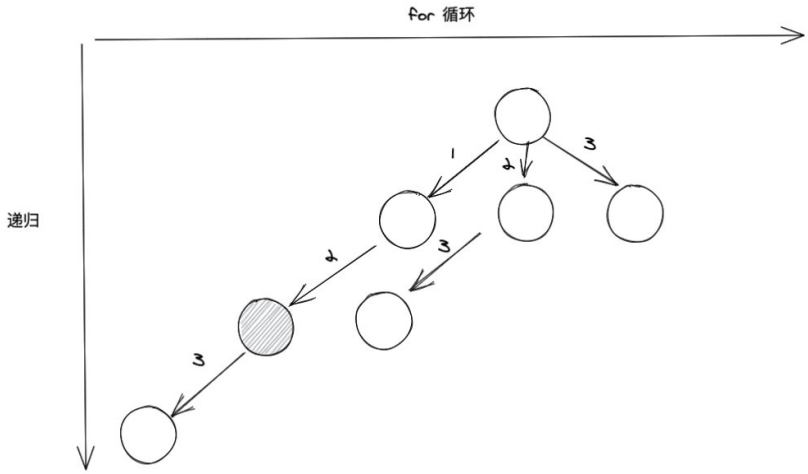
> for 循环用来枚举分割点，其实区间 dp 分割区间就是类似的做法

以上图来说， 我们会在每一个节点进行加入到结果集这一次操作。



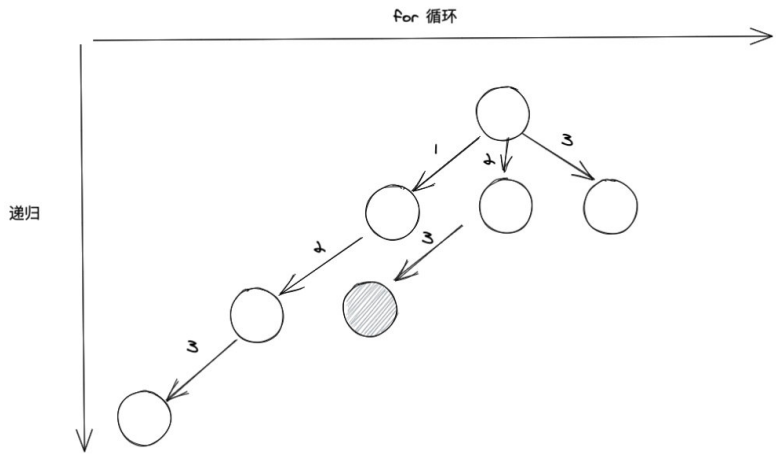
N 叉树的遍历（N = 3）

对于上面的灰色节点， 加入结果集就是 [1]。



N 叉树的遍历（N = 3）

这个加入结果集就是 [1,2]。



N 叉树的遍历（N = 3）

这个加入结果集就是 [2,3]，以此类推。一共有六个子集，分别是 [1], [1,2], [1,2,3], [2], [2,3] 和 [3]。

而对于全排列问题则会在叶子节点加入到结果集，不过这都是细节问题。掌握了思想之后， 大家再去学习细节就会事半功倍。

下面我们来看下具体代码怎么写。

算法流程

1. 构造空间树。
2. 进行遍历。
3. 如遇到边界条件，即不再向下搜索，转而搜索另一条链。
4. 达到目标条件，输出结果。

算法模板

伪代码：

```
const visited = {}
function dfs(i) {
    if (满足特定条件) {
        // 返回结果 or 退出搜索空间
    }

    visited[i] = true // 将当前状态标为已搜索
    dosomething(i) // 对i做一些操作
    for (根据i能到达的下个状态j) {
        if (!visited[j]) { // 如果状态j没有被搜索过
            dfs(j)
        }
    }
    undo(i) // 恢复i
}
```

剪枝

回溯题目的另外一个考点是剪枝， 通过恰当地剪枝，可以有效减少时间，比如我通过剪枝操作将**石子游戏 V**的时间从 900 多 ms 优化到了 500 多 ms。

剪枝在每道题的技巧都是不一样的， 不过一个简单的原则就是**避免根本不可能是答案的递归**。

举个例子: 842. 将数组拆分成斐波那契序列

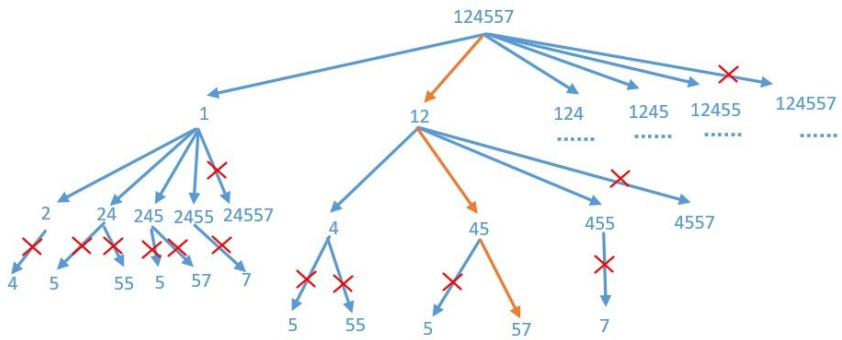
题目描述：

```
给定一个数字字符串 S，比如 S = "123456579"，我们可以将它分成斐波那契

形式上，斐波那契式序列是一个非负整数列表 F，且满足：

0 <= F[i] <= 2^31 - 1，（也就是说，每个整数都符合 32 位有符号整数
F.length >= 3；
对于所有的0 <= i < F.length - 2，都有 F[i] + F[i+1] = F[i+2]
另外，请注意，将字符串拆分成小块时，每个块的数字一定不要以零开头，除非

返回从 S 拆分出来的任意一组斐波那契式的序列块，如果不能拆分则返回 []。



示例 1：

输入："123456579"
输出：[123,456,579]
示例 2：

输入: "11235813"
输出: [1,1,2,3,5,8,13]
示例 3：

输入: "112358130"
输出: []
解释: 这项任务无法完成。
示例 4：

输入："0123"
输出：[]
解释：每个块的数字不能以零开头，因此 "01"，"2"，"3" 不是有效答案。
示例 5：

输入: "1101111"
输出: [110, 1, 111]
解释: 输出 [11,0,11,11] 也同样被接受。


提示：

1 <= S.length <= 200
字符串 S 中只含有数字。
```

还是直接套回溯模板即可解决。但是如果不进行合适地剪枝，很容易超时，这里我进行了四个剪枝操作，具体看代码。

```python
class Solution:
    def splitIntoFibonacci(self, S: str) -> List[int]:
        def backtrack(start, path):
            # 剪枝1
            if len(path) > 2 and path[-1] != path[-2] + pat
                return []
            if start >= len(S):
                if len(path) > 2:
                    return path
                return []

            cur = 0
            ans = []
            # 枚举分割点
            for i in range(start, len(S)):
                # 剪枝2
                if i > start and S[start] == '0':
                    return []
                cur = cur * 10 + int(S[i])
                # 剪枝3
                if cur > 2**31 - 1:
                    return []
                path.append(cur)
                ans = backtrack(i + 1, path)
                # 剪枝 4
                if len(ans) > 2:
                    return ans
                path.pop()
            return ans

        return backtrack(0, [])
```

剪枝过程用图表示就是这样的：



剪枝算法回溯的一大考点，大家一定套掌握。

笛卡尔积

一些回溯的题目，我们仍然也可以采用笛卡尔积的方式，将结果保存在返回值而不是路径中，这样就避免了回溯状态，并且由于结果在返回值中，因此可以使用记忆化递归， 进而优化为动态规划形式。

参考题目：

- 140. 单词拆分 II
- 401. 二进制手表
- 816. 模糊坐标

这类问题不同于子集和全排列，其组合是有规律的，我们可以使用笛卡尔积公式，将两个或更多子集联合起来。

经典题目

- 39. 组合总和
- 40. 组合总和 II
- 46. 全排列
- 47. 全排列 II
- 52. N 皇后 II
- 78. 子集
- 90. 子集 II
- 113. 路径总和 II
- 131. 分割回文串
- 1255. 得分最高的单词集合

总结

回溯的本质就是暴力枚举所有可能。要注意的是，由于回溯通常结果集都记录在回溯树的路径上，因此如果不进行撤销操作， 则可能在回溯后状态不正确导致结果有差异， 因此需要在递归到底部往上冒泡的时候进行撤销状态。

如果你每次递归的过程都拷贝了一份数据，那么就不需要撤销状态，相对地空间复杂度会有所增加。

滑动窗口（Sliding Window）

笔者最早接触滑动窗口是`滑动窗口协议`，滑动窗口协议（Sliding Window Protocol），属于 TCP 协议的一种应用，用于网络数据传输时的流量控制，以避免拥塞的发生。 发送方和接收方分别有一个窗口大小 w1 和 w2。窗口大小可能会根据网络流量的变化而有所不同，但是在更简单的实现中它们是固定的。窗口大小必须大于零才能进行任何操作。

我们算法中的滑动窗口也是类似，只不过包括的情况更加广泛。实际上上面的滑动窗口在某一个时刻就是固定窗口大小的滑动窗口，随着网络流量等因素改变窗口大小也会随着改变。接下来我们讲下算法中的滑动窗口。

介绍

滑动窗口是一种解决问题的思路和方法，通常用来解决一些连续问题。比如 LeetCode 的 209. 长度最小的子数组。更多滑动窗口题目见下方`题目列表`。

常见套路

滑动窗口主要用来处理连续问题。比如题目求解“连续子串 xxxx”，“连续子数组 xxxx”，就应该可以想到滑动窗口。能不能解决另说，但是这种敏感性还是要有的。

从类型上说主要有：

- 固定窗口大小
- 窗口大小不固定，求解最大的满足条件的窗口
- 窗口大小不固定，求解最小的满足条件的窗口（上面的 209 题就属于这种）

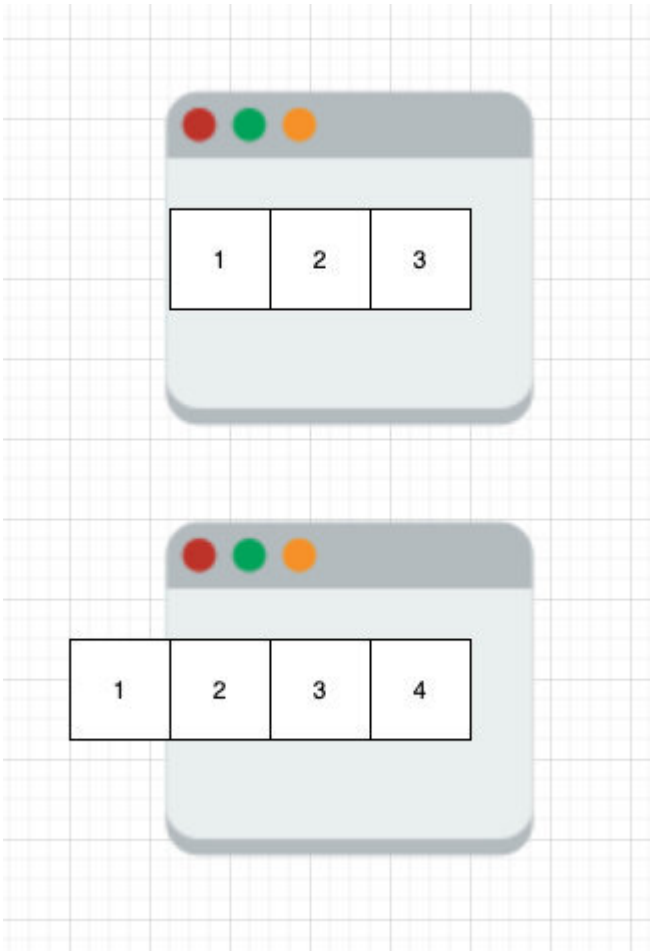
后面两种我们统称为`可变窗口`。当然不管是哪种类型基本的思路都是一样的，不一样的仅仅是代码细节。

固定窗口大小

对于固定窗口，我们只需要固定初始化左右指针 l 和 r，分别表示的窗口的左右顶点，并且保证：

1. l 初始化为 0
2. 初始化 r，使得 r - l + 1 等于窗口大小
3. 同时移动 l 和 r
4. 判断窗口内的连续元素是否满足题目限定的条件

- 4.1 如果满足，再判断是否需要更新最优解，如果需要则更新最优解
- 4.2 如果不满足，则继续。

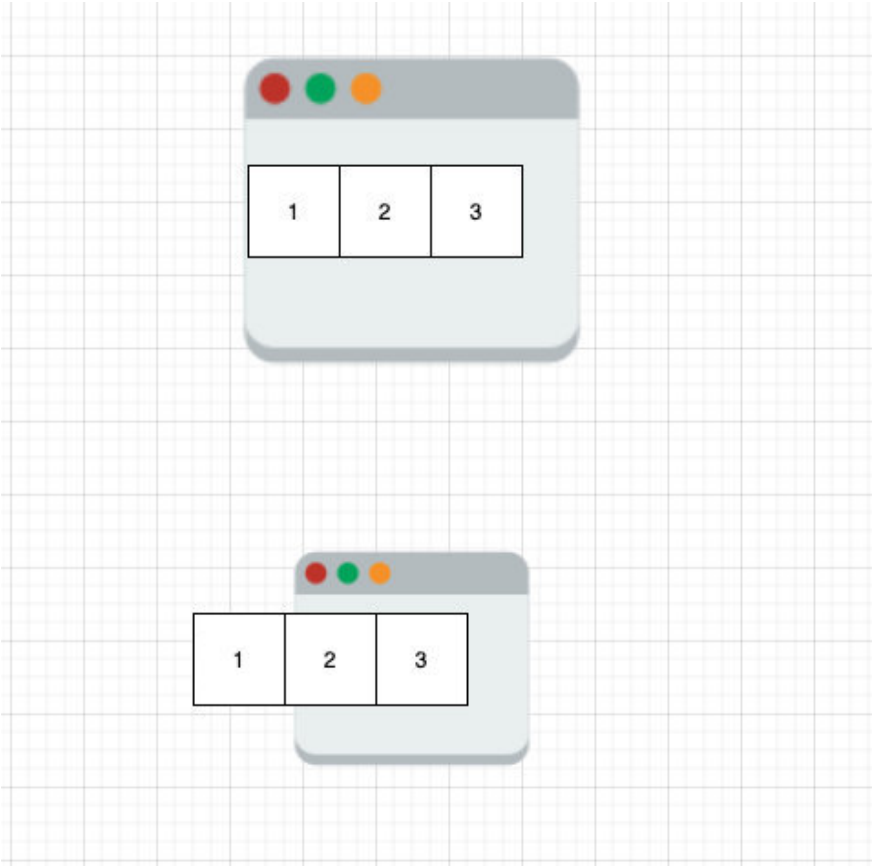


可变窗口大小

对于可变窗口，我们同样固定初始化左右指针 l 和 r，分别表示的窗口的左右顶点。后面有所不同，我们需要保证：

1. l 和 r 都初始化为 0
2. r 指针移动一步
3. 判断窗口内的连续元素是否满足题目限定的条件
    - 3.1 如果满足，再判断是否需要更新最优解，如果需要则更新最优解。并尝试通过移动 l 指针缩小窗口大小。循环执行 3.1
    - 3.2 如果不满足，则继续。

形象地来看的话，就是 r 指针不停向右移动，l 指针仅仅在窗口满足条件之后才会移动，起到窗口收缩的效果。



模板代码

伪代码

```
初始化慢指针 = 0
初始化 ans

for 快指针 in 可迭代集合
    更新窗口内信息
    while 窗口内不符合题意
        扩展或者收缩窗口
        慢指针移动
    更新答案
返回 ans
```

代码

以下是 209 题目的代码，使用 Python 编写，大家意会即可。

```
class Solution:
    def minSubArrayLen(self, s: int, nums: List[int]) -> in
        l = total = 0
        ans = len(nums) + 1
        for r in range(len(nums)):
            total += nums[r]
            while total >= s:
                ans = min(ans, r - l + 1)
                total -= nums[l]
                l += 1
        return  0 if ans == len(nums) + 1 else ans
```

题目列表（有题解）

以下题目有的信息比较直接，有的题目信息比较隐蔽，需要自己发掘

- 【Python，JavaScript】滑动窗口（3. 无重复字符的最长子串）
- 76. 最小覆盖子串
- 209. 长度最小的子数组
- 【Python】滑动窗口（438. 找到字符串中所有字母异位词）
- 【904. 水果成篮】（Python3）
- 【930. 和相同的二元子数组】（Java，Python）
- 【992. K 个不同整数的子数组】滑动窗口（Python）
- 978. 最长湍流子数组
- 【1004. 最大连续 1 的个数 III】滑动窗口（Python3）
- 【1234. 替换子串得到平衡字符串】[Java/C++/Python] Sliding Window
- 【1248. 统计「优美子数组」】滑动窗口（Python）
- 1658. 将 x 减到 0 的最小操作数

扩展阅读

- LeetCode Sliding Window Series Discussion

位运算

我这里总结了几道位运算的题目分享给大家，分别是 136 和 137， 260 和 645， 总共加起来四道题。 四道题全部都是位运算的套路，如果你想练习位运算的话，不要错过哦～～

前菜

开始之前我们先了解下异或，后面会用到。

1. 异或的性质

两个数字异或的结果 `a^b` 是将 a 和 b 的二进制每一位进行运算，得出的数字。 运算的逻辑是果同一位的数字相同则为 0，不同则为 1

1. 异或的规律

2. 任何数和本身异或则为 `0`

3. 任何数和 0 异或是 `本身`

4. 异或运算满足交换律，即：

`a ^ b ^ c = a ^ c ^ b`

OK，我们来看下这三道题吧。

136. 只出现一次的数字 1

题目大意是除了一个数字出现一次，其他都出现了两次，让我们找到出现一次的数。我们执行一次全员异或即可。

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        single_number = 0
        for num in nums:
            single_number ^= num
        return single_number
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

137. 只出现一次的数字 2

题目大意是除了一个数字出现一次，其他都出现了三次，让我们找到出现一次的数。 灵活运用位运算是本题的关键。

Python3:

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        res = 0
        for i in range(32):
            cnt = 0  # 记录当前 bit 有多少个1
            bit = 1 << i  # 记录当前要操作的 bit
            for num in nums:
                if num & bit != 0:
                    cnt += 1
            if cnt % 3 != 0:
                # 不等于0说明唯一出现的数字在这个 bit 上是1
                res |= bit

        return res - 2 ** 32 if res > 2 ** 31 - 1 else res
```

- 为什么 Python 最后需要对返回值进行判断?

如果不这么做的话测试用例是[-2,-2,1,1,-3,1,-3,-3,-4,-2] 的时候，就会输出 4294967292。 其原因在于 Python 是动态类型语言，在这种情况下其会将符号位置的 1 看成了值，而不是当作符号“负数”。 这是不对的。 正确答案应该是 - 4，-4 的二进制码是 1111...100，就变成 2^32-4=4294967292，解决办法就是 减去 2 ** 32 。

> 之所以这样不会有问题的原因还在于题目限定的数组范围不会超过 2 ** 32

JavaScript:

```
var singleNumber = function (nums) {
  let res = 0;
  for (let i = 0; i < 32; i++) {
    let cnt = 0;
    let bit = 1 << i;
    for (let j = 0; j < nums.length; j++) {
      if (nums[j] & bit) cnt++;
    }
    if (cnt % 3 != 0) res = res | bit;
  }
  return res;
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

645. 错误的集合

和上面的 `137. 只出现一次的数字2` 思路一样。这题没有限制空间复杂度，因此直接 hashmap 存储一下没问题。 不多说了，我们来看一种空间复杂度$O(1)$的解法。

由于和 `137. 只出现一次的数字2` 思路基本一样，我直接复用了代码。具体思路是，将 nums 的所有索引提取出一个数组 idx，那么由 idx 和 nums 组成的数组构成 singleNumbers 的输入，其输出是唯二不同的两个数。

但是我们不知道哪个是缺失的，哪个是重复的，因此我们需要重新进行一次遍历，判断出哪个是缺失的，哪个是重复的。

```
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        ret = 0  # 所有数字异或的结果
        a = 0
        b = 0
        for n in nums:
            ret ^= n
        # 找到第一位不是0的
        h = 1
        while(ret & h == 0):
            h <<= 1
        for n in nums:
            # 根据该位是否为0将其分为两组
            if (h & n == 0):
                a ^= n
            else:
                b ^= n

        return [a, b]

    def findErrorNums(self, nums: List[int]) -> List[int]:
        nums = [0] + nums
        idx = []
        for i in range(len(nums)):
            idx.append(i)
        a, b = self.singleNumbers(nums + idx)
        for num in nums:
            if a == num:
                return [a, b]
        return [b, a]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

260. 只出现一次的数字 3

题目大意是除了两个数字出现一次，其他都出现了两次，让我们找到这个两个数。

我们进行一次全员异或操作，得到的结果就是那两个只出现一次的不同的数字的异或结果。

我们刚才讲了异或的规律中有一个 `任何数和本身异或则为0` ， 因此我们的思路是能不能将这两个不同的数字分成两组 A 和 B。 分组需要满足两个条件.

1. 两个独特的的数字分成不同组
2. 相同的数字分成相同组

这样每一组的数据进行异或即可得到那两个数字。

问题的关键点是我们怎么进行分组呢?

由于异或的性质是，同一位相同则为 0，不同则为 1. 我们将所有数字异或的结果一定不是 0，也就是说至少有一位是 1.

我们随便取一个， 分组的依据就来了， 就是你取的那一位是 0 分成 1 组，那一位是 1 的分成一组。 这样肯定能保证 `2. 相同的数字分成相同组` ，不同的数字会被分成不同组么。 很明显当然可以， 因此我们选择是 1，也就是 说 `两个独特的的数字` 在那一位一定是不同的，因此两个独特元素一定会被分成不同组。

```python
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        ret = 0  # 所有数字异或的结果
        a = 0
        b = 0
        for n in nums:
            ret ^= n
        # 找到第一位不是0的
        h = 1
        while(ret & h == 0):
            h <<= 1
        for n in nums:
            # 根据该位是否为0将其分为两组
            if (h & n == 0):
                a ^= n
            else:
                b ^= n

        return [a, b]
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

相关题目

- 190. 颠倒二进制位（简单）
- 191. 位 1 的个数（简单）
- 338. 比特位计数（中等）
- 1072. 按列翻转得到最大值等行数（中等）

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

小岛问题

LeetCode 上有很多小岛题，虽然官方没有这个标签， 但是在我这里都差不多。不管是思路还是套路都比较类似，大家可以结合起来练习。

不严谨地讲，小岛问题是 DFS 的子专题。

套路

这种题目的套路都是 DFS，从一个或多个入口 DFS 即可。 DFS 的时候，我们往四个方向延伸即可。

一个最经典的代码模板：

```
seen = set()
def dfs(i, j):
  if i 越界 or j 越界: return
  if (i, j) in seen: return
  temp = board[i][j]
  # 标记为访问过
  seen.add((i, j))
  # 上
  dfs(i + 1, j)
  # 下
  dfs(i - 1, j)
  # 右
  dfs(i, j + 1)
  # 左
  dfs(i, j - 1)
  # 撤销标记
  seen.remove((i, j))
# 单点搜索
dfs(0, 0)
# 多点搜索
for i in range(M):
   for j in range(N):
      dfs(i, j)
```

有时候我们甚至可以不用 visited 来标记每个 cell 的访问情况， 而是直接原地标记，这种算法的空间复杂度会更好。这也是一个很常用的技巧，大家要熟练掌握。

```
def dfs(i, j):
  if i 越界 or j 越界: return
  if board[i][j] == -1: return
  temp = board[i][j]
  # 标记为访问过
  board[i][j] = -1
  # 上
  dfs(i + 1, j)
  # 下
  dfs(i - 1, j)
  # 右
  dfs(i, j + 1)
  # 左
  dfs(i, j - 1)
  # 撤销标记
  board[i][j] = temp
# 单点搜索
dfs(0, 0)
# 多点搜索
for i in range(M):
   for j in range(N):
      dfs(i, j)
```

相关题目

- 200. 岛屿数量
- 695. 岛屿的最大面积(字节跳动原题)
- 1162. 地图分析
- 463.岛屿的周长

上面四道题都可以使用常规的 DFS 来做。 并且递归的方向都是上下左右四个方向。更有意思的是，都可以采用原地修改的方式，来减少开辟 visited 的空间。

其中 463 题， 只是在做 DFS 的时候，需要注意相邻的各自边长可能会被重复计算， 因此需要减去。这里我的思路是：

- 遇到陆地就加 4
- 继续判断其左侧和上方是否为陆地
  - 如果是的话，会出现重复计算，这个时候重复计算的是 2，因此减去 2 即可
  - 如果不是，则不会重复计算， 不予理会即可

注意，右侧和下方的就不需要算了，否则还是会重复计算。

代码：

```
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int
        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or grid[i
                return 0
            grid[i][j] = -1
            ans = 4 + dfs(i + 1, j) + dfs(i - 1, j) + \
                dfs(i, j + 1) + dfs(i, j - 1)
            if i > 0 and grid[i - 1][j] != 0:
                ans -= 2
            if j > 0 and grid[i][j - 1] != 0:
                ans -= 2
            return ans

        m, n = len(grid), len(grid[0])
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    return dfs(i, j)
```

当然， 你选择判断右侧和下方也是一样的，只需要改**两行**代码即可，这两种算法没有什么区别。代码：

```
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int
        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or grid[i
                return 0
            grid[i][j] = -1
            ans = 4 + dfs(i + 1, j) + dfs(i - 1, j) + \
                dfs(i, j + 1) + dfs(i, j - 1)
            # 这里需要变
            if i < m - 1 and grid[i + 1][j] != 0:
                ans -= 2
            # 这里需要变
            if j < n - 1 and grid[i][j + 1] != 0:
                ans -= 2
            return ans

        m, n = len(grid), len(grid[0])
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    return dfs(i, j)
```

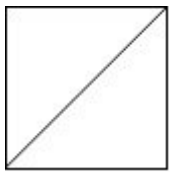
如果你下次碰到了小岛题目， 或者可以抽象为小岛类模型的题目，可以尝试使用本节给大家介绍的模板。这种题目的规律性很强， 类似的还有石子游戏，石子游戏大多数可以使用 DP 来做，这就是一种套路。

扩展

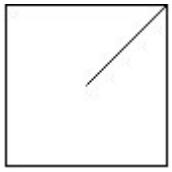
实际上，很多题都有小岛题的影子，所谓的小岛题的核心是求连通区域。如果你能将问题转化为求连通区域，那么就可以使用本节的思路去做。比如 959. 由斜杠划分区域

题目描述：

```
在由 1 x 1 方格组成的 N x N 网格 grid 中，每个 1 x 1 方块由 /、\

（请注意，反斜杠字符是转义的，因此 \ 用 "\\" 表示。）。

返回区域的数目。

示例 1：

输入：
[
  " /",
  "/ "
]
输出：2
解释：2x2 网格如下：
```

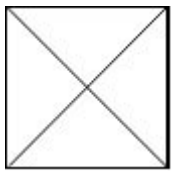


```
示例 2：

输入：
[
  " /",
  "  "
]
输出：1
解释：2x2 网格如下：
```



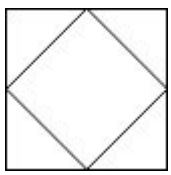
示例 3:

输入:
[
  "\\/",
  "/\\"
]
输出: 4
解释: (回想一下，因为 \ 字符是转义的，所以 "\\/" 表示 \/，而 "/\\"
2x2 网格如下:

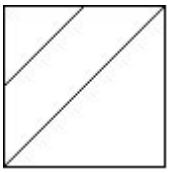


示例 4:

输入:
[
  "/\\",
  "\\/"
]
输出: 5
解释: (回想一下，因为 \ 字符是转义的，所以 "/\\" 表示 /\，而 "\\/"
2x2 网格如下:



```
示例 5：

输入：
[
  "//",
  "/ "
]
输出：3
解释：2x2 网格如下：
```



```
提示：

1 <= grid.length == grid[0].length <= 30
grid[i][j] 是 '/'、'\'、或 ' '。
```

实际上，如果你将题目中的 "/" 和 "\" 都转化为 一个 3 x 3 的网格之后，问题就变成了求连通区域的个数，就可以用本节的思路去解决了。具体留给读者去思考吧，这里给大家贴一个 Python3 的代码。

```
class Solution:
    def regionsBySlashes(self, grid: List[str]) -> int:
        m, n = len(grid), len(grid[0])
        new_grid = [[0 for _ in range(3 * n)] for _ in rang
        ans = 0
        # 预处理，生成新的 3 * m * 3 * n 的网格
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '/':
                    new_grid[3 * i][3 * j + 2] = 1
                    new_grid[3 * i + 1][3 * j + 1] = 1
                    new_grid[3 * i + 2][3 * j] = 1
                if grid[i][j] == '\\':
                    new_grid[3 * i][3 * j] = 1
                    new_grid[3 * i + 1][3 * j + 1] = 1
                    new_grid[3 * i + 2][3 * j + 2] = 1·
        def dfs(i, j):
            if 0 <= i < 3 * m and 0 <= j < 3 * n and new_gr
                new_grid[i][j] = 1
                dfs(i + 1, j)
                dfs(i - 1, j)
                dfs(i, j + 1)
                dfs(i, j - 1)
        for i in range(3 * m):
            for j in range(3 * n):
                if new_grid[i][j] == 0:
                    ans += 1
                    dfs(i, j)
        return ans
```

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

我是如何用最大公约数秒杀算法题的

关于最大公约数有专门的研究。 而在 LeetCode 中虽然没有直接让你求解最大公约数的题目。但是却有一些间接需要你求解最大公约数的题目。

比如：

- 914. 卡牌分组
- 365. 水壶问题
- 1071. 字符串的最大公因子

因此如何求解最大公约数就显得重要了。

如何求最大公约数?

定义法

```
def GCD(a: int, b: int) -> int:
    smaller = min(a, b)
    while smaller:
        if a % smaller == 0 and b % smaller == 0:
            return smaller
        smaller -= 1
```

复杂度分析

- 时间复杂度：最好的情况是执行一次循环体，最坏的情况是循环到 smaller 为 1，因此总的时间复杂度为 $O(N)$，其中 N 为 a 和 b 中较小的数。
- 空间复杂度：$O(1)$。

辗转相除法

如果我们需要计算 a 和 b 的最大公约数，运用辗转相除法的话。首先，我们先计算出 a 除以 b 的余数 c，把问题转化成求出 b 和 c 的最大公约数；然后计算出 b 除以 c 的余数 d，把问题转化成求出 c 和 d 的最大公约数；再然后计算出 c 除以 d 的余数 e，把问题转化成求出 d 和 e 的最大公约数。..... 以此类推，逐渐把两个较大整数之间的运算转化为两个较小整数之间的运算，直到两个数可以整除为止。

```
def GCD(a: int, b: int) -> int:
    return a if b == 0 else GCD(b, a % b)
```

复杂度分析

- 时间复杂度：$O(log(max(a, b)))$
- 空间复杂度：空间复杂度取决于递归的深度，因此空间复杂度为 $O(log(max(a, b)))$

更相减损术

辗转相除法如果 a 和 b 都很大的时候，a % b 性能会较低。在中国，《九章算术》中提到了一种类似辗转相减法的 更相减损术。它的原理是：`两个正整数 a 和 b（a>b），它们的最大公约数等于 a-b 的差值 c 和较小数 b 的最大公约数。` 。

```
def GCD(a: int, b: int) -> int:
    if a == b:
        return a
    if a < b:
        return GCD(b - a, a)
    return GCD(a - b, b)
```

上面的代码会报栈溢出。原因在于如果 a 和 b 相差比较大的话，递归次数会明显增加，要比辗转相除法递归深度增加很多，最坏时间复杂度为 O(max(a, b)))。这个时候我们可以将 `辗转相除法` 和 `更相减损术` 做一个结合，从而在各种情况都可以获得较好的性能。

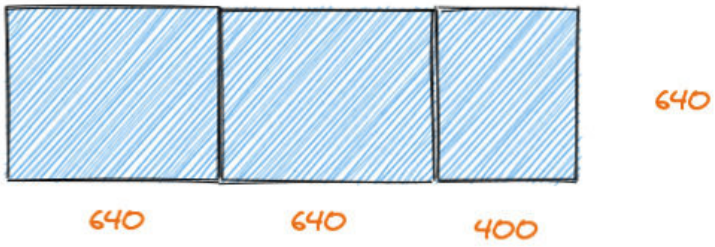
形象化解释

下面我们对上面的过程进行一个表形象地讲解，实际上这也是教材里面的讲解方式，我只是照搬过来，增加一下自己的理解罢了。我们来通过一个例子来讲解：

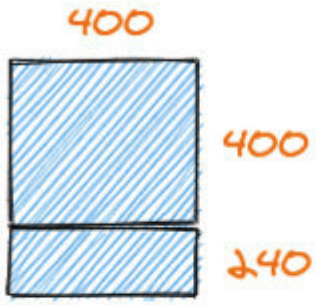
假如我们有一块 1680 米 * 640 米 的土地，我们希望讲起分成若干正方形的土地，且我们想让正方形土地的边长尽可能大，我们应该如何设计算法呢?

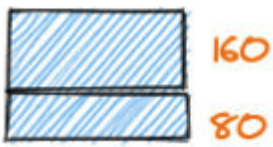
实际上这正是一个最大公约数的应用场景，我们的目标就是求解 1680 和 640 的最大公约数。

将 1680 米 * 640 米 的土地分割，相当于对将 400 米 * 640 米 的土地进行分割。 为什么呢？ 假如 400 米 * 640 米分割的正方形边长为 x，那么有 640 % x == 0，那么肯定也满足剩下的两块 640 米 * 640 米的。



我们不断进行上面的分割：





直到边长为 80，没有必要进行下去了。

实例解析

题目描述

```
给你三个数字 a，b，c，你需要找到第 n 个（n 从 0 开始）有序序列的值，

比如：
n = 8
a = 2
b = 5
c = 7

由于 2，5，7 构成的整数倍构成的有序序列为 [1, 2, 4, 5, 6, 7, 8, 1

注意：我们约定，有序序列的第一个永远是 1。
```

思路

大家可以通过 这个网站 在线验证。

一个简单的思路是使用堆来做，唯一需要注意的是去重，我们可以使用一个哈希表来记录出现过的数字，以达到去重的目的。

代码：

```
ss Solution:
    def solve(self, n, a, b, c):
        seen = set()
        h = [(a, a, 1), (b, b, 1), (c, c, 1)]
        heapq.heapify(h)

        while True:
            cur, base, times = heapq.heappop(h)
            if cur not in seen:
                n -= 1
                seen.add(cur)
            if n == 0:
                return cur
            heapq.heappush(h, (base * (times + 1), base, t
```

对于此解法不理解的可先看下我之前写的 几乎刷完了力扣所有的堆题，我发现了这些东西。。。（第二弹）

然而这种做法时间复杂度太高，有没有更好的做法呢?

实际上，我们可对搜索空间进行二分。首先思考一个问题，如果给定一个数字 x，那么有序序列中小于等于 x 的值有几个。

答案是 x // a + x // b + x // c 吗?

> // 是地板除

可惜不是的。比如 a = 2, b = 4, n = 4，答案显然不是 4 // 2 + 4 // 4 = 3，而是 2。这里出错的原因在于 4 被计算了两次，一次是 $2 *2 = 4$，另一次是 $4* 1 = 4$。

为了解决这个问题，我们可以通过集合论的知识。

一点点集合知识：

- 如果把有序序列中小于等于 x 的可以被 x 整除，且是 a 的倍数的值构成的集合为 SA，集合大小为 A
- 如果把有序序列中小于等于 x 的可以被 x 整除，且是 b 的倍数的值构成的集合为 SB，集合大小为 B
- 如果把有序序列中小于等于 x 的可以被 x 整除，且是 c 的倍数的值构成的集合为 SC，集合大小为 C

那么最终的答案就是 SA ，SB，SC 构成的大的集合（需要去重）的中的数字的个数，也就是：

$$A + B + C - sizeof(SA \cap SB) - sizeof(SB \cap SC) - sizeof(SA \cap SC) +$$

问题转化为 A 和 B 集合交集的个数如何求？

> A 和 B，B 和 C， A 和 C ，甚至是 A，B，C 的交集求法都是一样的。

实际上， SA 和 SB 的交集个数就是 x // lcm(A, B)，其中 lcm 为 A 和 B 的最小公倍数。而最小公倍数则可以通过最大公约数计算出来：

```
def lcm(x, y):
    return x * y // gcd(x, y)
```

接下来就是二分套路了，二分部分看不懂的建议看下我的二分专题。

代码（Python3）

```
class Solution:
    def solve(self, n, a, b, c):
        def gcd(x, y):
            if y == 0:
                return x
            return gcd(y, x % y)

        def lcm(x, y):
            return x * y // gcd(x, y)

        def possible(mid):
            return (mid // a + mid // b + mid // c - mid //

        l, r = 1, n * max(a, b, c)
        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l
```

复杂度分析

- 时间复杂度：$logn$。
- 空间复杂度：gcd 和 lcm 的递归树深度，基本可忽略不计。

总结

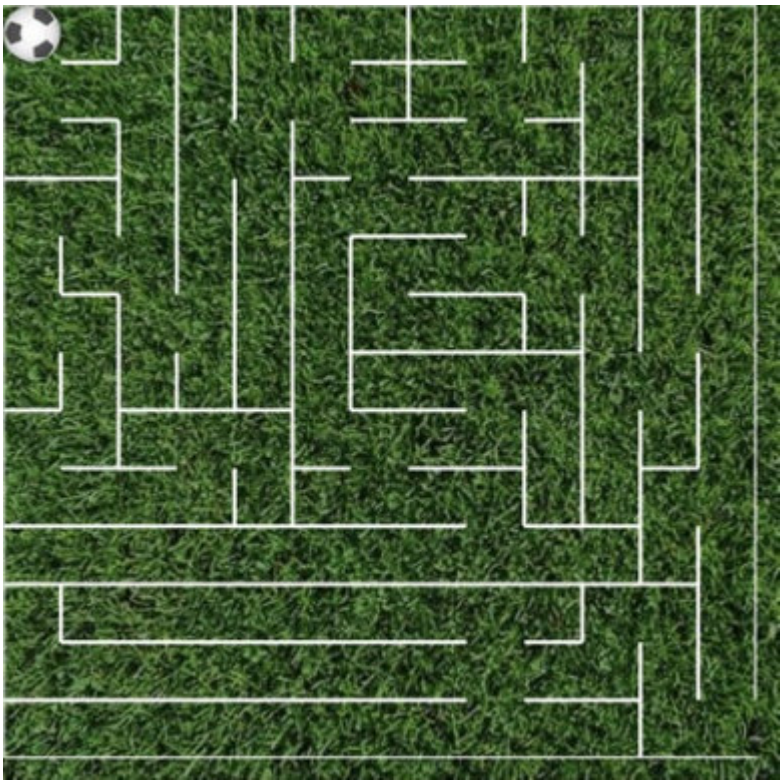
通过这篇文章，我们不仅明白了最大公约数的**概念以及求法**。也形象化地感知到了最大公约数计算的**原理**。最大公约数和最小公倍数是两个相似的概念， 关于最大公约数和最小公倍数的题目在力扣中不算少，大家可以通过**数学标签**找到这些题。更多关于算法中的数学知识，可以参考这篇文章刷算法题必备的数学考点汇总

> 这篇文章的第二篇也马上要发布了。

并查集

背景

相信大家都玩过下面的迷宫游戏。你的目标是从地图的某一个角落移动到地图的出口。规则很简单，仅仅你不能穿过墙。



实际上，这道题并不能够使用并查集来解决。 不过如果我将规则变成，“是否存在一条从入口到出口的路径”，那么这就是一个简单的联通问题，只不过用肉眼去观察实在是太费力了。幸好，我们有计算器可以帮助我们。并查集算法就可以用来解决这个问题，并且比暴力枚举效率高。如果地图不变，而不断改变入口和出口的位置，并查集的效果高的超出你的想象。

另外并查集还可以在人工智能中用作图像人脸识别等，感兴趣的可以查找一下相关资料。

概述

并查集使用的是一种**树型**的数据结构，用于处理一些不交集（Disjoint Sets）的合并及查询问题。比如让你求两个人是否间接认识，两个地点之间是否有至少一条路径。上面的例子其实都可以抽象为联通性问题。即如果两个点联通，那么这两个点就有至少一条路径能够将其连接起来。值得

注意的是，并查集只能回答“联通与否”，而不能回答诸如“具体的联通路径是什么”。如果要回答“具体的联通路径是什么”这个问题，则需要借助其他算法，比如广度优先遍历。

并查集（Union-find Algorithm）定义了两个用于此数据结构的操作：

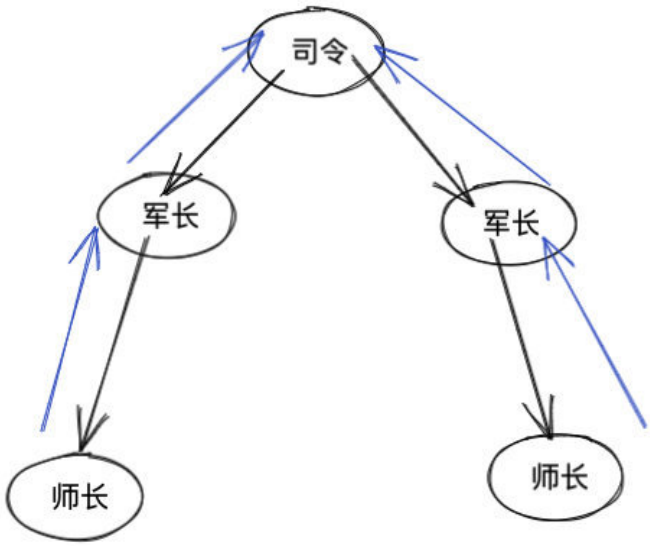
- Find：确定元素属于哪一个子集。它可以被用来确定两个元素是否属于同一子集。
- Union：将两个子集合并成同一个集合。

形象解释

比如有两个司令。 司令下有若干军长，军长下有若干师长。。。

判断两个节点是否联通

我们如何判断某两个师长是否归同一个司令管呢（连通性）？



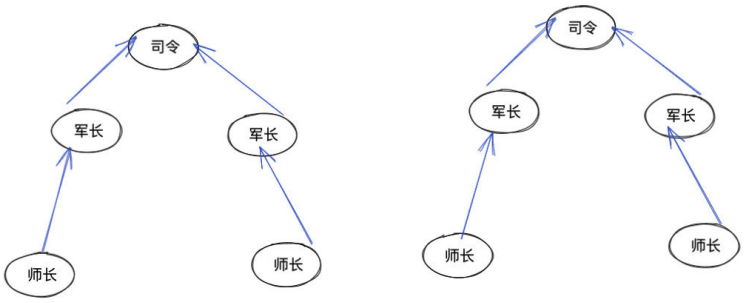
很简单，我们顺着师长，往上找，找到司令。 如果两个师长找到的是同一个司令，那么两个人就归同一个司令管。

代码上我们可以用 parent[x] = y 表示 x 的 parent 是 y，通过不断沿着搜索 parent 搜索找到 root，然后比较 root 是否相同即可得出结论。 这里的 root 实际上就是上文提到的**集合代表**。

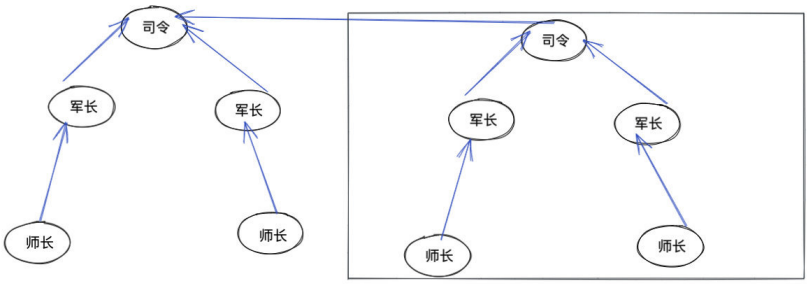
这个不断往上找的操作，我们一般称为 find，使用 ta 我们可以很容易地求出两个节点是否连通。

合并两个联通区域

如图有两个司令：



我们将其合并为一个联通域，最简单的方式就是直接将其中一个司令指向另外一个即可：



以上就是三个核心 API `find`， `connnected` 和 `union`， 的形象化解释，下面我们来看下代码实现。

核心 API

首先我们初始化每一个点都是一个连通域，类似下图：

为了更加精确的定义这些方法，需要定义如何表示集合。一种常用的策略是为每个集合选定一个固定的元素，称为代表，以表示整个集合。接着，Find(x) 返回 x 所属集合的代表，而 Union 使用两个集合的代表作为参数进行合并。

> 这里的代表就是上面的“司令”。

find

```
def find(self, x):
    while x != self.parent[x]:
        x = self.parent[x]
    return x
```

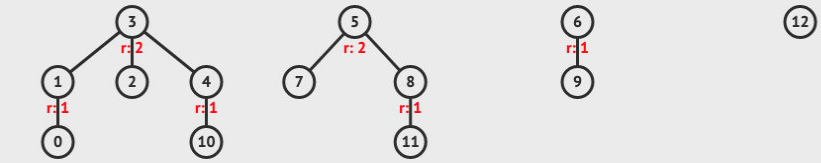
也可使用递归来实现。

```
def find(self, x):
    if x != self.parent[x]:
        self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
    return x
```

这里我在递归实现的 find 过程进行了路径的压缩，每次往上查找之后都会将树的高度降低到 2。

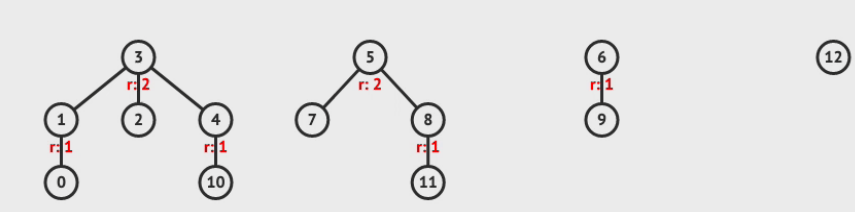
这有什么用呢？我们知道每次 find 都会从当前节点往上不断搜索，直到到达根节点，因此 find 的时间复杂度大致相等于节点的深度，最坏的情况节点的深度等于树的高度，树的高度如果不加控制则可能为节点数，因此 find 的时间复杂度可能会退化到 $O(n)$。而如果进行路径压缩，那么树的平均高度不会超过 $logn$，具体证明略。不过给大家画了一个图来辅助大家理解。

比如对于如下的一个图：

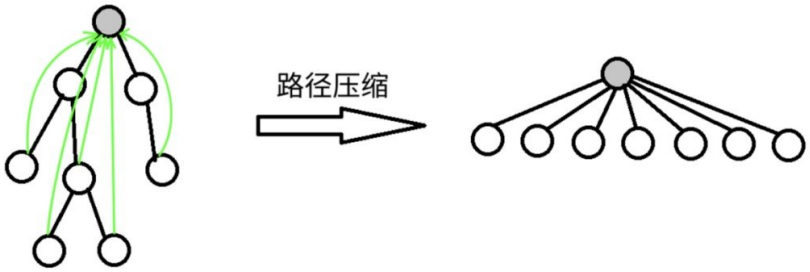


图中 r:1 表示 秩为 1，r 是 rank 的简写。这里的秩其实对应的就是上文的 size。（下同，不再赘述）

调用 find(0) 会逐步找到 3 ，在找到 3 的过程上会将路径上的节点都指向根节点。



极限情况下，每一个路径都会被压缩，这种情况下继续查找的时间复杂度就是 $O(1)$。



connected

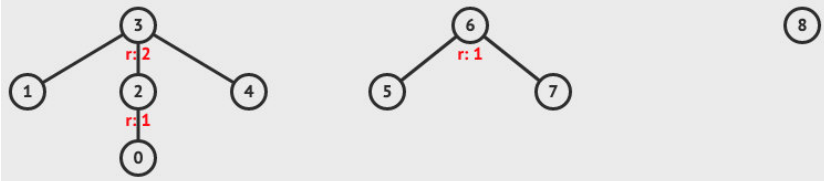
直接利用上面实现好的 find 方法即可。如果两个节点的祖先相同，那么其就联通。

```
def connected(self, p, q):
    return self.find(p) == self.find(q)
```

union

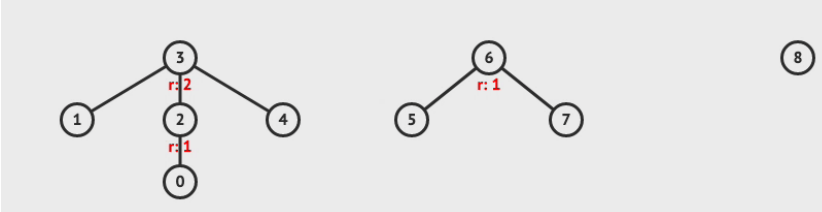
将其中一个节点挂到另外一个节点的祖先上，这样两者祖先就一样了。也就是说，两个节点联通了。

对于如下的一个图：



如果我们将 0 和 7 进行一次合并。即 `union(0, 7)` ，则会发生如下过程。

- 找到 0 的根节点 3
- 找到 7 的根节点 6
- 将 6 指向 3。（为了使得合并之后的树尽可能平衡，一般选择将小树挂载到大树上面，下面的代码模板会体现这一点。3 的秩比 6 的秩大，这样更利于树的平衡，避免出现极端的情况）



代码：

```
def union(self, p, q):
    if self.connected(p, q): return
    self.parent[self.find(p)] = self.find(q)
```

这里我并没有判断秩的大小关系，目的是方便大家理清主脉络。完整代码见下面代码区。

不带权并查集

平时做题过程，遇到的更多的是不带权的并查集。相比于带权并查集，其实现过程也更加简单。

带路径压缩的代码模板

```
class UF:
    def __init__(self, M):
        self.parent = {}
        self.size = {}
        self.cnt = 0
        # 初始化 parent, size 和 cnt
        for i in range(M):
            self.parent[i] = i
            self.cnt += 1
            self.size[i] = 1

    def find(self, x):
        if x != self.parent[x]:
            self.parent[x] = self.find(self.parent[x])
            return self.parent[x]
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        # 小的树挂到大的树上，  使树尽量平衡
        leader_p = self.find(p)
        leader_q = self.find(q)
        if self.size[leader_p] < self.size[leader_q]:
            self.parent[leader_p] = leader_q
            self.size[leader_q] += self.size[leader_p]
        else:
            self.parent[leader_q] = leader_p
            self.size[leader_p] += self.size[leader_q]
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)
```

带权并查集

上面讲到的其实都是有向无权图，因此仅仅使用 parent 表示节点关系就可以了。而如果使用的是有向带权图呢？实际上除了维护 parent 这样的节点指向关系，我们还需要维护节点的权重，一个简单的想法是使用另外一个哈希表 weight 存储节点的权重关系。比如 `weight[a] = 1` 表示 `a` 到其父节点的权重是 `1` 。

如果是带权的并查集，其查询过程的路径压缩以及合并过程会略有不同，因为我们不仅关心节点指向的变更，也关心权重如何更新。比如：

```
a    b
^    ^
|    |
|    |
x    y
```

如上表示的是 x 的父节点是 a，y 的父节点是 b，现在我需要将 x 和 y 进行合并。

```
a    b
^    ^
|    |
|    |
x -> y
```

假设 x 到 a 的权重是 w(xa)，y 到 b 的权重为 w(yb)，x 到 y 的权重是 w(xy)。合并之后会变成如图的样子：

```
a -> b
^    ^
|    |
|    |
x    y
```

那么 a 到 b 的权重应该被更新为什么呢？我们知道 w(xa) + w(ab) = w(xy) + w(yb)，也就是说 a 到 b 的权重 w(ab) = w(xy) + w(yb) - w(xa)。

当然上面关系式是加法，减法，取模还是乘法，除法等完全由题目决定，我这里只是举了一个例子。不管怎么样，这种运算一定需要满足**可传导性**。

带路径压缩的代码模板

这里以加法型带权并查集为例，讲述一下代码应该如何书写。

```
class UF:
    def __init__(self, M):
        # 初始化 parent, weight
        self.parent = {}
        self.weight = {}
        for i in range(M):
            self.parent[i] = i
            self.weight[i] = 0

   def find(self, x):
        if self.parent[x] != x:
            ancestor, w = self.find(self.parent[x])
            self.parent[x] = ancestor
            self.weight[x] += w
        return self.parent[x], self.weight[x]
    def union(self, p, q, dist):
        if self.connected(p, q): return
        leader_p, w_p = self.find(p)
        leader_q, w_q = self.find(q)
        self.parent[leader_p] = leader_q
        self.weight[leader_p] = dist + w_q - w_p
    def connected(self, p, q):
        return self.find(p)[0] == self.find(q)[0]
```

典型题目：

- 399. 除法求值

应用

- 检测图是否有环

思路： 只需要将边进行合并，并在合并之前判断是否已经联通即可，如果合并之前已经联通说明存在环。

代码：

```
uf = UF()
for a, b in edges:
    if uf.connected(a, b): return False
    uf.union(a, b)
return True
```

题目推荐：

- 684. 冗余连接
- Forest Detection

- 最小生成树经典算法 Kruskal

练习

关于并查集的题目不少，官方给的数据是 30 道（截止 2020-02-20），但是有一些题目虽然官方没有贴 `并查集` 标签，但是使用并查集来说确非常简单。这类题目如果掌握模板，那么刷这种题会非常快，并且犯错的概率会大大降低，这就是模板的好处。

我这里总结了几道并查集的题目：

- 547. 朋友圈
- 721. 账户合并
- 990. 等式方程的可满足性
- 1202. 交换字符串中的元素
- 1697. 检查边长度限制的路径是否存在

上面的题目前面四道都是无权图的连通性问题，第五道题是带权图的连通性问题。两种类型大家都要会，上面的题目关键字都是**连通性**，代码都是套模板。看完这里的内容，建议拿上面的题目练下手，检测一下学习成果。

总结

如果题目有连通，等价的关系，那么你就可以考虑并查集，另外使用并查集的时候要注意路径压缩，否则随着树的高度增加复杂度会逐渐增大。

对于带权并查集实现起来比较复杂，主要是路径压缩和合并这块不一样，不过我们只要注意节点关系，画出如下的图：

```
a -> b
^    ^
|    |
|    |
x    y
```

就不难看出应该如何更新拉。

本文提供的题目模板是西法我用的比较多的，用了它不仅出错概率大大降低，而且速度也快了很多，整个人都更自信了呢 ^_^

平衡二叉树专题

力扣关于平衡二叉树的题目还是有一些的，并且都非常经典，推荐大家练习。今天给大家精选了 4 道题，如果你彻底搞明白了这几道题，碰到其他的平衡二叉树的题目应该不至于没有思路。当你领会了我的思路之后，建议再找几个题目练手，巩固一下学习成果。

110. 平衡二叉树（简单）

最简单的莫过于判断一个树是否为平衡二叉树了，我们来看下。

题目描述

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

思路

由于平衡二叉树定义为就是**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。**用伪代码描述就是：

```
if abs(高度(root.left) - 高度(root.right)) <= 1 and root.lef
    print('是平衡二叉树')
else:
    print('不是平衡二叉树')
```

而 root.left 和 root.right **如何判断是否是二叉平衡树就和 root 是一样的了**，可以看出这个问题有明显的递归性。

因此我们首先需要知道如何计算一个子树的高度。这个可以通过递归的方式轻松地计算出来。计算子树高度的 Python 代码如下：

```
def dfs(node):
    if not node: return 0
    l = dfs(node.left)
    r = dfs(node.right)
    return max(l, r) + 1
```

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def dfs(node):
            if not node: return 0
            l = dfs(node.left)
            r = dfs(node.right)
            return max(l, r)  + 1
        if not root: return True
        if abs(dfs(root.left) -  dfs(root.right)) > 1: retu
        return self.isBalanced(root.left) and self.isBalanc
```

复杂度分析

- 时间复杂度：对于 isBalanced 来说，由于每个节点最多被访问一次，这部分的时间复杂度为 $O(N)$，而 dfs 函数 每次被调用的次数不超过 $log N$，因此总的时间复杂度为 $O(NlogN)$，其中 N 为树的节点总数。

- 空间复杂度：由于使用了递归，这里的空间复杂度的瓶颈在栈空间，因此空间复杂度为 $O(h)$，其中 h 为树的高度。

108. 将有序数组转换为二叉搜索树（简单）

108 和 109 基本是一样的，只不过数据结构不一样，109 变成了链表了而已。由于链表操作比数组需要考虑更多的因素，因此 109 是 中等难度。

题目描述

```
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

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

      0
     / \
   -3   9
   /   /
 -10  5
```

思路

对于这个问题或者 `给定一个二叉搜索树，将其改为平衡（后面会讲）` 基本思路都是一样的。

题目的要求是将有序数组转化为：

1. 高度平衡的二叉树
2. 二叉搜索树

由于平衡二叉树是左右两个子树的高度差的绝对值不超过 1。因此一种简单的方法是**选择中点作为根节点，根节点左侧的作为左子树，右侧的作为右子树即可。**原因很简单，这样分配可以保证左右子树的节点数目差不超过 1。因此高度差自然也不会超过 1 了。

上面的操作同时也满足了二叉搜索树，原因就是题目给的数组是有序的。

> 你也可以选择别的数作为根节点，而不是中点，这也可以看出答案是不唯一的。

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> TreeNode
        if not nums: return None
        mid = (len(nums) - 1) // 2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid + 1:])
        return root
```

复杂度分析

- 时间复杂度：由于每个节点最多被访问一次，因此总的时间复杂度为 $O(N)$，其中 N 为数组长度。
- 空间复杂度：由于使用了递归，这里的空间复杂度的瓶颈在栈空间，因此空间复杂度为 $O(h)$，其中 h 为树的高度。同时由于是平衡二叉树，因此 h 就是 $log N$。

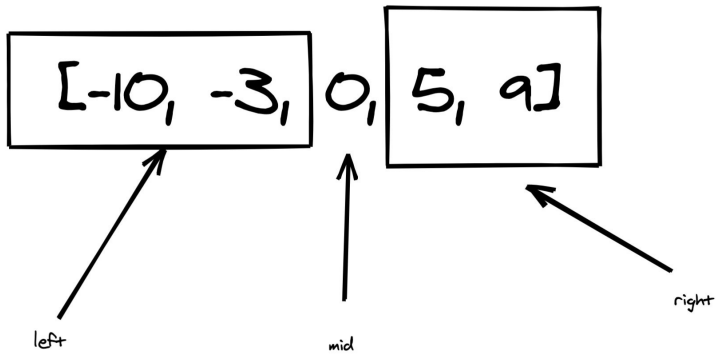
109. 有序链表转换二叉搜索树（中等）

题目描述

```
`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

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

      0
     / \
   -3   9
   /   /
 -10  5
```

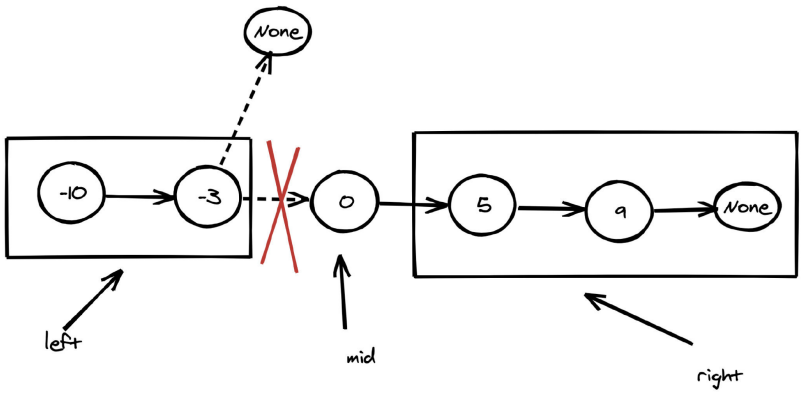
思路

和 108 思路一样。 不同的是数据结构的不同，因此我们需要关注的是链表和数组的操作差异。



（数组的情况）

我们再来看下链表：



（链表的情况）

找到中点，只需要使用经典的快慢指针即可。同时为了防止环的出现，我们需要斩断指向 mid 的 next 指针，因此需要记录一下中点前的一个节点，这只需要用一个变量 pre 记录即可。

代码

代码支持： Python3

Python3 Code:

```python
class Solution:
    def sortedListToBST(self, head: ListNode) -> TreeNode:
        if not head:
            return head
        pre, slow, fast = None, head, head

        while fast and fast.next:
            fast = fast.next.next
            pre = slow
            slow = slow.next
        if pre:
            pre.next = None
        node = TreeNode(slow.val)
        if slow == fast:
            return node
        node.left = self.sortedListToBST(head)
        node.right = self.sortedListToBST(slow.next)
        return node
```

复杂度分析

- 时间复杂度：由于每个节点最多被访问一次，因此总的时间复杂度为 $O(N)$，其中 N 为链表长度。
- 空间复杂度：由于使用了递归，这里的空间复杂度的瓶颈在栈空间，因此空间复杂度为 $O(h)$，其中 h 为树的高度。同时由于是平衡二叉树，因此 h 就是 $log N$。

1382. 将二叉搜索树变平衡（中等）

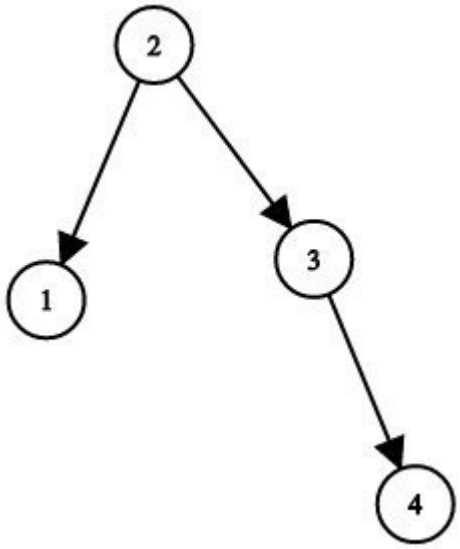
题目描述

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

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

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

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。
```

思路

由于 二叉搜索树的中序遍历是一个有序数组 ，因此问题很容易就转化为 108. 将有序数组转换为二叉搜索树（简单） 。

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def inorder(self, node):
        if not node: return []
        return self.inorder(node.left) + [node.val] + self.
    def balanceBST(self, root: TreeNode) -> TreeNode:
        nums = self.inorder(root)
        def dfs(start, end):
            if start == end: return TreeNode(nums[start])
            if start > end: return None
            mid = (start + end) // 2
            root = TreeNode(nums[mid])
            root.left = dfs(start, mid - 1)
            root.right = dfs(mid + 1, end)
            return root
        return dfs(0, len(nums) - 1)
```

复杂度分析

- 时间复杂度：由于每个节点最多被访问一次，因此总的时间复杂度为 $O(N)$，其中 N 为链表长度。
- 空间复杂度：虽然使用了递归，但是瓶颈不在栈空间，而是开辟的长度为 N 的 nums 数组，因此空间复杂度为 $O(N)$，其中 N 为树的节点总数。

总结

本文通过四道关于二叉平衡树的题帮助大家识别此类型题目背后的思维逻辑，我们来总结一下学到的知识。

平衡二叉树指的是：`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。`

如果需要让你判断一个树是否是平衡二叉树，只需要死扣定义，然后用递归即可轻松解决。

如果需要你将一个数组或者链表（逻辑上都是线性的数据结构）转化为平衡二叉树，只需要随便选一个节点，并分配一半到左子树，另一半到右子树即可。

同时，如果要求你转化为平衡二叉搜索树，则可以选择排序数组(或链表)的中点，左边的元素为左子树， 右边的元素为右子树即可。

> 小提示 1： 如果不需要是二叉搜索树则不需要排序，否则需要排序。
>
> 小提示 2： 你也可以不选择中点， 算法需要相应调整，感兴趣的同学可以试试。
>
> 小提示 3： 链表的操作需要特别注意环的存在。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

蓄水池抽样

力扣中关于蓄水池抽样问题官方标签是 2 道，根据我的做题情况来看，可能有三四道。比重算是比较低的，大家可以根据自己的实际情况选择性掌握。

蓄水池抽样的算法思维很巧妙，代码简单且容易理解，就算不掌握它，作为了解也是很不错的。

问题描述

给出一个数据流，我们需要在此数据流中随机选取 k 个数。由于这个数据流的长度很大，因此需要边遍历边处理，而不能将其一次性全部加载到内存。

请写出一个随机选择算法，使得数据流中所有数据被**等概率**选中。

这种问题的表达形式有很多。比如让你随机从一个矩形中抽取 k 个点，随机从一个单词列表中抽取 k 个单词等等，要求你等**概率随机抽取**。不管描述怎么变，其本质上都是一样的。今天我们就来看看如何做这种题。

算法描述

这个算法叫蓄水池抽样算法（reservoid sampling）。

其基本思路是：

- 构建一个大小为 k 的数组，将数据流的前 k 个元素放入数组中。
- 对数据流的前 k 个数**先**不进行任何处理。
- 从数据流的第 k + 1 个数开始，在 [1, i] 之间选一个数 rand，其中 i 表示当前是第几个数。
- 如果 rand 大于等于 k 什么都不做
- 如果 rand 小于 k， 将 rand 和 i 交换，也就是说选择当前的数代替已经被选中的数（备胎）。
- 最终返回幸存的备胎即可

这种算法的核心在于先以某一种概率选取数，并在后续过程以另一种概率换掉之前已经被选中的数。因此实际上每个数被最终选中的概率都是**被选中的概率 * 不被替换的概率**。

伪代码：

> 伪代码参考的某一本算法书，并略有修改。

```
Init : a reservoir with the size:  k
for i= k+1 to N
    if(random(1, i) < k) {
        SWAP the Mth value and ith value
    }
```

这样可以保证被选择的数是等概率的吗？答案是肯定的。

- 当 i <= k ，i 被选中的概率是 1。
- 到第 k + 1 个数时，第 k + 1 个数被选中的概率（走进上面的 if 分支的概率）是 $\frac{k}{k+1}$，到第 k + 2 个数时，第 k + 2 个数被选中的概率（走进上面的 if 分支的概率）是 $\frac{k}{k+2}$，以此类推。那么第 n 个数被选中的概率就是 $\frac{k}{n}$
- 上面分析了被选中的概率，接下来分析不被替换的概率。到第 k + 1 个数时，前 k 个数被替换的概率是 $\frac{1}{k}$。到前 k + 2 个数时，第 k + 2 个数被替换的概率是 $\frac{1}{k}$，以此类推。也就是说所有的被替换的概率都是 $\frac{1}{k}$。知道了被替换的概率，那么不被替换的概率其实就是 1 - 被替换的概率。

因此对于前 k 个数，最终被选择的概率都是 1 * 不被 k + 1 替换的概率 * 不被 k + 2 替换的概率 * ... 不被 n 替换的概率，即 1 * (1 - 被 k + 1 替换的概率) * (1 - 被 k + 2 替换的概率) * ... (1 - 被 n 替换的概率)，即 $1 \times (1 - \frac{k}{k+1} \times \frac{1}{k}) \times (1 - \frac{k}{k+2} \times \frac{1}{k}) \times ... \times (1 - \frac{k}{n} \times \frac{1}{k}) = \frac{k}{n} $。

对于 第 i (i > k) 个数，最终被选择的概率是 第 i 步被选中的概率 * 不被第 i + 1 步替换的概率 * ... * 不被第 n 步被替换的概率， 即 $\frac{k}{k+1} \times (1 - \frac{k}{k+2} \times \frac{1}{k}) \times ... \times (1 - \frac{k}{n} \times \frac{1}{k}) = \frac{k}{n} $。

总之，不管是哪个数，被选中的概率都是 $\frac{k}{n}$，满足概率相等的需求。

相关题目

- 382. 链表随机节点
- 398. 随机数索引
- 497. 非重叠矩形中的随机点

总结

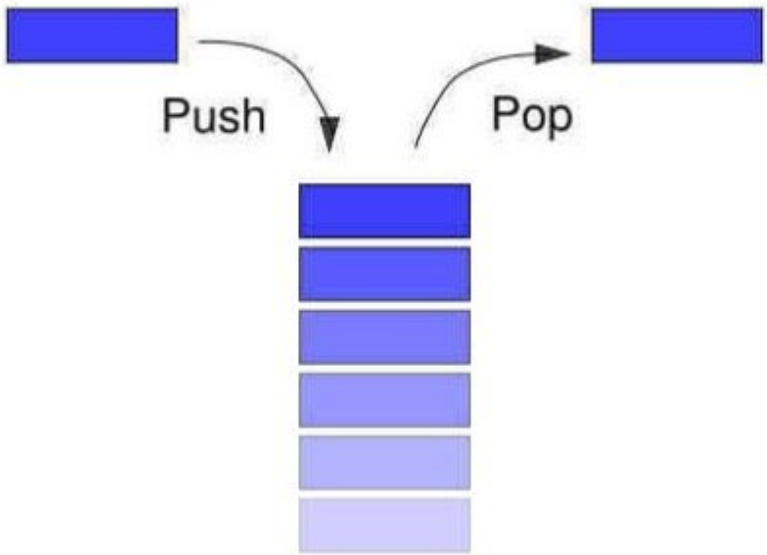
蓄水池抽样算法核心代码非常简单。但是却不容易想到，尤其是之前没见过的情况下。其核心点在于每个数被最终选中的概率都是**被选中的概率 * 不被替换的概率**。于是我们可以采取某一种动态手段，使得每一轮都有概

率选中和替换一些数字。 上面我们有给出了概率相等的证明过程，大家不妨自己尝试证明一下。之后结合文末的相关题目练习一下，效果会更好。

单调栈

顾名思义， 单调栈是一种栈。因此要学单调栈，首先要彻底搞懂栈。

栈是什么?



栈是一种受限的数据结构， 体现在只允许新的内容从一个方向插入或删除，这个方向我们叫栈顶，而从其他位置获取内容是不被允许的

栈最显著的特征就是 LIFO(Last In, First Out - 后进先出)

举个例子：

栈就像是一个放书本的抽屉，进栈的操作就好比是想抽屉里放一本书，新进去的书永远在最上层，而退栈则相当于从里往外拿书本，永远是从最上层开始拿，所以拿出来的永远是最后进去的哪一个

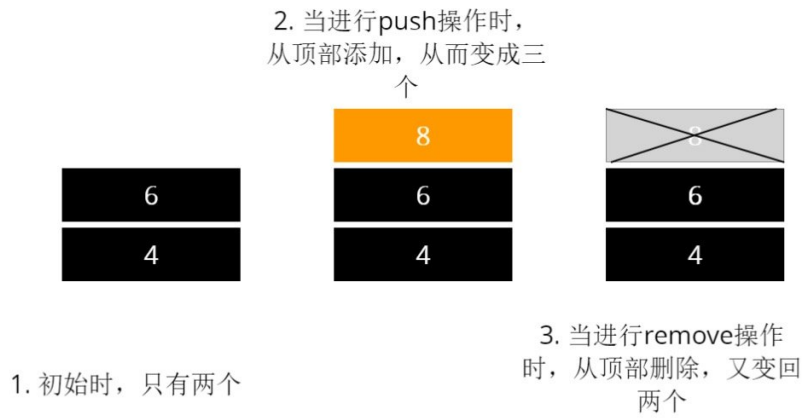
栈的常用操作

1. 进栈 - push - 将元素放置到栈顶
2. 退栈 - pop - 将栈顶元素弹出
3. 栈顶 - top - 得到栈顶元素的值
4. 是否空栈 - isEmpty - 判断栈内是否有元素

栈的常用操作时间复杂度

由于栈只在尾部操作就行了，我们用数组进行模拟的话，可以很容易达到O(1)的时间复杂度。当然也可以用链表实现，即链式栈。

1. 进栈 - O(1)
2. 出栈 - O(1)



应用

- 函数调用栈
- 浏览器前进后退
- 匹配括号
- 单调栈用来寻找下一个更大（更小）元素

题目推荐

- 394. 字符串解码
- 946. 验证栈序列
- 1381. 设计一个支持增量操作的栈

单调栈又是什么？

单调栈是一种特殊的栈。栈本来就是一种受限的数据结构了，单调栈在此基础上又受限了一次（受限++）。

单调栈要求栈中的元素是单调递减或者单调递减的。

> 是否严格递减或递减可以根据实际情况来。

这里我用 [a,b,c] 表示一个栈。 其中 左侧为栈底，右侧为栈顶。单调增还是单调减取决于出栈顺序。如果出栈的元素是单调增的，那就是单调递增栈，如果出栈的元素是单调减的，那就是单调递减栈。

比如：

- [1,2,3,4] 就是一个单调递减栈（因为此时的出栈顺序是 4，3，2，1。下同，不再赘述）
- [3,2,1] 就是一个单调递增栈
- [1,3,2] 就不是一个合法的单调栈

那这个限制有什么用呢？这个限制（特性）能够解决什么用的问题呢？

适用场景

单调栈适合的题目是求解**下一个大于 xxx**或者**下一个小于 xxx**这种题目。所有当你有这种需求的时候，就应该想到单调栈。

那么为什么单调栈适合求解**下一个大于 xxx**或者**下一个小于 xxx**这种题目？原因很简单，我这里通过一个例子给大家讲解一下。

> 这里举的例子是单调递减栈

比如我们需要依次将数组 [1,3,4,5,2,9,6] 压入单调栈。

1. 首先压入 1，此时的栈为：[1]
2. 继续压入 3，此时的栈为：[1,3]
3. 继续压入 4，此时的栈为：[1,3,4]
4. 继续压入 5，此时的栈为：[1,3,4,5]
5. **如果**继续压入 2，此时的栈为：[1,3,4,5,2] 不满足单调递减栈的特性， 因此需要调整。如何调整？由于栈只有 pop 操作，因此我们只好不断 pop，直到满足单调递减为止。
6. 上面其实我们并没有压入 2，而是先 pop，pop 到压入 2 依然可以保持单调递减再 压入 2，此时的栈为：[1,2]
7. 继续压入 9，此时的栈为：[1,2,9]
8. **如果**继续压入 6，则不满足单调递减栈的特性， 我们故技重施，不断 pop，直到满足单调递减为止。此时的栈为：[1,2,6]

注意这里的栈仍然是非空的，如果有的题目需要用到所有数组的信息，那么很有可能因没有考虑边界而不能通过所有的测试用例。 这里介绍一个技巧 - **哨兵法**，这个技巧经常用在单调栈的算法中。

对于上面的例子，我可以在原数组 [1,3,4,5,2,9,6] 的右侧添加一个小于数组中最小值的项即可，比如 -1。此时的数组是 [1,3,4,5,2,9,6,-1]。这种技巧可以简化代码逻辑，大家尽量掌握。

上面的例子如果你明白了，就不难理解为啥单调栈适合求解**下一个大于 xxx**或者**下一个小于 xxx**这种题目了。比如上面的例子，我们就可以很容易地求出**在其之后第一个小于其本身的位置**。比如 3 的索引是 1，小于 3 的第一个索引是 4，2 的索引 4，小于 2 的第一个索引是 0，但是其在 2 的索引 4 之后，因此不符合条件，也就是不存在**在 2 之后第一个小于 2 本身的位置**。

上面的例子，我们在第 6 步开始 pop，第一个被 pop 出来的是 5，因此 5 之后的第一个小于 5 的索引就是 4。同理被 pop 出来的 3，4，5 也都是 4。

如果用 ans 来表示**在其之后第一个小于其本身的位置**，ans[i] 表示 arr[i] 之后第一个小于 arr[i] 的位置， ans[i] 为 -1 表示这样的位置不存在，比如前文提到的 2。那么此时的 ans 是 [-1,4,4,4,-1,-1,-1]。

第 8 步，我们又开始 pop 了。此时 pop 出来的是 9，因此 9 之后第一个小于 9 的索引就是 6。

这个算法的过程用一句话总结就是，**如果压栈之后仍然可以保持单调性，那么直接压。否则先弹出栈的元素，直到压入之后可以保持单调性。** 这个算法的原理用一句话总结就是，**被弹出的元素都是大于当前元素的，并且由于栈是单调增的，因此在其之后小于其本身的最近的就是当前元素了**

下面给大家推荐几道题，大家趁着知识还在脑子来，赶紧去刷一下吧~

伪代码

上面的算法可以用如下的伪代码表示，同时这是一个通用的算法模板，大家遇到单调栈的题目可以直接套。

建议大家用自己熟悉的编程语言实现一遍，以后改改符号基本就能用。

```
class Solution:
    def monostoneStack(self, arr: List[int]) -> List[int]:
        stack = []
        ans = 定义一个长度和 arr 一样长的数组，并初始化为 -1
        循环 i in  arr:
            while stack and arr[i] > arr[栈顶元素]:
                peek = 弹出栈顶元素
                ans[peek] = i - peek
            stack.append(i)
        return ans
```

复杂度分析

- 时间复杂度：由于 arr 的元素最多只会入栈，出栈一次，因此时间复杂度仍然是 $O(N)$，其中 N 为数组长度。
- 空间复杂度：由于使用了栈， 并且栈的长度最大是和 arr 长度一致，因此空间复杂度是 $O(N)$，其中 N 为数组长度。

代码

这里提高两种编程语言的单调栈模板供大家参考。

Python3：

```python
class Solution:
    def monostoneStack(self, T: List[int]) -> List[int]:
        stack = []
        ans = [0] * len(T)
        for i in range(len(T)):
            while stack and T[i] > T[stack[-1]]:
                peek = stack.pop(-1)
                ans[peek] = i - peek
            stack.append(i)
        return ans
```

JS:

```javascript
var monostoneStack = function (T) {
  let stack = [];
  let result = [];
  for (let i = 0; i < T.length; i++) {
    result[i] = 0;
    while (stack.length > 0 && T[stack[stack.length - 1]] <
      let peek = stack.pop();
      result[peek] = i - peek;
    }
    stack.push(i);
  }
  return result;
};
```

题目推荐

下面几个题帮助你理解单调栈， 并让你明白什么时候可以用单调栈进行算法优化。

- 42. 接雨水
- 84. 柱状图中最大的矩形
- 739.每日温度

- 1. 去除重复字母
- 1. 移掉 K 位数字
- 1. 下一个更大元素 I
- 1. 最短无序连续子数组
-

1. 股票价格跨度

总结

单调栈本质就是栈， 栈本身就是一种受限的数据结构。其受限指的是只能在一端进行操作。而单调栈在栈的基础上进一步受限，即要求栈中的元素始终保持单调性。

由于栈中都是单调的，因此其天生适合解决**在其之后第一个小于其本身的位置**的题目。大家如果遇到题目需要找**在其之后第一个小于其本身的位置**的题目，就可是考虑使用单调栈。

单调栈的写法相对比较固定，大家可以自己参考我的伪代码自己总结一份模板，以后直接套用可以大大提高做题效率和容错率。

91 天学算法

很多教育机构宣传的 7 天，一个月搞定算法面试的，我大概都了解了下，不怎么靠谱。学习算法这东西，还是要考积累，没有量变是不可能有质变的。还有的人选择看书，这是一个不错的选择。但是很多人选了过时的或者质量差的书，又或者不会去写书中给的练习题，导致效果很差。

基于这几个原因，我组织了一个 91 天刷题活动，通过一个相对比较长的时间（91 天）给出最新的学习路径，并强制大家打卡这种高强度练习来让大家**在 91 天后遇见更好的自己**。详细活动介绍可以点下方链接查看。另外往期的讲义也在下面了，大家可以看看合不合你的口味。

最后送给大家一句话： **坚持下去，会有突然间成长的一天**。

- 91 天学算法第三期视频会议总结
- 第一期讲义-二分法
- 第一期讲义-双指针
- 第三期正在火热进行中

二分查找

二分查找又称 `折半搜索算法` 。 狭义地来讲，二分查找是一种在有序数组查找某一特定元素的搜索算法。这同时也是大多数人所知道的一种说法。实际上， 广义的二分查找是将问题的规模缩小到原有的一半。类似的，三分法就是将问题规模缩小为原来的 1/3。

本文给大家带来的内容则是 `狭义地二分查找` ，如果想了解其他广义上的二分查找可以查看我之前写的一篇博文 从老鼠试毒问题来看二分法

> 尽管二分查找的基本思想相对简单，但细节可以令人难以招架 ... — 高德纳

当乔恩·本特利将二分搜索问题布置给专业编程课的学生时，百分之 90 的学生在花费数小时后还是无法给出正确的解答，主要因为这些错误程序在面对边界值的时候无法运行，或返回错误结果。1988 年开展的一项研究显示，20 本教科书里只有 5 本正确实现了二分搜索。不仅如此，本特利自己 1986 年出版的《编程珠玑》一书中的二分搜索算法存在整数溢出的问题，二十多年来无人发现。Java 语言的库所实现的二分搜索算法中同样的溢出问题存在了九年多才被修复。

可见二分查找并不简单， 本文就试图带你走近 ta，明白 ta 的底层逻辑，并提供模板帮助大家写书 bug free 的二分查找代码。

大家可以看完讲义结合 LeetCode Book 二分查找练习一下

问题定义

给定一个由数字组成的有序数组 nums，并给你一个数字 target。问 nums 中是否存在 target。如果存在， 则返回其在 nums 中的索引。如果不存在，则返回 - 1。

这是二分查找中最简单的一种形式。当然二分查找也有很多的变形，这也是二分查找容易出错，难以掌握的原因。

常见变体有：

- 如果存在多个满足条件的元素，返回最左边满足条件的索引。
- 如果存在多个满足条件的元素，返回最右边满足条件的索引。
- 数组不是整体有序的。 比如先升序再降序，或者先降序再升序。
- 将一维数组变成二维数组。
- 。。。

接下来，我们逐个进行查看。

前提

- 数组是有序的（如果无序，我们也可以考虑排序，不过要注意排序的复杂度）

术语

二分查找中使用的术语：

- target —— 要查找的值
- index —— 当前位置
- l 和 r —— 左右指针
- mid —— 左右指针的中点，用来确定我们应该向左查找还是向右查找的索引

常见题型

查找一个数

算法描述：

- 先从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；
- 如果目标元素大于中间元素，则在数组大于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。
- 如果目标元素小于中间元素，则在数组小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。
- 如果在某一步骤数组为空，则代表找不到。

复杂度分析

- 平均时间复杂度： $O(logN)$
- 最坏时间复杂度： $O(logN)$
- 最优时间复杂度： $O(1)$
- 空间复杂度
  - 迭代: $O(1)$
  - 递归： $O(logN)$（无尾调用消除）

> 后面的复杂度也是类似的，不再赘述。

这种搜索算法每一次比较都使搜索范围缩小一半，是典型的二分查找。

这个是二分查找中最简答的一种类型了，我们先来搞定它。 我们来一个具体的例子， 这样方便大家增加代入感。假设 nums 为 `[1,3,4,6,7,8,10,13,14]`， target 为 4·。

- 刚开始数组中间的元素为 7
- 7 > 4 ，由于 7 右边的数字都大于 7 ，因此不可能是答案。我们将范围缩写到了 7 的左侧。

- 此时中间元素为 3
- 3 < 4，由于 3 左边的数字都小于 3 ，因此不可能是答案。我们将范围缩写到了 3 的右侧。
- 此时中间元素为 4，正好是我们要找的，返回其索引 2 即可。

如何将上面的算法转换为容易理解的可执行代码呢？就算是这样一个简简单单，朴实无华的二分查找， 不同的人写出来的差别也是很大的。 如果没有一个思维框架指导你，那么你在不同的时间可能会写出差异很大的代码。这样的话，你犯错的几率会大大增加。

这里给大家介绍一个我经常使用的思维框架和代码模板。

思维框架

首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点

> 你可以定义别的搜索区间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的搜索区间。

- 由于定义的搜索区间为 [left, right]，因此当 left <= right 的时候，搜索区间都不为空，此时我们都需要继续搜索。 也就是说终止搜索条件应该为 left <= right。

> 举个例子容易明白一点。 比如对于区间 [4,4]，其包含了一个元素 4，因此搜索区间不为空，需要继续搜索（试想 4 恰好是我们要找的 target，如果不继续搜索， 会错过正确答案）。而当搜索区间为 [left, right) 的时候，同样对于 [4,4]，这个时候搜索区间却是空的，因为这样的一个区间不存在任何数字·。

- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则提前返回 mid（只需要找到一个满足条件的即可）
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候搜索区间可缩小为 [mid + 1, right] （mid 以及 mid 左侧的数字被我们排除在外）
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候搜索区间可缩小为 [left, mid - 1] （mid 以及 mid 右侧的数字被我们排除在外）
- 循环结束都没有找到，则说明找不到，返回 -1 表示未找到。

代码模板

Java

```java
public int binarySearch(int[] nums, int target) {
    // 左右都闭合的区间 [l, r]
    int left = 0;
    int right = nums.length - 1;

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        if (nums[mid] < target)
                  // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        if (nums[mid] > target)
            // 搜索区间变为 [left, mid - 1]
            right = mid - 1;
    }
    return -1;
}
```

Python

```python
def binarySearch(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (left + right) >> 1
        if nums[mid] == target: return mid
        # 搜索区间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 搜索区间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    return -1
```

JavaScript

```
function binarySearch(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target) return mid;
    if (nums[mid] < target)
      // 搜索区间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 搜索区间变为 [left, mid - 1]
      right = mid - 1;
  }
  return -1;
}
```

C++

```
int binarySearch(vector<int>& nums, int target){
  if(nums.size() == 0)
    return -1;

  int left = 0, right = nums.size() - 1;
  while(left <= right){
    int mid = left + ((right - left) >> 1);
    if(nums[mid] == target){ return mid; }
    // 搜索区间变为 [mid+1, right]
    else if(nums[mid] < target)
    left = mid + 1;
    // 搜索区间变为 [left, mid - 1]
    else
    right = mid - 1;
  }
  return -1;
}
```

寻找最左边的满足条件的值

和 `查找一个数` 类似， 我们仍然套用 `查找一个数` 的思维框架和代码模板。

思维框架

- 首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点。
- 终止搜索条件为 left <= right。

- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则收缩右边界，我们找到了一个备胎，继续看看左边还有没有了（**注意这里不一样**）
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候搜索区间可缩小为 [mid + 1, right]
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候搜索区间可缩小为 [left, mid - 1]
- 由于不会提前返回，因此我们需要检查最终的 left，看 nums[left]是否等于 target。
  - 如果不等于 target，或者 left 出了右边边界了，说明至死都没有找到一个备胎，则返回 -1.
  - 否则返回 left 即可，备胎转正。

代码模板

> 实际上 nums[mid] > target 和 nums[mid] == target 是可以合并的。
> 我这里为了清晰，就没有合并，大家熟悉之后合并起来即可。

Java

```java
public int binarySearchLeft(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        }
        if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        }
        if (nums[mid] == target) {
            // 收缩右边界
            right = mid - 1;
        }
    }
    // 检查是否越界
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}
```

Python

```python
def binarySearchLeft(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) >> 1
        if nums[mid] == target:
            # 收缩右边界
            r = mid - 1;
        # 搜索区间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 搜索区间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    if l >= len(nums) or nums[l] != target: return -1
    return l
```

JavaScript

```javascript
function binarySearchLeft(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target)
      // 收缩右边界
      right = mid - 1;
    if (nums[mid] < target)
      // 搜索区间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 搜索区间变为 [left, mid - 1]
      right = mid - 1;
  }
  // 检查是否越界
  if (left >= nums.length || nums[left] != target) return -
  return left;
}
```

C++

```cpp
int binarySearchLeft(vector<int>& nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] == target) {
            // 收缩右边界
            right = mid - 1;
        }
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        }
        if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        }
    }
    // 检查是否越界
    if (left >= nums.size() || nums[left] != target)
        return -1;
    return left;
}
```

例题解析

给你一个严格递增的数组 nums ，让你找到第一个满足 nums[i] == i 的索引，如果没有这样的索引，返回 -1。（你的算法需要有 logN 的复杂度）。

首先我们做一个小小的变换，将原数组 nums 转换为 A，其中 A[i] = nums[i] - i。这样新的数组 A 就是一个不严格递增的数组。这样原问题转换为 在一个不严格递增的数组 A 中找第一个等于 0 的索引。接下来，我们就可以使用最左满足模板，找到最左满足 nums[i] == i 的索引。

代码：

```python
class Solution:
    def solve(self, nums):
        l, r = 0, len(nums) - 1
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] >= mid:
                r = mid - 1
            else:
                l = mid + 1
        return l if l < len(nums) and nums[l] == l else -1
```

寻找最右边的满足条件的值

和 `查找一个数` 类似， 我们仍然套用 `查找一个数` 的思维框架和代码模板。

> 有没有感受到框架和模板的力量?

思维框架

- 首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点。

> 你可以定义别的搜索区间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的搜索区间。

- 由于我们定义的搜索区间为 [left, right]，因此当 left <= right 的时候，搜索区间都不为空。 也就是说我们的终止搜索条件为 left <= right。

> 举个例子容易明白一点。 比如对于区间 [4,4]，其包含了一个元素 4，因此搜索区间不为空。而当搜索区间为 [left, right) 的时候，同样对于 [4,4]，这个时候搜索区间却是空的。

- 循环体内，我们不断计算 mid ，并将 nums[mid] 与 目标值比对。
  - 如果 nums[mid] 等于目标值， 则收缩左边界，我们找到了一个备胎，继续看看右边还有没有了
  - 如果 nums[mid] 小于目标值， 说明目标值在 mid 右侧，这个时候搜索区间可缩小为 [mid + 1, right]
  - 如果 nums[mid] 大于目标值， 说明目标值在 mid 左侧，这个时候搜索区间可缩小为 [left, mid - 1]
- 由于不会提前返回，因此我们需要检查最终的 right，看 nums[right] 是否等于 target。
  - 如果不等于 target，或者 right 出了左边边界了，说明至死都没有找到一个备胎，则返回 -1.
  - 否则返回 right 即可，备胎转正。

代码模板

> 实际上 nums[mid] < target 和 nums[mid] == target 是可以合并的。我这里为了清晰，就没有合并，大家熟悉之后合并起来即可。

Java

```java
public int binarySearchRight(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        }
        if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        }
        if (nums[mid] == target) {
            // 收缩左边界
            left = mid + 1;
        }
    }
    // 检查是否越界
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

Python

```python
def binarySearchRight(nums, target):
    # 左右都闭合的区间 [l, r]
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) >> 1
        if nums[mid] == target:
            # 收缩左边界
            l = mid + 1;
        # 搜索区间变为 [mid+1, right]
        if nums[mid] < target: l = mid + 1
        # 搜索区间变为 [left, mid - 1]
        if nums[mid] > target: r = mid - 1
    if r < 0 or nums[r] != target: return -1
    return r
```

JavaScript

```javascript
function binarySearchRight(nums, target) {
  let left = 0;
  let right = nums.length - 1;
  while (left <= right) {
    const mid = Math.floor(left + (right - left) / 2);
    if (nums[mid] == target)
      // 收缩左边界
      left = mid + 1;
    if (nums[mid] < target)
      // 搜索区间变为 [mid+1, right]
      left = mid + 1;
    if (nums[mid] > target)
      // 搜索区间变为 [left, mid - 1]
      right = mid - 1;
  }
  // 检查是否越界
  if (right < 0 || nums[right] != target) return -1;
  return right;
}
```

C++

```cpp
int binarySearchRight(vector<int>& nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.size() - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
         if (nums[mid] == target) {
            // 收缩左边界
            left = mid + 1;
        }
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        }
        if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        }
    }
    // 检查是否越界
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

寻找最左插入位置

上面我们讲了 `寻找最左满足条件的值` 。如果找不到，就返回 -1。那如果我想让你找不到不是返回 -1，而是应该插入的位置，使得插入之后列表仍然有序呢?

比如一个数组 nums: [1,3,4]，target 是 2。我们应该将其插入（注意不是真的插入）的位置是索引 1 的位置，即 [1,**2**,3,4]。因此 `寻找最左插入位置` 应该返回 1，而 `寻找最左满足条件` 应该返回-1。

另外如果有多个满足条件的值，我们返回最左侧的。 比如一个数组 nums: [1,2,2,2,3,4]，target 是 2，我们应该插入的位置是 1。

思维框架

如果你将**寻找最左插入位置**看成是**寻找最左满足**大于等于 x 的值，那就可以和前面的知识产生联系，使得代码更加统一。唯一的区别点在于**前面是最左满足等于 x**，这里是**最左满足大于等于 x**。

具体算法：

- 首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点。

> 你可以定义别的搜索区间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的搜索区间。

- 由于我们定义的搜索区间为 [left, right]，因此当 left <= right 的时候，搜索区间都不为空。 也就是说我们的终止搜索条件为 left <= right。
- 当 A[mid] >= x，说明找到一个备胎，我们令 r = mid - 1 将 mid 从搜索区间排除，继续看看有没有更好的备胎。
- 当 A[mid] < x，说明 mid 根本就不是答案，直接更新 l = mid + 1，从而将 mid 从搜索区间排除。
- 最后搜索区间的 l 就是最好的备胎，备胎转正。

代码模板

Python

```
def bisect_left(nums, x):
    # 内置 api
    bisect.bisect_left(nums, x)
    # 手写
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if A[mid] >= x: r = mid - 1
        else: l = mid + 1
    return l
```

其他语言暂时空缺，欢迎 PR

寻找最右插入位置

思维框架

如果你将**寻找最右插入位置**看成是**寻找最右满足**大于 x 的值，那就可以和前面的知识产生联系，使得代码更加统一。唯一的区别点在于**前面是最左满足等于 x**，这里是**最左满足大于 x**。

具体算法：

- 首先定义搜索区间为 [left, right]，注意是左右都闭合，之后会用到这个点。

> 你可以定义别的搜索区间形式，不过后面的代码也相应要调整，感兴趣的可以试试别的搜索区间。

- 由于我们定义的搜索区间为 [left, right]，因此当 left <= right 的时候，搜索区间都不为空。 也就是说我们的终止搜索条件为 left <= right。
- 当 A[mid] > x，说明找到一个备胎，我们令 r = mid - 1 将 mid 从搜索区间排除，继续看看有没有更好的备胎。
- 当 A[mid] <= x，说明 mid 根本就不是答案，直接更新 l = mid + 1，从而将 mid 从搜索区间排除。
- 最后搜索区间的 l 就是最好的备胎，备胎转正。

代码模板

Python

```
def bisect_right(nums, x):
    # 内置 api
    bisect.bisect_right(nums, x)
    # 手写
    l, r = 0, len(A) - 1
    while l <= r:
        mid = (l + r) // 2
        if A[mid] <= x: l = mid + 1
        else: r = mid - 1
    return l
```

其他语言暂时空缺，欢迎 PR

局部有序（先降后升或先升后降）

LeetCode 有原题 33. 搜索旋转排序数组 和 81. 搜索旋转排序数组 II， 我们直接拿过来讲解好了。

其中 81 题是在 33 题的基础上增加了 `包含重复元素` 的可能，实际上 33 题的进阶就是 81 题。通过这道题，大家可以感受到”包含重复与否对我们算法的影响“。 我们直接上最复杂的 81 题，这个会了，可以直接 AC 第 33 题。

81. 搜索旋转排序数组 II

题目描述

```
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

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？
```

思路

这是一个我在网上看到的前端头条技术终面的一个算法题。我们先不考虑重复元素。

题目要求时间复杂度为 logn，因此基本就是二分法了。 这道题目不是直接的有序数组，不然就是 easy 了。

首先要知道，我们随便选择一个点，将数组分为前后两部分，其中一部分一定是有序的。

具体步骤：

- 我们可以先找出 mid，然后根据 mid 来判断，mid 是在有序的部分还是无序的部分

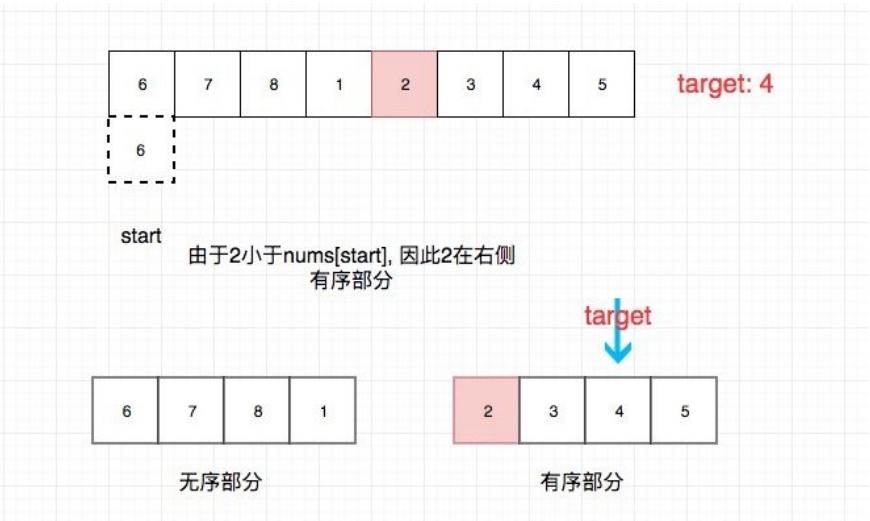
假如 mid 小于 start，则 mid 一定在右边有序部分，即 [mid,end] 部分有序。假如 mid 大于 start，则 mid 一定在左边有序部分，即 [start,mid]部分有序。**这是这类题目的突破口。**

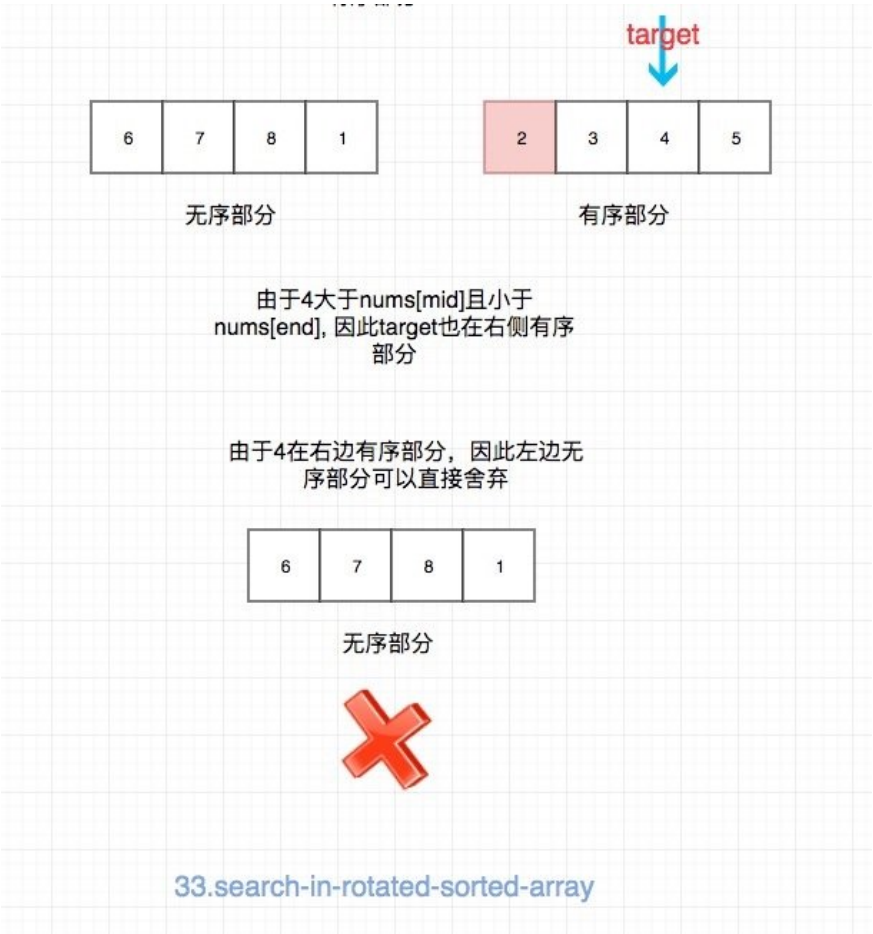
> 注意我没有考虑等号，之后我会讲。

- 然后我们继续判断 target 在哪一部分， 就可以舍弃另一部分了。

也就是说只需要比较 target 和**有序部分**的边界关系就行了。 比如 mid 在右侧有序部分，即[mid,end] 有序。那么我们只需要判断 target >= mid && target <= end 就能知道 target 在右侧有序部分，我们就 可以舍弃左边部分了(通过 start = mid + 1 实现)， 反之亦然。

我们以([6,7,8,1,2,3,4,5], 4)为例讲解一下：





接下来，我们考虑重复元素的问题。如果存在重复数字，就可能会发生 nums[mid] == nums[start] 了，比如 30333 。这个时候可以选择舍弃 start，也就是 start 右移一位。有的同学会担心”会不会错失目标元素？“。其实这个担心是多余的，前面我们已经介绍了”搜索区间“。由于搜索区间同时包含 start 和 mid ，因此去除一个 start ，我们还有 mid。假如 3 是我们要找的元素， 这样进行下去绝对不会错过，而是收缩”搜索区间“到一个元素 3 ，我们就可以心安理得地返回 3 了。

代码（Python）

```python
class Solution:
    def search(self, nums, target):
        l, r = 0, len(nums)-1
        while l <= r:
            mid = l + (r-l)//2
            if nums[mid] == target:
                return True
            while l < mid and nums[l] == nums[mid]:  # tri
                l += 1
            # the first half is ordered
            if nums[l] <= nums[mid]:
                # target is in the first half
                if nums[l] <= target < nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
            # the second half is ordered
            else:
                # target is in the second half
                if nums[mid] < target <= nums[r]:
                    l = mid + 1
                else:
                    r = mid - 1
        return False
```

复杂度分析

- 时间复杂度：$O(log N)$
- 空间复杂度：$O(1)$

扩展

如果题目不是让你返回 true 和 false，而是返回最左/最右等于 targrt 的索引呢？这不就又和前面的知识建立联系了么？比如我让你在一个旋转数组中找最左等于 target 的索引，其实就是 面试题 10.03. 搜索旋转数组。

思路和前面的最左满足类似，仍然是通过压缩区间，更新备胎，最后返回备胎的方式来实现。 具体看代码吧。

Python Code:

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            mid = l + (r - l) // 2
            # # the first half is ordered
            if nums[l] < nums[mid]:
                # target is in the first half
                if nums[l] <= target <= nums[mid]:
                    r = mid - 1
                else:
                    l = mid + 1
            # # the second half is ordered
            elif nums[l] > nums[mid]:
                # target is in the second half
                if nums[l] <= target or target <= nums[mid]
                    r = mid - 1
                else:
                    l = mid + 1
            elif nums[l] == nums[mid]:
                if nums[l] != target:
                    l += 1
                else:
                    # l 是一个备胎
                    r = l - 1
        return l if l < len(nums) and nums[l] == target els
```

二维数组

二维数组的二分查找和一维没有本质区别， 我们通过两个题来进行说明。

74. 搜索二维矩阵

题目地址

https://leetcode-cn.com/problems/search-a-2d-matrix/

题目描述

```
编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有

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
```

思路

简单来说就是将一个一维有序数组切成若干长度相同的段，然后将这些段拼接成一个二维数组。你的任务就是在这个拼接成的二维数组中找到target。

需要注意的是，数组是不存在重复元素的。

> 如果有重复元素，我们该怎么办?

算法：

- 选择矩阵左下角作为起始元素 Q
- 如果 Q > target，右方和下方的元素没有必要看了（相对于一维数组的右边元素）
- 如果 Q < target，左方和上方的元素没有必要看了（相对于一维数组的左边元素）
- 如果 Q == target ，直接 返回 True
- 交回了都找不到，返回 False

代码(Python)

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target:
        m = len(matrix)
        if m == 0:
            return False
        n = len(matrix[0])

        x = m - 1
        y = 0
        while x >= 0 and y < n:
            if matrix[x][y] > target:
                x -= 1
            elif matrix[x][y] < target:
                y += 1
            else:
                return True
        return False
```

复杂度分析

- 时间复杂度：最坏的情况是只有一行或者只有一列，此时时间复杂度为 $O(M * N)$。更多的情况下时间复杂度为 $O(M + N)$
- 空间复杂度：$O(1)$

力扣 240. 搜索二维矩阵 II 发生了一点变化，不再是 `每行的第一个整数大于前一行的最后一个整数` ，而是 `每列的元素从上到下升序排列` 。我们仍然可以选择左下进行二分。

寻找最值(改进的二分)

上面全部都是找到给定值，这次我们试图寻找最值（最小或者最大）。我们以最小为例，讲解一下这种题如何切入。

153. 寻找旋转排序数组中的最小值

题目地址

https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/

题目描述

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
```

二分法

思路

和查找指定值得思路一样。我们还是：

- 初始化首尾指针 l 和 r
- 如果 nums[mid] 大于 nums[r]，说明 mid 在左侧有序部分，由于最小的一定在右侧，因此可以收缩左区间，即 l = mid + 1
- 否则收缩右侧，即 r = mid（不可以 r = mid - 1）

> 这里多判断等号没有意义，因为题目没有让我们找指定值

- 当 l >= r 或者 nums[l] < nums[r] 的时候退出循环

> nums[l] < nums[r]，说明区间 [l, r] 已经是整体有序了，因此 nums[l] 就是我们想要找的

代码（Python）

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1

        while l < r:
            # important
            if nums[l] < nums[r]:
                return nums[l]
            mid = (l + r) // 2
            # left part
            if nums[mid] > nums[r]:
                l = mid + 1
            else:
                # right part
                r = mid
        # l or r is not important
        return nums[l]
```

复杂度分析

- 时间复杂度：$O(log N)$
- 空间复杂度：$O(1)$

另一种二分法

思路

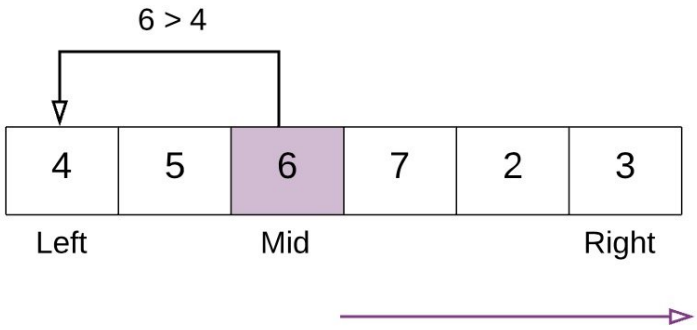
我们当然也也可以和 nums[l] 比较，而不是上面的 nums[r]，我们发现：

- 旋转点左侧元素**都大于**数组第一个元素
- 旋转点右侧元素**都小于**数组第一个元素

这样就建立了 nums[mid] 和 nums[0] 的联系。

具体算法：

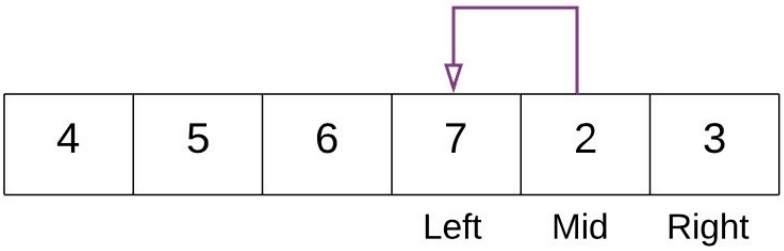
1. 找到数组的中间元素 mid。
2. 如果中间元素 > 数组第一个元素，我们需要在 mid 右边搜索。



- 如果中间元素 <= 数组第一个元素，我们需要在 mid 左边搜索。

上面的例子中，中间元素 6 比第一个元素 4 大，因此在中间点右侧继续搜索。

1. 当我们找到旋转点时停止搜索，当以下条件满足任意一个即可：
2. nums[mid] > nums[mid + 1]，因此 mid+1 是最小值。
3. nums[mid - 1] > nums[mid]，因此 mid 是最小值。



代码（Python）

```
class Solution:
    def findMin(self, nums):
        # If the list has just one element then return that
        if len(nums) == 1:
            return nums[0]

        # left pointer
        left = 0
        # right pointer
        right = len(nums) - 1

        # if the last element is greater than the first ele
        # e.g. 1 < 2 < 3 < 4 < 5 < 7. Already sorted array.
        # Hence the smallest element is first element. A[0]
        if nums[right] > nums[0]:
            return nums[0]

        # Binary search way
        while right >= left:
            # Find the mid element
            mid = left + (right - left) / 2
            # if the mid element is greater than its next e
            # This point would be the point of change. From
            if nums[mid] > nums[mid + 1]:
                return nums[mid + 1]
            # if the mid element is lesser than its previou
            if nums[mid - 1] > nums[mid]:
                return nums[mid]

            # if the mid elements value is greater than the
            # the least value is still somewhere to the rig
            if nums[mid] > nums[0]:
                left = mid + 1
            # if nums[0] is greater than the mid value then
            else:
                right = mid - 1
```

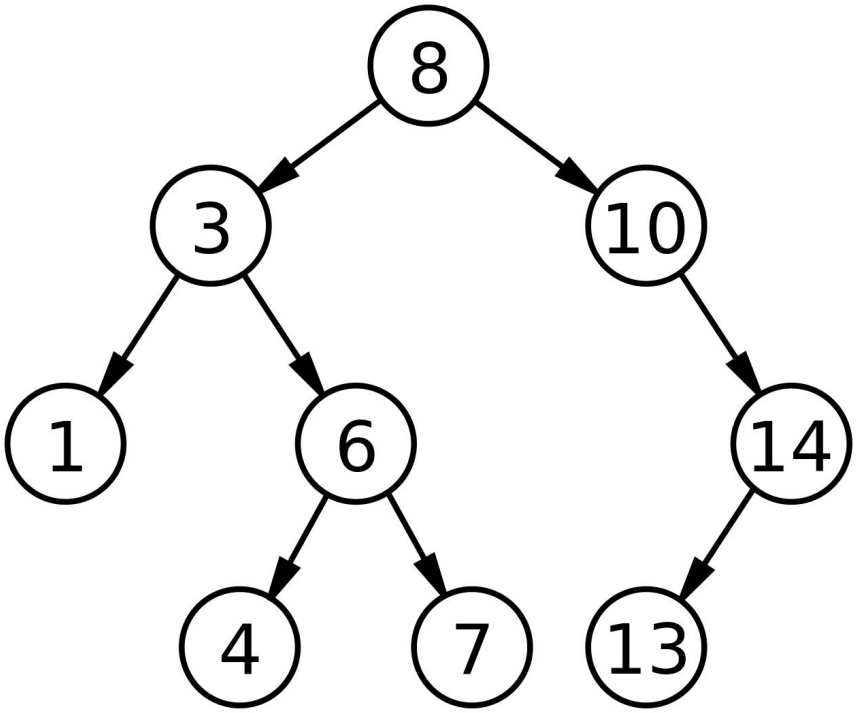
复杂度分析

- 时间复杂度：$O(log N)$
- 空间复杂度：$O(1)$

二叉树

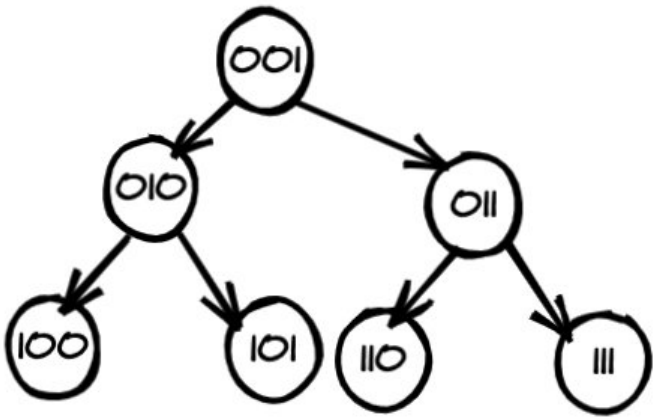
对于一个给定的二叉树，其任意节点最多只有两个子节点。 从这个定义，我们似乎可以嗅出一点二分法的味道， 但是这并不是二分。但是，二叉树中却和二分有很多联系，我们来看一下。

最简单的，如果这个二叉树是一个二叉搜索树（BST）。 那么实际上，在一个二叉搜索树种进行搜索的过程就是二分法。

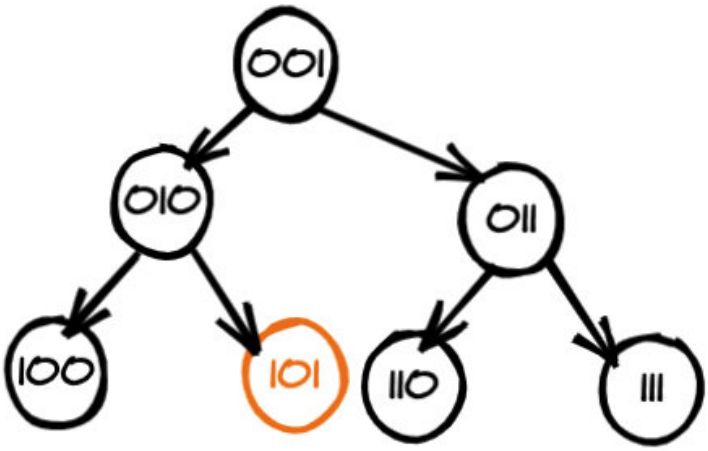


如上图，我们需要在这样一个二叉搜索树中搜索 7。那么我们的搜索路径则会是 8 -> 3 -> 6 -> 7，这也是一种二分法。只不过相比于普通的**有序序列查找给定值**二分， 其时间复杂度的下界更差，原因在于二叉搜索树并不一定是二叉平衡树。

上面讲了二叉搜索树，我们再来看一种同样特殊的树 - 完全二叉树。 如果我们给一颗完全二叉树的所有节点进行编号（二进制），依次为 01,10,11, ...。



那么实际上，最后一行的编号就是从根节点到该节点的路径。 其中 0 表示向左， 1 表示向右。(第一位数字不用)。 我们以最后一行的 101 为例，我们需要执行一次左，然后一次右。



其实原理也不难，如果你用数组表示过完全二叉树，那么就很容易理解。我们可以发现， 父节点的编号都是左节点的二倍，并且都是右节点的二倍 + 1。从二进制的角度来看就是：**父节点的编号左移一位就是左节点的编号，左移一位 + 1 就是右节点的编号**。 因此反过来， 知道了子节点的最后一位，我们就能知道它是父节点的左节点还是右节点啦。

题目推荐

- 875. 爱吃香蕉的珂珂
- 300. 最长上升子序列
- 354. 俄罗斯套娃信封问题
- 面试题 17.08. 马戏团人塔

> 后面三个题建议一起做

总结

二分查找是一种非常重要且难以掌握的核心算法，大家一定要好好领会。有的题目直接二分就可以了，有的题目二分只是其中一个环节。不管是哪种，都需要我们对二分的思想和代码模板非常熟悉才可以。

二分查找的基本题型有：

- 查找满足条件的元素，返回对应索引
- 如果存在多个满足条件的元素，返回最左边满足条件的索引。
- 如果存在多个满足条件的元素，返回最右边满足条件的索引。
- 数组不是整体有序的。 比如先升序再降序，或者先降序再升序。
- 将一维数组变成二维数组。
- 局部有序查找最大（最小）元素
- 。。。

不管是哪一种类型，我们的思维框架都是类似的，都是：

- 先定义**搜索区间**（非常重要）

- 根据搜索区间定义循环结束条件
- 取中间元素和目标元素做对比（目标元素可能是需要找的元素或者是数组第一个，最后一个元素等）（非常重要）
- 根据比较的结果收缩区间，舍弃非法解（也就是二分）

> 如果是整体有序通常只需要 nums[mid] 和 target 比较即可。如果是局部有序，则可能需要与其周围的特定元素进行比较。

大家可以使用这个思维框架并结合本文介绍的几种题型进行练习，必要的情况可以使用我提供的解题模板，提供解题速度的同时，有效地降低出错的概率。

特别需要注意的是**有无重复元素对二分算法影响很大**，我们需要小心对待。

【91算法-基础篇】05.双指针

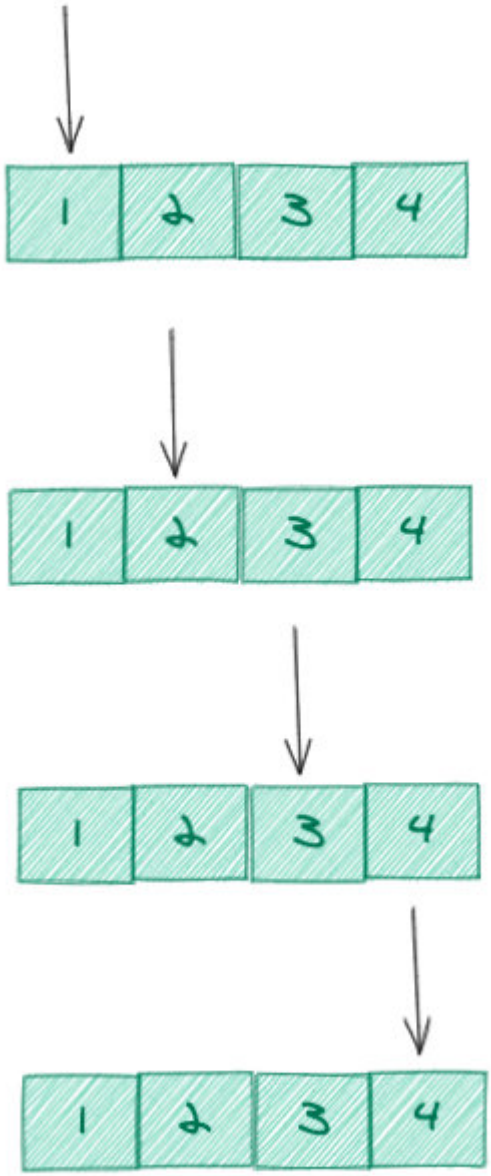
力扣加加，一个努力做西湖区最好的算法题解的团队。就在今天它给大家带来了《91 天学算法》，帮助大家摆脱困境，征服算法。

什么是双指针

顾名思议，双指针就是**两个指针**，但是不同于 C，C++中的指针， 其是一种**算法思想**。

如果说，我们迭代一个数组，并输出数组每一项，我们需要一个指针来记录当前遍历的项，这个过程我们叫单指针（index）的话。

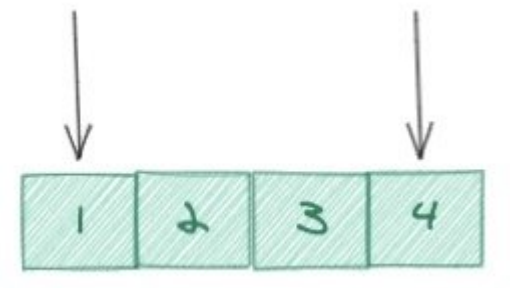
```
for(int i = 0;i < nums.size(); i++) {
  输出(nums[i]);
}
```



（图 1）

那么双指针实际上就是有两个这样的指针，最为经典的就是二分法中的左右双指针啦。

```
int l = 0;
int r = nums.size() - 1;

while (l < r) {
    if(一定条件) return 合适的值，一般是 l 和 r 的中点
    if(一定条件) l++
    if(一定条件) r--
}
// 因为 l == r，因此返回 l 和 r 都是一样的
return l
```



（图 2）

读到这里，你发现双指针是一个很宽泛的概念，就好像数组，链表一样，其类型会有很多很多， 比如二分法经常用到 `左右端点双指针` 。滑动窗口会用到 `快慢指针和固定间距指针` 。 因此双指针其实是一种综合性很强的类型，类似于数组，栈等。 但是我们这里所讲述的双指针，往往指的是某几种类型的双指针，而不是“只要有两个指针就是双指针了”。

> 有了这样一个**算法框架**，或者算法思维，有很大的好处。它能帮助你理清思路，当你碰到新的问题，在脑海里进行搜索的时候，**双指针**这个词就会在你脑海里闪过，闪过的同时你可以根据**双指针**的所有套路和这道题进行**穷举匹配**，这个思考解题过程本来就像是算法，我会在进阶篇《搜索算法》中详细阐述。

那么究竟我们算法中提到的双指针指的是什么呢？我们一起来看下算法中双指针的常见题型吧。

常见题型有哪些?

这里我将其分为三种类类型，分别是：

1. 快慢指针（两个指针步长不同）
2. 左右端点指针（两个指针分别指向头尾，并往中间移动，步长不确定）
3. 固定间距指针（两个指针间距相同，步长相同）

> 上面是我自己的分类，没有参考别人。可以发现我的分类标准已经覆盖了几乎所有常见的情况。 大家在平时做题的时候一定要养成这样的习惯，将题目类型进行总结，当然这个总结可以是别人总结好的，也可以是自己独立总结的。不管是哪一种，都要进行一定的消化吸收，把它们变成真正属于自己的知识。

不管是哪一种双指针，只考虑双指针部分的话，由于最多还是会遍历整个数组一次，因此时间复杂度取决于步长，如果步长是 1，2 这种常数的话，那么时间复杂度就是 O(N)，如果步长是和数据规模有关（比如二分法），其时间复杂度就是 O(logN)。并且由于不管规模多大，我们都只需要最多两个指针，因此空间复杂度是 O(1)。下面我们就来看看双指针的常见套路有哪些。

常见套路

快慢指针

1. 判断链表是否有环

这里给大家推荐两个非常经典的题目，一个是力扣 287 题，一个是 142 题。其中 142 题我在我的 LeetCode 题解仓库中的每日一题板块出过，并且给了很详细的证明和解答。而 287 题相对不直观，比较难以想到，这道题曾被官方选定为每日一题，也是相当经典的。

- 287. 寻找重复数
- 【每日一题】- 2020-01-14 - 142. 环形链表 II · Issue #274 · azl397985856/leetcode
- 读写指针。典型的是 `删除重复元素`

这里推荐我仓库中的一道题， 我这里写了一个题解，横向对比了几个相似题目，并剖析了这种题目的本质是什么，让你看透题目本质，推荐阅读。

- 80.删除排序数组中的重复项 II

左右端点指针

1. 二分查找。

二分查找会在专题篇展开，这里不多说，大家先知道就行了。

1. 暴力枚举中“从大到小枚举”（剪枝）

一个典型的题目是我之前参加官方每日一题的时候给的一个解法，大家可以看下。这种解法是可以 AC 的。同样地，这道题我也给出了三种方法，帮助大家从多个纬度看清这个题目。强烈推荐大家做到一题多解。这对于

你做题很多帮助。除了一题多解，还有一个大招是多题同解，这部分我们放在专题篇介绍。

find-the-longest-substring-containing-vowels-in-even

1. 有序数组。

区别于上面的二分查找，这种算法指针移动是连续的，而不是跳跃性的，典型的是 LeetCode 的 `两数和` ，以及 `N数和` 系列问题。

固定间距指针

1. 一次遍历（One Pass）求链表的中点
2. 一次遍历（One Pass）求链表的倒数第 k 个元素
3. 固定窗口大小的滑动窗口

模板(伪代码)

我们来看下上面三种题目的算法框架是什么样的。这个时候我们没必要纠结具体的语言，这里我直接使用了伪代码，就是防止你掉进细节。

当你掌握了这种算法的细节，就应该找几个题目试试。一方面是检测自己是否真的掌握了，另一方面是“细节”，”细节“是人类，尤其是软件工程师最大的敌人，毕竟我们都是 `差不多先生` 。

1. 快慢指针

```
l = 0
r = 0
while 没有遍历完
  if 一定条件
    l += 1
  r += 1
return 合适的值
```

1. 左右端点指针

```
l = 0
r = n - 1
while l < r
  if 找到了
    return 找到的值
  if 一定条件1
    l += 1
  else if  一定条件2
    r -= 1
return 没找到
```

1. 固定间距指针

```
l = 0
r = k
while 没有遍历完
  自定义逻辑
  l += 1
  r += 1
return 合适的值
```

题目推荐

如果你 `差不多` 理解了上面的东西，那么可以拿下面的题练练手。Let's Go!

左右端点指针

- 16.3Sum Closest (Medium)
- 713.Subarray Product Less Than K (Medium)
- 977.Squares of a Sorted Array (Easy)
- Dutch National Flag Problem

> 下面是二分类型

- 33.Search in Rotated Sorted Array (Medium)
- 875.Koko Eating Bananas（Medium）
- 881.Boats to Save People（Medium）

快慢指针

- 26.Remove Duplicates from Sorted Array（Easy）
- 141.Linked List Cycle (Easy)
- 142.Linked List Cycle II（Medium）

- 287.Find the Duplicate Number（Medium）
- 202.Happy Number (Easy)

固定间距指针

- 1456.Maximum Number of Vowels in a Substring of Given Length（Medium）

> 固定窗口大小的滑动窗口见专题篇的滑动窗口专题（暂未发布）

其他

有时候也不能太思维定式，比如 https://leetcode-cn.com/problems/consecutive-characters/ 这道题根本就没必要双指针什么的。

再比如：https://lucifer.ren/blog/2020/05/31/101.symmetric-tree/

精选题解

这里是我以前写的题解。 这里的题解一般都是多个题目，而不是针对某一具体题目的。熟悉我的朋友应该知道了， 这是我所说的**第二阶段** 。

第一阶段按照 tag 去刷， 第二阶段则要一题多解，多题同解，挖掘题目背后的东西。而这个系列大多数就是做了这个事情。其他不是**一题多解，多题同解**的，基本就是**大厂真题解析**。

- 《日程安排》专题
- 《构造二叉树》专题
- 字典序列删除
- 百度的算法面试题 - 祖玛游戏
- 西法的刷题秘籍】一次搞定前缀和
- 字节跳动的算法面试题是什么难度?
- 字节跳动的算法面试题是什么难度？（第二弹）
- 《我是你的妈妈呀》 - 第一期
- 一文带你看懂二叉树的序列化
- 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 你的衣服我扒了 - 《最长公共子序列》
- 一文看懂《最大子序列和问题》
- 状压 DP 入门
- 一行代码就可以 AC

一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～

我花了几天时间，从力扣中精选了四道相同思想的题目，来帮助大家解套，如果觉得文章对你有用，记得点赞分享，让我看到你的认可，有动力继续做下去。

这就是接下来要给大家讲的四个题，其中 1081 和 316 题只是换了说法而已。

- 316. 去除重复字母(困难)
- 321. 拼接最大数(困难)
- 402. 移掉 K 位数字(中等)
- 1081. 不同字符的最小子序列（中等）

402. 移掉 K 位数字（中等）

我们从一个简单的问题入手，识别一下这种题的基本形式和套路，为之后的三道题打基础。

题目描述

```
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

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。
```

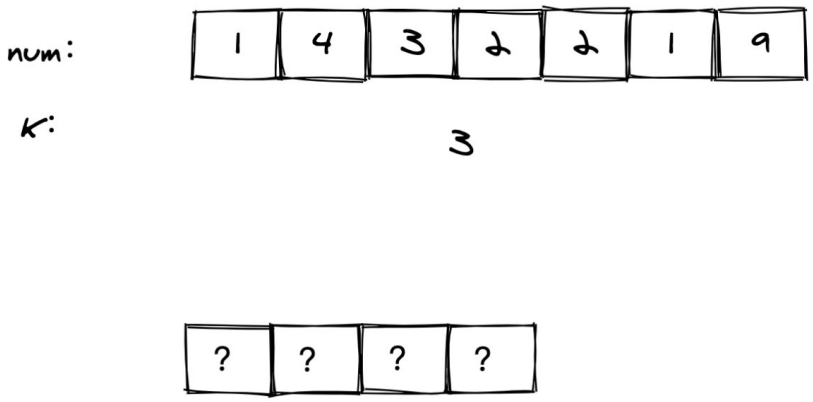
前置知识

- 数学

思路

这道题让我们从一个字符串数字中删除 k 个数字，使得剩下的数最小。也就说，我们要保持原来的数字的相对位置不变。

以题目中的 `num = 1432219, k = 3` 为例，我们需要返回一个长度为 4 的字符串，问题在于： 我们怎么才能求出这四个位置依次是什么呢?



（图 1）

暴力法的话，我们需要枚举 `C_n^(n - k)` 种序列（其中 n 为数字长度），并逐个比较最大。这个时间复杂度是指数级别的，必须进行优化。

一个思路是：

- 从左到右遍历
- 对于每一个遍历到的元素，我们决定是**丢弃**还是**保留**

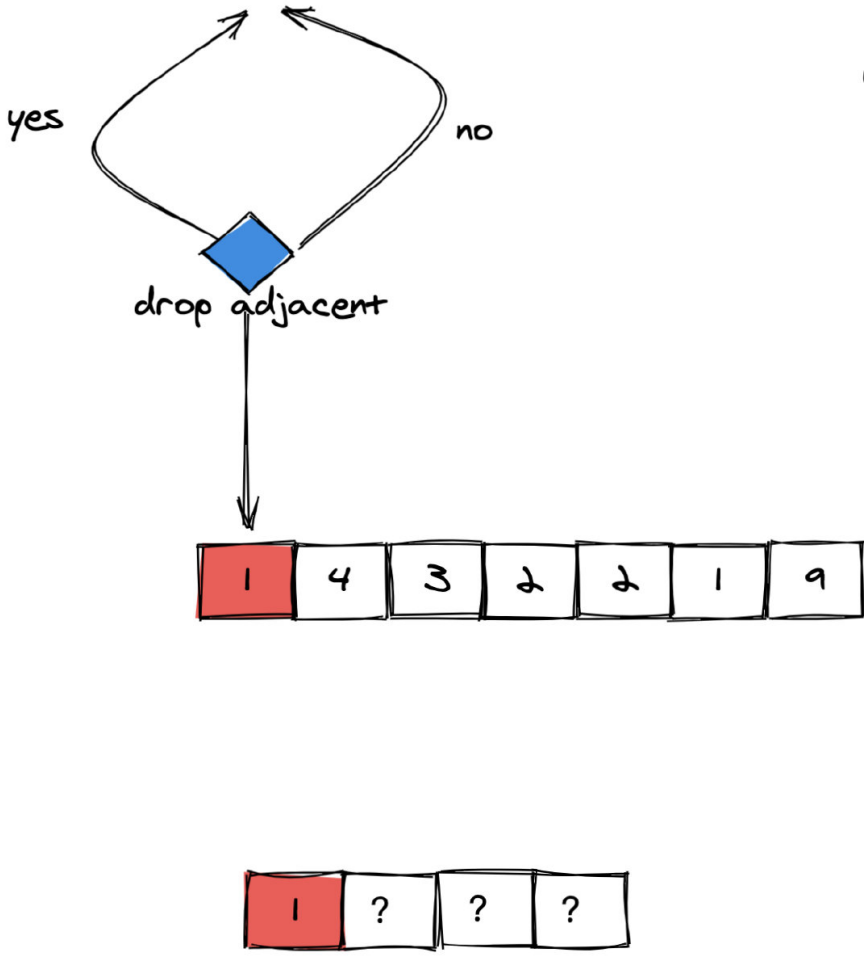
问题的关键是：我们怎么知道，一个元素是应该保留还是丢弃呢？

这里有一个前置知识：**对于两个数 123a456 和 123b456，如果 a > b，那么数字 123a456 大于 数字 123b456，否则数字 123a456 小于等于数字 123b456**。也就说，两个**相同位数**的数字大小关系取决于第一个不同的数的大小。

因此我们的思路就是：

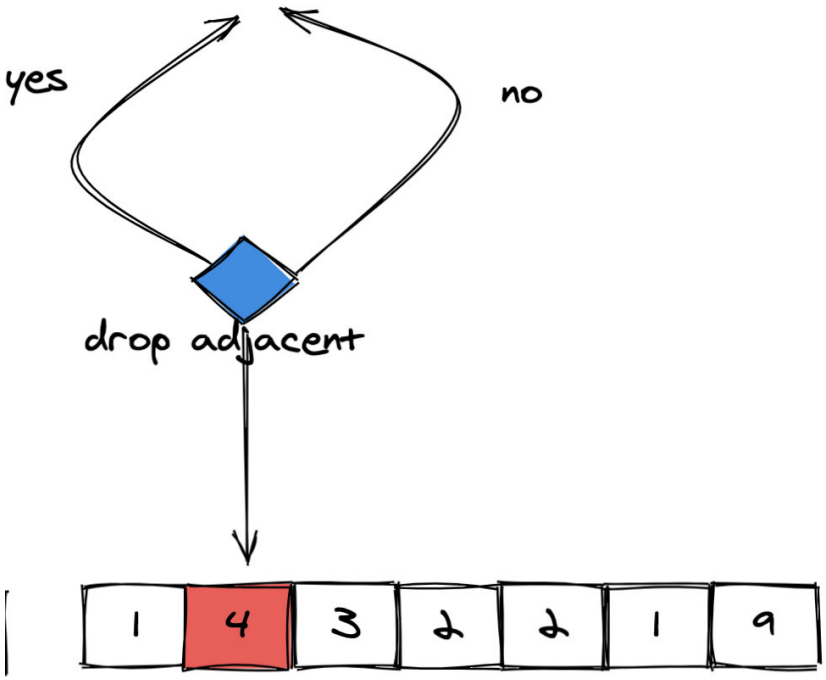
- 从左到右遍历
- 对于遍历到的元素，我们选择保留。
- 但是我们可以选择性丢弃前面相邻的元素。
- 丢弃与否的依据如上面的前置知识中阐述中的方法。

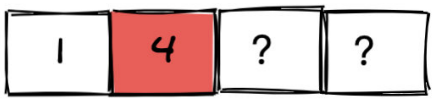
以题目中的 `num = 1432219, k = 3` 为例的图解过程如下：



（图 2）

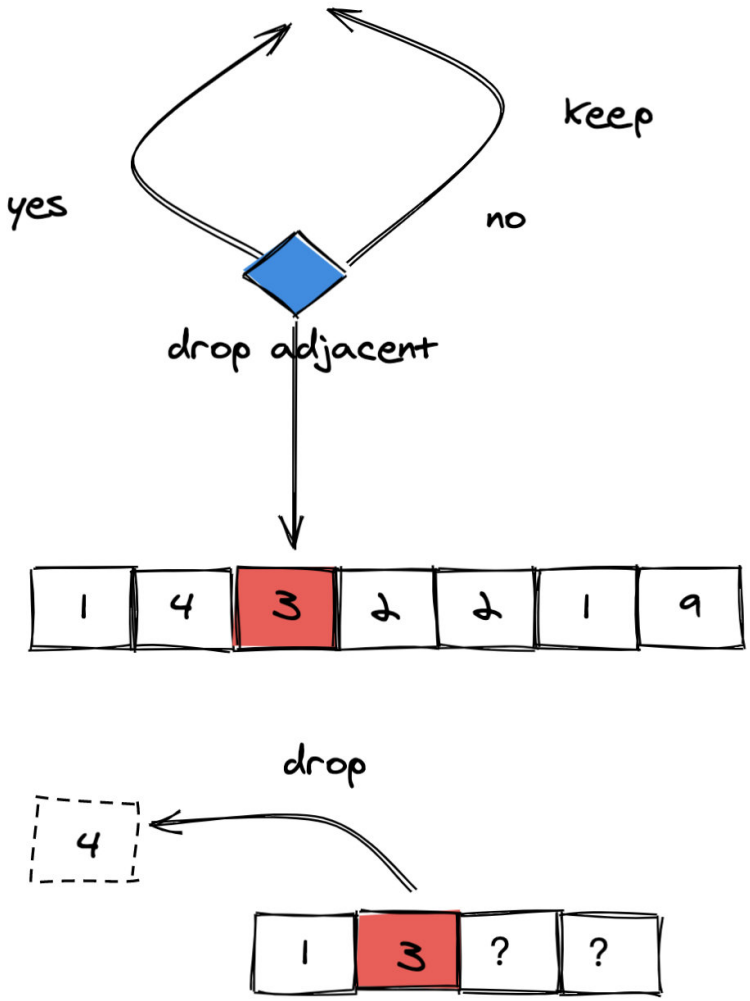
由于没有左侧相邻元素，因此**没办法丢弃**。





（图 3）

由于 4 比左侧相邻的 1 大。如果选择丢弃左侧的 1，那么会使得剩下的数字更大（开头的数从 1 变成了 4）。因此我们仍然选择**不丢弃**。



（图 4）

由于 3 比左侧相邻的 4 小。 如果选择丢弃左侧的 4，那么会使得剩下的数字更小（开头的数从 4 变成了 3）。因此我们选择**丢弃**。

。。。

后面的思路类似，我就不继续分析啦。

然而需要注意的是，如果给定的数字是一个单调递增的数字，那么我们的算法会永远**选择不丢弃**。这个题目中要求的，我们要永远确保**丢弃** k 个矛盾。

一个简单的思路就是：

- 每次丢弃一次，k 减去 1。当 k 减到 0 ，我们可以提前终止遍历。
- 而当遍历完成，如果 k 仍然大于 0。不妨假设最终还剩下 x 个需要丢弃，那么我们需要选择删除末尾 x 个元素。

上面的思路可行，但是稍显复杂。

（图 5）

我们需要把思路逆转过来。刚才我的关注点一直是**丢弃**，题目要求我们丢弃 k 个。反过来说，不就是让我们保留 $n - k$ 个元素么？其中 n 为数字长度。 那么我们只需要按照上面的方法遍历完成之后，再截取前**n - k**个元素即可。

按照上面的思路，我们来选择数据结构。由于我们需要**保留**和**丢弃相邻**的元素，因此使用栈这种在一端进行添加和删除的数据结构是再合适不过了，我们来看下代码实现。

代码（Python）

```
class Solution(object):
    def removeKdigits(self, num, k):
        stack = []
        remain = len(num) - k
        for digit in num:
            while k and stack and stack[-1] > digit:
                stack.pop()
                k -= 1
            stack.append(digit)
        return ''.join(stack[:remain]).lstrip('0') or '0'
```

复杂度分析

- 时间复杂度：虽然内层还有一个 while 循环，但是由于每个数字最多仅会入栈出栈一次，因此时间复杂度仍然为 $O(N)$，其中 N 为数字长度。
- 空间复杂度：我们使用了额外的栈来存储数字，因此空间复杂度为 $O(N)$，其中 N 为数字长度。

> 提示： 如果题目改成求删除 k 个字符之后的最大数，我们只需要将 stack[-1] > digit 中的大于号改成小于号即可。

316. 去除重复字母（困难）

题目描述

```
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

示例 1:

输入: "bcabc"
输出: "abc"
示例 2:

输入: "cbacdcbc"
输出: "acdb"
```

前置知识

- 字典序
- 数学

思路

与上面题目不同，这道题没有一个全局的删除次数 k。而是对于每一个在字符串 s 中出现的字母 c 都有一个 k 值。这个 k 是 c 出现次数 - 1。

沿用上面的知识的话，我们首先要做的就是计算每一个字符的 k，可以用一个字典来描述这种关系，其中 key 为 字符 c，value 为其出现的次数。

具体算法：

- 建立一个字典。其中 key 为 字符 c，value 为其出现的剩余次数。
- 从左往右遍历字符串，每次遍历到一个字符，其剩余出现次数 - 1.
- 对于每一个字符，如果其对应的剩余出现次数大于 1，我们**可以**选择丢弃（也可以选择不丢弃），否则不可以丢弃。
- 是否丢弃的标准和上面题目类似。如果栈中相邻的元素字典序更大，那么我们选择丢弃相邻的栈中的元素。

还记得上面题目的边界条件么？如果栈中剩下的元素大于 $n - k$，我们选择截取前 $n - k$ 个数字。然而本题中的 k 是分散在各个字符中的，因此这种思路不可行的。

不过不必担心。由于题目是要求只出现一次。我们可以在遍历的时候简单地判断其是否在栈上即可。

代码：

```
class Solution:
    def removeDuplicateLetters(self, s) -> int:
        stack = []
        remain_counter = collections.Counter(s)

        for c in s:
            if c not in stack:
                while stack and c < stack[-1] and  remain_
                    stack.pop()
                stack.append(c)
            remain_counter[c] -= 1
        return ''.join(stack)
```

复杂度分析

- 时间复杂度：由于判断当前字符是否在栈上存在需要 $O(N)$ 的时间，因此总的时间复杂度就是 $O(N ^ 2)$，其中 N 为字符串长度。
- 空间复杂度：我们使用了额外的栈来存储数字，因此空间复杂度为 $O(N)$，其中 N 为字符串长度。

查询给定字符是否在一个序列中存在的方法。根本上来说，有两种可能：

- 有序序列： 可以二分法，时间复杂度大致是 $O(N)$。
- 无序序列： 可以使用遍历的方式，最坏的情况下时间复杂度为 $O(N)$。我们也可以使用空间换时间的方式，使用 N的空间 换取 $O(1)$的时间复杂度。

由于本题中的 stack 并不是有序的，因此我们的优化点考虑空间换时间。而由于每种字符仅可以出现一次，这里使用 hashset 即可。

代码（Python）

```
class Solution:
    def removeDuplicateLetters(self, s) -> int:
        stack = []
        seen = set()
        remain_counter = collections.Counter(s)

        for c in s:
            if c not in seen:
                while stack and c < stack[-1] and  remain_c
                    seen.discard(stack.pop())
                seen.add(c)
                stack.append(c)
            remain_counter[c] -= 1
        return ''.join(stack)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为字符串长度。
- 空间复杂度：我们使用了额外的栈和 hashset，因此空间复杂度为 $O(N)$，其中 N 为字符串长度。

> LeetCode 《1081. 不同字符的最小子序列》 和本题一样，不再赘述。

321. 拼接最大数（困难）

题目描述

```
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

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]
```

前置知识

- 分治
- 数学

思路

和第一道题类似，只不不过这一次是两个**数组**，而不是一个，并且是求最大数。

最大最小是无关紧要的，关键在于是两个数组，并且要求从两个数组选取的元素个数加起来一共是 k。

然而在一个数组中取 k 个数字，并保持其最小（或者最大），我们已经会了。但是如果问题扩展到两个，会有什么变化呢?

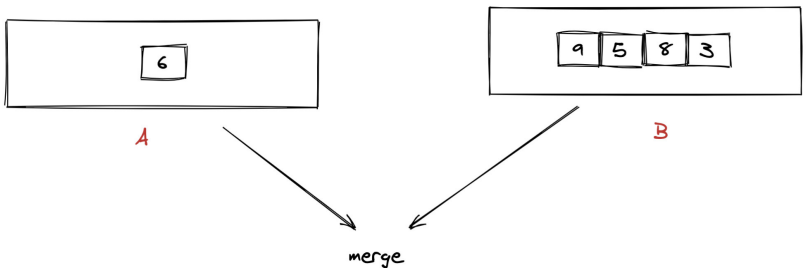
实际上，问题本质并没有发生变化。 假设我们从 nums1 中取了 k1 个，从 num2 中取了 k2 个，其中 k1 + k2 = k。而 k1 和 k2 这 两个子问题我们是会解决的。由于这两个子问题是相互独立的，因此我们只需要分别求解，然后将结果合并即可。

假如 k1 和 k2 个数字，已经取出来了。那么剩下要做的就是将这个长度分别为 k1 和 k2 的数字，合并成一个长度为 k 的数组合并成一个最大的数组。

以题目的 `nums1 = [3, 4, 6, 5] nums2 = [9, 1, 2, 5, 8, 3] k = 5` 为例。 假如我们从 num1 中取出 1 个数字，那么就要从 nums2 中取出 4 个数字。

运用第一题的方法，我们计算出应该取 nums1 的 [6]，并取 nums2 的 [9,5,8,3]。 如何将 [6] 和 [9,5,8,3]，使得数字尽可能大，并且保持相对位置不变呢?

实际上这个过程有点类似 `归并排序` 中的**治**，而上面我们分别计算 num1 和 num2 的最大数的过程类似 `归并排序` 中的**分**。



（图 6）

代码：

> 我们将从 num1 中挑选的 k1 个数组成的数组称之为 A，将从 num2 中挑选的 k2 个数组成的数组称之为 B，

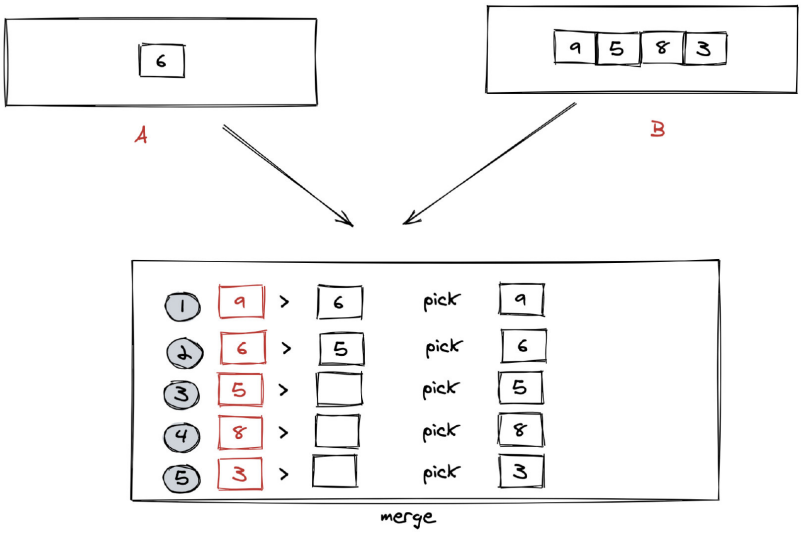
```
def merge(A, B):
    ans = []
    while A or B:
        bigger = A if A > B else B
        ans.append(bigger[0])
        bigger.pop(0)
    return ans
```

这里需要说明一下。 在很多编程语言中：**如果 A 和 B 是两个数组，当前仅当 A 的首个元素字典序大于 B 的首个元素，A > B 返回 true，否则返回 false**。

比如：

```
A = [1,2]
B = [2]
A < B # True

A = [1,2]
B = [1,2,3]
A < B # False
```

以合并 [6] 和 [9,5,8,3] 为例，图解过程如下：



（图 7）

具体算法：

- 从 nums1 中 取 $min(i, len(nums1))$ 个数形成新的数组 A（取的逻辑同第一题），其中 i 等于 0,1,2, ... k。
- 从 nums2 中 对应取 $min(j, len(nums2))$ 个数形成新的数组 B（取的逻辑同第一题），其中 j 等于 k - i。
- 将 A 和 B 按照上面的 merge 方法合并
- 上面我们暴力了 k 种组合情况，我们只需要将 k 种情况取出最大值即可。

代码（Python）

```
class Solution:
    def maxNumber(self, nums1, nums2, k):

        def pick_max(nums, k):
            stack = []
            drop = len(nums) - k
            for num in nums:
                while drop and stack and stack[-1] < num:
                    stack.pop()
                    drop -= 1
                stack.append(num)
            return stack[:k]

        def merge(A, B):
            ans = []
            while A or B:
                bigger = A if A > B else B
                ans.append(bigger[0])
                bigger.pop(0)
            return ans

        return max(merge(pick_max(nums1, i), pick_max(nums2
```

复杂度分析

- 时间复杂度：pick_max 的时间复杂度为 $O(M + N)$ ，其中 M 为 nums1 的长度，N 为 nums2 的长度。 merge 的时间复杂度为 $O(k)$，再加上外层遍历所有的 k 中可能性。因此总的时间复杂度为 $O(k^2 * (M + N))$。
- 空间复杂度：我们使用了额外的 stack 和 ans 数组，因此空间复杂度为 $O(max(M, N, k))$，其中 M 为 nums1 的长度，N 为 nums2 的长度。

总结

这四道题都是删除或者保留若干个字符，使得剩下的数字最小（或最大）或者字典序最小（或最大）。而解决问题的前提是要有一定**数学前提**。而基于这个数学前提，我们贪心地删除栈中相邻的字符。如果你会了这个套路，那么这四个题目应该都可以轻松解决。

`316. 去除重复字母（困难）`，我们使用 hashmap 代替了数组的遍历查找，属于典型的空间换时间方式，可以认识到数据结构的灵活使用是多么的重要。背后的思路是怎么样的？为什么想到空间换时间的方式，我在文中也进行了详细的说明，这都是值得大家思考的问题。然而实际上，这些题目中使用的栈也都是空间换时间的思想。大家下次碰到**需要空间换取时间**的场景，是否能够想到本文给大家介绍的**栈**和**哈希表**呢？

321. 拼接最大数（困难） 则需要我们能够对问题进行分解，这绝对不是一件简单的事情。但是对难以解决的问题进行分解是一种很重要的技能，希望大家能够通过这道题加深这种**分治**思想的理解。 大家可以结合我之前写过的几个题解练习一下，它们分别是：

- 【简单易懂】归并排序（Python）
- 一文看懂《最大子序列和问题》

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

【西法的刷题秘籍】一次搞定前缀和

我花了几天时间，从力扣中精选了五道相同思想的题目，来帮助大家解套，如果觉得文章对你有用，记得点赞分享，让我看到你的认可，有动力继续做下去。

- 467. 环绕字符串中唯一的子字符串(中等)
- 795. 区间子数组个数(中等)
- 904. 水果成篮(中等)
- 992. K 个不同整数的子数组（困难）
- 1109. 航班预订统计(中等)

前四道题都是滑动窗口的子类型，我们知道滑动窗口适合在题目要求连续的情况下使用， 而前缀和也是如此。二者在连续问题中，对于**优化时间复杂度**有着很重要的意义。 因此如果一道题你可以用暴力解决出来，而且题目恰好有连续的限制， 那么滑动窗口和前缀和等技巧就应该被想到。

除了这几道题， 还有很多题目都是类似的套路， 大家可以在学习过程中进行体会。今天我们就来一起学习一下。

前菜

我们从一个简单的问题入手，识别一下这种题的基本形式和套路，为之后的四道题打基础。当你了解了这个套路之后， 之后做这种题就可以直接套。

需要注意的是这四道题的前置知识都是 `滑动窗口` ， 不熟悉的同学可以先看下我之前写的 滑动窗口专题（思路 + 模板）

母题 0

有 N 个的正整数放到数组 A 里，现在要求一个新的数组 B，新数组的第 i 个数 B[i]是原数组 A 第 0 到第 i 个数的和。

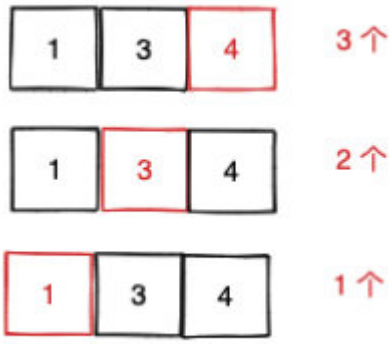
这道题可以使用前缀和来解决。 前缀和是一种重要的预处理，能大大降低查询的时间复杂度。我们可以简单理解为“数列的前 n 项的和”。这个概念其实很容易理解，即一个数组中，第 n 位存储的是数组前 n 个数字的和。

对 [1,2,3,4,5,6] 来说，其前缀和可以是 pre=[1,3,6,10,15,21]。我们可以使用公式 pre[i]=pre[$i-1$]+nums[i]得到每一位前缀和的值，从而通过前缀和进行相应的计算和解题。其实前缀和的概念很简单，但困难的是如何在题目中使用前缀和以及如何使用前缀和的关系来进行解题。

母题 1

如果让你求一个数组的连续子数组总个数，你会如何求？其中连续指的是数组的索引连续。 比如 [1,3,4]，其连续子数组有： `[1]，[3]，[4]，[1,3]，[3,4] ，[1,3,4]` ，你需要返回 6。

一种思路是总的连续子数组个数等于：**以索引为 0 结尾的子数组个数 + 以索引为 1 结尾的子数组个数 + ... + 以索引为 n - 1 结尾的子数组个数**，这无疑是完备的。



同时**利用母题 0 的前缀和思路， 边遍历边求和。**

参考代码(JS)：

```
function countSubArray(nums) {
  let ans = 0;
  let pre = 0;
  for (_ in nums) {
    pre += 1;
    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

而由于以索引为 i 结尾的子数组个数就是 i + 1，因此这道题可以直接用等差数列求和公式 `(1 + n) * n / 2` ，其中 n 数组长度。

母题 2

我继续修改下题目， 如果让你求一个数组相邻差为 1 连续子数组的总个数呢？其实就是**索引差 1 的同时，值也差 1。**

和上面思路类似，无非就是增加差值的判断。

参考代码(JS)：

```
function countSubArray(nums) {
  let ans = 1;
  let pre = 1;
  for (let i = 1; i < nums.length; i++) {
    if (nums[i] - nums[i - 1] == 1) {
      pre += 1;
    } else {
      pre = 0;
    }

    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

如果我值差只要大于 1 就行呢？其实改下符号就行了，这不就是求上升子序列个数么？这里不再继续赘述， 大家可以自己试试。

母题 3

我们继续扩展。

如果我让你求出不大于 k 的子数组的个数呢？不大于 k 指的是子数组的全部元素都不大于 k。 比如 [1,3,4] 子数组有 `[1], [3], [4], [1,3], [3,4] , [1,3,4]` ，不大于 3 的子数组有 `[1], [3], [1,3]` ，那么 [1,3,4] 不大于 3 的子数组个数就是 3。 实现函数 atMostK(k, nums)。

参考代码（JS）：

```
function countSubArray(k, nums) {
  let ans = 0;
  let pre = 0;
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] <= k) {
      pre += 1;
    } else {
      pre = 0;
    }

    ans += pre;
  }
  return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

母题 4

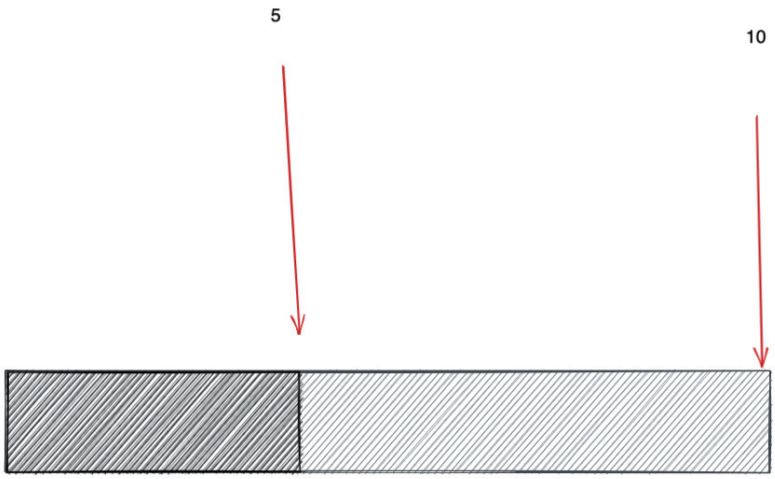
如果我让你求出子数组最大值刚好是 k 的子数组的个数呢？ 比如 [1,3,4] 子数组有 `[1], [3], [4], [1,3], [3,4] , [1,3,4]`，子数组最大值刚好是 3 的子数组有 `[3], [1,3]` ，那么 [1,3,4] 子数组最大值刚好是 3 的子数组个数就是 2。实现函数 exactK(k, nums)。

实际上是 exactK 可以直接利用 atMostK，即 atMostK(k) - atMostK(k - 1)，原因见下方母题 5 部分。

母题 5

如果我让你求出子数组最大值刚好是 介于 k1 和 k2 的子数组的个数呢? 实现函数 betweenK(k1, k2, nums)。

实际上是 betweenK 可以直接利用 atMostK，即 atMostK(k1, nums) - atMostK(k2 - 1, nums)，其中 k1 > k2。前提是值是离散的， 比如上面我出的题都是整数。 因此我可以直接 减 1，因为 **1 是两个整数最小的间隔**。



如上，`小于等于 10 的区域`减去`小于 5 的区域`就是`大于等于 5 且小于等于 10 的区域`。

注意我说的是小于 5， 不是小于等于 5。 由于整数是离散的，最小间隔是 1。因此小于 5 在这里就等价于 小于等于 4。这就是 betweenK(k1, k2, nums) = atMostK(k1) - atMostK(k2 - 1) 的原因。

因此不难看出 exactK 其实就是 betweenK 的特殊形式。 当 k1 == k2 的时候， betweenK 等价于 exactK。

因此 atMostK 就是灵魂方法，一定要掌握，不明白建议多看几遍。

有了上面的铺垫， 我们来看下第一道题。

467. 环绕字符串中唯一的子字符串（中等）

题目描述

```
把字符串 s 看作是“abcdefghijklmnopqrstuvwxyz”的无限环绕字符串，月

现在我们有了另一个字符串 p 。你需要的是找出 s 中有多少个唯一的 p 的非

注意: p 仅由小写的英文字母组成，p 的大小可能超过 10000。



示例 1:

输入: "a"
输出: 1
解释: 字符串 S 中只有一个"a"子字符。


示例 2:

输入: "cac"
输出: 2
解释: 字符串 S 中的字符串“cac”只有两个子串“a”、“c”。.


示例 3:

输入: "zab"
输出: 6
解释: 在字符串 S 中有六个子串“z”、“a”、“b”、“za”、“ab”、“zab”。.
```

前置知识

- 滑动窗口

思路

题目是让我们找 p 在 s 中出现的非空子串数目，而 s 是固定的一个无限循环字符串。由于 p 的数据范围是 10^5 ，因此暴力找出所有子串就需要 10^10 次操作了，应该会超时。而且题目很多信息都没用到，肯定不对。

仔细看下题目发现，这不就是母题 2 的变种么？话不多说， 直接上代码，看看有多像。

> 为了减少判断， 我这里用了一个黑科技， p 前面加了个 `^` 。

```
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int
        p = '^' + p
        w = 1
        ans = 0
        for i in range(1,len(p)):
            if ord(p[i])-ord(p[i-1]) == 1 or ord(p[i])-ord(
                w += 1
            else:
                w = 1
            ans += w
        return ans
```

如上代码是有问题。 比如 `cac` 会被计算为 3，实际上应该是 2。根本原因在于 c 被错误地计算了两次。因此一个简单的思路就是用 set 记录一下访问过的子字符串即可。比如：

```
{
    c,
    abc,
    ab,
    abcd
}
```

而由于 set 中的元素一定是连续的，因此上面的数据也可以用 hashmap 存：

```
{
    c: 3
    d: 4
    b: 1
}
```

含义是：

- 以 b 结尾的子串最大长度为 1，也就是 b。
- 以 c 结尾的子串最大长度为 3，也就是 abc。
- 以 d 结尾的子串最大长度为 4，也就是 abcd。

至于 c ，是没有必要存的。我们可以通过母题 2 的方式算出来。

具体算法：

- 定义一个 len_mapper。key 是 字母， value 是 长度。 含义是以 key 结尾的最长连续子串的长度。

> 关键字是：最长

- 用一个变量 w 记录连续子串的长度，遍历过程根据 w 的值更新 len_mapper
- 返回 len_mapper 中所有 value 的和。

比如: abc，此时的 len_mapper 为:

```
{
    c: 3
    b: 2
    a: 1
}
```

再比如：abcab，此时的 len_mapper 依旧。

再比如: abcazabc，此时的 len_mapper：

```
{
    c: 4
    b: 3
    a: 2
    z: 1
}
```

这就得到了去重的目的。这种算法是不重不漏的，因为最长的连续子串一定是包含了比它短的连续子串，这个思想和 1297. 子串的最大出现次数 剪枝的方法有异曲同工之妙。

代码（Python）

```python
class Solution:
    def findSubstringInWraproundString(self, p: str) -> int
        p = '^' + p
        len_mapper = collections.defaultdict(lambda: 0)
        w = 1
        for i in range(1,len(p)):
            if ord(p[i])-ord(p[i-1]) == 1 or ord(p[i])-ord(
                w += 1
            else:
                w = 1
            len_mapper[p[i]] = max(len_mapper[p[i]], w)
        return sum(len_mapper.values())
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为字符串 p 的长度。

- 空间复杂度：由于最多存储 26 个字母， 因此空间实际上是常数，故空间复杂度为 $O(1)$。

795. 区间子数组个数（中等）

题目描述

```
给定一个元素都是正整数的数组 A ，正整数 L  以及  R (L <= R)。

求连续、非空且其中最大元素满足大于等于 L  小于等于 R 的子数组个数。

例如 :
输入:
A = [2, 1, 4, 3]
L = 2
R = 3
输出: 3
解释: 满足条件的子数组: [2], [2, 1], [3].
注意:

L, R  和  A[i] 都是整数，范围在  [0, 10^9]。
数组  A  的长度范围在[1, 50000]。
```

前置知识

- 滑动窗口

思路

由母题 5，我们知道 **betweenK 可以直接利用 atMostK，即 atMostK(k1) - atMostK(k2 - 1)，其中 k1 > k2**。

由母题 2，我们知道如何求满足一定条件（这里是元素都小于等于 R）子数组的个数。

这两个结合一下， 就可以解决。

代码（Python）

> 代码是不是很像

```
class Solution:
    def numSubarrayBoundedMax(self, A: List[int], L: int, R
        def notGreater(R):
            ans = cnt = 0
            for a in A:
                if a <= R: cnt += 1
                else: cnt = 0
                ans += cnt
            return  ans

        return notGreater(R) - notGreater(L - 1)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$。

904. 水果成篮（中等）

题目描述

在一排树中，第 i 棵树产生 tree[i] 型的水果。
你可以从你选择的任何树开始，然后重复执行以下步骤：

把这棵树上的水果放进你的篮子里。如果你做不到，就停下来。
移动到当前树右侧的下一棵树。如果右边没有树，就停下来。
请注意，在选择一颗树后，你没有任何选择：你必须执行步骤 1，然后执行步骤

你有两个篮子，每个篮子可以携带任何数量的水果，但你希望每个篮子只携带一

用这个程序你能收集的水果树的最大总量是多少？

示例 1：

输入：[1,2,1]
输出：3
解释：我们可以收集 [1,2,1]。
示例 2：

输入：[0,1,2,2]
输出：3
解释：我们可以收集 [1,2,2]
如果我们从第一棵树开始，我们将只能收集到 [0, 1]。
示例 3：

输入：[1,2,3,2,2]
输出：4
解释：我们可以收集 [2,3,2,2]
如果我们从第一棵树开始，我们将只能收集到 [1, 2]。
示例 4：

输入：[3,3,3,1,2,1,1,2,3,3,4]
输出：5
解释：我们可以收集 [1,2,1,1,2]
如果我们从第一棵树或第八棵树开始，我们将只能收集到 4 棵水果树。

提示：

1 <= tree.length <= 40000
0 <= tree[i] < tree.length

前置知识

- 滑动窗口

思路

题目花里胡哨的。我们来抽象一下，就是给你一个数组， 让你**选定一个子数组， 这个子数组最多只有两种数字**，这个选定的子数组最大可以是多少。

这不就和母题 3 一样么？只不过 k 变成了固定值 2。另外由于题目要求整个窗口最多两种数字，我们用哈希表存一下不就好了吗?

> set 是不行了的。 因此我们不但需要知道几个数字在窗口， 我们还要知道每个数字出现的次数，这样才可以使用滑动窗口优化时间复杂度。

代码（Python）

```
class Solution:
    def totalFruit(self, tree: List[int]) -> int:
        def atMostK(k, nums):
            i = ans = 0
            win = defaultdict(lambda: 0)
            for j in range(len(nums)):
                if win[nums[j]] == 0: k -= 1
                win[nums[j]] += 1
                while k < 0:
                    win[nums[i]] -= 1
                    if win[nums[i]] == 0: k += 1
                    i += 1
                ans = max(ans, j - i + 1)
            return ans

        return atMostK(2, tree)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(k)$。

992. K 个不同整数的子数组（困难）

题目描述

```
给定一个正整数数组 A，如果 A 的某个子数组中不同整数的个数恰好为 K，则

（例如，[1,2,3,1,2] 中有 3 个不同的整数：1，2，以及 3。）

返回 A 中好子数组的数目。



示例 1：

输入：A = [1,2,1,2,3], K = 2
输出：7
解释：恰好由 2 个不同整数组成的子数组：[1,2], [2,1], [1,2], [2,3]
示例 2：

输入：A = [1,2,1,3,4], K = 3
输出：3
解释：恰好由 3 个不同整数组成的子数组：[1,2,1,3], [2,1,3], [1,3,4


提示：

1 <= A.length <= 20000
1 <= A[i] <= A.length
1 <= K <= A.length
```

前置知识

- 滑动窗口

思路

由母题 5，知：exactK = atMostK(k) - atMostK(k - 1)， 因此答案便呼之欲出了。其他部分和上面的题目 `904. 水果成篮` 一样。

> 实际上和所有的滑动窗口题目都差不多。

代码（Python）

```
class Solution:
    def subarraysWithKDistinct(self, A, K):
        return self.atMostK(A, K) - self.atMostK(A, K - 1)

    def atMostK(self, A, K):
        counter = collections.Counter()
        res = i = 0
        for j in range(len(A)):
            if counter[A[j]] == 0:
                K -= 1
            counter[A[j]] += 1
            while K < 0:
                counter[A[i]] -= 1
                if counter[A[i]] == 0:
                    K += 1
                i += 1
            res += j - i + 1
        return res
```

复杂度分析

- 时间复杂度：$O(N)$，中 N 为数组长度。
- 空间复杂度：$O(k)$。

1109. 航班预订统计（中等）

题目描述

这里有 n 个航班，它们分别从 1 到 n 进行编号。

我们这儿有一份航班预订表，表中第 i 条预订记录 bookings[i] = [i,

请你返回一个长度为 n 的数组 answer，按航班编号顺序返回每个航班上预订

示例：

输入：bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
输出：[10,55,45,25,25]

提示：

1 <= bookings.length <= 20000
1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000
1 <= bookings[i][2] <= 10000

前置知识

- 前缀和

思路

这道题的题目描述不是很清楚。我简单分析一下题目：

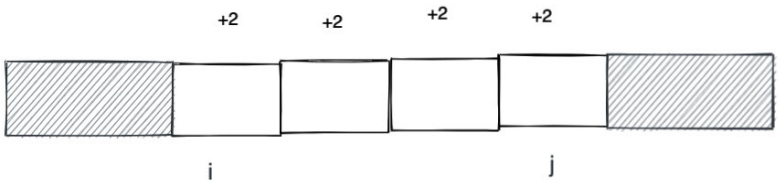
[i, j, k] 其实代表的是 第 i 站上来了 k 个人， 一直到 第 j 站都在飞机上，到第 j + 1 就不在飞机上了。所以第 i 站到第 j 站的**每一站**都会因此多 k 个人。

理解了题目只会不难写出下面的代码。

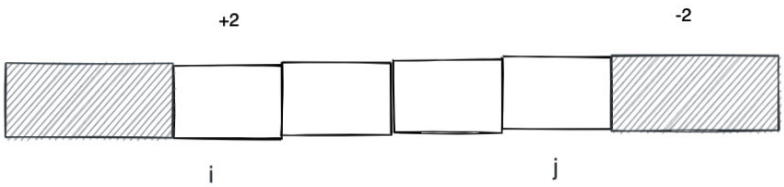
```
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]],
        counter = [0] * n

        for i, j, k in bookings:
            while i <= j:
                counter[i - 1] += k
                i += 1
        return counter
```

如上的代码复杂度太高，无法通过全部的测试用例。

注意到里层的 while 循环是连续的数组全部加上一个数字，不难想到可以利用母题 0 的前缀和思路优化。



一种思路就是在 i 的位置 + k， 然后利用前缀和的技巧给 i 到 n 的元素都加上 k。但是题目需要加的是一个区间， j + 1 及其之后的元素会被多加一个 k。一个简单的技巧就是给 j + 1 的元素减去 k，这样正负就可以抵消。



> 1. 拼车 是这道题的换皮题， 思路一模一样。

代码（Python）

```
class Solution:
    def corpFlightBookings(self, bookings: List[List[int]],
        counter = [0] * (n + 1)

        for i, j, k in bookings:
            counter[i - 1] += k
            if j < n: counter[j] -= k
        for i in range(n + 1):
            counter[i] += counter[i - 1]
        return counter[:-1]
```

复杂度分析

- 时间复杂度：$O(N)$，中 N 为数组长度。
- 空间复杂度：$O(N)$。

总结

这几道题都是滑动窗口和前缀和的思路。力扣类似的题目还真不少，大家只有多留心，就会发现这个套路。

前缀和的技巧以及滑动窗口的技巧都比较固定，且有模板可套。 难点就在于我怎么才能想到可以用这个技巧呢?

我这里总结了两点：

1. 找关键字。比如题目中有连续，就应该条件反射想到滑动窗口和前缀和。比如题目求最大最小就想到动态规划和贪心等等。想到之后，就可以和题目信息对比快速排除错误的算法，找到可行解。这个思考的时间会随着你的题感增加而降低。
2. 先写出暴力解，然后找暴力解的瓶颈， 根据瓶颈就很容易知道应该用什么数据结构和算法去优化。

最后推荐几道类似的题目， 供大家练习，一定要自己写出来才行哦。

- 303. 区域和检索 - 数组不可变
- 1171. 从链表中删去总和值为零的连续节点
- 1186.删除一次得到子数组最大和
- 1310. 子数组异或查询
- 1371. 每个元音包含偶数次的最长子字符串
- 1402. 做菜顺序

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。

更多算法套路可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

字节跳动的算法面试题是什么难度?

由于 lucifer 我是一个小前端， 最近也在准备写一个《前端如何搞定算法面试》的专栏，因此最近没少看各大公司的面试题。都说字节跳动算法题比较难，我就先拿 ta 下手，做了几套 。这次我们就拿一套 `2018 年的前端校招（第四批）` 来看下字节的算法笔试题的难度几何。地址：https://www.nowcoder.com/test/8536639/summary

> 实际上，这套字节的前端岗位笔试题和后端以及算法岗位的笔试题也只有一道题目（红包的设计题被换成了另外一个设计题）不一样而已，因此也不需要担心你不是前端，题目类型和难度和你的岗位不匹配。

这套题一共四道题， 两道问答题， 两道编程题。

其中一道问答题是 LeetCode 426 的原题，只不过题型变成了找茬（改错）。可惜的是 LeetCode 的 426 题是一个会员题目，没有会员的就看不来了。不过，剑指 Offer 正好也有这个题，并且力扣将剑指 Offer 全部的题目都 OJ 化了。 这道题大家可以去 https://leetcode-cn.com/problems/er-cha-sou-suo-shu-yu-shuang-xiang-lian-biao-lcof 提交答案。简单说一下这个题目的思路，我们只需要中序遍历即可得到一个有序的数列，同时在中序遍历过程中将 pre 和 cur 节点通过指针串起来即可。

另一个问答是红包题目，这里不多说了。我们重点看一下剩下两个算法编程题。

> 两个问答题由于不能在线判题，我没有做，只做了剩下两个编程题。

球队比赛

第一个编程题是一个球队比赛的题目。

题目描述

有三只球队，每只球队编号分别为球队 1，球队 2，球队 3，这三只球队一共需要进行 n 场比赛。现在已经踢完了 k 场比赛，每场比赛不能打平，踢赢一场比赛得一分，输了不得分不减分。已知球队 1 和球队 2 的比分相差 d1 分，球队 2 和球队 3 的比分相差 d2 分，每场比赛可以任意选择两只队伍进行。求如果打完最后的 (n-k) 场比赛，有没有可能三只球队的分数打平。

思路

假设球队 1，球队 2，球队 3 此时的胜利次数分别为 a，b，c，球队 1，球队 2，球队 3 总的胜利次数分别为 n1，n2，n3。

我一开始的想法是只要保证 n1，n2，n3 相等且都小于等于 n / 3 即可。如果题目给了 n1，n2，n3 的值就直接：

```
print(n1 == n2 == n3 == n / 3)
```

可是不仅 n1，n2，n3 没给， a，b，c 也没有给。

实际上此时我们的信息仅仅是：

```
① a + b + c = k
② a - b = d1 or b - a = d1
③ b - c = d2 or c - b = d2
```

其中 k 和 d1，d2 是已知的。a ，b，c 是未知的。 也就是说我们需要枚举所有的 a，b，c 可能性，解方程求出合法的 a，b，c，并且 合法的 a，b，c 都小于等于 n / 3 即可。

> 这个 a,b,c 的求解数学方程就是中学数学难度， 三个等式化简一下即可，具体见下方代码区域。

- a 只需要再次赢得 n / 3 - a 次
- b 只需要再次赢得 n / 3 - b 次
- c 只需要再次赢得 n / 3 - c 次

```
n1 = a + n / 3 - a = n / 3
n2 = b + (n / 3 - b) = n / 3
n3 = c + (n / 3 - c) = n / 3
```

代码(Python)

> 牛客有点让人不爽， 需要 print 而不是 return

```
t = int(input())
for i in range(t):
    n, k, d1, d2 = map(int, input().split(" "))
    if n % 3 != 0:
        print('no')
        continue
    abcs = []
    for r1 in [-1, 1]:
        for r2 in [-1, 1]:
            a = (k + 2 * r1 * d1 + r2 * d2) / 3
            b = (k + -1 * r1 * d1 + r2 * d2) / 3
            c = (k + -1 * r1 * d1 + -2 * r2 * d2) / 3
            a + r1
            if  0 <= a <= k and 0 <= b <= k and 0 <= c <= k
                abcs.append([a, b, c])
    flag = False
    for abc in abcs:
        if len(abc) > 0 and max(abc) <= n / 3:
            flag = True
            break
    if flag:
        print('yes')
    else:
        print('no')
```

复杂度分析

- 时间复杂度：$O(t)$
- 空间复杂度：$O(t)$

小结

感觉这个难度也就是力扣中等水平吧，力扣也有一些数学等式转换的题目， 比如 494.target-sum

转换字符串

题目描述

有一个仅包含’a’和’b’两种字符的字符串 s，长度为 n，每次操作可以把一个字符做一次转换（把一个’a’设置为’b’，或者把一个’b’置成’a’）；但是操作的次数有上限 m，问在有限的操作数范围内，能够得到最大连续的相同字符的子串的长度是多少。

思路

看完题我就有种似曾相识的感觉。

> 每次对妹子说出这句话的时候，她们都会觉得好假 ^_^

不过这次是真的。”哦，不！每次都是真的“。 这道题其实就是我之前写的滑动窗口的一道题【1004. 最大连续 1 的个数 III】滑动窗口（Python3）的换皮题。 专题地址：https://github.com/azl397985856/leetcode/blob/master/thinkings/slide-window.md

所以说，如果这道题你完全没有思路的话。说明：

- 抽象能力不够。
- 滑动窗口问题理解不到位。

第二个问题可以看我上面贴的地址，仔细读读，并完成课后练习即可解决。

第一个问题就比较困难了， 不过多看我的题解也可以慢慢提升的。比如：

- 《割绳子》 实际上就是 343. 整数拆分 的换皮题。
- 力扣 230 和 力扣 645 就是换皮题，详情参考位运算专题
- 以及 你的衣服我扒了 - 《最长公共子序列》
- 以及 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 以及 一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～
- 等等

回归这道题。其实我们只需要稍微抽象一下， 就是一个纯算法题。 抽象的另外一个好处则是将很多不同的题目返璞归真，从而可以在茫茫题海中逃脱。这也是我开启《我是你的妈妈呀》 的原因之一。

如果我们把 a 看成是 0 ， b 看成是 1。或者将 b 看成 1， a 看成 0。不就抽象成了：

```
给定一个由若干 0 和 1 组成的数组 A，我们最多可以将 m 个值从 0 变成 1

返回仅包含 1 的最长（连续）子数组的长度。
```

这就是 力扣 1004. 最大连续 1 的个数 III 原题。

因此实际上我们要求的是上面两种情况:

1. a 表示 0， b 表示 1
2. a 表示 1， b 表示 0

的较大值。

> lucifer 小提示： 其实我们也可以仅仅考虑一种情况，比如 a 看成是 0 ， b 看成是 1。这个时候， 我们操作变成了两种情况，0 变成 1 或者 1 变成 0，同时求解的也变成了最长连续 0 或者 最长连续 1 。由于这种抽象操作起来更麻烦， 我们不考虑。

问题得到了抽象就好解决了。我们只需要记录下加入窗口的是 0 还是 1：

- 如果是 1，我们什么都不用做
- 如果是 0，我们将 m 减 1

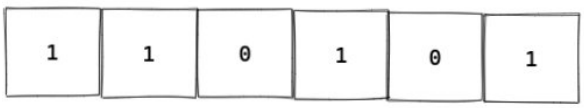
相应地，我们需要记录移除窗口的是 0 还是 1:

- 如果是 1，我们什么都不做
- 如果是 0，说明加进来的时候就是 1，加进来的时候我们 m 减去了 1，这个时候我们再加 1。

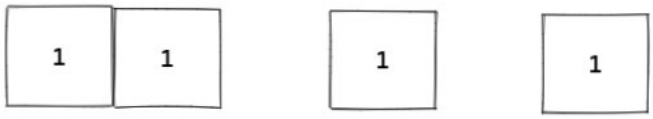
> lucifer 小提示： 实际上题目中是求**连续** a 或者 b 的长度。看到连续，大家也应该有滑动窗口的敏感度， 别管行不行， 想到总该有的。

我们拿 A = [1, 1, 0, 1, 0, 1], m = 1 来说。看下算法的具体过程：

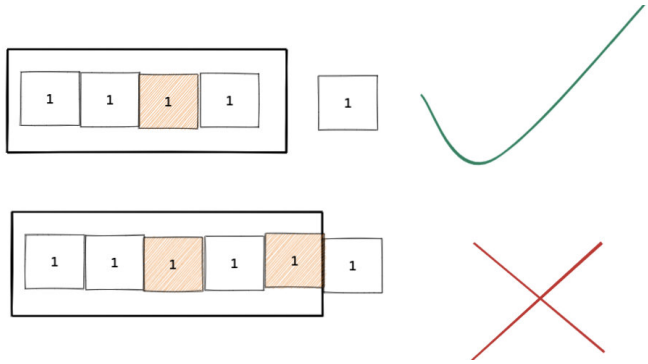
> lucifer 小提示： 左侧的数字表示此时窗口大小，黄色格子表示修补的墙，黑色方框表示的是窗口。



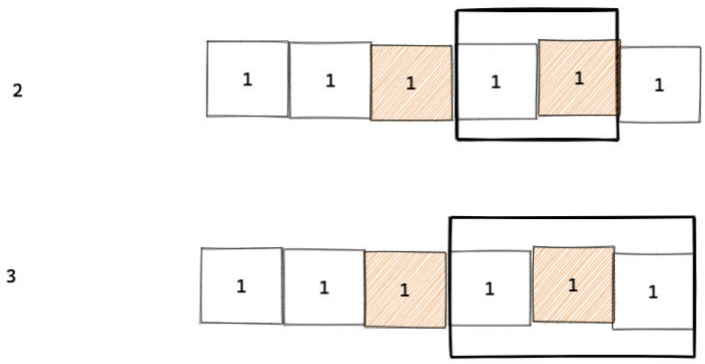
这里我形象地将 0 看成是洞，1 看成是墙， 我们的目标就是补洞，使得连续的墙最长。



每次碰到一个洞，我们都去不加选择地修补。由于 m 等于 1， 也就是说我们最多补一个洞。因此需要在修补超过一个洞的时候，我们需要调整窗口范围，使得窗口内最多修补一个墙。由于窗口表示的就是连续的墙（已有的或者修补的），因此最终我们返回窗口的最大值即可。



> 由于下面的图窗口内有两个洞，这和”最多补一个洞“冲突， 我们需要收缩窗口使得满足“最多补一个洞”的先决条件。



因此最大的窗口就是 max(2, 3, 4, ...) = 4。

> lucifer 小提示： 可以看出我们不加选择地修补了所有的洞，并调整窗口，使得窗口内最多有 m 个修补的洞，因此窗口的最大值就是答案。然而实际上，我们并不需要真的”修补“（0 变成 1），而是仅仅修改 m 的值即可。

我们先来看下抽象之后的**其中一种情况**的代码：

```
class Solution:
    def longestOnes(self, A: List[int], m: int) -> int:
        i = 0
        for j in range(len(A)):
            m -= 1 - A[j]
            if m < 0:
                m += 1 - A[i]
                i += 1
        return j - i + 1
```

因此**完整代码**就是：

```
class Solution:
    def longestOnes(self, A: List[int], m: int) -> int:
        i = 0
        for j in range(len(A)):
            m -= 1 - A[j]
            if m < 0:
                m += 1 - A[i]
                i += 1
        return j - i + 1
    def longestAorB(self, A:List[int], m: int) -> int:
        return max(self.longestOnes(map(lambda x: 0 if x ==
```

这里的两个 map 会生成两个不同的数组。 我只是为了方便大家理解才新建的两个数组， 实际上根本不需要，具体见后面的代码.

代码(Python)

```
i = 0
n, m = map(int, input().split(" "))
s = input()
ans = 0
k = m #   存一下，后面也要用这个初始值
# 修补 b
for j in range(n):
    m -= ord(s[j]) - ord('a')
    if m < 0:
        m += ord(s[i]) - ord('a')
        i += 1
ans = j - i + 1
i = 0
# 修补 a
for j in range(n):
    k += ord(s[j]) - ord('b')
    if k < 0:
        k -= ord(s[i]) - ord('b')
        i += 1
print(max(ans, j - i + 1))
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

小结

这道题就是一道换了皮的力扣题，难度中等。如果你能将问题抽象，同时又懂得滑动窗口，那这道题就很容易。我看了题解区的参考答案， 内容比较混乱，不够清晰。这也是我写下这篇文章的原因之一。

总结

这一套字节跳动的题目一共四道，一道设计题，三道算法题。

其中三道算法题从难度上来说，基本都是中等难度。从内容来看，基本都是力扣的换皮题。但是如果我不说他们是换皮题， 你们能发现么？ 如果你可以的话，说明你的抽象能力已经略有小成了。如果看不出来也没有关系，关注我。 手把手扒皮给你们看，扒多了慢慢就会了。切记，不要盲目做题！如果你做了很多题， 这几道题还是看不出套路，说明你该缓缓，改变下刷题方式了。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 36K+ star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

字节跳动的算法面试题是什么难度？（第二弹）

由于 lucifer 我是一个小前端， 最近也在准备写一个《前端如何搞定算法面试》的专栏，因此最近没少看各大公司的面试题。都说字节跳动算法题比较难，我就先拿 ta 下手，做了几套 。这次我们就拿一套 `字节跳动2017秋招编程题汇总` 来看下字节的算法笔试题的难度几何。地址：https://www.nowcoder.com/test/6035789/summary

这套题一共 11 道题， 三道编程题， 八道问答题。本次给大家带来的就是这三道编程题。更多精彩内容，请期待我的搞定算法面试专栏。

其中有一道题《异或》我没有通过所有的测试用例， 小伙伴可以找找茬，第一个找到并在公众号力扣加加留言的小伙伴奖励现金红包 10 元。

1. 头条校招

题目描述

```
头条的 2017 校招开始了！为了这次校招，我们组织了一个规模宏大的出题团队
a<=b<=c
b-a<=10
c-b<=10
所有出题人一共出了 n 道开放性题目。现在我们想把这 n 道题分布到若干场考

输入描述:
输入的第一行包含一个整数 n，表示目前已经出好的题目数量。

第二行给出每道题目的难度系数 d1,d2,...,dn。

数据范围

对于 30%的数据，1 ≤ n,di ≤ 5;

对于 100%的数据，1 ≤ n ≤ 10^5,1 ≤ di ≤ 100。

在样例中，一种可行的方案是添加 2 个难度分别为 20 和 50 的题目，这样可

输出描述:
输出只包括一行，即所求的答案。
示例 1
输入
4
20 35 23 40
输出
2
```

思路

这道题看起来很复杂， 你需要考虑很多的情况。，属于那种没有技术含量，但是考验编程能力的题目，需要思维足够严密。这种**模拟的题目**，就是题目让我干什么我干什么。 类似之前写的囚徒房间问题，约瑟夫环也是模拟，只不过模拟之后需要你剪枝优化。

这道题的情况其实很多， 我们需要考虑每一套题中的难度情况， 而不需要考虑不同套题的难度情况。题目要求我们满足： `a<=b<=c b-a<=10 c-b<=10` ，也就是题目难度从小到大排序之后，相邻的难度不能大于 10 。

因此我们的思路就是先排序，之后从小到大遍历，如果满足相邻的难度不大于 10 ，则继续。如果不满足， 我们就只能让字节的老师出一道题使得满足条件。

由于只需要比较同一套题目的难度，因此我的想法就是**比较同一套题目的第二个和第一个，以及第三个和第二个的 diff**。

- 如果 diff 小于 10，什么都不做，继续。

- 如果 diff 大于 10，我们必须补充题目。

这里有几个点需要注意。

对于第二题来说：

- 比如 1 **30** 40 这样的难度。 我可以在 1，30 之间加一个 21，这样 1，21，30 就可以组成一一套。
- 比如 1 **50** 60 这样的难度。 我可以在 1，50 之间加 21， 41 才可以组成一套，自身（50）是无论如何都没办法组到这套题中的。

不难看出， 第二道题的临界点是 diff = 20 。 小于等于 20 都可以将自身组到套题，增加一道即可，否则需要增加两个，并且自身不能组到当前套题。

对于第三题来说：

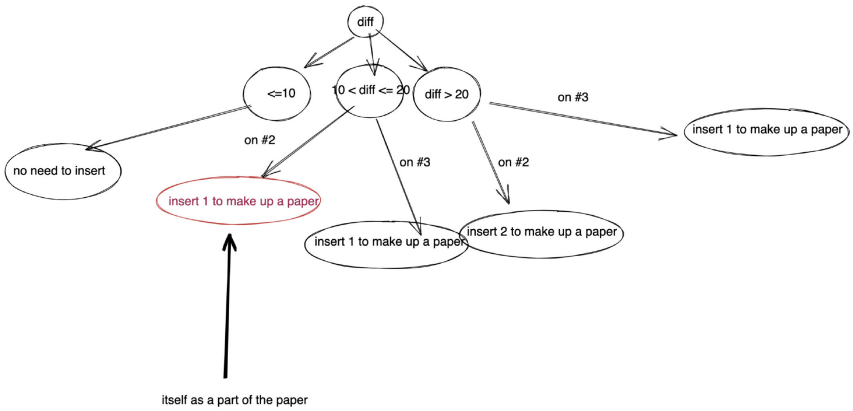
- 比如 1 20 **40**。 我可以在 20，40 之间加一个 30，这样 1，20，30 就可以组成一一套，自身（40）是无法组到这套题的。
- 比如 1 20 **60**。 也是一样的，我可以在 20，60 之间加一个 30，自身（60）同样是没办法组到这套题中的。

不难看出， 第三道题的临界点是 diff = 10 。 小于等于 10 都可以将自身组到套题，否则需要增加一个，并且自身不能组到当前套题。

这就是所有的情况了。

有的同学比较好奇，我是怎么思考的。 我是怎么**保障不重不漏**的。

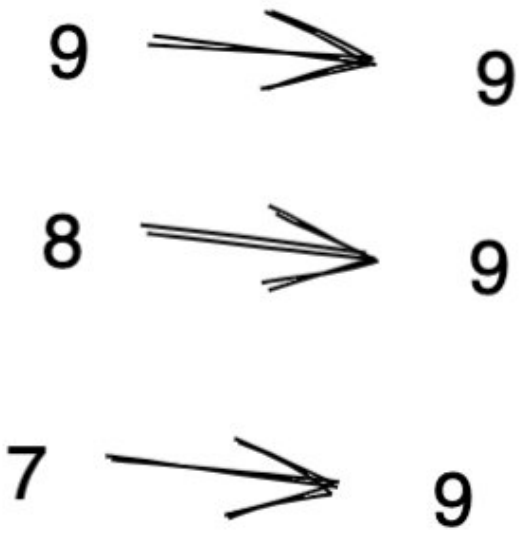
实际上，这道题就是一个决策树， 我画个决策树出来你就明白了。



> 图中红色边框表示自身可以组成套题的一部分， 我也用文字进行了说明。#2 代表第二题， #3 代表第三题。

从图中可以看出， 我已经考虑了所有情况。如果你能够像我一样画出这个决策图，我想你也不会漏的。当然我的解法并不一定是最优的，不过确实是一个非常好用，具有普适性的思维框架。

需要特别注意的是，由于需要凑整， 因此你需要使得题目的总数是 3 的倍数向上取整。



代码

```
n = int(input())
nums = list(map(int, input().split()))
cnt = 0
cur = 1
nums.sort()
for i in range(1, n):
    if cur == 3:
        cur = 1
        continue
    diff = nums[i] - nums[i - 1]
    if diff <= 10:
        cur += 1
    if 10 < diff <= 20:
        if cur == 1:
            cur = 3
        if cur == 2:
            cur = 1
        cnt += 1
    if diff > 20:
        if cur == 1:
            cnt += 2
        if cur == 2:
            cnt += 1
        cur = 1
print(cnt + 3 - cur)
```

复杂度分析

- 时间复杂度：由于使用了排序， 因此时间复杂度为 $O(NlogN)$。（假设使用了基于比较的排序）
- 空间复杂度：$O(1)$

2. 异或

题目描述

```
给定整数 m 以及 n 各数字 A1,A2,..An，将数列 A 中所有元素两两异或，

输入描述:
第一行包含两个整数 n,m.

第二行给出 n 个整数 A1，A2，...，An。

数据范围

对于 30%的数据，1 <= n, m <= 1000

对于 100%的数据，1 <= n, m, Ai <= 10^5

输出描述:
输出仅包括一行，即所求的答案

输入例子 1:
3 10
6 5 10

输出例子 1:
2
```

前置知识

- 异或运算的性质
- 如何高效比较两个数的大小（从高位到低位）

首先普及一下前置知识。 第一个是异或运算：

异或的性质：两个数字异或的结果 a^b 是将 a 和 b 的二进制每一位进行运算，得出的数字。 运算的逻辑是如果同一位的数字相同则为 0，不同则为 1

异或的规律：

1. 任何数和本身异或则为 0
2. 任何数和 0 异或是本身
3. 异或运算满足交换律，即： a ^ b ^ c = a ^ c ^ b

同时建议大家去看下我总结的几道位运算的经典题目。 位运算系列

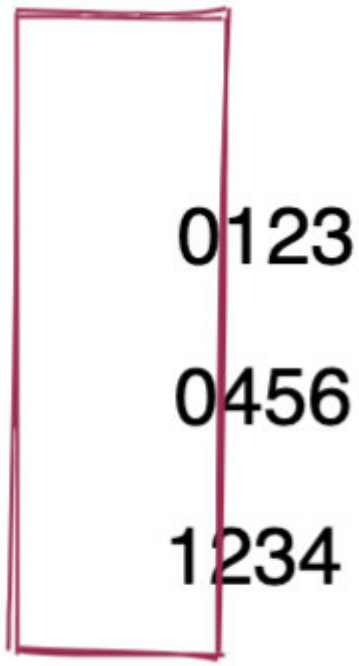
其次要知道一个常识， 即比较两个数的大小， 我们是从高位到低位比较，这样才比较高效。

比如：

```
123
456
1234
```

这三个数比较大小， 为了方便我们先补 0 ，使得大家的位数保持一致。

```
0123
0456
1234
```



先比较第一位，1 比较 0 大， 因此 1234 最大。再比较第二位， 4 比 1 大， 因此 456 大于 123，后面位不需要比较了。这其实就是剪枝的思想。

有了这两个前提，我们来试下暴力法解决这道题。

思路

暴力法就是枚举 $N^2 / 2$ 中组合， 让其两两按位异或，将得到的结果和 m 进行比较， 如果比 m 大， 则计数器 + 1， 最后返回计数器的值即可。

暴力的方法就如同题目描述的那样， 复杂度为 N^2。 一定过不了所有的测试用例， 不过大家实在没有好的解法的情况可以兜底。不管是牛客笔试还是实际的面试都是可行的。

接下来，让我们来**分析一下暴力为什么低效，以及如何选取数据结构和算法能够使得这个过程变得高效。** 记住这句话， 几乎所有的优化都是基于这种思维产生的，除非你开启了上帝模式，直接看了答案。 只不过等你熟悉了之后，这个思维过程会非常短， 以至于变成条件反射， 你感觉不到有这个过程， 这就是**有了题感。**

其实我刚才说的第二个前置知识就是我们优化的关键之一。

我举个例子， 比如 3 和 5 按位异或。

3 的二进制是 011， 5 的二进制是 101，

```
011
101
```

按照我前面讲的异或知识， 不难得出其异或结构就是 110。

上面我进行了三次异或：

1. 第一次是最高位的 0 和 1 的异或， 结果为 1。
2. 第二次是次高位的 1 和 0 的异或， 结果为 1。
3. 第三次是最低位的 1 和 1 的异或， 结果为 0。

那如何 m 是 1 呢？ 我们有必要进行三次异或么？ 实际上进行第一次异或的时候已经知道了一定比 m（m 是 1） 大。因为第一次异或的结构导致其最高位为 1，也就是说其最小也不过是 100，也就是 4，一定是大于 1 的。这就是**剪枝**， 这就是算法优化的关键。

> 看出我一步一步的思维过程了么？所有的算法优化都需要经过类似的过程。

因此我的算法就是从高位开始两两异或，并且异或的结果和 m 对应的二进制位比较大小。

- 如果比 m 对应的二进制位大或者小，我们提前退出即可。
- 如果相等，我们继续往低位移动重复这个过程。

这虽然已经剪枝了，但是极端情况下，性能还是很差。比如：

```
m: 1111
a: 1010
b: 0101
```

a，b 表示两个数，我们比较到最后才发现，其异或的值和 m 相等。因此极端情况，算法效率没有得到改进。

这里我想到了一点，就是如果一个数 a 的前缀和另外一个数 b 的前缀是一样的，那么 c 和 a 或者 c 和 b 的异或的结构前缀部分一定也是一样的。比如：

```
a: 111000
b: 111101
c: 101011
```

a 和 b 有共同的前缀 111，c 和 a 异或过了，当再次和 b 异或的时候，实际上前三位是没有必要进行的，这也是重复的部分。这就是算法可以优化的部分， 这就是剪枝。

分析算法，找到算法的瓶颈部分，然后选取合适的数据结构和算法来优化到。 这句话很重要， 请务必记住。

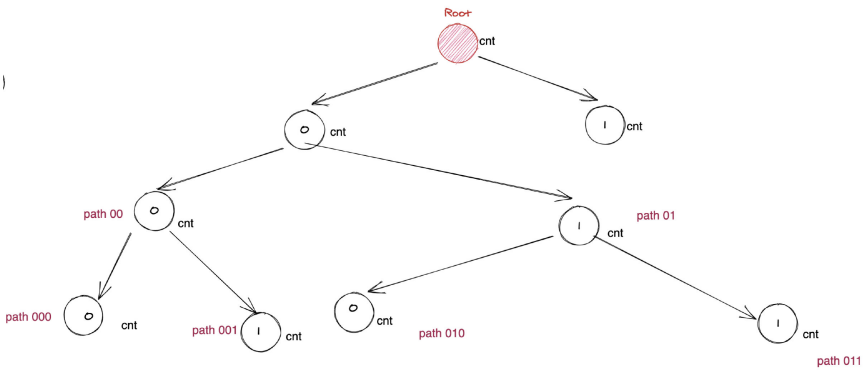
在这里，我们用的就是剪枝技术，关于剪枝，91 天学算法也有详细的介绍。

回到前面讲到的算法瓶颈， 多个数是有共同前缀的， 前缀部分就是我们浪费的运算次数， 说到前缀大家应该可以想到前缀树。如果不熟悉前缀树的话，看下我的这个前缀树专题，里面的题全部手写一遍就差不多了。

因此一种想法就是建立一个前缀树， **树的根就是最高的位**。 由于题目要求异或， 我们知道异或是二进制的位运算， 因此这棵树要存二进制才比较好。

反手看了一眼数据范围：m, n<=10^5 。 10^5 = 2 ^ x，我们的目标是求出满足条件的 x 的 ceil（向上取整），因此 x 应该是 17。

树的每一个节点存储的是：**n 个数中，从根节点到当前节点形成的前缀有多少个是一样的**，即多少个数的前缀是一样的。这样可以剪枝，提前退出的时候，就直接取出来用了。比如异或的结果是 1， m 当前二进制位是 0 ，那么这个前缀有 10 个，我都不需要比较了， 计数器直接 + 10 。



> 我用 17 直接复杂度过高，目前仅仅通过了 70 % - 80 % 测试用例，希望大家可以帮我找找毛病，我猜测是语言的锅。

代码

```
class TreeNode:
    def __init__(self):
        self.cnt = 1
        self.children = [None] * 2
def solve(num, i, cur):
    if cur == None or i == -1: return 0
    bit = (num >> i) & 1
    mbit = (m >> i) & 1
    if bit == 0 and mbit == 0:
        return (cur.children[1].cnt if cur.children[1] else
    if bit == 1 and mbit == 0:
        return (cur.children[0].cnt if cur.children[0] else
    if bit == 0 and mbit == 1:
        return solve(num, i - 1, cur.children[1])
    if bit == 1 and mbit == 1:
        return solve(num, i - 1, cur.children[0])

def preprocess(nums, root):
    for num in nums:
        cur = root
        for i in range(16, -1, -1):
            bit = (num >> i) & 1
            if cur.children[bit]:
                cur.children[bit].cnt += 1
            else:
                cur.children[bit] = TreeNode()
            cur = cur.children[bit]

n, m = map(int, input().split())
nums = list(map(int, input().split()))
root = TreeNode()
preprocess(nums, root)
ans = 0
for num in nums:
    ans += solve(num, 16, root)
print(ans // 2)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

3. 字典序

题目描述

```
给定整数 n 和 m，将 1 到 n 的这 n 个整数按字典序排列之后，求其中的第
对于 n=11，m=4，按字典序排列依次为 1，10，11，2，3，4，5，6，7
对于 n=200，m=25，按字典序排列依次为 1 10 100 101 102 103 104

输入描述：
输入仅包含两个整数 n 和 m。

数据范围：

对于 20%的数据，1 <= m <= n <= 5 ；

对于 80%的数据，1 <= m <= n <= 10^7 ；

对于 100%的数据，1 <= m <= n <= 10^18.

输出描述：
输出仅包括一行，即所求排列中的第 m 个数字.
示例 1
输入
11 4
输出
2
```

前置知识

- 十叉树
- 完全十叉树
- 计算完全十叉树的节点个数
- 字典树

思路

和上面题目思路一样， 先从暴力解法开始，尝试打开思路。

暴力兜底的思路是直接生成一个长度为 n 的数组， 排序，选第 m 个即可。代码：

```
n, m = map(int, input().split())

nums  = [str(i) for i in range(1, n + 1)]
print(sorted(nums)[m - 1])
```

复杂度分析

- 时间复杂度：取决于排序算法， 不妨认为是 $O(NlogN)$
- 空间复杂度: $O(N)$

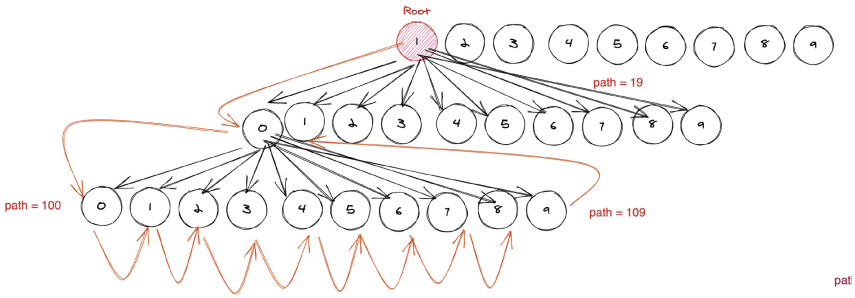
这种算法可以 pass 50 % case。

上面算法低效的原因是开辟了 N 的空间，并对整 N 个 元素进行了排序。

一种简单的优化方法是将排序换成堆，利用堆的特性求第 k 大的数， 这样时间复杂度可以减低到 $mlogN$。

我们继续优化。实际上，你如果把字典序的排序结构画出来， 可以发现他本质就是一个十叉树，并且是一个完全十叉树。

接下来，我带你继续分析。



如图， 红色表示根节点。节点表示一个十进制数， **树的路径存储真正的数字**，比如图上的 100，109 等。 这不就是上面讲的前缀树么?

如图黄色部分， 表示字典序的顺序，注意箭头的方向。因此本质上，**求字典序第 m 个数， 就是求这棵树的前序遍历的第 m 个节点。**

因此一种优化思路就是构建一颗这样的树，然后去遍历。 构建的复杂度是 $O(N)$，遍历的复杂度是 $O(M)$。因此这种算法的复杂度可以达到 $O(max(m, n))$ ，由于 n >= m，因此就是 $O(N)$。

实际上， 这样的优化算法依然是无法 AC 全部测试用例的，会超内存限制。 因此我们的思路只能是不使用 N 的空间去构造树。想想也知道， 由于 N 最大可能为 10^18，一个数按照 4 字节来算， 那么这就有 400000000 字节，大约是 381 M，这是不能接受的。

上面提到这道题就是一个完全十叉树的前序遍历，问题转化为求完全十叉树的前序遍历的第 m 个数。

> 十叉树和二叉树没有本质不同， 我在二叉树专题部分， 也提到了 N 叉树都可以用二叉树来表示。

对于一个节点来说，第 m 个节点：

- 要么就是它本身
- 要么其孩子节点中

- 要么在其兄弟节点
- 要么在兄弟节点的孩子节点中

究竟在上面的四个部分的哪，取决于其孩子节点的个数。

- count > m ， m 在其孩子节点中，我们需要深入到子节点。
- count <= m ， m 不在自身和孩子节点, 我们应该跳过所有孩子节点，直接到兄弟节点。

这本质就是一个递归的过程。

需要注意的是，我们并不会真正的在树上走，因此上面提到的**深入到子节点**， 以及 **跳过所有孩子节点，直接到兄弟节点**如何操作呢?

你仔细观察会发现： 如果当前节点的前缀是 x ，那么其第一个子节点（就是最小的子节点）是 x * 10，第二个就是 x * 10 + 1，以此类推。因此：

- 深入到子节点就是 x * 10。
- 跳过所有孩子节点，直接到兄弟节点就是 x + 1。

ok，铺垫地差不多了。

接下来，我们的重点是**如何计算给定节点的孩子节点的个数**。

这个过程和完全二叉树计算节点个数并无二致，这个算法的时间复杂度应该是 $O(logN*logN)$。 如果不会的同学，可以参考力扣原题： 222. 完全二叉树的节点个数 ，这是一个难度为中等的题目。

> 因此这道题本身被划分为 hard，一点都不为过。

这里简单说下，计算给定节点的孩子节点的个数的思路， 我的 91 天学算法里出过这道题。

一种简单但非最优的思路是分别计算左右子树的深度。

- 如果当前节点的左右子树高度相同，那么左子树是一个满二叉树，右子树是一个完全二叉树。
- 否则（左边的高度大于右边），那么左子树是一个完全二叉树，右子树是一个满二叉树。

如果是满二叉树，当前节点数 是 2 ^ depth，而对于完全二叉树，我们继续递归即可。

```python
class Solution:
    def countNodes(self, root):
        if not root:
            return 0
        ld = self.getDepth(root.left)
        rd = self.getDepth(root.right)
        if ld == rd:
            return 2 ** ld + self.countNodes(root.right)
        else:
            return 2 ** rd + self.countNodes(root.left)

    def getDepth(self, root):
        if not root:
            return 0
        return 1 + self.getDepth(root.left)
```

复杂度分析

- 时间复杂度：$O(logN * log N)$
- 空间复杂度：$O(logN)$

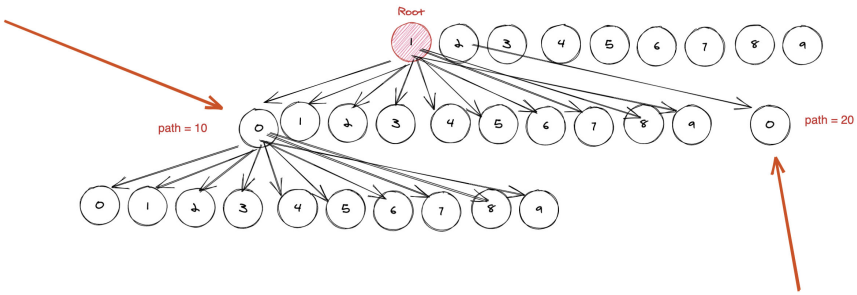
而这道题， 我们可以更简单和高效。

比如我们要计算 1 号节点的子节点个数。

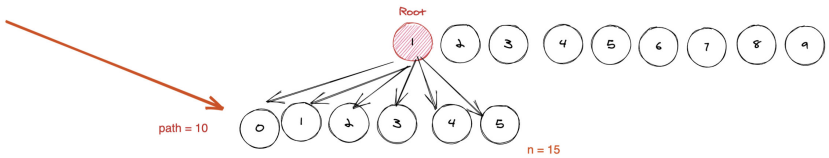
- 它的孩子节点个数是 。。。
- 它的孙子节点个数是 。。。
- 。。。

全部加起来即可。

它的孩子节点个数是 `20 - 10 = 10` 。 也就是它的**右边的兄弟节点的第一个子节点** 减去 它的**第一个子节点**。



由于是完全十叉树，而不是满十叉树 。因此你需要考虑边界情况，比如题目的 n 是 15。 那么 1 的子节点个数就不是 20 - 10 = 10 了， 而是 15 - 10 + 1 = 16。



其他也是类似的过程， 我们只要：

- Go deeper and do the same thing

或者：

- Move to next neighbor and do the same thing

不断重复，直到 m 降低到 0 。

代码

```
def count(c1, c2, n):
    steps = 0
    while c1 <= n:
        steps += min(n + 1, c2) - c1
        c1 *= 10
        c2 *= 10
    return steps
def findKthNumber(n: int, k: int) -> int:
    cur = 1
    k = k - 1
    while k > 0:
        steps = count(cur, cur + 1, n)
        if steps <= k:
            cur += 1
            k -= steps
        else:
            cur *= 10
            k -= 1
    return cur
n, m = map(int, input().split())
print(findKthNumber(n, m))
```

复杂度分析

- 时间复杂度：$O(logM * log N)$
- 空间复杂度：$O(1)$

总结

其中三道算法题从难度上来说，基本都是困难难度。从内容来看，基本都是力扣的换皮题，且都或多或少和树有关。如果大家一开始没有思路，建议大家先给出暴力的解法兜底，再画图或举简单例子打开思路。

我也刷了很多字节的题了，还有一些难度比较大的题。如果你第一次做，那么需要你思考比较久才能想出来。加上面试紧张，很可能做不出来。这个时候就更需要你冷静分析，先暴力打底，慢慢优化。有时候即使给不了最优解，让面试官看出你的思路也很重要。 比如小兔的棋盘 想出最优解难度就不低，不过你可以先暴力 DFS 解决，再 DP 优化会慢慢帮你打开思路。有时候面试官也会引导你，给你提示， 加上你刚才“发挥不错”，说不定一下子就做出最优解了，这个我深有体会。

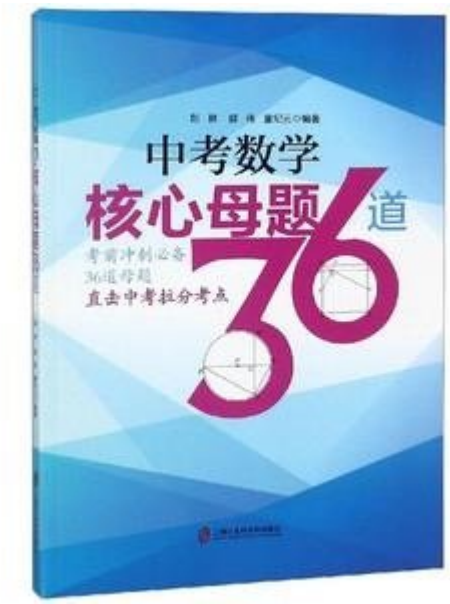
另外要提醒大家的是， 刷题要适量，不要贪多。要完全理清一道题的来龙去脉。多问几个为什么。 这道题暴力法怎么做？暴力法哪有问题？怎么优化？为什么选了这个算法就可以优化？为什么这种算法要用这种数据结构来实现？

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 36K+ star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

《我是你的妈妈呀》 - 第一期

记得我初中的时候，学校发的一个小册子的名字就是母题啥的。



大概意思是市面上的题（尤其是中考题）都是这些母题生的，都是它们的儿子。

熟悉我的朋友应该知道，我有一个风格：”喜欢用通俗易懂的语言以及图片，还原解题过程“。包括我是如何抽象的，如何与其他题目建立联系的等。比如：

- 一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～
- 超级详细记忆化递归，图解，带你一次攻克三道 Hard 套路题（44. 通配符匹配）
- 穿上衣服我就不认识你了？来聊聊最长上升子序列
- 扒一扒这种题的外套（343. 整数拆分）

如果把这个思考过程称之为自顶向下的话，那么实际上能写出来取决于你：

- 是否有良好的抽象能力
- 是否有足够的基础知识
- 是否能与学过的基础知识建立联系

如果反着呢？我先把所有抽象之后的纯粹的东西掌握，也就是母题。那么遇到新的题，我就往上套呗？这就是我在《LeetCode 题解仓库》中所说的**只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。** 这种思路就是**自底向上**。（有点像动态规划?） 市面上的题那么多，但是题目类型就是那几种。甚至出题人出题的时候都是根据以前的题目变个条件，变个说法从而搞出一个“新”的题。

这个专题的目标就是从反的方向来，我们先学习和记忆底层的被抽象过的经典的题目。遇到新的题目，就往这些母题上套即可。

那让我们来自底向上看下第一期的这八道母题吧~

母题 1

题目描述

给你两个有序的非空数组 nums1 和 nums2，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

- 初始化 ans 为无限大
- 使用两个指针，一个指针指向数组 1，一个指针指向数组 2
- 比较两个指针指向的数字的大小，并更新较小的那个的指针，使其向后移动一位。更新的过程顺便计算 ans
- 最后返回 ans

代码

```
def f(nums1, nums2):
    i = j = 0
    ans = float('inf')
    while i < len(nums1) and j < len(nums2):
        ans = min(ans, abs(nums1[i] - nums2[j]))
        if  nums1[i] < nums2[j]:
            i += 1
        else:
            j += 1
    return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

母题 2

题目描述

给你两个非空数组 nums1 和 nums2，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

数组没有说明是有序的，可以选择暴力。两两计算绝对值，返回最小的即可。

代码：

```
def f(nums1, nums2):
    ans = float('inf')
    for num1 in nums1:
        for num2 in nums2:
            ans = min(ans, abs(num1 - num2))
    return ans
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(1)$

由于暴力的时间复杂度是 $O(N^2)$，因此其实也可以先排序将问题转换为母题 1，然后用母题 1 的解法求解。

复杂度分析

- 时间复杂度：$O(NlogN)$
- 空间复杂度：$O(1)$

母题 3

题目描述

给你 k 个有序的非空数组，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

继续使用母题 1 的思路，使用 k 个 指针即可。

复杂度分析

- 时间复杂度：$O(klogM)$，其中 M 为 k 个非空数组的长度的最小值。
- 空间复杂度：$O(1)$

我们也可以使用堆来处理，代码更简单，逻辑更清晰。这里我们使用小顶堆，作用就是选出最小值。

代码

```
def f(matrix):
    ans = float('inf')
    max_value = max(nums[0] for nums in matrix)
    heap = [(nums[0], i, 0) for i, nums in enumerate(nums)]
    heapq.heapify(heap)

    while True:
        min_value, row, idx = heapq.heappop(heap)
        if max_value - min_value < ans:
            ans = max_value - min_value
        if idx == len(matrix[row]) - 1:
            break
        max_value = max(max_value, matrix[row][idx + 1])
        heapq.heappush(heap, (matrix[row][idx + 1], row, id

    return ans
```

复杂度分析

建堆的时间和空间复杂度为 $O(k)$。

while 循环会执行 M 次，其中 M 为 k 个非空数组的长度的最小值。heappop 和 heappush 的时间复杂度都是 logk。因此 while 循环总的时间复杂度为 $O(Mlogk)$。

- 时间复杂度：$O(max(Mlogk, k))$，其中 M 为 k 个非空数组的长度的最小值。
- 空间复杂度：$O(k)$

母题 4

题目描述

给你 k 个非空数组，让你从每个数组中分别挑一个，使得二者差的绝对值最小。

思路

先排序，然后转换为母题 3

母题 5

题目描述

给你两个有序的非空数组 nums1 和 nums2，让你将两个数组合并，使得新的数组有序。

LeetCode 地址： https://leetcode-cn.com/problems/merge-sorted-array/

思路

和母题 1 类似。

代码

```
def f(nums1, nums2):
    i = j = 0
    ans = []
    while i < len(nums1) and j < len(nums2):
        if nums1[i] < nums2[j]:
            ans.append(nums1[i])
            i += 1
        else:
            ans.append(nums2[j])
            j += 1
    if nums1:
        ans += nums2[j:]
    else:
        ans += nums1[i:]
    return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

母题 6

题目描述

给你 k 个有序的非空数组 nums1 和 nums2，让你将 k 个数组合并，使得新的数组有序。

思路

和母题 5 类似。 只不过不是两个，而是多个。我们继续套用堆的思路。

代码

```
import heapq


def f(matrix):
    ans = []
    heap = []
    for row in matrix:
        heap += row
    heapq.heapify(heap)

    while heap:
        cur = heapq.heappop(heap)
        ans.append(cur)

    return ans
```

复杂度分析

建堆的时间和空间复杂度为 $O(N)$。

heappop 的时间复杂度为 $O(logN)$。

- 时间复杂度：$O(NlogN)$，其中 N 是矩阵中的数字总数。
- 空间复杂度：$O(N)$，其中 N 是矩阵中的数字总数。

母题 7

题目描述

给你两个有序的链表 root1 和 root2，让你将两个链表合并，使得新的链表有序。

LeetCode 地址：https://leetcode-cn.com/problems/merge-two-sorted-lists/

思路

和母题 5 类似。 不同的地方在于数据结构从数组变成了链表，我们只需要注意链表的操作即可。

这里我使用了迭代和递归两种方式。

> 大家可以把母题 5 使用递归写一下。

代码

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为两个链表中较短的那个的长度。
- 空间复杂度：$O(N)$，其中 N 为两个链表中较短的那个的长度。

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        ans = cur = ListNode(0)
        while l1 and l2:
            if l1.val < l2.val:
                cur.next = l1
                cur = cur.next
                l1 = l1.next
            else:
                cur.next = l2
                cur = cur.next
                l2 = l2.next


        if l1:
            cur.next = l1
        else:
            cur.next = l2
        return ans.next
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为两个链表中较短的那个的长度。
- 空间复杂度：$O(1)$

母题 8

题目描述

给你 k 个有序的链表，让你将 k 个链表合并，使得新的链表有序。

LeetCode 地址：https://leetcode-cn.com/problems/merge-k-sorted-lists/

思路

和母题 7 类似，我们使用递归可以轻松解决。其实本质上就是

代码

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
    def mergeKLists(self, lists: List[ListNode]) -> ListNo
        if not lists: return None
        if len(lists) == 1: return lists[0]
        return self.mergeTwoLists(lists[0], self.mergeKLis
```

复杂度分析

mergeKLists 执行了 k 次，每次都执行一次 mergeTwoLists，mergeTwoLists 的时间复杂度前面已经分析过了，为 $O(N)$，其中 N 为两个链表中较短的那个的长度。

- 时间复杂度：$O(k * N)$，其中 N 为两个链表中较短的那个的长度
- 空间复杂度：$O(max(k, N))$

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        if not l1: return l2
        if not l2: return l1
        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
            l2.next = self.mergeTwoLists(l1, l2.next)
            return l2
    def mergeKLists(self, lists: List[ListNode]) -> ListNo
        if not lists: return None
        if len(lists) == 1: return lists[0]
        return self.mergeTwoLists(self.mergeKLists(lists[:
```

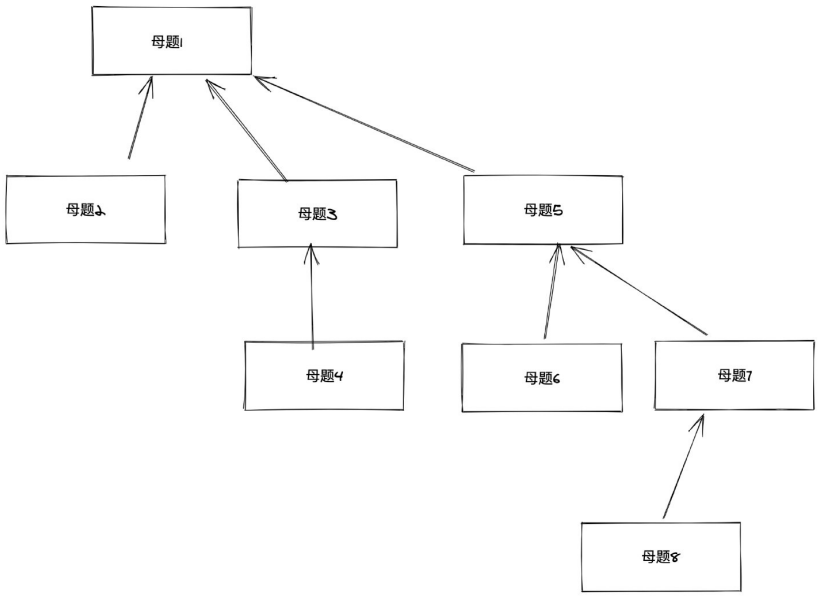
复杂度分析

mergeKLists 执行了 logk 次，每次都执行一次 mergeTwoLists，mergeTwoLists 的时间复杂度前面已经分析过了，为 $O(N)$，其中 N 为两个链表中较短的那个的长度。

- 时间复杂度：$O(Nlogk)$，其中 N 为两个链表中较短的那个的长度
- 空间复杂度：$O(max(logk, N))$，其中 N 为两个链表中较短的那个的长度

全家福

最后送大家一张全家福：



子题

实际子题数量有很多，这里提供几个供大家练习。一定要练习，不能眼高手低。多看我的题解，多练习，多总结，你也可以的。

- 面试题 17.14. 最小 K 个数
- 1200. 最小绝对差
- 632. 最小区间
- 两数和，三数和，四数和。。。 k 数和

总结

母题就是**抽象之后的纯粹的东西**。如果你掌握了母题，即使没有掌握抽象的能力，依然有可能套出来。但是随着题目做的变多，“抽象能力”也会越来越强。因为你知道这些题背后是怎么产生的。

本期给大家介绍了八道母题， 大家可以在之后的刷题过程中尝试使用母题来套模板。之后会给大家带来更多的母题。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

一文带你看懂二叉树的序列化

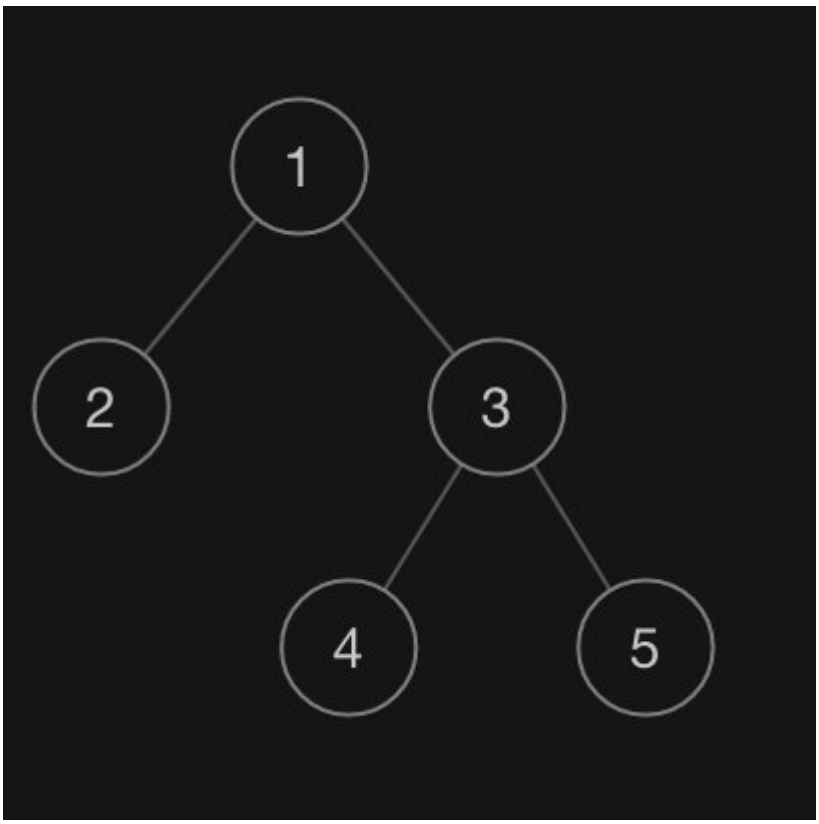
我们先来看下什么是序列化，以下定义来自维基百科：

> 序列化（serialization）在计算机科学的数据处理中，是指将数据结构或对象状态转换成可取用格式（例如存成文件，存于缓冲，或经由网络中发送），以留待后续在相同或另一台计算机环境中，能恢复原先状态的过程。依照序列化格式重新获取字节的结果时，可以利用它来产生与原始对象相同语义的副本。对于许多对象，像是使用大量引用的复杂对象，这种序列化重建的过程并不容易。面向对象中的对象序列化，并不概括之前原始对象所关系的函数。这种过程也称为对象编组（marshalling）。从一系列字节提取数据结构的反向操作，是反序列化（也称为解编组、deserialization、unmarshalling）。

可见，序列化和反序列化在计算机科学中的应用还是非常广泛的。就拿 LeetCode 平台来说，其允许用户输入形如：

```
[1,2,3,null,null,4,5]
```

这样的数据结构来描述一颗树：



([1,2,3,null,null,4,5] 对应的二叉树)

其实序列化和反序列化只是一个概念，不是一种具体的算法，而是很多的算法。并且针对不同的数据结构，算法也会不一样。本文主要讲述的是二叉树的序列化和反序列化。看完本文之后，你就可以放心大胆地去 AC 以下两道题：

- 449. 序列化和反序列化二叉搜索树(中等)
- 297. 二叉树的序列化与反序列化(困难)

前置知识

阅读本文之前，需要你对树的遍历以及 BFS 和 DFS 比较熟悉。如果你还不熟悉，推荐阅读一下相关文章之后再来看。或者我这边也写了一个总结性的文章二叉树的遍历，你也可以看看。

前言

我们知道：二叉树的深度优先遍历，根据访问根节点的顺序不同，可以将其分为 `前序遍历` ， `中序遍历` ， `后序遍历` 。即如果先访问根节点就是前序遍历，最后访问根节点就是后续遍历，其它则是中序遍历。而左右节点的相对顺序是不会变的，一定是先左后右。

> 当然也可以设定为先右后左。

并且知道了三种遍历结果中的任意两种即可还原出原有的树结构。这不就是序列化和反序列化么？如果对这个比较陌生的同学建议看看我之前写的《构造二叉树系列》

有了这样一个前提之后算法就自然而然了。即先对二叉树进行两次不同的遍历，不妨假设按照前序和中序进行两次遍历。然后将两次遍历结果序列化，比如将两次遍历结果以逗号“,” join 成一个字符串。 之后将字符串反序列即可，比如将其以逗号“,” split 成一个数组。

序列化：

```
class Solution:
    def preorder(self, root: TreeNode):
        if not root: return []
        return [str(root.val)] +self. preorder(root.left) +
    def inorder(self, root: TreeNode):
        if not root: return []
        return  self.inorder(root.left) + [str(root.val)] +
    def serialize(self, root):
        ans = ''
        ans += ','.join(self.preorder(root))
        ans += '$'
        ans += ','.join(self.inorder(root))

        return ans
```

反序列化：

这里我直接用了力扣 `105. 从前序与中序遍历序列构造二叉树` 的解法，一行代码都不改。

```
class Solution:
    def deserialize(self, data: str):
        preorder, inorder = data.split('$')
        if not preorder: return None
        return self.buildTree(preorder.split(','), inorder.

    def buildTree(self, preorder: List[int], inorder: List
        # 实际上inorder 和 preorder 一定是同时为空的，因此你无论
        if not preorder:
            return None
        root = TreeNode(preorder[0])

        i = inorder.index(root.val)
        root.left = self.buildTree(preorder[1:i + 1], inor
        root.right = self.buildTree(preorder[i + 1:], inor

        return root
```

实际上这个算法是不一定成立的，原因在于树的节点可能存在重复元素。也就是说我前面说的 `知道了三种遍历结果中的任意两种即可还原出原有的树结构` 是不对的，严格来说应该是**如果树中不存在重复的元素，那么知道了三种遍历结果中的任意两种即可还原出原有的树结构**。

聪明的你应该发现了，上面我的代码用了 `i = inorder.index(root.val)` ，如果存在重复元素，那么得到的索引 i 就可能不是准确的。但是，如果题目限定了没有重复元素则可以用这种算

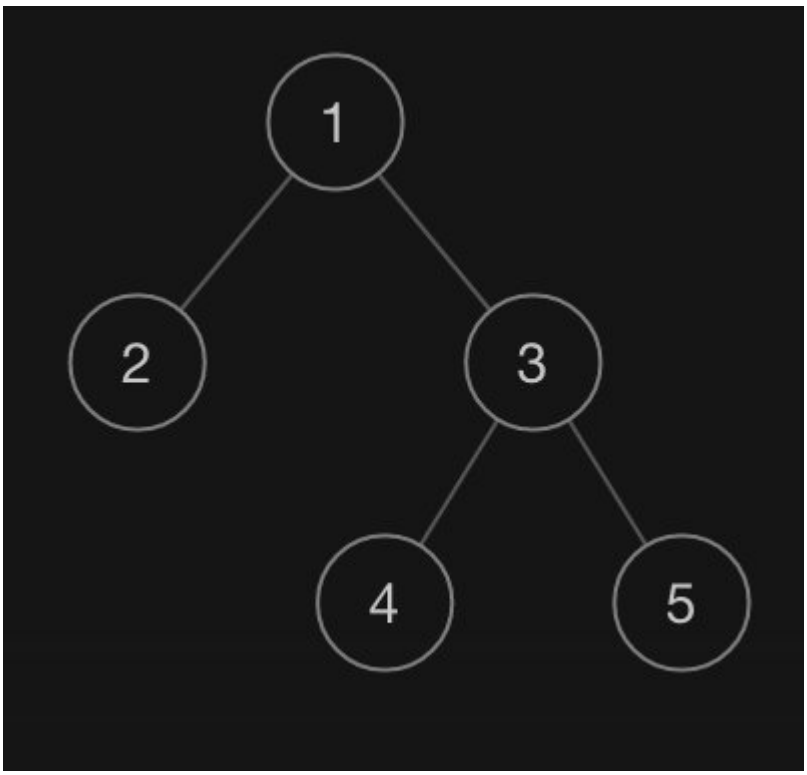
法。但是现实中不出现重复元素不太现实，因此需要考虑其他方法。那究竟是什么样的方法呢? 接下来进入正题。

DFS

序列化

我们来模仿一下力扣的记法。 比如：`[1,2,3,null,null,4,5]`(本质上是 BFS 层次遍历)，对应的树如下：

> 选择这种记法，而不是 DFS 的记法的原因是看起来比较直观



序列化的代码非常简单， 我们只需要在普通的遍历基础上，增加对空节点的输出即可（普通的遍历是不处理空节点的）。

比如我们都树进行一次前序遍历的同时增加空节点的处理。选择前序遍历的原因是容易知道根节点的位置，并且代码好写，不信你可以试试。

因此序列化就仅仅是普通的 DFS 而已，直接给大家看看代码。

Python 代码：

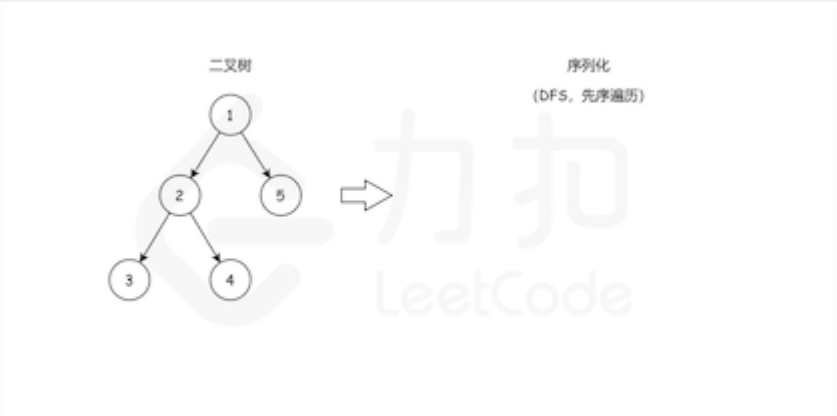
```python
class Codec:
    def serialize_dfs(self, root, ans):
        # 空节点也需要序列化，否则无法唯一确定一棵树，后不赘述。
        if not root: return ans + '#,'
        # 节点之间通过逗号（,）分割
        ans += str(root.val) + ','
        ans = self.serialize_dfs(root.left, ans)
        ans = self.serialize_dfs(root.right, ans)
        return ans
    def serialize(self, root):
        # 由于最后会添加一个额外的逗号，因此需要去除最后一个字符，后
        return self.serialize_dfs(root, '')[:-1]
```

Java 代码：

```java
public class Codec {
    public String serialize_dfs(TreeNode root, String str)
        if (root == null) {
            str += "None,";
        } else {
            str += str.valueOf(root.val) + ",";
            str = serialize_dfs(root.left, str);
            str = serialize_dfs(root.right, str);
        }
        return str;
    }

    public String serialize(TreeNode root) {
        return serialize_dfs(root, "");
    }
}
```

`[1,2,3,null,null,4,5]` 会被处理为 `1,2,#,#,3,4,#,#,5,#,#`

我们先看一个短视频：



（动画来自力扣）

反序列化

反序列化的第一步就是将其展开。以上面的例子来说，则会变成数组：`[1,2,#,#,3,4,#,#,5,#,#]`，然后我们同样执行一次前序遍历，每次处理一个元素，重建即可。由于我们采用的前序遍历，因此第一个是根元素，下一个是其左子节点，下下一个是其右子节点。

Python 代码：

```python
    def deserialize_dfs(self, nodes):
        if nodes:
            if nodes[0] == '#':
                nodes.pop(0)
                return None
            root = TreeNode(nodes.pop(0))
            root.left = self.deserialize_dfs(nodes)
            root.right = self.deserialize_dfs(nodes)
            return root
        return None

    def deserialize(self, data: str):
        nodes = data.split(',')
        return self.deserialize_dfs(nodes)
```

Java 代码：

```
    public TreeNode deserialize_dfs(List<String> l) {
        if (l.get(0).equals("None")) {
            l.remove(0);
            return null;
        }

        TreeNode root = new TreeNode(Integer.valueOf(l.get(
        l.remove(0);
        root.left = deserialize_dfs(l);
        root.right = deserialize_dfs(l);

        return root;
    }

    public TreeNode deserialize(String data) {
        String[] data_array = data.split(",");
        List<String> data_list = new LinkedList<String>(Ar
        return deserialize_dfs(data_list);
    }
```

复杂度分析

- 时间复杂度：每个节点都会被处理一次，因此时间复杂度为 $O(N)$，其中 N 为节点的总数。
- 空间复杂度：空间复杂度取决于栈深度，因此空间复杂度为 $O(h)$，其中 h 为树的深度。

BFS

序列化

实际上我们也可以使用 BFS 的方式来表示一棵树。在这一点上其实就和力扣的记法是一致的了。

我们知道层次遍历的时候实际上是有层次的。只不过有的题目需要你记录每一个节点的层次信息，有些则不需要。

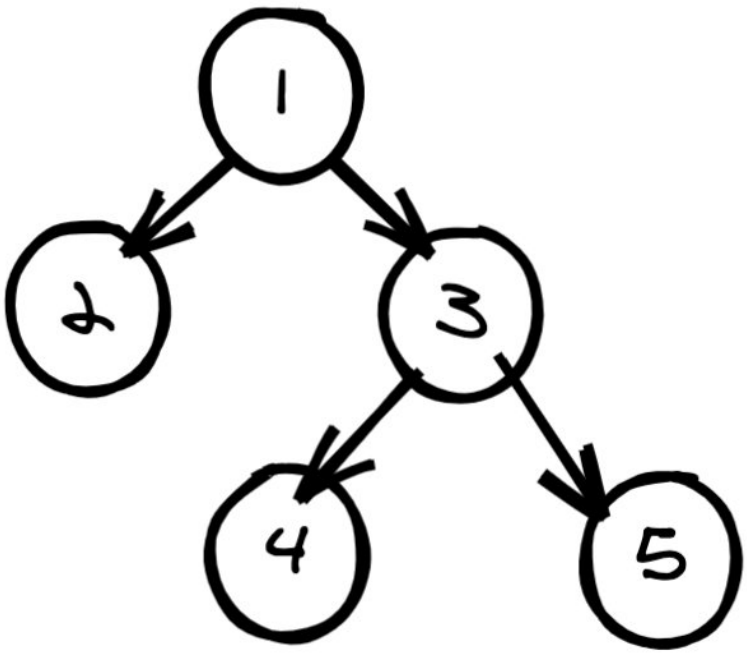
这其实就是一个朴实无华的 BFS，唯一不同则是增加了空节点。

Python 代码：

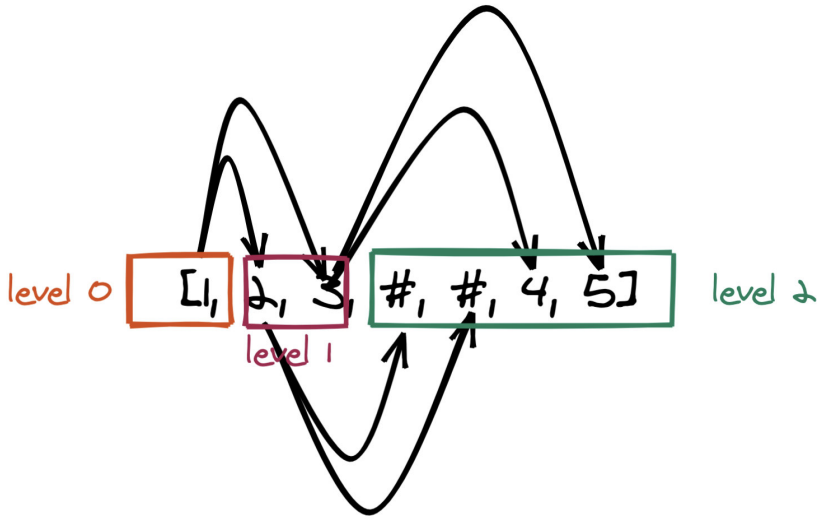
```
class Codec:
    def serialize(self, root):
        ans = ''
        queue = [root]
        while queue:
            node = queue.pop(0)
            if node:
                ans += str(node.val) + ','
                queue.append(node.left)
                queue.append(node.right)
            else:
                ans += '#,'
        return ans[:-1]
```

反序列化

如图有这样一棵树：



那么其层次遍历为 [1,2,3,#,#, 4, 5]。我们根据此层次遍历的结果来看下如何还原二叉树，如下是我画的一个示意图：



容易看出：

- level x 的节点一定指向 level x + 1 的节点，如何找到 level + 1 呢？这很容易通过层次遍历来做到。
- 对于给的的 level x，从左到右依次对应 level x + 1 的节点，即第 1 个节点的左右子节点对应下一层的第 1 个和第 2 个节点，第 2 个节点的左右子节点对应下一层的第 3 个和第 4 个节点。。。
- 接上，其实如果你仔细观察的话，实际上 level x 和 level x + 1 的判断是无需特别判断的。我们可以把思路逆转过来：`即第 1 个节点的左右子节点对应第 1 个和第 2 个节点，第 2 个节点的左右子节点对应第 3 个和第 4 个节点。。。`（注意，没了下一层三个字）

因此我们的思路也是同样的 BFS，并依次连接左右节点。

Python 代码：

```
    def deserialize(self, data: str):
        if data == '#': return None
        # 数据准备
        nodes = data.split(',')
        if not nodes: return None
        # BFS
        root = TreeNode(nodes[0])
        queue = [root]
        # 已经有 root 了，因此从 1 开始
        i = 1

        while i < len(nodes) - 1:
            node = queue.pop(0)
            #
            lv = nodes[i]
            rv = nodes[i + 1]
            i += 2
            # 对于给的的 level x，从左到右依次对应 level x + 1
            # node 是 level x 的节点，l 和 r 则是 level x + 1
            if lv != '#':
                l = TreeNode(lv)
                node.left = l
                queue.append(l)

            if rv != '#':
                r = TreeNode(rv)
                node.right = r
                queue.append(r)
        return root
```

复杂度分析

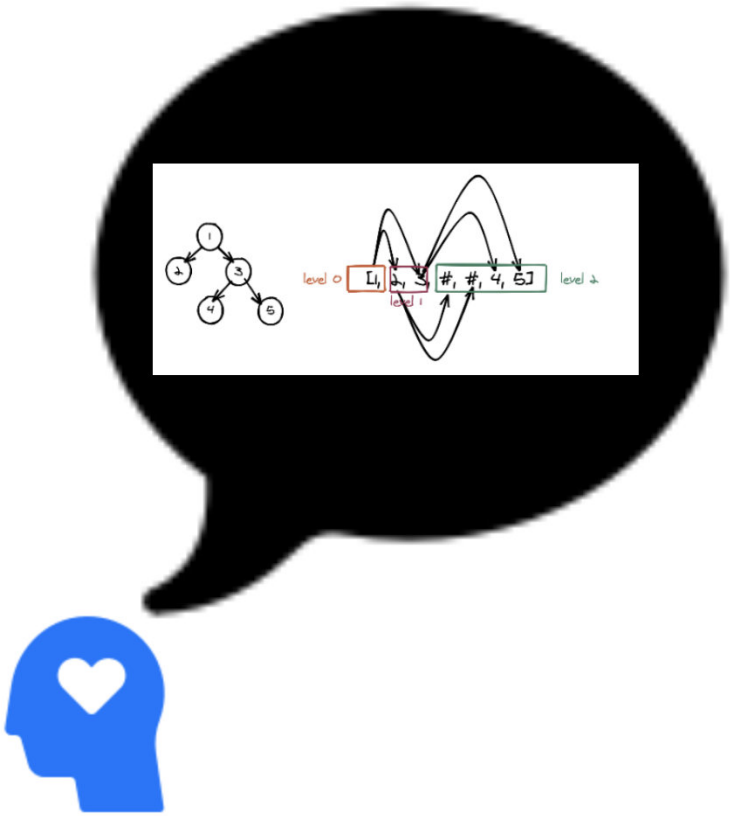
- 时间复杂度：每个节点都会被处理一次，因此时间复杂度为 $O(N)$，其中 N 为节点的总数。
- 空间复杂度：$O(N)$，其中 N 为节点的总数。

总结

除了这种方法还有很多方案， 比如括号表示法。 关于这个可以参考力扣 606. 根据二叉树创建字符串，这里就不再赘述了。

本文从 BFS 和 DFS 角度来思考如何序列化和反序列化一棵树。 如果用 BFS 来序列化，那么相应地也需要 BFS 来反序列化。如果用 DFS 来序列化，那么就需要用 DFS 来反序列化。

我们从马后炮的角度来说，实际上对于序列化来说，BFS 和 DFS 都比较常规。对于反序列化，大家可以像我这样举个例子，画一个图。可以先在纸上，电脑上，如果你熟悉了之后，也可以画在脑子里。



（Like This）

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

穿上衣服我就不认识你了？来聊聊最长上升子序列

最长上升子序列是一个很经典的算法题。有的会直接让你求最长上升子序列，有的则会换个说法，但最终考察的还是最长上升子序列。那么问题来了，它穿上衣服你还看得出来是么？

如果你完全看不出来了，说明抽象思维还不到火候。经常看我的题解的同学应该会知道，我经常强调 `抽象思维` 。没有抽象思维，所有的题目对你来说都是新题。你无法将之前做题的经验迁移到这道题，那你做的题意义何在？

虽然抽象思维很难练成，但是幸好算法套路是有限的，经常考察的题型更是有限的。从这些入手，或许可以让你轻松一些。本文就从一个经典到不行的题型《最长上升子序列》，来帮你进一步理解 `抽象思维` 。

> 注意。 本文是帮助你识别套路，从横向上理清解题的思维框架，并没有采用最优解，所有的题目给的解法都不是最优的，但是都可以通过所有的测试用例。如果你想看最优解，可以直接去讨论区看。或者期待我的 `深入剖析系列` 。

300. 最长上升子序列

题目地址

https://leetcode-cn.com/problems/longest-increasing-subsequence

题目描述

```
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(n2) 。
进阶: 你能将算法的时间复杂度降低到 O(n log n) 吗?
```

思路

> 美团和华为都考了这个题。

题目的意思是让我们从给定数组中挑选若干数字，这些数字满足：`如果 i < j 则 nums[i] < nums[j]`。问：一次可以挑选最多满足条件的数字是多少个。

最长上升子序列 @lucifer

这种子序列求极值的题目，应该要考虑到贪心或者动态规划。这道题贪心是不可以的，我们考虑动态规划。

按照动态规划定义状态的套路，我们有**两种常见**的定义状态的方式：

- dp[i] : 以 i 结尾（一定包括 i）所能形成的最长上升子序列长度, 答案是 max(dp[i])，其中 i = 0,1,2, ..., n - 1
- dp[i] : 以 i 结尾（可能包括 i）所能形成的最长上升子序列长度，答案是 dp[-1] （-1 表示最后一个元素）

容易看出第二种定义方式由于无需比较不同的 dp[i] 就可以获得答案，因此更加方便。但是想了下，状态转移方程会很不好写，因为 dp[i] 的末尾数字（最大的）可能是 任意 j < i 的位置。

第一种定义方式虽然需要比较不同的 dp[i] 从而获得结果，但是我们可以在循环的时候顺便得出，对复杂度不会有影响，只是代码多了一点而已。因此我们**选择第一种建模方式**。

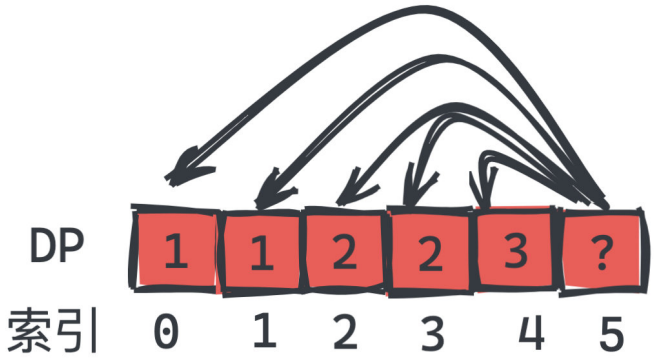
`dp[i] : 以 i 结尾（一定包括 i）所能形成的最长上升子序列`

`ans 不一定在 dp[-1] 产生`
`而是 max(dp[i]), 其中 i = 0, 1, 2, n - 1`

由于 dp[j] 中一定会包括 j，且以 j 结尾， 那么 nums[j] 一定是其所形成的序列中最大的元素，那么如果位于其后（意味着 i > j）的 nums[i] > nums[j]，那么 nums[i] 一定能够融入 dp[j] 从而形成更大的序列，这个序列的长度是 dp[j] + 1。因此状态转移方程就有了：`dp[i] = dp[j] + 1 (其中 i > j, nums[i] > nums[j])`

以 `[10, 9, 2, 5, 3, 7, 101, 18]` 为例，当我们计算到 dp[5]的时候，我们需要往回和 0，1，2，3，4 进行比较。

```
nums [10, 9, 2, 5, 3, 7, 101, 18]
```



具体的比较内容是：

```
nums[5] > nums[0] no
nums[5] > nums[1] no
nums[5] > nums[2] yes
nums[5] > nums[3] yes
nums[5] > nums[4] yes
```

最后从三个中选一个最大的 + 1 赋给 dp[5]即可。

```
max(dp[2], dp[3], dp[4])+ 1
```

记住这个状态转移方程，后面我们还会频繁用到。

代码

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0: return 0
        dp = [1] * n
        ans = 1
        for i in range(n):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    ans = max(ans, dp[i])
        return  ans
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(N)$

435. 无重叠区间

题目地址

https://leetcode-cn.com/problems/non-overlapping-intervals/

题目描述

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

可以认为区间的终点总是大于它的起点。
区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。
示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。
示例 2:

输入: [[1,2], [1,2], [1,2]]

输出: 2

解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。
示例 3:

输入: [[1,2], [2,3]]

输出: 0

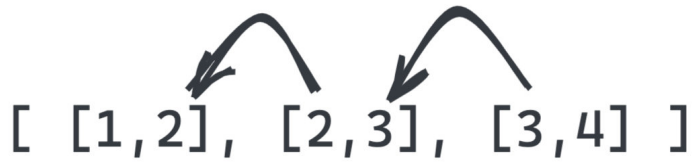
解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

思路

我们先来看下最终**剩下**的区间。由于剩下的区间都是不重叠的，因此剩下的**相邻区间的后一个区间的开始时间一定是不小于前一个区间的结束时间的**。 比如我们剩下的区间是 `[ [1,2], [2,3], [3,4] ]` 。就是第一个区间的 2 小于等于 第二个区间的 2，第二个区间的 3 小于等于第三个区间的 3。

不难发现如果我们将 `前面区间的结束` 和 `后面区间的开始` 结合起来看，其就是一个**非严格递增序列**。而我们的目标就是删除若干区间，从而**剩下最长的非严格递增子序列**。这不就是上面的题么？只不过上面是严格递增，这不重要，就是改个符号的事情。 上面的题你可以看成是删除了若干数字，然后剩下**剩下最长的严格递增子序列**。 **这就是抽象的力量，这就是套路。**

如果对区间按照起点或者终点进行排序，那么就转化为上面的最长递增子序列问题了。和上面问题不同的是，由于是一个区间。因此实际上，我们是需要拿**后面的开始时间**和**前面的结束时间**进行比较。



而由于：

- 题目求的是需要移除的区间，因此最后 return 的时候需要做一个转化。
- 题目不是要求严格递增，而是可以相等，因此我们的判断条件要加上等号。

> 这道题还有一种贪心的解法，其效率要比动态规划更好，但由于和本文的主题不一致，就不在这里讲了。

代码

你看代码多像

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[i
        n = len(intervals)
        if n == 0: return 0
        dp = [1] * n
        ans = 1
        intervals.sort(key=lambda a: a[1])

        for i in range(len(intervals)):
            for j in range(i - 1, -1, -1):
                if intervals[i][0] >= intervals[j][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    # 由于我事先进行了排序，因此倒着找的时候，找到
                    # 这也是为什么我按照结束时间排序的原因。
                    break
            dp[i] = max(dp[i], dp[i - 1])
            ans = max(ans, dp[i])

        return n - ans
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(N)$

646. 最长数对链

题目地址

https://leetcode-cn.com/problems/maximum-length-of-pair-chain/

题目描述

```
给出 n 个数对。 在每一个数对中，第一个数字总是比第二个数字小。

现在，我们定义一种跟随关系，当且仅当 b < c 时，数对(c, d) 才可以跟在

给定一个对数集合，找出能够形成的最长数对链的长度。你不需要用到所有的数

示例 :

输入: [[1,2], [2,3], [3,4]]
输出: 2
解释: 最长的数对链是 [1,2] -> [3,4]
注意:

给出数对的个数在 [1, 1000] 范围内。
```

思路

和上面的 `435. 无重叠区间` 是换皮题，唯一的区别这里又变成了严格增加。没关系，我们把等号去掉就行了。并且由于这道题求解的是最长的长度，因此转化也不需要了。

> 当然，这道题也有一种贪心的解法，其效率要比动态规划更好，但由于和本文的主题不一致，就不在这里讲了。

代码

这代码更像了！

```
class Solution:
    def findLongestChain(self, pairs: List[List[int]]) -> 
        n = len(pairs)
        dp = [1] * n
        ans = 1
        pairs.sort(key=lambda a: a[0])
        for i in range(n):
            for j in range(i):
                if pairs[i][0] > pairs[j][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    ans = max(ans, dp[i])
        return ans
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(N)$

452. 用最少数量的箭引爆气球

题目地址

https://leetcode-cn.com/problems/minimum-number-of-arrows-to-burst-balloons/

题目描述

```
在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一

Example:

输入:
[[10,16], [2,8], [1,6], [7,12]]

输出:
2

解释:
对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（
```

思路

把气球看成区间，几个箭可以全部射爆，意思就是有多少不重叠的区间。注意这里重叠的情况也可以射爆。这么一抽象，就和上面的 `646. 最长数对链` 一模一样了，不用我多说了吧?

> 当然，这道题也有一种贪心的解法，其效率要比动态规划更好，但由于和本文的主题不一致，就不在这里讲了。

代码

代码像不像?

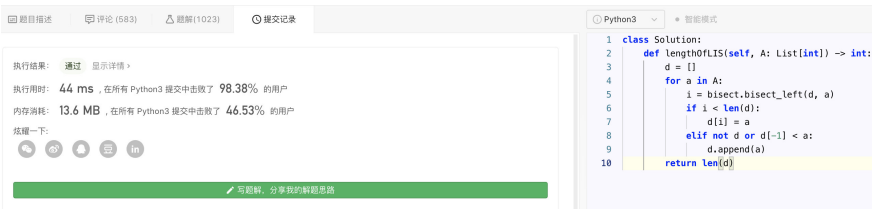
```python
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) ->
        n = len(points)
        if n == 0: return 0
        dp = [1] * n
        cnt = 1
        points.sort(key=lambda a:a[1])

        for i in range(n):
            for j in range(0, i):
                if points[i][0] > points[j][1]:
                    dp[i] = max(dp[i], dp[j] + 1)
                    cnt = max(cnt, dp[i])
        return cnt
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(N)$

优化

大家想看效率高的，其实也不难。 LIS 也可以用 **贪心 + 二分** 达到不错的效率。代码如下：



代码文字版如下：

```
class Solution:
    def lengthOfLIS(self, A: List[int]) -> int:
        d = []
        for a in A:
            i = bisect.bisect_left(d, a)
            if i < len(d):
                d[i] = a
            elif not d or d[-1] < a:
                d.append(a)
        return len(d)
```

More

其他的我就不一一说了。

比如 673. 最长递增子序列的个数 （滴滴面试题）。 不就是求出最长序列，之后再循环比对一次就可以得出答案了么?

491. 递增子序列 由于需要找到所有的递增子序列，因此动态规划就不行了，妥妥回溯就行了，套一个模板就出来了。回溯的模板可以看我之前写的回溯专题。

最后推荐两道题大家练习一下，别看它们是 hard， 其实掌握了我这篇文章的内容一点都不难。

- 面试题 08.13. 堆箱子

参考代码：

```
class Solution:
    def pileBox(self, box: List[List[int]]) -> int:
        box = sorted(box, key=sorted)
        n = len(box)
        dp = [0 if i == 0 else box[i - 1][2] for i in range
        ans = max(dp)

        for i in range(1, n + 1):
            for j in range(i + 1, n + 1):
                if box[j - 1][0] > box[i - 1][0] and box[j
                    dp[j] = max(dp[j], dp[i] + box[j - 1][2
                    ans = max(ans , dp[j])
        return ans
```

- 354. 俄罗斯套娃信封问题

参考代码：

```
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> 
        if not envelopes: return 0
        n = len(envelopes)
        dp = [1] * n
        envelopes.sort()
        for i in range(n):
            for j in range(i + 1, n):
                if envelopes[i][0] < envelopes[j][0] and en
                    dp[j] = max(dp[j], dp[i] + 1)
        return max(dp)
```

- 960. 删列造序 III

参考代码：

```
class Solution:
    def minDeletionSize(self, A):
        keep = 1
        m, n = len(A), len(A[0])
        dp = [1] * n
        for j in range(n):
            for k in range(j + 1, n):
                if all([A[i][k] >= A[i][j] for i in range(m
                    dp[k] = max(dp[k], dp[j] + 1)
                    keep = max(keep, dp[k])
        return n - keep
```

> 小任务：请尝试使用贪心在 NlogN 的时间内完成算法。（参考我上面的代码就行）

- 5644. 得到子序列的最少操作次数

由于这道题数据范围是 10^5，因此只能使用 $NlogN$ 的贪心才行。

> 关于为什么 10 ^ 5 就必须使用 $NlogN$ 甚至更优的算法我在刷题技巧提过。更多复杂度速查可参考我的刷题插件，公众号《力扣加加》回复插件获取即可。

参考代码：

```python
class Solution:
    def minOperations(self, target: List[int], A: List[int]
        def LIS(A):
            d = []
            for a in A:
                i = bisect.bisect_left(d, a)
                if d and i < len(d):
                    d[i] = a
                else:
                    d.append(a)
            return len(d)
        B = []
        target = { t:i for i, t in enumerate(target)}
        for a in A:
            if a in target: B.append(target[a])
        return len(target) - LIS(B)
```

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。

欢迎长按关注

你的衣服我扒了 - 《最长公共子序列》

之前出了一篇穿上衣服我就不认识你了？来聊聊最长上升子序列，收到了大家的一致好评。今天给大家带来的依然是换皮题 - 最长公共子序列系列。

最长公共子序列是一个很经典的算法题。有的会直接让你求最长上升子序列，有的则会换个说法，但最终考察的还是最长公共子序列。那么问题来了，它穿上衣服你还看得出来是么？

如果你完全看不出来了，说明抽象思维还不到火候。经常看我的题解的同学应该会知道，我经常强调 `抽象思维` 。没有抽象思维，所有的题目对你来说都是新题。你无法将之前做题的经验迁移到这道题，那你做的题意义何在？

虽然抽象思维很难练成，但是幸好算法套路是有限的，经常考察的题型更是有限的。从这些入手，或许可以让你轻松一些。本文就从一个经典到不行的题型《最长公共子序列》，来帮你进一步理解 `抽象思维` 。

> 注意。 本文是帮助你识别套路，从横向上理清解题的思维框架，并没有采用最优解，所有的题目给的解法可能不是最优的，但是都可以通过所有的测试用例。如果你想看最优解，可以直接去讨论区看。或者期待我的 `深入剖析系列` 。

718. 最长重复子数组

题目地址

https://leetcode-cn.com/problems/maximum-length-of-repeated-subarray/

题目描述

```
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

示例 1:

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]。
说明:

1 <= len(A), len(B) <= 1000
0 <= A[i], B[i] < 100
```

前置知识

- 哈希表
- 数组
- 二分查找
- 动态规划

思路

这就是最经典的最长公共子序列问题。一般这种求解**两个数组或者字符串求最大或者最小**的题目都可以考虑动态规划，并且通常都定义 dp[i][j] 为 `以 A[i], B[j] 结尾的 xxx` 。这道题就是： `以 A[i], B[j] 结尾的两个数组中公共的、长度最长的子数组的长度` 。

> 关于状态转移方程的选择可以参考： 穿上衣服我就不认识你了？来聊聊最长上升子序列

算法很简单：

- 双层循环找出所有的 i, j 组合，时间复杂度 $O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
  - 如果 A[i] == B[j]，dp[i][j] = dp[i - 1][j - 1] + 1
  - 否则，dp[i][j] = 0
- 循环过程记录最大值即可。

记住这个状态转移方程，后面我们还会频繁用到。

关键点解析

- dp 建模套路

代码

代码支持：Python

Python Code:

```
class Solution:
    def findLength(self, A, B):
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
        return ans
```

复杂度分析

- 时间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。

> 二分查找也是可以的，不过并不容易想到，大家可以试试。

1143.最长公共子序列

题目地址

https://leetcode-cn.com/problems/longest-common-subsequence

题目描述

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。

提示:

1 <= text1.length <= 1000 1 <= text2.length <= 1000 输入的字符串只含有小写英文字符。

前置知识

- 数组
- 动态规划

思路

和上面的题目类似，只不过数组变成了字符串（这个无所谓），子数组（连续）变成了子序列 （非连续）。

算法只需要一点小的微调： `如果 A[i] != B[j]，那么 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])`

关键点解析

- dp 建模套路

代码

> 你看代码多像

代码支持：Python

Python Code:

```
class Solution:
    def longestCommonSubsequence(self, A: str, B: str) -> 
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j -
        return ans
```

复杂度分析

- 时间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。

1035. 不相交的线

题目地址

https://leetcode-cn.com/problems/uncrossed-lines/description/

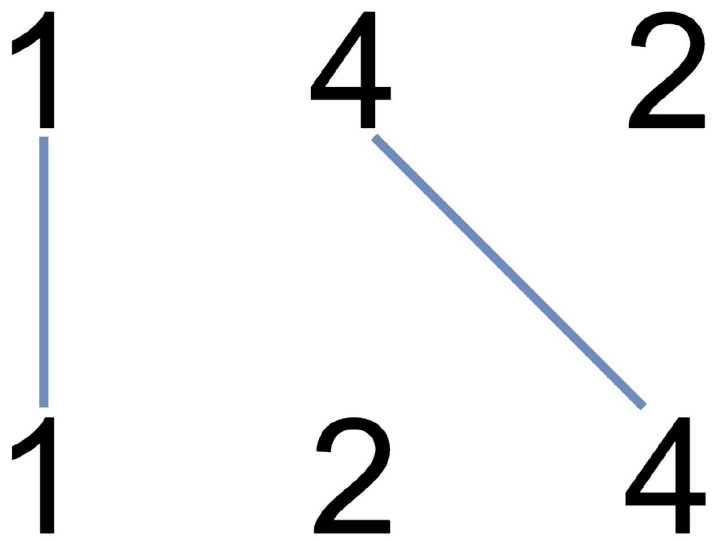
题目描述

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 <= A.length <= 500 1 <= B.length <= 500 1 <= A[i], B[i] <= 2000

前置知识

- 数组
- 动态规划

思路

从图中可以看出，如果想要不相交，则必然相对位置要一致，换句话说就是：**公共子序列**。因此和上面的 `1143.最长公共子序列` 一样，属于换皮题，代码也是一模一样。

关键点解析

- dp 建模套路

代码

> 你看代码多像

代码支持：Python

Python Code:

```
class Solution:
    def longestCommonSubsequence(self, A: str, B: str) -> i
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i][j -
        return ans
```

复杂度分析

- 时间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。

总结

第一道是“子串”题型，第二和第三则是“子序列”。不管是“子串”还是“子序列”，状态定义都是一样的，不同的只是一点小细节。

只有熟练掌握基础的数据结构与算法，才能对复杂问题迎刃有余。 基础算法，把它彻底搞懂，再去面对出题人的各种换皮就不怕了。相反，如果你不去思考题目背后的逻辑，就会刷地很痛苦。题目稍微一变化你就不会了，这也是为什么很多人说**刷了很多题，但是碰到新的题目还是不会做**的原因之一。关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

欢迎长按关注

一文看懂《最大子序列和问题》

最大子序列和是一道经典的算法题， leetcode 也有原题《53.maximum-sum-subarray》，今天我们就来彻底攻克它。

题目描述

求取数组中最大连续子序列和，例如给定数组为 A = [1， 3， -2， 4，-5]， 则最大连续子序列和为 6，即 1 + 3 +（-2）+ 4 = 6。 去

首先我们来明确一下题意。

- 题目说的子数组是连续的
- 题目只需要求和，不需要返回子数组的具体位置。
- 数组中的元素是整数，但是可能是正数，负数和 0。
- 子序列的最小长度为 1。

比如：

- 对于数组 [1, -2, 3, 5, -3, 2], 应该返回 3 + 5 = 8
- 对于数组 [0, -2, 3, 5, -1, 2], 应该返回 3 + 5 + -1 + 2 = 9
- 对于数组 [-9, -2, -3, -5, -3], 应该返回 -2

解法一 - 暴力法（超时法）

一般情况下，先从暴力解分析，然后再进行一步步的优化。

思路

我们来试下最直接的方法，就是计算所有的子序列的和，然后取出最大值。 记 Sum[i,....,j]为数组 A 中第 i 个元素到第 j 个元素的和，其中 0 <= i <= j < n， 遍历所有可能的 Sum[i,....,j] 即可。

我们去枚举以 0,1,2...n-1 开头的所有子序列即可， 对于每一个开头的子序列，我们都去枚举从当前开始到 n-1 的所有情况。

这种做法的时间复杂度为 O(N^2), 空间复杂度为 O(1)。

代码

JavaScript:

```
function LSS(list) {
  const len = list.length;
  let max = -Number.MAX_VALUE;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum = 0;
    for (let j = i; j < len; j++) {
      sum += list[j];
      if (sum > max) {
        max = sum;
      }
    }
  }

  return max;
}
```

Java：

```
class MaximumSubarrayPrefixSum {
  public int maxSubArray(int[] nums) {
      int len = nums.length;
      int maxSum = Integer.MIN_VALUE;
      int sum = 0;
      for (int i = 0; i < len; i++) {
        sum = 0;
        for (int j = i; j < len; j++) {
          sum += nums[j];
          maxSum = Math.max(maxSum, sum);
        }
      }
      return maxSum;
  }
}
```

Python 3:

```python
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = -sys.maxsize
        sum = 0
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += nums[j]
                maxSum = max(maxSum, sum)

        return maxSum
```

空间复杂度非常理想，但是时间复杂度有点高。怎么优化呢？我们来看下下一个解法。

解法二 - 分治法

思路

我们来分析一下这个问题， 我们先把数组平均分成左右两部分。

此时有三种情况：

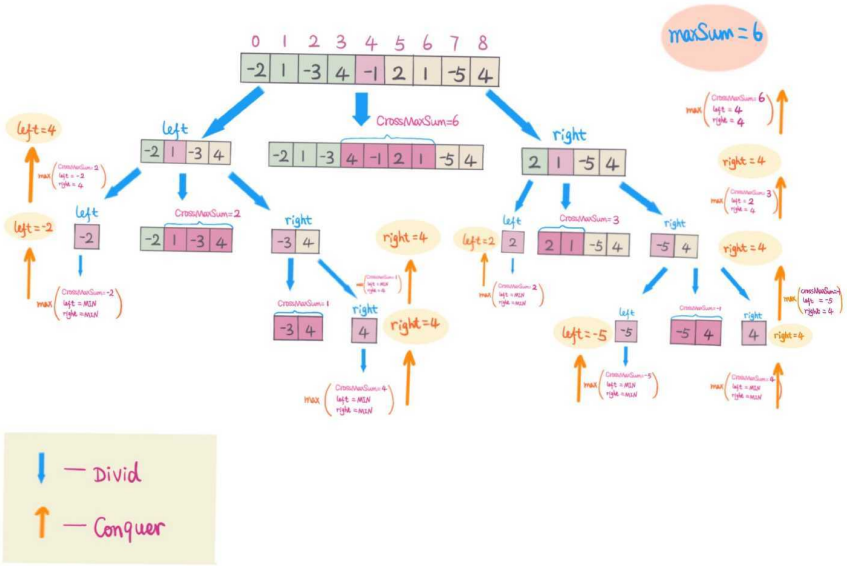
- 最大子序列全部在数组左部分
- 最大子序列全部在数组右部分
- 最大子序列横跨左右数组

对于前两种情况，我们相当于将原问题转化为了规模更小的同样问题。

对于第三种情况，由于已知循环的起点（即中点），我们只需要进行一次循环，分别找出 左边和右边的最大子序列即可。

所以一个思路就是我们每次都对数组分成左右两部分，然后分别计算上面三种情况的最大子序列和， 取出最大的即可。

举例说明，如下图：



(by snowan)

这种做法的时间复杂度为 O(N*logN), 空间复杂度为 O(1)。

代码

JavaScript:

```
function helper(list, m, n) {
  if (m === n) return list[m];
  let sum = 0;
  let lmax = -Number.MAX_VALUE;
  let rmax = -Number.MAX_VALUE;
  const mid = ((n - m) >> 1) + m;
  const l = helper(list, m, mid);
  const r = helper(list, mid + 1, n);
  for (let i = mid; i >= m; i--) {
    sum += list[i];
    if (sum > lmax) lmax = sum;
  }

  sum = 0;

  for (let i = mid + 1; i <= n; i++) {
    sum += list[i];
    if (sum > rmax) rmax = sum;
  }

  return Math.max(l, r, lmax + rmax);
}

function LSS(list) {
  return helper(list, 0, list.length - 1);
}
```

Java:

```
class MaximumSubarrayDivideConquer {
  public int maxSubArrayDividConquer(int[] nums) {
      if (nums == null || nums.length == 0) return 0;
      return helper(nums, 0, nums.length - 1);
    }
    private int helper(int[] nums, int l, int r) {
      if (l > r) return Integer.MIN_VALUE;
      int mid = (l + r) >>> 1;
      int left = helper(nums, l, mid - 1);
      int right = helper(nums, mid + 1, r);
      int leftMaxSum = 0;
      int sum = 0;
      // left surfix maxSum start from index mid - 1 to l
      for (int i = mid - 1; i >= l; i--) {
        sum += nums[i];
        leftMaxSum = Math.max(leftMaxSum, sum);
      }
      int rightMaxSum = 0;
      sum = 0;
      // right prefix maxSum start from index mid + 1 to r
      for (int i = mid + 1; i <= r; i++) {
        sum += nums[i];
        rightMaxSum = Math.max(sum, rightMaxSum);
      }
      // max(left, right, crossSum)
      return Math.max(leftMaxSum + rightMaxSum + nums[mid],
    }
}
```

Python 3 :

```
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        return self.helper(nums, 0, len(nums) - 1)
    def helper(self, nums, l, r):
        if l > r:
            return -sys.maxsize
        mid = (l + r) // 2
        left = self.helper(nums, l, mid - 1)
        right = self.helper(nums, mid + 1, r)
        left_suffix_max_sum = right_prefix_max_sum = 0
        sum = 0
        for i in reversed(range(l, mid)):
            sum += nums[i]
            left_suffix_max_sum = max(left_suffix_max_sum,
        sum = 0
        for i in range(mid + 1, r + 1):
            sum += nums[i]
            right_prefix_max_sum = max(right_prefix_max_sur
        cross_max_sum = left_suffix_max_sum + right_prefix_
        return max(cross_max_sum, left, right)
```

解法三 - 动态规划

思路

我们来思考一下这个问题， 看能不能将其拆解为规模更小的同样问题，并且能找出 递推关系。

我们不妨假设问题 Q(list, i) 表示 list 中以索引 i 结尾的情况下最大子序列和， 那么原问题就转化为 Q(list, i), 其中 i = 0,1,2...n-1 中的最大值。

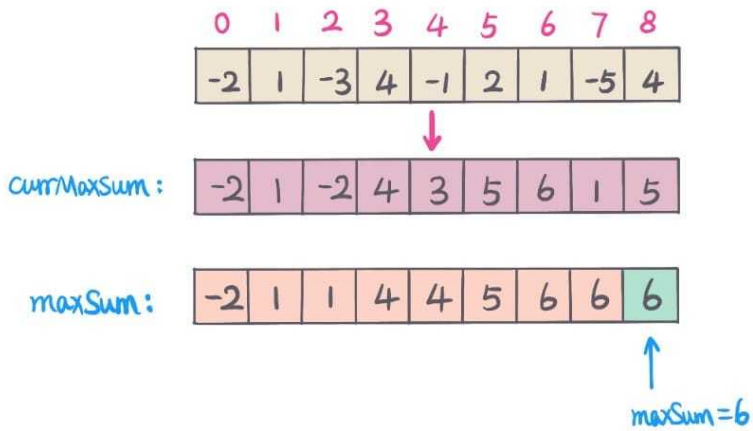
我们继续来看下递归关系，即 Q(list, i)和 Q(list, i - 1)的关系， 即如何根据 Q(list, i - 1) 推导出 Q(list, i)。

如果已知 Q(list, i - 1)， 我们可以将问题分为两种情况，即以索引为 i 的元素终止， 或者只有一个索引为 i 的元素。

- 如果以索引为 i 的元素终止， 那么就是 Q(list, i - 1) + list[i]
- 如果只有一个索引为 i 的元素，那么就是 list[i]

分析到这里，递推关系就很明朗了，即 `Q(list, i) = Math.max(0, Q(list, i - 1)) + list[i]`

举例说明，如下图：



(by snowan)

这种算法的时间复杂度 O(N), 空间复杂度为 O(1)

代码

JavaScript:

```
function LSS(list) {
  const len = list.length;
  let max = list[0];
  for (let i = 1; i < len; i++) {
    list[i] = Math.max(0, list[i - 1]) + list[i];
    if (list[i] > max) max = list[i];
  }

  return max;
}
```

Java:

```
class MaximumSubarrayDP {
  public int maxSubArray(int[] nums) {
     int currMaxSum = nums[0];
     int maxSum = nums[0];
     for (int i = 1; i < nums.length; i++) {
       currMaxSum = Math.max(currMaxSum + nums[i], nums[i]
       maxSum = Math.max(maxSum, currMaxSum);
     }
     return maxSum;
  }
}
```

Python 3:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        max_sum_ending_curr_index = max_sum = nums[0]
        for i in range(1, n):
            max_sum_ending_curr_index = max(max_sum_ending_
            max_sum = max(max_sum_ending_curr_index, max_su

        return max_sum
```

解法四 - 数学分析

思路

我们来通过数学分析来看一下这个题目。

我们定义函数 S(i)，它的功能是计算以 0（包括 0）开始加到 i（包括 i）的值。

那么 S(j) - S(i - 1) 就等于 从 i 开始（包括 i）加到 j（包括 j）的值。

我们进一步分析，实际上我们只需要遍历一次计算出所有的 S(i), 其中 i 等于 0,1,2....,n-1。 然后我们再减去之前的 S(k),其中 k 等于 0，1，i - 1，中的最小值即可。 因此我们需要 用一个变量来维护这个最小值，还需要一个变量维护最大值。

这种算法的时间复杂度 O(N), 空间复杂度为 O(1)。

其实很多题目，都有这样的思想， 比如之前的《每日一题 - 电梯问题》。

代码

JavaScript:

```
function LSS(list) {
  const len = list.length;
  let max = list[0];
  let min = 0;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum += list[i];
    if (sum - min > max) max = sum - min;
    if (sum < min) {
      min = sum;
    }
  }

  return max;
}
```

Java:

```
class MaxSumSubarray {
  public int maxSubArray3(int[] nums) {
      int maxSum = nums[0];
      int sum = 0;
      int minSum = 0;
      for (int num : nums) {
        // prefix Sum
        sum += num;
        // update maxSum
        maxSum = Math.max(maxSum, sum - minSum);
        // update minSum
        minSum = Math.min(minSum, sum);
      }
      return maxSum;
  }
}
```

Python 3:

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = nums[0]
        minSum = sum = 0
        for i in range(n):
            sum += nums[i]
            maxSum = max(maxSum, sum - minSum)
            minSum = min(minSum, sum)

        return maxSum
```

总结

我们使用四种方法解决了 《最大子序列和问题》，并详细分析了各个解法的思路以及复杂度，相信下次你碰到相同或者类似的问题 的时候也能够发散思维，做到 一题多解，多题一解 。

实际上，我们只是求出了最大的和，如果题目进一步要求出最大子序列和的子序列呢？ 如果要题目允许不连续呢？ 我们又该如何思考和变通？如何将数组改成二维，求解最大矩阵和怎么计算？ 这些问题留给读者自己来思考。

简单难度题目合集

这里的题目难度比较小， 大多是模拟题，或者是很容易看出解法的题目，另外简单题目一般使用暴力法都是可以解决的。 这个时候只有看一下数据范围，思考下你的算法复杂度就行了。

当然也不排除很多 hard 题目也可以暴力模拟，大家平时多注意数据范围即可。

以下是我列举的经典题目（带 91 字样的表示出自 **91 天学算法**活动）：

- 面试题 17.12. BiNode
- 0001. 两数之和
- 0020. 有效的括号
- 0021. 合并两个有序链表
- 0026. 删除排序数组中的重复项
- 0053. 最大子序和
- 0066. 加一 91
- 0088. 合并两个有序数组
- 0101. 对称二叉树
- 0104. 二叉树的最大深度
- 0108. 将有序数组转换为二叉搜索树
- 0121. 买卖股票的最佳时机
- 0122. 买卖股票的最佳时机 II
- 0125. 验证回文串
- 0136. 只出现一次的数字
- 0155. 最小栈
- 0160. 相交链表 91
- 0167. 两数之和 II 输入有序数组
- 0169. 多数元素
- 0172. 阶乘后的零
- 0190. 颠倒二进制位
- 0191. 位 1 的个数
- 0198. 打家劫舍
- 0203. 移除链表元素
- 0206. 反转链表
- 0219. 存在重复元素 II
- 0226. 翻转二叉树
- 0232. 用栈实现队列 91
- 0263. 丑数
- 0283. 移动零
- 0342. 4 的幂
- 0349. 两个数组的交集
- 0371. 两整数之和

- 401. 二进制手表
- 0437. 路径总和 III
- 0455. 分发饼干
- 0575. 分糖果
- 821. 字符的最短距离 91
- 0874. 模拟行走机器人
- 1260. 二维网格迁移
- 1332. 删除回文子序列

题目地址（面试题 17.12. BiNode）

https://leetcode-cn.com/problems/binode-lcci/

题目描述

```
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

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。
```

前置知识

- 二叉查找树
- 递归
- 二叉树的遍历

公司

- 暂无

思路

实际上这就是一个考察二叉树遍历 + 二叉搜索(查找)树性质的题目。这类题目特别需要注意的是指针操作，这一点和**链表反转系列**题目是一样的。

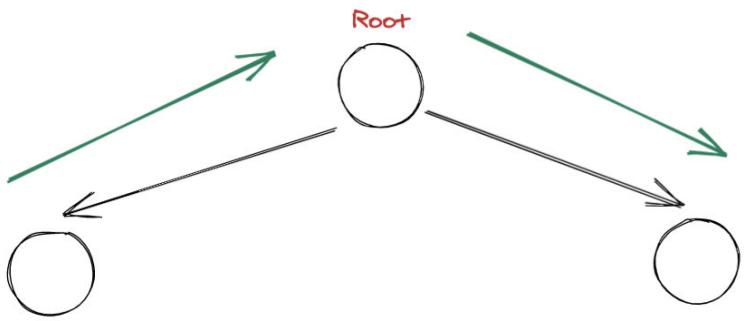
首先我们要知道一个性质： **对于一个二叉查找树来说，其中序遍历结果是一个有序数组**。 而题目要求你输出的恰好就是有序数组（虽然没有明说， 不过从测试用例也可以看出）。

因此一个思路就是中序遍历， 边遍历边改变指针即可。 这里有两个注意点:

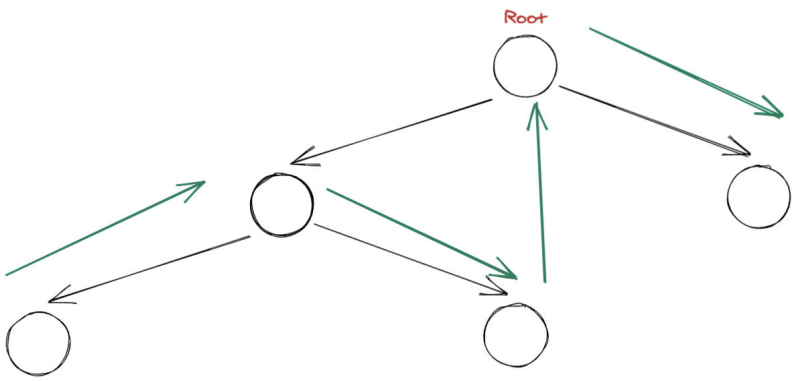
1. 指针操作小心互相引用，导致死循环。
2. 你需要返回的是最左下角的节点，而不是题目给的 root。
3. 对于第一个问题， 其实只要注意操作指针的顺序，以及在必要的时候重置指针即可。
4. 对于第二个问题，我用了一个黑科技，让代码看起来简洁又高效。如果不懂的话， 你也可以换个朴素的写法。

理解了上面的内容的话， 那让我们进入正题。

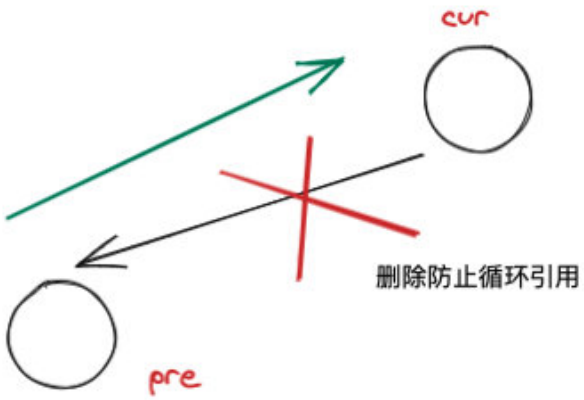
其中绿色是我们要增加的连线，而黑色是是原本的连线。



我们再来看一个复杂一点的:



实际上，不管多么复杂。 我们只需要进行一次**中序遍历**，同时记录前驱节点。然后修改前驱节点和当前节点的指针即可，整个过程就好像是链表反转。



核心代码（假设 pre 我们已经正确计算出了）：

```
cur.left = None
pre.right = cur
pre = cur
```

剩下的就是如何计算 pre，这个也不难，直接看代码：

```
self.pre = None
def dfs(root):
    dfs(root.left)
    # 上面的指针改变逻辑写到这里
    self.pre = root
    dfs(root.right)
```

问题得以解决。

这里还有最后一个问题就是返回值，题目要返回的实际上是最左下角的值。如何取到最左下角的节点呢？我们来看下核心代码你就懂了，代码比较简单。

```
    self.pre = self.ans = None
    def dfs(root):
        if not root: return
        dfs(root.left)
        root.left = None
        if self.pre: self.pre.right = root
        # 当第一次执行到下面这一行代码，恰好是在最左下角，此时 self
        if self.pre is None: self.ans = root
        self.pre = root
        dfs(root.right)
```

关键点

- 指针操作
- 返回值的处理

代码

```
class Solution:
    def convertBiNode(self, root: TreeNode) -> TreeNode:
        self.pre = self.ans = None
        def dfs(root):
            if not root: return
            dfs(root.left)
            root.left = None
            if self.pre: self.pre.right = root
            if self.pre is None: self.ans = root
            self.pre = root

            dfs(root.right)
        dfs(root)
        return self.ans
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点总数。
- 空间复杂度：$O(h)$，其中 h 为树的高度。

相关题目

- 206.reverse-linked-list
- 92.reverse-linked-list-ii
- 25.reverse-nodes-in-k-groups-cn

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(1. 两数之和)

https://leetcode-cn.com/problems/two-sum

题目描述

```
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

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的

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]
```

前置知识

- 哈希表

公司

- 字节
- 百度
- 腾讯
- adobe
- airbnb
- amazon
- apple
- bloomberg
- dropbox
- facebook
- linkedin
- microsoft
- uber
- yahoo
- yelp

思路

最容易想到的就是暴力枚举，我们可以利用两层 for 循环来遍历每个元素，并查找满足条件的目标元素。

伪代码：

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < i;j ++){
    if (nums[i] + nums[j] == target) return [j, i]
  }
}
```

不过这样时间复杂度为 O(N^2)，空间复杂度为 O(1)，时间复杂度较高，我们要想办法进行优化。

这里我们可以增加一个 Map 记录已经遍历过的数字及其对应的索引值。这样当遍历一个新数字的时候就去 Map 里查询 **target 与该数的差值 diff 是否已经在前面的数字中出现过**。如果出现过，说明 diff + 当前数 = target，我们就找到了一组答案。

关键点

- 求和转换为求差
- 借助 Map 结构将数组中每个元素及其索引相互对应
- 以空间换时间，将查找时间从 O(N) 降低到 O(1)

代码

- 语言支持：JS, Go，CPP

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
const twoSum = function (nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    const diff = target - nums[i];
    if (map.has(diff)) {
      return [map.get(diff), i];
    }
    map.set(nums[i], i);
  }
};
```

Go Code:

```go
func twoSum(nums []int, target int) []int {
    m := make(map[int]int)
    for i, _ := range nums {
        diff := target - nums[i]
        if j, ok := m[diff]; ok {
            return []int{i, j}
        } else {
            m[nums[i]] = i
        }
    }
    return []int{}
}
```

CPP Code:

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& A, int target) {
        unordered_map<int, int> m;
        for (int i = 0; i < A.size(); ++i) {
            int t = target - A[i];
            if (m.count(t)) return { m[t], i };
            m[A[i]] = i;
        }
        return {};
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(20. 有效的括号)

https://leetcode-cn.com/problems/valid-parentheses/description

题目描述

```
给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串，判断字符串是

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
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
```

前置知识

- 栈

公司

- 阿里
- 百度

- 腾讯
- 字节
- airbnb
- amazon
- bloomberg
- facebook
- google
- microsoft
- twitter
- zenefits

栈

思路

关于这道题的思路，邓俊辉讲的非常好，没有看过的同学可以看一下，视频地址。

使用栈，遍历输入字符串

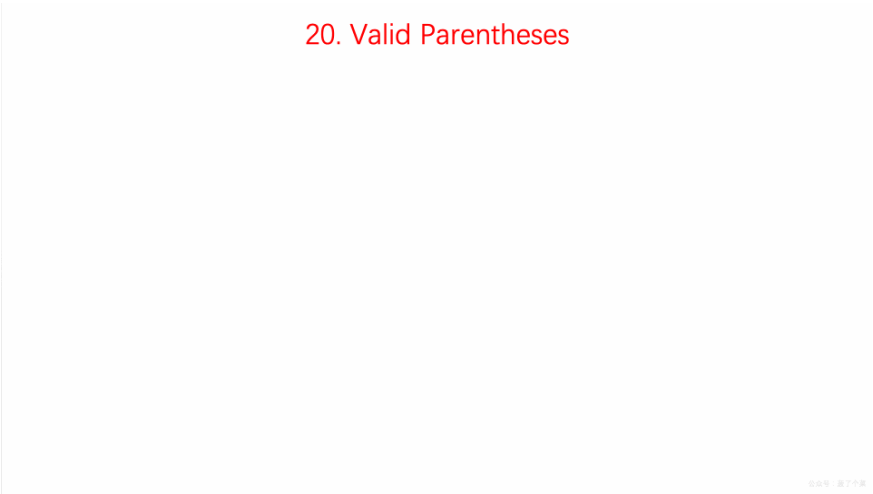
如果当前字符为左半边括号时，则将其压入栈中

如果遇到右半边括号时，分类讨论：

1）如栈不为空且为对应的左半边括号，则取出栈顶元素，继续循环

2）若此时栈为空，则直接返回 false

3）若不为对应的左半边括号，反之返回 false



（图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

值得注意的是，如果题目要求只有一种括号，那么我们其实可以使用更简洁，更省内存的方式 - 计数器来进行求解，而不必要使用栈。 而之所以多种括号不可以使用计数器在 $O(1)$ 空间做到是因为类似这种用例会无法

处理 "([)]"。

关键点解析

1. 栈的基本特点和操作
2. 可以用数组来模拟栈

比如 入： push 出：pop 就是栈。 入： push 出 shift 就是队列。 但是这种算法实现的队列在头部删除元素的时候时间复杂度比较高，具体大家可以参考一下双端队列 deque。

代码

代码支持：JS，Python

Javascript Code:

```
/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function (s) {
  let valid = true;
  const stack = [];
  const mapper = {
    "{": "}",
    "[": "]",
    "(": ")",
  };

  for (let i in s) {
    const v = s[i];
    if (["(", "[", "{"].indexOf(v) > -1) {
      stack.push(v);
    } else {
      const peak = stack.pop();
      if (v !== mapper[peak]) {
        return false;
      }
    }
  }

  if (stack.length > 0) return false;

  return valid;
};
```

Python Code:

```
    class Solution:
        def isValid(self,s):
          stack = []
          map = {
            "{":"}",
            "[":"]",
            "(":")"
          }
          for x in s:
            if x in map:
              stack.append(map[x])
            else:
              if len(stack)!=0:
                top_element = stack.pop()
                if x != top_element:
                  return False
                else:
                  continue
              else:
                return False
          return len(stack) == 0
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

O(1) 空间

思路

基本思路是修改参数，将参数作为我们的栈。 随着我们不断遍历， s 慢慢变成了一个栈。

因此 Python，Java，JS 等**字符串不可变**的语言无法使用此方法达到 $O(1)$。

具体参考： No stack O(1) space complexity O(n) time complexity solution in C++-space-complexity-O(n)-time-complexity-solution-in-C++/244061>)

代码

代码支持：C++

C++:

```
class Solution {
public:
    bool isValid(string s) {
        int top = -1;
        for(int i =0;i<s.length();++i){
            if(top<0 || !isMatch(s[top], s[i])){
                ++top;
                s[top] = s[i];
            }else{
                --top;
            }
        }
        return top == -1;
    }
    bool isMatch(char c1, char c2){
        if(c1 == '(' && c2 == ')') return true;
        if(c1 == '[' && c2 == ']') return true;
        if(c1 == '{' && c2 == '}') return true;
        return false;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

正则匹配

思路

我们不断通过消除 '[]' ， '()', '{}' ， 最后判断剩下的是否是空串即可，就像开心消消乐一样。

代码

代码支持：Python，JavaScript

Python:

```
class Solution:
     def isValid(self, s):

         while '[]' in s or '()' in s or '{}' in s:
             s = s.replace('[]','').replace('()','').replace
         return not len(s)
```

JavaScript:

```
var isValid = function (s) {
  while (s.includes("[]") || s.includes("()") || s.includes
    s = s.replace("[]", "").replace("()", "").replace("{}",
  }
  s = s.replace("[]", "").replace("()", "").replace("{}", "
  return s.length === 0;
};
```

复杂度分析

- 时间复杂度：取决于正则引擎的实现
- 空间复杂度：取决于正则引擎的实现

相关题目

- 32. 最长有效括号

扩展

- 如果让你检查 XML 标签是否闭合如何检查， 更进一步如果要你实现一个简单的 XML 的解析器，应该怎么实现?
- 事实上，这类问题还可以进一步扩展，我们可以去解析类似 HTML 等标记语法， 比如 `<p></p> <body></body>`

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(21. 合并两个有序链表)

https://leetcode-cn.com/problems/merge-two-sorted-lists

题目描述

> 将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个
>
> 示例：
>
> 输入：1->2->4, 1->3->4
> 输出：1->1->2->3->4->4

前置知识

- 递归
- 链表

公司

- 阿里
- 字节
- 腾讯
- amazon
- apple
- linkedin
- microsoft

公司

- 阿里、字节、腾讯

思路

本题可以使用递归来解，将两个链表头部较小的一个与剩下的元素合并，并返回排好序的链表头，当两条链表中的一条为空时终止递归。

关键点

- 掌握链表数据结构

- 考虑边界情况

代码

- 语言支持：CPP, JS

CPP Code:

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (l1 == nullptr) {
            return l2;
        } else if (l2 == nullptr) {
            return l1;
        } else if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

JS Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
const mergeTwoLists = function (l1, l2) {
  if (l1 === null) {
    return l2;
  }
  if (l2 === null) {
    return l1;
  }
  if (l1.val < l2.val) {
    l1.next = mergeTwoLists(l1.next, l2);
    return l1;
  } else {
    l2.next = mergeTwoLists(l1, l2.next);
    return l2;
  }
};
```

复杂度分析

M、N 是两条链表 l1、l2 的长度

- 时间复杂度：$O(M+N)$
- 空间复杂度：$O(M+N)$

扩展

- 你可以使用迭代的方式求解么?

迭代的 CPP 代码如下：

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
        ListNode head, *tail = &head;
        while (a && b) {
            if (a->val <= b->val) {
                tail->next = a;
                a = a->next;
            } else {
                tail->next = b;
                b = b->next;
            }
            tail = tail->next;
        }
        tail->next = a ? a : b;
        return head.next;
    }
};
```

迭代的 JS 代码如下：

```
var mergeTwoLists = function (l1, l2) {
  const prehead = new ListNode(-1);

  let prev = prehead;
  while (l1 != null && l2 != null) {
    if (l1.val <= l2.val) {
      prev.next = l1;
      l1 = l1.next;
    } else {
      prev.next = l2;
      l2 = l2.next;
    }
    prev = prev.next;
  }
  prev.next = l1 === null ? l2 : l1;

  return prehead.next;
};
```

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(26. 删除排序数组中的重复项)

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/description/

题目描述

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]);
}
```

前置知识

- 数组
- 双指针

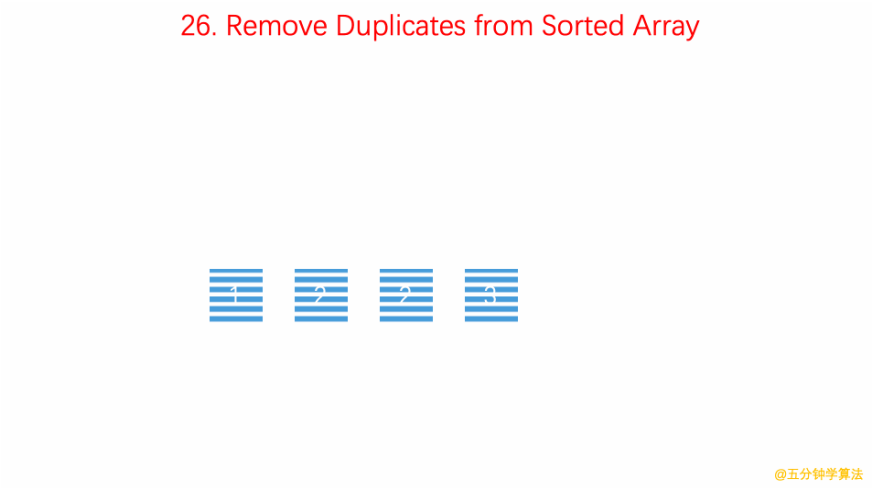
公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- facebook
- microsoft

思路

使用快慢指针来记录遍历的坐标。

- 开始时这两个指针都指向第一个数字
- 如果两个指针指的数字相同，则快指针向前走一步
- 如果不同，则两个指针都向前走一步
- 当快指针走完整个数组后，慢指针当前的坐标加 1 就是数组中不同数字的个数



（图片来自： https://github.com/MisterBooo/LeetCodeAnimation）

实际上这就是双指针中的快慢指针。在这里快指针是读指针， 慢指针是写指针。**从读写指针考虑， 我觉得更符合本质**。

关键点解析

- 双指针

这道题如果不要求，O(n) 的时间复杂度， O(1) 的空间复杂度的话，会很简单。 但是这道题是要求的，这种题的思路一般都是采用双指针

- 如果是数据是无序的，就不可以用这种方式了，从这里也可以看出排序在算法中的基础性和重要性。
- 注意 nums 为空时的边界条件。

代码

- 语言支持：JS，Python，C++

Javascript Code:

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function (nums) {
  const size = nums.length;
  if (size == 0) return 0;
  let slowP = 0;
  for (let fastP = 0; fastP < size; fastP++) {
    if (nums[fastP] !== nums[slowP]) {
      slowP++;
      nums[slowP] = nums[fastP];
    }
  }
  return slowP + 1;
};
```

Python Code:

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if nums:
            slow = 0
            for fast in range(1, len(nums)):
                if nums[fast] != nums[slow]:
                    slow += 1
                    nums[slow] = nums[fast]
            return slow + 1
        else:
            return 0
```

C++ Code:

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        if(nums.empty()) return 0;
        int fast,slow;
        fast=slow=0;
        while(fast!=nums.size()){
            if(nums[fast]==nums[slow]) fast++;
            else {
                slow++;
                nums[slow]=nums[fast];
                fast++;
            }
        }
        return slow+1;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(53. 最大子序和)

https://leetcode-cn.com/problems/maximum-subarray/

题目描述

```
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

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。
```

前置知识

- 滑动窗口
- 动态规划

公司

- 阿里
- 百度
- 字节
- 腾讯
- bloomberg
- linkedin
- microsoft

思路

这道题求解连续最大子序列和，以下从时间复杂度角度分析不同的解题思路。

解法一 - 暴力解 （暴力出奇迹， 噢耶！）

一般情况下，先从暴力解分析，然后再进行一步步的优化。

原始暴力解：（超时）

求子序列和，那么我们要知道子序列的首尾位置，然后计算首尾之间的序列和。用 2 个 for 循环可以枚举所有子序列的首尾位置。 然后用一个 for 循环求解序列和。这里时间复杂度太高， `O(n^3)` .

复杂度分析

- 时间复杂度：$O(N ^ 3)$, 其中 N 是数组长度
- 空间复杂度：$O(1)$

解法二 - 前缀和 + 暴力解

优化暴力解：（震惊，居然 AC 了）

在暴力解的基础上，用前缀和我们可以优化到暴力解 `O(n^2)` , 这里以空间换时间。 这里可以使用原数组表示 `prefixSum` , 省空间。

求序列和可以用前缀和（ `prefixSum`) 来优化，给定子序列的首尾位置 `(l, r),` 那么序列和 `subarraySum=prefixSum[r] - prefixSum[l - 1];` 用一个全局变量 `maxSum` , 比较每次求解的子序列和， `maxSum = max(maxSum, subarraySum)` .

复杂度分析

- 时间复杂度：$O(N ^ 2)$, 其中 N 是数组长度
- 空间复杂度：$O(N)$

> 如果用更改原数组表示前缀和数组，空间复杂度降为 `O(1)`

但是时间复杂度还是太高，还能不能更优化。答案是可以，前缀和还可以优化到 `O(n)` .

解法三 - 优化前缀和 - from @lucifer

我们定义函数 `S(i)` ，它的功能是计算以 `0（包括 0）` 开始加到 `i（包括 i）` 的值。

那么 `S(j) - S(i - 1)` 就等于 从 `i` 开始（包括 i）加到 `j` （包括 j）的值。

我们进一步分析，实际上我们只需要遍历一次计算出所有的 `S(i)` , 其中 `i = 0,1,2,....,n-1。` 然后我们再减去之前的 `S(k)` ,其中 `k = 0,1,2,...,i-1` ，中的最小值即可。 因此我们需要 用一个变量来维护这个最小值，还需要一个变量维护最大值。

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：$O(1)$

解法四 - 分治法

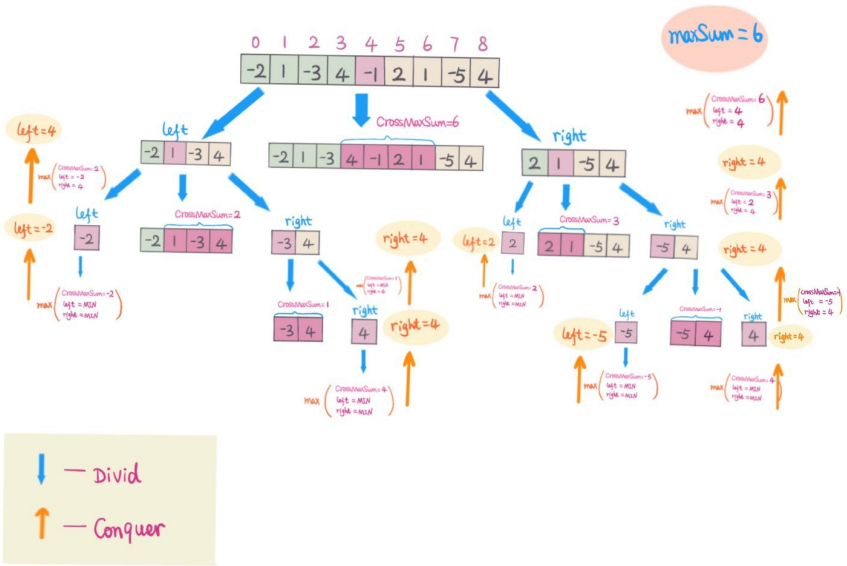
我们把数组 `nums` 以中间位置（ `m`)分为左（ `left`)右(`right`)两部分. 那么有， `left = nums[0]...nums[m - 1]` 和 `right = nums[m + 1]...nums[n-1]`

最大子序列和的位置有以下三种情况：

1. 考虑中间元素 `nums[m]` , 跨越左右两部分，这里从中间元素开始，往左求出后缀最大，往右求出前缀最大, 保持连续性。
2. 不考虑中间元素，最大子序列和出现在左半部分，递归求解左边部分最大子序列和
3. 不考虑中间元素，最大子序列和出现在右半部分，递归求解右边部分最大子序列和

分别求出三种情况下最大子序列和，三者中最大值即为最大子序列和。

举例说明，如下图：



复杂度分析

- 时间复杂度：$O(NlogN)$, 其中 N 是数组长度
- 空间复杂度：$O(logN)$

解法五 - 动态规划

动态规划的难点在于找到状态转移方程，

`dp[i] - 表示到当前位置 i 的最大子序列和`

状态转移方程为： `dp[i] = max(dp[i - 1] + nums[i], nums[i])`

初始化： `dp[0] = nums[0]`

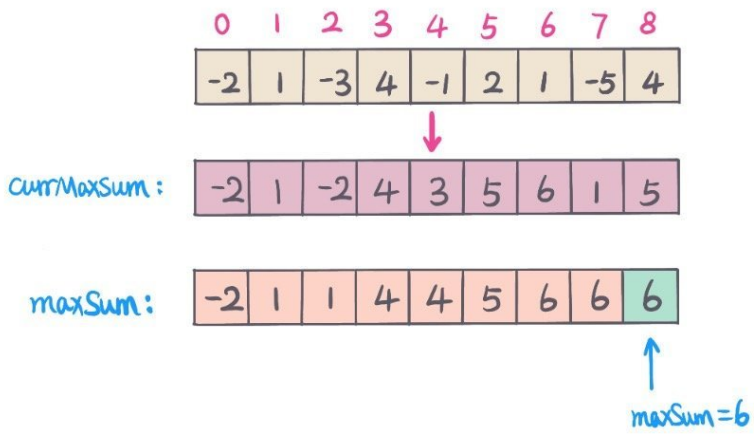
从状态转移方程中，我们只关注前一个状态的值，所以不需要开一个数组记录位置所有子序列和，只需要两个变量，

`currMaxSum - 累计最大和到当前位置i`

`maxSum - 全局最大子序列和 `：

- `currMaxSum = max(currMaxSum + nums[i], nums[i])`
- `maxSum = max(currMaxSum, maxSum)`

如图：



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：$O(1)$

关键点分析

1. 暴力解，列举所有组合子序列首尾位置的组合，求解最大的子序列和, 优化可以预先处理，得到前缀和
2. 分治法，每次从中间位置把数组分为左右中三部分， 分别求出左右中（这里中是包括中间元素的子序列）最大和。对左右分别深度递归，三者中最大值即为当前最大子序列和。
3. 动态规划，找到状态转移方程，求到当前位置最大和。

代码 (`Java/Python3/Javascript`)

解法二 - 前缀和 + 暴力

Java code

```java
class MaximumSubarrayPrefixSum {
  public int maxSubArray(int[] nums) {
      int len = nums.length;
      int maxSum = Integer.MIN_VALUE;
      int sum = 0;
      for (int i = 0; i < len; i++) {
        sum = 0;
        for (int j = i; j < len; j++) {
          sum += nums[j];
          maxSum = Math.max(maxSum, sum);
        }
      }
      return maxSum;
  }
}
```

Python3 code `(TLE)`

```python
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = -sys.maxsize
        sum = 0
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += nums[j]
                maxSum = max(maxSum, sum)

        return maxSum
```

Javascript code from **@lucifer**

```
function LSS(list) {
  const len = list.length;
  let max = -Number.MAX_VALUE;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum = 0;
    for (let j = i; j < len; j++) {
      sum += list[j];
      if (sum > max) {
        max = sum;
      }
    }
  }

  return max;
}
```

解法三 - 优化前缀和

Java code

```
class MaxSumSubarray {
  public int maxSubArray3(int[] nums) {
      int maxSum = nums[0];
      int sum = 0;
      int minSum = 0;
      for (int num : nums) {
        // prefix Sum
        sum += num;
        // update maxSum
        maxSum = Math.max(maxSum, sum - minSum);
        // update minSum
        minSum = Math.min(minSum, sum);
      }
      return maxSum;
  }
}
```

Python3 code

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        maxSum = nums[0]
        minSum = sum = 0
        for i in range(n):
            sum += nums[i]
            maxSum = max(maxSum, sum - minSum)
            minSum = min(minSum, sum)

        return maxSum
```

Javascript code from **@lucifer**

```javascript
function LSS(list) {
  const len = list.length;
  let max = list[0];
  let min = 0;
  let sum = 0;
  for (let i = 0; i < len; i++) {
    sum += list[i];
    if (sum - min > max) max = sum - min;
    if (sum < min) {
      min = sum;
    }
  }

  return max;
}
```

解法四 - 分治法

Java code

```
class MaximumSubarrayDivideConquer {
  public int maxSubArrayDividConquer(int[] nums) {
      if (nums == null || nums.length == 0) return 0;
      return helper(nums, 0, nums.length - 1);
    }
    private int helper(int[] nums, int l, int r) {
      if (l > r) return Integer.MIN_VALUE;
      int mid = (l + r) >>> 1;
      int left = helper(nums, l, mid - 1);
      int right = helper(nums, mid + 1, r);
      int leftMaxSum = 0;
      int sum = 0;
      // left surfix maxSum start from index mid - 1 to l
      for (int i = mid - 1; i >= l; i--) {
        sum += nums[i];
        leftMaxSum = Math.max(leftMaxSum, sum);
      }
      int rightMaxSum = 0;
      sum = 0;
      // right prefix maxSum start from index mid + 1 to r
      for (int i = mid + 1; i <= r; i++) {
        sum += nums[i];
        rightMaxSum = Math.max(sum, rightMaxSum);
      }
      // max(left, right, crossSum)
      return Math.max(leftMaxSum + rightMaxSum + nums[mid],
    }
}
```

Python3 code

```python
import sys
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        return self.helper(nums, 0, len(nums) - 1)
    def helper(self, nums, l, r):
        if l > r:
            return -sys.maxsize
        mid = (l + r) // 2
        left = self.helper(nums, l, mid - 1)
        right = self.helper(nums, mid + 1, r)
        left_suffix_max_sum = right_prefix_max_sum = 0
        sum = 0
        for i in reversed(range(l, mid)):
            sum += nums[i]
            left_suffix_max_sum = max(left_suffix_max_sum,
        sum = 0
        for i in range(mid + 1, r + 1):
            sum += nums[i]
            right_prefix_max_sum = max(right_prefix_max_sur
        cross_max_sum = left_suffix_max_sum + right_prefix_
        return max(cross_max_sum, left, right)
```

Javascript code from **@lucifer**

```
function helper(list, m, n) {
  if (m === n) return list[m];
  let sum = 0;
  let lmax = -Number.MAX_VALUE;
  let rmax = -Number.MAX_VALUE;
  const mid = ((n - m) >> 1) + m;
  const l = helper(list, m, mid);
  const r = helper(list, mid + 1, n);
  for (let i = mid; i >= m; i--) {
    sum += list[i];
    if (sum > lmax) lmax = sum;
  }

  sum = 0;

  for (let i = mid + 1; i <= n; i++) {
    sum += list[i];
    if (sum > rmax) rmax = sum;
  }

  return Math.max(l, r, lmax + rmax);
}

function LSS(list) {
  return helper(list, 0, list.length - 1);
}
```

解法五 - 动态规划

Java code

```
class MaximumSubarrayDP {
  public int maxSubArray(int[] nums) {
     int currMaxSum = nums[0];
     int maxSum = nums[0];
     for (int i = 1; i < nums.length; i++) {
       currMaxSum = Math.max(currMaxSum + nums[i], nums[i]
       maxSum = Math.max(maxSum, currMaxSum);
     }
     return maxSum;
  }
}
```

Python3 code

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        max_sum_ending_curr_index = max_sum = nums[0]
        for i in range(1, n):
            max_sum_ending_curr_index = max(max_sum_ending_
            max_sum = max(max_sum_ending_curr_index, max_su

        return max_sum
```

Javascript code from **@lucifer**

```javascript
function LSS(list) {
  const len = list.length;
  let max = list[0];
  for (let i = 1; i < len; i++) {
    list[i] = Math.max(0, list[i - 1]) + list[i];
    if (list[i] > max) max = list[i];
  }

  return max;
}
```

扩展

- 如果数组是二维数组，求最大子数组的和?
- 如果要求最大子序列的乘积?

相似题

- Maximum Product Subarray
- Longest Turbulent Subarray

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(160. 相交链表)

https://leetcode-cn.com/problems/intersection-of-two-linked-lists/

题目描述

> 编写一个程序，找到两个单链表相交的起始节点。

前置知识

- 链表
- 双指针

解法一: 哈希法

有 A, B 这两条链表, 先遍历其中一个，比如 A 链表, 并将 A 中的所有节点存入哈希表。

遍历 B 链表,检查节点是否在哈希表中, 第一个存在的就是相交节点

- 伪代码

```
data = new Set() // 存放A链表的所有节点的地址

while A不为空{
  哈希表中添加A链表当前节点
  A指针向后移动
}

while B不为空{
  if 如果哈希表中含有B链表当前节点
    return B
  B指针向后移动
}

return null // 两条链表没有相交点
```

- 代码支持: JS

JS Code:

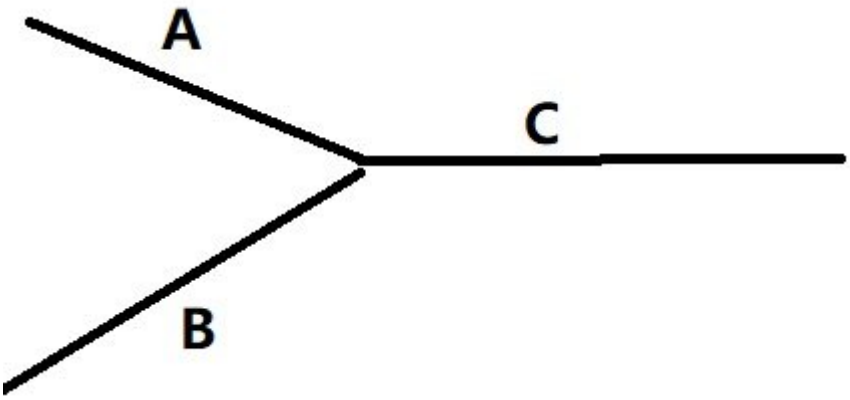
```
let data = new Set();
while (A !== null) {
  data.add(A);
  A = A.next;
}
while (B !== null) {
  if (data.has(B)) return B;
  B = B.next;
}
return null;
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

解法二：双指针

- 例如使用 a, b 两个指针分别指向 A, B 这两条链表, 两个指针相同的速度向后移动,
- 当 a 到达链表的尾部时,重定位到链表 B 的头结点
- 当 b 到达链表的尾部时,重定位到链表 A 的头结点。
- a, b 指针相遇的点为相交的起始节点，否则没有相交点



（图 5）

为什么 a, b 指针相遇的点一定是相交的起始节点? 我们证明一下：

1. 将两条链表按相交的起始节点继续截断，链表 1 为: A + C，链表 2 为: B + C
2. 当 a 指针将链表 1 遍历完后,重定位到链表 B 的头结点,然后继续遍历直至相交点(a 指针遍历的距离为 A + C + B)
3. 同理 b 指针遍历的距离为 B + C + A
4. 伪代码

```
a = headA
b = headB
while a,b指针不相等时 {
    if a指针为空时
      a指针重定位到链表 B的头结点
    else
      a指针向后移动一位
    if b指针为空时
      b指针重定位到链表 A的头结点
    else
      b指针向后移动一位
}
return a
```

- 代码支持: JS, Python, Go, PHP

JS Code:

```
var getIntersectionNode = function (headA, headB) {
  let a = headA,
    b = headB;
  while (a != b) {
    a = a === null ? headB : a.next;
    b = b === null ? headA : b.next;
  }
  return a;
};
```

Python Code：

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: L
        a, b = headA, headB
        while a != b:
            a = a.next if a else headB
            b = b.next if b else headA
        return a
```

Go Code:

```
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func getIntersectionNode(headA, headB *ListNode) *ListNode
    // a=A(a单独部分)+C(a相交部分); b=B(b单独部分)+C(b相交部分)
    // a+b=b+a=A+C+B+C=B+C+A+C
    a := headA
    b := headB
    for a != b {
        if a == nil {
            a = headB
        } else {
            a = a.Next
        }
        if b == nil {
            b = headA
        } else {
            b = b.Next
        }
    }
    return a
}
```

PHP Code:

```php
/**
 * Definition for a singly-linked list.
 * class ListNode {
 *     public $val = 0;
 *     public $next = null;
 *     function __construct($val) { $this->val = $val; }
 * }
 */
class Solution
{
    /**
     * @param ListNode $headA
     * @param ListNode $headB
     * @return ListNode
     */
    function getIntersectionNode($headA, $headB)
    {
        $a = $headA;
        $b = $headB;
        while ($a !== $b) { // 注意, 这里要用 !==
            $a = $a ? $a->next : $headB;
            $b = $b ? $b->next : $headA;
        }
        return $a;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

题目地址(66. 加一)

https://leetcode-cn.com/problems/plus-one

题目描述

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1:

```
输入: [1,2,3]
输出: [1,2,4]
解释: 输入数组表示数字 123。
示例 2:

输入: [4,3,2,1]
输出: [4,3,2,2]
解释: 输入数组表示数字 4321。
```

> lucifer 提示： 不要加直接数组转化为数字做加法再转回来。

前置知识

- 数组的遍历(正向遍历和反向遍历)

思路

这道题其实我们可以把它想象成小学生练习加法，只不过现在是固定的“加一”那么我们只需要考虑如何通过遍历来实现这个加法的过程就好了。

加法我们知道要从低位到高位进行运算，那么只需要对数组进行一次反向遍历即可。

伪代码：

```
for(int i = n - 1; i > - 1; i --) {
  内部逻辑
}
```

内部逻辑的话，其实有三种情况：

```
1. 个位上的数字小于9
    17
+   1
= 18
2. 个位数上等于9，其他位数可以是0-9的任何数，但是首位不等于9
    199
+     1
=  200

    109
+      1
=  110
3. 所有位数都为9
    99
+    1
= 100

      999
+       1
=  1000
```

第一种情况是最简单的，我们只需将数组的最后一位进行+1 操作就好了

第二种情况稍微多了一个步骤：我们需要把个位的 carry 向前进一位并在计算是否有更多的进位

第三种其实和第二种是一样的操作，只是由于我们知道数组的长度是固定的，所以当我们遇到情况三的时候需要扩大数组的长度。我们只需要在结果数组前多加上一位就好了。

```
// 首先我们要从数组的最后一位开始我们的计算得出我们新的sum
sum = arr[arr.length - 1] + 1

// 接下来我们需要判断这个新的sum是否超过9
sum > 9 ?

// 假如大于 9, 那么我们会更新这一位为 0 并且将carry值更改为1
carry = 1
arr[i] = 0

// 假如不大于 9，更新最后一位为sum并直接返回数组
arr[arr.length - 1] = sum
return arr

// 接着我们要继续向数组的倒数第二位重复进行我们上一步的操作
...

// 当我们完成以后，如果数组第一位时的sum大于0，那么我们就要给数组的首
result = new array with size of arr.length + 1
result[0] = 1
result[1] ..... result[result.length - 1]  = 0
```

代码

代码支持：Python3，JS, CPP, Go, PHP

Python3 Code：

```
class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        carry = 1
        for i in range(len(digits) - 1, -1, -1):
            digits[i], carry = (carry + digits[i]) % 10, (
        return [carry] + digits if carry else digits
```

JS Code：

```
var plusOne = function (digits) {
  var carry = 1; // 我们将初始的 +1 也当做是一个在个位的 carry
  for (var i = digits.length - 1; i > -1; i--) {
    if (carry) {
      var sum = carry + digits[i];
      digits[i] = sum % 10;
      carry = sum > 9 ? 1 : 0; // 每次计算都会更新下一步需要用到
    }
  }
  if (carry === 1) {
    digits.unshift(1); // 如果carry最后停留在1，说明有需要额外的
  }
  return digits;
};
```

CPP Code:

```
class Solution {
public:
    vector<int> plusOne(vector<int>& A) {
        int i = A.size() - 1, carry = 1;
        for (; i >= 0 && carry; --i) {
            carry += A[i];
            A[i] = carry % 10;
            carry /= 10;
        }
        if (carry) A.insert(begin(A), carry);
        return A;
    }
};
```

Go code:

```
func plusOne(digits []int) []int {
    for i := len(digits) - 1; i >= 0; i-- {
        digits[i]++
        if digits[i] != 10 { // 不产生进位，直接返回
            return digits
        }
        digits[i] = 0 // 产生进位，继续计算下一位
    }
    // 全部产生进位
    digits[0] = 1
    digits = append(digits, 0)
    return digits
}
```

PHP code:

```php
class Solution {

    /**
     * @param Integer[] $digits
     * @return Integer[]
     */
    function plusOne($digits) {
        $len = count($digits);
        for ($i = $len - 1; $i >= 0; $i--) {
            $digits[$i]++;
            if ($digits[$i] != 10) { // 不产生进位，直接返回
                return $digits;
            }
            $digits[$i] = 0; // 产生进位，继续计算下一位
        }
        // 全部产生进位
        $digits[0] = 1;
        $digits[$len] = 0;
        return $digits;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 面试题 02.05. 链表求和

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（88. 合并两个有序数组）

https://leetcode-cn.com/problems/merge-sorted-array/

题目描述

```
给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使

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 num
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]
```

公司

- 阿里
- 腾讯
- 百度
- 字节
- loomberg
- facebook
- microsoft

前置知识

- 归并排序

思路

符合直觉的做法是 `将nums2插到num1的末尾，然后排序`

具体代码：

```
// 这种解法连m都用不到
// 这显然不是出题人的意思
if (n === 0) return;
let current2 = 0;
for (let i = nums1.length - 1; i >= nums1.length - n; i--)
  nums1[i] = nums2[current2++];
}
nums1.sort((a, b) => a - b); // 当然你可以自己写排序，这里懒得写
```

这道题目其实和基本排序算法中的 `merge sort` 非常像。

我们先来回顾一下 merge sort 的 merge 过程。merge 的过程 `可以` 是先比较两个数组的**头元素**，然后将较小的推到最终的数组中，并将其从原数组中出队列。不断循环直到两个数组都为空。

具体代码如下：

```
// 将nums1 和 nums2 合并
function merge(nums1, nums2) {
  let ret = [];
  while (nums1.length || nums2.length) {
    // 为了方便大家理解，这里代码有点赘余
    if (nums1.length === 0) {
      ret.push(nums2.shift());
      continue;
    }

    if (nums2.length === 0) {
      ret.push(nums1.shift());
      continue;
    }
    const a = nums1[0];
    const b = nums2[0];
    if (a > b) {
      ret.push(nums2.shift());
    } else {
      ret.push(nums1.shift());
    }
  }
  return ret;
}
```

但是 merge sort 很多时候，合并的时候我们通常是新建一个数组，但是这道题目要求的是 `原地修改` .这就和 merge sort 的 merge 过程有点不同。这里要求原地修改。如果使用类似上面的方法，如果采用从头开始遍

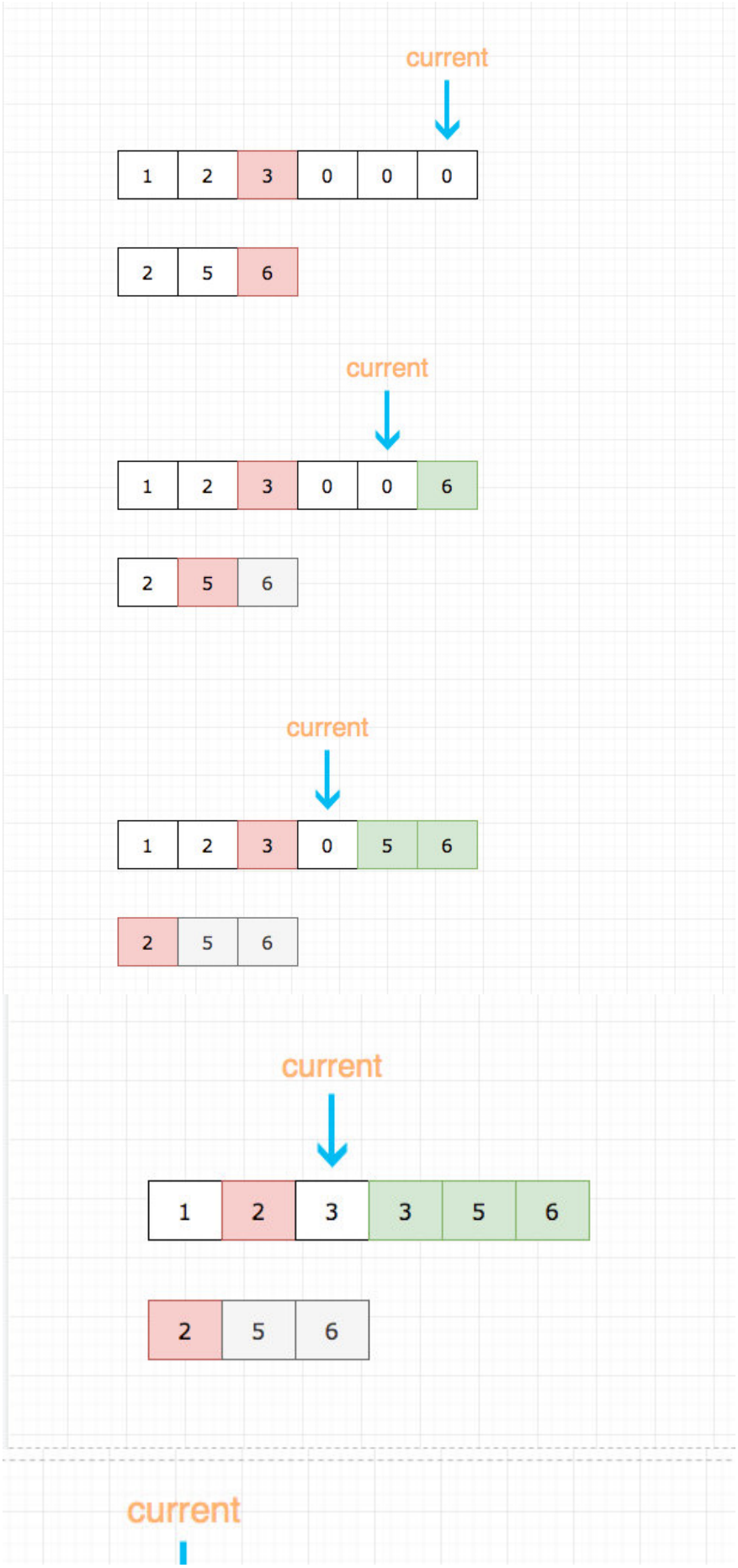
历，**需要将 nums1 的前 m 个数组放到另一个数组中避免写指针写入的干扰。** 这样空间复杂度就是 $O(m)$。其实我们能只要从**后往前比较，并从后往前插入即可。**

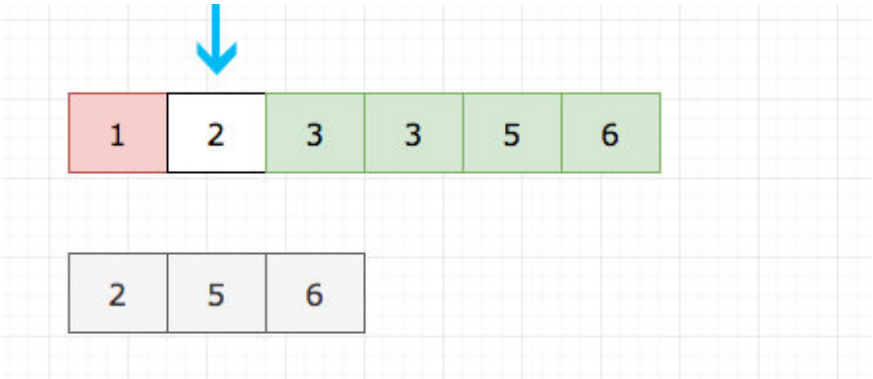
我们需要三个指针：

1. 写指针 current， 用于记录当前填补到那个位置了
2. m 用于记录 nums1 数组处理到哪个元素了
3. n 用于记录 nums2 数组处理到哪个元素了

如图所示：

- 灰色代表 num2 数组已经处理过的元素
- 红色代表当前正在进行比较的元素
- 绿色代表已经就位的元素


current
1 2 3 0 0 0
2 5 6
current
1 2 3 0 0 6
2 5 6
current
1 2 3 0 5 6
2 5 6
current
1 2 3 3 5 6
2 5 6
current



关键点解析

- 从后往前比较，并从后往前插入，这样可避免写指针影响，同时将空间复杂度降低到 $O(1)$

代码

代码支持：Python3, C++, Java, JavaScript

JavaSCript Code:

```
var merge = function (nums1, m, nums2, n) {
  // 设置一个指针，指针初始化指向nums1的末尾（根据#62，应该是index
  // 然后不断左移指针更新元素
  let current = m + n - 1;

  while (current >= 0) {
    // 没必要继续了
    if (n === 0) return;

    // 为了方便大家理解，这里代码有点赘余
    if (m < 1) {
      nums1[current--] = nums2[--n];
      continue;
    }

    if (n < 1) {
      nums1[current--] = nums1[--m];
      continue;
    }
    // 取大的填充 nums1的末尾
    // 然后更新 m 或者 n
    if (nums1[m - 1] > nums2[n - 1]) {
      nums1[current--] = nums1[--m];
    } else {
      nums1[current--] = nums2[--n];
    }
  }
};
```

C++ code:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums
        int current = m + n - 1;
        while (current >= 0) {
            if (n == 0) return;
            if (m < 1) {
                nums1[current--] = nums2[--n];
                continue;
            }
            if (n < 1) {
                nums1[current--] = nums1[--m];
                continue;
            }
            if (nums1[m - 1] > nums2[n - 1]) nums1[current-
            else nums1[current--] = nums2[--n];
        }
    }
};
```

Java Code:

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int
        int i=m-1, j=n-1, k=m+n-1;
        // 合并
        while(i>=0 && j>=0)
        {
            if(nums1[i] > nums2[j])
            {
                nums1[k--] = nums1[i--];
            }
            else
            {
                nums1[k--] = nums2[j--];
            }
        }
        // 合并剩余的nums2
        while(j>=0)
        {
            nums1[k--] = nums2[j--];
        }
    }
}
```

Python Code:

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[
        """
        Do not return anything, modify nums1 in-place inste
        """
        pos = m + n - 1
        while m > 0 and n > 0:
            if nums1[m - 1] < nums2[n - 1]:
                nums1[pos] = nums2[n - 1]
                n -= 1
            else:
                nums1[pos] = nums1[m - 1]
                m -= 1
            pos -= 1
        while n > 0:
            nums1[pos] = nums2[n - 1]
            n -= 1
            pos -= 1
```

复杂度分析

- 时间复杂度：$O(M + N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(101. 对称二叉树)

https://leetcode-cn.com/problems/symmetric-tree/

题目描述

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?

公司

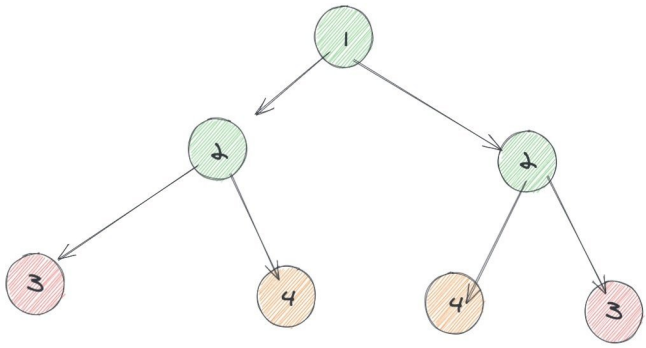
- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- linkedin
- microsoft

前置知识

- 二叉树
- 递归

思路

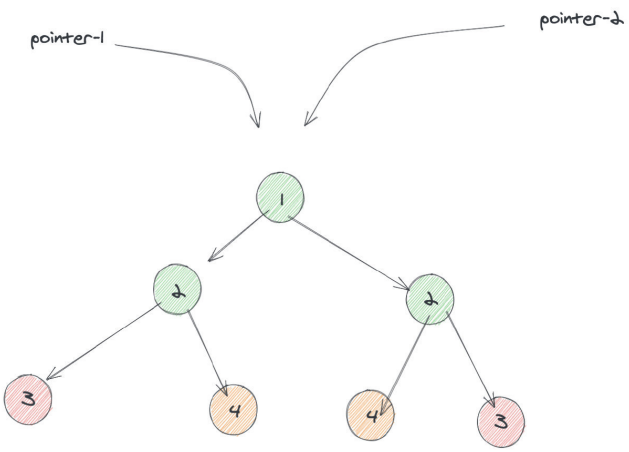
看到这题的时候，我的第一直觉是 DFS。然后我就想：`如果左子树是镜像，并且右子树也是镜像，是不是就说明整体是镜像？` 。经过几秒的思考， 这显然是不对的，不符合题意。



很明显其中左子树中的节点会和右子树中的节点进行比较，我把比较的元素进行了颜色区分，方便大家看。

这里我的想法是：`遍历每一个节点的时候，如果我都可以通过某种方法知道它对应的对称节点是谁，这样的话我直接比较两者是否一致就行了。`

因此想法是两次遍历，第一次遍历的同时将遍历结果存储到哈希表中，然后第二次遍历去哈希表取。这种方法可行，但是需要 N 的空间（N 为节点总数）。我想到如果两者可以同时进行遍历，是不是就省去了哈希表的开销。



如果不明白的话，我举个简单例子：

```
给定一个数组，检查它是否是镜像对称的。例如，数组 [1,2,2,3,2,2,1] 是
```

如果用哈希表的话大概是：

```
seen = dict()
for i, num in enumerate(nums):
    seen[i] = num
for i, num in enumerate(nums):
    if  seen[len(nums) - 1 - i] != num:
      return False
return True
```

而同时遍历的话大概是这样的：

```
l = 0
r = len(nums) - 1

while l < r:
   if nums[l] != nums[r]: return False
   l += 1
   r -= 1
return True
```

> 其实更像本题一点的话应该是从中间分别向两边扩展 😂

代码

代码支持：C++, Java, Python3

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        return root==NULL?true:recur(root->left, root->right
    }

    bool recur(TreeNode* l, TreeNode* r)
    {
        if(l == NULL && r==NULL)
        {
            return true;
        }
        // 只存在一个子节点 或者左右不相等
        if(l==NULL || r==NULL || l->val != r->val)
        {
            return false;
        }

        return recur(l->left, r->right) && recur(l->right,
    }
};
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root == null)
        {
            return true;
        }
        else{
            return recur(root.left, root.right);
        }
        // return root == null ? true : recur(root.left, r
    }

    public boolean recur(TreeNode l, TreeNode r)
    {
        if(l == null && r==null)
        {
            return true;
        }
        // 只存在一个子节点 或者左右不相等
        if(l==null || r==null || l.val != r.val)
        {
            return false;
        }

        return recur(l.left, r.right) && recur(l.right, r.
    }
}
```

Python3 Code:

```
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        def dfs(root1, root2):
            if root1 == root2 == None: return True
            if not root1 or not root2: return False
            if root1.val != root2.val: return False
            return dfs(root1.left, root2.right) and dfs(roo
        if not root: return True
        return dfs(root.left, root.right)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为节点数。
- 空间复杂度：递归的深度最高为节点数，因此空间复杂度是 $O(N)$，其中 N 为节点数。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（104. 二叉树的最大深度）

https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/description/

题目描述

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 。

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节
- apple
- linkedin
- uber
- yahoo

思路

由于树是一种递归的数据结构，因此用递归去解决的时候往往非常容易，这道题恰巧也是如此，

- 用递归实现的代码如下：

```
var maxDepth = function (root) {
  if (!root) return 0;
  if (!root.left && !root.right) return 1;
  return 1 + Math.max(maxDepth(root.left), maxDepth(root.r
};
```

如果使用迭代呢？ 我们首先应该想到的是树的各种遍历，由于我们求的是深度，因此 使用层次遍历（BFS）是非常合适的。 我们只需要记录有多少层即可。相关思路请查看binary-tree-traversal

关键点解析

- 队列
- 队列中用 Null(一个特殊元素)来划分每层，或者在对每层进行迭代之前保存当前队列元素的个数（即当前层所含元素个数）
- 树的基本操作- 遍历 - 层次遍历（BFS）

代码

- 语言支持：JS，C++，Java，Python, Go, PHP

JS Code:

```javascript
/*
 * @lc app=leetcode id=104 lang=javascript
 *
 * [104] Maximum Depth of Binary Tree
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function (root) {
  if (!root) return 0;
  if (!root.left && !root.right) return 1;

  // 层次遍历 BFS
  let cur = root;
  const queue = [root, null];
  let depth = 1;

  while ((cur = queue.shift()) !== undefined) {
    if (cur === null) {
      // 注意⚠️：不处理会无限循环，进而堆栈溢出
      if (queue.length === 0) return depth;
      depth++;
      queue.push(null);
      continue;
    }
    const l = cur.left;
    const r = cur.right;

    if (l) queue.push(l);
    if (r) queue.push(r);
  }

  return depth;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) return 0;
        auto q = vector<TreeNode*>();
        auto d = 0;
        q.push_back(root);
        while (!q.empty())
        {
            ++d;
            auto sz = q.size();
            for (auto i = 0; i < sz; ++i)
            {
                auto t = q.front();
                q.erase(q.begin());
                if (t->left != nullptr) q.push_back(t->left
                if (t->right != nullptr) q.push_back(t->rig
            }
        }
        return d;
    }
};
```

Java Code:

```java
/**
 * 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) {
        if(root == null)
        {
            return 0;
        }
        // 队列
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);
        int res = 0;
        // 按层扩展
        while(!queue.isEmpty())
        {
            // 拿出该层所有节点，并压入子节点
            int size = queue.size();
            while(size > 0)
            {
                TreeNode node = queue.poll();

                if(node.left != null)
                {
                    queue.offer(node.left);
                }
                if(node.right != null)
                {
                    queue.offer(node.right);
                }
                size-=1;
            }
            // 统计层数
            res +=1;
        }
        return res;
    }
}
```

Python Code:

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root: return 0
        q, depth = [root, None], 1
        while q:
            node = q.pop(0)
            if node:
                if node.left: q.append(node.left)
                if node.right: q.append(node.right)
            elif q:
                q.append(None)
                depth += 1
        return depth
```

Go Code:

```go
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
// BFS
func maxDepth(root *TreeNode) int {
    if root == nil {
        return 0
    }

    depth := 1
    q := []*TreeNode{root, nil} // queue
    var node *TreeNode
    for len(q) > 0 {
        node, q = q[0], q[1:] // pop
        if node != nil {
            if node.Left != nil {
                q = append(q, node.Left)
            }
            if node.Right != nil {
                q = append(q, node.Right)
            }
        } else if len(q) > 0 { // 注意要判断队列是否只有一个 ni
            q = append(q, nil)
            depth++
        }
    }
    return depth
}
```

PHP Code:

```php
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     public $val = null;
 *     public $left = null;
 *     public $right = null;
 *     function __construct($value) { $this->val = $value;
 * }
 */
class Solution
{

    /**
     * @param TreeNode $root
     * @return Integer
     */
    function maxDepth($root)
    {
        if (!$root) return 0;

        $depth = 1;
        $arr = [$root, null];
        while ($arr) {
            /** @var TreeNode $node */
            $node = array_shift($arr);
            if ($node) {
                if ($node->left) array_push($arr, $node->le
                if ($node->right) array_push($arr, $node->r
            } elseif ($arr) {
                $depth++;
                array_push($arr, null);
            }
        }
        return $depth;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关题目

- 102.binary-tree-level-order-traversal
- 103.binary-tree-zigzag-level-order-traversal

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(108. 将有序数组转换为二叉搜索树)

https://leetcode-cn.com/problems/convert-sorted-array-to-binary-search-tree/

题目描述

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

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

```
      0
     / \
   -3   9
   /   /
 -10  5
```

前置知识

- 二叉搜索树
- 平衡二叉树
- 递归

公司

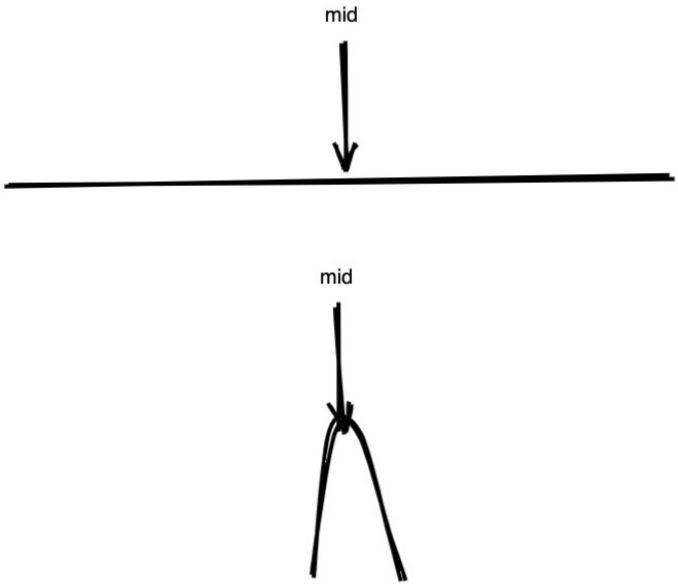
- 阿里
- 腾讯
- 百度
- 字节
- airbnb

思路

由于输入是一个**升序排列的有序数组**。因此任意选择一点，将其作为根节点，其左部分左节点，其右部分右节点即可。 因此我们很容易写出递归代码。

而题目要求是**高度平衡**的二叉搜索树，因此我们必须要取中点。 不难证明：`由于是中点，因此左右两部分差不会大于 1，也就是说其形成的左右子树节点数最多相差 1，因此左右子树高度差的绝对值不超过 1` 。

形象一点来看就像你提起一根绳子，从中点提的话才能使得两边绳子长度相差最小。



关键点

- 找中点

代码

代码支持：JS，C++，Java，Python

JS Code:

```
var sortedArrayToBST = function (nums) {
  // 由于数组是排序好的，因此一个思路就是将数组分成两半，一半是左子树
  // 然后运用“树的递归性质”递归完成操作即可。
  if (nums.length === 0) return null;
  const mid = nums.length >> 1;
  const root = new TreeNode(nums[mid]);

  root.left = sortedArrayToBST(nums.slice(0, mid));
  root.right = sortedArrayToBST(nums.slice(mid + 1));
  return root;
};
```

Python Code:

```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> TreeNode
        if not nums: return None
        mid = (len(nums) - 1) // 2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid + 1:])
        return root
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：每次递归都 copy 了 N 的 空间，因此空间复杂度为 $O(N ^ 2)$

然而，实际上没必要开辟新的空间：

C++ Code:

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return reBuild(nums, 0, nums.size()-1);
    }

    TreeNode* reBuild(vector<int>& nums, int left, int rig
    {
        // 终止条件：中序遍历为空
        if(left > right)
        {
            return NULL;
        }
        // 建立当前子树的根节点
        int mid = (left+right)/2;
        TreeNode * root = new TreeNode(nums[mid]);

        // 左子树的下层递归
        root->left = reBuild(nums, left, mid-1);
        // 右子树的下层递归
        root->right = reBuild(nums, mid+1, right);
        // 返回根节点
        return root;
    }
};
```

Java Code:

```
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return dfs(nums, 0, nums.length - 1);
    }

    private TreeNode dfs(int[] nums, int lo, int hi) {
        if (lo > hi) {
            return null;
        }
        int mid = lo + (hi - lo) / 2;
        TreeNode root = new TreeNode(nums[mid]);
        root.left = dfs(nums, lo, mid - 1);
        root.right = dfs(nums, mid + 1, hi);
        return root;
    }
}
```

Python Code:

```
class Solution(object):
    def sortedArrayToBST(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        return self.reBuild(nums, 0, len(nums)-1)

    def reBuild(self, nums, left, right):
         # 终止条件:
        if left > right:
            return
        # 建立当前子树的根节点
        mid = (left + right)//2
        root = TreeNode(nums[mid])
        # 左右子树的下层递归
        root.left = self.reBuild(nums, left, mid-1)
        root.right = self.reBuild(nums, mid+1, right)

        return root
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：由于是平衡二叉树，因此隐式调用栈的开销为 $O(logN)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

公众号【 力扣加加】 知乎专栏【 Lucifer - 知乎】

点关注，不迷路！

题目地址(121. 买卖股票的最佳时机)

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/description/

题目描述

```
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

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）
     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
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。
```

前置知识

- 数组

公司

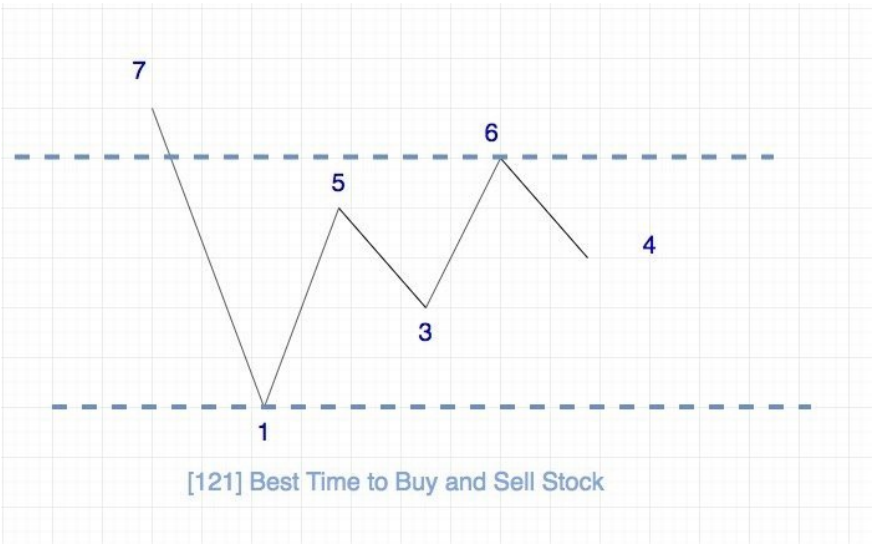
- 阿里
- 腾讯
- 百度
- 字节
- amazon
- bloomberg
- facebook
- microsoft
- uber

思路

由于我们是想获取到最大的利润，我们的策略应该是低点买入，高点卖出。

由于题目对于交易次数有限制，只能交易一次，因此问题的本质其实就是求波峰浪谷的差值的最大值。

用图表示的话就是这样：



关键点解析

- 这类题只要你在心中（或者别的地方）画出上面这种图就很容易解决

代码

语言支持：JS，C++，Java，Python

JS Code:

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
  let min = prices[0];
  let profit = 0;
  // 7 1 5 3 6 4
  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
      profit = Math.max(profit, prices[i] - min);
    } else {
      min = Math.min(min, prices[i]);
    }
  }

  return profit;
};
```

C++ Code:

```
/**
 * 系统上C++的测试用例中的输入有[]，因此需要加一个判断
 */
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;
        auto min = prices[0];
        auto profit = 0;
        for (auto i = 1; i < prices.size(); ++i) {
            if (prices[i] > prices[i -1]) {
                profit = max(profit, prices[i] - min);
            } else {
                min = std::min(min, prices[i]);;
            }
        }
        return profit;
    }
};
```

Java Code:

```java
class Solution {
    public int maxProfit(int[] prices) {
        int minprice = Integer.MAX_VALUE;
        int maxprofit = 0;
        for (int price: prices) {
            maxprofit = Math.max(maxprofit, price - minpric
             minprice = Math.min(price, minprice);
        }
        return maxprofit;
    }
}
```

Python Code:

```python
class Solution:
    def maxProfit(self, prices: 'List[int]') -> int:
        if not prices: return 0

        min_price = float('inf')
        max_profit = 0

        for price in prices:
            if price < min_price:
                min_price = price
            elif max_profit < price - min_price:
                max_profit = price - min_price
        return max_profit
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 122.best-time-to-buy-and-sell-stock-ii
- 309.best-time-to-buy-and-sell-stock-with-cooldown

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(122. 买卖股票的最佳时机 II)

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/description/

题目描述

```
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（

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）
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格
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
     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
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。


提示：

1 <= prices.length <= 3 * 10 ^ 4
0 <= prices[i] <= 10 ^ 4
```

前置知识

- 数组

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg

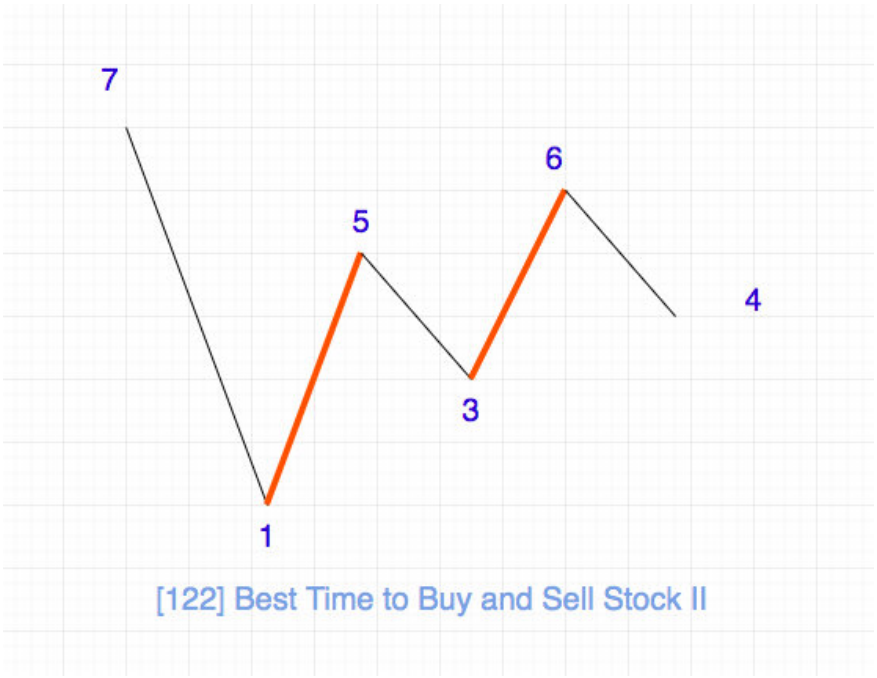
思路

由于我们是想获取到最大的利润，我们的策略应该是低点买入，高点卖出。

由于题目对于交易次数没有限制，因此只要能够赚钱的机会我们都不应该放过。

> 如下图，我们只需要求出加粗部分的总和即可

用图表示的话就是这样：



关键点解析

- 这类题只要你在心中（或者别的地方）画出上面这种图就很容易解决

代码

语言支持：JS，C++，Java

JS Code:

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
  let profit = 0;

  for (let i = 1; i < prices.length; i++) {
    if (prices[i] > prices[i - 1]) {
      profit = profit + prices[i] - prices[i - 1];
    }
  }

  return profit;
};
```

C++ Code:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int res = 0;
        for(int i=1;i<prices.size();i++)
        {
            if(prices[i] > prices[i-1])
            {
                res += prices[i] - prices[i-1];
            }
        }
        return res;
    }
};
```

Java Code:

```
class Solution {
    public int maxProfit(int[] prices) {
        int res = 0;
        for(int i=1;i<prices.length;i++)
        {
            if(prices[i] > prices[i-1])
            {
                res += prices[i] - prices[i-1];
            }
        }
        return res;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 121.best-time-to-buy-and-sell-stock
- 309.best-time-to-buy-and-sell-stock-with-cooldown

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(125. 验证回文串)

https://leetcode-cn.com/problems/valid-palindrome/description/

题目描述

```
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

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
```

前置知识

- 回文
- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- facebook
- microsoft
- uber
- zenefits

思路

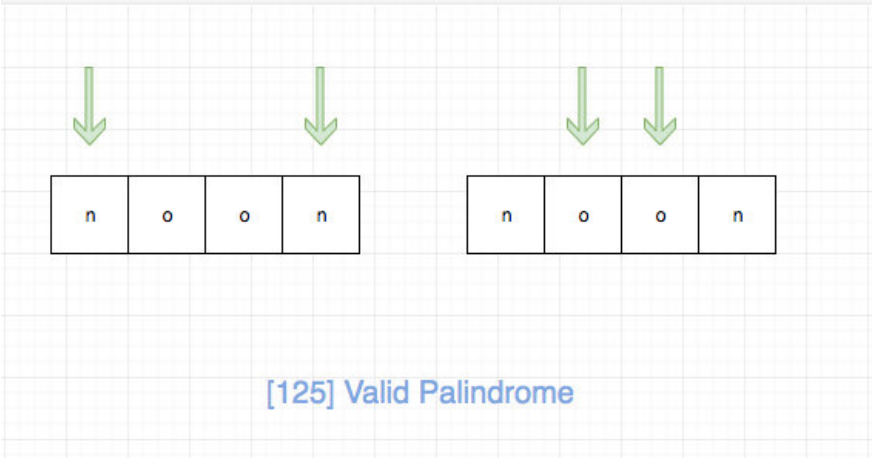
这是一道考察回文的题目，而且是最简单的形式，即判断一个字符串是否是回文。

针对这个问题，我们可以使用头尾双指针，

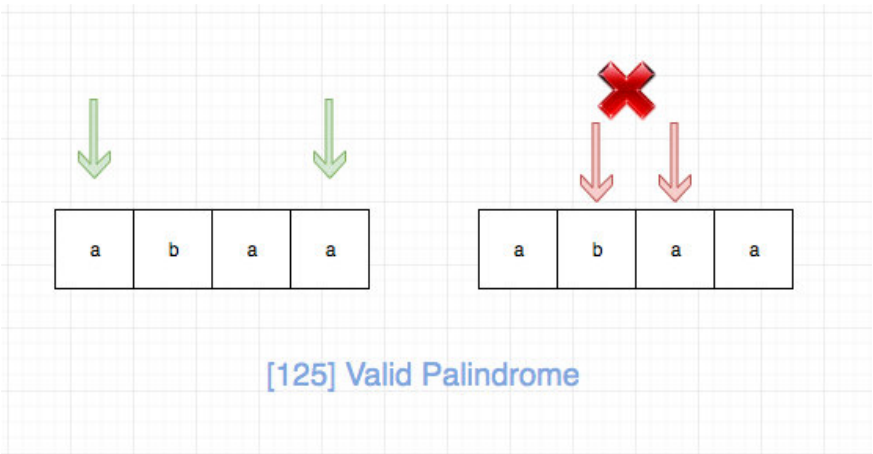
- 如果两个指针的元素不相同，则直接返回 false,
- 如果两个指针的元素相同，我们同时更新头尾指针，循环。 直到头尾指针相遇。

时间复杂度为 O(n).

拿“noon”这样一个回文串来说，我们的判断过程是这样的：



拿“abaa”这样一个不是回文的字符串来说，我们的判断过程是这样的：



关键点解析

- 双指针

代码

- 语言支持：JS，C++，Python

JavaScript Code:

```
/*
 * @lc app=leetcode id=125 lang=javascript
 *
 * [125] Valid Palindrome
 */
// 只处理英文字符（题目忽略大小写，我们前面全部转化成了小写， 因此这里
function isValid(c) {
  const charCode = c.charCodeAt(0);
  const isDigit =
    charCode >= "0".charCodeAt(0) && charCode <= "9".charC
  const isChar = charCode >= "a".charCodeAt(0) && charCode

  return isDigit || isChar;
}
/**
 * @param {string} s
 * @return {boolean}
 */
var isPalindrome = function (s) {
  s = s.toLowerCase();
  let left = 0;
  let right = s.length - 1;

  while (left < right) {
    if (!isValid(s[left])) {
      left++;
      continue;
    }
    if (!isValid(s[right])) {
      right--;
      continue;
    }

    if (s[left] === s[right]) {
      left++;
      right--;
    } else {
      break;
    }
  }

  return right <= left;
};
```

C++ Code:

```
class Solution {
public:
    bool isPalindrome(string s) {
        if (s.empty())
            return true;
        const char* s1 = s.c_str();
        const char* e = s1 + s.length() - 1;
        while (e > s1) {
            if (!isalnum(*s1)) {++s1; continue;}
            if (!isalnum(*e)) {--e; continue;}
            if (tolower(*s1) != tolower(*e)) return false;
            else {--e; ++s1;}
        }
        return true;
    }
};
```

Python Code:

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        left, right = 0, len(s) - 1
        while left < right:
            if not s[left].isalnum():
                left += 1
                continue
            if not s[right].isalnum():
                right -= 1
                continue
            if s[left].lower() == s[right].lower():
                left += 1
                right -= 1
            else:
                break
        return right <= left

    def isPalindrome2(self, s: str) -> bool:
        """
        使用语言特性进行求解
        """
        s = ''.join(i for i in s if i.isalnum()).lower()
        return s == s[::-1]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(136. 只出现一次的数字)

https://leetcode-cn.com/problems/single-number/

题目描述

```
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

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
```

前置知识

- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

根据题目描述，由于加上了时间复杂度必须是 O(n)，并且空间复杂度为 O(1)的条件，因此不能用排序方法，也不能使用 map 数据结构。

我们可以利用二进制异或的性质来完成，将所有数字异或即得到唯一出现的数字。

关键点

1. 异或的性质 两个数字异或的结果 `a^b` 是将 a 和 b 的二进制每一位进行运算，得出的数字。 运算的逻辑是 如果同一位的数字相同则为 0，不同则为 1
2. 异或的规律
3. 任何数和本身异或则为 `0`
4. 任何数和 0 异或是 `本身`
5. 很多人只是记得异或的性质和规律，但是缺乏对其本质的理解，导致很难想到这种解法（我本人也没想到）
6. bit 运算

代码

- 语言支持：JS，C，C++，Java，Python

JavaScrip Code：

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var singleNumber = function (nums) {
  let ret = 0;
  for (let index = 0; index < nums.length; index++) {
    const element = nums[index];
    ret = ret ^ element;
  }
  return ret;
};
```

C Code：

```
int singleNumber(int* nums, int numsSize){
    int res=0;
    for(int i=0;i<numsSize;i++)
    {
        res ^= nums[i];
    }

    return res;
}
```

C++ Code：

```cpp
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        auto ret = 0;
        for (auto i : nums) ret ^= i;
        return ret;
    }
};

// C++ one-liner
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        return accumulate(nums.cbegin(), nums.cend(), 0, bi
    }
};
```

Java Code：

```java
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for(int n:nums)
        {
            // 异或
            res ^= n;
        }
        return res;
    }
}
```

Python Code:

```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        single_number = 0
        for num in nums:
            single_number ^= num
        return single_number
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

延伸

有一个 n 个元素的数组，除了两个数只出现一次外，其余元素都出现两次，让你找出这两个只出现一次的数分别是几，要求时间复杂度为 O(n) 且再开辟的内存空间固定(与 n 无关)。

和上面一样，只是这次不是一个数字，而是两个数字。还是按照上面的思路，我们进行一次全员异或操作， 得到的结果就是那两个只出现一次的不同的数字的异或结果。

我们刚才讲了异或的规律中有一个 `任何数和本身异或则为0` ， 因此我们的思路是能不能将这两个不同的数字分成两组 A 和 B。 分组需要满足两个条件.

1. 两个独特的的数字分成不同组
2. 相同的数字分成相同组

这样每一组的数据进行异或即可得到那两个数字。

问题的关键点是我们怎么进行分组呢?

由于异或的性质是，同一位相同则为 0，不同则为 1. 我们将所有数字异或的结果一定不是 0，也就是说至少有一位是 1.

我们随便取一个， 分组的依据就来了， 就是你取的那一位是 0 分成 1 组，那一位是 1 的分成一组。 这样肯定能保证 `2. 相同的数字分成相同组` ，不同的数字会被分成不同组么。 很明显当然可以， 因此我们选择是 1，也就是 说 `两个独特的的数字` 在那一位一定是不同的，因此两个独特元素一定会被分成不同组。

Done!

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(155. 最小栈)

https://leetcode-cn.com/problems/min-stack/

题目描述

```
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

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","getM
[[],[-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。
```

前置知识

- 栈

公司

- amazon
- bloomberg
- google
- snapchat
- uber
- zenefits
- 阿里
- 腾讯
- 百度
- 字节

两个栈

思路

我们使用两个栈：

- 一个栈存放全部的元素，push，pop都是正常操作这个正常栈。
- 另一个存放最小栈。 每次push，如果比最小栈的栈顶还小，我们就push进最小栈，否则不操作
- 每次pop的时候，我们都判断其是否和最小栈栈顶元素相同，如果相同，那么我们pop掉最小栈的栈顶元素即可

关键点

- 往minstack中 push的判断条件。 应该是stack为空或者x小于等于minstack栈顶元素

代码

- 语言支持：JS，C++，Java，Python

JavaScript Code:

```
/**
 * initialize your data structure here.
 */
var MinStack = function() {
    this.stack = []
    this.minStack = []
};

/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function(x) {
    this.stack.push(x)
    if (this.minStack.length == 0 ||  x <= this.minStack[th
        this.minStack.push(x)
    }
};

/**
 * @return {void}
 */
MinStack.prototype.pop = function() {
    const x = this.stack.pop()
    if (x !== void 0 &&  x === this.minStack[this.minStack.
        this.minStack.pop()
    }
};

/**
 * @return {number}
 */
MinStack.prototype.top = function() {
    return this.stack[this.stack.length - 1]
};

/**
 * @return {number}
 */
MinStack.prototype.min = function() {
    return this.minStack[this.minStack.length - 1]
};

/**
 * Your MinStack object will be instantiated and called as
 * var obj = new MinStack()
 * obj.push(x)
 * obj.pop()
```

```
 * var param_3 = obj.top()
 * var param_4 = obj.min()
 */
```

C++ Code:

```
class MinStack {
    stack<int> data;
    stack<int> helper;
public:
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        data.push(x);
        if(helper.empty() || helper.top() >= x)
        {
            helper.push(x);
        }

    }

    void pop() {
        int top = data.top();
        data.pop();
        if(top == helper.top())
        {
            helper.pop();
        }

    }

    int top() {
        return data.top();
    }

    int getMin() {
        return helper.top();
    }
};

/**
 * Your MinStack object will be instantiated and called as
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

Java Code:

```java
public class MinStack {

    // 数据栈
    private Stack<Integer> data;
    // 辅助栈
    private Stack<Integer> helper;

    /**
     * initialize your data structure here.
     */
    public MinStack() {
        data = new Stack<>();
        helper = new Stack<>();
    }

    public void push(int x) {
        // 辅助栈在必要的时候才增加
        data.add(x);
        if (helper.isEmpty() || helper.peek() >= x) {
            helper.add(x);
        }
    }

    public void pop() {
        // 关键 3：data 一定得 pop()
        if (!data.isEmpty()) {
            // 注意：声明成 int 类型，这里完成了自动拆箱，从 Inte
            // 因此下面的比较可以使用 "==" 运算符
            int top = data.pop();
            if(top == helper.peek()){
                helper.pop();
            }
        }
    }

    public int top() {
        if(!data.isEmpty()){
            return data.peek();
        }
    }

    public int getMin() {
        if(!helper.isEmpty()){
            return helper.peek();
        }
    }
}
```

Python3 Code:

```
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.minstack = []

    def push(self, x: int) -> None:
        self.stack.append(x)
        if not self.minstack or x <= self.minstack[-1]:
            self.minstack.append(x)

    def pop(self) -> None:
        tmp = self.stack.pop()
        if tmp == self.minstack[-1]:
            self.minstack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def min(self) -> int:
        return self.minstack[-1]


# Your MinStack object will be instantiated and called as s
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.min()
```

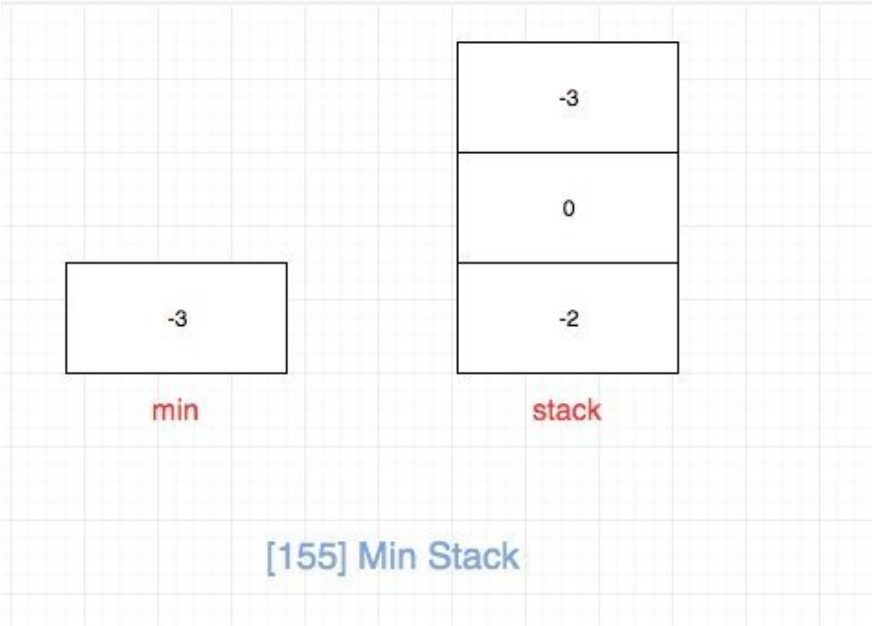
复杂度分析

- 时间复杂度：O(1)
- 空间复杂度：O(1)

一个栈

思路

符合直觉的方法是，每次对栈进行修改操作（push和pop）的时候更新最小值。 然后getMin只需要返回我们计算的最小值即可， top也是直接返回栈顶元素即可。 这种做法每次修改栈都需要更新最小值，因此时间复杂度是O(n).



是否有更高效的算法呢？答案是有的。

我们每次入栈的时候，保存的不再是真正的数字，而是它与当前最小值的差（当前元素没有入栈的时候的最小值）。 这样我们pop和top的时候拿到栈顶元素再加上**上一个**最小值即可。 另外我们在push和pop的时候去更新min，这样getMin的时候就简单了，直接返回min。

> 注意上面加粗的“上一个”，不是“当前的最小值”

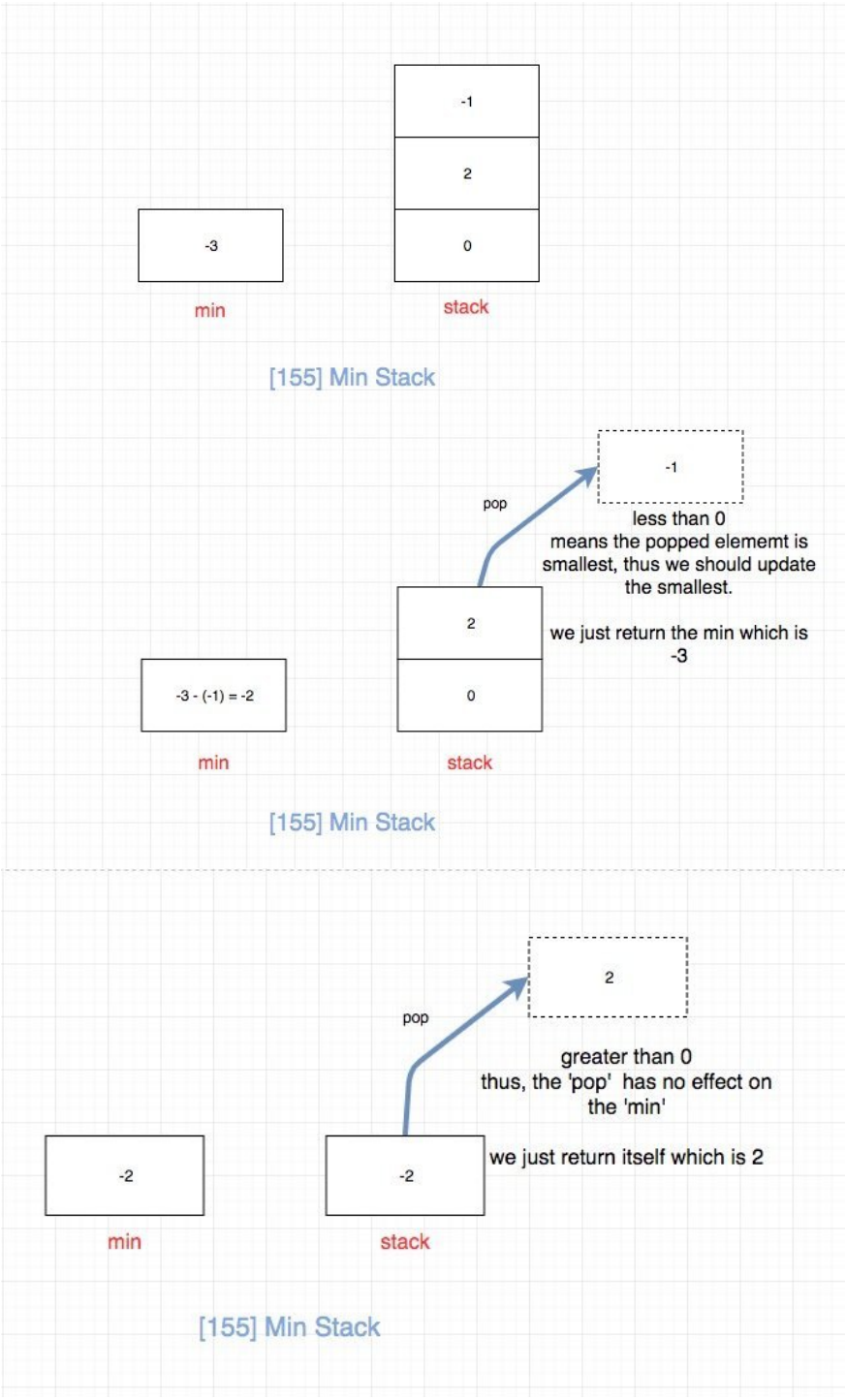
经过上面的分析，问题的关键转化为“如何求得上一个最小值”，解决这个的关键点在于利用min。

pop或者top的时候：

- 如果栈顶元素小于0，说明栈顶是当前最小的元素，它出栈会对min造成影响，我们需要去更新min。 上一个最小的是“min - 栈顶元素”,我们需要将上一个最小值更新为当前的最小值

  > 因为栈顶元素入栈的时候的通过 `栈顶元素 = 真实值 - 上一个最小的元素` 得到的， 而真实值 = min， 因此可以得出 `上一个最小的元素 = 真实值 -栈顶元素`

- 如果栈顶元素大于0，说明它对最小值 `没有影响` ，上一个最小值就是上上个最小值。



关键点

- 最小栈存储的不应该是真实值，而是真实值和min的差值
- top的时候涉及到对数据的还原，这里千万注意是**上一个**最小值

代码

- 语言支持：JS，C++，Java，Python

Javascript Code:

```
/*
 * @lc app=leetcode id=155 lang=javascript
 *
 * [155] Min Stack
 */
/**
 * initialize your data structure here.
 */
var MinStack = function() {
  this.stack = [];
  this.minV = Number.MAX_VALUE;
};

/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function(x) {
  // update 'min'
  const minV = this.minV;
  if (x < this.minV) {
    this.minV = x;
  }
  return this.stack.push(x - minV);
};

/**
 * @return {void}
 */
MinStack.prototype.pop = function() {
  const item = this.stack.pop();
  const minV = this.minV;

  if (item < 0) {
    this.minV = minV - item;
    return minV;
  }
  return item + minV;
};

/**
 * @return {number}
 */
MinStack.prototype.top = function() {
  const item = this.stack[this.stack.length - 1];
  const minV = this.minV;

  if (item < 0) {
```

```
    return minV;
  }
  return item + minV;
};

/**
 * @return {number}
 */
MinStack.prototype.min = function() {
  return this.minV;
};

/**
 * Your MinStack object will be instantiated and called as
 * var obj = new MinStack()
 * obj.push(x)
 * obj.pop()
 * var param_3 = obj.top()
 * var param_4 = obj.min()
 */
```

C++ Code:

```
class MinStack {
    stack<long> data;
    long min = INT_MAX;
public:
    /** initialize your data structure here. */
    MinStack() {

    }

    void push(int x) {
        data.push(x - min);
        if(x < min)
        {
            min = x;
        }

    }

    void pop() {
        long top = data.top();
        data.pop();
        // 更新最小值
        if(top < 0)
        {
            min -= top;
        }

    }

    int top() {
        long top = data.top();
        // 最小值为 min
        if (top < 0)
        {
            return min;
        }
        else{
            return min+top;
        }
    }

    int getMin() {
        return min;
    }
};

/**
 * Your MinStack object will be instantiated and called as
```

```
 * MinStack* obj = new MinStack();
 * obj->push(x);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

Java Code:

```java
class MinStack {
    long min;
    Stack<Long> stack;

    /** initialize your data structure here. */
    public MinStack() {
        stack = new Stack<>();
    }

    public void push(int x) {
        if (stack.isEmpty()) {
            stack.push(0L);
            min = x;
        }
        else {
            stack.push(x - min);
            if (x < min)
                min = x;
        }
    }

    public void pop() {
        long p = stack.pop();

        if (p < 0) {
            // if (p < 0), the popped value is the min
            // Recall p is added by this statement: stack.p
            // So, p = x - old_min
            // old_min = x - p
            // again, if (p < 0), x is the min so:
            // old_min = min - p
            min = min - p;
        }
    }

    public int top() {
        long p = stack.peek();

        if (p < 0) {
            return (int) min;
        }
        else {
            // p = x - min
            // x = p + min
            return (int) (p + min);
        }
    }
```

```
    public int getMin() {
        return (int) min;
    }
}
```

Python Code:

```python
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.minV = float('inf')
        self.stack = []

    def push(self, x: int) -> None:
        self.stack.append(x - self.minV)
        if x < self.minV:
            self.minV = x

    def pop(self) -> None:
        if not self.stack:
            return
        tmp = self.stack.pop()
        if tmp < 0:
            self.minV -= tmp

    def top(self) -> int:
        if not self.stack:
            return
        tmp = self.stack[-1]
        if tmp < 0:
            return self.minV
        else:
            return self.minV + tmp

    def min(self) -> int:
        return self.minV



# Your MinStack object will be instantiated and called as s
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.min()
```

复杂度分析

- 时间复杂度：O(1)
- 空间复杂度：O(1)

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经37K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(167. 两数之和 II - 输入有序数组)

https://leetcode-cn.com/problems/two-sum-ii-input-array-is-sorted/

题目描述

这是 leetcode 头号题目 `two sum` 的第二个版本，难度简单。

```
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 i

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 。
```

前置知识

- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- amazon

思路

由于题目没有对空间复杂度有求，用一个 hashmap 存储已经访问过的数字即可。

假如题目空间复杂度有要求，由于数组是有序的，只需要双指针即可。一个 left 指针，一个 right 指针， 如果 left + right 值 大于 target 则 right 左移动， 否则 left 右移，代码见下方 python code。

> 如果数组无序，需要先排序（从这里也可以看出排序是多么重要的操作）

关键点解析

- 由于是有序的，因此双指针更好

代码

- 语言支持：JS，C++，Java，Python

Javascript Code:

```
/**
 * @param {number[]} numbers
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function (numbers, target) {
  const visited = {}; // 记录出现的数字，  空间复杂度N

  for (let index = 0; index < numbers.length; index++) {
    const element = numbers[index];
    if (visited[target - element] !== void 0) {
      return [visited[target - element], index + 1];
    }
    visited[element] = index + 1;
  }
  return [];
};
```

C++ Code:

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int n = numbers.size();
        int left = 0;
        int right = n-1;
        while(left <= right)
        {
            if(numbers[left] + numbers[right] == target)
            {
                return {left + 1, right + 1};
            }
            else if (numbers[left] + numbers[right] > targe
            {
                right--;
            }
            else
            {
                left++;
            }
        }
        return {-1, -1};
    }
};
```

Java Code:

```java
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int n = numbers.length;
        int left = 0;
        int right = n-1;
        while(left <= right)
        {
            if(numbers[left] + numbers[right] == target)
            {
                return new int[]{left + 1, right + 1};
            }
            else if (numbers[left] + numbers[right] > targe
            {
                right--;
            }
            else
            {
                left++;
            }
        }

        return new int[]{-1, -1};
    }
}
```

Python Code:

```python
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> L
        visited = {}
        for index, number in enumerate(numbers):
            if target - number in visited:
                return [visited[target-number], index+1]
            else:
                visited[number] = index + 1

# 双指针思路实现
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> L
        left, right = 0, len(numbers) - 1
        while left < right:
            if numbers[left] + numbers[right] < target:
                left += 1
            if numbers[left] + numbers[right] > target:
                right -= 1
            if numbers[left] + numbers[right] == target:
                return [left+1, right+1]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(169. 多数元素)

https://leetcode-cn.com/problems/majority-element/

题目描述

```
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

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
```

前置知识

- 投票算法

公司

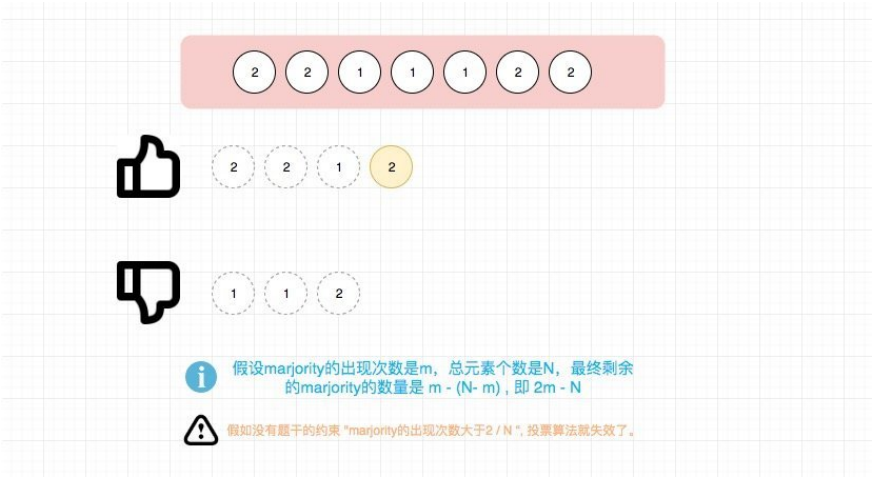
- 阿里
- 腾讯
- 百度
- 字节
- adobe
- zenefits

思路

符合直觉的做法是利用额外的空间去记录每个元素出现的次数，并用一个单独的变量记录当前出现次数最多的元素。

但是这种做法空间复杂度较高，有没有可能进行优化呢？ 答案就是用"投票算法"。

投票算法的原理是通过不断消除不同元素直到没有不同元素，剩下的元素就是我们要找的元素。



关键点解析

- 投票算法

代码

- 语言支持：JS，Python, CPP

Javascript Code:

```
var majorityElement = function (nums) {
  let count = 1;
  let majority = nums[0];
  for (let i = 1; i < nums.length; i++) {
    if (count === 0) {
      majority = nums[i];
    }
    if (nums[i] === majority) {
      count++;
    } else {
      count--;
    }
  }
  return majority;
};
```

Python Code:

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        count, majority = 1, nums[0]
        for num in nums[1:]:
            if count == 0:
                majority = num
            if num == majority:
                count += 1
            else:
                count -= 1
        return majority
```

CPP Code:

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int ans = 0, cnt = 0;
        for (int n : nums) {
            if (ans == n) ++cnt;
            else if (cnt > 0) --cnt;
            else {
                ans = n;
                cnt = 1;
            }
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(172. 阶乘后的零)

https://leetcode-cn.com/problems/factorial-trailing-zeroes/

题目描述

```
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) 。
```

前置知识

- 递归

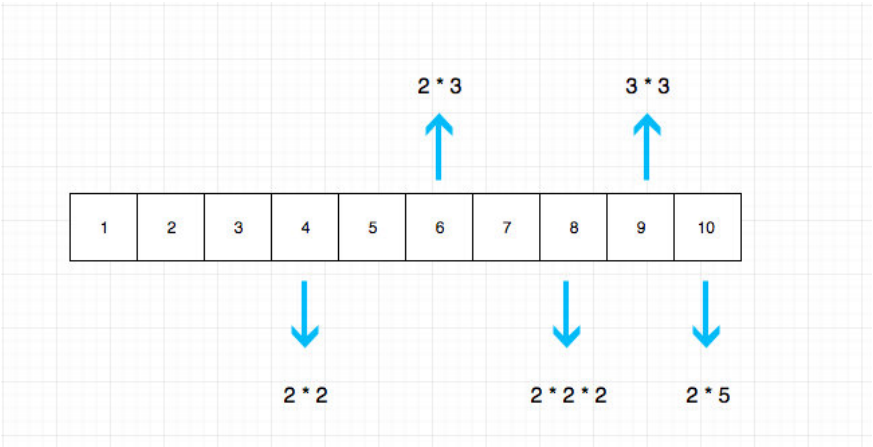
公司

- 阿里
- 腾讯
- 百度
- bloomberg

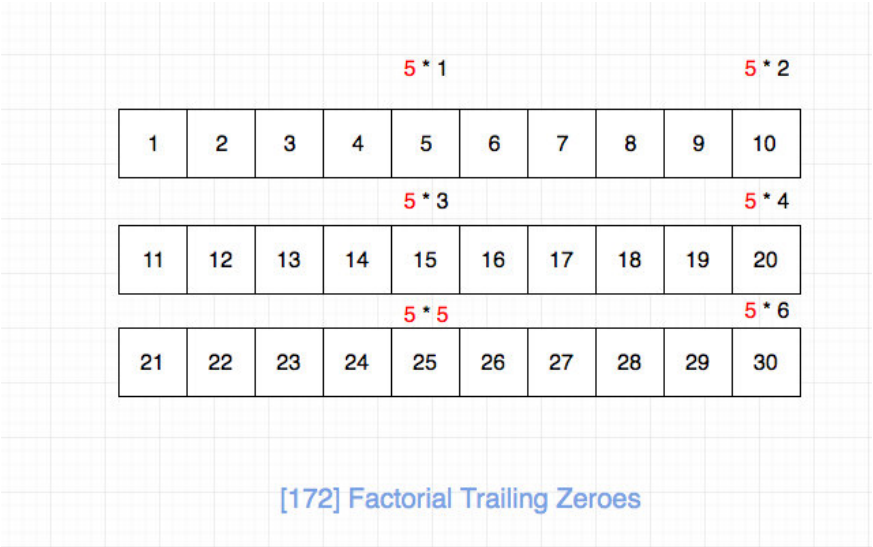
思路

我们需要求解这 n 个数字相乘的结果末尾有多少个 0，由于题目要求 log 的复杂度，因此暴力求解是不行的。

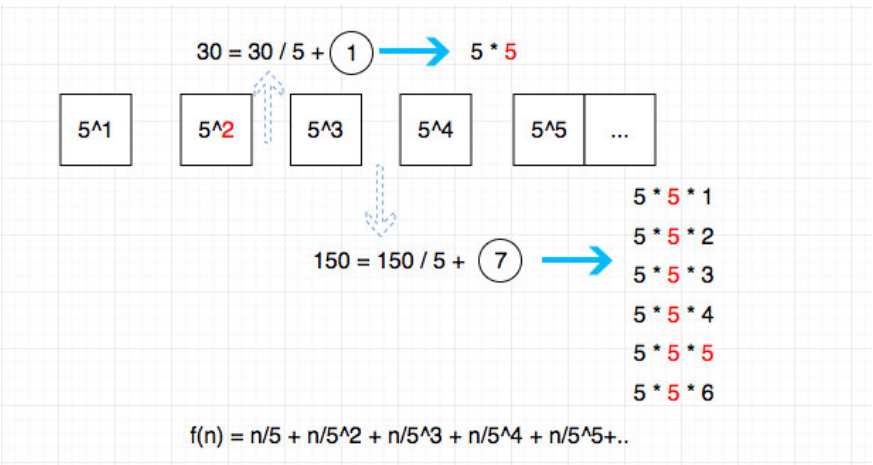
通过观察，我们发现如果想要结果末尾是 0，必须是分解质因数之后，2 和 5 相乘才行，同时因数分解之后发现 5 的个数远小于 2， 因此我们只需要求解这 n 数字分解质因数之后一共有多少个 5 即可.



如图如果 n 为 30，那么结果应该是图中红色 5 的个数，即 7。



我们的结果并不是直接 f(n) = n / 5, 比如 n 为 30， 25 中是有两个 5 的。类似，n 为 150，会有 7 个这样的数字，通过观察我们发现规律 `f(n) = n/5 + n/5^2 + n/5^3 + n/5^4 + n/5^5+..`



如果可以发现上面的规律，用递归还是循环实现这个算式就看你的了。

关键点解析

- 数论

代码

- 语言支持：JS，Python，C++, Java

Javascript Code:

```
/*
 * @lc app=leetcode id=172 lang=javascript
 *
 * [172] Factorial Trailing Zeroes
 */
/**
 * @param {number} n
 * @return {number}
 */
var trailingZeroes = function (n) {
  // tag: 数论

  // if (n === 0) return n;

  // 递归:  f(n) = n / 5 + f(n / 5)
  // return Math.floor(n / 5)  + trailingZeroes(Math.floor
  let count = 0;
  while (n >= 5) {
    count += Math.floor(n / 5);
    n = Math.floor(n / 5);
  }
  return count;
};
```

Python Code:

```python
class Solution:
    def trailingZeroes(self, n: int) -> int:
        count = 0
        while n >= 5:
            n = n // 5
            count += n
        return count


# 递归
class Solution:
    def trailingZeroes(self, n: int) -> int:
        if n == 0: return 0
        return n // 5 + self.trailingZeroes(n // 5)
```

C++ Code:

```cpp
class Solution {
public:
    int trailingZeroes(int n) {
        int res = 0;
        while(n >= 5)
        {
            n/=5;
            res += n;
        }
        return res;
    }
};
```

Java Code:

```java
class Solution {
    public int trailingZeroes(int n) {
        int res = 0;
        while(n >= 5)
        {
            n/=5;
            res += n;
        }
        return res;
    }
}
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(190. 颠倒二进制位)

https://leetcode-cn.com/problems/reverse-bits/

题目描述

```
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
     因此返回 964176192，其二进制表示形式为 00111001011110000001
示例 2：

输入：11111111111111111111111111111101
输出：10111111111111111111111111111111
解释：输入的二进制串 11111111111111111111111111111101 表示无符号
     因此返回 3221225471 其二进制表示形式为 10111111111111111111


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
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


进阶：
如果多次调用这个函数，你将如何优化你的算法？
```

前置知识

- 双指针

公司

- 阿里
- 腾讯
- 百度
- airbnb

- apple

思路

这道题是给定一个 32 位的无符号整型，让你按位翻转， 第一位变成最后一位， 第二位变成倒数第二位。。。

那么思路就是 `双指针`

> 这个指针可以加引号

- n 从高位开始逐步左， res 从低位（0）开始逐步右移
- 逐步判断，如果该位是 1，就 res + 1 , 如果是该位是 0， 就 res + 0
- 32 位全部遍历完，则遍历结束

关键点解析

1. 可以用任何数字和 1 进行位运算的结果都取决于该数字最后一位的特性简化操作和提高性能

eg :

- n & 1 === 1, 说明 n 的最后一位是 1
- n & 1 === 0, 说明 n 的最后一位是 0

- 对于 JS，ES 规范在之前很多版本都是没有无符号整形的， 转化为无符号，可以用一个 trick `n >>> 0`

- 双"指针" 模型

- bit 运算

代码

- 语言支持：JS，C++，Python

JavaScript Code：

```
/**
 * @param {number} n - a positive integer
 * @return {number} - a positive integer
 */
var reverseBits = function (n) {
  let res = 0;
  for (let i = 0; i < 32; i++) {
    res = (res << 1) + (n & 1);
    n = n >>> 1;
  }

  return res >>> 0;
};
```

C++ Code：

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        auto ret = 0;
        for (auto i = 0; i < 32; ++i) {
            ret = (ret << 1) + (n & 1);
            n >>= 1;
        }
        return ret;
    }
};
```

Python Code:

```
class Solution:
    # @param n, an integer
    # @return an integer
    def reverseBits(self, n):
        result = 0
        for i in range(32):
            result = (result << 1) + (n & 1)
            n >>= 1
        return result
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

拓展

不使用迭代也可以完成相同的操作：

1. 两两相邻的 1 位对调
2. 两两相邻的 2 位对调
3. 两两相邻的 4 位对调
4. 两两相邻的 8 位对调
5. 两两相邻的 16 位对调

C++代码如下：

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        auto ret = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555
        ret = ((ret & 0xcccccccc) >> 2) | ((ret & 0x3333333
        ret = ((ret & 0xf0f0f0f0) >> 4) | ((ret & 0x0f0f0f0
        ret = ((ret & 0xff00ff00) >> 8) | ((ret & 0x00ff00f
        return ((ret & 0xffff0000) >> 16) | ((ret & 0x0000f
    }
};
```

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(191. 位 1 的个数)

https://leetcode-cn.com/problems/number-of-1-bits/

题目描述

```
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’



示例 1：

输入：00000000000000000000000000001011
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三
示例 2：

输入：00000000000000000000000010000000
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一
示例 3：

输入：11111111111111111111111111111101
输出：31
解释：输入的二进制串 11111111111111111111111111111101 中，共有


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
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


进阶：
如果多次调用这个函数，你将如何优化你的算法？
```

前置知识

- 位运算

公司

- 阿里

- 腾讯
- 百度
- 字节
- apple
- microsoft

思路

这个题目的大意是： 给定一个无符号的整数， 返回其用二进制表式的时候的 1 的个数。

这里用一个 trick， 可以轻松求出。 就是 `n & (n - 1)` 可以 `消除` n 最后的一个 1 的原理。

> 为什么能消除最后一个 1， 其实也比较简单，大家自己想一下

这样我们可以不断进行 `n = n & (n - 1)` 直到 n === 0 ， 说明没有一个 1 了。 这个时候 `我们消除了多少1变成一个1都没有了， 就说明n有多少个1了` 。

关键点解析

1. `n & (n - 1)` 可以 `消除` n 最后的一个 1 的原理 简化操作
2. bit 运算

代码

语言支持：JS, C++，Python

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=191 lang=javascript
 *
 */
/**
 * @param {number} n - a positive integer
 * @return {number}
 */
var hammingWeight = function (n) {
  let count = 0;
  while (n !== 0) {
    n = n & (n - 1);
    count++;
  }

  return count;
};
```

C++ code:

```cpp
class Solution {
public:
    int hammingWeight(uint32_t v) {
        auto count = 0;
        while (v != 0) {
            v &= (v - 1);
            ++count;
        }
        return count;
    }
};
```

Python Code:

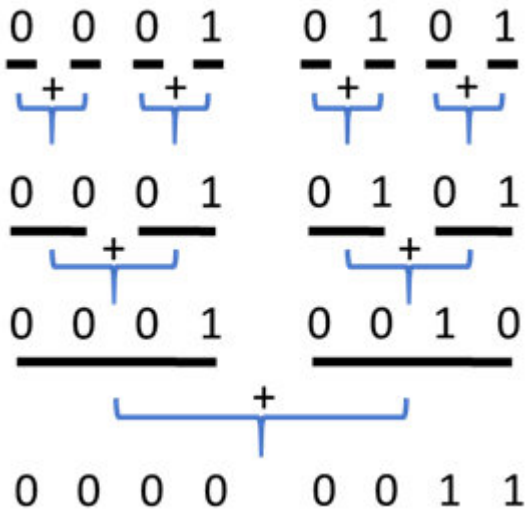
```python
class Solution(object):
    def hammingWeight(self, n):
        """
        :type n: int
        :rtype: int
        """
        count = 0
        while n:
            n &= n - 1
            count += 1
        return count
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(N)$

扩展

可以使用位操作来达到目的。例如 8 位的整数 21:



C++ Code：

```
const uint32_t ODD_BIT_MASK = 0xAAAAAAAA;
const uint32_t EVEN_BIT_MASK = 0x55555555;
const uint32_t ODD_2BIT_MASK = 0xCCCCCCCC;
const uint32_t EVEN_2BIT_MASK = 0x33333333;
const uint32_t ODD_4BIT_MASK = 0xF0F0F0F0;
const uint32_t EVEN_4BIT_MASK = 0x0F0F0F0F;
const uint32_t ODD_8BIT_MASK = 0xFF00FF00;
const uint32_t EVEN_8BIT_MASK = 0x00FF00FF;
const uint32_t ODD_16BIT_MASK = 0xFFFF0000;
const uint32_t EVEN_16BIT_MASK = 0x0000FFFF;

class Solution {
public:

    int hammingWeight(uint32_t v) {
        v = (v & EVEN_BIT_MASK) + ((v & ODD_BIT_MASK) >> 1
        v = (v & EVEN_2BIT_MASK) + ((v & ODD_2BIT_MASK) >>
        v = (v & EVEN_4BIT_MASK) + ((v & ODD_4BIT_MASK) >>
        v = (v & EVEN_8BIT_MASK) + ((v & ODD_8BIT_MASK) >>
        return (v & EVEN_16BIT_MASK) + ((v & ODD_16BIT_MASK
    }
};
```

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(198. 打家劫舍)

https://leetcode-cn.com/problems/house-robber/

入选理由

1. 爬楼梯换皮题给大家出一个，后面再列举几个大家有时间自己做做

题目描述

```
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

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



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
     偷窃到的最高金额 = 2 + 9 + 1 = 12 。


提示:

0 <= nums.length <= 100
0 <= nums[i] <= 400
```

前置知识

- 动态规划

公司

- 阿里

- 腾讯
- 百度
- 字节
- airbnb
- linkedin

思路

这是一道非常典型且简单的动态规划问题，但是在这里我希望通过这个例子， 让大家对动态规划问题有一点认识。

为什么别人的动态规划可以那么写，为什么没有用 dp 数组就搞定了。 比如别人的爬楼梯问题怎么就用 fibnacci 搞定了？为什么？在这里我们就来看下。

思路还是和其他简单的动态规划问题一样，我们本质上在解决 `对于第[i]个房子，我们抢还是不抢。` 的问题。

判断的标准就是总价值哪个更大， 那么对于抢的话 `就是当前的房子可以抢的价值 + dp[i - 2]`

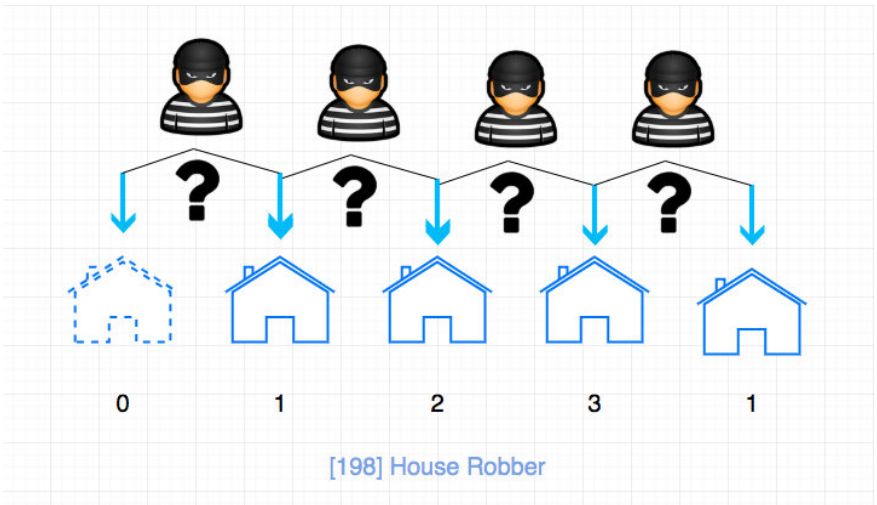
> i - 1 不能抢，否则会触发警铃

如果不抢的话，就是 `dp[i - 1]` .

> 这里的 dp 其实就是 `子问题` .

状态转移方程也不难写 `dp[i] = Math.max(dp[i - 2] + nums[i - 2], dp[i - 1]);` （注：这里为了方便计算，令 `dp[0]` 和 `dp[1]` 都等于 0，所以 `dp[i]` 对应的是 `nums[i - 2]` ）

上述过程用图来表示的话，是这样的：



我们仔细观察的话，其实我们只需要保证前一个 dp[i - 1] 和 dp[i - 2] 两个变量就好了， 比如我们计算到 i = 6 的时候，即需要计算 dp[6]的时候，我们需要 dp[5], dp[4]，但是我们 不需要 dp[3], dp[2] ...

因此代码可以简化为：

```
let a = 0;
let b = 0;

for (let i = 0; i < nums.length; i++) {
  const temp = b;
  b = Math.max(a + nums[i], b);
  a = temp;
}

return b;
```

如上的代码，我们可以将空间复杂度进行优化，从 O(n)降低到 O(1), 类似的优化在 DP 问题中不在少数。

> 动态规划问题是递归问题查表，避免重复计算，从而节省时间。 如果我们对问题加以分析和抽象，有可能对空间上进一步优化

关键点解析

代码

- 语言支持：JS，C++，Python

JavaScript Code：

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var rob = function (nums) {
  // Tag: DP
  const dp = [];
  dp[0] = 0;
  dp[1] = 0;

  for (let i = 2; i < nums.length + 2; i++) {
    dp[i] = Math.max(dp[i - 2] + nums[i - 2], dp[i - 1]);
  }

  return dp[nums.length + 1];
};
```

C++ Code：

> 与 JavaScript 代码略有差异，但状态迁移方程是一样的。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) return 0;
        auto sz = nums.size();
        if (sz == 1) return nums[0];
        auto prev = nums[0];
        auto cur = max(prev, nums[1]);
        for (auto i = 2; i < sz; ++i) {
            auto tmp = cur;
            cur = max(nums[i] + prev, cur);
            prev = tmp;
        }
        return cur;
    }
};
```

Python Code:

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        if not nums:
            return 0

        length = len(nums)
        if length == 1:
            return nums[0]
        else:
            prev = nums[0]
            cur = max(prev, nums[1])
            for i in range(2, length):
                cur, prev = max(prev + nums[i], cur), cur
            return cur
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 337.house-robber-iii

其他题目推荐

- Minimum-Sum-Subsequence

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(203. 移除链表元素)

https://leetcode-cn.com/problems/remove-linked-list-elements/

题目描述

```
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
```

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这个一个链表基本操作的题目，思路就不多说了。

虽然题目比较简单，但是实际面试出现的频率却不低， 因此大家一定要能够写出 bug free 的代码才可以。

链表的题目 90% 的 bug 都出现在：

1. 头尾节点的处理
2. 指针循环引用导致死循环

因此大家对这两个问题要保持 100% 的警惕。

关键点解析

- 链表的基本操作（删除指定节点）
- 虚拟节点 dummy 简化操作

> 其实设置 dummy 节点就是为了处理特殊位置（头节点），这这道题就是如果头节点是给定的需要删除的节点呢？ 为了保证代码逻辑的一致性，即不需要为头节点特殊定制逻辑，才采用的虚拟节点。

- 如果连续两个节点都是要删除的节点，这个情况容易被忽略。 eg:

```
// 只有下个节点不是要删除的节点才更新 current
if (!next || next.val !== val) {
  current = next;
}
```

代码

- 语言支持：JS，Python

Javascript Code:

```
/**
 * @param {ListNode} head
 * @param {number} val
 * @return {ListNode}
 */
var removeElements = function (head, val) {
  const dummy = {
    next: head,
  };
  let current = dummy;

  while (current && current.next) {
    let next = current.next;
    if (next.val === val) {
      current.next = next.next;
      next = next.next;
    }

    if (!next || next.val !== val) {
      current = next;
    }
  }

  return dummy.next;
};
```

Python Code:

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def removeElements(self, head: ListNode, val: int) -> L
        prev = ListNode(0)
        prev.next = head
        cur = prev
        while cur.next:
            if cur.next.val == val:
                cur.next = cur.next.next
            else:
                cur = cur.next
        return prev.next
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(206. 反转链表)

https://leetcode-cn.com/problems/reverse-linked-list/

题目描述

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?
```

前置知识

- 链表

公司

- 阿里
- 百度
- 腾讯
- adobe
- amazon
- apple
- bloomberg
- facebook
- microsoft
- snapchat
- twitter
- uber
- yahoo
- yelp
- zenefits

思路

这个就是常规操作了，使用一个变量记录前驱 pre，一个变量记录后继 next，不断更新 `current.next = pre` 就好了。

链表的题目 90% 的 bug 都出现在：

1. 头尾节点的处理
2. 指针循环引用导致死循环

因此大家对这两个问题要保持 100% 的警惕。

关键点解析

- 链表的基本操作（交换）
- 虚拟节点 dummy 简化操作
- 注意更新 current 和 pre 的位置， 否则有可能出现溢出

代码

语言支持：JS, C++, Python,Java

JavaScript Code：

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var reverseList = function (head) {
  if (!head || !head.next) return head;

  let cur = head;
  let pre = null;

  while (cur) {
    const next = cur.next;
    cur.next = pre;
    pre = cur;
    cur = next;
  }

  return pre;
};
```

C++ Code：

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* cur = head;
        ListNode* next = NULL;
        while (cur != NULL) {
            next = cur->next;
            cur->next = prev;
            prev = cur;
            cur = next;
        }
        return prev;
    }
};
```

Python Code:

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if not head: return None
        prev = None
        cur = head
        while cur:
            cur.next, prev, cur = prev, cur, cur.next
        return prev
```

Java Code:

```
/**
 * 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, cur = head;

        while (cur != null) {
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }

        return pre;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

拓展

通过单链表的定义可以得知，单链表也是递归结构，因此，也可以使用递归的方式来进行 reverse 操作。

> 由于单链表是线性的，使用递归方式将导致栈的使用也是线性的，当链表长度达到一定程度时，递归会导致爆栈，因此，现实中并不推荐使用递归方式来操作链表。

1. 除第一个节点外，递归将链表 reverse
2. 将第一个节点添加到已 reverse 的链表之后

> 这里需要注意的是，每次需要保存已经 reverse 的链表的头节点和尾节点

C++实现

```cpp
// 普通递归
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* tail = nullptr;
        return reverseRecursive(head, tail);
    }

    ListNode* reverseRecursive(ListNode *head, ListNode *&t
        if (head == nullptr) {
            tail = nullptr;
            return head;
        }
        if (head->next == nullptr) {
            tail = head;
            return head;
        }
        auto h = reverseRecursive(head->next, tail);
        if (tail != nullptr) {
            tail->next = head;
            tail = head;
            head->next = nullptr;
        }
        return h;
    }
};

//  (类似) 尾递归
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if (head == nullptr) return head;
        return reverseRecursive(nullptr, head, head->next);
    }

    ListNode* reverseRecursive(ListNode *prev, ListNode *he
    {
        if (next == nullptr) return head;
        auto n = next->next;
        next->next = head;
        head->next = prev;
        return reverseRecursive(head, next, n);
    }
};
```

JavaScript 实现

```
var reverseList = function (head) {
  // 递归结束条件
  if (head === null || head.next === null) {
    return head;
  }

  // 递归反转 子链表
  let newReverseList = reverseList(head.next);
  // 获取原来链表的第 2 个节点 newReverseListTail
  let newReverseListTail = head.next;
  // 调整原来头结点和第 2 个节点的指向
  newReverseListTail.next = head;
  head.next = null;

  // 将调整后的链表返回
  return newReverseList;
};
```

Python 实现

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if not head or not head.next: return head
        ans = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关题目

- 92.reverse-linked-list-ii
- 25.reverse-nodes-in-k-groups

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(219. 存在重复元素 II)

https://leetcode-cn.com/problems/contains-duplicate-ii/

题目描述

```
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



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
```

前置知识

- hashmap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

用一个 hashmap 存储已经访问过的数字，每次访问都查看 hashmap 中是否有这个元素，有的话拿出索引进行比对，是否满足条件（相隔不大于 k），如果满足返回 true 即可。

可以看出，这道题就是两数和的进阶版。大家可以将这两道题结合起来理解哦~

公司

- airbnb
- palantir

关键点解析

- 空间换时间

代码

- 语言支持：JS，Python，C++, Java

Javascript Code:

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {boolean}
 */
var containsNearbyDuplicate = function (nums, k) {
  const visited = {};
  for (let i = 0; i < nums.length; i++) {
    const num = nums[i];
    if (visited[num] !== undefined && i - visited[num] <= k
      return true;
    }
    visited[num] = i;
  }
  return false;
};
```

Python Code:

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: i
        d = {}
        for index, num in enumerate(nums):
            if num in d and index - d[num] <= k:
                return True
            d[num] = index
        return False
```

C++ Code：

```
class Solution {
public:
    bool containsNearbyDuplicate(vector<int>& nums, int k)
        auto m = unordered_map<int, int>();
        for (int i = 0; i < nums.size(); ++i) {
            auto iter = m.find(nums[i]);
            if (iter != m.end()) {
                if (i - m[nums[i]] <= k) {
                    return true;
                }
            }
            m[nums[i]] = i;
        }
        return false;
    }
};
```

Java Code:

```
class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int
        Map<Integer, Integer> map = new HashMap<>();
        for(int i=0;i<nums.length;i++)
        {
            if(map.get(nums[i]) != null && (i-map.get(nums
            {
                return true;
            }
            map.put(nums[i], i);
        }
        return false;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(226. 翻转二叉树)

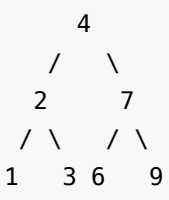
https://leetcode-cn.com/problems/invert-binary-tree/

题目描述

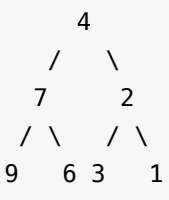
翻转一棵二叉树。

示例：

输入：



输出：



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

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个经典的面试问题，难度不大，大家可以用它练习一下递归和迭代。

算法：

遍历树（随便怎么遍历），然后将左右子树交换位置。

关键点解析

- 递归简化操作
- 如果树很高，建议使用栈来代替递归
- 这道题目对顺序没要求的，因此队列数组操作都是一样的，无任何区别

代码

- 语言支持：JS，Python，C++

Javascript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
var invertTree = function (root) {
  if (!root) return root;
  // 递归
  //   const left = root.left;
  //   const right = root.right;
  //   root.right = invertTree(left);
  //   root.left = invertTree(right);
  // 我们用stack来模拟递归
  // 本质上递归是利用了执行栈，执行栈也是一种栈
  // 其实这里使用队列也是一样的，因为这里顺序不重要

  const stack = [root];
  let current = null;
  while ((current = stack.shift())) {
    const left = current.left;
    const right = current.right;
    current.right = left;
    current.left = right;
    if (left) {
      stack.push(left);
    }
    if (right) {
      stack.push(right);
    }
  }
  return root;
};
```

Python Code:

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        if not root:
            return None
        stack = [root]
        while stack:
            node = stack.pop(0)
            node.left, node.right = node.right, node.left
            if node.left:
                stack.append(node.left)
            if node.right:
                stack.append(node.right)
        return root
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return root;
        auto q = queue<TreeNode*>();
        q.push(root);
        while (!q.empty()) {
            auto n = q.front(); q.pop();
            swap(n->left, n->right);
            if (n->left != nullptr) {
                q.push(n->left);
            }
            if (n->right != nullptr) {
                q.push(n->right);
            }
        }
        return root;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(232. 用栈实现队列)

https://leetcode-cn.com/problems/implement-queue-using-stacks/

题目描述

```
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
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
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
```

前置知识

- 栈
- 队列

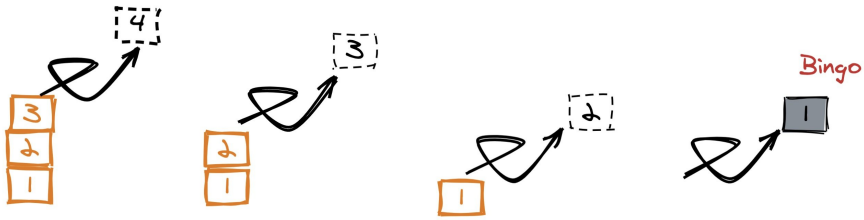
思路

题目要求用栈的原生操作来实现队列，也就是说需要用到 pop 和 push 但是我们知道 pop 和 push 都是在栈顶的操作，而队列的 enque 和 deque 则是在队列的两端的操作，这么一看一个 stack 好像不太能完成。

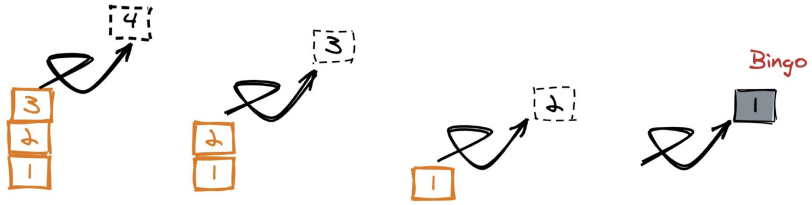
我们来分析一下过程。

假如向栈中分别 push 四个数字 `1, 2, 3, 4`，那么此时栈的情况应该是：

如果此时按照题目要求 pop 或者 peek 的话， 应该是返回 1 才对，而 1 在栈底我们无法直接操作。如果想要返回 1，我们首先要将 2，3，4 分别出栈才行。

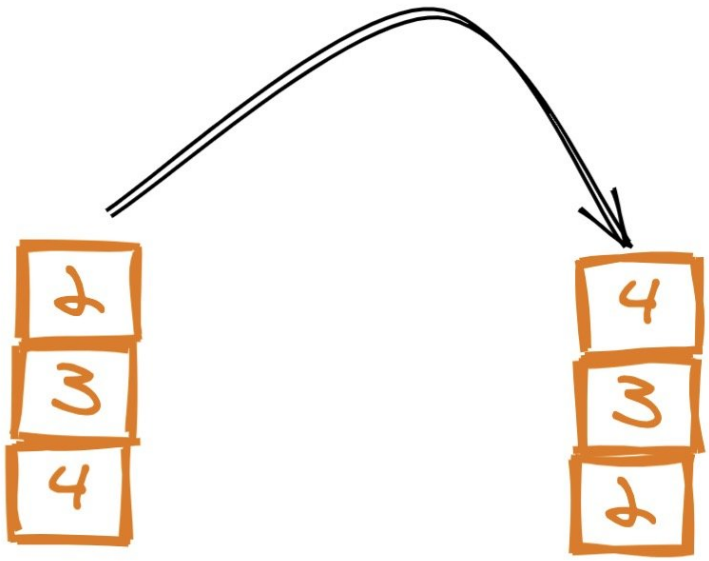


然而，如果我们这么做，1 虽然是正常返回了，但是 2，3，4 不就永远消失了么？ 一种简答方法就是，将 2，3，4 **存** 起来。而题目又说了，只能使用栈这种数据结构，那么我们考虑使用一个额外的栈来存放弹出的 2，3，4。

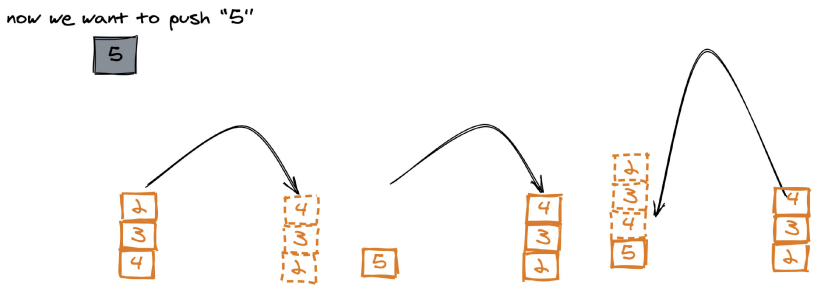


（pop 出来不扔掉，而是存起来）

整个过程类似这样：



比如，这个时候，我们想 push 一个 5，那么大概就是这样的：



然而这一过程，我们也可以发生在 push 阶段。

总之，就是我们需要在 push 或者 pop 的时候，将数组在两个栈之间倒腾一次。

关键点

- 在 push 的时候利用辅助栈(双栈)

代码

- 语言支持：JS, Python, Java

Javascript Code:

```javascript
/*
 * @lc app=leetcode id=232 lang=javascript
 *
 * [232] Implement Queue using Stacks
 */
/**
 * Initialize your data structure here.
 */
var MyQueue = function () {
  // tag: queue stack array
  this.stack = [];
  this.helperStack = [];
};

/**
 * Push element x to the back of queue.
 * @param {number} x
 * @return {void}
 */
MyQueue.prototype.push = function (x) {
  let cur = null;
  while ((cur = this.stack.pop())) {
    this.helperStack.push(cur);
  }
  this.helperStack.push(x);

  while ((cur = this.helperStack.pop())) {
    this.stack.push(cur);
  }
};

/**
 * Removes the element from in front of queue and returns t
 * @return {number}
 */
MyQueue.prototype.pop = function () {
  return this.stack.pop();
};

/**
 * Get the front element.
 * @return {number}
 */
MyQueue.prototype.peek = function () {
  return this.stack[this.stack.length - 1];
};

/**
```

```
 * Returns whether the queue is empty.
 * @return {boolean}
 */
MyQueue.prototype.empty = function () {
  return this.stack.length === 0;
};

/**
 * Your MyQueue object will be instantiated and called as s
 * var obj = new MyQueue()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.peek()
 * var param_4 = obj.empty()
 */
```

Python Code:

```python
class MyQueue:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.stack = []
        self.help_stack = []

    def push(self, x: int) -> None:
        """
        Push element x to the back of queue.
        """
        while self.stack:
            self.help_stack.append(self.stack.pop())
        self.help_stack.append(x)
        while self.help_stack:
            self.stack.append(self.help_stack.pop())

    def pop(self) -> int:
        """
        Removes the element from in front of queue and retu
        """
        return self.stack.pop()

    def peek(self) -> int:
        """
        Get the front element.
        """
        return self.stack[-1]

    def empty(self) -> bool:
        """
        Returns whether the queue is empty.
        """
        return not bool(self.stack)


# Your MyQueue object will be instantiated and called as su
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()
```

Java Code

```
class MyQueue {
    Stack<Integer> pushStack = new Stack<> ();
    Stack<Integer> popStack = new Stack<> ();

    /** Initialize your data structure here. */
    public MyQueue() {

    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        while (!popStack.isEmpty()) {
            pushStack.push(popStack.pop());
        }
        pushStack.push(x);
    }

    /** Removes the element from in front of queue and retu
    public int pop() {
        while (!pushStack.isEmpty()) {
            popStack.push(pushStack.pop());
        }
        return popStack.pop();
    }

    /** Get the front element. */
    public int peek() {
        while (!pushStack.isEmpty()) {
            popStack.push(pushStack.pop());
        }
        return popStack.peek();
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return pushStack.isEmpty() && popStack.isEmpty();
    }
}

/**
 * Your MyQueue object will be instantiated and called as s
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.empty();
 */
```

复杂度分析

- 时间复杂度：`O(N)`，其中 N 为 栈中元素个数，因为每次我们都要倒腾一次。
- 空间复杂度：`O(N)`，其中 N 为 栈中元素个数，多使用了一个辅助栈，这个辅助栈的大小和原栈的大小一样。

扩展

- 类似的题目有用队列实现栈，思路是完全一样的，大家有兴趣可以试一下。
- 栈混洗也是借助另外一个栈来完成的，从这点来看，两者有相似之处。

延伸阅读

实际上现实中也有使用两个栈来实现队列的情况，那么为什么我们要用两个 stack 来实现一个 queue?

其实使用两个栈来替代一个队列的实现是为了在多进程中分开对同一个队列对读写操作。一个栈是用来读的，另一个是用来写的。当且仅当读栈满时或者写栈为空时，读写操作才会发生冲突。

当只有一个线程对栈进行读写操作的时候，总有一个栈是空的。在多线程应用中，如果我们只有一个队列，为了线程安全，我们在读或者写队列的时候都需要锁住整个队列。而在两个栈的实现中，只要写入栈不为空，那么 `push` 操作的锁就不会影响到 `pop` 。

- reference
- further reading

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

题目地址(263. 丑数)

https://leetcode-cn.com/problems/ugly-number/

题目描述

```
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 是丑数。
输入不会超过 32 位有符号整数的范围: [−231,  231 − 1]。
```

前置知识

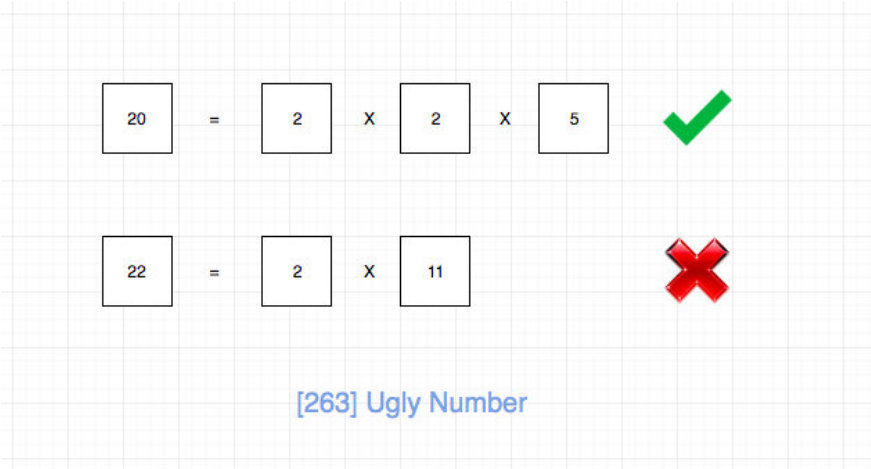
- 数学
- 因数分解

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

题目要求给定一个数字，判断是否为“丑陋数”(ugly number), 丑陋数是指只包含质因子 2, 3, 5 的正整数。



根据定义，我们将给定数字除以 2、3、5(顺序无所谓)，直到无法整除。如果得到 1，说明是所有因子都是 2 或 3 或 5，如果不是 1，则不是丑陋数。

这就好像我们判断一个数字是否为 n(n 为大于 1 的正整数)的幂次方一样，我们只需要 不断除以 n，直到无法整除，如果得到 1，那么就是 n 的幂次方。 这道题的不同在于 它不再是某一个数字的幂次方，而是三个数字（2，3，5），不过解题思路还是一样的。

转化为代码可以是：

```
while (num % 2 === 0) num = num / 2;
while (num % 3 === 0) num = num / 3;
while (num % 5 === 0) num = num / 5;

return num === 1;
```

> 我下方给出的代码是用了递归实现，只是给大家看下不同的写法而已。

关键点

- 数论
- 因数分解

代码

- 语言支持：JS, C++, Java, Python

Javascript Code:

```
/*
 * @lc app=leetcode id=263 lang=javascript
 *
 * [263] Ugly Number
 */
/**
 * @param {number} num
 * @return {boolean}
 */
var isUgly = function (num) {
  // TAG: 数论
  if (num <= 0) return false;
  if (num === 1) return true;

  const list = [2, 3, 5];

  if (list.includes(num)) return true;

  for (let i of list) {
    if (num % i === 0) return isUgly(Math.floor(num / i));
  }
  return false;
};
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(logN)$

C++ Code:

```
class Solution {
public:
    bool isUgly(int num) {
        int ugly[] = {2,3,5};
        for(int u : ugly)
        {
            while(num%u==0 && num%u < num)
            {
                num/=u;
            }
        }
        return num == 1;
    }
};
```

Java Code:

```java
class Solution {
    public boolean isUgly(int num) {
        int [] ugly = {2,3,5};
        for(int u : ugly)
        {
            while(num%u==0 && num%u < num)
            {
                num/=u;
            }
        }
        return num == 1;
    }
}
```

Python Code:

```python
# 非递归写法
class Solution:
    def isUgly(self, num: int) -> bool:
        if num <= 0:
            return False
        for i in (2, 3, 5):
            while num % i == 0:
                num /= i
        return num == 1
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(283. 移动零)

https://leetcode-cn.com/problems/move-zeroes/

题目描述

```
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

示例:

输入: [0,1,0,3,12]
输出: [1,3,12,0,0]
说明:

必须在原数组上操作，不能拷贝额外的数组。
尽量减少操作次数。
```

前置知识

- 数组
- 双指针

公司

- 阿里
- 腾讯
- 百度
- 字节
- bloomberg
- facebook

思路

如果题目没有要求 modify in-place 的话，我们可以先遍历一遍将包含 0 的和不包含 0 的存到两个数组，然后拼接两个数组即可。 但是题目要求 modify in-place， 也就是不需要借助额外的存储空间，刚才的方法空间复杂度是 O(n).

那么如果 modify in-place ， 空间复杂度降低为 1 呢?

其实可以使用**读写双指针**来做。具体来说使用一个慢指针表示写指针，快指针表示读指针。

具体来说：读指针不断往后移动。如果遇到非 0，则将读到的值写入写指针，触发写指针移动（其他情况写指针不动），读指针走到头算法结束。经过这样的处理，最终写指针的位置前面就是所有的非 0 数了， 最后将写指针后的 位置全部修改为 0 即可。

关键点解析

- 读写双指针

代码

- 语言支持：JS, C++, Java，Python

JavaScript Code:

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-p
 */
var moveZeroes = function (nums) {
  let index = 0;
  for (let i = 0; i < nums.length; i++) {
    const num = nums[i];
    if (num !== 0) {
      nums[index++] = num;
    }
  }

  for (let i = index; i < nums.length; i++) {
    nums[index++] = 0;
  }
};
```

C++ Code：

> 解题思想与上面 JavaScript 一致，做了少许代码优化（非性能优化，因为时间复杂度都是 O(n)）： 增加一个游标来记录下一个待处理的元素的位置，这样只需要写一次循环即可。

```cpp
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        vector<int>::size_type nonZero = 0;
        vector<int>::size_type next = 0;
        while (next < nums.size()) {
            if (nums[next] != 0) {
                // 使用 std::swap() 会带来 8ms 的性能损失
                // swap(nums[next], nums[nonZero]);
                auto tmp = nums[next];
                nums[next] = nums[nonZero];
                nums[nonZero] = tmp;
                ++nonZero;
            }
            ++next;
        }
    }
};
```

Java Code:

```java
class Solution {
    public void moveZeroes(int[] nums) {
        // 双指针
        int i = 0;
        for(int j=0; j<nums.length; j++)
        {
            // 不为0，前移
            if(nums[j] != 0)
            {
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
                i++;
            }
        }
    }
}
```

Python Code:

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instea
        """
        slow = fast = 0
        while fast < len(nums):
            if nums[fast] != 0:
                nums[fast], nums[slow] = nums[slow], nums[
                slow += 1
            fast += 1
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(342. 4 的幂)

https://leetcode-cn.com/problems/power-of-four/

题目描述

```
给定一个整数（32 位有符号整数），请编写一个函数来判断它是否是 4 的幂次

示例 1:

输入: 16
输出: true
示例 2:

输入: 5
输出: false
进阶:
你能不使用循环或者递归来完成本题吗?
```

前置知识

- 数论

公司

- 百度
- twosigma

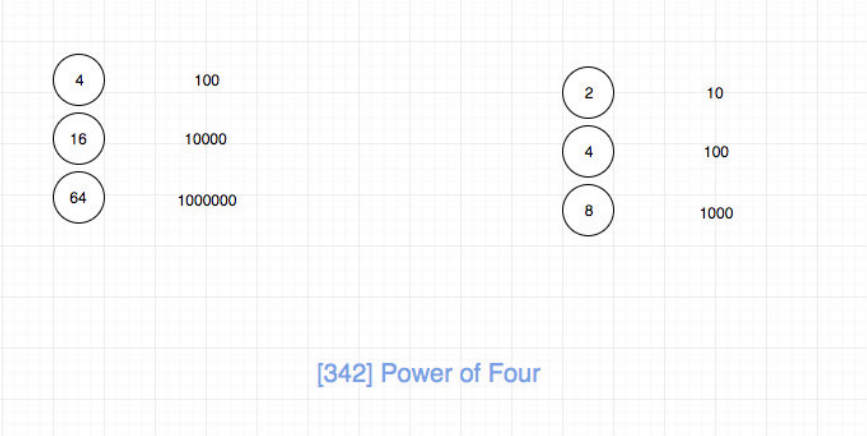
思路

符合直觉的做法是不停除以 4 直到不能整除，然后判断是否为 1 即可。
代码如下：

```
while (num && num % 4 == 0) {
  num /= 4;
}
return num == 1;
```

但是这道题目有一个 follow up: “你是否可以不使用循环/递归完成”。因此我们需要换种思路。

我们先来看下，4 的幂次方用 2 进制表示是什么样的.



发现规律： 4 的幂次方的二进制表示 1 的位置都是在奇数位（且不在最低位），其他位置都为 0

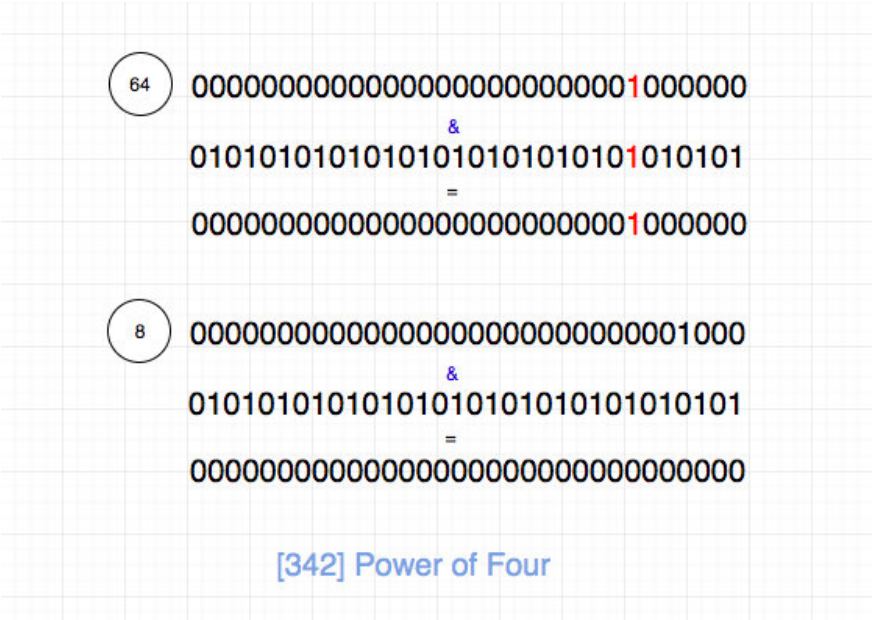
我们还可以发现： 2 的幂次方的特点是最低位之外，其他位置有且仅有一个 1（1 可以在任意位置）

我们进一步分析，如果一个数字是四的幂次方，那么只需要满足：

1. 是 2 的幂次方， 就能保证最低位之外，其他位置有且仅有一个 1
2. 这个 1 不在偶数位置，一定在奇数位置

对于第一点，如果保证一个数字是 2 的幂次方呢？ 显然不能不停除以 2，看结果是否等于 1，这样就循环了。 我们可以使用一个 trick， 如果一个数字 n 是 2 的幂次方，那么 n & (n - 1) 一定等于 0， 这个可以作为思考题，大家思考一下。

对于第二点，我们可以取一个特殊数字，这个特殊数字，奇数位置都是 1，偶数位置都是 0，然后和这个特殊数字 `求与` ， 如果等于本身，那么毫无疑问，这个 1 不再偶数位置，一定在奇数位置，因为如果在偶数位置， `求与` 的结果就是 0 了 题目要求 n 是 32 位有符号整形，那么我们的特殊数字就应该是 `01010101010101010101010101010101` (不用数了，一共 32 位)。



如上图，64 和这个特殊数字求与，得到的是本身。 8 是 2 的次方，但是不是 4 的次方，我们求与结果就是 0 了。

为了体现自己的逼格，我们可以使用计算器，来找一个逼格比较高的数字，这里我选了十六进制，结果是 `0x55555555` 。

代码见下方代码区。

说实话，这种做法不容易想到，其实还有一种方法。 如果一个数字是 4 的幂次方，那么只需要满足：

1. 是二的倍数
2. 减去 1 是三的倍数

代码如下：

```
return num > 0 && (num & (num - 1)) === 0 && (num - 1) % 3
```

关键点

- 数论
- 2 的幂次方特点（数学性质以及二进制表示）
- 4 的幂次方特点（数学性质以及二进制表示）

代码

语言支持：JS, Python

JavaScript Code：

```
/*
 * @lc app=leetcode id=342 lang=javascript
 *
 * [342] Power of Four
 */
/**
 * @param {number} num
 * @return {boolean}
 */
var isPowerOfFour = function (num) {
  // tag: 数论

  if (num === 1) return true;
  if (num < 4) return false;

  if ((num & (num - 1)) !== 0) return false;

  return (num & 0x55555555) === num;
};
```

Python Code:

```python
class Solution:
    def isPowerOfFour(self, num: int) -> bool:
        if num == 1:
            return True
        elif num < 4:
            return False
        else:
            if not num & (num-1) == 0:
                return False
            else:
                return num & 0x55555555 == num

    # 另一种解法：将数字转化为二进制表示的字符串，利用字符串的相关操作
    def isPowerOfFour(self, num: int) -> bool:
        binary_num = bin(num)[2:]
        return binary_num.strip('0') == '1' and len(binary_
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(349. 两个数组的交集)

https://leetcode-cn.com/problems/intersection-of-two-arrays/

题目描述

```
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。
```

前置知识

- hashtable

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

先遍历第一个数组，将其存到 hashtable 中，然后遍历第二个数组，如果在 hashtable 中存在就 push 到 ret，然后清空 hashtable，最后返回 ret 即可。

关键点解析

- 空间换时间

代码

代码支持：JS, Python

Javascript Code:

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number[]}
 */
var intersection = function (nums1, nums2) {
  const visited = {};
  const ret = [];
  for (let i = 0; i < nums1.length; i++) {
    const num = nums1[i];

    visited[num] = num;
  }

  for (let i = 0; i < nums2.length; i++) {
    const num = nums2[i];

    if (visited[num] !== undefined) {
      ret.push(num);
      visited[num] = undefined;
    }
  }

  return ret;
};
```

Python Code:

```python
class Solution:
    def intersection(self, nums1: List[int], nums2: List[i
        visited, result = {}, []
        for num in nums1:
            visited[num] = num
        for num in nums2:
            if num in visited:
                result.append(num)
                visited.pop(num)
        return result

    # 另一种解法：利用 Python 中的集合进行计算
    def intersection(self, nums1: List[int], nums2: List[i
        return set(nums1) & set(nums2)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(371. 两整数之和)

https://leetcode-cn.com/problems/sum-of-two-integers/

题目描述

```
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
```

前置知识

- 位运算

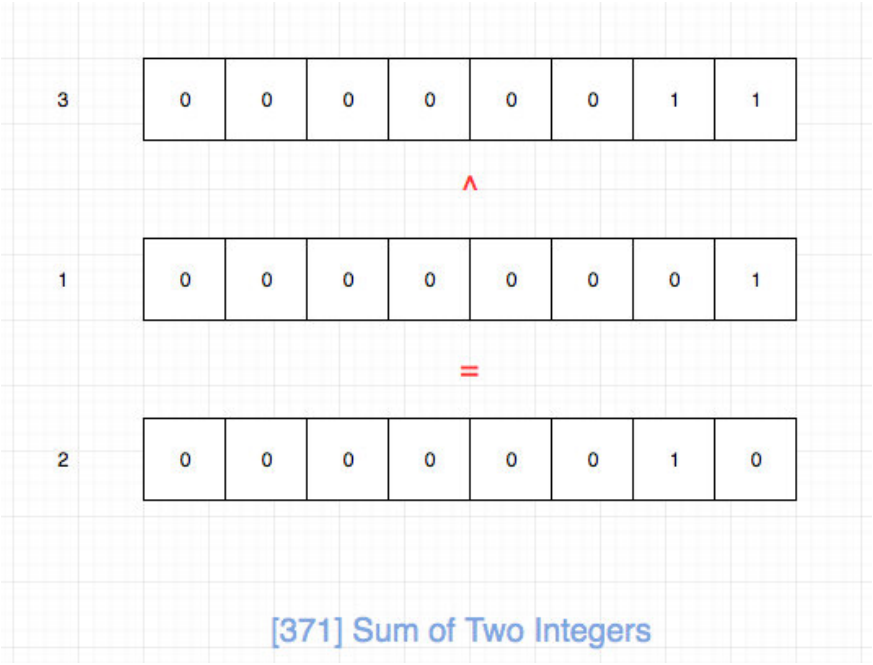
公司

- 阿里
- 腾讯
- 百度
- 字节

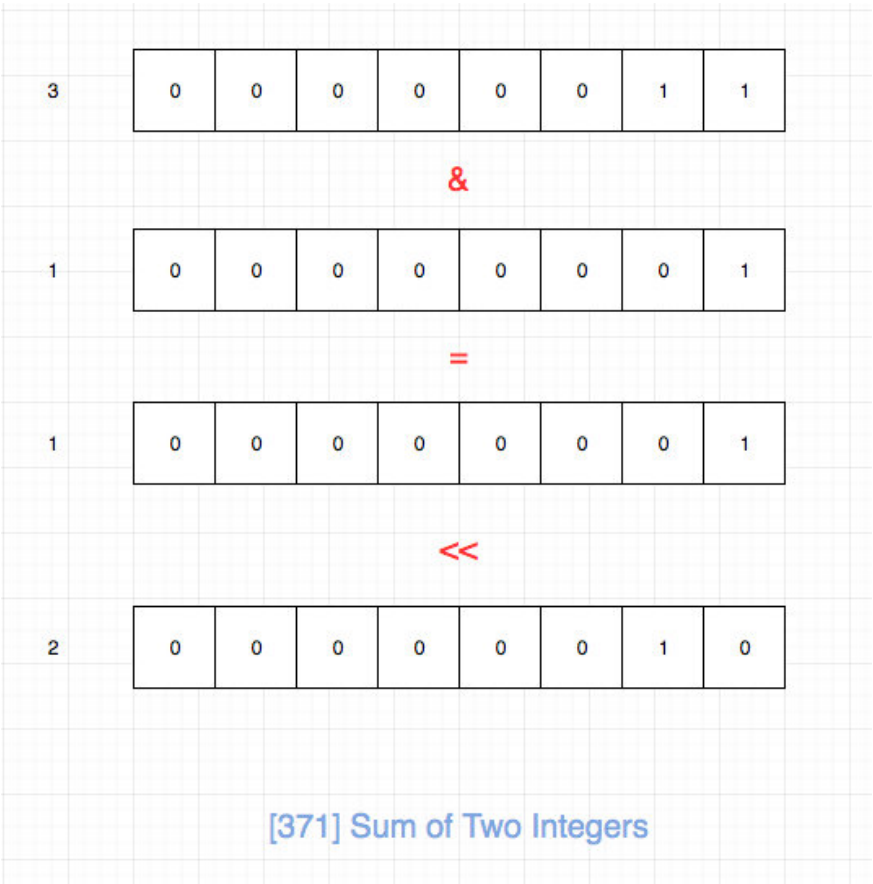
思路

不能使用加减法来求加法。 我们只能朝着位元算的角度来思考了。

由于 `异或` 是 `相同则位0，不同则位1` ，因此我们可以把异或看成是一种不进位的加减法。



由于 `与` 是 `全部位1则位1，否则位0` ，因此我们可以求与之后左移一位来表示进位。



然后我们对上述两个元算结果递归求解即可。 递归的结束条件就是其中一个为 0，我们直接返回另一个。

关键点解析

- 位运算
- 异或是一种不进位的加减法
- 求与之后左移一位来可以表示进位

代码

代码支持：JS，C++，Java，Python
Javascript Code:

```
/*
 * @lc app=leetcode id=371 lang=javascript
 *
 * [371] Sum of Two Integers
 */
/**
 * @param {number} a
 * @param {number} b
 * @return {number}
 */
var getSum = function (a, b) {
  if (a === 0) return b;

  if (b === 0) return a;

  return getSum(a ^ b, (a & b) << 1);
};
```

C++ Code:

```
class Solution {
public:
    int getSum(int a, int b) {
        if(a==0) return b;
        if(b==0) return a;

        while(b!=0)
        {
            // 防止 AddressSanitizer 对有符号左移的溢出保护处理
            auto carry = ((unsigned int ) (a & b))<<1;
            // 计算无进位的结果
            a = a^b;
            //将存在进位的位置置1
            b =carry;
        }
        return a;
    }
};
```

Java Code:

```
class Solution {
    public int getSum(int a, int b) {
        if(a==0) return b;
        if(b==0) return a;

        while(b!=0)
        {
            int carry = a&b;
            // 计算无进位的结果
            a = a^b;
            //将存在进位的位置置1
            b =carry<<1;
        }
        return a;
    }
}
```

Python Code:

```python
# python整数类型为Unifying Long Integers, 即无限长整数类型.
# 模拟 32bit 有符号整型加法
class Solution:
    def getSum(self, a: int, b: int) -> int:
        a &= 0xFFFFFFFF
        b &= 0xFFFFFFFF
        while b:
            carry = a & b
            a ^= b
            b = ((carry) << 1) & 0xFFFFFFFF
            # print((a, b))
        return a if a < 0x80000000 else ~(a^0xFFFFFFFF)
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

> 由于题目数据规模不会变化，因此其实复杂度分析是没有意义的。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(401. 二进制手表)

https://leetcode-cn.com/problems/binary-watch/

题目描述

二进制手表顶部有 4 个 LED 代表 小时（0-11），底部的 6 个 LED 代表

每个 LED 代表一个 0 或 1，最低位在右侧。

例如，上面的二进制手表读取 “3:25”。

给定一个非负整数 n 代表当前 LED 亮着的数量，返回所有可能的时间。

示例：

输入: n = 1
返回: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:0

提示：

输出的顺序没有要求。
小时不会以零开头，比如 “01:00” 是不允许的，应为 “1:00”。
分钟必须由两位数组成，可能会以零开头，比如 “10:2” 是无效的，应为 “10
超过表示范围（小时 0-11，分钟 0-59）的数据将会被舍弃，也就是说不会出

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/binary-watch
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

前置知识

- 笛卡尔积
- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

一看题目就是一个笛卡尔积问题。

即给你一个数字 num， 我可以将其分成两部分。其中一部分（不妨设为 a）给小时，另一部分给分（就是 num - a）。 最终的结果就是 a 能表示的所有小时的集合和 num - a所能表示的分的集合的笛卡尔积。

用代码表示就是：

```
# 枚举小时
for a in possible_number(i):
    # 小时确定了，分就是 num - i
    for b in possible_number(num - i, True):
        ans.add(str(a) + ":" + str(b).rjust(2, '0'))
```

枚举所有可能的 (a, num - a) 组合即可。

核心代码：

```
for i in range(min(4, num + 1)):
    for a in possible_number(i):
        for b in possible_number(num - i, True):
            ans.add(str(a) + ":" + str(b).rjust(2, '0'))
```

代码

```
class Solution:
    def readBinaryWatch(self, num: int) -> List[str]:
        def possible_number(count, minute=False):
            if count == 0: return [0]
            if minute:
                return filter(lambda a: a < 60, map(sum, co
            return filter(lambda a: a < 12, map(sum, combi
        ans = set()
        for i in range(min(4, num + 1)):
            for a in possible_number(i):
                for b in possible_number(num - i, True):
                    ans.add(str(a) + ":" + str(b).rjust(2,
        return list(ans)
```

进一步思考，实际上，我们要找的就是 a 和 b 相加等于 num，并且 a 和 b 就是二进制表示中 1 的个数。 因此可以将逻辑简化为：

```
class Solution:
    def readBinaryWatch(self, num: int) -> List[str]:
        return [str(a) + ":" + str(b).rjust(2, '0') for a i
```

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(437. 路径总和 III)

https://leetcode-cn.com/problems/path-sum-iii/

题目描述

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

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
```

前置知识

- hashmap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是要我们求解出任何一个节点出发到子孙节点的路径中和为指定值。 注意这里，不一定是从根节点出发，也不一定在叶子节点结束。

一种简单的思路就是直接递归解决，空间复杂度 O(n) 时间复杂度介于 O(nlogn) 和 O(n^2)， 具体代码：

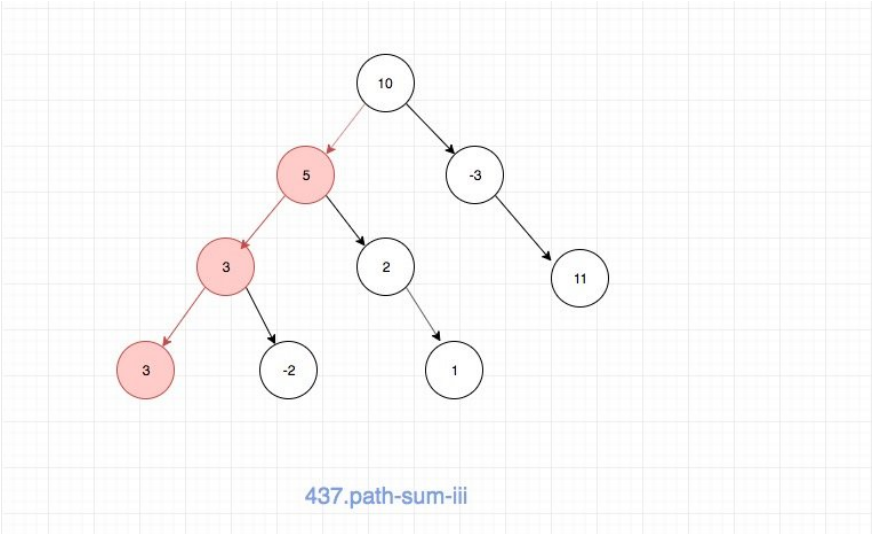
```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
// the number of the paths starting from self
function helper(root, sum) {
  if (root === null) return 0;
  const l = helper(root.left, sum - root.val);
  const r = helper(root.right, sum - root.val);

  return l + r + (root.val === sum ? 1 : 0);
}
/**
 * @param {TreeNode} root
 * @param {number} sum
 * @return {number}
 */
var pathSum = function (root, sum) {
  // 空间复杂度O(n) 时间复杂度介于O(nlogn) 和 O(n^2)
  // tag: dfs tree
  if (root === null) return 0;
  // the number of the paths starting from self
  const self = helper(root, sum);
  // we don't know the answer, so we just pass it down
  const l = pathSum(root.left, sum);
  // we don't know the answer, so we just pass it down
  const r = pathSum(root.right, sum);

  return self + l + r;
};
```

但是还有一种空间复杂度更加优秀的算法，利用 hashmap 来避免重复计算，时间复杂度和空间复杂度都是 O(n)。 这种思路是 `subarray-sum-equals-k` 的升级版本，如果那道题目你可以 O(n)解决，这道题目难度就不会很大， 只是将数组换成了二叉树。关于具体的思路可以看这道题目

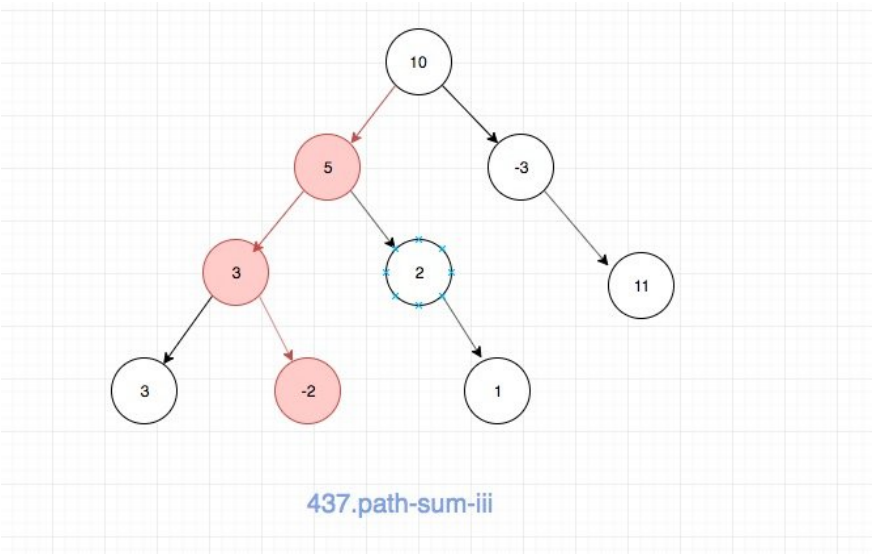
这里有一个不一样的地方，这里我说明一下，就是为什么要有 `hashmap[acc] = hashmap[acc] - 1;` ， 原因很简单，就是我们 DFS 的时候，从底部往上回溯的时候，map 的值应该也回溯。如果你对回溯法比较熟悉的话， 应该很容易理解，如果不熟悉可以参考这道题目， 这道题目就是通过 `tempList.pop()` 来完成的。

另外我画了一个图，相信看完你就明白了。

当我们执行到底部的时候：



接着往上回溯：



很容易看出，我们的 hashmap 不应该有第一张图的那个记录了，因此需要减去。

具体实现见下方代码区。

关键点解析

- 通过 hashmap，以时间换空间
- 对于这种连续的元素求和问题，有一个共同的思路，可以参考这道题目

代码

- 语言支持：JS

```
/*
 * @lc app=leetcode id=437 lang=javascript
 *
 * [437] Path Sum III
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function helper(root, acc, target, hashmap) {
  // see also : https://leetcode.com/problems/subarray-sum-

  if (root === null) return 0;
  let count = 0;
  acc += root.val;
  if (acc === target) count++;
  if (hashmap[acc - target] !== void 0) {
    count += hashmap[acc - target];
  }
  if (hashmap[acc] === void 0) {
    hashmap[acc] = 1;
  } else {
    hashmap[acc] += 1;
  }
  const res =
    count +
    helper(root.left, acc, target, hashmap) +
    helper(root.right, acc, target, hashmap);

  // 这里要注意别忘记了
  hashmap[acc] = hashmap[acc] - 1;

  return res;
}

var pathSum = function (root, sum) {
  const hashmap = {};
  return helper(root, 0, sum, hashmap);
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(455. 分发饼干)

https://leetcode-cn.com/problems/assign-cookies/

题目描述

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

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.

前置知识

- 贪心算法
- 双指针

公司

- 阿里
- 腾讯
- 字节

思路

本题可用贪心求解。给一个孩子的饼干应当尽量小并且能满足孩子，大的留来满足胃口大的孩子。因为胃口小的孩子最容易得到满足，所以优先满足胃口小的孩子需求。按照从小到大的顺序使用饼干尝试是否可满足某个孩子。

算法：

- 将需求因子 g 和 s 分别从小到大进行排序
- 使用贪心思想，配合两个指针，每个饼干只尝试一次，成功则换下一个孩子来尝试，不成功则换下一个饼干🍪 来尝试。

关键点

- 先排序再贪心

代码

语言支持：JS

```
/**
 * @param {number[]} g
 * @param {number[]} s
 * @return {number}
 */
const findContentChildren = function (g, s) {
    g = g.sort((a, b) => a - b);
    s = s.sort((a, b) => a - b);
    let gi = 0; // 胃口值
    let sj = 0; // 饼干尺寸
    let res = 0;
    while (gi < g.length && sj < s.length) {
        // 当饼干 sj >= 胃口 gi 时，饼干满足胃口，更新满足的孩子数
        if (s[sj] >= g[gi]) {
            gi++;
            sj++;
            res++;
        } else {
            // 当饼干 sj < 胃口 gi 时，饼干不能满足胃口，需要换大
            sj++;
        }
    }
    return res;
};
```

复杂度分析

- 时间复杂度：由于使用了排序，因此时间复杂度为 O(NlogN)
- 空间复杂度：O(1)

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经37K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(504. 七进制数)

https://leetcode-cn.com/problems/base-7/

题目描述

```
给定一个整数，将其转化为7进制，并以字符串形式输出。

示例 1:

输入: 100
输出: "202"


示例 2:

输入: -7
输出: "-10"


注意: 输入范围是 [-1e7, 1e7] 。
```

前置知识

-

公司

- 暂无

思路

这道题很经典也很重要。 如果你把这道题搞懂了，那么所有的进制转化题目对你来说都不是问题。 另外有的题目虽然不是直接让你进制转化，不过使用进制转化却实实在在可以优化代码。

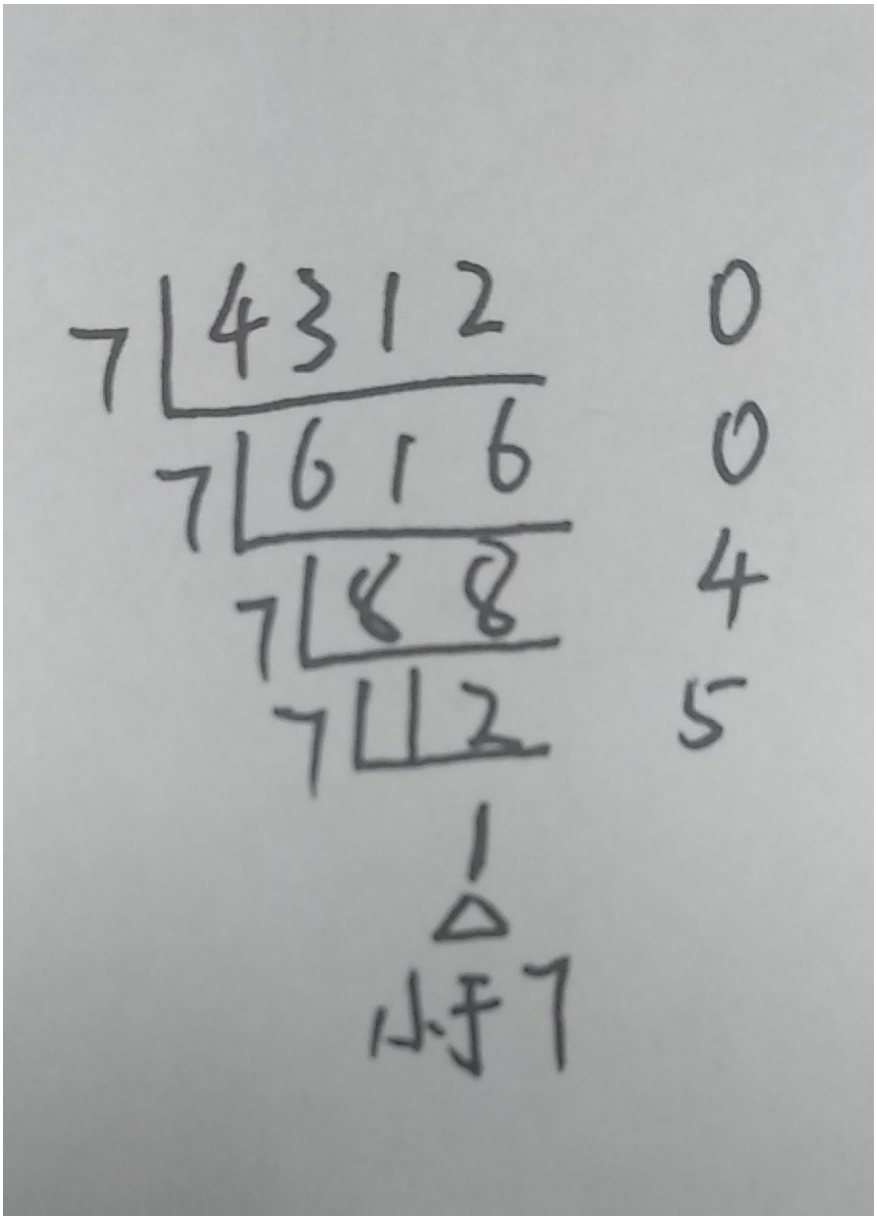
10 进制转化任意进制的思路都是**除 x 取余**，其中 x 为进制数，比如 2 进制就是 除 2 取余，7 进制就是除 7 取余。这个大家可能都知道，这里再带大家回顾一下。

比如一个数 4321 ，需要转化为 7 进制。那么可以：

- 先将 4321 除以 7，其中余数为 0 ， 除数为 616

- 继续将 616 采用同样的方法直到小于 7

将此过冲的余数**反序**就是答案了。图解：



（图片来自网络）

如图，4312 的 7 进制就是 15400。

关键点

- 除 x 取余，并逆序输出

代码

- 语言支持：Python3

Python3 Code:

递归：

```
class Solution:
    def convertToBase7(self, num: int) -> str:
        if num < 0:
            return "-" + self.convertToBase7(-num)
        if num < 7:
            return str(num)
        return self.convertToBase7(num // 7) + str(num % 7)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(h)$，其中 h 为递归栈的深度。

迭代：

```
class Solution:
    def convertToBase7(self, num: int) -> str:
        if num == 0:
            return 0
        ans = []
        is_negative = num < 0
        num = abs(num)
        while num > 0:
            num, remain = num // 7, num % 7
            ans.append(str(remain))

        return "-" + "".join(ans[::-1]) if is_negative else
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(575. 分糖果)

https://leetcode-cn.com/problems/distribute-candies/

题目描述

```
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

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
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
注意:

数组的长度为[2, 10,000]，并且确定为偶数。
数组中数字的大小在范围[-100,000, 100,000]内。
```

前置知识

- 数组

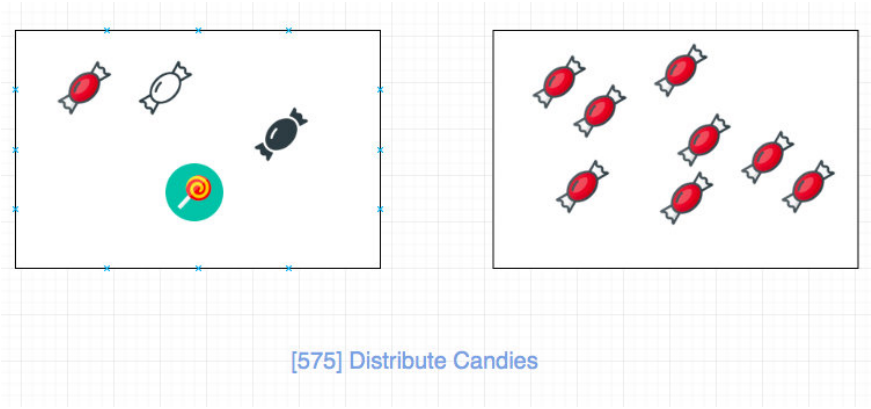
公司

- 阿里
- 字节

思路

由于糖果是偶数，并且我们只需要做到两个人糖果数量一样即可。

考虑两种情况:



- 如果糖果种类大于 n / 2（糖果种类数为 n），妹妹最多可以获得的糖果种类应该是 `n / 2` (因为妹妹只有 n / 2 个糖).
- 糖果种类数小于 n / 2, 妹妹能够得到的糖果种类可以是糖果的种类数（糖果种类本身就这么多）.

因此我们发现，妹妹能够获得的糖果种类的制约因素其实是糖果种类数。

关键点解析

- 这是一道逻辑题目，因此如果逻辑分析清楚了，代码是自然而然的

代码

- 语言支持：JS, Python

Javascript Code:

```
/*
 * @lc app=leetcode id=575 lang=javascript
 *
 * [575] Distribute Candies
 */
/**
 * @param {number[]} candies
 * @return {number}
 */
var distributeCandies = function (candies) {
  const count = new Set(candies);
  return Math.min(count.size, candies.length >> 1);
};
```

Python Code:

```
class Solution:
    def distributeCandies(self, candies: List[int]) -> int:
        return min(len(set(candies)), len(candies) >> 1)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(665. 非递减数列)

https://leetcode-cn.com/problems/non-decreasing-array/

题目描述

```
给你一个长度为 n 的整数数组，请你判断在 最多 改变 1 个元素的情况下，i

我们是这样定义一个非递减数列的： 对于数组中所有的 i (0 <= i <= n-2)



示例 1:

输入: nums = [4,2,3]
输出: true
解释: 你可以通过把第一个4变成1来使得它成为一个非递减数列。


示例 2:

输入: nums = [4,2,1]
输出: false
解释: 你不能在只改变一个元素的情况下将其变为非递减数列。




说明:

1 <= n <= 10 ^ 4
- 10 ^ 5 <= nums[i] <= 10 ^ 5
```

前置知识

- 数组
- 贪心

公司

- 暂无

思路

这道题简单就简单在它限定了**在 最多 改变 1 个元素的情况下**，如果不限定这个条件，让你求最少改变多少个数字，就有点难度了。

对于这道题来说，我们可以从左到右遍历数组 A， 如果 A[i-1] > A[i]，我们找到了一个递减对。遍历过程中的递减对个数大于 1，则可提前返回 False。

于是，大家可能写出如下代码：

```
class Solution:
    def checkPossibility(self, A: List[int]) -> bool:
        count = 0
        for i in range(1, len(A)):
            if A[i] < A[i - 1]:
                if count == 1: return False
                count += 1
        return True
```

上面代码是有问题的，问题在于类似 `[3,4,2,3]` 的测试用例会无法通过。问题在于递减对的计算方式有问题。

对于 `[3,4,2,3]` 来说，其递减对不仅仅有 (4,2)。其实应该还包括 (4,3)。 这提示我们在这个时候应该将 2 修改为不小于前一项的数，也就是 4，此时数组为 `[3,4,4,3]` 。这样后续判断就会多一个(4,3) 递减对。

而如果是 `[3,4,3,3]` ，在这个例子中应该将**索引为 1 的**修改为 3，即 [3,3,3,3]，而不是将索引为 2 的修改为 4，因为**末尾数字越小，对形成递增序列越有利，这就是贪心的思想**。代码上，我们没有必要修改前一项，而是**假设 ta 已经被修改了**即可。之所以可以假设被修改是因为题目只需要返回是否可组成非递减数列，而不需要返回具体的非递减数列是什么，这一点需要大家注意。

大家可以继续找几个测试用例，发现一下问题的规律。比如我找的几个用例： `[4,2,3]` `[4,2,1]` `[1,2,1,2]` `[1,1,1,]` `[]` 。这样就可以写代码了。

关键点

- 考虑各种边界情况，贪心改变数组的值

代码

- 语言支持：Python3

Python3 Code:

```
class Solution(object):
    def checkPossibility(self, A):
        N = len(A)
        count = 0
        for i in range(1, N):
            if A[i] < A[i - 1]:
                count += 1
                if count > 1:
                    return False
                # [4,2,3]  [4,2,1]   [1,2,1,2]  [1,1,1,] []
                if i >= 2 and A[i] < A[i - 2]:
                    A[i] = A[i - 1]

        return True
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

相关专题

- 最长上升子序列

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(821. 字符的最短距离)

https://leetcode-cn.com/problems/shortest-distance-to-a-character

题目描述

```
给定一个字符串 S 和一个字符 C。返回一个代表字符串 S 中每个字符到字符串

示例 1:

输入: S = "loveleetcode", C = 'e'
输出: [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]
说明:

- 字符串 S 的长度范围为 [1, 10000]。
- C 是一个单字符，且保证是字符串 S 里的字符。
- S 和 C 中的所有字母均为小写字母。
```

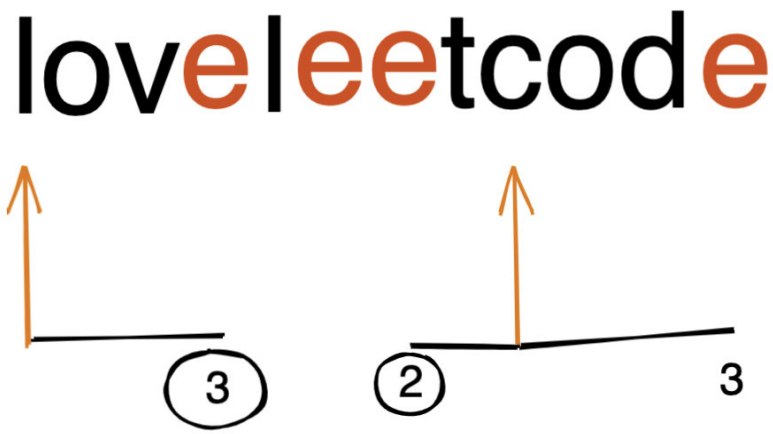
前置知识

- 数组的遍历(正向遍历和反向遍历)

思路

这道题就是让我们求的是向左或者向右距离目标字符最近的距离。

我画了个图方便大家理解：



比如我们要找第一个字符 l 的最近的字符 e，直观的想法就是向左向右分别搜索，遇到字符 e 就停止，比较两侧的距离，并取较小的即可。如上图，l 就是 3，c 就是 2。

这种直观的思路用代码来表示的话是这样的：

Python Code:

```python
class Solution:
    def shortestToChar(self, S: str, C: str) -> List[int]:
        ans = []

        for i in range(len(S)):
            # 从 i 向左向右扩展
            l = r = i
            # 向左找到第一个 C
            while l > -1:
                if S[l] == C: break
                l -= 1
            # 向左找到第一个 C
            while r < len(S):
                if S[r] == C: break
                r += 1
            # 如果至死没有找到，则赋值一个无限大的数字，由于题目的数
            if l == -1: l = -10000
            if r == len(S): r = 10000
            # 选较近的即可
            ans.append(min(r - i, i - l))
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

由于题目的数据范围是 10^4，因此通过所有的测试用例是没有问题的。

但是实际上，我们可以在线性的时间内解决。这里的关键点和上面的解法类似，也是两端遍历。不过不再是盲目的查找，因为这样做会有很多没有必要的计算。

我们可以使用空间换时间的方式来解，这里我使用类似单调栈的解法来解，大家也可以使用其他手段。关于单调栈的技巧，不在这里展开，感兴趣的可以期待我后面的专题。

```
class Solution:
    def shortestToChar(self, S: str, C: str) -> List[int]:
        ans = [10000] * len(S)
        stack = []
        for i in range(len(S)):
            while stack and S[i] == C:
                ans[stack.pop()] = i - stack[-1]
            if S[i] != C:stack.append(i)
            else: ans[i] = 0
        for i in range(len(S) - 1, -1, -1):
            while stack and S[i] == C:
                ans[stack.pop()] = min(ans[stack[-1]], sta
            if S[i] != C:stack.append(i)
            else: ans[i] = 0

        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

实际上，我们根本不需要栈来存储。原因很简单，那就是每次我们碰到目标字符 C 的时候， 我们就把栈**全部清空**了，因此我们用一个变量标识即可，具体参考后面的代码区。

> 如果碰到目标字符 C 的时候，不把栈清空，那么这个栈的空间多半是不能省的，反之可以省。

代码

代码支持：Python3，Java, CPP, Go, PHP

Python3 Code：

```python
class Solution:
    def shortestToChar(self, S: str, C: str) -> List[int]:
        pre = -10000
        ans = []

        for i in range(len(S)):
            if S[i] == C: pre = i
            ans.append(i - pre)
        pre = 20000
        for i in range(len(S) - 1, -1, -1):
            if S[i] == C: pre = i
            ans[i] = min(ans[i], pre - i)
        return ans
```

Java Code：

```java
class Solution {
    public int[] shortestToChar(String S, char C) {
        int N = S.length();
        int[] ans = new int[N];
        int prev = -10000;

        for (int i = 0; i < N; ++i) {
            if (S.charAt(i) == C) prev = i;
            ans[i] = i - prev;
        }

        prev = 20000;
        for (int i = N-1; i >= 0; --i) {
            if (S.charAt(i) == C) prev = i;
            ans[i] = Math.min(ans[i], prev - i);
        }

        return ans;
    }
}
```

CPP Code:

```
class Solution {
public:
    vector<int> shortestToChar(string S, char C) {
        vector<int> ans(S.size(), 0);
        int prev = -10000;
        for(int i = 0; i < S.size(); i ++){
            if(S[i] == C) prev = i;
            ans[i] = i - prev;
        }
        prev = 20000;
        for(int i = S.size() - 1; i >= 0; i --){
            if(S[i] == C) prev = i;
            ans[i] = min(ans[i], prev - i);
        }
        return ans;
    }
};
```

Go Code:

```go
func shortestToChar(S string, C byte) []int {
    N := len(S)
    ans := make([]int, N)

    pre := -N // 最大距离
    for i := 0; i < N; i++ {
        if S[i] == C {
            pre = i
        }
        ans[i] = i - pre
    }

    pre = N*2 // 最大距离
    for i := N - 1; i >= 0; i-- {
        if S[i] == C {
            pre = i
        }
        ans[i] = min(ans[i], pre-i)
    }
    return ans
}

func min(a, b int) int {
    if a < b {
        return a
    }
    return b
}
```

PHP Code:

```
class Solution
{

    /**
     * @param String $S
     * @param String $C
     * @return Integer[]
     */
    function shortestToChar($S, $C)
    {
        $N = strlen($S);
        $ans = [];

        $pre = -$N;
        for ($i = 0; $i < $N; $i++) {
            if ($S[$i] == $C) {
                $pre = $i;
            }
            $ans[$i] = $i - $pre;
        }

        $pre = $N * 2;
        for ($i = $N - 1; $i >= 0; $i--) {
            if ($S[$i] == $C) {
                $pre = $i;
            }
            $ans[$i] = min($ans[$i], $pre - $i);
        }
        return $ans;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（874. 模拟行走机器人）

https://leetcode-cn.com/problems/walking-robot-simulation/submissions/

题目描述

```
机器人在一个无限大小的网格上行走，从点 (0, 0) 处开始出发，面向北方。i

-2：向左转 90 度
-1：向右转 90 度
1 <= x <= 9：向前移动 x 个单位长度
在网格上有一些格子被视为障碍物。

第 i 个障碍物位于网格点 (obstacles[i][0], obstacles[i][1])

如果机器人试图走到障碍物上方，那么它将停留在障碍物的前一个网格方块上，

返回从原点到机器人的最大欧式距离的平方。



示例 1：

输入: commands = [4,-1,3], obstacles = []
输出: 25
解释: 机器人将会到达 (3, 4)
示例 2：

输入: commands = [4,-1,4,-2,4], obstacles = [[2,4]]
输出: 65
解释: 机器人在左转走到 (1, 8) 之前将被困在 (1, 4) 处


提示：

0 <= commands.length <= 10000
0 <= obstacles.length <= 10000
-30000 <= obstacle[i][0] <= 30000
-30000 <= obstacle[i][1] <= 30000
答案保证小于 2 ^ 31
```

前置知识

- hashtable

公司

- 暂无

思路

这道题之所以是简单难度，是因为其没有什么技巧。你只需要看懂题目描述，然后把题目描述转化为代码即可。

唯一需要注意的是查找障碍物的时候如果你采用的是 `线形查找` 会很慢，很可能会超时。

> 我实际测试了一下，确实会超时

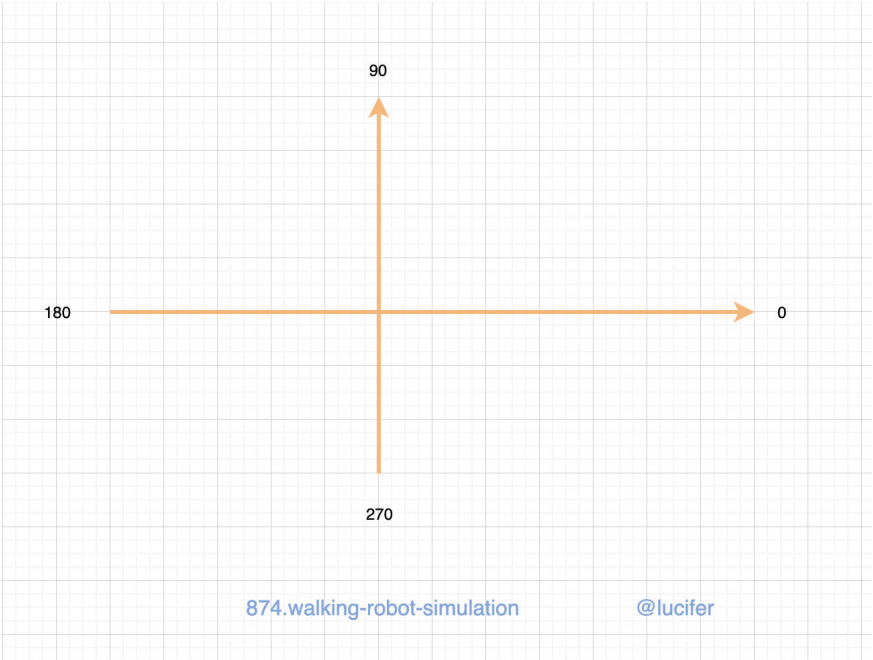
- 一种方式是使用排序，然后二分查找，如果采用基于比较的排序算法，那么这种算法的瓶颈在于排序本身，也就是$O(NlogN)$。
- 另一种方式是使用集合，将 obstacles 放入集合，然后需要的时候进行查询，查询的时候的时间复杂度为$O(1)$。

这里我们采用第二种方式。

接下来我们来“翻译”一下题目。

- 由于机器人只能往前走。因此机器人往东西南北哪个方向走取决于它的 `朝向` 。
- 我们使用枚举来表示当前机器人的 `朝向` 。
- 题目只有两种方式改变 `朝向` ，一种是左转（-2），另一种是右转（-1）。
- 题目要求的是机器人在 `运动过程中距离原点的最大值` ，而不是最终位置距离原点的距离。

为了代码书写简单，我建立了一个直角坐标系。用 `机器人的朝向和 x 轴正方向的夹角度数` 来作为枚举值，并且这个度数是 `0 <= deg < 360` 。我们不难知道，其实这个取值就是 `0` ， `90` ，`180` ，`270` 四个值。那么当 0 度的时候，我们只需要不断地 x+1，90 度的时候我们不断地 y + 1 等等。



关键点解析

- 理解题意，这道题容易理解错题意，求解为 `最终位置距离原点的距离`
- 建立坐标系
- 空间换时间

代码

代码支持： Python3

Python3 Code:

```
class Solution:
    def robotSim(self, commands: List[int], obstacles: List
        pos = [0, 0]
        deg = 90
        ans = 0
        obstaclesSet = set(map(tuple, obstacles))

        for command in commands:
            if command == -1:
                deg = (deg + 270) % 360
            elif command == -2:
                deg = (deg + 90) % 360
            else:
                if deg == 0:
                    i = 0
                    while i < command and not (pos[0] + 1,
                        pos[0] += 1
                        i += 1
                if deg == 90:
                    i = 0
                    while i < command and not (pos[0], pos
                        pos[1] += 1
                        i += 1
                if deg == 180:
                    i = 0
                    while i < command and not (pos[0] - 1,
                        pos[0] -= 1
                        i += 1
                if deg == 270:
                    i = 0
                    while i < command and not (pos[0], pos
                        pos[1] -= 1
                        i += 1
                ans = max(ans, pos[0] ** 2 + pos[1] ** 2)
        return ans
```

复杂度分析

- 时间复杂度：$O(N * M)$, 其中 N 为 commands 的长度， M 为 commands 数组的平均值。
- 空间复杂度：$O(obstacles)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（1128. 等价多米诺骨牌对的数量）

https://leetcode-cn.com/problems/number-of-equivalent-domino-pairs/

题目描述

```
给你一个由一些多米诺骨牌组成的列表 dominoes。

如果其中某一张多米诺骨牌可以通过旋转 0 度或 180 度得到另一张多米诺骨牌

形式上，dominoes[i] = [a, b] 和 dominoes[j] = [c, d] 等价的前提

在 0 <= i < j < dominoes.length 的前提下，找出满足 dominoes[i]



示例：

输入：dominoes = [[1,2],[2,1],[3,4],[5,6]]
输出：1


提示：

1 <= dominoes.length <= 40000
1 <= dominoes[i][j] <= 9
```

前置知识

- 组合计数

“排序” + 计数

思路

我们可以用一个哈希表存储所有的 [a,b] 对的计数信息。为了让形如 [3,4] 和 [4,3]被算到一起，我们可以对其进行排序处理。由于 dominoe 长度固定为 2，因此只需要**判断两者的大小并选择性交换即可**。

接下来，可以使用组合公式 $C*{n}^{2}$ *计算等价骨牌对的数量，其中 n 为单个骨牌的数量。比如**排序后** [3,4] 骨牌对有 5 个，那么 n 就是 5，由 [3,4]构成的等价骨牌对的数量就是 $C*{5}^{2} = 5\times(5-1)\div2 = 10$*

个。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def numEquivDominoPairs(self, dominoes: List[List[int]]
        n = len(dominoes)
        cnt = 0
        cntMapper = dict()

        for a, b in dominoes:
            k = str(a) + str(b) if a > b else str(b) + str(
            cntMapper[k] = cntMapper.get(k, 0) + 1
        for k in cntMapper:
            v = cntMapper[k]
            if v > 1:
                cnt += (v * (v - 1)) // 2
        return cnt
```

复杂度分析

令 N 为数组长度。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

状态压缩 + 一次遍历

思路

观察到题目给的数据范围是 `1 <= dominoes[i][j] <= 9`。这个数字很小，很容易让人想到状态压缩。关于状态压缩这部分可以看我之前写过的题解状压 DP 是什么？这篇题解带你入门

由于数字不会超过 9，因此使用一个 5 bit 表示就足够了。

我们可以用两个 5 bit 分别表示 a 和 b，即一共 10 个 bit 就够了。10 个 bit 一共最多 1024 种状态，因此使用一个 1024 大小的数组是足够的。

上面代码我们是先进行一次遍历，求出计数信息。然后再次遍历计算总和。实际上，我们可以将两者合二为一，专业的话来说就是 **One Pass**，中文是一次遍历。

注意到我们前面计算总和用到了组合公式 C_{n}^{2}，等价于 $n\times(n-1)\div{2}$，这其实就是等差数列 `1,2,3...n-1` 的求和公式。同时注意到我们的计数信息也是每次增加 1 的，即从 0 -> 1, 1 -> 2, n - 1 -> n。也就是说**我们的计数信息其实就是公差为 1 的等差数列，正好对应前面写的等差数列**。那我们是不是可以从 1 开始累加计数信息，直到 n -1（注意不是 n）。

> 力扣中有好几个题目都使用到了这种 One Pass 技巧。

代码

代码支持：Python3

Python3 Code:

```
        counts = [0] * 1024
         ans = 0
         for a, b in dominoes:
             if a >= b: v = a <<5 | b
             else: v = b << 5 | a
             ans += counts[v]
             counts[v] += 1
         return ans
```

复杂度分析

令 N 为数组长度。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1024)$

状态压缩优化

思路

代码上，我使用了 int 来存储，因此实际上会用 32 个字节（取决于不同的编程语言），这并没有发挥二进制的状态压缩的优点。由于 `1 <= dominoes[i][j] <= 9`，我们也可直接用 9 进制来存，刚好 `9 * 9 = 81` 种状态。 这样开辟一个大小为 81 的数组即可。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def numEquivDominoPairs(self, dominoes: List[List[int]]
        counts = [0] * 9 * 9
        ans = 0
        for a, b in dominoes:
            v = min((a - 1) * 9 + (b - 1), (b - 1) * 9 + (a
            ans += counts[v]
            counts[v] += 1
        return ans
```

复杂度分析

令 N 为数组长度。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(81)$

关键点

- 使用状态压缩可提高性能
- 使用求和公式技巧可在一次遍历内计算结果

题目地址（1260. 二维网格迁移）

https://leetcode-cn.com/problems/shift-2d-grid/description/

题目描述

```
给你一个 n 行 m 列的二维网格 grid 和一个整数 k。你需要将 grid 迁移

每次「迁移」操作将会引发下述活动：

位于 grid[i][j] 的元素将会移动到 grid[i][j + 1]。
位于 grid[i][m - 1] 的元素将会移动到 grid[i + 1][0]。
位于 grid[n - 1][m - 1] 的元素将会移动到 grid[0][0]。
请你返回 k 次迁移操作后最终得到的 二维网格。



示例 1：



输入：grid = [[1,2,3],[4,5,6],[7,8,9]], k = 1
输出：[[9,1,2],[3,4,5],[6,7,8]]
示例 2：



输入：grid = [[3,8,1,9],[19,7,2,5],[4,6,11,10],[12,0,21,13]]
输出：[[12,0,21,13],[3,8,1,9],[19,7,2,5],[4,6,11,10]]
示例 3：

输入：grid = [[1,2,3],[4,5,6],[7,8,9]], k = 9
输出：[[1,2,3],[4,5,6],[7,8,9]]


提示：

1 <= grid.length <= 50
1 <= grid[i].length <= 50
-1000 <= grid[i][j] <= 1000
0 <= k <= 100
```

前置知识

- 数组
- 数学

公司

- 字节

模拟法

思路

我们直接翻译题目，没有任何 hack 的做法。

代码

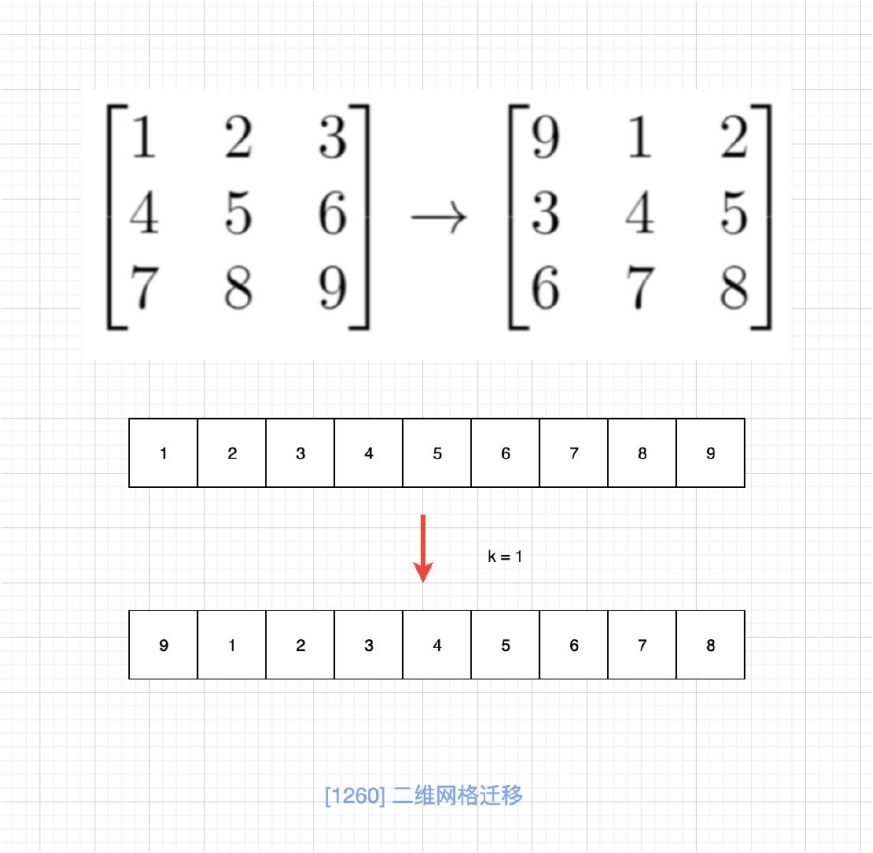
```
from copy import deepcopy

class Solution:
    def shiftGrid(self, grid: List[List[int]], k: int) -> l
        n = len(grid)
        m = len(grid[0])
        for _ in range(k):
            old = deepcopy(grid)
            for i in range(n):
                for j in range(m):
                    if j == m - 1:
                            grid[(i + 1) % n][0] = old[i][j
                    elif i == n - 1 and j == m - 1:
                        grid[0][0] = old[i][j]
                    else:
                        grid[i][j + 1] = old[i][j]
        return grid
```

由于是 easy，上述做法勉强可以过，我们考虑优化。

数学分析

思路

我们仔细观察矩阵会发现，其实这样的矩阵迁移是有规律的。 如图：



[1260] 二维网格迁移

因此这个问题就转化为我们一直的一维矩阵转移问题，LeetCode 也有原题189. 旋转数组，同时我也写了一篇文章文科生都能看懂的循环移位算法专门讨论这个，最终我们使用的是三次旋转法，相关数学证明也有写，很详细，这里不再赘述。

LeetCode 真的是喜欢换汤不换药呀 😂

代码

Python 代码：

```
#
# @lc app=leetcode.cn id=1260 lang=python3
#
# [1260] 二维网格迁移
#

# @lc code=start


class Solution:
    def shiftGrid(self, grid: List[List[int]], k: int) -> l
        n = len(grid)
        m = len(grid[0])
        # 二维到一维
        arr = [grid[i][j] for i in range(n) for j in range(
        # 取模，缩小k的范围，避免无意义的运算
        k %= m * n
        res = []
        # 首尾交换法

        def reverse(l, r):
            while l < r:
                t = arr[l]
                arr[l] = arr[r]
                arr[r] = t
                l += 1
                r -= 1
        # 三次旋转
        reverse(0, m * n - k - 1)
        reverse(m * n - k, m * n - 1)
        reverse(0, m * n - 1)
        # 一维到二维
        row = []
        for i in range(m * n):
            if i > 0 and i % m == 0:
                res.append(row)
                row = []
            row.append(arr[i])
        res.append(row)

        return res

# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$

- 空间复杂度：$O(1)$

相关题目

- 189. 旋转数组

参考

- 文科生都能看懂的循环移位算法

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（1332. 删除回文子序列）

https://leetcode-cn.com/problems/remove-palindromic-subsequences/

题目描述

给你一个字符串 s，它仅由字母 'a' 和 'b' 组成。每一次删除操作都可以从

返回删除给定字符串中所有字符（字符串为空）的最小删除次数。

「子序列」定义：如果一个字符串可以通过删除原字符串某些字符而不改变原字

「回文」定义：如果一个字符串向后和向前读是一致的，那么这个字符串就是一

示例 1：

输入：s = "ababa"
输出：1
解释：字符串本身就是回文序列，只需要删除一次。
示例 2：

输入：s = "abb"
输出：2
解释："abb" -> "bb" -> "".
先删除回文子序列 "a"，然后再删除 "bb"。
示例 3：

输入：s = "baabb"
输出：2
解释："baabb" -> "b" -> "".
先删除回文子序列 "baab"，然后再删除 "b"。
示例 4：

输入：s = ""
输出：0

提示：

0 <= s.length <= 1000
s 仅包含字母 'a' 和 'b'
在真实的面试中遇到过这道题?

前置知识

- 回文

公司

- 暂无

思路

这又是一道“抖机灵”的题目，类似的题目有1297.maximum-number-of-occurrences-of-a-substring

由于只有 a 和 b 两个字符。其实最多的消除次数就是 2。因为我们无论如何都可以先消除全部的 1 再消除全部的 2（先消除 2 也一样），这样只需要两次即可完成。 我们再看一下题目给的一次消除的情况，题目给的例子是“ababa”，我们发现其实它本身就是一个回文串，所以才可以一次全部消除。那么思路就有了：

- 如果 s 是回文，则我们需要一次消除
- 否则需要两次
- 一定要注意特殊情况， 对于空字符串，我们需要 0 次

判读回文的话只需要两个指针夹逼即可，具体思路可参考125. 验证回文串

关键点解析

- 注意审题目，一定要利用题目条件“只含有 a 和 b 两个字符”否则容易做的很麻烦

代码

代码支持：Python3、Java

Python3 Code:

```
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        if s == '':
            return 0
        def isPalindrome(s):
            l = 0
            r = len(s) - 1
            while l < r:
                if s[l] != s[r]:
                    return False
                l += 1
                r -= 1
            return True
        return 1 if isPalindrome(s) else 2
```

如果你觉得判断回文不是本题重点，也可以简单实现：

Python3 Code:

```
class Solution:
    def removePalindromeSub(self, s: str) -> int:
        if s == '':
            return 0
        return 1 if s == s[::-1] else 2
```

Java Code：

```
class Solution {
    public int removePalindromeSub(String s) {
        if ("".equals(s)) {
            return 0;
        }
        if (s.equals(new StringBuilder(s).reverse().toStri
            return 1;
        }
        return 2;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

中等难度题目合集

中等题目是力扣比例最大的部分，因此这部分我的题解也是最多的。 大家不要太过追求难题，先把中等难度题目做熟了再说。

这部分的题目要不需要我们挖掘题目的内含信息， 将其抽象成简单题目。 要么是一些写起来比较麻烦的题目， 一些人编码能力不行就挂了。因此大家一定要自己做， 即使看了题解”会了“，也要自己码一遍。自己不亲自写一遍，里面的细节永远不知道。

以下是我列举的经典题目（带 91 字样的表示出自 **91 天学算法**活动）：

- 面试题 17.09. 第 k 个数
- 面试题 17.23. 最大黑方阵🆕

- 0002. 两数相加

- 0003. 无重复字符的最长子串
- 0005. 最长回文子串
- 0011. 盛最多水的容器
- 0015. 三数之和
- 0017. 电话号码的字母组合
- 0019. 删除链表的倒数第 N 个节点
- 0022. 括号生成
- 0024. 两两交换链表中的节点
- 0029. 两数相除
- 0031. 下一个排列
- 0033. 搜索旋转排序数组
- 0039. 组合总和
- 0040. 组合总和 II
- 0046. 全排列
- 0047. 全排列 II
- 0048. 旋转图像
- 0049. 字母异位词分组
- 0050. Pow(x, n)
- 0055. 跳跃游戏
- 0056. 合并区间
- 0060. 第 k 个排列
- 0061. 旋转链表 91
- 0062. 不同路径
- 0073. 矩阵置零
- 0075. 颜色分类
- 0078. 子集
- 0079. 单词搜索
- 0080. 删除排序数组中的重复项 II

- 0086. 分隔链表
- 0090. 子集 II
- 0091. 解码方法
- 0092. 反转链表 II
- 0094. 二叉树的中序遍历
- 0095. 不同的二叉搜索树 II
- 0096. 不同的二叉搜索树
- 0098. 验证二叉搜索树
- 0102. 二叉树的层序遍历
- 0103. 二叉树的锯齿形层次遍历
- 0113. 路径总和 II
- 0129. 求根到叶子节点数字之和
- 0130. 被围绕的区域
- 0131. 分割回文串
- 0139. 单词拆分
- 0144. 二叉树的前序遍历
- 0147. 对链表进行插入排序 🆕
- 0150. 逆波兰表达式求值
- 0152. 乘积最大子数组
- 0199. 二叉树的右视图
- 0200. 岛屿数量
- 0201. 数字范围按位与
- 0208. 实现 Trie (前缀树)
- 0209. 长度最小的子数组
- 0211. 添加与搜索单词 * 数据结构设计
- 0215. 数组中的第 K 个最大元素
- 0221. 最大正方形
- 0227. 基本计算器 II🆕
- 0229. 求众数 II
- 0230. 二叉搜索树中第 K 小的元素
- 0236. 二叉树的最近公共祖先
- 0238. 除自身以外数组的乘积
- 0240. 搜索二维矩阵 II
- 0279. 完全平方数
- 0309. 最佳买卖股票时机含冷冻期
- 0322. 零钱兑换
- 0328. 奇偶链表
- 0334. 递增的三元子序列
- 0337. 打家劫舍 III
- 0343. 整数拆分
- 0365. 水壶问题
- 0378. 有序矩阵中第 K 小的元素
- 0380. 常数时间插入、删除和获取随机元素
- 0394. 字符串解码 91

- 0416. 分割等和子集
- 0445. 两数相加 II
- 0454. 四数相加 II
- 0464. 我能赢么
- 0494. 目标和
- 0516. 最长回文子序列
- 0513. 找树左下角的值 91
- 0518. 零钱兑换 II
- 0547. 朋友圈
- 0560. 和为 K 的子数组
- 0609. 在系统中查找重复文件
- 0611. 有效三角形的个数
- 0686. 重复叠加字符串匹配 🆕
- 0718. 最长重复子数组
- 0754. 到达终点数字
- 0785. 判断二分图
- 0816. 模糊坐标 🆕
- 0820. 单词的压缩编码
- 0875. 爱吃香蕉的珂珂
- 0877. 石子游戏
- 0886. 可能的二分法
- 0900. RLE 迭代器
- 0911. 在线选举 🆕
- 0912. 排序数组
- 0935. 骑士拨号器
- 0978. 最长湍流子数组 🆕
- 0987. 二叉树的垂序遍历 91
- 1011. 在 D 天内送达包裹的能力
- 1014. 最佳观光组合
- 1015. 可被 K 整除的最小整数
- 1019. 链表中的下一个更大节点
- 1020. 飞地的数量
- 1023. 驼峰式匹配
- 1031. 两个非重叠子数组的最大和
- 1104. 二叉树寻路
- 1131.绝对值表达式的最大值
- 1186. 删除一次得到子数组最大和
- 1218. 最长定差子序列
- 1227. 飞机座位分配概率
- 1261. 在受污染的二叉树中查找元素
- 1262. 可被三整除的最大和
- 1297. 子串的最大出现次数
- 1310. 子数组异或查询
- 1334. 阈值距离内邻居最少的城市

- 1371.每个元音包含偶数次的最长子字符串
- 1381. 设计一个支持增量操作的栈 91
- 1558. 得到目标数组的最少函数调用次数 🆕
- 1574. 删除最短的子数组使剩余数组有序 🆕
- 1631. 最小体力消耗路径 🆕
- 1658. 将 x 减到 0 的最小操作数 🆕

题目地址（面试题 17.09. 第 k 个数）

https://leetcode-cn.com/problems/get-kth-magic-number-lcci/

题目描述

```
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

示例 1:

输入: k = 5

输出: 9
```

前置知识

- 堆
- 状态机
- 动态规划

公司

- 暂无

堆

思路

思路很简单。 就是使用一个小顶堆，然后每次从小顶堆取一个， 取 k 次即可。

唯一需要注意的是重复数字，比如 `3 * 5 = 15` ， `5 * 3 = 15` 。关于去重， 用 set 就对了。

关键点

- 去重

代码(Python)

```python
from heapq import heappop, heappush
class Solution:
    def getKthMagicNumber(self, k: int) -> int:
        heap = [1]
        numbers = set()
        # 每次从小顶堆取一个， 取 k 次即可
        while k:
            cur = heappop(heap)
            if cur not in numbers:
                k -= 1
                heappush(heap, cur * 3)
                heappush(heap, cur * 5)
                heappush(heap, cur * 7)
            numbers.add(cur)
        return cur
```

状态机

思路

这个思路力扣的题目还是蛮多的， 比如西法我之前写的一个 《原来状态机也可以用来刷 LeetCode？》.

思路很简单， 就是用三个指针记录因子是 3，5，7 的最小值的索引。

代码（Python）

```python
class Solution:
    def getKthMagicNumber(self, k: int) -> int:
        p3 = p5 = p7 = 0
        state = [1] + [0] * (k - 1)

        for i in range(1, k):
            state[i] = min(state[p3] * 3, state[p5] * 5, st
            if 3 * state[p3] == state[i]: p3 += 1
            if 5 * state[p5] == state[i]: p5 += 1
            if 7 * state[p7] == state[i]: p7 += 1
        return state[-1]
```

复杂度分析

- 时间复杂度：$O(k)$
- 空间复杂度：$O(k)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 36K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（面试题 17.23. 最大黑方阵）

https://leetcode-cn.com/problems/max-black-square-lcci/

题目描述

```
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

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

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
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
```

前置知识

- 动态规划

公司

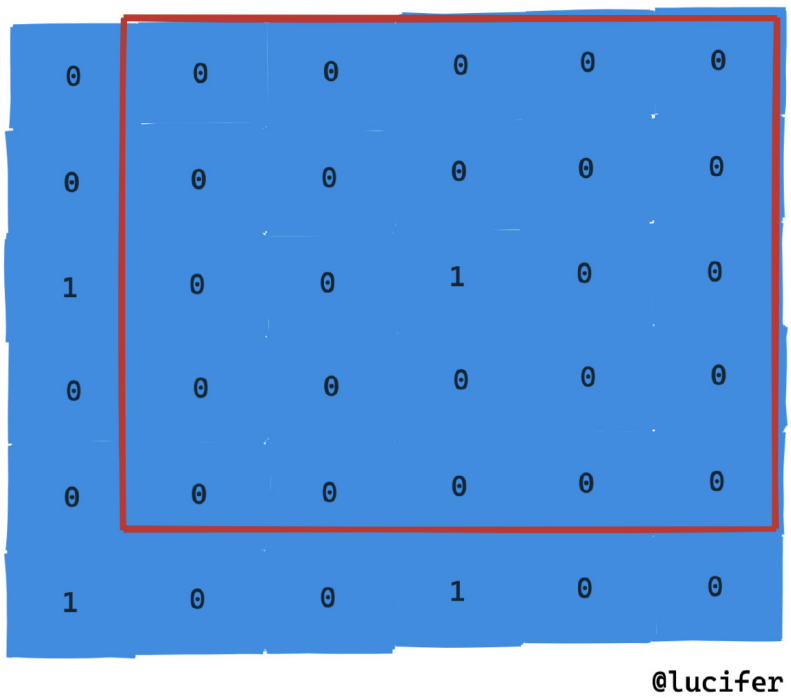
- 暂无

思路

看了下数据范围，矩阵大小不超过 200×200，因此答案应该就是暴力，这个数据范围差不多 N 的三次方的复杂度都可以通过，其中 N 为矩阵的边长。原因我也在之前的文章来和大家聊聊我是如何刷题的（第三弹）中讲过了，那就是 200^3 刚好是是 800 万，再多就很容易超过 **1000 万**了。

乍一看，这道题和 221. 最大正方形，实则不然。 这道题是可以空心的，只要边长是部分都是 0 即可，这就完全不同了。

如下图，红色部分就是答案。只需要保证边全部是 0 就好了，所以里面有一个 1 无所谓的。

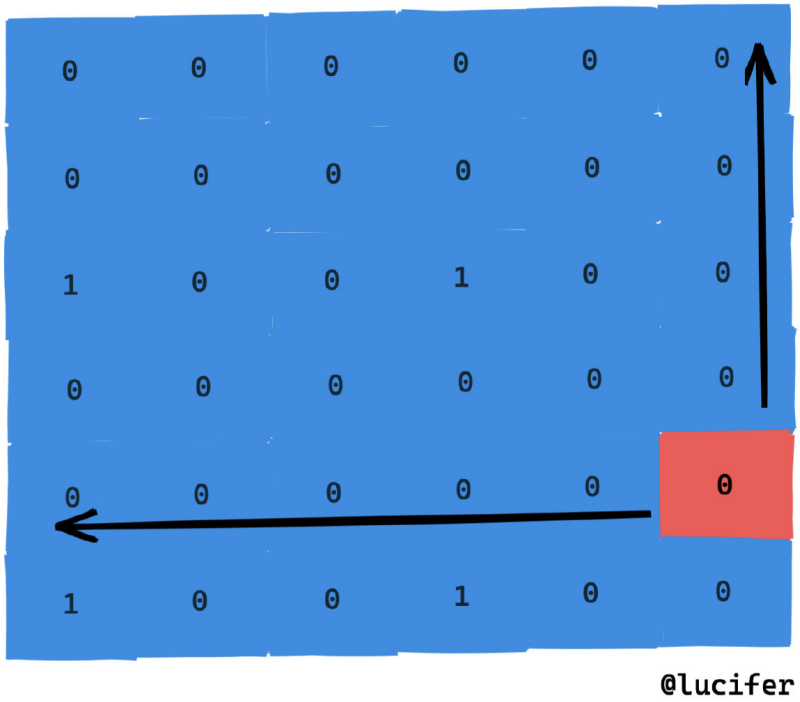


我们不妨从局部入手，看能不能打开思路。

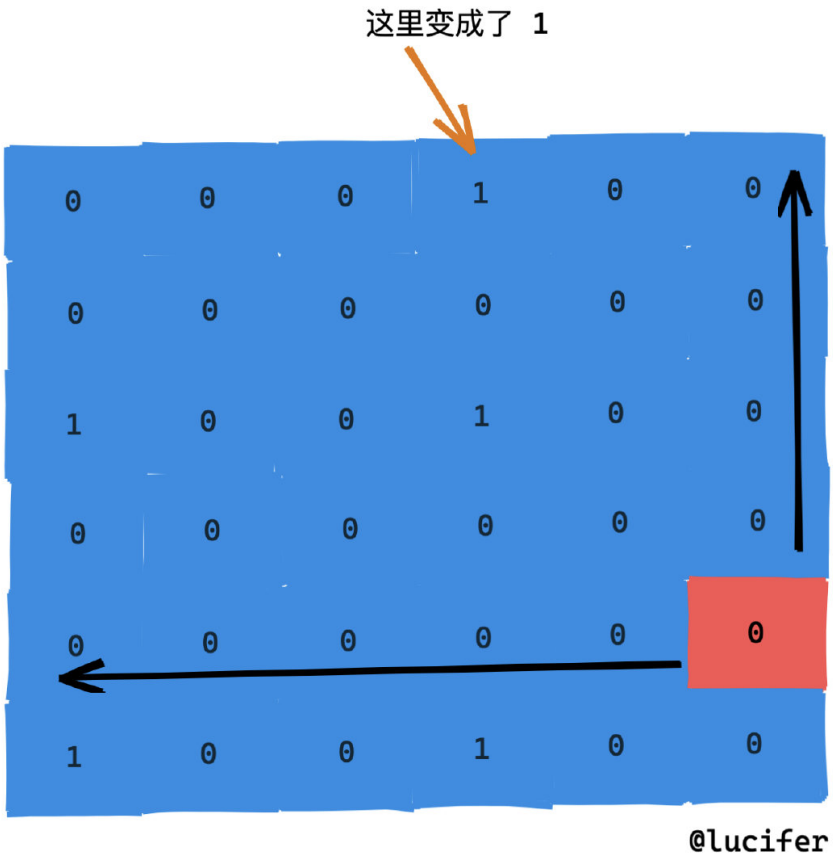
> 这是一种常用的技巧，当你面对难题无从下手的时候可以画个图，从特殊情况和局部情况开始，帮助我们打开思路。

比如我想计算以如下图红色格子为右下角的最大黑方阵。我们可以从当前点向上向左扩展直到非 0。

在上面的例子中，不难看出其最大黑方阵不会超过 min(4, 5)。



那答案直接就是 4 么？ 对于这种情况是的， 但是也存在其他情况。比如：



因此解空间上界虽然是 4，但是下界仍然可能为 1。

> 下界为 1 是因为我们只对值为 0 的格子感兴趣，如果格子为 0 ，最差最大方阵就是自身。

上面我已经将算法锁定为暴力求解了。对于解空间的暴力求解怎么做？无非就是**枚举所有解空间，最多减减枝**。

直白一点来说就是：

- 1 可以么?
- 2 可以么?
- 3 可以么?
- 4 可以么?

这叫做特殊。

如果你已经搞懂了这个特殊情况， 那么一般情况也就不难了。

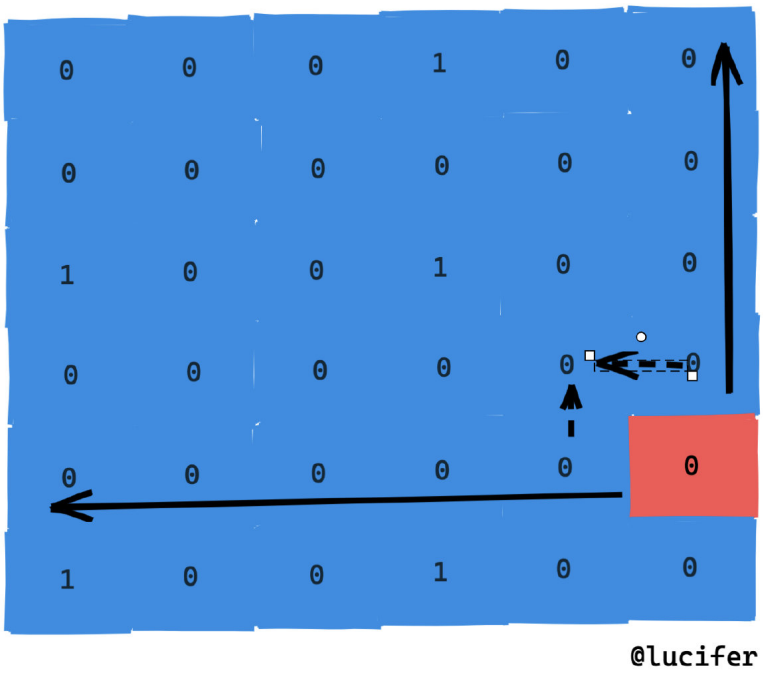
算法描述：

1. 从左到右从上到下扫描一次矩阵。
2. 如果格子值是 0，则分别向上和向左扫描直到第一个不是 0 的格子，那么最大方阵的上界就是 min(向左延伸的长度, 向上延伸的长度)。
3. 逐步尝试[1, 上界] 直到无法形成方阵，最后可行的方阵就是以当前点能 形成的最大方阵。
4. 扫描过程维护最大值，最后返回最大值以及顶点信息即可。

现在的难点只剩下第三点部分**如何逐步尝试[1, 上界]**。实际上这个也不难，只需要：

- 在向左延伸的同时向上探测
- 在向上延伸的同时向左探测

看一下图或许好理解一点。



如上图就是尝试 2 是否可行，如果可行，我们继续**得寸进尺**，直到不可行或者到上界。

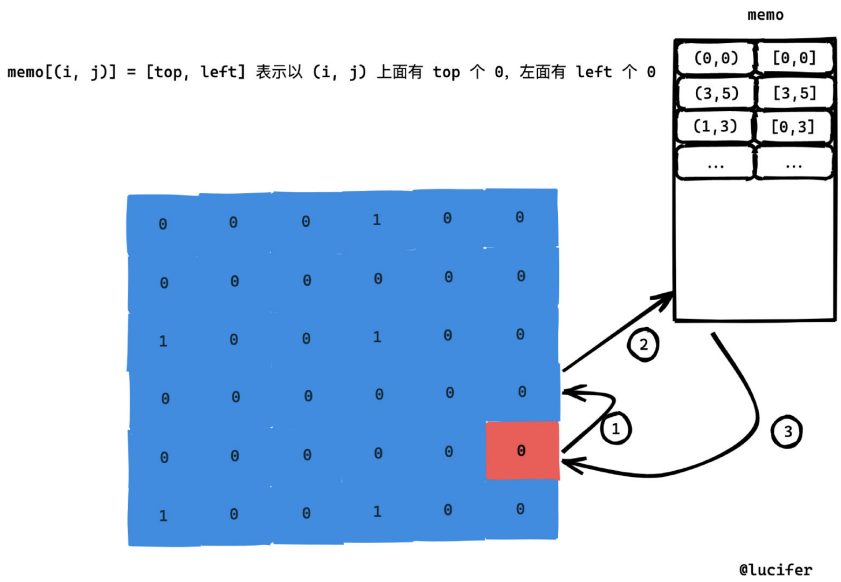
接下来，分析一下算法的时间复杂度。

- 由于每一个为 0 的点都需要向左向上探测，那么最坏就是 O(N)，其中 N 为边长。
- 向左向上的同时需要继续探测，这个复杂度最坏的情况也是 O(N)。

由于我们需要对最多$O(N^2)$个点执行上面的逻辑，因此总的时间复杂度就是 $O(N^4)$

而实际上每一个格子都是一个独立的子问题， 因此可以使用一个 memo（比如哈希表）将每个格子的扩展结果保存起来，这样可以将复杂度优化到 $O(N^3)$。

比如上面提到的向上向左探测的过程，如果上面和左面格子的扩展结果已经计算出来了，那么直接用就行了，这部分延伸的复杂度可以降低到 $O(1)$。因此不难看出， 当前格子的计算依赖于左侧和上方格子，因此使用**从左到右从上到下扫描矩阵** 是正确的选择，因为我们需要在遍历当当前格子的时候**左侧和上方格子的结果已经被计算出来了**。



1. (4,5) 找到上方相邻的格子，如果是 1 直接返回。
2. 如果上方格子值是 0 ， 去 memo 中查询。
3. memo 返回 结果，我们只需要将这个结果 + 1 就是可以向上延伸的最大长度了。

比如现在要计算以坐标(4,5) 为右下角的最大方阵的边长。第一步要向上探测到 (3,5)，到达(3,5) 之后无需继续往上延伸而是从 memo 中获取。(4,5) 上方的 0 的格子就是(3,5) 上方的格子个数 + 1。

最后一个问题。什么数据结构可以实现上面查询过程 $O(1)$ 时间呢? hashmap 可以，数组也可以。

- 使用哈希 map 的好处是无非事先开辟空间。缺点是如果数据量太大，可能会因为哈希表的冲突处理导致超时。比如石子游戏使用哈希表存就很容易超时。
- 使用数组好处和坏处几乎和哈希表是相反的。数组需要实现指定大小， 但是数组是不会冲突的，也不需要计算哈希键，因此在很多情况下性能更好。进一步使用数组这种内存连续的数据结构对 CPU 友好，因此同样复杂度会更快。 而哈希表使用了链表或者树，因此对 CPU 缓存就没那么友好了。

综上，我推荐大家使用数组来存储。

这道题差不多就是这样了。实际上，这就是动态规划优化，其实也没什么神奇嘛，很多时候都是**暴力枚举 + 记忆化**而已。

代码

代码支持：Java，Python

Java Code:

```
class Solution {
    public int[] findSquare(int[][] matrix) {
        int [] res = new int [0];
        int [][][] dp = new int [2][matrix.length+1][matri
        int max = 0
        for(int i=1;i<=matrix.length;i++){
            for(int j=1;j<=matrix[0].length;j++){
                if(matrix[i-1][j-1]==0){
                    dp[0][i][j] = dp[0][i-1][j]+1;
                    dp[1][i][j] = dp[1][i][j-1]+1;
                    int bound = Math.min(dp[0][i][j], dp[1
                    for(int k=0;k<bound;k++){
                        if(dp[1][i-k][j]>=k+1&&dp[0][i][j-k
                            if(k+1>max){
                                res = new int [3];
                                max = k+1;
                                res[0] = i-k-1;
                                res[1] = j-k-1;
                                res[2] = max;
                            }
                        }
                    }
                }
            }
        }
        return res;
    }
}
```

Python Code:

```
class Solution:
    def findSquare(self, matrix: List[List[int]]) -> List[
        n = len(matrix)
        dp = [[[0, 0] for _ in range(n + 1)] for _ in range
        ans = []
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                if matrix[i - 1][j - 1] == 0:
                    dp[i][j][0] = dp[i-1][j][0] + 1
                    dp[i][j][1] = dp[i][j-1][1] + 1
                    upper = min(dp[i][j][0], dp[i][j][1])
                    for k in range(upper):
                        if min(dp[i-k][j][1], dp[i][j-k][0]
                            if not ans or k + 1 > ans[2]:
                                ans = [i-k-1, j-k-1, k + 1]

        return ans
```

复杂度分析

- 时间复杂度：$O(N^3)$，其中 N 为矩阵边长。
- 空间复杂度：空间的瓶颈在于 memo，而 memo 的大小为矩阵的大小，因此空间复杂度为 $O(N^2)$，其中 N 为矩阵边长。

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（475. Increasing Digits）

https://binarysearch.com/problems/Increasing-Digits

题目描述

```
Given an integer n, return the number of positive integers

Example 1
Input
n = 2
Output
36
Explanation
We have 12, 13, 14, ..., 89.

Example 2
Input
n = 1
Output
9
```

前置知识

- 动态规划

思路

动态规划问题的关键就是：假设部分子问题已经解决了，并仅仅考虑局部，思考如何将已解决的子问题变成更大的子问题，这样就相当于向目标走进了一步。

我们可以定义状态 dp[i][j]， i 表示数字的位数，j 表示数字的结尾数字。

于是转移方程就是：dp[i][j] += dp[i - 1][k]，其中 k 是所有小于 j 的非负整数。最后只要返回 dp[n-1] 的和即可。

初始化的时候，由于题目限定了整数，因此需要初始化除了第一位的所有情况都为 1。

关键点

- 数位 DP

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def solve(self, n):
        dp = [[0] * 10 for _ in range(n)]
        dp[0] = [0] + [1] * 9

        for i in range(1, n):
            for j in range(1, 10):
                for k in range(j):
                    dp[i][j] += dp[i - 1][k]
        return sum(dp[-1])
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（168. Longest Contiguously Strictly Increasing Sublist After Deletion）

https://binarysearch.com/problems/Longest-Contiguously-Strictly-Increasing-Sublist-After-Deletion

题目描述

```
Given a list of integers nums, return the maximum length of

Constraints

n ≤ 100,000 where n is the length of nums
Example 1
Input
nums = [30, 1, 2, 3, 4, 5, 8, 7, 22]
Output
7
Explanation
If you remove 8 in the list you can get [1, 2, 3, 4, 5, 7,
```

前置知识

- 动态规划

思路

出这道题就是为了让大家明白一点**对于连续性的 DP 问题通常我们的策略都是一层循环 + 一维 DP（有时候可滚动数组优化）**。比如今天这个题。

动态规划问题的关键就是：假设部分子问题已经解决了，并仅仅考虑局部，思考如何将已解决的子问题变成更大的子问题，这样就相当于向目标走进了一步。

我们可以定义状态：

- dp[i][0] 表示以 nums[i] 结尾的删除 0 个数的情况下的最长严格递增子数组。
- dp[i][1] 表示以 nums[i] 结尾的删除 1 个数的情况下的最长严格递增子数组。

> 你也可定义两个一维数组，而不是一个二维数组。比如 dp0[i] 表示以 nums[i] 结尾的删除 0 个数的情况下的最长严格递增子数组。dp1[i] 表示以 nums[i] 结尾的删除 1 个数的情况下的最长严格递增子数组

接下来，我们需要分情况讨论。

- 如果 nums[i] > nums[i-1]，那么 dp[i][0] 和 dp[i][1] 都可以在前一个的基础上 + 1。也就是：

```
dp[i][0] = dp[i-1][0] + 1
dp[i][1] = dp[i-1][1] + 1
```

- 否则 dp[i][0] = dp[i][1] = 1

最终返回遍历过程中的 dp[i][0] 和 dp[i][1] 的最大值，用一个变量记录即可。

上面的算法少考虑了一个问题，那就是如果 nums[i] > nums[i-2]，我们其实可以选择 nums[i-1]，转而和 dp[i-2] 产生联系。也就是 dp[i][1] = dp[i-2][0] + 1。这个 1 就是将 nums[i-1] 删掉的一个操作。

需要注意的是判断 nums[i] > nums[i-2] 不是在 nums[i] <= nums[i-1] 才需要考虑的。 比如 [1,2,3.0,4] 这种情况。当遍历到 4 的时候，虽然 4 > 0，但是我们不是和 dp[i-1] 结合，这样答案就小了，而是要和 nums[i-2] 结合。

扩展一下，如果题目限定了最多删除 k 个呢?

- 首先状态中列的长度要变成 k
- 其次，我们往前比较的时候要比较 nums[i-1], nums[i-2], ... , nums[i-k-1]，取这 k + 1 种情况的最大值。

关键点

- 连续性 DP

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def solve(self, nums):
        n = len(nums)
        if not n: return 0
        dp = [[1, 0] for _ in range(n)]
        ans = 1

        for i in range(1,n):
            if nums[i] > nums[i-1]:
                dp[i][0] = dp[i-1][0] + 1
                dp[i][1] = dp[i-1][1] + 1
            else:
                dp[i][0] = 1
                dp[i][1] = 1
            if i > 1 and nums[i] > nums[i-2]:
                dp[i][1] = max(dp[i][1], 1 + dp[i-2][0])
            ans = max(ans, dp[i][0], dp[i][1])

        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（726.Consecutive Wins）

https://binarysearch.com/problems/Consecutive-Wins

题目描述

```
You are given integers n and k. Given that n represents the

Constraints

n ≤ 10,000
k ≤ 10
Example 1
Input
n = 3
k = 2
Output
7
Explanation
Here are the ways in which we can win 2 or fewer times cons

"LLL"
"WLL"
"LWL"
"LLW"
"WWL"
"LWW"
```

前置知识

- 递归树
- 动态规划

公司

- 暂无

思路

定义 f(i, j) 表示 i 次比赛连续赢 j 次的总的可能数目。 其实也就是**长度为 n - i 的字符串中连续 w 的次数不超过 j 的总的可能数**，其中 字符串中的字符只能是 L 或 W。

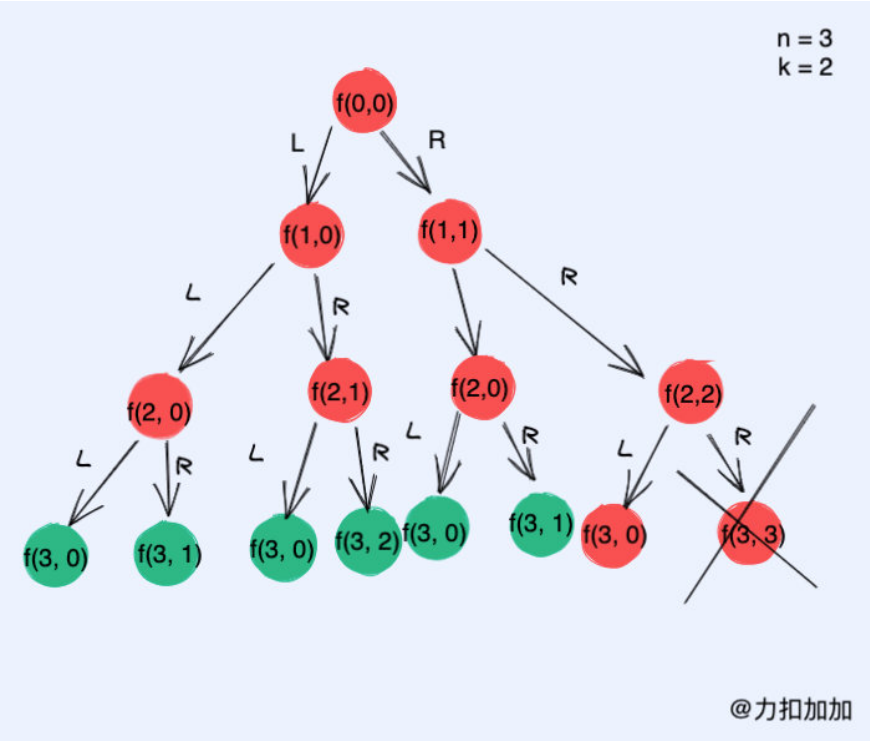
经过这样的定义之后，我们的答案应该是 f(0, 0)。

实际上，也就是问动态规划的转移方程是什么。

对于 f(0, 0)，我们可以：

- 在末尾增加一个 L，也就是输一局。用公式表示就是 f(1, 0)
- 在末尾增加一个 W，也就是赢一局。用公式表示就是 f(1, 1)

用图来表示就是如下的样子：



不是一般性，我们可以得出如下的转移方程：

$$f(i,j) = \begin{cases} f(i+1,0) + f(i+1,j+1) & j < k \\ f(i+1,0) & j = k \end{cases}$$

那么我们的目标其实就是计算图中叶子节点（绿色节点）的总个数。

> 注意 f(3,3) 是不合法的，我们不应该将其计算进去。

上面使我们的递归树代码，可以看出有很多重复的计算。这提示我们使用记忆化递归来解决。使用记忆化递归，总的时间复杂度 节点总数 * 单个节点的操作数。树的节点总数是 n * k，单个节点的操作是常数。故总的时间复杂度为 $O(n * k)，空间复杂度是使用的递归深度 + 记忆化使用的额外空间。其中递归深度是 n，记忆化的空间为 $n *k$，忽略低次项后空间复杂度为 $O(n* k)$*

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def solve(self, n, k):
        @lru_cache(None)
        def dp(i, cnt):
            if i == n:
                return 1
            ans = dp(i + 1, 0)  # place L
            if cnt < k:
                ans += dp(i + 1, cnt + 1)  # place W if I c
            return ans

        return dp(0, 0) % (10 ** 9 + 7)
```

复杂度分析

- 时间复杂度：$O(n * k)$
- 空间复杂度：$O(n * k)$

题目地址（941. Number of Substrings with Single Character Difference）

https://binarysearch.com/problems/Number-of-Substrings-with-Single-Character-Difference

题目描述

```
You are given two lowercase alphabet strings s and t. Retu

Constraints

0 ≤ n ≤ 100 where n is the length of s
0 ≤ m ≤ 100 where m is the length of t
Example 1
Input
s = "ab"
t = "db"
Output
4
Explanation
We can have the following substrings:

"a" changed to "d"
"a" changed to "b"
"b" changed to "d"
"ab" changed to "db"
```

前置知识

- 动态规划

暴力法

思路

暴力的做法是枚举所有的子串 s[i:i+k] 和 s[j:j+k]，其中 0 <= i < m - k, 0 <= j < n - k, 其中 m 和 n 分别为 s 和 t 的长度。

代码上可通过两层循环固定 i 和 j，再使用一层循环确定 k，k 从 0 开始计算。

如果子串不相同的字符：

- 个数为 0 ，则继续寻找。
- 个数为 1， 我们找到了一个可行的解，计数器 + 1
- 个数大于 1，直接 break，寻找下一个子串

最后返回计数器的值即可。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, s, t):
        ans = 0
        for i in range(len(s)):
            for j in range(len(t)):
                mismatches = 0
                for k in range(min(len(s) - i, len(t) - j)):
                    mismatches += s[i + k] != t[j + k]
                    if mismatches == 1:
                        ans += 1
                    elif mismatches > 1:
                        break
        return ans
```

复杂度分析

令 n 为 s 长度，m 为 t 长度。

- 时间复杂度：$O(m \times n \times min(m,n))$
- 空间复杂度：$O(1)$

动态规划

思路

实际上，我们也可通过空间换时间的方式。先对数据进行预处理，然后使用动态规划来求解。

具体来说，我们可以定义二维矩阵 prefix, prefix[i][j] 表示以 s[i] 和 t[j] 结尾的最长前缀的长度（注意我这里的前缀不一定从 s 和 t 的第一个字符开始算）。比如 s = 'qbba', t = 'abbd', 那么 prefix[3][3] 就等于 bb 的长度，也就是 2。

类似地，定义二维矩阵 suffix, suffix[i][j] 表示以 s[i] 和 t[j] 结尾的最长后缀的长度。

这样，我就可以通过两层循环固定确定 i 和 j。如果 s[i] != s[j]，我们找到了 (prefix[i-1][j-1] + 1) * (suffix[i-1][j-1] + 1) 个符合条件的字符组合。也就是前缀+1 和后缀长度+1 的**笛卡尔积**。

关键点

- 建立前后缀 dp 数组，将问题转化为前后缀的笛卡尔积

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, s, t):
        m, n = len(s), len(t)
        prefix = [[0] * (n + 1) for _ in range(m + 1)]
        suffix = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if s[i - 1] == t[j - 1]:
                    prefix[i][j] = prefix[i - 1][j - 1] + 1

        for i in range(m - 1, -1, -1):
            for j in range(n - 1, -1, -1):
                if s[i] == t[j]:
                    suffix[i][j] = suffix[i + 1][j + 1] + 1

        ans = 0
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if s[i - 1] != t[j - 1]:
                    ans += (prefix[i - 1][j - 1] + 1) * (su
        return ans
```

复杂度分析

令 n 为 s 长度，m 为 t 长度。

- 时间复杂度：$O(m \times n)$
- 空间复杂度：$O(m \times n)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 39K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（325.Bus Fare）

https://binarysearch.com/problems/Bus-Fare

题目描述

```
You are given a list of sorted integers days , where you mu

There are 3 types of bus tickets.

1 day pass for 2 dollars
7 day pass for 7 dollars
30 day pass for 25 dollars
Constraints

n ≤ 100,000 where n is the length of days
Example 1
Input
days = [1, 3, 4, 5, 29]
Output
9
Explanation
The lowest cost can be achieved by purchasing a 7 day pass

Example 2
Input
days = [1]
Output
2
```

前置知识

- 递归树
- 多指针

公司

- 暂无

暴力 DP

思路

定义专状态 dp[i] 为截止第 i + 1 天（包括）需要多少钱，因此答案就是 dp[n]，其中 n 为 max(days)，由于 day 是升序的，因此就是 day 最后一项。

使用两层暴力寻找。外层控制 i 天， 内层循环控制 j 天，其中 i <= j。每次我们只考虑进行一次操作：

- 买一张天数 2 的票
- 买一张天数 7 的票
- 买一张天数 25 的票

对于每一个 [i, j]对，我们对计算一遍，求出最小值就可以了。

代码：

```
class Solution:
    def solve(self, days):
        n = len(days)
        prices = [2, 7, 25]
        durations = [1, 7, 30]
        dp = [float("inf")] * (n + 1)
        dp[0] = 0

        for i in range(1, n + 1):
            for j in range(i, n + 1):
                # 如何第 i + 1天到第 j 天的天数小于等于 2，那么我
                # 7 和 25 的逻辑也是一样
        return dp[-1]
```

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, days):
        n = len(days)
        prices = [2, 7, 25]
        durations = [1, 7, 30]
        dp = [float("inf")] * (n + 1)
        # dp[i] 表示截止第 i + 1 天（包括）需要多少钱，因此答案就是
        dp[0] = 0

        for i in range(1, n + 1):
            for j in range(i, n + 1):
                for price, duration in zip(prices, duration
                    if days[j - 1] - days[i - 1] + 1 <= dur
                        dp[j] = min(dp[j], dp[i - 1] + pric
        return dp[-1]
```

复杂度分析

令 m 和 n 分别为 prices 和 days 的长度。

- 时间复杂度：$O(m * n^2)$
- 空间复杂度：$O(n)$

多指针优化

思路

这种算法需要枚举所有的 [i,j] 组合，之后再枚举所有的票，这无疑是完备的，但是复杂度很高。需要进行优化，接下来我们看下如何进行优化。

由于不同的票价的策略是一致的，因此我们可以先仅考虑天数为 2 的。假如两天的票内层循环的 j 找到了第一个满足条件的 i，不妨假设 i 的值是 x。

那么下一次循环，i 指针不必从 0 开始，而是直接从 x 开始，因此 x 之前肯定都无法满足： `days[j - 1] - days[i - 1] + 1 <= duration` 。这是优化的关键，这点其实和《组成三角形的个数》题目类似。关键都在于**指针不回退，达到优化的效果**。 实际上，思想上来看**单调栈**也是类似的。

上面我说了是 i 不回退，实际上不同的天数的票对应的**上一次指针位置是不同的**，我们可以使用一个长度为 m 的指针数组来表示不同天数的票上一次 i 指针的位置。相比于双指针，多指针的编码会稍微复杂一点。

由于 i 指针不需要回退，因此省略了一层 n 的循环，时间复杂度可以从 $O(m *n^2)$ *降低到* $O(m \ n)$。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, days):
        prices = [2, 7, 25]
        durations = [1, 7, 30]
        n = len(days)
        m = len(prices)
        dp = [float("inf")] * (n + 1)
        dp[0] = 0
        pointers = [0] * m
        for i in range(1, n + 1):
            for j in range(m):
                while days[i - 1] - days[pointers[j]] >= du
                    pointers[j] += 1
                dp[i] = min(dp[i], dp[pointers[j]] + prices
        return dp[-1]
```

复杂度分析

令 m 和 n 分别为 prices 和 days 的长度。

- 时间复杂度：$O(m * n)$
- 空间复杂度：$O(m + n)$

题目地址（Minimum Dropping Path Sum）

https://binarysearch.com/problems/Minimum-Dropping-Path-Sum

题目描述

```
You are given a two-dimensional list of integers matrix. Re

Constraints

1 ≤ n ≤ 250 where n is the number of rows in matrix
2 ≤ m ≤ 250 where m is the number of columns in matrix
Example 1
Input
matrix = [
    [4, 5, -2],
    [2, 6, 1],
    [3, 1, 2]
]
Output
1
Explanation
We can take -2 from the first row, 2 from the second row, a

Example 2
Input
matrix = [
    [3, 0, 3],
    [2, 1, 3],
    [-2, 3, 0]
]
Output
1
Explanation
We can take 0 from the first row, 3 from the second row, ar
```

前置知识

- 动态规划

公司

- 暂无

思路

这道题是不同路径（或者杨辉三角）的换皮题。

这道题是让我们每一行选择一个数，使得数字和最小。唯一的限制是相邻行拿的数字不能列相同(其实就是不能上下紧挨着拿)。

一个可能的暴力思路是：

- 先取第一行。第一行有 n （n 为列数）个选择，那就挨个试。
- 接下来取第二行。第二行有 n-1 个选择，那就挨个试。
- 接下来取第三行。第三行有 n-1 个选择，那就挨个试。
- 。。。

不要小看暴力法， 这是一种解决问题的思维习惯。

如果你将上面的过程画成一棵树的话，那么可以看出时间复杂度大概是和底层的节点数是一个数量级的，是指数级别的。就算不画树，你也不难看出大概的计算次数是 n *(n -1)* (n - 1) ...（一共 m - 1 个 n -1）。那么我们可以优化么?

实际上是可以的。我们先不考虑题目的限制”相邻行拿的数字不能列相同“。那么我们的策略就变成了贪婪， 只要每一行都取最小的不就行了？时间复杂度是 $O(m * n)$。

那么加上这个限制会有什么不同么？以题目的例子为例：

```
matrix = [
    [3, 0, 3],
    [2, 1, 3],
    [-2, 3, 0]
]
```

贪心的做法第一行要选 0，第二行要选 1，不过违反了限制。那我们有必要把所有的选择第一行和第二行的组合计算出来么（就像上面的暴力法那样）？其实我们只**记录上一行的最小和次小值**即可。如果出现了上面的情况，那么我们可以考虑：

- 选择 1 和上一行次小值（3 + 1）
- 选择行次小值和上一行最小值（2 + 0）

剩下的逻辑也是如此。

最终我们返回**选择到达**最后一行的**最小值**即可。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, matrix):
        dp = [(0, -1)]
        m, n = len(matrix), len(matrix[0])
        for i in range(m):
            next_dp = [(float("inf"), -1), (float("inf"), -
            for j in range(n):
                for v, k in dp:
                    if k == j:
                        continue
                    nxt = matrix[i][j] + v
                    if nxt < next_dp[0][0]:
                        next_dp = [(nxt, j), next_dp[0]]
                    elif nxt < next_dp[1][0]:
                        next_dp[1] = (nxt, j)
                    break
            dp = next_dp # rolling array
        return dp[0][0]
```

复杂度分析

- 时间复杂度：$O(m*n)$
- 空间复杂度：$O(1)$ （使用了滚动数组优化）

相关题目

- Painting-Houses (换皮题。代码一模一样，直接复制粘贴代码就可以AC)

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(2. 两数相加)

https://leetcode-cn.com/problems/add-two-numbers/

题目描述

```
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

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
```

前置知识

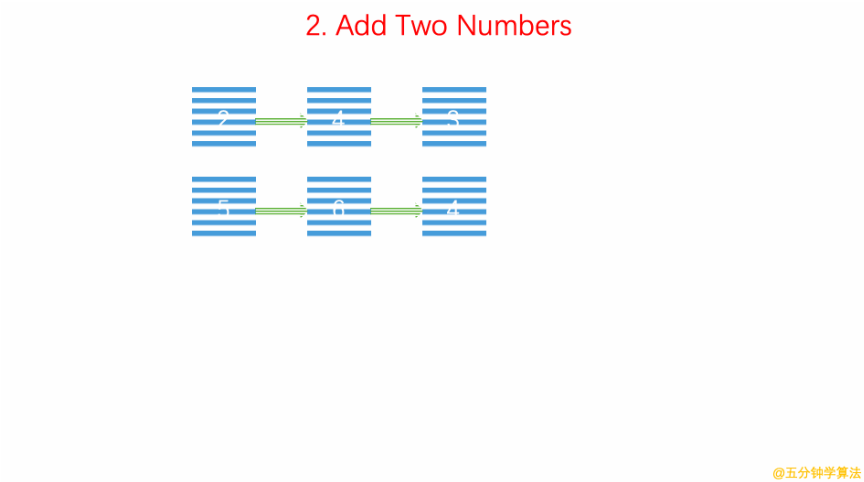
- 链表

公司

- 阿里
- 百度
- 腾讯

思路

设立一个表示进位的变量 carried，建立一个新链表，把输入的两个链表从头往后同时处理，每两个相加，将结果加上 carried 后的值作为一个新节点到新链表后面，并更新 carried 值即可。



(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

实际上两个大数相加也是一样的思路， 只不过具体的操作 API 不一样而已。这种题目思维难度不大，难点在于心细，因此大家一定要自己写一遍，确保 bug free。

关键点解析

1. 链表这种数据结构的特点和使用
2. 用一个 carried 变量来实现进位的功能，每次相加之后计算 carried，并用于下一位的计算

代码

- 语言支持：JS，C++，Java，Python

JavaScript Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
var addTwoNumbers = function (l1, l2) {
  if (l1 === null || l2 === null) return null;

  // 使用dummyHead可以简化对链表的处理，dummyHead.next指向新链表
  let dummyHead = new ListNode(0);
  let cur1 = l1;
  let cur2 = l2;
  let cur = dummyHead; // cur用于计算新链表
  let carry = 0; // 进位标志

  while (cur1 !== null || cur2 !== null) {
    let val1 = cur1 !== null ? cur1.val : 0;
    let val2 = cur2 !== null ? cur2.val : 0;
    let sum = val1 + val2 + carry;
    let newNode = new ListNode(sum % 10); // sum%10取模结果
    carry = sum >= 10 ? 1 : 0; // sum>=10，carry=1，表示有进
    cur.next = newNode;
    cur = cur.next;

    if (cur1 !== null) {
      cur1 = cur1.next;
    }

    if (cur2 !== null) {
      cur2 = cur2.next;
    }
  }

  if (carry > 0) {
    // 如果最后还有进位，新加一个节点
    cur.next = new ListNode(carry);
  }

  return dummyHead.next;
};
```

C++ Code:

> C++代码与上面的 JavaScript 代码略有不同：将 carry 是否为 0 的判断放到了 while 循环中

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* ret = nullptr;
        ListNode* cur = nullptr;
        int carry = 0;
        while (l1 != nullptr || l2 != nullptr || carry != 0
            carry += (l1 == nullptr ? 0 : l1->val) + (l2 ==
            auto temp = new ListNode(carry % 10);
            carry /= 10;
            if (ret == nullptr) {
                ret = temp;
                cur = ret;
            }
            else {
                cur->next = temp;
                cur = cur->next;
            }
            l1 = l1 == nullptr ? nullptr : l1->next;
            l2 = l2 == nullptr ? nullptr : l2->next;
        }
        return ret;
    }
};
```

Java Code:

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2)
        ListNode dummyHead = new ListNode(0);
        ListNode cur = dummyHead;
        int carry = 0;

        while(l1 != null || l2 != null)
        {
            int sum = carry;
            if(l1 != null)
            {
                sum += l1.val;
                l1 = l1.next;
            }
            if(l2 != null)
            {
                sum += l2.val;
                l2 = l2.next;
            }
            // 创建新节点
            carry = sum / 10;
            cur.next = new ListNode(sum % 10);
            cur = cur.next;

        }
        if (carry > 0) {
            cur.next = new ListNode(carry);
        }
        return dummyHead.next;
    }
}
```

Python Code:

```
class Solution:
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        res=ListNode(0)
        head=res
        carry=0
        while l1 or l2 or carry!=0:
            sum=carry
            if l1:
                sum+=l1.val
                l1=l1.next
            if l2:
                sum+=l2.val
                l2=l2.next
            # set value
            if sum<=9:
                res.val=sum
                carry=0
            else:
                res.val=sum%10
                carry=sum//10
            # creat new node
            if l1 or l2 or carry!=0:
                res.next=ListNode(0)
                res=res.next
        return head
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

拓展

通过单链表的定义可以得知，单链表也是递归结构，因此，也可以使用递归的方式来进行 reverse 操作。

> 由于单链表是线性的，使用递归方式将导致栈的使用也是线性的，
> 当链表长度达到一定程度时，递归可能会导致爆栈，

算法

1. 将两个链表的第一个节点值相加，结果转为 0-10 之间的个位数，并设置进位信息
2. 将两个链表第一个节点以后的链表做带进位的递归相加
3. 将第一步得到的头节点的 next 指向第二步返回的链表

C++实现

```
// 普通递归
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        return addTwoNumbers(l1, l2, 0);
    }

private:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2, int
        if (l1 == nullptr && l2 == nullptr && carry == 0) r
        carry += (l1 == nullptr ? 0 : l1->val) + (l2 == nu
        auto ret = new ListNode(carry % 10);
        ret->next = addTwoNumbers(l1 == nullptr ? l1 : l1->
                                  l2 == nullptr ? l2 : l2->n
                                  carry / 10);
        return ret;
    }
};
// 类似尾递归
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* head = nullptr;
        addTwoNumbers(head, nullptr, l1, l2, 0);
        return head;
    }

private:
    void addTwoNumbers(ListNode*& head, ListNode* cur, List
        if (l1 == nullptr && l2 == nullptr && carry == 0) r
        carry += (l1 == nullptr ? 0 : l1->val) + (l2 == nu
        auto temp = new ListNode(carry % 10);
        if (cur == nullptr) {
            head = temp;
            cur = head;
        } else {
            cur->next = temp;
            cur = cur->next;
        }
        addTwoNumbers(head, cur, l1 == nullptr ? l1 : l1->n
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$，其中 N 的空间是调用栈的开销。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(3. 无重复字符的最长子串)

https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/

题目描述

```
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
```

前置知识

- 哈希表
- 滑动窗口

公司

- 阿里
- 字节
- 腾讯

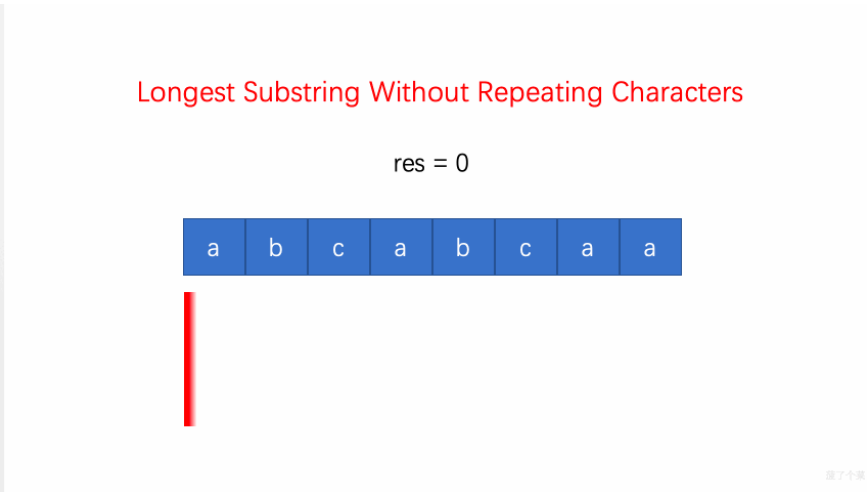
思路

题目要求连续， 我们考虑使用滑动窗口。 而这道题就是**窗口大小不固定**的滑动窗口题目，然后让我们求满足条件的窗口大小的最大值，这是一种非常常见的滑动窗口题目。

算法：

用一个 hashmap 来建立字符和其出现位置之间的映射。同时维护一个滑动窗口，窗口内的都是没有重复的字符，去尽可能的扩大窗口的大小，窗口不停的向右滑动。

1. 如果当前遍历到的字符从未出现过，那么直接扩大右边界；
2. 如果当前遍历到的字符出现过，则缩小窗口（左边索引向右移动），然后继续观察当前遍历到的字符；
3. 重复（1）（2），直到窗口内无重复元素；
4. 维护一个全局最大窗口 res，每次用出现过的窗口大小来更新结果 res，最后返回 res 获取结果;
5. 最后返回 res 即可;



(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点

- mapper 记录出现过并且没有被删除的字符
- 滑动窗口记录当前 index 开始的最大的不重复的字符序列

代码

代码支持：C++，Java，Python3

C++ Code:

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {

        int ans = 0, start = 0;
        int n = s.length();
        //
        map<char, int> mp;

        for(int i=0;i<n;i++)
        {
            char alpha = s[i];
            if(mp.count(alpha))
            {
                start = max(start, mp[alpha]+1);
            }
            ans = max(ans, i-start+1);
            // 字符位置
            mp[alpha] = i;
        }

        return ans;
    }
};
```

Java Code:

```java
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int ans = 0, start = 0;
        int n = s.length();
        //
        Map<Character, Integer> map = new HashMap<>();

        for(int i=0;i<n;i++)
        {
            char alpha = s.charAt(i);
            if(map.containsKey(alpha))
            {
                start = Math.max(start, map.get(alpha)+1);
            }
            ans = Math.max(ans, i-start+1);
            // 字符位置
            map.put(alpha, i);
        }

        return ans;
    }
}
```

Python3 Code:

```python
from collections import defaultdict


class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        l = 0
        ans = 0
        counter = defaultdict(lambda: 0)

        for r in range(len(s)):
            while counter.get(s[r], 0) != 0:
                counter[s[l]] = counter.get(s[l], 0) - 1
                l += 1
            counter[s[r]] += 1
            ans = max(ans, r - l + 1)

        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（5. 最长回文子串）

https://leetcode-cn.com/problems/longest-palindromic-substring/

题目描述

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"

前置知识

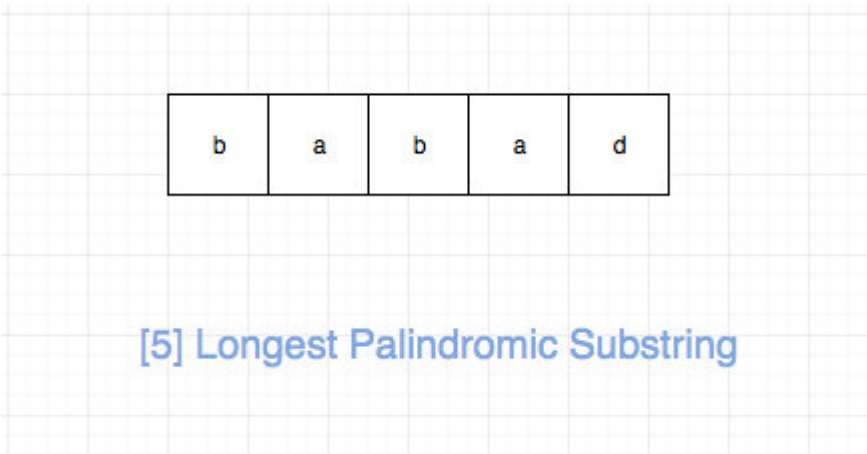
- 回文

公司

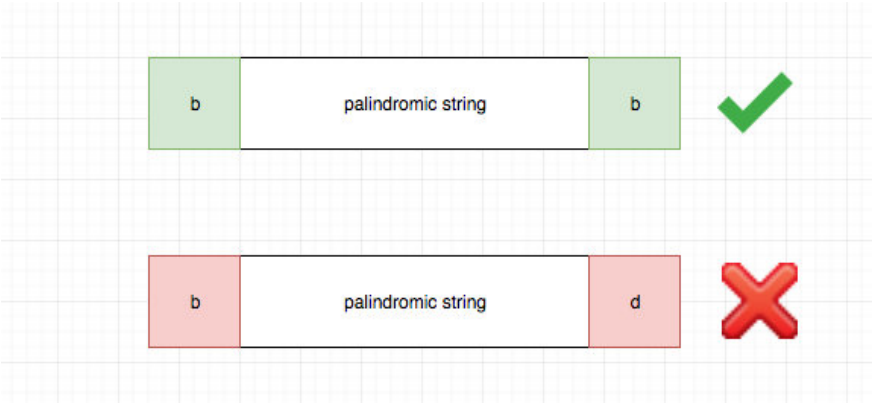
- 阿里
- 百度
- 腾讯

思路

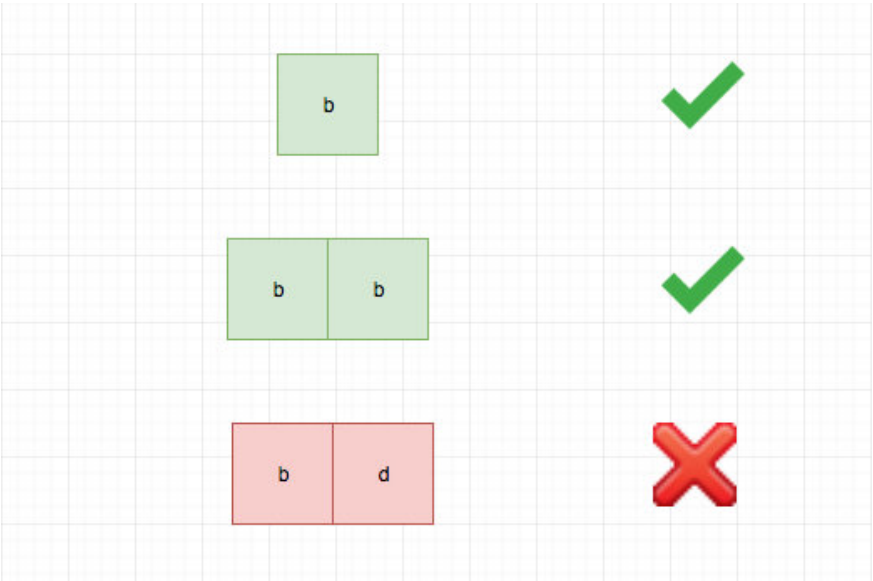
这是一道最长回文的题目，要我们求出给定字符串的最大回文子串。



解决这类问题的核心思想就是两个字“延伸”，具体来说**如果在一个不是回文字符串的字符串两端添加任何字符，或者在回文串左右分别加不同的字符，得到的一定不是回文串**



base case 就是一个字符（轴对称点是本身），或者两个字符（轴对称点是介于两者之间的虚拟点）。



事实上，上面的分析已经建立了大问题和小问题之间的关联，基于此，我们可以建立动态规划模型。

我们可以用 dp[i][j] 表示 s 中从 i 到 j（包括 i 和 j）是否可以形成回文， 状态转移方程只是将上面的描述转化为代码即可：

```
if (s[i] === s[j] && dp[i + 1][j - 1]) {
  dp[i][j] = true;
}
```

关键点

- ”延伸“（extend）

代码

代码支持：Python，JavaScript，CPP

Python Code：

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        if n == 0:
            return ""
        res = s[0]
        def extend(i, j, s):
            while(i >= 0 and j < len(s) and s[i] == s[j]):
                i -= 1
                j += 1
            return s[i + 1:j]

        for i in range(n - 1):
            e1 = extend(i, i, s)
            e2 = extend(i, i + 1, s)
            if max(len(e1), len(e2)) > len(res):
                res = e1 if len(e1) > len(e2) else e2
        return res
```

JavaScript Code：

```javascript
/*
 * @lc app=leetcode id=5 lang=javascript
 *
 * [5] Longest Palindromic Substring
 */
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function (s) {
  // babad
  // tag : dp
  if (!s || s.length === 0) return "";
  let res = s[0];

  const dp = [];

  // 倒着遍历简化操作，  这么做的原因是dp[i][..]依赖于dp[i + 1][..
  for (let i = s.length - 1; i >= 0; i--) {
    dp[i] = [];
    for (let j = i; j < s.length; j++) {
      if (j - i === 0) dp[i][j] = true;
      // specail case 1
      else if (j - i === 1 && s[i] === s[j]) dp[i][j] = tru
      // specail case 2
      else if (s[i] === s[j] && dp[i + 1][j - 1]) {
        // state transition
        dp[i][j] = true;
      }

      if (dp[i][j] && j - i + 1 > res.length) {
        // update res
        res = s.slice(i, j + 1);
      }
    }
  }

  return res;
};
```

CPP Code:

```
class Solution {
private:
    int expand(string &s, int L, int R) {
        while (L >= 0 && R < s.size() && s[L] == s[R]) {
            --L;
            ++R;
        }
        return R - L - 1;
    }
public:
    string longestPalindrome(string s) {
        if (s.empty()) return s;
        int start = 0, maxLen = 0;
        for (int i = 0; i < s.size(); ++i) {
            int len1 = expand(s, i, i);
            int len2 = expand(s, i, i + 1);
            int len = max(len1, len2);
            if (len > maxLen) {
                start = i - (len - 1) / 2;
                maxLen = len;
            }
        }
        return s.substr(start, maxLen);
    }
};
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N^2)$

相关题目

- 516.longest-palindromic-subsequence

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(11. 盛最多水的容器)

https://leetcode-cn.com/problems/container-with-most-water/description/

题目描述

```
给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 （i, a

说明：你不能倾斜容器，且  n  的值至少为 2。

![11.container-with-most-water-question](https://tva1.sina

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



示例:
`
输入：[1,8,6,2,5,4,8,3,7]
输出：49
```

前置知识

- 双指针

公司

- 字节
- 腾讯
- 百度
- 阿里

思路

题目中说 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。 ，因此符合直觉的解法就是固定两个端点，计算可以承载的水量，然后不断更新最大值，最后返回最大值即可。这种算法，需要两层循环，时间复杂度是 $O(n^2)$。

代码（JS）：

```
let max = 0;
for (let i = 0; i < height.length; i++) {
  for (let j = i + 1; j < height.length; j++) {
    const currentArea = Math.abs(i - j) * Math.min(height[
    if (currentArea > max) {
      max = currentArea;
    }
  }
}
return max;
```

虽然解法效率不高，但是可以通过（JS 可以通过，Python 不可以，其他语言没有尝试）。那么有没有更优的解法呢?

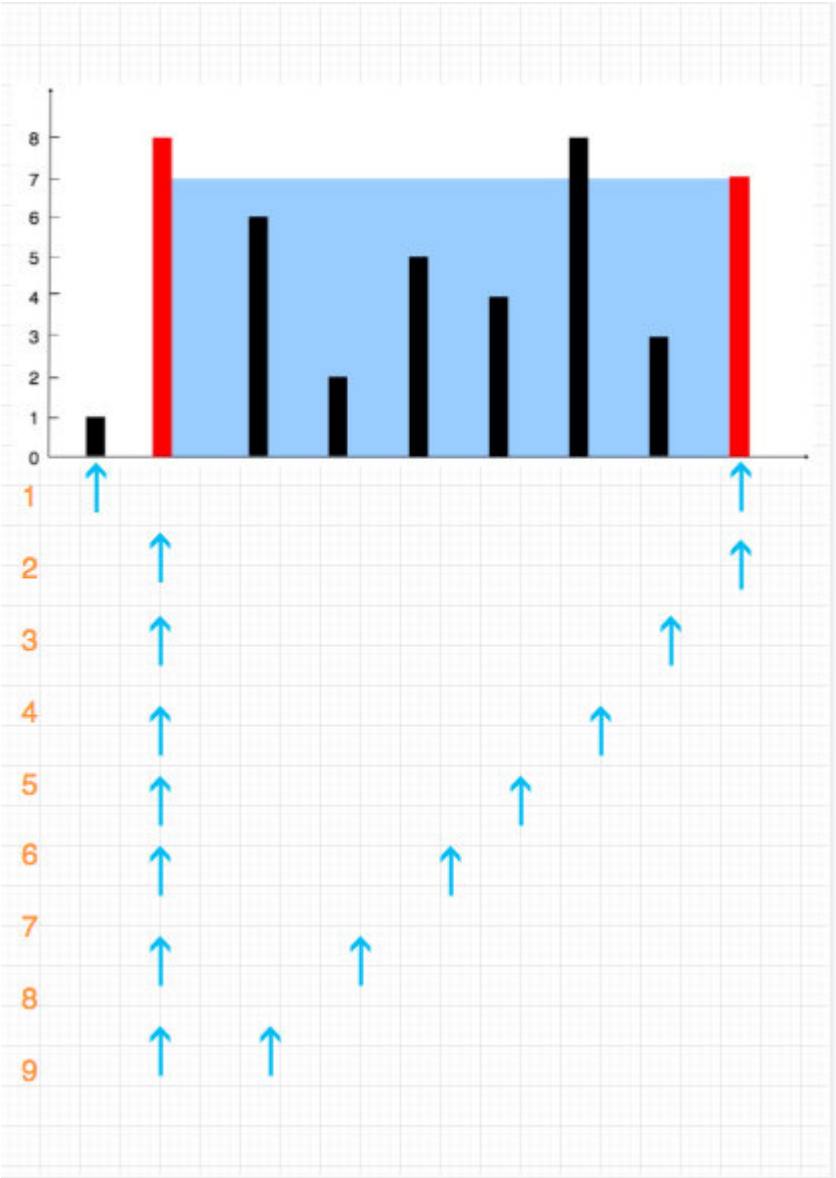
我们来换个角度来思考这个问题，上述的解法是通过两两组合，这无疑是完备的。我们换个角度思考，是否可以:

- 先计算长度为 n 的面积
- 然后计算长度为 n-1 的面积
- ...
- 计算长度为 1 的面积。

很显然这种解法也是完备的，但是似乎时间复杂度还是 $O(n ^ 2)$, 不要着急，我们继续优化。

考虑一下，如果我们计算 n-1 长度的面积的时候，是可以直接排除一半的结果的。

如图:



比如我们计算 n 面积的时候，假如左侧的线段高度比右侧的高度低，那么我们通过左移**右指针**来将长度缩短为 n - 1 的做法是没有意义的，因为 `新形成的面积变成了(n-1) * heightOfLeft， 这个面积一定比刚才的长度为 n 的面积 (n * heightOfLeft) 小` 。

也就是说**最大面积一定是当前的面积或者通过移动短的端点得到。**

关键点解析

- 双指针优化时间复杂度

代码

- 语言支持：JS，C++,Python

JavaScript Code:

```
/**
 * @param {number[]} height
 * @return {number}
 */
var maxArea = function (height) {
  if (!height || height.length <= 1) return 0;

  let leftPos = 0;
  let rightPos = height.length - 1;
  let max = 0;
  while (leftPos < rightPos) {
    const currentArea =
      Math.abs(leftPos - rightPos) *
      Math.min(height[leftPos], height[rightPos]);
    if (currentArea > max) {
      max = currentArea;
    }
    // 更新小的
    if (height[leftPos] < height[rightPos]) {
      leftPos++;
    } else {
      // 如果相等就随便了
      rightPos--;
    }
  }

  return max;
};
```

C++ Code:

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        auto ret = 0ul, leftPos = 0ul, rightPos = height.si
        while( leftPos < rightPos)
        {
            ret = std::max(ret, std::min(height[leftPos], h
            if (height[leftPos] < height[rightPos]) ++leftP
            else --rightPos;
        }
        return ret;
    }
};
```

Python Code:

```
class Solution:
    def maxArea(self, heights):
        l, r =  0, len(heights) - 1
        ans = 0
        while l < r:
            ans = max(ans, (r - l) * min(heights[l], height
            if heights[r] > heights[l]:
                l += 1
            else:
                r -= 1
        return ans
```

复杂度分析

- 时间复杂度：由于左右指针移动的次数加起来正好是 n， 因此时间复杂度为 $O(N)$。
- 空间复杂度：$O(1)$。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(15. 三数之和)

https://leetcode-cn.com/problems/3sum/

题目描述

```
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

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]
]
```

前置知识

- 排序
- 双指针
- 分治

公司

- 阿里
- 字节

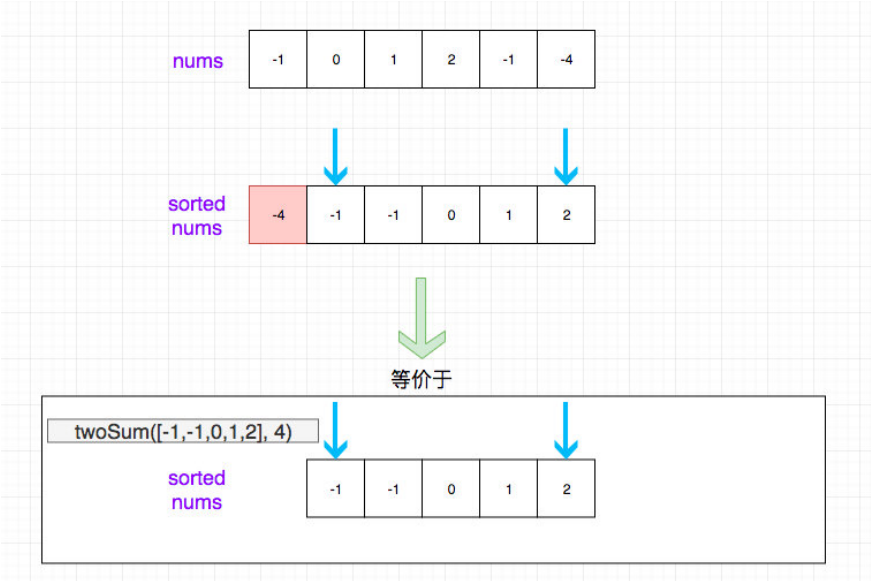
思路

采用 `分治` 的思想找出三个数相加等于 0，我们可以数组依次遍历，每一项 a[i]我们都认为它是最终能够用组成 0 中的一个数字，那么我们的目标就是找到剩下的元素（除 a[i]） `两个` 相加等于-a[i].

通过上面的思路，我们的问题转化为了 `给定一个数组，找出其中两个相加等于给定值` ，我们成功将问题转换为了另外一道力扣的简单题目1. 两数之和。这个问题是比较简单的， 我们只需要对数组进行排序，然后双指针

解决即可。 加上需要外层遍历依次数组，因此总的时间复杂度应该是 O(N^2)。

思路如图所示：



在这里之所以要排序解决是因为， 我们算法的瓶颈在这里不在于排序，而在于 O(N^2)，如果我们瓶颈是排序，就可以考虑别的方式了。

关键点解析

- 排序之后，用双指针
- 分治

代码

代码支持： JS，CPP

JS Code:

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var threeSum = function (nums) {
  if (nums.length < 3) return [];
  const list = [];
  nums.sort((a, b) => a - b);
  for (let i = 0; i < nums.length; i++) {
    //nums is sorted,so it's impossible to have a sum = 0
    if (nums[i] > 0) break;
    // skip duplicated result without set
    if (i > 0 && nums[i] === nums[i - 1]) continue;
    let left = i + 1;
    let right = nums.length - 1;

    // for each index i
    // we want to find the triplet [i, left, right] which s
    while (left < right) {
      // since left < right, and left > i, no need to compa
      if (nums[left] + nums[right] + nums[i] === 0) {
        list.push([nums[left], nums[right], nums[i]]);
        // skip duplicated result without set
        while (nums[left] === nums[left + 1]) {
          left++;
        }
        left++;
        // skip duplicated result without set
        while (nums[right] === nums[right - 1]) {
          right--;
        }
        right--;
        continue;
      } else if (nums[left] + nums[right] + nums[i] > 0) {
        right--;
      } else {
        left++;
      }
    }
  }
  return list;
};
```

CPP Code:

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& A) {
        sort(begin(A), end(A));
        vector<vector<int>> ans;
        int N = A.size();
        for (int i = 0; i < N - 2; ++i) {
            if (i && A[i] == A[i - 1]) continue;
            int L = i + 1, R = N - 1;
            while (L < R) {
                int sum = A[i] + A[L] + A[R];
                if (sum == 0) ans.push_back({ A[i], A[L], A
                if (sum >= 0) {
                    --R;
                    while (L < R && A[R] == A[R + 1]) --R;
                }
                if (sum <= 0) {
                    ++L;
                    while (L < R && A[L] == A[L - 1]) ++L;
                }
            }
        }
        return ans;
    }
}
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：取决于排序算法的空间消耗

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(17. 电话号码的字母组合)

https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number

题目描述

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。

前置知识

- 回溯

- 笛卡尔积

公司

- 阿里
- 百度
- 字节
- 腾讯

回溯

思路

由于要求所有的可能性，因此考虑使用回溯法进行求解。回溯是一种通过穷举所有可能情况来找到所有解的算法。如果一个候选解最后被发现并不是可行解，回溯算法会舍弃它，并在前面的一些步骤做出一些修改，并重新尝试找到可行解。究其本质，其实就是枚举。

如果没有更多的数字需要被输入，说明当前的组合已经产生。

如果还有数字需要被输入：

- 遍历下一个数字所对应的所有映射的字母
- 将当前的字母添加到组合最后，也就是 str + tmp[r]

关键点

- 回溯
- 回溯模板

代码

- 语言支持：JS, C++, Java, Python

JavaScript Code:

```
/**
 * @param {string} digits
 * @return {string[]}
 */
const letterCombinations = function (digits) {
  if (!digits) {
    return [];
  }
  const len = digits.length;
  const map = new Map();
  map.set("2", "abc");
  map.set("3", "def");
  map.set("4", "ghi");
  map.set("5", "jkl");
  map.set("6", "mno");
  map.set("7", "pqrs");
  map.set("8", "tuv");
  map.set("9", "wxyz");
  const result = [];

  function generate(i, str) {
    if (i == len) {
      result.push(str);
      return;
    }
    const tmp = map.get(digits[i]);
    for (let r = 0; r < tmp.length; r++) {
      generate(i + 1, str + tmp[r]);
    }
  }
  generate(0, "");
  return result;
};
```

C++ Code:

```
class Solution {
public:
    string letterMap[10] = {" "," ","abc","def","ghi","jkl"
    vector<string> res;
    vector<string> letterCombinations(string digits) {
        if(digits == "")
        {
            return res;
        }
        dfs(digits, 0, "");
        return res;
    }

    void dfs(string digits, int index, string s)
    {
        if(index == digits.length())
        {
            res.push_back(s);
            return;
        }
        // 获取当前数字
        char c = digits[index];
        // 获取数字对应字母
        string letters = letterMap[c-'0'];
        for(int i = 0 ; i < letters.length() ; i ++)
        {
            dfs(digits, index+1, s+letters[i]);
        }
    }
}
```

Java Code:

```
class Solution {

    private String letterMap[] = {
            " ",    //0
            "",     //1
            "abc",  //2
            "def",  //3
            "ghi",  //4
            "jkl",  //5
            "mno",  //6
            "pqrs", //7
            "tuv",  //8
            "wxyz"  //9
    };
    private ArrayList<String> res;
    public List<String> letterCombinations(String digits) {
        res = new ArrayList<String>();
        if(digits.equals(""))
        {
            return res;
        }
        dfs(digits, 0, "");
        return res;
    }

    public void dfs(String digits, int index, String s)
    {
        if(index == digits.length())
        {
            res.add(s);
            return;
        }
        // 获取当前数字
        Character c = digits.charAt(index);
        // 获取数字对应字母
        String letters = letterMap[c-'0'];
        for(int i = 0 ; i < letters.length() ; i ++)
        {
            dfs(digits, index+1, s+letters.charAt(i));
        }
    }
}
```

Python Code:

```
class Solution(object):
    def letterCombinations(self, digits):
        """
        :type digits: str
        :rtype: List[str]
        """
        if not digits:
            return []
        # 0-9
        self.d = [" "," ","abc","def","ghi","jkl","mno","pq
        self.res = []
        self.dfs(digits, 0, "")
        return self.res

    def dfs(self, digits, index, s):
        # 递归的终止条件,用index记录每次遍历到字符串的位置
        if index == len(digits):
            self.res.append(s)
            return
        # 获取当前数字
        c = digits[index]
        # print(c, int(c))
        # 获取数字对应字母
        letters = self.d[int(c)]
        # 遍历字符串
        for l in letters:
            # 调用下一层
            self.dfs(digits, index+1, s+l)
```

复杂度分析

N + M 是输入数字的总数

- 时间复杂度：O(2^N)，其中 N 为 digits 对于的所有可能的字母的和。
- 空间复杂度：O(2^N)，其中 N 为 digits 对于的所有可能的字母的和。

笛卡尔积

思路

不难发现， 题目要求的是一个笛卡尔积。

比如 digits = 'ab'，其实就是 a 对应的集合 {'a', 'b', 'c'} 和 b 对应的集合 {'d', 'e', 'f'} 笛卡尔积。

简单回忆一下笛卡尔积的内容。对于两个集合 A 和 B，A×B = {(x,y)|x∈A∧y∈B}。

例如，A={a,b}, B={0,1,2}，则：

- A×B={(a, 0), (a, 1), (a, 2), (b, 0), (b, 1), (b, 2)}
- B×A={(0, a), (0, b), (1, a), (1, b), (2, a), (2, b)}

实际上，力扣关于笛卡尔积优化的题目并不少。 比如：

- 140. 单词拆分 II
- 95. 不同的二叉搜索树 II
- 96.unique-binary-search-trees
- 等等

知道了这一点之后，就不难写出如下代码。

由于我们使用了笛卡尔积优化， 因此可以改造成纯函数，进而使用记忆化递归，进一步降低时间复杂度， 这是一个常见的优化技巧。

关键点

- 笛卡尔积
- 记忆化递归

代码

代码支持：Python3

```
# 输入："23"
# 输出：["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        mapper = [" ", " ", "abc", "def", "ghi",
                  "jkl", "mno", "pqrs", "tuv", "wxyz"]
            @lru_cache(None)
        def backtrack(digits, start):
            if start >= len(digits):
                return ['']
            ans = []
            for i in range(start, len(digits)):
                for c in mapper[int(digits[i])]:
                    # 笛卡尔积
                    for p in backtrack(digits, i + 1):
                        # 需要过滤诸如  "d", "e", "f" 等长度不
                        if start == 0:
                            if len(c + p) == len(digits):
                                ans.append(c + p)
                        else:
                            ans.append(c + p)
            return ans
        if not digits:
            return []
        return backtrack(digits, 0)
```

复杂度分析

N + M 是输入数字的总数

- 时间复杂度：O(N^2)，其中 N 为 digits 对于的所有可能的字母的和。
- 空间复杂度：O(N^2)，其中 N 为 digits 对于的所有可能的字母的和。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(19. 删除链表的倒数第 N 个节点)

https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/

题目描述

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?

前置知识

- 链表
- 双指针

公司

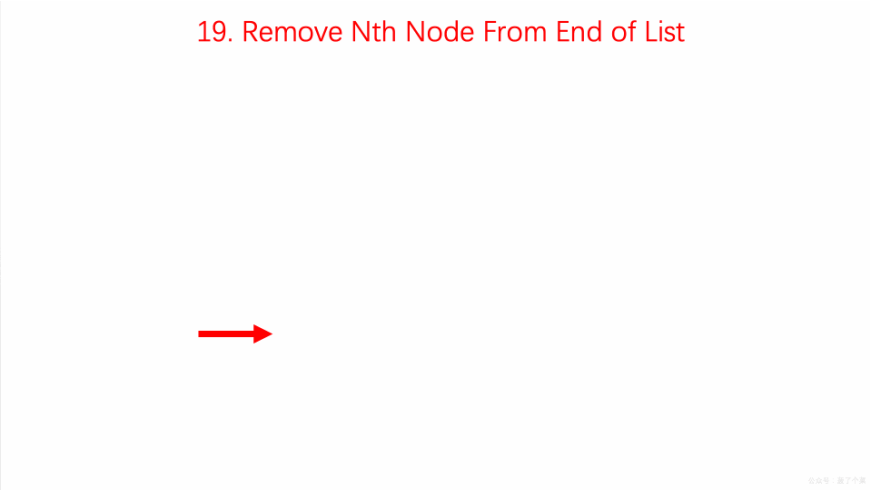
- 阿里
- 百度
- 腾讯
- 字节

思路

这里我们可以使用双指针算法，不妨设为指针 A 和 指针 B。指针 A 先移动 n 次， 指针 B 再开始移动。当 A 到达 null 的时候， 指针 B 的位置正好是倒数第 n。这个时候将 B 的指针指向 B 的下下个指针即可完成删除工作。

算法：

- 设置虚拟节点 dummyHead 指向 head（简化判断，使得头结点不需要特殊判断）
- 设定双指针 p 和 q，初始都指向虚拟节点 dummyHead
- 移动 q，直到 p 与 q 之间相隔的元素个数为 n
- 同时移动 p 与 q，直到 q 指向的为 NULL
- 将 p 的下一个节点指向下下个节点



(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

1. 链表这种数据结构的特点和使用
2. 使用双指针
3. 使用一个 dummyHead 简化操作

代码

代码支持: JS, Java，CPP

Javascript Code:

```
/**
 * @param {ListNode} head
 * @param {number} n
 * @return {ListNode}
 */
var removeNthFromEnd = function (head, n) {
  let i = -1;
  const noop = {
    next: null,
  };

  const dummyHead = new ListNode(); // 增加一个dummyHead 简化
  dummyHead.next = head;

  let currentP1 = dummyHead;
  let currentP2 = dummyHead;

  while (currentP1) {
    if (i === n) {
      currentP2 = currentP2.next;
    }

    if (i !== n) {
      i++;
    }

    currentP1 = currentP1.next;
  }

  currentP2.next = ((currentP2 || noop).next || noop).next;

  return dummyHead.next;
};
```

Java Code:

```
/**
 * 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)
        TreeNode dummy = new TreeNode(0);
        dummy.next = head;
        TreeNode first = dummy;
        TreeNode second = dummy;

        if (int i=0; i<=n; i++) {
            first = first.next;
        }

        while (first != null) {
            first = first.next;
            second = second.next;
        }

        second.next = second.next.next;

        return dummy.next;
    }
}
```

CPP Code:

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *p = head, *q = head;
        while (n--) q = q->next;
        if (!q) {
            head = head->next;
            delete p;
            return head;
        }
        while (q->next) p = p->next, q = q->next;
        q = p->next;
        p->next = q->next;
        delete q;
        return head;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(22. 括号生成)

https://leetcode-cn.com/problems/generate-parentheses

题目描述

```
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

示例：

输入：n = 3
输出：[
       "((()))",
       "(()())",
       "(())()",
       "()(())",
       "()()()"
     ]
```

前置知识

- DFS
- 回溯法

公司

- 阿里
- 百度
- 腾讯
- 字节

思路

本题是 20. 有效括号 的升级版。

由于我们需要求解所有的可能， 因此回溯就不难想到。回溯的思路和写法相对比较固定，并且回溯的优化手段大多是剪枝。

不难想到， 如果左括号的数目小于右括号，我们可以提前退出，这就是这道题的剪枝。 比如 `())....` ，后面就不用看了，直接退出即可。回溯的退出条件也不难想到，那就是：

- 左括号数目等于右括号数目

- 左括号数目 + 右括号数目 = 2 * n

由于我们需要剪枝， 因此必须从左开始遍历。（WHY？）

因此这道题我们可以使用深度优先搜索(回溯思想)，从空字符串开始构造，做加法， 即 dfs(左括号数, 右括号数目, 路径)， 我们从 dfs(0, 0, '') 开始。

伪代码：

```
res = []
def dfs(l, r, s):
   if l > n or r > n: return
   if (l == r == n): res.append(s)
   # 剪枝，提高算法效率
   if l > r: return
   # 加一个左括号
   dfs(l + 1, r, s + '(')
   # 加一个右括号
   dfs(l, r + 1, s + ')')
dfs(0, 0, '')
return res
```

由于字符串的不可变性， 因此我们无需 `撤销 s 的选择` 。但是当你使用 C++ 等语言的时候， 就需要注意撤销 s 的选择了。类似：

```
s.push_back(')');
dfs(l, r + 1, s);
s.pop_back();
```

关键点

- 当 l < r 时记得剪枝

代码

- 语言支持：JS，Python3，CPP

JS Code:

```
/**
 * @param {number} n
 * @return {string[]}
 * @param l 左括号已经用了几个
 * @param r 右括号已经用了几个
 * @param str 当前递归得到的拼接字符串结果
 * @param res 结果集
 */
const generateParenthesis = function (n) {
  const res = [];

  function dfs(l, r, str) {
    if (l == n && r == n) {
      return res.push(str);
    }
    // l 小于 r 时不满足条件 剪枝
    if (l < r) {
      return;
    }
    // l 小于 n 时可以插入左括号，最多可以插入 n 个
    if (l < n) {
      dfs(l + 1, r, str + "(");
    }
    // r < l 时 可以插入右括号
    if (r < l) {
      dfs(l, r + 1, str + ")");
    }
  }
  dfs(0, 0, "");
  return res;
};
```

Python Code:

```python
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        res = []
        def dfs(l, r, s):
            if l > n or r > n: return
            if (l == r == n): res.append(s)
            if l < r: return
            # 加一个左括号
            dfs(l + 1, r, s + '(')
            # 加一个右括号
            dfs(l, r + 1, s + ')')
        dfs(0, 0, '')
        return res
```

CPP Code:

```cpp
class Solution {
private:
    vector<string> ans;
    void generate(int leftCnt, int rightCnt, string &s) {
        if (!leftCnt && !rightCnt) {
            ans.push_back(s);
            return;
        }
        if (leftCnt) {
            s.push_back('(');
            generate(leftCnt - 1, rightCnt, s);
            s.pop_back();
        }
        if (rightCnt > leftCnt) {
            s.push_back(')');
            generate(leftCnt, rightCnt - 1, s);
            s.pop_back();
        }
    }
public:
    vector<string> generateParenthesis(int n) {
        string s;
        generate(n, n, s);
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：O(2^N)
- 空间复杂度：O(2^N)

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

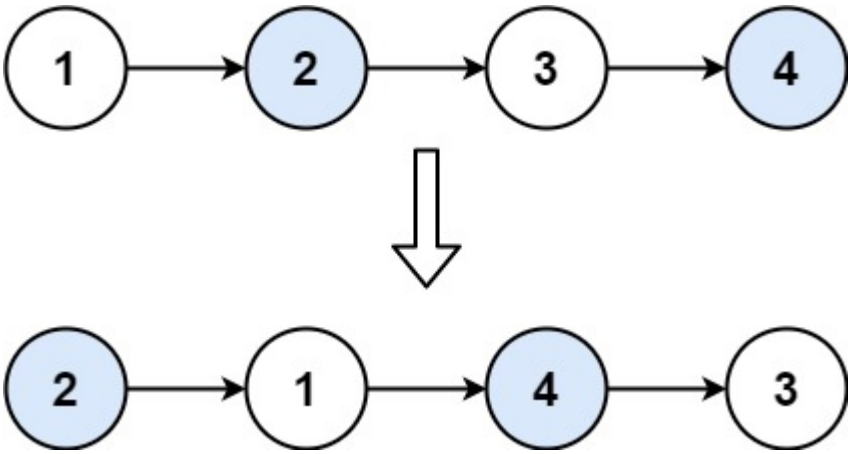
题目地址(24. 两两交换链表中的节点)

https://leetcode-cn.com/problems/swap-nodes-in-pairs/

题目描述

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 1:
输入: head = [1,2,3,4]
输出: [2,1,4,3]

示例 2:
输入: head = []
输出: []

示例 3:
输入: head = [1]
输出: [1]

提示:
链表中节点的数目在范围 [0, 100] 内
0 <= Node.val <= 100
```

前置知识

- 链表

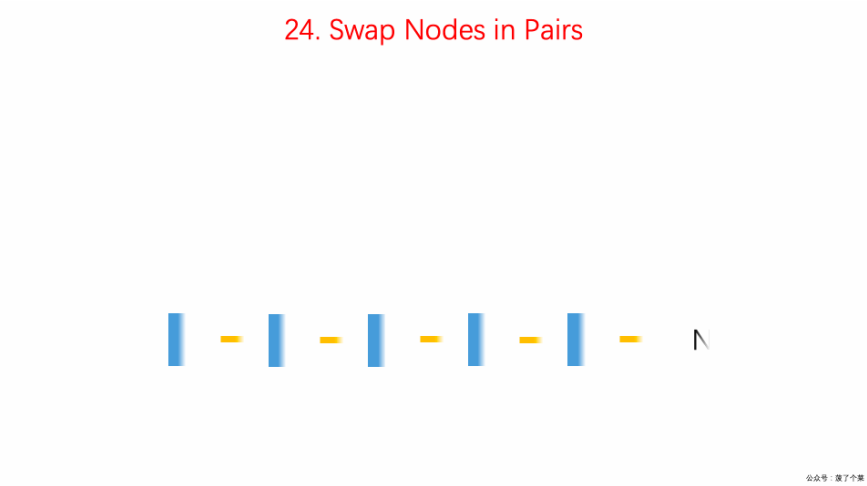
公司

- 阿里
- 腾讯
- 百度
- 字节

思路

设置一个 dummy 节点简化操作，dummy next 指向 head。

1. 初始化 first 为第一个节点
2. 初始化 second 为第二个节点
3. 初始化 current 为 dummy
4. first.next = second.next
5. second.next = first
6. current.next = second
7. current 移动两格
8. 重复



（图片来自： https://github.com/MisterBooo/LeetCodeAnimation）

关键点解析

1. 链表这种数据结构的特点和使用
2. dummyHead 简化操作

代码

- 语言支持：JS，Python3, Go, PHP, CPP

JS Code:

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var swapPairs = function (head) {
  const dummy = new ListNode(0);
  dummy.next = head;
  let current = dummy;
  while (current.next != null && current.next.next != null)
    // 初始化双指针
    const first = current.next;
    const second = current.next.next;

    // 更新双指针和 current 指针
    first.next = second.next;
    second.next = first;
    current.next = second;

    // 更新指针
    current = current.next.next;
  }
  return dummy.next;
};
```

Python Code:

```python
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        """
        用递归实现链表相邻互换：
        第一个节点的 next 是第三、第四个节点交换的结果，第二个节点的
        第三个节点的 next 是第五、第六个节点交换的结果，第四个节点的
        以此类推
        :param ListNode head
        :return ListNode
        """
        # 如果为 None 或 next 为 None，则直接返回
        if not head or not head.next:
            return head

        _next = head.next
        head.next = self.swapPairs(_next.next)
        _next.next = head
        return _next
```

Go Code:

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func swapPairs(head *ListNode) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }

    next := head.Next
    head.Next = swapPairs(next.Next) // 剩下的节点递归已经处理
    next.Next = head
    return next
}
```

PHP Code:

```
/**
 * Definition for a singly-linked list.
 * class ListNode {
 *     public $val = 0;
 *     public $next = null;
 *     function __construct($val = 0, $next = null) {
 *         $this->val = $val;
 *         $this->next = $next;
 *     }
 * }
 */
class Solution
{

    /**
     * @param ListNode $head
     * @return ListNode
     */
    function swapPairs($head)
    {
        if (!$head || !$head->next) return $head;

        /** @var ListNode $next */
        $next = $head->next;
        $head->next = (new Solution())->swapPairs($next->ne
        $next->next = $head;
        return $next;
    }
}
```

CPP Code:

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode h, *tail = &h;
        while (head && head->next) {
            auto p = head, q = head->next;
            head = q->next;
            q->next = p;
            tail->next = q;
            tail = p;
        }
        tail->next = head;
        return h.next;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(29. 两数相除)

https://leetcode-cn.com/problems/divide-two-integers/

题目描述

```
给定两个整数，被除数 dividend 和除数 divisor。将两数相除，要求不使用

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



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 [−231,  231 −
```

前置知识

- 二分法

公司

- Facebook
- Microsoft
- Oracle

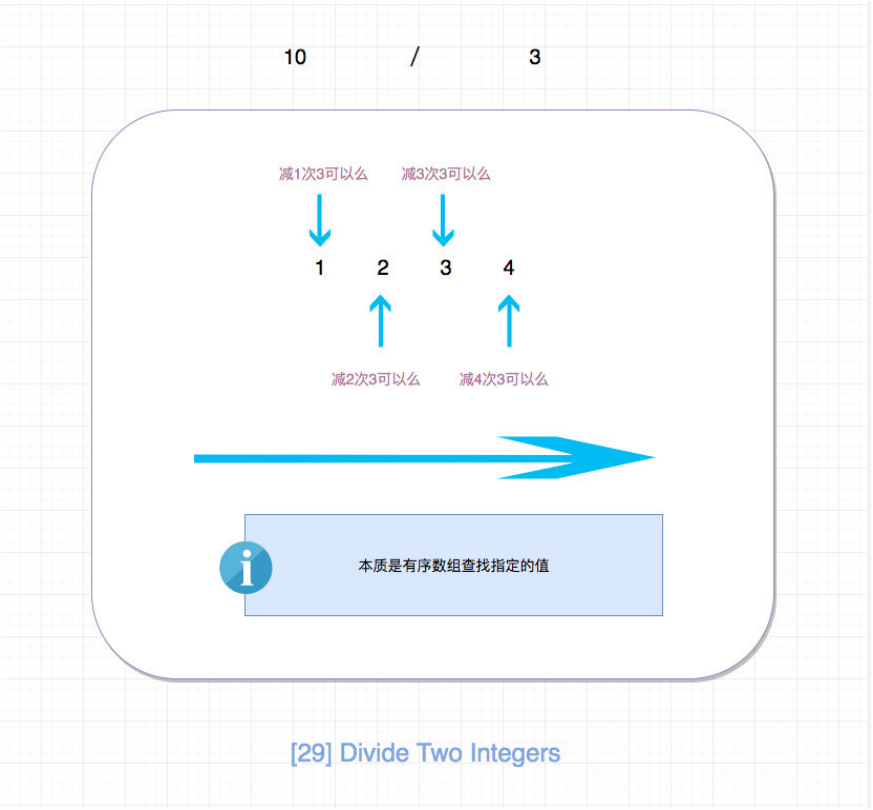
思路

符合直觉的做法是，减数一次一次减去被减数，不断更新差，直到差小于0，我们减了多少次，结果就是多少。

核心代码：

```
let acc = divisor;
let count = 0;

while (dividend - acc >= 0) {
  acc += divisor;
  count++;
}

return count;
```

这种做法简单直观，但是性能却比较差. 下面来介绍一种性能更好的方法。



通过上面这样的分析，我们直到可以使用二分法来解决，性能有很大的提升。

关键点解析

- 二分查找
- 正负数的判断中，这样判断更简单。

```
const isNegative = dividend > 0 !== divisor > 0;
```

或者利用异或：

```
const isNegative = dividend ^ (divisor < 0);
```

代码

- 语言支持：JS, Python3, CPP

```
/*
 * @lc app=leetcode id=29 lang=javascript
 *
 * [29] Divide Two Integers
 */
/**
 * @param {number} dividend
 * @param {number} divisor
 * @return {number}
 */
var divide = function (dividend, divisor) {
  if (divisor === 1) return dividend;

  // 这种方法很巧妙，即符号相同则为正，不同则为负
  const isNegative = dividend > 0 !== divisor > 0;

  const MAX_INTERGER = Math.pow(2, 31);

  const res = helper(Math.abs(dividend), Math.abs(divisor))

  // overflow
  if (res > MAX_INTERGER - 1 || res < -1 * MAX_INTERGER) {
    return MAX_INTERGER - 1;
  }

  return isNegative ? -1 * res : res;
};

function helper(dividend, divisor) {
  // 二分法
  if (dividend <= 0) return 0;
  if (dividend < divisor) return 0;
  if (divisor === 1) return dividend;

  let acc = 2 * divisor;
  let count = 1;

  while (dividend - acc > 0) {
    acc += acc;
    count += count;
  }
  // 直接使用位移运算，比如acc >> 1会有问题
  const last = dividend - Math.floor(acc / 2);

  return count + helper(last, divisor);
}
```

Python3 Code:

```
class Solution:
    def divide(self, dividend: int, divisor: int) -> int:
        """
        二分法
        :param int divisor
        :param int dividend
        :return int
        """
        # 错误处理
        if divisor == 0:
            raise ZeroDivisionError
        if abs(divisor) == 1:
            result = dividend if 1 == divisor else -divider
            return min(2**31-1, max(-2**31, result))

        # 确定结果的符号
        sign = ((dividend >= 0) == (divisor >= 0))

        result = 0
        # abs也可以直接写在while条件中，不过可能会多计算几次
        _divisor = abs(divisor)
        _dividend = abs(dividend)

        while _divisor <= _dividend:
            r, _dividend = self._multi_divide(_divisor, _d
            result += r

        result = result if sign else -result

        # 注意返回值不能超过32位有符号数的表示范围
        return min(2**31-1, max(-2**31, result))

    def _multi_divide(self, divisor, dividend):
        """
        翻倍除法，如果可以被除，则下一步除数翻倍，直至除数大于被除数，
        返回商加总的结果与被除数的剩余值；
        这里就不做异常处理了；
        :param int divisor
        :param int dividend
        :return tuple result, left_dividend
        """
        result = 0
        times_count = 1
        while divisor <= dividend:
            dividend -= divisor
            result += times_count
            times_count += times_count
```

```
            divisor += divisor
        return result, dividend
```

CPP Code:

```cpp
class Solution {
public:
    int divide(int dividend, int divisor) {
        if (!divisor) return 0;  // divide-by-zero error
        bool pos1 = dividend > 0, pos2 = divisor > 0, pos =
        if (pos1) dividend = -dividend;
        if (pos2) divisor = -divisor;
        int q = 0, d = divisor, t = 1;
        while (t > 0 && dividend < 0) {
            if (dividend - d <= 0) {
                dividend -= d;
                q -= t;
                if ((INT_MIN >> 1) < d) {
                    t <<= 1;
                    d <<= 1;
                }
            } else {
                d >>= 1;
                t >>= 1;
            }
        }
        return pos? -q : q;
    }
};
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

相关题目

- 875.koko-eating-bananas

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(31. 下一个排列)

https://leetcode-cn.com/problems/next-permutation/

题目描述

> 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
>
> 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`

前置知识

- 回溯法

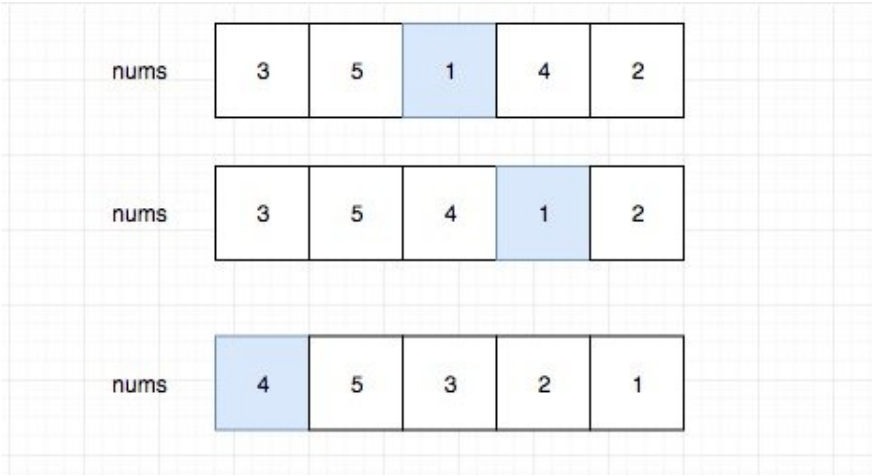
公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的方法是按顺序求出所有的排列，如果当前排列等于 nums，那么我直接取下一个但是这种做法不符合 constant space 要求（题目要求直接修改原数组）,时间复杂度也太高，为 O(n!),肯定不是合适的解。

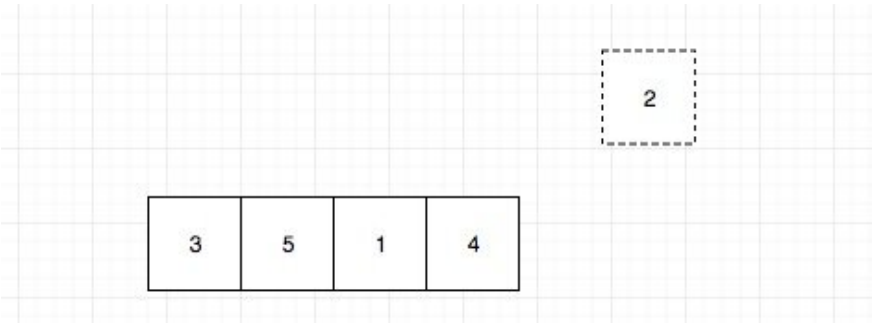
这种题目比较抽象，写几个例子通常会帮助理解问题的规律。我找了几个例子，其中蓝色背景表示的是当前数字找下一个更大排列的时候`需要改变的元素`.



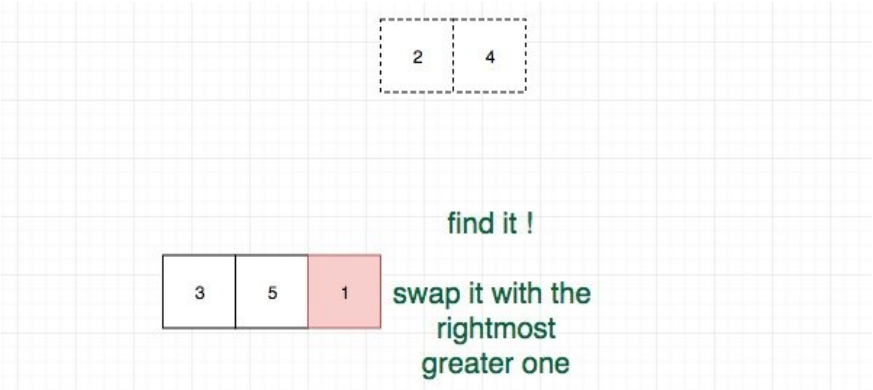
我们不难发现，蓝色的数字都是从后往前第一个不递增的元素，并且我们的下一个更大的排列 只需要改变蓝色的以及之后部分即可，前面的不需要变。

那么怎么改变蓝色的以及后面部分呢？为了使增量最小， 由于前面我们观察发现，其实剩下的元素从左到右是递减的，而我们想要变成递增的，我们只需要不断交换首尾元素即可。

另外我们也可以以回溯的角度来思考这个问题，让我们先回溯一次：

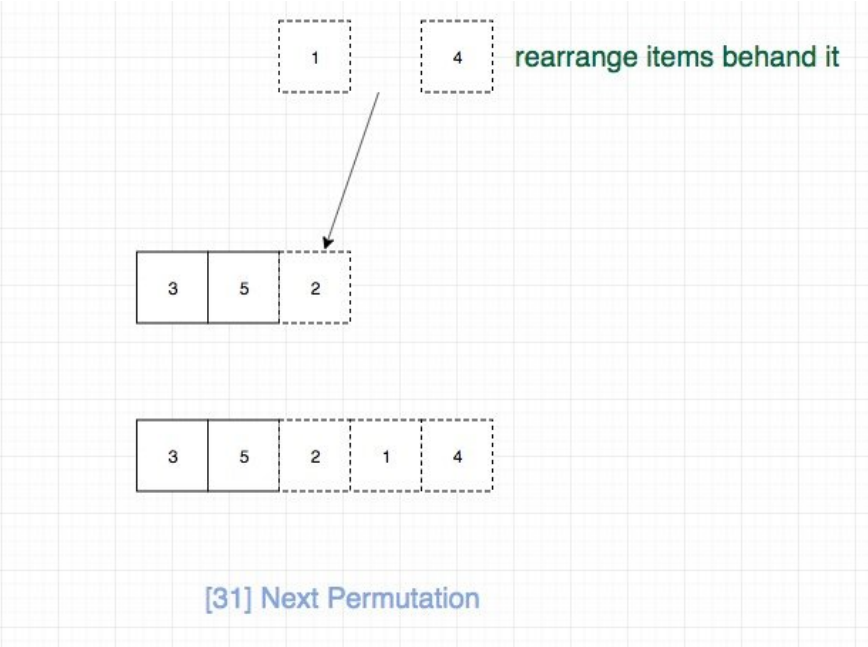


这个时候可以选择的元素只有 2，我们无法组成更大的排列，我们继续回溯，直到如图：



我们发现我们可以交换 4 或者 2 实现变大的效果，但是要保证变大的幅度最小（下一个更大）， 我们需要选择最小的，由于之前我们发现后面是从左到右递减的，显然就是交换最右面大于 1 的。

之后就是不断交换使之幅度最小：



关键点解析

- 写几个例子通常会帮助理解问题的规律
- 在有序数组中首尾指针不断交换位置即可实现 reverse
- 找到从右边起 `第一个大于nums[i]的` ，并将其和 nums[i]进行交换

代码

语言支持: Javascript, Python3, CPP

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=31 lang=javascript
 *
 * [31] Next Permutation
 */

function reverseRange(A, i, j) {
  while (i < j) {
    const temp = A[i];
    A[i] = A[j];
    A[j] = temp;
    i++;
    j--;
  }
}
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-p
 */
var nextPermutation = function (nums) {
  // 时间复杂度O(n) 空间复杂度O(1)
  if (nums == null || nums.length <= 1) return;

  let i = nums.length - 2;
  // 从后往前找到第一个降序的,相当于找到了我们的回溯点
  while (i > -1 && nums[i + 1] <= nums[i]) i--;

  // 如果找了就swap
  if (i > -1) {
    let j = nums.length - 1;
    // 找到从右边起第一个大于nums[i]的，并将其和nums[i]进行交换
    // 因为如果交换的数字比nums[i]还要小肯定不符合题意
    while (nums[j] <= nums[i]) j--;
    const temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
  }

  // 最后我们只需要将剩下的元素从左到右，依次填入当前最小的元素就可以
  // [i + 1, A.length -1]的元素进行反转

  reverseRange(nums, i + 1, nums.length - 1);
};
```

Python3 Code:

```python
class Solution:
    def nextPermutation(self, nums):
        """
        Do not return anything, modify nums in-place instea
        :param list nums
        """
        # 第一步，从后往前，找到下降点
        down_index = None
        for i in range(len(nums)-2, -1, -1):
            if nums[i] < nums[i+1]:
                down_index = i
                break
        # 如果没有下降点，重新排列
        if down_index is None:
            nums.reverse()
        # 如果有下降点
        else:
            # 第二步，从后往前，找到比下降点大的数，对换位置
            for i in range(len(nums)-1, i, -1):
                if nums[down_index] < nums[i]:
                    nums[down_index], nums[i] = nums[i], nu
                    break
            # 第三部，重新排列下降点之后的数
            i, j = down_index+1, len(nums)-1
            while i < j:
                nums[i], nums[j] = nums[j], nums[i]
                i += 1
                j -= 1
```

CPP Code:

```cpp
class Solution {
public:
  void nextPermutation(vector<int>& nums) {
    int i = nums.size() - 2, j = nums.size() - 1;
    while (i >= 0 && nums[i] >= nums[i + 1]) --i;
    if (i >= 0) {
      while (j > i && nums[j] <= nums[i]) --j;
      swap(nums[i], nums[j]);
    }
    reverse(nums.begin() + i + 1, nums.end());
  }
};
```

相关题目

- 46.next-permutation
- 47.permutations-ii
- 60.permutation-sequence(TODO)

题目地址(33. 搜索旋转排序数组)

https://leetcode-cn.com/problems/search-in-rotated-sorted-array/

题目描述

```
给你一个升序排列的整数数组 nums ，和一个整数 target 。

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,

请你在数组中搜索 target ，如果数组中存在这个目标值，则返回它的索引，否


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
示例 3：

输入：nums = [1], target = 0
输出：-1


提示：

1 <= nums.length <= 5000
-10^4 <= nums[i] <= 10^4
nums 中的每个值都 独一无二
nums 肯定会在某个点上旋转
-10^4 <= target <= 10^4
```

前置知识

- 数组
- 二分法

公司

- 阿里

- 腾讯
- 百度
- 字节

思路

这是一个我在网上看到的前端头条技术终面的一个算法题。

题目要求时间复杂度为 logn，因此基本就是二分法了。 这道题目不是直接的有序数组，不然就是 easy 了。

首先要知道，我们随便选择一个点，将数组分为前后两部分，其中一部分一定是有序的。

具体步骤：

- 我们可以先找出 mid，然后根据 mid 来判断，mid 是在有序的部分还是无序的部分

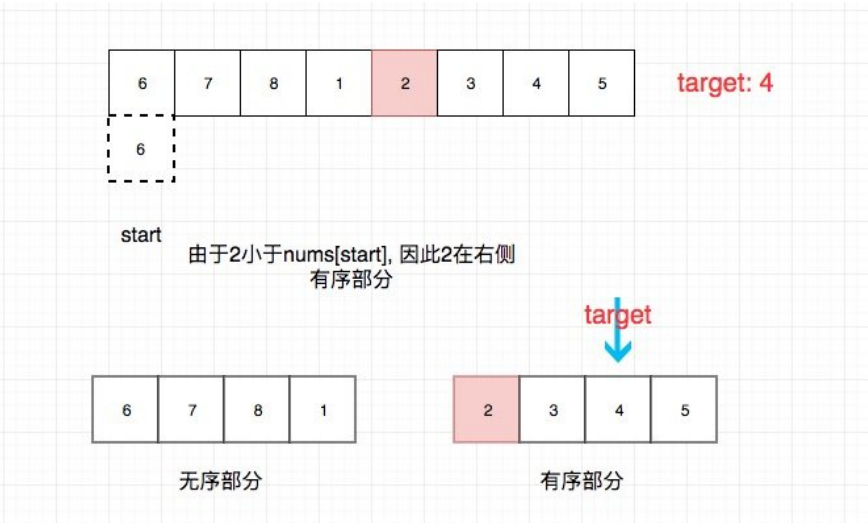
假如 mid 小于 start，则 mid 一定在右边有序部分。 假如 mid 大于等于 start， 则 mid 一定在左边有序部分。

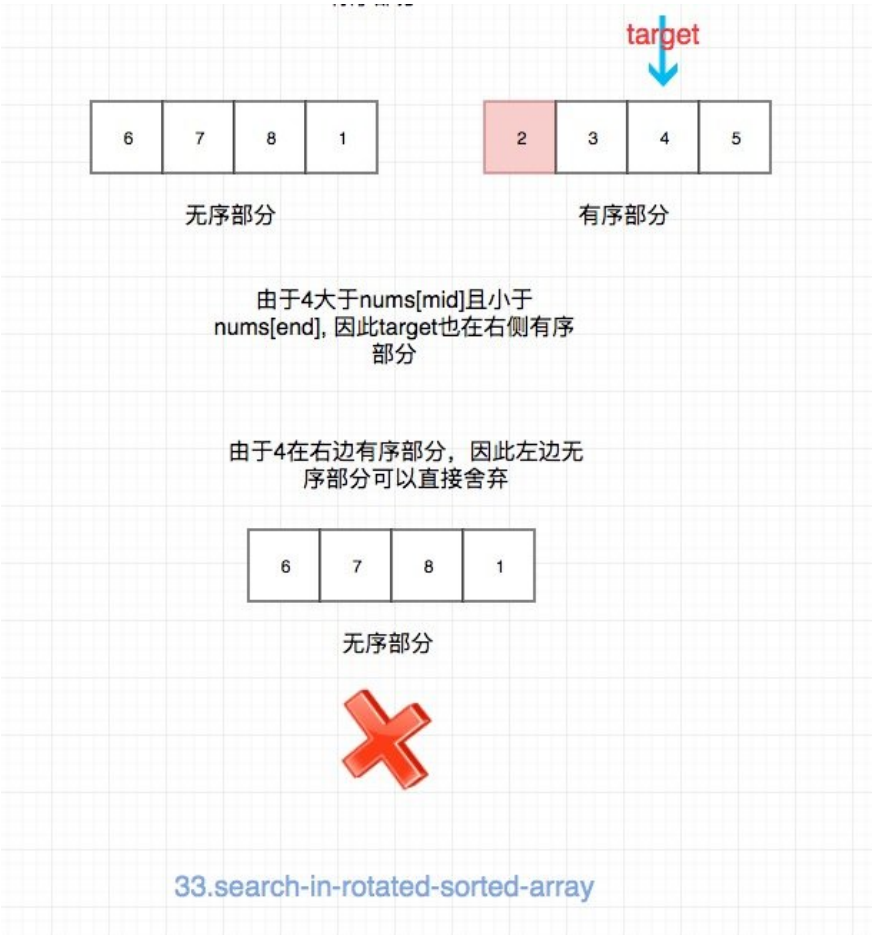
> 注意等号的考虑

- 然后我们继续判断 target 在哪一部分， 我们就可以舍弃另一部分了

我们只需要比较 target 和有序部分的边界关系就行了。 比如 mid 在右侧有序部分，即[mid, end] 那么我们只需要判断 target >= mid && target <= end 就能知道 target 在右侧有序部分，我们就 可以舍弃左边部分了(start = mid + 1)， 反之亦然。

我们以([6,7,8,1,2,3,4,5], 4)为例讲解一下：





关键点解析

- 二分法
- 找出有序区间，然后根据 target 是否在有序区间舍弃一半元素

代码

- 语言支持: Javascript，Python3

```
/*
 * @lc app=leetcode id=33 lang=javascript
 *
 * [33] Search in Rotated Sorted Array
 */
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var search = function (nums, target) {
  // 时间复杂度：O(logn)
  // 空间复杂度：O(1)
  // [6,7,8,1,2,3,4,5]
  let start = 0;
  let end = nums.length - 1;

  while (start <= end) {
    const mid = start + ((end - start) >> 1);
    if (nums[mid] === target) return mid;

    // [start, mid]有序

    // ⚠注意这里的等号
    if (nums[mid] >= nums[start]) {
      //target 在 [start, mid] 之间

      // 其实target不可能等于nums[mid]， 但是为了对称，我还是加上
      if (target >= nums[start] && target <= nums[mid]) {
        end = mid - 1;
      } else {
        //target 不在 [start, mid] 之间
        start = mid + 1;
      }
    } else {
      // [mid, end]有序

      // target 在 [mid, end] 之间
      if (target >= nums[mid] && target <= nums[end]) {
        start = mid + 1;
      } else {
        // target 不在 [mid, end] 之间
        end = mid - 1;
      }
    }
  }
```

```
    return -1;
};
```

Python3 Code:

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        """用二分法，先判断左右两边哪一边是有序的，再判断是否在有序的
        if len(nums) <= 0:
            return -1

        left = 0
        right = len(nums) - 1
        while left < right:
            mid = (right - left) // 2 + left
            if nums[mid] == target:
                return mid

            # 如果中间的值大于最左边的值，说明左边有序
            if nums[mid] > nums[left]:
                if nums[left] <= target <= nums[mid]:
                    right = mid
                else:
                    # 这里 +1，因为上面是 <= 符号
                    left = mid + 1
            # 否则右边有序
            else:
                # 注意：这里必须是 mid+1，因为根据我们的比较方式，
                if nums[mid+1] <= target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid

        return left if nums[left] == target else -1
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(39. 组合总和)

https://leetcode-cn.com/problems/combination-sum/

题目描述

```
给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 car

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]
]


提示：

1 <= candidates.length <= 30
1 <= candidates[i] <= 200
candidate 中的每个元素都是独一无二的。
1 <= target <= 500
```

前置知识

- 回溯法

公司

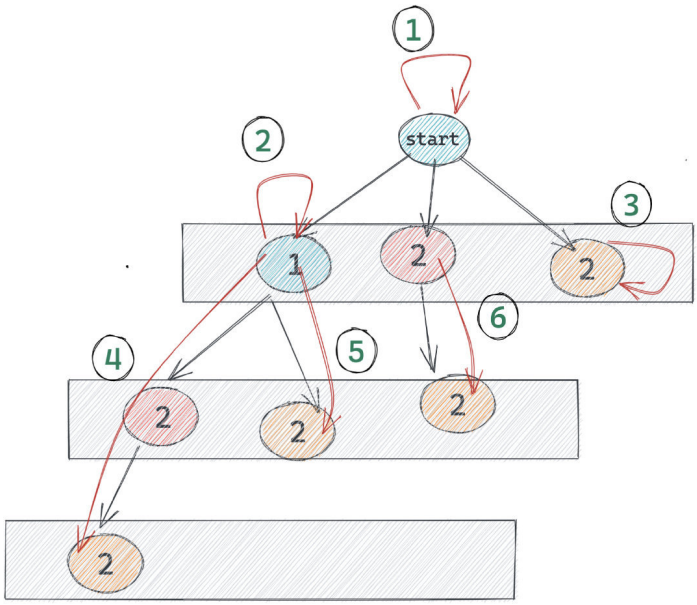
- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。网上也有大神给出了这种回溯法解题的通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：



backtrack @lucifer

> 每一层灰色的部分，表示当前有哪些节点是可以选择的， 红色部分则是选择路径。1，2，3，4，5，6 则分别表示我们的 6 个子集。
>
> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript，Python3,CPP

JS Code:

```
function backtrack(list, tempList, nums, remain, start) {
  if (remain < 0) return;
  else if (remain === 0) return list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, remain - nums[i], i);
    tempList.pop();
  }
}
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum = function (candidates, target) {
  const list = [];
  backtrack(
    list,
    [],
    candidates.sort((a, b) => a - b),
    target,
    0
  );
  return list;
};
```

Python3 Code:

```python
class Solution:
    def combinationSum(self, candidates: List[int], target:
        """
        回溯法，层层递减，得到符合条件的路径就加入结果集中，超出则剪枝
        主要是要注意一些细节，避免重复等；
        """
        size = len(candidates)
        if size <= 0:
            return []

        # 先排序，便于后面剪枝
        candidates.sort()

        path = []
        res = []
        self._find_path(target, path, res, candidates, 0, s

        return res

    def _find_path(self, target, path, res, candidates, beg
        """沿着路径往下走"""
        if target == 0:
            res.append(path.copy())
        else:
            for i in range(begin, size):
                left_num = target - candidates[i]
                # 如果剩余值为负数，说明超过了，剪枝
                if left_num < 0:
                    break
                # 否则把当前值加入路径
                path.append(candidates[i])
                # 为避免重复解，我们把比当前值小的参数也从下一次寻找
                # 因为根据他们得出的解一定在之前就找到过了
                self._find_path(left_num, path, res, candid
                # 记得把当前值移出路径，才能进入下一个值的路径
                path.pop()
```

CPP Code:

```
class Solution {
private:
    vector<vector<int>> res;
    void dfs(vector<int> &c, int t, int start, vector<int>
        if (!t) {
            res.push_back(v);
            return;
        }
        for (int i = start; i < c.size() && t >= c[i]; ++i
            v.push_back(c[i]);
            dfs(c, t - c[i], i, v);
            v.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum(vector<int>& candida
        sort(candidates.begin(), candidates.end());
        vector<int> v;
        dfs(candidates, target, 0, v);
        return res;
    }
};
```

相关题目

- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址(40. 组合总和 II)

https://leetcode-cn.com/problems/combination-sum-ii/

题目描述

```
给定一个数组 candidates 和一个目标数 target ，找出 candidates 中

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]
]
```

前置知识

- 回溯法

公司

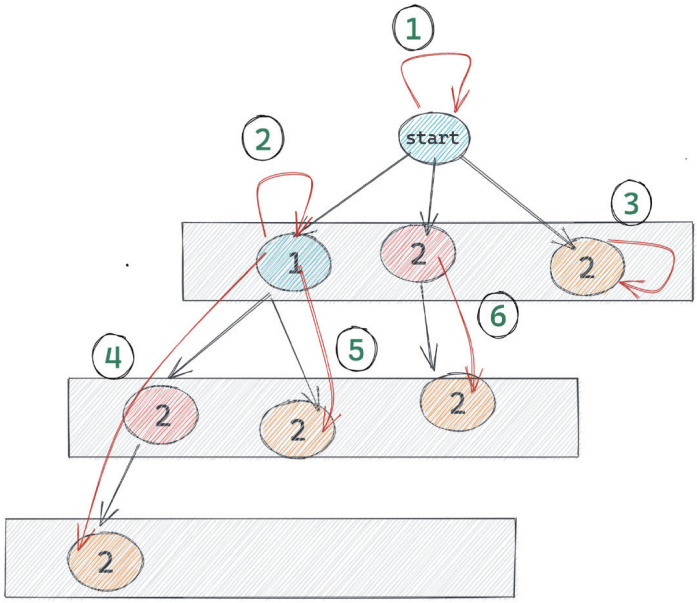
- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求极值` ，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。网上也有大神给出了这种回溯法解题的通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：



backtrack @lucifer

> 每一层灰色的部分，表示当前有哪些节点是可以选择的， 红色部分则是选择路径。1，2，3，4，5，6 则分别表示我们的 6 个子集。
>
> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

对于一个数组 [1,1,3]，任选其中两项，其组合有 3 种。分别是 (1,3), (1,1) 和 (1,3)。实际上，我们可以将两个 (1,3) 看成一样的（部分题目不能看成一样的，但本题必须看成一样的）。我们可以排序的方式进行剪枝处理。即先对数组进行一次排序，不妨进行一次升序排序。接下来我们需要修改 backrack 函数内部。先来看出修改之前的代码：

```
 if target == 0:
    res.append(path.copy())
else:
    for i in range(begin, size):
        left_num = target - candidates[i]
        if left_num < 0:
            break
        path.append(candidates[i])
        self._find_path(candidates, path, res, left_num, i-
        path.pop()
```

这里的逻辑一句话概括其实就是 **分别尝试选择 candidates[i] 和不选择 candidates[i]**。对应上面的 [1,1,3]任选两项的例子就是：

- 选择第一个 1，不选择第二个 1，选择 3。就是 [1,3]
- 不选择第一个 1，选择第二个 1，选择 3。就是 [1,3]
- ...

那么如果将代码改为：

```
 if target == 0:
    res.append(path.copy())
else:
    for i in range(begin, size):
        # 增加下面一行代码
        if i > begin and candidates[i] == candidate[i - 1]:
        left_num = target - candidates[i]
        if left_num < 0:
            break
        path.append(candidates[i])
        self._find_path(candidates, path, res, left_num, i-
        path.pop()
```

经过这样的处理，重复的都会被消除。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript, Python3, CPP

```
function backtrack(list, tempList, nums, remain, start) {
  if (remain < 0) return;
  else if (remain === 0) return list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    // 和39.combination-sum 的其中一个区别就是这道题candidates可
    // 代码表示就是下面这一行。注意 i > start 这一条件
    if (i > start && nums[i] == nums[i - 1]) continue; // s
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, remain - nums[i], i + 1
    tempList.pop();
  }
}
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum2 = function (candidates, target) {
  const list = [];
  backtrack(
    list,
    [],
    candidates.sort((a, b) => a - b),
    target,
    0
  );
  return list;
};
```

Python3 Code:

```python
class Solution:
    def combinationSum2(self, candidates: List[int], target
        """
        与39题的区别是不能重用元素，而元素可能有重复；
        不能重用好解决，回溯的index往下一个就行；
        元素可能有重复，就让结果的去重麻烦一些；
        """
        size = len(candidates)
        if size == 0:
            return []

        # 还是先排序，主要是方便去重
        candidates.sort()

        path = []
        res = []
        self._find_path(candidates, path, res, target, 0, s

        return res

    def _find_path(self, candidates, path, res, target, beg
        if target == 0:
            res.append(path.copy())
        else:
            for i in range(begin, size):
                left_num = target - candidates[i]
                if left_num < 0:
                    break
                # 如果存在重复的元素，前一个元素已经遍历了后一个元素
                if i > begin and candidates[i] == candidate
                    continue
                path.append(candidates[i])
                # 开始的 index 往后移了一格
                self._find_path(candidates, path, res, left
                path.pop()
```

CPP Code:

```
class Solution {
    vector<vector<int>> ans;
    void backtrack(vector<int> &A, int target, int start, v
        if (!target) {
            ans.push_back(path);
            return;
        }
        for (int i = start; i < A.size() && target >= A[i];
            if (i != start && A[i] == A[i - 1]) continue;
            path.push_back(A[i]);
            dfs(A, target - A[i], i + 1, path);
            path.pop_back();
        }
    }
public:
    vector<vector<int>> combinationSum2(vector<int>& A, int
        sort(A.begin(), A.end());
        vector<int> path;
        backtrack(A, target, 0, path);
        return ans;
    }
};
```

相关题目

- 39.combination-sum
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(46. 全排列)

https://leetcode-cn.com/problems/permutations/

题目描述

```
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]
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

回溯的基本思路清参考上方的回溯专题。

以 [1,2,3] 为例，我们的逻辑是：

- 先从 [1,2,3] 选取一个数。
- 然后继续从 [1,2,3] 选取一个数，并且这个数不能是已经选取过的数。

> 如何确保这个数不能是已经选取过的数？我们可以直接在已经选取的数字中线性查找，也可以将已经选取的数字中放到 hashset 中，这样就可以在 $O(1)$ 的时间来判断是否已经被选取了，只不过需要额外的空间。

- 重复这个过程直到选取的数字个数达到了 3。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript, Python3,CPP

Javascript Code:

```
function backtrack(list, tempList, nums) {
  if (tempList.length === nums.length) return list.push([..
  for (let i = 0; i < nums.length; i++) {
    if (tempList.includes(nums[i])) continue;
    tempList.push(nums[i]);
    backtrack(list, tempList, nums);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permute = function (nums) {
  const list = [];
  backtrack(list, [], nums);
  return list;
};
```

Python3 Code:

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        """itertools库内置了这个函数"""
        import itertools
        return itertools.permutations(nums)

    def permute2(self, nums: List[int]) -> List[List[int]]:
        """自己写回溯法"""
        res = []
        def backtrack(nums, pre_list):
            if len(nums) <= 0:
                res.append(pre_list)
            else:
                for i in nums:
                    # 注意copy一份新的调用，否则无法正常循环
                    p_list = pre_list.copy()
                    p_list.append(i)
                    left_nums = nums.copy()
                    left_nums.remove(i)
                    backtrack(left_nums, p_list)
        backtrack(nums, [])
        return res

    def permute3(self, nums: List[int]) -> List[List[int]]:
        """回溯的另一种写法"""
        res = []
        length = len(nums)
        def backtrack(start=0):
            if start == length:
                # nums[:] 返回 nums 的一个副本，指向新的引用，这
                res.append(nums[:])
            for i in range(start, length):
                nums[start], nums[i] = nums[i], nums[start]
                backtrack(start+1)
                nums[start], nums[i] = nums[i], nums[start]
        backtrack()
        return res
```

CPP Code:

```
class Solution {
    vector<vector<int>> ans;
    void dfs(vector<int> &nums, int start) {
        if (start == nums.size() - 1) {
            ans.push_back(nums);
            return;
        }
        for (int i = start; i < nums.size(); ++i) {
            swap(nums[i], nums[start]);
            dfs(nums, start + 1);
            swap(nums[i], nums[start]);
        }
    }
public:
    vector<vector<int>> permute(vector<int>& nums) {
        dfs(nums, 0);
        return ans;
    }
};
```

复杂度分析 令 N 为数组长度。

- 时间复杂度：$O(N!)$
- 空间复杂度：$O(N)$

相关题目

- 31.next-permutation
- 39.combination-sum
- 40.combination-sum-ii
- 47.permutations-ii
- 60.permutation-sequence
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址(47. 全排列 II)

https://leetcode-cn.com/problems/permutations-ii/

题目描述

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]
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

回溯的基本思路清参考上方的回溯专题。

这道题和第 46 题不一样的点在于其有重复元素。比如题目给的 [1,1,2]。

如果按照 46 题的解法，那么就会有重复的排列。 回顾一下 46 题我们的逻辑。以 [1,1,2] 为例，我们的逻辑是：

- 先从 [1,1,2] 选取一个数。
- 然后继续从 [1,1,2] 选取一个数，并且这个数不能是已经选取过的数。
- 重复这个过程直到选取的数字达到了 3。

如果继续沿用上面的逻辑，那么我们是永远无法拿到全部三个数字的，因此我们的逻辑需要变化。

这里我的算法是记录每一个被选取的索引，而不是值，这就保证了**同一个数字不会被选取多次，并且可以选取所有数字了**。

不过这还是有一个问题。 仍然以 [1,1,2] 为例，如果第一次选取了第一个 1，第二次选取了第二个 1，这就产生了一个组合 [1,1,2]。 如果继续第一次选取了第二个 1，而第二次选取了第一个 1，那么又会产生组合 [1,1,2]，可以看出这两个组合是重复的。（题目认为这两种情况应该算一种）

一个解决方案是对 nums 进行一次排序，并规定如果 **i > 0 and nums[i] == nums[i - 1] and visited[i - 1]**， 则不进行选取即可。

经过这样的处理，每次实际上都是从后往前依次重复的数。仍然以上面的 [1,1,2] 为例。[1,1,2] 这个排列一定是先取的第二个 1，再取第一个 1，最后取的 2。这样可以避免重复的原因在于：如果先取的第一个 1，那么永远无法取到三个数，便形成了一个不可行解。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持: Javascript，Python3, CPP

JS Code:

```
/*
 * @lc app=leetcode id=47 lang=javascript
 *
 * [47] Permutations II
 */
function backtrack(list, nums, tempList, visited) {
  if (tempList.length === nums.length) return list.push([..
  for (let i = 0; i < nums.length; i++) {
    // 和46.permutations的区别是这道题的nums是可以重复的
    // 我们需要过滤这种情况
    if (visited[i]) continue; // 同一个数字不能用两次
    if (i > 0 && nums[i] === nums[i - 1] && visited[i - 1])

    visited[i] = true;
    tempList.push(nums[i]);
    backtrack(list, nums, tempList, visited);
    visited[i] = false;
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permuteUnique = function (nums) {
  const list = [];
  backtrack(
    list,
    nums.sort((a, b) => a - b),
    [],
    []
  );
  return list;
};
```

Python3 code:

```python
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[
        """与46题一样，当然也可以直接调用itertools的函数，然后去重
        return list(set(itertools.permutations(nums)))

    def permuteUnique(self, nums: List[int]) -> List[List[
        """自己写回溯法，与46题相比，需要去重"""
        # 排序是为了方便去重
        nums.sort()
        res = []
        def backtrack(nums, pre_list):
            if len(nums) <= 0:
                res.append(pre_list)
            else:
                for i in range(len(nums)):
                    # 同样值的数字只能使用一次
                    if i > 0 and nums[i] == nums[i-1]:
                        continue
                    p_list = pre_list.copy()
                    p_list.append(nums[i])
                    left_nums = nums.copy()
                    left_nums.pop(i)
                    backtrack(left_nums, p_list)
        backtrack(nums, [])
        return res
```

CPP Code:

```
class Solution {
private:
  vector<vector<int>> ans;
  void permute(vector<int> nums, int start) {
    if (start == nums.size() - 1) {
      ans.push_back(nums);
      return;
    }
    for (int i = start; i < nums.size(); ++i) {
      if (i != start && nums[i] == nums[start]) continue;
      swap(nums[i], nums[start]);
      permute(nums, start + 1);
    }
  }
public:
  vector<vector<int>> permuteUnique(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    permute(nums, 0);
    return ans;
  }
};
```

相关题目

- 31.next-permutation
- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 60.permutation-sequence
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址(48. 旋转图像)

https://leetcode-cn.com/problems/rotate-image/

题目描述

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

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]
]
```

示例 2:

```
给定 matrix =
[
  [ 5, 1, 9,11],
  [ 2, 4, 8,10],
  [13, 3, 6, 7],
  [15,14,12,16]
],

原地旋转输入矩阵，使其变为:
[
  [15,13, 2, 5],
  [14, 3, 4, 1],
  [12, 6, 8, 9],
  [16, 7,10,11]
]
```

前置知识

- 原地算法
- 矩阵

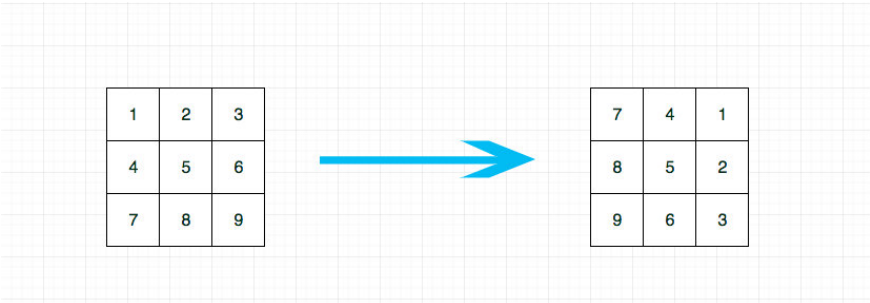
公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目让我们 in-place，也就说空间复杂度要求 O(1)，如果没有这个限制的话，很简单。

通过观察发现，我们只需要将第 i 行变成第 n - i - 1 列， 因此我们只需要保存一个原有矩阵，然后按照这个规律一个个更新即可。

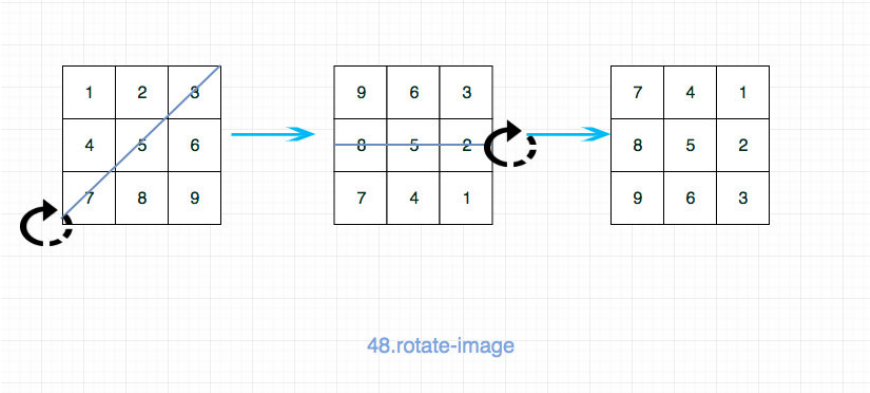


代码：

```
var rotate = function (matrix) {
  // 时间复杂度O(n^2) 空间复杂度O(n)
  const oMatrix = JSON.parse(JSON.stringify(matrix)); // c
  const n = oMatrix.length;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      matrix[j][n - i - 1] = oMatrix[i][j];
    }
  }
};
```

如果要求空间复杂度是 O(1)的话，我们可以用一个 temp 记录即可，这个时候就不能逐个遍历了。 比如遍历到 1 的时候，我们把 1 存到 temp，然后更新 1 的值为 7。 1 被换到了 3 的位置，我们再将 3 存到 temp，依次类推。 但是这种解法写起来比较麻烦，这里我就不写了。

事实上有一个更加巧妙的做法，我们可以巧妙地利用对称轴旋转达到我们的目的，如图，我们先进行一次以对角线为轴的翻转，然后 再进行一次以水平轴心线为轴的翻转即可。



这种做法的时间复杂度是 O(n^2) ， 空间复杂度是 O(1)

关键点解析

- 矩阵旋转操作

代码

- 语言支持: Javascript， Python3, CPP

```javascript
/*
 * @lc app=leetcode id=48 lang=javascript
 *
 * [48] Rotate Image
 */
/**
 * @param {number[][]} matrix
 * @return {void} Do not return anything, modify matrix in-
 */
var rotate = function (matrix) {
  // 时间复杂度O(n^2) 空间复杂度O(1)

  // 做法：  先沿着对角线翻转，然后沿着水平线翻转
  const n = matrix.length;
  function swap(arr, [i, j], [m, n]) {
    const temp = arr[i][j];
    arr[i][j] = arr[m][n];
    arr[m][n] = temp;
  }
  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - i; j++) {
      swap(matrix, [i, j], [n - j - 1, n - i - 1]);
    }
  }

  for (let i = 0; i < n / 2; i++) {
    for (let j = 0; j < n; j++) {
      swap(matrix, [i, j], [n - i - 1, j]);
    }
  }
};
```

Python3 Code:

```python
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place inst
        先做矩阵转置（即沿着对角线翻转），然后每个列表翻转；
        """
        n = len(matrix)
        for i in range(n):
            for j in range(i, n):
                matrix[i][j], matrix[j][i] = matrix[j][i],
        for m in matrix:
            m.reverse()

    def rotate2(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place inst
        通过内置函数zip，可以简单实现矩阵转置，下面的代码等于先整体翻
        不过这种写法的空间复杂度其实是O(n);
        """
        matrix[:] = map(list, zip(*matrix[::-1]))
```

CPP Code:

```cpp
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int N = matrix.size();
        for (int i = 0; i < N / 2; ++i) {
            for (int j = i; j < N - i - 1; ++j) {
                int tmp = matrix[i][j];
                matrix[i][j] = matrix[N - j - 1][i];
                matrix[N - j - 1][i] = matrix[N - i - 1][N
                matrix[N - i - 1][N - j - 1] = matrix[j][N
                matrix[j][N - i - 1] = tmp;
            }
        }
    }
};
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(49. 字母异位词分组)

https://leetcode-cn.com/problems/group-anagrams/

题目描述

```
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

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。
```

前置知识

- 哈希表
- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

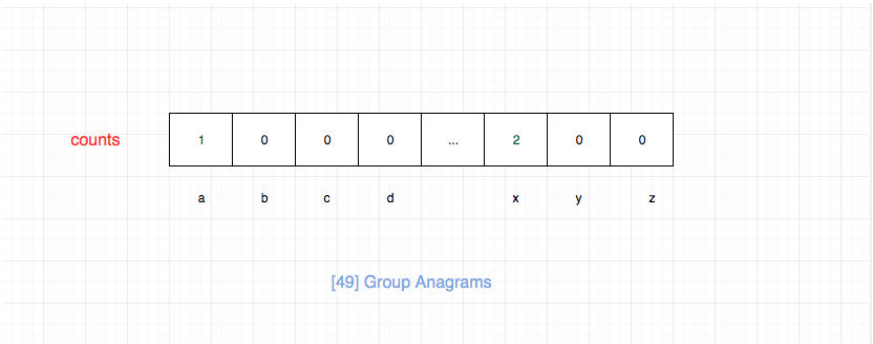
一个简单的解法就是遍历数组，然后对每一项都进行排序，然后将其添加到 hashTable 中，最后输出 hashTable 中保存的值即可。

这种做法空间复杂度 O(n)， 假设排序算法用的快排，那么时间复杂度为 O(n * klogk), n 为数组长度，k 为字符串的平均长度

代码：

```
var groupAnagrams = function (strs) {
  const hashTable = {};

  function sort(str) {
    return str.split("").sort().join("");
  }

  // 这个方法需要排序，因此不是很优，但是很直观，容易想到
  for (let i = 0; i < strs.length; i++) {
    const str = strs[i];
    const key = sort(str);
    if (!hashTable[key]) {
      hashTable[key] = [str];
    } else {
      hashTable[key].push(str);
    }
  }

  return Object.values(hashTable);
};
```

下面我们介绍另外一种方法，我们建立一个 26 长度的 counts 数组（如果区分大小写，我们可以建立 52 个，如果支持其他字符依次类推）。 然后我们给每一个字符一个固定的数组下标，然后我们只需要更新每个字符出现的次数。 最后形成的 counts 数组如果一致，则说明他们可以通过 交换顺序得到。这种算法空间复杂度 O(n), 时间复杂度 O(n * k), n 为数组长度，k 为字符串的平均长度.



实际上，这就是桶排序的基本思想。 很多题目都用到了这种思想， 读者可以留心一下。

关键点解析

- 桶排序

代码

- 语言支持: Javascript，Python3, CPP

JS Code:

```
/*
 * @lc app=leetcode id=49 lang=javascript
 *
 * [49] Group Anagrams
 */
/**
 * @param {string[]} strs
 * @return {string[][]}
 */
var groupAnagrams = function (strs) {
  // 类似桶排序

  let counts = [];
  const hashTable = {};
  for (let i = 0; i < strs.length; i++) {
    const str = strs[i];
    counts = Array(26).fill(0);
    for (let j = 0; j < str.length; j++) {
      counts[str[j].charCodeAt(0) - "a".charCodeAt(0)]++;
    }
    const key = counts.join("-");
    if (!hashTable[key]) {
      hashTable[key] = [str];
    } else {
      hashTable[key].push(str);
    }
  }

  return Object.values(hashTable);
};
```

Python3 Code:

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[s
        """
        思路同上，在Python中，这里涉及到3个知识点：
        1. 使用内置的 defaultdict 字典设置默认值；
        2. 内置的 ord 函数，计算ASCII值（等于chr）或Unicode值(等
        3. 列表不可哈希，不能作为字典的键，因此这里转为元组；
        """
        str_dict = collections.defaultdict(list)
        for s in strs:
          s_key = [0] * 26
          for c in s:
            s_key[ord(c)-ord('a')] += 1
          str_dict[tuple(s_key)].append(s)
        return list(str_dict.values())
```

CPP Code:

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& A)
        unordered_map<string, int> m;
        vector<vector<string>> ans;
        for (auto &s : A) {
            auto p = s;
            sort(p.begin(), p.end());
            if (!m.count(p)) {
                m[p] = ans.size();
                ans.push_back({});
            }
            ans[m[p]].push_back(s);
        }
        return ans;
    }
};
```

复杂度分析

其中 N 为 strs 的长度， M 为 strs 中字符串的平均长度。

- 时间复杂度：$O(N * M)$
- 空间复杂度：$O(N * M)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（50. Pow(x, n)）

https://leetcode-cn.com/problems/powx-n/description/

题目描述

```
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 = 1/22 = 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 [-231, 231 - 1] 。
```

前置知识

- 递归
- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节

解法零 - 遍历法

思路

这道题是让我们实现数学函数 `幂` ，因此直接调用系统内置函数是不被允许的。

符合直觉的做法是 `将x乘以n次` ，这种做法的时间复杂度是$O(N)$。

经实际测试，这种做法果然超时了。测试用例通过 291/304，在 `0.00001\n2147483647` 这个测试用例挂掉了。如果是面试，这个解法可以作为一种兜底解法。

代码

语言支持: Python3

Python3 Code:

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n < 0:
            return 1 / self.myPow(x, -n)
        res = 1
        for _ in range(n):
            res *= x
        return res
```

解法一 - 普通递归（超时法）

思路

首先我们要知道：

- 如果想要求 x ^ 4，那么我们可以求 (x^2)^2
- 如果是奇数，会有一点不同。 比如 x ^ 5 就等价于 x * (x^2)^2。

> 当然 x ^ 5 可以等价于 (x ^ 2) ^ 2.5, 但是这不相当于直接调用了幂函数了么。对于整数，我们可以很方便的模拟，但是小数就不方便了。

我们的思路就是：

- 将 n 地板除 2，我们不妨设结果为 a
- 那么 myPow(x, n) 就等价于 `myPow(x, a) * myPow(x, n - a)`

很可惜这种算法也会超时，原因在于重复计算会比较多，你可以试一下缓存一下计算看能不能通过。

> 如果你搞不清楚有哪些重复计算，建议画图理解一下。

代码

语言支持: Python3

Python3 Code:

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n == 1:
            return x
        if n < 0:
            return 1 / self.myPow(x, -n)
        return self.myPow(x, n // 2) * self.myPow(x, n - n
```

解法二 - 优化递归

思路

上面的解法每次直接 myPow 都会调用两次自己。我们不从缓存计算角度，而是从减少这种调用的角度来优化。

我们考虑 myPow 只调用一次自身可以么？ 没错，是可以的。

我们的思路就是：

- 如果 n 是偶数，我们将 n 折半，底数变为 x^2
- 如果 n 是奇数， 我们将 n 减去 1 ，底数不变，得到的结果再乘上底数 x

这样终于可以 AC。

代码

语言支持: Python3, CPP

Python3 Code:

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        if n == 1:
            return x
        if n < 0:
            return 1 / self.myPow(x, -n)
        return self.myPow(x * x, n // 2) if n % 2 == 0 else
```

CPP Code:

```cpp
class Solution {
    double myPow(double x, long n) {
        if (n < 0) return 1 / myPow(x, -n);
        if (n == 0) return 1;
        if (n == 1) return x;
        if (n == 2) return x * x;
        return myPow(myPow(x, n / 2), 2) * (n % 2 ? x : 1);
    }
public:
    double myPow(double x, int n) {
        return myPow(x, (long)n);
    }
};
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(logN)$

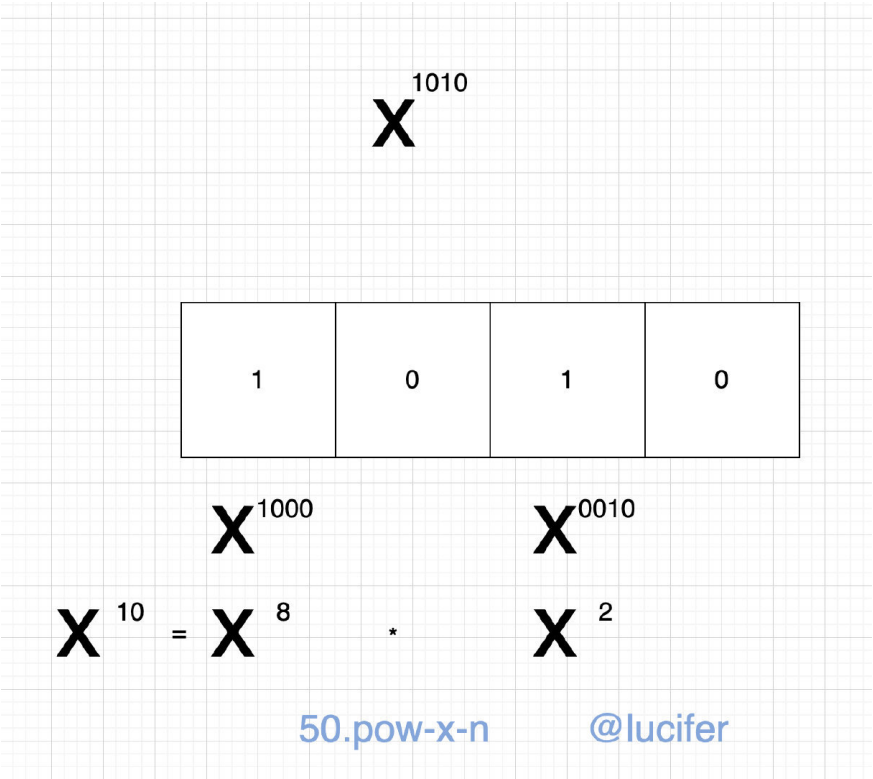
解法三 - 位运算

思路

我们来从位（bit）的角度来看一下这道题。如果你经常看我的题解和文章的话，可能知道我之前写过几次相关的“从位的角度思考分治法”，比如 LeetCode 458.可怜的小猪。

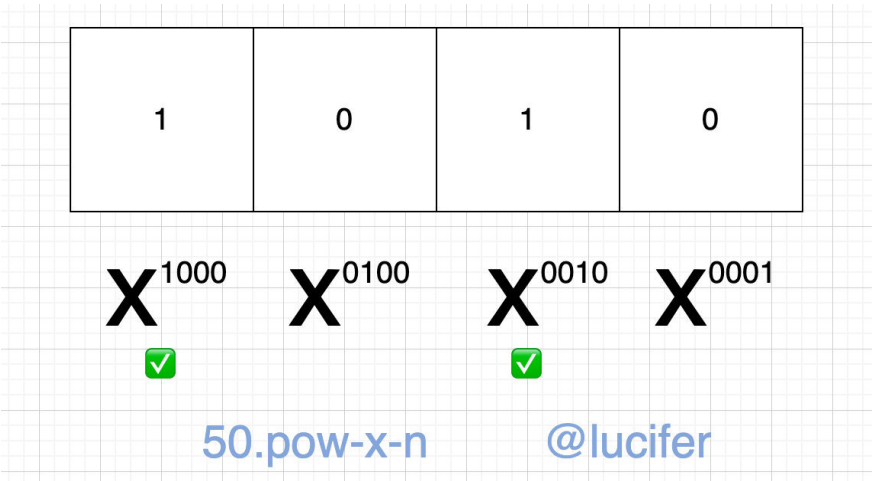
以 x 的 10 次方举例。10 的 2 进制是 1010，然后用 2 进制转 10 进制的方法把它展成 2 的幂次的和。

$$x^{10} = x^{(1010)_2} = x^{1*2^3+0*2^2+1*2^1+0*2^0} = x^{1*2^3} * x^{0*2^2} x^{1*2^1} * x^{0*2^0}$$



因此我们的算法就是：

- 不断的求解 x 的 2^0 次方，x 的 2^1 次方，x 的 2^2 次方等等。
- 将 n 转化为二进制表示
- 将 n 的二进制表示中 `1的位置` pick 出来。比如 n 的第 i 位为 1，那么就将 x^i pick 出来。
- 将 pick 出来的结果相乘



这里有两个问题：

第一个问题是 `似乎我们需要存储 x^i 以便后续相乘的时候用到` 。实际上，我们并不需要这么做。我们可以采取一次遍历的方式来完成，具体看代码。

第二个问题是，如果我们从低位到高位计算的时候，我们如何判断最高位置是否为 1？我们需要一个 bitmask 来完成，这种算法我们甚至需要借助一个额外的变量。 然而我们可以 hack 一下，直接从高位到低位进行计算，这个时候我们只需要判断最后一位是否为 1 就可以了，这个就简单了，我们直接和 1 进行一次 `与运算` 即可。

代码

语言支持: Python3

Python3 Code:

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n < 0:
            return 1 / self.myPow(x, -n)
        res = 1
        while n:
            if n & 1 == 1:
                res *= x
            x *= x
            n >>= 1
        return res
```

复杂度分析

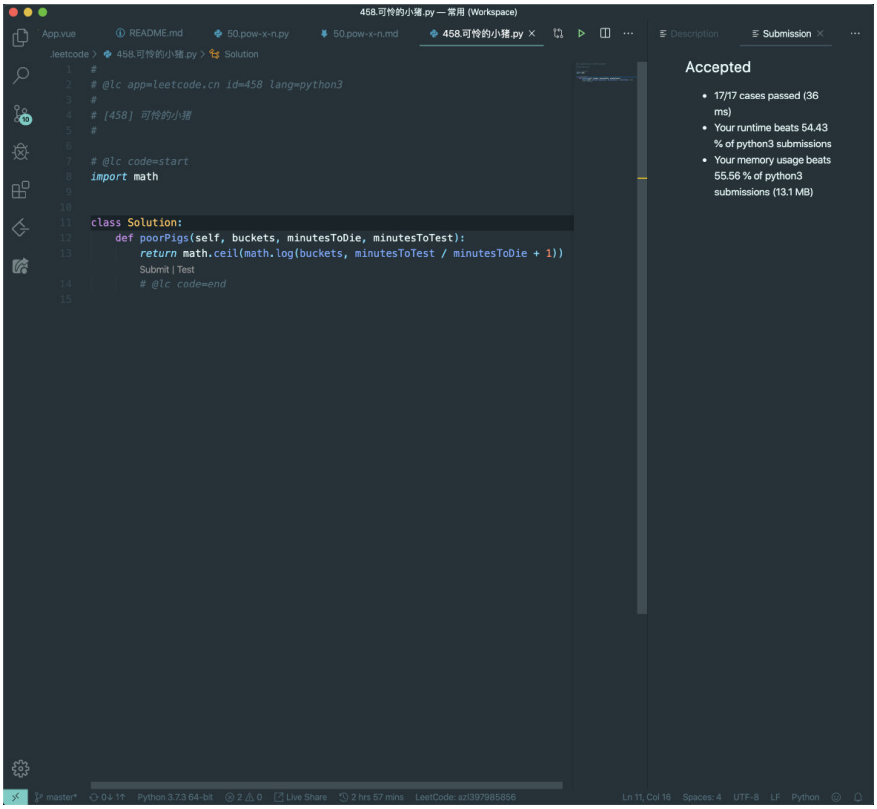
- 时间复杂度：$O(logN)$
- 空间复杂度：$O(1)$

关键点解析

- 超时分析
- hashtable
- 数学分析
- 位运算
- 二进制转十进制

相关题目

- 458.可怜的小猪



大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(55. 跳跃游戏)

https://leetcode-cn.com/problems/jump-game/

题目描述

```
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
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
```

前置知识

- 贪心

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是一道典型的 `贪心` 类型题目。思路就是用一个变量记录当前能够到达的最大的索引，并逐个遍历数组中的元素去更新这个索引，遍历完成判断这个索引是否大于 `数组长度 - 1` 即可。

我在贪心的专题中，讲到了这道题的升级版，大家可以去看一下。

关键点解析

- 记录和更新当前位置能够到达的最大的索引

代码

- 语言支持: Javascript,C++,Java,Python3

Javascript Code:

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
var canJump = function (nums) {
  let max = 0; // 能够走到的数组下标

  for (let i = 0; i < nums.length; i++) {
    if (max < i) return false; // 当前这一步都走不到，后面更走不
    max = Math.max(nums[i] + i, max);
  }

  return max >= nums.length - 1;
};
```

C++ Code:

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int n=nums.size();
        int k=0;
        for(int i=0;i<n;i++)
        {
            if(i>k){
                return false;
            }
            // 能跳到最后一个位置
            if(k>=n-1){
                return true;
            }
            // 从当前位置能跳的最远的位置
            k = max(k, i+nums[i]);
        }
        return k >= n-1;
    }
};
```

Java Code:

```
class Solution {
    public boolean canJump(int[] nums) {
        int n=nums.length;
        int k=0;
        for(int i=0;i<n;i++)
        {
            if(i>k){
                return false;
            }
            // 能跳到最后一个位置
            if(k>=n-1){
                return true;
            }
            // 从当前位置能跳的最远的位置
            k = Math.max(k, i+nums[i]);
        }
        return k >= n-1;
    }
}
```

Python3 Code:

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        """思路同上"""
        _max = 0
        _len = len(nums)
        for i in range(_len-1):
            if _max < i:
                return False
            _max = max(_max, nums[i] + i)
            # 下面这个判断可有可无，但提交的时候数据会好看点
            if _max >= _len - 1:
                return True
        return _max >= _len - 1
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(56. 合并区间)

https://leetcode-cn.com/problems/merge-intervals/

题目描述

```
给出一个区间的集合，请合并所有重叠的区间。



示例 1:

输入: intervals = [[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: intervals = [[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。
注意：输入类型已于2019年4月15日更改。 请重置默认代码定义以获取新方法签



提示：

intervals[i][0] <= intervals[i][1]
```

前置知识

- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

- 先对数组进行排序，排序的依据就是每一项的第一个元素的大小。

- 然后我们对数组进行遍历，遍历的时候两两运算（具体运算逻辑见下）
- 判断是否相交，如果不相交，则跳过
- 如果相交，则合并两项

关键点解析

- 对数组进行排序简化操作
- 如果不排序，需要借助一些 hack,这里不介绍了

代码

- 语言支持: Javascript，Python3

```javascript
/*
 * @lc app=leetcode id=56 lang=javascript
 *
 * [56] Merge Intervals
 */
/**
 * @param {number[][]} intervals
 * @return {number[][]}
 */

function intersected(a, b) {
  if (a[0] > b[1] || a[1] < b[0]) return false;
  return true;
}

function mergeTwo(a, b) {
  return [Math.min(a[0], b[0]), Math.max(a[1], b[1])];
}
var merge = function (intervals) {
  // 这种算法需要先排序
  intervals.sort((a, b) => a[0] - b[0]);
  for (let i = 0; i < intervals.length - 1; i++) {
    const cur = intervals[i];
    const next = intervals[i + 1];

    if (intersected(cur, next)) {
      intervals[i] = undefined;
      intervals[i + 1] = mergeTwo(cur, next);
    }
  }
  return intervals.filter((q) => q);
};
```

Python3 Code:

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[Lis
        """先排序，后合并"""
        if len(intervals) <= 1:
            return intervals

        # 排序
        def get_first(a_list):
            return a_list[0]
        intervals.sort(key=get_first)

        # 合并
        res = [intervals[0]]
        for i in range(1, len(intervals)):
            if intervals[i][0] <= res[-1][1]:
                res[-1] = [res[-1][0], max(res[-1][1], inte
            else:
                res.append(intervals[i])

        return res
```

复杂度分析

令 n 为 intervals 的长度。

- 时间复杂度：由于采用了排序，因此复杂度大概为 $O(nlogn)$
- 空间复杂度：$O(n)$

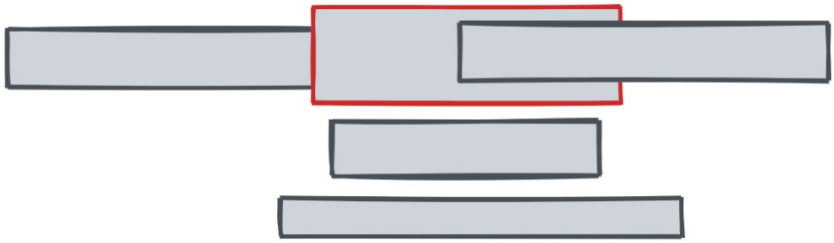
扩展

这道题是让你合并区间。这里还有一道删除区间的题目，大家可以结合起来练习 Interval-Carving。

参考 Python3 代码：

```python
class Solution:
    def solve(self, intervals, cut):
        ans = []
        for s, e in intervals:
            if s < cut[0]: ans.append([s, min(e, cut[0])])
            if cut[1] < e: ans.append([max(s, cut[1]), e])
        return ans
```

另外下面的图是我思考时候的草图，红色表示需要删除的区间，灰色是题目给的区间。



> 注意是 if 不是 else if 。 否则的话，你的判断会很多。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(60. 第 k 个排列)

https://leetcode-cn.com/problems/permutation-sequence/

题目描述

```
给出集合 [1,2,3,…,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：

"123"
"132"
"213"
"231"
"312"
"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"
```

前置知识

- 数学
- 回溯
- factorial

公司

- 阿里
- 百度
- 字节
- Twitter

思路

LeetCode 上关于排列的题目截止目前（2020-01-06）主要有三种类型：

- 生成全排列
- 生成下一个排列
- 生成第 k 个排列（我们的题目就是这种）

我们不可能求出所有的排列，然后找到第 k 个之后返回。因为排列的组合是 N！，要比 2^n 还要高很多，非常有可能超时。我们必须使用一些巧妙的方法。

我们以题目中的 n= 3 k = 3 为例：

- "123"
- "132"
- "213"
- "231"
- "312"
- "321"

可以看出 1xx，2xx 和 3xx 都有两个。如果你了解阶乘的话，应该知道这实际上是 2！个。

以上面的例子为例，假设我们想要找的是第 3 个。那么我们可以**直接跳到** 2 开头，因此我们知道以 1 开头的排列有 2 个，可以直接跳过，问题缩小了，我们将 2 加入到结果集的第一位，不断重复上述的逻辑，直到结果集的长度为 n 即可。

关键点解析

- 找规律
- 排列组合

代码

- 语言支持: Python3

```python
import math

class Solution:
    def getPermutation(self, n: int, k: int) -> str:
        res = ""
        candidates = [str(i) for i in range(1, n + 1)]

        while n != 0:
            facto = math.factorial(n - 1)
            # i 表示前面被我们排除的组数，也就是k所在的组的下标
            # k // facto 是不行的， 比如在 k % facto == 0的情况
            i = math.ceil(k / facto) - 1
            # 我们把candidates[i]加入到结果集，然后将其弹出candi
            res += candidates[i]
            candidates.pop(i)
            # k 缩小了 facto *  i
            k -= facto * i
            # 每次迭代我们实际上就处理了一个元素，n 减去 1，当n ==
            n -= 1
        return res
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(61. 旋转链表)

https://leetcode-cn.com/problems/rotate-list/

题目描述

```
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

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
```

快慢指针法

前置知识

- 求单链表的倒数第 N 个节点

思路一

1. 采用快慢指
2. 快指针与慢指针都以每步一个节点的速度向后遍历
3. 快指针比慢指针先走 N 步
4. 当快指针到达终点时，慢指针正好是倒数第 N 个节点

思路一代码

- 伪代码

```
快指针 = head;
慢指针 = head;
while (快指针.next) {
  if (N-- <= 0) {
    慢指针 = 慢指针.next;
  }
  快指针 = 快指针.next;
}
```

- 语言支持: JS

JS Code:

```
let slow = (fast = head);
while (fast.next) {
  if (k-- <= 0) {
    slow = slow.next;
  }
  fast = fast.next;
}
```

思路二

1. 获取单链表的倒数第 N + 1 与倒数第 N 个节点
2. 将倒数第 N + 1 个节点的 next 指向 null
3. 将链表尾节点的 next 指向 head
4. 返回倒数第 N 个节点

例如链表 A -> B -> C -> D -> E 右移 2 位，依照上述步骤为：

1. 获取节点 C 与 D
2. A -> B -> C -> null, D -> E
3. D -> E -> A -> B -> C -> nul
4. 返回节点 D

> 注意：假如链表节点长度为 len，
> 则右移 K 位与右移动 k % len 的效果是一样的
> 就像是长度为 1000 米的环形跑道，
> 你跑 1100 米与跑 100 米到达的是同一个地点

思路二代码

- 伪代码

```
获取链表的长度
k = k % 链表的长度
获取倒数第k + 1,倒数第K个节点与链表尾节点
倒数第k + 1个节点.next = null
链表尾节点.next = head
return 倒数第k个节点
```

- 语言支持: JS, JAVA, Python, CPP, Go, PHP

JS Code:

```
var rotateRight = function (head, k) {
  if (!head || !head.next) return head;
  let count = 0,
    now = head;
  while (now) {
    now = now.next;
    count++;
  }
  k = k % count;
  let slow = (fast = head);
  while (fast.next) {
    if (k-- <= 0) {
      slow = slow.next;
    }
    fast = fast.next;
  }
  fast.next = head;
  let res = slow.next;
  slow.next = null;
  return res;
};
```

JAVA Code:

```java
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head == null || head.next == null) return head;
        int count = 0;
        ListNode now = head;
        while(now != null){
            now = now.next;
            count++;
        }
        k = k % count;
        ListNode slow = head, fast = head;
        while(fast.next != null){
            if(k-- <= 0){
                slow = slow.next;
            }
            fast = fast.next;
        }
        fast.next = head;
        ListNode res = slow.next;
        slow.next = null;
        return res;
    }
}
```

Python Code:

```python
class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNo
        # 双指针
        if head:
            p1 = head
            p2 = head
            count = 1
            i = 0
            while i < k:
                if p2.next:
                    count += 1
                    p2 = p2.next
                else:
                    k = k % count
                    i = -1
                    p2 = head
                i += 1

            while p2.next:
                p1 = p1.next
                p2 = p2.next

            if p1.next:
                tmp = p1.next
            else:
                return head
            p1.next = None
            p2.next = head
            return tmp
```

CPP Code:

```
class Solution {
    int getLength(ListNode *head) {
        int len = 0;
        for (; head; head = head->next, ++len);
        return len;
    }
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head) return NULL;
        int len = getLength(head);
        k %= len;
        if (k == 0) return head;
        auto p = head, q = head;
        while (k--) q = q->next;
        while (q->next) {
            p = p->next;
            q = q->next;
        }
        auto h = p->next;
        q->next = head;
        p->next = NULL;
        return h;
    }
};
```

Go Code:

```go
/**
 * Definition for singly-linked list.
 * type ListNode struct {
 *     Val int
 *     Next *ListNode
 * }
 */
func rotateRight(head *ListNode, k int) *ListNode {
    if head == nil || head.Next == nil {
        return head
    }
    n := 0
    p := head
    for p != nil {
        n++
        p = p.Next
    }
    k = k % n
    // p 为快指针, q 为慢指针
    p = head
    q := head
    for p.Next!=nil {
        p = p.Next
        if k>0 {
            k--
        } else {
            q = q.Next
        }
    }
    // 更新指针
    p.Next = head
    head = q.Next
    q.Next = nil

    return head
}
```

PHP Code:

```php
/**
 * Definition for a singly-linked list.
 * class ListNode {
 *     public $val = 0;
 *     public $next = null;
 *     function __construct($val) { $this->val = $val; }
 * }
 */
class Solution
{

    /**
     * @param ListNode $head
     * @param Integer $k
     * @return ListNode
     */
    function rotateRight($head, $k)
    {
        if (!$head || !$head->next) return $head;

        $p = $head;
        $n = 0;
        while ($p) {
            $n++;
            $p = $p->next;
        }
        $k = $k % $n;
        $p = $q = $head; // $p 快指针; $q 慢指针
        while ($p->next) {
            $p = $p->next;
            if ($k > 0) $k--;
            else $q = $q->next;
        }
        $p->next = $head;
        $head = $q->next;
        $q->next = null;

        return $head;
    }
}
```

复杂度分析

- 时间复杂度：节点最多只遍历两遍，时间复杂度为$O(N)$
- 空间复杂度：未使用额外的空间，空间复杂度$O(1)$

题目地址(62. 不同路径)

https://leetcode-cn.com/problems/unique-paths/

题目描述

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

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

前置知识

- 排列组合
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

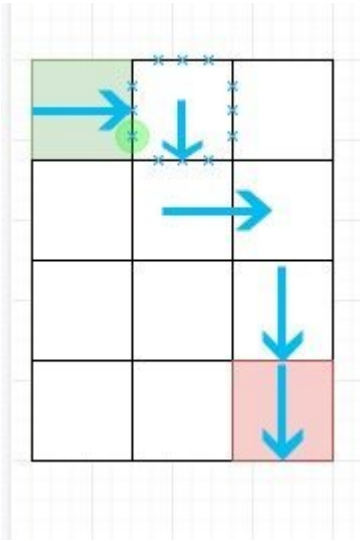
思路

首先这道题可以用排列组合的解法来解，需要一点高中的知识。

$$C_n^r = \frac{n!}{r!(n-r)!}$$

而这道题我们也可以用动态规划来解。其实这是一道典型的适合使用动态规划解决的题目，它和爬楼梯等都属于动态规划中最简单的题目，因此也经常会被用于面试之中。

读完题目你就能想到动态规划的话，建立模型并解决恐怕不是难事。其实我们很容易看出，由于机器人只能右移动和下移动， 因此第[i, j]个格子的总数应该等于[i - 1, j] + [i, j -1]， 因为第[i,j]个格子一定是从左边或者上面移动过来的。



这不就是二维平面的爬楼梯么？和爬楼梯又有什么不同呢?

代码大概是：

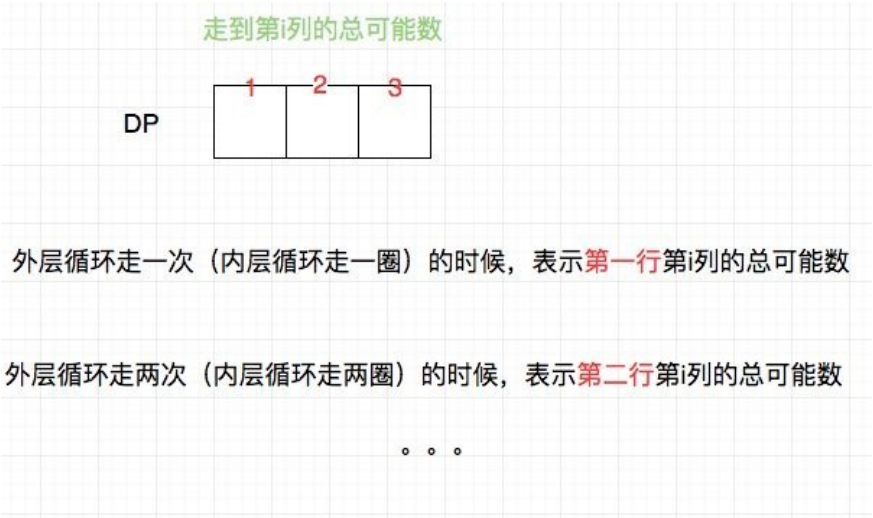
Python Code:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        d = [[1] * n for _ in range(m)]

        for col in range(1, m):
            for row in range(1, n):
                d[col][row] = d[col - 1][row] + d[col][row

        return d[m - 1][n - 1]
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(M * N)$

由于 dp[i][j] 只依赖于左边的元素和上面的元素，因此空间复杂度可以进一步优化， 优化到 O(n).



具体代码请查看代码区。

当然你也可以使用记忆化递归的方式来进行，由于递归深度的原因，性能比上面的方法差不少：

> 直接暴力递归的话可能会超时。

Python3 Code:

```
class Solution:

    @lru_cache
    def uniquePaths(self, m: int, n: int) -> int:
        if m == 1 or n == 1:
            return 1
        return self.uniquePaths(m - 1, n) + self.uniquePath
```

关键点

- 排列组合原理
- 记忆化递归
- 基本动态规划问题
- 空间复杂度可以进一步优化到 O(n), 这会是一个考点

代码

代码支持 JavaScript，Python3, CPP

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=62 lang=javascript
 *
 * [62] Unique Paths
 *
 * https://leetcode.com/problems/unique-paths/description/
 */
/**
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
var uniquePaths = function (m, n) {
  const dp = Array(n).fill(1);

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      dp[j] = dp[j] + dp[j - 1];
    }
  }

  return dp[n - 1];
};
```

Python3 Code:

```python
class Solution:

    def uniquePaths(self, m: int, n: int) -> int:
        dp = [1] * n
        for _ in range(1, m):
            for j in range(1, n):
                dp[j] += dp[j - 1]
        return dp[n - 1]
```

CPP Code:

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> dp(n + 1, 0);
        dp[n - 1] = 1;
        for (int i = m - 1; i >= 0; --i) {
            for (int j = n - 1; j >= 0; --j) dp[j] += dp[j
        }
        return dp[0];
    }
};
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(N)$

扩展

你可以做到比$O(M * N)$更快，比$O(N)$更省内存的算法么？这里有一份资料可供参考。

> 提示： 考虑数学

相关题目

- 70. 爬楼梯
- 63. 不同路径 II
- 【每日一题】- 2020-09-14 -小兔的棋盘

题目地址(73. 矩阵置零)

https://leetcode-cn.com/problems/set-matrix-zeroes/

题目描述

```
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

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
你能想出一个常数空间的解决方案吗?
```

前置知识

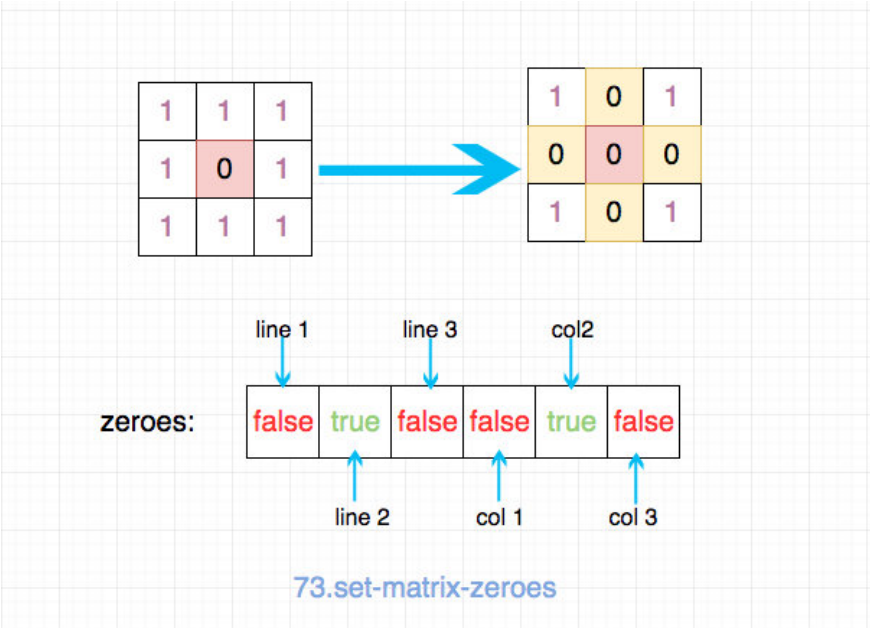
- 状态压缩

公司

- 阿里
- 百度
- 字节

思路

符合直觉的想法是，使用一个 m + n 的数组来表示每一行每一列是否”全部是 0“， 先遍历一遍去构建这样的 m + n 数组，然后根据这个 m + n 数组去修改 matrix 即可。



这样的时间复杂度 O(m * n), 空间复杂度 O(m + n).

代码如下：

```
var setZeroes = function (matrix) {
  if (matrix.length === 0) return matrix;
  const m = matrix.length;
  const n = matrix[0].length;
  const zeroes = Array(m + n).fill(false);

  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      const item = matrix[i][j];

      if (item === 0) {
        zeroes[i] = true;
        zeroes[m + j] = true;
      }
    }
  }

  for (let i = 0; i < m; i++) {
    if (zeroes[i]) {
      matrix[i] = Array(n).fill(0);
    }
  }

  for (let i = 0; i < n; i++) {
    if (zeroes[m + i]) {
      for (let j = 0; j < m; j++) {
        matrix[j][i] = 0;
      }
    }
  }

  return matrix;
};
```

但是这道题目还有一个 follow up， 要求使用 O(1)的时间复杂度。因此上述的方法就不行了。 但是我们要怎么去存取这些信息（哪一行哪一列应该全部为 0）呢?

一种思路是使用第一行第一列的数据来代替上述的 zeros 数组。 这样我们就不必借助额外的存储空间，空间复杂度自然就是 O(1)了。

由于我们不能先操作第一行和第一列， 因此我们需要记录下”第一行和第一列是否全是 0“这样的一个数据，最后根据这个信息去 修改第一行和第一列。

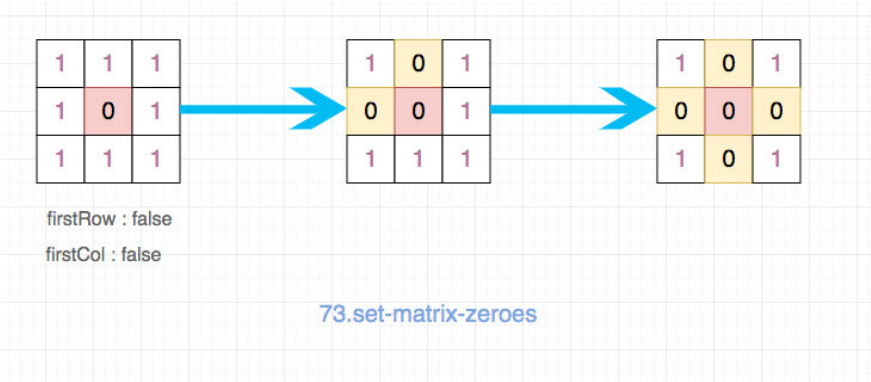
具体步骤如下：

- 记录下”第一行和第一列是否全是 0“这样的一个数据

- 遍历除了第一行和第一列之外的所有的数据，如果是 0，那就更新第一行第一列中对应的元素为 0

  > 你可以把第一行第一列看成我们上面那种解法使用的 m + n 数组。

- 根据第一行第一列的数据，更新 matrix
- 最后根据我们最开始记录的”第一行和第一列是否全是 0“去更新第一行和第一列即可



关键点

- 使用第一行和第一列来替代我们 m + n 数组
- 先记录下”第一行和第一列是否全是 0“这样的一个数据，否则会因为后续对第一行第一列的更新造成数据丢失
- 最后更新第一行第一列

代码

- 语言支持：JS，Python3

```
/*
 * @lc app=leetcode id=73 lang=javascript
 *
 * [73] Set Matrix Zeroes
 */
/**
 * @param {number[][]} matrix
 * @return {void} Do not return anything, modify matrix in-
 */
var setZeroes = function (matrix) {
  if (matrix.length === 0) return matrix;
  const m = matrix.length;
  const n = matrix[0].length;

  // 时间复杂度 O(m * n), 空间复杂度 O(1)
  let firstRow = false; // 第一行是否应该全部为0
  let firstCol = false; // 第一列是否应该全部为0

  for (let i = 0; i < m; i++) {
    for (let j = 0; j < n; j++) {
      const item = matrix[i][j];
      if (item === 0) {
        if (i === 0) {
          firstRow = true;
        }
        if (j === 0) {
          firstCol = true;
        }
        matrix[0][j] = 0;
        matrix[i][0] = 0;
      }
    }
  }

  for (let i = 1; i < m; i++) {
    for (let j = 1; j < n; j++) {
      const item = matrix[i][j];
      if (matrix[0][j] == 0 || matrix[i][0] == 0) {
        matrix[i][j] = 0;
      }
    }
  }

  // 最后处理第一行和第一列

  if (firstRow) {
    for (let i = 0; i < n; i++) {
      matrix[0][i] = 0;
```

```
    }
  }

  if (firstCol) {
    for (let i = 0; i < m; i++) {
      matrix[i][0] = 0;
    }
  }

  return matrix;
};
```

Python3 Code:

直接修改第一行和第一列为 0 的解法：

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place inst
        """
        def setRowZeros(matrix: List[List[int]], i:int) ->
            C = len(matrix[0])
            matrix[i] = [0] * C

        def setColZeros(matrix: List[List[int]], j:int) ->
            R = len(matrix)
            for i in range(R):
                matrix[i][j] = 0

        isCol = False
        R = len(matrix)
        C = len(matrix[0])

        for i in range(R):
            if matrix[i][0] == 0:
                isCol = True
            for j in range(1, C):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0
                    matrix[0][j] = 0
        for j in range(1, C):
            if matrix[0][j] == 0:
                setColZeros(matrix, j)

        for i in range(R):
            if matrix[i][0] == 0:
                setRowZeros(matrix, i)

        if isCol:
            setColZeros(matrix, 0)
```

另一种方法是用一个特殊符合标记需要改变的结果，只要这个特殊标记不在我们的题目数据范围（0 和 1）即可，这里用 None。

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        这题要解决的问题是，必须有个地方记录判断结果，但又不能影响下
        直接改为0的话，会影响下一步的判断条件；
        因此，有一种思路是先改为None，最后再将None改为0；
        从条件上看，如果可以将第一行、第二行作为记录空间，那么，用No
        """
        rows = len(matrix)
        cols = len(matrix[0])
        # 遍历矩阵，用None记录要改的地方，注意如果是0则要保留，否则
        for r in range(rows):
            for c in range(cols):
                if matrix[r][c] is not None and matrix[r][
                    # 改值
                    for i in range(rows):
                        matrix[i][c] = None if matrix[i][c]
                    for j in range(cols):
                        matrix[r][j] = None if matrix[r][j]
        # 再次遍历，将None改为0
        for r in range(rows):
            for c in range(cols):
                if matrix[r][c] is None:
                    matrix[r][c] = 0
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

扩展

为什么选择第一行第一列，选择其他行和列可以么？为什么？

题目地址(75. 颜色分类)

https://leetcode-cn.com/problems/sort-colors/

题目描述

```
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，

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
你能想出一个仅使用常数空间的一趟扫描算法吗?
```

前置知识

- 荷兰国旗问题
- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

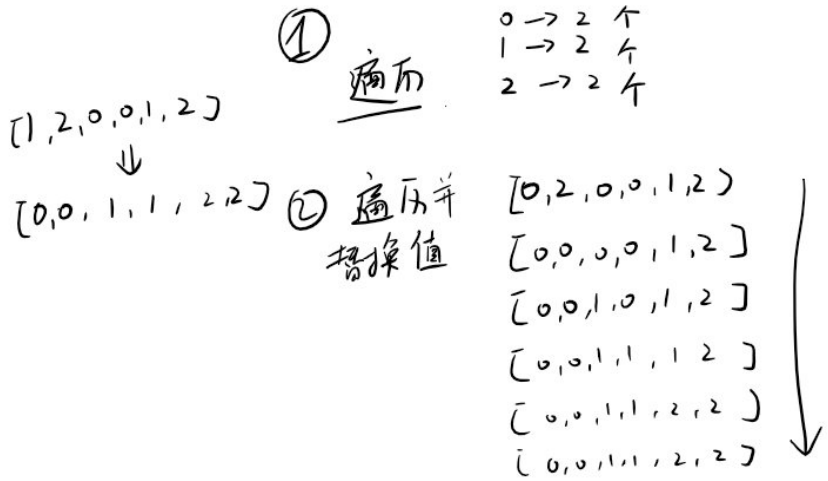
思路

这个问题是典型的荷兰国旗问题（https://en.wikipedia.org/wiki/Dutch_national_flag_problem）。 因为我们可以将红白蓝三色小球想象成条状物，有序排列后正好组成荷兰国旗。

解法一 - 计数排序

- 遍历数组，统计红白蓝三色球（0，1，2）的个数
- 根据红白蓝三色球（0，1，2）的个数重排数组

这种思路的时间复杂度：$O(n)$，需要遍历数组两次（Two pass）。



解法二 - 挡板法

我们可以把数组分成三部分，前部（全部是 0），中部（全部是 1）和后部（全部是 2）三个部分。每一个元素（红白蓝分别对应 0、1、2）必属于其中之一。

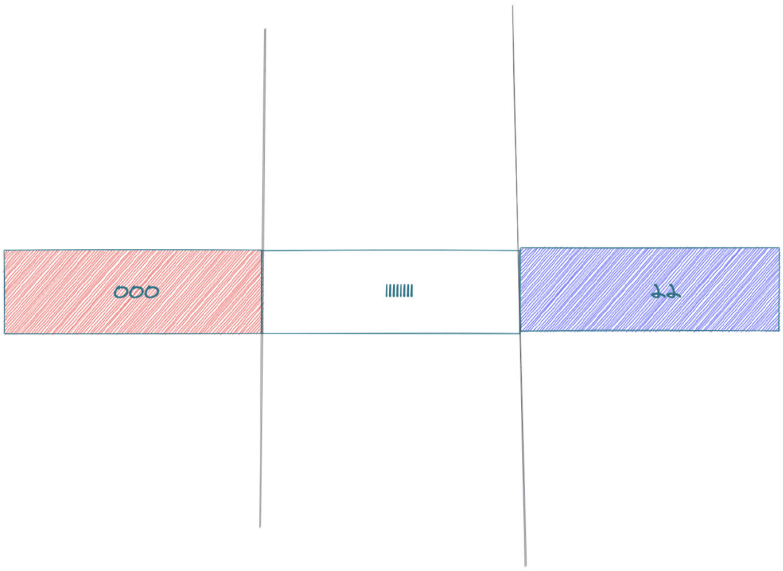
核心目标就是确定三个部分的分割点，不难知道分割点有两个。

并且我们如果将前部和后部各排在数组的前边和后边，中部自然就排好了。

具体来说，可以用三个指针。

- begin 指向前部的末尾的下一个元素（刚开始默认前部无 0，所以指向第一个位置）
- end 指向后部开头的前一个位置（刚开始默认后部无 2，所以指向最后一个位置）
- 遍历指针 current，从头开始进行遍历。

形象地来说地话就是有两个挡板，这两个挡板具体在哪事先我们不知道，我们的目标就是移动挡板到合适位置，并且使得挡板间的每一部分都是同一个的颜色。



还是以题目给的样例来说，初始化挡板位置为最左侧和最右侧：

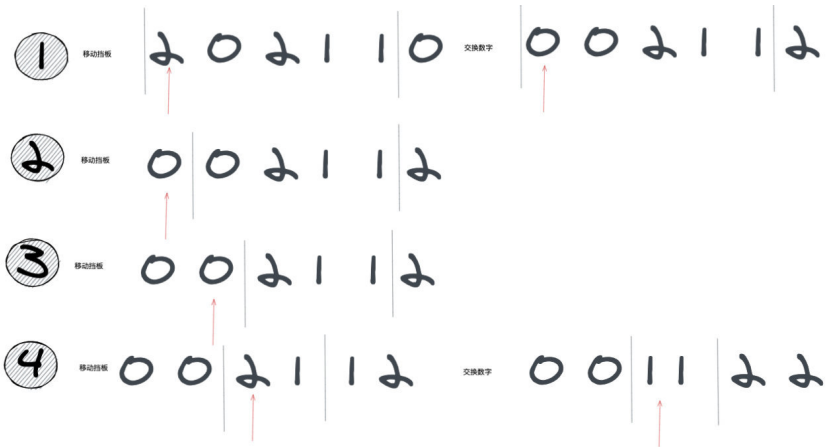
读取第一个元素是 2，它应该在右边，那么我们移动右边地挡板，使得 2 跑到挡板的右边。

> 带有背景色的圆圈 1 是第一步的意思。

并将其和移动挡板后挡板右侧地元素进行一次交换，这意味着“被移动挡板右侧元素已就位”。

。。。

整个过程大概是这样的：



。。。

这种思路的时间复杂度也是$O(n)$, 只需要遍历数组一次。空间复杂度为$O(1)，因为我们没有使用额外的空间。

关键点解析

- 荷兰国旗问题
- counting sort

代码

代码支持： Python3, CPP

Python3 Code:

```python
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instea
        """
        p0 = cur = 0
        p2 = len(nums) - 1

        while cur <= p2:
            if nums[cur] == 0:
                nums[cur], nums[p0] = nums[p0], nums[cur]
                p0 += 1
                cur += 1
            elif nums[cur] == 2:
                nums[cur], nums[p2] = nums[p2], nums[cur]
                p2 -= 1
            else:
                cur += 1
```

CPP Code:

```cpp
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int r = 0, g = 0, b = 0;
        for (int n : nums) {
            if (n == 0) {
                nums[b++] = 2;
                nums[g++] = 1;
                nums[r++] = 0;
            } else if (n == 1) {
                nums[b++] = 2;
                nums[g++] = 1;
            } else nums[b++] = 2;
        }
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 面试题 02.04. 分割链表

参考代码：

```
class Solution:
    def partition(self, head: ListNode, x: int) -> ListNode
        l1 = cur = head
        while cur:
            if cur.val < x:
                cur.val, l1.val = l1.val, cur.val
                l1 = l1.next
            cur = cur.next
        return head
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为链表长度。
- 空间复杂度：$O(1)$。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(78. 子集)

https://leetcode-cn.com/problems/subsets/

题目描述

```
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],
  []
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

回溯的基本思路清参考上方的回溯专题。

子集类题目和全排列题目不一样的点在于其需要在递归树的所有节点执行**加入结果集** 这一操作，而不像全排列需要在叶子节点执行**加入结果集**。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持：JS，C++，Java，Python

JavaScript Code:

```
function backtrack(list, tempList, nums, start) {
  list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, i + 1);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var subsets = function (nums) {
  const list = [];
  backtrack(list, [], nums, 0);
  return list;
};
```

C++ Code：

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        auto ret = vector<vector<int>>();
        auto tmp = vector<int>();
        backtrack(ret, tmp, nums, 0);
        return ret;
    }

    void backtrack(vector<vector<int>>& list, vector<int>&
        list.push_back(tempList);
        for (auto i = start; i < nums.size(); ++i) {
            tempList.push_back(nums[i]);
            backtrack(list, tempList, nums, i + 1);
            tempList.pop_back();
        }
    }
};
```

Java Code：

```
class Solution {
    // 结果
    List<List<Integer>> res = new ArrayList();
    public List<List<Integer>> subsets(int[] nums) {
        backtrack(nums, 0, new ArrayList<Integer>());
        return res;
    }

    public void backtrack(int[] nums, int start, ArrayList<
    {
        // 注意：深拷贝
        res.add(new ArrayList(track));
        for(int i=start; i<nums.length;i++)
        {
            // 做选择
            track.add(nums[i]);
            // 回溯
            backtrack(nums, i+1, track);
            // 撤销选择
            track.remove(track.size()-1);
        }
    }
}
```

python Code：

```python
class Solution:
    def subsets(self, nums):
        self.res = []
        self.track = []
        self.backtrack(nums, 0, self.track)

        return self.res

    def backtrack(self, nums, start, track):
        # 注意深拷贝
        self.res.append(list(self.track))
        for i in range(start, len(nums)):
            # 做选择
            self.track.append(nums[i])
            # 回溯
            self.backtrack(nums, i+1, self.track)
            # 撤销选择
            self.track.pop()
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(79. 单词搜索)

https://leetcode-cn.com/problems/word-search/

题目描述

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

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
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

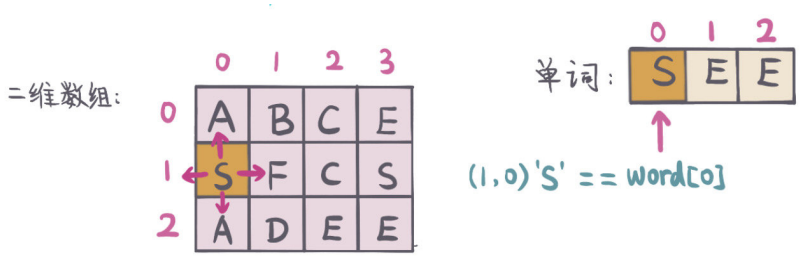
思路

在 2D 表中搜索是否有满足给定单词的字符组合，要求所有字符都是相邻的（方向不限). 题中也没有要求字符的起始和结束位置。

在起始位置不确定的情况下，扫描二维数组，找到字符跟给定单词的第一个字符相同的，四个方向（上，下，左，右）分别 DFS 搜索， 如果任意方向满足条件，则返回结果。不满足，回溯，重新搜索。

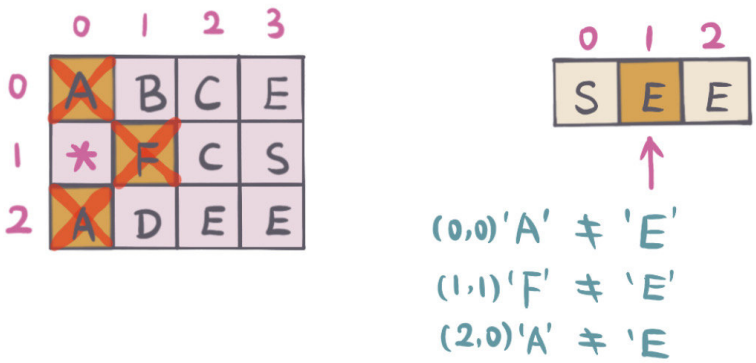
举例说明：如图二维数组，单词："SEE"

```
1. 扫描二维数组，找到board[1,0] = word[0]，匹配单词首字母。
2. 做DFS（上，下，左，右 四个方向）

如下图：
```



起始位置（1，0），判断相邻的字符是否匹配单词下一个字符 `E` .

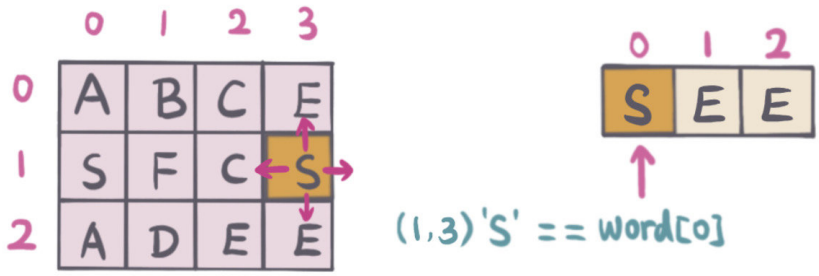
```
1. 标记当前字符（1，0）为已经访问过，board[1][0] = '*'
2. 上（0，0）字符为 'A' 不匹配，
3. 下（2，0）字符为 'A'，不匹配，
4. 左（-1，0）超越边界，不匹配，
5. 右（1，1）字符 'F'，不匹配

如下图：
```



由于从起始位置 DFS 都不满足条件，所以

1. 回溯，标记起始位置（1，0）为未访问。board[1][0] = 'S'.
2. 然后继续扫描二维数组，找到下一个起始位置（1，3）

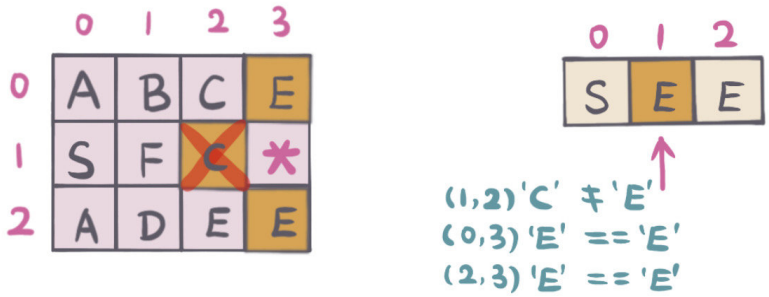
如下图：



起始位置（1，3），判断相邻的字符是否匹配单词下一个字符 `E`.

1. 标记当前字符（1，3）为已经访问过，board[1][3] = '*'
2. 上（0，3）字符为 'E'，匹配，继续DFS搜索（参考位置为（0，3）位置[
3. 下（2，3）字符为 'E'，匹配，#2匹配，先进行#2 DFS搜索，由于#2 DF
4. 左（1，2）字符为 'C'，不匹配，
5. 右（1，4）超越边界，不匹配

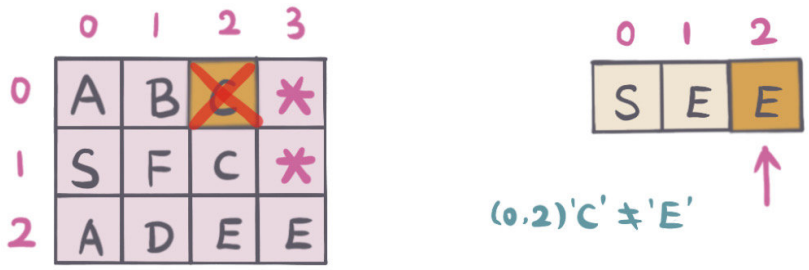
如下图：



位置（0，3）满足条件，继续 DFS，判断相邻的字符是否匹配单词下一个字符 `E`

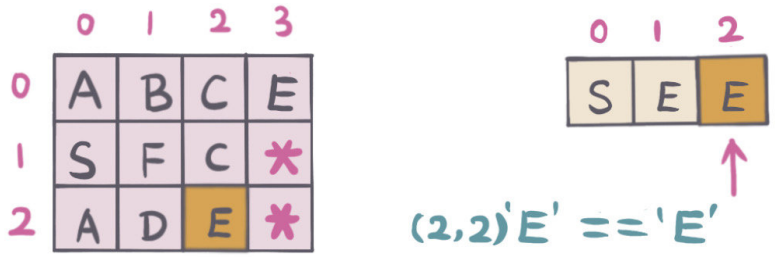
1. 标记当前字符（0，3）为已经访问过，board[0][3] = '*'
2. 上 （-1，3）超越边界，不匹配
3. 下（1，3）已经访问过，
4. 左（0，2）字符为 'C'，不匹配
5. 右（1，4）超越边界，不匹配

如下图

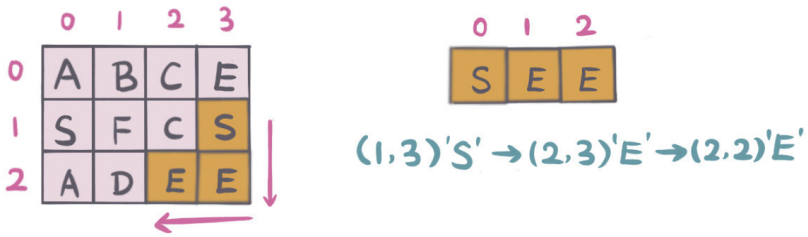


从位置（0，3）DFS 不满足条件，继续位置（2，3）DFS 搜索

```
1. 回溯，标记起始位置（0，3）为未访问。board[0][3] = 'E'.
2. 回到满足条件的位置（2，3），继续DFS搜索，判断相邻的字符是否匹配单词
3. 上（1，3）已访问过
4. 下（3，3）超越边界，不匹配
5. 左（2，2）字符为 'E'，匹配
6. 右（2，4）超越边界，不匹配

如下图:
```



单词匹配完成，满足条件，返回 `True` .



复杂度分析

- *时间复杂度：* `O(m*n) - m 是二维数组行数， n 是二维数组列数`
- *空间复杂度：* `O(1) -` 这里在原数组中标记当前访问过，没有用到额外空间

> **注意**：如果用 Set 或者是 boolean[][]来标记字符位置是否已经访问过，需要额外的空间 `O(m*n)` .

关键点分析

- 遍历二维数组的每一个点，找到起始点相同的字符，做 DFS
- DFS 过程中，要记录已经访问过的节点，防止重复遍历，这里（Java Code 中）用 `*` 表示当前已经访问过，也可以用 Set 或者是 boolean[][]数组记录访问过的节点位置。
- 是否匹配当前单词中的字符，不符合回溯，这里记得把当前 `*` 重新设为当前字符。如果用 Set 或者是 boolean[][]数组，记得把当前位置重设为没有访问过。

代码（ `Java/Javascript/Python3` ）

Java Code

```java
public class LC79WordSearch {
  public boolean exist(char[][] board, String word) {
    if (board == null || word == null) return false;
    if (word.length() == 0) return true;
    if (board.length == 0) return false;
    int rows = board.length;
    int cols = board[0].length;
    for (int r = 0; r < rows; r++) {
      for (int c = 0; c < cols; c++) {
        // scan board, start with word first character
        if (board[r][c] == word.charAt(0)) {
          if (helper(board, word, r, c, 0)) {
            return true;
          }
        }
      }
    }
    return false;
  }

  private boolean helper(char[][] board, String word, int
    // already match word all characters, return true
    if (start == word.length()) return true;
    if (!isValid(board, r, c) ||
        board[r][c] != word.charAt(start)) return false;
    // mark visited
    board[r][c] = '*';
    boolean res = helper(board, word, r - 1, c, start + 1)
        ||  helper(board, word, r + 1, c, start + 1)
        ||  helper(board, word, r, c - 1, start + 1)
        ||  helper(board, word, r, c + 1, start + 1);
    // backtracking to start position
    board[r][c] = word.charAt(start);
    return res;
  }

  private boolean isValid(char[][] board, int r, int c) {
    return r >= 0 && r < board.length && c >= 0 && c < boa
  }
}
```

Python3 Code

```python
class Solution:
    def exist(self, board: List[List[str]], word: str) -> b
        m = len(board)
        n = len(board[0])

        def dfs(board, r, c, word, index):
            if index == len(word):
                return True
            if r < 0 or r >= m or c < 0 or c >= n or board
                return False
            board[r][c] = '*'
            res = dfs(board, r - 1, c, word, index + 1) or
            board[r][c] = word[index]
            return res

        for r in range(m):
            for c in range(n):
                if board[r][c] == word[0]:
                    if dfs(board, r, c, word, 0):
                        return True
        return False
```

Javascript Code from **@lucifer**

```
/*
 * @lc app=leetcode id=79 lang=javascript
 *
 * [79] Word Search
 */
function DFS(board, row, col, rows, cols, word, cur) {
  // 边界检查
  if (row >= rows || row < 0) return false;
  if (col >= cols || col < 0) return false;

  const item = board[row][col];

  if (item !== word[cur]) return false;

  if (cur + 1 === word.length) return true;

  // 如果你用hashmap记录访问的字母，那么你需要每次backtrack的时候
  // 这里我们使用一个little trick

  board[row][col] = null;

  // 上下左右
  const res =
    DFS(board, row + 1, col, rows, cols, word, cur + 1) ||
    DFS(board, row - 1, col, rows, cols, word, cur + 1) ||
    DFS(board, row, col - 1, rows, cols, word, cur + 1) ||
    DFS(board, row, col + 1, rows, cols, word, cur + 1);

  board[row][col] = item;

  return res;
}
/**
 * @param {character[][]} board
 * @param {string} word
 * @return {boolean}
 */
var exist = function (board, word) {
  if (word.length === 0) return true;
  if (board.length === 0) return false;

  const rows = board.length;
  const cols = board[0].length;

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      const hit = DFS(board, i, j, rows, cols, word, 0);
      if (hit) return true;
```

```
      }
    }
    return false;
};
```

参考（References）

1. 回溯法 Wiki

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（80.删除排序数组中的重复项 II）

https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array-ii/

题目描述

```
给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素最多出现

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

示例 1:

给定 nums = [1,1,1,2,2,3],

函数应返回新长度 length = 5, 并且原数组的前五个元素被修改为 1, 1,

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 = [0,0,1,1,1,1,2,3,3],

函数应返回新长度 length = 7, 并且原数组的前五个元素被修改为 0, 0,

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

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]);
}
```

前置知识

- 双指针

公司

- 阿里
- 百度
- 字节

思路

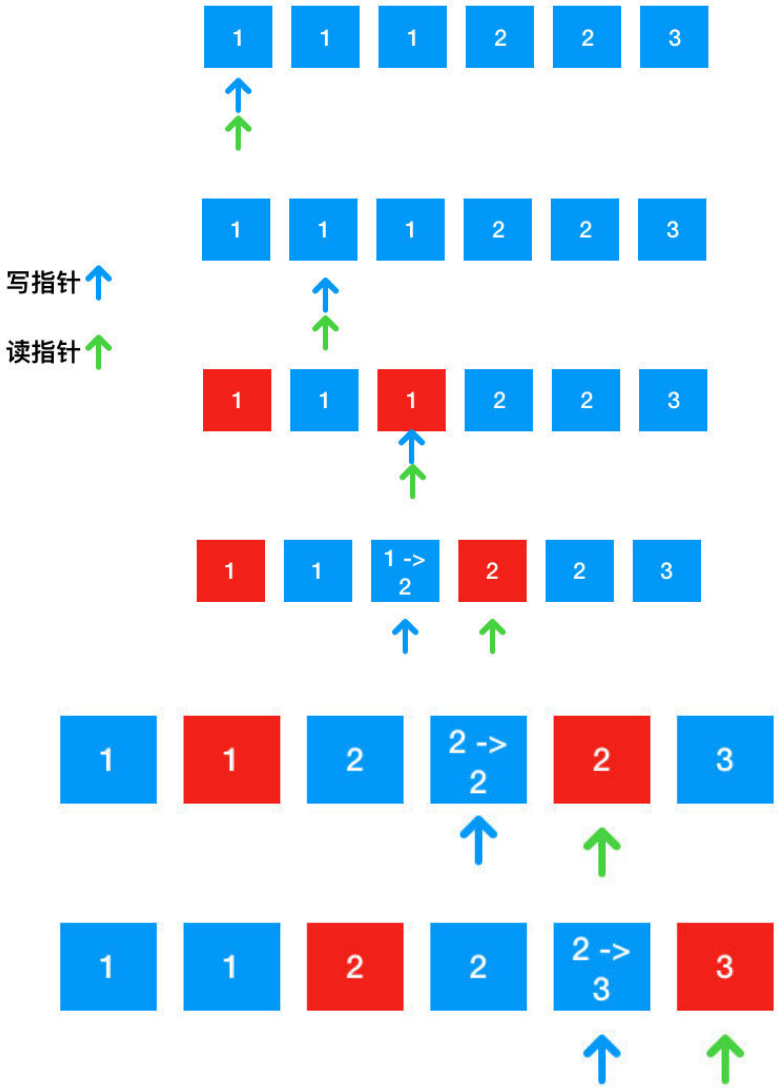
”删除排序“类题目截止到现在（2020-1-15）一共有四道题：

这道题是26.remove-duplicates-from-sorted-array 的进阶版本，唯一的不同是不再是全部元素唯一，而是全部元素不超过 2 次。实际上这种问题可以更抽象一步，即“删除排序数组中的重复项，使得相同数字最多出现 k 次” 。 那么这道题 k 就是 2， 26.remove-duplicates-from-sorted-array 的 k 就是 1。

上一题我们使用了快慢指针来实现，这道题也是一样，只不过逻辑稍有不同。 其实快慢指针本质是读写指针，在这里我们的快指针实际上就是读指针，而慢指针恰好相当于写指针。”快慢指针的说法“便于描述和记忆，“读写指针”的说法更便于理解本质。本文中，以下内容均描述为快慢指针。

- 初始化快慢指针 slow ， fast ，全部指向索引为 0 的元素。
- fast 每次移动一格
- 慢指针选择性移动，即只有写入数据之后才移动。是否写入数据取决于 slow - 2 对应的数字和 fast 对应的数字是否一致。
- 如果一致，我们不应该写。 否则我们就得到了三个相同的数字，不符合题意
- 如果不一致，我们需要将 fast 指针的数据写入到 slow 指针。
- 重复这个过程，直到 fast 走到头，说明我们已无数字可写。

图解（红色的两个数字，表示我们需要比较的两个数字）：



关键点分析

- 快慢指针
- 读写指针
- 删除排序问题

代码

代码支持： Python, CPP

Python Code：

```python
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        # 写指针
        i = 0
        K = 2
        for num in nums:
            if i < K or num != nums[i-K]:
                nums[i] = num
                i += 1
        return i
```

CPP Code:

```cpp
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int i = 0;
        int k = 2;
        for (int num : nums) {
            if (i < k || num != nums[i - k]) {
                nums[i] = num;
                i++;
            }
        }
        return i;
    }
};
```

复杂度分析

令 n 为数组长度。

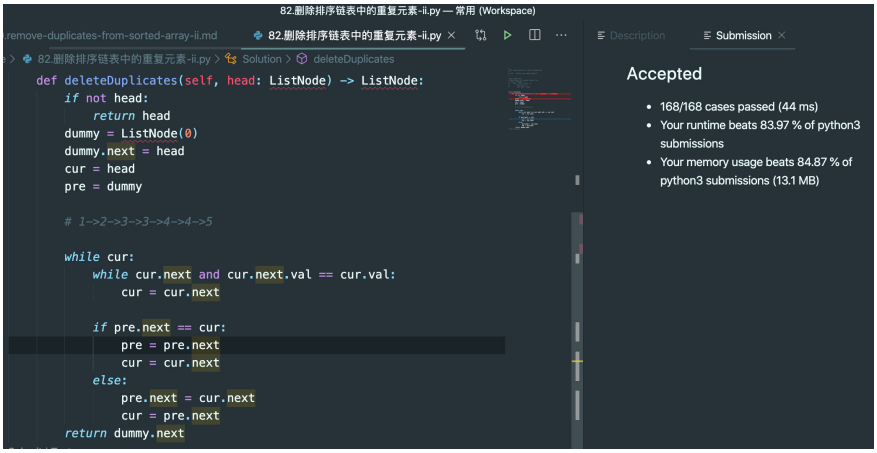
- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

基于这套代码，你可以轻易地实现 k 为任意正整数的算法。

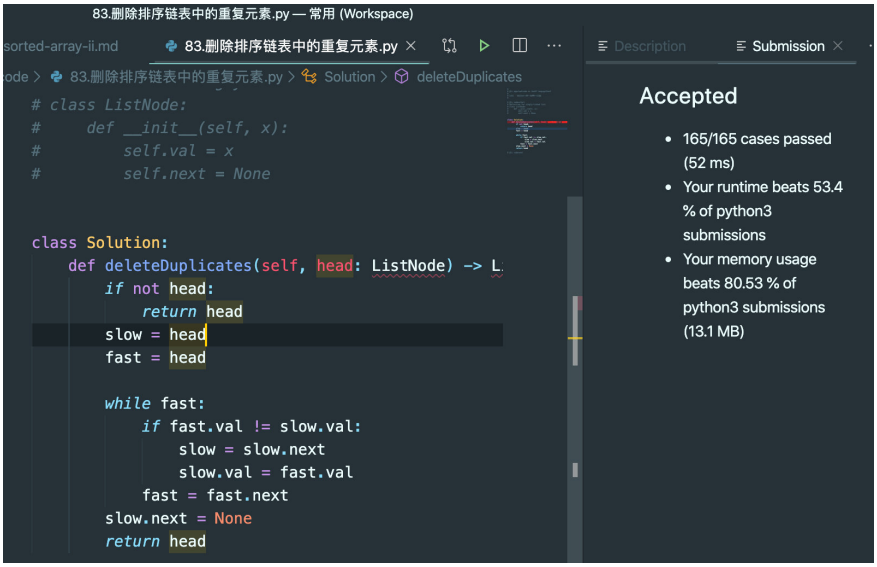
相关题目

正如上面所说，相关题目一共有三道（排除自己）。其中一道我们仓库已经讲到了。剩下两道原理类似，但是实际代码和细节有很大不同，原因就在于数组可以随机访问，而链表不行。 感兴趣的可以做一下剩下的两道链表题。

- 1. 删除排序链表中的重复元素 II



- 1. 删除排序链表中的重复元素



大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(86. 分隔链表)

https://leetcode-cn.com/problems/partition-list/

题目描述

```
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

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
```

前置知识

- 链表

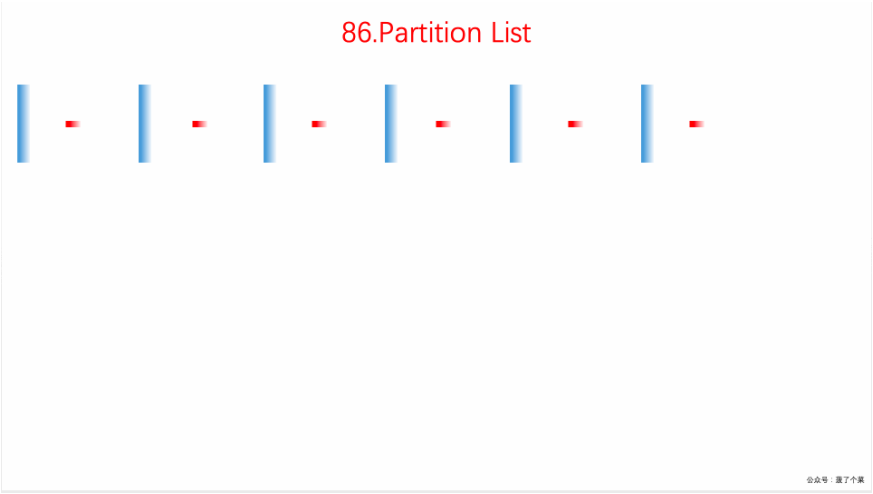
公司

- 阿里
- 腾讯
- 百度
- 字节

思路

- 设定两个虚拟节点，dummyHead1 用来保存小于该值的链表，dummyHead2 来保存大于等于该值的链表
- 遍历整个原始链表，将小于该值的放于 dummyHead1 中，其余的放置在 dummyHead2 中

遍历结束后，将 dummyHead2 插入到 dummyHead1 后面



(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

- 链表的基本操作（遍历）
- 虚拟节点 dummy 简化操作
- 遍历完成之后记得 `currentL1.next = null;` 否则会内存溢出

> 如果单纯的遍历是不需要上面操作的，但是我们的遍历会导致 currentL1.next 和 currentL2.next 中有且仅有一个不是 null， 如果不这么操作的话会导致两个链表成环，造成溢出。

代码

- 语言支持: Javascript，Python3, CPP

```
/**
 * @param {ListNode} head
 * @param {number} x
 * @return {ListNode}
 */
var partition = function (head, x) {
  const dummyHead1 = {
    next: null,
  };
  const dummyHead2 = {
    next: null,
  };

  let current = {
    next: head,
  };
  let currentL1 = dummyHead1;
  let currentL2 = dummyHead2;
  while (current.next) {
    current = current.next;
    if (current.val < x) {
      currentL1.next = current;
      currentL1 = current;
    } else {
      currentL2.next = current;
      currentL2 = current;
    }
  }

  currentL2.next = null;

  currentL1.next = dummyHead2.next;

  return dummyHead1.next;
};
```

Python3 Code:

```python
class Solution:
    def partition(self, head: ListNode, x: int) -> ListNode
        """在原链表操作，思路基本一致，只是通过指针进行区分而已"""
        # 在链表最前面设定一个初始node作为锚点，方便返回最后的结果
        first_node = ListNode(0)
        first_node.next = head
        # 设计三个指针，一个指向小于x的最后一个节点，即前后分离点
        # 一个指向当前遍历节点的前一个节点
        # 一个指向当前遍历的节点
        sep_node = first_node
        pre_node = first_node
        current_node = head

        while current_node is not None:
            if current_node.val < x:
                # 注意有可能出现前一个节点就是分离节点的情况
                if pre_node is sep_node:
                    pre_node = current_node
                    sep_node = current_node
                    current_node = current_node.next
                else:
                    # 这段次序比较烧脑
                    pre_node.next = current_node.next
                    current_node.next = sep_node.next
                    sep_node.next = current_node
                    sep_node = current_node
                    current_node = pre_node.next
            else:
                pre_node = current_node
                current_node = pre_node.next

        return first_node.next
```

CPP Code:

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode dummy, geHead, *ltTail = &dummy, *geTail =
        while (head) {
            auto p = head;
            head = head->next;
            if (p->val < x) {
                ltTail->next = p;
                ltTail = p;
            } else {
                geTail->next = p;
                geTail = p;
            }
        }
        ltTail->next = geHead.next;
        geTail->next = NULL;
        return dummy.next;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(90. 子集 II)

https://leetcode-cn.com/problems/subsets-ii/

题目描述

```
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

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],
  []
]
```

前置知识

- 回溯

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

回溯的基本思路清参考上方的回溯专题。

这道题需要求子集，因此首先我们需要在所有的节点都执行加入结果集的操作，而不是像全排列那样在叶子节点才执行。

另外一个需要注意的是本题是包含重复元素的。以题目中的 [1,2,2] 为例，第一个 2 和第二个 2 是没有区别的。也就是说交换两者的位置也仅算一种情况，而不是多个。

如果是 [1,2,2,2,2,2] 呢？如果还是以 78 题的逻辑来做会有很多重复结果，那么我们如何避免上面的重复计算？

一种可行的方式是先排序，排序之后规定一种**针对相邻且相等的情况的取数逻辑**，使得无论多少个相邻的同样数字**仅有一种取法**。

而这个取数逻辑其实很简单，那就是**i > start && nums[i] === nums[i - 1]**，其中 i 为当前遍历的索引， start 为遍历的起始索引。（大家可以结合上面的回溯专题的图来理解）

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持：JS，C++，Python3

JavaScript Code：

```
function backtrack(list, tempList, nums, start) {
  list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    // 和78.subsets的区别在于这道题nums可以有重复
    // 因此需要过滤这种情况
    if (i > start && nums[i] === nums[i - 1]) continue;
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, i + 1);
    tempList.pop();
  }
}
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var subsetsWithDup = function (nums) {
  const list = [];
  backtrack(
    list,
    [],
    nums.sort((a, b) => a - b),
    0,
    []
  );
  return list;
};
```

C++ Code：

```cpp
class Solution {
private:
    void subsetsWithDup(vector<int>& nums, size_t start, ve
        res.push_back(tmp);
        for (auto i = start; i < nums.size(); ++i) {
            if (i > start && nums[i] == nums[i - 1]) contir
            tmp.push_back(nums[i]);
            subsetsWithDup(nums, i + 1, tmp, res);
            tmp.pop_back();
        }
    }
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        auto tmp = vector<int>();
        auto res = vector<vector<int>>();
        sort(nums.begin(), nums.end());
        subsetsWithDup(nums, 0, tmp, res);
        return res;
    }
};
```

Python Code:

```python
class Solution:
    def subsetsWithDup(self, nums: List[int], sorted: bool=
        """回溯法，通过排序参数避免重复排序"""
        if not nums:
            return [[]]
        elif len(nums) == 1:
            return [[], nums]
        else:
            # 先排序，以便去重
            # 注意，这道题排序花的时间比较多
            # 因此，增加一个参数，使后续过程不用重复排序，可以大幅提
            if not sorted:
                nums.sort()
            # 回溯法
            pre_lists = self.subsetsWithDup(nums[:-1], sort
            all_lists = [i+[nums[-1]] for i in pre_lists] +
            # 去重
            result = []
            for i in all_lists:
                if i not in result:
                    result.append(i)
            return result
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 113.path-sum-ii
- 131.palindrome-partitioning

题目地址(91. 解码方法)

https://leetcode-cn.com/problems/decode-ways/

题目描述

一条包含字母 A-Z 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

给定一个只包含数字的非空字符串，请计算解码方法的总数。

题目数据保证答案肯定是一个 32 位的整数。

示例 1：

```
输入："12"
输出：2
解释：它可以解码为 "AB"（1 2）或者 "L"（12）。
```

示例 2：

```
输入："226"
输出：3
解释：它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6
```

示例 3：

```
输入：s = "0"
输出：0
```

示例 4：

```
输入：s = "1"
输出：1
```

示例 5：

```
输入：s = "2"
输出：1
```

提示：

```
1 <= s.length <= 100
```

s 只包含数字，并且可以包含前导零。

前置知识

- 爬楼梯问题
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目和爬楼梯问题有异曲同工之妙。

这也是一道典型的动态规划题目。我们来思考：

- 对于一个数字来说[1,9]这九个数字能够被识别为一种编码方式
- 对于两个数字来说[10, 26]这几个数字能被识别为一种编码方式

我们考虑用 dp[i]来切分子问题， 那么 dp[i]表示的意思是当前字符串的以索引 i 结尾的子问题。这样的话，我们最后只需要取 dp[s.length] 就可以解决问题了。

关于递归公式，让我们 `先局部后整体` 。

对于局部，我们遍历到一个元素的时候，有两种方式来组成编码方式，第一种是这个元素本身（需要自身是[1,9]）,第二种是它和前一个元素组成[10, 26]。

用伪代码来表示的话就是： `dp[i] = 以自身去编码（一位） + 以前面的元素和自身去编码（两位）` ，这显然是完备的，这样我们就通过层层推导得到结果。

关键点解析

- 爬楼梯问题（我把这种题目统称为爬楼梯问题）

代码

代码支持： JS，CPP

JS Code:

```
/**
 * @param {string} s
 * @return {number}
 */
var numDecodings = function (s) {
  if (s == null || s.length == 0) {
    return 0;
  }
  const dp = Array(s.length + 1).fill(0);
  dp[0] = 1;
  dp[1] = s[0] !== "0" ? 1 : 0;
  for (let i = 2; i < s.length + 1; i++) {
    const one = +s.slice(i - 1, i);
    const two = +s.slice(i - 2, i);

    if (two >= 10 && two <= 26) {
      dp[i] = dp[i - 2];
    }

    if (one >= 1 && one <= 9) {
      dp[i] += dp[i - 1];
    }
  }

  return dp[dp.length - 1];
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

实际上，我们也可以使用滚动数组优化。

CPP code:

```
class Solution {
public:
  int numDecodings(string s) {
    int pre2 = 0, pre1 = 1;
    for (int i = 0; i < s.size() && pre1; ++i) {
      int cur = 0;
      if (s[i] != '0') cur += pre1;
      if (i != 0 && s[i - 1] != '0' && (s[i - 1] - '0') * 1
        cur += pre2;
      pre2 = pre1;
      pre1 = cur;
    }
    return pre1;
  }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

扩展

如果编码的范围不再是 1-26，而是三位的话怎么办？

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(92. 反转链表 II)

https://leetcode-cn.com/problems/reverse-linked-list-ii/

题目描述

```
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
```

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路（四点法）

这道题和206.reverse-linked-list 有点类似，并且这道题是 206 的升级版。 让我们反转某一个区间，而不是整个链表，我们可以将 206 看作本题的特殊情况（special case）。

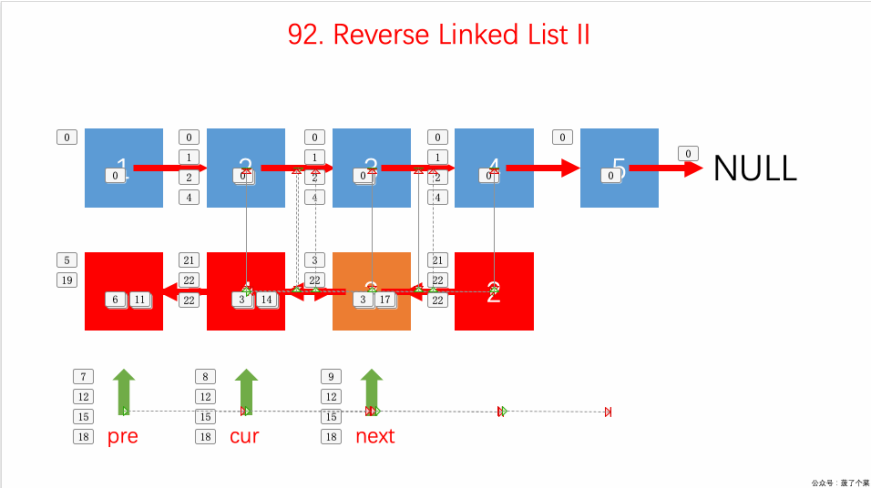
核心在于**取出需要反转的这一小段链表，反转完后再插入到原先的链表中。**

以本题为例：

反转的是 2,3,4 这三个点，那么我们可以先取出 2，用 cur 指针指向 2，然后当取出 3 的时候，我们将 3 指向 2 的，把 cur 指针前移到 3，依次类推，到 4 后停止，这样我们得到一个新链表 4->3->2, cur 指针指向 4。

对于原链表来说，有两个点的位置很重要，需要用指针记录下来，分别是 1 和 5，把新链表插入的时候需要这两个点的位置。用 pre 指针记录 1 的位置当 4 结点被取走后，5 的位置需要记下来

这样我们就可以把反转后的那一小段链表加入到原链表中



(图片来自网络)

- 首先我们直接返回 head 是不行的。 当 m 不等于 1 的时候是没有问题的，但只要 m 为 1，就会有问题。
- 其次如果链表商都小于 4 的时候，p1，p2，p3，p4 就有可能为空。为了防止 NPE，我们也要充分地判空。

```
class Solution:
    def reverseBetween(self, head: ListNode, m: int, n: int
        pre = None
        cur = head
        i = 0
        p1 = p2 = p3 = p4 = None
        # 一坨逻辑
        if p1:
            p1.next = p3
        else:
            dummy.next = p3
        if p2:
            p2.next = p4
        return head
```

如上代码是不可以的，我们考虑使用 dummy 节点。

```
class Solution:
    def reverseBetween(self, head: ListNode, m: int, n: int)
        pre = None
        cur = head
        i = 0
        p1 = p2 = p3 = p4 = None
        dummy = ListNode(0)
        dummy.next = head
        # 一坨逻辑
        if p1:
            p1.next = p3
        else:
            dummy.next = p3
        if p2:
            p2.next = p4

        return dummy.next
```

关于链表反转部分, 顺序比较重要，我们需要：

- 先 cur.next = pre
- 再 更新 p2 和 p2.next(其中要设置 p2.next = None，否则会互相应用，造成无限循环)
- 最后更新 pre 和 cur

上述的顺序不能错，不然会有问题。原因就在于 `p2.next = None` ，如果这个放在最后，那么我们的 cur 会提前断开。

```
    while cur:
            i += 1
            if i == m - 1:
                p1 = cur
            next = cur.next
            if m < i <= n:
                cur.next = pre

            if i == m:
                p2 = cur
                p2.next = None

            if i == n:
                p3 = cur

            if i == n + 1:
                p4 = cur

            pre = cur
            cur = next
```

关键点解析

- 四点法
- 链表的基本操作
- 考虑特殊情况 m 是 1 或者 n 是链表长度的情况，我们可以采用虚拟节点 dummy 简化操作
- 用四个变量记录特殊节点， 然后操作这四个节点使之按照一定方式连接即可。
- 注意更新 current 和 pre 的位置， 否则有可能出现溢出

代码

我把这个方法称为 `四点法`

语言支持：JS, Python3， CPP

JavaScript Code:

```
/*
 * @lc app=leetcode id=92 lang=javascript
 *
 * [92] Reverse Linked List II
 *
 * https://leetcode.com/problems/reverse-linked-list-ii/des
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} m
 * @param {number} n
 * @return {ListNode}
 */
var reverseBetween = function (head, m, n) {
  // 虚拟节点，简化操作
  const dummyHead = {
    next: head,
  };

  let cur = dummyHead.next; // 当前遍历的节点
  let pre = cur; // 因为要反转，因此我们需要记住前一个节点
  let index = 0; // 链表索引，用来判断是否是特殊位置（头尾位置）

  // 上面提到的四个特殊节点
  let p1 = (p2 = p3 = p4 = null);

  while (cur) {
    const next = cur.next;
    index++;

    // 对 (m - n) 范围内的节点进行反转
    if (index > m && index <= n) {
      cur.next = pre;
    }

    // 下面四个if都是边界，用于更新四个特殊节点的值
    if (index === m - 1) {
      p1 = cur;
    }
    if (index === m) {
      p2 = cur;
```

```
    }

    if (index === n) {
      p3 = cur;
    }

    if (index === n + 1) {
      p4 = cur;
    }

    pre = cur;

    cur = next;
  }

  // 两个链表合并起来
  (p1 || dummyHead).next = p3; // 特殊情况需要考虑
  p2.next = p4;

  return dummyHead.next;
};
```

Python Code:

```
class Solution:
    def reverseBetween(self, head: ListNode, m: int, n: int
        if not head.next or n == 1:
            return head
        dummy = ListNode()
        dummy.next = head
        pre = None
        cur = head
        i = 0
        p1 = p2 = p3 = p4 = None
        while cur:
            i += 1
            next = cur.next
            if m < i <= n:
                cur.next = pre
            if i == m - 1:
                p1 = cur
            if i == m:
                p2 = cur
            if i == n:
                p3 = cur
            if i == n + 1:
                p4 = cur
            pre = cur
            cur = next
        if not p1:
            dummy.next = p3
        else:
            p1.next = p3
        p2.next = p4
        return dummy.next
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 25.reverse-nodes-in-k-groups
- 206.reverse-linked-list

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(94. 二叉树的中序遍历)

https://leetcode-cn.com/problems/binary-tree-inorder-traversal/

题目描述

```
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?
```

前置知识

- 二叉树
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

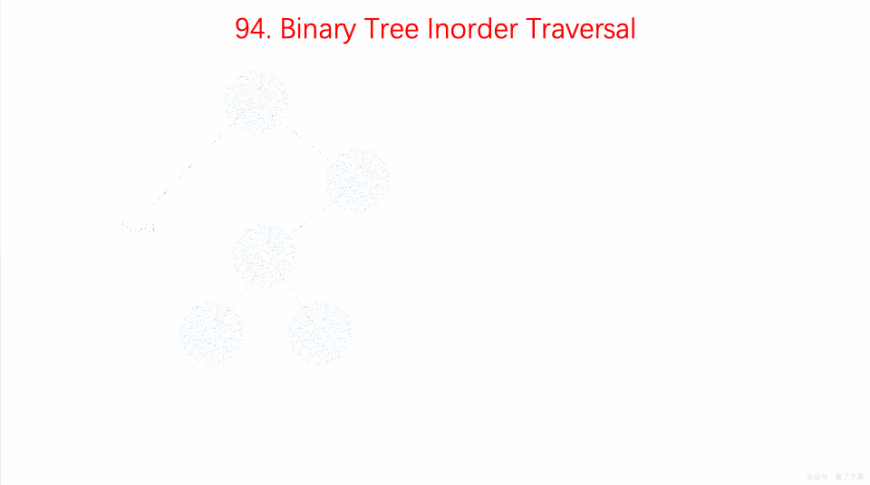
递归的方式相对简单，非递归的方式借助栈这种数据结构实现起来会相对轻松。

如果采用非递归，可以用栈(Stack)的思路来处理问题。

中序遍历的顺序为左-根-右，具体算法为：

- 从根节点开始，先将根节点压入栈

- 然后再将其所有左子结点压入栈，取出栈顶节点，保存节点值
- 再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中， 重复上步骤



(图片来自： https://github.com/MisterBooo/LeetCodeAnimation)

关键点解析

- 二叉树的基本操作（遍历）
  > 不同的遍历算法差异还是蛮大的
- 如果非递归的话利用栈来简化操作
- 如果数据规模不大的话，建议使用递归
- 递归的问题需要注意两点，一个是终止条件，一个如何缩小规模
- 终止条件，自然是当前这个元素是 null（链表也是一样）
- 由于二叉树本身就是一个递归结构， 每次处理一个子树其实就是缩小了规模， 难点在于如何合并结果，这里的合并结果其实就是 `left.concat(mid).concat(right)` , mid 是一个具体的节点，left 和 right `递归求出即可`

代码

- 语言支持：JS，C++，Python3, Java

JavaScript Code：

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var inorderTraversal = function (root) {
  // 1. Recursive solution
  // if (!root) return [];
  // const left = root.left ? inorderTraversal(root.left) :
  // const right = root.right ? inorderTraversal(root.right
  // return left.concat([root.val]).concat(right);

  // 2. iterative solutuon
  if (!root) return [];
  const stack = [root];
  const ret = [];
  let left = root.left;

  let item = null; // stack 中弹出的当前项

  while (left) {
    stack.push(left);
    left = left.left;
  }

  while ((item = stack.pop())) {
    ret.push(item.val);
    let t = item.right;

    while (t) {
      stack.push(t);
      t = t.left;
    }
  }

  return ret;
};
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<TreeNode*> s;
        vector<int> v;
        while (root != NULL || !s.empty()) {
            for (; root != NULL; root = root->left)
                s.push_back(root);
            v.push_back(s.back()->val);
            root = s.back()->right;
            s.pop_back();
        }
        return v;
    }
};
```

Python Code:

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def inorderTraversal(self, root: TreeNode) -> List[int]
        """
        1. 递归法可以一行代码完成，无需讨论；
        2. 迭代法一般需要通过一个栈保存节点顺序，我们这里直接使用列表
          - 首先，我要按照中序遍历的顺序存入栈，这边用的逆序，方便从
          - 在存入栈时加入一个是否需要深化的参数
          - 在回头取值时，这个参数应该是否，即直接取值
          - 简单调整顺序，即可实现前序和后序遍历
        """
        # 递归法
        # if root is None:
        #     return []
        # return self.inorderTraversal(root.left)\
        #     + [root.val]\
        #     + self.inorderTraversal(root.right)
        # 迭代法
        result = []
        stack = [(1, root)]
        while stack:
            go_deeper, node = stack.pop()
            if node is None:
                continue
            if go_deeper:
                # 左右节点还需继续深化，并且入栈是先右后左
                stack.append((1, node.right))
                # 节点自身已遍历，回头可以直接取值
                stack.append((0, node))
                stack.append((1, node.left))
            else:
                result.append(node.val)
        return result
```

Java Code:

- recursion

```java
/**
 * 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 LinkedList<>();
    public List<Integer> inorderTraversal(TreeNode root) {
        inorder(root);
        return res;
    }

    public void inorder (TreeNode root) {
        if (root == null) return;

        inorder(root.left);

        res.add(root.val);

        inorder(root.right);
    }
}
```

- iteration

```java
/**
 * 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> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<> ();
        Stack<TreeNode> stack = new Stack<> ();

        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
        return res;
    }
}
```

相关专题

- 二叉树的遍历

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（95. 不同的二叉搜索树 II）

https://leetcode-cn.com/problems/unique-binary-search-trees-ii/

题目描述

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
```

前置知识

- 二叉搜索树
- 分治

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个经典的使用分治思路的题目。基本思路和96.unique-binary-search-trees一样。

只是我们需要求解的不仅仅是数字，而是要求解所有的组合。我们假设问题 f(1, n) 是求解 1 到 n（两端包含）的所有二叉树。那么我们的目标就是求解 f(1, n)。

我们将问题进一步划分为子问题，假如左侧和右侧的树分别求好了，我们是不是只要运用组合的原理，将左右两者进行合并就好了，我们需要两层循环来完成这个过程。

关键点解析

- 分治法

代码

语言支持：Python3, CPP

Python3 Code:

```python
class Solution:
    def generateTrees(self, n: int) -> List[TreeNode]:
        if not n:
            return []

        def generateTree(start, end):
            if start > end:
                return [None]
            res = []
            for i in range(start, end + 1):
                ls = generateTree(start, i - 1)
                rs = generateTree(i + 1, end)
                for l in ls:
                    for r in rs:
                        node = TreeNode(i)
                        node.left = l
                        node.right = r
                        res.append(node)

            return res

        return generateTree(1, n)
```

CPP Code:

```
class Solution {
private:
    vector<TreeNode*> generateTrees(int first, int last) {
        if (first > last) return { NULL };
        vector<TreeNode*> v;
        for (int i = first; i <= last; ++i) {
            auto lefts = generateTrees(first, i - 1);
            auto rights = generateTrees(i + 1, last);
            for (auto left : lefts) {
                for (auto right : rights) {
                    v.push_back(new TreeNode(i));
                    v.back()->left = left;
                    v.back()->right = right;
                }
            }
        }
        return v;
    }
public:
    vector<TreeNode*> generateTrees(int n) {
        if (n <= 0) return {};
        return generateTrees(1, n);
    }
};
```

复杂度分析 令 C(N) 为 N 的卡特兰数。

- 时间复杂度：$O(N*C(N))$
- 空间复杂度：$O(C(N))$

相关题目

- 96.unique-binary-search-trees

题目地址（96. 不同的二叉搜索树）

https://leetcode-cn.com/problems/unique-binary-search-trees/

题目描述

```
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:

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

前置知识

- 二叉搜索树
- 分治

公司

- 阿里
- 腾讯
- 百度
- 字节

岗位信息

- 腾讯（广州）- 安卓 - 社招 - 三面

思路

这是一个经典的使用分治思路的题目。

对于数字 n ，我们可以 1- n 这样的离散整数分成左右两部分。我们不妨设其分别为 A 和 B。那么问题转化为 A 和 B 所能组成的 BST 的数量的笛卡尔积。而对于 A 和 B 以及原问题除了规模，没有不同，这不就是分治思路么？至于此，我们只需要考虑边界即可，边界很简单就是 n 小于等于 1 的时候，我们返回 1。

具体来说：

- 生成一个[1:n + 1] 的数组
- 我们遍历一次数组，对于每一个数组项，我们执行以下逻辑
- 对于每一项，我们都假设其是断点。断点左侧的是 A，断点右侧的是 B。
- 那么 A 就是 i - 1 个数， 那么 B 就是 n - i 个数
- 我们递归，并将 A 和 B 的结果相乘即可。

> 其实我们发现，题目的答案只和 n 有关，和具体 n 个数的具体组成，只要是有序数组即可。

题目没有明确 n 的取值范围，我们试一下暴力递归。

代码（Python3）：

```
class Solution:
    def numTrees(self, n: int) -> int:
        if n <= 1:
            return 1
        res = 0
        for i in range(1, n + 1):
            res += self.numTrees(i - 1) * self.numTrees(n -
        return res
```

上面的代码会超时，并没有栈溢出，因此我们考虑使用 hashmap 来优化，代码见下方代码区。

关键点解析

- 分治法
- 笛卡尔积
- 记忆化递归

代码

语言支持：Python3, CPP

Python3 Code:

```
class Solution:
    visited = dict()

    def numTrees(self, n: int) -> int:
        if n in self.visited:
            return self.visited.get(n)
        if n <= 1:
            return 1
        res = 0
        for i in range(1, n + 1):
            res += self.numTrees(i - 1) * self.numTrees(n -
        self.visited[n] = res
        return res
```

CPP Code:

```
class Solution {
    vector<int> visited;
    int dp(int n) {
        if (visited[n]) return visited[n];
        int ans = 0;
        for (int i = 0; i < n; ++i) ans += dp(i) * dp(n - i
        return visited[n] = ans;
    }
public:
    int numTrees(int n) {
        visited.assign(n + 1, 0);
        visited[0] = 1;
        return dp(n);
    }
};
```

复杂度分析

- 时间复杂度：一层循环是 N，另外递归深度是 N，因此总的时间复杂度是 $O(N^2)$
- 空间复杂度：递归的栈深度和 visited 的大小都是 N，因此总的空间复杂度为 $O(N)$

相关题目

- 95.unique-binary-search-trees-ii

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(98. 验证二叉搜索树)

https://leetcode-cn.com/problems/validate-binary-search-tree/

题目描述

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 。
```

前置知识

- 中序遍历

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

中序遍历

这道题是让你验证一棵树是否为二叉查找树（BST）。 由于中序遍历的性质 `如果一个树遍历的结果是有序数组，那么他也是一个二叉查找树(BST)` , 我们只需要中序遍历，然后两两判断是否有逆序的元素对即可，如果有，则不是 BST，否则即为一个 BST。

定义法

根据定义，一个结点若是在根的左子树上，那它应该小于根结点的值而大于左子树最小值；若是在根的右子树上，那它应该大于根结点的值而小于右子树最大值。也就是说，每一个结点必须落在某个取值范围：

1. 根结点的取值范围为（考虑某个结点为最大或最小整数的情况）：(long_min, long_max)
2. 左子树的取值范围为：(current_min, root.value)
3. 右子树的取值范围为：(root.value, current_max)

关键点解析

- 二叉树的基本操作（遍历）
- 中序遍历一个二叉查找树（BST）的结果是一个有序数组
- 如果一个树遍历的结果是有序数组，那么他也是一个二叉查找树(BST)

代码

中序遍历

- 语言支持：JS，C++, Java

JavaScript Code：

```
/*
 * @lc app=leetcode id=98 lang=javascript
 *
 * [98] Validate Binary Search Tree
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isValidBST = function (root) {
  if (root === null) return true;
  if (root.left === null && root.right === null) return tru
  const stack = [root];
  let cur = root;
  let pre = null;

  function insertAllLefts(cur) {
    while (cur && cur.left) {
      const l = cur.left;
      stack.push(l);
      cur = l;
    }
  }
  insertAllLefts(cur);

  while ((cur = stack.pop())) {
    if (pre && cur.val <= pre.val) return false;
    const r = cur.right;

    if (r) {
      stack.push(r);
      insertAllLefts(r);
    }
    pre = cur;
  }

  return true;
};
```

C++ Code：

```cpp
// 递归
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        TreeNode* prev = nullptr;
        return validateBstInorder(root, prev);
    }

private:
    bool validateBstInorder(TreeNode* root, TreeNode*& prev
        if (root == nullptr) return true;
        if (!validateBstInorder(root->left, prev)) return f
        if (prev != nullptr && prev->val >= root->val) retu
        prev = root;
        return validateBstInorder(root->right, prev);
    }
};

// 迭代
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        auto s = vector<TreeNode*>();
        TreeNode* prev = nullptr;
        while (root != nullptr || !s.empty()) {
            while (root != nullptr) {
                s.push_back(root);
                root = root->left;
            }
            root = s.back();
            s.pop_back();
            if (prev != nullptr && prev->val >= root->val)
            prev = root;
            root = root->right;
        }
        return true;
    }
};
```

Java Code:

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isValidBST(TreeNode root) {
        Stack<Integer> stack = new Stack<> ();
        TreeNode previous = null;

        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            if (previous != null && root.val <= previous.va
            previous = root;
            root = root.right;
        }
        return true;
    }
}
```

定义法

- 语言支持：C++，Python3, Java, JS

C++ Code：

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
// 递归
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        return helper(root, LONG_MIN, LONG_MAX);
    }
private:
    bool helper(const TreeNode* root, long min, long max) {
        if (root == nullptr) return true;
        if (root->val >= max || root->val <= min) return fa
        return helper(root->left, min, root->val) && helper
    }
};

// 循环
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        if (root == nullptr) return true;
        auto ranges = queue<pair<long, long>>();
        ranges.push(make_pair(LONG_MIN, LONG_MAX));
        auto nodes = queue<const TreeNode*>();
        nodes.push(root);
        while (!nodes.empty()) {
            auto sz = nodes.size();
            for (auto i = 0; i < sz; ++i) {
                auto range = ranges.front();
                ranges.pop();
                auto n = nodes.front();
                nodes.pop();
                if (n->val >= range.second || n->val <= ran
                    return false;
                }
                if (n->left != nullptr) {
                    ranges.push(make_pair(range.first, n->v
                    nodes.push(n->left);
                }
                if (n->right != nullptr) {
                    ranges.push(make_pair(n->val, range.sec
```

```
                    nodes.push(n->right);
                }
            }
        }
        return true;
    }
};
```

Python Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def isValidBST(self, root: TreeNode, area: tuple=(-floa
        """思路如上面的分析，用Python表达会非常直白
        :param root TreeNode 节点
        :param area tuple 取值区间
        """
        if root is None:
            return True

        is_valid_left = root.left is None or\
                    (root.left.val < root.val and area[0] <
        is_valid_right = root.right is None or\
                    (root.right.val > root.val and area[0] <

        # 左右节点都符合，说明本节点符合要求
        is_valid = is_valid_left and is_valid_right

        # 递归下去
        return is_valid\
            and self.isValidBST(root.left, (area[0], root.v
            and self.isValidBST(root.right, (root.val, area
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isValidBST(TreeNode root) {
        return helper(root, null, null);
    }

    private boolean helper(TreeNode root, Integer lower, In
        if (root == null) return true;

        if (lower != null && root.val <= lower) return fals
        if (higher != null && root.val >= higher) return fa

        if (!helper(root.left, lower, root.val)) return fal
        if (!helper(root.right, root.val, higher)) return t

        return true;
    }
}
```

JavaScript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isValidBST = function (root) {
  if (!root) return true;
  return valid(root);
};

function valid(root, min = -Infinity, max = Infinity) {
  if (!root) return true;
  const val = root.val;
  if (val <= min) return false;
  if (val >= max) return false;
  return valid(root.left, min, val) && valid(root.right, va
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关题目

230.kth-smallest-element-in-a-bst

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(102. 二叉树的层序遍历)

https://leetcode-cn.com/problems/binary-tree-level-order-traversal/

题目描述

```
给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左至



示例：
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],
  [9,20],
  [15,7]
]
```

前置知识

- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个典型的二叉树遍历问题， 关于二叉树遍历，我总结了一个专题，大家可以先去看下那个，然后再来刷这道题。

这道题可以借助 `队列` 实现，首先把 root 入队，然后入队一个特殊元素 Null(来表示每层的结束)。

然后就是 while(queue.length), 每次处理一个节点，都将其子节点（在这里是 left 和 right）放到队列中。

然后不断的出队， 如果出队的是 null，则表式这一层已经结束了，我们就继续 push 一个 null。

如果不入队特殊元素 Null 来表示每层的结束，则在 while 循环开始时保存当前队列的长度，以保证每次只遍历一层（参考下面的 C++ Code）。

> 如果采用递归方式，则需要将当前节点，当前所在的 level 以及结果数组传递给递归函数。在递归函数中，取出节点的值，添加到 level 参数对应结果数组元素中（参考下面的 C++ Code 或 Python Code）。

关键点解析

- 队列
- 队列中用 Null(一个特殊元素)来划分每层
- 树的基本操作- 遍历 - 层次遍历（BFS）
- 注意塞入 null 的时候，判断一下当前队列是否为空，不然会无限循环

代码

- 语言支持：JS，C++，Python3

Javascript Code:

```
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrder = function (root) {
  if (!root) return [];
  const items = []; // 存放所有节点
  const queue = [root, null]; // null 简化操作
  let levelNodes = []; // 存放每一层的节点

  while (queue.length > 0) {
    const t = queue.shift();

    if (t) {
      levelNodes.push(t.val);
      if (t.left) {
        queue.push(t.left);
      }
      if (t.right) {
        queue.push(t.right);
      }
    } else {
      // 一层已经遍历完了
      items.push(levelNodes);
      levelNodes = [];
      if (queue.length > 0) {
        queue.push(null);
      }
    }
  }

  return items;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */

 // 迭代
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        auto ret = vector<vector<int>>();
        if (root == nullptr) return ret;
        auto q = vector<TreeNode*>();
        q.push_back(root);
        auto level = 0;
        while (!q.empty())
        {
            auto sz = q.size();
            ret.push_back(vector<int>());
            for (auto i = 0; i < sz; ++i)
            {
                auto t = q.front();
                q.erase(q.begin());
                ret[level].push_back(t->val);
                if (t->left != nullptr) q.push_back(t->left
                if (t->right != nullptr) q.push_back(t->rig
            }
            ++level;
        }
        return ret;
    }
};

// 递归
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> v;
        levelOrder(root, 0, v);
        return v;
    }
private:
    void levelOrder(TreeNode* root, int level, vector<vecto
        if (root == NULL) return;
```

```
        if (v.size() < level + 1) v.resize(level + 1);
        v[level].push_back(root->val);
        levelOrder(root->left, level + 1, v);
        levelOrder(root->right, level + 1, v);
    }
};
```

Python Code：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]
        """递归法"""
        if root is None:
            return []

        result = []

        def add_to_result(level, node):
            """递归函数
            :param level int 当前在二叉树的层次
            :param node TreeNode 当前节点
            """
            if level > len(result) - 1:
                result.append([])

            result[level].append(node.val)
            if node.left:
                add_to_result(level+1, node.left)
            if node.right:
                add_to_result(level+1, node.right)

        add_to_result(0, root)
        return result
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树中节点总数。
- 空间复杂度：$O(N)$，其中 N 为树中节点总数。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

扩展

实际上这道题方法很多， 比如经典的三色标记法。

相关题目

- 103.binary-tree-zigzag-level-order-traversal
- 104.maximum-depth-of-binary-tree

相关专题

- 二叉树的遍历

题目地址(103. 二叉树的锯齿形层次遍历)

https://leetcode-cn.com/problems/binary-tree-zigzag-level-order-traversal/

题目描述

和leetcode 102 基本是一样的，思路是完全一样的。

```
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

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]
]
```

前置知识

- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一个典型的二叉树遍历问题， 关于二叉树遍历，我总结了一个专题，大家可以先去看下那个，然后再来刷这道题。

这道题可以借助 `队列` 实现，首先把root入队，然后入队一个特殊元素Null(来表示每层的结束)。

然后就是while(queue.length), 每次处理一个节点，都将其子节点（在这里是left和right）放到队列中。

然后不断的出队， 如果出队的是null，则表式这一层已经结束了，我们就继续push一个null。

关键点解析

- 队列
- 队列中用Null(一个特殊元素)来划分每层
- 树的基本操作- 遍历 - 层次遍历（BFS）

代码

- 语言支持：JS，C++

JavaScript Code：

```
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var zigzagLevelOrder = function(root) {
  if (!root) return [];
  const items = [];
  let isOdd = true;
  let levelNodes = [];

  const queue = [root, null];


  while(queue.length > 0) {
      const t = queue.shift();

      if (t) {
          levelNodes.push(t.val)
          if (t.left) {
            queue.push(t.left)
          }
          if (t.right) {
            queue.push(t.right)
          }
      } else {
        if (!isOdd) {
          levelNodes = levelNodes.reverse();
        }
        items.push(levelNodes)
        levelNodes = [];
        isOdd = !isOdd;
        if (queue.length > 0) {
            queue.push(null);
        }
      }
  }

  return items

};
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        auto ret = vector<vector<int>>();
        if (root == nullptr) return ret;
        auto queue = vector<const TreeNode*>{root};
        auto isOdd = true;
        while (!queue.empty()) {
            auto sz = queue.size();
            auto level = vector<int>();
            for (auto i = 0; i < sz; ++i) {
                auto n = queue.front();
                queue.erase(queue.begin());
                if (isOdd) level.push_back(n->val);
                else level.insert(level.begin(), n->val);
                if (n->left != nullptr) queue.push_back(n->
                if (n->right != nullptr) queue.push_back(n-
            }
            isOdd = !isOdd;
            ret.push_back(level);
        }
        return ret;
    }
};
```

拓展

由于二叉树是递归结构，因此，可以采用递归的方式来处理。在递归时需要保留当前的层次信息（从0开始），作为参数传递给下一次递归调用。

描述

1. 当前层次为偶数时，将当前节点放到当前层的结果数组尾部
2. 当前层次为奇数时，将当前节点放到当前层的结果数组头部
3. 递归对左子树进行之字形遍历，层数参数为当前层数+1
4. 递归对右子树进行之字形遍历，层数参数为当前层数+1

C++实现

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        auto ret = vector<vector<int>>();
        zigzagLevelOrder(root, 0, ret);
        return ret;
    }
private:
    void zigzagLevelOrder(const TreeNode* root, int level,
        if (root == nullptr || level < 0) return;
        if (ret.size() <= level) {
            ret.push_back(vector<int>());
        }
        if (level % 2 == 0) ret[level].push_back(root->val)
        else ret[level].insert(ret[level].begin(), root->va
        zigzagLevelOrder(root->left, level + 1, ret);
        zigzagLevelOrder(root->right, level + 1, ret);
    }
};
```

相关题目

- 102.binary-tree-level-order-traversal
- 104.maximum-depth-of-binary-tree

题目地址(113. 路径总和 II)

https://leetcode-cn.com/problems/path-sum-ii/

题目描述

```
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

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  5   1
返回:

[
   [5,4,11,2],
   [5,8,4,5]
]
```

前置知识

- 回溯法

公司

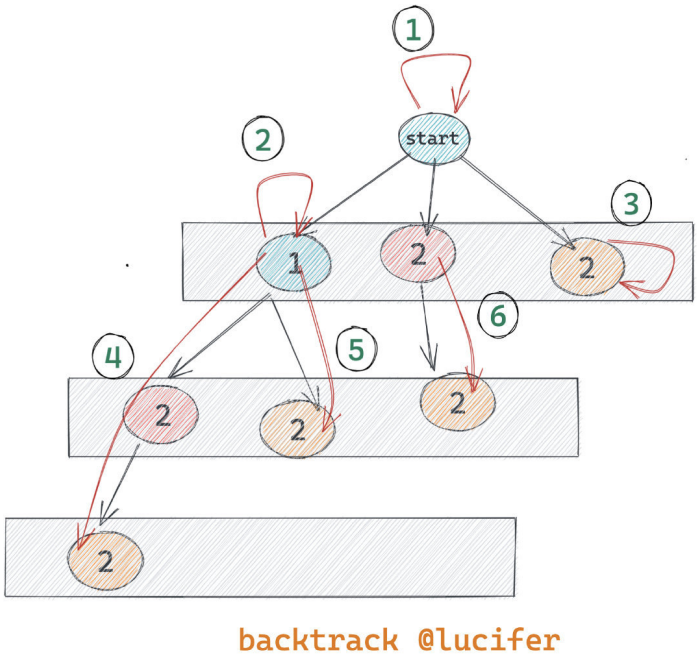
- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是求集合，并不是 `求值` ，而是枚举所有可能，因此动态规划不是特别切合，因此我们需要考虑别的方法。

这种题目其实有一个通用的解法，就是回溯法。网上也有大神给出了这种回溯法解题的通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

我们先来看下通用解法的解题思路，我画了一张图：



> 图是 78.subsets，都差不多，仅做参考。

通用写法的具体代码见下方代码区。

关键点解析

- 回溯法
- backtrack 解题公式

代码

- 语言支持：JS，C++，Python3

JavaScript Code：

```
/*
 * @lc app=leetcode id=113 lang=javascript
 *
 * [113] Path Sum II
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function backtrack(root, sum, res, tempList) {
  if (root === null) return;
  if (root.left === null && root.right === null && sum ===
    return res.push([...tempList, root.val]);

  tempList.push(root.val);
  backtrack(root.left, sum - root.val, res, tempList);

  backtrack(root.right, sum - root.val, res, tempList);
  tempList.pop();
}
/**
 * @param {TreeNode} root
 * @param {number} sum
 * @return {number[][]}
 */
var pathSum = function (root, sum) {
  if (root === null) return [];
  const res = [];
  backtrack(root, sum, res, []);
  return res;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int sum) {
        auto ret = vector<vector<int>>();
        auto temp = vector<int>();
        backtrack(root, sum, ret, temp);
        return ret;
    }
private:
    void backtrack(const TreeNode* root, int sum, vector<ve
        if (root == nullptr) return;
        tempList.push_back(root->val);
        if (root->val == sum && root->left == nullptr && ro
            ret.push_back(tempList);
        } else {
            backtrack(root->left, sum - root->val, ret, ter
            backtrack(root->right, sum - root->val, ret, te
        }
        tempList.pop_back();
    }
};
```

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def pathSum(self, root: TreeNode, sum: int) -> List[Lis
        if not root:
            return []

        result = []

        def trace_node(pre_list, left_sum, node):
            new_list = pre_list.copy()
            new_list.append(node.val)
            if not node.left and not node.right:
                # 这个判断可以和上面的合并，但分开写会快几毫秒，可
                if left_sum == node.val:
                    result.append(new_list)
            else:
                if node.left:
                    trace_node(new_list, left_sum-node.val,
                if node.right:
                    trace_node(new_list, left_sum-node.val,

        trace_node([], sum, root)
        return result
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 131.palindrome-partitioning

题目地址(129. 求根到叶子节点数字之和)

https://leetcode-cn.com/problems/sum-root-to-leaf-numbers/

题目描述

```
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

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.
```

前置知识

- 递归

公司

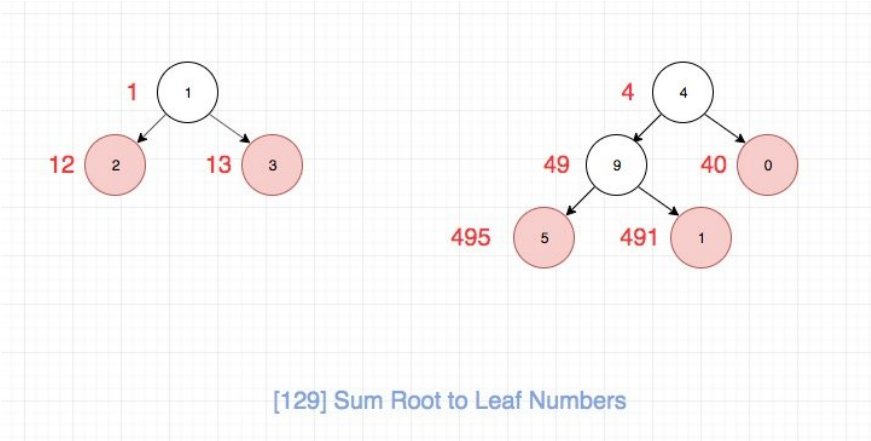
- 阿里
- 百度
- 字节

思路

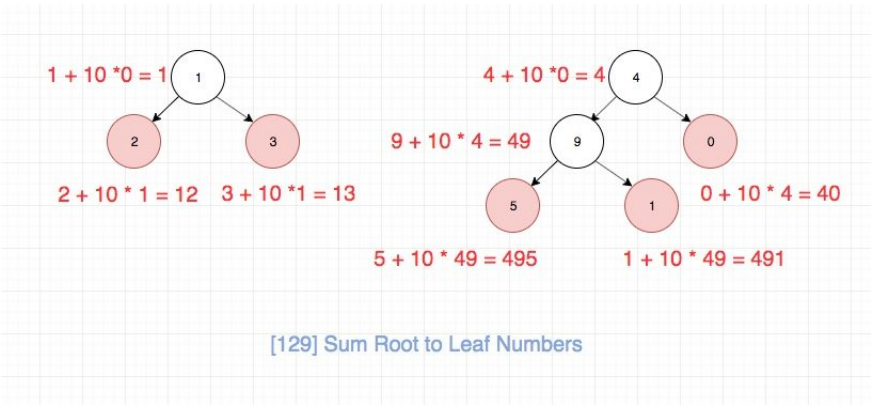
这是一道非常适合训练递归的题目。虽然题目不难，但是要想一次写正确，并且代码要足够优雅却不是很容易。

这里我们的思路是定一个递归的 helper 函数，用来帮助我们完成递归操作。 递归函数的功能是将它的左右子树相加，注意这里不包括这个节点本身，否则会多加， 我们其实关注的就是叶子节点的值，然后通过层层回溯到 root，返回即可。

整个过程如图所示：



那么数字具体的计算逻辑，如图所示，相信大家通过这个不难发现规律：



关键点解析

- 递归分析

代码

- 语言支持：JS，C++，Python, Go, PHP

JS Code：

```
/*
 * @lc app=leetcode id=129 lang=javascript
 *
 * [129] Sum Root to Leaf Numbers
 */
function helper(node, cur) {
  if (node === null) return 0;
  const next = node.val + cur * 10;

  if (node.left === null && node.right === null) return nex

  const l = helper(node.left, next);
  const r = helper(node.right, next);

  return l + r;
}
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var sumNumbers = function (root) {
  // tag: `tree` `dfs` `math`
  return helper(root, 0);
};
```

C++ Code：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        return helper(root, 0);
    }
private:
    int helper(const TreeNode* root, int val) {
        if (root == nullptr) return 0;
        auto ret = root->val + val * 10;
        if (root->left == nullptr && root->right == nullptr)
            return ret;
        auto l = helper(root->left, ret);
        auto r = helper(root->right, ret);
        return l + r;
    }
};
```

Python Code:

```python
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def sumNumbers(self, root: TreeNode) -> int:

        def helper(node, cur_val):
            if not node: return 0
            next_val = cur_val * 10 + node.val

            if not (node.left or node.right):
                return next_val

            left_val = helper(node.left, next_val)
            right_val = helper(node.right, next_val)

            return left_val + right_val

        return helper(root, 0)
```

Go Code:

```go
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func sumNumbers(root *TreeNode) int {
    return helper(root, 0)
}

func helper(root *TreeNode, cur int) int {
    if root == nil {
        return 0 // 当前非叶子节点，不计算
    }

    next := cur*10 + root.Val
    if root.Left == nil && root.Right == nil {
        return next // 当前为叶子节点，计算
    }

    l := helper(root.Left, next)
    r := helper(root.Right, next)
    return l + r
}
```

PHP Code:

```php
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     public $val = null;
 *     public $left = null;
 *     public $right = null;
 *     function __construct($value) { $this->val = $value;
 * }
 */
class Solution
{

    /**
     * @param TreeNode $root
     * @return Integer
     */
    function sumNumbers($root)
    {
        return (new Solution())->helper($root, 0);
    }

    /**
     * @param TreeNode $root
     * @param int $cur
     * @return int
     */
    function helper($root, $cur)
    {
        if (!$root) return 0; // 当前不是叶子节点
        $next = $cur * 10 + $root->val;
        if (!$root->left && !$root->right) return $next; //

        $l = (new Solution())->helper($root->left, $next);
        $r = (new Solution())->helper($root->right, $next);
        return $l + $r;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

拓展

通常来说，可以利用队列、栈等数据结构将递归算法转为递推算法。

描述

使用两个队列：

1. 当前和队列：保存上一层每个结点的当前和（比如 49 和 40）
2. 结点队列：保存当前层所有的非空结点

每次循环按层处理结点队列。处理步骤：

1. 从结点队列取出一个结点
2. 从当前和队列将上一层对应的当前和取出来
3. 若左子树非空，则将该值乘以 10 加上左子树的值，并添加到当前和队列中
4. 若右子树非空，则将该值乘以 10 加上右子树的值，并添加到当前和队列中
5. 若左右子树均为空时，将该节点的当前和加到返回值中

实现

- 语言支持：C++，Python

C++ Code：

```
class Solution {
public:
    int sumNumbers(TreeNode* root) {
        if (root == nullptr) return 0;
        auto ret = 0;
        auto runningSum = vector<int>{root->val};
        auto queue = vector<const TreeNode*>{root};
        while (!queue.empty()) {
            auto sz = queue.size();
            for (auto i = 0; i < sz; ++i) {
                auto n = queue.front();
                queue.erase(queue.begin());
                auto tmp = runningSum.front();
                runningSum.erase(runningSum.begin());
                if (n->left != nullptr) {
                    runningSum.push_back(tmp * 10 + n->left
                    queue.push_back(n->left);
                }
                if (n->right != nullptr) {
                    runningSum.push_back(tmp * 10 + n->righ
                    queue.push_back(n->right);
                }
                if (n->left == nullptr && n->right == nullp
                    ret += tmp;
                }
            }
        }
        return ret;
    }
};
```

Python Code:

```python
class Solution:
    def sumNumbers(self, root: TreeNode) -> int:
        if not root: return 0
        result = 0
        node_queue, sum_queue = [root], [root.val]
        while node_queue:
            for i in node_queue:
                cur_node = node_queue.pop(0)
                cur_val = sum_queue.pop(0)
                if cur_node.left:
                    node_queue.append(cur_node.left)
                    sum_queue.append(cur_val * 10 + cur_noc
                if cur_node.right:
                    node_queue.append(cur_node.right)
                    sum_queue.append(cur_val * 10 + cur_noc
                if not (cur_node.left or cur_node.right):
                    result += cur_val
        return result
```

相关题目

- sum-of-root-to-leaf-binary-numbers

> 这道题和本题太像了，跟一道题没啥区别

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(130. 被围绕的区域)

https://leetcode-cn.com/problems/surrounded-regions/

题目描述

```
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
```

前置知识

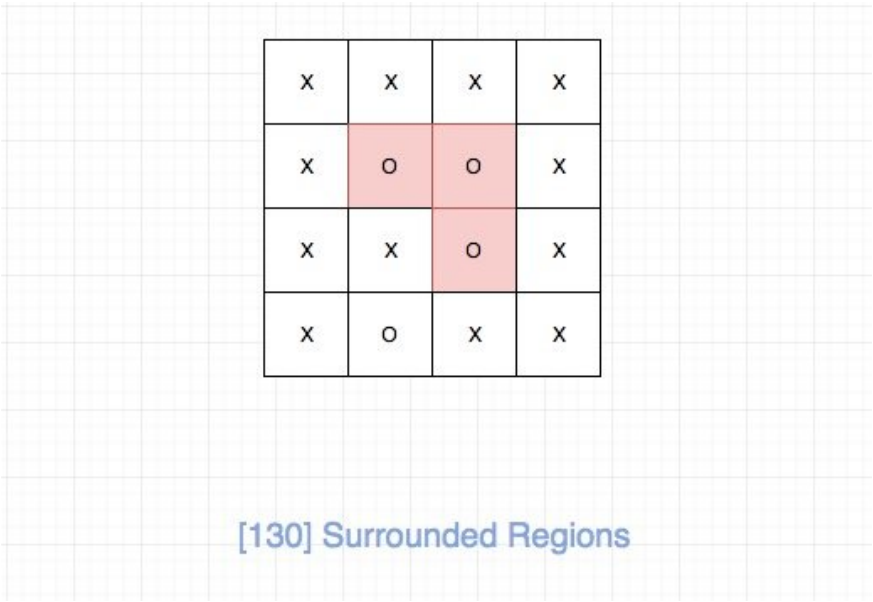
- DFS

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

我们需要将所有被 X 包围的 O 变成 X，并且题目明确说了边缘的所有 O 都是不可以变成 X 的。

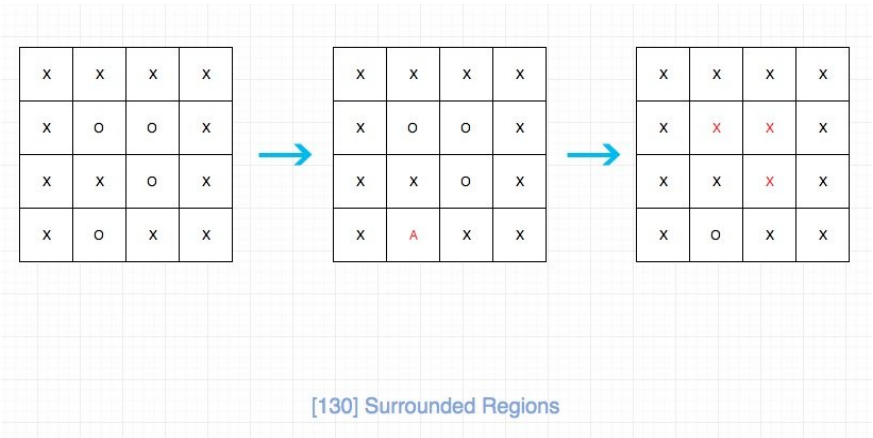


其实我们观察会发现，我们除了边缘的 O 以及和边缘 O 连通的 O 是不需要变成 X 的，其他都要变成 X。

经过上面的思考，问题转化为连通区域问题。 这里我们需要标记一下 `边缘的0以及和边缘0连通的0` 。 我们当然可以用额外的空间去存，但是对于这道题目而言，我们完全可以 mutate。这样就空间复杂度会好一点。

整个过程如图所示：

> 我将 `边缘的0以及和边缘0连通的0` 标记为了 "A"



关键点解析

- 二维数组 DFS 解题模板
- 转化问题为 `连通区域问题`
- 直接 mutate 原数组，节省空间

代码

- 语言支持：JS，Python3, CPP

```
/*
 * @lc app=leetcode id=130 lang=javascript
 *
 * [130] Surrounded Regions
 */
// 将O以及周边的O转化为A
function mark(board, i, j, rows, cols) {
  if (i < 0 || i > rows - 1 || j < 0 || j > cols - 1 || boa
    return;

  board[i][j] = "A";
  mark(board, i + 1, j, rows, cols);
  mark(board, i - 1, j, rows, cols);
  mark(board, i, j + 1, rows, cols);
  mark(board, i, j - 1, rows, cols);
}
/**
 * @param {character[][]} board
 * @return {void} Do not return anything, modify board in-p
 */
var solve = function (board) {
  const rows = board.length;
  if (rows === 0) return [];
  const cols = board[0].length;

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (i === 0 || i == rows - 1 || j === 0 || j === cols
        mark(board, i, j, rows, cols);
      }
    }
  }

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (board[i][j] === "O") {
        board[i][j] = "X";
      } else if (board[i][j] === "A") {
        board[i][j] = "O";
      }
    }
  }

  return board;
};
```

Python Code：

```python
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place inste
        """
        # 如果数组长或宽小于等于2，则不需要替换
        if len(board) <= 2 or len(board[0]) <= 2:
            return

        row, col = len(board), len(board[0])

        def dfs(i, j):
            """
            深度优先算法，如果符合条件，替换为A并进一步测试，否则停止
            """
            if i < 0 or j < 0 or i >= row or j >= col or bo
                return
            board[i][j] = 'A'

            dfs(i - 1, j)
            dfs(i + 1, j)
            dfs(i, j - 1)
            dfs(i, j + 1)

        # 从外围开始
        for i in range(row):
            dfs(i, 0)
            dfs(i, col-1)

        for j in range(col):
            dfs(0, j)
            dfs(row-1, j)

        # 最后完成替换
        for i in range(row):
            for j in range(col):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                elif board[i][j] == 'A':
                    board[i][j] = 'O'
```

CPP Code:

```
class Solution {
    int M, N, dirs[4][2] = {{0, 1}, {0, -1}, {1, 0}, {-1, 0
    void dfs(vector<vector<char>> &board, int x, int y) {
        if (x < 0 || x >= M || y < 0 || y >= N || board[x]
        board[x][y] = '#';
        for (auto &dir : dirs) dfs(board, x + dir[0], y + d
    }
public:
    void solve(vector<vector<char>>& board) {
        if (board.empty() || board[0].empty()) return;
        M = board.size(), N = board[0].size();
        for (int i = 0; i < M; ++i) {
            dfs(board, i, 0);
            dfs(board, i, N - 1);
        }
        for (int j = 0; j < N; ++j) {
            dfs(board, 0, j);
            dfs(board, M - 1, j);
        }
        for (auto &row : board) {
            for (auto &cell : row) {
                cell = cell == '#' ? 'O' : 'X';
            }
        }
    }
};
```

相关题目

- 200.number-of-islands

> 解题模板是一样的

复杂度分析

- 时间复杂度：$O(row * col)$
- 空间复杂度：$O(row * col)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(131. 分割回文串)

https://leetcode-cn.com/problems/palindrome-partitioning/

题目描述

```
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"]
]
```

前置知识

- 回溯法

公司

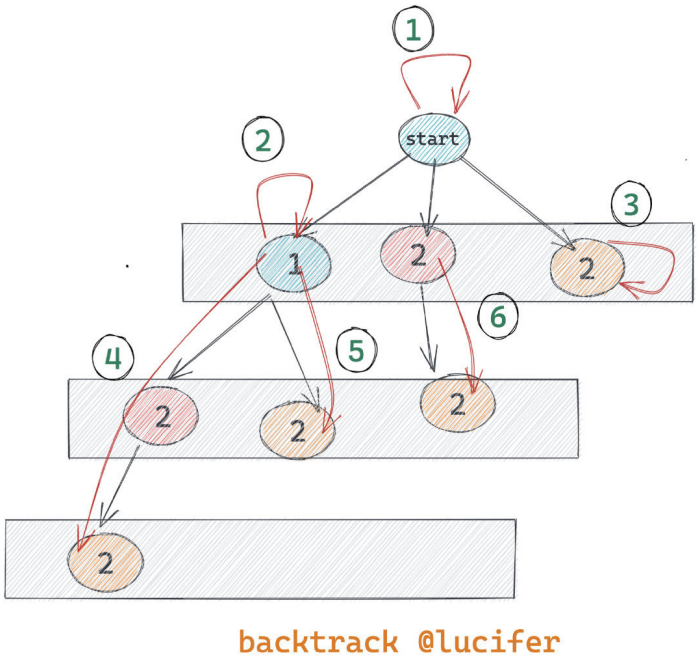
- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道求解所有可能性的题目， 这时候可以考虑使用回溯法。 回溯法解题的模板我们已经在很多题目中用过了， 这里就不多说了。大家可以结合其他几道题目加深一下理解。

这种题目其实有一个通用的解法，就是回溯法。网上也有大神给出了这种回溯法解题的通用写法>)，这里的所有的解法使用通用方法解答。 除了这道题目还有很多其他题目可以用这种通用解法，具体的题目见后方相关题目部分。

这里我画了一个图：



> 图是 78.subsets，都差不多，仅做参考。

关键点解析

- 回溯法

代码

- 语言支持：JS，Python3, CPP

JS Code:

```
/*
 * @lc app=leetcode id=131 lang=javascript
 *
 * [131] Palindrome Partitioning
 */

function isPalindrom(s) {
  let left = 0;
  let right = s.length - 1;

  while (left < right && s[left] === s[right]) {
    left++;
    right--;
  }

  return left >= right;
}
function backtrack(s, list, tempList, start) {
  const sliced = s.slice(start);

  if (isPalindrom(sliced) && tempList.join("").length === s
    list.push([...tempList]);

  for (let i = 0; i < sliced.length; i++) {
    const sub = sliced.slice(0, i + 1);
    if (isPalindrom(sub)) {
      tempList.push(sub);
    } else {
      continue;
    }
    backtrack(s, list, tempList, start + i + 1);
    tempList.pop();
  }
}
/**
 * @param {string} s
 * @return {string[][]}
 */
var partition = function (s) {
  // "aab"
  // ["aa", "b"]
  // ["a", "a", "b"]
  const list = [];
  backtrack(s, list, [], 0);
  return list;
};
```

Python Code:

```python
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        """回溯法"""

        res = []

        def helper(s, tmp):
            """
            如果是空字符串，说明已经处理完毕
            否则逐个字符往前测试，判断是否是回文
            如果是，则处理剩余字符串，并将已经得到的列表作为参数
            """
            if not s:
                res.append(tmp)
            for i in range(1, len(s) + 1):
                if s[:i] == s[:i][::-1]:
                    helper(s[i:], tmp + [s[:i]])

        helper(s, [])
        return res
```

CPP Code:

```
    class Solution {
private:
  vector<vector<string>> ans;
  vector<string> tmp;
  bool isPalindrome(string &s, int first, int last) {
    while (first < last && s[first] == s[last]) ++first, --
    return first >= last;
  }
  void dfs(string &s, int start) {
    if (start == s.size()) { ans.push_back(tmp); return; }
    for (int i = start; i < s.size(); ++i) {
      if (isPalindrome(s, start, i)) {
        tmp.push_back(s.substr(start, i - start + 1));
        dfs(s, i + 1);
        tmp.pop_back();
      }
    }
  }
public:
  vector<vector<string>> partition(string s) {
    dfs(s, 0);
    return ans;
  }
};
```

相关题目

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 78.subsets
- 90.subsets-ii
- 113.path-sum-ii

题目地址(139. 单词拆分)

https://leetcode-cn.com/problems/word-break/

题目描述

```
给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否

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 ap
     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", "
输出: false
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

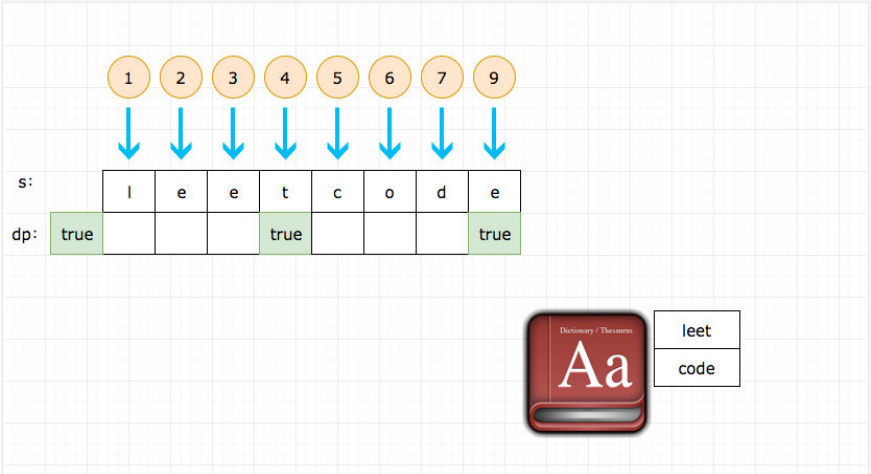
思路

这道题是给定一个字典和一个句子，判断该句子是否可以由字典里面的单词组出来，一个单词可以用多次。

暴力的方法是无解的，复杂度极其高。 我们考虑其是否可以拆分为小问题来解决。 对于问题 `(s, wordDict)` 我们是否可以用(s', wordDict) 来解决。 其中 s' 是 s 的子序列， 当 s'变成寻常(长度为 0)的时候问题就解决了。 我们状态转移方程变成了这道题的难点。

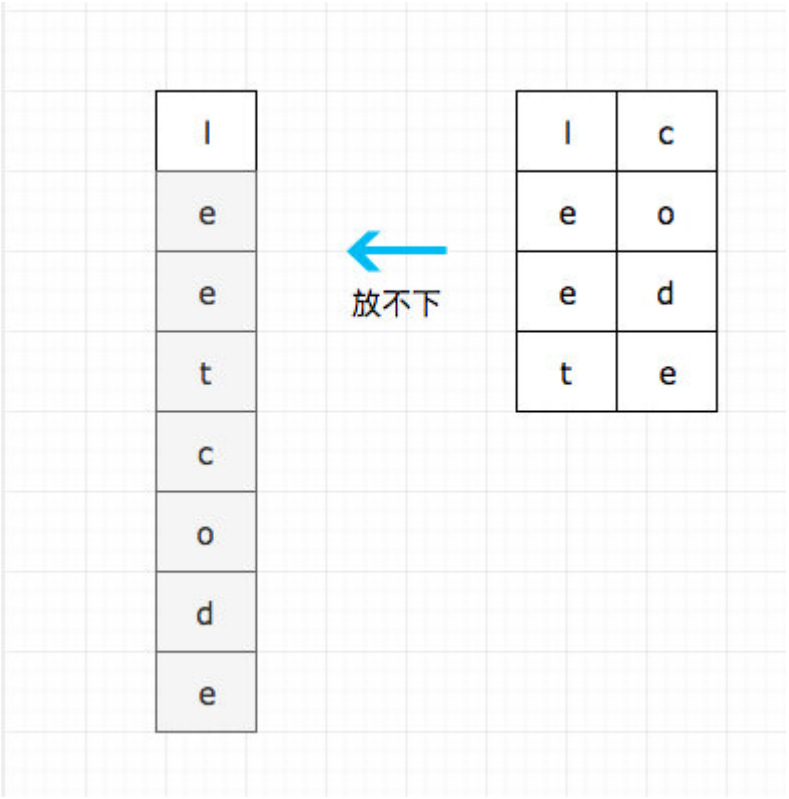
我们可以建立一个数组 dp, dp[i]代表 字符串 s.substring(0, i) 能否由字典里面的单词组成， 值得注意的是，这里我们无法建立 dp[i] 和 dp[i - 1] 的关系， 我们可以建立的是 dp[i - word.length] 和 dp[i] 的关系。

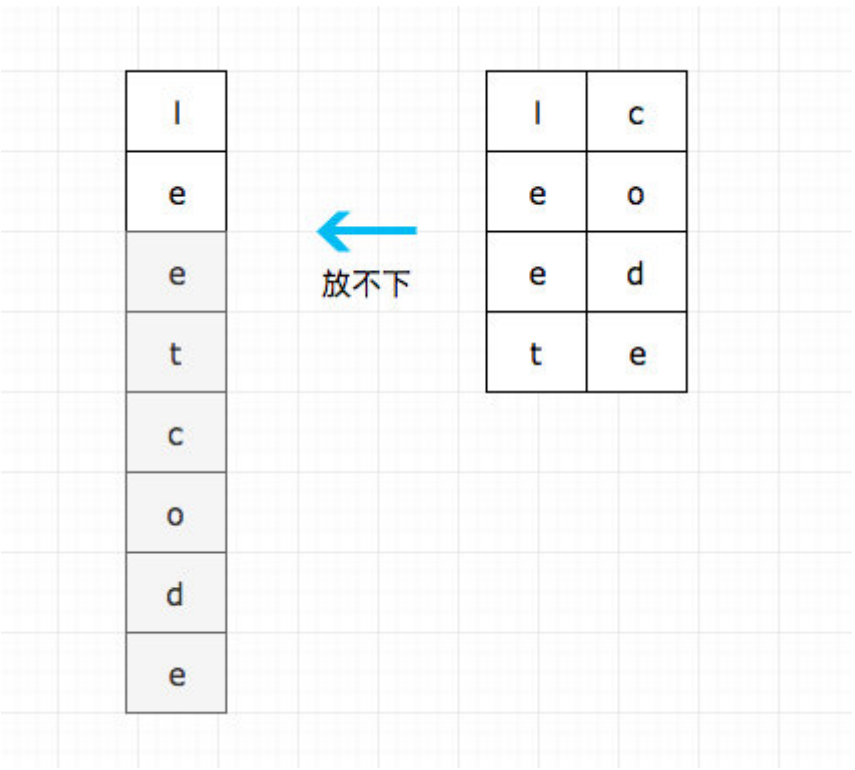
我们用图来感受一下：

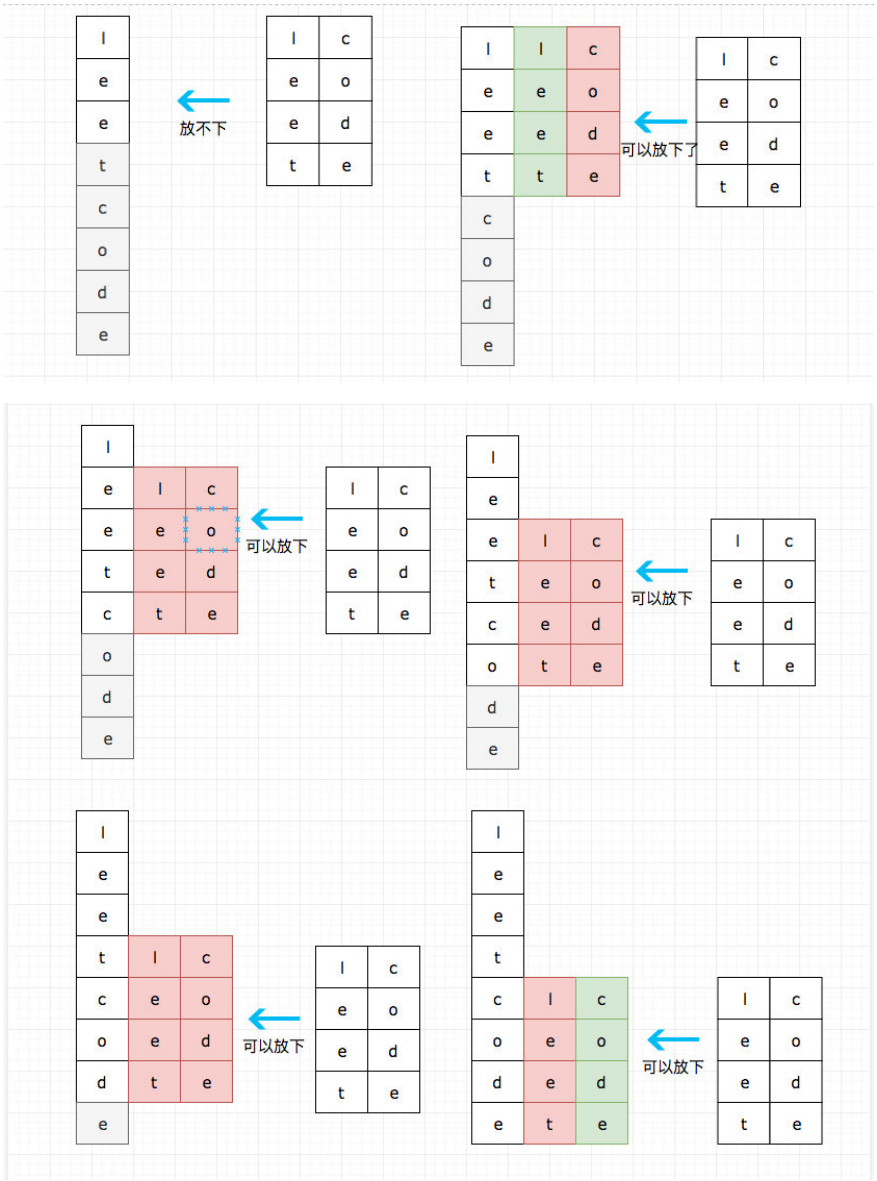


没有明白也没有关系，我们分步骤解读一下：

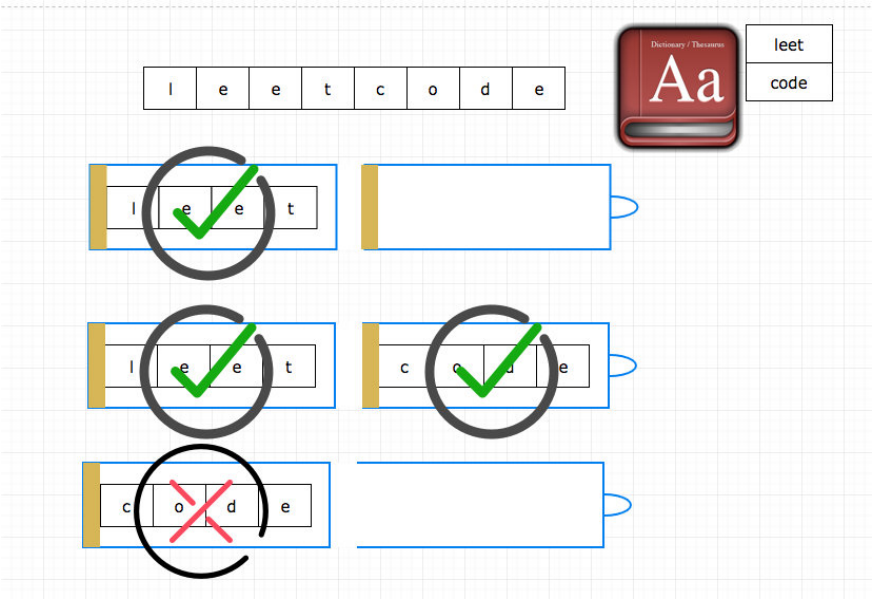
（以下的图左边都代表 s，右边都是 dict，灰色代表没有处理的字符，绿色代表匹配成功，红色代表匹配失败）


l
e
e
t
c
o
d
e
放不下
l c
e o
e d
t e


l
e
e
t
c
o
d
e
放不下
l c
e o
e d
t e



上面分步解释了算法的基本过程，下面我们感性认识下这道题，我把它比喻为 你正在 往一个老式手电筒🔦 中装电池



代码

代码支持： JS，CPP

JS Code:

```
/**
 * @param {string} s
 * @param {string[]} wordDict
 * @return {boolean}
 */
var wordBreak = function (s, wordDict) {
  const dp = Array(s.length + 1);
  dp[0] = true;
  for (let i = 0; i < s.length + 1; i++) {
    for (let word of wordDict) {
      if (word.length <= i && dp[i - word.length]) {
        if (s.substring(i - word.length, i) === word) {
          dp[i] = true;
        }
      }
    }
  }

  return dp[s.length] || false;
};
```

CPP Code:

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& dict) {
        unordered_set<string> st(begin(dict), end(dict));
        int N = s.size();
        vector<bool> dp(N + 1);
        dp[0] = true;
        for (int i = 1; i <= N; ++i) {
            for (int j = 0; j < i && !dp[i]; ++j) {
                dp[i] = dp[j] && st.count(s.substr(j, i - j
            }
        }
        return dp[N];
    }
};
```

复杂度分析

令 S 和 W 分别为字符串和字典的长度。

- 时间复杂度：$O(S ^ 3)$
- 空间复杂度：$O(S + W)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(144. 二叉树的前序遍历)

https://leetcode-cn.com/problems/binary-tree-preorder-traversal/

题目描述

```
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?
```

前置知识

- 递归
- 栈

公司

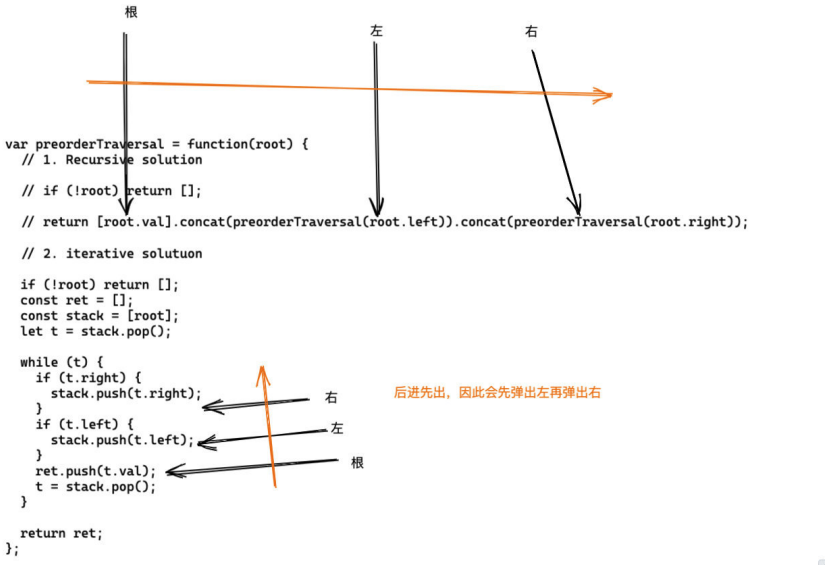
- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目是前序遍历，这个和之前的 `leetcode 94 号问题 – 中序遍历` 完全不一回事。

前序遍历是 `根左右` 的顺序，注意是 `根` 开始，那么就很简单。直接先将根节点入栈，然后 看有没有右节点，有则入栈，再看有没有左节点，有则入栈。 然后出栈一个元素，重复即可。

> 其他树的非递归遍历可没这么简单



（迭代 VS 递归）

关键点解析

- 二叉树的基本操作（遍历）
  > 不同的遍历算法差异还是蛮大的
- 如果非递归的话利用栈来简化操作
- 如果数据规模不大的话，建议使用递归
- 递归的问题需要注意两点，一个是终止条件，一个如何缩小规模
- 终止条件，自然是当前这个元素是 null（链表也是一样）
- 由于二叉树本身就是一个递归结构， 每次处理一个子树其实就是缩小了规模， 难点在于如何合并结果，这里的合并结果其实就是 `mid.concat(left).concat(right)` , mid 是一个具体的节点，left 和 right `递归求出即可`

代码

- 语言支持：JS，C++

JavaScript Code:

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var preorderTraversal = function (root) {
  // 1. Recursive solution

  // if (!root) return [];

  // return [root.val].concat(preorderTraversal(root.left)

  // 2. iterative solutuon

  if (!root) return [];
  const ret = [];
  const stack = [root];
  let t = stack.pop();

  while (t) {
    if (t.right) {
      stack.push(t.right);
    }
    if (t.left) {
      stack.push(t.left);
    }
    ret.push(t.val);
    t = stack.pop();
  }

  return ret;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> v;
        vector<TreeNode*> s;
        while (root != NULL || !s.empty()) {
            while (root != NULL) {
                v.push_back(root->val);
                s.push_back(root);
                root = root->left;
            }
            root = s.back()->right;
            s.pop_back();
        }
        return v;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关专题

- 二叉树的遍历

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（147. 对链表进行插入排序）

https://leetcode-cn.com/problems/insertion-sort-list/

题目描述

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
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

插入排序算法：

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
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
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
```

思路

这道题属于链表题目中的修改指针。看过我链表专题的小伙伴应该熟悉我解决这种链表问题的套路了吧？

关于链表的题目，我总结了四个技巧**虚拟头**， **快慢指针**，**穿针引线** 和 **先穿再排后判空**，这道题我们使用到了两个，分别是**虚拟头**和**先穿再排后判空**。这四个技巧的具体内容可以查看我的《链表专题》。

链表问题首先我们看返回值，如果返回值不是原本链表的头的话，我们可以使用虚拟节点。

此时我们的代码是这样的：

```
class Solution:
    def insertionSortList(self, head: ListNode) -> ListNode
        ans = ListNode(float("-inf"))
        # do domething
        return ans.next
```

接着，我们理清算法整体框架，把细节留出来。

我们的算法应该有两层循环，外层控制当前需要插入的元素，内层选择插入的位置。

此时我们的代码是这样的：

```
class Solution:
    def insertionSortList(self, head: ListNode) -> ListNode
        ans = ListNode(float("-inf"))

        def insert(to_be_insert):
            # 选择插入的位置，并插入

        while head:
            insert(head)
            head = head.next
        return ans.next
```

而选择插入位置的代码也不难，只需要每次都从头出发遍历，找到第一个大于 to_be_insert 的，在其前面插入即可（如果有的话）：

```
# ans 就是上面我提到的虚拟节点
ans = cur
while cur.next and cur.next.val < to_be_insert.val:
    cur = cur.next
```

而链表插入是一个基本操作，不多解释，直接看代码：

```
to_be_insert.next = cur.next
cur.next = to_be_insert
```

于是完整代码就呼之欲出了：

```
class Solution:
    def insertionSortList(self, head: ListNode) -> ListNode
        ans = ListNode(float("-inf"))

        def helper(inserted):
            cur = ans
            while cur.next and cur.next.val < inserted.val
                cur = cur.next
            inserted.next = cur.next
            cur.next = inserted

        while head:
            helper(head)
            head = head.next
        return ans.next
```

到这里还没完，继续使用我的第二个技巧**先穿再排后判空**。由于这道题没有穿，我们直接排序和判空即可。

next 改变的代码一共两行，因此只需要关注这两行代码即可：

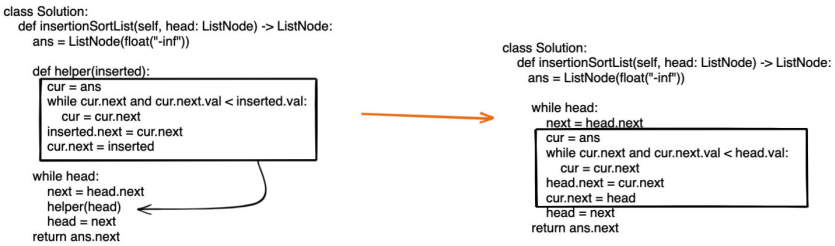
```
inserted.next = cur.next
cur.next = inserted
```

inserted 的 next 改变实际上会影响外层，解决方案很简单，**留个联系方式**就好。这我在上面的文章也反复提到过。

```
class Solution:
    def insertionSortList(self, head: ListNode) -> ListNode
        ans = ListNode(float("-inf"))

        def insert(to_be_insert):
            # 选择插入的位置，并插入
            # 这里 to_to_insert 的 next 会被修改，进而影响外层的

        while head:
            # 留下联系方式
            next = head.next
            insert(head)
            # 使用联系方式更新 head
            head = next
        return ans.next
```

> 不熟悉这个技巧的看下上面提到的我写的链表文章即可。

如果你上面代码你会了，将 insert 代码整个复制出来就变成大部分人的解法了。不过我还是建议新手按照我的这个模式一步步来，稳扎稳打，不要着急。



代码

代码支持：Python3，Java，JS, CPP

Python Code:

```
class Solution:
    def insertionSortList(self, head: ListNode) -> ListNode
        ans = ListNode(float("-inf"))

        while head:
            next = head.next
            cur = ans
            while cur.next and cur.next.val < head.val:
                cur = cur.next
            head.next = cur.next
            cur.next = head
            head = next
        return ans.next
```

Java Code:

```
class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode ans = new ListNode(-1);
        while( head != null ){
            ListNode next = head.next;
            ListNode cur = ans;
            while(cur.next != null && cur.next.val < head.v
                cur = cur.next;
            }
            head.next = cur.next;
            cur.next = head;
            head = next;
        }

        return ans.next;
    }
}
```

JS Code:

```
var insertionSortList = function (head) {
  ans = new ListNode(-1);
  while (head != null) {
    next = head.next;
    cur = ans;
    while (cur.next != null && cur.next.val < head.val) {
      cur = cur.next;
    }
    head.next = cur.next;
    cur.next = head;
    head = next;
  }

  return ans.next;
};
```

CPP Code:

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode dummy, *p;
        while (head) {
            auto *n = head;
            head = head->next;
            p = &dummy;
            while (p->next && p->next->val < n->val) p = p-
            n->next = p->next;
            p->next = n;
        }
        return dummy.next;
    }
};
```

复杂度分析

- 时间复杂度：$O(N^2)$，其中 N 为链表长度。
- 空间复杂度：$O(1)$。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(150. 逆波兰表达式求值)

https://leetcode-cn.com/problems/evaluate-reverse-polish-notation/

题目描述

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

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

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"
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
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，

前置知识

- 栈

公司

- 阿里
- 腾讯

思路

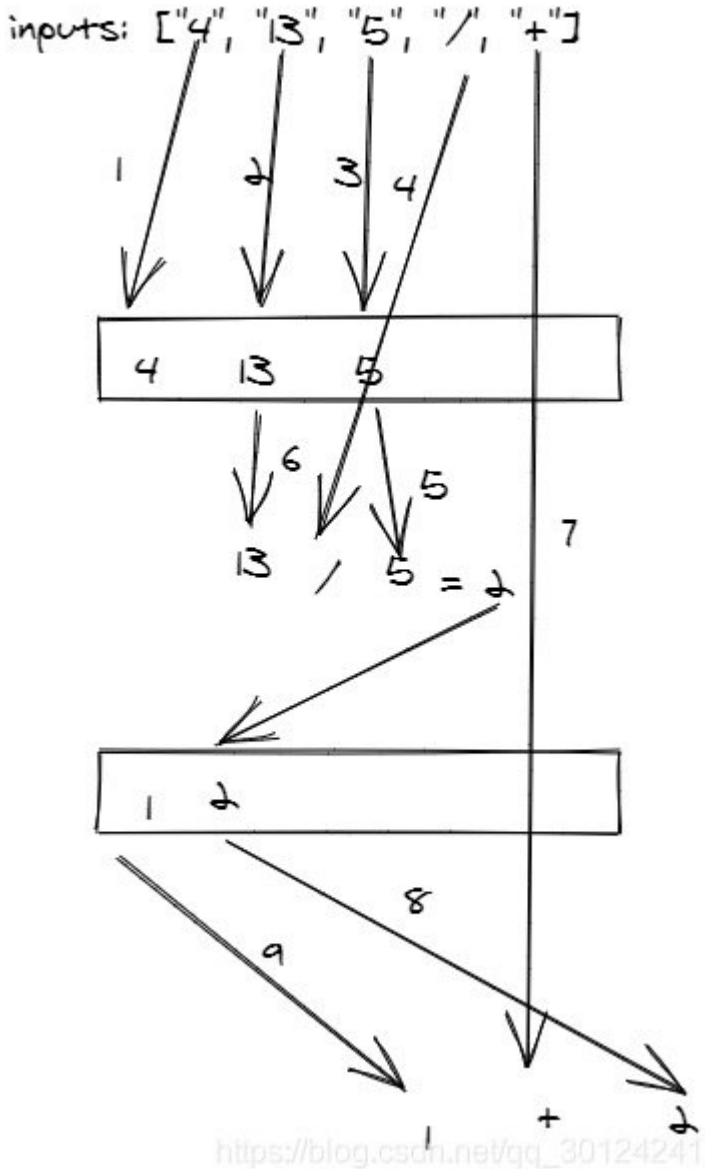
逆波兰表达式又叫做后缀表达式。在通常的表达式中，二元运算符总是置于与之相关的两个运算对象之间，这种表示法也称为 `中缀表示` 。

波兰逻辑学家 J.Lukasiewicz 于 1929 年提出了另一种表示表达式的方法，按此方法，每一运算符都置于其运算对象之后，故称为 `后缀表示` 。

> 逆波兰表达式是一种十分有用的表达式，它将复杂表达式转换为可以依靠简单的操作得到计算结果的表达式。例如(a+b)*(c+d)转换为ab+cd+*

思路就是：

- 遍历列表，依次入栈，直到遇到算数运算符。
- 将栈顶两个元素出栈运算，将结果压栈
- 重复以上过程直到所有的 token 都处理完毕。



关键点

1. 栈的基本用法
2. 如果你用的是 JS 的话，需要注意/ 和 其他很多语言是不一样的
3. 如果你用的是 JS 的话，需要先将字符串转化为数字。否则有很多意想不到的结果
4. 操作符的顺序应该是 先出栈的是第二位，后出栈的是第一位。 这在不符合交换律的操作中很重要， 比如减法和除法。

代码

代码支持：JS，Python，Java, CPP

JS Code:

```
/**
 * @param {string[]} tokens
 * @return {number}
 */
var evalRPN = function (tokens) {
  // 这种算法的前提是 tokens是有效的,
  // 当然这由算法来保证
  const stack = [];

  for (let index = 0; index < tokens.length; index++) {
    const token = tokens[index];
    // 对于运算数,  我们直接入栈
    if (!Number.isNaN(Number(token))) {
      stack.push(token);
    } else {
      // 遇到操作符，我们直接大胆运算，不用考虑算术优先级
      // 然后将运算结果入栈即可

      // 当然如果题目进一步扩展，允许使用单目等其他运算符，我们的算法
      const a = Number(stack.pop());
      const b = Number(stack.pop());
      if (token === "*") {
        stack.push(b * a);
      } else if (token === "/") {
        stack.push((b / a) >> 0);
      } else if (token === "+") {
        stack.push(b + a);
      } else if (token === "-") {
        stack.push(b - a);
      }
    }
  }

  return stack.pop();
};
```

Python Code:

```python
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        if len(tokens) > 2:
            stack = []
            operations = ['+', '-', '*', '/']
            for token in tokens:
                if token in operations:
                    b = int(stack.pop())
                    a = int(stack.pop())
                    if '+' == token:
                        tmp = a + b
                    elif '-' == token:
                        tmp = a - b
                    elif '*' == token:
                        tmp = a * b
                    else:
                        tmp = int(a / b)
                    stack.append(tmp)
                else:
                    stack.append(token)
            return stack[0]
        return int(tokens[-1])
```

Java Code:

```java
class Solution {
    public static int evalRPN(String[] tokens) {
    int[] numStack = new int[tokens.length / 2 + 1];
    int index = 0;
    for (String s : tokens) {
        if (s.equals("+")) {
                numStack[index - 2] += numStack[--index];
            } else if (s.equals("-")) {
                numStack[index - 2] -= numStack[--index];
            } else if (s.equals("*")) {
                numStack[index - 2] *= numStack[--index];
            } else if (s.equals("/")) {
                numStack[index - 2] /= numStack[--index];
            } else {
                numStack[index++] = Integer.parseInt(s);
            }
    }
    return numStack[0];
    }
}
```

CPP Code:

```
    class Solution {
public:
  int evalRPN(vector<string>& tokens) {
    stack<int> s;
    for (string t : tokens) {
      if (isdigit(t.back())) s.push(stoi(t));
      else {
        int n = s.top();
        s.pop();
        switch(t[0]) {
          case '+': s.top() += n; break;
          case '-': s.top() -= n; break;
          case '*': s.top() *= n; break;
          case '/': s.top() /= n; break;
        }
      }
    }
    return s.top();
  }
};
```

扩展

逆波兰表达式中只改变运算符的顺序，并不会改变操作数的相对顺序，这是一个重要的性质。另外逆波兰表达式完全不关心操作符的优先级，这在中缀表达式中是做不到的，这很有趣，感兴趣的可以私下查找资料研究下为什么会这样。

题目地址(152. 乘积最大子数组)

https://leetcode-cn.com/problems/maximum-product-subarray/

题目描述

```
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



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。
```

前置知识

- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

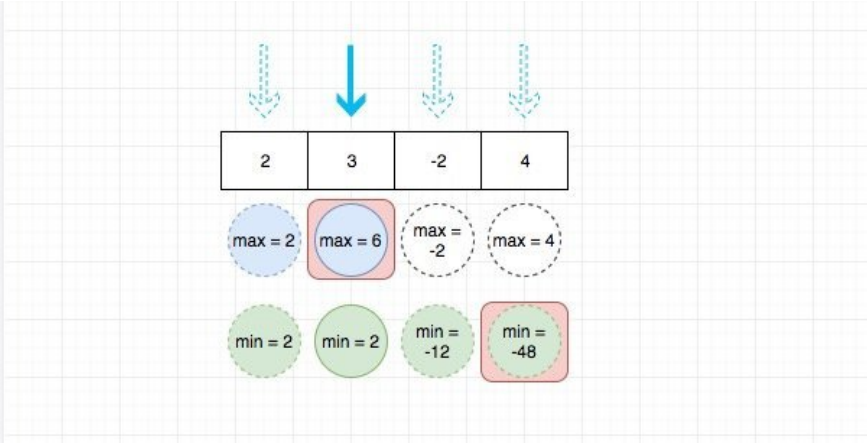
思路

这道题目要我们求解连续的 n 个数中乘积最大的积是多少。这里提到了连续，笔者首先想到的就是滑动窗口，但是这里比较特殊，我们不能仅仅维护一个最大值，因此最小值（比如-20）乘以一个比较小的数（比如-10）可能就会很大。 因此这种思路并不方便。

首先来暴力求解,我们使用两层循环来枚举所有可能项，这种解法的时间复杂度是 O(n^2), 代码如下：

```
var maxProduct = function (nums) {
  let max = nums[0];
  let temp = null;
  for (let i = 0; i < nums.length; i++) {
    temp = nums[i];
    for (let j = i + 1; j < nums.length; j++) {
      temp *= nums[j];
      max = Math.max(temp, max);
    }
  }

  return max;
};
```

前面说了`最小值（比如-20）乘以一个比较小的数（比如-10）可能就会很大`。因此我们需要同时记录乘积最大值和乘积最小值，然后比较元素和这两个的乘积，去不断更新最大值。当然，我们也可以选择只取当前元素。因此实际上我们的选择有三种，而如何选择就取决于哪个选择带来的价值最大（乘积最大或者最小）。



这种思路的解法由于只需要遍历一次，其时间复杂度是 O(n)，代码见下方代码区。

关键点

- 同时记录乘积最大值和乘积最小值

代码

代码支持：Python3，JavaScript, CPP

Python3 Code:

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n = len(nums)
        max__dp = [1] * (n + 1)
        min_dp = [1] * (n + 1)
        ans = float('-inf')

        for i in range(1, n + 1):
            max__dp[i] = max(max__dp[i - 1] * nums[i - 1],
                             min_dp[i - 1] * nums[i - 1], n
            min_dp[i] = min(max__dp[i - 1] * nums[i - 1],
                            min_dp[i - 1] * nums[i - 1], nu
            ans = max(ans, max__dp[i])
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

当我们知道动态转移方程的时候，其实应该发现了。我们的 dp[i] 只和 dp[i - 1]有关，这是一个空间优化的信号，告诉我们 `可以借助两个额外变量记录即可` 。

Python3 Code:

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n = len(nums)
        a = b = 1
        ans = float('-inf')

        for i in range(1, n + 1):
            temp = a
            a = max(a * nums[i - 1],
                    b * nums[i - 1], nums[i - 1])
            b = min(temp * nums[i - 1],
                    b * nums[i - 1], nums[i - 1])
            ans = max(ans, a)
        return ans
```

JavaScript Code:

```
var maxProduct = function (nums) {
  let max = nums[0];
  let min = nums[0];
  let res = nums[0];

  for (let i = 1; i < nums.length; i++) {
    let tmp = min;
    min = Math.min(nums[i], Math.min(max * nums[i], min * r
    max = Math.max(nums[i], Math.max(max * nums[i], tmp * r
    res = Math.max(res, max);
  }
  return res;
};
```

CPP Code:

```
class Solution {
public:
    int maxProduct(vector<int>& A) {
        int maxProd = 1, minProd = 1, ans = INT_MIN;
        for (int n : A) {
            int a = n * maxProd, b = n * minProd;
            maxProd = max({n, a, b});
            minProd = min({n, a, b});
            ans = max(ans, maxProd);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

题目地址(153. 寻找旋转排序数组中的最小值)

https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/

题目描述

```
已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 旋转 后，得
若旋转 4 次，则可以得到 [4,5,6,7,0,1,2]
若旋转 4 次，则可以得到 [0,1,2,4,5,6,7]

注意，数组 [a[0], a[1], a[2], ..., a[n-1]] 旋转一次 的结果为数组

给你一个元素值 互不相同 的数组 nums ，它原来是一个升序排列的数组，并按

示例 1:

输入：nums = [3,4,5,1,2]
输出：1
解释：原数组为 [1,2,3,4,5] ，旋转 3 次得到输入数组。

示例 2:

输入：nums = [4,5,6,7,0,1,2]
输出：0
解释：原数组为 [0,1,2,4,5,6,7] ，旋转 4 次得到输入数组。

示例 3:

输入：nums = [11,13,15,17]
输出：11
解释：原数组为 [11,13,15,17] ，旋转 4 次得到输入数组。

提示：

n == nums.length
1 <= n <= 5000
-5000 <= nums[i] <= 5000
nums 中的所有整数 互不相同
nums 原来是一个升序排序的数组，并进行了 1 至 n 次旋转
```

前置知识

- 二分

公司

- 暂无

思路

这道题和普通的旋转数组找指定值类似，都是利用数组有序使用二分进行求解。不过和普通的整体有序数组不同，大家需要先判断 mid 是在**左侧有序部分还是右侧有序部分**。

这道题也不例外。而由于这道题是求最小值，而不是一个指定的值。我们可以不照搬我的二分讲义中提供的二分模板，而是换一种方式，这样代码会更简洁。

解空间： (l, r)

这样整体模板就是：

```
while l < r:
   # your code here
return nums[l] # or nums[r]
```

和普通旋转数组找指定值类似，我们先判断 mid 在左边有序部分还是右边有序部分。

- 如果 mid 在左边有序部分，那么直接 l = mid + 1
- 否则 r = mid

注意由于我们的 mid 是向下取整得到的，因此 r = mid 并不会导致死循环，而 l = mid 则可能会死循环。

另外**如果左端点的值小于右端点的值则可以提前退出**，这点需要注意。

关键点

- 如果左端点的值小于右端点的值则可以提前退出

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1

        while l < r:
            # important
            if nums[l] < nums[r]:
                return nums[l]
            mid = (l + r) // 2
            # left part
            if nums[mid] > nums[r]:
                l = mid + 1
            else:
                # right part
                r = mid
        # l or r is not important
        return nums[l]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(logn)$
- 空间复杂度：$O(1)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(199. 二叉树的右视图)

https://leetcode-cn.com/problems/binary-tree-right-side-view/

题目描述

```
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

示例:

输入: [1,2,3,null,5,null,4]
输出: [1, 3, 4]
解释:

   1            <---
 /   \
2     3         <---
 \     \
  5     4       <---
```

前置知识

- 队列

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

> 这道题和 leetcode 102 号问题《102.binary-tree-level-order-traversal》很像

这道题可以借助 `队列` 实现，首先把 root 入队，然后入队一个特殊元素 Null(来表示每层的结束)。

然后就是 while(queue.length), 每次处理一个节点，都将其子节点（在这里是 left 和 right）放到队列中。

然后不断的出队， 如果出队的是 null，则表式这一层已经结束了，我们就继续 push 一个 null。

关键点解析

- 队列
- 队列中用 Null(一个特殊元素)来划分每层
- 树的基本操作- 遍历 - 层次遍历（BFS）
- 二叉树的右视图可以看作是层次遍历每次只取每一层的最右边的元素

代码

语言支持：JS，C++

Javascript Code:

```
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var rightSideView = function(root) {
  if (!root) return [];

  const ret = [];
  const queue = [root, null];

  let levelNodes = [];

  while (queue.length > 0) {
    const node = queue.shift();
    if (node !== null) {
      levelNodes.push(node.val);
      if (node.right) {
        queue.push(node.right);
      }
      if (node.left) {
        queue.push(node.left);
      }
    } else {
      // 一层遍历已经结束
      ret.push(levelNodes[0]);
      if (queue.length > 0) {
        queue.push(null);
      }
      levelNodes = [];
    }
  }

  return ret;
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        auto ret = vector<int>();
        if (root == nullptr) return ret;
        auto q = queue<const TreeNode*>();
        q.push(root);
        while (!q.empty()) {
            auto sz = q.size();
            for (auto i = 0; i < sz; ++i) {
                auto n = q.front();
                q.pop();
                if (n->left != nullptr ) q.push(n->left);
                if (n->right != nullptr ) q.push(n->right);
                if (i == sz - 1) ret.push_back(n->val);
            }
        }
        return ret;
    }
};
```

扩展

假如题目变成求二叉树的左视图呢?

很简单我们只需要取 queue 的最后一个元素即可。 或者存的时候反着来也行

> 其实我们没必要存储 levelNodes，而是只存储每一层最右的元素，这样空间复杂度就不是 n 了， 就是 logn 了。

题目地址(200. 岛屿数量)

https://leetcode-cn.com/problems/number-of-islands/

题目描述

```
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

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连

此外，你可以假设该网格的四条边均被水包围。



示例 1:

输入: grid = [
  ["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: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
输出: 3


提示:

m == grid.length
n == grid[i].length
1 <= m, n <= 300
grid[i][j] 的值为 '0' 或 '1'
```

前置知识

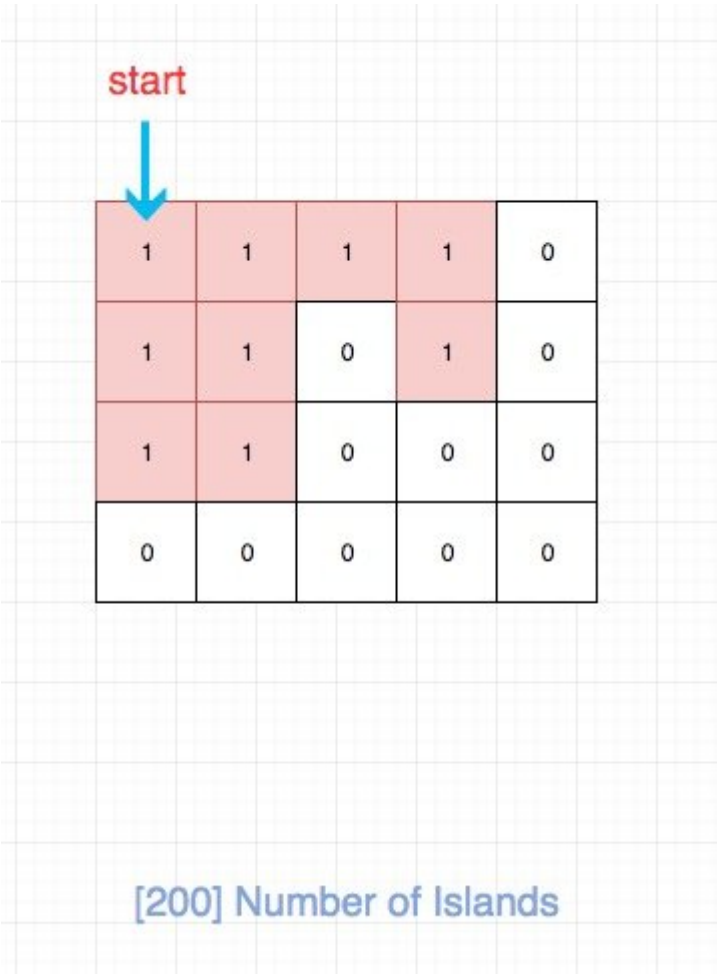
- DFS

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

如图，我们其实就是要求红色区域的个数，换句话说就是求连续区域的个数。



符合直觉的做法是用 DFS 来解：

- 我们需要建立一个 visited 数组用来记录某个位置是否被访问过。
- 对于一个为 `1` 且未被访问过的位置，我们递归进入其上下左右位置上为 `1` 的数，将其 visited 变成 true。
- 重复上述过程
- 找完相邻区域后，我们将结果 res 自增 1，然后我们在继续找下一个为 `1` 且未被访问过的位置，直至遍历完.

但是这道题目只是让我们求连通区域的个数，因此我们其实不需要额外的空间去存储 visited 信息。 注意到上面的过程，我们对于数字为 0 的其实不会进行操作的，也就是对我们“没用”。 因此对于已经访问的元素， 我们可以将其置为 0 即可。

关键点解析

- 二维数组 DFS 解题模板
- 将已经访问的元素置为 0，省去 visited 的空间开销

代码

- 语言支持：C++, Java, JS, python3

C++ Code：

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int res = 0;
        for(int i=0;i<grid.size();i++)
        {
            for(int j=0;j<grid[0].size();j++)
            {
                if(grid[i][j] == '1')
                {
                    dfs(grid, i, j);
                    res += 1;
                }
            }
        }
        return res;

    }
    void dfs(vector<vector<char>>& grid, int i, int j)
    {
        // edge
        if(i<0 || i>= grid.size() || j<0 || j>= grid[0].siz
        {
            return;
        }
        grid[i][j] = '0';
        dfs(grid, i+1, j);
        dfs(grid, i-1, j);
        dfs(grid, i, j+1);
        dfs(grid, i, j-1);
    }
};
```

Java Code：

```
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].len

        int count = 0;
        for (int row = 0; row < grid.length; row++) {
            for (int col = 0; col < grid[0].length; col++)
                if (grid[row][col] == '1') {
                    dfs(grid, row, col);
                    count++;
                }
            }
        }
        return count;
    }

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

Javascript Code:

```
/*
 * @lc app=leetcode id=200 lang=javascript
 *
 * [200] Number of Islands
 */
function helper(grid, i, j, rows, cols) {
  if (i < 0 || j < 0 || i > rows - 1 || j > cols - 1 || gr
    return;

  grid[i][j] = "0";

  helper(grid, i + 1, j, rows, cols);
  helper(grid, i, j + 1, rows, cols);
  helper(grid, i - 1, j, rows, cols);
  helper(grid, i, j - 1, rows, cols);
}
/**
 * @param {character[][]} grid
 * @return {number}
 */
var numIslands = function (grid) {
  let res = 0;
  const rows = grid.length;
  if (rows === 0) return 0;
  const cols = grid[0].length;
  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      if (grid[i][j] === "1") {
        helper(grid, i, j, rows, cols);
        res++;
      }
    }
  }
  return res;
};
```

python code:

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid: return 0

        count = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    self.dfs(grid, i, j)
                    count += 1

        return count

    def dfs(self, grid, i, j):
        if i < 0 or j < 0 or i >= len(grid) or j >= len(gri
            return
        grid[i][j] = '0'
        self.dfs(grid, i + 1, j)
        self.dfs(grid, i - 1, j)
        self.dfs(grid, i, j + 1)
        self.dfs(grid, i, j - 1)
```

复杂度分析

- 时间复杂度：$O(m * n)$
- 空间复杂度：$O(m * n)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

相关题目

- 695. 岛屿的最大面积

题目地址(201. 数字范围按位与)

https://leetcode-cn.com/problems/bitwise-and-of-numbers-range/

题目描述

```
给定范围 [m, n]，其中 0 <= m <= n <= 2147483647，返回此范围内所有

示例 1:

输入: [5,7]
输出: 4
示例 2:

输入: [0,1]
输出: 0
```

前置知识

- 位运算

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

一个显而易见的解法是， 从 m 到 n 依次进行 `求与` 的操作。

```
let res = m;
for (let i = m + 1; i <= n; i++) {
  res = res & i;
}
return res;
```

但是， 如果你把这个 solution 提交的话，很显然不会通过， 会超时。

我们依旧还是用 trick 来简化操作。 我们利用的性质是， n 个连续数字求与的时候，前 m 位都是 1.

举题目给的例子：[5,7] 共 5， 6，7 三个数字， 用二进制表示 101, 110,111, 这三个数字特点是第一位都是 1，后面几位求与一定是 0.

再来一个明显的例子：[20, 24], 共 20， 21， 22， 23，24 五个数字，二进制表示就是

```
0001 0100
0001 0101
0001 0110
0001 0111
0001 1000
```

这五个数字特点是第四位都是 1，后面几位求与一定是 0.

因此我们的思路就是， 求出这个数字区间的数字前多少位都是 1 了，那么他们求与的结果一定是前几位数字，然后后面都是 0.

关键点解析

- n 个连续数字求与的时候，前 m 位都是 1
- 可以用递归实现， 个人认为比较难想到
- bit 运算

代码：

```
n > m ? rangeBitwiseAnd(m / 2, n / 2) << 1 : m;
```

> 每次问题规模缩小一半， 这是二分法吗?

代码

语言支持：JavaSCript，Python3

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=201 lang=javascript
 *
 * [201] Bitwise AND of Numbers Range
 *
 */
/**
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
var rangeBitwiseAnd = function (m, n) {
  let count = 0;
  while (m !== n) {
    m = m >> 1;
    n = n >> 1;
    count++;
  }

  return n << count;
};
```

Python Code:

```python
class Solution:
    def rangeBitwiseAnd(self, m: int, n: int) -> int:
        cnt = 0
        while m != n:
            m >>= 1
            n >>= 1
            cnt += 1

        return m << cnt
```

复杂度分析

- 时间复杂度：最坏的情况我们需要循环 N 次，最好的情况是一次都不需要， 因此时间复杂度取决于我们移动的位数，具体移动的次数取决于我们的输入，平均来说时间复杂度为 $O(N)$，其中 N 为 M 和 N 的二进制表示的位数。
- 空间复杂度：$O(1)$

题目地址(208. 实现 Trie (前缀树))

https://leetcode-cn.com/problems/implement-trie-prefix-tree/

题目描述

```
实现一个 Trie (前缀树)，包含 insert, search, 和 startsWith 这三

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。
```

前置知识

- 前缀树

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道很直接的题目，上来就让你实现 `前缀树（字典树）` 。这算是基础数据结构中的 知识了，不清楚什么是字典树的可以查阅相关资料。

我们看到题目给出的使用方法 `new Trie` ，
`insert` ，`search` 和 `startWith` .

为了区分 `search` 和 `startWith` 我们需要增加一个标示来区分当前节点是否是某个单词的结尾。 因此节点的数据结构应该是:

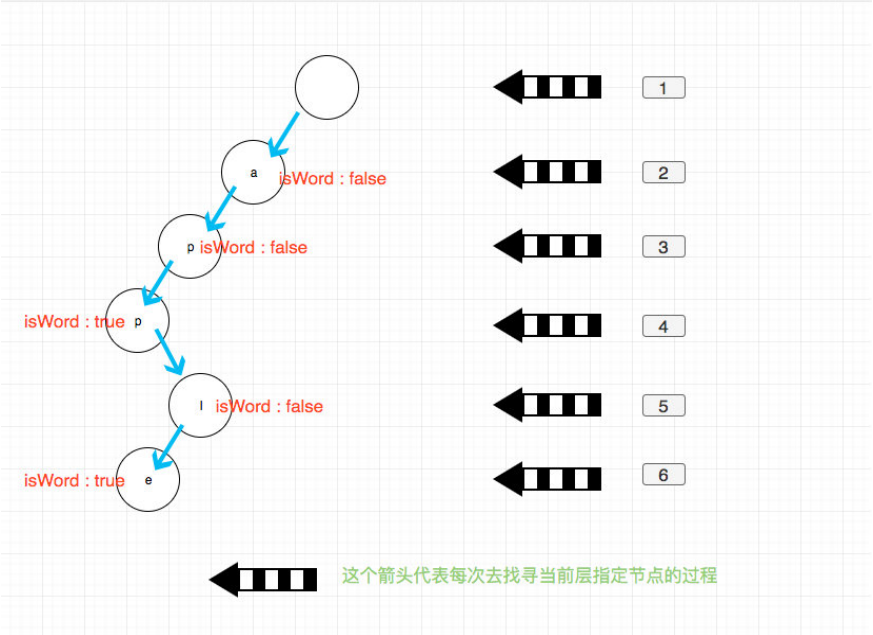
```
function TrieNode(val) {
  this.val = val; // 当前的字母
  this.children = []; // 题目要求字典仅有a-z，那么其长度最大为26
  this.isWord = false;
}
```

每次 insert 我们其实都是从根节点出发，一个一个找到我们需要添加的节点，修改 children 的值.

我们应该修改哪一个 child 呢？ 我们需要一个函数来计算索引

```
function computeIndex(c) {
  return c.charCodeAt(0) - "a".charCodeAt(0);
}
```

其实不管 insert， search 和 startWith 的逻辑都是差不多的，都是从 root 出发， 找到我们需要操作的 child， 然后进行相应操作（添加，修改，返回）。



关键点解析

- 前缀树

代码

```
/*
 * @lc app=leetcode id=208 lang=javascript
 *
 * [208] Implement Trie (Prefix Tree)
 *
 * https://leetcode.com/problems/implement-trie-prefix-tree
 *
 * algorithms
 * Medium (36.93%)
 * Total Accepted:    172K
 * Total Submissions: 455.5K
 * Testcase Example:  '["Trie","insert","search","search","
 *
 * Implement a trie with insert, search, and startsWith met
 *
 * Example:
 *
 *
 * Trie trie = new Trie();
 *
 * trie.insert("apple");
 * trie.search("apple");   // returns true
 * trie.search("app");     // returns false
 * trie.startsWith("app"); // returns true
 * trie.insert("app");
 * trie.search("app");     // returns true
 *
 *
 * Note:
 *
 *
 * You may assume that all inputs are consist of lowercase
 * All inputs are guaranteed to be non-empty strings.
 *
 *
 */
function TrieNode(val) {
  this.val = val;
  this.children = [];
  this.isWord = false;
}

function computeIndex(c) {
  return c.charCodeAt(0) - "a".charCodeAt(0);
}
/**
 * Initialize your data structure here.
 */
```

```
var Trie = function () {
  this.root = new TrieNode(null);
};

/**
 * Inserts a word into the trie.
 * @param {string} word
 * @return {void}
 */
Trie.prototype.insert = function (word) {
  let ws = this.root;
  for (let i = 0; i < word.length; i++) {
    const c = word[i];
    const current = computeIndex(c);
    if (!ws.children[current]) {
      ws.children[current] = new TrieNode(c);
    }
    ws = ws.children[current];
  }
  ws.isWord = true;
};

/**
 * Returns if the word is in the trie.
 * @param {string} word
 * @return {boolean}
 */
Trie.prototype.search = function (word) {
  let ws = this.root;
  for (let i = 0; i < word.length; i++) {
    const c = word[i];
    const current = computeIndex(c);
    if (!ws.children[current]) return false;
    ws = ws.children[current];
  }
  return ws.isWord;
};

/**
 * Returns if there is any word in the trie that starts wit
 * @param {string} prefix
 * @return {boolean}
 */
Trie.prototype.startsWith = function (prefix) {
  let ws = this.root;
  for (let i = 0; i < prefix.length; i++) {
    const c = prefix[i];
    const current = computeIndex(c);
```

```
    if (!ws.children[current]) return false;
    ws = ws.children[current];
  }
  return true;
};

/**
 * Your Trie object will be instantiated and called as such
 * var obj = new Trie()
 * obj.insert(word)
 * var param_2 = obj.search(word)
 * var param_3 = obj.startsWith(prefix)
 */
```

相关题目

- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址(209. 长度最小的子数组)

https://leetcode-cn.com/problems/minimum-size-subarray-sum/

题目描述

```
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 ≥



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
```

前置知识

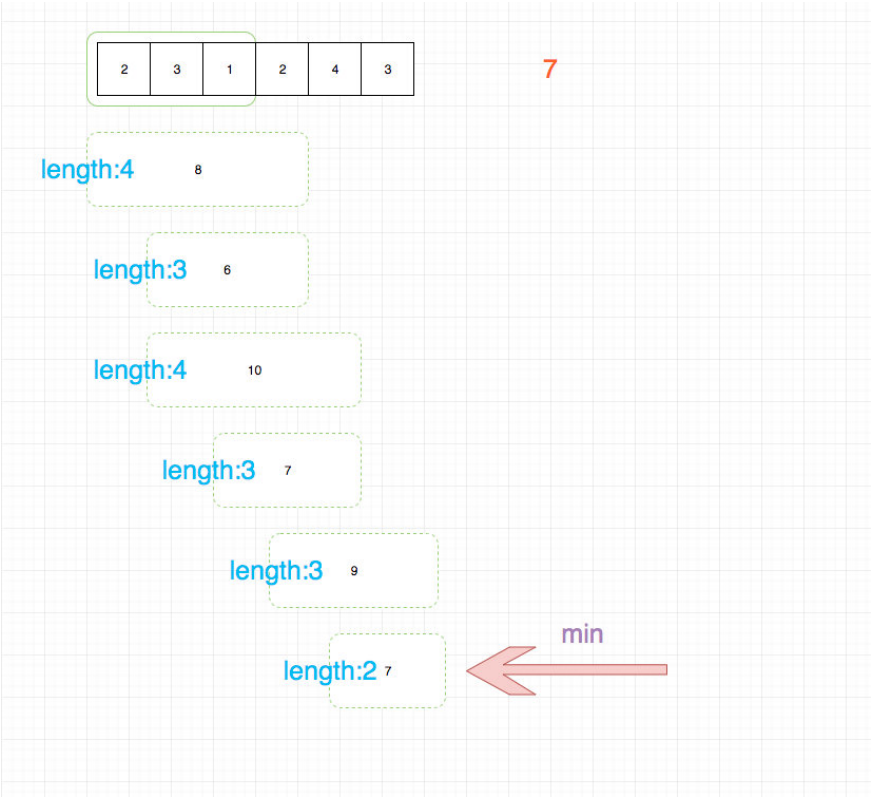
- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

用滑动窗口来记录序列， 每当滑动窗口中的 sum 超过 s， 就去更新最小值，并根据先进先出的原则更新滑动窗口，直至 sum 刚好小于 s



> 这道题目和 leetcode 3 号题目有点像，都可以用滑动窗口的思路来解决

关键点

- 滑动窗口简化操作(滑窗口适合用于求解这种要求 `连续` 的题目)

代码

- 语言支持：JS，C++，Python

Python Code：

```
class Solution:
    def minSubArrayLen(self, s: int, nums: List[int]) -> in
        l = total = 0
        ans = len(nums) + 1
        for r in range(len(nums)):
            total += nums[r]
            while total >= s:
                ans = min(ans, r - l + 1)
                total -= nums[l]
                l += 1
        return  0 if ans == len(nums) + 1 else ans
```

JavaScript Code：

```javascript
/*
 * @lc app=leetcode id=209 lang=javascript
 *
 * [209] Minimum Size Subarray Sum
 *
 */
/**
 * @param {number} s
 * @param {number[]} nums
 * @return {number}
 */
var minSubArrayLen = function (s, nums) {
  if (nums.length === 0) return 0;
  const slideWindow = [];
  let acc = 0;
  let min = null;

  for (let i = 0; i < nums.length + 1; i++) {
    const num = nums[i];

    while (acc >= s) {
      if (min === null || slideWindow.length < min) {
        min = slideWindow.length;
      }
      acc = acc - slideWindow.shift();
    }

    slideWindow.push(num);

    acc = slideWindow.reduce((a, b) => a + b, 0);
  }

  return min || 0;
};
```

C++ Code：

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int num_len= nums.size();
        int left=0, right=0, total=0, min_len= num_len+1;
        while (right < num_len) {
            do {
                total += nums[right++];
            } while (right < num_len && total < s);
            while (left < right && total - nums[left] >= s)
            if (total >=s && min_len > right - left)
                min_len = right- left;
        }
        return min_len <= num_len ? min_len: 0;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组大小。
- 空间复杂度：$O(1)$

欢迎关注我的公众号《脑洞前端》获取更多更新鲜的 LeetCode 题解

扩展

如果题目要求是 sum = s, 而不是 sum >= s 呢?

eg:

```
var minSubArrayLen = function (s, nums) {
  if (nums.length === 0) return 0;
  const slideWindow = [];
  let acc = 0;
  let min = null;

  for (let i = 0; i < nums.length + 1; i++) {
    const num = nums[i];

    while (acc > s) {
      acc = acc - slideWindow.shift();
    }
    if (acc === s) {
      if (min === null || slideWindow.length < min) {
        min = slideWindow.length;
      }
      slideWindow.shift();
    }

    slideWindow.push(num);

    acc = slideWindow.reduce((a, b) => a + b, 0);
  }

  return min || 0;
};
```

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（211. 添加与搜索单词 - 数据结构设计）

https://leetcode-cn.com/problems/design-add-and-search-words-data-structure/

题目描述

```
请你设计一个数据结构，支持 添加新单词 和 查找字符串是否与任何先前添加的

实现词典类 WordDictionary ：

WordDictionary() 初始化词典对象
void addWord(word) 将 word 添加到数据结构中，之后可以对它进行匹配
bool search(word) 如果数据结构中存在字符串与 word 匹配，则返回 tr


示例：

输入：
["WordDictionary","addWord","addWord","addWord","search","s
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]
输出：
[null,null,null,null,false,true,true,true]

解释：
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True


提示：

1 <= word.length <= 500
addWord 中的 word 由小写英文字母组成
search 中的 word 由 '.' 或小写英文字母组成
最调用多 50000 次 addWord 和 search
```

前置知识

- 前缀树

公司

- 阿里
- 腾讯

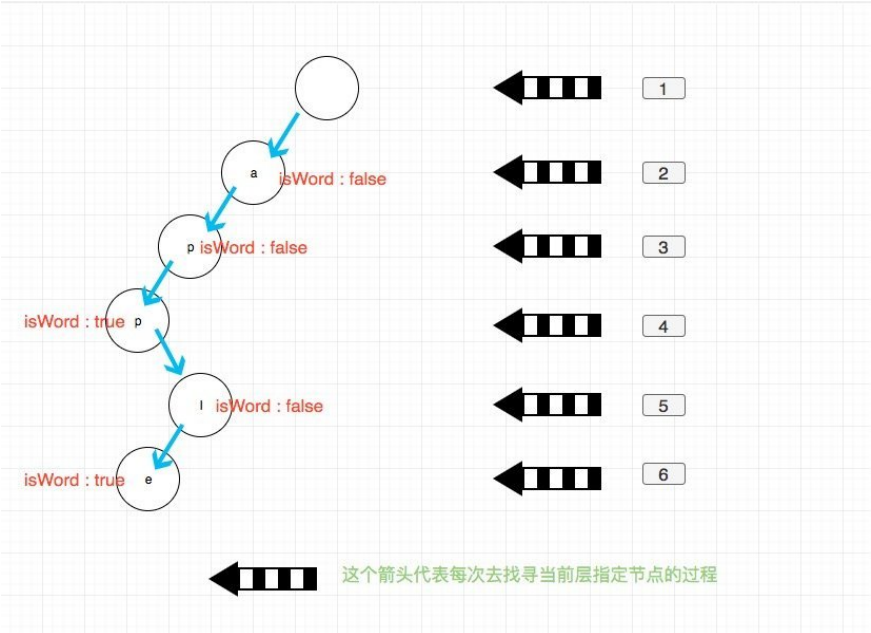
思路

我们首先不考虑字符"."的情况。这种情况比较简单，我们 addWord 直接添加到数组尾部，search 则线性查找即可。

接下来我们考虑特殊字符“.”，其实也不难，只不过 search 的时候，判断如果是“.”, 我们认为匹配到了，继续往后匹配即可。

上面的代码复杂度会比较高，我们考虑优化。如果你熟悉前缀树的话，应该注意到这可以使用前缀树来进行优化。前缀树优化之后每次查找复杂度是$O(h)$, 其中 h 是前缀树深度，也就是最长的字符串长度。

关于前缀树，LeetCode 有很多题目。有的是直接考察，让你实现一个前缀树，有的是间接考察，比如本题。前缀树代码见下方，大家之后可以直接当成前缀树的解题模板使用。



由于我们这道题需要考虑特殊字符".", 因此我们需要对标准前缀树做一点改造，insert 不做改变，我们只需要改变 search 即可，代码(Python 3)：

```
def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for i, w in enumerate(word):
            if w == '.':
                wizards = []
                for k in curr.keys():
                    if k == '#':
                        continue
                    wizards.append(self.search(word[:i] + k
                return any(wizards)
            if w not in curr:
                return False
            curr = curr[w]
        return "#" in curr
```

标准的前缀树搜索我也贴一下代码，大家可以对比一下：

```
def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                return False
            curr = curr[w]
        return "#" in curr
```

关键点

- 前缀树（也叫字典树），英文名 Trie（读作 tree 或者 try）

代码

- 语言支持：Python3

Python3 Code：

关于 Trie 的代码:

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for i, w in enumerate(word):
            if w == '.':
                wizards = []
                for k in curr.keys():
                    if k == '#':
                        continue
                    wizards.append(self.search(word[:i] + k
                return any(wizards)
            if w not in curr:
                return False
            curr = curr[w]
        return "#" in curr
```

主逻辑代码:

```
class WordDictionary:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = Trie()

    def addWord(self, word: str) -> None:
        """
        Adds a word into the data structure.
        """
        self.trie.insert(word)

    def search(self, word: str) -> bool:
        """
        Returns if the word is in the data structure. A wo
        """
        return self.trie.search(word)


# Your WordDictionary object will be instantiated and calle
# obj = WordDictionary()
# obj.addWord(word)
# param_2 = obj.search(word)
```

相关题目

- 0208.implement-trie-prefix-tree
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words

题目地址(215. 数组中的第K个最大元素)

https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

题目描述

```
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

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。
```

前置知识

- 堆
- Quick Select

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题要求在一个无序的数组中，返回第K大的数。根据时间复杂度不同，这题有3种不同的解法。

解法一 （排序）

很直观的解法就是给数组排序，这样求解第 `K` 大的数，就等于是从小到大排好序的数组的第 `（n-K）` 小的数 (n 是数组的长度）。

例如：

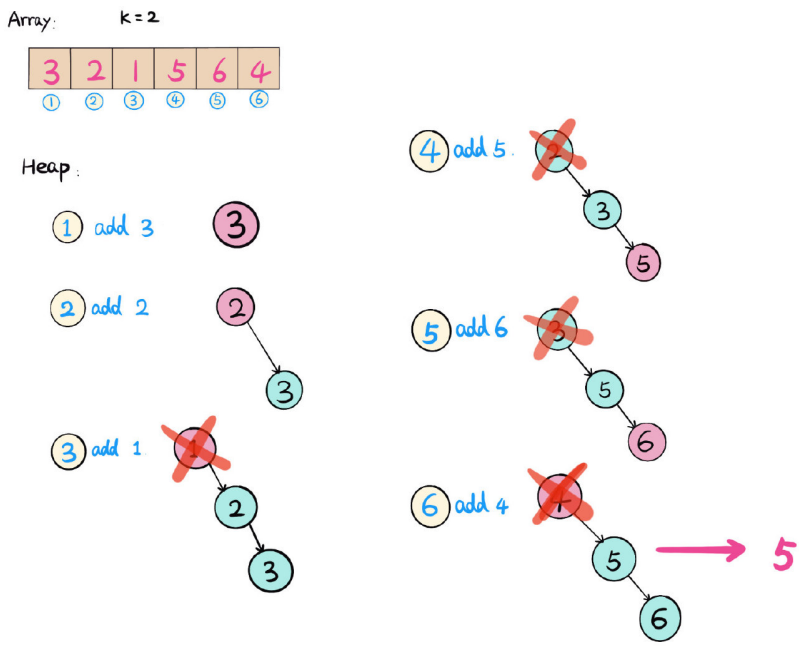
```
[3,2,1,5,6,4], k = 2
1. 数组排序：
 [1,2,3,4,5,6],
2. 找第（n-k）小的数
 n-k=4, nums[4]=5 （第2大的数）
```

时间复杂度: `O(nlogn) - n 是数组长度。`

解法二 - 小顶堆（Heap）

可以维护一个大小为 `K` 的小顶堆，堆顶是最小元素，当堆的 `size > K` 的时候，删除堆顶元素. 扫描一遍数组，最后堆顶就是第 `K` 大的元素。直接返回。

例如：



时间复杂度: `O(n * logk) , n is array length` *空间复杂度:* `O(k)`

跟排序相比，以空间换时间。

解法三 - Quick Select

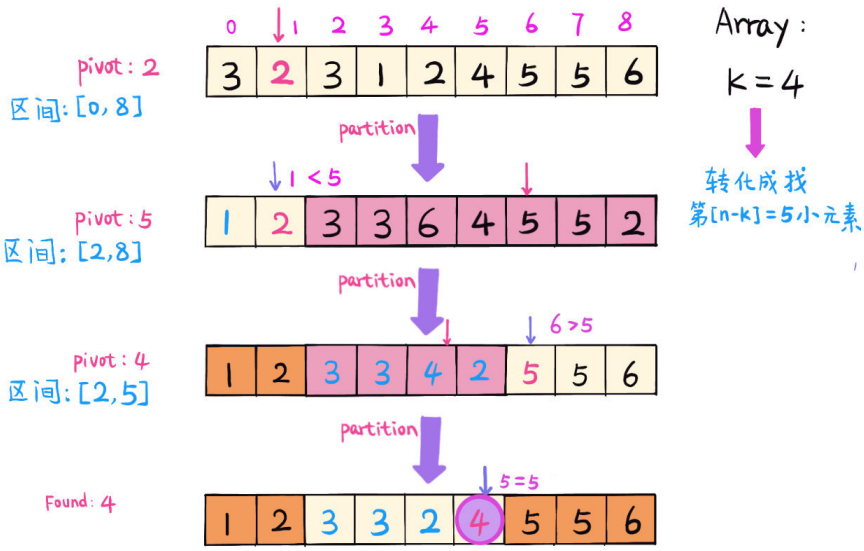
Quick Select 类似快排，选取pivot，把小于pivot的元素都移到pivot之前，这样pivot所在位置就是第pivot index 小的元素。 但是不需要完全给数组排序，只要找到当前pivot的位置是否是在第(n-k)小的位置，如果是，找到第k大的数直接返回。

具体步骤：

1. 在数组区间随机取`pivot index = left + random[right-left]`.
2. 根据pivot 做 partition，在数组区间，把小于pivot的数都移到pivot
3. 得到pivot的位置 index，`compare(index, (n-k))`.
    a. index == n-k -> 找到第`k`大元素，直接返回结果。
    b. index < n-k -> 说明在`index`右边，继续找数组区间`[index+
    c. index > n-k -> 那么第`k`大数在`index`左边，继续查找数组区

例子，【3,2,3,1,2,4,5,5,6]， k = 4

如下图：



时间复杂度:

- 平均是：`O(n)`
- 最坏的情况是：`O(n * n)`

关键点分析

1. 直接排序很简单
2. 堆（Heap）主要是要维护一个K大小的小顶堆，扫描一遍数组，最后堆顶元素即是所求。
3. Quick Select, 关键是是取pivot，对数组区间做partition，比较pivot的位置，类似二分，取pivot左边或右边继续递归查找。

代码（Java code）

解法一 - 排序

```
class KthLargestElementSort {
 public int findKthlargest2(int[] nums, int k) {
    Arrays.sort(nums);
    return nums[nums.length - k];
  }
}
```

解法二 - Heap （PriorityQueue）

```
class KthLargestElementHeap {
  public int findKthLargest(int[] nums, int k) {
      PriorityQueue<Integer> pq = new PriorityQueue<>();
      for (int num : nums) {
        pq.offer(num);
        if (pq.size() > k) {
          pq.poll();
        }
      }
      return pq.poll();
  }
}
```

解法三 - Quick Select

```
class KthLargestElementQuickSelect {
    static Random random = new Random();
    public int findKthLargest3(int[] nums, int k) {
      int len = nums.length;
      return select(nums, 0, len - 1, len - k);
    }

    private int select(int[] nums, int left, int right, int
      if (left == right) return nums[left];
      // random select pivotIndex between left and right
      int pivotIndex = left + random.nextInt(right - left);
      // do partition, move smaller than pivot number into
      int pos = partition(nums, left, right, pivotIndex);
      if (pos == k) {
        return nums[pos];
      } else if (pos > k) {
        return select(nums, left, pos - 1, k);
      } else {
        return select(nums, pos + 1, right, k);
      }
    }

    private int partition(int[] nums, int left, int right,
      int pivot = nums[pivotIndex];
      // move pivot to end
      swap(nums, right, pivotIndex);
      int pos = left;
      // move smaller num to pivot left
      for (int i = left; i <= right; i++) {
        if (nums[i] < pivot) {
          swap(nums, pos++, i);
        }
      }
      // move pivot to original place
      swap(nums, right, pos);
      return pos;
    }

    private void swap(int[] nums, int i, int j) {
      int tmp = nums[i];
      nums[i] = nums[j];
      nums[j] = tmp;
    }
}
```

参考（References）

1. Quick Select Wiki

题目地址(220. 存在重复元素 III)

https://leetcode-cn.com/problems/contains-duplicate-iii/

题目描述

```
在整数数组 nums 中，是否存在两个下标 i 和 j，使得 nums [i] 和 num

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
```

前置知识

- 哈希表

公司

- 暂无

暴力（超时）

思路

最简单的思路就是双层循环，找出所有的两两组合。然后逐个判断其是否满足 `nums [i] 和 nums [j] 的差的绝对值最大为 t，并且 i 和 j 之间的差的绝对值最大为 k`。

代码

```
class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int]
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                if abs(nums[i] - nums[j]) <= t and j - i  <
                    return True
        return False
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(1)$

暴力 + 剪枝 （超时）

思路

上述的内存循环可以稍微优化一下， 之前我们从 i + 1 到 len(nums)，实际上我们只需要 i + 1 到 min(len(nums), i + k + 1)。这样我们的 `j - i <= k` 也可以省略了。

代码

```
class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int]
        for i in range(len(nums)):
            for j in range(i + 1, min(len(nums), i + k + 1)
                if abs(nums[i] - nums[j]) <= t:
                    return True
        return False
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：$O(1)$

分桶 （通过）

思路

这道题是 219. 存在重复元素 II 的进阶版。那道题的条件是 `nums[i] == nums[j]`，而这道题则更加宽泛，是 `nums [i] 和 nums [j] 的差的绝对值小于等于 t` 。

这里我们介绍一种分桶的思想，其基本思想和桶排序是类似的。

具体来说，我们可使用 (t + 1) 个桶。将所有数除以 (t+1) 的结果**作为编号存到一个哈希表中**，不难知道哈希表的编号范围为 [0, t]，因此哈希表最大容量为 (t+1)。

经过这样的处理，如果两个数字的编号相同，那么意味着其绝对值差小于等于 t。

那么如果两个数字的编号不同，是否意味着其绝对值差大于 t 呢？也不一定，**相邻编号**也可能是绝对值差小于等于 t 。因此我们只需要检查以下三种情况即可。

1. 当前编号
2. 左边相邻的编号
3. 右边相邻的编号

另外由于题目限定是索引差小于等于 k，因此我们可以固定一个窗口大小为 k 的滑动窗口，每次都仅处理窗口内的元素，这样可以保证桶内的数任意两个数都满足**索引之差的绝对值小于等于 k**。 因此我们需要清除哈希表中过期（不在窗口内）的信息。

具体算法：

- 我们将数据分到 M 个桶 中。
- 每个数字 nums[i] 都被我们分配到一个桶中
- 分配的依据就是 nums[i] // (t + 1)
- 这样相邻桶内的数字最多相差 `2 * t + 1`
- 不相邻的桶一定不满足相差小于等于 t
- 同一个桶内的数字最多相差 `t`
- 因此如果命中同一个桶内，那么直接返回 True
- 如果命中相邻桶，我们再判断一下是否满足 相差 <= t
- 否则返回 False

需要注意的是，由于题目有索引相差 k 的要求，因此要维护一个大小为 k 的窗口，定期清除桶中 `过期` 的数字。

关键点

- 分桶排序思想的应用

代码

我们使用哈希表来模拟桶，key 就是桶号，value 就是数字本身。

Python 3:

```
class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int]
        bucket = dict()
        if t < 0: return False
        for i in range(len(nums)):
            nth = nums[i] // (t + 1)
            if nth in bucket:
                return True
            if nth - 1 in bucket and abs(nums[i] - bucket[n
                return True
            if nth + 1 in bucket and abs(nums[i] - bucket[n
                return True
            bucket[nth] = nums[i]
            if i >= k: bucket.pop(nums[i - k] // (t + 1))
        return False
```

C++

```
class Solution {
public:
    bool containsNearbyAlmostDuplicate(vector<int>& nums, 
        if(t<0) return false;
        //t+1可能会溢出，所以要+ 1LL
        long long mod = t + 1LL;
        unordered_map<long long,long long> buck;
        for(int i=0;i<nums.size();i++)
        {
            long long nth = nums[i] / mod;
            //可能nums[i]为负数，比如-4 / 5 以及 -4 / 5都等于0,
            if(nums[i] < 0) nth--;
            //这里要用find 不能直接[],因为可能本身存储的数字就为0
            if(buck.find(nth)!=buck.end())
                return true;
            else if(buck.find(nth-1)!=buck.end() && abs(nur
                return true;
            else if(buck.find(nth+1)!=buck.end() && abs(nur
                return true;
            buck[nth] = nums[i];
            if(i >= k)
            {
                buck.erase(nums[i - k] / mod);
            }
        }
        return false;
    }
};
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：由于过期的会被清除，因此哈希表大小不会大于 k，因此空间复杂度为 $O(min(n,k))$

扩展

实际上我们也可以一次遍历，并将遍历到的数字全部放到平衡二叉搜索树。这样我们只需要查找一下是否存在一个 x，满足 nums[i] - t <= x <= nums[i] + t 即可。

当然，我们仍然需要像方法三（桶排序）那样将过期的数排除。我们可以调用二叉平衡的删除方法。不过这种做法时间和空间并不优秀，给大家做扩展思路好了。

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(221. 最大正方形)

https://leetcode-cn.com/problems/maximal-square/

题目描述

```
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

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
```

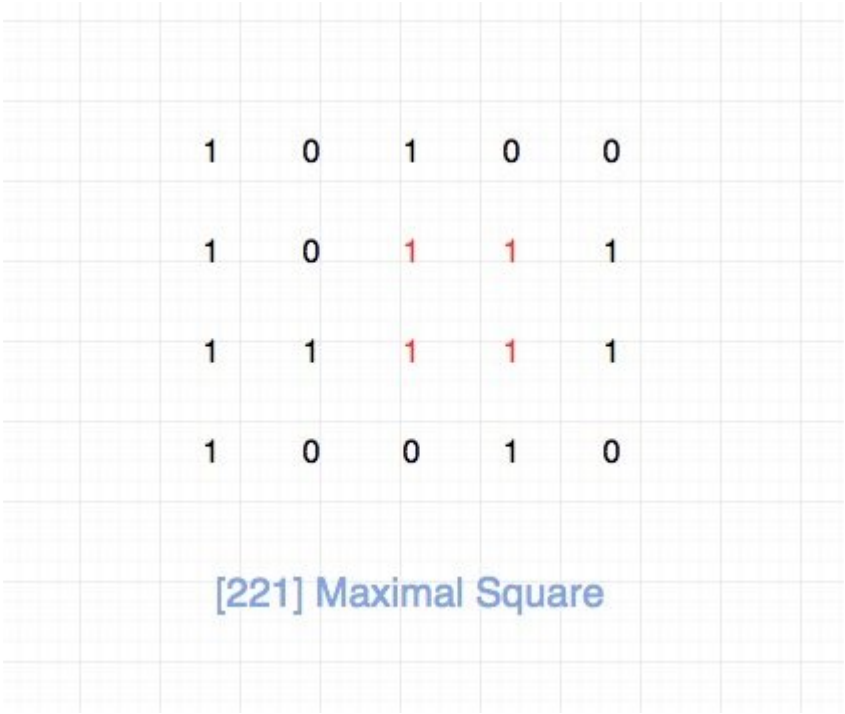
前置知识

- 动态规划
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

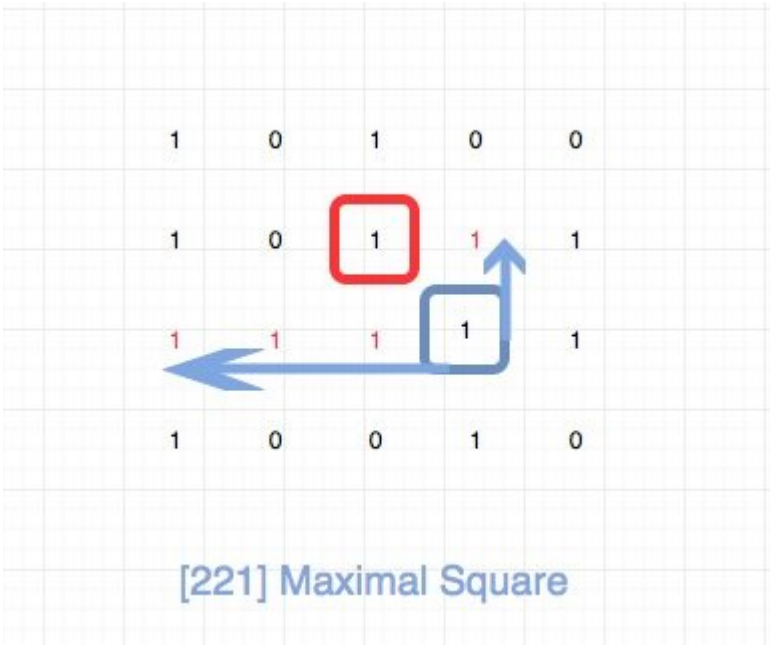


符合直觉的做法是暴力求解处所有的正方形，逐一计算面积，然后记录最大的。这种时间复杂度很高。

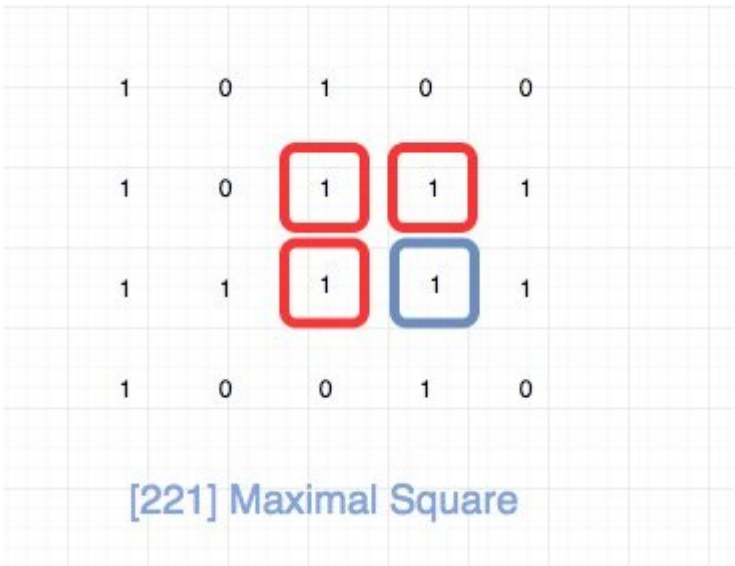
我们考虑使用动态规划，我们使用 dp[i][j]表示以 matrix[i][j]为右下角的顶点的可以组成的最大正方形的边长。 那么我们只需要计算所有的 i，j 组合，然后求出最大值即可。

我们来看下 dp[i][j] 怎么推导。 首先我们要看 matrix[i][j], 如果 matrix[i][j]等于 0，那么就不用看了，直接等于 0。 如果 matrix[i][j]等于 1，那么我们将 matrix[[i][j]分别往上和往左进行延伸，直到碰到一个 0 为止。

如图 dp[3][3] 的计算。 matrix[3][3]等于 1，我们分别往上和往左进行延伸，直到碰到一个 0 为止，上面长度为 1，左边为 3。 dp[2][2]等于 1（之前已经计算好了），那么其实这里的瓶颈在于三者的最小值, 即 `Min(1, 1, 3)` , 也就是 `1` 。 那么 dp[3][3] 就等于 `Min(1, 1, 3) + 1` 。



dp[i - 1][j - 1]我们直接拿到，关键是 `往上和往左进行延伸` ，最直观的做法是我们内层加一个循环去做就好了。 但是我们仔细观察一下，其实我们根本不需要这样算。 我们可以直接用 dp[i - 1][j]和 dp[i][j -1]。 具体就是 `Min(dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j]) + 1` 。



事实上，这道题还有空间复杂度 O(N)的解法，其中 N 指的是列数。 大家可以去这个leetcode 讨论看一下。

关键点解析

- DP
- 递归公式可以利用 dp[i - 1][j]和 dp[i][j -1]的计算结果，而不用重新计算
- 空间复杂度可以降低到 O(n), n 为列数

代码

代码支持：Python，JavaScript：

Python Code：

```python
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int
        res = 0
        m = len(matrix)
        if m == 0:
            return 0
        n = len(matrix[0])
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1],
                res = max(res, dp[i][j])
        return res ** 2
```

JavaScript Code：

```
/*
 * @lc app=leetcode id=221 lang=javascript
 *
 * [221] Maximal Square
 */
/**
 * @param {character[][]} matrix
 * @return {number}
 */
var maximalSquare = function (matrix) {
  if (matrix.length === 0) return 0;
  const dp = [];
  const rows = matrix.length;
  const cols = matrix[0].length;
  let max = Number.MIN_VALUE;

  for (let i = 0; i < rows + 1; i++) {
    if (i === 0) {
      dp[i] = Array(cols + 1).fill(0);
    } else {
      dp[i] = [0];
    }
  }

  for (let i = 1; i < rows + 1; i++) {
    for (let j = 1; j < cols + 1; j++) {
      if (matrix[i - 1][j - 1] === "1") {
        dp[i][j] = Math.min(dp[i - 1][j - 1], dp[i - 1][j],
        max = Math.max(max, dp[i][j]);
      } else {
        dp[i][j] = 0;
      }
    }
  }

  return max * max;
};
```

复杂度分析

- 时间复杂度：$O(M * N)$，其中 M 为行数，N 为列数。
- 空间复杂度：$O(M * N)$，其中 M 为行数，N 为列数。

题目地址(227. 基本计算器 II)

https://leetcode-cn.com/problems/basic-calculator-ii/

题目描述

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

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。

来源：力扣（LeetCode）
链接：https://leetcode-cn.com/problems/basic-calculator-ii
著作权归领扣网络所有。商业转载请联系官方授权，非商业转载请注明出处。

前置知识

- 栈

公司

- 暂无

思路

计算器的题目基本都和栈有关，这道题也不例外。

由题目信息可知，s 中一共包含以下几种数据：

- 空格
- 数字
- 操作符。这里有 + - * /

而对于操作符来说又可以进一步细分：

- 一元操作符 + -
- 二元操作符 * /

对于一元操作符来说，我们只需要知道一个操作数即可。这个操作数就是操作符右边的数字。为了达到这个效果，我们需要一点小小的 trick。

```
1 + 2
```

我们可以在前面补充一个 + 号，变成：

```
+ 1 + 2
# 可看成
(+1)(+2)
```

再比如:

```
(-1)(+2)(+3)(-4)
```

> 括号只是逻辑分组，实际并不存在。下同，不再赘述。

而对于二元操作符来说，我们需要知道两个操作数，这两个操作数分别是操作符两侧的两个数字。

```
(5) / (2)
```

再比如

```
(3) * (4)
```

简单来说就是，一元操作符绑定一个操作数。而二元操作符绑定两个操作数。

算法：

- 从左到右遍历 s
- 如果是数字，则更新数字
- 如果是空格，则跳过

- 如果是运算符，则按照运算符规则计算，并将计算结果重新入栈，具体见代码。最后更新 pre_flag 即可。

为了简化判断， 我使用了两个哨兵。一个是 s 末尾的 $，另一个是最开始的 pre_flag。

关键点解析

- 区分一目和二目运算符，并使用栈来简化操作
- 记录 pre_flag，即上一次出现的操作符
- 使用哨兵简化操作。一个是 s 的 $ ，另一个是 pre_flag 的 +

代码

代码支持：Python。

Python Code：

```python
class Solution:
    def calculate(self, s: str) -> int:
        stack = []
        s += '$'
        pre_flag = '+'
        num = 0

        for c in s:
            if c.isdigit():
                num = num * 10 + int(c)
            elif c == ' ': continue
            else:
                if pre_flag == '+':
                    stack.append(num)
                elif pre_flag == '-':
                    stack.append(-num)
                elif pre_flag == '*':
                    stack.append(stack.pop() * num)
                elif pre_flag == '/':
                    stack.append(int(stack.pop() / num))
                pre_flag = c
                num = 0
        return sum(stack)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

扩展

1. 基本计算器 和这道题差不多，官方难度困难。就是多了个括号而已。所以基本上可以看做是这道题的扩展。题目描述：

```
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

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。
```

拿题目中最难的例子来说 "(1+(4+5+2)-3)+(6+8)"。我们可以将其拆分为：

- 6+8 (= 14)
- 4 + 5 + 2 (=11)
- (11) - 3 (=8)
- 1 + (8) (=9)
- 9 + (14) (=23)

简单来说就是将括号里面的内容提取出来，提取出来就是上面的问题了。用上面的方法计算出结果，然后将结果作为一个数字替换原来的表达式。

比如我们先按照上面的算法计算出 6 + 8 的结果是 14，然后将 14 替换原来的 (6+8)，那么原问题就转化为了 `(1+(4+5+2)-3)+14` 。这样一步一步就可以得到答案。

因此我们可以使用递归，每次遇到 `(` 则开启一轮新的递归，遇到 `)` 则退出一层递归即可。

Python 代码：

```python
class Solution:
    def calculate(self, s: str) -> int:
        def dfs(s, start):
            stack = []
            pre_flag = '+'
            num = 0
            i = start
            while i < len(s):
                c = s[i]
                if  c == ' ':
                    i += 1
                    continue
                elif c == '(':
                    i, num = dfs(s, i+1)
                elif c.isdigit():
                    num = num * 10 + int(c)
                else:
                    if pre_flag == '+':
                        stack.append(num)
                    elif pre_flag == '-':
                        stack.append(-num)
                    if c == ')': break
                    pre_flag = c
                    num = 0
                i += 1
            return i, sum(stack)
        s += '$'
        return dfs(s, 0)[1]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

补充：一些同学反映：思路和我的一样，代码也类似，为什么执行不正确？这里我强调一点：

- 注意语句 `if c == ')': break` 的位置。如果放在其他位置，需要在其前手动增加语句，代码类似：

```python
if c == ')':
    if pre_flag == '+':
        stack.append(num)
    elif pre_flag == '-':
        stack.append(-num)
    break
```

以 "(1+(4+5+2)-3)+(6+8)" 来说，(4+5+2) 加起来就是 11，如果 break 前不执行上面的语句就会漏掉 2 变成 了 9，而不是 11。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(229. 求众数 II)

https://leetcode-cn.com/problems/majority-element-ii/

题目描述

```
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。

进阶：尝试设计时间复杂度为 O(n)、空间复杂度为 O(1)的算法解决此问题。



示例 1:

输入：[3,2,3]
输出：[3]
示例 2:

输入：nums = [1]
输出：[1]
示例 3:

输入：[1,1,1,3,3,2,2,2]
输出：[1,2]


提示：

1 <= nums.length <= 5 * 104
-10^9 <= nums[i] <= 10^9
```

前置知识

- 摩尔投票法

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题目和169.majority-element 很像。

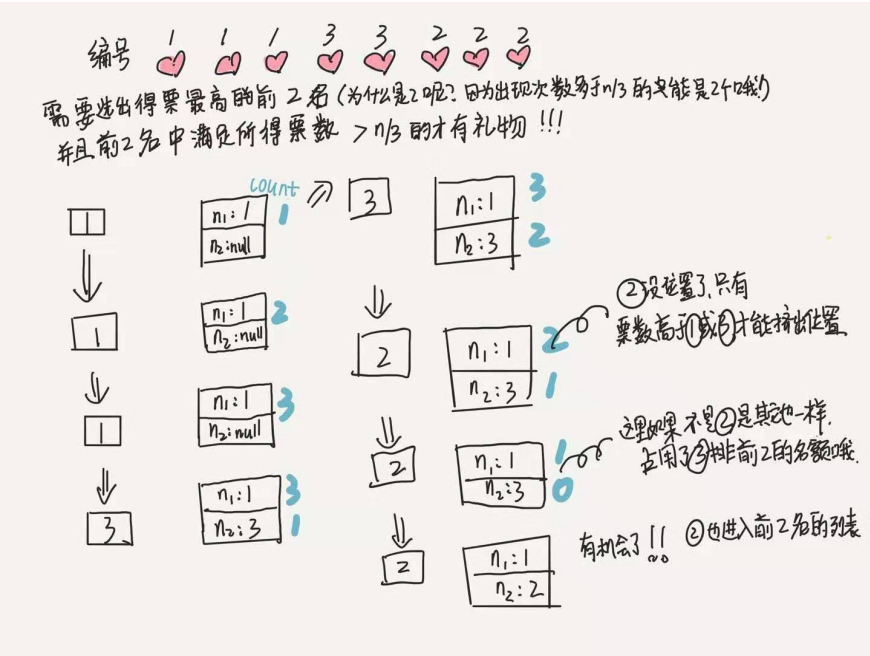
我们仍然可以采取同样的方法 - “摩尔投票法”， 具体的思路可以参考上面的题目。

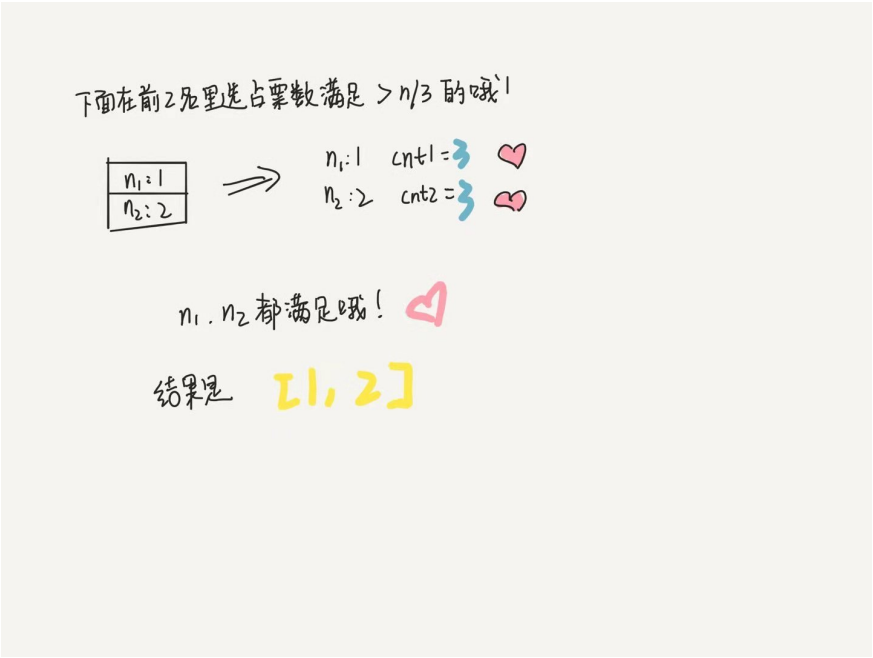
但是这里有一个不同的是这里的众数不再是超过 `1 / 2` ,而是超过 `1 / 3` 。 题目也说明了，超过三分之一的有可能有多个（实际上就是 0，1，2 三种可能）。

因此我们不能只用一个 counter 来解决了。 我们的思路是同时使用两个 counter，其他思路和上一道题目一样。

最后需要注意的是两个 counter 不一定都满足条件，这两个 counter 只是出现次数最多的两个数字。 有可能不满足出现次数大于 1/3， 因此最后我们需要进行过滤筛选。

这里画了一个图，大家可以感受一下：





关键点解析

- 摩尔投票法
- 两个 counter
- 最后得到的只是出现次数最多的两个数字，有可能不满足出现次数大于 1/3

代码

代码支持：CPP，JS

CPP Code:

```
class Solution {
public:
    vector<int> majorityElement(vector<int>& nums) {
        int c1 = 0, c2 = 0, v1 = 0, v2 = 1;
        for (int n : nums) {
            if (v1 == n) ++c1;
            else if (v2 == n) ++c2;
            else if (!c1) v1 = n, ++c1;
            else if (!c2) v2 = n, ++c2;
            else --c1, --c2;
        }
        c1 = c2 = 0;
        for (int n : nums) {
            if (v1 == n) ++c1;
            if (v2 == n) ++c2;
        }
        vector<int> v;
        if (c1 > nums.size() / 3) v.push_back(v1);
        if (c2 > nums.size() / 3) v.push_back(v2);
        return v;
    }
};
```

JS Code：

```
/*
 * @lc app=leetcode id=229 lang=javascript
 *
 * [229] Majority Element II
 */
/**
 * @param {number[]} nums
 * @return {number[]}
 */
var majorityElement = function (nums) {
  const res = [];
  const len = nums.length;
  let n1 = null,
    n2 = null,
    cnt1 = 0,
    cnt2 = 0;

  for (let i = 0; i < len; i++) {
    if (n1 === nums[i]) {
      cnt1++;
    } else if (n2 === nums[i]) {
      cnt2++;
    } else if (cnt1 === 0) {
      n1 = nums[i];
      cnt1++;
    } else if (cnt2 === 0) {
      n2 = nums[i];
      cnt2++;
    } else {
      cnt1--;
      cnt2--;
    }
  }

  cnt1 = 0;
  cnt2 = 0;

  for (let i = 0; i < len; i++) {
    if (n1 === nums[i]) {
      cnt1++;
    } else if (n2 === nums[i]) {
      cnt2++;
    }
  }

  if (cnt1 > (len / 3) >>> 0) {
    res.push(n1);
  }
```

```
  if (cnt2 > (len / 3) >>> 0) {
    res.push(n2);
  }

  return res;
};
```

Java 代码：

```java
/*
 * @lc app=leetcode id=229 lang=java
 *
 * [229] Majority Element II
 */
class Solution {
    public List<Integer> majorityElement(int[] nums) {
        List<Integer> res = new ArrayList<Integer>();
        if (nums == null || nums.length == 0)
            return res;
        int n1 = nums[0], n2 = nums[0], cnt1 = 0, cnt2 = 0,
        for (int i = 0; i < len; i++) {
            if (nums[i] == n1)
                cnt1++;
            else if (nums[i] == n2)
                cnt2++;
            else if (cnt1 == 0) {
                n1 = nums[i];
                cnt1 = 1;
            } else if (cnt2 == 0) {
                n2 = nums[i];
                cnt2 = 1;
            } else {
                cnt1--;
                cnt2--;
            }
        }
        cnt1 = 0;
        cnt2 = 0;
        for (int i = 0; i < len; i++) {
            if (nums[i] == n1)
                cnt1++;
            else if (nums[i] == n2)
                cnt2++;
        }
        if (cnt1 > len / 3)
            res.add(n1);
        if (cnt2 > len / 3 && n1 != n2)
            res.add(n2);
        return res;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

扩展

如果题目中 3 变成了 k，怎么解决?

大家可以自己思考一下，我这里给一个参考链接：
https://leetcode.com/problems/majority-element-ii/discuss/63500/JAVA-Easy-Version-To-Understand!!!!!!!!!!!!/64925

这个实现说实话不是很好，大家可以优化一下。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(230. 二叉搜索树中第 K 小的元素)

https://leetcode-cn.com/problems/kth-smallest-element-in-a-bst/

题目描述

```
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

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
```

前置知识

- 中序遍历

公司

- 阿里
- 腾讯

- 百度
- 字节

思路

解法一：

由于‘中序遍历一个二叉查找树（BST）的结果是一个有序数组’，因此我们只需要在遍历到第 k 个，返回当前元素即可。 中序遍历相关思路请查看binary-tree-traversal

解法二：

联想到二叉搜索树的性质，root 大于左子树，小于右子树，如果左子树的节点数目等于 K-1，那么 root 就是结果，否则如果左子树节点数目小于 K-1，那么结果必然在右子树，否则就在左子树。 因此在搜索的时候同时返回节点数目，跟 K 做对比，就能得出结果了。

关键点解析

- 中序遍历

代码

解法一：

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=230 lang=javascript
 *
 * [230] Kth Smallest Element in a BST
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function (root, k) {
  const stack = [root];
  let cur = root;
  let i = 0;

  function insertAllLefts(cur) {
    while (cur && cur.left) {
      const l = cur.left;
      stack.push(l);
      cur = l;
    }
  }
  insertAllLefts(cur);

  while ((cur = stack.pop())) {
    i++;
    if (i === k) return cur.val;
    const r = cur.right;

    if (r) {
      stack.push(r);
      insertAllLefts(r);
    }
  }

  return -1;
};
```

Java Code:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
private int count = 1;
private int res;

public int KthSmallest (TreeNode root, int k) {
    inorder(root, k);
    return res;
}

public void inorder (TreeNode root, int k) {
    if (root == null) return;

    inorder(root.left, k);

    if (count++ == k) {
        res = root.val;
        return;
    }

    inorder(root.right, k);
}
```

解法二:

JavaScript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function nodeCount(node) {
  if (node === null) return 0;

  const l = nodeCount(node.left);
  const r = nodeCount(node.right);

  return 1 + l + r;
}
/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function (root, k) {
  const c = nodeCount(root.left);
  if (c === k - 1) return root.val;
  else if (c < k - 1) return kthSmallest(root.right, k - c
  return kthSmallest(root.left, k);
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

扩展

这道题有一个 follow up：

`What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?`

建议大家思考一下。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(236. 二叉树的最近公共祖先)

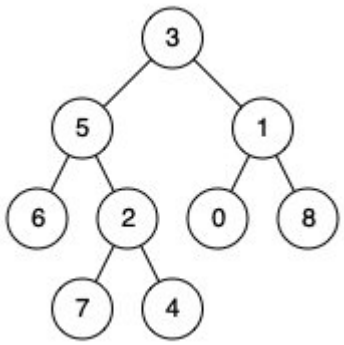
https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/

题目描述

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

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

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。

前置知识

- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

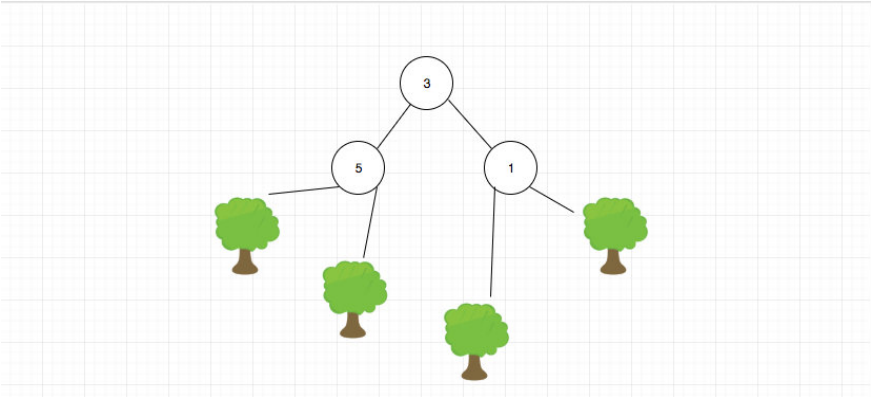
这道题目是求解二叉树中，两个给定节点的最近的公共祖先。是一道非常经典的二叉树题目。

我们之前说过树是一种递归的数据结构，因此使用递归方法解决二叉树问题从写法上来看是最简单的，这道题目也不例外。

用递归的思路去思考树是一种非常重要的能力。

如果大家这样去思考的话，问题就会得到简化，我们的目标就是分别在左右子树进行查找 p 和 q。 如果 p 没有在左子树，那么它一定在右子树（题目限定 p 一定在树中）， 反之亦然。

对于具体的代码而言就是，我们假设这个树就一个结构，然后尝试去解决，然后在适当地方去递归自身即可。 如下图所示：



我们来看下核心代码：

```
// 如果我们找到了p，直接进行返回，那如果下面就是q呢？ 其实这没有影响，
if (!root || root === p || root === q) return root;
const left = lowestCommonAncestor(root.left, p, q); // 去左
const right = lowestCommonAncestor(root.right, p, q); // 去
if (!left) return right; // 左子树找不到，返回右子树
if (!right) return left; // 右子树找不到，返回左子树
return root; // 左右子树分别有一个，则返回root
```

> 如果没有明白的话，请多花时间消化一下

关键点解析

- 用递归的思路去思考树

代码

代码支持： JavaScript， Python3

- JavaScript Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
var lowestCommonAncestor = function (root, p, q) {
  if (!root || root === p || root === q) return root;
  const left = lowestCommonAncestor(root.left, p, q);
  const right = lowestCommonAncestor(root.right, p, q);
  if (!left) return right; // 左子树找不到，返回右子树
  if (!right) return left; // 右子树找不到，返回左子树
  return root; // 左右子树分别有一个，则返回root
};
```

- Python Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'Tr
        if not root or root == p or root == q:
            return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        if not left:
            return right
        if not right:
            return left
        else:
            return root
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

扩展

如果递归的结束条件改为 `if (!root || root.left === p || root.right === q) return root;` 代表的是什么意思，对结果有什么样的影响?

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（238. 除自身以外数组的乘积）

https://leetcode-cn.com/problems/product-of-array-except-self/

题目描述

```
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



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

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，
```

前置知识

- 数组

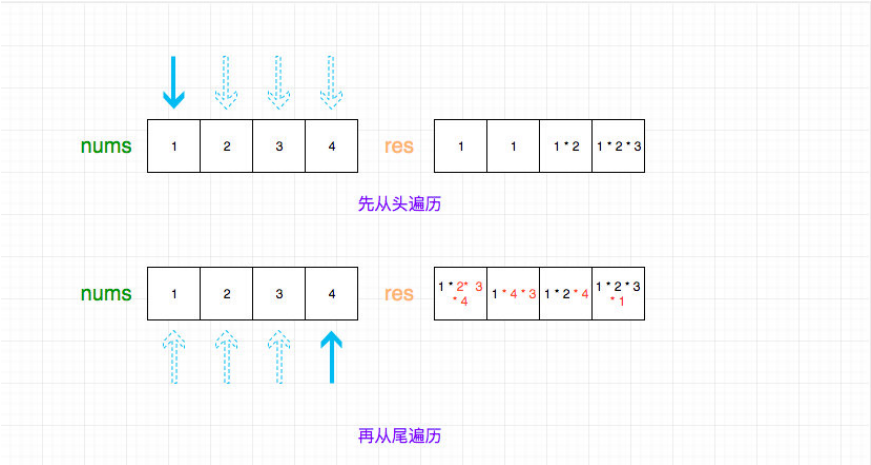
公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这道题的意思是给定一个数组，返回一个新的数组，这个数组每一项都是其他项的乘积。 符合直觉的思路是两层循环，时间复杂度是 O(n^2),但是题目要求 `Please solve it without division and in O(n)` 。

因此我们需要换一种思路，由于输出的每一项都需要用到别的元素，因此一次遍历是绝对不行的。 考虑我们先进行一次遍历， 然后维护一个数组，第 i 项代表前 i 个元素（不包括 i）的乘积。 然后我们反向遍历一次，然后维护另一个数组，同样是第 i 项代表前 i 个元素（不包括 i）的乘积。



有意思的是第一个数组和第二个数组的反转（reverse）做乘法（有点像向量运算）就是我们想要的运算。

其实我们进一步观察，我们不需要真的创建第二个数组（第二个数组只是做中间运算使用），而是直接修改第一个数组即可。

关键点解析

- 两次遍历， 一次正向，一次反向。
- 维护一个数组，第 i 项代表前 i 个元素（不包括 i）的乘积

代码

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */
var productExceptSelf = function (nums) {
  const ret = [];

  for (let i = 0, temp = 1; i < nums.length; i++) {
    ret[i] = temp;
    temp *= nums[i];
  }
  // 此时ret[i]存放的是前i个元素相乘的结果(不包含第i个)

  // 如果没有上面的循环的话,
  // ret经过下面的循环会变成ret[i]存放的是后i个元素相乘的结果(不包含

  // 我们的目标是ret[i]存放的所有数字相乘的结果(不包含第i个)

  // 因此我们只需要对于上述的循环产生的ret[i]基础上运算即可
  for (let i = nums.length - 1, temp = 1; i >= 0; i--) {
    ret[i] *= temp;
    temp *= nums[i];
  }
  return ret;
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

题目地址(240. 搜索二维矩阵 II)

https://leetcode-cn.com/problems/search-a-2d-matrix-ii/

题目描述

```
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。

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。
```

前置知识

- 数组

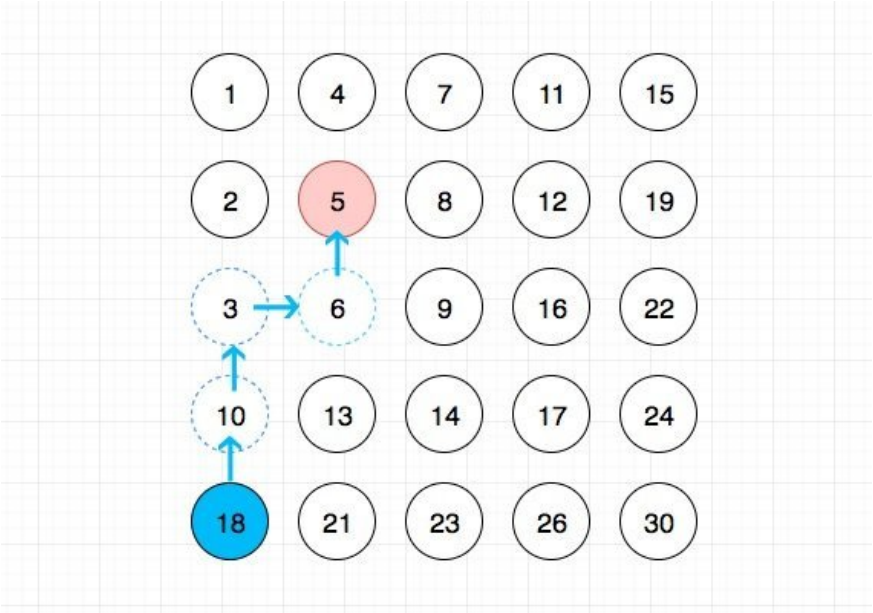
公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的做法是两层循环遍历，时间复杂度是 O(m * n), 有没有时间复杂度更好的做法呢？ 答案是有，那就是充分运用矩阵的特性（横向纵向都递增）， 我们可以从角落（左下或者右上）开始遍历，这样时间复杂

度是 O(m + n).



其中蓝色代表我们选择的起点元素， 红色代表目标元素。

关键点解析

- 从角落开始遍历，利用递增的特性简化时间复杂

代码

代码支持：JavaScript, Python3

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=240 lang=javascript
 *
 * [240] Search a 2D Matrix II
 *
 * https://leetcode.com/problems/search-a-2d-matrix-ii/desc
 *
 *
 */
/**
 * @param {number[][]} matrix
 * @param {number} target
 * @return {boolean}
 */
var searchMatrix = function (matrix, target) {
  if (!matrix || matrix.length === 0) return false;

  let colIndex = 0;
  let rowIndex = matrix.length - 1;
  while (rowIndex > 0 && target < matrix[rowIndex][colIndex
    rowIndex--;
  }

  while (colIndex < matrix[0].length) {
    if (target === matrix[rowIndex][colIndex]) return true;
    if (target > matrix[rowIndex][colIndex]) {
      colIndex++;
    } else if (rowIndex > 0) {
      rowIndex--;
    } else {
      return false;
    }
  }

  return false;
};
```

Python Code:

```python
class Solution:
    def searchMatrix(self, matrix, target):
        m = len(matrix)
        if m == 0:
            return False
        n = len(matrix[0])
        i = m - 1
        j = 0

        while i >= 0 and j < n:
            if matrix[i][j] == target:
                return True
            if matrix[i][j] > target:
                i -= 1
            else:
                j += 1
        return False
```

复杂度分析

- 时间复杂度：$O(M + N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(279. 完全平方数)

https://leetcode-cn.com/problems/perfect-squares/

题目描述

```
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

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.
```

前置知识

- 递归
- 动态规划

公司

- 阿里
- 百度
- 字节

思路

直接递归处理即可，但是这种暴力的解法很容易超时。如果你把递归的过程化成一棵树的话（其实就是递归树）， 可以看出中间有很多重复的计算。

如果能将重复的计算缓存下来，说不定能够解决时间复杂度太高的问题。

> 递归对内存的要求也很高， 如果数字非常大，也会面临爆栈的风险，将递归转化为循环可以解决。

递归 + 缓存的方式代码如下：

```
const mapper = {};

function d(n, level) {
  if (n === 0) return level;

  let i = 1;
  const arr = [];

  while (n - i * i >= 0) {
    const hit = mapper[n - i * i];
    if (hit) {
      arr.push(hit + level);
    } else {
      const depth = d(n - i * i, level + 1) - level;
      mapper[n - i * i] = depth;
      arr.push(depth + level);
    }
    i++;
  }

  return Math.min(...arr);
}
/**
 * @param {number} n
 * @return {number}
 */
var numSquares = function (n) {
  return d(n, 0);
};
```

如果使用 DP，其实本质上和递归 + 缓存 差不多。

DP 的代码见代码区。

关键点解析

- 如果用递归 + 缓存， 缓存的设计很重要 我的做法是 key 就是 n, value 是以 n 为起点，到达底端的深度。 下次取出缓存的时候用当前的 level + 存的深度 就是我们想要的 level.
- 使用动态规划的核心点还是选和不选的问题

```
for (let i = 1; i <= n; i++) {
  for (let j = 1; j * j <= i; j++) {
    // 不选 (dp[i])  还是  选 (dp[i - j * j])
    dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
  }
}
```

代码

代码支持：CPP，JS

CPP Code:

```
class Solution {
public:
    int numSquares(int n) {
        static vector<int> dp{0};
        while (dp.size() <= n) {
            int m = dp.size(), minVal = INT_MAX;
            for (int i = 1; i * i <= m; ++i) minVal = min(m
            dp.push_back(minVal);
        }
        return dp[n];
    }
};
```

JS Code:

```
/**
 * @param {number} n
 * @return {number}
 */
var numSquares = function (n) {
  if (n <= 0) {
    return 0;
  }

  const dp = Array(n + 1).fill(Number.MAX_VALUE);
  dp[0] = 0;
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j * j <= i; j++) {
      // 不选 (dp[i])  还是  选 (dp[i - j * j])
      dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
    }
  }

  return dp[n];
};
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(309. 最佳买卖股票时机含冷冻期)

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/

题目描述

```
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

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]
```

前置知识

- 动态规划

公司

- 阿里
- 腾讯
- 字节

思路

这是一道典型的 DP 问题， DP 问题的核心是找到状态和状态转移方程。

这道题目的状态似乎比我们常见的那种 DP 问题要多，这里的状态有 buy sell cooldown 三种， 我们可以用三个数组来表示这这三个状态，buy,sell, cooldown.

- buy[i]表示第 i 天，且以 buy 结尾的最大利润
- sell[i]表示第 i 天，且以 sell 结尾的最大利润
- cooldown[i]表示第 i 天，且以 sell 结尾的最大利润

我们思考一下，其实 cooldown 这个状态数组似乎没有什么用，因此 cooldown 不会对 `profit` 产生 任何影响。 我们可以进一步缩小为两种状态。

- buy[i] 表示第 i 天，且以 buy 或者 coolwown 结尾的最大利润
- sell[i] 表示第 i 天，且以 sell 或者 cooldown 结尾的最大利润

对应的状态转移方程如下：

> 这个需要花点时间来理解

```
buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
```

我们来分析一下，buy[i]对应第 i 的 action 只能是 buy 或者 cooldown。

- 如果是 cooldown，那么 profit 就是 buy[i - 1]
- 如果是 buy，那么就是 `前一个卖的profit减去今天买股票花的钱` ，即 sell[i -2] - prices[i]

> 注意这里是 i - 2，不是 i-1 ，因为有 cooldown 的限制

sell[i]对应第 i 的 action 只能是 sell 或者 cooldown。

- 如果是 cooldown，那么 profit 就是 sell[i - 1]
- 如果是 sell，那么就是 `前一次买的时候获取的利润加上这次卖的钱` ，即 buy[i - 1] + prices[i]

关键点解析

- 多状态动态规划

代码

```
/*
 * @lc app=leetcode id=309 lang=javascript
 *
 * [309] Best Time to Buy and Sell Stock with Cooldown
 *
 */
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function (prices) {
  if (prices == null || prices.length <= 1) return 0;

  // 定义状态变量
  const buy = [];
  const sell = [];
  // 寻常
  buy[0] = -prices[0];
  buy[1] = Math.max(-prices[0], -prices[1]);
  sell[0] = 0;
  sell[1] = Math.max(0, prices[1] - prices[0]);
  for (let i = 2; i < prices.length; i++) {
    // 状态转移方程
    // 第i天只能是买或者cooldown
    // 如果买利润就是sell[i - 2] - prices[i], 注意这里是i - 2,
    // cooldown就是buy[i -1]
    buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
    // 第i天只能是卖或者cooldown
    // 如果卖利润就是buy[i  -1] + prices[i]
    // cooldown就是sell[i -1]
    sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
  }

  return Math.max(buy[prices.length - 1], sell[prices.length
};
```

相关题目

- 121.best-time-to-buy-and-sell-stock
- 122.best-time-to-buy-and-sell-stock-ii

题目地址(322. 零钱兑换)

https://leetcode-cn.com/problems/coin-change/

题目描述

```
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

你可以认为每种硬币的数量是无限的。



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
示例 3:

输入: coins = [1], amount = 0
输出: 0
示例 4:

输入: coins = [1], amount = 1
输出: 1
示例 5:

输入: coins = [1], amount = 2
输出: 2


提示:

1 <= coins.length <= 12
1 <= coins[i] <= 231 - 1
0 <= amount <= 104
```

前置知识

- 贪心算法

- 动态规划

公司

- 腾讯
- 百度
- 字节
- 阿里巴巴（盒马生鲜）

岗位信息

- 阿里巴巴（盒马生鲜）：前端技术二面

思路

思路

假如我们把 coin 逆序排列，然后从面值大的开始取，如果取了当前硬币后金额仍有小于 amount，则继续取。

举个例子：

```
eg: 对于 [1,2,5] 组成 11 块

- 排序[5,2,1]

- 取第一个5, 更新amout 为 11 - 5 = 6 (1口)
      6 > 5 继续更新 为 6 - 5 = 1 (2口)
      1 < 5 退出

- 取第二个2
      1 < 2 退出

- 取最后一个元素，也就是1

      1 === 1 更新为 1 - 1 = 0 (3口)

- amout 为 0 退出


因此结果是 3
```

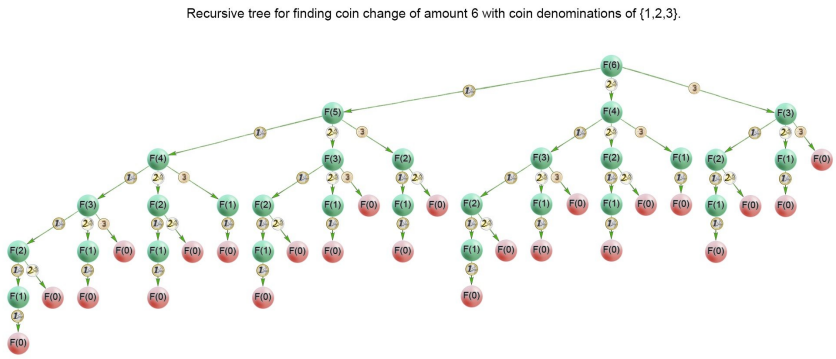
熟悉贪心算法的同学应该已经注意到了，这就是贪心算法，贪心算法想要使得 amount **尽快地变得更小**。

贪心算法通常时间复杂度更低，但对这道题目来说，贪心是不正确的！要证明它的错误，只需要随便举一个反例即可。 比如 `coins = [1, 5, 11] amout = 15` , 使用贪心就会得到错误的结果。 因此这种做法有时候不靠谱，我们还是采用靠谱的做法.

如果我们暴力求解，对于所有的组合都计算一遍，然后比较， 那么这样的复杂度是 2 的 n 次方（这个可以通过数学公式证明，这里不想啰嗦了），这个是不可以接受的。

暴力法枚举过程实际上有很多重复子问题，我们一般称重叠子问题。对于重叠子问题，我们可以使用备忘录来解决。

以 coins = [1,2,3], amount = 6 来说，我们可以画出如下的递归树。



（图片来自https://leetcode.com/problems/coin-change/solution/）

如上图 F(1) 被重复计算了 13 次！！如何消除重叠子问题？答案是记忆化递归或者动态规划，二者没有本质区别。

这里以自底向上的动态规划为例讲解一下。比较容易想到的是二维数组存放 F(n) 。

定义 dp[i][j] 为使用 coins 的前 j 项组成 金额 i 的最少硬币数。对于动态规划问题，最关键的是决策（不包含决策的是递推式动态规划）。对于动态规划的决策的技巧就是：仅关心最后一步和前一步，不考虑其他部分是如何达成的。

对于这道题来说，最后一步就是选择第 j 个硬币还是不选择第 j 个硬币。

- 如果选择第 j 个硬币，那么 dp[i][j] = min(dp[i][j - 1], dp[i - coins[j - 1]][j] + 1)

> 注意：dp[i - coins[j - 1]][j] 含义是硬币无限取， dp[i - coins[j - 1]][j - 1] 的含义就变成了硬币最多取一次

- 否则 dp[i][j] = dp[i][j - 1]

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        if amount < 0:
            return - 1
        dp = [[amount + 1 for _ in range(len(coins) + 1)]
              for _ in range(amount + 1)]

        # 初始化第一行为0，其他为最大值（也就是amount + 1）
        for j in range(len(coins) + 1):
            dp[0][j] = 0

        for i in range(1, amount + 1):
            for j in range(1, len(coins) + 1):
                # 注意：dp[i - coins[j - 1]][j] 含义是硬币无限
                if i - coins[j - 1] >= 0:
                    dp[i][j] = min(
                        dp[i][j - 1], dp[i - coins[j - 1]]
                    else:
                        dp[i][j] = dp[i][j - 1]

        return -1 if dp[-1][-1] == amount + 1 else dp[-1][-
```

复杂度分析

- 时间复杂度：$O(amonut * len(coins))$
- 空间复杂度：$O(amount * len(coins))$

dp[i][j] 依赖于 `dp[i][j - 1]` 和 `dp[i - coins[j - 1]][j] + 1)` 这是一个优化的信号，我们可以将其优化到一维。

用 dp[i] 来表示组成 i 块钱，需要最少的硬币数，那么

1. 第 j 个硬币我可以选择不拿 这个时候， 硬币数 = dp[i]
2. 第 j 个硬币我可以选择拿 这个时候， 硬币数 = dp[i - coins[j]] + 1

和 01 背包问题不同， 硬币是可以拿任意个，对于每一个 dp[i] 我们都选择遍历一遍 coin， 不断更新 dp[i]

关键点解析

- 分析出是典型的完全背包问题

代码

- 语言支持：JS，C++,Python3

JavaScript Code：

```
var coinChange = function (coins, amount) {
  if (amount === 0) {
    return 0;
  }
  const dp = Array(amount + 1).fill(Number.MAX_VALUE);
  dp[0] = 0;
  for (let i = 1; i < dp.length; i++) {
    for (let j = 0; j < coins.length; j++) {
      if (i - coins[j] >= 0) {
        dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
      }
    }
  }

  return dp[dp.length - 1] === Number.MAX_VALUE ? -1 : dp[
};
```

C++ Code：

> C++中采用 INT_MAX，因此判断时需要加上 `dp[a - coin] < INT_MAX` 以防止溢出

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        auto dp = vector<int>(amount + 1, INT_MAX);
        dp[0] = 0;
        for (auto a = 1; a <= amount; ++a) {
            for (const auto & coin : coins) {
                if (a >= coin && dp[a - coin] < INT_MAX) {
                    dp[a] = min(dp[a], dp[a-coin] + 1);
                }
            }
        }
        return dp[amount] == INT_MAX ? -1 : dp[amount];
    }
};
```

Python3 Code:

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) ->
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0

        for i in range(1, amount + 1):
            for j in range(len(coins)):
                if i >= coins[j]:
                    dp[i] = min(dp[i], dp[i - coins[j]] + 1

        return -1 if dp[-1] == amount + 1 else dp[-1]
```

复杂度分析

- 时间复杂度：$O(amonut * len(coins))$
- 空间复杂度：$O(amount)$

扩展

这是一道很简单描述的题目， 因此很多时候会被用到大公司的电面中。

相似问题:

518.coin-change-2

题目地址(328. 奇偶链表)

https://leetcode-cn.com/problems/odd-even-linked-list/

题目描述

```
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

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

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。
```

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的想法是，先遍历一遍找出奇数的节点。然后再遍历一遍找出偶数节点，最后串起来。

但是有两个问题，如果不修改节点的话，需要借助额外的空间，空间复杂度是 N。如果修改的话，会对第二次遍历（遍历偶数节点）造成影响。

因此可以采用一种做法： 遍历一次，每一步同时修改两个节点就好了，这样就可以规避上面两个问题。

关键点解析

- 用虚拟节点来简化操作
- 循环的结束条件设置为 `odd && odd.next && even && even.next` ，不应该是 `odd && even` ，否则需要记录一下奇数节点的最后一个节点，复杂了操作

代码

- 语言支持：JS，C++

JavaScript Code:

```javascript
/*
 * @lc app=leetcode id=328 lang=javascript
 *
 * [328] Odd Even Linked List
 *
 *
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var oddEvenList = function (head) {
  if (!head || !head.next) return head;

  const dummyHead1 = {
    next: head,
  };
  const dummyHead2 = {
    next: head.next,
  };

  let odd = dummyHead1.next;
  let even = dummyHead2.next;

  while (odd && odd.next && even && even.next) {
    const oddNext = odd.next.next;
    const evenNext = even.next.next;

    odd.next = oddNext;
    even.next = evenNext;

    odd = oddNext;
    even = evenNext;
  }

  odd.next = dummyHead2.next;

  return dummyHead1.next;
};
```

C++ Code：

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if (head == nullptr) return head;
        auto odd = head, evenHead = head->next, even = head
        // 因为“每次循环之后依然保持odd在even之前”，循环条件可以只
        while (even != nullptr && even->next != nullptr) {
            odd->next = even->next;
            odd = odd->next;
            even->next = odd->next;
            even = even->next;
        }
        odd->next = evenHead;
        return head;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(334. 递增的三元子序列)

https://leetcode-cn.com/problems/increasing-triplet-subsequence/

题目描述

```
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
```

前置知识

- 双指针

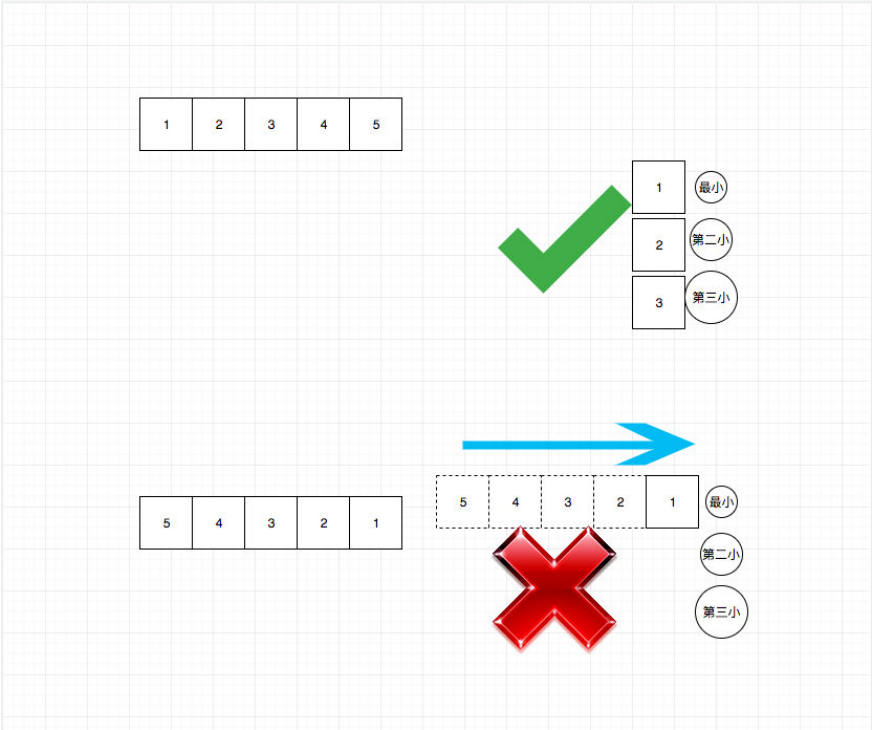
公司

- 百度
- 字节

思路

这道题是求解顺序数字是否有三个递增的排列， 注意这里没有要求连续的，因此诸如滑动窗口的思路是不可以的。 题目要求 O(n)的时间复杂度和 O(1)的空间复杂度，因此暴力的做法就不用考虑了。

我们的目标就是 依次 找到三个数字，其顺序是递增的。因此我们的做法可以是依次遍历， 然后维护三个变量，分别记录最小值，第二小值，第三小值。只要我们能够填满这三个变量就返回 true，否则返回 false。



关键点解析

- 维护三个变量，分别记录最小值，第二小值，第三小值。只要我们能够填满这三个变量就返回 true，否则返回 false

代码

代码支持： JS

JS Code：

```
/*
/**
 * @param {number[]} nums
 * @return {boolean}
 */
var increasingTriplet = function (nums) {
  if (nums.length < 3) return false;
  let n1 = Number.MAX_VALUE;
  let n2 = Number.MAX_VALUE;

  for (let i = 0; i < nums.length; i++) {
    if (nums[i] <= n1) {
      n1 = nums[i];
    } else if (nums[i] <= n2) {
      n2 = nums[i];
    } else {
      return true;
    }
  }

  return false;
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(337. 打家劫舍 III)

https://leetcode-cn.com/problems/house-robber-iii/

题目描述

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。

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.

前置知识

- 二叉树
- 动态规划

公司

- 阿里

- 腾讯
- 百度
- 字节

思路

和 198.house-robber 类似，这道题也是相同的思路。 只不过数据结构从数组换成了树。

我们仍然是对每一项进行决策：**如果我抢的话，所得到的最大价值是多少。如果我不抢的话，所得到的最大价值是多少。**

- 遍历二叉树，都每一个节点我们都需要判断抢还是不抢。
  - 如果抢了的话， 那么我们不能继续抢其左右子节点
  - 如果不抢的话，那么我们可以继续抢左右子节点，当然也可以不抢。抢不抢取决于哪个价值更大。
- 抢不抢取决于哪个价值更大。

这是一个明显的递归问题，我们使用递归来解决。由于没有重复子问题，因此没有必要 cache ， 也没有必要动态规划。

关键点

- 对每一个节点都分析，是抢还是不抢

代码

语言支持：JS, C++，Java，Python

JavaScript Code：

```
function helper(root) {
  if (root === null) return [0, 0];
  // 0: rob 1: notRob
  const l = helper(root.left);
  const r = helper(root.right);

  const robed = root.val + l[1] + r[1];
  const notRobed = Math.max(l[0], l[1]) + Math.max(r[0], r

  return [robed, notRobed];
}
/**
 * @param {TreeNode} root
 * @return {number}
 */
var rob = function (root) {
  const [robed, notRobed] = helper(root);
  return Math.max(robed, notRobed);
};
```

C++ Code:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {
 * };
 */
class Solution {
public:
    int rob(TreeNode* root) {
        pair<int, int> res = dfs(root);
        return max(res.first, res.second);
    }

    pair<int, int> dfs(TreeNode* root)
    {
        pair<int, int> res = {0, 0};
        if(root == NULL)
        {
            return res;
        }

        pair<int, int> left = dfs(root->left);
        pair<int, int> right = dfs(root->right);
        // 0 代表不偷，1 代表偷
        res.first = max(left.first, left.second) + max(righ
        res.second = left.first + right.first + root->val;
        return res;
    }

};
```

Java Code:

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int rob(TreeNode root) {
        int[] res = dfs(root);
        return Math.max(res[0], res[1]);
    }

    public int[] dp(TreeNode root)
    {
        int[] res = new int[2];
        if(root == null)
        {
            return res;
        }

        int[] left = dfs(root.left);
        int[] right = dfs(root.right);
        // 0 代表不偷，1 代表偷
        res[0] = Math.max(left[0], left[1]) + Math.max(righ
        res[1] = left[0] + right[0] + root.val;
        return res;
    }
}
```

Python Code:

```
class Solution:
    def rob(self, root: TreeNode) -> int:
        def dfs(node):
            if not node:
                return [0, 0]
            [l_rob, l_not_rob] = dfs(node.left)
            [r_rob, r_not_rob] = dfs(node.right)
            return [node.val + l_not_rob + r_not_rob, max(
        return max(dfs(root))


# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点个数。
- 空间复杂度：$O(h)$，其中 h 为树的高度。

相关题目

- 198.house-robber

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(343. 整数拆分)

https://leetcode-cn.com/problems/integer-break/

题目描述

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。

前置知识

- 递归
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

希望通过这篇题解让大家知道“题解区的水有多深”，让大家知道“什么才是好的题解”。

我看了很多人的题解直接就是两句话，然后跟上代码:

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [1] * (n + 1)
        for i in range(3, n + 1):
            for j in range(1, i):
                dp[i] = max(j * dp[i - j], j * (i - j), dp
        return dp[n]
```

这种题解说实话，只针对那些”自己会， 然后去题解区看看有没有新的更好的解法的人“。但是大多数看题解的人是那种 `自己没思路，不会做的人` 。那么这种题解就没什么用了。

我认为 `好的题解应该是新手友好的，并且能够将解题人思路完整展现的题解` 。比如看到这个题目，我首先想到了什么（对错没有关系），然后头脑中经过怎么样的筛选将算法筛选到具体某一个或某几个。我的最终算法是如何想到的，有没有一些先行知识。

当然我也承认自己有很多题解也是直接给的答案，这对很多人来说用处不大，甚至有可能有反作用，给他们一种”我已经会了“的假象。实际上他们根本不懂解题人本身原本的想法， 也许是写题解的人觉得”这很自然“，也可能”只是为了秀技“。

Ok，下面来讲下 `我是如何解这道题的` 。

抽象

首先看到这道题，自然而然地先对问题进行抽象，这种抽象能力是必须的。LeetCode 实际上有很多这种穿着华丽外表的题，当你把这个衣服扒开的时候，会发现都是差不多的，甚至两个是一样的，这样的例子实际上有很多。 就本题来说，就有一个剑指 Offer 的原题《剪绳子》和其本质一样，只是换了描述方式。类似的有力扣 137 和 645 等等，大家可以自己去归纳总结。

> 137 和 645 我贴个之前写的题解 https://leetcode-cn.com/problems/single-number/solution/zhi-chu-xian-yi-ci-de-shu-xi-lie-wei-yun-suan-by-3/

培养自己抽象问题的能力，不管是在算法上还是工程上。 务必记住这句话！

数学是一门非常抽象的学科，同时也很方便我们抽象问题。为了显得我的题解比较高级，引入一些你们看不懂的数学符号也是很有必要的（开玩笑，没有什么高级数学符号啦）。

> 实际上这道题可以用纯数学角度来解，但是我相信大多数人并不想看。即使你看了，大多人的感受也是“好 nb，然而并没有什么用”。

这道题抽象一下就是：

$$n == \sum_{i=1}^{n} n_i$$

令： （图 1） 求：

$$max(\prod_{i=1}^{n} n_i)$$

（图 2）

第一直觉

经过上面的抽象，我的第一直觉这可能是一个数学题，我回想了下数学知识，然后用数学法 AC 了。 数学就是这么简单平凡且枯燥。

然而如果没有数学的加持的情况下，我继续思考怎么做。我想是否可以枚举所有的情况（如图 1），然后对其求最大值（如图 2）。

问题转化为如何枚举所有的情况。经过了几秒钟的思考，我发现这是一个很明显的递归问题。 具体思考过程如下：

- 我们将原问题抽象为 f(n)
- 那么 f(n) 等价于 max(1 * fn(n - 1), 2 * f(n - 2), ..., (n - 1) * f(1))。

用数学公式表示就是：

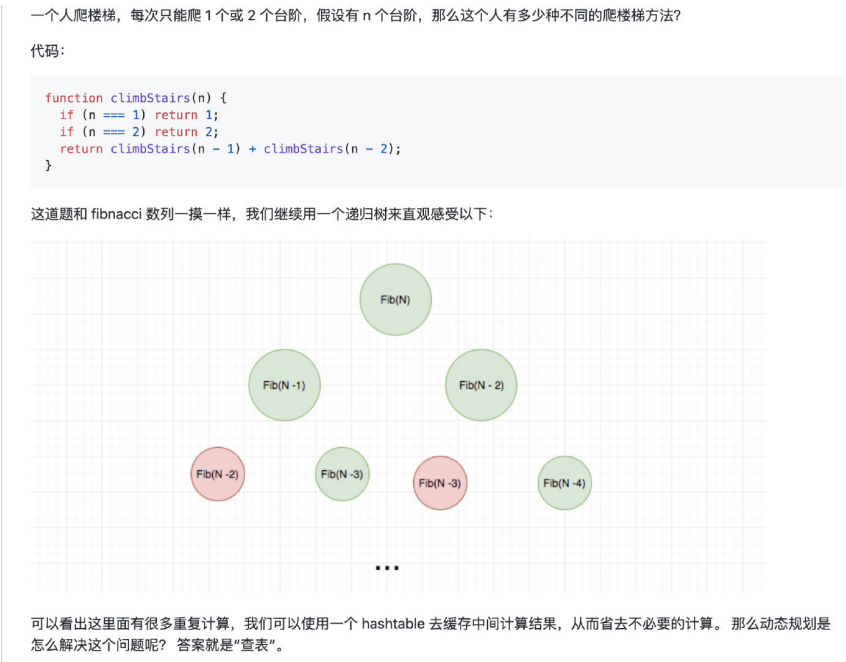
$$max(\prod_{i=1}^{n} i * f(n-i)))$$

（图 3）

截止目前，是一点点数学 + 一点点递归，我们继续往下看。现在问题是不是就很简单啦？直接翻译图三为代码即可，我们来看下这个时候的代码：

```python
class Solution:
    def integerBreak(self, n: int) -> int:
        if n == 2: return 1
        res = 0
        for i in range(1, n):
            res = max(res, max(i * self.integerBreak(n - i)
        return res
```

毫无疑问，超时了。原因很简单，就是算法中包含了太多的重复计算。如果经常看我的题解的话，这句话应该不陌生。我随便截一个我之前讲过这个知识点的图。



(图 4)

> 原文链接：https://github.com/azl397985856/leetcode/blob/master/thinkings/dynamic-programming.md

大家可以尝试自己画图理解一下。

> 看到这里，有没有种殊途同归的感觉呢?

考虑优化

如上，我们可以考虑使用记忆化递归的方式来解决。只是用一个 hashtable 存储计算过的值即可。

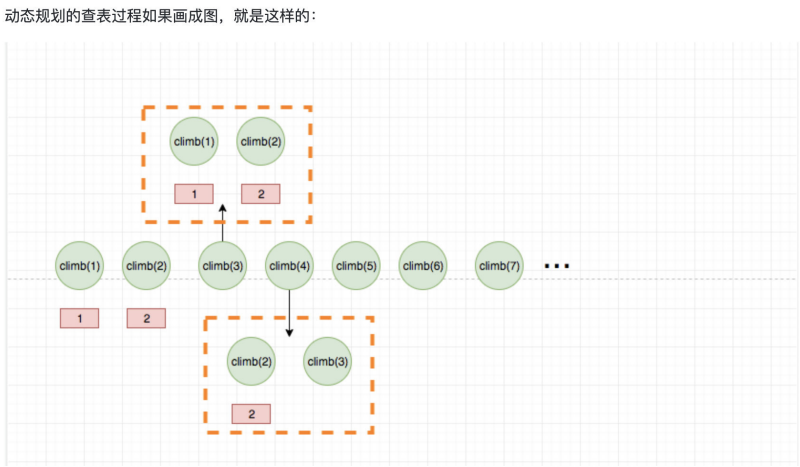
```
class Solution:
    @lru_cache()
    def integerBreak(self, n: int) -> int:
        if n == 2: return 1
        res = 0
        for i in range(1, n):
            res = max(res, max(i * self.integerBreak(n - i)
        return res
```

为了简单起见（偷懒起见），我直接用了 lru_cache 注解， 上面的代码是可以 AC 的。

动态规划

看到这里的同学应该发现了，这个套路是不是很熟悉？下一步就是将其改造成动态规划了。

如图 4，我们的思考方式是从顶向下，这符合人们思考问题的方式。将其改造成如下图的自底向上方式就是动态规划。



(图 5)

现在再来看下文章开头的代码：

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [1] * (n + 1)
        for i in range(3, n + 1):
            for j in range(1, i):
                dp[i] = max(j * dp[i - j], j * (i - j), dp
        return dp[n]
```

dp table 存储的是图 3 中 f(n)的值。一个自然的想法是令 dp[i] 等价于 f(i)。而由于上面分析了原问题等价于 f(n)，那么很自然的原问题也等价于 dp[n]。

而 dp[i]等价于 f(i)，那么上面针对 f(i) 写的递归公式对 dp[i] 也是适用的，我们拿来试试。

```
// 关键语句
res = max(res, max(i * self.integerBreak(n - i),i * (n - i)
```

翻译过来就是：

```
dp[i] = max(dp[i], max(i * dp(n - i),i * (n - i)))
```

而这里的 n 是什么呢？我们说了 `dp是自底向下的思考方式` ，那么在达到 n 之前是看不到整体的 `n` 的。因此这里的 n 实际上是 1,2,3,4... n。

自然地，我们用一层循环来生成上面一系列的 n 值。接着我们还要生成一系列的 i 值，注意到 n - i 是要大于 0 的，因此 i 只需要循环到 n - 1 即可。

思考到这里，我相信上面的代码真的是 `不难得出` 了。

关键点

- 数学抽象
- 递归分析
- 记忆化递归
- 动态规划

代码

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [1] * (n + 1)
        for i in range(3, n + 1):
            for j in range(1, i):
                dp[i] = max(j * dp[i - j], j * (i - j), dp
        return dp[n]
```

总结

培养自己的解题思维很重要， 不要直接看别人的答案。而是要将别人的东西变成自己的， 而要做到这一点，你就要知道“他们是怎么想到的”，“想到这点是不是有什么前置知识”，“类似题目有哪些”。

最优解通常不是一下子就想到了，这需要你在不那么优的解上摔了很多次跟头之后才能记住的。因此在你没有掌握之前，不要直接去看最优解。在你掌握了之后，我不仅鼓励你去写最优解，还鼓励去一题多解，从多个解决思考问题。 到了那个时候， 萌新也会惊讶地呼喊“哇塞， 这题还可以这么解啊？”。 你也会低调地发出“害，解题就是这么简单平凡且枯燥。”的声音。

扩展

正如我开头所说，这种套路实在是太常见了。希望大家能够识别这种问题的本质，彻底掌握这种套路。另外我对这个套路也在我的新书《LeetCode 题解》中做了介绍，本书目前刚完成草稿的编写，如果你想要第一时间获取到我们的题解新书，那么请发送邮件到 `azl397985856@gmail.com` ，标题著明“书籍《LeetCode 题解》预定”字样。。

题目地址(365. 水壶问题)

https://leetcode-cn.com/problems/water-and-jug-problem/

题目描述

```
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

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
```

BFS（超时）

前置知识

- BFS
- 最大公约数

公司

- 阿里
- 百度
- 字节

思路

两个水壶的水我们考虑成状态，然后我们不断进行倒的操作，改变状态。那么初始状态就是（0 0） 目标状态就是 （any, z）或者 (z, any)，其中 any 指的是任意升水。

已题目的例子，其过程示意图，其中括号表示其是由哪个状态转移过来的：

0 0 3 5(0 0) 3 0 (0 0)0 5(0 0) 3 2(0 5) 0 3(0 0) 0 2(3 2) 2 0(0 2) 2 5(2 0) 3 4(2 5) bingo

代码

```
class Solution:
    def canMeasureWater(self, x: int, y: int, z: int) -> bo
        if x + y < z:
            return False
        queue = [(0, 0)]
        seen = set((0, 0))

        while(len(queue) > 0):
            a, b = queue.pop(0)
            if a ==z or b == z or a + b == z:
                return True
            states = set()

            states.add((x, b))
            states.add((a, y))
            states.add((0, b))
            states.add((a, 0))
            states.add((min(x, b + a), 0 if b < x - a else
            states.add((0 if a + b < y else a - (y - b), mi
            for state in states:
                if state in seen:
                    continue;
                queue.append(state)
                seen.add(state)
        return False
```

复杂度分析

- 时间复杂度：由于状态最多有$O((x + 1) *(y + 1))$ 种，因此总的时间复杂度为$O(x* y)$。
- 空间复杂度：我们使用了队列来存储状态，set 存储已经访问的元素，空间复杂度和状态数目一致，因此空间复杂度是$O(x * y)$。

上面的思路很直观，但是很遗憾这个算法在 LeetCode 的表现是 TLE(Time Limit Exceeded)。不过如果你能在真实面试中写出这样的算法，我相信大多数情况是可以过关的。

我们来看一下有没有别的解法。实际上，上面的算法就是一个标准的 BFS。如果从更深层次去看这道题，会发现这道题其实是一道纯数学问题，类似的纯数学问题在 LeetCode 中也会有一些，不过大多数这种题目，我们仍然可以采取其他方式 AC。那么让我们来看一下如何用数学的方式来解这个题。

数学法 - 最大公约数

思路

这是一道关于 `数论` 的题目，确切地说是关于 `裴蜀定理` （英语：Bézout's identity）的题目。

摘自 wiki 的定义：

```
对任意两个整数 a、b，设 d是它们的最大公约数。那么关于未知数  x和  y的

ax+by=m

有整数解 （x,y）当且仅当  m是  d的整数倍。裴蜀等式有解时必然有无穷多
```

因此这道题可以完全转化为 `裴蜀定理` 。还是以题目给的例子 `x = 3, y = 5, z = 4` ，我们其实可以表示成 `3 * 3 - 1 * 5 = 4` , 即 `3 * x - 1 * y = z` 。我们用 a 和 b 分别表示 3 升的水壶和 5 升的水壶。那么我们可以：

- 倒满 a（**1**）
- 将 a 倒到 b
- 再次倒满 a（**2**）
- 再次将 a 倒到 b（a 这个时候还剩下 1 升）
- 倒空 b（**-1**）
- 将剩下的 1 升倒到 b
- 将 a 倒满（**3**）
- 将 a 倒到 b
- b 此时正好是 4 升

上面的过程就是 `3 * x - 1 * y = z` 的具体过程解释。

也就是说我们只需要求出 x 和 y 的最大公约数 d，并判断 z 是否是 d 的整数倍即可。

代码

代码支持：Python3，JavaScript

Python Code：

```python
class Solution:
    def canMeasureWater(self, x: int, y: int, z: int) -> bo
        if x + y < z:
            return False

        if (z == 0):
            return True

        if (x == 0):
            return y == z

        if (y == 0):
            return x == z

        def GCD(a, b):
            smaller = min(a, b)
            while smaller:
                if a % smaller == 0 and b % smaller == 0:
                    return smaller
                smaller -= 1

        return z % GCD(x, y) == 0
```

JavaScript：

```
/**
 * @param {number} x
 * @param {number} y
 * @param {number} z
 * @return {boolean}
 */
var canMeasureWater = function (x, y, z) {
  if (x + y < z) return false;

  if (z === 0) return true;

  if (x === 0) return y === z;

  if (y === 0) return x === z;

  function GCD(a, b) {
    let min = Math.min(a, b);
    while (min) {
      if (a % min === 0 && b % min === 0) return min;
      min--;
    }
    return 1;
  }

  return z % GCD(x, y) === 0;
};
```

实际上求最大公约数还有更好的方式，比如辗转相除法：

```
def GCD(a, b):
    if b == 0: return a
    return GCD(b, a % b)
```

复杂度分析

- 时间复杂度：$O(log(max(a, b)))$
- 空间复杂度：空间复杂度取决于递归的深度，因此空间复杂度为 $O(log(max(a, b)))$

关键点分析

- 数论
- 裴蜀定理

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(378. 有序矩阵中第 K 小的元素)

https://leetcode-cn.com/problems/kth-smallest-element-in-a-sorted-matrix/

题目描述

给定一个 n x 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
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 ≤ k ≤ n2 。

前置知识

- 二分查找
- 堆

公司

- 阿里
- 腾讯
- 字节

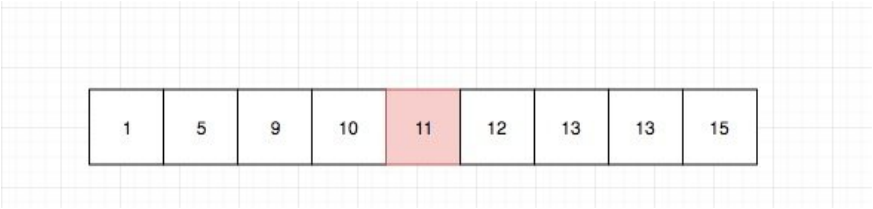
思路

显然用大顶堆可以解决，时间复杂度 Klogn n 为总的数字个数, 但是这种做法没有利用题目中 sorted matrix 的特点，因此不是一种好的做法.

一个巧妙的方法是二分法，我们分别从第一个和最后一个向中间进行扫描，并且计算出中间的数值与数组中的进行比较， 可以通过计算中间值在这个数组中排多少位，然后得到比中间值小的或者大的数字有多少个，然后与 k 进行比较，如果比 k 小则说明中间值太小了，则向后移动，否则向前移动。

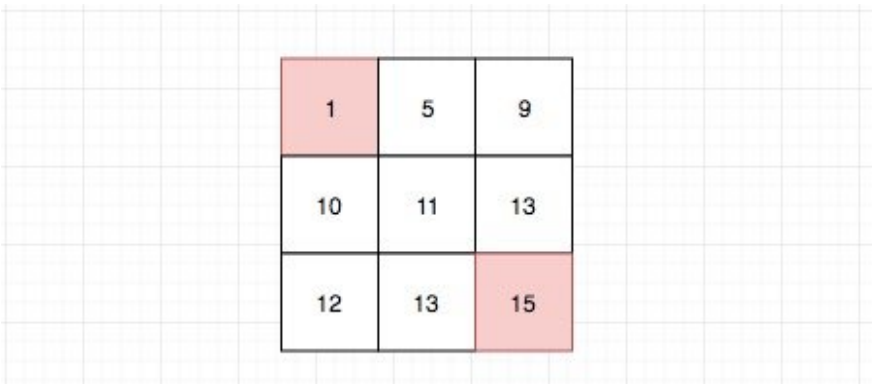
这个题目的二分确实很难想，我们来一步一步解释。

最普通的二分法是有序数组中查找指定值（或者说满足某个条件的值）。由于是有序的，我们可以根据索引关系来确定大小关系， 因此这种思路比较直接，但是对于这道题目索引大小和数字大小没有直接的关系，因此这种二分思想就行不通了。

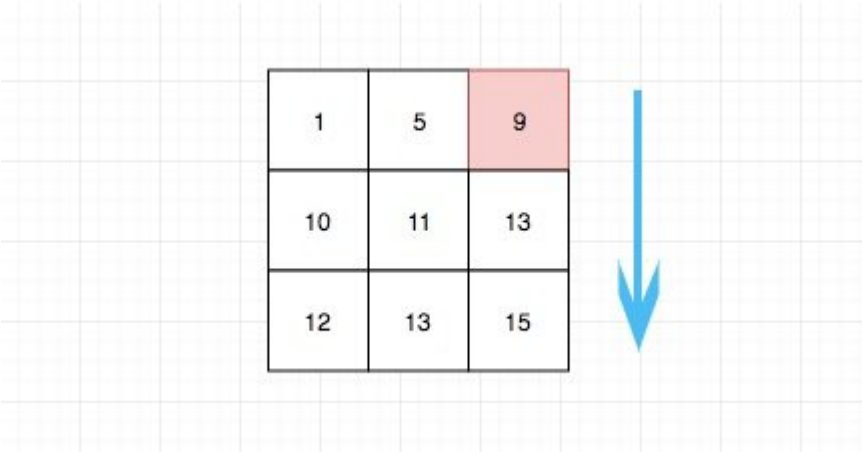


(普通的基于索引判断的二分法)

- 我们能够找到矩阵中最大的元素（右下角）和最小的元素（左上角）。我们可以求出值的中间，而不是上面那种普通二分法的索引的中间。



- 找到中间值之后，我们可以拿这个值去计算有多少元素是小于等于它的。 具体方式就是比较行的最后一列，如果中间值比最后一列大，说明中间元素肯定大于这一行的所有元素。 否则我们从后往前遍历直到不大于。

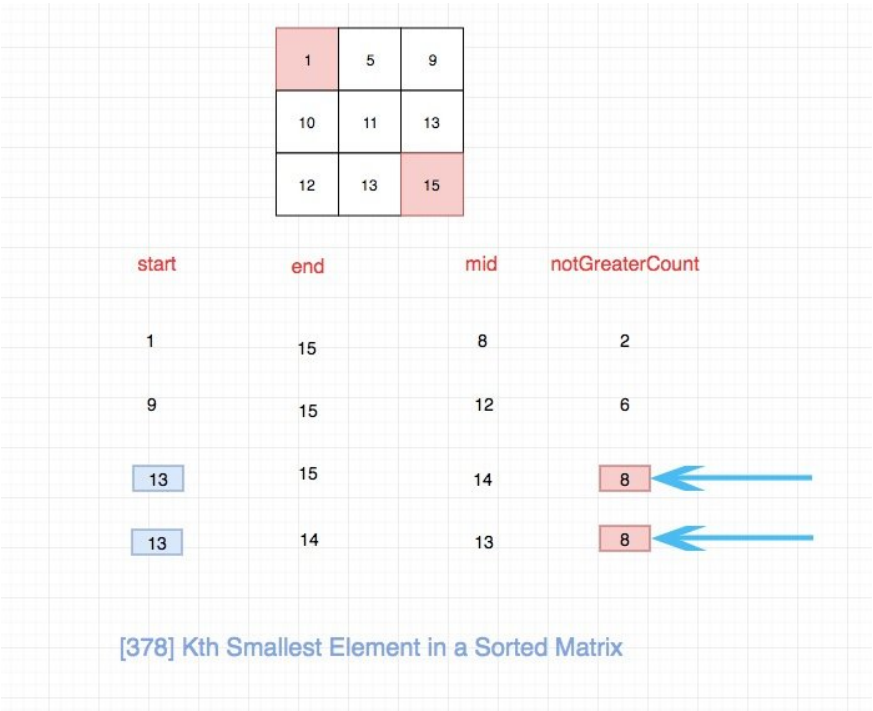


- 上一步我们会计算一个 count，我们拿这个 count 和 k 进行比较
- 如果 count 小于 k，说明我们选择的中间值太小了，肯定不符合条件，我们需要调整左区间为 mid + 1
- 如果 count 大于 k，说明我们选择的中间值正好或者太大了。我们调整右区间 mid

> 由于 count 大于 k 也可能我们选择的值是正好的， 因此这里不能调整为 mid - 1， 否则可能会得不到结果

- 最后直接返回 start, end, 或者 mid 都可以，因此三者最终会收敛到矩阵中的一个元素，这个元素也正是我们要找的元素。

整个计算过程是这样的：



这里有一个大家普遍都比较疑惑的点，也是我当初非常疑惑，困扰我很久的点， leetcode 评论区也有很多人来问，就是“能够确保最终我们找到的元素一定在矩阵中么？”

答案是可以，`相等的时候一定在matrix里面。 因为原问题一定有解，找下界使得start不断的逼近于真实的元素 .`

我是看了评论区一个大神的评论才明白的，以下是@GabrielaSong的评论原文：

```
The lo we returned is guaranteed to be an element in the ma
Let us assume element m is the kth smallest number in the r
When we are about to reach convergence, if mid=m-1, its cou
so we would set lo as (m-1)+1=m, in this case the hi will t
and if mid=m+1, its count value would be k+x-1, so we would
To sum up, because the number lo found by binary search fir
 The equal sign guarantees there exists and only exists one
 So lo must be the only element satisfying this element in
```

更多解释,可以参考leetcode discuss

> 如果是普通的二分查找，我们是基于索引去找，因此不会有这个问题。

关键点解析

- 二分查找
- 有序矩阵的套路（文章末尾还有一道有序矩阵的题目）
- 堆（优先级队列）

代码

```
/*
 * @lc app=leetcode id=378 lang=javascript
 *
 * [378] Kth Smallest Element in a Sorted Matrix
 */
function notGreaterCount(matrix, target) {
  // 等价于在matrix 中搜索mid，搜索的过程中利用有序的性质记录比mid小

  // 我们选择左下角，作为开始元素
  let curRow = 0;
  // 多少列
  const COL_COUNT = matrix[0].length;
  // 最后一列的索引
  const LAST_COL = COL_COUNT - 1;
  let res = 0;

  while (curRow < matrix.length) {
    // 比较最后一列的数据和target的大小
    if (matrix[curRow][LAST_COL] < target) {
      res += COL_COUNT;
    } else {
      let i = COL_COUNT - 1;
      while (i < COL_COUNT && matrix[curRow][i] > target) {
        i--;
      }
      // 注意这里要加1
      res += i + 1;
    }
    curRow++;
  }

  return res;
}
/**
 * @param {number[][]} matrix
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function (matrix, k) {
  if (matrix.length < 1) return null;
  let start = matrix[0][0];
  let end = matrix[matrix.length - 1][matrix[0].length - 1]
  while (start < end) {
    const mid = start + ((end - start) >> 1);
    const count = notGreaterCount(matrix, mid);
    if (count < k) start = mid + 1;
    else end = mid;
  }
```

```
    // 返回start,mid, end 都一样
    return start;
};
```

复杂度分析

- 时间复杂度：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)，因此总的时间复杂度为 $O(nlog(r-l))$。
- 空间复杂度：$O(1)$。

相关题目

- 240.search-a-2-d-matrix-ii

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（380. 常数时间插入、删除和获取随机元素）

https://leetcode-cn.com/problems/insert-delete-getrandom-o1/description/

题目描述

```
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。

insert(val)：当元素 val 不存在时，向集合中插入该项。
remove(val)：元素 val 存在时，从集合中移除该项。
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();
```

思路

这是一个设计题。这道题的核心就是考察基本数据结构和算法的操作以及复杂度。

我们来回顾一下基础知识：

- 数组支持随机访问，其按照索引查询的时间复杂度为$O(1)$,按值查询的时间复杂度为$O(N)$， 而插入和删除的时间复杂度为$O(N)$。
- 链表不支持随机访问，其查询的时间复杂度为$O(N)$,但是对于插入和删除的复杂度为$O(1)$（不考虑找到选要处理的节点花费的时间）。
- 对于哈希表，正常情况下其查询复杂度平均为$O(N)$，插入和删除的复杂度为$O(1)$。

由于题目要求 getRandom 返回要随机并且要在$O(1)$复杂度内，那么如果单纯使用链表或者哈希表肯定是不行的。

而又由于对于插入和删除也需要复杂度为$O(1)$，因此单纯使用数组也是不行的，因此考虑多种使用数据结构来实现。

> 实际上 LeetCode 设计题，几乎没有单纯一个数据结构搞定的，基本都需要多种数据结构结合，这个时候需要你对各种数据结构以及其基本算法的复杂度有着清晰的认知。

对于 getRandom 用数组很简单。对于判断是否已经有了存在的元素，我们使用哈希表也很容易做到。因此我们可以将数组随机访问，以及哈希表$O(1)$按检索值的特性结合起来，即同时使用这两种数据结构。

对于删除和插入，我们需要一些技巧。

对于插入：

- 我们直接往 append，并将其插入哈希表即可。
- 对于删除，我们需要做到 O(1)。删除哈希表很明显可以，但是对于数组，平均时间复杂度为 O(1)。

因此如何应付删除的这种性能开销呢？ 我们知道对于数据删除，我们的时间复杂度来源于

1. `查找到要删除的元素`
2. 以及 `重新排列被删除元素后面的元素` 。

对于 1，我们可以通过哈希表来实现。 key 是插入的数字，value 是数组对应的索引。删除的时候我们根据 key 反查出索引就可以快速找到。

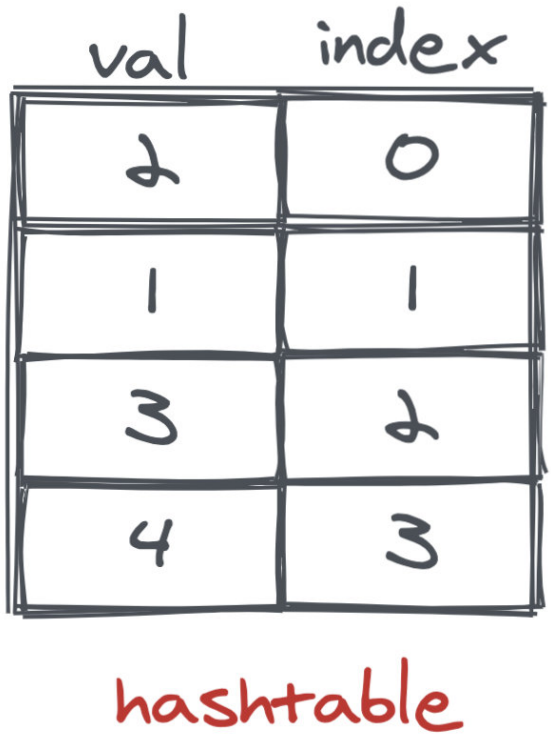
> 题目说明了不会存在重复元素，所以我们可以这么做。思考一下，如果没有这个限制会怎么样?

对于 2，我们可以通过和数组最后一项进行交换的方式来实现，这样就避免了数据移动。同时数组其他项的索引仍然保持不变，非常好！

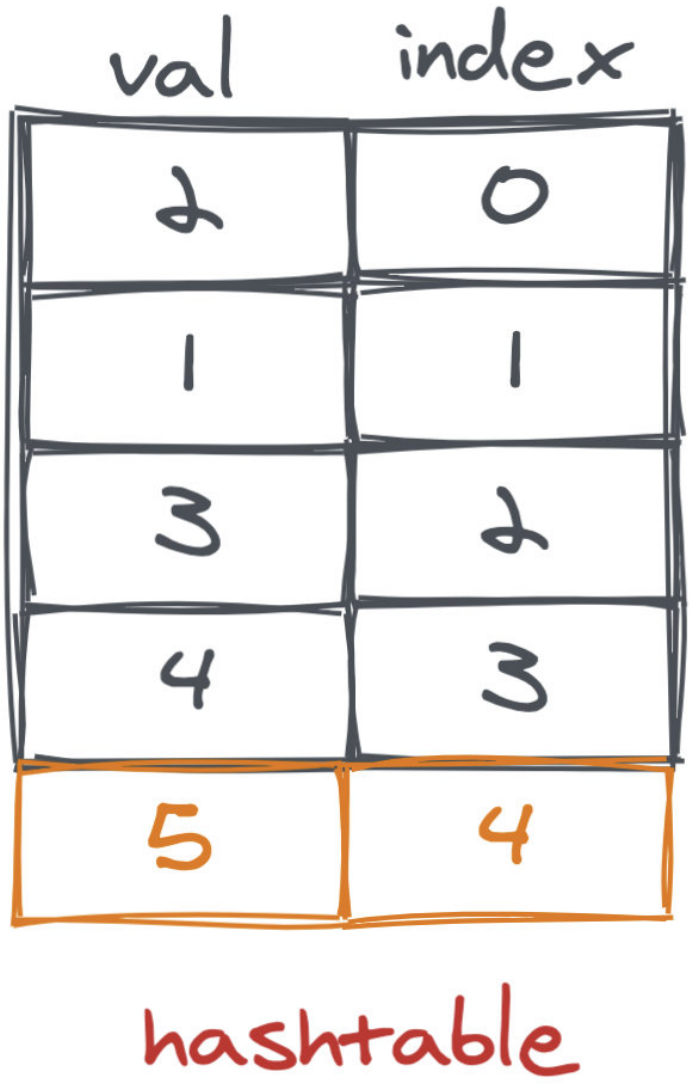
> 相应地，我们插入的时候，需要维护哈希表

图解：

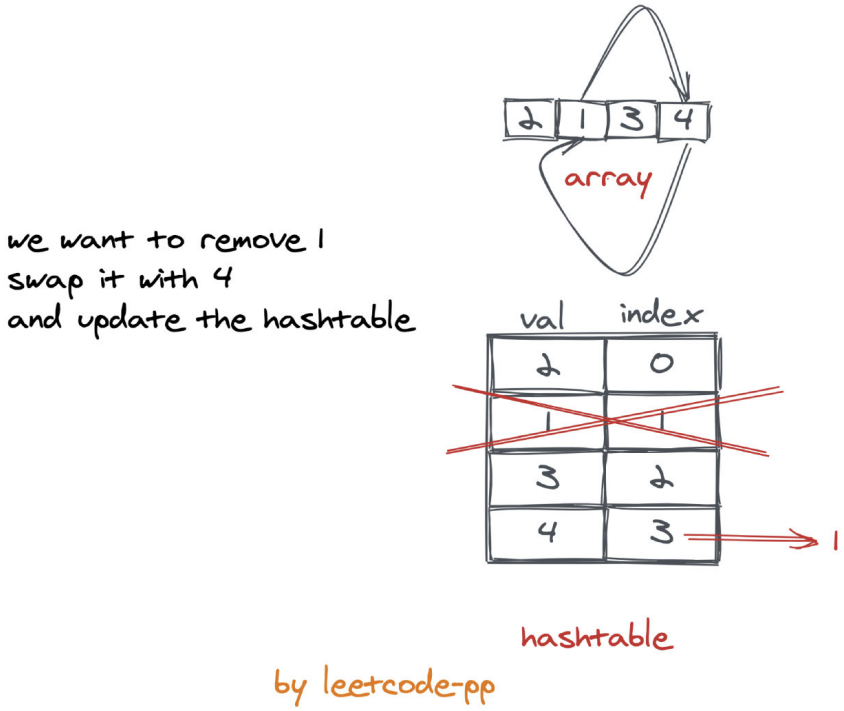
以依次【1，2，3，4】之后为初始状态，那么此时状态是这样的：



而当要插入一个新的 5 的时候， 我们只需要分别向数组末尾和哈希表中插入这条记录即可。



而删除的时候稍微有一点复杂：



关键点解析

- 数组
- 哈希表
- 数组 + 哈希表
- 基本算法时间复杂度分析

代码

```python
from random import random


class RandomizedSet:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.data = dict()
        self.arr = []
        self.n = 0

    def insert(self, val: int) -> bool:
        """
        Inserts a value to the set. Returns true if the set
        """
        if val in self.data:
            return False
        self.data[val] = self.n
        self.arr.append(val)
        self.n += 1

        return True

    def remove(self, val: int) -> bool:
        """
        Removes a value from the set. Returns true if the s
        """
        if val not in self.data:
            return False
        i = self.data[val]
        # 更新data
        self.data[self.arr[-1]] = i
        self.data.pop(val)
        # 更新arr
        self.arr[i] = self.arr[-1]
        # 删除最后一项
        self.arr.pop()
        self.n -= 1

        return True

    def getRandom(self) -> int:
        """
        Get a random element from the set.
        """
```

```
        return self.arr[int(random() * self.n)]


# Your RandomizedSet object will be instantiated and calle
# obj = RandomizedSet()
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

题目地址(394. 字符串解码)

https://leetcode-cn.com/problems/decode-string/

题目描述

```
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_stri

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

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

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"
```

前置知识

- 栈
- 括号匹配

使用栈

思路

题目要求将一个经过编码的字符解码并返回解码后的字符串。题目给定的条件是只有四种可能出现的字符

1. 字母
2. 数字
3. [
4.]

并且输入的方括号总是满足要求的（成对出现），数字只表示重复次数。

那么根据以上条件，可以看出其括号符合栈先进后出的特性以及递归的特质，稍后我们使用递归来解。

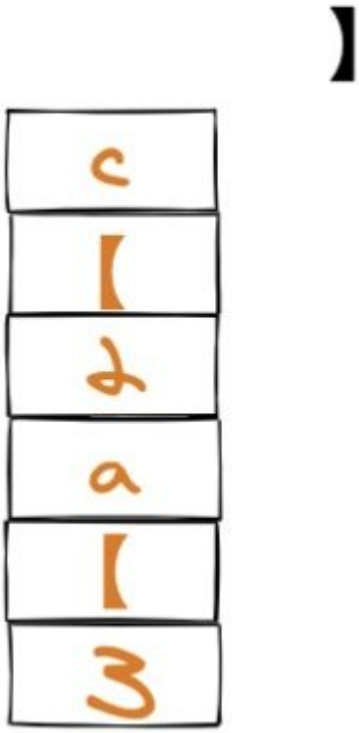
那么现在看一下迭代的解法。

我们可以利用 stack 来实现这个操作，遍历这个字符串 s，判断每一个字符的类型：

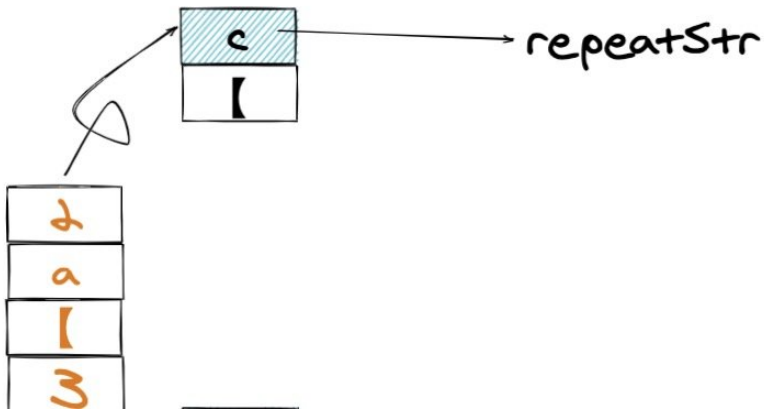
- 如果是字母 --> 添加到 stack 当中
- 如果是数字 --> 先不着急添加到 stack 中 --> 因为有可能有多位
- 如果是 [--> 说明重复字符串开始 --> 将数字入栈 --> 并且将数字清零
- 如果是] --> 说明重复字符串结束 --> 将重复字符串重复前一步储存的数字遍

拿题目给的例子 `s = "3[a2[c]]"` 来说：

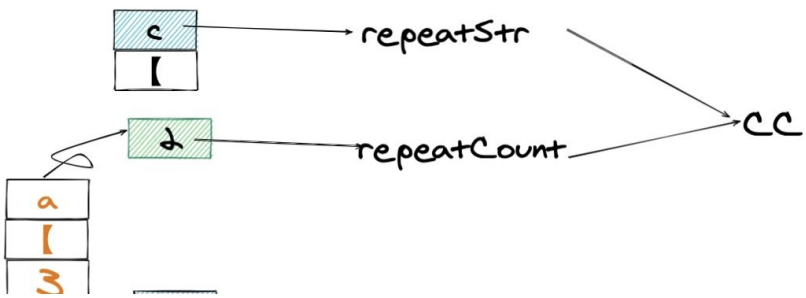
在遇到 `】` 之前，我们不断执行压栈操作：



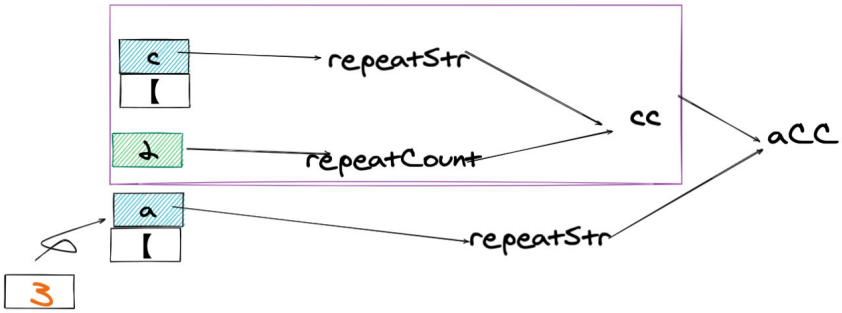
当遇到 】 的时候，说明我们应该出栈了，不断出栈知道对应的 【 ，这中间的就是 repeatStr。



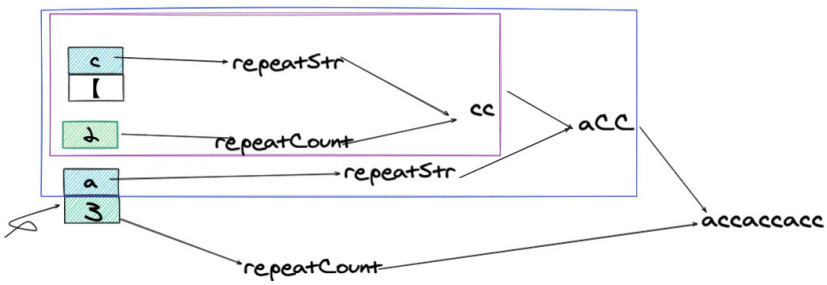
但是要重复几次呢？ 我们需要继续出栈，直到非数字为止，这个数字我们记录为 repeatCount。



而最终的字符串就是 repeatCount 个 repeatStr 拼接的形式。**并将其看成一个字母压入栈中**。



继续，后面的逻辑是一样的：



（最终图）

代码

代码支持：Python

Python：

```python
class Solution:
    def decodeString(self, s: str) -> str:
        stack = []
        for c in s:
            if c == ']':
                repeatStr = ''
                repeatCount = ''
                while stack and stack[-1] != '[':
                    repeatStr = stack.pop() + repeatStr
                # pop 掉 "["
                stack.pop()
                while stack and stack[-1].isnumeric():
                    repeatCount = stack.pop() + repeatCount
                stack.append(repeatStr * int(repeatCount))
            else:
                stack.append(c)
        return "".join(stack)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为解码后的 s 的长度。
- 空间复杂度：$O(N)$，其中 N 为解码后的 s 的长度。

递归

思路

递归的解法也是类似。由于递归的解法并不比迭代书写简单，以及递归我们将在第三节讲述。

主逻辑仍然和迭代一样。 只不过每次碰到左括号就进入递归，碰到右括号就跳出递归返回即可。

唯一需要注意的是，我这里使用了 start 指针跟踪当前遍历到的位置， 因此如果使用递归需要在递归返回后更新指针。

代码

```
class Solution:

    def decodeString(self, s: str) -> str:
        def dfs(start):
            repeat_str = repeat_count = ''
            while start < len(s):
                if s[start].isnumeric():
                    repeat_count += s[start]
                elif s[start] == '[':
                    # 更新指针
                    start, t_str = dfs(start + 1)
                    # repeat_count 仅作用于 t_str，而不作用于当
                    repeat_str = repeat_str + t_str * int(r
                    repeat_count = ''
                elif s[start] == ']':
                    return start, repeat_str
                else:
                    repeat_str += s[start]
                start += 1
            return repeat_str
        return dfs(0)
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为解码后的 s 的长度。
- 空间复杂度：$O(N)$，其中 N 为解码后的 s 的长度。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

欢迎长按关注

题目地址(416. 分割等和子集)

https://leetcode-cn.com/problems/partition-equal-subset-sum/

题目描述

```
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

注意:

每个数组中的元素不会超过 100
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.
```

前置知识

- DFS
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

抽象能力不管是在工程还是算法中都占据着绝对重要的位置。比如上题我们可以抽象为：

给定一个非空数组，和是 sum，能否找到这样的一个子序列，使其和为 2/sum

我们做过二数和，三数和， 四数和，看到这种类似的题会不会舒适一点，思路更开阔一点。

老司机们看到转化后的题，会立马想到背包问题，这里会提供**深度优先搜索**和**背包**两种解法。

深度优先遍历

我们再来看下题目描述，sum 有两种情况，

1. 如果 sum % 2 === 1, 则肯定无解，因为 sum/2 为小数，而数组全由整数构成，子数组和不可能为小数。
2. 如果 sum % 2 === 0, 需要找到和为 2/sum 的子序列 针对 2，我们要在 nums 里找到满足条件的子序列 subNums。 这个过程可以类比为在一个大篮子里面有 N 个球，每个球代表不同的数字，我们用一小篮子去抓取球，使得拿到的球数字和为 2/sum。那么很自然的一个想法就是，对大篮子里面的每一个球，我们考虑取它或者不取它，如果我们足够耐心，最后肯定能穷举所有的情况，判断是否有解。上述思维表述为伪代码如下：

```
令 target = sum / 2, nums 为输入数组, cur 为当前当前要选择的数字
nums 为输入数组，target为当前求和目标，cur为当前判断的数
function dfs(nums, target, cur)
  如果target < 0 或者 cur > nums.length
    return false
  否则
    如果 target = 0, 说明找到答案了，返回true
    否则
      取当前数或者不取，进入递归 dfs(nums, target - nums[cur],
```

因为对每个数都考虑取不取，所以这里时间复杂度是 O(2 ^ n), 其中 n 是 nums 数组长度，

javascript 实现

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  return dfs(nums, sum, 0);
};

function dfs(nums, target, cur) {
  if (target < 0 || cur > nums.length) {
    return false;
  }
  return (
    target === 0 ||
    dfs(nums, target - nums[cur], cur + 1) ||
    dfs(nums, target, cur + 1)
  );
}
```

不出所料，这里是超时了，我们看看有没优化空间

1. 如果 nums 中最大值 > 2/sum, 那么肯定无解
2. 在搜索过程中，我们对每个数都是取或者不取，并且数组中所有项都为正数。我们设取的数和为 `pickedSum`，不难得 pickedSum <= 2/sum， 同时要求丢弃的数为 `discardSum`，不难得 pickedSum <= 2 / sum。

我们同时引入这两个约束条件加强剪枝：

优化后的代码如下

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  nums = nums.sort((a, b) => b - a);
  if (sum < nums[0]) {
    return false;
  }
  return dfs(nums, sum, sum, 0);
};

function dfs(nums, pickRemain, discardRemain, cur) {
  if (pickRemain === 0 || discardRemain === 0) {
    return true;
  }

  if (pickRemain < 0 || discardRemain < 0 || cur > nums.le
    return false;
  }

  return (
    dfs(nums, pickRemain - nums[cur], discardRemain, cur +
    dfs(nums, pickRemain, discardRemain - nums[cur], cur +
  );
}
```

leetcode 是 AC 了，但是时间复杂度 O(2 ^ n), 算法时间复杂度很差，我们看看有没更好的。

DP 解法

在用 DFS 是时候，我们是不关心取数的规律的，只要保证接下来要取的数在之前没有被取过即可。那如果我们有规律去安排取数策略的时候会怎么样呢，比如第一次取数安排在第一位，第二位取数安排在第二位，在判断第 i 位是取数的时候，我们是已经知道前 i-1 个数每次是否取的所有子序列组合，记集合 S 为这个子序列的和。再看第 i 位取数的情况, 有两种情况取或者不取

1. 取的情况，如果 target-nums[i]在集合 S 内，则返回 true，说明前 i 个数能找到和为 target 的序列
2. 不取的情况，如果 target 在集合 S 内，则返回 true，否则返回 false

也就是说，前 i 个数能否构成和为 target 的子序列取决为前 i-1 数的情况。

记 F[i, target] 为 nums 数组内前 i 个数能否构成和为 target 的子序列的可能，则状态转移方程为

`F[i, target] = F[i - 1, target] || F[i - 1, target - nums[i]]`

状态转移方程出来了，代码就很好写了，DFS + DP 都可以解，有不清晰的可以参考下 递归和动态规划， 这里只提供 DP 解法

伪代码表示

```
n = nums.length
target 为 nums 各数之和
如果target不能被2整除,
  返回false

令dp为n * target 的二维矩阵, 并初始为false
遍历0:n, dp[i][0] = true 表示前i个数组成和为0的可能

遍历 0 到 n
  遍历 0 到 target
    if 当前值j大于nums[i]
      dp[i + 1][j] = dp[i][j-nums[i]] || dp[i][j]
    else
      dp[i+1][j] = dp[i][j]
```

算法时间复杂度 O(n*m), 空间复杂度 O(n*m), m 为 sum(nums) / 2

javascript 实现

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  } else {
    sum = sum / 2;
  }

  const dp = Array.from(nums).map(() =>
    Array.from({ length: sum + 1 }).fill(false)
  );

  for (let i = 0; i < nums.length; i++) {
    dp[i][0] = true;
  }

  for (let i = 0; i < dp.length - 1; i++) {
    for (let j = 0; j < dp[0].length; j++) {
      dp[i + 1][j] =
        j - nums[i] >= 0 ? dp[i][j] || dp[i][j - nums[i]]
    }
  }

  return dp[nums.length - 1][sum];
};
```

再看看有没有优化空间，看状态转移方程 `F[i, target] = F[i - 1, target] || F[i - 1, target - nums[i]]` 第 n 行的状态只依赖于第 n-1 行的状态，也就是说我们可以把二维空间压缩成一维

伪代码

```
遍历 0 到 n
  遍历 j 从 target 到 0
    if 当前值j大于nums[i]
      dp[j] = dp[j-nums[i]] || dp[j]
    else
      dp[j] = dp[j]
```

时间复杂度 O(n*m), 空间复杂度 O(n) javascript 实现

```
var canPartition = function (nums) {
  let sum = nums.reduce((acc, num) => acc + num, 0);
  if (sum % 2) {
    return false;
  }
  sum = sum / 2;
  const dp = Array.from({ length: sum + 1 }).fill(false);
  dp[0] = true;

  for (let i = 0; i < nums.length; i++) {
    for (let j = sum; j > 0; j--) {
      dp[j] = dp[j] || (j - nums[i] >= 0 && dp[j - nums[i]]
    }
  }

  return dp[sum];
};
```

其实这道题和 leetcode 518 是换皮题，它们都可以归属于背包问题

背包问题

背包问题描述

有 N 件物品和一个容量为 V 的背包。放入第 i 件物品耗费的费用是 Ci，得到的 价值是 Wi。求解将哪些物品装入背包可使价值总和最大。

背包问题的特性是，每种物品，我们都可以选择放或者不放。令 F[i, v]表示前 i 件物品放入到容量为 v 的背包的状态。

针对上述背包，F[i, v]表示能得到最大价值，那么状态转移方程为

```
F[i, v] = max{F[i-1, v], F[i-1, v-Ci] + Wi}
```

针对 416. 分割等和子集这题，F[i, v]的状态含义就表示前 i 个数能组成和为 v 的可能，状态转移方程为

```
F[i, v] = F[i-1, v] || F[i-1, v-Ci]
```

再回过头来看下leetcode 518, 原题如下

> 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。

带入背包思想，F[i,v] 表示用前 i 种硬币能兑换金额数为 v 的组合数，状态转移方程为 `F[i, v] = F[i-1, v] + F[i-1, v-Ci]`

javascript 实现

```
/**
 * @param {number} amount
 * @param {number[]} coins
 * @return {number}
 */
var change = function (amount, coins) {
  const dp = Array.from({ length: amount + 1 }).fill(0);
  dp[0] = 1;
  for (let i = 0; i < coins.length; i++) {
    for (let j = 1; j <= amount; j++) {
      dp[j] = dp[j] + (j - coins[i] >= 0 ? dp[j - coins[i]]
    }
  }
  return dp[amount];
};
```

复杂度分析

- 时间复杂度：$O(amount * len(coins))$
- 空间复杂度：$O(amount)$

参考

- 背包九讲 基本上看完前四讲就差不多够刷题了。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(424. 替换后的最长重复字符)

https://leetcode-cn.com/problems/longest-repeating-character-replacement/

题目描述

```
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

注意：字符串长度 和 k 不会超过 104。



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。
```

前置知识

-

公司

- 暂无

最长连续 1 模型

思路

这道题其实就是我之前写的滑动窗口的一道题【1004. 最大连续 1 的个数 III】滑动窗口（Python3）的换皮题。字节跳动2018 年的校招（第四批）也考察了这道题的换皮题。

这道题和那两道题差不多。唯一的不同是这道题是 **26 种可能**（因为每一个大写字母都可能是最终的最长重复子串包含的字母），而上面两题是一种可能。这也不难，枚举 26 种情况，取最大值就行了。

最长连续 1 模型可以直接参考上面的链接。其核心算法简单来说，就是**维护一个可变窗口，窗口内的不重复字符小于等于 k，最终返回最大窗口的大小即可。**

关键点

- 最长连续 1 模型

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        def fix(c, k):
            ans = i = 0
            for j in range(len(s)):
                k -= s[j] != c
                while i < j and k < 0:
                    k += s[i] != c
                    i += 1
                ans = max(ans, j - i + 1)
            return ans

        ans = 0
        for i in range(26):
            ans = max(ans, fix(chr(ord("A") + i), k))
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(26 * n)$
- 空间复杂度：$O(1)$

空间换时间

思路

我们也可以用一个长度为 26 的数组 counts 记录每个字母出现的频率，如果窗口大小大于 `最大频率+k` ，我们需要收缩窗口。

这提示我们继续使用滑动窗口技巧。和上面一样，也是**维护一个可变窗口，窗口内的不重复字符小于等于 k，最终返回最大窗口的大小即可。**

关键点

- 空间换时间

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        if not s: return 0
        counts = [0] * 26
        i = most_fraq = 0
        for j in range(len(s)):
            counts[ord(s[j]) - ord("A")] += 1
            most_fraq = max(most_fraq, counts[ord(s[j]) - c
            if i < j and j - i + 1 - most_fraq > k:
                counts[ord(s[i]) - ord("A")] -= 1
                i += 1
        return j - i + 1
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(26)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(445. 两数相加 II)

https://leetcode-cn.com/problems/add-two-numbers-ii/

题目描述

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

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

前置知识

- 链表
- 栈

公司

- 腾讯
- 百度
- 字节

思路

由于需要从低位开始加，然后进位。 因此可以采用栈来简化操作。 依次将两个链表的值分别入栈 stack1 和 stack2，然后相加入栈 stack，进位操作用一个变量 carried 记录即可。

最后根据 stack 生成最终的链表即可。

> 也可以先将两个链表逆置，然后相加，最后将结果再次逆置。

关键点解析

- 栈的基本操作
- carried 变量记录进位
- 循环的终止条件设置成 `stack.length > 0` 可以简化操作
- 注意特殊情况， 比如 1 + 99 = 100

代码

- 语言支持：JS，C++, Python3

JavaScript Code:

```
/*
 * @lc app=leetcode id=445 lang=javascript
 *
 * [445] Add Two Numbers II
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
var addTwoNumbers = function (l1, l2) {
  const stack1 = [];
  const stack2 = [];
  const stack = [];

  let cur1 = l1;
  let cur2 = l2;
  let curried = 0;

  while (cur1) {
    stack1.push(cur1.val);
    cur1 = cur1.next;
  }

  while (cur2) {
    stack2.push(cur2.val);
    cur2 = cur2.next;
  }

  let a = null;
  let b = null;

  while (stack1.length > 0 || stack2.length > 0) {
    a = Number(stack1.pop()) || 0;
    b = Number(stack2.pop()) || 0;

    stack.push((a + b + curried) % 10);

    if (a + b + curried >= 10) {
      curried = 1;
    } else {
```

```
        curried = 0;
      }
    }

    if (curried === 1) {
      stack.push(1);
    }

    const dummy = {};

    let current = dummy;

    while (stack.length > 0) {
      current.next = {
        val: stack.pop(),
        next: null,
      };

      current = current.next;
    }

    return dummy.next;
};
```

C++ Code：

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        auto carry = 0;
        auto ret = (ListNode*)nullptr;
        auto s1 = vector<int>();
        toStack(l1, s1);
        auto s2 = vector<int>();
        toStack(l2, s2);
        while (!s1.empty() || !s2.empty() || carry != 0) {
            auto v1 = 0;
            auto v2 = 0;
            if (!s1.empty()) {
                v1 = s1.back();
                s1.pop_back();
            }
            if (!s2.empty()) {
                v2 = s2.back();
                s2.pop_back();
            }
            auto v = v1 + v2 + carry;
            carry = v / 10;
            auto tmp = new ListNode(v % 10);
            tmp->next = ret;
            ret = tmp;
        }
        return ret;
    }
private:
    // 此处若返回而非传入vector，跑完所有测试用例多花8ms
    void toStack(const ListNode* l, vector<int>& ret) {
        while (l != nullptr) {
            ret.push_back(l->val);
            l = l->next;
        }
    }
};

// 逆置，相加，再逆置。跑完所有测试用例比第一种解法少花4ms
class Solution {
```

```
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        auto rl1 = reverseList(l1);
        auto rl2 = reverseList(l2);
        auto ret = add(rl1, rl2);
        return reverseList(ret);
    }
private:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = NULL;
        ListNode* cur = head;
        ListNode* next = NULL;
        while (cur != NULL) {
            next = cur->next;
            cur->next = prev;
            prev = cur;
            cur = next;
        }
        return prev;
    }

    ListNode* add(ListNode* l1, ListNode* l2) {
        ListNode* ret = nullptr;
        ListNode* cur = nullptr;
        int carry = 0;
        while (l1 != nullptr || l2 != nullptr || carry != 0
            carry += (l1 == nullptr ? 0 : l1->val) + (l2 ==
            auto temp = new ListNode(carry % 10);
            carry /= 10;
            if (ret == nullptr) {
                ret = temp;
                cur = ret;
            }
            else {
                cur->next = temp;
                cur = cur->next;
            }
            l1 = l1 == nullptr ? nullptr : l1->next;
            l2 = l2 == nullptr ? nullptr : l2->next;
        }
        return ret;
    }
};
```

Python Code:

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) ->
        def listToStack(l: ListNode) -> list:
            stack, c = [], l
            while c:
                stack.append(c.val)
                c = c.next
            return stack

        # transfer l1 and l2 into stacks
        stack1, stack2 = listToStack(l1), listToStack(l2)

        # add stack1 and stack2
        diff = abs(len(stack1) - len(stack2))
        stack1 = ([0]*diff + stack1 if len(stack1) < len(st
        stack2 = ([0]*diff + stack2 if len(stack2) < len(st
        stack3 = [x + y for x, y in zip(stack1, stack2)]

        # calculate carry for each item in stack3 and add c
        carry = 0
        for i, val in enumerate(stack3[::-1]):
            index = len(stack3) - i - 1
            carry, stack3[index] = divmod(val + carry, 10)
            if carry and index == 0:
                stack3 = [1] + stack3
            elif carry:
                stack3[index - 1] += 1

        # transfer stack3 to a linkedList
        result = ListNode(0)
        c = result
        for item in stack3:
            c.next = ListNode(item)
            c = c.next

        return result.next
```

复杂度分析

其中 M 和 N 分别为两个链表的长度。

- 时间复杂度：$O(M + N)$

- 空间复杂度：$O(M + N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(454. 四数相加 II)

https://leetcode-cn.com/problems/4sum-ii/

题目描述

```
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,

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 ≤ 5

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) + (
2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (
```

前置知识

- hashTable

公司

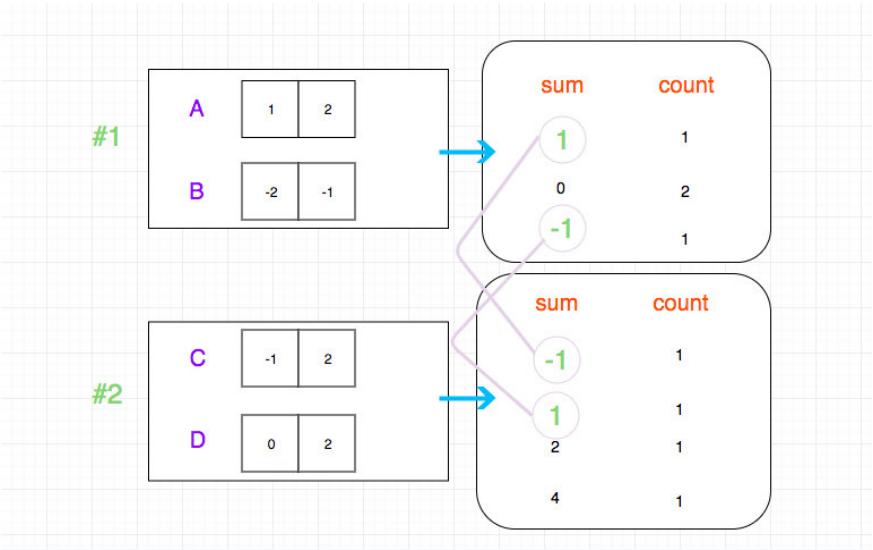
- 阿里
- 字节

思路

如果按照常规思路去完成查找需要四层遍历，时间复杂是 O(n^4), 显然是行不通的。 因此我们有必要想一种更加高效的算法。

我一个思路就是我们将四个数组分成两组，两两结合。 然后我们分别计算 `两两结合能够算出的和有哪些，以及其对应的个数` 。

如图：



这个时候我们得到了两个 `hashTable` ， 我们只需要进行简单的数学运算就可以得到结果。

关键点解析

- 空间换时间
- 两两分组，求出两两结合能够得出的可能数，然后合并即可。

代码

语言支持： `JavaScript` ， `Python3`

`JavaScript` ：

```
/*
 * @lc app=leetcode id=454 lang=javascript
 *
 * [454] 4Sum II
 *
 * https://leetcode.com/problems/4sum-ii/description/
/**
 * @param {number[]} A
 * @param {number[]} B
 * @param {number[]} C
 * @param {number[]} D
 * @return {number}
 */
var fourSumCount = function (A, B, C, D) {
  const sumMapper = {};
  let res = 0;
  for (let i = 0; i < A.length; i++) {
    for (let j = 0; j < B.length; j++) {
      sumMapper[A[i] + B[j]] = (sumMapper[A[i] + B[j]] || 0
    }
  }

  for (let i = 0; i < C.length; i++) {
    for (let j = 0; j < D.length; j++) {
      res += sumMapper[-(C[i] + D[j])] || 0;
    }
  }

  return res;
};
```

Python3 :

```
class Solution:
    def fourSumCount(self, A: List[int], B: List[int], C: L
        mapper = {}
        res = 0
        for i in A:
            for j in B:
                mapper[i + j] = mapper.get(i + j, 0) + 1

        for i in C:
            for j in D:
                res += mapper.get(-1 * (i + j), 0)
        return res
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N^2)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(456. 132 模式)

https://leetcode-cn.com/problems/132-pattern/

题目描述

```
给你一个整数数组 nums ，数组中共有 n 个整数。132 模式的子序列 由三个

如果 nums 中存在 132 模式的子序列 ，返回 true ；否则，返回 false



进阶：很容易想到时间复杂度为 O(n^2) 的解决方案，你可以设计一个时间复杂

示例 1：

输入：nums = [1,2,3,4]
输出：false
解释：序列中不存在 132 模式的子序列。

示例 2：

输入：nums = [3,1,4,2]
输出：true
解释：序列中有 1 个 132 模式的子序列： [1, 4, 2] 。

示例 3：

输入：nums = [-1,3,2,0]
输出：true
解释：序列中有 3 个 132 模式的的子序列：[-1, 3, 2]、[-1, 3, 0] 和



提示：

n == nums.length
1 <= n <= 104
-109 <= nums[i] <= 109
```

前置知识

- 单调栈 🆕

公司

- 暂无

思路

132 模式指的是满足：大小关系是 1 < 2 < 3 ，索引关系是 1 < 3 < 2 的三个数。

一个简单的思路是使用一层**从左到右**的循环固定 3，遍历的同时维护最小值，这个最小值就是 1（如果固定的 3 不等于 1 的话）。 接下来使用另外一个嵌套寻找枚举符合条件的 2 即可。 这里的符合条件指的是大于 1 且小于 3。这种做法的时间复杂度为 $O(n^2)$，并不是一个好的做法，我们需要对其进行优化。

实际上，我们也可以枚举 2 的位置，这样目标变为找到一个大于 2 的数和一个小于 2 的数。由于 2 在序列的右侧，因此我们需要**从右往左**进行遍历。又由于题目只需要找到一个 312 模式，因此我们应该贪心地选择尽可能大的 2（只要不大于 3 即可），这样才**更容易找到 1**（换句话说不会错过 1）。

首先考虑找到 32 模式。我们可以使用从右往左遍历，遇到一个比后一位大的数。我们就找到了一个可行的 32 模式。和上面思路类似，维护一个全局最小值即可，这样就不会错过答案。可是这样就无法做到前面提到的**贪心地选择尽可能大的 2**，我们选择的 2 实际上是尽可能小的 2。那如何找到尽可能大的并且比当前数小的 2 呢?

其实，我们可以维护一个递增栈。每次遇到一个比栈顶大的数就 pop 栈，直到栈顶比当前数字还大。那么最后一次 pop 出去的就是满足条件的最大的 2 了。找到了 32 模式，接下来，我们只需要找到一个比 2 小的数就可以直接返回 True 了。

关键点

- 固定 2, 从右往左遍历,使用单调栈获取最大的小于当前数的 2，并将当前数作为 3 。

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def find132pattern(self, nums: List[int]) -> bool:
        s2, stack = float("-inf"), []

        for i in range(len(nums) - 1, -1, -1):
            if nums[i] < s2:
                return True
            while stack and stack[-1] < nums[i]:
                s2 = stack.pop()
            stack.append(nums[i])
        return False
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（464. 我能赢么）

https://leetcode-cn.com/problems/can-i-win/

题目描述

```
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

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），

给定一个整数 maxChoosableInteger （整数池中可选择的最大数）和另一个

你可以假设 maxChoosableInteger 不会大于 20， desiredTotal 不会大

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），
同样地，第一个玩家选择任意其他整数，第二个玩家都会赢。
```

前置知识

- 动态规划
- 回溯

公司

- 阿里
- linkedin

暴力解（超时）

思路

题目的函数签名如下：

```
def canIWin(self, maxChoosableInteger: int, desiredTotal: i
```

即给你两个整数 maxChoosableInteger 和 desiredTotal，让你返回一个布尔值。

两种特殊情况

首先考虑两种特殊情况，后面所有的解法这两种特殊情况都适用，因此不再赘述。

- 如果 desiredTotal 是小于等于 maxChoosableInteger 的，直接返回 True，这不难理解。
- 如果 [1, maxChoosableInteger] 全部数字之和小于 desiredTotal，谁都无法赢，返回 False。

一般情况

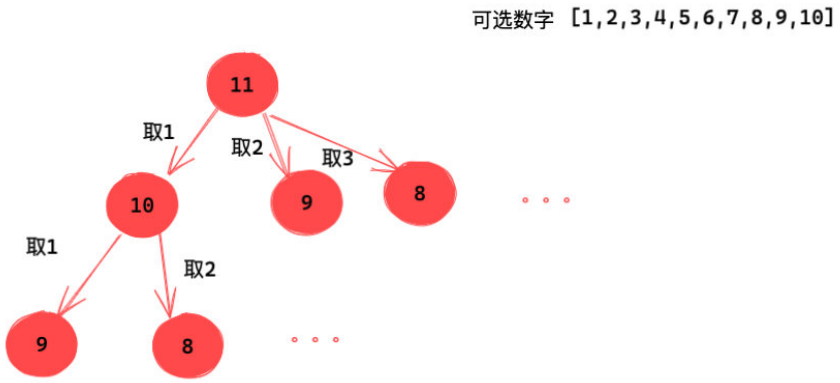
考虑完了特殊情况，我们继续思考一般情况。

首先我们来简化一下问题， 如果数字可以随便选呢？这个问题就简单多了，和爬楼梯没啥区别。这里考虑暴力求解，使用 DFS + 模拟的方式来解决。

注意到每次可选的数字都不变，都是 [1, maxChoosableInteger] ，因此无需通过参数传递。或者你想传递的话，把引用往下传也是可以的。

> 这里的 [1, maxChoosableInteger] 指的是一个左右闭合的区间。

为了方便大家理解，我画了一个逻辑树：



接下来，我们写代码遍历这棵树即可。

可重复选的暴力核心代码如下：

```
class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTota
        # acc 表示当前累计的数字和
        def dfs(acc):
            if acc >= desiredTotal:
                return False
            for n in range(1, maxChoosableInteger + 1):
                # 对方有一种情况赢不了，我就选这个数字就能赢了，返回
                if not dfs(acc + n):
                    return True
            return False

        # 初始化集合，用于保存当前已经选择过的数。
        return dfs(0)
```

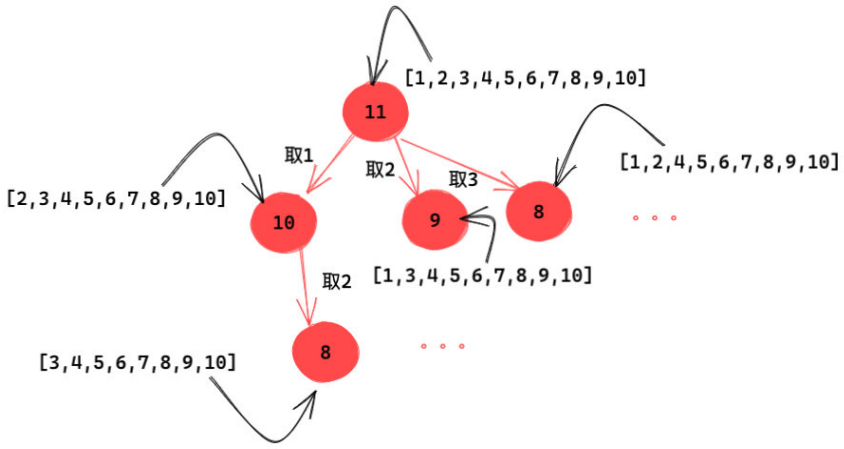
上面代码已经很清晰了，并且加了注释，我就不多解释了。我们继续来看下**如果数字不允许重复选** 会怎么样?

一个直观的思路是使用 set 记录已经被取的数字。当选数字的时候，如果是在 set 中则不取即可。由于可选数字在**动态变化**。也就是说上面的逻辑树部分，每个树节点的可选数字都是不同的。

那怎么办呢？很简单，通过参数传递呗。而且：

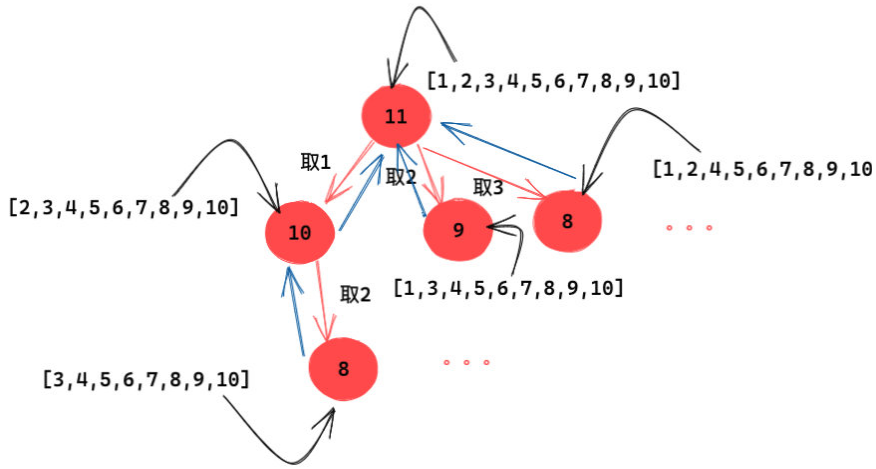
- 要么 set 是值传递，这样不会相互影响。
- 要么每次递归返回的是时候主动回溯状态。 关于这块不熟悉的，可以看下我之前写过的回溯专题。

如果使用值传递，对应是这样的：



每一个 set 引用地址都是不同的

如果在每次递归返回的是时候主动回溯状态，对应是这样的：



每一个 set 引用地址都是一样的

注意图上的蓝色的新增的线，他们表示递归返回的过程。我们需要在返回的过程**撤销选择**。比如我选了数组 2， 递归返回的时候再把数字 2 从 set 中移除。

简单对比下两种方法。

- 使用 set 的值传递，每个递归树的节点都会存一个完整的 set，空间大概是 **节点的数目 X set 中数字个数**，因此空间复杂度大概是 $O(2^maxChoosableInteger * maxChoosableInteger)$， 这个空间根本不可想象，太大了。
- 使用本状态回溯的方式。由于每次都要从 set 中移除指定数字，时间复杂度是 $O(maxChoosableInteger X 节点数)$，这样做时间复杂度又太高了。

这里我用了第二种方式 - 状态回溯。和上面代码没有太大的区别，只是加了一个 set 而已，唯一需要注意的是需要在回溯过程恢复状态（picked.remove(n)）。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTota
        if desiredTotal <= maxChoosableInteger:
            return True
        if sum(range(maxChoosableInteger + 1)) < desiredTot
            return False
        # picked 用于保存当前已经选择过的数。
        # acc 表示当前累计的数字和
        def backtrack(picked, acc):
            if acc >= desiredTotal:
                return False
            if len(picked) == maxChoosableInteger:
                # 说明全部都被选了，没得选了，返回 False， 代表输
                return False
            for n in range(1, maxChoosableInteger + 1):
                if n not in picked:
                    picked.add(n)
                    # 对方有一种情况赢不了，我就选这个数字就能赢了
                    if not backtrack(picked, acc + n):
                        picked.remove(n)
                        return True
                    picked.remove(n)
            return False

        # 初始化集合，用于保存当前已经选择过的数。
        return backtrack(set(), 0)
```

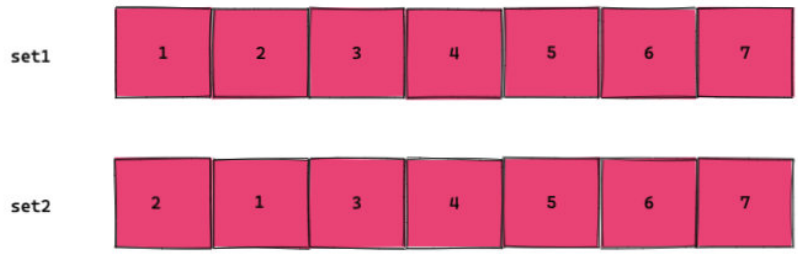
状态压缩 + 回溯

思路

有的同学可能会问， 为什么不使用记忆化递归？这样可以有效减少逻辑树的节点数，从指数级下降到多项式级。这里的原因在于 set 是不可直接序列化的，因此不可直接存储到诸如哈希表这样的数据结构。

而如果你自己写序列化，比如最粗糙的将 set 转换为字符串或者元祖存。看起来可行，set 是 ordered 的，因此如果想正确序列化还需要排序。当然你可用一个 orderedhashset，不过效率依然不好，感兴趣的可以研究一下。

如下图，两个 set 应该一样，但是遍历的结果顺序可能不同，如果不排序就可能有错误。



至此，问题的关键基本上锁定为找到一个**可以序列化且容量大大减少的数据结构**来存是不是就可行了？

注意到 **maxChoosableInteger 不会大于 20** 这是一个强有力的提示。由于 20 是一个不大于 32 的数字， 因此这道题很有可能和状态压缩有关，比如用 4 个字节存储状态。力扣相关的题目还有不少， 具体大家可参考文末的相关题目。

我们可以将状态进行压缩，使用位来模拟。实际上使用状态压缩和上面**思路一模一样，只是 API 不一样**罢了。

假如我们使用的这个用来代替 set 的数字名称为 picked。

- picked 第一位表示数字 1 的使用情况。
- picked 第二位表示数字 2 的使用情况。
- picked 第三位表示数字 3 的使用情况。
- 。。。

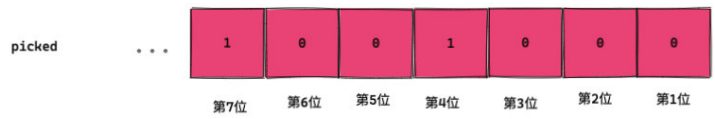
比如我们刚才用了集合，用到的集合 api 有：

- in 操作符，判断一个数字是否在集合中
- add(n) 函数， 用于将一个数加入到集合
- len()，用于判断集合的大小

那我们其实就用位来模拟实现这三个 api 就罢了。详细可参考我的这篇题解 - 面试题 01.01. 判定字符是否唯一

如果实现 add 操作?

这个不难。 比如我要模拟 picked.add(n)，只要将 picked 第 n 为置为 1 就行。也就是说 1 表示在集合中，0 表示不在。

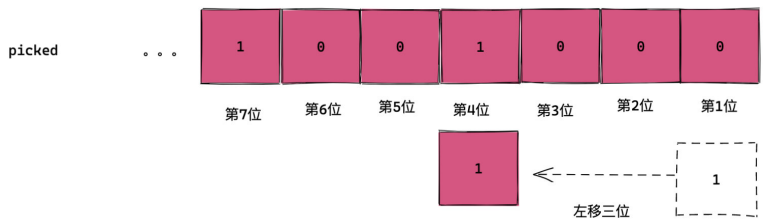


使用**或运算和位移运算**可以很好的完成这个需求。

位移运算

```
1 << a
```

指的是 1 的二进制表示全体左移 a 位， 右移也是同理



| 操作

```
a | b
```

指的是 a 和 b 每一位都进行或运算的结构。 常见的用法是 a 和 b 其中一个当成是 seen。 这样就可以当**二值**数组和哈希表用了。 比如：

```
seen = 0b0000000
a = 0b0000001
b = ob0000010

seen |= a 后,   seen 为 0b0000001
seen |= b 后,   seen 为 0b0000011
```

这样我就可以知道 a 和 b 出现过了。 当然 a， b 以及其他你需要统计的数字只能用一位。 典型的是题目只需要存 26 个字母，那么一个 int(32 bit) 足够了。 如果是包括大写，那就是 52， 就需要至少 52 bit。

如何实现 in 操作符?

有了上面的铺垫就简单了。比如要模拟 n in picked。那只要判断 picked 的第 n 位是 0 还是 1 就行了。如果是 0 表示不在 picked 中，如果是 1 表示在 picked 中。

用**或运算和位移运算**可以很好的完成这个需求。

& 操作

```
a & b
```

指的是 a 和 b 每一位都进行与运算的结构。 常见的用法是 a 和 b 其中一个是 mask。 这样就可以得指定位是 0 还是 1 了。 比如：

```
mask = 0b0000010
a & mask == 1 说明 a 在第二位（从低到高）是 1
a & mask == 0 说明 a 在第二位（从低到高）是 0
```

如何实现 len

其实只要逐个 bit 比对，如果当前 bit 是 1 则计数器 + 1，最后返回计数器的值即可。

这没有问题。而实际上，我们只关心集合大小是否等于 maxChoosableInteger。也就是我只关心**第 maxChoosableInteger 位以及低于 maxChoosableInteger 的位是否全部是 1**。

这就简单了，我们只需要将 1 左移 maxChoosableInteger + 1 位再减去 1 即可。一行代码搞定：

```
picked == (1 << (maxChoosableInteger + 1)) - 1
```

上面代码返回 true 表示满了， 否则没满。

至此大家应该感受到了，使用位来代替 set 思路上没有任何区别。不同的仅仅是 API 而已。如果你只会使用 set 不会使用位运算进行状态压缩，只能说明你对位 的 api 不熟而已。多练习几道就行了，文末我列举了几道类似的题目，大家不要错过哦~

关键点分析

- 回溯
- 动态规划
- 状态压缩

代码

代码支持：Java,CPP,Python3,JS

Java Code:

```
public class Solution {
    public boolean canIWin(int maxChoosableInteger, int des

        if (maxChoosableInteger >= desiredTotal) return tru
        if ((1 + maxChoosableInteger) * maxChoosableIntege

        Boolean[] dp = new Boolean[(1 << maxChoosableIntege
        return dfs(maxChoosableInteger, desiredTotal, 0, dp
    }

    private boolean dfs(int maxChoosableInteger, int desire
        if (dp[state] != null)
            return dp[state];
        for (int i = 1; i <= maxChoosableInteger; i++){
            int tmp = (1 << (i - 1));
            if ((tmp & state) == 0){
                if (desiredTotal - i <= 0 || !dfs(maxChoosa
                    dp[state] = true;
                    return true;
                }
            }
        }
        dp[state] = false;
        return false;
    }
}
```

C++ Code:

```
class Solution {
public:
    bool canIWin(int maxChoosableInteger, int desiredTotal)
        int sum = (1+maxChoosableInteger)*maxChoosableInteg
        if(sum < desiredTotal){
            return false;
        }
        unordered_map<int,int> d;
        return dfs(maxChoosableInteger,0,desiredTotal,0,d);
    }

    bool dfs(int n,int s,int t,int S,unordered_map<int,int>
        if(d[S]) return  d[S];
        int& ans = d[S];

        if(s >= t){
            return ans = true;
        }
        if(S == (((1 << n)-1) << 1)){
            return ans = false;
        }

        for(int m = 1;m <=n;++m){
            if(S & (1 << m)){
                continue;
            }
            int nextS = S|(1 << m);
            if(s+m >= t){
                return ans = true;
            }
            bool r1 = dfs(n,s+m,t,nextS,d);
            if(!r1){
                return ans = true;
            }
        }
        return ans = false;
    }
};
```

Python3 Code:

```python
class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTota
        if desiredTotal <= maxChoosableInteger:
            return True
        if sum(range(maxChoosableInteger + 1)) < desiredTot
            return False

        @lru_cache(None)
        def dp(picked, acc):
            if acc >= desiredTotal:
                return False
            if picked == (1 << (maxChoosableInteger + 1)) -
                return False
            for n in range(1, maxChoosableInteger + 1):
                if picked & 1 << n == 0:
                    if not dp(picked | 1 << n, acc + n):
                        return True
            return False

        return dp(0, 0)
```

JS Code:

```
var canIWin = function (maxChoosableInteger, desiredTotal)
  // 直接获胜
  if (maxChoosableInteger >= desiredTotal) return true;

  // 全部拿完也无法到达
  var sum = (maxChoosableInteger * (maxChoosableInteger + 1
  if (desiredTotal > sum) return false;

  // 记忆化
  var dp = {};

  /**
   * @param {number} total 剩余的数量
   * @param {number} state 使用二进制位表示抽过的状态
   */
  function f(total, state) {
    // 有缓存
    if (dp[state] !== undefined) return dp[state];

    for (var i = 1; i <= maxChoosableInteger; i++) {
      var curr = 1 << i;
      // 已经抽过这个数
      if (curr & state) continue;
      // 直接获胜
      if (i >= total) return (dp[state] = true);
      // 可以让对方输
      if (!f(total - i, state | curr)) return (dp[state] =
    }

    // 没有任何让对方输的方法
    return (dp[state] = false);
  }

  return f(desiredTotal, 0);
};
```

相关题目

- 面试题 01.01. 判定字符是否唯一 纯状态压缩，无 DP
- 698. 划分为 k 个相等的子集
- 1681. 最小不兼容性

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(494. 目标和)

https://leetcode-cn.com/problems/target-sum/

题目描述

```
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

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。
```

前置知识

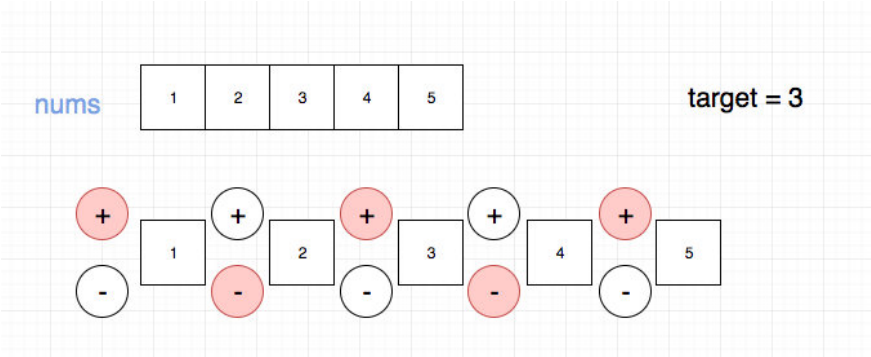
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

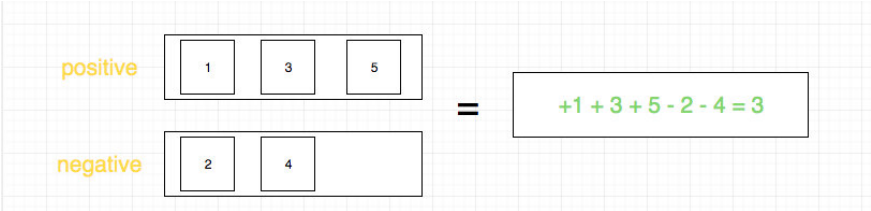
思路

题目是给定一个数组，让你在数字前面添加 `+` 或者 `-`，使其和等于 target.



暴力法的时间复杂度是指数级别的，因此我们不予考虑。我们需要换种思路.

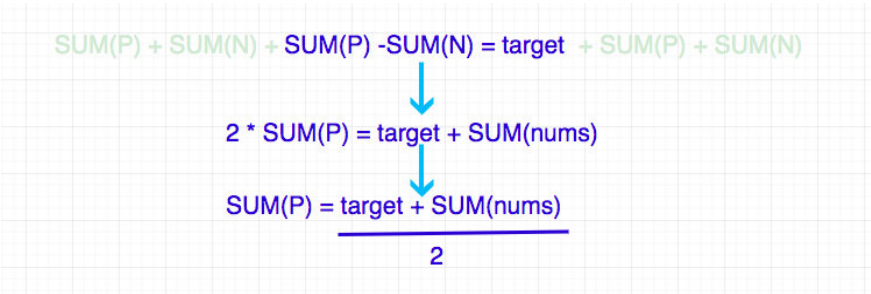
我们将最终的结果分成两组，一组是我们添加了 `+` 的，一组是我们添加了 `-` 的。



如上图，问题转化为如何求其中一组，我们不妨求前面标 `+` 的一组

> 如果求出一组，另一组实际就已知了，即总集和这一组的差集。

通过进一步分析，我们得到了这样的关系：



因此问题转化为，求解 `sumCount(nums, target)`,即 nums 数组中能够组成 target 的总数一共有多少种，这是一道我们之前做过的题目，使用动态规划可以解决。

关键点解析

- 对元素进行分组，分组的依据是符号， 是 `+` 或者 `-`

- 通过数学公式推导可以简化我们的求解过程，这需要一点 数学知识和数学意识

代码

```
/*
 * @lc app=leetcode id=494 lang=javascript
 *
 * [494] Target Sum
 *
 */
// 这个是我们熟悉的问题了
// 我们这里需要求解的是nums里面有多少种可以组成target的方式
var sumCount = function (nums, target) {
  // 这里通过观察，我们没必要使用二维数组去存储这些计算结果
  // 使用一维数组可以有效节省空间
  const dp = Array(target + 1).fill(0);
  dp[0] = 1;
  for (let i = 0; i < nums.length; i++) {
    for (let j = target; j >= nums[i]; j--) {
      dp[j] += dp[j - nums[i]];
    }
  }
  return dp[target];
};
const add = (nums) => nums.reduce((a, b) => (a += b), 0);
/**
 * @param {number[]} nums
 * @param {number} S
 * @return {number}
 */
var findTargetSumWays = function (nums, S) {
  const sum = add(nums);
  if (sum < S) return 0;
  if ((S + sum) % 2 === 1) return 0;
  return sumCount(nums, (S + sum) >> 1);
};
```

复杂度分析

- 时间复杂度：$O(N * target)$
- 空间复杂度：$O(target)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

扩展

如果这道题目并没有限定所有的元素以及 target 都是正数怎么办?

题目地址(516. 最长回文子序列)

https://leetcode-cn.com/problems/longest-palindromic-subsequence/

题目描述

```
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

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
```

前置知识

- 动态规划

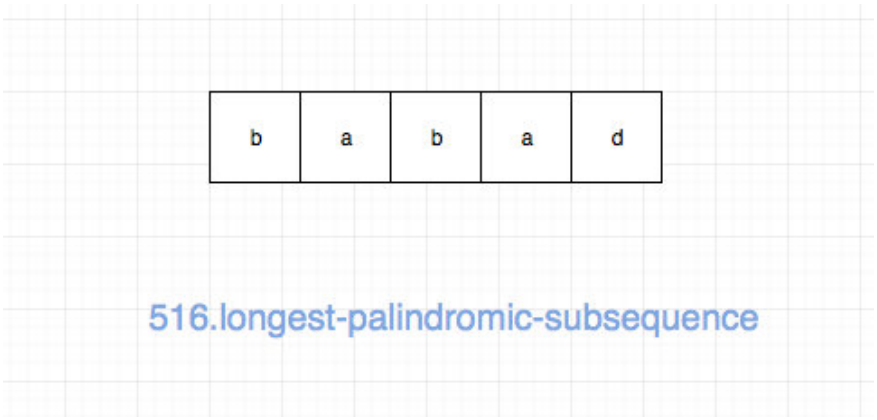
公司

- 阿里

- 腾讯
- 百度
- 字节

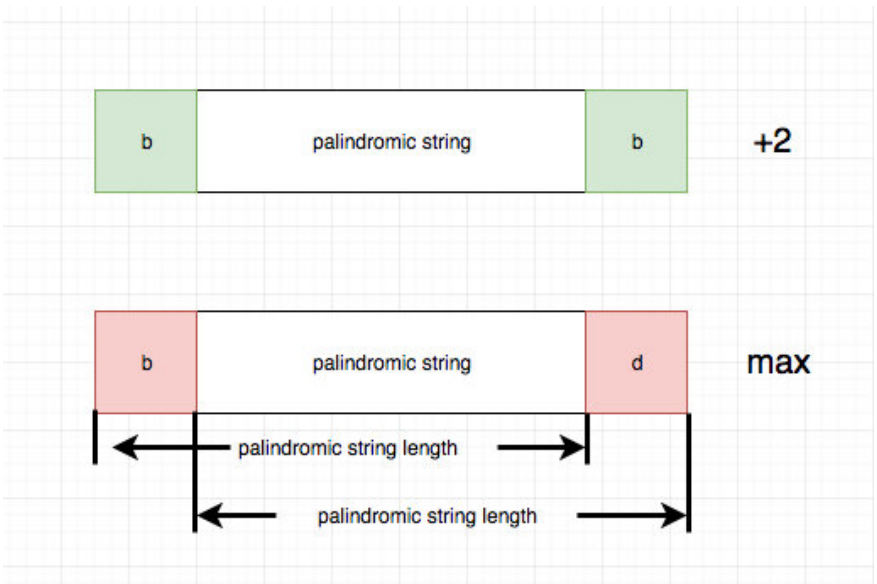
思路

这是一道最长回文的题目，要我们求出给定字符串的最大回文子序列。



解决这类问题的核心思想就是两个字“延伸”，具体来说

- 如果一个字符串是回文串，那么在它左右分别加上一个相同的字符，那么它一定还是一个回文串，因此 `回文长度增加2`
- 如果一个字符串不是回文串，或者在回文串左右分别加不同的字符，得到的一定不是回文串,因此 `回文长度不变，我们取[i][j-1]和[i+1][j]的较大值`

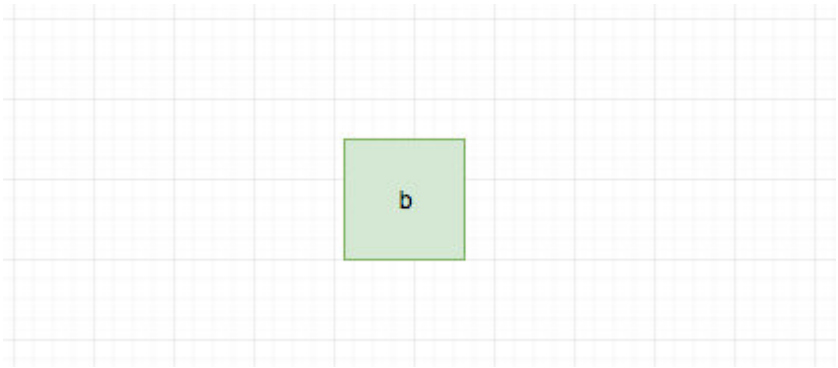


事实上，上面的分析已经建立了大问题和小问题之间的关联， 基于此，我们可以建立动态规划模型。

我们可以用 dp[i][j] 表示 s 中从 i 到 j（包括 i 和 j）的回文序列长度， 状态转移方程只是将上面的描述转化为代码即可：

```
if (s[i] === s[j]) {
  dp[i][j] = dp[i + 1][j - 1] + 2;
} else {
  dp[i][j] = Math.max(dp[i][j - 1], dp[i + 1][j]);
}
```

base case 就是一个字符（轴对称点是本身）



关键点

- ”延伸“（extend）

代码

```
/*
 * @lc app=leetcode id=516 lang=javascript
 *
 * [516] Longest Palindromic Subsequence
 */
/**
 * @param {string} s
 * @return {number}
 */
var longestPalindromeSubseq = function (s) {
  // bbbab 返回4
  // tag : dp
  const dp = [];

  for (let i = s.length - 1; i >= 0; i--) {
    dp[i] = Array(s.length).fill(0);
    for (let j = i; j < s.length; j++) {
      if (i - j === 0) dp[i][j] = 1;
      else if (s[i] === s[j]) {
        dp[i][j] = dp[i + 1][j - 1] + 2;
      } else {
        dp[i][j] = Math.max(dp[i][j - 1], dp[i + 1][j]);
      }
    }
  }

  return dp[0][s.length - 1];
};
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N^2)$

相关题目

- 5.longest-palindromic-substring

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（513. 找树左下角的值）

https://leetcode-cn.com/problems/find-bottom-left-tree-value/

题目描述

给定一个二叉树，在树的最后一行找到最左边的值。

示例 1:

输入:

```
    2
   / \
  1   3
```

输出:
1

示例 2:

输入:

```
        1
       / \
      2   3
     /   / \
    4   5   6
       /
      7
```

输出:
7

BFS

思路

其实问题本身就告诉你怎么做了

> 在树的最后一行找到最左边的值。

问题再分解一下

- 找到树的最后一行
- 找到那一行的第一个节点

不用层序遍历简直对不起这个问题，这里贴一下层序遍历的流程

```
令curLevel为第一层节点也就是root节点
定义nextLevel为下层节点
遍历node in curLevel,
  nextLevel.push(node.left)
  nextLevel.push(node.right)
令curLevel = nextLevel, 重复以上流程直到curLevel为空
```

代码

- 代码支持：JS，Python，Java，CPP, Go, PHP

JS Code：

```
var findBottomLeftValue = function (root) {
  let curLevel = [root];
  let res = root.val;
  while (curLevel.length) {
    let nextLevel = [];
    for (let i = 0; i < curLevel.length; i++) {
      curLevel[i].left && nextLevel.push(curLevel[i].left);
      curLevel[i].right && nextLevel.push(curLevel[i].right
    }
    res = curLevel[0].val;
    curLevel = nextLevel;
  }
  return res;
};
```

Python Code:

```python
class Solution(object):
    def findBottomLeftValue(self, root):
        queue = collections.deque()
        queue.append(root)
        while queue:
            length = len(queue)
            res = queue[0].val
            for _ in range(length):
                cur = queue.popleft()
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
        return res
```

Java：

```java
class Solution {
    Map<Integer,Integer> map = new HashMap<>();
    int maxLevel = 0;
    public int findBottomLeftValue(TreeNode root) {
        if (root == null) return 0;
        LinkedList<TreeNode> deque = new LinkedList<>();
        deque.add(root);
        int res = 0;
        while(!deque.isEmpty()) {
            int size = deque.size();
            for (int i = 0; i < size; i++) {
                TreeNode node = deque.pollFirst();
                if (i == 0) {
                    res = node.val;
                }
                if (node.left != null)deque.addLast(node.le
                if (node.right != null)deque.addLast(node.r
            }
        }
        return res;
    }
}
```

CPP：

```
class Solution {
public:
    int findBottomLeftValue_bfs(TreeNode* root) {
        queue<TreeNode*> q;
        TreeNode* ans = NULL;
        q.push(root);
        while (!q.empty()) {
            ans = q.front();
            int size = q.size();
            while (size--) {
                TreeNode* cur = q.front();
                q.pop();
                if (cur->left )
                    q.push(cur->left);
                if (cur->right)
                    q.push(cur->right);
            }
        }
        return ans->val;
    }
}
```

Go Code:

```go
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func findBottomLeftValue(root *TreeNode) int {
    res := root.Val
    curLevel := []*TreeNode{root} // 一层层遍历
    for len(curLevel) > 0 {
        res = curLevel[0].Val
        var nextLevel []*TreeNode
        for _, node := range curLevel {
            if node.Left != nil {
                nextLevel = append(nextLevel, node.Left)
            }
            if node.Right != nil {
                nextLevel = append(nextLevel, node.Right)
            }
        }
        curLevel = nextLevel
    }
    return res
}
```

PHP Code:

```php
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     public $val = null;
 *     public $left = null;
 *     public $right = null;
 *     function __construct($value) { $this->val = $value;
 * }
 */
class Solution
{

    /**
     * @param TreeNode $root
     * @return Integer
     */
    function findBottomLeftValue($root)
    {
        $curLevel = [$root];
        $res = $root->val;
        while (count($curLevel)) {
            $nextLevel = [];
            $res = $curLevel[0]->val;
            foreach ($curLevel as $node) {
                if ($node->left) $nextLevel[] = $node->left
                if ($node->right) $nextLevel[] = $node->rig
            }
            $curLevel = $nextLevel;
        }
        return $res;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点数。
- 空间复杂度：$O(Q)$，其中 Q 为队列长度，最坏的情况是满二叉树，此时和 N 同阶，其中 N 为树的节点总数

DFS

思路

树的最后一行找到最左边的值，转化一下就是找第一个出现的深度最大的节点，这里用先序遍历去做，其实中序遍历也可以，只需要保证左节点在右节点前被处理即可。 具体算法为，先序遍历 root，维护一个最大深度

的变量，记录每个节点的深度，如果当前节点深度比最大深度要大，则更新最大深度和结果项。

代码

代码支持：JS，Python，Java，CPP

JS Code：

```
function findBottomLeftValue(root) {
  let maxDepth = 0;
  let res = root.val;

  dfs(root.left, 0);
  dfs(root.right, 0);

  return res;

  function dfs(cur, depth) {
    if (!cur) {
      return;
    }
    const curDepth = depth + 1;
    if (curDepth > maxDepth) {
      maxDepth = curDepth;
      res = cur.val;
    }
    dfs(cur.left, curDepth);
    dfs(cur.right, curDepth);
  }
}
```

Python Code:

```python
class Solution(object):

    def __init__(self):
        self.res = 0
        self.max_level = 0

    def findBottomLeftValue(self, root):
        self.res = root.val
        def dfs(root, level):
            if not root:
                return
            if level > self.max_level:
                self.res = root.val
                self.max_level = level
            dfs(root.left, level + 1)
            dfs(root.right, level + 1)
        dfs(root, 0)

        return self.res
```

Java Code:

```java
class Solution {
    int max = 0;
    Map<Integer,Integer> map = new HashMap<>();
    public int findBottomLeftValue(TreeNode root) {
        if (root == null) return 0;
        dfs(root,0);
        return map.get(max);
    }

    void dfs (TreeNode node,int level){
        if (node == null){
            return;
        }
        int curLevel = level+1;
        dfs(node.left,curLevel);
        if (curLevel > max && !map.containsKey(curLevel)){
            map.put(curLevel,node.val);
            max = curLevel;
        }
        dfs(node.right,curLevel);
    }

}
```

CPP:

```cpp
class Solution {
public:
    int res;
    int max_depth = 0;
    void findBottomLeftValue_core(TreeNode* root, int depth
        if (root->left || root->right) {
            if (root->left)
                findBottomLeftValue_core(root->left, depth
            if (root->right)
                findBottomLeftValue_core(root->right, depth
        } else {
            if (depth > max_depth) {
                res = root->val;
                max_depth = depth;
            }
        }
    }
    int findBottomLeftValue(TreeNode* root) {
        findBottomLeftValue_core(root, 1);
        return res;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点总数。
- 空间复杂度：$O(h)$，其中 h 为树的高度。

题目地址（518. 零钱兑换 II）

https://leetcode-cn.com/problems/coin-change-2/

题目描述

```
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

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

注意:

你可以假设:

0 <= amount (总金额) <= 5000
1 <= coin (硬币面额) <= 5000
硬币种类不超过 500 种
结果符合 32 位符号整数
```

前置知识

- 动态规划
- 背包问题

公司

- 阿里
- 百度
- 字节

思路

这个题目和 coin-change 的思路比较类似。

进一步我们可以对问题进行空间复杂度上的优化（这种写法比较难以理解，但是相对更省空间）

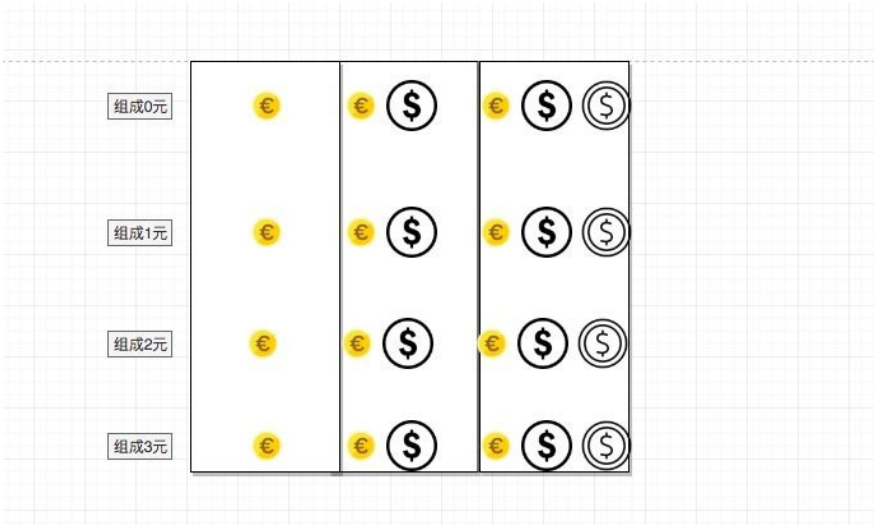
用 dp[i] 来表示组成 i 块钱，需要最少的硬币数，那么

1. 第 j 个硬币我可以选择不拿 这个时候， 组成数 = dp[i]
2. 第 j 个硬币我可以选择拿 这个时候， 组成数 = dp[i - coins[j]] + dp[i]
3. 和 01 背包问题不同， 硬币是可以拿任意个，属于完全背包问题
4. 对于每一个 dp[i] 我们都选择遍历一遍 coin， 不断更新 dp[i]

eg:

```
if (amount === 0) return 1;

const dp = [Array(amount + 1).fill(1)];

for (let i = 1; i < amount + 1; i++) {
  dp[i] = Array(coins.length + 1).fill(0);
  for (let j = 1; j < coins.length + 1; j++) {
    // 从1开始可以简化运算
    if (i - coins[j - 1] >= 0) {
      // 注意这里是coins[j -1]而不是coins[j]
      dp[i][j] = dp[i][j - 1] + dp[i - coins[j - 1]][j]; //
    } else {
      dp[i][j] = dp[i][j - 1];
    }
  }
}

return dp[dp.length - 1][coins.length];
```

- 当我们选择一维数组去解的时候，内外循环将会对结果造成影响



eg:

```
// 这种答案是不对的。
// 原因在于比如amount = 5, coins = [1,2,5]
// 这种算法会将[1,2,2] [2,1,2] [2, 2, 1] 算成不同的

if (amount === 0) return 1;

const dp = [1].concat(Array(amount).fill(0));

for (let i = 1; i < amount + 1; i++) {
  for (let j = 0; j < coins.length; j++) {
    if (i - coins[j] >= 0) {
      dp[i] = dp[i] + dp[i - coins[j]];
    }
  }
}

return dp[dp.length - 1];

// 正确的写法应该是内外循环调换一下, 具体可以看下方代码区
```

关键点解析

- 动态规划

代码

代码支持：Python3，JavaScript：

JavaSCript Code:

```javascript
/*
 * @lc app=leetcode id=518 lang=javascript
 *
 * [518] Coin Change 2
 *
 */
/**
 * @param {number} amount
 * @param {number[]} coins
 * @return {number}
 */
var change = function (amount, coins) {
  if (amount === 0) return 1;

  const dp = [1].concat(Array(amount).fill(0));

  for (let j = 0; j < coins.length; j++) {
    for (let i = 1; i < amount + 1; i++) {
      if (i - coins[j] >= 0) {
        dp[i] = dp[i] + dp[i - coins[j]];
      }
    }
  }

  return dp[dp.length - 1];
};
```

Python Code:

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount + 1)
        dp[0] = 1

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] += dp[i - coins[j]]

        return dp[-1]
```

复杂度分析

- 时间复杂度：$O(amount)$
- 空间复杂度：$O(amount * len(coins))$

扩展 1

这是一道很简单描述的题目， 因此很多时候会被用到大公司的电面中。

相似问题:

322.coin-change

扩展 2

Python 二维解法（不推荐，但是可以帮助理解）：

```
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [[0 for _ in range(len(coins) + 1)] for _ in r
        for j in range(len(coins) + 1):
            dp[0][j] = 1

        for i in range(amount + 1):
            for j in range(1, len(coins) + 1):
                if i >= coins[j - 1]:
                    dp[i][j] = dp[i - coins[j - 1]][j] + dp
                else:
                    dp[i][j] = dp[i][j - 1]
        return dp[-1][-1]
```

复杂度分析

- 时间复杂度：$O(amount * len(coins))$
- 空间复杂度：$O(amount * len(coins))$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（547. 朋友圈）

https://leetcode-cn.com/problems/friend-circles/

题目描述

```
班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递

给定一个 N \* N 的矩阵 M，表示班级中学生之间的朋友关系。如果 M[i

示例 1:

输入:
[[1,1,0],
[1,1,0],
[0,0,1]]
输出: 2
说明：已知学生 0 和学生 1 互为朋友，他们在一个朋友圈。
第 2 个学生自己在一个朋友圈。所以返回 2。
示例 2:

输入:
[[1,1,0],
[1,1,1],
[0,1,1]]
输出: 1
说明：已知学生 0 和学生 1 互为朋友，学生 1 和学生 2 互为朋友，所以学
注意：

N 在[1,200]的范围内。
对于所有学生，有 M[i][i] = 1。
如果有 M[i][j] = 1，则有 M[j][i] = 1。
```

前置知识

- 并查集

公司

- 阿里
- 腾讯
- 百度

- 字节

思路

并查集有一个功能是可以轻松计算出连通分量，然而本题的朋友圈的个数，本质上就是连通分量的个数，因此用并查集可以完美解决。

为了简单更加清晰，我将并查集模板代码单尽量独拿出来。

代码

`find`，`union`，`connected` 都是典型的模板方法。 懂的同学可能也发现了，我没有做路径压缩，这直接导致 find union connected 的时间复杂度最差的情况退化到 $O(N)$。

当然优化也不难，我们只需要给每一个顶层元素设置一个 size 用来表示连通分量的大小，这样 union 的时候我们将小的拼接到大的上即可。 另外 find 的时候我们甚至可以路径压缩，将树高限定到常数，这样时间复杂度可以降低到 $O(1)$。

```python
class UF:
    parent = {}
    cnt = 0
    def __init__(self, M):
        n = len(M)
        for i in range(n):
            self.parent[i] = i
            self.cnt += 1

    def find(self, x):
        while x != self.parent[x]:
            x = self.parent[x]
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        self.parent[self.find(p)] = self.find(q)
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)

class Solution:
    def findCircleNum(self, M: List[List[int]]) -> int:
        n = len(M)
        uf = UF(M)
        for i in range(n):
            for j in range(i):
                if M[i][j] == 1:
                    uf.union(i, j)
        return uf.cnt
```

复杂度分析

- 时间复杂度：平均 $O(logN)$，最坏的情况是 $O(N)$
- 空间复杂度：我们使用了 parent， 因此空间复杂度为 $O(N)$

相关专题

- 并查集专题

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(560. 和为 K 的子数组)

https://leetcode-cn.com/problems/subarray-sum-equals-k/

题目描述

```
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

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 :

数组的长度为 [1, 20,000]。
数组中元素的范围是 [-1000, 1000] ，且整数 k 的范围是 [-1e7, 1e7]
```

前置知识

- 哈希表
- 前缀和

公司

- 阿里
- 腾讯
- 字节

思路

符合直觉的做法是暴力求解所有的子数组，然后分别计算和，如果等于 k,count 就+1.这种做法的时间复杂度为 O(n^2)，代码如下：

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt, n =  0, len(nums)
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += nums[j]
                if (sum == k): cnt += 1
        return cnt
```

实际上刚开始看到这题目的时候，我想“是否可以用滑动窗口解决？”。但是很快我就放弃了，因为看了下数组中项的取值范围有负数，这样我们扩张或者收缩窗口就比较复杂。第二个想法是前缀和，保存一个数组的前缀和，然后利用差分法得出任意区间段的和，这种想法是可行的，代码如下：

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
      cnt, n =  0, len(nums)
      pre = [0] * (n + 1)
      for i in range(1, n + 1):
          pre[i] = pre[i - 1] + nums[i - 1]
      for i in range(1, n + 1):
          for j in range(i, n + 1):
              if (pre[j] - pre[i - 1] == k): cnt += 1
      return cnt
```

然而题目要求的仅仅是求**总数**，而不需要把所有的子数组求出。因此我们可直接使用下面的时间复杂度为 $O(n)$ 的算法。

这种做法的核心点在于， 题目让求的子数组总个数其实等价于：

- 以索引 0 结尾的子数组个数
- 以索引 1 结尾的子数组个数
- 。。。
- 以索引 n - 1 结尾的子数组个数

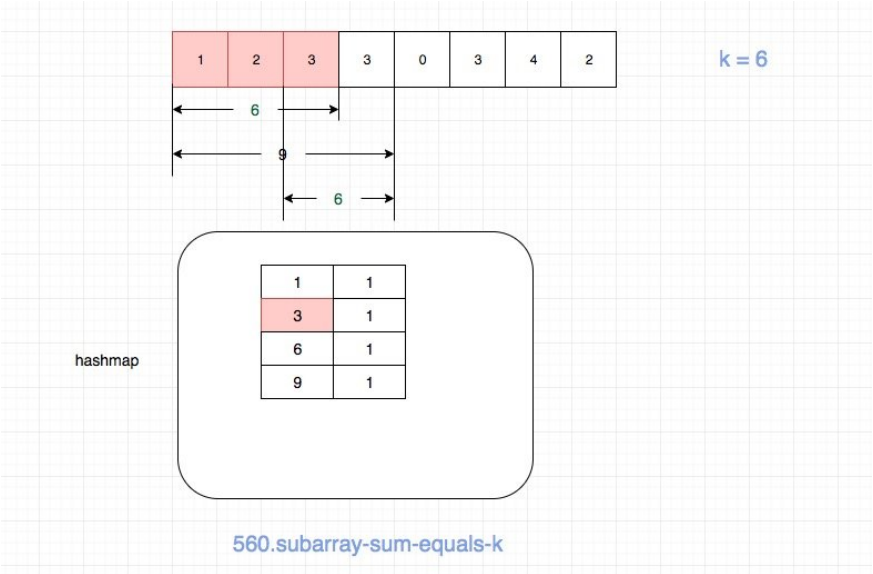
而以索引 i 结尾的子数组个数等于：前缀和为 acc - k 的子数组个数，其中 acc 为当前的前缀和。为了能够快速取出前缀和为 acc - k 的个数，我们可将其存到哈希中。

具体算法：

- 维护一个 hashmap，hashmap 的 key 为累加值 acc，value 为累加值 acc 出现的次数。
- 迭代数组，然后不断更新 acc 和 hashmap，如果 acc 等于 k，那么很明显应该+1. 如果 hashmap[acc - k] 存在，我们就把它加到结果中

去即可。

语言比较难以解释，我画了一个图来演示 nums = [1,2,3,3,0,3,4,2], k = 6 的情况。



如图，当访问到 nums[3]的时候，hashmap 如图所示，这个时候 count 为 2. 其中之一是[1,2,3],这个好理解。还有一个是[3,3].

这个[3,3]正是我们通过 hashmap[acc - k]即 hashmap[9 - 6]得到的。

关键点解析

- 前缀和
- 可以利用 hashmap 记录和的累加值来避免重复计算

代码

- 语言支持：JS, Python

Javascript Code:

```javascript
/*
 * @lc app=leetcode id=560 lang=javascript
 *
 * [560] Subarray Sum Equals K
 */
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var subarraySum = function (nums, k) {
  const hashmap = {};
  let acc = 0;
  let count = 0;

  for (let i = 0; i < nums.length; i++) {
    acc += nums[i];

    if (acc === k) count++;

    if (hashmap[acc - k] !== void 0) {
      count += hashmap[acc - k];
    }

    if (hashmap[acc] === void 0) {
      hashmap[acc] = 1;
    } else {
      hashmap[acc] += 1;
    }
  }

  return count;
};
```

Python Code:

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        d = {}
        acc = count = 0
        for num in nums:
            acc += num
            if acc == k:
                count += 1
            if acc - k in d:
                count += d[acc-k]
            if acc in d:
                d[acc] += 1
            else:
                d[acc] = 1
        return count
```

扩展

这是一道类似的题目，但是会稍微复杂一点, 题目地址: 437.path-sum-iii

题目地址(609. 在系统中查找重复文件)

https://leetcode-cn.com/problems/find-duplicate-file-in-system/

题目描述

给定一个目录信息列表，包括目录路径，以及该目录中的所有包含内容的文件，

输入列表中的单个目录信息字符串的格式如下：

"root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ..

这意味着有 n 个文件（f1.txt, f2.txt ... fn.txt 的内容分别是 f1_c

该输出是重复文件路径组的列表。对于每个组，它包含具有相同内容的文件的所

"directory_path/file_name.txt"

示例 1：

输入：
["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "r
输出：
[["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.

注：

最终输出不需要顺序。
您可以假设目录名、文件名和文件内容只有字母和数字，并且文件内容的长度在
给定的文件数量在 [1, 20000] 个范围内。
您可以假设在同一目录中没有任何文件或目录共享相同的名称。
您可以假设每个给定的目录信息代表一个唯一的目录。目录路径和文件信息用一

超越竞赛的后续行动：

假设您有一个真正的文件系统，您将如何搜索文件？广度搜索还是宽度搜索？
如果文件内容非常大（GB级别），您将如何修改您的解决方案？
如果每次只能读取 1 kb 的文件，您将如何修改解决方案？
修改后的解决方案的时间复杂度是多少？其中最耗时的部分和消耗内存的部分是
如何确保您发现的重复文件不是误报？

前置知识

- 哈希表

思路

思路就是hashtable去存储，key为文件内容，value为fullfilename， 遍历一遍去填充hashtable， 最后将hashtable中的值打印出来即可。

> 当且仅当有重复内容，我们才打印，因此我们需要过滤一下， 类似filter(q => q.length >= 2)
>
> ### 关键点解析

- hashtable

代码

```
/**
 * @param {string[]} paths
 * @return {string[][]}
 */
var findDuplicate = function(paths) {
  const hashmap = {};

  for (let path of paths) {
    const [folder, ...files] = path.split(" ");
    for (let file of files) {
      const lpi = file.indexOf("(");
      const rpi = file.lastIndexOf(")");
      const filename = file.slice(0, lpi);
      const content = file.slice(lpi, rpi);
      const fullname = `${folder}/${filename}`;
      if (!hashmap[content]) hashmap[content] = [];
      hashmap[content].push(fullname);
    }
  }

  return Object.values(hashmap).filter(q => q.length >= 2);
};
```

题目地址(611. 有效三角形的个数)

https://leetcode-cn.com/problems/valid-triangle-number/

题目描述

```
给定一个包含非负整数的数组，你的任务是统计其中可以组成三角形三条边的三

示例 1:

输入: [2,2,3,4]
输出: 3
解释:
有效的组合是:
2,3,4 (使用第一个 2)
2,3,4 (使用第二个 2)
2,2,3
注意:

数组长度不超过1000。
数组里整数的范围为 [0, 1000]。
```

前置知识

- 排序
- 双指针
- 二分法
- 三角形边的关系

公司

- 腾讯
- 百度
- 字节

暴力法（超时）

思路

首先要有一个数学前提：`如果三条线段中任意两条的和都大于第三边，那么这三条线段可以组成一个三角形`。即给定三个线段 a，b，c，如果满足 a + b > c and a + c > b and b + c > a，则线段 a，b，c 可以构成三角形，否则不可以。

力扣中有一些题目是需要一些数学前提的，不过这些数学前提都比较简单，一般不会超过高中数学知识，并且也不会特别复杂。一般都是小学初中知识即可。

> 如果你在面试中碰到不知道的数学前提，可以寻求面试官提示试试。

关键点解析

- 三角形边的关系
- 三层循环确定三个线段

代码

代码支持: Python

```
class Solution:
    def is_triangle(self, a, b, c):
        if a == 0 or b == 0 or c == 0: return False
        if a + b > c and a + c > b and b + c > a: return Tr
        return False
    def triangleNumber(self, nums: List[int]) -> int:
        n = len(nums)
        ans = 0
        for i in range(n - 2):
            for j in range(i + 1, n - 1):
                for k in range(j + 1, n):
                    if self.is_triangle(nums[i], nums[j], r

        return ans
```

复杂度分析

- 时间复杂度：$O(N ^ 3)$，其中 N 为 数组长度。
- 空间复杂度：$O(1)$

优化的暴力法

思路

暴力法的时间复杂度为 $O(N ^ 3)$， 其中 N 最大为 1000。一般来说， $O(N ^ 3)$ 的算法在数据量 <= 500 是可以 AC 的。1000 的数量级则需要考虑 $O(N ^ 2)$ 或者更好的解法。

OK，到这里了。我给大家一个干货。 应该是其他博主不太会提的。原因可能是他们不知道， 也可能是他们觉得太小儿科不需要说。

1. 由于前面我根据数据规模推测到到了解法的复杂度区间是 $N ^ 2$, $N ^ 2 * logN$，不可能是 N （WHY？）。
2. 降低时间复杂度的方法主要有： `空间换时间` 和 `排序换时间` （我们一般都是使用基于比较的排序方法）。而 `排序换时间` 仅仅在总体复杂度大于 $O(NlogN)$ 才适用（原因不用多说了吧？）。

这里由于总体的时间复杂度是 $O(N ^ 3)$，因此我自然想到了 `排序换时间` 。当我们对 nums 进行一次排序之后，我发现：

- is_triangle 函数有一些判断是无效的

```
    def is_triangle(self, a, b, c):
        if a == 0 or b == 0 or c == 0: return False
        # a + c > b 和  b + c > a 是无效的判断，因为恒成立
        if a + b > c and a + c > b and b + c > a: return Tr
        return False
```

- 因此我们的目标变为找到 `a + b > c` 即可，因此第三层循环是可以提前退出的。

```
for i in range(n - 2):
    for j in range(i + 1, n - 1):
        k = j + 1
        while k < n and num[i] + nums[j] > nums[k]:
            k += 1
        ans += k - j - 1
```

- 这也仅仅是减枝而已，复杂度没有变化。通过进一步观察，发现 k 没有必要每次都从 j + 1 开始。而是从上次找到的 k 值开始就行。原因很简单， 当 nums[i] + nums[j] > nums[k] 时，我们想要找到下一个满足 nums[i] + nums[j] > nums[k] 的 新的 k 值，由于进行了排序，因此这个 k 肯定比之前的大（单调递增性），因此上一个 k 值之前的数都是无效的，可以跳过。

```
for i in range(n - 2):
    k = i + 2
    for j in range(i + 1, n - 1):
        while k < n and nums[i] + nums[j] > nums[k]:
            k += 1
        ans += k - j - 1
```

由于 K 不会后退，因此最内层循环总共最多执行 N 次，因此总的时间复杂度为 $O(N ^ 2)$。

> 这个复杂度分析有点像单调栈，大家可以结合起来理解。

关键点分析

- 排序

代码

```
class Solution:
    def triangleNumber(self, nums: List[int]) -> int:
        n = len(nums)
        ans = 0
        nums.sort()
        for i in range(n - 2):
            if nums[i] == 0: continue
            k = i + 2
            for j in range(i + 1, n - 1):
                while k < n and nums[i] + nums[j] > nums[k]
                    k += 1
                ans += k - j - 1
        return ans
```

复杂度分析

- 时间复杂度：$O(N ^ 2)$
- 空间复杂度：取决于排序算法

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(673. 最长递增子序列的个数)

https://leetcode-cn.com/problems/number-of-longest-increasing-subsequence/

题目描述

```
给定一个未排序的整数数组，找到最长递增子序列的个数。

示例 1:

输入: [1,3,5,4,7]
输出: 2
解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。
示例 2:

输入: [2,2,2,2,2]
输出: 5
解释: 最长递增子序列的长度是1，并且存在5个子序列的长度为1，因此输出5。
注意: 给定的数组长度不超过 2000 并且结果一定是32位有符号整数。
```

前置知识

- 动态规划

公司

- 暂无

思路

这道题其实就是**最长上升子序列（LIS）**的变种题。如果对 LIS 不了解的可以先看下我之前写的一篇文章穿上衣服我就不认识你了？来聊聊最长上升子序列，里面将这种题目的套路讲得很清楚了。

回到这道题。题目让我们求最长递增子序列的个数，而不是通常的**最长递增子序列的长度**。 因此我想到使用另外一个变量记录**最长递增子序列的个数**信息即可。类似的套路有**股票问题**，这种问题的套路在于只是单独存储一个状态以无法满足条件，对于这道题来说，我们存储的单一状态就是**最长递增子序列的长度**。那么一个自然的想法是**不存储最长递增子序列的长度，而是仅存储最长递增子序列的个数**可以么？这是不可以的，因为**最长递增子序列的个数** 隐式地条件是你要先找到最长的递增子序列才行。

如何存储两个状态呢？一般有两种方式：

- 二维数组 dp[i][0] 第一个状态 dp[i][1] 第二个状态
- dp1[i] 第一个状态 dp2[i] 第二个状态

使用哪个都可以，空间复杂度也是一样的，使用哪种看你自己。这里我们使用第一种，并且 dp[i][0] 表示 以 nums[i] 结尾的最长上升子序列的长度，dp[i][1] 表示 以 nums[i] 结尾的长度为 dp[i][0] 的子序列的个数。

明确了要多存储一个状态之后，我们来看下状态如何转移。

LIS 的一般过程是这样的：

```
for i in range(n):
    for j in range(i + 1, n):
        if nums[j] > nums[i]:
            # ...
```

这道题也是类似，遍历到 nums[j] 的时候往前遍历所有的 满足 i < j 的 i。

- 如果 nums[j] <= nums[i]， nums[j] 无法和前面任何的序列拼接成**递增子序列**
- 否则说明我们可以拼接。但是拼接与否取决于拼接之后会不会更长。如果更长了就拼，否则不拼。

上面是 LIS 的常规思路，下面我们加一点逻辑。

- 如果拼接后的序列更长，那么 dp[j][1] = dp[i][1] （这点容易忽略）
- 如果拼接之后序列一样长， 那么 dp[j][1] += dp[i][1]。
- 如果拼接之后变短了，则不应该拼接。

关键点解析

- 最长上升子序列问题
- dp[j][1] = dp[i][1] 容易忘记

代码

代码支持: Python

```python
class Solution:
    def findNumberOfLIS(self, nums: List[int]) -> int:
        n = len(nums)
        # dp[i][0] ->  LIS
        # dp[i][1] -> NumberOfLIS
        dp = [[1, 1] for i in range(n)]
        ans = [1, 1]
        longest = 1
        for i in range(n):
            for j in range(i + 1, n):
                if nums[j] > nums[i]:
                    if dp[i][0] + 1 > dp[j][0]:
                        dp[j][0] = dp[i][0] + 1
                        # 下面这行代码容易忘记，导致出错
                        dp[j][1] = dp[i][1]
                        longest = max(longest, dp[j][0])
                    elif dp[i][0] + 1 == dp[j][0]:
                        dp[j][1] += dp[i][1]
        return sum(dp[i][1] for i in range(n) if dp[i][0] =
```

复杂度分析

令 N 为数组长度。

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

扩展

这道题也可以使用线段树来解决，并且性能更好，不过由于不算是常规解法，因此不再这里展开，感兴趣的同学可以尝试一下。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(686. 重复叠加字符串匹配)

https://leetcode-cn.com/problems/repeated-string-match/description/

题目描述

```
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

注意：字符串 "abc" 重复叠加 0 次是 ""，重复叠加 1 次是 "abc"，重复



示例 1：

输入：a = "abcd", b = "cdabcdab"
输出：3
解释：a 重复叠加三遍后为 "abcdabcdabcd"，此时 b 是其子串。
示例 2：

输入：a = "a", b = "aa"
输出：2
示例 3：

输入：a = "a", b = "a"
输出：1
示例 4：

输入：a = "abc", b = "wxyz"
输出：-1


提示：

1 <= a.length <= 104
1 <= b.length <= 104
a 和 b 由小写英文字母组成
```

前置知识

- set

公司

- 暂无

思路

首先，一个容易发现的点是**如果 b 中包含有 a 中没有的字符， 那么一定需要返回 - 1**。因此使用集合存储 a 和 b 的所有字符，并比较 b 是否是 a 的子集，如果不是，则直接返回 - 1。

接着我们逐个尝试：

- 两个 a 是否可以?
- 三个 a 是否可以?
- 。。。
- n 个 a 是否可以?

如果可以，则直接返回 n 即可。关于是否可以的判断， 我们可以使用任何语言自带的 indexof 算法，Python 中 可以使用 `b in a` 判读 b 时候是 a 的子串。

代码:

```
cnt = 1
while True:
    if b in a * cnt:
        return cnt
    cnt += 1
return -1
```

上面的代码有 BUG，会在一些情况无限循环。比如：

```
 a = "abcabcabcabc"
 b = "abac"
```

因此我们必须设计出口，并返回 -1。问题的我们的上界是什么呢?

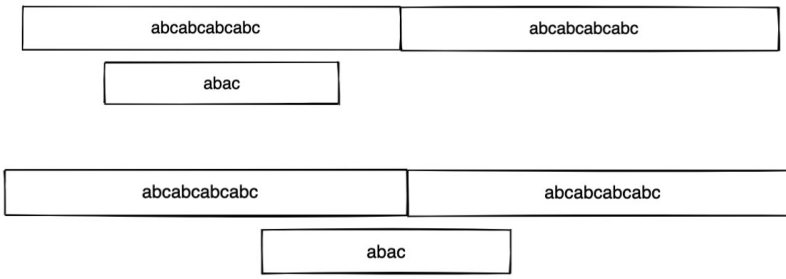
这里有个概念叫**解空间**。这是一个很重要的概念。 我举个简单的例子。你要在一个数组 A 中找某一个数的索引，题目保证这个数字一定在数组中存在。那么这道题的解空间就是**[0, n -1]**，其中 n 为数组长度。你的解不可能在这个范围外。

回到本题，如果 a 经过 n 次可以匹配成功， 那么最终 a 的长度范围是 [len(b), 2 * len(a) + len(b)]，下界是 len(b) 容易理解， 关键是上界。

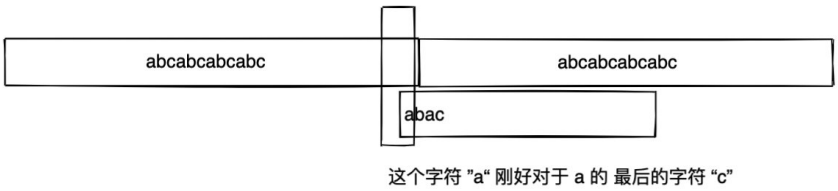
还是以上面的例子来说。

```
a = "abcabcabcabc"
b = "abac"
```

abac 如果可以在其中匹配到，一定是以下几种情况：



临界情况就是：



这个字符 ”a“ 刚好对于 a 的 最后的字符 “c”

因此最终 a 的长度的临界值就是 2 * len(a) + len(b)。**超过这个范围再多次的叠加也没有意义。**

关键点解析

- 答案是有限的， 搞清楚解空间是关键

代码

代码支持: Python

```python
class Solution:
    def repeatedStringMatch(self, a: str, b: str) -> int:
        if not set(b).issubset(set(a)):
            return -1
        cnt = 1
        while len(a * cnt) < 2 * len(a) + len(b):
            if b in a * cnt:
                return cnt
            cnt += 1
        return -1
```

复杂度分析

- 时间复杂度：b in a 的时间复杂度为 M + N（取决于内部算法），因此总的时间复杂度为 $O((M + N) ^ 2)$，其中 M 和 N 为 a 和 b 的长度。
- 空间复杂度：由于使用了 set，因此空间复杂度为 $O(M +N)$，其中 M 和 N 为 a 和 b 的长度。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(714. 买卖股票的最佳时机含手续费)

https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/

题目描述

```
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

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

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

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.
```

前置知识

- 动态规划

公司

- 暂无

记忆化递归

思路

首先明确一点。 如果没有交易费用，那么和另一道力扣的简单难度题目一样。 多了交易费用导致问题稍微复杂了一点。加了交易费用有什么不同呢?

举一个例子大家就明白了。 比如 prices = [1,1,2,2,6] fee = 3。 如果没有按照无交易费的原则**只要有利可图就交易**，那么我们的收益为 1。实际上，在 1 买入， 6 卖出却可以得到 3 的收益。其根本原因在于**交易次数对结果是有影响的**。

这道题不能使用上述的贪心策略，而必须使用动态规划。

> 动态规划和贪心有着很强的关联性。

定义 dp[i] 为到第 i 天(0=< i <n)为止的收益，那么答案就是 dp[n-1]。但是由于题目限定了只能**在手上没有股票的时候才能买，手上有股票的时候才能卖（这很显然）**。因此我们需要多定义一个维度的状态，表示现在手上是否有股票。标识是否手上有股票只需要一个长度为 2 的列表就可以了。

因此我们可以使用 dp[i][0] 表示没有股票的情况下的最大利润，dp[i][1] 表示有股票的情况下的最大利润。

那么：

- 如果第 i 天手上没有股票，要么是 i - 1 没股票，今天什么都不做。要么是 i - 1 手上有股票，今天卖了它。因此 dp[i][0] 就是这两种情况的利润最大值。
- 如果第 i 天手上有股票，要么是 i - 1 有股票，今天什么都不做。要么是 i - 1 手上没有股票，今天买了它。因此 dp[i][0] 就是这两种情况的利润最大值。

最后返回 dp[n-1][0] 即可。

> 由于 dp[n-1][1] <= dp[n-1][0] ， 因此直接返回 dp[n-1][0] 即可。

base case 见下方代码的递归出口。

关键点

- 记忆化递归

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int
        def dp(i):
            if i == 0:
                return 0, -prices[0] - fee
            sell, buy = dp(i - 1)
            return max(sell, buy + prices[i]), max(buy, se

        return dp(len(prices) - 1)[0]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

DP

思路

使用 dp 的思路和上面一样，只是代码形式不同而已。

关键点

- 滚动数组优化技巧

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int
        n = len(prices)
        dp = [[0 for i in range(2)]] * n
        for i in range(n):
            if i == 0:
                dp[i][0] = 0
                dp[i][1] = -1 * prices[i]
            else:
                dp[i][0] = max(dp[i - 1][1] + prices[i] - f
                dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] -

        return dp[-1][0]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

使用 DP 数组的好处就是可以使用滚动数组优化。比如这道题使用滚动数组可将空间进一步压缩。

```
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int
        n = len(prices)
        # [手里没股票，手里有股票]
        dp = [0, 0]
        for i in range(n):
            if i == 0:
                dp[0] = 0
                dp[1] = -1 * prices[i] - fee
            else:
                dp[0] = max(dp[0], dp[1] + prices[i])
                dp[1] = max(dp[1], dp[0] - prices[i] - fee)

        return dp[0]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（718. 最长重复子数组）

https://leetcode-cn.com/problems/maximum-length-of-repeated-subarray/

题目描述

```
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

示例 1:

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]。
说明:

1 <= len(A), len(B) <= 1000
0 <= A[i], B[i] < 100
```

前置知识

- 哈希表
- 数组
- 二分查找
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

关于这个类型， 我专门写了一个专题《你的衣服我扒了 - 《最长公共子序列》》，里面讲了三道题，其中就有这个。

这就是最经典的最长公共子序列问题。一般这种求解**两个数组或者字符串求最大或者最小**的题目都可以考虑动态规划，并且通常都定义 dp[i][j] 为 `以 A[i], B[j] 结尾的 xxx`。这道题就是：`以 A[i], B[j] 结尾的两个数组中公共的、长度最长的子数组的长度`。算法很简单：

- 双层循环找出所有的 i, j 组合，时间复杂度 $O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
  - 如果 A[i] == B[j]，dp[i][j] = dp[i - 1][j - 1] + 1
  - 否则，dp[i][j] = 0
- 循环过程记录最大值即可。

关键点解析

- dp 建模套路

代码

代码支持：Python

Python Code:

```
class Solution:
    def findLength(self, A, B):
        m, n = len(A), len(B)
        ans = 0
        dp = [[0 for _ in range(n + 1)] for _ in range(m +
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if A[i - 1] == B[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
        return ans
```

复杂度分析

- 时间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。

更多

- 你的衣服我扒了 - 《最长公共子序列》

扩展

二分查找也是可以的，不过并不容易想到，大家可以试试。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(754. 到达终点数字)

https://leetcode-cn.com/problems/reach-a-number/

题目描述

```
在一根无限长的数轴上，你站在0的位置。终点在target的位置。

每次你可以选择向左或向右移动。第 n 次移动（从 1 开始），可以走 n 步。

返回到达终点需要的最小移动次数。

示例 1:

输入: target = 3
输出: 2
解释:
第一次移动，从 0 到 1 。
第二次移动，从 1 到 3 。
示例 2:

输入: target = 2
输出: 3
解释:
第一次移动，从 0 到 1 。
第二次移动，从 1 到 -1 。
第三次移动，从 -1 到 2 。
注意:

target是在[-10^9, 10^9]范围中的非零整数。
```

前置知识

- 数学

公司

思路

不难看出， 问题的本质就是一个有限序列， 1，2，3，4... 。我们的目标是给这个序列的元素添加正负号，是的其和为 target。这和 494.target-sum 的思路是一样的。

拿题目的 target = 3 来说， 就是 1 + 2 = 3。

拿题目的 target = 2 来说， 就是 1 - 2 + 3 = 2。

为什么是有限序列?

因为我们始终可以在 target 次以内走到 target。严格来说， 最少在根号 target 左右就可以走到 target。

和 494.target-sum 不同的是， 这道题数组是无限的，看起来似乎更难，实际上更简单， 因为数组是有规律的，每次都递增 1。

> 由于 target 正负是对称的， 因此 target 最少走多少布，-target 也是多少步。因此我们只考虑一种情况即可， 不妨只考虑正数的情况。

其实，只要找出第一个满足 1 + 2 + 3 + ... + steps > target 的 steps 即可。

- 如果 steps 是偶数， 那么我们总可以找出若干数字使其变为符号，满足 1 + 2 + 3 + ... + steps == target
- 如果 steps 是奇数， 1 + 2 + 3 + ... + steps + steps + 1 或者 1 + 2 + 3 + ... + steps + steps + 1 + steps + 2 中有且仅有一个是偶数。我们仍然可以套用上面的方法，找出若干数字使其变为符号，满足 1 + 2 + 3 + ... + steps + steps + 1 == target 或者 1 + 2 + 3 + ... + steps + steps + 1 + steps + 2 == target

关键点解析

- 对元素进行分组，分组的依据是符号， 是 `+` 或者 `-`
- 通过数学公式推导可以简化我们的求解过程，这需要一点 `数学知识和数学意识`

代码(Python)

Python Code:

```
class Solution(object):
    def reachNumber(self, target):
        target = abs(target)
        steps = 0
        while target > 0:
            steps += 1
            target -= steps
        if target & 1 == 0: return steps
        steps += 1
        if (target - steps) & 1 == 0: return steps
        return steps + 1
```

相关题目

- 494.target-sum

题目地址（785. 判断二分图）

https://leetcode-cn.com/problems/is-graph-bipartite/

题目描述

```
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

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

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。
```

前置知识

- 图的遍历
- DFS

公司

- 暂无

思路

和 886 思路一样。 我甚至**直接拿过来 dfs 函数一行代码没改就 AC 了**。

唯一需要调整的地方是 graph 。 我将其转换了一下，具体可以看代码，非常简单易懂。

具体算法：

- 设置一个长度为 N 的数组 colors，colors[i] 表示 节点 i 的颜色，0 表示无颜色， 1 表示一种颜色， - 1 表示另一种颜色。
- 初始化 colors 全部为 0
- 构图（这里有邻接矩阵） 使得 grid[i][j] 表示 i 和 j 是否有连接（这里用 0 表示无， 1 表示有）
- 遍历图。
  - 如果当前节点未染色，则染色，不妨染为颜色 1
  - 递归遍历其邻居
    - 如果邻居没有染色， 则染为另一种颜色。即 color * - 1，其中 color 为当前节点的颜色
    - 否则，判断当前节点和邻居的颜色是否一致，不一致则返回 False，否则返回 True

强烈建议两道题一起练习一下。

关键点

- 图的建立和遍历
- colors 数组

代码

```
class Solution:
    def dfs(self, grid, colors, i, color, N):
        colors[i] = color
        for j in range(N):
            if grid[i][j] == 1:
                if colors[j] == color:
                    return False
                if colors[j] == 0 and not self.dfs(grid, co
                    return False
        return True

    def isBipartite(self, graph: List[List[int]]) -> bool:
        N = len(graph)
        grid = [[0] * N for _ in range(N)]
        colors = [0] * N
        for i in range(N):
            for j in graph[i]:
                grid[i][j] = 1
        for i in range(N):
            if colors[i] == 0 and not self.dfs(grid, colors
                return False
        return True
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

相关问题

- 886. 可能的二分法

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(790. 多米诺和托米诺平铺)

https://leetcode-cn.com/problems/domino-and-tromino-tiling/

题目描述

```
有两种形状的瓷砖：一种是 2x1 的多米诺形，另一种是形如 "L" 的托米诺形

XX  <- 多米诺

XX  <- "L" 托米诺
X


给定 N 的值，有多少种方法可以平铺 2 x N 的面板？返回值 mod 10^9 +

（平铺指的是每个正方形都必须有瓷砖覆盖。两个平铺不同，当且仅当面板上有

示例：
输入：3
输出：5
解释：
下面列出了五种不同的方法，不同字母代表不同瓷砖：
XYZ XXZ XYY XXY XYY
XYZ YYZ XZZ XYY XXY

提示：

N  的范围是 [1, 1000]
```

前置知识

- 动态规划

公司

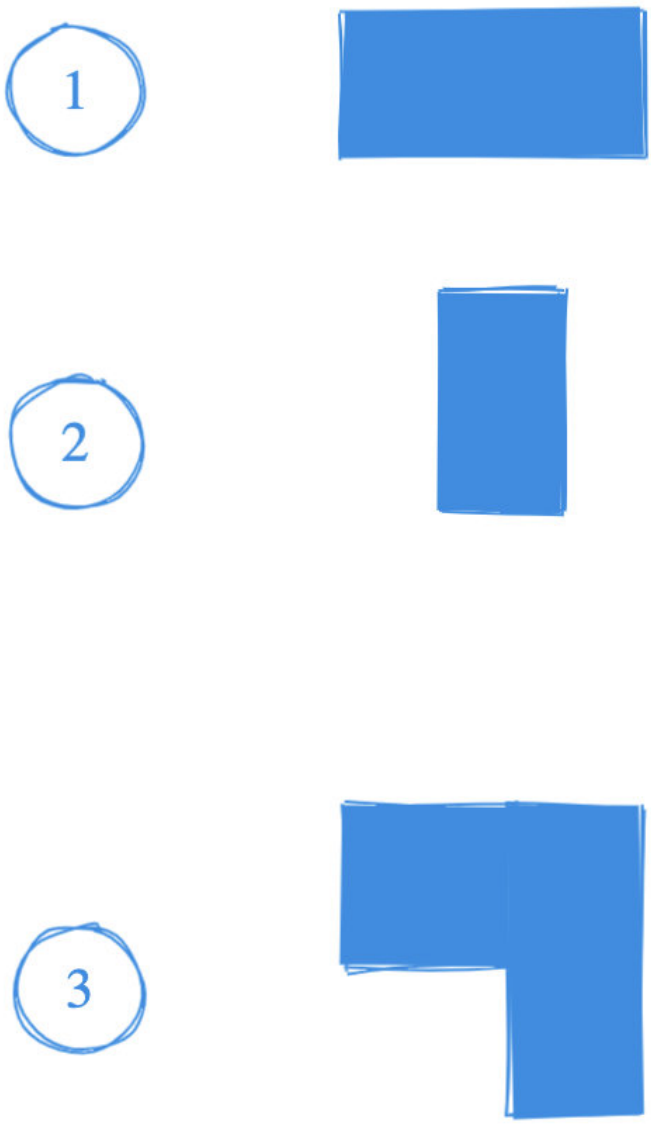
- 暂无

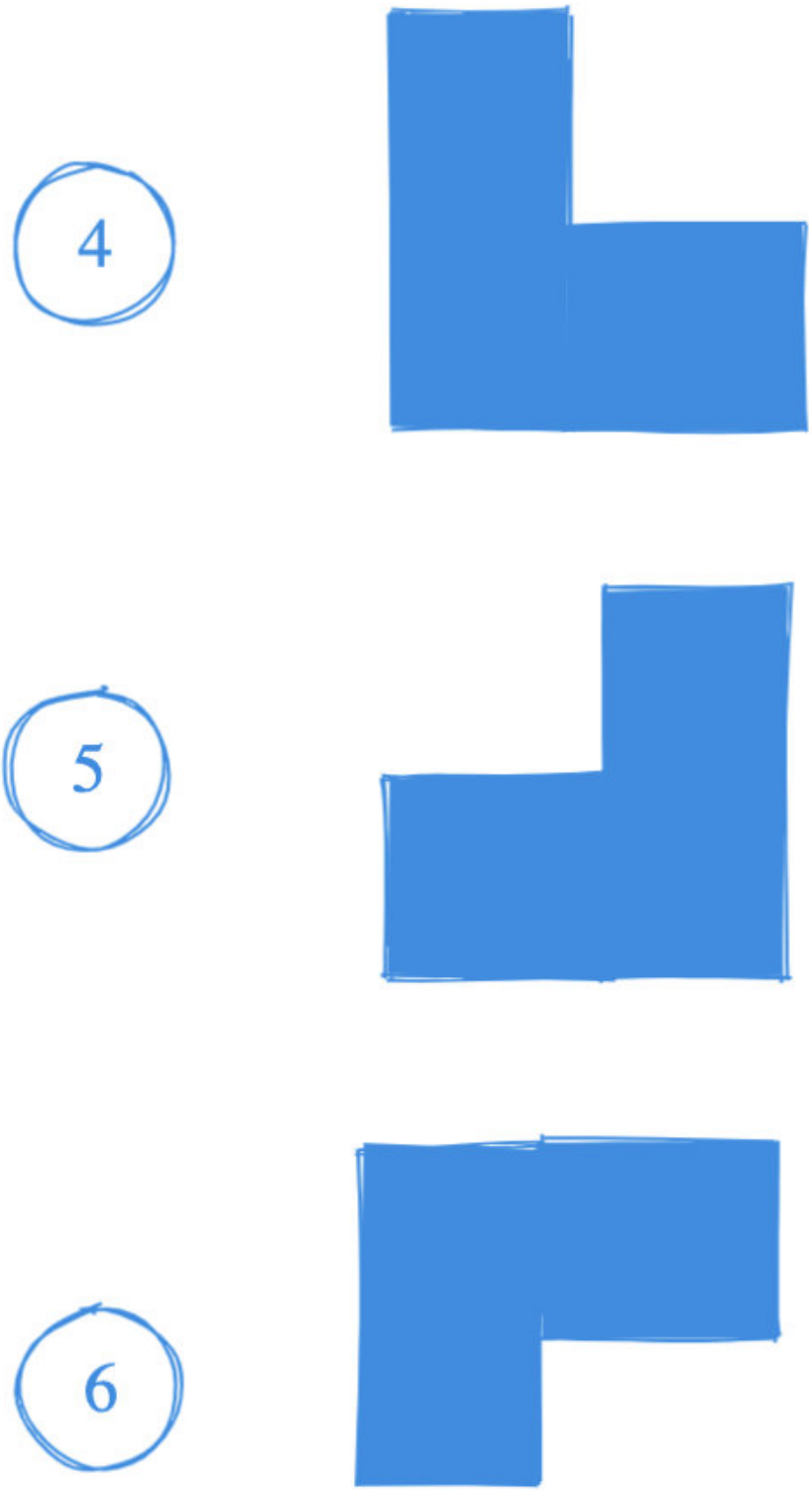
思路

这种题目和铺瓷砖一样，这种题目基本都是动态规划可解。做这种题目的诀窍就是将所有的可能都列举出来，然后分析问题的最优子结构。最后根据子问题之间的递推关系解决问题。

如果题目只有 XX 型，或者只有 L 型，实际上就很简单了。而这道题是 XX 和 L，则稍微有点难度。力扣还有一些其他更复杂度的铺瓷砖，都是给你若干瓷砖，让你刚好铺满一个形状。大家做完这道题之后可以去尝试一下其他题相关目。

以这道题来说，所有可能的情况无非就是以下 6 种：





而题目要求的是**刚好铺满** 2 * N 的情况的总的可能数。

如上图 1，2，3，5 可能是刚好铺满 2 *N 的瓷砖的**最后一块砖**，换句话说 4 和 6 不能是刚好铺满 2* N 的最后一块瓷砖。

为了方便描述，我们令 F(n) 表示刚好铺满 2 * n 的瓷砖的总的可能数，因此题目要求的其实就是 F(n)。

- 如果最后一块选择了形状 1，那么**此时**的刚好铺满 2 * n 的瓷砖的总的可能数是 F(n-2)
- 如果最后一块选择了形状 2，那么**此时**的刚好铺满 2 * n 的瓷砖的总的可能数是 F(n-1)
- 如果最后一块选择了形状 3，那么**此时**的刚好铺满 2 * n 的瓷砖的总的可能数是？
- 如果最后一块选择了形状 5，那么**此时**的刚好铺满 2 * n 的瓷砖的总的可能数是？
- 如果最后一块选择了形状 4 和 6，那么**此时**的刚好铺满 2 * n 的瓷砖的总的可能数是 0。换句话说 4 和 6 不可能是刚好铺满的最后一块砖。

虽然 4 和 6 不可能是刚好铺满的最后一块砖，但其实可以是中间状态，中间状态可以进一步转移到**刚好铺满的状态**。比如股票问题就是这样，虽然我们的最终答案不可能是买入之后，一定是卖出，但是中间的过程可以卖出，通过卖出转移到最终状态。

现在的问题是如何计算：最后一块选择了形状 3 和 最后一块选择了形状 5 的总的可能数，以及 4 和 6 在什么情况下选择。实际上，我们只需要考虑选择我 3 和 5 的总的可能数就行了，其原因稍后你就知道了。

为了表示所有的情况，我们需要另外一个状态定义和转移。 我们令 T(n) 表示刚好铺满 2 * (N - 1)瓷砖，最后一列只有一块瓷砖的总的可能数。对应上图中的 4 和 6。

经过这样的定义，那么就有：

- 如果最后一块选择了形状 3，那么刚好铺满 2 * (N - 1)瓷砖，最后一列只有一块瓷砖的总的可能数是 T(n -1)
- 如果最后一块选择了形状 5，那么刚好铺满 2 * (N - 1)瓷砖，最后一列只有一块瓷砖的总的可能数是 T(n-1)

> 大家可以根据基本图形画一下试试就知道了。同时你也应该理解了 4 和 6 在什么情况下使用。

根据以上的信息有如下公式:

```
F(n) = F(n−1) + F(n−2) + 2 ∗ T(n−1)
```

由于上述等式有两个变量，因此至少需要两个这样的等式才可解。而上面的等式是 F(n) = xxx，因此一个直觉找到一个类似 T(n) = xxx 的公式。不难发现如下等式：

```
T(n) = F(n−2) + T(n−1)
```

将上面两个公式进行合并。具体来说就是：

```
F(n) = F(n-1) + F(n-2) + 2 * T(n-1)
T(n) = F(n-2) + T(n-1) -> T(n-1) = F(n-3) + T(n-2) -> 2 * T
```

进一步：

```
F(n) = F(n-1) + 2 * F(n-3) + F(n-2) + 2T(n-2) = F(n-1) + F(
```

至此，我们得出了状态转移方程：

```
F(n) = 2 * F(n-1) + F(n-3)
```

关键点

- 识别最优子结构
- 对一块瓷砖能拼成的图形进行分解，并对每一种情况进行讨论

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def numTilings(self, N: int) -> int:
        dp = [0] * (N + 3)
        # f(3) = 2 * f(2) + f(0) = 2 + f(0) = 1 -> f(0) = -
        # f(4) = 2 * f(3) + f(1) = 2 + f(1) = 2 -> f(1) = 0
        dp[0] = -1
        dp[1] = 0
        dp[2] = 1
        # f(n) = f(n-1) + f(n-2) + 2 * T(n-1)
        # 2 * T(n-1) = 2 * f(n-3) + 2 * T(n-2)
        # f(n) = f(n-1) + 2 * f(n-3) + f(n-2) + 2T(n-2) = f
        for i in range(3, N + 3):
            dp[i] = 2 * dp[i-1] + dp[i-3]
        return dp[-1] % (10 ** 9 + 7)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$

- 空间复杂度：$O(n)$

使用滚动数组优化可以将空间复杂度降低到 $O(1)$，大家可以试试。

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(799. 香槟塔)

https://leetcode-cn.com/problems/champagne-tower/

题目描述

```
我们把玻璃杯摆成金字塔的形状，其中第一层有1个玻璃杯，第二层有2个，依次

从顶层的第一个玻璃杯开始倾倒一些香槟，当顶层的杯子满了，任何溢出的香槟

例如，在倾倒一杯香槟后，最顶层的玻璃杯满了。倾倒了两杯香槟后，第二层的

现在当倾倒了非负整数杯香槟后，返回第 i 行 j 个玻璃杯所盛放的香槟占玻璃



示例 1:
输入: poured(倾倒香槟总杯数) = 1, query_glass(杯子的位置数) = 1,
输出: 0.0
解释: 我们在顶层（下标是（0，0））倒了一杯香槟后，没有溢出，因此所有在

示例 2:
输入: poured(倾倒香槟总杯数) = 2, query_glass(杯子的位置数) = 1,
输出: 0.5
解释: 我们在顶层（下标是（0，0）倒了两杯香槟后，有一杯量的香槟将从顶层


注意:

poured 的范围[0, 10 ^ 9]。
query_glass 和query_row 的范围 [0, 99]。
```

前置知识

- 动态规划
- 杨辉三角

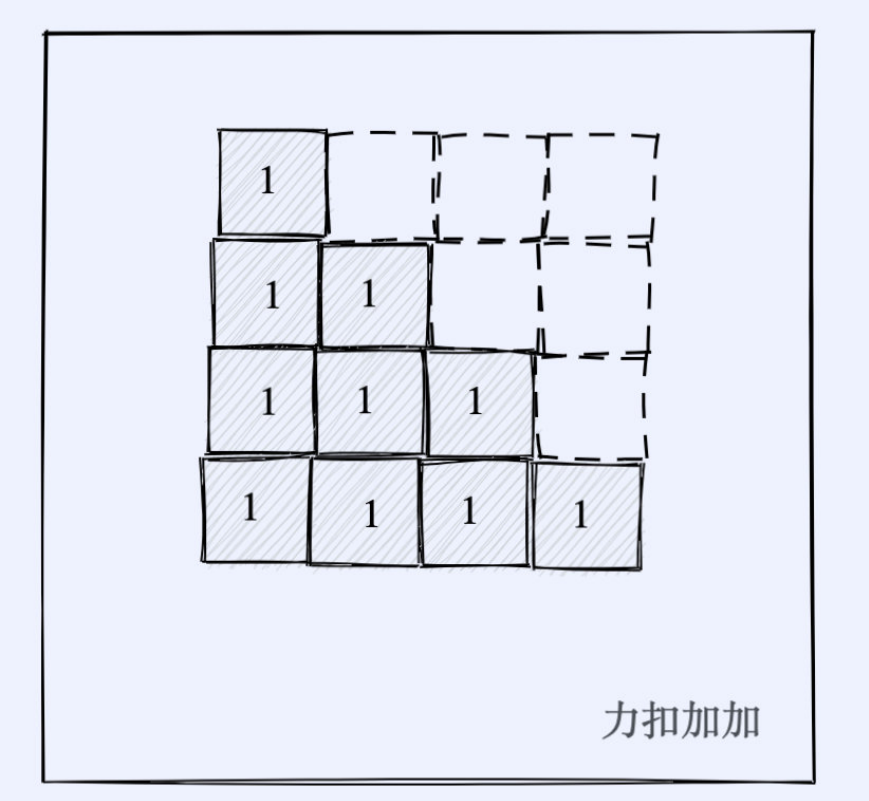
公司

- 暂无

思路

这道题和杨辉三角问题类似，实现的基本思路都是从上到下模拟。如果大家对杨辉三角问题不熟悉，建议先看下杨辉三角。杨辉三角也是动态规划中很经典的问题。

由题目可知杯子的数目是第一行一个，第二行两个。。。第 i 行 i 个 (i >= 1)。因此建立一个二维数组即可。为了简单，我们可以建立一个大小为 R *R 的二维矩阵 A ，其中 R 为香槟塔的高度。虽然这样的建立方式会造成一半的空间浪费。但是题目的条件是* ***query_glass 和 query_row 的范围 [0, 99]****，因此即便如此问题也不大。当然你也可以直接开辟一个 100* 100 的矩阵。



（用 R * R 的二维矩阵 A 进行模拟，如图虚线的部分是没有被使用的空间，也就是”浪费“的空间）

接下来，我们只需要按照题目描述进行模拟即可。具体来说：

- 先将第一行第一列的杯子注满香槟。即 A[0][0] = poured
- 接下来从上到下，从左到右进行模拟。
- 模拟的过程就是
    1. 计算溢出的容量
    2. 将溢出的容量平分到下一层的两个酒杯中。（只需要平分到下一层即可，不用关心下一层满之后的溢出问题，因为之后会考虑，下面的代码也会体现这一点）

关键点

- 不必模拟多步，而是只模拟一次即可

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def champagneTower(self, poured, R, C):
        # 这种初始化方式有一半空间是浪费的
        A = [[0] * (R+1) for _ in range(R+1)]
        A[0][0] = poured
        # 从上到下，从左到右模拟每一行每一列
        for i in range(R + 1):
            for j in range(i+1):
                overflow = (A[i][j] - 1.0) / 2.0
                # 不必模拟多步，而是只模拟一次即可。也就是说我们无
                if overflow > 0 and i < R and j <= C:
                    A[i+1][j] += overflow
                    if j+1<=C: A[i+1][j+1] += overflow

        return min(1, A[R][C]) # 最后的结果如果大于 1，说明流到
```

复杂度分析

- 时间复杂度：$O(R^2)$
- 空间复杂度：$O(R^2)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(801. 使序列递增的最小交换次数)

https://leetcode-cn.com/problems/minimum-swaps-to-make-sequences-increasing/

题目描述

```
我们有两个长度相等且不为空的整型数组 A 和 B 。

我们可以交换 A[i] 和 B[i] 的元素。注意这两个元素在各自的序列中应该处

在交换过一些元素之后，数组 A 和 B 都应该是严格递增的（数组严格递增的

给定数组 A 和 B ，请返回使得两个数组均保持严格递增状态的最小交换次数。

示例:
输入: A = [1,3,5,4], B = [1,2,3,7]
输出: 1
解释:
交换 A[3] 和 B[3] 后，两个数组如下:
A = [1, 3, 5, 7] ， B = [1, 2, 3, 4]
两个数组均为严格递增的。

注意:

A, B 两个数组的长度总是相等的，且长度的范围为 [1, 1000]。
A[i], B[i] 均为 [0, 2000]区间内的整数。
```

前置知识

- 动态规划

公司

- 暂无

思路

要想解决这道题，需要搞定两个关键点。

关键点一：无需考虑全部整体，而只需要考虑相邻两个数字即可

这其实也是可以使用动态规划解决问题的关键条件。对于这道题来说，**最小**的子问题就是当前项和前一项组成的局部，**无法**再小了，**没有必要**再大了。

为什么只关心两个数字即可？因为要使得整个数组递增，**假设**前面的 i - 2 项已经满足递增了，那么现在**采取某种方式**使得满足 A[i] > A[i-1] 即可(B 也是同理)。

> 因为 A[i - 1] > A[i-2] 已经成立，因此如果 A[i] > A[i - 1]，那么整体就递增了。

这提示我们可以使用动态规划来完成。 如果上面的这些没有听懂，则很有可能对动态规划不熟悉，建议先看下基础知识。

关键点二：相邻两个数字的大小关系有哪些?

由于题目一定有解，因此交换相邻项中的**一个或两个**一定能满足**两个数组都递增**的条件。换句话说，如下的情况是不可能存在的：

```
A:[1,2,4]
B:[1,5,1]
```

因为无论怎么交换都无法得到两个递增的序列。那相邻数字的大小关系究竟有哪些呢？实际上大小关系一共有四种。为了描述方便，先列举两个条件，之后直接用 q1 和 q2 来引用这两个关系。

```
q1: A[i-1] < A[i] and B[i-1] < B[i]
q2: A[i-1] < B[i] and B[i-1] < A[i]
```

- q1 表示的是两个数组本身就已经递增了，你**可以选择**不交换。
- q2 表示的是两个数组必须进行一次交换，你**可以选择**交换 i 或者交换 i - 1。

铺垫已经有了，接下来我们来看下这四种关系。

关系一：q1 满足 q2 满足。换不换都行，换 i 或者 i - 1 都行, 也可以都换

关系二：q1 不满足 q2 不满足。无解，对应上面我举的不可能存在的情况

关系三：q1 满足 q2 不满足。换不换都行，但是如果换需要都换。

关系四：q1 不满足 q2 满足 。必须换，换 i 或者 i - 1

接下来按照上面的四种关系进行模拟即可解决。

关键点

- 无需考虑全部整体，而只需要考虑相邻两个数字即可
- 分情况讨论
- 从题目的**一定有解**条件入手

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def minSwap(self, A: List[int], B: List[int]) -> int:
        n = len(A)
        swap = [n] * n
        no_swap = [n] * n
        swap[0] = 1
        no_swap[0] = 0

        for i in range(1, len(A)):
            q1 = A[i-1] < A[i] and B[i-1] < B[i]
            q2 = A[i-1] < B[i] and B[i-1] < A[i]
            if q1 and q2:
                no_swap[i] = min(swap[i-1], no_swap[i-1]) #
                swap[i] = min(swap[i-1], no_swap[i-1]) + 1
            if q1 and not q2:
                swap[i] = swap[i-1] + 1 # 都换
                no_swap[i] = no_swap[i-1] # 都不换
            if not q1 and q2:
                swap[i] = no_swap[i-1] + 1 # 换 i
                no_swap[i] = swap[i-1] # 换 i - 1

        return min(swap[n-1], no_swap[n-1])
```

实际上，我们也可以将逻辑进行合并，这样代码更加简洁。力扣中国题解区很多都是这种写法。即：

```
if q1:
    no_swap[i] = no_swap[i-1] # 都不换
    swap[i] = swap[i-1] + 1 # 都换
if q2:
    swap[i] = min(swap[i], no_swap[i-1] + 1) # 换 i
    no_swap[i] =  min(no_swap[i], swap[i-1]) # 换 i - 1
```

可以看出，这种写法和上面逻辑是一致的。

逻辑合并之后的代码，更简短。但由于两个分支可能都执行到，因此不太容易直接写出。

代码：

```
class Solution:
    def minSwap(self, A: List[int], B: List[int]) -> int:
        n = len(A)
        swap = [n] * n
        no_swap = [n] * n
        swap[0] = 1
        no_swap[0] = 0

        for i in range(1, len(A)):
            # 如果交换之前有序，则可以不交换
            if A[i-1] < A[i] and B[i-1] < B[i]:
                no_swap[i] = no_swap[i-1]
                swap[i] = swap[i-1] + 1
            # 否则至少需要交换一次（交换当前项或者前一项）
            if A[i-1] < B[i] and B[i-1] < A[i]:
                swap[i] = min(swap[i], no_swap[i-1] + 1) #
                no_swap[i] =  min(no_swap[i], swap[i-1]) #

        return min(swap[n-1], no_swap[n-1])
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（816. 模糊坐标）

https://leetcode-cn.com/problems/ambiguous-coordinates

题目描述

```
我们有一些二维坐标，如 "(1, 3)" 或 "(2, 0.5)"，然后我们移除所有逗号

原始的坐标表示法不会存在多余的零，所以不会出现类似于"00", "0.0", "0

最后返回的列表可以是任意顺序的。而且注意返回的两个数字中间（逗号之后）

示例 1:
输入: "(123)"
输出: ["(1, 23)", "(12, 3)", "(1.2, 3)", "(1, 2.3)"]
示例 2:
输入: "(00011)"
输出:  ["(0.001, 1)", "(0, 0.011)"]
解释:
0.0, 00, 0001 或 00.01 是不被允许的。
示例 3:
输入: "(0123)"
输出: ["(0, 123)", "(0, 12.3)", "(0, 1.23)", "(0.1, 23)", "
示例 4:
输入: "(100)"
输出: [(10, 0)]
解释:
1.0 是不被允许的。


提示:

4 <= S.length <= 12.
S[0] = "(", S[S.length - 1] = ")", 且字符串 S 中的其他元素都是
```

前置知识

- 回溯
- 笛卡尔积

公司

- 暂无

思路

这个也是一个明显的笛卡尔积的题目。

我们先将题目简化一下，不考虑题目给的那些限制， 只要将其分割成逗号分割的两部分即可，不用考虑是否是有效的（比如不能是 001 等）。

那么代码大概是：

Python3 Code:

```
class Solution:

    def subset(self, s: str):
        ans = []
        for i in range(1, len(s)):
            ans.append(s[:i] + "." + s[i:])
        ans.append(s)
        return ans

    def ambiguousCoordinates(self, s: str) -> List[str]:
        ans = []
        s = s[1:-1]
        for i in range(1, len(s)):
            x = self.subset(s[:i])
            y = self.subset(s[i:])
            for i in x:
                for j in y:
                    ans.append('(' + i + ', ' + j + ')')
        return ans
```

我简单解释一下上面代码的意思。

- 将字符串分割成两部分， 其所有的可能性无非就是枚举切割点，这里使用了一个 for 循环。
- subset(s) 的功能是在 s 的第 0 位后，第一位后，第 n - 2 位后插入一个小数点 "."，其实就是构造一个有效的数字而已。
- 因此 x 和 y 就是分割形成的两部分的有效分割集合，**答案自然就是 x 和 y 的笛卡尔积**。

如果上面的代码你会了，这道题无非就是增加几个约束， 我们剪几个不合法的枝即可。具体代码见下方代码区，可以看出，代码仅仅是多了几个 if 判断而已。

上面的目标很常见，请**务必掌握**。

关键点

- 笛卡尔积优化

代码

代码支持：Python3

```
class Solution:
    # "123" => ["1.23", "12.3", "123"]
    def subset(self, s: str):
        ans = []

        #  带小数点的
        for i in range(1, len(s)):
            # 不允许 00.111,  0.0, 01.1, 1.0
            if s[0] == '0' and i > 1:
                continue
            if s[-1] == '0':
                continue
            ans.append(s[:i] + "." + s[i:])
        # 不带小数点的（不允许 001）
        if s == '0' or not s.startswith('0'):
            ans.append(s)
        return ans

    def ambiguousCoordinates(self, s: str) -> List[str]:
        ans = []
        s = s[1:-1]
        for i in range(1, len(s)):
            x = self.subset(s[:i])
            y = self.subset(s[i:])
            for i in x:
                for j in y:
                    ans.append('(' + i + ', ' + j + ')')
        return ans
```

复杂度分析

- 时间复杂度：$O(N^3)$
- 空间复杂度：$O(N^2)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（820. 单词的压缩编码）

https://leetcode-cn.com/problems/short-encoding-of-words/

题目描述

```
给定一个单词列表，我们将这个列表编码成一个索引字符串 S 与一个索引列表

例如，如果这个列表是 ["time", "me", "bell"]，我们就可以将其表示为

对于每一个索引，我们可以通过从字符串 S 中索引的位置开始读取字符串，直到

那么成功对给定单词列表进行编码的最小字符串长度是多少呢?



示例:

输入: words = ["time", "me", "bell"]
输出: 10
说明: S = "time#bell#" ,  indexes = [0, 2, 5] 。


提示:

1 <= words.length <= 2000
1 <= words[i].length <= 7
每个单词都是小写字母 。
```

前置知识

- 前缀树

公司

- 阿里
- 字节

思路

读完题目之后就发现如果将列表中每一个单词分别倒序就是一个后缀树问题。比如 `["time", "me", "bell"]` 倒序之后就是 ["emit", "em", "lleb"]，我们要求的结果无非就是 "emit" 的长度 + "llem"的长度 + "##"的长度（em 和 emit 有公共前缀，计算一个就好了）。

因此符合直觉的想法是使用前缀树 + 倒序插入的形式来模拟后缀树。

下面的代码看起来复杂，但是很多题目我都是用这个模板，稍微调整下细节就能 AC。我这里总结了一套前缀树专题

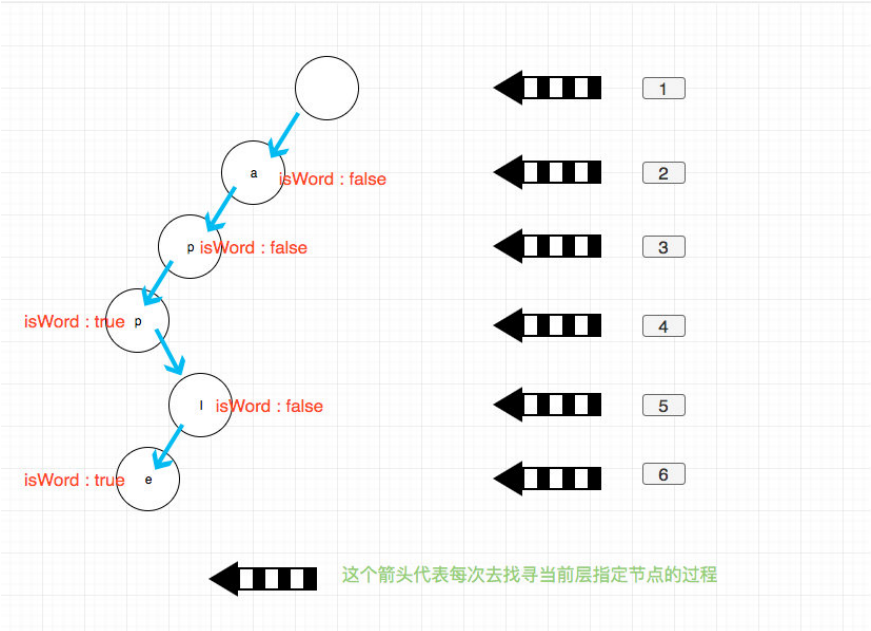
- 208.implement-trie-prefix-tree
- 211.add-and-search-word-data-structure-design
- 212.word-search-ii
- 472.concatenated-words

前缀树的 api 主要有以下几个：

- `insert(word)`：插入一个单词
- `search(word)`：查找一个单词是否存在
- `startWith(word)`：查找是否存在以 word 为前缀的单词

其中 startWith 是前缀树最核心的用法，其名称前缀树就从这里而来。大家可以先拿 208 题开始，熟悉一下前缀树，然后再尝试别的题目。

一个前缀树大概是这个样子：



如图每一个节点存储一个字符，然后外加一个控制信息表示是否是单词结尾，实际使用过程可能会有细微差别，不过变化不大。

这道题需要考虑 edge case， 比如这个列表是 ["time", "time", "me", "bell"] 这种包含重复元素的情况，这里我使用 hashset 来去重。

关键点

- 前缀树
- 去重

代码

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for w in word:
            curr = curr[w]
        # len(curr) == 1 means we meet '#'
        # when we search 'em'(which reversed from 'me')
        # the result is len(curr) > 1
        # cause the curr look like { '#': 1, i: {...}}
        return len(curr) == 1
class Solution:
    def minimumLengthEncoding(self, words: List[str]) -> in
        trie = Trie()
        cnt = 0
        words = set(words)
        for word in words:
            trie.insert(word[::-1])
        for word in words:
            if trie.search(word[::-1]):
                cnt += len(word) + 1
        return cnt
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为单词长度列表中的总字符数，比如 ["time", "me"]，就是 4 + 2 = 6。
- 空间复杂度：$O(N)$，其中 N 为单词长度列表中的总字符数，比如 ["time", "me"]，就是 4 + 2 = 6。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址(873. 最长的斐波那契子序列的长度)

https://leetcode-cn.com/problems/length-of-longest-fibonacci-subsequence/

题目描述

```
如果序列 X_1, X_2, ..., X_n 满足下列条件，就说它是 斐波那契式 的：

n >= 3
对于所有 i + 2 <= n，都有 X_i + X_{i+1} = X_{i+2}

给定一个严格递增的正整数数组形成序列，找到 A 中最长的斐波那契式的子序列

（回想一下，子序列是从原序列 A 中派生出来的，它从 A 中删掉任意数量的元



示例 1：

输入: [1,2,3,4,5,6,7,8]
输出: 5
解释:
最长的斐波那契式子序列为：[1,2,3,5,8] 。


示例 2：

输入: [1,3,7,11,12,14,18]
输出: 3
解释:
最长的斐波那契式子序列有：
[1,11,12]，[3,11,14] 以及 [7,11,18] 。




提示：

3 <= A.length <= 1000
1 <= A[0] < A[1] < ... < A[A.length - 1] <= 10^9
（对于以 Java，C，C++，以及 C# 的提交，时间限制被减少了 50%）
```

前置知识

- 动态规划

公司

- 暂无

思路

和一般的 DP 不同，这道题是已知状态转移方程。所以我勉强也归类到 DP 吧。

这道题的思路是两两枚举数组中的数字，不妨称其为 a 和 b。接下来，我们以 a 和 b 为斐波那契的起点， 很明显斐波那契数列的下一个数字应该是 a + b，这是题目给出的信息。

- 如果 a + b 不在数组中，直接终止，继续枚举下一个。
- 如果 a + b 在数组中，说明我们找到了一个长度为 3 的斐波那契子数列。那么继续尝试扩展斐波那契数列长度到 4。。。

上面的枚举需要 $O(n^2)$的时间复杂度，枚举过程记录最大长度并返回即可。

对于每次枚举，我们都需要不断重复检查 a + b 是否在数组中，直到不再数组中为止。因此最坏的情况是一直在数组中，这个时间复杂度大概是数组中最大值和最小值的差值的对数。用公式表示就是 $log(m1 - m2)$，其中 m1 为数组 最大值， m2 为数组最小值。

关键点

- 使用集合存储数组中的所有数，然后枚举数组中的两两组合并，去集合中不断延伸斐波那契数列

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def lenLongestFibSubseq(self, A: List[int]) -> int:
        s = set(A)
        ans = 0
        for i in range(len(A)):
            for j in range(i + 1, len(A)):
                a, b = A[j], A[i] + A[j]
                t = 2
                while b in s:
                    a, b = b, a + b
                    t += 1
                ans = max(ans, t)
        return 0 if ans < 3 else ans
```

复杂度分析

令 n 为数组长度, m1 为数组最大值，m2 为数组最小值。

- 时间复杂度：$O(n^2log(m1-m2))$
- 空间复杂度：$O(n)$

扩展

这道题还有时间复杂度更好的做法， 感兴趣的可以参考 力扣官方题解

结尾

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（875. 爱吃香蕉的珂珂）

https://leetcode-cn.com/problems/koko-eating-bananas/description/

题目描述

```
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

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

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
```

前置知识

- 二分查找

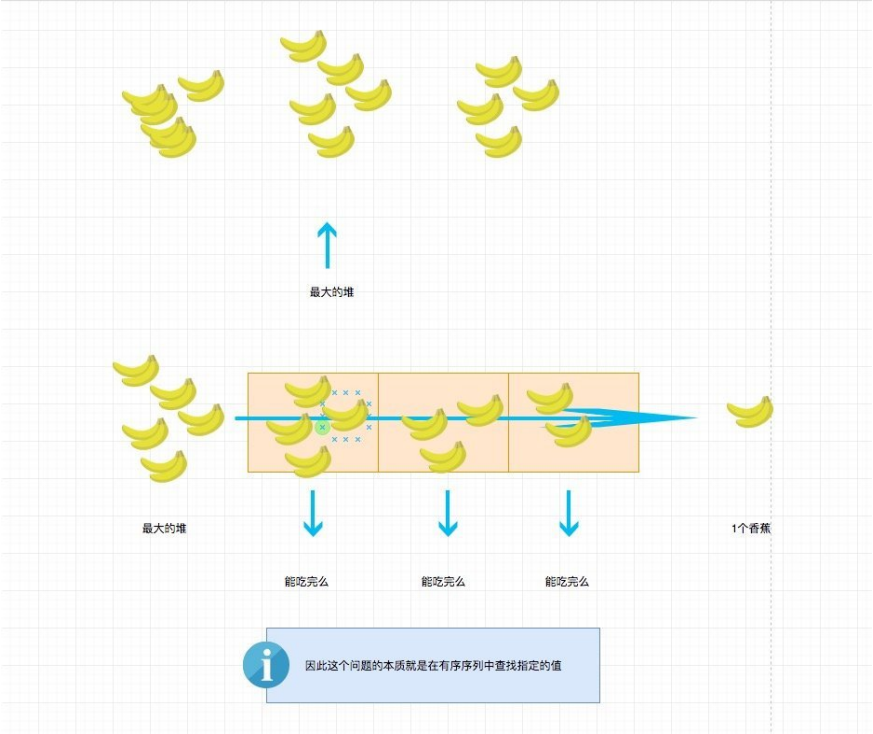
公司

- 字节

思路

符合直觉的做法是，选择最大的堆的香蕉数，然后试一下能不能行，如果不行则直接返回上次计算的结果，如果行，我们减少 1 个香蕉，试试行不行，依次类推。计算出刚好不行的即可。这种解法的时间复杂度比较高，为 $O(N * M)$，其中 N 为 piles 长度， M 为 Piles 中最大的数。。

这道题如果能看出来是二分法解决，那么其实很简单。为什么它是二分问题呢？我这里画了个图，我相信你看了就明白了。



> 香蕉堆的香蕉个数上限是 10^9， 珂珂这也太能吃了吧?

关键点解析

- 二分查找模板

代码

代码支持：Python，JavaScript

Python Code:

```python
class Solution:
    def solve(self, piles, k):
        def possible(mid):
            t = 0
            for pile in piles:
                t += (pile + mid - 1) // mid
            return t <= k

        l, r = 1, max(piles)

        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l
```

JavaScript Code:

```
function canEatAllBananas(piles, H, mid) {
  let h = 0;
  for (let pile of piles) {
    h += Math.ceil(pile / mid);
  }

  return h <= H;
}
/**
 * @param {number[]} piles
 * @param {number} H
 * @return {number}
 */
var minEatingSpeed = function (piles, H) {
  let lo = 1,
    hi = Math.max(...piles);
  // [l, r) ,  左闭右开的好处是如果能找到，那么返回 l 和 r 都是一样
  while (lo <= hi) {
    let mid = lo + ((hi - lo) >> 1);
    if (canEatAllBananas(piles, H, mid)) {
      hi = mid - 1;
    } else {
      lo = mid + 1;
    }
  }

  return lo; //  不能选择hi
};
```

复杂度分析

- 时间复杂度：$O(max(N, N * logM))$，其中 N 为 piles 长度， M 为 Piles 中最大的数。
- 空间复杂度：$O(1)$

模板

分享几个常用的的二分法模板。

查找一个数

```java
public int binarySearch(int[] nums, int target) {
    // 左右都闭合的区间 [l, r]
    int left = 0;
    int right = nums.length - 1;

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        else if (nums[mid] > target)
            // 搜索区间变为 [left, mid - 1]
            right = mid - 1;
    }
    return -1;
}
```

寻找最左边的满足条件的值

```java
public int binarySearchLeft(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右边界
            right = mid - 1;
        }
    }
    // 检查是否越界
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}
```

寻找最右边的满足条件的值

```java
public int binarySearchRight(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩左边界
            left = mid + 1;
        }
    }
    // 检查是否越界
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

> 如果题目重点不是二分，也就是说二分只是众多步骤中的一步，大家也可以直接调用语言的 API，比如 Python 的 bisect 模块。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(877. 石子游戏)

https://leetcode-cn.com/problems/stone-game/

题目描述

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

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

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

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 tru
```

提示：

```
2 <= piles.length <= 500
piles.length 是偶数。
1 <= piles[i] <= 500
sum(piles) 是奇数。
```

前置知识

- 动态规划

公司

- 阿里
- 字节

思路

由于 piles 是偶数的，并且 piles 的总和是奇数的。

因此 Alex `可以做到` 要不拿的全部是奇数，要么全部是偶数。

举个例子： 比如 Alex 第一次先拿第一个

这里有两种情况：

1. Lee 如果拿了第二块（偶数），那么 Alex 继续拿第三块，以此类推。。。
2. Lee 如果拿了最后一块（偶数），那么 Alex 继续拿倒数第二块，以此类推。。。

因此 Alex `可以` 做到只拿奇数或者偶数，只是他可以控制的，因此他要做的就是数一下，奇数加起来多还是偶数加起来多就好了。 奇数多就全部选奇数，偶数就全部选偶数。 Lee 是没有这种自由权的。

关键点解析

- 可以用 DP（动态规划）
- 可以从数学的角度去分析

代码

```
/**
 * @param {number[]} piles
 * @return {boolean}
 */
var stoneGame = function(piles) {
  return true;
};
```

扩展

腾讯面试题：一共 100 只弓箭 你和你的对手共用。你们每次只能射出一支箭或者两支箭，射击交替进行，设计一个算法，保证自己获胜。

答案： 先手，剩下的是 3 的倍数就行（100-1=99），然后按照 3 的倍数射箭必赢。 比如你先拿了 1，剩下 99 个。 对手拿了 1，你就拿 2。这样持续 33 次就赢了。如果对手拿了 2 个，你就拿 1 个，这样持续 33 次你也是赢的。

> 这是一种典型的博弈问题， 你和对手交替进行，对手的行动影响你接下来的策略。 这算是一种最简单的博弈问题了

题目地址（886. 可能的二分法）

https://leetcode-cn.com/problems/is-graph-bipartite/

题目描述

```
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

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

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
```

前置知识

- 图的遍历
- DFS

公司

- 暂无

思路

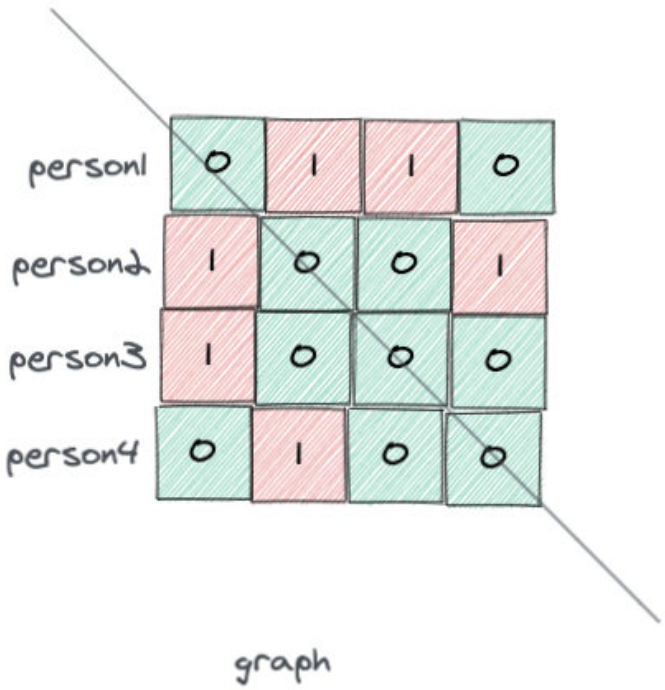
这是一个图的问题。解决这种问题一般是要遍历图才行的，这也是图的套路。 那么遍历的话，你要有一个合适的数据结构。 比较常见的图存储方式是邻接矩阵和邻接表。

而我们这里为了操作方便（代码量），直接使用邻接矩阵。由于是互相不喜欢，不存在一个喜欢另一个，另一个不喜欢一个的情况，因此这是无向图。而无向图邻接矩阵实际上是会浪费空间，具体看我下方画的图。

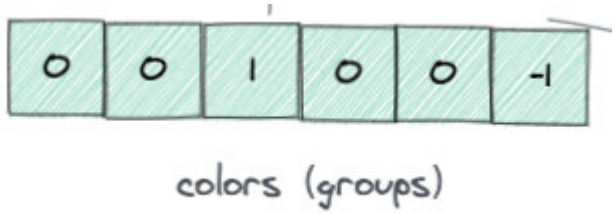
而题目给我们的二维矩阵并不是现成的邻接矩阵形式，因此我们需要自己生成。

我们用 1 表示互相不喜欢（dislike each other）。

```
        graph = [[0] * N for i in range(N)]
        for a, b in dislikes:
            graph[a - 1][b - 1] = 1
            graph[b - 1][a - 1] = 1
```



同时可以用 hashmap 或者数组存储 N 个人的分组情况， 业界关于这种算法一般叫染色法，因此我们命名为 colors，其实对应的本题叫 groups 更合适。



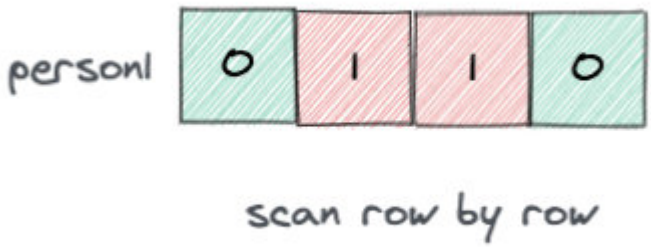
我们用：

- 0 表示没有分组
- 1 表示分组 1
- -1 表示分组 2

之所以用 0，1，-1，而不是 0，1，2 是因为我们会在不能分配某一组的时候尝试分另外一组，这个时候有其中一组转变为另外一组就可以直接乘以-1，而 0，1，2 这种就稍微麻烦一点而已。

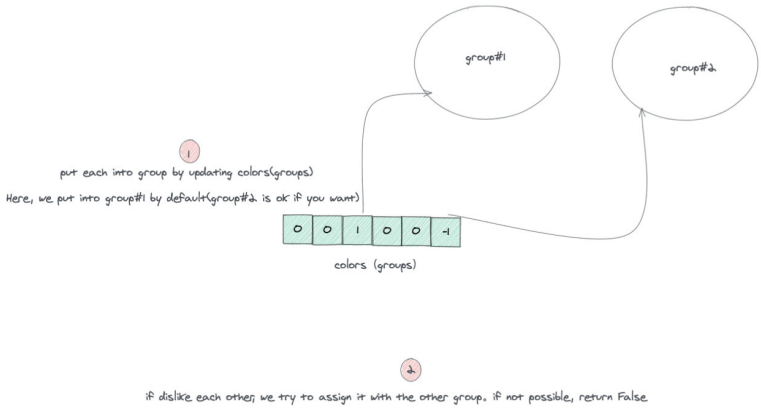
具体算法：

- 遍历每一个人，尝试给他们进行分组，比如默认分配组 1.



- 然后遍历这个人讨厌的人，尝试给他们分另外一组，如果不可以分配另外一组，则返回 False

那问题的关键在于如何判断“不可以分配另外一组”呢?



实际上，我们已经用 colors 记录了分组信息，对于每一个人如果分组确定了，我们就更新 colors，那么对于一个人如果分配了一个组，并且他讨厌的人也被分组之后，**分配的组和它只能是一组**，那么“就是不可以分配另外一组”。

代码表示就是：

```
# 其中j 表示当前是第几个人，N表示总人数。 dfs的功能就是根据colors和
if colors[j] == 0 and not self.dfs(graph, colors, j, -1 * 
```

关键点

- 二分图
- 染色法
- 图的建立和遍历
- colors 数组

代码

```
class Solution:
    def dfs(self, graph, colors, i, color, N):
        colors[i] = color
        for j in range(N):
            # dislike eachother
            if graph[i][j] == 1:
                if colors[j] == color:
                    return False
                if colors[j] == 0 and not self.dfs(graph, c
                    return False
        return True

    def possibleBipartition(self, N: int, dislikes: List[Li
        graph = [[0] * N for i in range(N)]
        colors = [0] * N
        for a, b in dislikes:
            graph[a - 1][b - 1] = 1
            graph[b - 1][a - 1] = 1
        for i in range(N):
            if colors[i] == 0 and not self.dfs(graph, colo
                return False
        return True
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

相关问题

- 785. 判断二分图

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(898. 子数组按位或操作)

https://leetcode-cn.com/problems/bitwise-ors-of-subarrays/

题目描述

我们有一个非负整数数组 A。

对于每个（连续的）子数组 B = [A[i], A[i+1], ..., A[j]] （ i <=

返回可能结果的数量。 （多次出现的结果在最终答案中仅计算一次。）

示例 1:

输入: [0]
输出: 1
解释:
只有一个可能的结果 0 。

示例 2:

输入: [1,1,2]
输出: 3
解释:
可能的子数组为 [1]，[1]，[2]，[1, 1]，[1, 2]，[1, 1, 2]。
产生的结果为 1，1，2，1，3，3 。
有三个唯一值，所以答案是 3 。

示例 3:

输入: [1,2,4]
输出: 6
解释:
可能的结果是 1，2，3，4，6，以及 7 。

提示:

1 <= A.length <= 50000
0 <= A[i] <= 10^9

前置知识

- 【西法带你学算法】一次搞定前缀和

公司

- 暂无

思路

我们首先需要对问题进行分解，分解的思路和 【西法带你学算法】一次搞定前缀和 中提到的一样。这里简单介绍一下，如果还不明白的，建议看下那篇文章。

题目需要求的是所有子数组或运算后的结果的数目（去重）。一个朴素的思路是求出所有的子数组，然后对其求或，然后放到 hashset 中去重，最后返回 hashset 的大小即可。

我们可以使用固定两个端点的方式在 $O(n^2)$ 的时间计算出所有的子数组，并在 $O(n)$ 的时间求或，因此这种朴素的算法的时间复杂度是 $O(n^2 + n)$。

要点 1

而由于子数组具有连续性，也就是说如果子数组 A[i:j] 的或是 OR(i,j)。那么子数组 A[i:j+1] 的或就是 OR(i,j) | A[j+1]，也就是说，我们**无需重复计算 OR(i, j)**。基于这种思路，我们可以写出 $O(n)$ 的代码。

要点 2

所有的子数组其实就是:

- 以索引为 0 结束的子数组
- 以索引为 1 结束的子数组
- 。。。

因此，我们可以边遍历边计算，并使用上面提到的技巧，用之前计算的结果推导下一步的结果。

算法（假设当前遍历到了索引 i）：

- 用 pres 记录上一步的子数组异或值集合，也就是**以索引 i - 1 结尾的子数组异或值集合**
- 遍历 pres，使用 pres 中的每一项和当前数进行或运算，并将结果重新放入 pres。最后别忘了把自身也放进去。

> 为了防止迭代 pres 过程改变 pres 的值，我们可以用另外一个中间临时集合承载结果。

- 将 pres 中的所有数加入 ans，其中 ans 为我们要返回的一个 hashset

关键点

- 子数组是连续的，有很多性质可以利用

代码

- 语言支持：Python3

Python3 Code:

```
class Solution(object):
    def subarrayBitwiseORs(self, A):
        pres = set([0])
        ans = set()
        for a in A:
            nxt = set()
            for pre in pres:
                nxt.add(a | pre)
                nxt.add(a)
            pres = nxt
            ans |= nxt
        return len(ans)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(900. RLE 迭代器)

https://leetcode-cn.com/problems/rle-iterator/

题目描述

编写一个遍历游程编码序列的迭代器。

迭代器由 RLEIterator(int[] A) 初始化，其中 A 是某个序列的游程编码

迭代器支持一个函数：next(int n)，它耗尽接下来的 n 个元素（n >= 1）

例如，我们以 A = [3,8,0,9,2,5] 开始，这是序列 [8,8,8,5,5] 的游程

示例：

输入：["RLEIterator","next","next","next","next"], [[[3,8,0,
输出：[null,8,8,5,-1]
解释：
RLEIterator 由 RLEIterator([3,8,0,9,2,5]) 初始化。
这映射到序列 [8,8,8,5,5]。
然后调用 RLEIterator.next 4次。

.next(2) 耗去序列的 2 个项，返回 8。现在剩下的序列是 [8, 5, 5]。

.next(1) 耗去序列的 1 个项，返回 8。现在剩下的序列是 [5, 5]。

.next(1) 耗去序列的 1 个项，返回 5。现在剩下的序列是 [5]。

.next(2) 耗去序列的 2 个项，返回 -1。 这是由于第一个被耗去的项是 5，
但第二个项并不存在。由于最后一个要耗去的项不存在，我们返回 -1。

提示：

0 <= A.length <= 1000
A.length 是偶数。
0 <= A[i] <= 10^9
每个测试用例最多调用 1000 次 RLEIterator.next(int n)。
每次调用 RLEIterator.next(int n) 都有 1 <= n <= 10^9 。

前置知识

- 哈夫曼编码和游程编码

公司

- 暂无

思路

这是一个游程编码的典型题目。

该算法分为两个部分，一个是初始化，一个是调用 `next(n)` .

我们需要做的就是初始化的时候，记住这个A。 然后每次调用 `next(n)` 的时候只需要

判断n是否大于Ai

- 如果大于A[i], 那就说明不够，我们移除数组前两项，更新n，重复1
- 如果小于A[i], 则说明够了，更新A[i]

这样做，我们每次都要更新A，还有一种做法就是不更新A，而是 `伪更新` ，即用一个变量记录，当前访问到的数组位置。

> 很多时候我们需要原始的，那么就必须这种放了，我的解法就是这种方法。

关键点解析

代码

```
/**
 * @param {number[]} A
 */
var RLEIterator = function(A) {
    this.A = A;
    this.current = 0;
};


/**
 * @param {number} n
 * @return {number}
 */
RLEIterator.prototype.next = function(n) {
    const A = this.A;
    while(this.current < A.length && A[this.current] < n){
        n = n - A[this.current];
        this.current += 2;
    }

    if(this.current >= A.length){
        return -1;
    }

    A[this.current] = A[this.current] - n; // 更新Count
    return A[this.current + 1]; // 返回element
};

/**
 * Your RLEIterator object will be instantiated and called
 * var obj = new RLEIterator(A)
 * var param_1 = obj.next(n)
 */
```

扩展阅读

哈夫曼编码和游程编码

题目地址(911. 在线选举)

https://leetcode-cn.com/problems/online-election/

题目描述

```
在选举中，第 i 张票是在时间为 times[i] 时投给 persons[i] 的。

现在，我们想要实现下面的查询函数： TopVotedCandidate.q(int t) 将

在 t 时刻投出的选票也将被计入我们的查询之中。在平局的情况下，最近获得

示例：

输入：["TopVotedCandidate","q","q","q","q","q","q"], [[[0,1,
输出：[null,0,1,1,0,0,1]
解释：
时间为 3，票数分布情况是 [0]，编号为 0 的候选人领先。
时间为 12，票数分布情况是 [0,1,1]，编号为 1 的候选人领先。
时间为 25，票数分布情况是 [0,1,1,0,0,1]，编号为 1 的候选人领先（因
在时间 15、24 和 8 处继续执行 3 个查询。


提示：

1 <= persons.length = times.length <= 5000
0 <= persons[i] <= persons.length
times 是严格递增的数组，所有元素都在 [0, 10^9] 范围中。
每个测试用例最多调用 10000 次 TopVotedCandidate.q。
TopVotedCandidate.q(int t) 被调用时总是满足 t >= times[0]。
```

前置知识

- 二分查找
- 哈希表

公司

- 暂无

思路

题目给了一个 times 数组， 我们可以记录 times 中每一时刻 t 的优胜者，只需要边遍历边统计票数，用两个全局参数 max_voted_person 和 max_voted_count 分别表示当前票数最多的人和其对应的票数即可。

由于题目要求如果票数相同取最近的，那么只需要在更新 max_voted_person 和 max_voted_count 的时候，增加**如果当前人票数和 max_voted_count 一致也更新 max_voted_person 和 max_voted_count**逻辑即可轻松实现。

由于题目没有说 person[i] 是 [0, N) 区间的值，使用数组统计不方便，因此这里我使用哈希表进行统计。

核心代码：

```
class TopVotedCandidate:

    def __init__(self, persons: List[int], times: List[int]
        vote_count = collections.defaultdict(int) # 哈希表统
        max_voted_person = -1
        max_voted_count = 0
        winner = []
        # zip([1,2,3], [4,5,6]) 会返回 [[1,4], [2,5], [3,6]]
        for p, t in zip(persons, times):
            vote_count[p] += 1
            if vote_count[p] >= max_voted_count:
                max_voted_count = vote_count[p]
                max_voted_person = p
            # 更新 winner
            winner.append(max_voted_person)
```

经过上面的处理生成了一个 winner 数组，winner 数组和 times 以及 persons 是等长的。

接下来就是查询了，查询的 api 如下：

```
q(int t) -> int
```

我们要做的就是使用 t 去前面生成好的 winner 数组找。由于 times 是有序的，因此查询过程我们就可以使用二分了。

比如:

```
times =  [2,4,5,6]
winner = [1,2,1,1]
```

表示的就是：

- 2，5，6 时刻的优胜者是 1
- 4 时刻优胜者是 2

如果 t 为 2， 4， 5， 6 我们直接返回 winner 对应的项目即可。比如 t 为 2，2 在 times 中 第 0 项，因此返回 winner[0]即可。

如果 t 为 3 呢？3 不在 [2,4,5,6] 中。根据题目要求，我们需要以 3 的最近的一个往前的时间点，也就是 2 ，我们仍然需要返回 winner[0]。

总的来说，其实我们需要找的位置就是一个最左插入位置，即将 t 插入 times 之后仍然保持有序的位置。比如 t 为 3 就是 [2,3,4,5,6]，我们需要返回 3 的前一个。关于最左插入我在二分查找 进行了详细的描述，不懂的可以看下。

关键点解析

- 使用哈希表记录 times 中每一个时刻的优胜信息
- 最左插入模板

代码

代码支持： Python3

```
class TopVotedCandidate:

    def __init__(self, persons: List[int], times: List[int]
        vote_count = collections.defaultdict(int)
        max_voted_person = -1
        max_voted_count = 0
        winner = []
        for p, t in zip(persons, times):
            vote_count[p] += 1
            if vote_count[p] >= max_voted_count:
                max_voted_count = vote_count[p]
                max_voted_person = p
            winner.append(max_voted_person)
        self.winner = winner
        self.times = times

    def q(self, t: int) -> int:
        winner = self.winner
        # times 是不重复的，也就是严格递增的，类似 [2,4,5,6]，这是
        # eg:
        # times  [2,4,5,6]
        # winner [1,2,1,1]
        i = bisect.bisect_left(self.times, t)
        if i != len(self.times) and self.times[i] == t:
            return winner[i]
        return winner[i - 1]
```

复杂度分析

- 时间复杂度：初始化的时间复杂度为 $O(N)$，q 的复杂度为 $O(logN)$，其中 N 为数组长度。
- 空间复杂度：我们使用了 vote_count 记录投票情况，因此空间复杂度 $O(N)$，其中 N 为数组长度。

题目地址(912. 排序数组)

https://leetcode-cn.com/problems/sort-an-array/

题目描述

```
给你一个整数数组 nums，请你将该数组升序排列。



示例 1：

输入：nums = [5,2,3,1]
输出：[1,2,3,5]
示例 2：

输入：nums = [5,1,1,2,0,0]
输出：[0,0,1,1,2,5]


提示：

1 <= nums.length <= 50000
-50000 <= nums[i] <= 50000
```

前置知识

- 数组
- 排序

公司

- 阿里
- 百度
- 字节

思路

这是一个很少见的直接考察 `排序` 的题目。 其他题目一般都是暗含 `排序` ，这道题则简单粗暴，直接让你排序。 并且这道题目的难度是 `Medium` ， 笔者感觉有点不可思议。

我们先来看题目的限制条件，这其实在选择算法的过程中是重要的。 看到这道题的时候，大脑就闪现出了各种排序算法， 这也算是一个复习 `排序算法` 的机会吧。

题目的限制条件是有两个，第一是元素个数不超过10k，这个不算大。 另外一个是数组中的每一项范围都是 `-50k` 到 `50k` （包含左右区间）。 看到这里，基本我就排除了时间复杂度为O(n^2)的算法。

> 我没有试时间复杂度 O(n^2) 的解法，大家可以试一下，看是不是会TLE。

剩下的就是基于比较的 `nlogn` 算法，以及基于特定条件的O(n)算法。

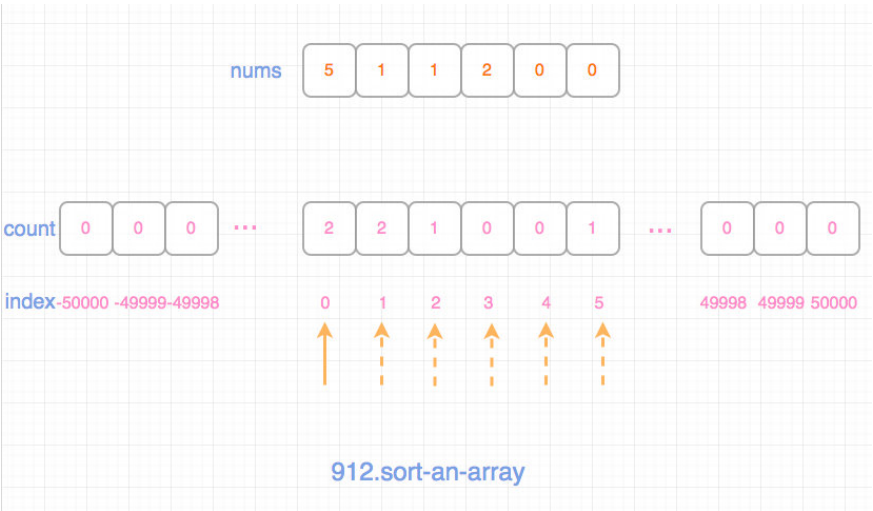
由于平时很少用到 `计数排序` 等O(n)的排序算法，一方面是空间复杂度不是常量，另一方面是其要求数据范围不是很大才行，不然会浪费很多空间。 但是这道题我感觉可以试一下。 在这里，我用了两种方法，一种是 `计数排序` ，一种是 `快速排序` 来解决。 大家也可以尝试用别的解法来解决。

解法一 - 计数排序

时间复杂度O(n)空间复杂度O(m) m 为数组中值的取值范围，在这道题就是 `50000 * 2 + 1` 。

我们只需要准备一个数组取值范围的数字，然后遍历一遍，将每一个元素放到这个数组对应位置就好了， 放的规则是 `索引为数字的值，value为出现的次数` 。

这样一次遍历，我们统计出了所有的数字出现的位置和次数。 我们再来一次遍历，将其输出到即可。



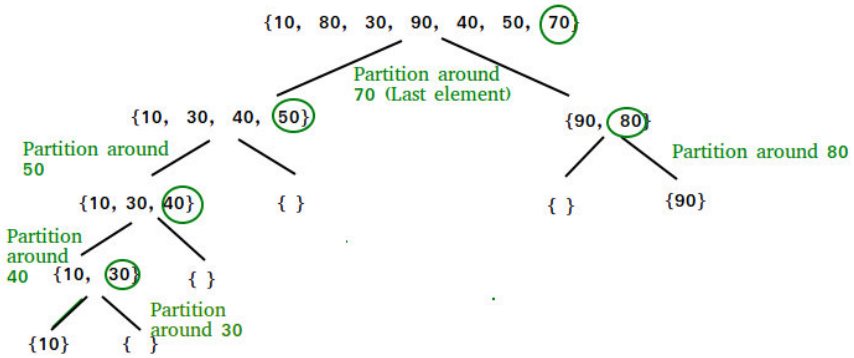
解法二 - 快速排序

快速排序和归并排序都是分支思想来进行排序的算法， 并且二者都非常流行。 快速排序的核心点在于选择轴元素。

每次我们将数组分成两部分，一部分是比pivot（轴元素）大的，另一部分是不比pivot大的。 我们不断重复这个过程， 直到问题的规模缩小的寻常（即只有一个元素的情况）。

快排的核心点在于如何选择轴元素，一般而言，选择轴元素有三种策略：

- 数组最左边的元素
- 数组最右边的元素
- 数组中间的元素（我采用的是这种，大家可以尝试下别的）
- 数组随机一项元素



(图片来自： https://www.geeksforgeeks.org/quick-sort/)

> 图片中的轴元素是最后面的元素，而提供的解法是中间元素，这点需要注意，但是这并不影响理解。

关键点解析

- 排序算法
- 注意题目的限制条件从而选择合适的算法

代码

计数排序：

代码支持： JavaScript

```
/**
 * @param {number[]} nums
 * @return {number[]}
 */
var sortArray = function(nums) {
    const counts = Array(50000 * 2 + 1).fill(0);
    const res = [];
    for(const num of nums) counts[50000 + num] += 1;
    for(let i in counts)  {
        while(counts[i]--) {
            res.push(i - 50000)
        }
    }
    return res;
};
```

快速排序：

代码支持： JavaScript

```
function swap(nums, a, b) {
    const temp = nums[a];
    nums[a] = nums[b];
    nums[b] = temp;
}

function helper(nums, start, end) {
    if (start >= end) return;
    const pivotIndex = start + ((end - start) >>> 1)
    const pivot = nums[pivotIndex]
    let i = start;
    let j = end;
    while (i <= j) {
        while (nums[i] < pivot) i++;
        while (nums[j] > pivot) j--;
        if (i <= j) {
            swap(nums, i, j);
            i++;
            j--;
        }
    }
    helper(nums, start, j);
    helper(nums, i, end);
}

/**
 * @param {number[]} nums
 * @return {number[]}
 */
var sortArray = function(nums) {
    helper(nums, 0, nums.length - 1);
    return nums;
};
```

扩展

- 你是否可以用其他方式排序算法解决

参考

- QuickSort - geeksforgeeks

题目地址(932. 漂亮数组)

https://leetcode-cn.com/problems/beautiful-array/

题目描述

对于某些固定的 N，如果数组 A 是整数 1, 2, ..., N 组成的排列，使得：

对于每个 i < j，都不存在 k 满足 i < k < j 使得 A[k] * 2 = A[i]

那么数组 A 是漂亮数组。

给定 N，返回任意漂亮数组 A（保证存在一个）。

示例 1：

输入：4
输出：[2,1,4,3]

示例 2：

输入：5
输出：[3,1,2,5,4]

提示：

1 <= N <= 1000

前置知识

- 分治

公司

- 暂无

思路

由数字的奇偶特性，可知：**奇数 + 偶数 = 奇数** 。

因此如果要使得：**对于每个 i < j，都不存在 k 满足 i < k < j 使得 A[k] * 2 = A[i] + A[j]** 成立，我们可以令 A[i] 和 A[j] 一个为奇数，另一个为偶数即可。

另外还有两个非常重要的性质，也是本题的突破口。那就是：

性质 1： 如果数组 A 是 漂亮数组，那么将 A 中的每一个数 x 进行 kx + b 的映射，其仍然为漂亮数组。其中 k 为不等于 0 的整数， b 为整数。 性质 2：如果数组 A 和 B 分别是不同奇偶性的漂亮数组，那么将 A 和 B 拼接起来仍为漂亮数组。

举个例子。我们要求长度为 N 的漂亮数组。那么一定是有 N / 2 个偶数 和 N - N / 2 个奇数。

> 这里的除法为地板除。

假设长度为 N / 2 和 N - N/2 的漂亮数组被计算出来了。那么我们只需要对长度为 N/2 的漂亮数组通过性质 1 变换成全部为偶数的漂亮数组，并将长度为 N - N/2 的漂亮数组也通过性质 1 变换成全部为奇数的漂亮数组。接下来利用性质 2 将其进行拼接即可得到一个漂亮数组。

刚才我们**假设长度为 N / 2 和 N - N/2 的漂亮数组被计算出来了**，实际上我们并没有计算出来，那么其实可以用同样的方法来计算。其实就是分治，将问题规模缩小了，问题本质不变。递归的终点自然是 N == 1，此时可直接返回 [1]。

关键点

- 利用性质**奇数 + 偶数 = 奇数**
- 对问题进行分解

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def beautifulArray(self, N: int) -> List[int]:
        @lru_cache(None)
        def dp(n):
            if n == 1:
                return [1]
            ans = []
            # [1,n] 中奇数比偶数多1或一样
            for a in dp(n - n // 2):
                ans += [a * 2 - 1]
            for b in dp(n // 2):
                ans += [b * 2]
            return ans

        return dp(N)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(n + logn)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

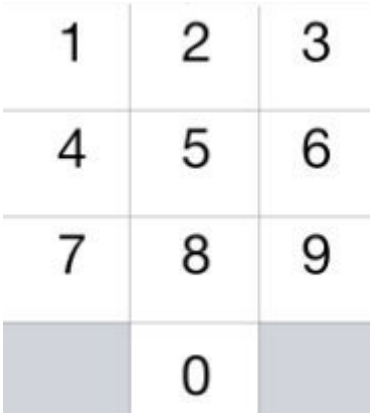
关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址 (935. 骑士拨号器）

https://leetcode-cn.com/problems/knight-dialer/

题目描述

国际象棋中的骑士可以按下图所示进行移动：



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

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

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

前置知识

- DFS
- 记忆化搜索

公司

- 暂无

深度优先遍历（DFS）

思路

这道题要求解一个数字。并且每一个格子能够跳的状态是确定的。 因此我们的思路就是“状态机”（动态规划），暴力遍历（BFS or DFS），这里我们使用 DFS。（注意这几种思路并无本质不同）

对于每一个号码键盘，我们可以转移的状态是确定的，我们做一个”预处理“，将这些状态转移记录到一个数组 jump，其中 jump[i] 表示 i 位置可以跳的点（用一个数组来表示）。问题转化为：

- 从 0 开始所有的路径
- 从 1 开始所有的路径
- 从 2 开始所有的路径
- ...
- 从 9 开始所有的路径

不管从几开始思路都是一样的。 我们使用一个函数 f(i, n) 表示 `骑士在 i 的位置，还剩下 N 步可以走` 的时候可以拨出的总的号码数。那么问题就是求解 `f(0, N) + f(1, N) + f(2, N) + ... + f(9, N)` 。对于 f(i, n)，我们初始化 cnt 为 0，由于 i 能跳的格子是 jump[i]，我们将其 `cnt += f(j, n - 1)` ，其中 j 属于 jump[i]，最终返回 cnt 即可。

不难看出，这种算法有大量重复计算，我们使用记忆化递归形式来减少重复计算。 这种算法勉强通过。

代码

```
class Solution:
    def knightDialer(self, N: int) -> int:
        cnt = 0
        jump = [[4, 6], [6, 8], [7, 9], [4, 8], [
            0, 3, 9], [], [0, 1, 7], [2, 6], [1, 3], [2, 4]
        visited = dict()

        def helper(i, n):
            if (i, n) in visited: return visited[(i, n)]
            if n == 1:
                return 1
            cnt = 0
            for j in jump[i]:
                cnt += helper(j, n - 1)
            visited[(i, n)] = cnt
            return cnt
        for i in range(10):
            cnt += helper(i, N)
        return cnt % (10**9 + 7)
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

朴素遍历

思路

我们使用迭代的形式来优化上述过程。我们初始化十个变量分别表示键盘不同位置能够拨出的号码数，并且初始化为 1。接下来我们只要循环 N - 1 次，不断更新状态即可。不过这种算法和上述算法并无本质不同。

代码

```
class Solution:
    def knightDialer(self, N: int) -> int:
        a0 = a1 = a2 = a3 = a4 = a5 = a6 = a7 = a8 = a9 = 1
        for _ in range(N - 1):
            a0, a1, a2, a3, a4, a5, a6, a7, a8, a9 = a4 + a
                a9, a4 + a8, a0 + a3 + a9, 0, a0 + a1 + a7,
        return (a0 + a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址 (947. 移除最多的同行或同列石头)

https://leetcode-cn.com/problems/most-stones-removed-with-same-row-or-column/

题目描述

n 块石头放置在二维平面中的一些整数坐标点上。每个坐标点上最多只能有一块

如果一块石头的 同行或者同列 上有其他石头存在，那么就可以移除这块石头。

给你一个长度为 n 的数组 stones ，其中 stones[i] = [xi, yi] 表示第

示例 1：

输入：stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]
输出：5
解释：一种移除 5 块石头的方法如下所示：
1. 移除石头 [2,2] ，因为它和 [2,1] 同行。
2. 移除石头 [2,1] ，因为它和 [0,1] 同列。
3. 移除石头 [1,2] ，因为它和 [1,0] 同行。
4. 移除石头 [1,0] ，因为它和 [0,0] 同列。
5. 移除石头 [0,1] ，因为它和 [0,0] 同行。
石头 [0,0] 不能移除，因为它没有与另一块石头同行/列。
示例 2：

输入：stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]
输出：3
解释：一种移除 3 块石头的方法如下所示：
1. 移除石头 [2,2] ，因为它和 [2,0] 同行。
2. 移除石头 [2,0] ，因为它和 [0,0] 同列。
3. 移除石头 [0,2] ，因为它和 [0,0] 同行。
石头 [0,0] 和 [1,1] 不能移除，因为它们没有与另一块石头同行/列。
示例 3：

输入：stones = [[0,0]]
输出：0
解释：[0,0] 是平面上唯一一块石头，所以不可以移除它。

提示：

1 <= stones.length <= 1000
0 <= xi, yi <= 104
不会有两块石头放在同一个坐标点上

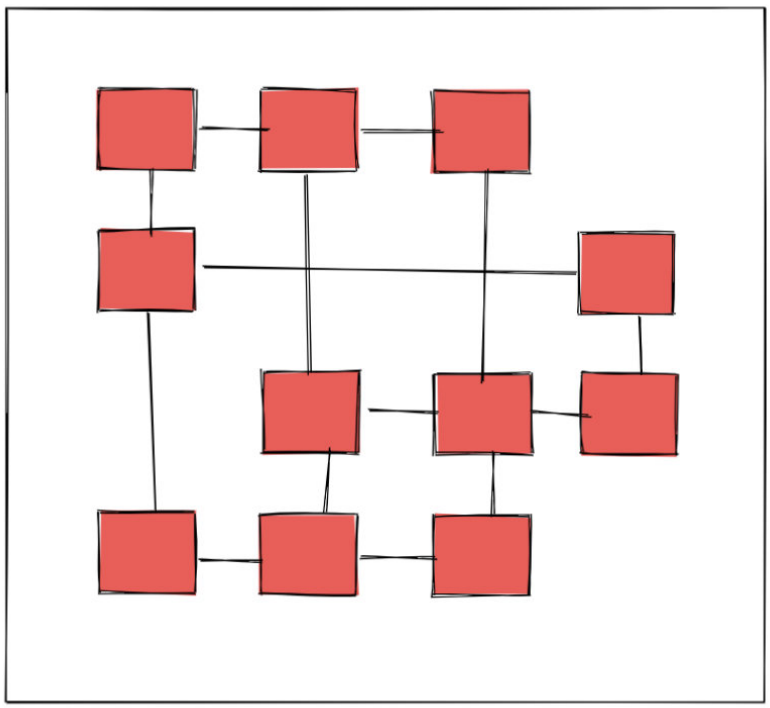
前置知识

- 并查集

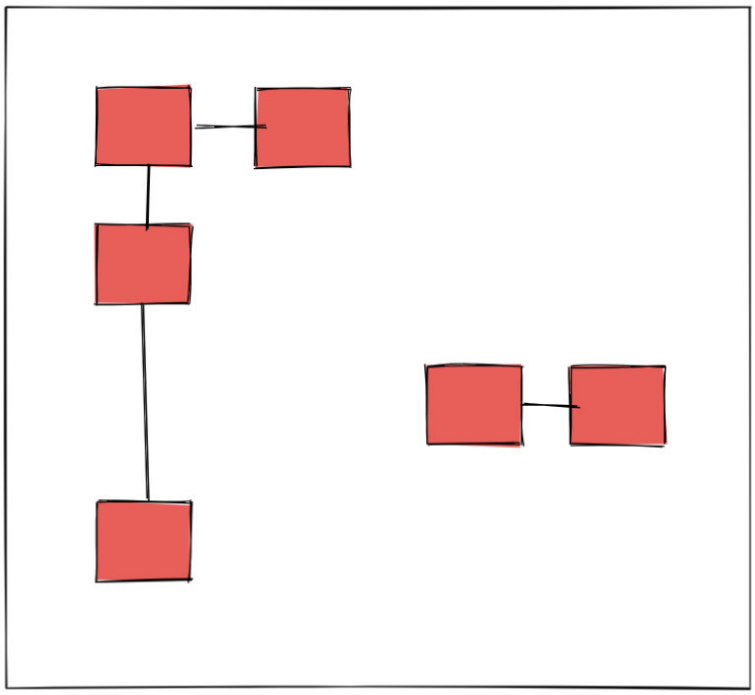
思路

读完题目之后， 看了下数据范围。我猜测可能和动态规划什么的有关，**而且时间复杂度是 $O(n^2)$ 左右**，其中 n 为 stones 的长度。继续看了下示例，然后跟着思考了一下，发现很像是某种联通关系。 类似的题目有很多，题目描述记不太清楚了。大概意思是给你一个二维网格，行和列需要**一起增加**，求最小和什么的。这类题目都是行和列具有某种绑定关系。 于是我想到了使用并查集来做。 后面想了一下，反正就是**求联通区域的个数**，因此使用 DFS 和 BFS 都可以。如果你想使用 DFS 或者 BFS 来解，可以结合我之前写的 小岛专题 练习一下哦。

继续分析下题目。 题目的意思是任意一个石头可以消除和它同行和同列的其他石子。于是我就想象出了下面这样一幅图，其中红色的方块表示石子，方块的连线表示离得最近的可以消除的石子。实际上，一个石子除了可以消除图中线条直接相连的石子，还可以消除邻居的邻居。**这提示我们使用并查集维护这种联通关系**，联通的依据自然就是列或者行一样。



上面是一个全联通的图。如下是有两个联通域的图。



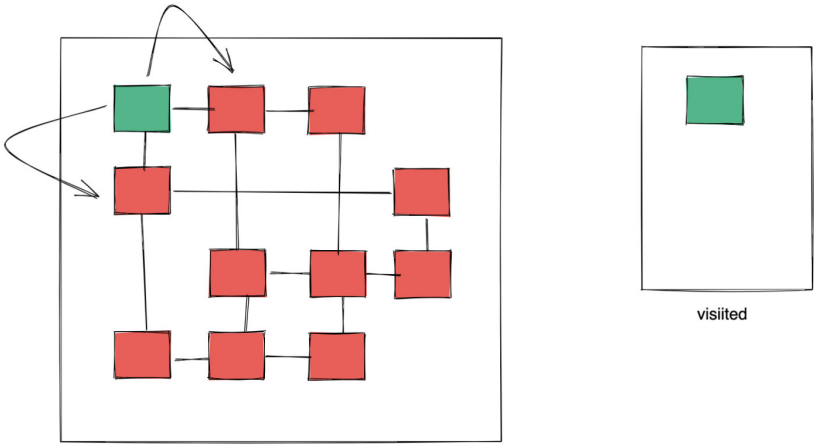
有了上面的知识，其实就可以将石子全部建立并查集的联系，并计算联通子图的个数。答案就是 n - 联通子图的个数，其中 n 为 stones 的长度。

核心代码：

```
n = len(stones)
# 标准并查集模板
uf = UF(n)
# 两个 for 循环作用是将所有石子两两合并
for i in range(n):
    for j in range(i + 1, n):
        # 如果行或者列相同，将其联通成一个子图
        if stones[i][0] == stones[j][0] or stones[i][1] ==
return n - uf.cnt
```

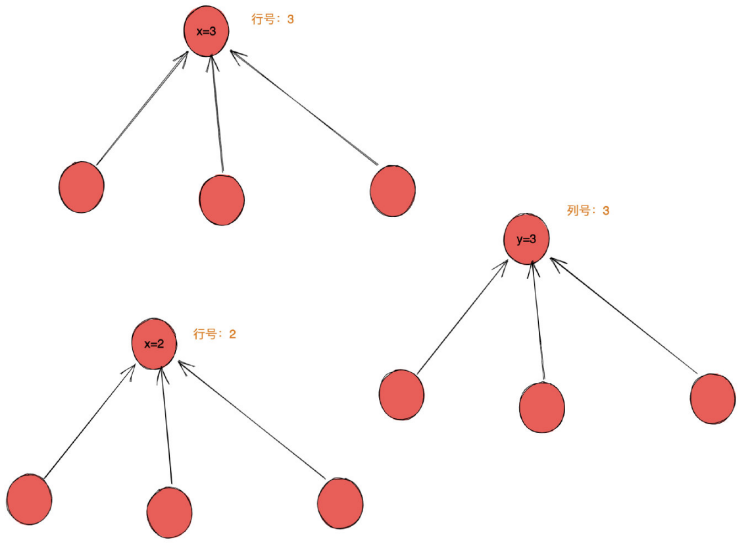
有的人想问，这可行么？即**我可以将一个联通子图的石子移除只剩下一个么？**

答案是肯定的。其实上面我提到了这道题也可使用 DFS 和 BFS 的方式来做。如果你使用 DFS 的方式来做，会发现其实 **DFS 路径的取反就是消除的顺序**，当然消除的顺序不唯一，因为遍历访问联通子图的序列并不唯一。 如果题目要求我们求移除顺序，那我们可以考虑使用 DFS 来做，同时记录路径信息即可。



使用遍历的方式（BFS 或者 DFS），由于每次访问一个石子都需要使用 visited 来记录访问信息防止环的产生，因此 visited 的逆序也是一个可行的移除顺序。不过这要求你的 visited 的是有序的。实现的方法有很多，有点偏题了，这里就不赘述了。

实际上，上面的并查集代码仍然可以优化。上面的思路是直接将点作为并查集的联通条件。实际上，我们可以将点的横纵坐标分别作为联通条件。即如果横坐标相同的联通到一个子图，纵坐标相同的联通到一个子图。如下图：



为了达到这个模板，我们不能再初始化的时候计算联通域数量了，即不能像上面那样 `uf = UF(n)`（此时联通域个数为 n）。因为横坐标，纵坐标分别有多少不重复的我们是不知道的，一种思路是先计算出**横坐标，纵坐标分别有多少不重复的**。这当然可以，还有一种思路是在 find 过程中计算，这样 one pass 即可完成，具体见下方代码区。

由于我们需要**区分横纵坐标**（上面图也可以看出来），因此可**用一种映射**区分二者。由于题目限定了横纵坐标取值为 10000 以内（包含 10000），一种思路就是将 x 或者 y 加上 10001。

核心代码：

```
n = len(stones)
uf = UF(0)
for i in range(n):
    uf.union(stones[i][0] + 10001, stones[i][1])
return n - uf.cnt
```

代码

并查集

代码支持： Python3

其中 `class UF` 部分是标准的无权并查集模板，我一行代码都没变。关于模板可以去 并查集 查看。

```
class UF:
    def __init__(self, M):
        self.parent = {}
        self.cnt = 0
        # 初始化 parent, size 和 cnt
        for i in range(M):
            self.parent[i] = i
            self.cnt += 1

    def find(self, x):
        if x != self.parent[x]:
            self.parent[x] = self.find(self.parent[x])
            return self.parent[x]
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        leader_p = self.find(p)
        leader_q = self.find(q)
        self.parent[leader_p] = leader_q
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)

class Solution:
    def removeStones(self, stones: List[List[int]]) -> int:
        n = len(stones)
        uf = UF(n)
        for i in range(n):
            for j in range(i + 1, n):
                if stones[i][0] == stones[j][0] or stones[i
        return n - uf.cnt
```

复杂度分析

令 n 为数组 stones 的长度。

- 时间复杂度：$O(n^2logn)$
- 空间复杂度：$O(n)$

优化的并查集

代码支持： Python3

```python
class UF:
    def __init__(self, M):
        self.parent = {}
        self.cnt = 0

    def find(self, x):
        if x not in self.parent:
            self.cnt += 1
            self.parent[x] = x
        if x != self.parent[x]:
            self.parent[x] = self.find(self.parent[x])
            return self.parent[x]
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        leader_p = self.find(p)
        leader_q = self.find(q)
        self.parent[leader_p] = leader_q
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)

class Solution:
    def removeStones(self, stones: List[List[int]]) -> int:
        n = len(stones)
        uf = UF(0)
        for i in range(n):
            uf.union(stones[i][0] + 10001, stones[i][1])
        return n - uf.cnt
```

复杂度分析

令 n 为数组 stones 的长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(n)$

DFS

代码支持： Java

> by 一位不愿透露姓名的热心网友。

```
public int removeStones(int[][] stones) {
        Set visit = new HashSet();
        int count = 0;
        int offset = 10000;
        HashMap <Integer,List<int []>>map = new HashMap();

        // 构造图 每一项是一个节点
        for (int i = 0; i < stones.length; i++) {
            int [] node = stones[i];
            List<int []> list =   map.getOrDefault(node[0]-
            list.add(node);
            map.put(node[0]-offset,list);

            List<int []> list1 = map.getOrDefault(node[1],
            list1.add(node);
            map.put(node[1],list1);
        }
        // 寻找联通分量
        for (int i = 0; i < stones.length; i++) {
            int [] node = stones[i];
            if (!visit.contains((node))){
                visit.add((node));
                dfs(node,visit,map);
                count++;
            }
        }
        return stones.length-count;
    }

    // 遍历节点
    public void dfs(int[]node, Set set,HashMap <Integer,Lis
        int offset = 10000;
        List<int []> list =   map.getOrDefault(node[0]-offs
        for (int i = 0; i < list.size(); i++) {
            int[] item = list.get(i);
            if (!set.contains((item))){
                set.add((item));
                dfs(item,set,map);
            }
        }
        List<int []> list2 =   map.getOrDefault(node[1],new
        for (int i = 0; i < list2.size(); i++) {
            int[] item = list2.get(i);
            if (!set.contains((item))){
                set.add((item));
                dfs(item,set,map);
            }
```

```
            }
        }
```

复杂度分析

令 n 为数组 stones 的长度。

- 时间复杂度：建图和遍历图的时间均为 $O(n)$
- 空间复杂度：$O(n)$

力扣的小伙伴的点下我头像的关注按钮，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。

题目地址 (959. 由斜杠划分区域）

https://leetcode-cn.com/problems/regions-cut-by-slashes/

题目描述

在由 1 x 1 方格组成的 N x N 网格 grid 中，每个 1 x 1 方块由 /、\

（请注意，反斜杠字符是转义的，因此 \ 用 "\\" 表示。）。

返回区域的数目。

示例 1：

输入：
[
  " /",
  "/ "
]
输出：2
解释：2x2 网格如下：

示例 2：

输入：
[
  " /",
  " "
]
输出：1
解释：2x2 网格如下：

示例 3：

输入：
[
  "\\/",
  "/\\"
]
输出：4
解释：（回想一下，因为 \ 字符是转义的，所以 "\\/" 表示 \/，而 "/\\'
2x2 网格如下：

示例 4：

输入：
[
  "/\\",
  "\\/"
]
输出：5
解释：（回想一下，因为 \ 字符是转义的，所以 "/\\" 表示 /\，而 "\\/'

```
2x2 网格如下：

示例 5：

输入：
[
  "//",
  "/ "
]
输出：3
解释：2x2 网格如下：



提示：

1 <= grid.length == grid[0].length <= 30
grid[i][j] 是 '/'、'\'、或 ' '。
```

前置知识

- BFS
- DFS
- 并查集

公司

- 暂无

并查集

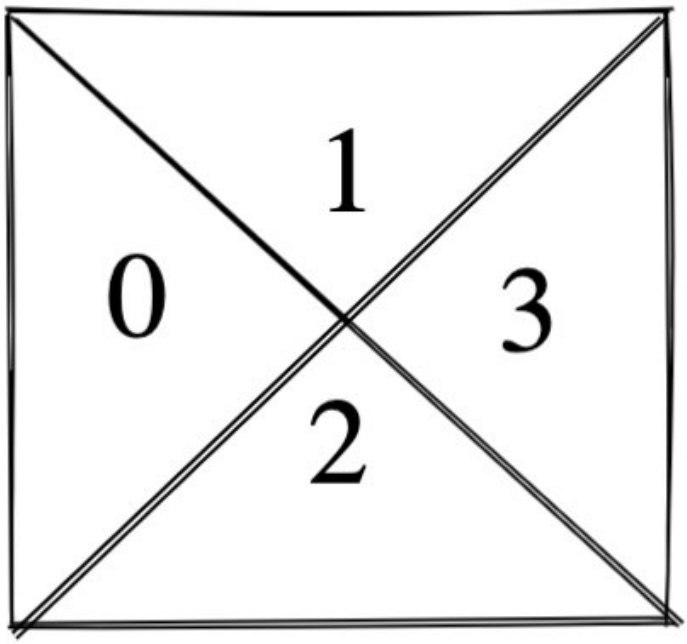
题目给了一个网格，网格有三个符号，分别是左斜杠，右斜杠和空格。我们要做的就是根据这些符号，将网格分成若干区域，并求区域的个数。

了解了题目之后，我们发现这其实就是一个求联通域个数的题目。这种题目一般有三种解法：**BFS**，**DFS** 和并查集。而如果题目需要求具体的联通信息，则需要使用 BFS 或 DFS 来完成。这里给大家提供 DFS 和并查集两种做法。

思路

使用并查集可以将网格按照如下方式进行逻辑上的划分，之所以进行如下划分的原因是一个网格最多只能被分成如下四个部分，而并查集的处理过程是**合并**，因此初始状态需要是一个个孤立的点，每一个点初始都是一个

独立的联通区域。这在我下方代码的初始化过程有所体现。



> 编号方式无所谓，你可以按照你的喜好编号。不过编号方式改变了，代码要做相应微调。

这里我直接使用了**不带权并查集模板** UF，没有改任何代码。

> 并查集模板在我的刷题插件中，插件可在我的公众号《力扣加加》回复插件获取

而一般的并查集处理信息都是一维的，本题却是二维的，如何存储？实际上很简单，我们只需要做一个简单的数学映射即可。

```
def get_pos(row, col):
    return row * n + col
```

如上代码会将原始格子 grid 的 grid[row][col] 映射到新的格子的一维坐标 `row * n + col`，其中 n 为列宽。而由于我们将一个格子拆成了四个，因此需要一个新的大网格来记录这些信息。而原始网格其实和旧的网格一一映射可确定，因此可以直接用原始网格，而不必新建一个新的大网格。如何做呢？其实将上面的坐标转换代码稍微修改就可以了。

```
def get_pos(row, col, i):
    return row * n + col + i
```

接下来就是并查集的部分了：

- 如果是 '/'，则将 0 和 1 合并，2 和 3 合并。
- 如果是 '\'，则将 0 和 2 合并，1 和 3 合并。
- 如果是 ' '，则将 0, 1, 2, 3 合并。

最终返回联通区域的个数即可。

需要特别注意的是当前格子可能和原始格子的上面，下面，左面和右面的格子联通。因此不能仅仅考虑上面的格子内部的联通，还需要考虑相邻的格子的联通。为了避免**重复计算**，我们不能考虑四个方向，而是只能考虑两个方向，这里我考虑了上面和左面。

代码

代码支持： Python3

Python Code:

```
class UF:
  def __init__(self, M):
      self.parent = {}
      self.cnt = 0
      # 初始化 parent, size 和 cnt
      for i in range(M):
          self.parent[i] = i
          self.cnt += 1

  def find(self, x):
      if x != self.parent[x]:
          self.parent[x] = self.find(self.parent[x])
          return self.parent[x]
      return x
  def union(self, p, q):
      if self.connected(p, q): return
      leader_p = self.find(p)
      leader_q = self.find(q)
      self.parent[leader_p] = leader_q
      self.cnt -= 1
  def connected(self, p, q):
      return self.find(p) == self.find(q)

class Solution:
    def regionsBySlashes(self, grid):
        n = len(grid)
        N = n * n * 4
        uf = UF(N)
        def get_pos(row, col, i):
            return (row * n + col) * 4 + i
        for row in range(n):
            for col in range(n):
                v = grid[row][col]
                if row > 0:
                    uf.union(get_pos(row - 1, col, 2), get_
                if col > 0:
                    uf.union(get_pos(row, col - 1, 3), get_
                if v == '/':
                    uf.union(get_pos(row, col, 0), get_pos(
                    uf.union(get_pos(row, col, 2), get_pos(
                if v == '\\':
                    uf.union(get_pos(row, col, 1), get_pos(
                    uf.union(get_pos(row, col, 0), get_pos(
                if v == ' ':
                    uf.union(get_pos(row, col, 0), get_pos(
                    uf.union(get_pos(row, col, 1), get_pos(
```

```
                    uf.union(get_pos(row, col, 2), get_pos(

        return uf.cnt
```

复杂度分析

令 n 为网格的边长。

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

DFS

思路

要使用 DFS 在二维网格计算联通区域，我们需要对数据进行预处理。如果不明白为什么，可以看下我之前写的小岛专题。

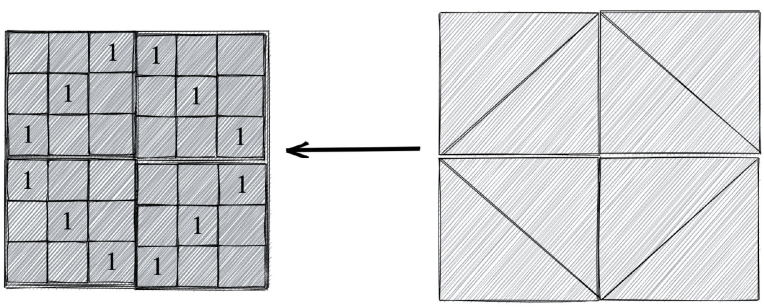
由于题目是“/” 和 "\" 将联通区域进行了分割。因此我们可以将 “/” 和 "\" 看成是陆地，其他部分看成是水。因此我们的目标就转化为小岛问题中的求水的区域个数。

至此，我们的预处理逻辑就清楚了。就是将题目中的“/” 和 "\" 改成 1，其他空格改成 0，然后从 0 启动搜索（可以是 BFS 或者 DFS），边搜索边将水变成陆地，最终启动搜索的次数就是水的区域个数。

将 “/” 和 "\" 直接变为 1 是肯定不行的。那将 “/” 和 "\" 变成一个 2 X 2 的格子呢？也是不行的，因为无法处理上面提到的相邻格子的联通情况。

因此我们需要将 “/” 和 "\" 变成一个 3 X 3 的格子。

> 4 X 4 以及更多的格子也是可以的，但没有必要了，那样只会徒增时间和空间。



代码

代码支持： Python3

Python Code:

```python
class Solution:
    def regionsBySlashes(self, grid: List[str]) -> int:
        m, n = len(grid), len(grid[0])
        new_grid = [[0 for _ in range(3 * n)] for _ in rang
        ans = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '/':
                    new_grid[3 * i][3 * j + 2] = 1
                    new_grid[3 * i + 1][3 * j + 1] = 1
                    new_grid[3 * i + 2][3 * j] = 1
                if grid[i][j] == '\\':
                    new_grid[3 * i][3 * j] = 1
                    new_grid[3 * i + 1][3 * j + 1] = 1
                    new_grid[3 * i + 2][3 * j + 2] = 1
        def dfs(i, j):
            if 0 <= i < 3 * m and 0 <= j < 3 * n and new_gr
                new_grid[i][j] = 1
                dfs(i + 1, j)
                dfs(i - 1, j)
                dfs(i, j + 1)
                dfs(i, j - 1)
        for i in range(3 * m):
            for j in range(3 * n):
                if new_grid[i][j] == 0:
                    ans += 1
                    dfs(i, j)
        return ans
```

复杂度分析

令 n 为网格的边长。

- 时间复杂度：虽然我们在 $9 *m* n$ 的网格中嵌套了 dfs，但由于每个格子最多只会被处理一次，因此时间复杂度仍然是 $O(n^2)$
- 空间复杂度：主要是 new_grid 的空间，因此空间复杂度是 $O(n^2)$

扩展

这道题的 BFS 解法留给大家来完成。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址 (978. 最长湍流子数组)

https://leetcode-cn.com/problems/longest-turbulent-subarray/

题目描述

```
当 A 的子数组 A[i], A[i+1], ..., A[j] 满足下列条件时，我们称其为

若 i <= k < j，当 k 为奇数时， A[k] > A[k+1]，且当 k 为偶数时，A
或 若 i <= k < j，当 k 为偶数时，A[k] > A[k+1] ，且当 k 为奇数时
也就是说，如果比较符号在子数组中的每个相邻元素对之间翻转，则该子数组是

返回 A 的最大湍流子数组的长度。



示例 1：

输入：[9,4,2,10,7,8,8,1,9]
输出：5
解释：(A[1] > A[2] < A[3] > A[4] < A[5])
示例 2：

输入：[4,8,12,16]
输出：2
示例 3：

输入：[100]
输出：1


提示：

1 <= A.length <= 40000
0 <= A[i] <= 10^9
```

前置知识

- 滑动窗口

公司

- 暂无

思路

我们先尝试从题目给的例子打开思路。

对于 A 为 [9,4,2,10,7,8,8,1,9] 来说，我用这样的一个数组 arr 来表示 [-, -, +, -, +, 0, -, +]。其含义是 arr[i] 表示 A[i] - A[i - 1]的符号，其中：**+ 表示正号，- 表示 负号，0 表示 A[i] 和 A[i - 1]相同的情况**，那么显然 arr 的长度始终为 A 的长度 - 1。

那么不难得出，题目给出的剩下两个例子的 arr 为：[+, +, +] 和 []。

通过观察不难发现， 实际题目要求的就是**正负相间的最大长度**。如上的三个例子分别为：

> 我用粗体表示答案部分

- [-, **-, +, -, +**, 0, -, +]，答案是 4 + 1
- [**+**, +, +]，答案是 1 + 1
- []，答案是 0 + 1

于是使用滑动窗口求解就不难想到了，实际上题目求的是**连续 xxxx**，你应该有滑动窗口的想法才对，对不对另说，想到是最起码的。

由于 0 是始终不可以出现在答案中的，因此这算是一个临界条件，大家需要注意特殊判断一下，具体参考代码部分。

代码

代码中使用了一个小技巧，就是 a ^ b >= 0 说明其符号相同，这样比相乘判断符号的好处是可以**避免大数溢出**。

```
class Solution:
    def maxTurbulenceSize(self, A: List[int]) -> int:
        ans = 1
        i = 0
        for j in range(2, len(A)):
            if (A[j] == A[j - 1]):
                i = j
            elif (A[j] - A[j - 1]) ^ (A[j - 1] - A[j - 2]) 
                i = j - 1
            ans = max(ans, j - i + 1)
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（987. 二叉树的垂序遍历）

https://leetcode-cn.com/problems/vertical-order-traversal-of-a-binary-tree

题目描述

给定二叉树，按垂序遍历返回其结点值。

对位于 (X, Y) 的每个结点而言，其左右子结点分别位于 (X-1, Y-1) 和 (

把一条垂线从 X = -infinity 移动到 X = +infinity ，每当该垂线与结

如果两个结点位置相同，则首先报告的结点值较小。

按 X 坐标顺序返回非空报告的列表。每个报告都有一个结点值列表。

示例 1：

输入：[3,9,20,null,null,15,7]
输出：[[9],[3,15],[20],[7]]
解释：
在不丧失其普遍性的情况下，我们可以假设根结点位于 (0, 0)：
然后，值为 9 的结点出现在 (-1, -1)；
值为 3 和 15 的两个结点分别出现在 (0, 0) 和 (0, -2)；
值为 20 的结点出现在 (1, -1)；
值为 7 的结点出现在 (2, -2)。

示例 2：

输入：[1,2,3,4,5,6,7]
输出：[[4],[2],[1,5,6],[3],[7]]
解释：
根据给定的方案，值为 5 和 6 的两个结点出现在同一位置。
然而，在报告 "[1,5,6]" 中，结点值 5 排在前面，因为 5 小于 6。

提示：

树的结点数介于 1 和 1000 之间。
每个结点值介于 0 和 1000 之间。

前置知识

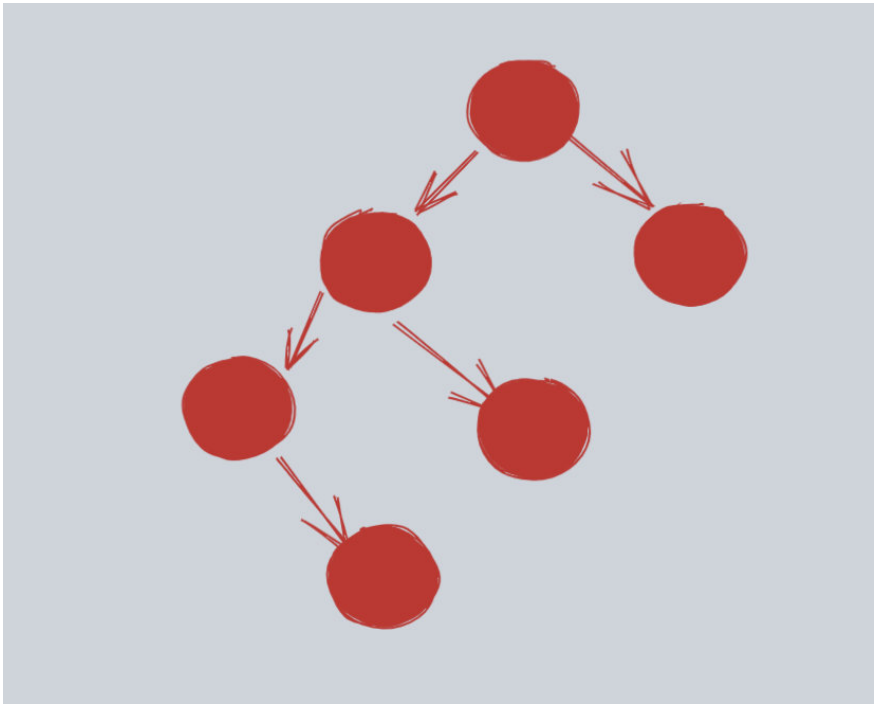
- DFS

- 排序

思路

经过前面几天的学习，希望大家对 DFS 和 BFS 已经有了一定的了解了。

我们先来简化一下问题。假如题目没有 `从上到下的顺序报告结点的值（Y 坐标递减）` ，甚至也没有 `如果两个结点位置相同，则首先报告的结点值较小。` 的限制。是不是就比较简单了？



如上图，我们只需要进行一次搜索，不妨使用 DFS（没有特殊理由，我一般都是 DFS），将节点存储到一个哈希表中，其中 key 为节点的 x 值，value 为横坐标为 x 的节点值列表（不妨用数组表示）。形如：

```
{
    1: [1,3,4]
    -1: [5]
}
```

> 数据是瞎编的，不和题目例子有关联

经过上面的处理， 这个时候只需要对哈希表中的数据进行一次排序输出即可。

ok，如果这个你懂了，我们尝试加上面的两个限制加上去。

1. 从上到下的顺序报告结点的值（Y 坐标递减）
2. 如果两个结点位置相同，则首先报告的结点值较小。

关于第一个限制。其实我们可以再哈希表中再额外增加一层来解决。形如：

```
{
    1: {
        -2,[1,3,4]
        -3,[5]

    },
    -1: {
        -3: [6]
    }
}
```

这样我们除了对 x 排序，再对里层的 y 排序即可。

再来看第二个限制。其实看到上面的哈希表结构就比较清晰了，我们再对值排序即可。

总的来说，我们需要进行三次排序，分别是对 x 坐标，y 坐标 和 值。

那么时间复杂度是多少呢？我们来分析一下：

- 哈希表最外层的 key 总个数是最大是树的宽度。
- 哈希表第二层的 key 总个数是树的高度。
- 哈希表值的总长度是树的节点数。

也就是说哈希表的总容量和树的总的节点数是同阶的。因此空间复杂度为 $O(N)$， 排序的复杂度大致为 $NlogN$，其中 N 为树的节点总数。

代码

- 代码支持：Python，JS，CPP

Python Code:

```
class Solution(object):
    def verticalTraversal(self, root):
        seen = collections.defaultdict(
            lambda: collections.defaultdict(list))

        def dfs(root, x=0, y=0):
            if not root:
                return
            seen[x][y].append(root.val)
            dfs(root.left, x-1, y+1)
            dfs(root.right, x+1, y+1)

        dfs(root)
        ans = []
        # x 排序、
        for x in sorted(seen):
            level = []
            # y 排序
            for y in sorted(seen[x]):
                # 值排序
                level += sorted(v for v in seen[x][y])
            ans.append(level)

        return ans
```

JS Code(by @suukii)：

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var verticalTraversal = function (root) {
  if (!root) return [];

  // 坐标集合以 x 坐标分组
  const pos = {};
  // dfs 遍历节点并记录每个节点的坐标
  dfs(root, 0, 0);

  // 得到所有节点坐标后，先按 x 坐标升序排序
  let sorted = Object.keys(pos)
    .sort((a, b) => +a - +b)
    .map((key) => pos[key]);

  // 再给 x 坐标相同的每组节点坐标分别排序
  sorted = sorted.map((g) => {
    g.sort((a, b) => {
      // y 坐标相同的，按节点值升序排
      if (a[0] === b[0]) return a[1] - b[1];
      // 否则，按 y 坐标降序排
      else return b[0] - a[0];
    });
    // 把 y 坐标去掉，返回节点值
    return g.map((el) => el[1]);
  });

  return sorted;

  // ********************************
  function dfs(root, x, y) {
    if (!root) return;

    x in pos || (pos[x] = []);
    // 保存坐标数据，格式是：[y, val]
    pos[x].push([y, root.val]);

    dfs(root.left, x - 1, y - 1);
    dfs(root.right, x + 1, y - 1);
```

```
    }
};
```

CPP(by @Francis-xsc):

```
class Solution {
public:
    struct node
    {
        int val;
        int x;
        int y;
        node(int v,int X,int Y):val(v),x(X),y(Y){};
    };
    static bool cmp(node a,node b)
    {
        if(a.x^b.x)
            return a.x<b.x;
        if(a.y^b.y)
            return a.y<b.y;
        return a.val<b.val;
    }
    vector<node> a;
    int minx=1000,maxx=-1000;
    vector<vector<int>> verticalTraversal(TreeNode* root) {
        dfs(root,0,0);
        sort(a.begin(),a.end(),cmp);
        vector<vector<int>>ans(maxx-minx+1);
        for(auto xx:a)
        {
            ans[xx.x-minx].push_back(xx.val);
        }
        return ans;
    }
    void dfs(TreeNode* root,int x,int y)
    {
        if(root==nullptr)
            return;
        if(x<minx)
            minx=x;
        if(x>maxx)
            maxx=x;
        a.push_back(node(root->val,x,y));
        dfs(root->left,x-1,y+1);
        dfs(root->right,x+1,y+1);
    }
};
```

复杂度分析

- 时间复杂度：$O(NlogN)$，其中 N 为树的节点总数。
- 空间复杂度：$O(N)$，其中 N 为树的节点总数。

题目地址(1004. 最大连续 1 的个数 III)

https://leetcode-cn.com/problems/max-consecutive-ones-iii/

题目描述

```
给定一个由若干 0 和 1 组成的数组 A，我们最多可以将 K 个值从 0 变成 1

返回仅包含 1 的最长（连续）子数组的长度。



示例 1：

输入：A = [1,1,1,0,0,0,1,1,1,1,0], K = 2
输出：6
解释：
[1,1,1,0,0,1,1,1,1,1,1]
粗体数字从 0 翻转到 1，最长的子数组长度为 6。

示例 2：

输入：A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3
输出：10
解释：
[0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
粗体数字从 0 翻转到 1，最长的子数组长度为 10。



提示：

1 <= A.length <= 20000
0 <= K <= A.length
A[i] 为 0 或 1
```

前置知识

-

公司

- 暂无

思路

这道题我在 字节跳动的算法面试题是什么难度？ 提到过这道题的换皮题。大家可以将这两道题目结合起来理解。接下来，我们看下这道题如何解决。

如果题目没有 `最多可以将 K 个值从 0 变成 1` 。这个条件，那么会很简单，是一个常规的滑动窗口模板题。 我们加上这个条件对问题有什么样的影响呢?

这道题我们只需要记录下加入窗口的是 0 还是 1：

- 如果是 1，我们什么都不用做
- 如果是 0，我们将 K 减 1

相应地，我们需要记录移除窗口的是 0 还是 1:

- 如果是 1，我们什么都不做
- 如果是 0，说明加进来的时候就是 0，加进来的时候我们 K 减去了 1，这个时候我们再加回去 1。

如果 K 减少到负数，则收缩窗口大小。不断用满足条件的窗口大小更新答案即可。

为什么这种思路可行?

这其实就是滑动窗口的常见套路。 这种算法可行的根本原因在于其本身就是暴力枚举的优化。如果让你用最暴力的解法如何求解呢？无非就是枚举所有的子数组，这需要 $O(N^2)$ 的时间复杂度，接下来判断子数组是否满足**最多将 k 个 0 变成 1，子数组全部为 1**。如果满足则更新答案即可。 如何对暴力解进行优化呢?

比如现在是判断的子数组 A[2:3]，我们计算出 A[2:3] 有一个 0，也就是说需要将一个 0 变成 1 才行。那么当我们继续判断子数组 A[2:4]，我们只需要判断 A[4]，同时结合 A[2:3]的计数信息即可。**滑动窗口就是专门优化这种每次只在端点变化，中间都不变，从而省去了中间即重复计算，进而将窗口内的计数信息从 O(w) 降低到 O(1)**，其中 w 为窗口大小。

接下来，我们继续优化。实际上也是没有必要两层循环枚举所有子数组的。而是在 $O(n)$ 的时间就可以枚举所有的合法子数组。为什么呢？这是双指针的常见套路。其实你可以换个角度思考：

所有的子数组就是

- 以索引 0 为右端点的所有子数组
- 加上以索引 1 为右端点的所有子数组
- 加上以索引 2 为右端点的所有子数组
- ...

- 加上以索引 n - 1 为右端点的所有子数组，其中 n 为数组长度

这样的话我们就可以使用双指针技巧。右指针模拟右端点，使用左指针模拟左端点了。**如果以索引 i 为右端点的子数组 0 的个数不大于 k，那么左指针 l 没必要右移，因为当前右指针 r 和所有的索引 i, 其中 l <= i <= r 的组合 0 的个数都不会大于 k，但是子数组还更短了，不可能是答案，因此我们直接右移右指针，这是算法的关键**。通过这种方式算法的时间复杂度可从 $O(n^2)$ 降低到 n。

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def longestOnes(self, A: List[int], K: int) -> int:
        i = ans = 0

        for j in range(len(A)):
            K -= A[j] == 0
            while K < 0:
                K += A[i] == 0
                i += 1
            ans = max(ans, j - i + 1)
        return ans
```

甚至更简洁：

```python
class Solution:
    def longestOnes(self, A: List[int], K: int) -> int:
        i = 0

        for j in range(len(A)):
            K -= 1 - A[j]
            if K < 0:
                K += 1 - A[i]
                i += 1
        return j - i + 1
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（1011. 在 D 天内送达包裹的能力）

https://leetcode-cn.com/problems/capacity-to-ship-packages-within-d-days/

题目描述

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

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
```

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 <= D <= weights.length <= 50000
1 <= weights[i] <= 500
```

前置知识

- 二分法

公司

- 阿里

思路

这道题和猴子吃香蕉 简直一摸一样，没有看过的建议看一下那道题。

像这种题如何你能发现本质的考点，那么 AC 是瞬间的事情。 这道题本质上就是从 1，2，3，4，。。。total（其中 toal 是总的货物重量）的有限离散数据中查找给定的数。这里我们不是直接查找 target，而是查找恰好能够在 D 天运完的载货量。

- 容量是 1 可以运完么?
- 容量是 2 可以运完么?
- 容量是 3 可以运完么?
- 。。。
- 容量是 total 可以运完么？（当然可以，因为 D 大于等于 1）

上面不断询问的过程如果回答是 yes 我们直接 return 即可。如果回答是 no，我们继续往下询问。

这是一个典型的二分问题，只不过我们的判断条件略有不同，大概是：

```
def canShip(opacity):
    # 指定船的容量是否可以在D天运完
    lo = 0
    hi = total
    while lo < hi:
        mid = (lo + hi) // 2
        if canShip(mid):
            hi = mid
        else:
            lo = mid + 1

    return lo
```

关键点解析

- 能够识别出是给定的有限序列查找一个数字（二分查找），要求你对二分查找以及变体十分熟悉

代码

语言支持：JS，Python

Python Code:

```python
class Solution:
    def shipWithinDays(self, weights: List[int], D: int) ->
        lo = 0
        hi = 0

        def canShip(opacity):
            days = 1
            remain = opacity
            for weight in weights:
                if weight > opacity:
                    return False
                remain -= weight
                if remain < 0:
                    days += 1
                    remain = opacity - weight
            return days <= D

        for weight in weights:
            hi += weight
        while lo < hi:
            mid = (lo + hi) // 2
            if canShip(mid):
                hi = mid
            else:
                lo = mid + 1

        return lo
```

js Code:

```
/**
 * @param {number[]} weights
 * @param {number} D
 * @return {number}
 */
var shipWithinDays = function (weights, D) {
  let high = weights.reduce((acc, cur) => acc + cur);
  let low = 0;

  while (low < high) {
    let mid = Math.floor((high + low) / 2);
    if (canShip(mid)) {
      high = mid;
    } else {
      low = mid + 1;
    }
  }

  return low;

  function canShip(opacity) {
    let remain = opacity;
    let count = 1;
    for (let weight of weights) {
      if (weight > opacity) {
        return false;
      }
      remain -= weight;
      if (remain < 0) {
        count++;
        remain = opacity - weight;
      }
      if (count > D) {
        return false;
      }
    }
    return count <= D;
  }
};
```

复杂度分析

- 时间复杂度：$O(logN)$
- 空间复杂度：$O(N)$

题目地址(1014. 最佳观光组合)

https://leetcode-cn.com/problems/best-sightseeing-pair/

题目描述

```
给定正整数数组 A，A[i] 表示第 i 个观光景点的评分，并且两个景点 i

一对景点（i < j）组成的观光组合的得分为（A[i] + A[j] + i - j）：景

返回一对观光景点能取得的最高分。

示例：

输入：[8,1,5,2,6]
输出：11
解释：i = 0, j = 2, A[i] + A[j] + i - j = 8 + 5 + 0 - 2 = 11

提示：

2 <= A.length <= 50000
1 <= A[i] <= 1000
```

前置知识

- 动态规划

公司

- 阿里
- 字节

思路

最简单的思路就是两两组合，找出最大的，妥妥超时，我们来看下代码：

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        res = 0
        for i in range(n - 1):
            for j in range(i + 1, n):
                res = max(res, A[i] + A[j] + i - j)
        return res
```

我们思考如何优化。 其实我们可以遍历一遍数组，对于数组的每一项 `A[j] - j` 我们都去前面找 `最大` 的 A[i] + i （这样才能保证结果最大）。

我们考虑使用动态规划来解决, 我们使用 dp[i] 来表示 数组 A 前 i 项的 `A[i] + i` 的最大值。

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        dp = [float('-inf')] * (n + 1)
        res = 0
        for i in range(n):
            dp[i + 1] = max(dp[i], A[i] + i)
            res = max(res, dp[i] + A[i] - i)
        return res
```

如上其实我们发现，dp[i + 1] 只和 dp[i] 有关，这是一个空间优化的信号。我们其实可以使用一个变量来记录，而不必要使用一个数组，代码见下方。

关键点解析

- 空间换时间
- dp 空间优化

代码

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        pre = A[0] + 0
        res = 0
        for i in range(1, n):
            res = max(res, pre + A[i] - i)
            pre = max(pre, A[i] + i)
        return res
```

小技巧

Python 的代码如果不使用 max，而是使用 if else 效率目测会更高，大家可以试一下。

```
class Solution:
    def maxScoreSightseeingPair(self, A: List[int]) -> int:
        n = len(A)
        pre = A[0] + 0
        res = 0
        for i in range(1, n):
            # res = max(res, pre + A[i] - i)
            # pre = max(pre, A[i] + i)
            res = res if res > pre + A[i] - i else pre + A
            pre = pre if pre > A[i] + i else A[i] + i
        return res
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(1015. 可被 K 整除的最小整数)

https://leetcode-cn.com/problems/smallest-integer-divisible-by-k/

题目描述

```
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。


提示：

1 <= K <= 10^5
```

前置知识

- 循环节

公司

- 暂无

思路

这道题是说给定一个 K 值，能否找到一个形如 1，11，111，1111 。。。这样的数字 n 使得 n % K == 0。

首先容易想到的是如果 K 是 2，4，5， 6，8 结尾的话，一定是不行的。直观的解法是从 1，11，111，1111 。。。 这样一直除下去，直到碰到可以整除的，我们返回即可。 但是如果这个数字根本就无法整除怎么办？没错，我们会无限循环下去。我们应该在什么时刻跳出循环，返回 - 1（表示不能整除）呢?

我们拿题目给出的不能整除的 2 来说。

- 1 // 2 等于 1
- 11 // 2 等于 1
- 111 // 2 等于 1
- ...

我们再来一个不能整除的例子 6:

- 1 // 6 等于 1
- 11 // 6 等于 5
- 111 // 6 等于 3
- 1111 // 6 等于 1
- 11111 // 6 等于 5
- ...

通过观察我们发现不断整除的过程，会陷入无限循环，对于 2 来说，其循环节就是 1。对于 6 来说，其循环节来说就是 153。而且由于我们的分母是 6，也就是说余数的可能性一共只有六种情况 0,1,2,3,4,5。

上面是感性的认识， 接下来我们从数学上予以证明。上面的算法用公式来表示就是 `mod = (10 \* mod + 1) % K` 。假如出现了相同的数，我们可以肯定之后会无限循环。比如 153 之后出现了 1，我们可以肯定之后一定是 35。。。 因为我们的 mod 只是和前一个 mod 有关，上面的公式是一个 `纯函数` 。

关键点解析

- 数学（无限循环与循环节）

代码

```python
#
# @lc app=leetcode.cn id=1015 lang=python3
#
# [1015] 可被 K 整除的最小整数
#

# @lc code=start


class Solution:
    def smallestRepunitDivByK(self, K: int) -> int:
        if K % 10 in [2, 4, 5, 6, 8]:
            return - 1
        seen = set()
        mod = 0
        for i in range(1, K + 1):
            mod = (mod * 10 + 1) % K
            if mod in seen:
                return -1
            if mod == 0:
                return ix
            seen.add(mod)
```

相关题目

- 166. 分数到小数

题目地址(1019. 链表中的下一个更大节点)

https://leetcode-cn.com/problems/next-greater-node-in-linked-list/

题目描述

```
给出一个以头节点 head 作为第一个节点的链表。链表中的节点分别编号为：n

每个节点都可能有下一个更大值（next larger value）：对于 node_i，如

返回整数答案数组 answer，其中 answer[i] = next_larger(node_{i+

注意：在下面的示例中，诸如 [2,1,5] 这样的输入（不是输出）是链表的序列



示例 1：

输入：[2,1,5]
输出：[5,5,0]
示例 2：

输入：[2,7,4,3,5]
输出：[7,0,5,5,0]
示例 3：

输入：[1,7,5,1,9,2,5,1]
输出：[7,9,9,9,0,5,0,0]


提示：

对于链表中的每个节点，1 <= node.val <= 10^9
给定列表的长度在 [0, 10000] 范围内
```

前置知识

- 链表
- 栈

公司

- 腾讯

- 字节

思路

看完题目就应该想到单调栈才行，LeetCode 上关于单调栈的题目还不少，难度都不小。但是一旦你掌握了这个算法，那么这些题目对你来说都不是问题了。

如果你不用单调栈，那么可以暴力$O(N^2)$的时间复杂度解决，只需要双层循环即可。但是这种做法应该是过不了关的。使用单调栈可以将时间复杂度降低到线性，当然需要额外的$O(N)$的空间复杂度。

顾名思义，单调栈即满足单调性的栈结构。与单调队列相比，其只在一端进行进出。为了描述方便，以下举例及代码以维护一个整数的单调递减栈为例。将一个元素插入单调栈时，为了维护栈的单调性，需要在保证将该元素插入到栈顶后整个栈满足单调性的前提下弹出最少的元素。

例如，栈中自顶向下的元素为 1，2，4，5 ，插入元素 3 时为了保证单调性需要依次弹出元素 :

- 最开始栈是这样的： [5,4,2,1]
- 为了维护递减特性，1,2 需要被移除。此时栈是这样的： [5,4]
- 我们将 3 push 到栈顶即可
- 此时栈是这样的： [5,4,3]

用代码描述如下：

Python Code:

```
def monoStack(list):
    st = []
    for v in list:
        while len(st) > 0 and v > st[-1]:
            st.pop()
        st.append(v)
    return st
monoStack([5, 4, 2, 1, 3]) # output: [5, 4, 3]
```

关键点

- 单调栈（单调递减栈）
- 单调栈的代码模板

代码

Python Code:

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def nextLargerNodes(self, head):
        res, st = [], []
        while head:
            while len(st) > 0 and head.val > st[-1][1]:
                res[st.pop()[0]] = head.val
            st.append((len(res), head.val))
            res.append(0)
            head = head.next
        return res
```

复杂度分析

其中 N 为链表的长度。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

扩展

甚至可以做到 O(1)的空间复杂度，请参考C# O(n) time O(1) space-time-O(1)-space>)

相关题目

- 每日一题 - 739.Daily Temperatures

题目地址(1020. 飞地的数量)

https://leetcode-cn.com/problems/number-of-enclaves/

题目描述

给出一个二维数组 A，每个单元格为 0（代表海）或 1（代表陆地）。

移动是指在陆地上从一个地方走到另一个地方（朝四个方向之一）或离开网格的

返回网格中无法在任意次数的移动中离开网格边界的陆地单元格的数量。

示例 1：

输入：[[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
输出：3
解释：
有三个 1 被 0 包围。一个 1 没有被包围，因为它在边界上。
示例 2：

输入：[[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
输出：0
解释：
所有 1 都在边界上或可以到达边界。

提示：

1 <= A.length <= 500
1 <= A[i].length <= 500
0 <= A[i][j] <= 1
所有行的大小都相同

前置知识

- DFS
- hashset

解法一 (暴力法)

思路

这是一个典型的可以使用 DFS 进行解决的一类题目， LeetCode 相关的题目有很多。

对于这种题目不管是思路还是代码都有很大的相似性，我们来看下。

暴力法的思路很简单，我们遍历整个矩阵：

- 如果遍历到 0，我们不予理会
- 如果遍历到 1. 我们将其加到 temp
- 不断拓展边界（上下左右）
- 如果 dfs 过程中碰到了边界，说明可以逃脱，我们将累加的 temp 清空
- 如果 dfs 过程之后没有碰到边界，说明无法逃脱。我们将 temp 加到 cnt
- 最终返回 cnt 即可

关键点解析

- visited 记录访问过的节点，防止无限循环。

代码

Python Code:

```
class Solution:
    temp = 0
    meetEdge = False

    def numEnclaves(self, A: List[List[int]]) -> int:
        cnt = 0
        m = len(A)
        n = len(A[0])
        visited = set()

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or (i, j)
                return
            visited.add((i, j))
            if A[i][j] == 1:
                self.temp += 1
            else:
                return
            if i == 0 or i == m - 1 or j == 0 or j == n - 1
                self.meetEdge = True
            dfs(i + 1, j)
            dfs(i - 1, j)
            dfs(i, j + 1)
            dfs(i, j - 1)
        for i in range(m):
            for j in range(n):
                dfs(i, j)
                if not self.meetEdge:
                    cnt += self.temp
                self.meetEdge = False
                self.temp = 0
        return cnt
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(M * N)$

解法二 (原地标记法)

公司

- 暂无

思路

上面的解法空间复杂度很差，我们考虑进行优化， 这里我们使用消除法。即使用题目范围外的数据原地标记是否访问， 这样时间复杂度可以优化到 $O(1)$，这是一种非常常见的优化技巧，请务必掌握，另外文章末尾的题目也是类似的技巧，大家可以结合起来练习。

- 从矩阵边界开始 dfs
- 如果碰到 1 就将其变成 0
- 如果碰到 0 则什么都不做
- 最后我们遍历整个矩阵，数一下 1 的个数即可。

关键点解析

- 原地标记

代码

Python Code:

```
#
# @lc app=leetcode.cn id=1020 lang=python3
#
# [1020] 飞地的数量
#

# @lc code=start


class Solution:

    def numEnclaves(self, A: List[List[int]]) -> int:
        cnt = 0
        m = len(A)
        n = len(A[0])

        def dfs(i, j):
            if i < 0 or i >= m or j < 0 or j >= n or A[i][j
                return
            A[i][j] = 0

            dfs(i + 1, j)
            dfs(i - 1, j)
            dfs(i, j + 1)
            dfs(i, j - 1)
        for i in range(m):
            dfs(i, 0)
            dfs(i, n - 1)
        for j in range(1, n - 1):
            dfs(0, j)
            dfs(m - 1, j)
        for i in range(m):
            for j in range(n):
                if A[i][j] == 1:
                    cnt += 1
        return cnt

        # @lc code=end
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(1)$

参考

- 200.number-of-islands

题目地址(1023. 驼峰式匹配)

https://leetcode-cn.com/problems/camelcase-matching/

题目描述

```
如果我们可以将小写字母插入模式串 pattern 得到待查询项 query，那么待

给定待查询列表 queries，和模式串 pattern，返回由布尔值组成的答案列表



示例 1：

输入：queries = ["FooBar","FooBarTest","FootBall","FrameBuf
输出：[true,false,true,true,false]
示例：
"FooBar" 可以这样生成："F" + "oo" + "B" + "ar"。
"FootBall" 可以这样生成："F" + "oot" + "B" + "all".
"FrameBuffer" 可以这样生成："F" + "rame" + "B" + "uffer".
示例 2：

输入：queries = ["FooBar","FooBarTest","FootBall","FrameBuf
输出：[true,false,true,false,false]
解释：
"FooBar" 可以这样生成："Fo" + "o" + "Ba" + "r".
"FootBall" 可以这样生成："Fo" + "ot" + "Ba" + "ll".
示例 3：

输出：queries = ["FooBar","FooBarTest","FootBall","FrameBuf
输入：[false,true,false,false,false]
解释：
"FooBarTest" 可以这样生成："Fo" + "o" + "Ba" + "r" + "T" + "e


提示：

1 <= queries.length <= 100
1 <= queries[i].length <= 100
1 <= pattern.length <= 100
所有字符串都仅由大写和小写英文字母组成。
```

前置知识

- 双指针

公司

- 暂无

思路

这道题是一道典型的双指针题目。不过这里的双指针并不是指向同一个数组或者字符串，而是指向多个，这道题是指向两个，分别是 query 和 pattern，这种题目非常常见，能够识别和掌握这种题目的解题模板非常重要。对 queries 的每一项我们的逻辑是一样的，这里就以其中一项为例进行讲解。

以 query 为 FooBar，pattern 为 FB 为例。

首先我们来简化一下问题，假如我们没有 `可以在任何位置插入每个字符，也可以插入 0 个字符。` 这个规则。我们的问题会比较简单，这个时候我们的算法是什么样的呢？一起来看下：

1. 首先我们建立两个指针 i 和 j 分别指向 query 和 pattern 的首字母。
2. 当 i 和 j 指向的字母相同的时候，我们同时向后移动两个指针一个单位。
3. 当 i 和 j 指向的字母不同的时候，我们直接返回 False

假如我们要找到的不是子串，而是子序列怎么办？我们不妨假设判断 pattern 是否是 query 的子序列。 其实 LeetCode 实际上也有这样的题目，我们来看下：

1. 首先我们建立两个指针 i 和 j 分别指向 query 和 pattern 的首字母。
2. 当 i 和 j 指向的字母相同的时候，我们同时向后移动两个指针一个单位。
3. 当 i 和 j 指向的字母不同的时候，我们移动 i 指针。
4. 当 i 超出 query 范围的时候，我们只需要判断 pattern 是否达到了终点即可。当然我们也可以提前退出。

我们直接参考下 LeetCode 392. 判断子序列。

代码：

> 给定字符串 s 和 t ，判断 s 是否为 t 的子序列

Python Code:

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        i = 0
        j = 0
        while j < len(t):
            if i < len(s) and s[i] == t[j]:
                i += 1
                j += 1
            else:
                j += 1
            if i >= len (s):
                return True
        return i == len(s)
```

然后我们加上 `可以在任何位置插入每个字符，也可以插入 0 个字符。` 这个规则。来看下有什么不同：

1. 首先我们建立两个指针 i 和 j 分别指向 query 和 pattern 的首字母。
2. 当 i 和 j 指向的字母相同的时候，我们同时向后移动两个指针一个单位。
3. 当 i 和 j 指向的字母不同的时候，我们继续判断 i 指向的元素是否是小写。
4. 如果是小写我们只把 i 向后移动一个单位。
5. 如果不是小写我们直接返回 False

关键点解析

- 双指针
- 字符串匹配
- 子序列
- 子串

代码

Python Code:

```
class Solution:
    def camelMatch(self, queries: List[str], pattern: str)
        res = []
        for query in queries:
            i = 0
            j = 0
            while i < len(query):
                if j < len(pattern) and query[i] == patter
                    i += 1
                    j += 1
                elif query[i].islower():
                    i += 1
                else:
                    break
            if i == len(query) and j == len(pattern):
                res.append(True)
            else:
                res.append(False)
        return res
```

复杂度分析

其中 N 为 queries 的长度， M 为 queries 的平均长度， P 为 pattern 的长度。

- 时间复杂度：$O(N *M* P)$
- 空间复杂度：$O(1)$

扩展

这是一个符合直觉的解法，但是却不是一个很优秀的解法，那么你有想到什么优秀的解法么？

参考

- 392. 判断子序列

题目地址(1031. 两个非重叠子数组的最大和)

https://leetcode-cn.com/problems/maximum-sum-of-two-non-overlapping-subarrays/

题目描述

```
给出非负整数数组 A ，返回两个非重叠（连续）子数组中元素的最大和，子数组

从形式上看，返回最大的 V，而 V = (A[i] + A[i+1] + ... + A[i+L-1

0 <= i < i + L - 1 < j < j + M - 1 < A.length, 或
0 <= j < j + M - 1 < i < i + L - 1 < A.length.


示例 1：

输入：A = [0,6,5,2,2,5,1,9,4], L = 1, M = 2
输出：20
解释：子数组的一种选择中，[9] 长度为 1，[6,5] 长度为 2。
示例 2：

输入：A = [3,8,1,3,2,1,8,9,0], L = 3, M = 2
输出：29
解释：子数组的一种选择中，[3,8,1] 长度为 3，[8,9] 长度为 2。
示例 3：

输入：A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3
输出：31
解释：子数组的一种选择中，[5,6,0,9] 长度为 4，[0,3,8] 长度为 3。


提示：

L >= 1
M >= 1
L + M <= A.length <= 1000
0 <= A[i] <= 1000
```

前置知识

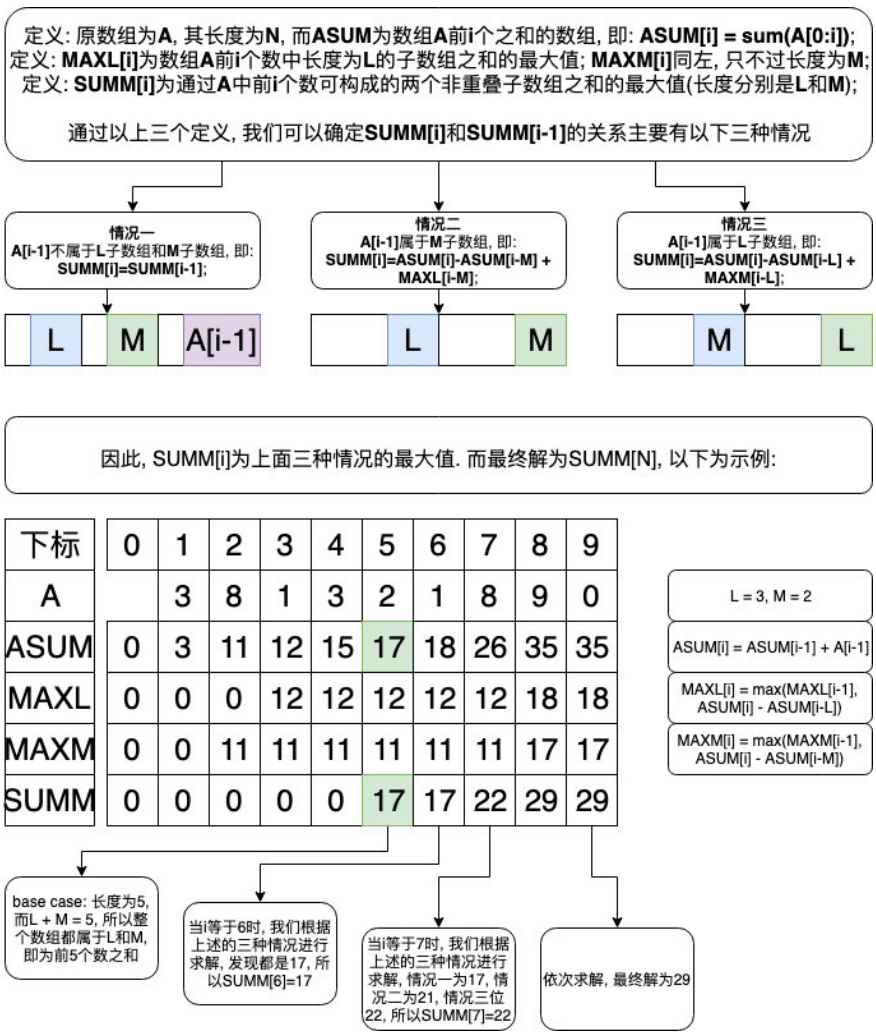
- 数组

公司

- 字节

思路(动态规划)

题目中要求在前 N(数组长度)个数中找出长度分别为 L 和 M 的非重叠子数组之和的最大值, 因此, 我们可以定义数组 A 中前 i 个数可构成的非重叠子数组 L 和 M 的最大值为 SUMM[i], 并找到 SUMM[i]和 SUMM[i-1]的关系, 那么最终解就是 SUMM[N]. 以下为图解:



关键点解析

1. 注意图中描述的都是 A[i-1], 而不是 A[i], 因为 base case 为空数组, 而不是 A[0];
2. 求解图中 ASUM 数组的时候, 注意定义的是 ASUM[i] = sum(A[0:i]), 因此当 i 等于 0 时, A[0:0]为空数组, 即: ASUM[0]为 0, 而 ASUM[1]才等于 A[0];

3. 求解图中 MAXL 数组时, 注意 i < L 时, 没有意义, 因为长度不够, 所以从 i = L 时才开始求解;
4. 求解图中 MAXM 数组时, 也一样, 要从 i = M 时才开始求解;
5. 求解图中 SUMM 数组时, 因为我们需要一个 L 子数组和一个 M 子数组, 因此长度要大于等于 L+M 才有意义, 所以要从 i = L + M 时开始求解.

代码

语言支持: Python, CPP

Python Code:

```python
class Solution:
    def maxSumTwoNoOverlap(self, a: List[int], l: int, m: i
        """

        define asum[i] as the sum of subarray, a[0:i]
        define maxl[i] as the maximum sum of l-length subar
        define maxm[i] as the maximum sum of m-length subar
        define msum[i] as the maximum sum of non-overlap l-

        case 1: a[i] is both not in l-length subarray and r
        case 2: a[i] is in l-length subarray, then msum[i]
        case 3: a[i] is in m-length subarray, then msum[i]

        so, msum[i] = max(msum[i - 1], asum[i] - asum[i-l]
        """

        alen, tlen = len(a), l + m
        asum = [0] * (alen + 1)
        maxl = [0] * (alen + 1)
        maxm = [0] * (alen + 1)
        msum = [0] * (alen + 1)

        for i in range(tlen):
            if i == 1:
                asum[i] = a[i - 1]
            elif i > 1:
                asum[i] = asum[i - 1] + a[i - 1]
            if i >= l:
                maxl[i] = max(maxl[i - 1], asum[i] - asum[i
            if i >= m:
                maxm[i] = max(maxm[i - 1], asum[i] - asum[i

        for i in range(tlen, alen + 1):
            asum[i] = asum[i - 1] + a[i - 1]
            suml = asum[i] - asum[i - l]
            summ = asum[i] - asum[i - m]
            maxl[i] = max(maxl[i - 1], suml)
            maxm[i] = max(maxm[i - 1], summ)
            msum[i] = max(msum[i - 1], suml + maxm[i - l],

        return msum[-1]
```

CPP Code:

```
class Solution {
private:
    int get(vector<int> &v, int i) {
        return (i >= 0 && i < v.size()) ? v[i] : 0;
    }
public:
    int maxSumTwoNoOverlap(vector<int>& A, int L, int M) {
        int N = A.size(), ans = 0;
        partial_sum(A.begin(), A.end(), A.begin());
        vector<int> maxLeft(N, 0), maxRight(N, 0);
        for (int i = L - 1; i < N; ++i) maxLeft[i] = max(ge
        for (int i = N - L; i >= 0; --i) maxRight[i] = max(
        for (int i = M - 1; i < N; ++i) {
            int sum = A[i] - get(A, i - M)
                + max(get(maxLeft, i - M), get(maxRight, i
            ans = max(ans, sum);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(N)$，其中 N 为数组长度。

扩展

1. 代码中, 求解了 4 个动态规划数组来求解最终值, 有没有可能只用两个数组来求解该题, 可以的话, 需要保留的又是哪两个数组?
2. 代码中, 求解的 4 动态规划数组的顺序能否改变, 哪些能改, 哪些不能改?

如果采用前缀和数组的话，可以只使用 O(n)的空间来存储前缀和，O(1)的动态规划状态空间来完成。C++代码如下:

```
class Solution {
public:
    int maxSumTwoNoOverlap(vector<int>& A, int L, int M) {
        auto tmp = vector<int>{A[0]};
        for (auto i = 1; i < A.size(); ++i) {
            tmp.push_back(A[i] + tmp[i - 1]);
        }
        auto res = tmp[L + M - 1], lMax = tmp[L - 1], mMax
        for (auto i = L + M; i < tmp.size(); ++i) {
            lMax = max(lMax, tmp[i - M] - tmp[i - M - L]);
            mMax = max(mMax, tmp[i - L] - tmp[i - L - M]);
            res = max(res, max(lMax + tmp[i] - tmp[i - M],
        }
        return res;
    }
};
```

题目地址(1043. 分隔数组以得到最大和)

https://leetcode-cn.com/problems/partition-array-for-maximum-sum/

题目描述

给你一个整数数组 arr，请你将该数组分隔为长度最多为 k 的一些（连续）子

返回将数组分隔变换后能够得到的元素最大和。

注意，原数组和分隔后的数组对应顺序应当一致，也就是说，你只能选择分隔数

示例 1：

输入：arr = [1,15,7,9,2,5,10], k = 3
输出：84
解释：
因为 k=3 可以分隔成 [1,15,7] [9] [2,5,10]，结果为 [15,15,15,9,
若是分隔成 [1] [15,7,9] [2,5,10]，结果就是 [1, 15, 15, 15, 10,

示例 2：

输入：arr = [1,4,1,5,7,3,6,1,9,9,3], k = 4
输出：83

示例 3：

输入：arr = [1], k = 1
输出：1

提示：

1 <= arr.length <= 500
0 <= arr[i] <= 109
1 <= k <= arr.length

前置知识

- 动态规划
- 记忆化递归

公司

- 暂无

记忆化递归

思路

这道题的思路无非就是暴力枚举所有的可能，求所有可能中的最大值即可。

因此我们可以枚举所有的 i，然后计算区间 [i:j] 的可能区间和，其中 j 的取值范围是 [i:i+k]。如何对区间求和呢？ 其实也容易，只需要用一个变量 max_ele 记录区间最大值（这在遍历的时候可以同时取得），然后 max_ele * (j-i+1) 即可，其中 j - i + 1 为区间的长度。这样我们就算出了区间 [i:i+k] 的区间和最大值。也就是说我们将问题规模缩小了，继续使用同样的方法直到问题缩小到寻常即可。使用递归可以轻松达到这一点。

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def maxSumAfterPartitioning(self, arr: List[int], k: in
        @lru_cache(None)
        def dp(i):
            if i >= len(arr): return 0
            ans = 0
            max_value = -1
            for steps in range(1, k + 1):
                if i + steps - 1 < len(arr): max_value = ma
                else: break
                ans = max(ans, max_value * steps +  dp(i +
            return ans
        return dp(0)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

记忆化递归

思路

同上。我们可以将上面的代码改成普通 dp 形式。

只要：

- 将递归的代码改成 for 循环
- 记忆化的地方用 dp 数组代替

即可轻松实现。

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def maxSumAfterPartitioning(self, nums: List[int], k: i
        n = len(nums)
        dp = [0] * (n+1)

        for i in range(1, n+1):
            max_ele = 0
            for j in range(i, min(n+1, i+k)):
                max_ele = max(max_ele, nums[j-1])
                # range: [i,j]
                dp[j] = max(dp[j], (j-i+1) * max_ele + dp[i
        return max(dp)
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(1104. 二叉树寻路)

https://leetcode-cn.com/problems/path-in-zigzag-labelled-binary-tree/

题目描述

```
在一棵无限的二叉树上，每个节点都有两个子节点，树中的节点 逐行 依次按 “

如下图所示，在奇数行（即，第一行、第三行、第五行……）中，按从左到右的顺

而偶数行（即，第二行、第四行、第六行……）中，按从右到左的顺序进行标记。

![](https://tva1.sinaimg.cn/large/007S8ZIlly1ghlu59hpv3j310

给你树上某一个节点的标号 label，请你返回从根节点到该标号为 label 节点

示例 1：

输入：label = 14
输出：[1,3,4,14]
示例 2：

输入：label = 26
输出：[1,2,6,10,26]

提示：

1 <= label <= 10^6
```

前置知识

- 二叉树

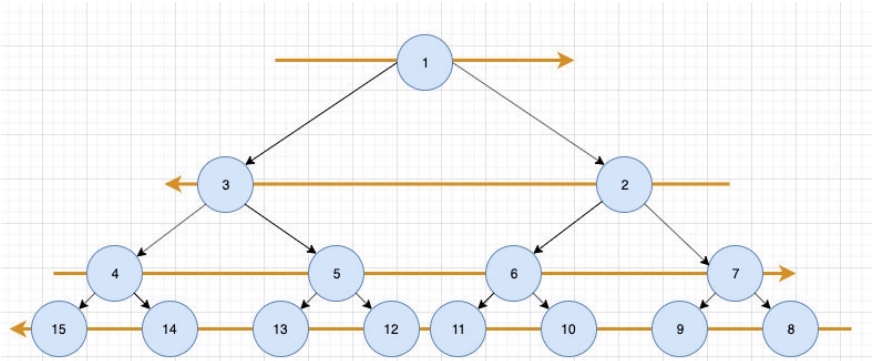
公司

- 暂无

思路

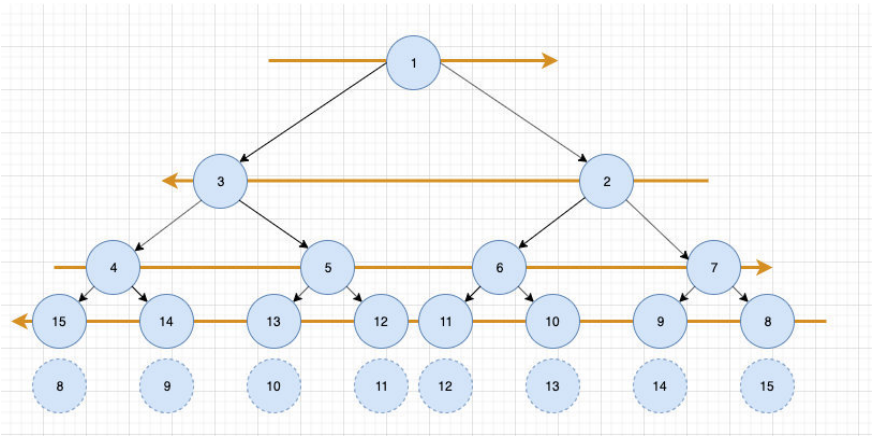
假如这道题不是之字形，那么就会非常简单。 我们可以根据子节点的 label 轻松地求出父节点的 label，公示是 label // 2（其中 label 为子节点的 label）。

如果是这样的话，这道题应该是 easy 难度，代码也不难写出。我们继续考虑之字形。我们不妨先观察一下，找下规律。

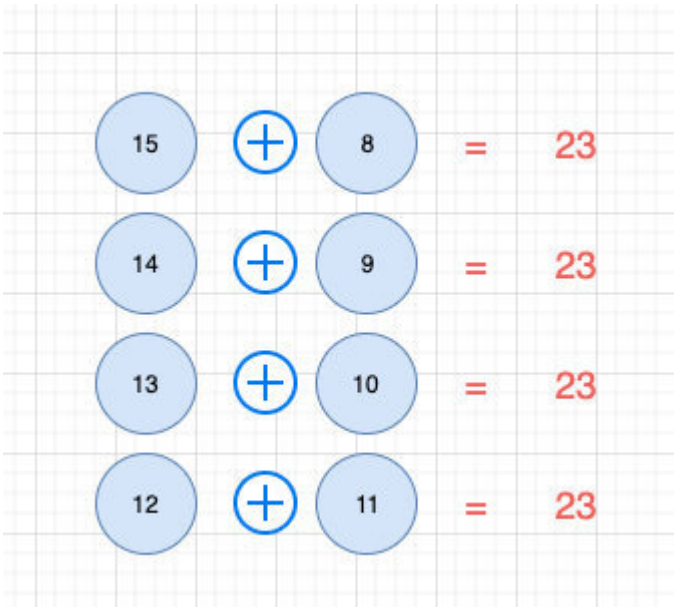


以上图最后一行为例，对于 15 节点，之字变换之前对应的应该是 8 节点。14 节点对应的是 9 节点。。。

全部列举出来是这样的：



我们发现之字变换前后的 label 相加是一个定值。



因此实际上只需要求解出每一层的这个定值，然后减去当前值就好了。（注意我们不需要区分偶数行和奇数行） 问题的关键转化为求解这个定值，这个定值其实很好求，因为每一层的最大值和最小值我们很容易求，而最大值和最小值的和正是我们要求的这个数字。

最大值和最小值这么求呢？由满二叉树的性质，我们知道每一层的最小值就是 `2 ** (level - 1)` ，而最大值是 `2 ** level - 1` 。因此我们只要知道 level 即可，level 非常容易求出，具体可以看下面代码。

关键点

- 满二叉树的性质：
- 最小值是 `2 ** (level - 1)` ，最大值是 `2 ** level - 1` ，其中 level 是树的深度。
- 假如父节点的索引为 i，那么左子节点就是 2*i， 右边子节点就是 2*i + 1。
- 假如子节点的索引是 i，那么父节点的索引就是 i // 2。
- 先思考一般情况（不是之字形）， 然后通过观察找出规律

代码

```python
class Solution:
    def pathInZigZagTree(self, label: int) -> List[int]:
        level = 0
        res = []
        while 2 ** level - 1 < label:
            level += 1

        while level > 0:
            res.insert(0, label)
            label = 2 ** (level - 1) + 2 ** level - 1 - lab
            label //= 2
            level -= 1
        return res
```

复杂度分析

- 时间复杂度：由于每次都在头部插入 res，因此时间复杂度为 $O(log_Label)$， 一共插入了 $O(log_Label)$ 次， 因此总的时间复杂度为 $O(logLabel * logLabel)$。
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1131. 绝对值表达式的最大值）

https://leetcode-cn.com/problems/maximum-of-absolute-value-expression/

题目描述

```
给你两个长度相等的整数数组，返回下面表达式的最大值：

|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|

其中下标 i，j 满足 0 <= i, j < arr1.length。

示例 1：

输入：arr1 = [1,2,3,4], arr2 = [-1,4,5,6]
输出：13
示例 2：

输入：arr1 = [1,-2,-5,0,10], arr2 = [0,-2,-1,-7,-4]
输出：20

提示：

2 <= arr1.length == arr2.length <= 40000
-10^6 <= arr1[i], arr2[i] <= 10^6
```

前置知识

- 数组

解法一（数学分析）

公司

- 阿里
- 腾讯
- 字节

思路

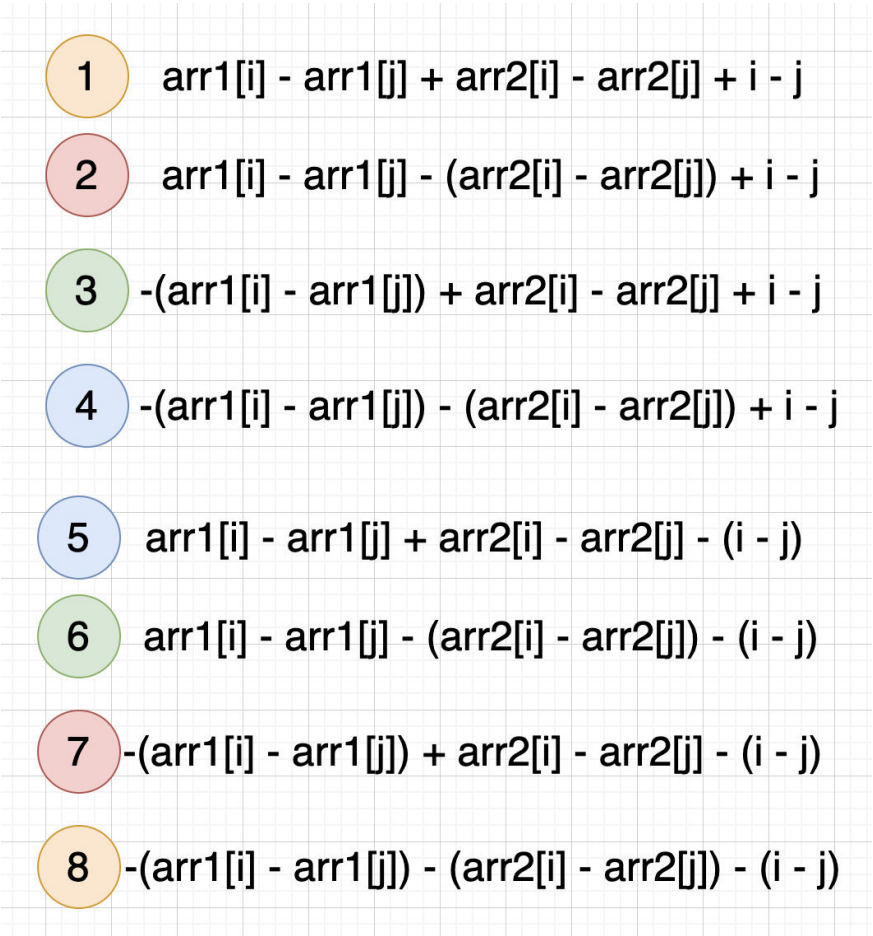
如图我们要求的是这样一个表达式的最大值。arr1 和 arr2 为两个不同的数组，且二者长度相同。i 和 j 是两个合法的索引。

> 红色竖线表示的是绝对值的符号

$$|arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|$$

我们对其进行分类讨论，有如下八种情况：

> |arr1[i] -arr1[j]| 两种情况 |arr2[i] -arr2[j]| 两种情况 |i - j| 两种情况 因此一共是 2 * 2 * 2 = 8 种

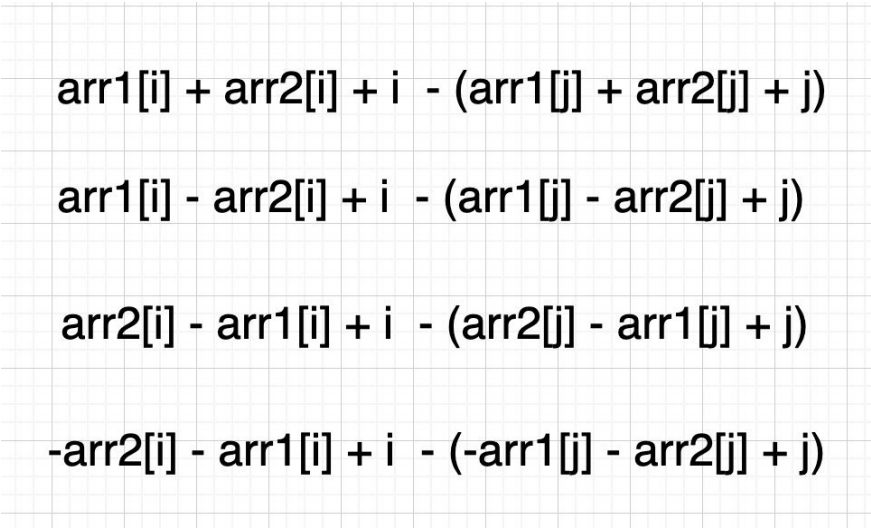


由于 i 和 j 之前没有大小关系，也就说二者可以相互替代。因此：

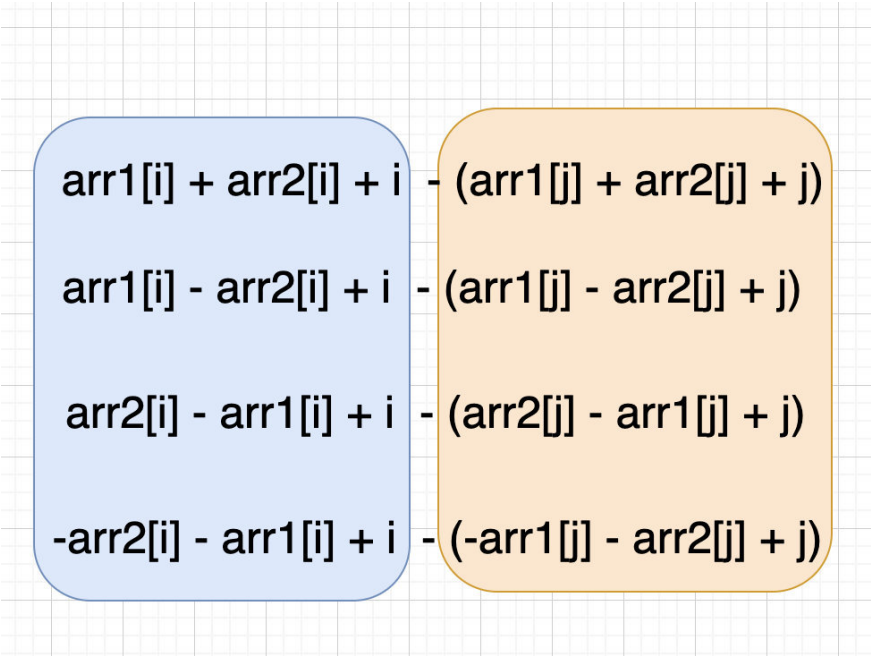
- 1 等价于 8
- 2 等价于 7
- 3 等价于 6
- 4 等价于 5

也就是说我们只需要计算 1，2，3，4 的最大值就可以了。（当然你可以选择其他组合，只要完备就行）

为了方便，我们将 i 和 j 都提取到一起：



容易看出等式的最大值就是前面的最大值，和后面最小值的差值。如图：



再仔细观察，会发现前面部分和后面部分是一样的，原因还是上面所说的 i 和 j 可以互换。因此我们要做的就是：

- 遍历一遍数组，然后计算四个表达式， arr1[i] + arr2[i] + i，arr1[i] - arr2[i] + i，arr2[i] - arr1[i] + i 和 -1 * arr2[i] - arr1[i] + i 的 最大值和最小值。
- 然后分别取出四个表达式最大值和最小值的差值（就是这个表达式的最大值）
- 四个表达式最大值再取出最大值

关键点

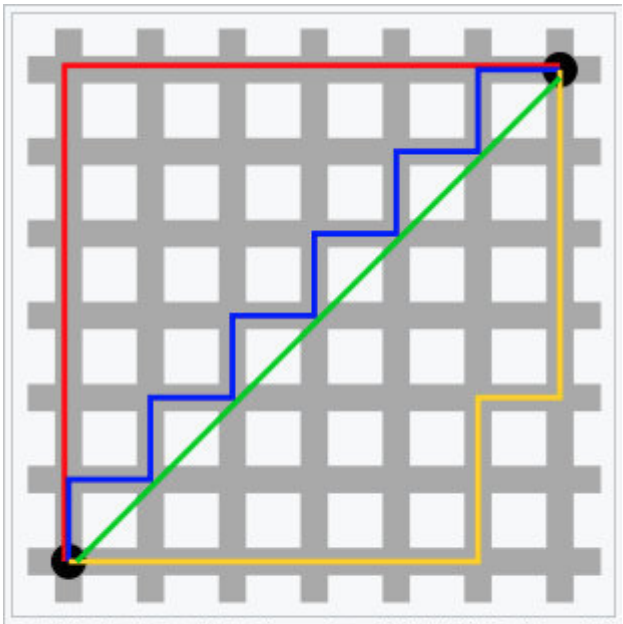
- 数学分析

代码

```
class Solution:
    def maxAbsValExpr(self, arr1: List[int], arr2: List[int
        A = []
        B = []
        C = []
        D = []
        for i in range(len(arr1)):
            a, b, c, d = arr1[i] + arr2[i] + i, arr1[i] - a
                i, arr2[i] - arr1[i] + i, -1 * arr2[i] - ar
            A.append(a)
            B.append(b)
            C.append(c)
            D.append(d)
        return max(max(A) - min(A), max(B) - min(B), max(C)
```

解法二（曼哈顿距离）

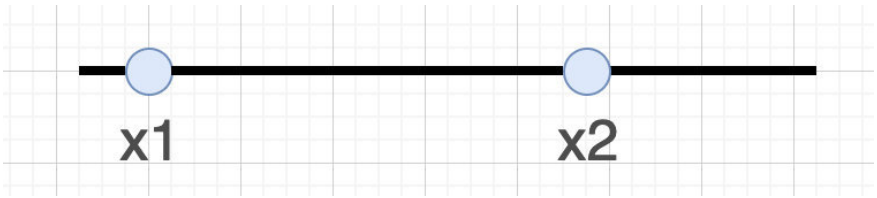
思路



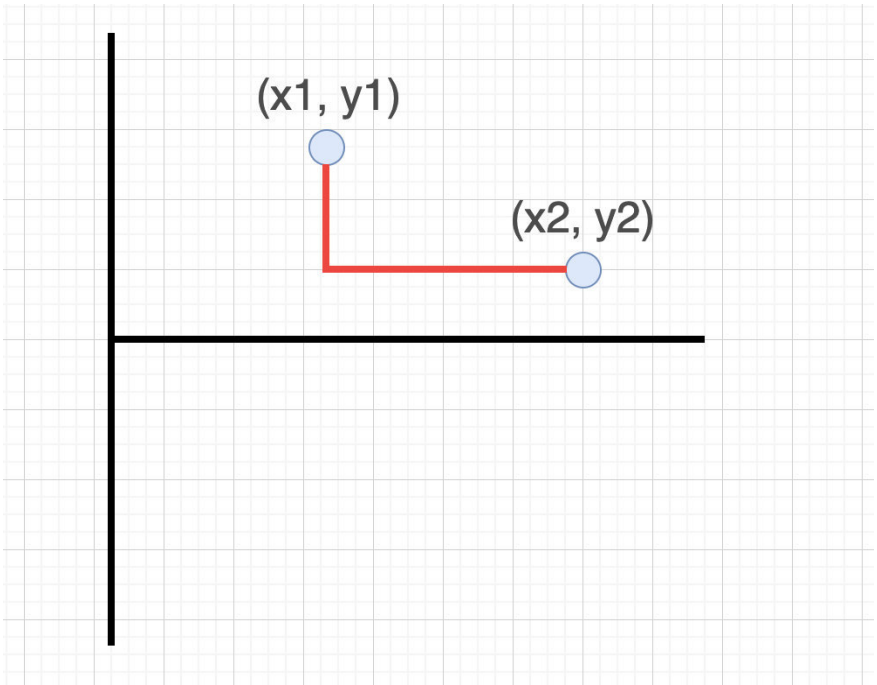
曼哈顿与欧几里得距离：红、蓝与黄线分别表示所有曼哈顿距离都拥有一样长度（12），而绿线表示欧几里得距离有$6\times\sqrt{2}\approx 8.48$的长度。

（图来自：https://zh.wikipedia.org/wiki/%E6%9B%BC%E5%93%88%E9%A0%93%E8%B7%9D%E9%9B%A2）

一维曼哈顿距离可以理解为一条线上两点之间的距离: |x1 - x2|，其值为 max(x1 - x2, x2 - x1)



在平面上，坐标（x1, y1）的点 P1 与坐标（x2, y2）的点 P2 的曼哈顿距离为：|x1-x2| + |y1 - y2|,其值为 max(x1 - x2 + y1 - y2, x2 - x1 + y1 - y2, x1 - x2 + y2 - y1, x2 -x1 + y2 - y1)



然后这道题目是更复杂的三维曼哈顿距离，其中(i, arr[i], arr[j])可以看作三位空间中的一个点，问题转化为曼哈顿距离最远的两个点的距离。 延续上面的思路，|x1-x2| + |y1 - y2| + |z1 - z2|,其值为 :

max(

x1 - x2 + y1 - y2 + z1 - z2,

x1 - x2 + y1 - y2 + z2 - z1,

x2 - x1 + y1 - y2 + z1 - z2,

x2 - x1 + y1 - y2 + z2 - z1,

x1 - x2 + y2 - y1 + z1 - z2,

x1 - x2 + y2 - y1 + z2- z1,

x2 -x1 + y2 - y1 + z1 - z2,

x2 -x1 + y2 - y1 + z2 - z1

)

我们可以将 1 和 2 放在一起方便计算：

max(

x1 + y1 + z1 - (x2 + y2 + z2),

x1 + y1 - z1 - (x2 + y2 - z2)

...

)

我们甚至可以扩展到 n 维，具体代码见下方。

关键点

- 曼哈顿距离
- 曼哈顿距离代码模板

> 解题模板可以帮助你快速并且更少错误的解题，更多解题模板请期待我的新书(未完成)

代码

```
class Solution:
    def maxAbsValExpr(self, arr1: List[int], arr2: List[int
        # 曼哈顿距离模板代码
        sign = [1, -1]
        n = len(arr1)
        dists = []
        # 三维模板
        for a in sign:
            for b in sign:
                for c in sign:
                    maxDist = float('-inf')
                    minDist = float('inf')
                    # 分别计算所有点的曼哈顿距离
                    for i in range(n):
                        dist = arr1[i] * a + arr2[i] * b +
                        maxDist = max(maxDist, dist)
                        minDist = min(minDist, dist)
                    # 将所有的点的曼哈顿距离放到dists中
                    dists.append(maxDist - minDist)
        return max(dists)
```

复杂度分析

- 时间复杂度：$O(N^3)$
- 空间复杂度：$O(N)$

总结

可以看出其实两种解法都是一样的，只是思考角度不一样。

相关题目

- 1030. 距离顺序排列矩阵单元格

- 1162. 地图分析

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(1186. 删除一次得到子数组最大和)

https://leetcode-cn.com/problems/maximum-subarray-sum-with-one-deletion/

题目描述

```
给你一个整数数组，返回它的某个 非空 子数组（连续元素）在执行一次可选的

换句话说，你可以从原数组中选出一个子数组，并可以决定要不要从中删除一个

注意，删除一个元素后，子数组 不能为空。

请看示例：

示例 1：

输入：arr = [1,-2,0,3]
输出：4
解释：我们可以选出 [1, -2, 0, 3]，然后删掉 -2，这样得到 [1, 0, 3]
示例 2：

输入：arr = [1,-2,-2,3]
输出：3
解释：我们直接选出 [3]，这就是最大和。
示例 3：

输入：arr = [-1,-1,-1,-1]
输出：-1
解释：最后得到的子数组不能为空，所以我们不能选择 [-1] 并从中删去 -1
     我们应该直接选择 [-1]，或者选择 [-1, -1] 再从中删去一个 -1。


提示：

1 <= arr.length <= 10^5
-10^4 <= arr[i] <= 10^4
```

前置知识

- 数组
- 动态规划

公司

- 字节

思路

暴力法

符合知觉的做法是求出所有的情况，然后取出最大的。 我们只需要两层循环接口，外循环用于确定我们丢弃的元素，内循环用于计算 subArraySum。

```
class Solution:
  def maximumSum(self, arr: List[int]) -> int:
      res = arr[0]
      def maxSubSum(arr, skip):
          res = maxSub = float("-inf")

          for i in range(len(arr)):
              if i == skip:
                  continue
              maxSub = max(arr[i], maxSub + arr[i])
              res = max(res, maxSub)
          return res
      # 这里循环到了len(arr)项，表示的是一个都不删除的情况
      for i in range(len(arr) + 1):
          res = max(res, maxSubSum(arr, i))
      return res
```

空间换时间

上面的做法在 LC 上会 TLE， 因此我们需要换一种思路，既然超时了，我们是否可以从空间换时间的角度思考呢？我们可以分别从头尾遍历，建立两个 subArraySub 的数组 l 和 r。 其实这个不难想到，很多题目都用到了这个技巧。

具体做法：

- 一层遍历， 建立 l 数组，l[i]表示从左边开始的以 arr[i]结尾的 subArraySum 的最大值
- 一层遍历， 建立 r 数组，r[i]表示从右边开始的以 arr[i]结尾的 subArraySum 的最大值
- 一层遍历， 计算 l[i - 1] + r[i + 1] 的最大值

  > l[i - 1] + r[i + 1]的含义就是删除 arr[i]的子数组最大值

- 上面的这个步骤得到了删除一个的子数组最大值， 不删除的只需要在上面循环顺便计算一下即可

```python
class Solution:
    def maximumSum(self, arr: List[int]) -> int:
        n = len(arr)
        l = [arr[0]] * n
        r = [arr[n - 1]] * n
        if n == 1:
            return arr[0]
        res = arr[0]
        for i in range(1, n):
            l[i] = max(l[i - 1] + arr[i], arr[i])
            res = max(res, l[i])
        for i in range(n - 2, -1, -1):
            r[i] = max(r[i + 1] + arr[i], arr[i])
            res = max(res, r[i])
        for i in range(1, n - 1):
            res = max(res, l[i - 1] + r[i + 1])

        return res
```

动态规划

上面的算法虽然时间上有所改善，但是正如标题所说，空间复杂度是O(n),有没有办法改进呢？答案是使用动态规划。

具体过程：

- 定义 max0，表示以 arr[i]结尾且一个都不漏的最大子数组和
- 定义 max1，表示以 arr[i]或者 arr[i - 1]结尾，可以漏一个的最大子数组和
- 遍历数组，更新 max1 和 max0（注意先更新 max1，因为 max1 用到了上一个 max0）
- 其中 `max1 = max(max1 + arr[i], max0)` , 即删除 arr[i - 1]或者删除 arr[i]
- 其中 `max0 = max(max0 + arr[i], arr[i])` ， 一个都不删除

```python
#
# @lc app=leetcode.cn id=1186 lang=python3
#
# [1186] 删除一次得到子数组最大和
#

# @lc code=start


class Solution:
    def maximumSum(self, arr: List[int]) -> int:
        # DP
        max0 = arr[0]
        max1 = arr[0]
        res = arr[0]
        n = len(arr)
        if n == 1:
            return max0

        for i in range(1, n):
            # 先更新max1，再更新max0，因为max1用到了上一个max0
            max1 = max(max1 + arr[i], max0)
            max0 = max(max0 + arr[i], arr[i])
            res = max(res, max0, max1)
        return res
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

关键点解析

- 空间换时间
- 头尾双数组
- 动态规划

相关题目

- 42.trapping-rain-water

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(1218. 最长定差子序列)

https://leetcode-cn.com/problems/longest-arithmetic-subsequence-of-given-difference/

题目描述

```
给你一个整数数组 arr 和一个整数 difference，请你找出 arr 中所有相等

示例 1：

输入：arr = [1,2,3,4], difference = 1
输出：4
解释：最长的等差子序列是 [1,2,3,4]。
示例 2：

输入：arr = [1,3,5,7], difference = 1
输出：1
解释：最长的等差子序列是任意单个元素。
示例 3：

输入：arr = [1,5,7,8,5,3,4,2,1], difference = -2
输出：4
解释：最长的等差子序列是 [7,5,3,1]。


提示：

1 <= arr.length <= 10^5
-10^4 <= arr[i], difference <= 10^4
```

前置知识

- 数组
- 动态规划

公司

- 腾讯

思路

最直观的思路是双层循环，我们暴力的枚举出以每一个元素为开始元素，以最后元素结尾的的所有情况。很明显这是所有的情况，这就是暴力法的精髓， 很明显这种解法会 TLE（超时），不过我们先来看一下代码，顺着这个思维继续思考。

暴力法

```
    def longestSubsequence(self, arr: List[int], difference:
        n = len(arr)
        res = 1
        for i in range(n):
            count = 1
            for j in range(i + 1, n):
                if arr[i] + difference * count == arr[j]:
                    count += 1

                if count > res:
                    res = count

        return res
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

动态规划

上面的时间复杂度是 O(n^2)， 有没有办法降低到 O(n)呢？很容易想到的是空间换时间的解决方案。

我的想法是将 `以每一个元素结尾的最长等差子序列的长度` 统统存起来，即 `dp[num] = maxLen` 这样我们遍历到一个新的元素的时候，就去之前的存储中去找 `dp[num - difference]` , 如果找到了，就更新当前的 `dp[num] = dp[num - difference] + 1` , 否则就是不进行操作（还是默认值 1）。

这种空间换时间的做法的时间和空间复杂度都是 O(n)。

关键点解析

- 将 `以每一个元素结尾的最长等差子序列的长度` 统统存起来

代码

```python
#
# @lc app=leetcode.cn id=1218 lang=python3
#
# [1218] 最长定差子序列
#

# @lc code=start


class Solution:

    # 动态规划
    def longestSubsequence(self, arr: List[int], difference
        n = len(arr)
        res = 1
        dp = {}
        for num in arr:
            dp[num] = 1
            if num - difference in dp:
                dp[num] = dp[num - difference] + 1

        return max(dp.values())

# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1227. 飞机座位分配概率）

https://leetcode-cn.com/problems/airplane-seat-assignment-probability/

题目描述

```
有 n 位乘客即将登机，飞机正好有 n 个座位。第一位乘客的票丢了，他随便选

剩下的乘客将会：

如果他们自己的座位还空着，就坐到自己的座位上，

当他们自己的座位被占用时，随机选择其他座位
第 n 位乘客坐在自己的座位上的概率是多少？



示例 1：

输入：n = 1
输出：1.00000
解释：第一个人只会坐在自己的位置上。
示例 2：

输入：n = 2
输出：0.50000
解释：在第一个人选好座位坐下后，第二个人坐在自己的座位上的概率是 0.5。


提示：

1 <= n <= 10^5
```

前置知识

- 记忆化搜索
- 动态规划

暴力递归

这是一道 LeetCode 为数不多的概率题，我们来看下。

公司

- 字节

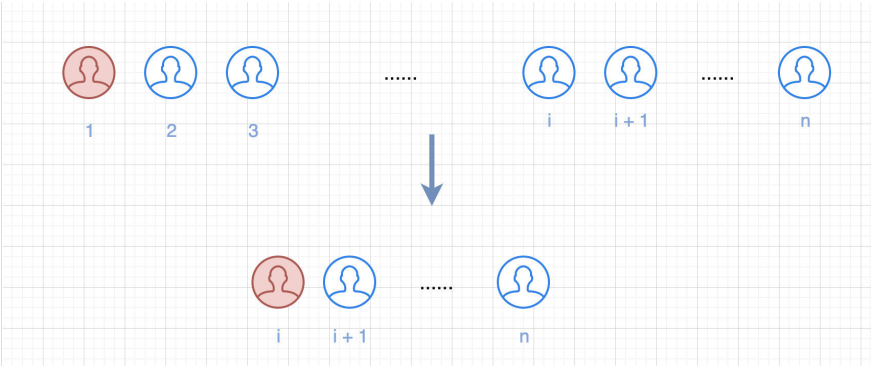
思路

我们定义原问题为 f(n)。对于第一个人来说，他有 n 中选择，就是分别选择 n 个座位中的一个。由于选择每个位置的概率是相同的，那么选择每个位置的概率应该都是 1 / n。

我们分三种情况来讨论：

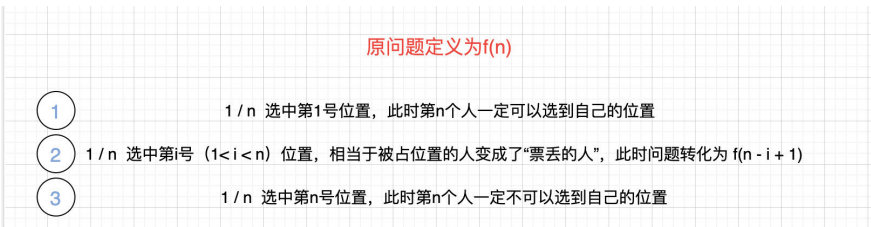
- 如果第一个人选择了第一个人的位置（也就是选择了自己的位置），那么剩下的人按照票上的座位做就好了，这种情况第 n 个人一定能做到自己的位置
- 如果第一个人选择了第 n 个人的位置，那么第 n 个人肯定坐不到自己的位置。
- 如果第一个人选择了第 i (1 < i < n)个人的位置，那么第 i 个人就相当于变成了“票丢的人”，此时问题转化为 f(n - i + 1)。

此时的问题转化关系如图：



（红色表示票丢的人）

整个过程分析：



代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5
        res = 1 / n
        for i in range(2, n):
            res += self.nthPersonGetsNthSeat(n - i + 1) * 1
        return res
```

上述代码会栈溢出。

暴力递归 + hashtable

思路

我们考虑使用记忆化递归来减少重复计算，虽然这种做法可以减少运行时间，但是对减少递归深度没有帮助。还是会栈溢出。

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    seen = {}

    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5
        if n in self.seen:
            return self.seen[n]
        res = 1 / n
        for i in range(2, n):
            res += self.nthPersonGetsNthSeat(n - i + 1) * 1
        self.seen[n] = res
        return res
```

动态规划

思路

上面做法会栈溢出。其实我们根本不需要运行就应该能判断出栈溢出，题目已经给了数据规模是 1 <= n <= 10 ** 5。 这个量级不管什么语言，除非使用尾递归，不然一般都会栈溢出，具体栈深度大家可以查阅相关资料。

既然是栈溢出，那么我们考虑使用迭代来完成。 很容易想到使用动态规划来完成。其实递归都写出来，写一个朴素版的动态规划也难不到哪去，毕竟动态规划就是记录子问题，并建立子问题之间映射而已，这和递归并无本质区别。

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5

        dp = [1, .5] * n

        for i in range(2, n):
            dp[i] = 1 / n
            for j in range(2, i):
                dp[i] += dp[i - j + 1] * 1 / n
        return dp[-1]
```

这种思路的代码超时了，并且仅仅执行了 35/100 testcase 就超时了。

数学分析

思路

我们还需要进一步优化时间复杂度，我们需要思考是否可以在线形的时间内完成。

我们继续前面的思路进行分析, 不难得出，我们不妨称其为等式 1：

```
f(n)
= 1/n + 0 + 1/n * (f(n-1) + f(n-2) + ... + f(2))
= 1/n * (f(n-1) + f(n-2) + ... + f(2) + 1)
= 1/n * (f(n-1) + f(n-2) + ... + f(2) + f(1))
```

似乎更复杂了？没关系，我们继续往下看，我们看下 f(n - 1)，我们不妨称其为等式 2。

```
f(n-1) = 1/(n-1) * (f(n-2) + f(n-3) + ... + f(1))
```

我们将等式 1 和等式 2 两边分别同时乘以 n 和 n - 1。

```
n * f(n) = f(n-1) + f(n-2) + f(n-3) + ... + f(1)
(n-1) * f(n-1) = f(n-2) + f(n-3) + ... + f(1)
```

之后我们使用错位相减技巧可以将等式进一步花间。

具体来说我们可以将两者相减：

```
n * f(n) - (n-1)*f(n-1) = f(n-1)
```

我们继续将 (n-1)*f(n-1) 移到等式右边，得到：

```
n * f(n) = n * f(n-1)
```

也就是说:

```
f(n) = f(n - 1)
```

当然前提是 n 大于 2。

既然如此，我们就可以减少一层循环， 我们用这个思路来优化一下上面的 dp 解法。这种解法终于可以 AC 了。

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        if n == 1:
            return 1
        if n == 2:
            return 0.5

        dp = [1, .5] * n

        for i in range(2, n):
            dp[i] = 1/n+(n-2)/n * dp[n-1]
        return dp[-1]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

优化数学分析

思路

上面我们通过数学分析，得出了当 n 大于 2 时：

```
f(n) = f(n - 1)
```

那么是不是意味着我们随便求出一个 n 就好了？ 比如我们求出 n = 2 的时候的值，是不是就知道 n 为任意数的值了。 我们不难想出 n = 2 时候，概率是 0.5，因此只要 n 大于 1 就是 0.5 概率，否则就是 1 概率。

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def nthPersonGetsNthSeat(self, n: int) -> float:
        return 1 if n == 1 else .5
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

关键点

- 概率分析
- 数学推导
- 动态规划
- 递归 + mapper
- 栈限制大小
- 尾递归

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1261. 在受污染的二叉树中查找元素）

https://leetcode-cn.com/problems/find-elements-in-a-contaminated-binary-tree/

题目描述

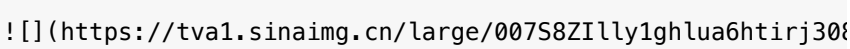
给出一个满足下述规则的二叉树：

root.val == 0
如果 treeNode.val == x 且 treeNode.left != null，那么 treeNo
如果 treeNode.val == x 且 treeNode.right != null，那么 treeN
现在这个二叉树受到「污染」，所有的 treeNode.val 都变成了 -1。

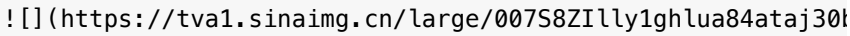
请你先还原二叉树，然后实现 FindElements 类：

FindElements(TreeNode* root) 用受污染的二叉树初始化对象，你需要先
bool find(int target) 判断目标值 target 是否存在于还原后的二叉树

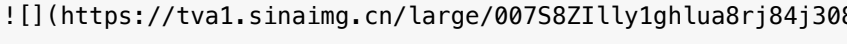
示例 1：



输入：
["FindElements","find","find"]
[[[-1,null,-1]],[1],[2]]
输出：
[null,false,true]
解释：
FindElements findElements = new FindElements([-1,null,-1]);
findElements.find(1); // return False
findElements.find(2); // return True
示例 2：



输入：
["FindElements","find","find","find"]
[[[-1,-1,-1,-1,-1]],[1],[3],[5]]
输出：
[null,true,true,false]
解释：
FindElements findElements = new FindElements([-1,-1,-1,-1,-
findElements.find(1); // return True
findElements.find(3); // return True
findElements.find(5); // return False
示例 3：



输入：
["FindElements","find","find","find","find"]
[[[-1,null,-1,-1,null,-1]],[2],[3],[4],[5]]
输出：

```
[null,true,false,false,true]
解释:
FindElements findElements = new FindElements([-1,null,-1,-1
findElements.find(2); // return True
findElements.find(3); // return False
findElements.find(4); // return False
findElements.find(5); // return True


提示:

TreeNode.val == -1
二叉树的高度不超过 20
节点的总数在 [1, 10^4] 之间
调用 find() 的总次数在 [1, 10^4] 之间
0 <= target <= 10^6
```

前置知识

- 二进制

暴力法

公司

- 暂无

思路

最简单想法就是递归建立树，然后 find 的时候递归查找即可，代码也很简单。

代码

Pythpn Code:

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class FindElements:
    node = None
    def __init__(self, root: TreeNode):
        def recover(node):
            if not node:
                return node;
            if node.left:
                node.left.val =  2 * node.val + 1
            if node.right:
                 node.right.val = 2 * node.val + 2
            recover(node.left)
            recover(node.right)
            return node
        root.val = 0
        self.node = recover(root)


    def find(self, target: int) -> bool:
        def findInTree(node, target):
            if not node:
                return False
            if node.val == target:
                return True
            return findInTree(node.left, target) or findIn
        return findInTree(self.node, target)




# Your FindElements object will be instantiated and called
# obj = FindElements(root)
# param_1 = obj.find(target)
```

上述代码会超时，我们来考虑优化。

空间换时间

思路

上述代码会超时，我们考虑使用空间换时间。 建立树的时候，我们将所有值存到一个集合中去。当需要 find 的时候，我们直接查找 set 即可，时间复杂度 O(1)。

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class FindElements:
    def __init__(self, root: TreeNode):
        # set 不能放在init外侧。 因为测试用例之间不会销毁FindElem
        self.seen = set()
        def recover(node):
            if not node:
                return node;
            if node.left:
                node.left.val =  2 * node.val + 1
                self.seen.add(node.left.val)
            if node.right:
                 node.right.val = 2 * node.val + 2
                 self.seen.add(node.right.val)
            recover(node.left)
            recover(node.right)
            return node
        root.val = 0
        self.seen.add(0)
        self.node = recover(root)


    def find(self, target: int) -> bool:
        return target in self.seen




# Your FindElements object will be instantiated and called
# obj = FindElements(root)
# param_1 = obj.find(target)
```

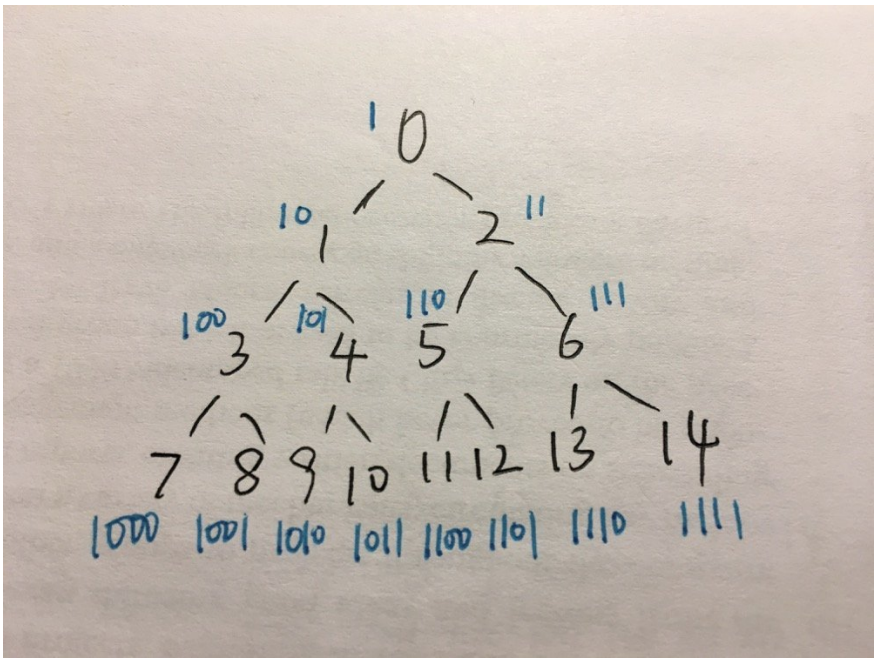
这种解法可以 AC，但是在数据量非常大的时候，可能 MLE，我们继续考虑优化。

二进制法

思路

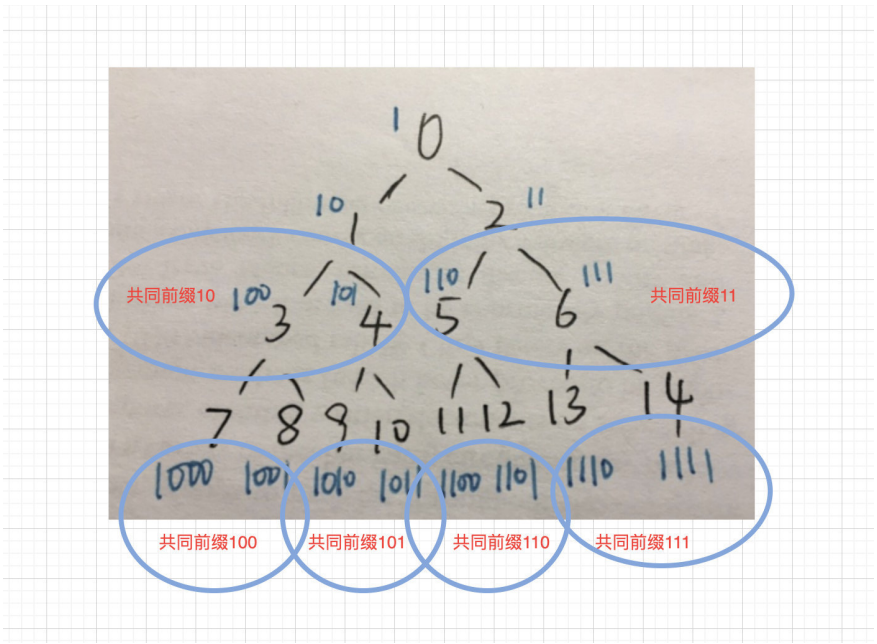
这是一种非常巧妙的做法。

如果我们把树中的数全部加 1 会怎么样?



（图参考 https://leetcode.com/problems/find-elements-in-a-contaminated-binary-tree/discuss/431229/Python-Special-Way-for-find()-without-HashSet-O(1)-Space-O(logn)-Time）

仔细观察发现，每一行的左右子树分别有不同的前缀：



Ok，那么算法就来了。为了便于理解，我们来举个具体的例子，比如 target 是 9，我们首先将其加 1，二进制表示就是 1010。不考虑第一位，就是 010，我们只要：

- 0 向左 👈
- 1 向右 👉
-
    - 0 向左 👈

就可以找到 9 了。

> 0 表示向左， 1 表示向右

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class FindElements:
    node = None
    def __init__(self, root: TreeNode):
        def recover(node):
            if not node:
                return node;
            if node.left:
                node.left.val =  2 * node.val + 1
            if node.right:
                 node.right.val = 2 * node.val + 2
            recover(node.left)
            recover(node.right)
            return node
        root.val = 0
        self.node = recover(root)


    def find(self, target: int) -> bool:
        node = self.node
        for bit in bin(target+1)[3:]:
            node = node and (node.left, node.right)[int(bit
        return bool(node)




# Your FindElements object will be instantiated and called
# obj = FindElements(root)
# param_1 = obj.find(target)
```

关键点解析

- 空间换时间
- 二进制思维
- 将 target + 1

更多题解可以访问我的LeetCode题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经37K star啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（1262. 可被三整除的最大和）

https://leetcode-cn.com/problems/greatest-sum-divisible-by-three/

题目描述

```
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
```

前置知识

- 数组
- 回溯法
- 排序

暴力法

公司

- 字节
- 网易有道

思路

一种方式是找出所有的能够被 3 整除的子集，然后挑选出和最大的。由于我们选出了所有的子集，那么时间复杂度就是 $O(2^N)$ ， 毫无疑问会超时。这里我们使用回溯法找子集，如果不清楚回溯法，可以参考我之前的题解，很多题目都用到了，比如78.subsets。

更多回溯题目，可以访问上方链接查看（可以使用一套模板搞定）：

- 39.combination-sum
- 40.combination-sum-ii
- 46.permutations
- 47.permutations-ii
- 90.subsets-ii
- 113.path-sum-ii
- 131.palindrome-partitioning

代码

```
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        self.res = 0
        def backtrack(temp, start):
            total = sum(temp)
            if total % 3 == 0:
                self.res = max(self.res, total)
            for i in range(start, len(nums)):
                temp.append(nums[i])
                backtrack(temp, i + 1)
                temp.pop(-1)


        backtrack([], 0)

        return self.res
```

减法 + 排序

减法的核心思想是，我们求出总和。如果总和不满足题意，我们尝试减去最小的数，使之满足题意。

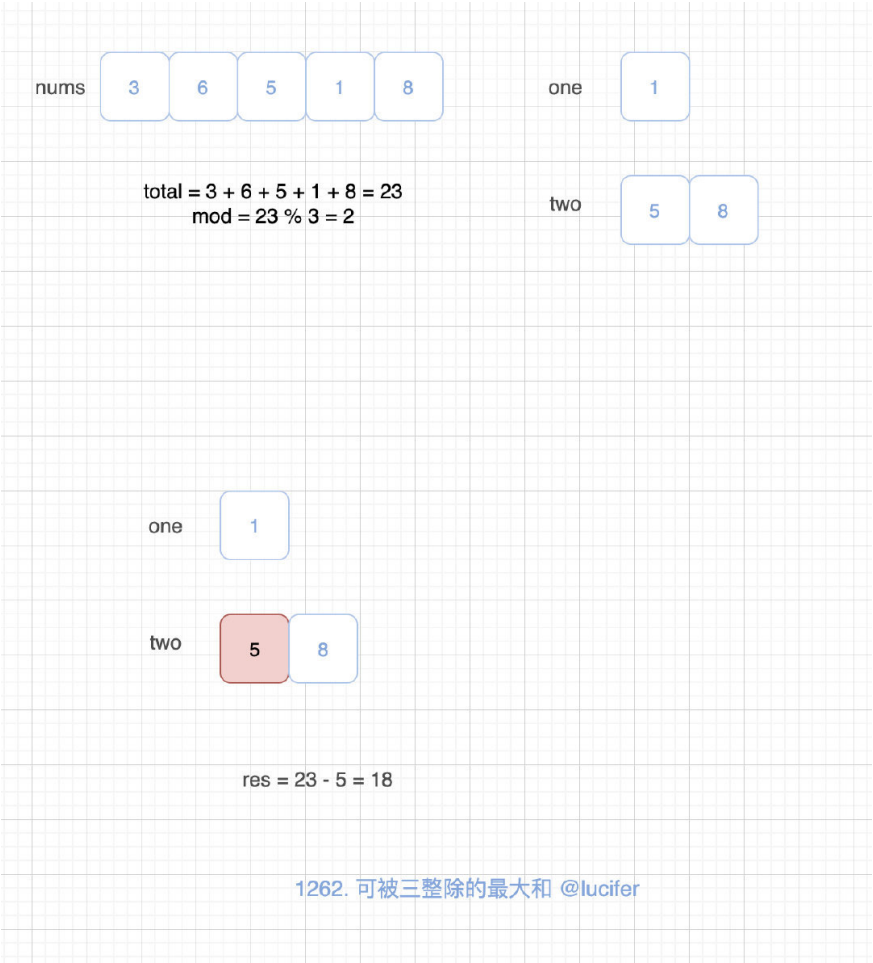
思路

这种算法的思想，具体来说就是：

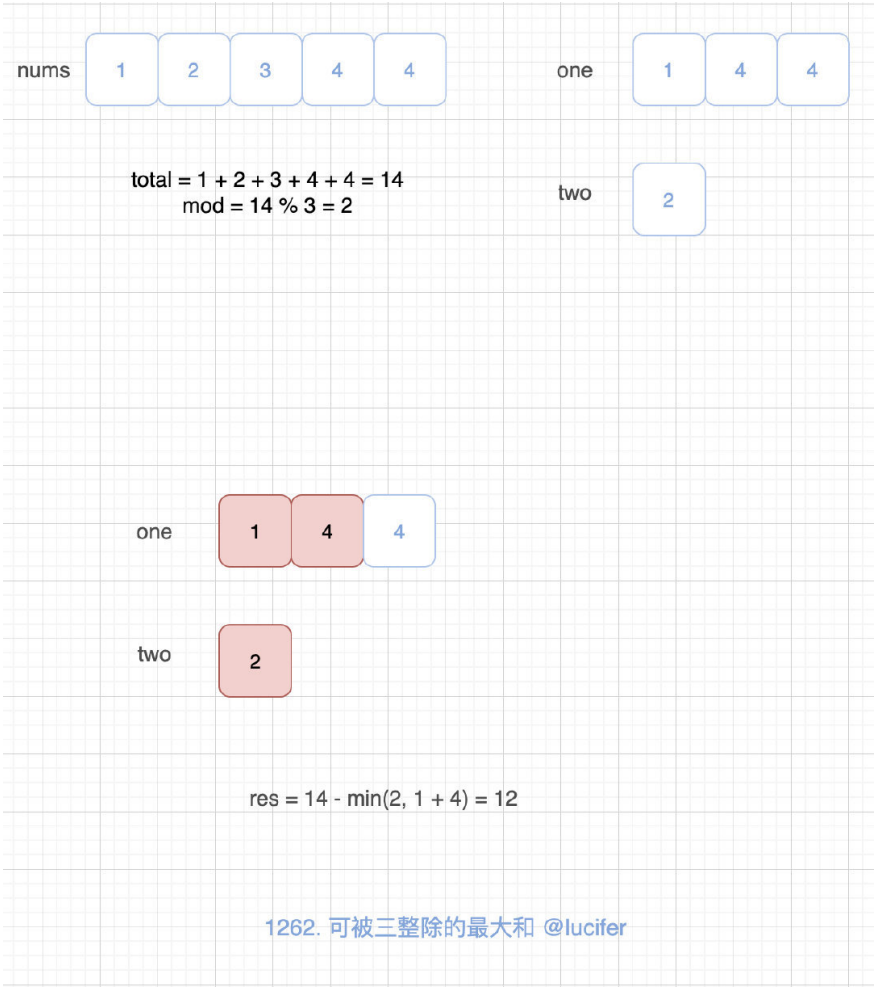
- 我们将所有的数字加起来，我们不妨设为 total
- total 除以 3，得到一个余数 mod， mod 可能值有 0，1，2.
- 同时我们建立两个数组，一个是余数为 1 的数组 one，一个是余数为 2 的数组 two
- 如果 mod 为 0，我们直接返回即可。
- 如果 mod 为 1，我们可以减去 one 数组中最小的一个（如果有的话），或者减去两个 two 数组中最小的（如果有的话），究竟减去谁取决谁更小。
- 如果 mod 为 2，我们可以减去 two 数组中最小的一个（如果有的话），或者减去两个 one 数组中最小的（如果有的话），究竟减去谁取决谁更小。

由于我们需要取 one 和 two 中最小的一个或者两个，因此对数组 one 和 two 进行排序是可行的，如果基于排序的话，时间复杂度大致为 $O(NlogN)$，这种算法可以通过。

以题目中的例 1 为例：


nums
3
6
5
1
8
one
1
total = 3 + 6 + 5 + 1 + 8 = 23
mod = 23 % 3 = 2
two
5
8
one
1
two
5
8
res = 23 - 5 = 18
1262. 可被三整除的最大和 @lucifer

以题目中的例 2 为例：


nums
1
2
3
4
4
one
1
4
4
total = 1 + 2 + 3 + 4 + 4 = 14
mod = 14 % 3 = 2
two
2
one
1
4
4
two
2
res = 14 - min(2, 1 + 4) = 12
1262. 可被三整除的最大和 @lucifer

代码

```python
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        one = []
        two = []
        total = 0

        for num in nums:
            total += num
            if num % 3 == 1:
                one.append(num)
            if num % 3 == 2:
                two.append(num)
        one.sort()
        two.sort()
        if total % 3 == 0:
            return total
        elif total % 3 == 1 and one:
            if len(two) >= 2 and one[0] > two[0] + two[1]:
                return total - two[0] - two[1]
            return total - one[0]
        elif total % 3 == 2 and two:
            if len(one) >= 2 and two[0] > one[0] + one[1]:
                return total - one[0] - one[1]
            return total - two[0]
        return 0
```

减法 + 非排序

思路

上面的解法使用到了排序。 我们其实观察发现，我们只是用到了 one 和 two 的最小的两个数。因此我们完全可以在线形的时间和常数的空间完成这个算法。我们只需要分别记录 one 和 two 的最小值和次小值即可，在这里，我使用了两个长度为 2 的数组来表示，第一项是最小值，第二项是次小值。

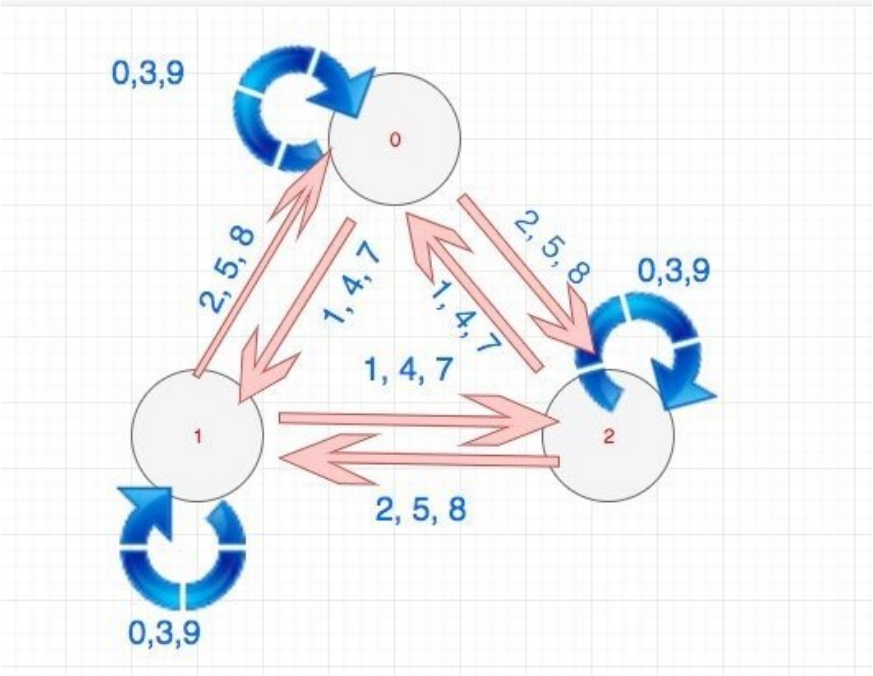
代码

```python
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        one = [float('inf')] * 2
        two = [float('inf')] * 2
        total = 0

        for num in nums:
            total += num
            if num % 3 == 1:
                if num < one[0]:
                    t = one[0]
                    one[0] = num
                    one[1] = t
                elif num < one[1]:
                    one[1] = num
            if num % 3 == 2:
                if num < two[0]:
                    t = two[0]
                    two[0] = num
                    two[1] = t
                elif num < two[1]:
                    two[1] = num
        if total % 3 == 0:
            return total
        elif total % 3 == 1 and one:
            if len(two) >= 2 and one[0] > two[0] + two[1]:
                return total - two[0] - two[1]
            return total - one[0]
        elif total % 3 == 2 and two:
            if len(one) >= 2 and two[0] > one[0] + one[1]:
                return total - one[0] - one[1]
            return total - two[0]
        return 0
```

有限状态机

思路

我在数据结构与算法在前端领域的应用 - 第二篇 中讲到了有限状态机。



状态机表示若干个状态以及在这些状态之间的转移和动作等行为的数学模型。通俗的描述状态机就是定义了一套状态变更的流程：状态机包含一个状态集合，定义当状态机处于某一个状态的时候它所能接收的事件以及可执行的行为，执行完成后，状态机所处的状态。

状态机使用非常广泛，比如正则表达式的引擎，编译器的词法和语法分析，网络协议，企业应用等很多领域都会用到。

拿本题中来说，我们从左到右扫描数组的过程，将会不断改变状态机的状态。

我们使用 state 数组来表示本题的状态：

- state[0] 表示 mod 为 0 的 最大和
- state[1] 表示 mod 为 1 的 最大和
- state[2] 表示 mod 为 1 的 最大和

我们的状态转移方程就会很容易。说到状态转移方程，你可能会想到动态规划。没错！这种思路可以直接翻译成动态规划，算法完全一样。如果你看过我上面提到的文章，那么状态转移方程对你来说就会很容易。如果你不清楚，那么请往下看：

- 我们从左往右不断读取数字，我们不妨设这个数字为 num。
- 如果 num % 3 为 0。 那么我们的 state[0], state[1], state[2] 可以直接加上 num（题目限定了 num 为非负）， 因为任何数字加上 3 的倍数之后，mod 3 的值是不变的。
- 如果 num % 3 为 1。 我们知道 state[2] + num 会变成一个能被三整除的数，但是这个数字不一定比当前的 state[0]大。 代码表示就是 `max(state[2] + num, state[0])` 。同理 state[1] 和 state[2] 的转移逻辑类似。

- 同理 num % 3 为 2 也是类似的逻辑。
- 最后我们返回 state[0]即可。

代码

```
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        state = [0, float('-inf'), float('-inf')]

        for num in nums:
            if num % 3 == 0:
                state = [state[0] + num, state[1] + num, st
            if num % 3 == 1:
                a = max(state[2] + num, state[0])
                b = max(state[0] + num, state[1])
                c = max(state[1] + num, state[2])
                state = [a, b, c]
            if num % 3 == 2:
                a = max(state[1] + num, state[0])
                b = max(state[2] + num, state[1])
                c = max(state[0] + num, state[2])
                state = [a, b, c]
        return state[0]
```

当然这个代码还可以简化：

```
class Solution:
    def maxSumDivThree(self, nums: List[int]) -> int:
        state = [0, float('-inf'), float('-inf')]

        for num in nums:
            temp = [0] * 3
            for i in range(3):
                temp[(i + num) % 3] = max(state[(i + num) %
            state = temp

        return state[0]
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

关键点解析

- 贪婪法
- 状态机
- 数学分析

扩展

实际上，我们可以采取加法（贪婪策略），感兴趣的可以试一下。

另外如果题目改成了 `请你找出并返回能被x整除的元素最大和` ，你只需要将我的解法中的 3 改成 x 即可。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1297. 子串的最大出现次数）

https://leetcode-cn.com/problems/maximum-number-of-occurrences-of-a-substring/

题目描述

```
给你一个字符串 s ，请你返回满足以下条件且出现次数最大的 任意 子串的出

子串中不同字母的数目必须小于等于 maxLetters 。
子串的长度必须大于等于 minSize 且小于等于 maxSize 。


示例 1:

输入: s = "aababcaab", maxLetters = 2, minSize = 3, maxSize
输出: 2
解释: 子串 "aab" 在原字符串中出现了 2 次。
它满足所有的要求: 2 个不同的字母，长度为 3 （在 minSize 和 maxSize
示例 2:

输入: s = "aaaa", maxLetters = 1, minSize = 3, maxSize = 3
输出: 2
解释: 子串 "aaa" 在原字符串中出现了 2 次，且它们有重叠部分。
示例 3:

输入: s = "aabcabcab", maxLetters = 2, minSize = 2, maxSize
输出: 3
示例 4:

输入: s = "abcde", maxLetters = 2, minSize = 3, maxSize = 3
输出: 0


提示:

1 <= s.length <= 10^5
1 <= maxLetters <= 26
1 <= minSize <= maxSize <= min(26, s.length)
s 只包含小写英文字母。
```

前置知识

- 字符串
- 滑动窗口

暴力法

题目给的数据量不是很大，为 1 <= maxLetters <= 26，我们试一下暴力法。

公司

- 字节

思路

暴力法如下：

- 先找出所有满足长度大于等于 minSize 且小于等于 maxSize 的所有子串。（平方的复杂度）
- 对于 maxLetter 满足题意的子串，我们统计其出现次数。时间复杂度为 O(k),其中 k 为子串长度
- 返回最大的出现次数

代码

Pythpn Code:

```
class Solution:
    def maxFreq(self, s: str, maxLetters: int, minSize: int
        n = len(s)
        letters = set()
        cnts = dict()
        res = 0
        for i in range(n - minSize + 1):
            length = minSize
            while i + length <= n and length <= maxSize:
                t = s[i:i + length]
                for c in t:
                    if len(letters) > maxLetters:
                        break
                    letters.add(c)
                if len(letters) <= maxLetters:
                    cnts[t] = cnts.get(t, 0) + 1
                    res = max(res, cnts[t])
                letters.clear()
                length += 1
        return res
```

上述代码会超时。我们来利用剪枝来优化。

剪枝

思路

还是暴力法的思路，不过我们在此基础上进行一些优化。首先我们需要仔细阅读题目，如果你足够细心或者足够有经验，可能会发现其实题目中 maxSize 没有任何用处，属于干扰信息。

也就是说我们没有必要统计 `长度大于等于 minSize 且小于等于 maxSize 的所有子串` ，而是统计长度为 minSize 的所有字串即可。原因是，如果一个大于 minSize 长度的字串若是满足条件，那么该子串其中必定有至少一个长度为 minSize 的字串满足条件。因此一个大于 minSize 长度的字串出现了 n 次，那么该子串其中必定有一个长度为 minSize 的子串出现了 n 次。

代码

代码支持 Python3，Java：

Python Code：

```
    def maxFreq(self, s: str, maxLetters: int, minSize: int, r
            counter, res = {}, 0
            for i in range(0, len(s) - minSize + 1):
                sub = s[i : i + minSize]
                if len(set(sub)) <= maxLetters:
                    counter[sub] = counter.get(sub, 0) + 1
                    res = max(res, counter[sub])
            return res;

# @lc code=end
```

Java Code：

```
    public int maxFreq(String s, int maxLetters, int minSize,
        Map<String, Integer> counter = new HashMap<>();
        int res = 0;
        for (int i = 0; i < s.length() - minSize + 1; i++) {
            String substr = s.substring(i, i + minSize);
            if (checkNum(substr, maxLetters)) {
                int newVal = counter.getOrDefault(substr, 0) +
                counter.put(substr, newVal);
                res = Math.max(res, newVal);
            }
        }
        return res;
    }
    public boolean checkNum(String substr, int maxLetters) {
        Set<Character> set = new HashSet<>();
        for (int i = 0; i < substr.length(); i++)
            set.add(substr.charAt(i));
        return set.size() <= maxLetters;
    }
```

复杂度分析

其中 N 为 s 长度

- 时间复杂度：$O(N * minSize)$
- 空间复杂度：$O(N * minSize)$

关键点解析

- 滑动窗口
- 识别题目干扰信息
- 看题目限制条件，对于本题有用的信息是 `1 <= maxLetters <= 26`

扩展

我们也可以使用滑动窗口来解决，感兴趣的可以试试看。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1310. 子数组异或查询）

https://leetcode-cn.com/problems/xor-queries-of-a-subarray/

题目描述

```
有一个正整数数组 arr，现给你一个对应的查询数组 queries，其中 querie

对于每个查询 i，请你计算从 Li 到 Ri 的 XOR 值（即 arr[Li] xor ar

并返回一个包含给定查询 queries 所有结果的数组。



示例 1：

输入：arr = [1,3,4,8], queries = [[0,1],[1,2],[0,3],[3,3]]
输出：[2,7,14,8]
解释：
数组中元素的二进制表示形式是：
1 = 0001
3 = 0011
4 = 0100
8 = 1000
查询的 XOR 值为：
[0,1] = 1 xor 3 = 2
[1,2] = 3 xor 4 = 7
[0,3] = 1 xor 3 xor 4 xor 8 = 14
[3,3] = 8
示例 2：

输入：arr = [4,8,2,10], queries = [[2,3],[1,3],[0,0],[0,3]]
输出：[8,0,4,4]


提示：

1 <= arr.length <= 3 * 10^4
1 <= arr[i] <= 10^9
1 <= queries.length <= 3 * 10^4
queries[i].length == 2
0 <= queries[i][0] <= queries[i][1] < arr.length
```

前置知识

- 前缀和

公司

- 暂无

暴力法

思路

最直观的思路是双层循环即可，果不其然超时了。

代码

```
class Solution:
    def xorQueries(self, arr: List[int], queries: List[List
         res = []
        for (L, R) in queries:
            i = L
            xor = 0
            while i <= R:
                xor ^= arr[i]
                i += 1
            res.append(xor)
        return res
```

前缀表达式

思路

比较常见的是前缀和，这个概念其实很容易理解，即一个数组中，第 n 位存储的是数组前 n 个数字的和。

对 [1,2,3,4,5,6] 来说，其前缀和可以是 pre=[1,3,6,10,15,21]。我们可以使用公式 pre[i]=pre[$i-1$]+nums[i]得到每一位前缀和的值，从而通过前缀和进行相应的计算和解题。其实前缀和的概念很简单，但困难的是如何在题目中使用前缀和以及如何使用前缀和的关系来进行解题。

这道题是对前缀异或。用代码表示就是：

```
pre[0] = 0
pre[i] = arr[0] ^ arr[1] ^ ... ^ arr[i - 1]
```

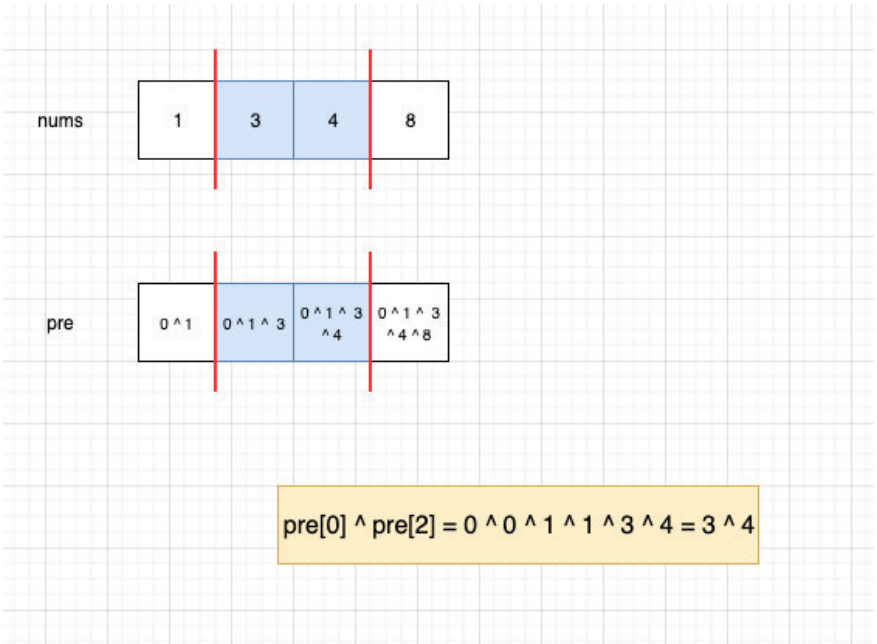
> ^ 表示异或

其中 pre 就是前缀异或数组，其本质和前缀和类似。接下来对一个区间进行异或，比如对 nums 的[0,2] 范围进行异或应该是 `nums[0] ^ nums[1] ^ nums[2]` 。建立了 pre 数组之后，我们就可以使用如下方式计算，而不是类似 `nums[0] ^ nums[1] ^ nums[2]` 的区间遍历形式。

```
pre[Li] ^ pre[Ri + 1] = (arr[0] ^ ... ^ arr[Li - 1]) ^ (arr
                      = (arr[0] ^ ... ^ arr[Li - 1]) ^ (arr
                      = (arr[Li] ^ ... ^ arr[Ri])
                      = arr[Li] ^ ... ^ arr[Ri]
```

上面成立的前提是异或的一个重要性质 `x ^ y ^ x = y` 。用自然语言来说就是异或一组数字，两两相同的会抵消，而上面的除了区间[Li,Ri]内的数，其他数都出现了两次，因此会被抵消。这样就达到了我们的目的。

也就是说如果要计算[Li,Ri] 的异或，不再需要遍历 [Li,Ri] 区间内的所有元素，而是直接用 pre[Li] ^ pre[Ri + 1] 计算即可。时间复杂度从 $O(R)$ 降低到了 $O(1)$， 其中 R 为区间长度，即 Ri - Li + 1。



代码

代码支持 Python3，Java，C++：

Python Code：

```
#
# @lc app=leetcode.cn id=1218 lang=python3
#
# [1218] 最长定差子序列
#

# @lc code=start


class Solution:
    def xorQueries(self, arr: List[int], queries: List[List
        pre = [0]
        res = []
        for i in range(len(arr)):
            pre.append(pre[i] ^ arr[i])
        for (L, R) in queries:
            res.append(pre[L] ^ pre[R + 1])
        return res

# @lc code=end
```

Java Code：

```
  public int[] xorQueries(int[] arr, int[][] queries) {

        int[] preXor = new int[arr.length];
        preXor[0] = 0;

        for (int i = 1; i < arr.length; i++)
            preXor[i] = preXor[i - 1] ^ arr[i - 1];

        int[] res = new int[queries.length];

        for (int i = 0; i < queries.length; i++) {

            int left = queries[i][0], right = queries[i][1
            res[i] = arr[right] ^ preXor[right] ^ preXor[le
        }

        return res;
    }
```

C++ Code:

```
class Solution {
public:
    vector<int> xorQueries(vector<int>& arr, vector<vector<
        vector<int>res;
        for(int i=1; i<arr.size(); ++i){
            arr[i]^=arr[i-1];
        }
        for(vector<int>temp :queries){
            if(temp[0]==0){
                res.push_back(arr[temp[1]]);
            }
            else{
                res.push_back(arr[temp[0]-1]^arr[temp[1]]);
            }
        }
        return res;
    }
};
```

复杂度分析

其中 N 为数组 arr 长度， M 为 queries 的长度。

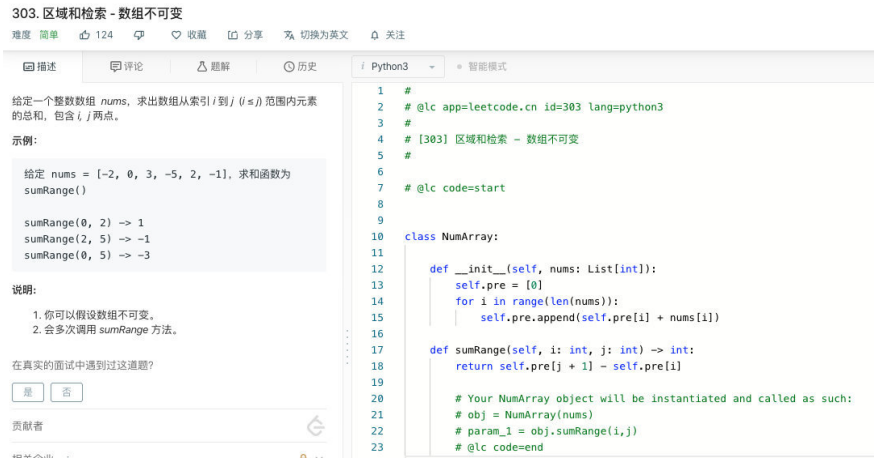
- 时间复杂度：$O(N * M)$
- 空间复杂度：$O(N)$

关键点解析

- 异或的性质 x ^ y ^ x = y
- 前缀表达式

相关题目

- 303. 区域和检索 - 数组不可变



- 1186.删除一次得到子数组最大和

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（1334. 阈值距离内邻居最少的城市）

https://leetcode-cn.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance/

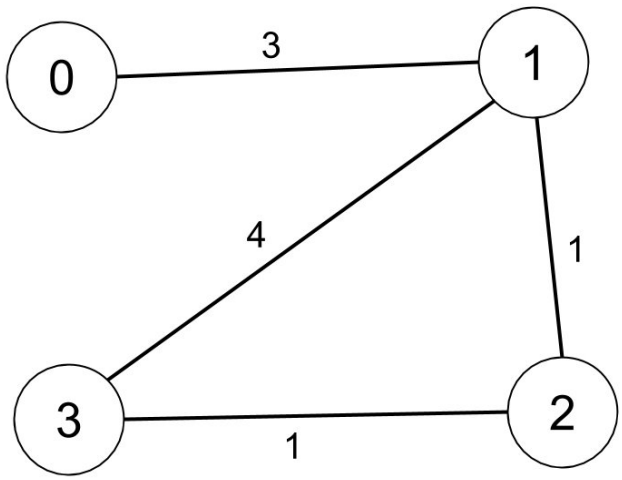
题目描述

有 n 个城市，按从 0 到 n-1 编号。给你一个边数组 edges，其中 edges[

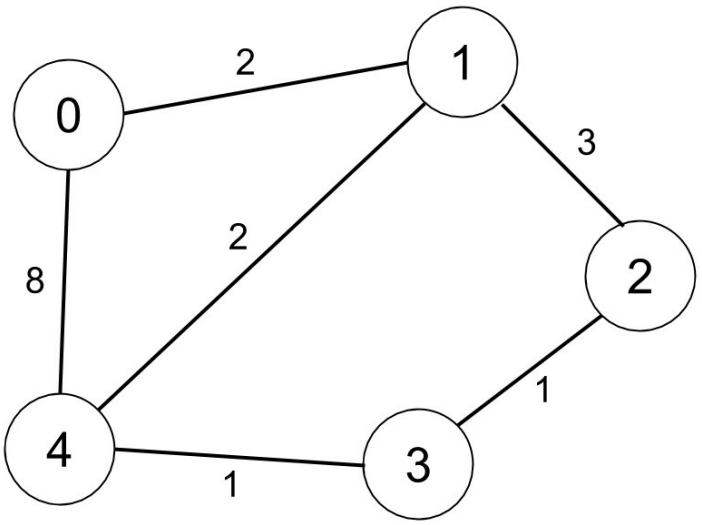
返回能通过某些路径到达其他城市数目最少、且路径距离 最大 为 distanceT

注意，连接城市 i 和 j 的路径的距离等于沿该路径的所有边的权重之和。

示例 1：



输入：n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], dist
输出：3
解释：城市分布图如上。
每个城市阈值距离 distanceThreshold = 4 内的邻居城市分别是：
城市 0 -> [城市 1, 城市 2]
城市 1 -> [城市 0, 城市 2, 城市 3]
城市 2 -> [城市 0, 城市 1, 城市 3]
城市 3 -> [城市 1, 城市 2]
城市 0 和 3 在阈值距离 4 以内都有 2 个邻居城市，但是我们必须返回城市
示例 2：



```
输入：n = 5, edges = [[0,1,2],[0,4,8],[1,2,3],[1,4,2],[2,3,1
输出：0
解释：城市分布图如上。
每个城市阈值距离 distanceThreshold = 2 内的邻居城市分别是：
城市 0 -> [城市 1]
城市 1 -> [城市 0, 城市 4]
城市 2 -> [城市 3, 城市 4]
城市 3 -> [城市 2, 城市 4]
城市 4 -> [城市 1, 城市 2, 城市 3]
城市 0 在阈值距离 4 以内只有 1 个邻居城市。


提示：

2 <= n <= 100
1 <= edges.length <= n * (n - 1) / 2
edges[i].length == 3
0 <= fromi < toi < n
1 <= weighti, distanceThreshold <= 10^4
所有 (fromi, toi) 都是不同的。
```

前置知识

- 动态规划
- Floyd-Warshall

公司

- 暂无

思路

这道题的本质就是：

1. 在一个无向图中寻找每两个城镇的最小距离，我们使用 Floyd-Warshall 算法（英语：Floyd-Warshall algorithm），中文亦称弗洛伊德算法，是解决任意两点间的最短路径的一种算法。
2. 筛选最小距离不大于 distanceThreshold 的城镇。
3. 统计每个城镇，其满足条件的城镇有多少个
4. 我们找出最少的即可

Floyd-Warshall 算法的时间复杂度和空间复杂度都是$O(N^3)$, 而空间复杂度可以优化到$O(N^2)$。Floyd-Warshall 的基本思想是对于每两个点之间的最小距离，要么经过中间节点 k，要么不经过，我们取两者的最小

值，这是一种动态规划思想，详细的解法可以参考Floyd-Warshall 算法 (wikipedia)

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], d
        # 构建dist矩阵
        dist = [[float('inf')] * n for _ in  range(n)]
        for i, j, w in edges:
            dist[i][j] = w
            dist[j][i] = w
        for i in range(n):
            dist[i][i] = 0
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    dist[i][j] = min(dist[i][j], dist[i][k]

        # 过滤
        res = 0
        minCnt = float('inf')
        for i in range(n):
            cnt = 0
            for d in dist[i]:
                if d <= distanceThreshold:
                    cnt += 1
            if cnt <= minCnt:
                minCnt = cnt
                res = i
        return res
```

复杂度分析

- 时间复杂度：$O(N^3)$
- 空间复杂度：$O(N^2)$

关键点解析

- Floyd-Warshall 算法
- 你可以将本文给的 Floyd-Warshall 算法当成一种解题模板使用

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1371. 每个元音包含偶数次的最长子字符串）

https://leetcode-cn.com/problems/find-the-longest-substring-containing-vowels-in-even-counts/

题目描述

```
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



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
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，


提示：

1 <= s.length <= 5 x 10^5
s 只包含小写英文字母。
```

前置知识

- 前缀和
- 状态压缩

暴力法 + 剪枝

公司

- 暂无

思路

首先拿到这道题的时候，我想到第一反应是滑动窗口行不行。 但是很快这个想法就被我否定了，因为滑动窗口（这里是可变滑动窗口）我们需要扩张和收缩窗口大小，而这里不那么容易。因为题目要求的是奇偶性，而不是类似“元音出现最多的子串”等。

突然一下子没了思路。那就试试暴力法吧。暴力法的思路比较朴素和直观。 那就是 双层循环找到所有子串，然后对于每一个子串，统计元音个数，如果子串的元音个数都是偶数，则更新答案，最后返回最大的满足条件的子串长度即可 。

这里我用了一个小的 trick。枚举所有子串的时候，我是从最长的子串开始枚举的，这样我找到一个满足条件的直接返回就行了（early return），不必维护最大值。 这样不仅减少了代码量，还提高了效率。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findTheLongestSubstring(self, s: str) -> int:
        for i in range(len(s), 0, -1):
            for j in range(len(s) - i + 1):
                sub = s[j:j + i]
                has_odd_vowel = False
                for vowel in ['a', 'e', 'i', 'o', 'u']:
                    if sub.count(vowel) % 2 != 0:
                        has_odd_vowel = True
                        break
                if not has_odd_vowel: return  i
        return 0
```

复杂度分析

- 时间复杂度：双层循环找出所有子串的复杂度是$O(n^2)$，统计元音个数复杂度也是$O(n)$，因此这种算法的时间复杂度为$O(n^3)$。
- 空间复杂度：$O(1)$

前缀和 + 剪枝

思路

上面思路中 `对于每一个子串，统计元音个数` ，我们仔细观察的话，会发现有很多重复的统计。那么优化这部分的内容就可以获得更好的效率。

对于这种连续的数字问题，这里我们考虑使用前缀和来优化。

经过这种空间换时间的策略之后，我们的时间复杂度会降低到$O(n ^ 2)$，但是相应空间复杂度会上升到$O(n)$，这种取舍在很多情况下是值得的。

代码

代码支持：Python3，Java

Python3 Code:

```python
class Solution:
    i_mapper = {
        "a": 0,
        "e": 1,
        "i": 2,
        "o": 3,
        "u": 4
    }
    def check(self, s, pre, l, r):
        for i in range(5):
            if s[l] in self.i_mapper and i == self.i_mapper
            else: cnt = 0
            if (pre[r][i] - pre[l][i] + cnt) % 2 != 0: retu
        return True
    def findTheLongestSubstring(self, s: str) -> int:
        n = len(s)

        pre = [[0] * 5 for _ in range(n)]

        # pre
        for i in range(n):
            for j in range(5):
                if s[i] in self.i_mapper and self.i_mapper
                    pre[i][j] = pre[i - 1][j] + 1
                else:
                    pre[i][j] = pre[i - 1][j]
        for i in range(n - 1, -1, -1):
            for j in range(n - i):
                if self.check(s, pre, j, i + j):
                    return i + 1
        return 0
```

Java Code：

```java
class Solution {
    public int findTheLongestSubstring(String s) {

        int len = s.length();

        if (len == 0)
            return 0;

        int[][] preSum = new int[len][5];
        int start = getIndex(s.charAt(0));
        if (start != -1)
            preSum[0][start]++;

        // preSum
        for (int i = 1; i < len; i++) {

            int idx = getIndex(s.charAt(i));

            for (int j = 0; j < 5; j++) {

                if (idx == j)
                    preSum[i][j] = preSum[i - 1][j] + 1;
                else
                    preSum[i][j] = preSum[i - 1][j];
            }
        }

        for (int i = len - 1; i >= 0; i--) {

            for (int j = 0; j < len - i; j++) {
                if (checkValid(preSum, s, j, i + j))
                    return i + 1;
            }
        }
        return 0;
    }


    public boolean checkValid(int[][] preSum, String s, int

        int idx = getIndex(s.charAt(left));

        for (int i = 0; i < 5; i++)
            if (((preSum[right][i] - preSum[left][i] + (idx
                return false;

        return true;
    }
```

```
    public int getIndex(char ch) {

        if (ch == 'a')
            return 0;
        else if (ch == 'e')
            return 1;
        else if (ch == 'i')
            return 2;
        else if (ch == 'o')
            return 3;
        else if (ch == 'u')
            return 4;
        else
            return -1;
    }
}
```

复杂度分析

- 时间复杂度：$O(n^2)$。
- 空间复杂度：$O(n)$

前缀和 + 状态压缩

思路

前面的前缀和思路，我们通过空间（prefix）换取时间的方式降低了时间复杂度。但是时间复杂度仍然是平方，我们是否可以继续优化呢?

实际上由于我们只关心奇偶性，并不关心每一个元音字母具体出现的次数。因此我们可以使用 `是奇数，是偶数` 两个状态来表示，由于只有两个状态，我们考虑使用位运算。

我们使用 5 位的二进制来表示以 i 结尾的字符串中包含各个元音的奇偶性，其中 0 表示偶数，1 表示奇数，并且最低位表示 a，然后依次是 e，i，o，u。比如 `10110` 则表示的是包含偶数个 a 和 o，奇数个 e，i，u，我们用变量 `cur` 来表示。

为什么用 0 表示偶数? 1 表示奇数?

回答这个问题，你需要继续往下看。

其实这个解法还用到了一个性质，这个性质是小学数学知识：

- 如果两个数字奇偶性相同，那么其相减一定是偶数。
- 如果两个数字奇偶性不同，那么其相减一定是奇数。

看到这里，我们再来看上面抛出的问题 `为什么用 0 表示偶数？1 表示奇数？` 。因为这里我们打算用异或运算，而异或的性质是：

如果对两个二进制做异或，会对其每一位进行位运算，如果相同则位 0，否则位 1。这和上面的性质非常相似。上面说 `奇偶性相同则位偶数，否则为奇数` 。因此很自然地 `用 0 表示偶数？1 表示奇数` 会更加方便。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def findTheLongestSubstring(self, s: str) -> int:
        mapper = {
            "a": 1,
            "e": 2,
            "i": 4,
            "o": 8,
            "u": 16
        }
        seen = {0: -1}
        res = cur = 0

        for i in range(len(s)):
            if s[i] in mapper:
                cur ^= mapper.get(s[i])
            # 全部奇偶性都相同，相减一定都是偶数
            if cur in seen:
                res = max(res, i - seen.get(cur))
            else:
                seen[cur] = i
        return res
```

复杂度分析

- 时间复杂度：$O(n)$。
- 空间复杂度：$O(n)$

关键点解析

- 前缀和
- 状态压缩

相关题目

- 掌握前缀表达式真的可以为所欲为！

题目地址(1381. 设计一个支持增量操作的栈)

https://leetcode-cn.com/problems/plus-one

题目描述

请你设计一个支持下述操作的栈。

实现自定义栈类 CustomStack ：

CustomStack(int maxSize)：用 maxSize 初始化对象，maxSize 是栈中
void push(int x)：如果栈还未增长到 maxSize ，就将 x 添加到栈顶。
int pop()：弹出栈顶元素，并返回栈顶的值，或栈为空时返回 -1 。
void inc(int k, int val)：栈底的 k 个元素的值都增加 val 。如果栈

示例：

输入：
["CustomStack","push","push","pop","push","push","push","in
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]
输出：
[null,null,null,2,null,null,null,null,null,103,202,201,-1]
解释：
CustomStack customStack = new CustomStack(3); // 栈是空的 []
customStack.push(1); // 栈变为 [1]
customStack.push(2); // 栈变为 [1, 2]
customStack.pop(); // 返回 2 --> 返回栈顶值 2，栈变为 [1]
customStack.push(2); // 栈变为 [1, 2]
customStack.push(3); // 栈变为 [1, 2, 3]
customStack.push(4); // 栈仍然是 [1, 2, 3]，不能添加其他元素使栈
customStack.increment(5, 100); // 栈变为 [101, 102, 103]
customStack.increment(2, 100); // 栈变为 [201, 202, 103]
customStack.pop(); // 返回 103 --> 返回栈顶值 103，栈变为 [201
customStack.pop(); // 返回 202 --> 返回栈顶值 202，栈变为 [201
customStack.pop(); // 返回 201 --> 返回栈顶值 201，栈变为 []
customStack.pop(); // 返回 -1 --> 栈为空，返回 -1

提示：

1 <= maxSize <= 1000
1 <= x <= 1000
1 <= k <= 1000
0 <= val <= 100
每种方法 increment，push 以及 pop 分别最多调用 1000 次

前置知识

- 栈
- 前缀和

increment 时间复杂度为 $O(k)$ 的方法

思路

首先我们来看一种非常符合直觉的方法，然而这种方法并不好，increment 操作需要的时间复杂度为 $O(k)$。

`push` 和 `pop` 就是普通的栈操作。 唯一要注意的是边界条件，这个已经在题目中指明了，具体来说就是：

- push 的时候要判断是否满了
- pop 的时候要判断是否空了

而做到上面两点，只需要一个 cnt 变量记录栈的当前长度，一个 size 变量记录最大容量，并在 pop 和 push 的时候更新 cnt 即可。

代码

```python
class CustomStack:

    def __init__(self, size: int):
        self.st = []
        self.cnt = 0
        self.size = size

    def push(self, x: int) -> None:
        if self.cnt < self.size:
            self.st.append(x)
            self.cnt += 1


    def pop(self) -> int:
        if self.cnt == 0: return -1
        self.cnt -= 1
        return self.st.pop()


    def increment(self, k: int, val: int) -> None:
        for i in range(0, min(self.cnt, k)):
            self.st[i] += val
```

复杂度分析

- 时间复杂度：push 和 pop 操作的时间复杂度为 $O(1)$（讲义有提到），而 increment 操作的时间复杂度为 $O(min(k, cnt))$
- 空间复杂度：$O(1)$

前缀和

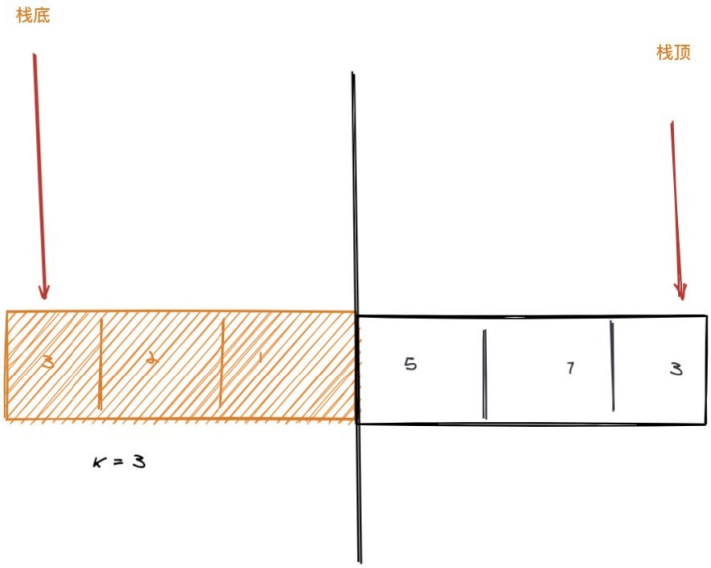
前缀和在讲义里面提到过，大家也可是看下我的文章 一次搞定前缀和

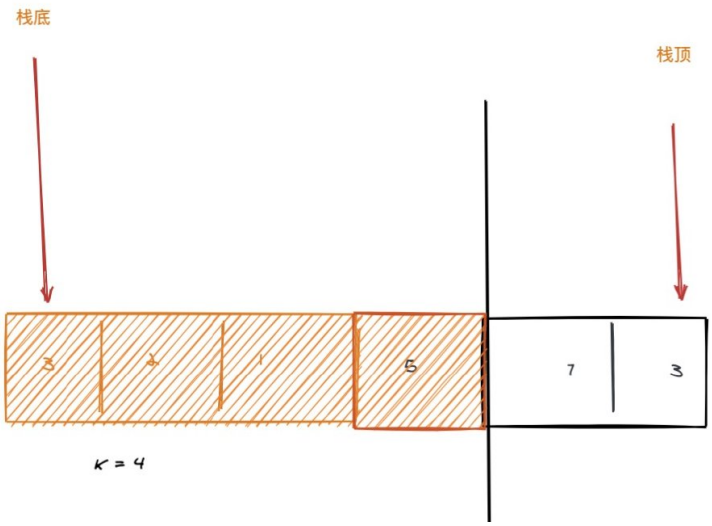
思路

和上面的思路类似，不过我们采用空间换时间的方式。采用一个额外的数组 incrementals 来记录每次 incremental 操作。

具体算法如下：

- 初始化一个大小为 maxSize 的数组 incrementals， 并全部填充 0
- push 操作不变，和上面一样
- increment 的时候，我们将用到 incremental 信息。那么这个信息是什么，从哪来呢？我这里画了一个图

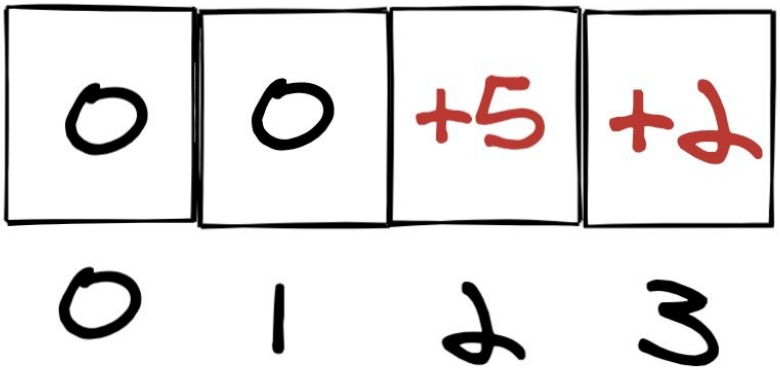




如图黄色部分是我们需要执行增加操作，我这里画了一个挡板分割，实际上这个挡板不存在。那么如何记录黄色部分的信息呢？我举个例子来说

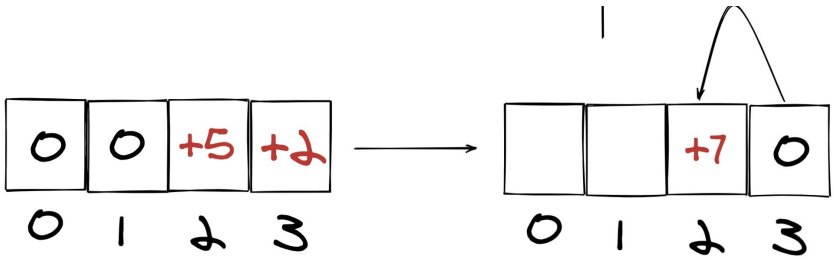
比如：

- 调用了 increment(3, 2)，就把 increment[3] 增加 2。
- 继续调用 increment(2, 5)，就把 increment[2] 增加 5。



而当我们 pop 的时候：

- 只需要将栈顶元素**加上 increment[cnt - 1]** 即可， 其中 cnt 为栈当前的大小。
- 另外，我们需要将 increment[cnt - 1] 更新到 increment[cnt - 2]，并将 increment[cnt - 1] 重置为 0。



代码

```
class CustomStack:

    def __init__(self, size: int):
        self.st = []
        self.cnt = 0
        self.size = size
        self.incrementals = [0] * size

    def push(self, x: int) -> None:
        if self.cnt < self.size:
            self.st.append(x)
            self.cnt += 1


    def pop(self) -> int:
        if self.cnt == 0: return -1
        if self.cnt >= 2:
            self.incrementals[self.cnt - 2] += self.increme
        ans = self.st.pop() + self.incrementals[self.cnt -
        self.incrementals[self.cnt - 1] = 0
        self.cnt -= 1
        return ans


    def increment(self, k: int, val: int) -> None:
            if self.cnt:
                self.incrementals[min(self.cnt, k) - 1] +=
```

复杂度分析

- 时间复杂度：全部都是 $O(1)$
- 空间复杂度：我们维护了一个大小为 maxSize 的数组，因此平均到每次的空间复杂度为 $O(maxSize / N)$，其中 N 为操作数。

优化的前缀和

思路

上面的思路无论如何，我们都需要维护一个大小为 $O(maxSize)$ 的数组 incremental 。而由于栈只能在栈顶进行操作，因此这实际上可以稍微优化一点，即维护一个大小为当前栈长度的 incrementals，而不是 $O(maxSize)$ 。

每次栈 push 的时候，incrementals 也 push 一个 0。每次栈 pop 的时候， incrementals 也 pop，这样就可以了。

> 这里的 incrementals 并不是一个栈，而是一个普通数组，因此可以随机访问。

代码

```
class CustomStack:

    def __init__(self, size: int):
        self.st = []
        self.cnt = 0
        self.size = size
        self.incrementals = []

    def push(self, x: int) -> None:
        if self.cnt < self.size:
            self.st.append(x)
            self.incrementals.append(0)
            self.cnt += 1


    def pop(self) -> int:
        if self.cnt == 0: return -1
        self.cnt -= 1
        if self.cnt >= 1:
            self.incrementals[-2] += self.incrementals[-1]
        return self.st.pop() + self.incrementals.pop()


    def increment(self, k: int, val: int) -> None:
        if self.incrementals:
            self.incrementals[min(self.cnt, k) - 1] += val
```

复杂度分析

- 时间复杂度：全部都是 $O(1)$
- 空间复杂度：我们维护了一个大小为 cnt 的数组，因此平均到每次的空间复杂度为 $O(cnt / N)$，其中 N 为操作数，cnt 为操作过程中的栈的最大长度（小于等于 maxSize）。

可以看出优化的解法在 maxSize 非常大的时候是很有意义的。

相关题目

- 155. 最小栈

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

欢迎长按关注

题目地址(1438. 绝对差不超过限制的最长连续子数组)

https://leetcode-cn.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/

题目描述

给你一个整数数组 nums ，和一个表示限制的整数 limit，请你返回最长连续

如果不存在满足条件的子数组，则返回 0 。

示例 1：

输入：nums = [8,2,4,7], limit = 4
输出：2
解释：所有子数组如下：
[8] 最大绝对差 |8-8| = 0 <= 4.
[8,2] 最大绝对差 |8-2| = 6 > 4.
[8,2,4] 最大绝对差 |8-2| = 6 > 4.
[8,2,4,7] 最大绝对差 |8-2| = 6 > 4.
[2] 最大绝对差 |2-2| = 0 <= 4.
[2,4] 最大绝对差 |2-4| = 2 <= 4.
[2,4,7] 最大绝对差 |2-7| = 5 > 4.
[4] 最大绝对差 |4-4| = 0 <= 4.
[4,7] 最大绝对差 |4-7| = 3 <= 4.
[7] 最大绝对差 |7-7| = 0 <= 4.
因此，满足题意的最长子数组的长度为 2 。

示例 2：

输入：nums = [10,1,2,4,7,2], limit = 5
输出：4
解释：满足题意的最长子数组是 [2,4,7,2]，其最大绝对差 |2-7| = 5 <=

示例 3：

输入：nums = [4,2,2,2,4,4,2,2], limit = 0
输出：3

提示：

1 <= nums.length <= 10^5
1 <= nums[i] <= 10^9
0 <= limit <= 10^9

前置知识

- 有序集合
- 二分法
- 滑动窗口
- 单调栈

公司

- 暂无

二分法

这道题核心的就是求连续子数组的最大值和最小值，而由于数据是静态的，因此没必要使用线段树。

这里我们可以手动维护一个有序数组 d。其中的数据表示的就是**某一个连续子数组**，只不过 d 是已经排好序的。比如原有的子数组是：[3,1,2]，那么 d 就是 [1,2,3]。我们可以使用二分法在 $logn$ 的时间找到插入点，并在最坏 $O(n)$ 的时间完成插入和删除。因此最坏时间复杂度是 $o(n^2)$。

接下来，我们使用滑动窗口技巧，代码上可使用双指针。而由于 d 的长度就是窗口的大小，因此使用一个指针表示右端点即可，因为左端点可通过**右端点 - d 的长度 + 1**得出。

思路

关键点

- 维护一个有序数组，并通过二分法找到插入位置

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def longestSubarray(self, A: List[int], limit: int) ->
        d = []
        ans = 1

        for i, a in enumerate(A):
            bisect.insort(d, a)
            if len(d) > 1:
                while d[-1] - d[0] > limit:
                    d.remove(A[i - len(d)+1])
                ans = max(ans, len(d))
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

有序集合

思路

和上面思路类似。 只不过将数据结构从数组变成了平衡树，这样插入和删除的复杂度可以降低到 $O(logn)$。Python 的 SortedList 可以达到此目的。Java 可用 TreeMap，C++ 可用 multiset 代替。

代码基本也是类似的，大家自己看下即可。

关键点

- 平衡二叉树优化插入和删除的时间复杂度

代码

- 语言支持：Python3

Python3 Code:

```
from sortedcontainers import SortedList
class Solution:
    def longestSubarray(self, A: List[int], limit: int) ->
        d = SortedList()
        ans = 1

        for i, a in enumerate(A):
            d.add(a)
            if len(d) > 1:
                while d[-1] - d[0] > limit:
                    d.remove(A[i - len(d)+1])
                ans = max(ans, len(d))
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(n)$

单调栈

思路

单调栈可快速得到最大值和最小值。因此我们可以使用两个栈，分别计算区间的最大值和最小值，接下来的思路和上面类似。即维护一个滑动窗口即可。

关于单调栈可参考我之前写的文章 单调栈解题模板秒杀八道题

关键点

- 单调栈获取最大最小值

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def longestSubarray(self, A: List[int], limit: int) ->
        q1, q2 = collections.deque(), collections.deque()
        ans = 1
        i = 0
        for j, a in enumerate(A):
            while q1 and q1[-1] < a:
                q1.pop()
            q1.append(a)
            while q2 and q2[-1] > a:
                q2.pop()
            q2.append(a)
            while i < j and q1 and q2 and q1[0] - q2[0] > 
                if A[i] == q1[0]: q1.popleft()
                elif A[i] == q2[0]: q2.popleft()
                i += 1
            ans = max(ans, j - i + 1)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（1558. 得到目标数组的最少函数调用次数）

https://leetcode-cn.com/problems/minimum-numbers-of-function-calls-to-make-target-array/

题目描述

```
func modify(arr, op, idx){
    //add by 1 index idx
    if (op == 0) {
        arr[idx] = arr[idx] + 1
    }
    //multiply by 2 all elements
    if (op == 1) {
        for(i = 0; i < arr.length; i++) {
            arr[i] = arr[i] * 2
        }
    }
}
```

给你一个与 nums 大小相同且初始值全为 0 的数组 arr ，请你调用以上函数

请你返回将 arr 变成 nums 的最少函数调用次数。

答案保证在 32 位有符号整数以内。

示例 1:

输入: nums = [1,5]
输出: 5
解释: 给第二个数加 1 : [0, 0] 变成 [0, 1] （1 次操作）。
将所有数字乘以 2 : [0, 1] -> [0, 2] -> [0, 4] （2 次操作）。
给两个数字都加 1 : [0, 4] -> [1, 4] -> [1, 5] （2 次操作）。
总操作次数为: 1 + 2 + 2 = 5 。
示例 2:

输入: nums = [2,2]
输出: 3
解释: 给两个数字都加 1 : [0, 0] -> [0, 1] -> [1, 1] （2 次操作）
将所有数字乘以 2 : [1, 1] -> [2, 2] （1 次操作）。
总操作次数为: 2 + 1 = 3 。
示例 3:

输入: nums = [4,2,5]
输出: 6
解释: （初始）[0,0,0] -> [1,0,0] -> [1,0,1] -> [2,0,2] -> [2,
示例 4:

输入: nums = [3,2,2,4]
输出: 7
示例 5:

输入: nums = [2,4,8,16]
输出: 8

提示:

1 <= nums.length <= 10^5
0 <= nums[i] <= 10^9

前置知识

- 模拟

公司

- 暂无

思路

我们采用模拟的思路。 模拟指的是题目让我干什么，我干什么。

由于只能进行两种操作， 因此总的操作数就是两种操作的和。这里使用两个变量分别记录两种操作的数目，最后将其和返回即可。

由于题目给的参数是目标值， 其实我们这里也可以采用逆向思考， 即从 nums 递归到全零数组，这对结果不会产生影响。

```
class Solution:
    def minOperations(self, nums: List[int]) -> int:
        max_multi = add = 0

        for num in nums:
            # your code here
        return max_multi + add
```

算法：

- 从左到右遍历数组中的每一项
- 如果该项是奇数，则需要减去 1，同时 add 操作 + 1
- 如果该项是大于 0 的偶数， 则需要进行 除 2 操作，同时 multi 操作 + 1
- 每次遍历都会产生一个 multi，而由于 multi 次数取决于数组最大项，因此我们需要维护全局最大的 multi
- 最后的结果就是 add + 全局最大的 multi

关键点

- 逆向思考
- 使用两个变量分别记录 add 和 multi 的次数
- multi 取决于整个数组最大的数，add 取决于数组出现奇数的次数

代码

代码支持：Python3

```python
class Solution:
    def minOperations(self, nums: List[int]) -> int:
        max_multi = add = 0

        for num in nums:
            multi = 0
            while num > 0:
                if num & 1 == 1:
                    add += 1
                    num -= 1
                if num >= 2:
                    multi += 1
                    num //= 2

            max_multi = max(max_multi, multi)
        return max_multi + add
```

复杂度分析

- 时间复杂度：$O(N * (max_multi + add))$，其中 N 为 nums 的长度。
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（1574. 删除最短的子数组使剩余数组有序）

https://leetcode-cn.com/problems/shortest-subarray-to-be-removed-to-make-array-sorted/description/

题目描述

```
给你一个整数数组 arr ，请你删除一个子数组（可以为空），使得 arr 中剩

一个子数组指的是原数组中连续的一个子序列。

请你返回满足题目要求的最短子数组的长度。



示例 1：

输入：arr = [1,2,3,10,4,2,3,5]
输出：3
解释：我们需要删除的最短子数组是 [10,4,2] ，长度为 3 。剩余元素形成非
另一个正确的解为删除子数组 [3,10,4] 。
示例 2：

输入：arr = [5,4,3,2,1]
输出：4
解释：由于数组是严格递减的，我们只能保留一个元素。所以我们需要删除长度
示例 3：

输入：arr = [1,2,3]
输出：0
解释：数组已经是非递减的了，我们不需要删除任何元素。
示例 4：

输入：arr = [1]
输出：0


提示：

1 <= arr.length <= 10^5
0 <= arr[i] <= 10^9
```

前置知识

- 双指针
- 滑动窗口

公司

- 暂无

思路

首先考虑如果题目不要求必须删除连续的子数组，而是任意的子序列。那么我们可以使用 LIS（最长上升子序列模型）求出 LIS 长度，然后用 n 减去它即可。对于 LIS 模型不熟悉的，可以看下我的这篇文章。

这道题是求极值的，我首先想到的 DP，简单思考了下没啥思路。 而这道题要求我们必须连续，那就考虑滑动窗口。

首先我们将数组分成三部分 A，B，C（A，B，C 都可为空）。由于删除必须连续，因此我们不能只删除 A，C，除此之外可以随便删除。这是题目条件，也是解决问题的关键之一。

不难看出题目的解空间上界是 n - 1，下界是 0，其中 n 为数组长度。

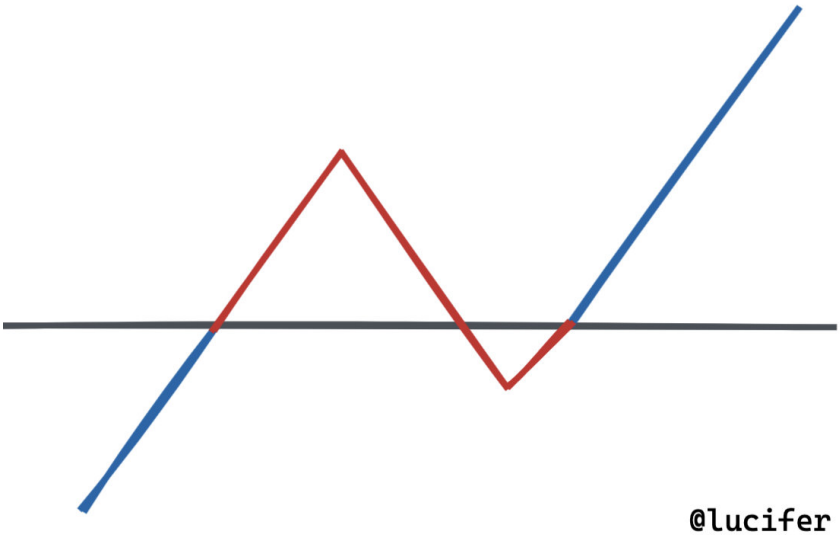
进一步思考。题目的上界其实也可以是 `n - 最长连续非递减子序列` 的长度。我们可以扫描一次数组，统计最长的连续非递减的子序列长度即可。

Java 代码：

```
ans = cnt = 1
for(int i = 1; i < A.length; i++ ) {
    if (A[i] >= A[i - 1]) {
        cnt++
    }
    else {
        ans = max(ans, cnt)
        cnt = 1
    }
}
```

这样 ans 就是 **最长连续非递减子序列** 的长度了。

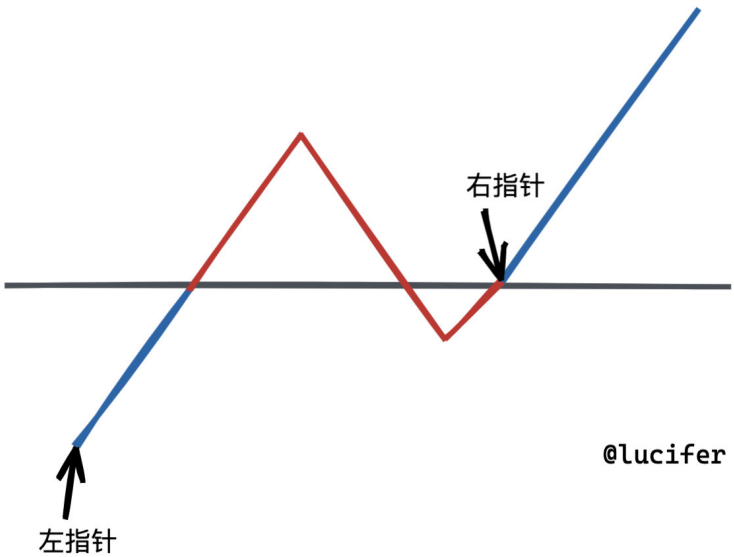
但是显然这只是上界， 并不是正确解。一个显而易见的反例是：



如图我们取蓝色部分，而将红色部分删除，答案可能会更小。

实际上，这道题的思路和11. 盛最多水的容器 有点类似。只是这道题比较隐蔽，不那么容易想到。因此大家可以先从那道题开始，了解下这个套路。

一个可行的思路是初始化两个指针，一个指向头部，一个指向从尾部起第一个拐点（如上图右边蓝色部分的左端点）。



假设左指针为 i 右指针为 j，我们只需要不断右移左指针，左移右指针，并根据 i 和 j 的相对大小更新窗口即可。

值得注意的是，左指针不应该超过右侧第一个拐点，右指针也不应该超过左侧第一个拐点，原因就是前面讲到的**不能只删除 A，C**。 因此左指针移动的过程是单调非递减的，右指针是单调非递增的。这是本题的重中之重。

具体来说：

- 如果 A[i] <= A[j]，我们可以选择删除 [i+1,j-1] 得到一个**候选解**。
- 如果 A[i] > A[j]，屋面不仅无法通过删除 [i+1,j-1] 得到候选解，并且大于 A[i] 的更不用看了，更加不会满足。又由于上面分析的 A[i] 移动过程是单调不递减的，因此就没有必要继续移动了。我们可以通过右移左指针来排序所有的 [i+1, j-1]，[i+2, j- 1]，.....。

当然这里面还有一些细节，大家需要看代码才能完整领会。强烈建议大家自己写画一个图，然后写一遍，特别需要注意各种边界的判断。

关键点

- 画图
- 边界条件的考察（比如+1 -1 等号）

代码

代码支持：C++，Python3

Python3 Code:

```python
class Solution:
    def findLengthOfShortestSubarray(self, A: List[int]) ->
        n = len(A)
        l, r = 0, n - 1

        while l < n - 1 and A[l] <= A[l + 1]:
            l += 1
        if l == n - 1:
            return 0
        while r > 0 and A[r] >= A[r - 1]:
            r -= 1
        ans = min(r, n - l - 1)
        i = 0
        while i <= l and r < n:
            if A[i] <= A[r]:
                # delete i + 1 ~ r - 1
                ans = min(ans, r - i - 1)
                i += 1
            else:
                # extend the sliding window
                r += 1
        return ans
```

C++ Code:

```cpp
class Solution {
public:
    int findLengthOfShortestSubarray(vector<int>& A) {
        int N = A.size(), left = 0, right = N - 1;
        while (left + 1 < N && A[left] <= A[left + 1]) ++le
        if (left == A.size() - 1) return 0;
        while (right > left && A[right - 1] <= A[right]) --
        int ans = min(N - left - 1, right), i = 0, j = righ
        while (i <= left && j < N) {
            if (A[j] >= A[i]) {
                ans = min(ans, j - i - 1);
                ++i;
            } else ++j;
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$

相关题目

- 11. 盛最多水的容器

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

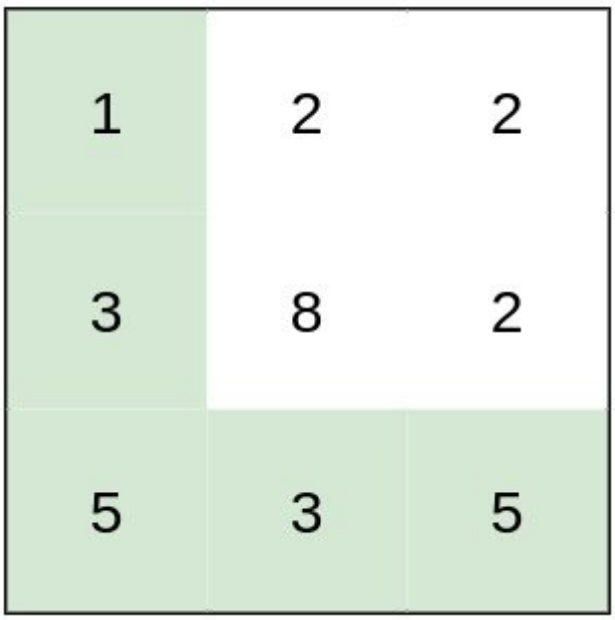
欢迎长按关注

题目地址（1631. 最小体力消耗路径）

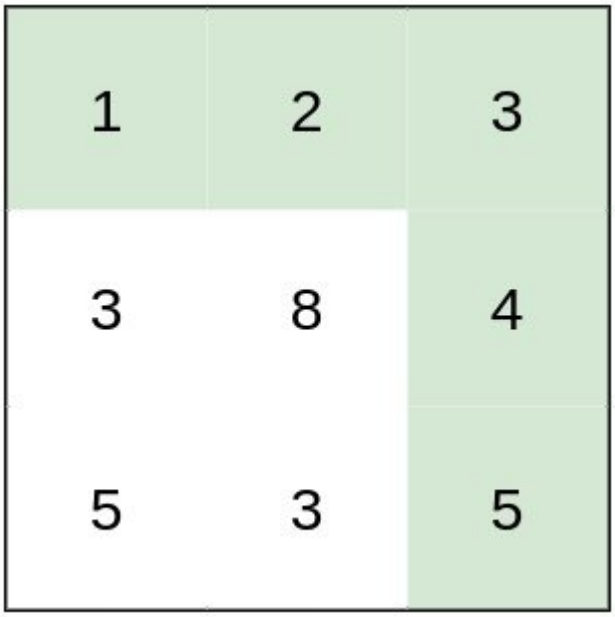
https://leetcode-cn.com/problems/path-with-minimum-effort/

题目描述

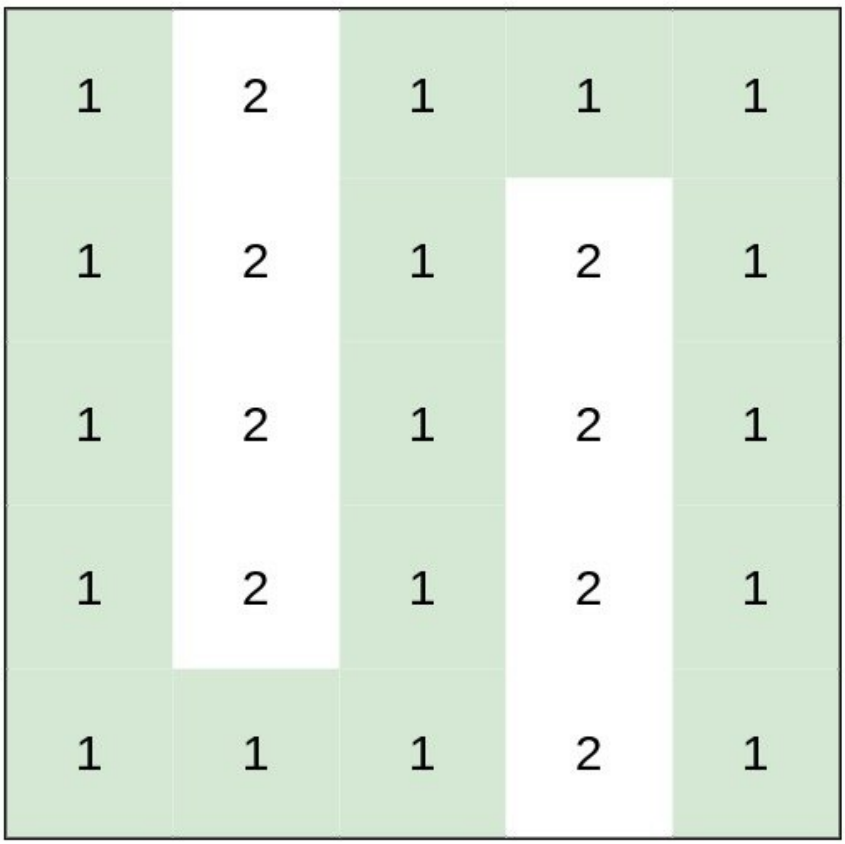
```
你准备参加一场远足活动。给你一个二维 rows x columns 的地图 heights

一条路径耗费的 体力值 是路径上相邻格子之间 高度差绝对值 的 最大值 决定

请你返回从左上角走到右下角的最小 体力消耗值 。


示例 1：
```




```
输入：heights = [[1,2,2],[3,8,2],[5,3,5]]
输出：2
解释：路径 [1,3,5,3,5] 连续格子的差值绝对值最大为 2 。
这条路径比路径 [1,2,2,2,5] 更优，因为另一条路劲差值最大值为 3 。
示例 2：
```



输入：heights = [[1,2,3],[3,8,4],[5,3,5]]
输出：1
解释：路径 [1,2,3,4,5] 的相邻格子差值绝对值最大为 1 ，比路径 [1,3,
示例 3：



```
输入：heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,
输出：0
解释：上图所示路径不需要消耗任何体力。


提示：

rows == heights.length
columns == heights[i].length
1 <= rows, columns <= 100
1 <= heights[i][j] <= 10e6
```

前置知识

- 二维矩阵
- 深度优先遍历
- 二分查找

公司

- 暂无

思路

如果采用暴力解法， 需要找出所有的路径，然后返回最小代价的即可，时间复杂度是指数级别。回头看一下数据范围是 10e6，因此这种解法是不行的。

由于题目的解空间是 [0, 10**6 - 1]。

> 对解空间这个概念不熟悉的，可以看我之前的一篇题解686. 重复叠加字符串匹配

本质上，我们需要进行发问：

- 0 可以么？
- 1 可以么？
- 2 可以么？
- 。。。

直到找到第一个不可以的，我们返回前一个即可。

关于可不可以，我们可以使用 DFS 来做，由于只需要找到一条满足条件的，或者找到一个不满足的提前退出，因此最坏的情况是一直符合，并走到终点，这种情况下时间复杂度是 $(m \times n)$，因此总的时间复杂度

是 $O(m \times n \times 10**6)$。

实际上，上面的不断发问的过程不就是一个连续的递增序列么？ 我们的目标不就是在一个连续递增序列找指定值么？于是二分法就不难想到。

而且这道题本质就是二分查找中的**查找最右侧满足条件的值**，关于这个问题，我已经在 【91 天学算法】二分查找 中进行了详细描述，并给出了代码模板，直接套就可以了。

值得注意的是，我们只需要找到一个满足条件的路径即可，因此可以利用短路剪枝。

```
return dfs(i + 1, j, heights[i][j], target) or dfs(i - 1, j
```

而不是写出下面的代码（下面的代码会超时）：

```
top = dfs(i + 1, j, heights[i][j], target)
bottom = dfs(i - 1, j, heights[i][j], target)
right = dfs(i, j + 1, heights[i][j], target)
left = dfs(i, j - 1, heights[i][j], target)
return top or bottom or right or left
```

代码

代码支持：Python3

```
class Solution:
    def minimumEffortPath(self, heights: List[List[int]]) -
        lo, hi = 0, 10**6 - 1
        m, n = len(heights), len(heights[0])
        def dfs(i, j, pre, target):
            if (i, j) in visited: return False
            if i < 0 or i >= m or j < 0 or j >= n or abs(he
            if i == m - 1 and j == n - 1: return True
            visited.add((i, j))
            return dfs(i + 1, j, heights[i][j], target) or
        # 查找最右侧满足条件的值
        while lo <= hi:
            visited = set()
            mid = (lo + hi) >> 1
            if dfs(0, 0, heights[0][0], mid): hi = mid - 1
            else: lo = mid + 1
        return lo
```

复杂度分析

m 为 矩阵的高度， n 为矩阵的长度。

- 时间复杂度：$O(4 \times m \times n \times log_2 10^6)$，其中 $log_2 10^6$ 为二分的次数， $4 \times m \times n$ 为每次 dfs 的时间。
- 空间复杂度：$O(m \times n)$，不管是递归的栈开销还是 visited 的开销都是 $O(m \times n)$。

相关问题

- 875. 爱吃香蕉的珂珂

题目地址（1658. 将 x 减到 0 的最小操作数）

https://leetcode-cn.com/problems/minimum-operations-to-reduce-x-to-zero

题目描述

```
给你一个整数数组 nums 和一个整数 x 。每一次操作时，你应当移除数组 nu

如果可以将 x 恰好 减到 0 ，返回 最小操作数 ；否则，返回 -1 。



示例 1:

输入: nums = [1,1,4,2,3], x = 5
输出: 2
解释: 最佳解决方案是移除后两个元素，将 x 减到 0 。
示例 2:

输入: nums = [5,6,7,8,9], x = 4
输出: -1
示例 3:

输入: nums = [3,2,20,1,1,3], x = 10
输出: 5
解释: 最佳解决方案是移除后三个元素和前两个元素（总共 5 次操作），将 x


提示:

1 <= nums.length <= 10^5
1 <= nums[i] <= 104
1 <= x <= 109
```

前置知识

- 堆
- 滑动窗口

公司

- 暂无

堆

思路

这里可以使用堆来解决。具体来说是我自己总结的**多路归并**题型。

> 关于这个算法套路，请期待后续的堆专题。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def minOperations(self, nums: List[int], x: int) -> int
        # 看数据范围，这种方法铁定超时（指数复杂度）
        h = [(0, 0, len(nums) - 1, x)]
        while h:
            moves,l,r,remain = heapq.heappop(h)
            if remain == 0: return moves
            if l + 1 < len(nums): heapq.heappush(h, (moves
            if r > 0: heapq.heappush(h, (moves + 1, l,r-1,
        return -1
```

复杂度分析

- 时间复杂度：$O(2^moves)$，其中 moves 为题目答案。最坏情况 moves 和 N 同阶，也就是 2^N。
- 空间复杂度：$O(1)$。

由于题目数组长度最大可以达到 10^5， 这提示我们此方法必然超时。

我们必须考虑时间复杂度更加优秀的方式。

动态规划（记忆化递归）

思路

由上面的解法， 我们不难想到使用动态规划来解决。

枚举所有的 l,r,x 组合，并找到最小的，其中 l 表示 左指针， r 表示右指针，x 表示剩余的数字。这里为了书写简单我使用了记忆化递归。

代码

代码支持：Python3

Python3 Code:

> Python 的 @lru_cache 是缓存计算结果的数据结构， None 表示不限制容量。

```
class Solution:
    def minOperations(self, nums: List[int], x: int) -> int
        n = len(nums)

        @lru_cache(None)
        def dp(l, r, x):
            if x == 0:
                return 0
            if x < 0 or r < 0 or l > len(nums) - 1:
                return n + 1
            return 1 + min(dp(l + 1, r, x - nums[l]), dp(l,

        ans = dp(0, len(nums) - 1, x)
        return -1 if ans > n else ans
```

复杂度分析

- 时间复杂度：$O(N^2 * h)$，其中 N 为数组长度, h 为 x 的减少速度，最坏的情况可以达到三次方的复杂度。
- 空间复杂度：$O(N)$，其中 N 为数组长度，这里的空间指的是递归栈的开销。

这种复杂度仍然无法通过 10^5 规模，需要继续优化算法。

滑动窗口

思路

实际上，我们也可以逆向思考。即：我们剩下的数组一定是原数组的中间部分。

那是不是就是说，我们只要知道数据中子序和等于 sum(nums) - x 的长度。用 nums 的长度减去它就好了？

由于我们的目标是 `最小操作数` ，因此我们只要求**和为定值的最长子序列**，这是一个典型的滑动窗口问题。

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def minOperations(self, nums: List[int], x: int) -> int
        # 逆向求解，滑动窗口
        i = 0
        target = sum(nums) - x
        win = 0
        ans = len(nums)
        if target == 0: return ans
        for j in range(len(nums)):
            win += nums[j]
            while i < j and win > target:
                win -= nums[i]
                i += 1
            if win == target:
                ans = min(ans, len(nums) - (j - i + 1))
        return -1 if ans == len(nums) else ans
```

复杂度分析

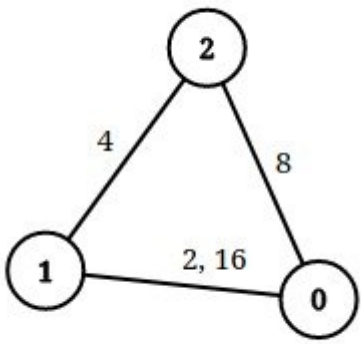
- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(1)$。

题目地址(1697. 检查边长度限制的路径是否存在)

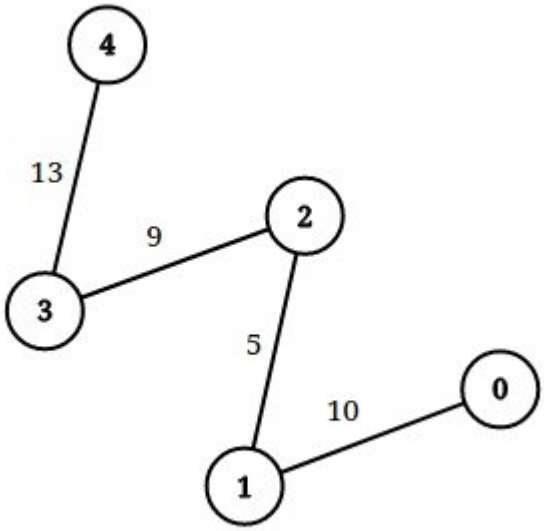
https://leetcode-cn.com/problems/checking-existence-of-edge-length-limited-paths/

题目描述

```
给你一个 n 个点组成的无向图边集 edgeList ，其中 edgeList[i] = [u

给你一个查询数组queries ，其中 queries[j] = [pj, qj, limitj] ，

请你返回一个 布尔数组 answer ，其中 answer.length == queries.le



示例 1：
```




```
输入：n = 3, edgeList = [[0,1,2],[1,2,4],[2,0,8],[1,0,16]],
输出：[false,true]
解释：上图为给定的输入数据。注意到 0 和 1 之间有两条重边，分别为 2 和
对于第一个查询，0 和 1 之间没有小于 2 的边，所以我们返回 false 。
对于第二个查询，有一条路径（0 -> 1 -> 2）两条边都小于 5 ，所以这个查
示例 2：
```




```
输入：n = 5, edgeList = [[0,1,10],[1,2,5],[2,3,9],[3,4,13]],
输出：[true,false]
解释：上图为给定数据。


提示：

2 <= n <= 105
1 <= edgeList.length, queries.length <= 105
edgeList[i].length == 3
queries[j].length == 3
0 <= ui, vi, pj, qj <= n - 1
ui != vi
pj != qj
1 <= disi, limitj <= 109
两个点之间可能有 多条 边。
```

前置知识

- 排序
- 并查集

公司

- 暂无

思路

本题和 1170. 比较字符串最小字母出现频次 类似， 都可以采取离线排序优化的方式来解。

具体来说，我们可以分别对 edges 和 queries 进行一次升序排序。接下来，遍历 queries。遍历 queries 的同时**将权值小于 limitj 的边进行合并**。接下来，我们只需要判断 pj 和 qj 是否已经在同一个联通域即可。因此如果 pj 和 qj 在同一个联通域，那么其联通的路径上的所有边必定都小于 limitj，其原因就是前面加粗的那句话。

注意到排序打乱了 queries 的索引，因此我们需要记录一下其原始索引。

做完这道题之后建议大家完成下方的相关题目，以巩固这个知识点。

关键点

- 离线查询优化

代码

- 语言支持：Python3

Python3 Code:

```
class UF:
    parent = {}
    size = {}
    cnt = 0
    def __init__(self, M):
        # 初始化 parent, size 和 cnt
        for i in range(M):
            self.parent[i] = i
            self.size[i] = 1

    def find(self, x):
        while x != self.parent[x]:
            x = self.parent[x]
            # 路径压缩
            self.parent[x] = self.parent[self.parent[x]];
        return x
    def union(self, p, q):
        if self.connected(p, q): return
        # 小的树挂到大的树上,  使树尽量平衡
        leader_p = self.find(p)
        leader_q = self.find(q)
        if self.size[leader_p] < self.size[leader_q]:
            self.parent[leader_p] = leader_q
            self.size[leader_p] += self.size[leader_q]
        else:
            self.parent[leader_q] = leader_p
            self.size[leader_q] += self.size[leader_p]
        self.cnt -= 1
    def connected(self, p, q):
        return self.find(p) == self.find(q)
class Solution:
    def distanceLimitedPathsExist(self, n: int, edgeList: l
        m = len(queries)
        edgeList.sort(key=lambda a:a[2])
        queries = [(fr, to, w, i) for i, [fr, to, w] in enu
        queries.sort(key=lambda a:a[2])
        ans = [False] * m
        uf = UF(n)
        j = 0
        for fr, to, w, i in queries:
            while j < len(edgeList) and edgeList[j][2] < w:
                uf.union(edgeList[j][0], edgeList[j][1])
                j += 1
            if uf.connected(fr, to): ans[i] = True
        return ans
```

复杂度分析

令 m, q edges 和 queries 的长度。

- 时间复杂度：$O(mlogm + qlogq)$
- 空间复杂度：$O(n + q)$

相关题目

- 1170. 比较字符串最小字母出现频次

题目地址(1737. 满足三条件之一需改变的最少字符数)

https://leetcode-cn.com/problems/change-minimum-characters-to-satisfy-one-of-three-conditions/

题目描述

```
给你两个字符串 a 和 b ，二者均由小写字母组成。一步操作中，你可以将 a

操作的最终目标是满足下列三个条件 之一 ：

a 中的 每个字母 在字母表中 严格小于 b 中的 每个字母 。
b 中的 每个字母 在字母表中 严格小于 a 中的 每个字母 。
a 和 b 都 由 同一个 字母组成。
返回达成目标所需的 最少 操作数。



示例 1：

输入：a = "aba", b = "caa"
输出：2
解释：满足每个条件的最佳方案分别是：
1）将 b 变为 "ccc"，2 次操作，满足 a 中的每个字母都小于 b 中的每个
2）将 a 变为 "bbb" 并将 b 变为 "aaa"，3 次操作，满足 b 中的每个字
3）将 a 变为 "aaa" 并将 b 变为 "aaa"，2 次操作，满足 a 和 b 由同
最佳的方案只需要 2 次操作（满足条件 1 或者条件 3）。
示例 2：

输入：a = "dabadd", b = "cda"
输出：3
解释：满足条件 1 的最佳方案是将 b 变为 "eee" 。


提示：

1 <= a.length, b.length <= 105
a 和 b 只由小写字母组成
```

前置知识

- 计数

- 枚举

公司

- 暂无

思路

三个条件中，**前两个条件其实是一样的**，因为如果你会了其中一个，那么你只需要将 A 和 B 交换位置就可以解出另外一个了。

对于前两个条件来说，我们可以枚举所有可能。以条件一 `A 中的 每个字母 在字母表中 严格小于 B 中的 每个字母` 为例。我们要做的就是枚举所有可能的 A 的最大字母 和 B 的最小字母（其中 A 的最大字母保证严格小于 B 的最大字母），并计算操作数，最后取最小值即可。

代码上，我们需要先统计 A 和 B 的字符出现次数信息，不妨分别记为 counter_A 和 counter_B。接下来，我们就可以执行核心的枚举逻辑了。

核心代码:

```
        # 枚举 A 的最大字母
        for i in range(1, 26):
            t = 0
            # 大于等于 i 的所有字符都需要进行一次操作
            for j in range(i, 26):
                t += counter_A[j]
            # 小于 i 的所有字符都需要进行一次操作
            for j in range(i):
                t += counter_B[j]
            # 枚举的所有情况中取最小的
            ans = min(ans, t)
```

而对于第三个条件，则比较简单，我们只需要将 A 和 B 改为同一个字母，并计算出操作数，取最小值即可。我们可能修改成的字母一共只有 26 种可能，因此直接枚举即可。

核心代码:

```
for i in range(26):
    ans = min(ans, len(A) + len(B) - counter_A[i] - counte
```

关键点

- 使用一个长度为 26 的数组计数不仅性能比哈希表好，并且在这里代码书写会更简单

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def minCharacters(self, A: str, B: str) -> int:
        counter_A = [0] * 26
        counter_B = [0] * 26
        for a in A:
            counter_A[ord(a) - ord('a')] += 1
        for b in B:
            counter_B[ord(b) - ord('a')] += 1
        ans = len(A) + len(B)
        for i in range(26):
            ans = min(ans, len(A) + len(B) - counter_A[i] -
        for i in range(1, 26):
            t = 0
            for j in range(i, 26):
                t += counter_A[j]
            for j in range(i):
                t += counter_B[j]
            ans = min(ans, t)
        for i in range(1, 26):
            t = 0
            for j in range(i, 26):
                t += counter_B[j]
            for j in range(i):
                t += counter_A[j]
            ans = min(ans, t)
        return ans
```

复杂度分析

令 m, n 分别为数组 A 和数组 B 的长度。

- 时间复杂度：$O(m + n)$
- 空间复杂度：$O(26)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

困难难度题目合集

困难难度题目从类型上说多是：

- 图
- 设计题
- 游戏场景题目
- 中等题目的 follow up

从解法上来说，多是：

- 图算法
- 动态规划
- 二分法
- DFS & BFS
- 状态压缩
- 剪枝

从逻辑上说， 要么就是非常难想到，要么就是非常难写代码。 这里我总结了几个技巧：

1. 看题目的数据范围， 看能否暴力模拟
2. 暴力枚举所有可能的算法往上套，比如图的题目。
3. 总结和记忆解题模板，减少解题压力

以下是我列举的经典题目（带 91 字样的表示出自 **91 天学算法**活动）：

- 0004. 寻找两个正序数组的中位数
- 0023. 合并 K 个升序链表
- 0025. K 个一组翻转链表
- 0030. 串联所有单词的子串
- 0032. 最长有效括号
- 0042. 接雨水
- 0052. N 皇后 II
- 0057. 插入区间 🆕
- 0084. 柱状图中最大的矩形
- 0085. 最大矩形
- 0124. 二叉树中的最大路径和
- 0128. 最长连续序列
- 0140. 单词拆分 II 🆕
- 0145. 二叉树的后序遍历
- 0212. 单词搜索 II
- 0239. 滑动窗口最大值
- 0295. 数据流的中位数
- 0297. 二叉树的序列化与反序列化 91

- 0301. 删除无效的括号
- 0312. 戳气球
- 330. 按要求补齐数组
- 0335. 路径交叉
- 0460. LFU 缓存
- 0472. 连接词
- 0488. 祖玛游戏
- 0493. 翻转对
- 0715. Range 模块 🆕
- 0768. 最多能完成排序的块 II 91
- 0887. 鸡蛋掉落
- 0895. 最大频率栈
- 0975. 奇偶跳 🆕
- 1032. 字符流
- 1168. 水资源分配优化
- 1203. 项目管理 🆕
- 1255. 得分最高的单词集合
- 1345. 跳跃游戏 IV
- 1449. 数位成本和为目标值的最大数字 🆕
- 5640. 与数组中元素的最大异或值 🆕

题目地址(LCP 20. 快速公交)

https://leetcode-cn.com/problems/meChtZ/

题目描述

小扣打算去秋日市集，由于游客较多，小扣的移动速度受到了人流影响：

小扣从 x 号站点移动至 x + 1 号站点需要花费的时间为 inc；
小扣从 x 号站点移动至 x - 1 号站点需要花费的时间为 dec。

现有 m 辆公交车，编号为 0 到 m-1。小扣也可以通过搭乘编号为 i 的公交车

假定小扣起始站点记作 0，秋日市集站点记作 target，请返回小扣抵达秋日市

注意：小扣可在移动过程中到达编号大于 target 的站点。

示例 1：

输入：target = 31, inc = 5, dec = 3, jump = [6], cost = [10]

输出：33

解释：
小扣步行至 1 号站点，花费时间为 5；
小扣从 1 号站台搭乘 0 号公交至 6 * 1 = 6 站台，花费时间为 10；
小扣从 6 号站台步行至 5 号站台，花费时间为 3；
小扣从 5 号站台搭乘 0 号公交至 6 * 5 = 30 站台，花费时间为 10；
小扣从 30 号站台步行至 31 号站台，花费时间为 5；
最终小扣花费总时间为 33。

示例 2：

输入：target = 612, inc = 4, dec = 5, jump = [3,6,8,11,5,10,

输出：26

解释：
小扣步行至 1 号站点，花费时间为 4；
小扣从 1 号站台搭乘 0 号公交至 3 * 1 = 3 站台，花费时间为 4；
小扣从 3 号站台搭乘 3 号公交至 11 * 3 = 33 站台，花费时间为 3；
小扣从 33 号站台步行至 34 站台，花费时间为 4；
小扣从 34 号站台搭乘 0 号公交至 3 * 34 = 102 站台，花费时间为 4；
小扣从 102 号站台搭乘 1 号公交至 6 * 102 = 612 站台，花费时间为 7
最终小扣花费总时间为 26。

提示：

1 <= target <= 10^9
1 <= jump.length, cost.length <= 10
2 <= jump[i] <= 10^6
1 <= inc, dec, cost[i] <= 10^6

前置知识

- 递归
- 回溯
- 动态规划

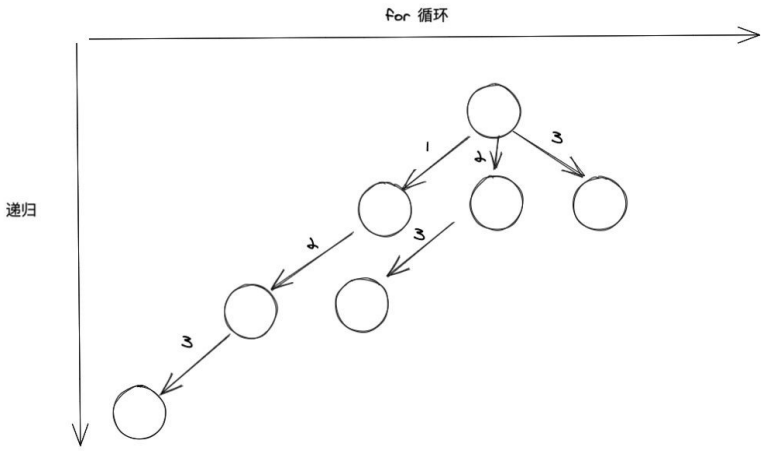
公司

- 暂无

思路

这道题我们可以直接模拟所有可能性，找出最小的即可。

那么如何模拟呢？这里的模拟思路其实和回溯是一样的。我们可以使用递归控制一个变量，递归函数内部控制另外一个变量。



N 叉树的遍历（$N = 3$）

具体来说，我们可以用递归控制当前位置这一变量，递归函数内部循环遍历 jumps。自然语言表达就是**对于每一个位置 pos，我们都可以选择我先走一步（之后怎么走不管）到终点或者先乘坐一个公交车（之后怎么走不管）到终点**。

核心代码:

```
def dfs(pos):
 if pos === target: return 0
 if pos > target: return float('inf')
 ans = (target - pos) * inc
 for jump in jumps:
      ans = min(ans, 乘坐本次公交的花费)
  return ans
dfs(0)
```

上面代码大体思路没有问题。 只是少考虑了一个点**小扣可在移动过程中到达编号大于 target 的站点。** 如果加上这个条件，我们递归到 pos 大于 target 的时候**并不能**认为其是一个非法解。

实际上，我们也可采取逆向思维，即从 target 出发返回 0，这无疑和从 0 出发到 target 的花费是等价的， 但是这样可以简化逻辑。为什么可以简化逻辑呢？是不需要考虑大于 target 了么？当然不是，那样会错过正解。我举个例子你就懂了。 比如：

```
target = 5
jumps = [3]
```

当前的位置 pos = 2，选择乘坐公交车会到达 2 * 3 = 6 。这样往回走一站就可以到达 target 了。如果采用逆向思维如何考虑这一点呢?

逆向思维是从 5 开始。先将 5 / 3 (整数除) 得到 1，余数是 2。 这意味着有两种到达此位置的方式：

- 先想办法到 1，再乘坐本次公交到 3（1 * 3 = 3），然后想办法往前走 2（5 % 3 = 2）.
- 先想办法到 2，再乘坐本次公交到 6（2 * 3 = 6），然后想办法往后走 1. (3 - 5 % 3 = 1)

> 这就考虑到了超过 target 的情况。

特殊地，如果可以整除，我们直接乘坐公交车就行了，无需走路 🚶 。

有的同学可能有疑问，为什么不继续下去。 比如：

- 先想办法到 3，再乘坐本次公交到 9（3 * 3 = 9），然后想办法往后走 1. (3 + 3 - 5 % 3 = 4)
- 。。。

这是没有必要的，因为这些情况一定都比上面两种情况花费更多。

关键点

- 逆向思维

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def busRapidTransit(self, target: int, inc: int, dec: i
        @lru_cache(None)
        def dfs(pos):
            if pos == 0: return 0
            if pos == 1: return inc
            ans = pos * inc
            for i, jump in enumerate(jumps):
                pre_pos, left = pos // jump, pos % jump
                if left == 0: ans = min(ans, cost[i] + dfs(
                else: ans = min(ans, cost[i] + dfs(pre_pos)
            return ans
        return dfs(target) % 1000000007
```

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(LCP 21. 追逐游戏)

https://leetcode-cn.com/problems/Za25hA/

题目描述

秋游中的小力和小扣设计了一个追逐游戏。他们选了秋日市集景区中的 N 个景点

小力和小扣只能沿景点间的小路移动。小力的目标是在最快时间内追到小扣，小

移动至相邻景点
留在原地
如果小力追到小扣（即两人于某一时刻出现在同一位置），则游戏结束。若小力

注意：小力和小扣一定会采取最优移动策略。

示例 1：

输入：edges = [[1,2],[2,3],[3,4],[4,1],[2,5],[5,6]], startA

输出：3

解释：

第一回合，小力移动至 2 号点，小扣观察到小力的行动后移动至 6 号点；
第二回合，小力移动至 5 号点，小扣无法移动，留在原地；
第三回合，小力移动至 6 号点，小力追到小扣。返回 3。

示例 2：

输入：edges = [[1,2],[2,3],[3,4],[4,1]], startA = 1, startB

输出：-1

解释：

小力如果不动，则小扣也不动；否则小扣移动到小力的对角线位置。这样小力无

提示：

edges 的长度等于图中节点个数
3 <= edges.length <= 10^5
1 <= edges[i][0], edges[i][1] <= edges.length 且 edges[i][0
1 <= startA, startB <= edges.length 且 startA != startB

前置知识

- BFS
- DFS
- 图论

公司

- 暂无

思路

为了方便描述，我们将追的人称为 A，被追的人称为 B。

首先，我们需要明确几个前提。

1. 给定 N 个节点， N 条边的图，那么图中有且仅有 1 个环。
2. 如果环的大小等于 3（只要三个节点才能成环），那么无论如何 A 都可以捉到 B。

有了上面的两个前提的话，我们继续来分析。如果环的大小大于 3，那么存在 A 无法追到 B 的可能。这个可能仅在 A 到环的入口的距离大于 B 到环的入口的距离 + 1。如果不满足这个条件，那么 A 一定可以追到 B。

> 之所以 + 1 是因为 A 先走 B 后走。

由于 B 尽量会让自己尽可能晚一点被抓到，那么 B 一定会去一个点，这个点满足：B 比 A 先到。（否则 B 还没到就被抓到了，即根本到不了）。满足条件的点可能不止一个，B 一定会去这些点中最晚被抓住的。最晚被抓住其实就等价于 A 到这个点的距离减去 B 到这个点的距离。由于游戏需要我们返回回合数，那么直接返回 A 到这个点的距离其实就可以了。

分析好了上面的点，基本思路就有了。剩下的问题在于如何通过代码来实现。

首先，我们需要找到图中的环的入口以及环的大小。这可以通过 DFS 来实现，通过扩展参数维护当前节点和父节点的深度信息。具体看代码即可。

其次，我们需要求 A 和 B 到图中所有点的距离，这个可以通过 BFS 来实现。具体看代码即可。

以上两个都是图的基本操作，也就是模板，不再赘述。不过对于检测环的入口来说，这个有点意思。检测环的入口，我们可以通过对 B 做 BFS，当 B 到达第一个环上的节点，就找到了环的入口。有的同学可能会问，如果 B 一开始就在环上呢？实际上，我们可以**认为** B 在的节点就是环的节点， 这对结果并没有影响。

为了更快地找到一个节点的所有邻居，我们需要将题目中给的 edges 矩阵转化为临接矩阵。

关键点

- 明确这道题中有且仅有一个环
- 当且仅当环的长度大于 3，A 到环入口的距离大于 B 到环入口的距离 + 1 才永远追不上
- 如何检测环，如果计算单点到图中所有点的距离

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def chaseGame(self, edges: List[List[int]], startA: int
        n = len(edges)
        graph = collections.defaultdict(list)
        for fr, to in edges:
            graph[fr].append(to)
            graph[to].append(fr)

        def bfs(fr, find_entry=False):
            dist = collections.defaultdict(lambda: float("
            q = collections.deque([fr])
            steps = 0
            nonlocal entry
            while q:
                for i in range(len(q)):
                    cur = q.popleft()
                    if cur in dist:
                        continue
                    if find_entry and cur in circle:
                        entry = cur
                        return
                    dist[cur] = steps
                    for neibor in graph[cur]:
                        q.append(neibor)
                steps += 1
            return dist

        parent = {}
        depth = collections.defaultdict(int)  # 可以被用作 v
        circle = set()
        entry = 0  # 环的入口

        def cal_circle(node, p):
            parent[node] = p
            depth[node] = depth[p] + 1
            for neibor in graph[node]:
                if neibor == p:
                    continue
                if neibor not in depth:
                    cal_circle(neibor, node)
                elif depth[neibor] < depth[node]:
                    # 检测到了环
                    cur = node
                    while cur != neibor:
                        circle.add(cur)
                        cur = parent[cur]
                    circle.add(neibor)
```

```
        cal_circle(1, 0)

        d1, d2 = bfs(startA), bfs(startB)
        bfs(startB, True)

        if len(circle) > 3:
            if d1[entry] > d2[entry] + 1:
                return -1
        if d1[startA] == 1:
            return 1
        ans = 1
        for i in range(1, n + 1):
            if d1[i] - d2[i] > 1:
                ans = max(ans, d1[i])
        return ans
```

参考资料

- 找环，然后分情况讨论

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（820.Number Stream to Intervals）

https://binarysearch.com/problems/Number-Stream-to-Intervals

题目描述

```
Implement a data structure with the following methods:

StreamSummary() constructs a new instance.
add(int val) adds the number val to the instance.
int[][] get() returns a sorted list of disjoint intervals s
Constraints

n ≤ 10,000 where n is the number of calls to add
m ≤ 10,000 where n is the number of calls to get
Example 1
Input
methods = ["constructor", "add", "add", "add", "add", "get"
arguments = [[], [1], [3], [2], [9], []]`
Output
[None, None, None, None, None, [
    [1, 3],
    [9, 9]
]]
Explanation
s = StreamSummary()
s.add(1)
s.add(3)
s.add(2)
s.add(9)
s.get() == [[1, 3], [9, 9]]
Example 2
Input
methods = ["constructor", "add", "add", "add", "add", "get"
arguments = [[], [1], [2], [4], [3], []]`
Output
[None, None, None, None, None, [
    [1, 4]
]]
Explanation
s = StreamSummary()
s.add(1)
s.add(2)
s.add(4)
s.add(3)
s.get() == [[1, 4]]
```

前置知识

- 哈希表
- 有序哈希表
- 二分法

思路

这道题是给我们一个数据流。由于是流，因此不是一次性给我们的。题目的意思是每次 add 都会增加一个 [val, val] 的左右闭合的区间。如果 add 的区间**与左边或者右边能够合并**，我们需要将其合并，get 需要返回合并之后的区间总和。

以题目中的:

```
s.add(1)
s.add(3)
s.add(2)
s.add(9)
```

为例。

我们分步看一下合并后的区间情况。

```
s.add(1) # [ [1,1] ]
s.add(3) # [ [1,1], [3,3] ]
s.add(2) # [ [1,1], [2,2], [3,3] ] 可合并为 [ [1,3] ]
s.add(9) # [ [1,3], [9,9] ]
```

因此这个时候调用 get 会返回 `[ [1,3], [9,9] ]` 。

题目意思就是这样，接下来我们只需要模拟即可。由于每次 add 都需要判断其是否会和前面的区间或者后面的区间进行合并，因此我们可以使用两个哈希表存储。

- 哈希表 start 其中 start[x] 表示以 x 为区间左端点的区间的右端点，也就是说其表示的是区间 [x, start[x]]。
- 哈希表 end 其中 end[x] 表示以 x 为区间右端点的区间的左端点，也就是说其表示的是区间 [end[x], x]。

这样 add 的时候就有四种情况：

- 仅和左边区间结合，也就是说 val - 1 在 end 中。此时 [a,val-1], [val+1,b] 可以和 [val,val] 合并为 [a,b]
- 仅和右边区间结合，也就是说 val + 1 在 start 中.此时 [val+1,b] 可以和 [val,val] 合并为 [val,b]
- 和左右边区间都结合，也就是说 val - 1 在 end 中 且 val + 1 在 start 中.此时 [a,val-1] 可以和 [val,val] 合并为 [a,val]
- 不和左右区间结合

根据上面的四种情况更新 start 和 end 即可。需要注意的是更新了区间（区间合并）之后，需要将原有的区间从哈希表移除，以免影响最终结果。

由于题目说明了 get 返回值需要是升序排序的，而普通的哈希表是乱序的。因此我们需要：

- get 部分对哈希表进行排序之后再返回

这种做法 add 时间复杂度为 $O(1)$，get 时间复杂度为 $mlogm$，m 为合并后的区间个数。

- 使用 SortedDict

由于 SortedDict 内部使用的是平衡树，因此 add 时间复杂度为 $O(logn)$， get 时间复杂度为 $O(m)$，m 为合并后的区间个数。

这两种方法都可以，大家可以根据 add 和 get 的调用频率以及 m 和 n 的大小关系决定使用哪一种。

代码

代码支持：Python3

Python3 Code:

```
from sortedcontainers import SortedDict


class StreamSummary:
    def __init__(self):
        self.start = SortedDict()
        self.end = SortedDict()

    def add(self, val):
        if val - 1 in self.end and val + 1 in self.start:
            # [a, val-1] + [val,val] + [val+1, b] -> [a, b]
            self.end[self.start[val + 1]] = self.end[val -
            self.start[self.end[val - 1]] = self.start[val
            del self.start[val + 1]
            del self.end[val - 1]
        elif val - 1 in self.end:
            # [a, val -1] + [val, val] -> [a, val]
            self.end[val] = self.end[val - 1]
            self.start[self.end[val]] = val
            del self.end[val - 1]
        elif val + 1 in self.start:
            # [val,val] + [val+1, b] -> [val, b]
            self.start[val] = self.start[val + 1]
            self.end[self.start[val]] = val
            del self.start[val + 1]
        else:
            self.start[val] = val
            self.end[val] = val

    def get(self):
        # iterate start or end get same correct answer
        ans = []
        for s, e in self.start.items():
            ans.append([s, e])
        return ans
```

复杂度分析

令 n 为数据流长度，m 为合并后的区间个数。

- 时间复杂度：add 时间复杂度为 $O(logn)$， get 时间复杂度为 $O(m)$
- 空间复杂度：$O(m)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 39K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（762.Number Stream to Intervals）

https://binarysearch.com/problems/Triple-Inversion

题目描述

```
Given a list of integers nums, return the number of pairs i

Constraints

n ≤ 100,000 where n is the length of nums
Example 1
Input
nums = [7, 1, 2]
Output
2
Explanation
We have the pairs (7, 1) and (7, 2)
```

前置知识

- 二分法

暴力法（超时）

思路

本题和力扣 493. 翻转对 和 剑指 Offer 51. 数组中的逆序对 一样，都是求逆序对的换皮题。

暴力的解法可以枚举所有可能的 j，然后往前找 i 使得满足 $nums[i] > nums[j] * 3$，我们要做的就是将满足这种条件的 i 数出来有几个即可。这种算法时间复杂度为 $O(n^2)$。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, A):
        ans = 0
        for i in range(len(A)):
            for j in range(i+1,len(A)):
                if A[i] > A[j] * 3: ans += 1
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(1)$

二分法

思路

这道题我们也可以反向思考。即思考：对于 nums 中的每一项 num，我们找前面出现过的大于 num * 3 的数。

我们可以自己构造有序序列 d，然后在 d 上做二分。如何构建 d 呢？很简单，就是将 nums 中已经遍历过的数字全部放到 d 中即可。

代码表示就是：

```
d = []
for a in A:
    bisect.insort(d, a)
```

bisect.insort 指的是使用二分找到插入点，并将数插入到数组中，使得**插入后数组仍然有序**。虽然使用了二分，使得找到插入点的时间复杂度为 $O(logn)$，但是由于数组的特性，插入导致的数组项后移的时间复杂度为 $O(n)$，因此总的时间复杂度为 $O(n^2)$。

Python3 Code:

```
class Solution:
    def solve(self, A):
        d = []
        ans = 0

        for a in A:
            i = bisect.bisect_right(d, a * 3)
            ans += len(d) - i
            bisect.insort(d, a)
```

由于上面的算法瓶颈在于数组的插入后移带来的时间。因此我们可以使用平衡二叉树来减少这部分时间，使用平衡二叉树可以使得插入时间稳定在 $O(logn)$，Python 可使用 SortedList 来实现， Java 可用 TreeMap 代替。

代码

代码支持：Python3

Python3 Code:

```
from sortedcontainers import SortedList
class Solution:
    def solve(self, A):
        d = SortedList()
        ans = 0

        for a in A:
            i = d.bisect_right(a * 3)
            ans += len(d) - i
            d.add(a)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(n)$

分治法

思路

我们接下来介绍更广泛使用的，效率更高的解法 `分治` 。我们进行一次归并排序，并在归并过程中计算逆序数，换句话说 `逆序对是归并排序的副产物` 。

> 如果不熟悉归并排序，可以先查下相关资料。如果你直接想看归并排序代码，那么将的代码统计 cnt 部分删除就好了。

归并排序实际上会把数组分成两个有序部分，我们不妨称其为左和右，归并排序的过程中会将左右两部分合并成一个有序的部分，对于每一个左右部分，我们分别计算其逆序数，然后全部加起来就是我们要求的逆序数。那么分别如何求解左右部分的逆序数呢?

首先我们知道归并排序的核心在于合并，而合并两个有序数组是一个简单题目。我这里给贴一下大概算法：

```
def mergeTwo(nums1, nums2):
    res = []
    i = j = 0
    while i < len(nums1) and j < len(nums2):
        if nums1[i] < nums[j]:
            res.append(nums[i])
            i += 1
        else:
            res.append(nums[j])
            j += 1
    while i < len(nums1) :
        res.append(num[i])
        i += 1
    while j < len(nums1) :
        res.append(num[j])
        j += 1
    return res
```

而我们要做的就是在上面的合并过程中统计逆序数。

> 为了方便描述，我将题目中的：i < j such that nums[i] > nums[j] * 3，改成 i < j such that nums[i] > nums[j]。也就是将 3 倍变成一倍。如果你理解了这个过程，只需要比较的时候乘以 3 就行，其他逻辑不变。

比如对于左：[1，2，**3**，4]右：[**2**，5]。由于我的算法是按照 [start, mid), [mid, end) 区间分割的，因此这里的 mid 为 3（具体可参考下方代码区）。其中 `i` ， `j` 指针如粗体部分（左数组的 3 和右数组的 2）。那么 逆序数就是 `mid - i + 1` 也就是 `3 - 2 + 1 = 2` 即 `(3, 2)` 和

`(4, 2)` 。其原因在于如果 3 大于 2，那么 3 后面不用看了，肯定都大于 2。 之后会变成：[1，2，**3**，4] 右：[2，**5**] （左数组的 3 和 右数组的 5），继续按照上面的方法计算直到无法进行即可。

```
class Solution:
    def solve(self, nums: List[int]) -> int:
        self.cnt = 0
        def merge(nums, start, mid, end):
            i, j, temp = start, mid + 1, []
            while i <= mid and j <= end:
                if nums[i] <= nums[j]:
                    temp.append(nums[i])
                    i += 1
                else:
                    self.cnt += mid - i + 1
                    temp.append(nums[j])
                    j += 1
            while i <= mid:
                temp.append(nums[i])
                i += 1
            while j <= end:
                temp.append(nums[j])
                j += 1

            for i in range(len(temp)):
                nums[start + i] = temp[i]


        def mergeSort(nums, start, end):
            if start >= end: return
            mid = (start + end) >> 1
            mergeSort(nums, start, mid)
            mergeSort(nums, mid + 1, end)
            merge(nums, start, mid,  end)
        mergeSort(nums, 0, len(nums) - 1)
        return self.cnt
```

注意上述算法在 mergeSort 中我们每次都开辟一个新的 temp，这样空间复杂度大概相当于 NlogN，实际上我们完全每必要每次 mergeSort 都开辟一个新的，而是大家也都用一个。具体见下方代码区。

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def solve(self, nums) -> int:
        self.cnt = 0
        def merge(nums, start, mid, end, temp):
            i, j = start, mid + 1
            while i <= mid and j <= end:
                if nums[i] <=  nums[j]:
                    temp.append(nums[i])
                    i += 1
                else:
                    temp.append(nums[j])
                    j += 1
            # 防住
            # 这里代码开始
            ti, tj = start, mid + 1
            while ti <= mid and tj <= end:
                if nums[ti] <=  3 * nums[tj]:
                    ti += 1
                else:
                    self.cnt += mid - ti + 1
                    tj += 1
            # 这里代码结束
            while i <= mid:
                temp.append(nums[i])
                i += 1
            while j <= end:
                temp.append(nums[j])
                j += 1
            for i in range(len(temp)):
                nums[start + i] = temp[i]
            temp.clear()


        def mergeSort(nums, start, end, temp):
            if start >= end: return
            mid = (start + end) >> 1
            mergeSort(nums, start, mid, temp)
            mergeSort(nums, mid + 1, end, temp)
            merge(nums, start, mid,  end, temp)
        mergeSort(nums, 0, len(nums) - 1, [])
        return self.cnt
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(nlogn)$
- 空间复杂度：$O(n)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（822. Kth-Pair-Distance）

https://binarysearch.com/problems/Kth-Pair-Distance

题目描述

```
Given a list of integers nums and an integer k, return the

Constraints

n ≤ 100,000 where n is the length of nums
Example 1
Input
nums = [1, 5, 3, 2]
k = 3
Output
2
Explanation
Here are all the pair distances:

abs(1 - 5) = 4
abs(1 - 3) = 2
abs(1 - 2) = 1
abs(5 - 3) = 2
abs(5 - 2) = 3
abs(3 - 2) = 1
Sorted in ascending order we have [1, 1, 2, 2, 3, 4].
```

前置知识

- 排序
- 二分法

堆（超时）

思路

堆很适合动态求极值。我在堆的专题中也说了，使用固定堆可求第 k 大的或者第 k 小的数。这道题是求第 k 小的绝对值差。于是可将所有决定值差动态加入到大顶堆中，并保持堆的大小为 k 不变。这样堆顶的就是第 k 小的绝对值差啦。

其实也可用小顶堆保存所有的绝对值差，然后弹出 k 次，最后一次弹出的就是第 k 小的绝对值差啦。

可惜的是，不管使用哪种方法都无法通过。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, A, k):
        A.sort()
        h = [(A[i] - A[i-1], i-1,i) for i in range(1, len(A
        heapq.heapify(h)

        while True:
            top, i, j = heapq.heappop(h)
            if not k: return top
            k -= 1
            if j + 1 < len(A): heapq.heappush(h, (A[j+1] -
```

二分法

思路

这道题是典型的计数二分。

计数二分基本就是求第 k 大（或者第 k 小）的数。其核心思想是找到一个数 x，使得小于等于 x 的数恰好有 k 个。

> 不能看出，有可能答案不止一个

对应到这道题来说就是找到一个绝对值差 diff，使得绝对值差小于等于 diff 的恰好有 k 个。

这种类型是否可用二分解决的关键在于：

如果小于等于 x 的数恰好有 p 个：

1. p 小于 k，那么可舍弃一半解空间
2. p 大于 k，同样可舍弃一半解空间

> 等于你看情况放

简单来说，就是让未知世界无机可乘。无论如何我都可以舍弃一半。

回到这道题，如果小于等于 diff 的绝对值差有大于 k 个，那么 diff 有点 大了，也就是说可以舍弃大于等于 diff 的所有值。反之也是类似，具体大家看代码吧。

最后只剩下两个问题：

- 确定解空间上下界
- 如果计算小于等于 diff 的有即可

第一个问题：下界是 0 ，下界是 max(nums) - min(min)。

第二个问题：可以使用双指针一次遍历解决。大家可以回忆趁此机会回忆一下双指针。具体地，**首先对数组排序**，然后使用右指针 j 和 左指针 i。如果 nums[j] - nums[i] 大于 diff，我们收缩 i 直到 nums[j] - nums[i] <= diff。这个时候，我们就可计算出以索引 j 结尾的绝对值差小于等于 diff 的个数，个数就是 j - i。我们可以使用滑动窗口技巧分别计算所有的 j 的个数，并将其累加起来就是答案。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, A, k):
        A.sort()
        def count_not_greater(diff):
            i = ans = 0
            for j in range(1, len(A)):
                while A[j] - A[i] > diff:
                    i += 1
                ans += j - i
            return ans
        l, r = 0, A[-1] - A[0]

        while l <= r:
            mid = (l + r) // 2
            if count_not_greater(mid) > k:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：由于进行了排序， 因此时间复杂度大约是 $O(nlogn)$

- 空间复杂度：取决于排序的空间消耗

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（796. Minimum Light Radius）

https://binarysearch.com/problems/Minimum-Light-Radius

题目描述

```
You are given a list of integers nums representing coordina

Constraints

n ≤ 100,000 where n is the length of nums
Example 1
Input
nums = [3, 4, 5, 6]
Output
0.5
Explanation
If we place the lamps on 3.5, 4.5 and 5.5 then with r = 0.5
```

前置知识

- 排序
- 二分法

二分法

思路

本题和力扣 475. 供暖器 类似。

这道题含义是给你一个数组 nums，让你在 [min(nums),max(nums)] 范围内放置 3 个灯，每个灯覆盖范围都是 r，让你求最小的 r。

之所以不选择小于 min(nums) 的位置和大于 max(nums) 的位置是因为没有必要。比如选取了小于 min(nums) 的位置 pos，那么选取 pos **一定不比选择 min(nums) 位置好**。

这道题的核心点还是一样的思维模型，即：

- 确定 r 的上下界，这里 r 的下界是 0 上界是 max(nums) - min(nums)。
- 对于上下界之间的所有可能 x 进行枚举（不妨从小到大枚举），检查半径为 x 是否可以覆盖所有，返回第一个可以覆盖所有的 x 即可。

注意到我们是在一个有序序列进行枚举，因此使用二分就应该想到。可使用二分的核心点在于：如果 x 不行，那么小于 x 的所有半径都必然不行。

接下来的问题就是给定一个半径 x，判断其是否可覆盖所有的房子。

这其实就是我们讲义中提到的**能力检测二分**，我定义的函数 possible 就是能力检测。

首先**对 nums 进行排序**，这在后面会用到。 然后从左开始模拟放置灯。先在 nums[0] + r 处放置一个灯，其可以覆盖 [0, 2 *r]。由于 nums 已经排好序了，那么这个等可以覆盖到的房间其实就是 nums 中坐标小于等于 2* \ r 所有房间，使用二分查找即可。对于 nums 右侧的所有的房间我们需要继续放置灯，采用同样的方式即可。

能力检测核心代码：

```
def possible(diameter):
    start = nums[0]
    end = start + diameter
    for i in range(LIGHTS):
        idx = bisect_right(nums, end)
        if idx >= N:
            return True
        start = nums[idx]
        end = start + diameter
    return False
```

由于我们想要找到满足条件的最小值，因此可直接套用**最左二分模板**。

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def solve(self, nums):
        nums.sort()
        N = len(nums)
        if N <= 3:
            return 0
        LIGHTS = 3
        # 这里使用的是直径，因此最终返回需要除以 2
        def possible(diameter):
            start = nums[0]
            end = start + diameter
            for i in range(LIGHTS):
                idx = bisect_right(nums, end)
                if idx >= N:
                    return True
                start = nums[idx]
                end = start + diameter
            return False

        l, r = 0, nums[-1] - nums[0]
        while l <= r:
            mid = (l + r) // 2
            if possible(mid):
                r = mid - 1
            else:
                l = mid + 1
        return l / 2
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：由于进行了排序， 因此时间复杂度大约是 $O(nlogn)$
- 空间复杂度：取决于排序的空间消耗

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（483. Largest Equivalent Set of Pairs）

https://binarysearch.com/problems/Largest-Equivalent-Set-of-Pairs

题目描述

```
Given a list of integers nums, find two sets such that the

Note that not all integers in nums need to be used and the

Constraints

n ≤ 30 where n is the length of nums
0 ≤ nums[i] ≤ 100
Example 1
Input
nums = [1, 4, 3, 5]
Output
5
Explanation
The two sets are [1, 4] and [5].
```

前置知识

- 动态规划

思路

假设题目要求我们找的两个子集分别为 A 和 B。 那么对于一个数来说，我们有三种选择：

- 将其加入 A
- 将其加入 B
- 既不加入 A，也不加入 B

> 不存在既加入 A 又加入 B 的情况。

因此我们要做的就是枚举 nums，对于每个数组执行三种操作。最终枚举完所有的数字之后，如果集合 A 和 集合 B 的和一样的，那么就返回任意一个的和即可。

一个简单的思路是分别维护两个集合的和。实际上，由于我们只关心 A 和 B 的和是否相等，而不关心其具体的值，因此我们可以维护 A 和 B 的差值。当 A 和 B 的差值为 0 的时候，说明 A 和 B 相等。

代码上，我们可以将 A 和 B 的差值 diff 作为参数传进来，而集合 A （或者 B）的和作为返回值。由于我们需要集合 A 的和尽可能大，因此我们可以将上面三种情况的最大值进行返回即可。

大家可以通过**画递归树**来直观感受这种算法。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def solve(self, nums):
        n = len(nums)

        @lru_cache(None)
        def dp(i, diff):
            if i == n:
                return 0 if diff == 0 else float("-inf")
            return max(
                dp(i + 1, diff),
                dp(i + 1, diff - nums[i]),
                dp(i + 1, diff + nums[i]) + nums[i],
            )

        return dp(0, 0)
```

复杂度分析

令 m 为数组长度， n 为最终两个子集的长度的较大者。（因为最坏的情况，我们选取的子集就是较大的）

- 时间复杂度：$O(m * n)$
- 空间复杂度：$O(m * n)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(4. 寻找两个正序数组的中位数)

https://leetcode-cn.com/problems/median-of-two-sorted-arrays/

题目描述

```
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 +

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
```

前置知识

- 中位数
- 分治法
- 二分查找

公司

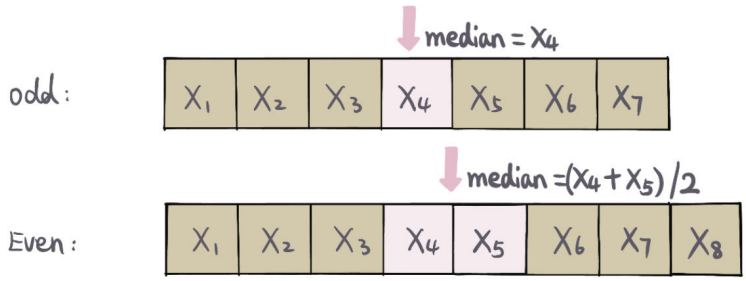
- 阿里
- 百度
- 腾讯

暴力法

思路

首先了解一下 Median 的概念，一个数组中 median 就是把数组分成左右等分的中位数。

如下图：



知道了概念，我们先来看下如何使用暴力法来解决。

> 试了一下，暴力解法也是可以被 Leetcode Accept 的。

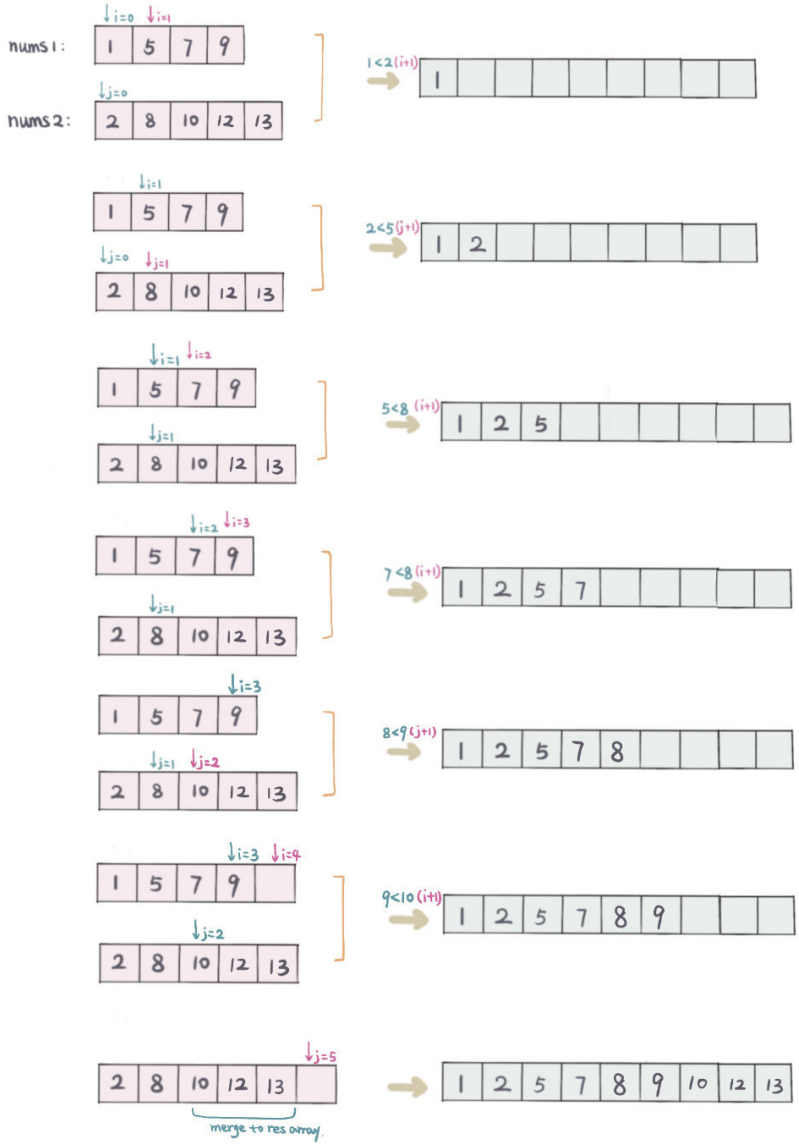
暴力解主要是要 merge 两个排序的数组 （A，B） 成一个排序的数组。

用两个 `pointer（i，j）` ， `i` 从数组 `A` 起始位置开始，即 `i=0` 开始， `j` 从数组 `B` 起始位置， 即 `j=0` 开始. 一一比较 `A[i]` 和 `B[j]` ，

1. 如果 `A[i] <= B[j]` , 则把 `A[i]` 放入新的数组中，i 往后移一位，即 `i+1` .
2. 如果 `A[i] > B[j]` , 则把 `B[j]` 放入新的数组中，j 往后移一位，即 `j+1` .
3. 重复步骤#1 和 #2，直到 `i` 移到 `A` 最后，或者 `j` 移到 `B` 最后。
4. 如果 `j` 移动到 `B` 数组最后，那么直接把剩下的所有 `A` 依次放入新的数组中.
5. 如果 `i` 移动到 `A` 数组最后，那么直接把剩下的所有 `B` 依次放入新的数组中.

> 整个过程类似归并排序的合并过程

Merge 的过程如下图。



时间复杂度和空间复杂度都是 `O(m+n)`，不符合题中给出 `O(log(m+n))` 时间复杂度的要求。

代码

代码支持： Java，JS：

Java Code：

```
class MedianTwoSortedArrayBruteForce {
    public double findMedianSortedArrays(int[] nums1, int[]
      int[] newArr = mergeTwoSortedArray(nums1, nums2);
      int n = newArr.length;
      if (n % 2 == 0) {
        // even
        return (double) (newArr[n / 2] + newArr[n / 2 - 1])
      } else {
        // odd
        return (double) newArr[n / 2];
      }
    }
    private int[] mergeTwoSortedArray(int[] nums1, int[] nu
      int m = nums1.length;
      int n = nums2.length;
      int[] res = new int[m + n];
      int i = 0;
      int j = 0;
      int idx = 0;
      while (i < m && j < n) {
        if (nums1[i] <= nums2[j]) {
          res[idx++] = nums1[i++];
        } else {
          res[idx++] = nums2[j++];
        }
      }
      while (i < m) {
        res[idx++] = nums1[i++];
      }
      while (j < n) {
        res[idx++] = nums2[j++];
      }
      return res;
    }
}
```

JS Code:

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function (nums1, nums2) {
  // 归并排序
  const merged = [];
  let i = 0;
  let j = 0;
  while (i < nums1.length && j < nums2.length) {
    if (nums1[i] < nums2[j]) {
      merged.push(nums1[i++]);
    } else {
      merged.push(nums2[j++]);
    }
  }
  while (i < nums1.length) {
    merged.push(nums1[i++]);
  }
  while (j < nums2.length) {
    merged.push(nums2[j++]);
  }

  const { length } = merged;
  return length % 2 === 1
    ? merged[Math.floor(length / 2)]
    : (merged[length / 2] + merged[length / 2 - 1]) / 2;
};
```

复杂度分析

- 时间复杂度：$O(max(m, n))$
- 空间复杂度：$O(m + n)$

二分查找

思路

如果我们把上一种方法的最终结果拿出来单独看的话，不难发现最终结果就是 nums1 和 nums 两个数组交错形成的新数组，也就是说 nums1 和 nums2 的相对位置并不会发生变化，这是本题的关键信息之一。

为了方便描述，不妨假设最终分割后，数组 nums1 左侧部分是 A，数组 nums2 左侧部分是 B。由于题中给出的数组都是排好序的，在排好序的数组中查找很容易想到可以用二分查找（Binary Search)·, 这里对数组长度

小的做二分以减少时间复杂度。对较小的数组做二分可行的原因在于如果一个数组的索引 i 确定了，那么另一个数组的索引位置 j 也是确定的，因为 (i+1) + (j+1) 等于 (m + n + 1) / 2，其中 m 是数组 A 的长度， n 是数组 B 的长度。具体来说，我们可以保证数组 A 和 数组 B 做 partition 之后， `len(Aleft)+len(Bleft)=(m+n+1)/2`

接下来需要特别注意四个指针：leftp1, rightp1, leftp2, rightp2，分别表示 A 数组分割点，A 数组分割点右侧数，B 数组分割点，B 数组分割点右侧数。不过这里有两个临界点需要特殊处理：

- 如果分割点左侧没有数，即分割点索引是 0，那么其左侧应该设置为无限小。
- 如果分割点右侧没有数，即分割点索引是数组长度-1，那么其左侧应该设置为无限大。

如果我们二分之后满足： `leftp1 < rightp2 and leftp2 < rightp1` ，那么说明分割是正确的，直接返回 `max(leftp1, leftp2)+min(rightp1, rightp2)` 即可。否则，说明分割无效，我们需要调整分割点。

如何调整呢？实际上只需要判断 leftp1 > rightp2 的大小关系即可。如果 leftp1 > rightp2，那么说明 leftp1 太大了，我们可以通过缩小上界来降低 leftp1，否则我们需要扩大下界。

核心代码：

```
if leftp1 > rightp2:
    hi = mid1 - 1
else:
    lo = mid1 + 1
```

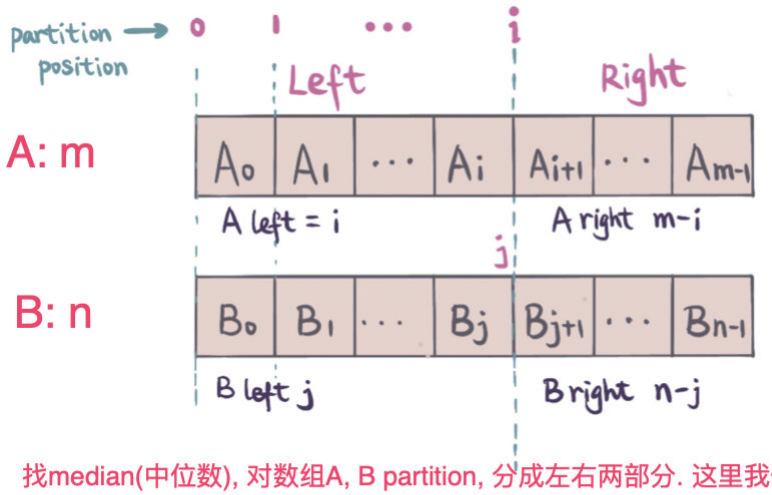
上面的调整上下界的代码是建立在**对数组 nums1 进行二分的基础上的**，如果我们对数组 nums2 进行二分，那么相应地需要改为：

```
if leftp2 > rightp1:
    hi = mid2 - 1
else:
    lo = mid2 + 1
```

下面我们通过一个具体的例子来说明。

比如对数组 A 的做 partition 的位置是区间 `[0,m]`

如图：



找median(中位数), 对数组A, B partition, 分成左右两部分. 这里我们用长度小的数组做二分.(m < n),
lo = 0, hi = m, i = lo + (hi - lo) / 2;
A以 i 做partition,
B以 j 做partition.
满足 i+j=(m+n+1)/2

```
if (Ai <= Bj+1 && Bj <= Ai+1) {
  // odd (奇数)
  if ((m+n) % 2 == 1) {
     median = max(Ai, Bj)
   } else {
      // Even (偶数)
      median = (max(Ai,Bj) + min(Ai+1,Bj+1))/2
   }
}
```

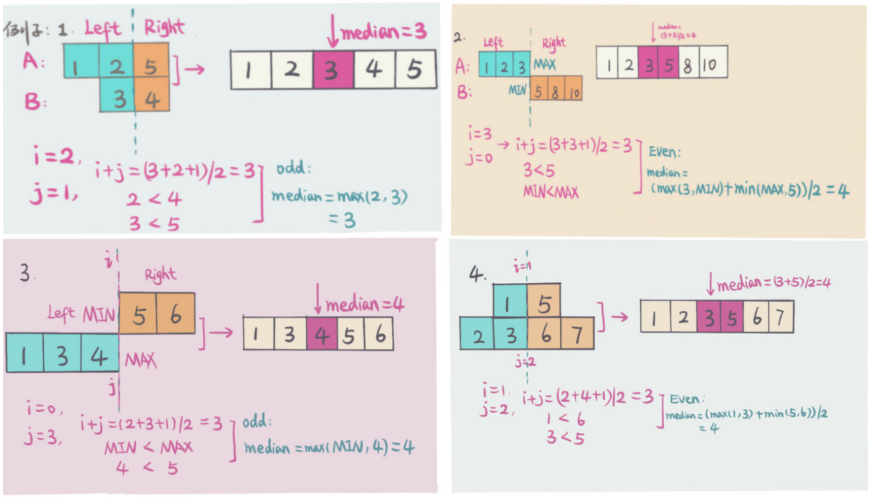
如果Ai>Bj+1, 说明左边的比右边的要大, 要往左半部分shift, hi = i - 1,
对A 区间 [lo, i-1] 二分, 对A, B重新做partition,
I = (lo+hi)/2, j = (m+n+1)/2 - i;

如果Bj>Ai+1, 说明右边的比左边的要大, 要往右半部分shift, lo = i + 1,
对A 区间 [i+1, hi] 二分, 对A, B重新做partition,
i = (lo+hi)/2, j = (m+n+1)/2 - i;

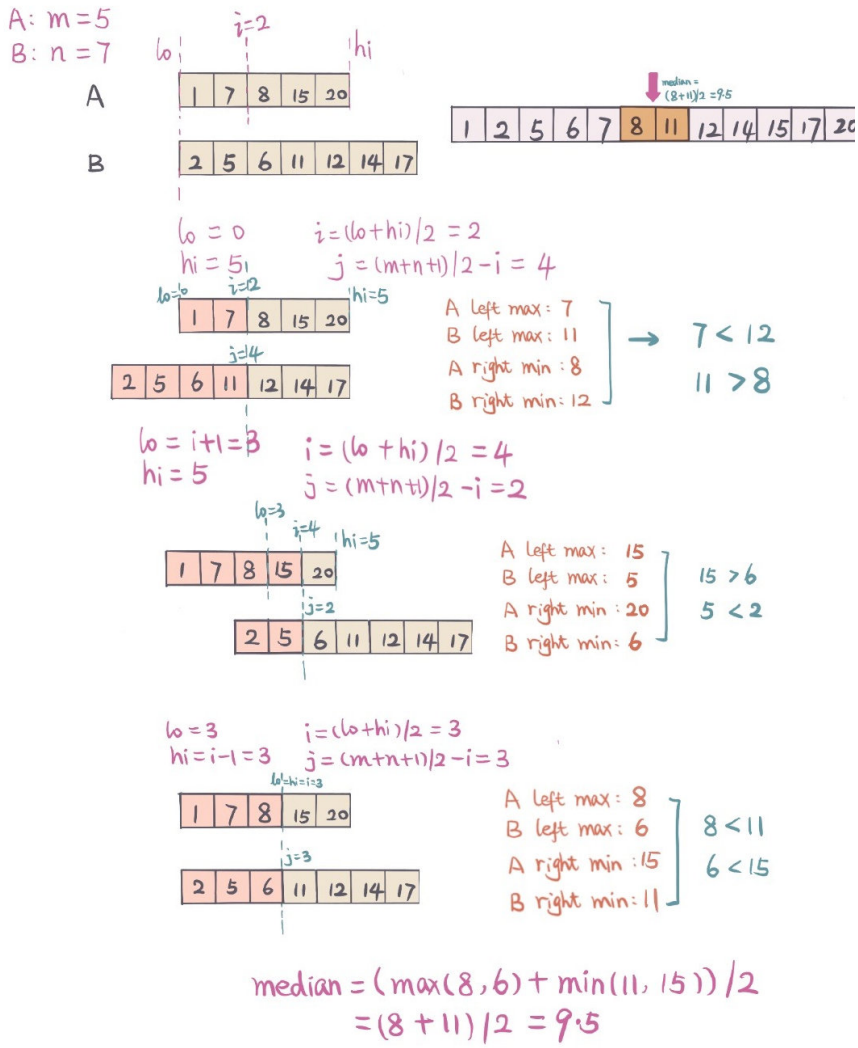
满足 (lo <= hi)条件, 直到找到median.

这里我们对数组小的做二分, 所以时间复杂度是:
O(log(min(m, n))

下图给出几种不同情况的例子（注意但左边或者右边没有元素的时候，左边用 `INF_MIN` ，右边用 `INF_MAX` 表示左右的元素：



下图给出具体做的 partition 解题的例子步骤，



这个算法关键在于：

1. 要 partition 两个排好序的数组成左右两等份，partition 需要满足 `len(Aleft)+len(Bleft)=(m+n+1)/2 - m是数组A的长度， n是数`

`组B的长度` ，

2. 且 partition 后 A 左边最大(`maxLeftA`), A 右边最小（ `minRightA`), B 左边最大（ `maxLeftB`), B 右边最小（ `minRightB`) 满足 `(maxLeftA <= minRightB && maxLeftB <= minRightA)`

关键点分析

- 有序数组容易想到二分查找
- 对小的数组进行二分可降低时间复杂度
- 根据 leftp1,rightp2,leftp2 和 rightp1 的大小关系确定结束点和收缩方向

代码

代码支持：JS，CPP， Python3，

JS Code:

```
/**
 * 二分解法
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number}
 */
var findMedianSortedArrays = function (nums1, nums2) {
  // make sure to do binary search for shorten array
  if (nums1.length > nums2.length) {
    [nums1, nums2] = [nums2, nums1];
  }
  const m = nums1.length;
  const n = nums2.length;
  let low = 0;
  let high = m;
  while (low <= high) {
    const i = low + Math.floor((high - low) / 2);
    const j = Math.floor((m + n + 1) / 2) - i;

    const maxLeftA = i === 0 ? -Infinity : nums1[i - 1];
    const minRightA = i === m ? Infinity : nums1[i];
    const maxLeftB = j === 0 ? -Infinity : nums2[j - 1];
    const minRightB = j === n ? Infinity : nums2[j];

    if (maxLeftA <= minRightB && minRightA >= maxLeftB) {
      return (m + n) % 2 === 1
        ? Math.max(maxLeftA, maxLeftB)
        : (Math.max(maxLeftA, maxLeftB) + Math.min(minRight
    } else if (maxLeftA > minRightB) {
      high = i - 1;
    } else {
      low = low + 1;
    }
  }
};
```

Java Code:

```
class MedianSortedTwoArrayBinarySearch {
  public static double findMedianSortedArraysBinarySearch(
     // do binary search for shorter length array, make su
     if (nums1.length > nums2.length) {
        return findMedianSortedArraysBinarySearch(nums2, nu
      }
      int m = nums1.length;
      int n = nums2.length;
      int lo = 0;
      int hi = m;
      while (lo <= hi) {
        // partition A position i
        int i = lo + (hi - lo) / 2;
        // partition B position j
        int j = (m + n + 1) / 2 - i;

        int maxLeftA = i == 0 ? Integer.MIN_VALUE : nums1[
        int minRightA = i == m ? Integer.MAX_VALUE : nums1

        int maxLeftB = j == 0 ? Integer.MIN_VALUE : nums2[
        int minRightB = j == n ? Integer.MAX_VALUE : nums2

        if (maxLeftA <= minRightB && maxLeftB <= minRightA
          // total length is even
          if ((m + n) % 2 == 0) {
            return (double) (Math.max(maxLeftA, maxLeftB) -
          } else {
            // total length is odd
            return (double) Math.max(maxLeftA, maxLeftB);
          }
        } else if (maxLeftA > minRightB) {
          // binary search left half
          hi = i - 1;
        } else {
          // binary search right half
          lo = i + 1;
        }
      }
      return 0.0;
    }
}
```

CPP Code:

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vecto
        if (nums1.size() > nums2.size()) swap(nums1, nums2)
        int M = nums1.size(), N = nums2.size(), L = 0, R =
        while (true) {
            int i = (L + R) / 2, j = K - i;
            if (i < M && nums2[j - 1] > nums1[i]) L = i + 1
            else if (i > L && nums1[i - 1] > nums2[j]) R =
            else {
                int maxLeft = max(i ? nums1[i - 1] : INT_MI
                if ((M + N) % 2) return maxLeft;
                int minRight = min(i == M ? INT_MAX : nums1
                return (maxLeft + minRight) / 2.0;
            }
        }
    }
};
```

Python3 Code:

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums
        N = len(nums1)
        M = len(nums2)
        if N > M:
            return self.findMedianSortedArrays(nums2, nums1

        lo = 0
        hi = N
        combined = N + M

        while lo <= hi:
            mid1 = lo + hi >> 1
            mid2 = ((combined + 1) >> 1) - mid1

            leftp1 = -float("inf") if mid1 == 0 else nums1
            rightp1 = float("inf") if mid1 == N else nums1

            leftp2 = -float("inf") if mid2 == 0 else nums2
            rightp2 = float("inf") if mid2 == M else nums2

            # Check if the partition is valid for the case
            if leftp1 <= rightp2 and leftp2 <= rightp1:
                if combined % 2 == 0:
                    return (max(leftp1, leftp2)+min(rightp1

                return max(leftp1, leftp2)
            else:
                if leftp1 > rightp2:
                    hi = mid1 - 1
                else:
                    lo = mid1 + 1
        return -1
```

复杂度分析

- 时间复杂度：$O(log(min(m, n)))$
- 空间复杂度：$O(log(min(m, n)))$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（23. 合并 K 个排序链表）

https://leetcode-cn.com/problems/merge-k-sorted-lists/

题目描述

```
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
```

前置知识

- 链表
- 归并排序

公司

- 阿里
- 百度
- 腾讯
- 字节

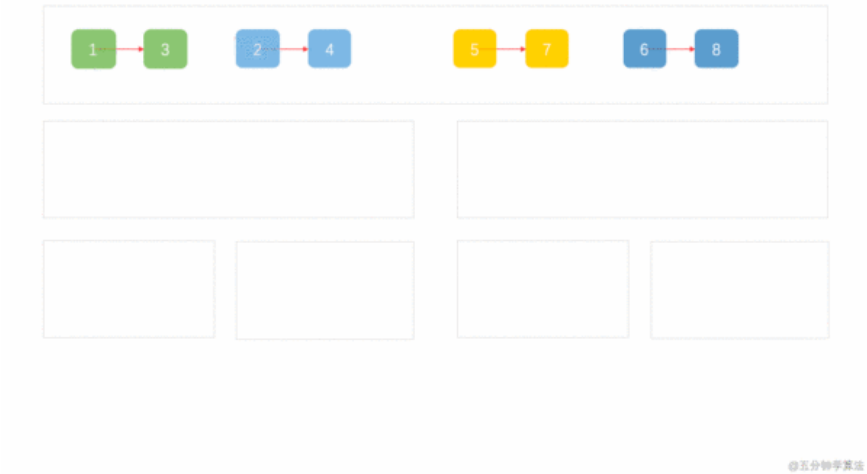
思路

这道题目是合并 k 个已排序的链表，号称 leetcode 目前 `最难` 的链表题。和之前我们解决的88.merge-sorted-array很像。 他们有两点区别：

1. 这道题的数据结构是链表，那道是数组。这个其实不复杂，毕竟都是线性的数据结构。
2. 这道题需要合并 k 个元素，那道则只需要合并两个。这个是两题的关键差别，也是这道题难度为 `hard` 的原因。

因此我们可以看出，这道题目是 `88.merge-sorted-array` 的进阶版本。其实思路也有点像，我们来具体分析下第二条。 如果你熟悉合并排序的话，你会发现它就是 `合并排序的一部分` 。

具体我们可以来看一个动画



（动画来自 https://zhuanlan.zhihu.com/p/61796021 ）

关键点解析

- 分治
- 归并排序(merge sort)

代码

代码支持 JavaScript， Python3, CPP

JavaScript Code：

```
/*
 * @lc app=leetcode id=23 lang=javascript
 *
 * [23] Merge k Sorted Lists
 *
 * https://leetcode.com/problems/merge-k-sorted-lists/descr
 *
 */
function mergeTwoLists(l1, l2) {
  const dummyHead = {};
  let current = dummyHead;
  // l1: 1 -> 3 -> 5
  // l2: 2 -> 4 -> 6
  while (l1 !== null && l2 !== null) {
    if (l1.val < l2.val) {
      current.next = l1; // 把小的添加到结果链表
      current = current.next; // 移动结果链表的指针
      l1 = l1.next; // 移动小的那个链表的指针
    } else {
      current.next = l2;
      current = current.next;
      l2 = l2.next;
    }
  }

  if (l1 === null) {
    current.next = l2;
  } else {
    current.next = l1;
  }
  return dummyHead.next;
}
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function (lists) {
  // 图参考： https://zhuanlan.zhihu.com/p/61796021
  if (lists.length === 0) return null;
  if (lists.length === 1) return lists[0];
  if (lists.length === 2) {
```

```
    return mergeTwoLists(lists[0], lists[1]);
  }

  const mid = lists.length >> 1;
  const l1 = [];
  for (let i = 0; i < mid; i++) {
    l1[i] = lists[i];
  }

  const l2 = [];
  for (let i = mid, j = 0; i < lists.length; i++, j++) {
    l2[j] = lists[i];
  }

  return mergeTwoLists(mergeKLists(l1), mergeKLists(l2));
};
```

Python3 Code：

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def mergeKLists(self, lists: List[ListNode]) -> ListNo
        n = len(lists)

        # basic cases
        if n == 0: return None
        if n == 1: return lists[0]
        if n == 2: return self.mergeTwoLists(lists[0], list

        # divide and conqure if not basic cases
        mid = n // 2
        return self.mergeTwoLists(self.mergeKLists(lists[:m


    def mergeTwoLists(self, l1: ListNode, l2: ListNode) ->
        res = ListNode(0)
        c1, c2, c3 = l1, l2, res
        while c1 or c2:
            if c1 and c2:
                if c1.val < c2.val:
                    c3.next = ListNode(c1.val)
                    c1 = c1.next
                else:
                    c3.next = ListNode(c2.val)
                    c2 = c2.next
                c3 = c3.next
            elif c1:
                c3.next = c1
                break
            else:
                c3.next = c2
                break

        return res.next
```

CPP Code:

```
class Solution {
private:
    ListNode* mergeTwoLists(ListNode* a, ListNode* b) {
        ListNode head(0), *tail = &head;
        while (a && b) {
            if (a->val < b->val) { tail->next = a; a = a->n
            else { tail->next = b; b = b->next; }
            tail = tail->next;
        }
        tail->next = a ? a : b;
        return head.next;
    }
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return NULL;
        for (int N = lists.size(); N > 1; N = (N + 1) / 2)
            for (int i = 0; i < N / 2; ++i) {
                lists[i] = mergeTwoLists(lists[i], lists[N
            }
        }
        return lists[0];
    }
};
```

复杂度分析

- 时间复杂度：$O(kn*logk)$
- 空间复杂度：$O(logk)$

相关题目

- 88.merge-sorted-array

扩展

这道题其实可以用堆来做，感兴趣的同学尝试一下吧。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(25. K 个一组翻转链表)

https://leetcode-cn.com/problems/reverse-nodes-in-k-group/

题目描述

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。

前置知识

- 链表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

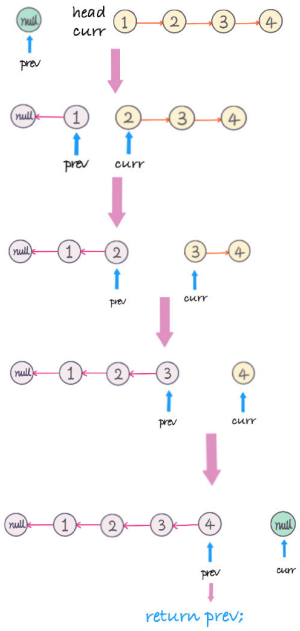
题意是以 `k` 个 nodes 为一组进行翻转，返回翻转后的 `linked list`.

从左往右扫描一遍 `linked list`，扫描过程中，以 k 为单位把数组分成若干段，对每一段进行翻转。给定首尾 nodes，如何对链表进行翻转。

链表的翻转过程，初始化一个为 `null` 的 `previous node (prev)`，然后遍历链表的同时，当前 `node (curr)` 的下一个（next）指向前一个 `node (prev)`，在改变当前 node 的指向之前，用一个临时变量记录当前 node 的下一个 `node (curr.next)`. 即

```
ListNode temp = curr.next;
curr.next = prev;
prev = curr;
curr = temp;
```

举例如图：翻转整个链表 `1->2->3->4->null` -> `4->3->2->1->null`

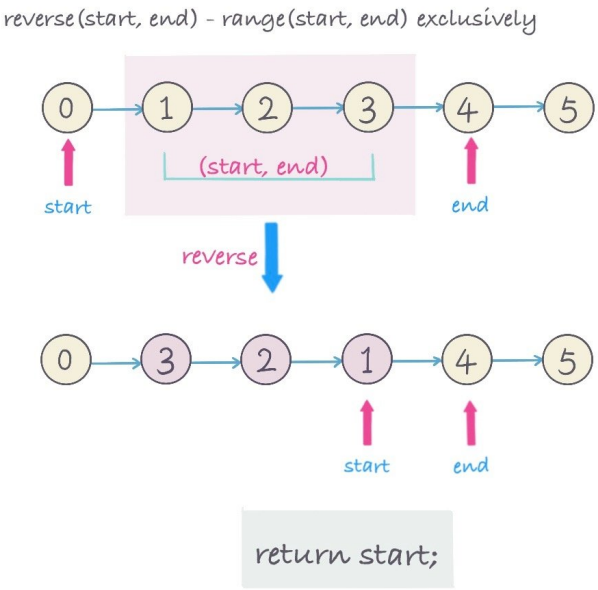


这里是对每一组（`k个nodes`）进行翻转，

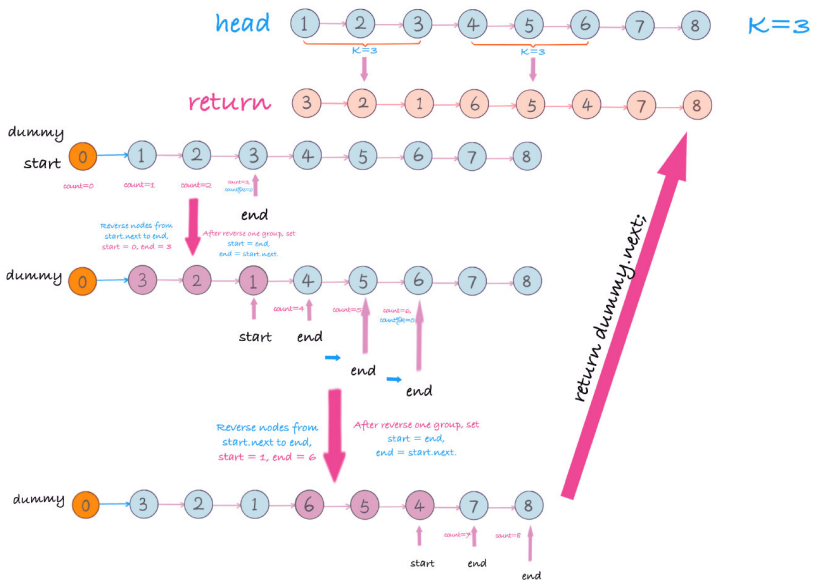
1. 先分组，用一个 `count` 变量记录当前节点的个数
2. 用一个 `start` 变量记录当前分组的起始节点位置的前一个节点
3. 用一个 `end` 变量记录要翻转的最后一个节点位置
4. 翻转一组（`k个nodes`）即 `(start, end) - start and end exclusively`。
5. 翻转后，`start` 指向翻转后链表, 区间 `(start, end)` 中的最后一个节点, 返回 `start` 节点。

6. 如果不需要翻转， `end` 就往后移动一个（ `end=end.next` ），每一次移动，都要 `count+1` .

如图所示 步骤 4 和 5： 翻转区间链表区间 `(start, end)`



举例如图， `head=[1,2,3,4,5,6,7,8], k = 3`



> **NOTE**: 一般情况下对链表的操作，都有可能会引入一个新的 `dummy node` ，因为 `head` 有可能会改变。这里 `head` 从1->3 ， `dummy (List(0))` 保持不变。

复杂度分析

- *时间复杂度:* `O(n) - n is number of Linked List`
- *空间复杂度:* `O(1)`

关键点分析

1. 创建一个 dummy node
2. 对链表以 k 为单位进行分组，记录每一组的起始和最后节点位置
3. 对每一组进行翻转，更换起始和最后的位置
4. 返回 `dummy.next` .

代码 (`Java/Python3/javascript`)

Java Code

```
class ReverseKGroupsLinkedList {
  public ListNode reverseKGroup(ListNode head, int k) {
      if (head == null || k == 1) {
        return head;
      }
      ListNode dummy = new ListNode(0);
      dummy.next = head;

      ListNode start = dummy;
      ListNode end = head;
      int count = 0;
      while (end != null) {
        count++;
        // group
        if (count % k == 0) {
          // reverse linked list (start, end]
          start = reverse(start, end.next);
          end = start.next;
        } else {
          end = end.next;
        }
      }
      return dummy.next;
    }

    /**
     * reverse linked list from range (start, end), return
     * for example:
     * 0->1->2->3->4->5->6->7->8
     * |           |
     * start       end
     *
     * After call start = reverse(start, end)
     *
     * 0->3->2->1->4->5->6->7->8
     *          |  |
     *       start end
     *       first
     *
     */
    private ListNode reverse(ListNode start, ListNode end)
      ListNode curr = start.next;
      ListNode prev = start;
      ListNode first = curr;
      while (curr != end){
        ListNode temp = curr.next;
        curr.next = prev;
        prev = curr;
```

```
            curr = temp;
        }
        start.next = prev;
        first.next = curr;
        return first;
    }
}
```

Python3 Cose

```
class Solution:
    # 翻转一个子链表，并且返回新的头与尾
    def reverse(self, head: ListNode, tail: ListNode, term
        cur = head
        pre = None
        while cur != terminal:
            next = cur.next
            cur.next = pre

            pre = cur
            cur = next
        return tail, head

    def reverseKGroup(self, head: ListNode, k: int) -> List
        ans = ListNode()
        ans.next = head
        pre = ans

        while head:
            tail = pre
            # 查看剩余部分长度是否大于等于 k
            for i in range(k):
                tail = tail.next
                if not tail:
                    return ans.next
            next = tail.next
            head, tail = self.reverse(head, tail, tail.next
            # 把子链表重新接回原链表
            pre.next = head
            tail.next = next
            pre = tail
            head = next

        return ans.next
```

javascript code

```
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
 */
var reverseKGroup = function (head, k) {
  // 标兵
  let dummy = new ListNode();
  dummy.next = head;
  let [start, end] = [dummy, dummy.next];
  let count = 0;
  while (end) {
    count++;
    if (count % k === 0) {
      start = reverseList(start, end.next);
      end = start.next;
    } else {
      end = end.next;
    }
  }
  return dummy.next;

  // 翻转stat -> end的链表
  function reverseList(start, end) {
    let [pre, cur] = [start, start.next];
    const first = cur;
    while (cur !== end) {
      let next = cur.next;
      cur.next = pre;
      pre = cur;
      cur = next;
    }
    start.next = pre;
    first.next = cur;
    return first;
  }
};
```

参考（References)

- Leetcode Discussion (yellowstone)

扩展 1

- 要求从后往前以 `k` 个为一组进行翻转。**(字节跳动（ByteDance）面试题)**

  例子，`1->2->3->4->5->6->7->8, k = 3`，

  从后往前以 `k=3` 为一组，

  - `6->7->8` 为一组翻转为 `8->7->6`，
  - `3->4->5` 为一组翻转为 `5->4->3`.
  - `1->2` 只有 2 个 nodes 少于 `k=3` 个，不翻转。

  最后返回：`1->2->5->4->3->8->7->6`

这里的思路跟从前往后以 `k` 个为一组进行翻转类似，可以进行预处理：

1. 翻转链表
2. 对翻转后的链表进行从前往后以 k 为一组翻转。
3. 翻转步骤 2 中得到的链表。

例子：`1->2->3->4->5->6->7->8, k = 3`

1. 翻转链表得到：`8->7->6->5->4->3->2->1`
2. 以 k 为一组翻转：`6->7->8->3->4->5->2->1`
3. 翻转步骤#2 链表：`1->2->5->4->3->8->7->6`

扩展 2

如果这道题你按照 92.reverse-linked-list-ii 提到的 `p1, p2, p3, p4` （四点法） 的思路来思考的话会很清晰。

代码如下（Python）：

```
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> List
        if head is None or k < 2:
            return head
        dummy = ListNode(0)
        dummy.next = head
        pre = dummy
        cur = head
        count = 0
        while cur:
            count += 1
            if count % k == 0:
                pre = self.reverse(pre, cur.next)
                # end 调到下一个位置
                cur = pre.next
            else:
                cur = cur.next
        return dummy.next
    # (p1, p4)  左右都开放

    def reverse(self, p1, p4):
        prev, curr = p1, p1.next
        p2 = curr
        # 反转
        while curr != p4:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
        # 将反转后的链表添加到原链表中
        # prev 相当于 p3
        p1.next = prev
        p2.next = p4
        # 返回反转前的头,  也就是反转后的尾部
        return p2

# @lc code=end
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 92.reverse-linked-list-ii

- 206.reverse-linked-list

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（30. 串联所有单词的子串）

https://leetcode-cn.com/problems/substring-with-concatenation-of-all-words/

题目描述

```
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 wo

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



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: []
```

前置知识

- 字符串
- 数组
- 哈希表

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

本题是要我们找出 words 中 所有单词按照任意顺序串联 形成的单词中恰好出现在 s 中的索引，因此顺序是不重要的。换句话说，我们只要统计每一个单词的出现情况即可。以题目中 s = "barfoothefoobarman", words = ["foo","bar"] 为例。 我们只需要统计 foo 出现了一次，bar 出现了一次即可。我们只需要在 s 中找到同样包含一次 foo 和一次 bar 的子串即可。由于 words 中的字符串都是等长的，因此编码上也会比较简单。

1. 我们的目标状态是 Counter(words)，即对 words 进行一次计数。
2. 我们只需从头开始遍历一次数组，每次截取 word 长度的字符，一共截取 words 长度次即可。
3. 如果我们截取的 Counter 和 Counter(words)一致，则加入到 res
4. 否则我们继续一个指针，继续执行步骤二
5. 重复执行这个逻辑直到达到数组尾部

关键点解析

- Counter

代码

语言支持：Python3, CPP

Python3 Code:

```python
from collections import Counter


class Solution:
    def findSubstring(self, s: str, words: List[str]) -> Li
        if not s or not words:
            return []
        res = []
        n = len(words)
        word_len = len(words[0])
        window_len = word_len * n
        target = Counter(words)
        i = 0
        while i < len(s) - window_len + 1:
            sliced = []
            start = i
            for _ in range(n):
                sliced.append(s[start:start + word_len])
                start += word_len
            if Counter(sliced) == target:
                res.append(i)
            i += 1
        return res
```

CPP Code:

```
class Solution {
private:
    int len, n;
    string s;
    bool rec(int i, unordered_map<string, int> &m, int cnt)
        if (cnt == n) return true;
        int &v = m[s.substr(i, len)];
        if (v) {
            v--;
            bool ret = rec(i + len, m, cnt + 1);
            v++;
            return ret;
        }
        return false;
    }
public:
    vector<int> findSubstring(string s, vector<string>& wo
        if (words.empty()) return {};
        this->s = s;
        len = words[0].size();
        n = words.size();
        unordered_map<string, int> m;
        for (string word : words) ++m[word];
        int end = s.size() - n * len;
        vector<int> v;
        for (int i = 0; i <= end; ++i) {
            if (rec(i, m, 0)) v.push_back(i);
        }
        return v;
    }
};
```

复杂度分析

其中 N 为 words 中的总字符数。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(32. 最长有效括号)

https://leetcode-cn.com/problems/longest-valid-parentheses/

题目描述

```
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

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 "()()"
```

前置知识

- 动态规划

暴力（超时）

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的做法是：分别计算以 i 开头的 最长有效括号（i 从 0 到 n - 1·），从中取出最大的即可。

代码

代码支持： Python

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        n = len(s)
        ans = 0

        def validCnt(start):
            # cnt 为 ) 的数量减去 ( 的数量
            cnt = 0
            ans = 0
            for i in range(start, n):
                if s[i] == '(':
                    cnt += 1
                if s[i] == ')':
                    cnt -= 1
                if cnt < 0:
                    return i - start
                if cnt == 0:
                    ans = max(ans, i - start + 1)
            return ans
        for i in range(n):
            ans = max(ans, validCnt(i))

        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

栈

思路

主要思路和常规的括号解法一样，遇到'('入栈，遇到')'出栈，并计算两个括号之间的长度。 因为这个题存在非法括号对的情况且求是合法括号对的最大长度 所以有两个注意点是:

1. **栈中存的是符号的下标**
2. **当栈为空时且当前扫描到的符号是')'时，需要将这个符号入栈作为分割符**
3. 栈中初始化一个 -1，作为**分割符**

代码

- 语言支持: Python, javascript, CPP

javascript code:

```
// 用栈来解
var longestValidParentheses = function (s) {
  let stack = new Array();
  let longest = 0;
  stack.push(-1);
  for (let i = 0; i < s.length; i++) {
    if (s[i] === "(") {
      stack.push(i);
    } else {
      stack.pop();
      if (stack.length === 0) {
        stack.push(i);
      } else {
        longest = Math.max(longest, i - stack[stack.length
      }
    }
  }
  return longest;
};
```

Python Code:

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        if not s:
            return 0
        res = 0
        stack = [-1]
        for i in range(len(s)):
            if s[i] == "(":
                stack.append(i)
            else:
                stack.pop()
                if not stack:
                    stack.append(i)
                else:
                    res = max(res, i - stack[-1])
        return res
```

CPP Code:

```
class Solution {
public:
    int longestValidParentheses(string s) {
        stack<int> st;
        st.push(-1);
        int ans = 0;
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == ')' && st.top() != -1 && s[st.top()
                st.pop();
                ans = max(ans, i - st.top());
            } else st.push(i);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

O(1) 空间

思路

我们可以采用解法一中的计数方法。

- 从左到右遍历一次，并分别记录左右括号的数量 left 和 right。
- 如果 right > left ，说明截止上次可以匹配的点到当前点这一段无法匹配，重置 left 和 right 为 0
- 如果 right == left, 此时可以匹配，此时有效括号长度为 left + right，我们获得一个局部最优解。如果其比全局最优解大，我们更新全局最优解

值得注意的是，对形如 `(((()` 这样的，更新全局最优解的逻辑永远无法执行。一种方式是再从右往左遍历一次即可，具体看代码。

> 类似的思想有哨兵元素，虚拟节点。只不过本题无法采用这种方法。

代码

代码支持：Java，Python3, CPP

Java Code:

```java
public class Solution {
    public int longestValidParentheses(String s) {
        int left = 0, right = 0, maxlength = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, left + righ
            }
            if (right > left) {
                left = right = 0;
            }
        }
        left = right = 0;
        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, left + righ
            }
            if (left > right) {
                left = right = 0;
            }
        }
        return maxlength;
    }
}
```

Python3 Code:

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        ans = l = r = 0
        for c in s:
            if c == '(':
                l += 1
            else:
                r += 1
            if l == r:
                ans = max(ans, l + r)
            if r > l:
                l = r = 0
        l = r = 0
        for c in s[::-1]:
            if c == '(':
                l += 1
            else:
                r += 1
            if l == r:
                ans = max(ans, l + r)
            if r < l:
                l = r = 0

        return ans
```

CPP Code:

```
class Solution {
public:
    int longestValidParentheses(string s) {
        int left = 0, right = 0, ans = 0, N = s.size();
        for (int i = 0; i < N; ++i) {
            left += s[i] == '(';
            right += s[i] == ')';
            if (left == right) ans = max(ans, left + right)
            else if (right > left) left = right = 0;
        }
        left = 0, right = 0;
        for (int i = N - 1; i >= 0; --i) {
            left += s[i] == '(';
            right += s[i] == ')';
            if (left == right) ans = max(ans, left + right)
            else if (left > right) left = right = 0;
        }
        return ans;
    }
};
```

动态规划

思路

所有的动态规划问题, 首先需要解决的就是如何寻找合适的子问题. 该题需要我们找到最长的有效括号对, 我们首先想到的就是定义**dp[i]为前 i 个字符串的最长有效括号对长度**, 但是随后我们会发现, 这样的定义, 我们无法找到 dp[i]和 dp[i-1]的任何关系. 所以, 我们需要重新找一个新的定义: 定义**dp[i]为以第 i 个字符结尾的最长有效括号对长度**. 然后, 我们通过下面这个例子找一下 dp[i]和 dp[i-1]之间的关系.

```
s = '(()())'
```

从上面的例子我们可以观察出一下几点结论(**描述中 i 为图中的 dp 数组的下标, 对应 s 的下标应为 i-1, 第 i 个字符的 i 从 1 开始**).

1. base case: 空字符串的最长有效括号对长度肯定为 0, 即: dp[0] = 0;
2. s 的第**1**个字符结尾的最长有效括号对长度为 0, s 的第**2**个字符结尾的最长有效括号对长度也为 0, 这个时候我们可以得出结论: 最长有效括号对不可能以'('结尾, 即: dp[1] = d[2] = 0;
3. 当 i 等于 3 时, 我们可以看出 dp[2]=0, dp[3]=2, 因为第 2 个字符(**s[1]**)和第 3 个字符(**s[2]**)是配对的; 当 i 等于 4 时, 我们可以看出 dp[3]=2, dp[4]=4, 因为我们配对的是第 1 个字符(**s[0]**)和第 4 个字符(**s[3]**); 因

此, 我们可以得出结论: 如果第**i**个字符和第**i-1-dp[i-1]**个字符是配对的, 则 dp[i] = dp[i-1] + 2, 其中: i-1-dp[i-1] >= 1, 因为第 0 个字符没有任何意义;

4. 根据第 3 条规则来计算的话, 我们发现 dp[5]=0, dp[6]=2, 但是显然, dp[6]应该为 6 才对, 但是我们发现可以将"(())"和"()"进行拼接, 即: dp[i] += dp[i-dp[i]], 即: dp[6] = 2 + dp[6-2] = 2 + dp[4] = 6

根据以上规则, 我们求解 dp 数组的结果为: [0, 0, 0, 2, 4, 0, 6, 0], 其中最长有效括号对的长度为 6. 以下为图解:

代码

代码支持：Python3， CPP

Python3 Code:

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        mlen = 0
        slen = len(s)
        dp = [0] * (slen + 1)
        for i in range(1, len(s) + 1):
            # 有效的括号对不可能会以'('结尾的
            if s[i - 1] == '(':
                continue

            left_paren = i - 2 - dp[i - 1]
            if left_paren >= 0 and s[left_paren] == '(':
                dp[i] = dp[i - 1] + 2

                # 拼接有效括号对
                if dp[i - dp[i]]:
                    dp[i] += dp[i - dp[i]]

                # 更新最大有效扩对长度
                if dp[i] > mlen:
                    mlen = dp[i]

        return mlen
```

CPP Code:

```cpp
class Solution {
public:
    int longestValidParentheses(string s) {
        vector<int> dp(s.size() + 1, 0);
        int ans = 0;
        for (int i = 0; i < s.size(); ++i) {
            if (s[i] == '(') continue;
            int start = i - dp[i] - 1;
            if (start >= 0 && s[start] == '(')
                dp[i + 1] = dp[i] + 2 + dp[start];
            ans = max(ans, dp[i + 1]);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

关键点解析

1. 第 3 点特征, 需要检查的字符是 s[i-1]和 s[i-2-dp[i-1]], 根据定义可知: i-1 >= dp[i-1], 但是 i-2 不一定大于 dp[i-1], 因此, 需要检查越界;
2. 第 4 点特征最容易遗漏, 还有就是不需要检查越界, 因为根据定义可知: i >= dp[i], 所以 dp[i-dp[i]]的边界情况是 dp[0];

相关题目

- 20.valid-parentheses

扩展

1. 如果判断的不仅仅只有'('和')', 还有'[', ']', '{'和'}', 该怎么办?
2. 如果输出的不是长度, 而是任意一个最长有效括号对的字符串, 该怎么办?

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(42. 接雨水)

https://leetcode-cn.com/problems/trapping-rain-water/

题目描述

> 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，

> 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
>
> 示例:
>
> 输入: [0,1,0,2,1,0,1,3,2,1,2,1]
> 输出: 6

前置知识

- 空间换时间
- 双指针
- 单调栈

公司

- 阿里
- 腾讯
- 百度
- 字节

双数组

思路

这是一道雨水收集的问题， 难度为 `hard` . 如图所示，让我们求下过雨之后最多可以积攒多少的水。

如果采用暴力求解的话，思路应该是 height 数组依次求和，然后相加。

- 伪代码

```
for (let i = 0; i < height.length; i++) {
  area += (h[i] - height[i]) * 1; // h为下雨之后的水位
}
```

问题转化为求 h，那么 h[i]又等于 `左右两侧柱子的最大值中的较小值` ， 即 `h[i] = Math.min(左边柱子最大值, 右边柱子最大值)`

如上图那么 h 为 [0, 1, 1, 2, 2, 2 ,2, 3, 2, 2, 2, 1]

问题的关键在于求解 `左边柱子最大值` 和 `右边柱子最大值` , 我们其实可以用两个数组来表示 `leftMax` ， `rightMax` ， 以 leftMax 为例，leftMax[i]代表 i 的左侧柱子的最大值，因此我们维护两个数组即可。

关键点解析

- 建模 `h[i] = Math.min(左边柱子最大值, 右边柱子最大值)` (h 为下雨之后的水位)

代码

- 代码支持: JS, Python3, C++:

JS Code:

```
/*
 * @lc app=leetcode id=42 lang=javascript
 *
 * [42] Trapping Rain Water
 *
 */
/**
 * @param {number[]} height
 * @return {number}
 */
var trap = function (height) {
  let max = 0;
  let volume = 0;
  const leftMax = [];
  const rightMax = [];

  for (let i = 0; i < height.length; i++) {
    leftMax[i] = max = Math.max(height[i], max);
  }

  max = 0;

  for (let i = height.length - 1; i >= 0; i--) {
    rightMax[i] = max = Math.max(height[i], max);
  }

  for (let i = 0; i < height.length; i++) {
    volume = volume + Math.min(leftMax[i], rightMax[i]) - h
  }

  return volume;
};
```

Python Code:

```python
class Solution:
    def trap(self, heights: List[int]) -> int:
        n = len(heights)
        l, r = [0] * n, [0] * n
        ans = 0
        for i in range(1, len(heights)):
            l[i] = max(l[i - 1], heights[i - 1])
        for i in range(len(heights) - 2, 0, -1):
            r[i] = max(r[i + 1], heights[i + 1])
        for i in range(len(heights)):
            ans += max(0, min(l[i], r[i]) - heights[i])
        return ans
```

C++ Code:

```cpp
int trap(vector<int>& heights)
{
    if(heights == null)
        return 0;
    int ans = 0;
    int size = heights.size();
    vector<int> left_max(size), right_max(size);
    left_max[0] = heights[0];
    for (int i = 1; i < size; i++) {
        left_max[i] = max(heights[i], left_max[i - 1]);
    }
    right_max[size - 1] = heights[size - 1];
    for (int i = size - 2; i >= 0; i--) {
        right_max[i] = max(heights[i], right_max[i + 1]);
    }
    for (int i = 1; i < size - 1; i++) {
        ans += min(left_max[i], right_max[i]) - heights[i];
    }
    return ans;
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

双指针

思路

上面代码比较好理解，但是需要额外的 N 的空间。从上面解法可以看出，我们实际上只关心左右两侧较小的那一个，并不需要两者都计算出来。具体来说：

- 如果 l[i + 1] < r[i] 那么 最终积水的高度由 i 的左侧最大值决定。
- 如果 l[i + 1] >= r[i] 那么 最终积水的高度由 i 的右侧最大值决定。

因此我们不必维护完整的两个数组，而是可以只进行一次遍历，同时维护左侧最大值和右侧最大值，使用常数变量完成即可。这是一个典型的双指针问题，

具体算法：

1. 维护两个指针 left 和 right，分别指向头尾。
2. 初始化左侧和右侧最高的高度都为 0。
3. 比较 height[left] 和 height[right]
    - 3.1 如果 height[left] < height[right], 那么瓶颈在于 height[left]，不需要考虑 height[right]
        - 3.1.1 如果 height[left] < left_max， 则当前格子积水面积为 (left_max - height[left])，否则无法积水，即积水面积为 0。也可将逻辑统一为盛水量为 max(0, left_max - height[left])
        - 3.1.2 左指针右移一位。（其实就是左指针的位置的雨水量已经计算完成了，我们移动到下个位置用同样的方法计算）
    - 3.2 否则 瓶颈在于 height[right]，不需要考虑 height[left]
        - 3.2.1 如果 height[right] < right_max, 则当前格子积水面积为 (right_max - height[left])，否则无法积水，即积水面积为 0。也可将逻辑统一为盛水量为 max(0, right_max - height[right])
        - 3.2.2 右指针右移一位。（其实就是右指针的位置的雨水量已经计算完成了，我们移动到下个位置用同样的方法计算）

代码

- 代码支持: Python, C++, Go, PHP:

Python Code:

```python
class Solution:
    def trap(self, heights: List[int]) -> int:
        n = len(heights)
        l_max = r_max = 0
        l, r = 0, n - 1
        ans = 0
        while l < r:
            if heights[l] < heights[r]:
                if heights[l] < l_max:
                    ans += l_max - heights[l]
                else:
                    l_max = heights[l]
                l += 1
            else:
                if heights[r] < r_max:
                    ans += r_max - heights[r]
                else:
                    r_max = heights[r]
                r -= 1
        return ans
```

C++ Code:

```cpp
class Solution {
public:
    int trap(vector<int>& heights)
{
    int left = 0, right = heights.size() - 1;
    int ans = 0;
    int left_max = 0, right_max = 0;
    while (left < right) {
        if (heights[left] < heights[right]) {
            heights[left] >= left_max ? (left_max = heights
            ++left;
        }
        else {
            heights[right] >= right_max ? (right_max = heig
            --right;
        }
    }
    return ans;
}

};
```

Go Code:

```go
func trap(height []int) int {
    if len(height) == 0 {
        return 0
    }

    l, r := 0, len(height)-1
    lMax, rMax := height[l], height[r]
    ans := 0
    for l < r {
        if height[l] < height[r] {
            if height[l] < lMax {
                ans += lMax - height[l]
            } else {
                lMax = height[l]
            }
            l++
        } else {
            if height[r] < rMax {
                ans += rMax - height[r]
            } else {
                rMax = height[r]
            }
            r--
        }
    }
    return ans
}
```

PHP Code:

```php
class Solution
{

    /**
     * @param Integer[] $height
     * @return Integer
     */
    function trap($height)
    {
        $n = count($height);
        if (!$n) return 0;

        $l = 0;
        $l_max = $height[$l];
        $r = $n - 1;
        $r_max = $height[$r];
        $ans = 0;
        while ($l < $r) {
            if ($height[$l] < $height[$r]) {
                if ($height[$l] < $l_max) $ans += $l_max -
                else $l_max = $height[$l];
                $l++;
            } else {
                if ($height[$r] < $r_max) $ans += $r_max-$h
                else $r_max = $height[$r];
                $r--;
            }
        }
        return $ans;
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

相关题目

- 84.largest-rectangle-in-histogram

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

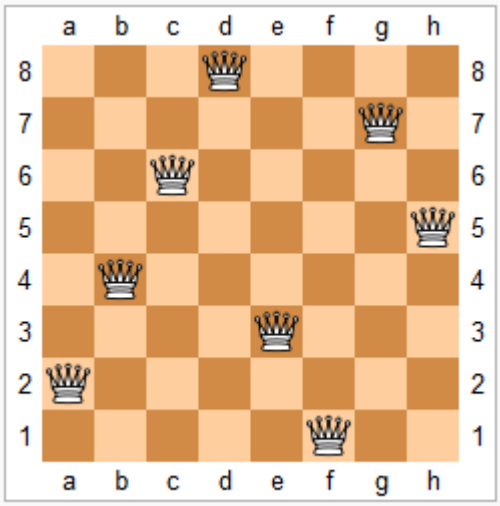
欢迎长按关注

题目地址(52. N皇后 II)

https://leetcode-cn.com/problems/n-queens-ii/

题目描述

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

给定一个整数 n，返回 n 皇后不同的解决方案的数量。

示例:

输入: 4
输出: 2
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.."]
]
```

前置知识

- 回溯
- 深度优先遍历

公司

- 阿里
- 百度
- 字节

思路

使用深度优先搜索配合位运算，二进制为 1 代表不可放置，0 相反

利用如下位运算公式：

- x & -x：得到最低位的 1 代表除最后一位 1 保留，其他位全部为 0
- x & (x-1)：清零最低位的 1 代表将最后一位 1 变成 0
- x & ((1 << n) - 1)：将 x 最高位至第 n 位(含)清零

关键点

- 位运算
- DFS（深度优先搜索）

代码

- 语言支持：JS

```
/**
 * @param {number} n
 * @return {number}
 * @param row 当前层
 * @param cols 列
 * @param pie 左斜线
 * @param na 右斜线
 */
const totalNQueens = function (n) {
    let res = 0;
    const dfs = (n, row, cols, pie, na) => {
        if (row >= n) {
            res++;
            return;
        }
        // 将所有能放置 Q 的位置由 0 变成 1，以便进行后续的位遍历
        // 也就是得到当前所有的空位
        let bits = (~(cols | pie | na)) & ((1 << n) - 1);
        while (bits) {
            // 取最低位的1
            let p = bits & -bits;
            // 把P位置上放入皇后
            bits = bits & (bits - 1);
            // row + 1 搜索下一行可能的位置
            // cols | p 目前所有放置皇后的列
            // (pie | p) << 1 和 (na | p) >> 1) 与已放置过皇后
            dfs(n, row + 1, cols | p, (pie | p) << 1, (na
        }
    }
    dfs(n, 0, 0, 0, 0);
    return res;
};
```

复杂度分析

- 时间复杂度：O(N!)
- 空间复杂度：O(N)

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(57. 插入区间)

https://leetcode-cn.com/problems/insert-interval/

题目描述

给出一个无重叠的 ，按照区间起始端点排序的区间列表。

在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果

示例 1:

输入: intervals = [[1,3],[6,9]], newInterval = [2,5]
输出: [[1,5],[6,9]]
示例 2:

输入: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInt
输出: [[1,2],[3,10],[12,16]]
解释: 这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。

注意：输入类型已在 2019 年 4 月 15 日更改。请重置为默认代码定义以获取

前置知识

- 排序

公司

- 阿里
- 腾讯
- 百度
- 字节

排序

思路

一个简单的思路是将 intervals 和 newInterval 合并成一个数组并排序，那么问题就和 56. 合并区间 类似，而56. 合并区间 是一个中等题，思路参考 56. 合并区间 即可。

代码

- 语言支持: Python3

```
class Solution:
    def insert(self, intervals: List[List[int]], newInterva
        intervals.append(newInterval)
        intervals.sort(key=lambda a: a[0])

        def intersected(a, b):
            if a[0] > b[1] or a[1] < b[0]:
                return False
            return True

        def mergeTwo(a, b):
            return [min(a[0], b[0]), max(a[1], b[1])]

        i = 0
        while i < len(intervals) - 1:
            cur = intervals[i]
            next = intervals[i + 1]
            if intersected(cur, next):
                intervals[i] = None
                intervals[i + 1] = mergeTwo(cur, next)
            i += 1

        return list(filter(lambda x: x, intervals))
```

复杂度分析

- 时间复杂度：由于采用了排序，因此复杂度大概为 $O(NlogN)$
- 空间复杂度：$O(1)$

一次扫描

思路

由于没有卡时间复杂度，因此上面的代码也不会超时。但是实际的面试可能并不会通过，我们必须考虑线性时间复杂度的做法。

> 力扣有很多测试用例卡的不好的题目，以至于暴力法都可以过，但是大家不要抱有侥幸心理，否则真真实面试的时候会后悔。

newInterval 相对于 intervals 的位置关系有多种：

- newInterval 在左侧
- newInterval 在右侧
- newInterval 在中间

而 newInterval 在中间又分为相交和不相交，看起来比较麻烦，实际却不然。来看下我的具体算法。

算法描述：

- 从左往右遍历，对于遍历到的每一项姑且称之为 interval。
    - 如果 interval 在 newInterval 左侧不相交，那么不断 push interval 到 ans。
    - 否则不断合并 interval 和 newInterval，直到合并之后的新区间和 interval 不重合，将合并后的**大区间** push 到 ans。融合的方法参考上方 56 题。
    - 后面不可能发生重合了（题目保证了），因此直接将后面的 interval 全部添加到 ans 即可
- 最终返回 ans

代码

- 语言支持: Python3

```
class Solution:
    def insert(self, intervals: List[List[int]], newInterva
        i, n = 0, len(intervals)
        ans = []

        def intersected(a, b):
            if a[0] > b[1] or a[1] < b[0]:
                return False
            return True
        # 前
        while i < n and intervals[i][1] < newInterval[0]:
            ans.append(intervals[i])
            i += 1
        # 中
        while i < n and intersected(intervals[i], newInterv
            newInterval = [min(intervals[i][0], newInterva
                           max(intervals[i][1], newInterva
            i += 1
        ans.append(newInterval)
        # 后
        while i < n:
            ans.append(intervals[i])
            i += 1
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

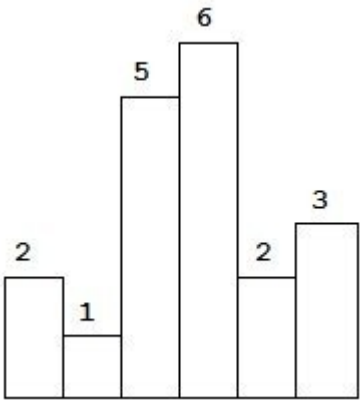
题目地址（84. 柱状图中最大的矩形）

https://leetcode-cn.com/problems/largest-rectangle-in-histogram/

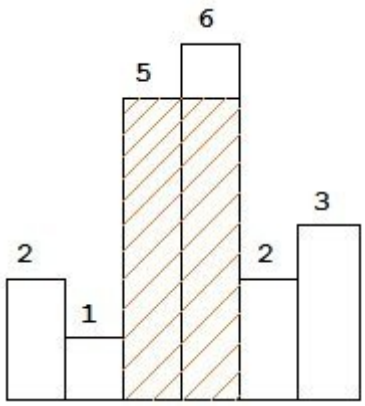
题目描述

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

```
示例:

输入: [2,1,5,6,2,3]
输出: 10
```

公司

- 阿里

- 腾讯
- 百度
- 字节

前置知识

- 单调栈

暴力枚举 - 左右端点法（TLE）

思路

我们暴力尝试 `所有可能的矩形` 。由于矩阵是二维图形， 我我们可以使用 `左右两个端点来唯一确认一个矩阵` 。因此我们使用双层循环枚举所有的可能性即可。 而矩形的面积等于 `（右端点坐标 - 左端点坐标 + 1）* 最小的高度` ，最小的高度我们可以在遍历的时候顺便求出。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> i
        n, ans = len(heights), 0
        if n != 0:
            ans = heights[0]
        for i in range(n):
            height = heights[i]
            for j in range(i, n):
                height = min(height, heights[j])
                ans = max(ans, (j - i + 1) * height)
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(1)$

暴力枚举 - 中心扩展法（TLE）

思路

我们仍然暴力尝试 `所有可能的矩形` 。只不过我们这一次从中心向两边进行扩展。对于每一个 i，我们计算出其左边第一个高度小于它的索引 p，同样地，计算出右边第一个高度小于它的索引 q。那么以 i 为最低点能够

构成的面积就是 `(q - p - 1) * heights[i]` 。这种算法毫无疑问也是正确的。我们证明一下，假设 f(i) 表示求以 i 为最低点的情况下，所能形成的最大矩阵面积。那么原问题转化为 `max(f(0), f(1), f(2), ..., f(n - 1))` 。

具体算法如下：

- 我们使用 l 和 r 数组。l[i] 表示 左边第一个高度小于它的索引，r[i] 表示 右边第一个高度小于它的索引。
- 我们从前往后求出 l，再从后往前计算出 r。
- 再次遍历求出所有的可能面积，并取出最大的。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n = len(heights)
        l, r, ans = [-1] * n, [n] * n, 0
        for i in range(1, n):
            j = i - 1
            while j >= 0 and heights[j] >= heights[i]:
                j -= 1
            l[i] = j
        for i in range(n - 2, -1, -1):
            j = i + 1
            while j < n and heights[j] >= heights[i]:
                j += 1
            r[i] = j
        for i in range(n):
            ans = max(ans, heights[i] * (r[i] - l[i] - 1))
        return ans
```

复杂度分析

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(N)$

优化中心扩展法（Accepted）

思路

实际上我们内层循环没必要一步一步移动，我们可以直接将 `j -= 1` 改成 `j = l[j]` ，`j += 1` 改成 `j = r[j]` 。

代码

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> 
        n = len(heights)
        l, r, ans = [-1] * n, [n] * n, 0

        for i in range(1, n):
            j = i - 1
            while j >= 0 and heights[j] >= heights[i]:
                j = l[j]
            l[i] = j
        for i in range(n - 2, -1, -1):
            j = i + 1
            while j < n and heights[j] >= heights[i]:
                j = r[j]
            r[i] = j
        for i in range(n):
            ans = max(ans, heights[i] * (r[i] - l[i] - 1))
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

单调栈（Accepted）

思路

实际上，读完第二种方法的时候，你应该注意到了。我们的核心是求左边第一个比 i 小的和右边第一个比 i 小的。 如果你熟悉单调栈的话，那么应该会想到这是非常适合使用单调栈来处理的场景。

从左到右遍历柱子，对于每一个柱子，我们想找到第一个高度小于它的柱子，那么我们就可以使用一个单调递增栈来实现。 如果柱子大于栈顶的柱子，那么说明不是我们要找的柱子，我们把它塞进去继续遍历，如果比栈顶小，那么我们就找到了第一个小于的柱子。 **对于栈顶元素，其右边第一个小于它的就是当前遍历到的柱子，左边第一个小于它的就是栈中下一个要被弹出的元素**，因此以当前栈顶为最小柱子的面积为**当前栈顶的柱子高度 * (当前遍历到的柱子索引 - 1 - (栈中下一个要被弹出的元素索引 + 1) + 1)**，化简一下就是 **当前栈顶的柱子高度 * (当前遍历到的柱子索引 - 栈中下一个要被弹出的元素索引 - 1)**。

这种方法只需要遍历一次，并用一个栈。由于每一个元素最多进栈出栈一次，因此时间和空间复杂度都是$O(N)$。

为了统一算法逻辑，减少边界处理，我在 heights 首尾添加了两个哨兵元素，**这样我们可以保证所有的柱子都会出栈。**

代码

代码支持： Python,CPP

Python Code:

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> i
        n, heights, st, ans = len(heights), [0] + heights +
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                ans = max(ans, heights[st.pop(-1)] * (i - s
            st.append(i)
        return ans
```

CPP Code:

```
class Solution {
public:
    int largestRectangleArea(vector<int>& A) {
        A.push_back(0);
        int N = A.size(), ans = 0;
        stack<int> s;
        for (int i = 0; i < N; ++i) {
            while (s.size() && A[s.top()] >= A[i]) {
                int h = A[s.top()];
                s.pop();
                int j = s.size() ? s.top() : -1;
                ans = max(ans, h * (i - j - 1));
            }
            s.push(i);
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

2020-05-30 更新：

有的观众反应看不懂为啥需要两个哨兵。 其实末尾的哨兵就是为了将栈清空，防止遍历完成栈中还有没参与运算的数据。

而前面的哨兵有什么用呢？ 我这里把上面代码进行了拆解：

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> i
        n, heights, st, ans = len(heights),[0] + heights +
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                a = heights[st[-1]]
                st.pop()
                # 如果没有前面的哨兵，这里的 st[-1] 可能会越界。
                ans = max(ans, a * (i - 1 - st[-1]))
            st.append(i)
        return ans
```

相关题目

- 42.trapping-rain-water

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（85. 最大矩形）

https://leetcode-cn.com/problems/maximal-rectangle/

题目描述

给定一个仅包含 0 和 1 的二维二进制矩阵，找出只包含 1 的最大矩形，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：6

前置知识

- 单调栈

公司

- 阿里
- 腾讯
- 百度
- 字节

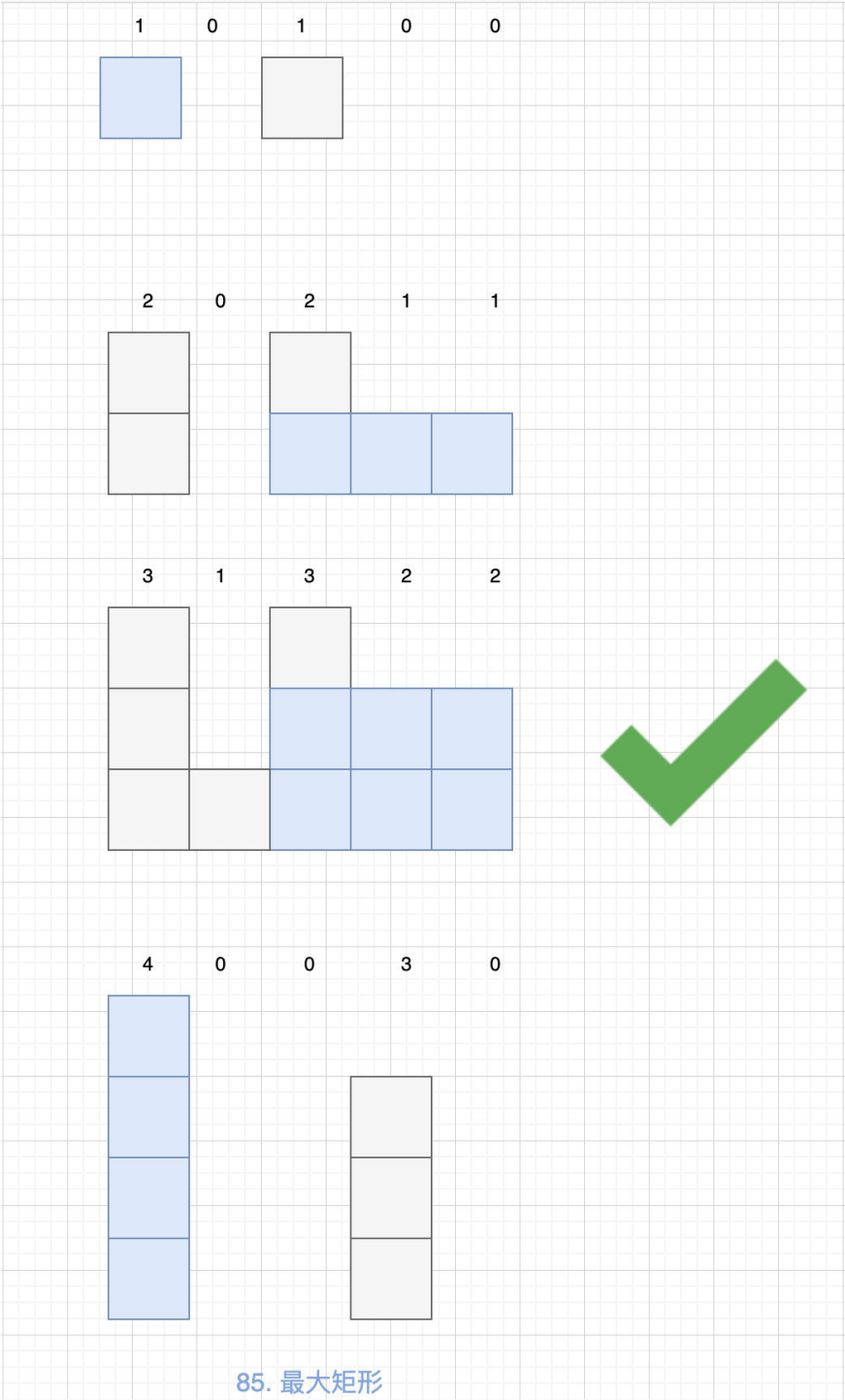
思路

我在 【84. 柱状图中最大的矩形】多种方法（Python3） 使用了多种方法来解决。 然而在这道题，我们仍然可以使用完全一样的思路去完成。 不熟悉的可以看下我的题解。本题解是基于那道题的题解来进行的。

拿题目给的例子来说：

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

我们逐行扫描得到 `84. 柱状图中最大的矩形` 中的 heights 数组：



85. 最大矩形

这样我们就可以使用 `84. 柱状图中最大的矩形` 中的解法来进行了，这里我们使用单调栈来解。

下面的代码直接将 84 题的代码封装成 API 调用了。

代码

代码支持：Python

Python Code:

```
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> i
        n, heights, st, ans = len(heights), [0] + heights +
        for i in range(n + 2):
            while st and heights[st[-1]] > heights[i]:
                ans = max(ans, heights[st.pop(-1)] * (i - s
            st.append(i)

        return ans
    def maximalRectangle(self, matrix: List[List[str]]) ->
        m = len(matrix)
        if m == 0: return 0
        n = len(matrix[0])
        heights = [0] * n
        ans = 0
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == "0":
                    heights[j] = 0
                else:
                    heights[j] += 1
            ans = max(ans, self.largestRectangleArea(height
        return ans
```

复杂度分析

- 时间复杂度：$O(M * N)$
- 空间复杂度：$O(N)$

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 38K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(87. 扰乱字符串)

https://leetcode-cn.com/problems/scramble-string/

题目描述

使用下面描述的算法可以扰乱字符串 s 得到字符串 t ：
如果字符串的长度为 1 ，算法停止
如果字符串的长度 > 1 ，执行下述步骤：
在一个随机下标处将字符串分割成两个非空的子字符串。即，如果已知字符串 s
随机 决定是要「交换两个子字符串」还是要「保持这两个子字符串的顺序不变」
在 x 和 y 这两个子字符串上继续从步骤 1 开始递归执行此算法。

给你两个 长度相等 的字符串 s1 和 s2，判断 s2 是否是 s1 的扰乱字符串

示例 1：

输入：s1 = "great", s2 = "rgeat"
输出：true
解释：s1 上可能发生的一种情形是：
"great" --> "gr/eat" // 在一个随机下标处分割得到两个子字符串
"gr/eat" --> "gr/eat" // 随机决定：「保持这两个子字符串的顺序不变
"gr/eat" --> "g/r / e/at" // 在子字符串上递归执行此算法。两个子字
"g/r / e/at" --> "r/g / e/at" // 随机决定：第一组「交换两个子字符
"r/g / e/at" --> "r/g / e/ a/t" // 继续递归执行此算法，将 "at"
"r/g / e/ a/t" --> "r/g / e/ a/t" // 随机决定：「保持这两个子字
算法终止，结果字符串和 s2 相同，都是 "rgeat"
这是一种能够扰乱 s1 得到 s2 的情形，可以认为 s2 是 s1 的扰乱字符串，

示例 2：

输入：s1 = "abcde", s2 = "caebd"
输出：false

示例 3：

输入：s1 = "a", s2 = "a"
输出：true

提示：

s1.length == s2.length
1 <= s1.length <= 30
s1 和 s2 由小写英文字母组成

前置知识

- 动态规划
- 递归

公司

- 暂无

思路

我们可以将 s1 和 s2 看成是两颗树 t1 和 t2 的中序遍历结果。

这样问题转为为**t1 是否可由 t2 经过翻转任意节点得到**，这里的翻转某一节点指的是交换该节点的左右子树，使得原本的左子树变成右子树，右子树变为左子树。这和 951. 翻转等价二叉树 是一样的。

而这道题比那道题难的点在于是**不知道树的原始结构**。我们可以将枚举所有可能，以 s1 来说：

- s1 的第一个字符可能是整棵树的根节点
- s1 的第二个字符可能是整棵树的根节点
- 。。。

确定了根节点，我们只需要使用同样的方法即可确定其他节点。这提示我们使用递归来解决。

> 如果对上面如何构造树不懂的，可以看下我之前写的 构造二叉树

上面提到了互为扰乱字符串必然存在相同的字符种类和个数，因此当我们确定了 s1 的根节点的时候，s2 的根节点只有两种类型。这是因为 s2 要保证分割后两部分的大小分别和 s1 的两部分大小完全一样。也就是说：**我们没有必要枚举 s1 和 s2 的所有可能的根节点组合**（这种组合有 n ^ 2 种，其中 n 为 s1 和 s2 的长度），而是**仅仅枚举 s1 的割点**（这样只有 n 种）。

实际上，我们没有必要先构造树 t1 和 t2，再去判断**t1 是否可由 t2 经过翻转任意节点得到**。而是将两个步骤结合到一起进行。

接下来，我们对 t1 的**每一个节点都执行进行翻转或者不翻转两种操作**，如果最终 t1 和 t2 相等了，说明 s1 和 s2 互为干扰字符串。实际上，我们没有必要真的翻转，而是直接比较 t1 的左子树和 t2 的右子树，t1 的右子树和 t2 的左子树**是否完全一样**。

代码：

> 我直接复制的 951. 翻转等价二叉树 的代码

```
class Solution:
    def flipEquiv(self, root1: TreeNode, root2: TreeNode) -
        if not root1 or not root2:
            return not root1 and not root2
        if root1.val != root2.val:
            return False
        # 不翻转
        if self.flipEquiv(root1.left, root2.left) and self.
            return True
        # 翻转
        if self.flipEquiv(root1.left, root2.right) and self
            return True
        # 不管翻转还是不翻转都不行，直接返回 False
        return False
```

另外，由于扰乱字符串的定义我们不难知道：如果 s1 和 s2 的字符种类和个数不完全相同，那么不可能是互为扰乱字符串（也就是说 s1 和 s2 排序后需要保持相同）。我们可以利用这点剪枝。

关键点

- 将其抽象为树的对比问题

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    @lru_cache(None)
    def isScramble(self, s1: str, s2: str) -> bool:
        if s1 == s2:
            return True
        # 剪枝
        if collections.Counter(s1) != collections.Counter(s
            return False
        # 枚举所有可能的根节点
        for i in range(1, len(s1)):
            # ----|-
            # -|----
            # 不进行翻转
            if self.isScramble(s1[:i], s2[:i]) and self.isS
                return True
            # 进行翻转
            if self.isScramble(s1[i:], s2[:-i]) and self.is
                return True
        # 不管翻转还是不翻转都不行，直接返回 False
        return False
```

复杂度分析

令 n 为字符串长度。

- 时间复杂度：$O(n^4)$， 其中 n^3 来自于 isScramble 的计算次数，n 来内每次 isScramble 的计算量。
- 空间复杂度：$O(n^3)$， 内存占用全部来自记忆化部分，因此空间复杂度等于数据规模，也就是 isScramble 三个参数的规模乘积。

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(124. 二叉树中的最大路径和)

https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/description/

题目描述

```
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，沿父节点-子节点连接，达到任

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
```

前置知识

- 树

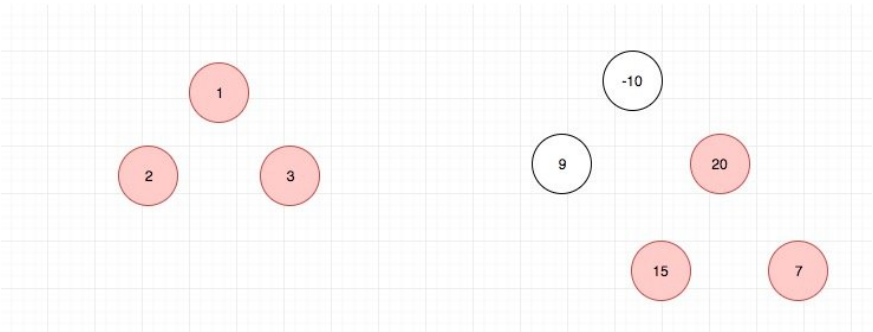
公司

- 阿里
- 腾讯
- 百度
- 字节

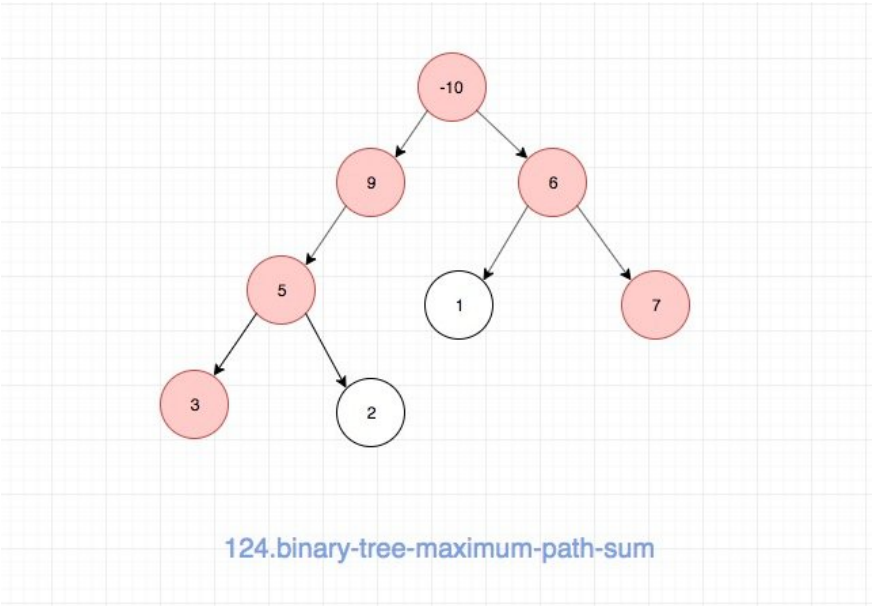
思路

这道题目的 path 让我误解了，然后浪费了很多时间来解这道题。我觉得 leetcode 给的 demo 太少了，不足以让我理解 path 的概念因此我这里自己画了一个图，来补充一下，帮助大家理解 path 的概念，不要像我一样理解错啦。

首先是官网给的两个例子：



接着是我自己画的一个例子：



如图红色的部分是最大路径上的节点。大家可以结合上面的 demo 来继续理解一下 path， 除非你理解了 path，否则不要往下看。

树的题目，基本都是考察递归思想的。因此我们需要思考如何去定义我们的递归函数，在这里我定义了一个递归函数，它的功能是，`返回以当前节点为根节点的MaxPath`

但是有两个条件:

1. 根节点必须选择
2. 左右子树只能选择一个

为什么要有这两个条件?

我的想法是原问题可以转化为：以每一个节点为根节点，分别求出 MaxPath，最后计算最大值，因此第一个条件需要满足.

对于第二个条件，由于递归函数子节点的返回值会被父节点使用，因此我们如果两个孩子都选择了就不符合 MaxPath 的定义了。实际上这道题，当遍历到某一个节点的时候，我们需要子节点的信息，然后同时结合自身的 val 来决定要不要选取左右子树以及选取的话要选哪一个， 因此这个过程本质上就是 `后序遍历`

基本算法就是不断调用递归函数，然后在调用过程中不断计算和更新 MaxPath，最后在主函数中将 MaxPath 返回即可。

关键点解析

- 递归
- 理解题目中的 path 定义

代码

代码支持：JavaScript，Java，Python, CPP

- JavaScript

```
/*
 * @lc app=leetcode id=124 lang=javascript
 *
 * [124] Binary Tree Maximum Path Sum
 */
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
function helper(node, payload) {
  if (node === null) return 0;

  const l = helper(node.left, payload);
  const r = helper(node.right, payload);

  payload.max = Math.max(
    node.val + Math.max(0, l) + Math.max(0, r),
    payload.max
  );

  return node.val + Math.max(l, r, 0);
}
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxPathSum = function (root) {
  if (root === null) return 0;
  const payload = {
    max: root.val,
  };
  helper(root, payload);
  return payload.max;
};
```

- Java

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
  int ans;
  public int maxPathSum(TreeNode root) {
    ans = Integer.MIN_VALUE;
    helper(root);   // recursion
    return ans;
  }

  public int helper(TreeNode root) {
    if (root == null) return 0;
    int leftMax = Math.max(0, helper(root.left));     // f
    int rightMax = Math.max(0, helper(root.right));   // f
    ans = Math.max(ans, leftMax+rightMax+root.val);   // f
    return max(leftMax, rightMax) + root.val;         // a
  }
}
```

- Python

```
class Solution:
    ans = float('-inf')
    def maxPathSum(self, root: TreeNode) -> int:
        def helper(node):
            if not node: return 0
            l = helper(node.left)
            r = helper(node.right)
            self.ans = max(self.ans, max(l,0) + max(r, 0) -
            return max(l, r, 0) + node.val
        helper(root)
        return self.ans
```

- CPP

```
class Solution {
private:
    int ans = INT_MIN;
    int postOrder(TreeNode *root) {
        if (!root) return INT_MIN;
        int L = max(0, postOrder(root->left)), R = max(0, p
        ans = max(ans, L + R + root->val);
        return root->val + max(L, R);
    }
public:
    int maxPathSum(TreeNode* root) {
        postOrder(root);
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(n)$，其中 n 为节点数。
- 空间复杂度：$O(h)$， 其中 h 为树高。

相关题目

- 113.path-sum-ii

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(128. 最长连续序列)

https://leetcode-cn.com/problems/longest-consecutive-sequence/

题目描述

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。
```

前置知识

- hashmap

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

这是一道最最长连续数字序列长度的题目， 官网给出的难度是 `hard` .

符合直觉的做法是先排序，然后用一个变量记录最大值，遍历去更新最大值即可，

代码:

```
if (nums.length === 0) return 0;
let count = 1;
let maxCount = 1;
// 这里其实可以不需要排序，这么做只不过是为了方便理解
nums = [...new Set(nums)].sort((a, b) => a - b);
for (let i = 0; i < nums.length - 1; i++) {
  if (nums[i + 1] - nums[i] === 1) {
    count++;
  } else {
    if (count > maxCount) {
      maxCount = count;
    }
    count = 1;
  }
}
return Math.max(count, maxCount);
```

但是需要排序时间复杂度会上升，题目要求时间复杂度为 O(n), 那么我们其实可以不用排序去解决的。

思路就是将之前”排序之后，通过比较前后元素是否相差 1 来判断是否连续“的思路改为 不排序而是 `直接遍历，然后在内部循环里面查找是否存在当前值的邻居元素` ，但是马上有一个 问题，内部我们 `查找是否存在当前值的邻居元素` 的过程如果使用数组，时间复杂度是 O(n), 那么总体的复杂度就是 O(n^2)，完全不可以接受。怎么办呢?

我们换个思路，用空间来换时间。比如用类似于 hashmap 这样的数据结构优化查询部分，将时间复杂度降低到 O(1), 代码见后面 `代码部分`

关键点解析

- 空间换时间

代码

代码支持：Java, Python，JS

Java Code:

```java
class Solution {
    public int longestConsecutive(int[] nums) {
        Set<Integer> set = new HashSet<Integer>();
        int ans = 0;
        for (int num : nums) {
            set.add(num);
        }
        for(int i = 0;i < nums.length; i ++) {
            int x = nums[i];
            // 说明x是连续序列的开头元素
            if  (!set.contains(x - 1)) {
                while(set.contains(x + 1)) {
                    x ++;
                }
            }
            ans = Math.max(ans, x - nums[i] + 1);
        }
        return ans;

    }
}
```

Python Code:

```python
class Solution:
    def longestConsecutive(self, A: List[int]) -> int:
        seen = set(A)
        ans = 0
        for a in A:
            t = a
            #  if 的作用是剪枝
            if t + 1 not in seen:
                while t - 1 in seen:
                    t -= 1
            ans = max(ans, a - t + 1)
        return ans
```

JS Code:

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var longestConsecutive = function (nums) {
  set = new Set(nums);
  let max = 0;
  let temp = 0;
  set.forEach((x) => {
    // 说明x是连续序列的开头元素。加这个条件相当于剪枝的作用，否则时间
    if (!set.has(x - 1)) {
      temp = x + 1;
      while (set.has(y)) {
        temp = temp + 1;
      }
      max = Math.max(max, y - x); // y - x 就是从x开始到最后有
    }
  });
  return max;
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(132. 分割回文串 II)

https://leetcode-cn.com/problems/palindrome-partitioning-ii/

题目描述

给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文。

返回符合要求的 最少分割次数 。

示例 1:

输入: s = "aab"
输出: 1
解释: 只需一次分割就可将 s 分割成 ["aa","b"] 这样两个回文子串。

示例 2:

输入: s = "a"
输出: 0

示例 3:

输入: s = "ab"
输出: 1

提示:

1 <= s.length <= 2000
s 仅由小写英文字母组成

前置知识

- 动态规划

公司

- 暂无

思路

为了能够得到最小的**分割次数**。我们需要枚举所有的分割可能，从中找到最小的。使用纯粹暴力的回溯并无法通过本题。因此需要性能上更加优秀的算法。

试想，如果 s[i:j] 是否是回文的信息已经计算好了，关于如何计算，这需要一点点预处理，我会在之后讲解。

现在不妨假设有一个函数：`judge(i,j)`，你可以输入 i 和 j，函数可以在 $O(1)$ 的时间返回 s[i:j] 是否为回文，如果是回文则返回 true，否则返回 false。

那么，我们就可以使用双层循环枚举所有的子串，并

```
for i in range(n):
    for j in range(i + 1, n):
        if judge(i + 1, j):
            # 你的逻辑
```

这里的动态规划转移为：`dp[j] = min(dp[j], dp[i] + 1)`，其中 dp[i] 表示将字符串 s[0:i] 分割成若干回文串的最小分割次数。那么答案自然就是 dp[n-1]。base case 为 dp[0] = 0，其他初始化为无穷大即可。

剩下的则是 judge 实现。 实际上，我们可以预处理二维数组 palindrome_pairs，其中 palindrome_pairs[i][j] 存放的是一个布尔值，表示 s[i:j] 是否是回文， 具体处理的过程也是使用动态规划技巧。其动态规划转移方程思想就是**中心扩展法**，这是一种回文问题中常用的技巧。

这种算法的思想为：如果 s[i:j] 是回文，那么在其左右加相同的字符，同样可以构成一个回文。在不是回文的字符串左右添加任意字符都不能得到回文串。因此转移方程就是：

```
palindrome_pairs[i][j] = (s[i] == s[j]) and palindrome_pai
```

这道题有一点被忽略，也算是一个难点吧。回顾下上面的代码：

```
for i in range(n):
    for j in range(i + 1, n):
        if judge(i + 1, j):
            dp[j] = min(dp[j], dp[i] + 1)
```

这里的代码有个问题，即如果 s[0:j] 本身就是一个回文，那么 dp[j] 应该是 0，这也算是一种特殊的 base case。

关键点

- 预处理。 将 s[i:j] 是否为回文的数据提前计算出来存储到一个二维数组中。接下来就是普通的动态规划。
- 如果 s[0:j] 本身就是一个回文，那么 dp[j] 应该是 0

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)
        palindrome_pairs = [[True] * n for _ in range(n)]

        for i in range(n - 1, -1, -1):
            for j in range(i + 1, n):
                palindrome_pairs[i][j] = (s[i] == s[j]) and

        def judge(i, j):
            return palindrome_pairs[i][j]

        dp = [float("inf")] * n
        dp[0] = 0
        for i in range(n):
            for j in range(i + 1, n):
                if palindrome_pairs[0][j]:
                    dp[j] = 0
                elif judge(i + 1, j):
                    dp[j] = min(dp[j], dp[i] + 1)
        return dp[-1]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（140. 单词拆分 II）

https://leetcode-cn.com/problems/word-break-ii/

题目描述

```
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

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"
]
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：
[]
```

微软有一道题和这个一样 题目地址 一样，感兴趣的可以看看完整面经。

前置知识

- 回溯
- 笛卡尔积

公司

- 暂无

暴力回溯

思路

实际上这道题就是暴力回溯就好了， 代码也比较简单。

代码

代码支持：Python3, CPP

Python3 Code:

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> Lis
        ans = []
        n = len(s)

        def backtrack(temp, start):
            if start == n: ans.append(temp[1:])
            for i in range(start, n):
                if s[start:i + 1] in wordDict:
                    backtrack(temp + " " + s[start:i + 1],
        backtrack('', 0)
        return ans
```

CPP Code:

```
class Solution {
    int maxLen = 0;
    unordered_set<string> ws;
    vector<int> m;
    vector<string> ans;
    bool dfs(string &s, int i, string tmp) {
        if (i == s.size()) {
            ans.push_back(tmp);
            return true;
        }
        if (m[i] == 0) return m[i];
        m[i] = 0;
        for (int j = min((int)s.size(), i + maxLen); j > i
            auto sub = s.substr(i, j - i);
            if (ws.count(sub) && dfs(s, j, tmp.size() ? tm
        }
        return m[i];
    }
public:
    vector<string> wordBreak(string s, vector<string>& dic
        ws = { dict.begin(), dict.end() };
        for (auto &w : dict) maxLen = max(maxLen, (int)w.s
        m.assign(s.size(), -1); // -1 = unvisited, 0 = can
        dfs(s, 0, "");
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(2^N)$
- 空间复杂度：$O(2^N)$

笛卡尔积优化

思路

上面的代码会超时，测试用例会挂在如下：

```
"aaaa
```

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> Lis
        n = len(s)
        @lru_cache(None)
        def backtrack(start):
            ans = []
            if start == n:
                ans.append('')
            for i in range(start, n):
                if s[start:i + 1] in wordDict:
                    if start == 0: temp = s[start:i + 1]
                    else: temp = " " + s[start:i + 1]
                    ps = backtrack(i + 1)
                    for p in ps:
                        ans.append(temp + p)
            return ans
        return backtrack(0)
```

复杂度分析

令 C 为字典的总长度， N 为字典中最长的单词的长度。

- 时间复杂度：$O(N^2)$
- 空间复杂度：$O(C)$

这种记忆化递归的方式和 DP 思想一模一样， 大家可以将其改造为 DP，这个留给大家来完成。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(145. 二叉树的后序遍历)

https://leetcode-cn.com/problems/binary-tree-postorder-traversal/

题目描述

```
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?
```

前置知识

- 栈
- 递归

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

相比于前序遍历，后续遍历思维上难度要大些，前序遍历是通过一个 stack，首先压入父亲结点，然后弹出父亲结点，并输出它的 value，之后压人其右儿子，左儿子即可。

然而后序遍历结点的访问顺序是：左儿子 -> 右儿子 -> 自己。那么一个结点需要两种情况下才能够输出： 第一，它已经是叶子结点； 第二，它不是叶子结点，但是它的儿子已经输出过。

那么基于此我们只需要记录一下当前输出的结点即可。对于一个新的结点，如果它不是叶子结点，儿子也没有访问，那么就需要将它的右儿子，左儿子压入。 如果它满足输出条件，则输出它，并记录下当前输出结点。输出在 stack 为空时结束。

关键点解析

- 二叉树的基本操作（遍历）
  > 不同的遍历算法差异还是蛮大的
- 如果非递归的话利用栈来简化操作
- 如果数据规模不大的话，建议使用递归
- 递归的问题需要注意两点，一个是终止条件，一个如何缩小规模
- 终止条件，自然是当前这个元素是 null（链表也是一样）
- 由于二叉树本身就是一个递归结构， 每次处理一个子树其实就是缩小了规模， 难点在于如何合并结果，这里的合并结果其实就是 `left.concat(right).concat(mid)` , mid 是一个具体的节点，left 和 right `递归求出即可`

代码

代码支持：JS, CPP

JS Code:

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var postorderTraversal = function (root) {
  // 1. Recursive solution

  // if (!root) return [];

  // return postorderTraversal(root.left).concat(postorder

  // 2. iterative solutuon

  if (!root) return [];
  const ret = [];
  const stack = [root];
  let p = root; // 标识元素，用来判断节点是否应该出栈

  while (stack.length > 0) {
    const top = stack[stack.length - 1];
    if (
      top.left === p ||
      top.right === p || // 子节点已经遍历过了
      (top.left === null && top.right === null) // 叶子元素
    ) {
      p = stack.pop();
      ret.push(p.val);
    } else {
      if (top.right) {
        stack.push(top.right);
      }
      if (top.left) {
        stack.push(top.left);
      }
    }
  }

  return ret;
};
```

CPP Code:

```cpp
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> ans;
        stack<TreeNode*> s;
        TreeNode *prev = NULL;
        while (root || s.size()) {
            while (root) {
                s.push(root);
                root = root->left;
            }
            root = s.top();
            if (!root->right || root->right == prev) {
                ans.push_back(root->val);
                s.pop();
                prev = root;
                root = NULL;
            } else root = root->right;
        }
        return ans;
    }
};
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

相关专题

- 二叉树的遍历

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(146. LRU 缓存机制)

https://leetcode-cn.com/problems/lru-cache/

题目描述

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 。

实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以正整数作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 -1 。
- `void put(int key, int 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：

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3],
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

思路

1. 确定需要使用的数据结构

   i. 根据题目要求,存储的数据需要保证顺序关系(逻辑层面) ===> 使用数组,链表等保证顺序关系
   ii. 同时需要对数据进行频繁的增删, 时间复杂度 O(1) ==> 使用链表等
   iii. 对数据进行读取时, 时间复杂度 O(1) ===> 使用哈希表 最终采取双向链表 + 哈希表
      > i. 双向链表按最后一次访问的时间的顺序进行排列, 链表头部为最近访问的节点
      > ii. 哈希表,以关键字为键,以链表节点的地址为值

2. put 操作 通过哈希表, 查看传入的关键字对应的链表节点, 是否存在
   i. 如果存在,
      i. 将该链表节点的值更新
      ii. 将该该链表节点调整至链表头部
   ii. 如果不存在
      i. 如果链表容量未满,
         i. 新生成节点,
         ii. 将该节点位置调整至链表头部
      ii. 如果链表容量已满
         i. 删除尾部节点
         ii. 新生成节点
         iii. 将该节点位置调整至链表头部
      iii. 将新生成的节点，按关键字为键，节点地址为值插入哈希表
3. get 操作 通过哈希表, 查看传入的关键字对应的链表节点, 是否存在
   i. 节点存在
      i. 将该节点位置调整至链表头部
      ii. 返回该节点的值
   ii. 节点不存在, 返回 null

伪代码如下：

```
var LRUCache = function(capacity) {
    保存一个该数据结构的最大容量
    生成一个双向链表,同时保存该链表的头结点与尾节点
    生成一个哈希表
};

function get (key) {
    if 哈希表中存在该关键字 {
        根据哈希表获取该链表节点
        将该节点放置于链表头部
        return 链表节点的值
    } else {
          return -1
    }
};

function put (key, value) {
    if 哈希表中存在该关键字 {
        根据哈希表获取该链表节点
        将该链表节点的值更新
        将该节点放置于链表头部
    } else {
        if 容量已满 {
            删除链表尾部的节点
            新生成一个节点
            将该节点放置于链表头部
        } else {
            新生成一个节点
            将该节点放置于链表头部
        }
    }
};
```

代码

语言支持: JS, Go, PHP, Python3

JS Code:

```
function ListNode(key, val) {
  this.key = key;
  this.val = val;
  this.pre = this.next = null;
}

var LRUCache = function (capacity) {
  this.capacity = capacity;
  this.size = 0;
  this.data = {};
  this.head = new ListNode();
  this.tail = new ListNode();
  this.head.next = this.tail;
  this.tail.pre = this.head;
};

function get(key) {
  if (this.data[key] !== undefined) {
    let node = this.data[key];
    this.removeNode(node);
    this.appendHead(node);
    return node.val;
  } else {
    return -1;
  }
}

function put(key, value) {
  let node;
  if (this.data[key] !== undefined) {
    node = this.data[key];
    this.removeNode(node);
    node.val = value;
  } else {
    node = new ListNode(key, value);
    this.data[key] = node;
    if (this.size < this.capacity) {
      this.size++;
    } else {
      key = this.removeTail();
      delete this.data[key];
    }
  }
  this.appendHead(node);
}

function removeNode(node) {
  let preNode = node.pre,
```

```
    nextNode = node.next;
  preNode.next = nextNode;
  nextNode.pre = preNode;
}

function appendHead(node) {
  let firstNode = this.head.next;
  this.head.next = node;
  node.pre = this.head;
  node.next = firstNode;
  firstNode.pre = node;
}

function removeTail() {
  let key = this.tail.pre.key;
  this.removeNode(this.tail.pre);
  return key;
}
```

Go Code:

```
type LRUCache struct {
    Cap   int
    Hash  map[int]*DoubleListNode
    Cache *DoubleList
    Size  int
}

func Constructor(capacity int) LRUCache {
    return LRUCache{
        Cap:   capacity,
        Hash:  make(map[int]*DoubleListNode),
        Cache: NewDoubleList(),
    }
}

func (this *LRUCache) Get(key int) int {
    n, ok := this.Hash[key]
    if !ok {
        return -1
    }

    // 设为最近使用过
    this.Cache.remove(n)
    this.Cache.append(n)

    return n.Val
}

func (this *LRUCache) Put(key int, value int) {
    n, ok := this.Hash[key]
    if !ok { // cache 不存在
        // 增加节点
        newNode := &DoubleListNode{Key: key, Val: value}
        this.Hash[key] = newNode
        this.Cache.append(newNode)

        if this.Cap == this.Size { // 已满
            // 删除尾节点
            tailNode := this.Cache.tail.Pre
            delete(this.Hash, tailNode.Key)
            this.Cache.remove(tailNode)
        } else {
            this.Size++
        }
    } else {
        // cache 存在
        // 设为最近使用过
        this.Cache.remove(n)
```

```
        this.Cache.append(n)

        n.Val = value // 更新值
    }
}

type DoubleListNode struct {
    Key, Val  int
    Pre, Next *DoubleListNode
}

type DoubleList struct {
    Head, tail *DoubleListNode
}

func NewDoubleList() *DoubleList {
    dl := &DoubleList{
        Head: &DoubleListNode{},
        tail: &DoubleListNode{},
    }
    dl.Head.Next = dl.tail
    dl.tail.Pre = dl.Head
    return dl
}

func (dl *DoubleList) append(n *DoubleListNode) {
    n.Next = dl.Head.Next
    n.Pre = dl.Head
    dl.Head.Next.Pre = n
    dl.Head.Next = n
}

func (dl *DoubleList) remove(n *DoubleListNode) {
    n.Pre.Next = n.Next
    n.Next.Pre = n.Pre
}
```

PHP Code:

```php
class LRUCache
{
    public $hash = []; // key => DoubleListNode
    public $cache;
    public $size = 0;
    public $cap; // 容量

    /**
     * @param Integer $capacity
     */
    function __construct($capacity)
    {
        $this->cap = $capacity;
        $this->cache = new DoubleList();
    }

    /**
     * @param Integer $key
     * @return Integer
     */
    function get($key)
    {
        if (!isset($this->hash[$key])) return -1;

        // 更新节点
        /** @var DoubleListNode $node */
        $node = $this->hash[$key];
        $this->cache->remove($node);
        $this->cache->append($node);

        return $node->val;
    }

    /**
     * @param Integer $key
     * @param Integer $value
     * @return NULL
     */
    function put($key, $value)
    {
        if (isset($this->hash[$key])) { // key 存在
            // 更新节点
            /** @var DoubleListNode $node */
            $node = $this->hash[$key];
            $this->cache->remove($node);
            $this->cache->append($node);
            $node->val = $value;
        } else { // key 不存在, 新增节点
```

```
            $node = new DoubleListNode($key, $value);
            $this->cache->append($node);
            $this->hash[$key] = $node;

            if ($this->size == $this->cap) {
                // 删除原节点
                $oldNode = $this->cache->tail->pre;
                $this->cache->remove($oldNode);
                unset($this->hash[$oldNode->key]);
            } else {
                $this->size++;
            }
        }
    }
}

class DoubleListNode
{
    public $key;
    public $val;
    public $pre;
    public $next;

    public function __construct($key = null, $val = null)
    {
        if ($key) $this->key = $key;
        if ($val) $this->val = $val;
    }
}

class DoubleList
{
    public $head;
    public $tail;

    public function __construct()
    {
        $this->head = new DoubleListNode();
        $this->tail = new DoubleListNode();
        $this->head->next = $this->tail;
        $this->tail->pre = $this->head;
    }

    public function append(DoubleListNode $node)
    {
        $node->pre = $this->head;
        $node->next = $this->head->next;
        $this->head->next->pre = $node;
```

```
            $this->head->next = $node;
        }

        public function remove(DoubleListNode $node)
        {
            $node->pre->next = $node->next;
            $node->next->pre = $node->pre;
        }
    }
```

Python Code:

> 来自 leetcode

```
class DLinkedNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None


class LRUCache:
    def __init__(self, capacity: int):
        self.cache = dict()
        # 使用伪头部和伪尾部节点
        self.head = DLinkedNode()
        self.tail = DLinkedNode()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.capacity = capacity
        self.size = 0

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        # 如果 key 存在，先通过哈希表定位，再移到头部
        node = self.cache[key]
        self.moveToHead(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key not in self.cache:
            # 如果 key 不存在，创建一个新的节点
            node = DLinkedNode(key, value)
            # 添加进哈希表
            self.cache[key] = node
            # 添加至双向链表的头部
            self.addToHead(node)
            self.size += 1
            if self.size > self.capacity:
                # 如果超出容量，删除双向链表的尾部节点
                removed = self.removeTail()
                # 删除哈希表中对应的项
                self.cache.pop(removed.key)
                self.size -= 1
        else:
            # 如果 key 存在，先通过哈希表定位，再修改 value，并移
            node = self.cache[key]
            node.value = value
            self.moveToHead(node)
```

```
    def addToHead(self, node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def removeNode(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def moveToHead(self, node):
        self.removeNode(node)
        self.addToHead(node)

    def removeTail(self):
        node = self.tail.prev
        self.removeNode(node)
        return node
```

复杂度分析

- 时间复杂度：$O(1)$
- 空间复杂度：$O(n)$ n 为容量的大小

题目地址(154. 寻找旋转排序数组中的最小值 II)

https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array-ii/

题目描述

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 旋转 后，得
若旋转 4 次，则可以得到 [4,5,6,7,0,1,4]
若旋转 7 次，则可以得到 [0,1,4,4,5,6,7]

注意，数组 [a[0], a[1], a[2], ..., a[n-1]] 旋转一次 的结果为数组

给你一个可能存在 重复 元素值的数组 nums ，它原来是一个升序排列的数组，

示例 1:

输入：nums = [1,3,5]
输出：1

示例 2:

输入：nums = [2,2,2,0,1]
输出：0

提示:

n == nums.length
1 <= n <= 5000
-5000 <= nums[i] <= 5000
nums 原来是一个升序排序的数组，并进行了 1 至 n 次旋转

进阶:

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？

前置知识

- 二分

公司

- 暂无

思路

和 153 题目类似，只不过这道题在 153 的基础上增加了“可能重复”的条件。

沿用 153 的思路：

- 如果左端点的值小于右端点的值则可以提前退出。
- 否则我们选取中点，并判断中点的位置是在左边有序部分还是右边有序部分。
- 如果在左边有序部分，那么 r = mid，如果在右边有序部分则 l = mid + 1

问题的关键是**有时候我们是没有办法判断 mid 是在左边有序部分还是右边有序部分的。** 比如 nums[mid] == nums[l]，就可能对应下面的两种情况：

1. [2,2,2,2,0,1,2] 此时 mid 在左侧有序部分
2. [2,0,1,2,2,2] 此时 mid 在右侧有序部分

这个时候我们无法排除一半。退而求其次，我们只能排除一个，这个是算法的关键，这同时意味着我们的算法无法在最坏的情况下达到 $logn$，这和我们平时用的比较多的二分是不一样的。

那么在这种无法判断是在左边有序部分还是右边有序部分的情况下，我们应该舍弃左端点还是右端点呢?

答案是选择舍弃右端点，因为舍弃右端点不会错过最小值。之所以选择舍弃右端点不会错过最小值有一个前提：**取中点逻辑是向下取整**，如果你取中点是向上取整情况就有所不同了。

另外需要注意的是，我们选择判断是在左侧有序部分还是右边有序部分，**需要用 mid 和右端点进行比较，而不能是左端点。**这是因为使用左端点无法在任何情况舍弃一半，读者不妨自己试试看。

关键点

- 比较右端点而不是左端点
- 如果左端点的值小于右端点的值则可以提前退出

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1

        while l < r:
            if nums[l] < nums[r]:
                return nums[l]
            mid = (l + r) // 2
            # [2,2,2,0,1]
            if nums[mid] > nums[r]:
                l = mid + 1
            elif nums[mid] < nums[r]:
                r = mid
            else:
                r -= 1

        return nums[l]  # or nums[r]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$，最坏的情况我们需要扫描整个数组。
- 空间复杂度：$O(1)$

题目地址（212. 单词搜索 II）

https://leetcode-cn.com/problems/word-search-ii/

题目描述

```
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

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

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
```

前置知识

- 前缀树
- 深度优先遍历
- 小岛专题
- 剪枝

公司

- 百度
- 字节

思路

我们需要对矩阵中每一项都进行深度优先遍历（DFS）。 递归的终点是

1. 超出边界
2. 递归路径上组成的单词不在 words 的前缀。

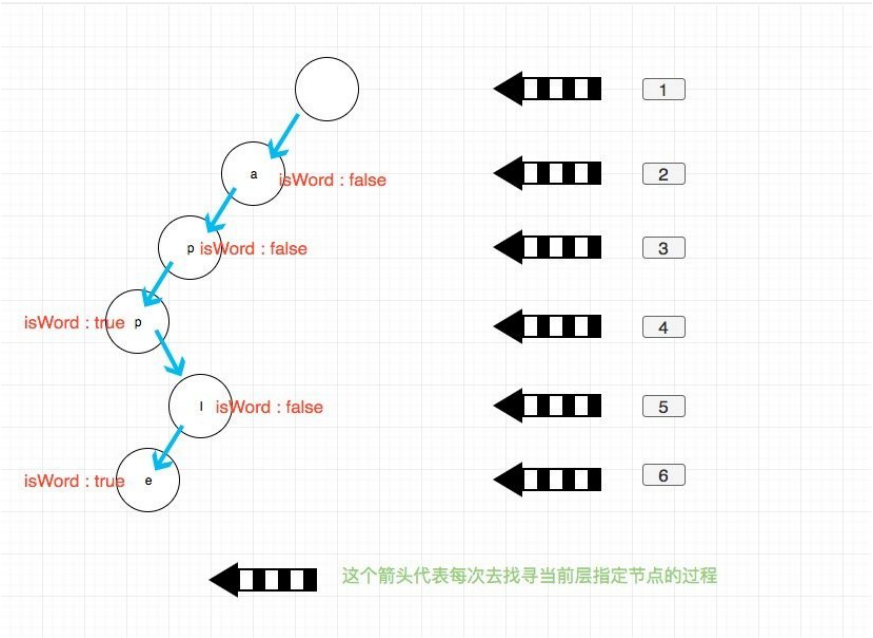
比如题目示例：words = ["oath","pea","eat","rain"]，那么对于 oa，oat 满足条件，因为他们都是 oath 的前缀。因此对于 a，oat 来说，它们**有希望**能找到 oath，但是 oaa 就不满足条件。这是一个关键点，我们的算法就是基于这个前提进行剪枝的，如果不剪枝则无法通过所有的测试用例。

这是一个典型的二维表格 DFS，和小岛专题套路一样：

- 四个方向 DFS。
- 为了防止环的出现，我们需要记录访问过的节点。
- 必须的时候考虑原地修改，减少 visited 的空间开销。

而返回结果是需要去重的。出于简单考虑，我们使用集合（set），最后返回的时候重新转化为 list。

刚才我提到了一个关键词“前缀”，我们考虑使用前缀树来优化。使得复杂度降低为$O(h)$, 其中 h 是前缀树深度，也就是最长的字符串长度。



关键点

- 前缀树（也叫字典树），英文名 Trie（读作 tree 或者 try）
- DFS
- 剪枝的技巧

代码

- 语言支持：Python3

Python3 Code：

关于 Trie 的代码:

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def startsWith(self, prefix):
        """
        Returns if there is any word in the trie that start
        :type prefix: str
        :rtype: bool
        """

        curr = self.Trie
        for w in prefix:
            if w not in curr:
                return False
            curr = curr[w]
        return True
```

主逻辑代码：

```
class Solution:
    def findWords(self, board: List[List[str]], words: List
        m = len(board)
        if m == 0:
            return []
        n = len(board[0])
        trie = Trie()
        seen = None
        res = set()
        for word in words:
            trie.insert(word)

        def dfs(s, i, j):
            if (i, j) in seen or i < 0 or i >= m or j < 0 o
                return
            s += board[i][j]
            seen[(i, j)] = True

            if s in words:
                res.add(s)
            dfs(s, i + 1, j)
            dfs(s, i - 1, j)
            dfs(s, i, j + 1)
            dfs(s, i, j - 1)

            del seen[(i, j)]

        for i in range(m):
            for j in range(n):
                seen = dict()
                dfs("", i, j)
        return list(res)
```

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0472.concatenated-words
- 0820.short-encoding-of-words
- 1032.stream-of-characters

题目地址(239. 滑动窗口最大值)

https://leetcode-cn.com/problems/sliding-window-maximum/

题目描述

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

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
```

前置知识

- 队列
- 滑动窗口

公司

- 阿里
- 腾讯
- 百度
- 字节

思路

符合直觉的想法是直接遍历 nums, 然后然后用一个变量 slideWindow 去承载 k 个元素， 然后对 slideWindow 求最大值，这是可以的，遍历一次的时间复杂度是 N，k 个元素求最大值时间复杂度是 k， 因此总的时间复杂度是 O(n * k).代码如下：

JavaScript:

```
var maxSlidingWindow = function (nums, k) {
  // bad 时间复杂度O(n * k)
  if (nums.length === 0 || k === 0) return [];
  let slideWindow = [];
  const ret = [];
  for (let i = 0; i < nums.length - k + 1; i++) {
    for (let j = 0; j < k; j++) {
      slideWindow.push(nums[i + j]);
    }
    ret.push(Math.max(...slideWindow));
    slideWindow = [];
  }
  return ret;
};
```

Python3:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) ->
        if k == 0: return []
        res = []
        for r in range(k - 1, len(nums)):
            res.append(max(nums[r - k + 1:r + 1]))
        return res
```

但是如果真的是这样，这道题也不会是 hard 吧？这道题有一个 follow up，要求你用线性的时间去完成。

其实，我们没必须存储窗口内的所有元素。 如果新进入的元素比前面的大，那么前面的元素就不再有利用价值，可以直接移除。这提示我们使用一个单调递增栈来完成。

但由于窗口每次向右移动的时候，位于窗口最左侧的元素是需要被擦除的，而栈只能在一端进行操作。

而如果你使用数组实现，就是可以在另一端操作了，但是时间复杂度仍然是 $O(k)$，和上面的暴力算法时间复杂度一样。

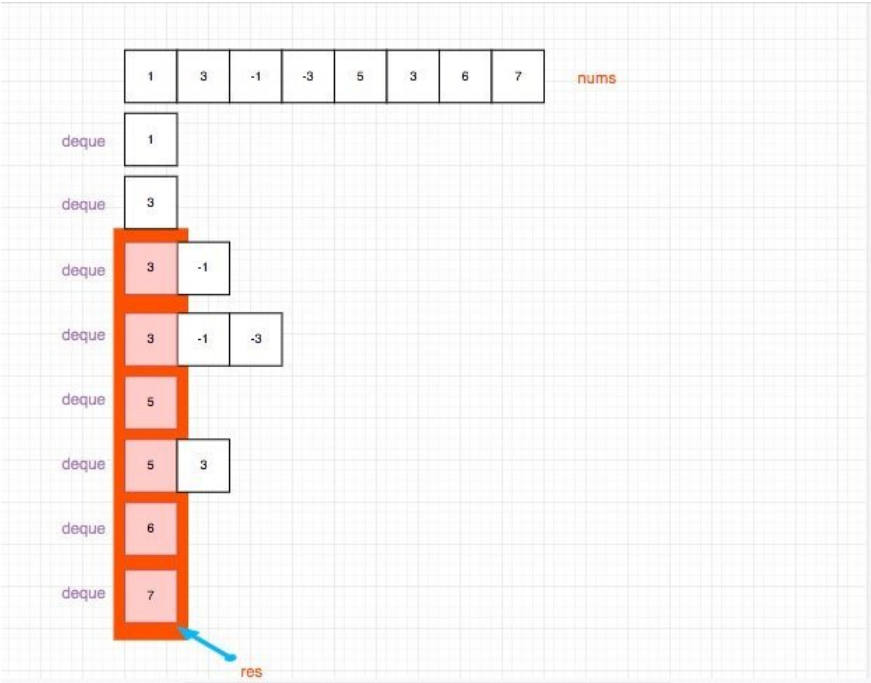
因此，我们考虑使用链表来实现，维护两个指针分别指向头部和尾部即可，这样做的时间复杂度是 $O(1)$，这就是双端队列。

因此思路就是用一个双端队列来保存 `接下来的滑动窗口可能成为最大值的数` 。

具体做法：

- 入队列
- 移除失效元素，失效元素有两种
- 一种是已经超出窗口范围了，比如我遍历到第 4 个元素，k = 3，那么 i = 0 的元素就不应该出现在双端队列中了 具体就是 `索引大于 i - k + 1的元素都应该被清除`
- 小于当前元素都没有利用价值了，具体就是 `从后往前遍历（双端队列是一个递减队列）双端队列，如果小于当前元素就出队列`

经过上面的分析，不难知道双端队列其实是一个递减的一个队列，因此队首的元素一定是最大的。用图来表示就是：



关键点解析

- 双端队列简化时间复杂度
- 滑动窗口

代码

JavaScript:

JS 的 deque 实现我这里没有写， 大家可以参考 collections/deque

```
var maxSlidingWindow = function (nums, k) {
  // 双端队列优化时间复杂度, 时间复杂度O(n)
  const deque = []; // 存放在接下来的滑动窗口可能成为最大值的数
  const ret = [];
  for (let i = 0; i < nums.length; i++) {
    // 清空失效元素
    while (deque[0] < i - k + 1) {
      deque.shift();
    }

    while (nums[deque[deque.length - 1]] < nums[i]) {
      deque.pop();
    }

    deque.push(i);

    if (i >= k - 1) {
      ret.push(nums[deque[0]]);
    }
  }
  return ret;
};
```

复杂度分析

- 时间复杂度：$O(N * k)$，如果使用双端队列优化的话，可以到 $O(N)$
- 空间复杂度：$O(k)$

Python3:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) ->
        q = collections.deque() # 本质就是单调队列
        ans = []
        for i in range(len(nums)):
            while q and nums[q[-1]] <= nums[i]: q.pop() # 维
            while q and i - q[0] >= k: q.popleft() # 移除失效
            q.append(i)
            if i >= k - 1: ans.append(nums[q[0]])
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$
- 空间复杂度：$O(k)$

扩展

为什么用双端队列

因为删除无效元素的时候，会清除队首的元素（索引太小了）或者队尾(元素太小了)的元素。 因此需要同时对队首和队尾进行操作，使用双端队列是一种合乎情理的做法。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(295. 数据流的中位数)

https://leetcode-cn.com/problems/find-median-from-data-stream/

题目描述

```
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

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:

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?
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?
```

前置知识

- 堆
- 队列

公司

- 阿里
- 百度
- 字节

思路

这道题目是求动态数据的中位数，在 leetcode 难度为 `hard` . 如果这道题是求静态数据的中位数，我们用数组去存储， 空间复杂度 O(1), 时间复杂度 O(1)

> 空间复杂度指的是除了存储数据之外额外开辟的用于计算等任务的内存空间

代码也比较简单

```
function findMedian(a) {
  return a.length % 2 === 0
    ? (a[a.length >> 1] + a[a.length >> (1 + 1)]) / 2
    : a[a.length >> 1];
}
```

但是题目要求是动态数据， 那么是否可以每次添加数据的时候，都去排一次序呢？ 假如我们每次插入都用 `快速排序` 进行排序的话，那么时间复杂度是 O(nlogn) + O(1)

> O(nlogn) 是排序的时间复杂度 O(1)是查询中位数的时间复杂度

如果你用这种思路进行的话， 恐怕 leetcode 会超时。

那么如何优化呢？ 答案是使用堆， Java， C++等语言都有 `优先级队列` 中这种数据结构， 优先级队列本质上就是一个堆。 关于堆和优先级队列的关系，我会放在《数据结构和算法》部分讲解。这里不赘述

如果借助堆这种数据结构， 就可以轻易实现了。

具体的做法是，建立两个堆，这两个堆需要满足:

1. 大顶堆元素都比小顶堆小（由于堆的特点其实只要比较堆顶即可）
2. 大顶堆元素不小于小顶堆，且最多比小顶堆多一个元素

满足上面两个条件的话，如果想要找到中位数，就比较简单了

- 如果两个堆数量相等（本质是总数为偶数）, 就两个堆顶元素的平均数
- 如果两个堆数量不相等（本质是总数为奇数）， 就取大顶堆的堆顶元素

比如对于[1,2,3] 求中位数:

再比如对于[1,2,3, 4] 求中位数:

关键点解析

- 用两个堆（一个大顶堆，一个小顶堆）来简化时间复杂度
- 用优先级队列简化操作

> JavaScript 不像 Java， C++等语言都有 `优先级队列` 中这种数据结构， 因此大家可以使用社区的实现 个人认为没有非要纠结于优先级队列怎么实现， 至少这道题不是考这个的 优先级队列的实现个人认为已经超过了这道题想考察的范畴

代码

代码支持：CPP，JS

JS Code：

如果不使用现成的 `优先级队列` 这种数据结构，代码可能是这样的：

```
/**
 * initialize your data structure here.
 */
var MedianFinder = function () {
  this.maxHeap = [];
  this.minHeap = [];
};

function minHeapify() {
  this.minHeap.unshift(null);
  const a = this.minHeap;

  // 为了方便大家理解，这里选用了粗暴的实现
  // 时间复杂度为O(n)
  // 其实可以降到O(logn)，  具体细节我不想在这里讲解和实现
  for (let i = a.length - 1; i >> 1 > 0; i--) {
    // 自下往上堆化
    if (a[i] < a[i >> 1]) {
      // 如果子元素更小，则交换位置
      const temp = a[i];
      this.minHeap[i] = a[i >> 1];
      this.minHeap[i >> 1] = temp;
    }
  }
  this.minHeap.shift(null);
}

function maxHeapify() {
  this.maxHeap.unshift(null);
  const a = this.maxHeap;

  // 为了方便大家理解，这里选用了粗暴的实现
  // 时间复杂度为O(n)
  // 其实可以降到O(logn)，  具体细节我不想在这里讲解和实现
  for (let i = a.length - 1; i >> 1 > 0; i--) {
    // 自下往上堆化
    if (a[i] > a[i >> 1]) {
      // 如果子元素更大，则交换位置
      const temp = a[i];
      this.maxHeap[i] = a[i >> 1];
      this.maxHeap[i >> 1] = temp;
    }
  }
  this.maxHeap.shift(null);
}

/**
 * @param {number} num
```

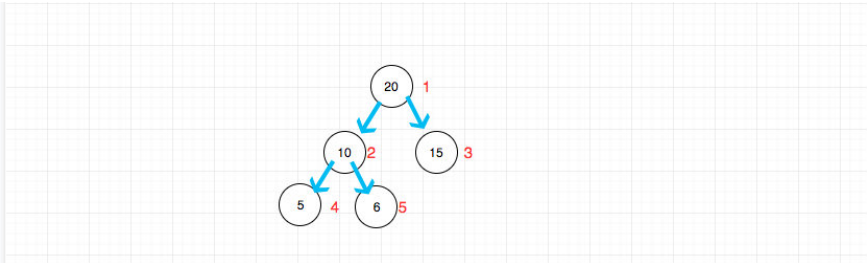
```
 * @return {void}
 */
MedianFinder.prototype.addNum = function (num) {
  // 为了大家容易理解，这部分代码写的比较冗余

  // 插入
  if (num >= (this.minHeap[0] || Number.MIN_VALUE)) {
    this.minHeap.push(num);
  } else {
    this.maxHeap.push(num);
  }
  // 调整两个堆的节点数量平衡
  // 使得大顶堆的数量最多大于小顶堆一个， 且一定不小于小顶堆数量
  if (this.maxHeap.length > this.minHeap.length + 1) {
    // 大顶堆的堆顶元素移动到小顶堆
    this.minHeap.push(this.maxHeap.shift());
  }

  if (this.minHeap.length > this.maxHeap.length) {
    // 小顶堆的堆顶元素移动到大顶堆
    this.maxHeap.push(this.minHeap.shift());
  }

  // 调整堆顶元素
  if (this.maxHeap[0] > this.minHeap[0]) {
    const temp = this.maxHeap[0];
    this.maxHeap[0] = this.minHeap[0];
    this.minHeap[0] = temp;
  }

  // 堆化
  maxHeapify.call(this);
  minHeapify.call(this);
};

/**
 * @return {number}
 */
MedianFinder.prototype.findMedian = function () {
  if ((this.maxHeap.length + this.minHeap.length) % 2 === 0
    return (this.minHeap[0] + this.maxHeap[0]) / 2;
  } else {
    return this.maxHeap[0];
  }
};

/**
 * Your MedianFinder object will be instantiated and calle
```

```
 * var obj = new MedianFinder()
 * obj.addNum(num)
 * var param_2 = obj.findMedian()
 */
```

其中 `minHeapify` 和 `maxHeapify` 的过程都有一个 hack 操作，就是：

```
this.heap.unshift(null);
// ....
this.heap.shift(null);
```

其实就是为了存储的数据从 1 开始，这样方便计算。 即对于下标为 i 的元素， `i >> 1` 一定是父节点的下标。



> 这是因为我用满二叉树来存储的堆

这个实现比较繁琐，下面介绍一种优雅的方式，假设 JS 和 Java 和 C++等语言一样有 `PriorityQueue` 这种数据结构，那么我们实现就比较简单了。

代码：

> 关于 PriorityQueue 的实现，感兴趣的可以看下
> https://github.com/janogonzalez/priorityqueuejs

```
var MedianFinder = function () {
  this.maxHeap = new PriorityQueue((a, b) => a - b);
  this.minHeap = new PriorityQueue((a, b) => b - a);
};

/**
 * @param {number} num
 * @return {void}
 */
MedianFinder.prototype.addNum = function (num) {
  // 我们的目标就是建立两个堆，一个大顶堆，一个小顶堆
  // 结合中位数的特点
  // 这两个堆需要满足：
  // 1. 大顶堆元素都比小顶堆小（由于堆的特点其实只要比较堆顶即可）
  // 2. 大顶堆元素不小于小顶堆，且最多比小顶堆多一个元素

  // 满足上面两个条件的话，如果想要找到中位数，就比较简单了
  // 如果两个堆数量相等（本质是总数为偶数），就两个堆顶元素的平均数
  // 如果两个堆数量不相等（本质是总数为奇数），  就取大顶堆的堆顶元素

  // 问题如果保证满足上述两个特点

  // 1. 保证第一点
  this.maxHeap.enq(num);
  // 由于小顶堆的所有数都来自大顶堆的堆顶元素（最大值）
  // 因此可以保证第一点
  this.minHeap.enq(this.maxHeap.deq());

  // 2. 保证第二点
  if (this.maxHeap.size() < this.minHeap.size()) {
    this.maxHeap.enq(this.minHeap.deq());
  }
};

/**
 * @return {number}
 */
MedianFinder.prototype.findMedian = function () {
  if (this.maxHeap.size() == this.minHeap.size())
    return (this.maxHeap.peek() + this.minHeap.peek()) / 2.
  else return this.maxHeap.peek();
};

/**
 * Your MedianFinder object will be instantiated and calle
 * var obj = new MedianFinder()
 * obj.addNum(num)
```

```
 * var param_2 = obj.findMedian()
 */
```

CPP Code：

```
class MedianFinder {
public:
    /** initialize your data structure here. */
    MedianFinder() {

    }

    void addNum(int num) {
        if (big_queue.empty()) {
            big_queue.push(num);
            return;
        }
        if (big_queue.size() == small_queue.size()) {
            if (num <= big_queue.top()) {
                big_queue.push(num);
            } else {
                small_queue.push(num);
            }
        } else if (big_queue.size() > small_queue.size()) {
            if (big_queue.top() > num) {
                small_queue.push(big_queue.top());
                big_queue.pop();
                big_queue.push(num);
            } else {
                small_queue.push(num);
            }
        } else if (big_queue.size() < small_queue.size()) {
            if (small_queue.top() > num) {
                big_queue.push(num);
            } else {
                big_queue.push(small_queue.top());
                small_queue.pop();
                small_queue.push(num);
            }
        }
    }

    double findMedian() {
        if (big_queue.size() == small_queue.size()) {
            return (big_queue.top() + small_queue.top()) *
        }
        if (big_queue.size() < small_queue.size()) {
            return small_queue.top();
        }
        return big_queue.top();
    }

private:
```

```
    std::priority_queue<int, std::vector<int>, std::greate
    std::priority_queue<int> big_queue; // 最大堆
};
```

题目地址（297. 二叉树的序列化与反序列化）

https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/

题目描述

```
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

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

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

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
```

思路(BFS)

如果我将一个二叉树的完全二叉树形式序列化，然后通过 BFS 反序列化，这不就是力扣官方序列化树的方式么？比如：

```
    1
   / \
  2   3
     / \
    4   5
```

序列化为 "[1,2,3,null,null,4,5]"。 这不就是我刚刚画的完全二叉树么？就是将一个普通的二叉树硬生生当成完全二叉树用了。

> 其实这并不是序列化成了完全二叉树，下面会纠正。

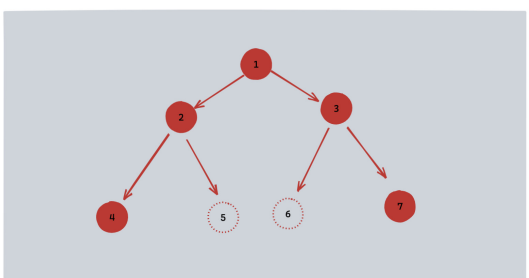
将一颗普通树序列化为完全二叉树很简单，只要将空节点当成普通节点入队处理即可。代码：

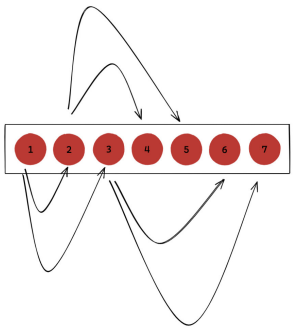
```
class Codec:

    def serialize(self, root):
        q = collections.deque([root])
        ans = ''
        while q:
            cur = q.popleft()
            if cur:
                ans += str(cur.val) + ','
                q.append(cur.left)
                q.append(cur.right)
            else:
                # 除了这里不一样，其他和普通的不记录层的 BFS 没区别
                ans += 'null,'
        # 末尾会多一个逗号，我们去掉它。
        return ans[:-1]
```

细心的同学可能会发现，我上面的代码其实并不是将树序列化成了完全二叉树，这个我们稍后就会讲到。另外后面多余的空节点也一并序列化了。这其实是可以优化的，优化的方式也很简单，那就是去除末尾的 null 即可。

你只要彻底理解我刚才讲的 `我们可以给完全二叉树编号，这样父子之间就可以通过编号轻松求出。比如我给所有节点从左到右从上到下依次从 1 开始编号。那么已知一个节点的编号是 i，那么其左子节点就是 2 * i，右子节点就是 2 * 1 + 1，父节点就是 (i + 1) / 2。` 这句话，那么反序列化对你就不是难事。

如果我用一个箭头表示节点的父子关系，箭头指向节点的两个子节点，那么大概是这样的：





我们刚才提到了：

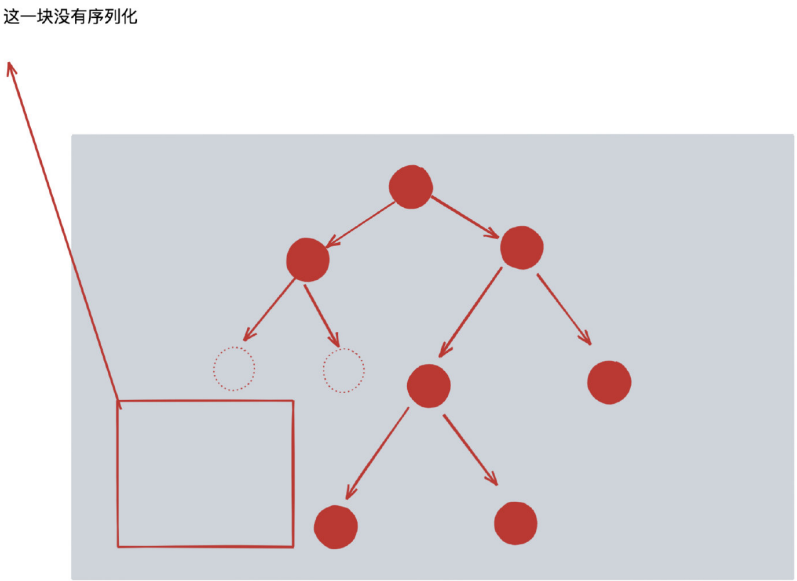
- 1 号节点的两个子节点的 2 号 和 3 号。
- 2 号节点的两个子节点的 4 号 和 5 号。

- 。。。
- i 号节点的两个子节点的 `2 * i` 号 和 `2 * 1 + 1` 号。

此时你可能会写出类似这样的代码：

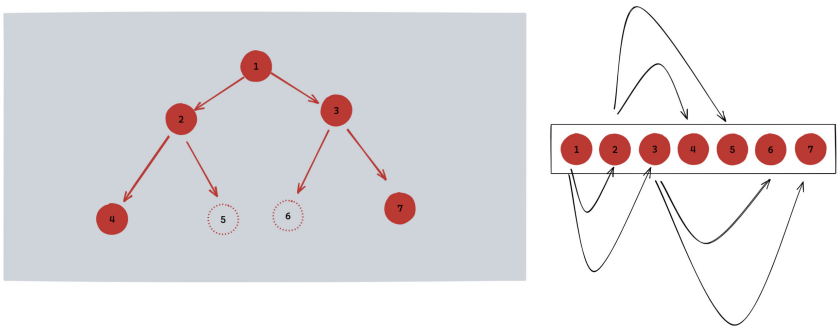
```
    def deserialize(self, data):
        if data == 'null': return None
        nodes = data.split(',')
        root = TreeNode(nodes[0])
        # 从一号开始编号，编号信息一起入队
        q = collections.deque([(root, 1)])
        while q:
            cur, i = q.popleft()
            # 2 * i 是左节点，而 2 * i 编号对应的其实是索引为 2 
            if 2 * i - 1 < len(nodes): lv = nodes[2 * i - 1
            if 2 * i < len(nodes): rv = nodes[2 * i]
            if lv != 'null':
                l = TreeNode(lv)
                # 将左节点和 它的编号 2 * i 入队
                q.append((l, 2 * i))
                cur.left = l
            if rv != 'null':
                r = TreeNode(rv)
                # 将右节点和 它的编号 2 * i + 1 入队
                q.append((r, 2 * i + 1))
                cur.right = r

        return root
```

但是上面的代码是不对的，因为我们序列化的时候其实不是完全二叉树，这也是上面我埋下的伏笔。因此遇到类似这样的 case 就会挂：



这也是我前面说”上面代码的序列化并不是一颗完全二叉树“的原因。

其实这个很好解决， 核心还是上面我画的那种图：



其实我们可以：

- 用三个指针分别指向数组第一项，第二项和第三项（如果存在的话），这里用 p1，p2，p3 来标记，分别表示当前处理的节点，当前处理的节点的左子节点和当前处理的节点的右子节点。
- p1 每次移动一位，p2 和 p3 每次移动两位。
- p1.left = p2; p1.right = p3。
- 持续上面的步骤直到 p1 移动到最后。

因此代码就不难写出了。反序列化代码如下：

```
def deserialize(self, data):
    if data == 'null': return None
    nodes = data.split(',')
    root = TreeNode(nodes[0])
    q = collections.deque([root])
    i = 0
    while q and i < len(nodes) - 2:
        cur = q.popleft()
        lv = nodes[i + 1]
        rv = nodes[i + 2]
        i += 2
        if lv != 'null':
            l = TreeNode(lv)
            q.append(l)
            cur.left = l
        if rv != 'null':
            r = TreeNode(rv)
            q.append(r)
            cur.right = r

    return root
```

这个题目虽然并不是完全二叉树的题目，但是却和完全二叉树很像，有借鉴完全二叉树的地方。

代码

- 代码支持：JS，Python, Go

JS Code：

```
const serialize = (root) => {
  const queue = [root];
  let res = [];
  while (queue.length) {
    const node = queue.shift();
    if (node) {
      res.push(node.val);
      queue.push(node.left);
      queue.push(node.right);
    } else {
      res.push("#");
    }
  }
  return res.join(",");
};

const deserialize = (data) => {
  if (data == "#") return null;

  const list = data.split(",");

  const root = new TreeNode(list[0]);
  const queue = [root];
  let cursor = 1;

  while (cursor < list.length) {
    const node = queue.shift();

    const leftVal = list[cursor];
    const rightVal = list[cursor + 1];

    if (leftVal != "#") {
      const leftNode = new TreeNode(leftVal);
      node.left = leftNode;
      queue.push(leftNode);
    }
    if (rightVal != "#") {
      const rightNode = new TreeNode(rightVal);
      node.right = rightNode;
      queue.push(rightNode);
    }
    cursor += 2;
  }
  return root;
};
```

Python Code:

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Codec:
    def serialize(self, root):
        ans = ''
        queue = [root]
        while queue:
            node = queue.pop(0)
            if node:
                ans += str(node.val) + ','
                queue.append(node.left)
                queue.append(node.right)
            else:
                ans += '#,'
        print(ans[:-1])
        return ans[:-1]



    def deserialize(self, data: str):
        if data == '#': return None
        nodes = data.split(',')
        if not nodes: return None
        root = TreeNode(nodes[0])
        queue = [root]
        # 已经有 root 了，因此从 1 开始
        i = 1

        while i < len(nodes) - 1:
            node = queue.pop(0)
            lv = nodes[i]
            rv = nodes[i + 1]
            i += 2
            if lv != '#':
                l = TreeNode(lv)
                node.left = l
                queue.append(l)

            if rv != '#':
                r = TreeNode(rv)
                node.right = r
                queue.append(r)
        return root
```

Go Code:

```go
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */

type Codec struct {
}

func Constructor() Codec {
    return Codec{}
}

// Serializes a tree to a single string.
func (this *Codec) serialize(root *TreeNode) string {
    ans := ""
    q := []*TreeNode{root} // queue
    var cur *TreeNode
    for len(q) > 0 {
        cur, q = q[0], q[1:]
        if cur != nil {
            ans += strconv.Itoa(cur.Val) + ","
            q = append(q, cur.Left)
            q = append(q, cur.Right)
        } else {
            ans += "#,"
        }
    }
    return ans[:len(ans)-1]
}

// Deserializes your encoded data to tree.
func (this *Codec) deserialize(data string) *TreeNode {
    if data == "#" {
        return nil
    }

    a := strings.Split(data, ",")
    var s string
    s, a = a[0], a[1:]
    v, _ := strconv.Atoi(s)
    root := &TreeNode{Val: v}
    q := []*TreeNode{root} // queue
    var cur, newNode *TreeNode
    for len(a) > 0 {
```

```
            cur, q = q[0], q[1:] // pop

            s, a = a[0], a[1:] // 左子树
            if s != "#" {
                v, _ := strconv.Atoi(s)
                newNode = &TreeNode{Val: v}
                cur.Left = newNode
                q = append(q, newNode)
            }

            if len(a) == 0 {
                return root
            }

            s, a = a[0], a[1:] // 右子树
            if s != "#" {
                v, _ := strconv.Atoi(s)
                newNode = &TreeNode{Val: v}
                cur.Right = newNode
                q = append(q, newNode)
            }
        }
        return root
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为树的节点数。
- 空间复杂度：$O(Q)$，其中 Q 为队列长度，最坏的情况是满二叉树，此时和 N 同阶，其中 N 为树的节点总数

题目地址(301. 删除无效的括号)

https://leetcode-cn.com/problems/remove-invalid-parentheses/

题目描述

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: [""]
```

前置知识

- BFS
- 队列

公司

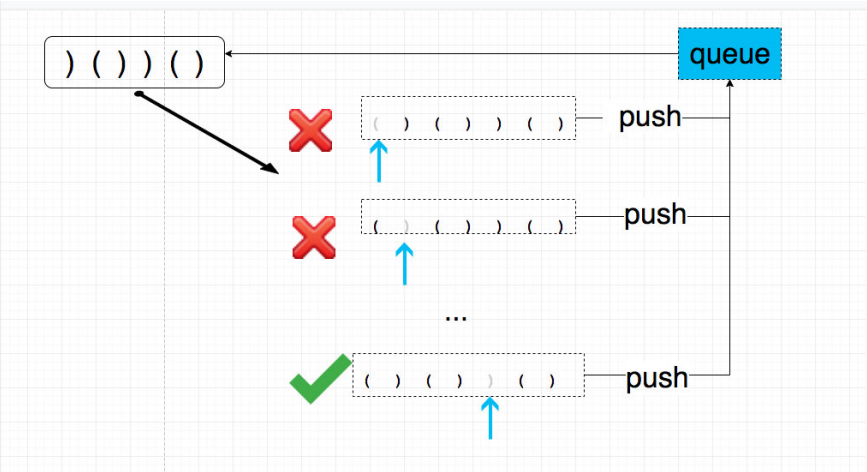
- 阿里
- 腾讯
- 百度
- 字节

思路

我们的思路是先写一个函数用来判断给定字符串是否是有效的。 然后再写一个函数，这个函数 依次删除第i个字符，判断是否有效，有效则添加进最终的返回数组。

这样的话实现的功能就是， 删除 一个 小括号使之有效的所有可能。因此只需要递归调用 依次删除第i个字符 的功能就可以了。

而且由于题目要求是要删除最少的小括号，因此我们的思路是使用广度优先遍历，而不是深度有限的遍历。



> 没有动图，请脑补

关键点解析

- 广度优先遍历
- 使用队列简化操作
- 使用一个visited的mapper， 来避免遍历同样的字符串

代码

```
var isValid = function(s) {
  let openParenthes = 0;
  for(let i = 0; i < s.length; i++) {
    if (s[i] === '(') {
      openParenthes++;
    } else if (s[i] === ')') {
      if (openParenthes === 0) return false;
      openParenthes--;
    }
  }
  return openParenthes === 0;
};
/**
 * @param {string} s
 * @return {string[]}
 */
var removeInvalidParentheses = function(s) {
  if (!s || s.length === 0) return [""];
  const ret = [];
  const queue = [s];
  const visited = {};
  let current = null;
  let removedParentheses = 0; // 只记录最小改动

  while ((current = queue.shift())) {
    let hit = isValid(current);
    if (hit) {
      if (!removedParentheses) {
       removedParentheses =  s.length - current.length
      }
      if (s.length - current.length > removedParentheses) r
      ret.unshift(current);
      continue;
    }
    for (let i = 0; i < current.length; i++) {
      if (current[i] !== ')' && current[i] !== '(') continu
      const subString = current.slice(0, i).concat(current.
      if (visited[subString]) continue;
      visited[subString] = true;
      queue.push(subString);
    }
  }

  return ret.length === 0 ? [""] : ret;
};
```

扩展

相似问题:

validParentheses

题目地址（312. 戳气球）

https://leetcode-cn.com/problems/burst-balloons/

题目描述

```
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

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 nums[l

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
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
```

前置知识

- 回溯法
- 动态规划

公司

- 阿里
- 腾讯
- 百度
- 字节

回溯法（超时）

回溯法

这道题就是要戳破所有的气球，求获得硬币的最大数量。我的第一反应就是暴力回溯。

但是这种暴力算法肯定会超时，为什么呢？因为题目给的气球数量有点多，最多 500 个；500 的阶乘，会超时爆栈；但是我们依然写一下代码，找下突破口，小伙伴们千万不要看不起暴力，暴力是优化的突破口；

如果小伙伴对回溯法不太熟悉，我建议你记住下面的模版，也可以看我之前写的文章，回溯法基本可以使用以下的模版写。

代码

```
var maxCoins = function (nums) {
  let res = Number.MIN_VALUE;
  backtrack(nums, 0);
  return res;
  // 回溯法，状态树很大
  function backtrack(nums, score) {
    if (nums.length == 0) {
      res = Math.max(res, score);
      return;
    }
    for (let i = 0, n = nums.length; i < n; i++) {
      let point =
        (i - 1 < 0 ? 1 : nums[i - 1]) *
        nums[i] *
        (i + 1 >= n ? 1 : nums[i + 1]);
      let tempNums = [].concat(nums);
      // 做选择 在 nums 中删除元素 nums[i]
      nums.splice(i, 1);
      // 递归回溯
      backtrack(nums, score + point);
      // 撤销选择
      nums = [...tempNums];
    }
  }
};
```

动态规划

思路

回溯法的缺点也很明显，复杂度很高，小伙伴们可以脑补一下执行过程的状态树，这里我偷个懒就不画了；

通过仔细观察这个状态树，我们会发现这个状态树的【选择】上，会有一些重复的选择分支；很明显存在了重复子问题；自然我就想到了能不能用动态规划来解决；

判读能不能用动态规划解决，还有一个问题，就是必须存在最优子结构；什么意思呢？其实就是根据局部最优，推导出答案；假设我们**戳破第 k 个气球**是最优策略的最后一步，和上一步有没有联系呢？根据题目意思，戳破第 k 个，前一个和后一个就变成相邻的了。**由于这种不稳定性，导致问题难以处理。一种解决方案是反向思考**。即我们不是给你一个 nums，一个个移除数。而是从空数组开始一个个添加。

> 由于题目说明了 nums 左右各存在一个虚拟的气球，因此这里说的空数组实际指的是 [1,1] 这种情况，即只有两个虚拟数字。

经过这样的反向思考，问题就变得简单起来了。经过这样的思考之后就使用动态规划解决就 ok 了。

既然用动态规划，那就老套路了，把动态规划的三个问题想清楚定义好；然后找出题目的【状态】和【选择】，然后根据【状态】枚举，枚举的过程中根据【选择】计算递推就能得到答案了。

那本题的【选择】是什么呢？就是戳哪一个气球。那【状态】呢？就是题目给的气球数量。

1. 定义状态

这里有个细节，就是题目说明有两个虚拟气球，nums[-1] = nums[n] = 1；如果当前戳破的气球是最后一个或者第一个，前面/后面没有气球了，不能乘以 0，而是乘以 1。

定义状态的最关键两个点，往子问题（问题规模变小）想，最后一步最优策略是什么；我们假设最后戳破的气球是 k，戳破 k 获得最大数量的银币就是 `nums[i] * nums[k] * nums[j]` 再加上前面戳破的最大数量和后面的最大数量，即：`nums[i] * nums[k] * nums[j] + 前面最大数量 + 后面最大数量` 。

那我们可以这样来定义状态，dp[i][j] = x 表示：戳破气球 i 和气球 j 之间（开区间，不包括 i 和 j）的所有气球，可以获得的最大硬币数为 x。为什么开区间？因为不能和已经计算过的产生联系，我们这样定义之后，利用**两个虚拟气球** 就可完成本题。

1. 状态转移方程

而对于 dp[i][j]，i 和 j 之间会有很多气球，到底该戳哪个先呢？我们直接设为 k，枚举选择最优的 k 就可以了。所以，最终的状态转移方程为：`dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + nums[k] * nums[i] * nums[j])` 。由于是开区间，因此 k 为 i + 1, i + 2... j - 1。

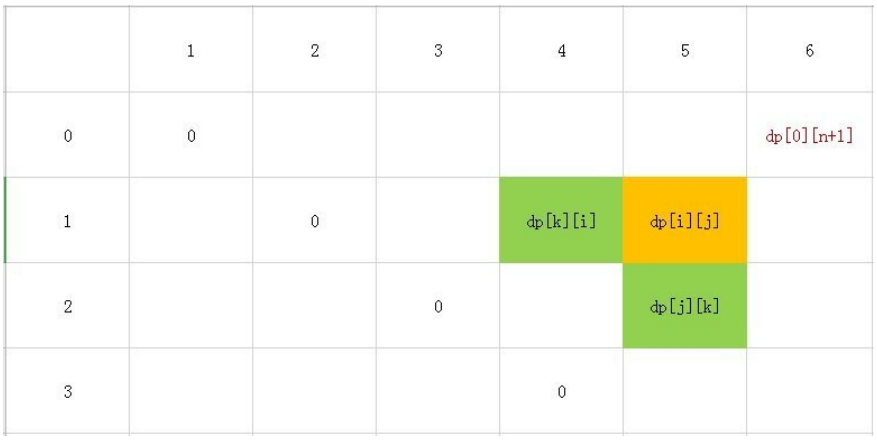
> 这就是典型的枚举分割点 DP，大家一定要掌握哦~

1. 初始值和边界

由于我们利用了两个虚拟气球，边界就是气球数 n + 2，当 i == j 时，很明显两个之间没有气球，为 0；

1. 如何枚举状态

因为我们最终要求的答案是 dp[0][n + 1]，就是戳破虚拟气球之间的所有气球获得的最大值。当 i == j 时，i 和 j 之间是没有气球的，所以枚举的状态很明显是 dp table 的左上部分，也就是 j 大于 i，如下图所示，只给出一部分方便思考。



> 图有错误。图中 dp[k][i] 应该是 dp[i][k]，dp[j][k] 应该是 dp[k][j]

从上图可以看出，我们需要从下到上，从左到右进行遍历。

关键点

- 这道题属于是切割型 dp
- 反向思考
- 遍历方向的确定

代码

代码支持： JS, Python

JS Code:

```
var maxCoins = function (nums) {
  let n = nums.length;
  // 添加两侧的虚拟气球
  let points = [1, ...nums, 1];
  let dp = Array.from(Array(n + 2), () => Array(n + 2).fil
  // 最后一行开始遍历,从下往上
  for (let i = n; i >= 0; i--) {
    // 从左往右
    for (let j = i + 1; j < n + 2; j++) {
      for (let k = i + 1; k < j; k++) {
        dp[i][j] = Math.max(
          dp[i][j],
          points[j] * points[k] * points[i] + dp[i][k] + dp
        );
      }
    }
  }
  return dp[0][n + 1];
};
```

Python Code:

```
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)
        points = [1] + nums + [1]
        dp = [[0] * (n + 2) for _ in range(n + 2)]

        for i in range(n, -1, -1):
            for j in range(i + 1, n + 2):
                for k in range(i + 1, j):
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp
        return dp[0][-1]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n ^ 3)$
- 空间复杂度：$O(n ^ 2)$

如果使用记忆化递归，时间复杂度和上面一样，空间复杂度是 $O(n)$，但是在力扣提交会超时，大家作为参考即可。

Python3 Code:

```
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)
        nums = [1] + nums + [1]

        @lru_cache(None)
        def dp(left, right):
            if left + 1 == right:
                return 0
            if left + 2 == right:
                return nums[left] * nums[left + 1] * nums[
            ans = 0
            for i in range(left + 1, right):
                ans = max(ans, nums[i] * nums[left] * nums
            return ans

        return dp(0, len(nums) - 1)
```

相关题目

- Maximum-Additive-Score-by-Removing-Numbers

总结

简单的 dp 题目会直接告诉你怎么定义状态，告诉你怎么选择计算，你只需要根据套路判断一下能不能用 dp 解题即可，而判断能不能，往往暴力就是突破口。

这道题如果从空数组反向思考，则避免了因为数组变化导致的状态变化而难以处理的问题，是一种常见的技巧。另外此题属于典型的分割 DP 问题。分割 DP 问题通常都是两层循环枚举所有的左右端点，再用一层循环枚举所有的割点，也就是三层循环，时间复杂度也是 $O(n^3)$。

更多题解可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 30K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(330. 按要求补齐数组)

https://leetcode-cn.com/problems/patching-array/

题目描述

```
给定一个已排序的正整数数组 nums，和一个正整数 n 。从 [1, n] 区间内选

示例 1:

输入: nums = [1,3], n = 6
输出: 1
解释:
根据 nums 里现有的组合 [1], [3], [1,3]，可以得出 1, 3, 4。
现在如果我们将 2 添加到 nums 中， 组合变为: [1], [2], [3], [1,3]
其和可以表示数字 1, 2, 3, 4, 5, 6，能够覆盖 [1, 6] 区间里所有的数
所以我们最少需要添加一个数字。
示例 2:

输入: nums = [1,5,10], n = 20
输出: 2
解释: 我们需要添加 [2, 4]。
示例 3:

输入: nums = [1,2,2], n = 5
输出: 0
```

前置知识

- 贪心
- 前缀和

公司

- 暂无

思路

这道题核心点正如标题所言： **贪心 + 维护端点信息**。

贪心的思想这里不多说了，思路和官方题解是一样的。

先不考虑需要增加数字的情况，即没有任何缺失的数字。

这里给了几个例子方便大家理解。

> 左侧是 nums 数组， 右侧是 nums 可以覆盖的区间 [start, end] （注意是左右都闭合）。当然如果你写出别的形式，比如左闭右开，那么代码要做一些调整。

[1] -> [1,1] [1,2] -> [1,3] [1,2,3] -> [1,6] [1,2,3,4] -> [1,10]

可以看出，可以覆盖的区间，总是 [1, x] ，其中 x 为 nums 的和。

接下来，我们考虑有些数字缺失导致无法覆盖的情况。

算法：

1. 初始化覆盖区间为 [0, 0] 表示啥都没覆盖，目标区间是 [1, n]
2. 如果数组当前数字无法达到前缀和，那么需要补充数字，更新区间为 [1, 前缀和]。
3. 如果数组当前数字无法达到前缀和，则什么都不需要做。

那么第二步补充数字的话需要补充什么数字呢？如果当前区间是 [1,x]，我们应该添加数字 x + 1，这样可以覆盖的区间为 [1,2*x+1]。如果你选择添加小于 x + 1 的数字，达到的效果肯定没这个区间大。而如果你选择添加大于 x + 1 的数字，那么会导致 x + 1 无法被覆盖。这就是贪心的思想。

关键点解析

- 维护端点信息，并用前缀和更新区间

代码

代码变量说明：

- furthest 表示区间右端点
- i 表示当前遍历到的数组索引
- ans 是需要返回的答案

```
class Solution:
    def minPatches(self, nums: List[int], n: int) -> int:
        furthest = i = ans = 0
        while furthest < n:
            # 可覆盖到，直接用前缀和更新区间
            if i < len(nums) and nums[i] <= furthest + 1:
                furthest += nums[i] #  [1, furthest] -> [1,
                i += 1
            else:
                # 不可覆盖到，增加一个数 furthest + 1，并用前缀和
                # 如果 nums[i] > furthest + 1，说明我们必须添加
                furthest = 2 * furthest + 1 # [1, furthest]
                ans += 1
        return ans
```

如果你的区间信息是左闭右开的，代码可以这么写：

```
class Solution:
    def minPatches(self, nums: List[int], n: int) -> int:
        furthest, i, ans = 1, 0, 0
        # 结束条件也要相应改变
        while furthest <= n:
            if i < len(nums) and nums[i] <= furthest:
                furthest += nums[i] #  [1, furthest) -> [1,
                i += 1
            else:
                furthest = 2 * furthest # [1, furthest) ->
                ans += 1
        return ans
```

复杂度分析

- 时间复杂度：$O(N)$。
- 空间复杂度：$O(1)$。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（335. 路径交叉）

https://leetcode-cn.com/problems/self-crossing/

题目描述

```
给定一个含有 n 个正数的数组 x。从点 (0,0) 开始，先向北移动 x[0] 米，

编写一个 O(1) 空间复杂度的一趟扫描算法，判断你所经过的路径是否相交。



示例 1:

┌────┐
│    │
└────┼──>
     │

输入: [2,1,1,2]
输出: true
示例 2:

┌──────┐
│      │
│
│
└──────────────>

输入: [1,2,3,4]
输出: false
示例 3:

┌────┐
│    │
└────┼>

输入: [1,1,1,1]
输出: true
```

前置知识

- 滑动窗口

公司

- 暂无

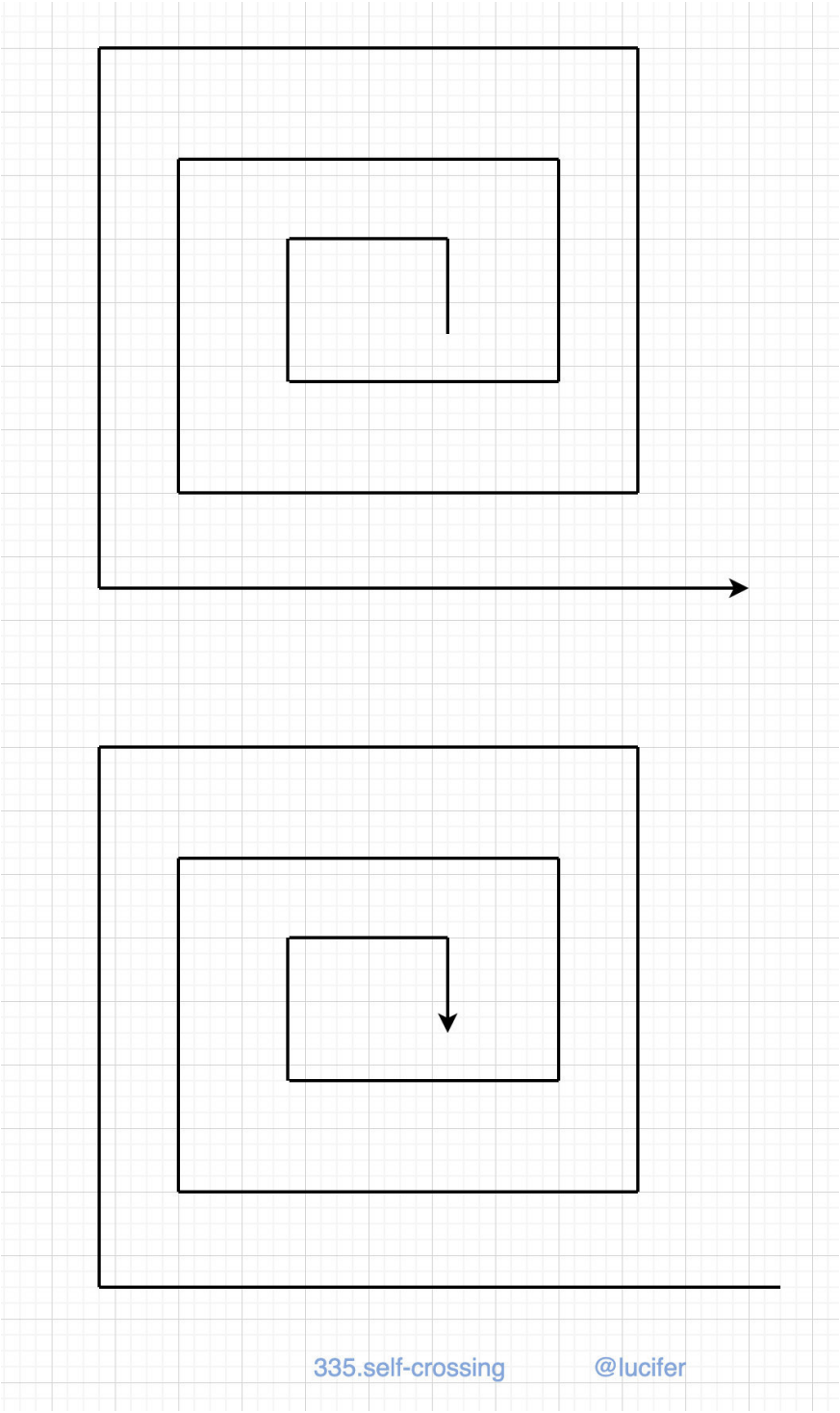
思路

符合直觉的做法是$O(N)$时间和空间复杂度的算法。这种算法非常简单，但是题目要求我们使用空间复杂度为$O(1)$的做法。

关于空间复杂度为$O(N)$的算法可以参考我之前的874.walking-robot-simulation。 思路基本是类似，只不过 obstacles（障碍物）不是固定的，而是我们不断遍历的时候动态生成的，我们每遇到一个点，就将其标记为 obstacle。随着算法的进行，我们的 obstacles 逐渐增大，最终和 N 一个量级。

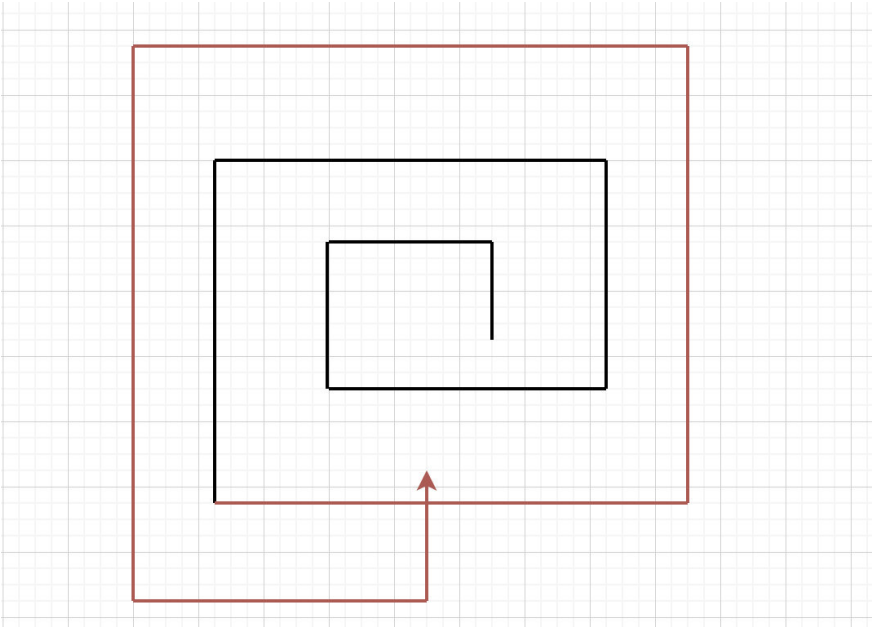
我们考虑进行优化。我们仔细观察发现，如果想让其不相交，从大的范围来看只有两种情况：

1. 我们画的圈不断增大。
2. 我们画的圈不断减少。



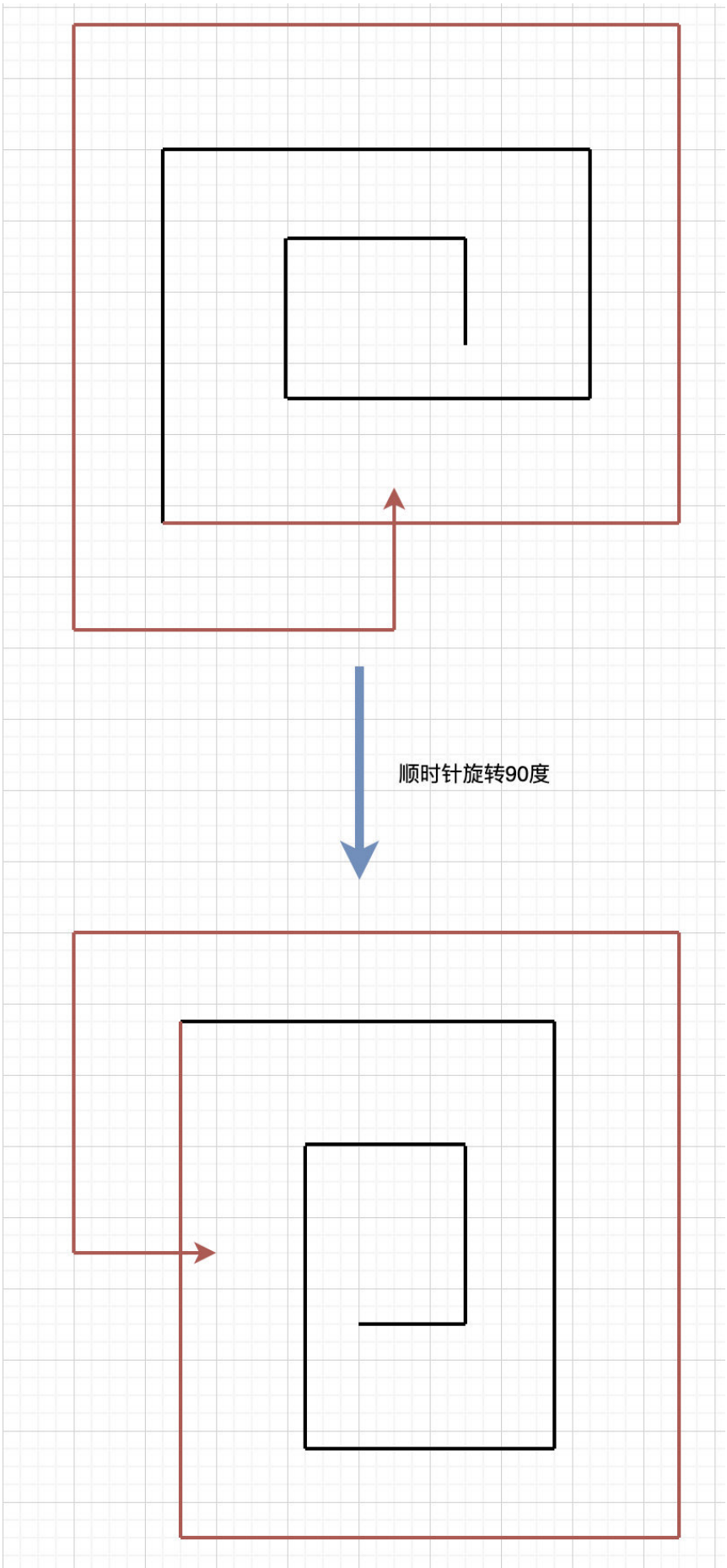
（有没有感觉像迷宫？）

这样我们会发现，其实我们画最新一笔的时候，并不是之前画的所有的都需要考虑，我们只需要最近的几个就可以了，实际上是最近的五个，不过不知道也没关系，我们稍后会讲解。



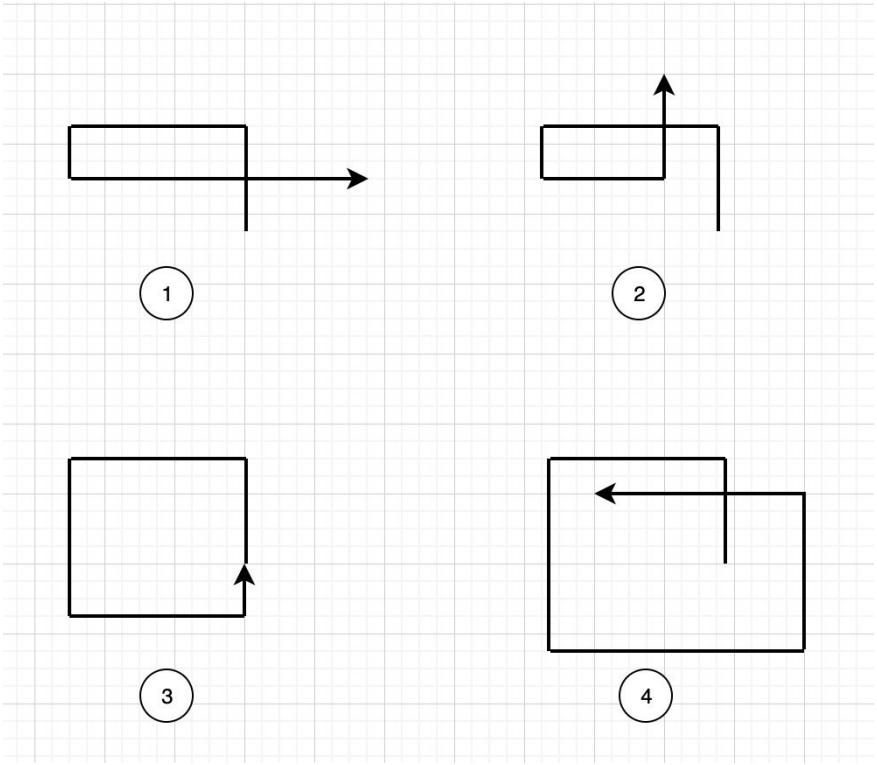
红色部分指的是我们需要考虑的，而剩余没有被红色标注的部分则无需考虑。不是因为我们无法与之相交，而是我们 一旦与之相交，则必然我们也一定会与红色标记部分相交 。

然而我们画的方向也是不用考虑的。比如我当前画的方向是从左到右，那和我画的方向是从上到下有区别么？在这里是没区别的，不信我帮你将上图顺时针旋转 90 度看一下：


顺时针旋转90度

方向对于我们考虑是否相交没有差别。

当我们仔细思考的时候，会发现其实相交的情况只有以下几种：



这个时候代码就呼之欲出了。

- 我们只需要遍历数组 x，假设当前是第 i 个元素。
- 如果 x[i] >= x[i - 2] and x[i - 1] <= x[i - 3]，则相交（第一种情况）
- 如果 x[i - 1] <= x[i - 3] and x[i - 2] <= x[i]，则相交（第二种情况）
- 如果 i > 3 and x[i - 1] == x[i - 3] and x[i] + x[i - 4] == x[i - 2]，则相交（第三种情况）
- 如果 i > 4 and x[i] + x[i - 4] >= x[i - 2] and x[i - 1] >= x[i - 3] - x[i - 5] \ and x[i - 1] <= x[i - 3] and x[i - 2] >= x[i - 4] and x[i - 3] >= x[i - 5] ，则相交（第四种情况）
- 否则不相交

关键点解析

- 一定要画图辅助
- 对于这种$O(1)$空间复杂度有固定的套路。常见的有：
- 直接修改原数组
- 滑动窗口（当前状态并不是和之前所有状态有关，而是仅和某几个有关）。

我们采用的是滑动窗口。但是难点就在于我们怎么知道当前状态和哪几个有关。对于这道题来说，画图或许可以帮助你打开思路。另外面试的时候说出$O(N)$的思路也不失为一个帮助你冷静分析问题的手段。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def isSelfCrossing(self, x: List[int]) -> bool:
        n = len(x)
        if n < 4:
            return False
        for i in range(3, n):
            if x[i] >= x[i - 2] and x[i - 1] <= x[i - 3]:
                return True
            if x[i - 1] <= x[i - 3] and x[i - 2] <= x[i]:
                return True
            if i > 3 and x[i - 1] == x[i - 3] and x[i] + x
                return True
            if i > 4 and x[i] + x[i - 4] >= x[i - 2] and x
                    and x[i - 1] <= x[i - 3] and x[i - 2] >
                return True
        return False
```

复杂度分析

其中 N 为数组长度。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(1)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(460. LFU缓存)

https://leetcode-cn.com/problems/lfu-cache/

题目描述

```
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

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
「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

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
```

前置知识

- 链表
- HashMap

公司

- 阿里
- 腾讯

- 百度
- 字节

思路

本题已被收录到我的新书中，敬请期待～

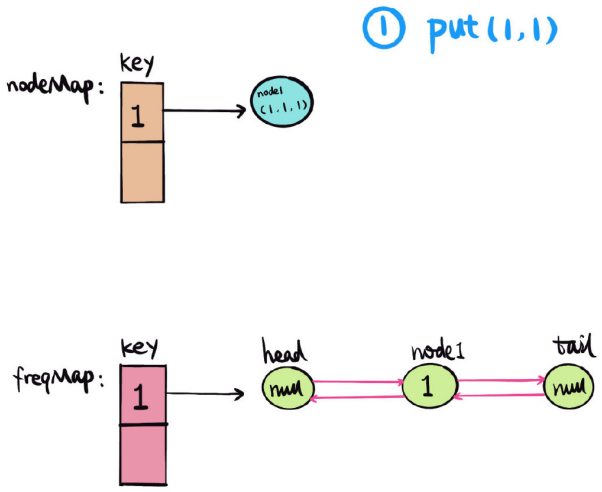
LFU（Least frequently used) 但内存容量满的情况下，有新的数据进来，需要更多空间的时候，就需要删除被访问频率最少的元素。

举个例子，比如说 cache 容量是 3，按顺序依次放入 `1，2，1，2，1，3` ， cache 已存满 3 个元素 (1，2，3)， 这时如果想放入一个新的元素 4 的时候，就需要腾出一个元素空间。 用 LFU，这里就淘汰 3， 因为 3 的次数只出现依次， 1 和 2 出现的次数都比 3 多。

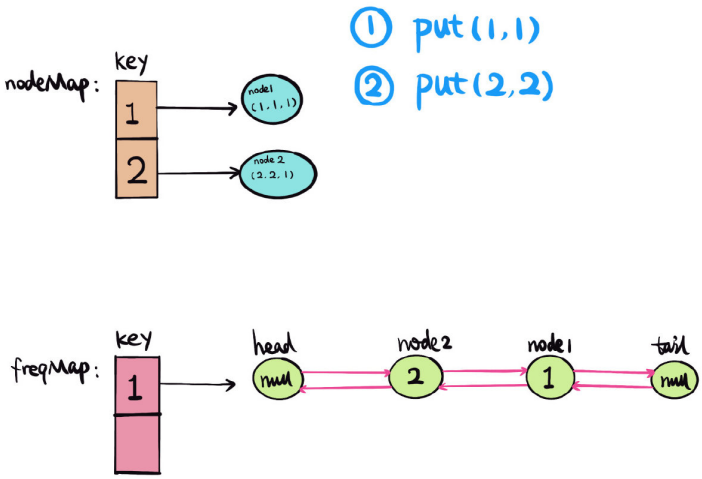
题中 `get` 和 `put` 都是 `O(1)` 的时间复杂度，那么删除和增加都是 `O(1)` ，可以想到用双链表，和 `HashMap` ，用一个 `HashMap, nodeMap,` 保存当前 `key` ，和 `node{key, value, frequent}` 的映射。 这样 `get(key)` 的操作就是 `O(1)` . 如果要删除一个元素，那么就需要另一个 `HashMap，freqMap，` 保存元素出现次数 `（frequent）` 和双链表 `（DoublyLinkedlist）` 映射， 这里双链表存的是 frequent 相同的元素。每次 `get` 或 `put` 的时候， `frequent+1` ，然后把 `node` 插入到双链表的 `head node, head.next=node` 每次删除 `freqent` 最小的双链表的 `tail node, tail.prev` 。

用给的例子举例说明：

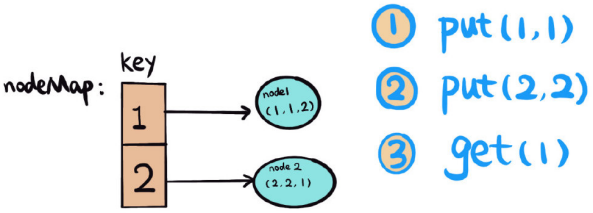
```
1. put(1, 1),
   - 首先查找 nodeMap 中有没有 key=1 对应的 value,
       没有就新建 node(key, value, freq) -> node1(1, 1, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value,
       没有就新建 doublylinkedlist(head, tail), 把 node1 插入
   如下图,
```

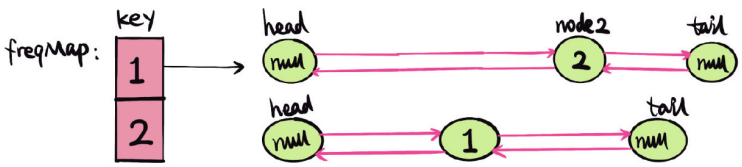



```
2. put(2, 2),
   - 首先查找 nodeMap 中有没有 key=2 对应的 value,
       没有就新建 node(key, value, freq) -> node2(2, 2, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value,
       没有就新建 doublylinkedlist(head, tail), 把 node2 插入
   如下图,
```

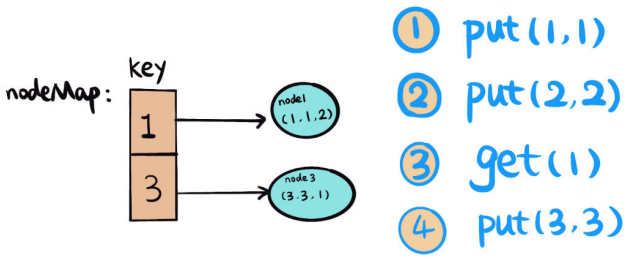


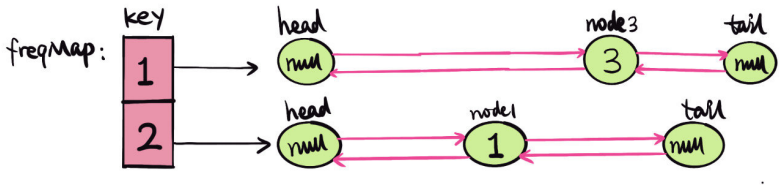
3. get(1),
   - 首先查找 nodeMap 中有没有 key=1 对应的 value，nodeMap:{[1
     找到 node1，把 node1 freq+1 -> node1(1,1,2)
   - 更新 freqMap，删除 freq=1，node1
   - 更新 freqMap，插入 freq=2，node1
   如下图，





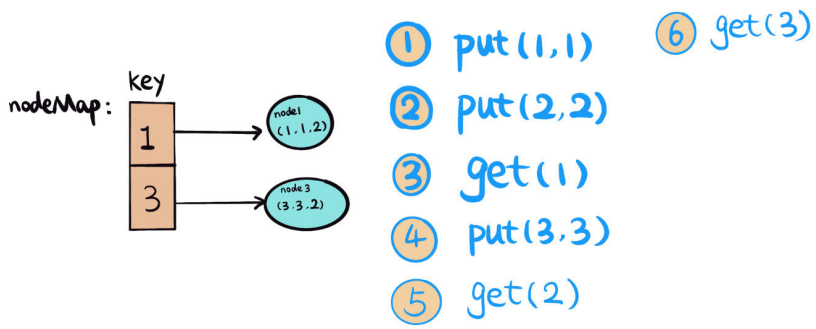
4. put(3, 3),
   - 判断 cache 的 capacity，已满，需要淘汰使用次数最少的元素，找
     如果 tailnode.prev != null，删除。然后从 nodeMap 中删除
   - 首先查找 nodeMap 中有没有 key=3 对应的 value，
     没有就新建 node(key, value, freq) -> node3(3, 3, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value，
     没有就新建 doublylinkedlist(head, tail)，把 node3 插入
   如下图，

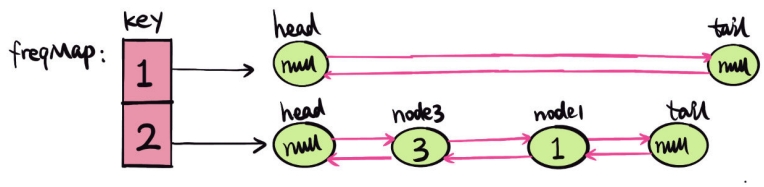




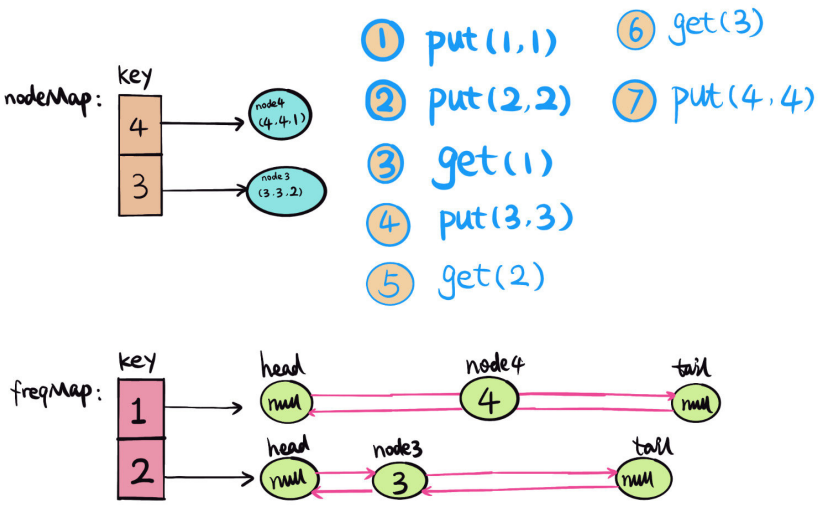
5. get(2)
   - 查找 nodeMap，如果没有对应的 key 的 value，返回 -1。

6. get(3)
   - 首先查找 nodeMap 中有没有 key=3 对应的 value，nodeMap:{[1
     找到 node3，把 node3 freq+1 -> node3(3,3,2)
   - 更新 freqMap，删除 freq=1，node3
   - 更新 freqMap，插入 freq=2，node3
   如下图，



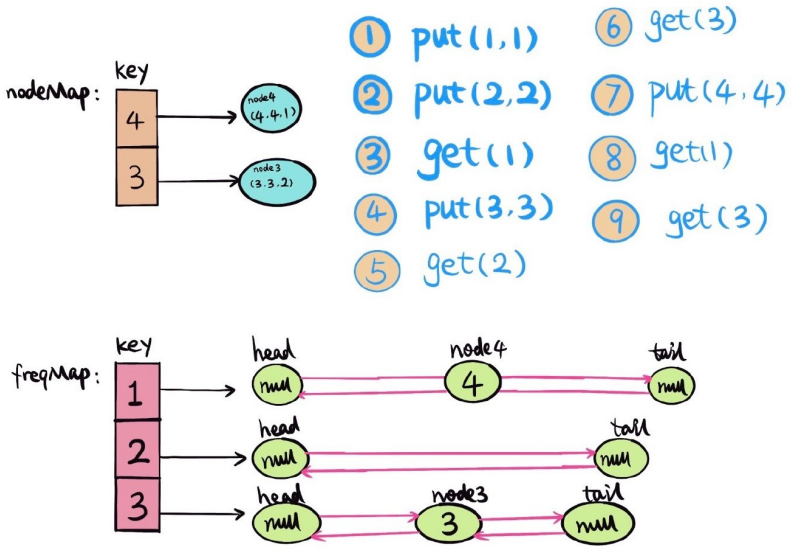


7. put(4, 4),
   - 判断 cache 的 capacity，已满，需要淘汰使用次数最少的元素，找
     如果 tailnode.prev != null，删除。然后从 nodeMap 中删除
   - 首先查找 nodeMap 中有没有 key=4 对应的 value，
     没有就新建 node(key, value, freq) -> node4(4, 4, 1),
   - 查找 freqMap 中有没有 freq=1 对应的 value，
     没有就新建 doublylinkedlist(head, tail)，把 node4 插入
   如下图，

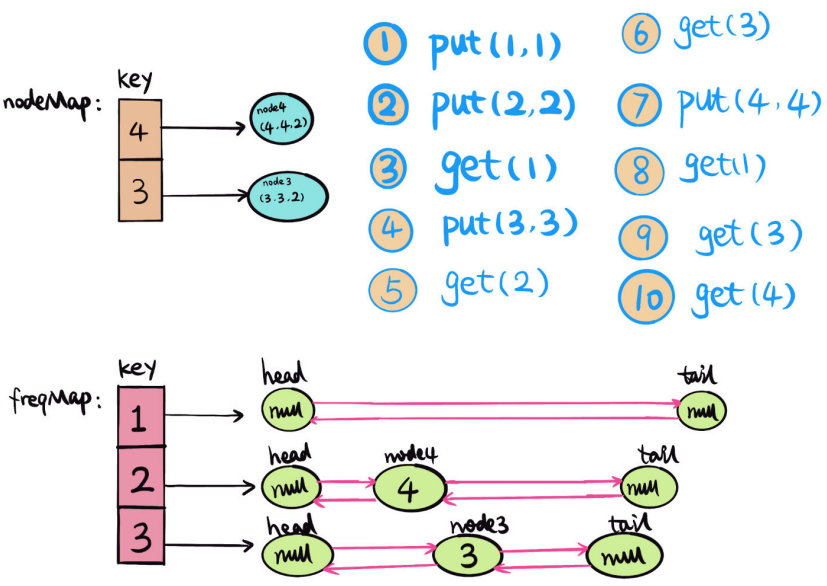


8. get(1)
   - 查找 nodeMap，如果没有对应的 key 的 value，返回 -1。

9. get(3)
   - 首先查找 nodeMap 中有没有 key=3 对应的 value，nodeMap:{[4
     找到 node3，把 node3 freq+1 -> node3(3,3,3)
   - 更新 freqMap，删除 freq=2，node3
   - 更新 freqMap，插入 freq=3，node3
   如下图，




```
10. get(4)
   - 首先查找 nodeMap 中有没有 key=4 对应的 value, nodeMap:{[4
       找到 node4, 把 node4 freq+1 -> node4(4,4,2)
   - 更新 freqMap, 删除 freq=1, node4
   - 更新 freqMap, 插入 freq=2, node4
   如下图,
```



关键点分析

用两个 `Map` 分别保存 `nodeMap {key, node}` 和 `freqMap{frequent, DoublyLinkedList}` 。实现 `get` 和 `put` 操作都是 `O(1)` 的时间复杂度。

可以用 Java 自带的一些数据结构，比如 HashLinkedHashSet，这样就不需要自己自建 Node，DoublelyLinkedList。可以很大程度的缩减代码量。

代码（Java code）

```
public class LC460LFUCache {
  class Node {
    int key, val, freq;
    Node prev, next;

    Node(int key, int val) {
      this.key = key;
      this.val = val;
      freq = 1;
    }
  }

  class DoubleLinkedList {
    private Node head;
    private Node tail;
    private int size;

    DoubleLinkedList() {
      head = new Node(0, 0);
      tail = new Node(0, 0);
      head.next = tail;
      tail.prev = head;
    }

    void add(Node node) {
      head.next.prev = node;
      node.next = head.next;
      node.prev = head;
      head.next = node;
      size++;
    }

    void remove(Node node) {
      node.prev.next = node.next;
      node.next.prev = node.prev;
      size--;
    }

    // always remove last node if last node exists
    Node removeLast() {
      if (size > 0) {
        Node node = tail.prev;
        remove(node);
        return node;
      } else return null;
    }
  }
```

```
// cache capacity
private int capacity;
// min frequent
private int minFreq;
Map<Integer, Node> nodeMap;
Map<Integer, DoubleLinkedList> freqMap;
public LC460LFUCache(int capacity) {
  this.minFreq = 0;
  this.capacity = capacity;
  nodeMap = new HashMap<>();
  freqMap = new HashMap<>();
}

public int get(int key) {
  Node node = nodeMap.get(key);
  if (node == null) return -1;
  update(node);
  return node.val;
}

public void put(int key, int value) {
  if (capacity == 0) return;
  Node node;
  if (nodeMap.containsKey(key)) {
    node = nodeMap.get(key);
    node.val = value;
    update(node);
  } else {
    node = new Node(key, value);
    nodeMap.put(key, node);
    if (nodeMap.size() == capacity) {
      DoubleLinkedList lastList = freqMap.get(minFreq);
      nodeMap.remove(lastList.removeLast().key);
    }
    minFreq = 1;
    DoubleLinkedList newList = freqMap.getOrDefault(node.
    newList.add(node);
    freqMap.put(node.freq, newList);
  }
}

private void update(Node node) {
  DoubleLinkedList oldList = freqMap.get(node.freq);
  oldList.remove(node);
  if (node.freq == minFreq && oldList.size == 0) minFreq-
  node.freq++;
  DoubleLinkedList newList = freqMap.getOrDefault(node.fr
  newList.add(node);
```

```
      freqMap.put(node.freq, newList);
    }
  }
```

参考（References）

1. LFU(Least frequently used) Cache
2. Leetcode discussion mylzsd-Solution-Using-Two-HashMap-and-One-DoubleLinkedList)
3. Leetcode discussion aaaeeeo-Solution-Using-Two-HashMap-and-One-DoubleLinkedList)

题目地址（472. 连接词）

https://leetcode-cn.com/problems/concatenated-words/

题目描述

```
给定一个不含重复单词的列表，编写一个程序，返回给定单词列表中所有的连接

连接词的定义为：一个字符串完全是由至少两个给定数组中的单词组成的。

示例:

输入: ["cat","cats","catsdogcats","dog","dogcatsdog","hippo

输出: ["catsdogcats","dogcatsdog","ratcatdogcat"]

解释: "catsdogcats"由"cats", "dog" 和 "cats"组成;
     "dogcatsdog"由"dog", "cats"和"dog"组成;
     "ratcatdogcat"由"rat", "cat", "dog"和"cat"组成。
说明:

给定数组的元素总数不超过 10000。
给定数组中元素的长度总和不超过 600000。
所有输入字符串只包含小写字母。
不需要考虑答案输出的顺序。
```

前置知识

- 前缀树

公司

- 阿里
- 字节

思路

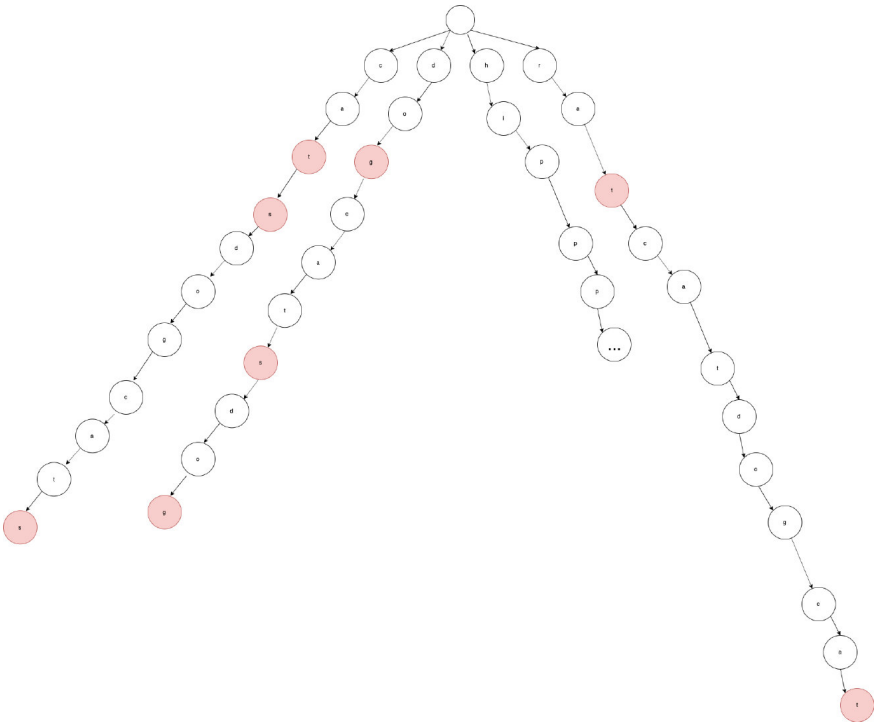
本题我的思路是直接使用前缀树来解决。**标准的前缀树模板**我在之前的题解中提到了，感兴趣的可以到下方的相关题目中查看。

这道题这里我们不需要 search，我们的做法是：

- 先进行一次遍历，将 words 全部插入（insert）到前缀树中。

- 然后再进行一次遍历，查找每一个单词有几个单词表中的单词组成
- 如果大于 2，则将其加入到 res 中
- 最后返回 res 即可

我们构造的前缀树大概是这样的：



问题的关键在于第二步中的**查找每一个单词有几个单词表中的单词组成**。其实如果你了解前缀树的话应该不难写出来。 比如查找 catsdogcats：

- 我们先从 c 到 a 到 t，发现 t 是单词结尾，我们数量 + 1
- 然后将剩下的部分“sdogcats”重新执行上述过程。
- s 发现找不到，我们返回 0
- 因此最终结果是 1

很明显这个逻辑是错误的，正确的划分应该是：

- 我们先从 c 到 a 到 t，再到 s，此时数量 + 1
- 然后将剩下的“dogcats”重复上述过程
- dog 找到了，数量 + 1
- 最后将 cats 加入。也找到了，数量再加 1

由于我们并不知道 cat 这里断开，结果更大？还是 cats 这里断开结果更大？因此我们的做法是将其全部递归求出，然后取出最大值即可。如果我们直接这样递归的话，可能会超时，卡在最后一个测试用例上。一个简单的方式是记忆化递归，从而避免重复计算，经测试这种方法能够通过。

关键点分析

- 前缀树

代码

代码支持：Python3

Python3 Code:

```
class Trie:

    def __init__(self):
        self.Trie = {}
        self.visited = {}

    def insert(self, word):
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def cntWords(self, word):
        if not word:
            return 0
        if word in self.visited:
            return self.visited[word]
        curr = self.Trie
        res = float('-inf')

        for i, w in enumerate(word):
            if w not in curr:
                return res
            curr = curr[w]
            if '#' in curr:
                res = max(res, 1 + self.cntWords(word[i + 1
        self.visited[word] = res
        return res


class Solution:
    def findAllConcatenatedWordsInADict(self, words: List[s
        self.trie = Trie()
        res = []

        for word in words:
            self.trie.insert(word)
        for word in words:
            if self.trie.cntWords(word) >= 2:
                res.append(word)
        return res
```

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0820.short-encoding-of-words
- 1032.stream-of-characters

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。目前已经 37K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(480. 滑动窗口中位数)

https://leetcode-cn.com/problems/sliding-window-median/

题目描述

中位数是有序序列最中间的那个数。如果序列的大小是偶数，则没有最中间的数

例如：

```
[2,3,4]，中位数是 3
[2,3]，中位数是 (2 + 3) / 2 = 2.5
```

给你一个数组 nums，有一个大小为 k 的窗口从最左端滑动到最右端。窗口中有

示例：

给出 nums = [1,3,-1,-3,5,3,6,7]，以及 k = 3。

```
窗口位置                      中位数
---------------               -----
[1  3  -1] -3  5  3  6  7       1
 1 [3  -1  -3] 5  3  6  7      -1
 1  3 [-1  -3  5] 3  6  7      -1
 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]      6
```

因此，返回该滑动窗口的中位数数组 [1,-1,-1,3,5,6]。

提示：

你可以假设 k 始终有效，即：k 始终小于输入的非空数组的元素个数。
与真实值误差在 10 ^ -5 以内的答案将被视作正确答案。

前置知识

- 二分查找

公司

- 暂无

思路

每次窗口移动都伴随左侧移除一个，右侧添加一个。而中位数是排序之后的中间数字。因此我们的思路是维护一个大小为 k 的有序数组，这个有序数组就是窗口内的数组**排序之后**的结果。

而在一个有序数组添加和移除数字，可以使用二分法在 $O(logk)$ 的时间找到，并在 $O(k)$ 的时间完成删除， $O(1)$ 的时间完成插入。因此总的时间复杂度为 $n*k$。

关键点

- 滑动窗口 + 二分

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def medianSlidingWindow(self, A: List[int], k: int) ->
        ans = []
        win = []

        for i, a in enumerate(A):
            bisect.insort(win, a)
            if i >= k:
                win.pop(bisect.bisect_left(win, A[i - k]))
            if i >= k - 1:
                if k & 1:
                    median = win[k // 2]
                else:
                    median = (win[k // 2] + win[k // 2 - 1]
                ans.append(median)
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：数组插入的时间复杂度为 $O(k)$, 因此总的时间复杂度为 $O(n * k)$
- 空间复杂度：使用了大小为 k 的数组，因此空间复杂度为 $O(k)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（483. 最小好进制）

https://leetcode-cn.com/problems/smallest-good-base/

题目描述

```
对于给定的整数 n, 如果n的k（k>=2）进制数的所有数位全为1，则称 k（k>=

以字符串的形式给出 n, 以字符串的形式返回 n 的最小好进制。



示例 1:

输入："13"
输出："3"
解释：13 的 3 进制是 111。
示例 2:

输入："4681"
输出："8"
解释：4681 的 8 进制是 11111。
示例 3:

输入："1000000000000000000"
输出："999999999999999999"
解释：1000000000000000000 的 999999999999999999 进制是 11。


提示：

n的取值范围是 [3, 10^18]。
输入总是有效且没有前导 0。
```

前置知识

- 二分法
- 进制转换

公司

- 暂无

思路

题目虽然很短，题意也不难理解。但是要想做出来还是要费一番功夫的。

题目的意思是给你一个数字 n，让你返回一个进制 k，使得 n 的 k 进制表示为 1111...111，即一个全 1 的表示，并且**需要返回满足条件最小的 k**。

朴素的思路是一个个尝试。不过就算想要暴力求解也要一点**进制转换**的知识。这个知识点是： 一个数字 n 的 k 进制表示可以按照 $a *k^0* + *b* k^1 + c *k*^2 + ... + *m* k^{N-1}$ 的方式转化为十进制，其中 N 为 数字 n 的 k 进制位数。 比如十进制 3 的二进制是为 11，其位数就是 2。再比如十进制的 199，位数为 3。由于我们要求的是位全为 1 的数，因此系数全部为 1，也就是 a,b,c ...., m 全部为 1。

因此我们可从 k = 2 开始枚举，直到 n - 1，线性尝试是否可满足条件。

> 一进制只能有 0， 不可能有 1，故不考虑。 由于 n 进行最多 n 个数，上限是 n - 1，因此我们的枚举上限也是 n - 1。

核心伪代码：

```
n = int(n)
// 上面提到的 base 进制转十进制公式
func sum_with(base, N):
    return sum(1 * base ** i for i in range(N))

for k=2 to n - 1:
    if sum_with(k, N) == n: return k
```

可问题是 N 如何求出呢?

朴素的思路仍然是线性枚举。但是我们的搜索区间如何确定呢？我们知道对于一个数 n 来说，其 2 进制表示的长度一定是大于 3 进制表示的长度的。更一般而言，如果 k1 > k2，那么对于一个数字 n 的 k1 进制表示的位数一定小于 k2 进行表示的位数。 因此我们的解空间就是 [1,x] 其中 x 为 n 的二进制表示的位数。也就是说，我们可逐一枚举 N 的值 N`。

注意到需要返回的是最小的 k 进制，结合前面说的进制越小 N 越大的知识，我们应该使用从后往前倒着遍历，这样遇到满足条件的 k 可直接返回，因此在平均意义上时间复杂度更低。

```
n = int(n)
// 上面提到的 base 进制转十进制公式
func sum_with(base, N):
    return sum(1 * base ** i for i in range(N))
for N=x to 1:
    for k=2 to n - 1:
        if sum_with(k, N) == n: return k
```

注意这里的 x 到 1 的枚举没有必要线性枚举，而是可使用二分搜索的方式进行，其依据是**如果进制 k 的 N 位表示大于 n，那么 N`表示就不用看了，肯定都大，其中 N`是大于 N 的整数**。

让我们来计算下上面算法的时间复杂度。外层二分枚举的时间复杂度 $O(loglogn)$，内层枚举的时间复杂度是 n，sum_with 的时间复杂度为 logn，因此总的时间复杂度为 $n\times loglogn\times logn$。

到这里为止，算法勉强可以通过了。不过仍然有优化空间。注意到 sum_with 部分其实就是一个等比数列求和，因此我们可以使用等比数列求和公式，从而将 sum_with 的时间复杂度降低到 $O(1)$。

即使如此算法的时间复杂度也是 $O(n\times loglogn)$，代入到题目是的数据范围 $[3, 10^{18}]$，也是在超时边缘。使用数学法降维打击可以获得更好的效率，感兴趣的可以研究一下。

关键点解析

- 利用等比数列求和公式可降低时间复杂度
- 从进制转换入手发现单调性，从而使用二分解决

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def smallestGoodBase(self, n: str) -> str:
        n = int(n)
        # 上面提到的 base 进制转十进制公式。
        # 使用等比数列求和公式可简化时间复杂度
        def sum_with(base, N):
            return (1 - base ** N) // (1 - base)
            # return sum(1 * base ** i for i in range(N))
        # bin(n) 会计算出 n 的二进制表示， 其会返回形如 '0b10111
        for N in range(len(bin(n)) - 2, 0, -1):
            l = 2
            r = n - 1
            while l <= r:
                mid = (l + r) // 2
                v = sum_with(mid, N)

                if v < n:
                    l = mid + 1
                elif v > n:
                    r = mid - 1
                else:
                    return str(mid)
```

复杂度分析

- 时间复杂度：$O(n\times loglogn)$
- 空间复杂度：$O(1)$

题目地址（488. 祖玛游戏）

https://leetcode-cn.com/problems/zuma-game/

题目描述

```
回忆一下祖玛游戏。现在桌上有一串球，颜色有红色(R)，黄色(Y)，蓝色(B)，

每一次，你可以从手里的球选一个，然后把这个球插入到一串球中的某个位置上

找到插入并可以移除掉桌上所有球所需的最少的球数。如果不能移除桌上所有的

示例:
输入: "WRRBBW", "RB"
输出: -1
解释: WRRBBW -> WRR[R]BBW -> WBBW -> WBB[B]W -> WW （翻译者标

输入: "WWRRBBWW", "WRBRW"
输出: 2
解释: WWRRBBWW -> WWRR[R]BBWW -> WWBBWW -> WWBB[B]WW -> WWW

输入:"G", "GGGGG"
输出: 2
解释: G -> G[G] -> GG[G] -> empty

输入: "RBYYBBRRB", "YRBGB"
输出: 3
解释: RBYYBBRRB -> RBYY[Y]BBRRB -> RBBBRRB -> RRRB -> B ->
标注:

你可以假设桌上一开始的球中，不会有三个及三个以上颜色相同且连着的球。
桌上的球不会超过20个，输入的数据中代表这些球的字符串的名字是 "board"
你手中的球不会超过5个，输入的数据中代表这些球的字符串的名字是 "hand"。
输入的两个字符串均为非空字符串，且只包含字符 'R','Y','B','G','W'。
```

前置知识

- 回溯
- 哈希表
- 双指针

公司

- 百度

思路

面试题困难难度的题目常见的题型有：

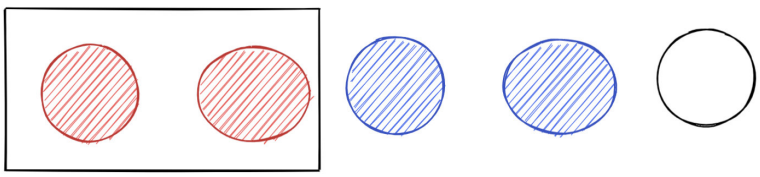
- DP
- 设计题
- 图
- 游戏

本题就是游戏类题目。 如果你是一个前端， 说不定还会考察你如何实现一个 zuma 游戏。这种游戏类的题目，可以简单可以困难， 比如力扣经典的石子游戏，宝石游戏等。这类题目没有固定的解法。我做这种题目的思路就是先暴力模拟，再尝试优化算法瓶颈。

注意下数据范围球的数目 <= 5，因此暴力法就变得可行。基本思路是暴力枚举手上的球可以消除的地方， 我们可以使用回溯法来完成暴力枚举的过程，在回溯过程记录最小值即可。由于回溯树的深度不会超过 5，因此这种解法应该可以 AC。

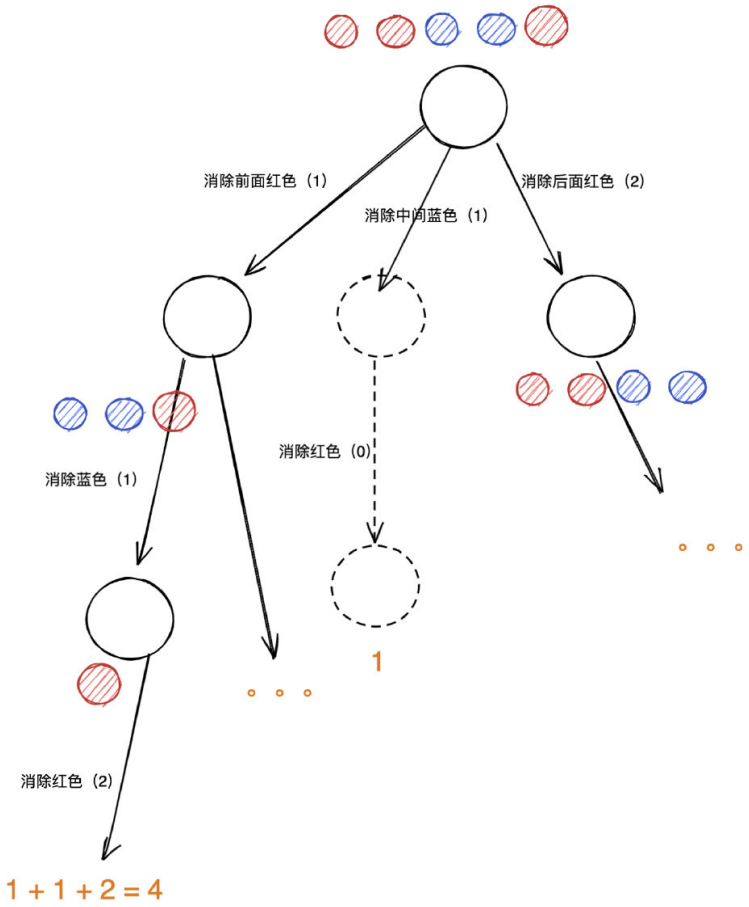
上面提到的 `可以消除的地方` ，指的是**连续相同颜色 + 手上相同颜色的球大于等于 3**，这也是题目说明的消除条件。

因此我们只需要两个指针记录连续相同颜色球的位置，如果可以消除，消除即可。



如图，我们记录了连续红球的位置， 如果手上有红球， 则可以尝试将其清除，这一次决策就是回溯树（决策树）的一个分支。之后我们会撤回到这个决策分支， 尝试其他可行的决策分支。

以 board = RRBBRR ， hand 为 RRBB 为例，其决策树为：



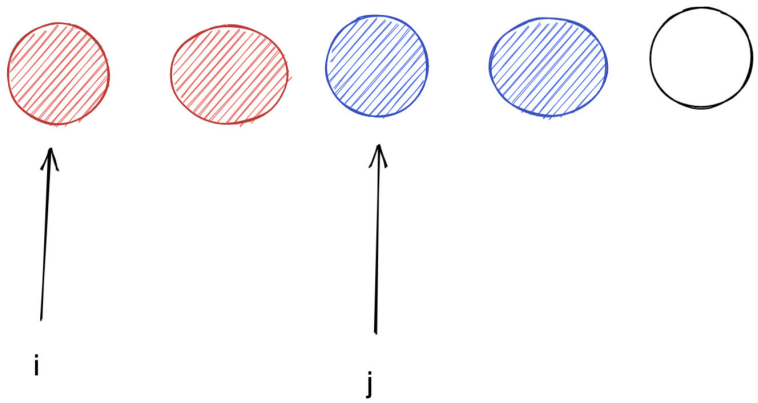
其中虚线表示无需手动干预，系统自动消除。叶子节点末尾的黄色表示全部消除需要的手球个数。路径上的文字后面的数字表示此次消除需要的手球个数

> 如果你对回溯不熟悉，可以参考下我之前写的几篇题解：比如 46.permutations。

可以看出， 如果选择先消除中间的蓝色，则只需要一步即可完成。

关于计算连续球位置的核心代码(Python3):

```
i = 0
while i < len(board):
    j = i + 1
    while j < len(board) and board[i] == board[j]: j += 1
    # 其他逻辑

    # 更新左指针
    i = j
```



具体算法：

1. 用哈希表存储手上的球的种类和个数，这么做是为了后面**快速判断连续的球是否可以被消除**。由于题目限制手上求不会超过 5，因此哈希表的最大容量就是 5，可以认为这是一个常数的空间。
2. 回溯。

   2.1 确认可以消除的位置，算法参考上面的代码。

   2.2 判断手上是否有足够相同颜色的球可以消除。

   2.3 回溯的过程记录全局最小值。

关键点解析

- 回溯模板
- 双指针写法

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def findMinStep(self, board: str, hand: str) -> int:
        def backtrack(board):
            if not board: return 0
            i = 0
            ans = 6
            while i < len(board):
                j = i + 1
                while j < len(board) and board[i] == board
                balls = 3 - (j - i)
                if counter[board[i]] >= balls:
                    balls = max(0, balls)
                    counter[board[i]] -= balls
                    ans = min(ans, balls + backtrack(board
                    counter[board[i]] += balls
                i = j
            return ans

        counter = collections.Counter(hand)
        ans = backtrack(board)
        return -1 if ans > 5 else ans
```

复杂度分析

- 时间复杂度：$O(2^(min(C, 5)))$，其中 C 为连续相同颜色球的次数，比如 WWRRRR， C 就是 2， WRBDD， C 就是 4。min(C, 5) 是因为题目限定了手上球的个数不大于 5。
- 空间复杂度：$O(min(C, 5) * Board)$，其中 C 为连续相同颜色球的次数，Board 为 Board 的长度。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 36K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（493. 翻转对）

https://leetcode-cn.com/problems/reverse-pairs/

题目描述

```
给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将

你需要返回给定数组中的重要翻转对的数量。

示例 1:

输入: [1,3,2,3,1]
输出: 2
示例 2:

输入: [2,4,3,5,1]
输出: 3
注意:

给定数组的长度不会超过50000。
输入数组中的所有数字都在32位整数的表示范围内。
```

前置知识

- 归并排序
- 逆序数
- 分治

公司

- 阿里
- 百度
- 字节

暴力法

思路

读完这道题你应该就能联想到逆序数才行。求解逆序数最简单的做法是使用双层循环暴力求解。我们仿照求解决逆序数的解法来解这道题（其实唯一的区别就是系数从 1 变成了 2）。

代码

代码支持：Python3

Python3 Code:

```
class Solution(object):
    def reversePairs(self, nums):
        n = len(nums)
        cnt = 0
        for i in range(n):
            for j in range(i + 1, n):
                if nums[i] > 2 * nums[j]:
                    cnt += 1
        return cnt
```

分治法

思路

如果你能够想到逆序数，那么你很可能直到使用类似归并排序的方法可以求解逆序数。实际上逆序数只是归并排序的副产物而已。

我们在正常的归并排序的代码中去计算逆序数即可。由于每次分治的过程，左右两段数组分别是有序的，因此我们可以减少一些运算。 从时间复杂度的角度上讲，我们从$O(N^2)$优化到了 $O(NlogN)$。

具体来说，对两段有序的数组，有序数组内部是不需要计算逆序数的。我们计算逆序数的逻辑只是计算两个数组之间的逆序数，我们假设两个数组是 A 和 B，并且 A 数组最大的元素不大于 B 数组最小的元素。而要做到这样，我们只需要常规的归并排序即可。

接下来问题转化为求解两个有序数组之间的逆序数，并且两个有序数组之间满足关系 `A数组最大的元素不大于B数组最小的元素` 。

关于计算逆序数的核心代码(Python3):

```
l = r = 0
while l < len(left) and r < len(right):
    if left[l] <= 2 * right[r]:
        l += 1
    else:
        self.cnt += len(left) - l
        r += 1
```

代码

代码支持：Python3

Python3 Code:

```
class Solution(object):
    def reversePairs(self, nums):
        self.cnt = 0

        def mergeSort(lst):
            L = len(lst)
            if L <= 1:
                return lst
            return mergeTwo(mergeSort(lst[:L//2]), mergeSo

        def mergeTwo(left, right):
            l = r = 0
            while l < len(left) and r < len(right):
                if left[l] <= 2 * right[r]:
                    l += 1
                else:
                    self.cnt += len(left) - l
                    r += 1
            return sorted(left+right)

        mergeSort(nums)
        return self.cnt
```

复杂度分析

- 时间复杂度：$O(NlogN)$
- 空间复杂度：$O(logN)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

对于具体的排序过程我们偷懒直接使用了语言内置的方法 sorted，这在很多时候是有用的，即使你是参加面试，这种方式通常也是允许的。省略非核心的考点，可以使得问题更加聚焦，这是一种解决问题的思路，在工作中也很有用。

关键点解析

- 归并排序
- 逆序数
- 分治
- 识别考点，其他非重点可以使用语言内置方法

扩展

这道题还有很多别的解法，感性的可以参考下这个题解 General principles behind problems similar to "Reverse Pairs"

题目地址(679. 24 点游戏)

https://leetcode-cn.com/problems/24-game/

题目描述

```
你有 4 张写有 1 到 9 数字的牌。你需要判断是否能通过 *，/，+，-，(，

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:

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
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，[
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 +
```

前置知识

- 回溯
- 数字精度问题
- 分治

公司

- 暂无

思路

题目给了我们四个数字，让我们通过 + - $\times$ $\div$ 将其组成 24。由于题目数据范围只可能是 4，因此使用暴力回溯所有可能性是一个可行的解。

我们先使用回溯找出 nums 的全部全排列，这一步需要枚举 $4 \times 3 \times 2 \times 1$ 种可能。由于四种运算都是双目运算（题目明确指出 - 不能当成负号），因此我们可以任意选出两个，继续枚举四种运算符。由于选出哪个对结果没有影响，因此不妨我们就选前两个。接下来，将前两个的运算结果和后面的数继续**使用同样的方法来解决**。 这样问题规模便从 4 缩小到了 3，这样不断进行下去直到得到一个数字为止。如果剩下的这唯一的数字是 24，那么我们就返回 true，否则返回 false。

值得注意的是，实数除存在精度误差。因此我们需要判断一下最后的结果离 24 不超过某个精度范围即可，比如如果结果和 24 误差不超过 10^{-6}，我们就认为是 24，返回 true 即可。

关键点

- 使用递归将问题分解成规模更小的同样问题
- 精度控制，即如果误差不超过某一个较小的数字就认为二者是相等的

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def judgePoint24(self, nums: List[int]) -> bool:
        if len(nums) == 1:
            return math.isclose(nums[0], 24)
        return any(self.judgePoint24([x] + rest) for a, b,
for x in [a+b, a-b, a*b, b and a/b])
```

复杂度分析

由于题目输入大小恒为 4 ，因此实际上我们算法复杂度也是一个定值。

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(715. Range 模块)

https://leetcode-cn.com/problems/range-module/

题目描述

```
Range 模块是跟踪数字范围的模块。你的任务是以一种有效的方式设计和实现以

addRange(int left, int right) 添加半开区间 [left, right)，跟踪
queryRange(int left, int right) 只有在当前正在跟踪区间 [left,
removeRange(int left, int right) 停止跟踪区间 [left, right) 中

示例：

addRange(10, 20): null
removeRange(14, 16): null
queryRange(10, 14): true （区间 [10, 14) 中的每个数都正在被跟踪
queryRange(13, 15): false （未跟踪区间 [13, 15) 中像 14, 14.0
queryRange(16, 17): true （尽管执行了删除操作，区间 [16, 17) 中

提示：

半开区间 [left, right) 表示所有满足 left <= x < right 的实数。
对 addRange, queryRange, removeRange 的所有调用中 0 < left <
在单个测试用例中，对 addRange 的调用总数不超过 1000 次。
在单个测试用例中，对  queryRange 的调用总数不超过 5000 次。
在单个测试用例中，对 removeRange 的调用总数不超过 1000 次。
```

前置知识

- 区间查找问题
- 二分查找

公司

- 暂无

思路

直观的思路是使用端点记录已经被跟踪的区间，我们需要记录的区间信息大概是这样的：[(1,2),(3,6),(8,12)]，这表示 [1,2), [3,6), [8,12) 被跟踪。

添加区间需要先查一下会不会和已有的区间和交集，如果有则融合。删除区间也是类似。关于判断是否有交集以及融合都可以采用一次遍历的方式来解决，优点是简单直接。

区间查询的话，由于被跟踪的区间是有序且不重叠的（重叠的会被我们合并），因此可是使用二分查找来加速。

官方给的解法其实就是这种。

代码：

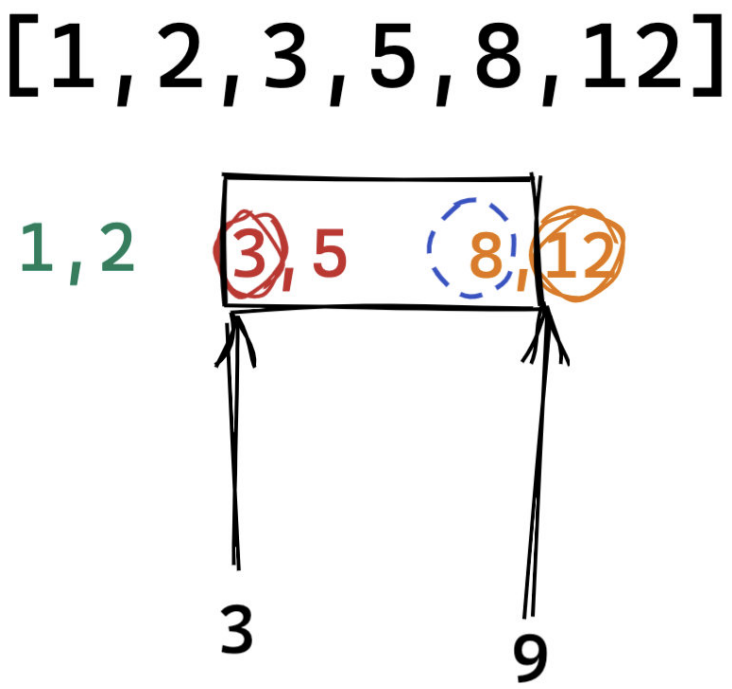
```
class RangeModule(object):
    def __init__(self):
        # [(1,2),(3,6),(8,12)]
        self.ranges = []
    def overlap(self, left, right):
        i, j = 0, len(self.ranges) - 1
        while i < len(self.ranges) and self.ranges[i][1] <
            i += 1
        while j >= 0 and self.ranges[j][0] > right:
            j -= 1
        return i, j

    def addRange(self, left, right):
        i, j = self.overlap(left, right)
        if i <= j:
            left = min(left, self.ranges[i][0])
            right = max(right, self.ranges[j][1])
        self.ranges[i:j+1] = [(left, right)]
    def queryRange(self, left, right):
        i = bisect.bisect_right(self.ranges, (left, float(
        return self.ranges and self.ranges[i][0] <= left ar

    def removeRange(self, left, right):
        i, j = self.overlap(left, right)
        merge = []
        for k in xrange(i, j+1):
            if self.ranges[k][0] < left:
                merge.append((self.ranges[k][0], left))
            if right < self.ranges[k][1]:
                merge.append((right, self.ranges[k][1]))
        self.ranges[i:j+1] = merge
```

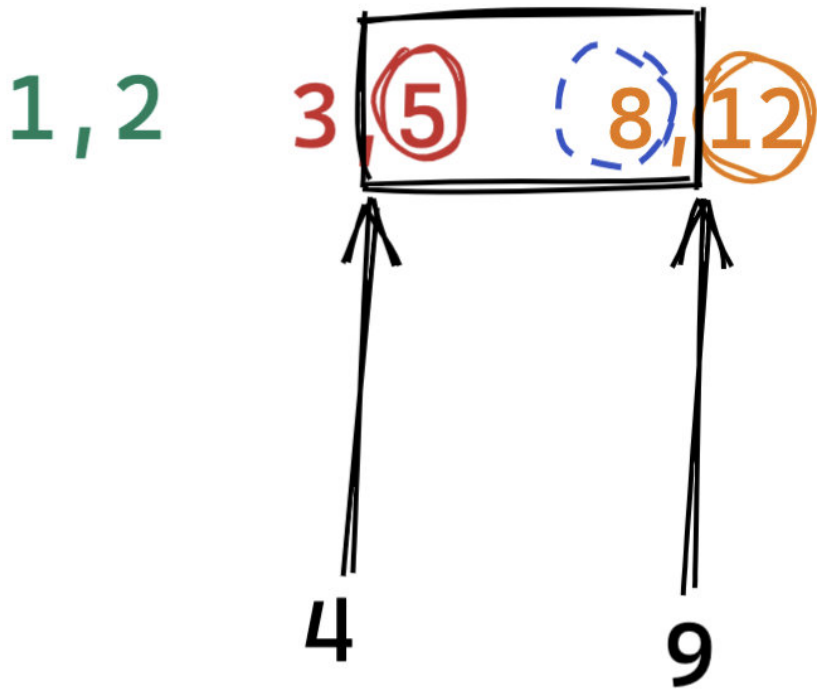
但其实这种做法 overlap 的时间复杂度是 $O(N)$，这部分可以优化。优化点点在于 overlap 的实现，实际上被跟踪的区间是有序的，因此这部分其实也可是二分查找。只不过我写了一半就发现不好根据结束时间查找。

参考了 这篇题解 后发现，其实我们可以将被跟踪的区块一维化处理，这样问题就简单了。比如我们不这样记录被跟踪的区间 [(1,2),(3,6),(8,12)]，而是这样：[1,2,3,5,8,12]。

经过这样的处理， 数组的奇数坐标就是区间的结束点，偶数坐标就是开始点啦。这样二分就不需要像上面一样使用元组，而是使用单值了。

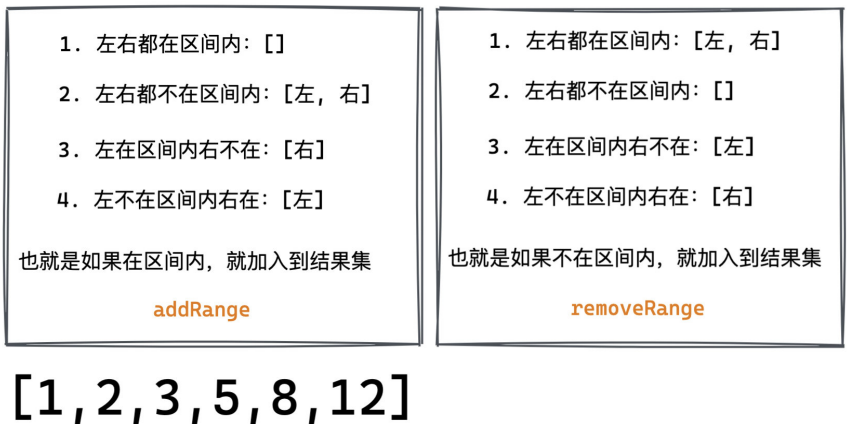
- 如何查询某一个区间 [s, e] 是否被跟踪呢？我们只需要将 s， e 分别在数组中查一下。如果 s 和 e 都是**同一个奇数坐标**即可。
- 插入和删除也是一样。先将 s， e 分别在数组中查一下，假设我们查到的分别为 i 和 j，接下来使用 [i, j] 更新原有区间即可。





使用不同颜色区分不同的区间，当我们要查 [3,9] 的时候。实线圈表示我们查到的索引，黑色的框框表示我们需要更新的区间。

区间更新逻辑如下：



[1,2,3,5,8,12]

关键点解析

- 二分查找的灵活使用（最左插入和最右插入）
- 将区间一维化处理

代码

为了明白 Python 代码的含义，你需要明白 bisect_left 和 bisect_right，关于这两点我在二分查找专题讲地很清楚了，大家可以看一下。实际上这两者的区别只在于目标数组有目标值的情况，因此如果你搞不懂，可以尝试代入这种特殊情况理解。

代码支持：Python3

Python3 Code:

```
class RangeModule(object):
    def __init__(self):
        # [1,2,3,5,8,12]
        self.ranges = []

    def overlap(self, left, right, is_odd):
        i = bisect_left(self.ranges, left)
        j = bisect_right(self.ranges, right)
        merge = []
        if i & 1 == int(is_odd):
            merge.append(left)
        if j & 1 == int(is_odd):
            merge.append(right)
        # 修改 ranges 的 [i:j-1] 部分
        self.ranges[i:j] = merge

    def addRange(self, left, right):
        # [1,2,3,5,8,12]，代入 left = 3, right = 5，此时需要
        return self.overlap(left, right, False)

    def removeRange(self, left, right):
        # [1,2,3,5,8,12]，代入 left = 3, right = 5，此时需要
        return self.overlap(left, right, True)

    def queryRange(self, left, right):
        # [1,2,3,5,8,12]，代入 left = 3, right = 5，此时需要
        i = bisect_right(self.ranges, left)
        j = bisect_left(self.ranges, right)
        return i & 1 == 1 and i == j  # 都在一个区间内
```

addRange 和 removeRange 中使用 bisect_left 找到左端点 l，使用 bisect_right 找到右端点，这样将 [left, right) 更新到区间 [l, r - 1] 即可。

复杂度分析

- 时间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。
- 空间复杂度：$O(m * n)$，其中 m 和 n 分别为 A 和 B 的 长度。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(768. 最多能完成排序的块 II)

https://leetcode-cn.com/problems/max-chunks-to-make-sorted-ii/

题目描述

```
这个问题和“最多能完成排序的块”相似，但给定数组中的元素可以重复，输入数

arr是一个可能包含重复元素的整数数组，我们将这个数组分割成几个“块”，并

我们最多能将数组分成多少块?

示例 1:

输入: arr = [5,4,3,2,1]
输出: 1
解释:
将数组分成2块或者更多块，都无法得到所需的结果。
例如，分成 [5, 4], [3, 2, 1] 的结果是 [4, 5, 1, 2, 3]，这不是有
示例 2:

输入: arr = [2,1,3,4,4]
输出: 4
解释:
我们可以把它分成两块，例如 [2, 1], [3, 4, 4]。
然而，分成 [2, 1], [3], [4], [4] 可以得到最多的块数。
注意:

arr的长度在[1, 2000]之间。
arr[i]的大小在[0, 10**8]之间。
```

前置知识

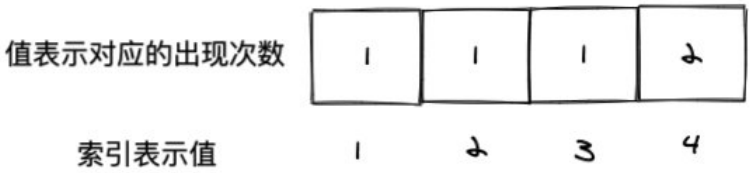
- 栈
- 队列

计数

思路

这里可以使用类似计数排序的技巧来完成。以题目给的 [2,1,3,4,4] 来说：

可以先计数，比如用一个数组来计数，其中数组的索引表示值，数组的值表示其对应的出现次数。比如上面，除了 4 出现了两次，其他均出现一次，因此 count 就是 [0,1,1,1,2]。



其中 counts[4] 就是 2，表示的就是 4 这个值出现了两次。

> 实际上 count 最开始的 0 是没有必要的，不过这样方便理解罢了。

如果我们使用数组来计数，那么空间复杂度就是 $upper - lower$，其中 upper 是 arr 的最大值， lower 是 arr 的最小值。

计数完毕之后， 我们要做的是比较当前的 arr 和最终的 arr（已经有序的 arr） 的计数数组的关系即可。

这里有一个关键点： **如果两个数组的计数信息是一致的，那么两个数组排序后的结果也是一致的。** 如果你理解计数排序，应该明白我的意思。不明白也没有关系， 我稍微解释一下你就懂了。

如果我把一个数组打乱，然后排序，得到的数组一定是确定的，即不管你怎么打乱排好序都是一个确定的有序序列。这个论点的正确性是毋庸置疑的。而实际上，一个数组无论怎么打乱，其计数结果也是确定的，这也是毋庸置疑的。反之，如果是两个不同的数组，打乱排序后的结果一定是不同的，计数也是同理。

（这两个数组排序后的结果以及计数信息是一致的）

因此我们的算法有了：

- 先排序 arr,不妨记排序后的 arr 为 sorted_arr
- 从左到右遍历 arr，比如遍历到了索引为 i 的元素，其中 0 <= i < len(arr)
- 如果 arr[:i+1] 的计数信息和 sorted_arr[:i+1] 的计数信息一致，那么说明可以分桶，否则不可以。

> arr[:i+1] 指的是 arr 的切片，从索引 0 到 索引 i 的一个切片。

关键点

- 计数

代码

- 语言支持：Python

Python Code:

```python
class Solution(object):
    def maxChunksToSorted(self, arr):
        count_a = collections.defaultdict(int)
        count_b = collections.defaultdict(int)
        ans = 0

        for a, b in zip(arr, sorted(arr)):
            count_a[a] += 1
            count_b[b] += 1
            if count_a == count_b: ans += 1

        return ans
```

复杂度分析

- 时间复杂度：内部 count_a 和 count_b 的比较时间复杂度也是 $O(N)$，因此总的时间复杂度为 $O(N^2)$，其中 N 为数组长度。
- 空间复杂度：使用了两个 counter，其大小都是 N，因此空间复杂度为 $O(N)$，其中 N 为数组长度。

优化的计数

思路

实际上，我们不需要两个 counter ，而是使用一个 counter 来记录 arr 和 sorted_arr 的 diff 即可。但是这也仅仅是空间上的一个常数优化而已。

我们还可以在时间上进一步优化， 去除内部 count_a 和 count_b 的比较，这样算法的瓶颈就是排序了。而去除的关键点就是我们上面提到的**记录 diff**，具体参考下方代码。

关键点

- 计数
- count 的边界条件

代码

- 语言支持：Python

Python Code:

```python
class Solution(object):
    class Solution(object):
    def maxChunksToSorted(self, arr):
        count = collections.defaultdict(int)
        non_zero_cnt = 0
        ans = 0

        for a, b in zip(arr, sorted(arr)):
            if count[a] == -1: non_zero_cnt -= 1
            if count[a] == 0: non_zero_cnt += 1
            count[a] += 1
            if count[b] == 1: non_zero_cnt -= 1
            if count[b] == 0: non_zero_cnt += 1
            count[b] -= 1
            if non_zero_cnt == 0: ans += 1

        return ans
```

复杂度分析

- 时间复杂度：瓶颈在于排序，因此时间复杂度为 $O(NlogN)$，其中 N 为数组长度。
- 空间复杂度：使用了一个 counter，其大小是 N，因此空间复杂度为 $O(N)$，其中 N 为数组长度。

单调栈

思路

通过题目给的三个例子，应该可以发现一些端倪。

- 如果 arr 是非递减的，那么答案为 1。
- 如果 arr 是非递增的，那么答案是 arr 的长度。

并且由于**只有分的块内部可以排序**，块与块之间的相对位置是不能变的。因此**直观上**我们的核心其实找到从左到右开始不减少（增加或者不变）的地方并分块。

比如对于 [5,4,3,2,1] 来说：

- 5 的下一个是 4，比 5 小，因此如果分块，那么永远不能变成 [1,2,3,4,5]。
- 同理，4 的下一个是 3，比 4 小，因此如果分块，那么永远不能变成 [1,2,3,4,5]。
- 。。。

最后就是不能只能是整体是一个大块，我们返回 1 即可。

我们继续分析一个稍微复杂一点的，即题目给的 [2,1,3,4,4]。

- 2 的下一个是 1，比 2 小，不能分块。
- 1 的下一个是 3，比 1 大，可以分块。
- 3 的下一个是 4，比 3 大，可以分块。
- 4 的下一个是 4，一样大，可以分块。

因此答案就是 4，分别是：

- [2,1]
- [3]
- [3]
- [4]

然而上面的算法步骤是不正确的，原因在于只考虑局部，没有考虑整体，比如 **[4,2,2,1,1]** 这样的测试用例，实际上只应该返回 1，原因是后面碰得到了 1，使得前面不应该分块。

因为把数组分成数个块，分别排序每个块后，组合所有的块就跟整个数组排序的结果一样，这就意味着后面块中的最小值一定大于前面块的最大值,这样才能保证分块有。因此直观上，我们又会觉得是不是”只要后面有较小值，那么前面大于它的都应该在一个块里面“，实际上的确如此。

有没有注意到我们一直在找下一个比当前小的元素？这就是一个信号，使用单调递增栈即可以空间换时间的方式解决。对单调栈不熟悉的小伙伴可以看下我的单调栈专题

不过这还不够，我们要把思路逆转！

> 这是《逆转裁判》 中经典的台词， 主角在深处绝境的时候，会突然冒出这句话，从而逆转思维，寻求突破口。

这里的话，我们将思路逆转，不是分割区块，而是**融合区块**。

比如 [2,1,3,4,4]，遍历到 1 的时候会发现 1 比 2 小，因此 2， 1 需要在一块，我们可以将 2 和 1 融合，并**重新压回栈**。那么融合成 1 还是 2 呢？答案是 2，因为 2 是瓶颈，这提示我们可以用一个递增栈来完成。

因此本质上**栈存储的每一个元素就代表一个块，而栈里面的每一个元素的值就是块的最大值**。

以 [2,1,3,4,4] 来说， stack 的变化过程大概是：

- [2]
- 1 被融合了，保持 [2] 不变
- [2,3]
- [2,3,4]
- [2,3,4,4]

简单来说，就是**将一个减序列压缩合并成最该序列的最大的值**。 因此最终返回 stack 的长度就可以了。

具体算法参考代码区，注释很详细。

代码

- 语言支持：Python，CPP，Java，JS, Go, PHP

Python Code:

```python
class Solution:
    def maxChunksToSorted(self, A: [int]) -> int:
        stack = []
        for a in A:
            # 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
            # 而栈中每一个元素都是一个块，并且栈的存的是块的最大值，
            if stack and stack[-1] > a:
                # 我们需要将融合后的区块的最大值重新放回栈
                # 而 stack 是递增的，因此 stack[-1] 是最大的
                cur = stack[-1]
                # 维持栈的单调递增
                while stack and stack[-1] > a: stack.pop()
                stack.append(cur)
            else:
                stack.append(a)
        # 栈存的是块信息，因此栈的大小就是块的数量
        return len(stack)
```

CPP Code:

```
class Solution {
public:
    int maxChunksToSorted(vector<int>& arr) {
        stack<int> stack;
        for(int i =0;i<arr.size();i++){
            // 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
            // 而栈中每一个元素都是一个块，并且栈的存的是块的最大值
            if(!stack.empty()&&stack.top()>arr[i]){
                // 我们需要将融合后的区块的最大值重新放回栈
                // 而 stack 是递增的，因此 stack[-1] 是最大的
                int cur = stack.top();
                // 维持栈的单调递增
                while(!stack.empty()&&stack.top()>arr[i]){
                    sstackta.pop();
                }

                stack.push(cur);
            }else{

                stack.push(arr[i]);
            }
        }
        // 栈存的是块信息，因此栈的大小就是块的数量
        return stack.size();
    }
};
```

JAVA Code:

```
class Solution {
    public int maxChunksToSorted(int[] arr) {
        LinkedList<Integer> stack = new LinkedList<Integer>
        for (int num : arr) {
            // 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
            // 而栈中每一个元素都是一个块，并且栈的存的是块的最大值
            if (!stack.isEmpty() && num < stack.getLast())
                // 我们需要将融合后的区块的最大值重新放回栈
                // 而 stack 是递增的，因此 stack[-1] 是最大的
                int cur = stack.removeLast();
                // 维持栈的单调递增
                while (!stack.isEmpty() && num < stack.getL
                    stack.removeLast();
                }
                stack.addLast(cur);
            } else {
                stack.addLast(num);
            }
        }
        // 栈存的是块信息，因此栈的大小就是块的数量
        return stack.size();
    }
}
```

JS Code:

```
var maxChunksToSorted = function (arr) {
  const stack = [];

  for (let i = 0; i < arr.length; i++) {
    a = arr[i];
    if (stack.length > 0 && stack[stack.length - 1] > a) {
      const cur = stack[stack.length - 1];
      while (stack && stack[stack.length - 1] > a) stack.po
      stack.push(cur);
    } else {
      stack.push(a);
    }
  }
  return stack.length;
};
```

Go Code:

```go
func maxChunksToSorted(arr []int) int {
    var stack []int // 单调递增栈，stack[-1] 栈顶
    for _, a := range arr {
        // 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
        // 而栈中每一个元素都是一个块，并且栈的存的是块的最大值，因此
        if len(stack) > 0 && stack[len(stack)-1] > a {
            // 我们需要将融合后的区块的最大值重新放回栈
            // 而 stack 是递增的，因此 stack[-1] 是最大的
            cur := stack[len(stack)-1]
            // 维持栈的单调递增
            for len(stack) > 0 && stack[len(stack)-1] > a {
                stack = stack[:len(stack)-1] // pop
            }
            stack = append(stack, cur) // push
        } else {
            stack = append(stack, a) // push
        }
    }
    // 栈存的是块信息，因此栈的大小就是块的数量
    return len(stack)
}
```

PHP Code:

```php
class Solution
{

    /**
     * @param Integer[] $arr
     * @return Integer
     */
    function maxChunksToSorted($arr)
    {
        $stack = []; // 单调递增栈，stack[-1] 栈顶
        foreach ($arr as $a) {
            // 遇到一个比栈顶小的元素，而前面的块不应该有比 a 小的
            // 而栈中每一个元素都是一个块，并且栈的存的是块的最大值
            if ($stack && $stack[count($stack) - 1] > $a) {
                $cur = $stack[count($stack) - 1];
                // 维持栈的单调递增
                while ($stack && $stack[count($stack) - 1]
                array_push($stack, $cur);
            } else array_push($stack, $a);
        }
        // 栈存的是块信息，因此栈的大小就是块的数量
        return count($stack);
    }
}
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(N)$，其中 N 为数组长度。

总结

实际上本题的单调栈思路和 【力扣加加】从排序到线性扫描(57. 插入区间) 以及 394. 字符串解码 都有部分相似，大家可以结合起来理解。

融合与 【力扣加加】从排序到线性扫描(57. 插入区间) 相似， 重新压栈和 394. 字符串解码 相似。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址(805. 数组的均值分割)

https://leetcode-cn.com/problems/split-array-with-same-average/

题目描述

```
给定的整数数组 A ，我们要将 A数组 中的每个元素移动到 B数组 或者 C数组

返回true ，当且仅当在我们的完成这样的移动后，可使得B数组的平均值和C数

示例:
输入:
[1,2,3,4,5,6,7,8]
输出: true
解释: 我们可以将数组分割为 [1,4,5,8] 和 [2,3,6,7], 他们的平均值都

注意:

A 数组的长度范围为 [1, 30].
A[i] 的数据范围为 [0, 10000].
```

前置知识

- 回溯

公司

- 暂无

思路

实际上分出的**两个列表 B 和 C 的均值都等于列表 A 的均值**，这是本题的入手点。以下是证明：

```
sum(B) * (N - K) = sum(C) * K
sum(B) * N = (sum(B) + sum(C)) * K
sum(B) / K = (sum(B) + sum(C)) / N
sum(B) / K = sum(A) / N
```

因此我们可以枚举所有的 A 的大小 i，相应地 B 的大小就是 n - i，其中 n 为数组 A 的大小。而由于**两个列表 B 和 C 的均值都等于列表 A 的均值**。因此可以提前计算出 A 的均值 avg，那么 A 的总和其实就是 i *avg*，*我们使用回溯找到一个和为 i* avg 的组合，即可返回 true，否则返回 false。

核心代码：

```
def splitArraySameAverage(self, A: List[int]) -> bool:
        n = len(A)
        avg = sum(A) / n

        for i in range(1, n // 2 + 1):
            for combination in combinations(A, i):
                if abs(sum(combination) - avg * i) < 1e-6:
                    return True
        return False
```

上面代码由于回溯里面嵌套了 sum，因此时间复杂度为**回溯的时间复杂度 * sum 的时间复杂度**，因此总的时间复杂度在最坏的情况下是 $n * 2^n$。

上面的代码思路上可行，但却有很多可以优化的地方。至少我们可以不用计算出来所有的组合之后再求和，而是直接计算**所有的和**的组合，这种算法的时间复杂度为 2^n。

核心代码：

```
def splitArraySameAverage(self, A: List[int]) -> bool:
        n = len(A)
        avg = sum(A) / n

        for i in range(1, n // 2 + 1):
            for s in combinationSum(A, i):
                if abs(s - avg * i) < 1e-6:
                    return True
        return False
```

但是遗憾的是，这仍然不足以通过所有的测试用例。接下来，我们可以通过剪枝的手段来达到 AC 的目的。 很多**回溯**的题目都是基于剪枝来完成的。剪枝是回溯问题的核心考点。

对于这道题来说，我们可以剪枝的点有两个：

- 剪枝一：对于一个数组 [1,1,3]，任选其中两项，其组合有 3 种。分别是 (1,1), (1,3) 和 (1,3)。实际上，我们可以将两个 (1,3) 看成一样的（部分题目不能看成一样的，但本题可以），如果能将生成同样的组

合剪枝掉就好了。我们可以排序的方式进行剪枝，具体参考 40. 组合总和 II

- 剪枝二：由于每个数字都是整数，那么其和一定也是整数，因此如果和是小数，那么其一定不可能，可以剪枝。

关键点

- 回溯解题模板
- 两个剪枝

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def combinationSum(self, candidates: List[int], count:
        size = len(candidates)
        if size == 0:
            return []

        # 还是先排序，主要是方便去重
        candidates.sort()

        ans = []
        self._find_path(candidates, ans, 0, count, 0, size)
        return ans

    def _find_path(self, candidates, ans, path_sum, count,
        if count == 0:
            ans.append(path_sum)
            return
        else:
            for i in range(begin, size):
                # 剪枝一。 注意这里的 i > begin 这个条件
                if i > begin and candidates[i] == candidate
                    continue
                self._find_path(candidates, ans, path_sum -

    def splitArraySameAverage(self, A: List[int]) -> bool:
        n = len(A)
        avg = sum(A) / n

        for i in range(1, n // 2 + 1):
            # 剪枝二
            if abs(i * avg - int(i * avg)) > 1e-6:
                continue
            for s in self.combinationSum(A, i):
                if abs(s - avg * i) < 1e-6:
                    return True
        return False
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(2^n)$
- 空间复杂度：$O(n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(839. 相似字符串组)

https://leetcode-cn.com/problems/similar-string-groups/

题目描述

```
如果交换字符串 X 中的两个不同位置的字母，使得它和字符串 Y 相等，那么称

例如，"tars" 和 "rats" 是相似的 (交换 0 与 2 的位置)； "rats" 和

总之，它们通过相似性形成了两个关联组：{"tars", "rats", "arts"} 和

给你一个字符串列表 strs。列表中的每个字符串都是 strs 中其它所有字符串
```

```
示例 1：

输入：strs = ["tars","rats","arts","star"]
输出：2
```

```
示例 2：

输入：strs = ["omv","ovm"]
输出：1
```

```
提示：

1 <= strs.length <= 100
1 <= strs[i].length <= 1000
sum(strs[i].length) <= 2 * 10^4
strs[i] 只包含小写字母。
strs 中的所有单词都具有相同的长度，且是彼此的字母异位词。
```

```
备注：

      字母异位词（anagram），一种把某个字符串的字母的位置（顺序）加以
```

前置知识

- 并查集

公司

- 暂无

思路

将字符串看成图中的点，字符串的相似关系看成边， 即如果两个字符串 s1, s2 相似就在两个字符串之间添加一条无向边(s1, s2)。

相似关系其实是**没有**联通性的。比如 s1 和 s2 相似，s2 和 s3 相似，那么 s1 和 s3 **不一定**相似，但是 s1 ，s2，s3 应该在一个**相似字符串数组**中。而题目仅要求我们返回相似字符串数组的个数。 而**在同一个相似字符串数组中的字符串却具有联通性**。这提示我们使用并查集维护字符串（图中的点）的联通关系。直接套一个标准的不带权并查集模板就好了，我将**标准不带权并查集封装成了一个 API 调用**，这样遇到其他需要用并查集的题目也可直接使用。

在调用并查集模板之前，我们需要知道图中点的个数，那自然就是字符串的总数了。接下来，我们将字符串两两合并，这需要 $O(N^2)$ 的时间复杂度, 其中 n 为字符串总数。核心代码：

```
uf = UF(n)
for i in range(n):
      for j in range(i + 1, n):
           if strs[i] == strs[j] or is_similar(list(strs[i]
              uf.union(i, j)
return uf.cnt
```

uf.cnt 为图中的联通分量的个数，正好对应题目的返回。

接下来，我们需要实现 is_similar 函数。 朴素的思路是遍历所有可能，即交换其中一个字符串（不妨称其为 s1）的任意两个字符。每次交换后都判断是否和另外一个字符串相等（不妨称其为 s2），代码表示其实 s1 == s2。由于每次判断两个字符相等的复杂度是线性的，因此这种算法 is_similar 的时间复杂度为 $O(m^3)$，其中 m 为字符串长度。这种算法会超时。

核心代码：

```
def is_similar(A, B):
    n = len(A)
    for i in range(n):
        for j in range(i + 1, n):
            A[i], A[j] = A[j], A[i]
            if A == B: return True
            A[i], A[j] = A[j], A[i]
    return False
```

实际上，我们只需要统计两个字符串不同字符的个数即可。这个不同字符指的是相同索引的字符不同。如果不同字符的个数等于 2 （或者 0）那么就可以认为两个字符串是相似的。

关键点

- 判断两个字符串是否相似的函数 is_similar 没有必须真实交换并判断，而是判断不相等字符是否等于 2

代码

- 语言支持：Python3

Python3 Code:

```
class UF:
  def __init__(self, M):
      self.parent = {}
      self.cnt = 0
      # 初始化 parent, size 和 cnt
      for i in range(M):
          self.parent[i] = i
          self.cnt += 1

  def find(self, x):
      if x != self.parent[x]:
          self.parent[x] = self.find(self.parent[x])
          return self.parent[x]
      return x
  def union(self, p, q):
      if self.connected(p, q): return
      leader_p = self.find(p)
      leader_q = self.find(q)
      self.parent[leader_p] = leader_q
      self.cnt -= 1
  def connected(self, p, q):
      return self.find(p) == self.find(q)

class Solution:
    def numSimilarGroups(self, strs: List[str]) -> int:
        n = len(strs)
        uf = UF(n)
        def is_similar(A, B):
            n = len(A)
            diff = 0
            for i in range(n):
                if A[i] != B[i]: diff += 1
            return diff == 2

        for i in range(n):
            for j in range(i + 1, n):
                if strs[i] == strs[j] or is_similar(list(st
                    uf.union(i, j)
        return uf.cnt
```

复杂度分析

令 n 为字符串总数，m 为字符串的平均长度。

- 时间复杂度：$O(n^2\times m)$
- 空间复杂度：$O(n)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

题目地址(887. 鸡蛋掉落)

原题地址：https://leetcode-cn.com/problems/super-egg-drop/

题目描述

```
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

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

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

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 <= K <= 100
1 <= N <= 10000
```

前置知识

- 递归

- 动态规划

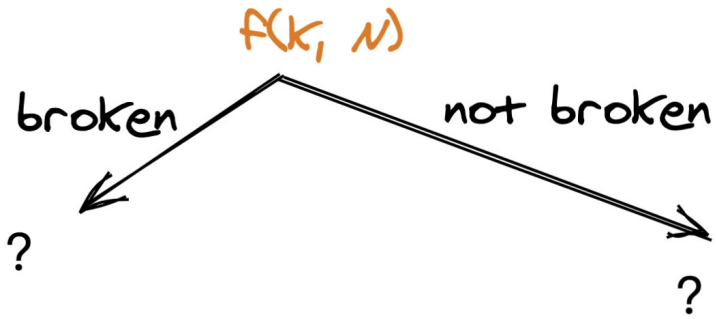
思路

本题也是 vivo 2020 年提前批的一个笔试题。时间一个小时，一共三道题，分别是本题，合并 k 个链表，以及种花问题。

这道题我在很早的时候做过，也写了题解。现在看来，思路没有讲清楚。没有讲当时的思考过程还原出来，导致大家看的不太明白。今天给大家带来的是 887.super-egg-drop 题解的**重制版**。思路更清晰，讲解更透彻，如果觉得有用，那就转发在看支持一下？OK，我们来看下这道题吧。

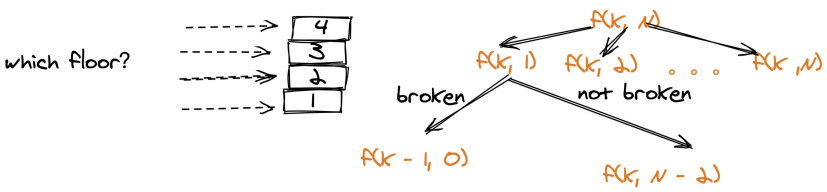
这道题乍一看很复杂，我们不妨从几个简单的例子入手，尝试打开思路。

假如有 2 个鸡蛋，6 层楼。 我们应该先从哪层楼开始扔呢？想了一会，没有什么好的办法。我们来考虑使用暴力的手段。



（图 1. 这种思路是不对的）

既然我不知道先从哪层楼开始扔是最优的，那我就依次模拟从第 1，第 2。。。第 6 层扔。每一层楼丢鸡蛋，都有两种可能，碎或者不碎。由于是最坏的情况，因此我们需要模拟两种情况，并取两种情况中的扔次数的较大值（较大值就是最坏情况）。 然后我们从六种扔法中选择最少次数的即可。



（图 2. 应该是这样的）

而每一次选择从第几层楼扔之后，剩下的问题似乎是一个规模变小的同样问题。嗯哼？递归？

为了方便描述，我将 f(i, j) 表示有 i 个鸡蛋， j 层楼，在最坏情况下，最少的次数。

伪代码：

```
def superEggDrop(K, N):
    ans = N
    # 暴力枚举从第 i 层开始扔
    for i in range(1, N + 1):
        ans = min(ans, max(self.superEggDrop(K - 1, i - 1)
    return ans
```

如上代码：

- self.superEggDrop(K - 1, i - 1) 指的是鸡蛋破碎的情况，我们就只剩下 K - 1 个鸡蛋， 并且 i - 1 个楼层需要 check。
- self.superEggDrop(K, N - i) + 1 指的是鸡蛋没有破碎的情况，我们仍然有 K 个鸡蛋， 并且剩下 N - i 个楼层需要 check。

接下来，我们增加两行递归的终止条件，这道题就完成了。

```
class Solution:
    def superEggDrop(self, K: int, N: int) -> int:
        if K == 1: return N
        if N == 0 or N == 1: return N
        ans = N
        # 暴力枚举从第 i 层开始扔
        for i in range(1, N + 1):
            ans = min(ans, max(self.superEggDrop(K - 1, i -
        return ans
```

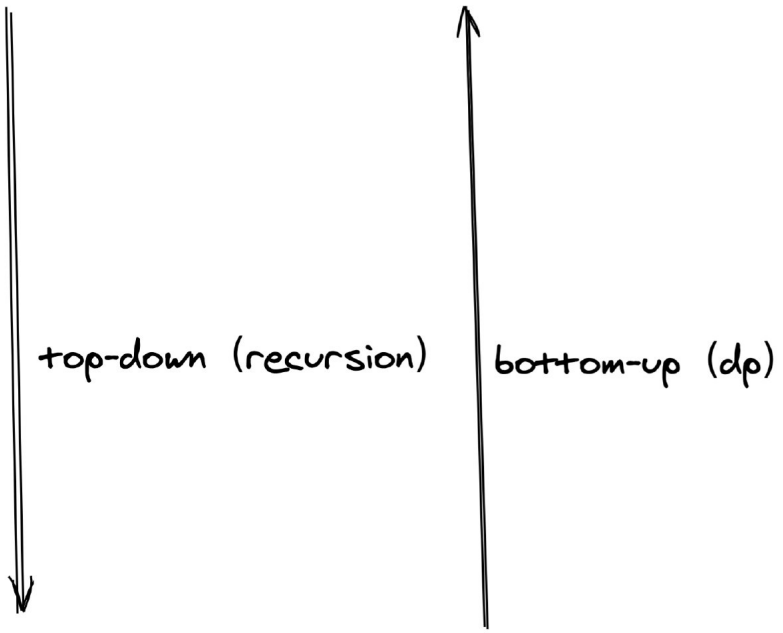
可是如何这就结束的话，这道题也不能是 hard，而且这道题是公认难度较大的 hard 之一。

上面的代码会 TLE，我们尝试使用记忆化递归来试一下，看能不能 AC。

```
class Solution:
    @lru_cache()
    def superEggDrop(self, K: int, N: int) -> int:
        if K == 1: return N
        if N == 0 or N == 1: return N
        ans = N
        # 暴力枚举从第 i 层开始扔
        for i in range(1, N + 1):
            ans = min(ans, max(self.superEggDrop(K - 1, i -
        return ans
```

性能比刚才稍微好一点，但是还是很容易挂。

那只好 bottom-up（动态规划）啦。



(图 3)

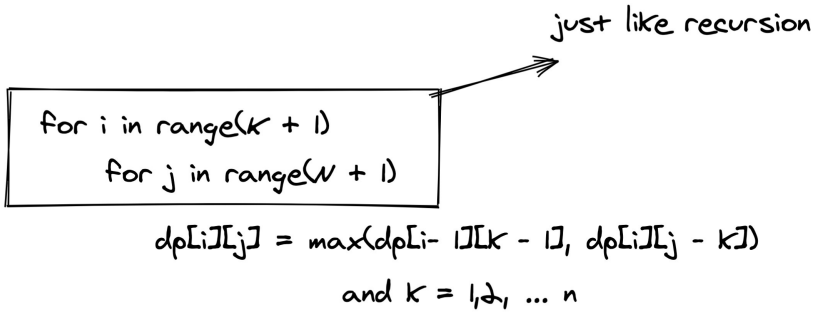
我将上面的过程简写成如下形式：

$$f(k, n) = \min(f(k - 1, i - 1), f(k, n - i))$$
$$\text{and } i = 1, 2, \dots n$$

(图 4)

与其递归地进行这个过程，我们可以使用迭代的方式。 相比于上面的递归式，减少了栈开销。然而两者有着很多的相似之处。

如果说递归是用函数调用来模拟所有情况， 那么动态规划就是用表来模拟。我们知道所有的情况，无非就是 N 和 K 的所有组合，我们怎么去枚举 K 和 N 的所有组合？ 当然是套两层循环啦！



（图 5. 递归 vs 迭代）

如上，你将 dp[i][j] 看成 superEggDrop(i, j)，是不是和递归是一摸一样?

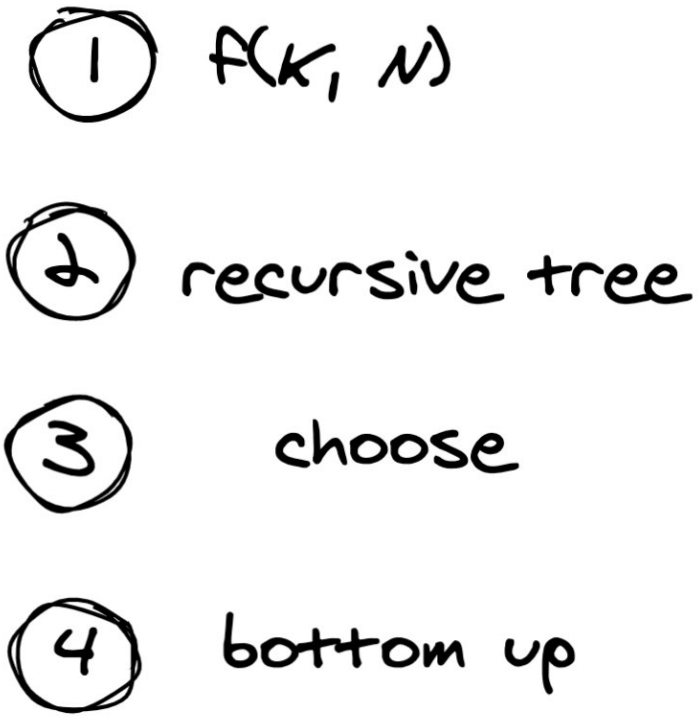
来看下迭代的代码：

```
class Solution:
    def superEggDrop(self, K: int, N: int) -> int:
        for i in range(K + 1):
            for j in range(N + 1):
                if i == 1:
                    dp[i][j] = j
                if j == 1 or j == 0:
                    dp[i][j] == j
                dp[i][j] = j
                for k in range(1, j + 1):
                    dp[i][j] = min(dp[i][j], max(dp[i - 1]
        return dp[K][N]
```

值得注意的是，在这里内外循环的顺序无关紧要，并且内外循坏的顺序对我们写代码来说复杂程度也是类似的，各位客官可以随意调整内外循环的顺序。比如这样也是可以的：

```
class Solution:
    def superEggDrop(self, K: int, N: int) -> int:
        dp = [[0] * (K + 1) for _ in range(N + 1)]

        for i in range(N + 1):
            for j in range( K + 1):
                if j == 1:
                    dp[i][j] = i
                if i == 1 or i == 0:
                    dp[i][j] == i
                dp[i][j] = i
                for k in range(1, i + 1):
                    dp[i][j] = min(dp[i][j], max(dp[k - 1]
        return dp[N][K]
        dp = [[0] * (N + 1) for _ in range(K + 1)]
```

总结一下，上面的解题方法思路是：



然而这样还是不能 AC。这正是这道题困难的地方。 **一道题目往往有不止一种状态转移方程，而不同的状态转移方程往往性能是不同的。**

那么这道题有没有性能更好的其他的状态转移方程呢?

把思路逆转!

> 这是《逆转裁判》 中经典的台词， 主角在深处绝境的时候，会突然冒出这句话，从而逆转思维，寻求突破口。

我们这样来思考这个问题。 既然题目要求最少的扔的次数，假设有一个函数 f(k, i)，他的功能是求出 k 个鸡蛋，扔 i 次所能检测的最高楼层。

我们只需要不断进行发问:

- ”f 函数啊 f 函数，我扔一次可以么？“， 也就是判断 f(k, 1) >= N 的返回值
- ”f 函数啊 f 函数，我扔两次呢？“， 也就是判断 f(k, 2) >= N 的返回值
- ...
- ”f 函数啊 f 函数，我扔 m 次呢？“， 也就是判断 f(k, m) >= N 的返回值

我们只需要返回第一个返回值为 true 的 m 即可。

> 想到这里，我条件发射地想到了二分法。 聪明的小朋友们，你们觉得二分可以么？为什么？欢迎评论区留言讨论。

那么这个神奇的 f 函数怎么实现呢？其实很简单。

- 摔碎的情况，可以检测的最高楼层是 `f(m - 1, k - 1) + 1` 。因为碎了嘛，我们多检测了摔碎的这一层。
- 没有摔碎的情况，可以检测的最高楼层是 `f(m - 1, k)` 。因为没有碎，也就是说我们啥都没检测出来（对能检测的最高楼层无贡献）。

我们来看下代码：

```
class Solution:
    def superEggDrop(self, K: int, N: int) -> int:
        def f(m, k):
            if k == 0 or m == 0: return 0
            return f(m - 1, k - 1) + 1 +  f(m - 1, k)
        m = 0
        while f(m, K) < N:
            m += 1
        return m
```

上面的代码可以 AC。我们来顺手优化成迭代式。

```
class Solution:
    def superEggDrop(self, K: int, N: int) -> int:
        dp = [[0] * (K + 1) for _ in range(N + 1)]
        m = 0
        while dp[m][K] < N:
            m += 1
            for i in range(1, K + 1):
                dp[m][i] = dp[m - 1][i - 1] + 1 + dp[m - 1]
        return m
```

代码

代码支持：JavaSCript，Python

Python:

```python
class Solution:
    def superEggDrop(self, K: int, N: int) -> int:
        dp = [[0] * (K + 1) for _ in range(N + 1)]
        m = 0
        while dp[m][K] < N:
            m += 1
            for i in range(1, K + 1):
                dp[m][i] = dp[m - 1][i - 1] + 1 + dp[m - 1]
        return m
```

JavaSCript:

```javascript
var superEggDrop = function (K, N) {
  // 不选择dp[K][M]的原因是dp[M][K]可以简化操作
  const dp = Array(N + 1)
    .fill(0)
    .map((_) => Array(K + 1).fill(0));

  let m = 0;
  while (dp[m][K] < N) {
    m++;
    for (let k = 1; k <= K; ++k) dp[m][k] = dp[m - 1][k - 1
  }
  return m;
};
```

复杂度分析

- 时间复杂度：$O(m * K)$，其中 m 为答案。
- 空间复杂度：$O(K * N)$

对为什么用加法的同学有疑问的可以看我写的《对《丢鸡蛋问题》的一点补充》。

总结

- 对于困难，先举几个简单例子帮助你思考。
- 递归和迭代的关系，以及如何从容地在两者间穿梭。
- 如果你还不熟悉动态规划，可以先从递归做起。多画图，当你做多了题之后，就会越来越从容。
- 对于动态规划问题，往往有不止一种状态转移方程，而不同的状态转移方程往往性能是不同的。

> 友情提示: 大家不要为了这个题目高空抛物哦。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

欢迎长按关注

题目地址（895. 最大频率栈）

https://leetcode-cn.com/problems/maximum-frequency-stack/

题目描述

```
实现 FreqStack，模拟类似栈的数据结构的操作的一个类。

FreqStack 有两个函数：

push(int x)，将整数 x 推入栈中。
pop()，它移除并返回栈中出现最频繁的元素。
如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。


示例：

输入：
["FreqStack","push","push","push","push","push","push","pop
[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]
输出：[null,null,null,null,null,null,null,5,7,5,4]
解释：
执行六次 .push 操作后，栈自底向上为 [5,7,5,7,4,5]。然后：

pop() -> 返回 5，因为 5 是出现频率最高的。
栈变成 [5,7,5,7,4]。

pop() -> 返回 7，因为 5 和 7 都是频率最高的，但 7 最接近栈顶。
栈变成 [5,7,5,4]。

pop() -> 返回 5 。
栈变成 [5,7,4]。

pop() -> 返回 4 。
栈变成 [5,7]。


提示：

对 FreqStack.push(int x) 的调用中 0 <= x <= 10^9。
如果栈的元素数目为零，则保证不会调用  FreqStack.pop()。
单个测试样例中，对 FreqStack.push 的总调用次数不会超过 10000。
单个测试样例中，对 FreqStack.pop 的总调用次数不会超过 10000。
所有测试样例中，对 FreqStack.push 和 FreqStack.pop 的总调用次数不
```

前置知识

- 设计题
- 栈
- 哈希表

公司

- 暂无

思路

设计题目基本都是选择好数据结构，那么算法实现就会很容易。 如果你不会这道题，并去看其他人的题解代码，会发现很多时候都比较容易理解。 你没有能做出来的原因很大程度上是因为**对基础数据结构**不熟悉。设计题基本不太会涉及到算法，如果有算法， 也比较有限，常见的有二分法。

对于这道题来说，我们需要涉及一个栈。 这个栈弹出的不是最近压入栈的，而是频率最高的。

> 实际上，这已经不是栈了，只是它愿意这么叫。

既然要弹出频率最高的，那么我们肯定要统计所有栈中数字的出现频率。由于数字范围比较大，因此使用哈希表是一个不错的选择。为了能更快的求出频率最高的，我们需要将频率最高的数字（或者其出现次数）存起来。

另外题目要求**如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素**。我们不妨就使用一个栈 fraq_stack 来维护，将相同频率的数字放到一个栈中。这样频率相同的我们直接出栈就可以取到**最接近栈顶的元素**啦。存储结构为：

```
{
   3: [1,2,3],
    2: [1,2,3,4],
    1: [1,2,3,4,5]
}
```

上面的结构表示：

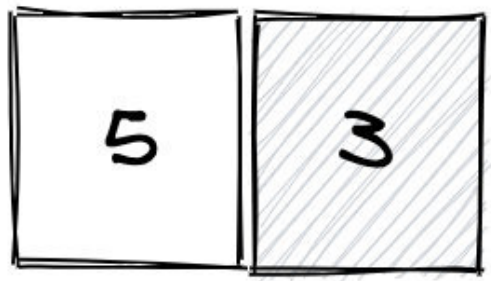
- 1,2,3 出现了 3 次
- 4 出现了 2 次
- 5 出现了 1 次

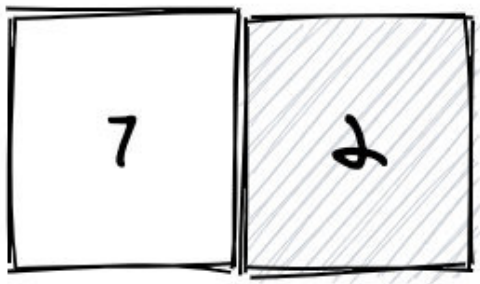
细心的同学可能发现了，频率为 2 的列表中同样存储了频率更高（这里是频率为 3）的数字。

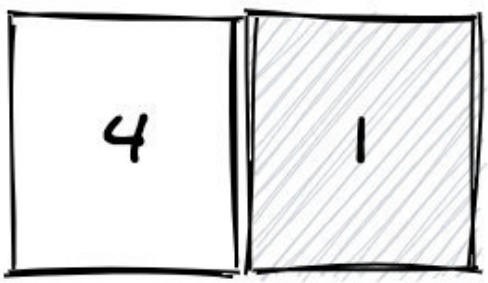
这是故意的。比如我们将 3 弹出，那么 3 其实就变成了频率为 2 的数字了。这个时候，我们如何将 3 插入到频率为 2 的栈的正确位置呢？其实你只要按照我上面的数据结构进行设计就没有这个问题啦。

我们以题目给的例子来讲解。

- 使用 fraq 来存储对应的数字出现次数。key 是数字，value 频率

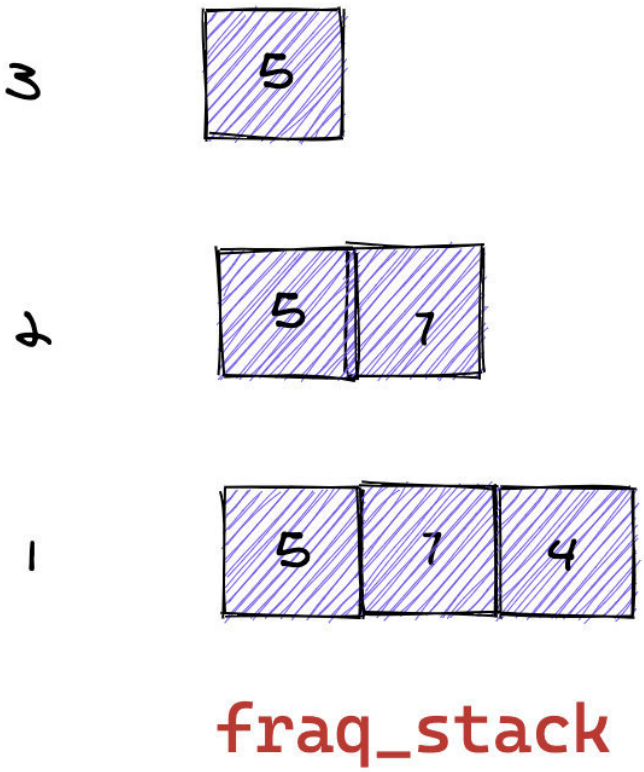




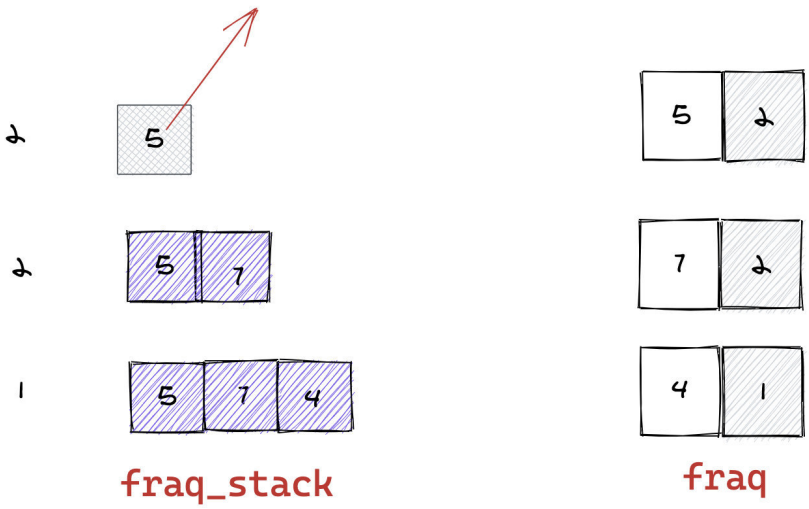


fraq

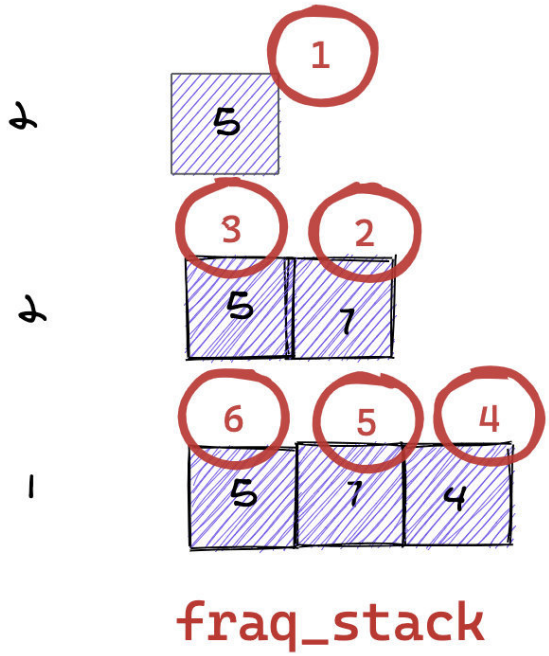
- 由于题目限制“如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。”，我们考虑使用栈来维护一个频率表 fraq_stack。key 是频率，value 是数字组成的栈。



- 同时用 max_fraq 记录当前的最大频率值。
- 第一次 pop 的时候，我们最大的频率是 3。由 fraq_stack 知道我们需要 pop 掉 5。



- 之后 pop 依次是这样的（红色数字表示顺序）



关键点解析

- 栈的基本性质
- hashtable 的基本性质
- fraq_stack 的设计。fraq_stack 中当前频率的栈要保存所有大于等于其频率的数字
- push 和 pop 的时候同时更新 fraq，max_fraq 和 fraq_stack。

代码

```
class FreqStack:

    def __init__(self):
        self.fraq = collections.defaultdict(lambda: 0)
        self.fraq_stack = collections.defaultdict(list)
        self.max_fraq = 0

    def push(self, x: int) -> None:
        self.fraq[x] += 1
        if self.fraq[x] > self.max_fraq:
            self.max_fraq = self.fraq[x]
        self.fraq_stack[self.fraq[x]].append(x)

    def pop(self) -> int:
        ans = self.fraq_stack[self.max_fraq].pop()
        self.fraq[ans] -= 1
        if not self.fraq_stack[self.max_fraq]:
            self.max_fraq -= 1
        return ans

# Your FreqStack object will be instantiated and called as
# obj = FreqStack()
# obj.push(x)
# param_2 = obj.pop()
```

复杂度分析

这里的复杂度为均摊复杂度。

- 时间复杂度：$O(1)$
- 空间复杂度：$O(1)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址 (975. 奇偶跳）

https://leetcode-cn.com/problems/odd-even-jump/

题目描述

给定一个整数数组 A，你可以从某一起始索引出发，跳跃一定次数。在你跳跃的

你可以按以下方式从索引 i 向后跳转到索引 j（其中 i < j）：

在进行奇数跳跃时（如，第 1，3，5... 次跳跃），你将会跳到索引 j，使得
在进行偶数跳跃时（如，第 2，4，6... 次跳跃），你将会跳到索引 j，使得
（对于某些索引 i，可能无法进行合乎要求的跳跃。）
如果从某一索引开始跳跃一定次数（可能是 0 次或多次），就可以到达数组的末

返回好的起始索引的数量。

示例 1：

输入：[10,13,12,14,15]
输出：2
解释：
从起始索引 i = 0 出发，我们可以跳到 i = 2，（因为 A[2] 是 A[1]，A
从起始索引 i = 1 和 i = 2 出发，我们可以跳到 i = 3，然后我们就无法
从起始索引 i = 3 出发，我们可以跳到 i = 4，到达数组末尾。
从起始索引 i = 4 出发，我们已经到达数组末尾。
总之，我们可以从 2 个不同的起始索引（i = 3, i = 4）出发，通过一定数
示例 2：

输入：[2,3,1,1,4]
输出：3
解释：
从起始索引 i=0 出发，我们依次可以跳到 i = 1，i = 2，i = 3：

在我们的第一次跳跃（奇数）中，我们先跳到 i = 1，因为 A[1] 是（A[1]，

在我们的第二次跳跃（偶数）中，我们从 i = 1 跳到 i = 2，因为 A[2] 是

在我们的第三次跳跃（奇数）中，我们从 i = 2 跳到 i = 3，因为 A[3] 是

我们不能从 i = 3 跳到 i = 4，所以起始索引 i = 0 不是好的起始索引。

类似地，我们可以推断：
从起始索引 i = 1 出发， 我们跳到 i = 4，这样我们就到达数组末尾。
从起始索引 i = 2 出发， 我们跳到 i = 3，然后我们就不能再跳了。
从起始索引 i = 3 出发， 我们跳到 i = 4，这样我们就到达数组末尾。
从起始索引 i = 4 出发，我们已经到达数组末尾。
总之，我们可以从 3 个不同的起始索引（i = 1, i = 3, i = 4）出发，通
示例 3：

输入：[5,1,3,4,2]
输出：3

解释：
我们可以从起始索引 1，2，4 出发到达数组末尾。

提示：

```
1 <= A.length <= 20000
0 <= A[i] < 100000
```

前置知识

- 单调栈

公司

- 暂无

思路

题目要求我们从数组某一个索引出发交替跳高和跳低（奇偶跳），如果可以跳到末尾，则计数器加一，最终返回计数器的值。

这种题目一般都是倒着思考比较容易。因为我虽然不知道你**从哪开始**可以跳到最后，但是我知道最终被计算进返回值的一定是在数组末尾**结束的**。

我们先尝试从题目给的例子打开思路。

以题目中的[10,13,12,14,15]为例。最终计入计数器的出发点一定是跳到了 15 上，15 这一步既可以是跳高（奇数跳）过来的，也可以是跳低（偶数跳）过来的。

- 如果是跳高过来的，那么一定是 14，因此只有 14 的下一个**最小的**比其大（或等于）的是 15。
- 不可能是跳低过来的，因为没有比它大的。而如果前面有比它大的，那一定是找一个数 x，x 的下一个**最大**的比其小（或等于）的是 15。

一开始我想到的是单调栈，单很快就发现这行不通。因为题目要求的并不是**下一个**比其大（或等于）的数，而是**下一个最小的**比其大（或等于）。

如果题目要求的是**下一个**比其大（或等于）的数。那么我可以写出如下的代码：

```
n = len(A)
next_higher, next_lower = [-1] * n, [-1] * n

stack = []
for i, a in enumerate(A):
    while stack and A[stack[-1]] <= A[i]:
        next_higher[stack.pop()] = i
    stack.append(i)
stack = []
for i, a in enumerate(A):
    while stack and A[stack[-1]] >= A[i]:
        next_lower[stack.pop()] = i
    stack.append(i)
```

对上面代码不熟悉的朋友，可以看下我之前写的 单调栈专题。

可是我们需要求的是**下一个最小的**比其大（或等于）呀。一种简单的方法是先对 A 进行排序再使用单调栈。比如我们进行升序排序，接下来只要遍历一次排好序的数组，同时结合单调栈即可。 由于已经进行了排序，因此后面的数一定是**不小于**前面的数的，且**对于任意相邻的数 a 和 b，a 的最小的大于等于它本身的数就是 b**，前提是 a 和 b 对应排序之前的索引 i 和 j 满足 i < j。这提示我们排序的时候需要额外记录原始索引。

代码：

```
A = sorted([a, i] for i, a in enumerate(A))
```

这里有 1 个细节。即排序的时候如何处理相等情况，比如 a 和 b 相等，是保持之前的相对顺序还是逆序还是都可以？实际上，我们想希望的是保持之前的相对顺序，这样才不会错过相等的情况的解。因此我这里排序的是时候是以 [a, i] 形式保存的数据。

由于除了要处理跳高，我们仍然需要处理跳低。而最关键的是跳低也需要我们**在 a 和 b 相等的情况下，保持之前的相对顺序**。 因此就不能通过简单的排序一次处理。比如我们不能这么干：

```
class Solution:
    def oddEvenJumps(self, A):
        n = len(A)
        next_higher, next_lower = [0] * n, [0] * n
        A = sorted([a, i] for i, a in enumerate(A))

        stack = []
        for _, i in A:
            # it means stack[-1]'s next bigger(or equal) is
            while stack and stack[-1] < i:
                next_higher[stack.pop()] = i
            stack.append(i)

        stack = []
        for _, i in A[::-1]:
            # it means stack[-1]'s next smaller(or equal) i
            while stack and stack[-1] < i:
                next_lower[stack.pop()] = i
            stack.append(i)

        # ...
```

解决这个问题的方法最简单的莫过于使用两次排序，具体见下方代码区。

现在已经有了两个数组，这两个数组可以帮助我们

- **快速**找到下一个最小的比其大（或等于）的数。（奇数跳）
- **快速**找到下一个最大的比其小（或等于）的数。（偶数跳）

数据已经预处理完毕。接下来，只需要从结果倒推即可。这提示我们从后往前遍历。

算法：

- 使用前面的方法预处理出 next_higher 和 next_lower
- 使用两个数组 higher 和 lower。higher[i] 表示是否可从 i 开始通过跳高的方式奇偶跳到达数组最后一项，lower 也是类似。
- 从后往前倒推遍历。如果 lower[next_higher[i]] 为 true 说明 i 是可以通过跳高的方式奇偶跳到达数组最后一项的。higher[next_lower[i]] 也是类似。
- 最后返回 higher 中 为 true 的个数。

代码

代码支持： Python3

Python Code:

```
class Solution:
    def oddEvenJumps(self, A):
        n = len(A)
        next_higher, next_lower = [0] * n, [0] * n

        stack = []
        for _, i in sorted([a, i] for i, a in enumerate(A)
            # it means stack[-1]'s next bigger(or equal) is
            while stack and stack[-1] < i:
                next_higher[stack.pop()] = i
            stack.append(i)

        stack = []
        for _, i in sorted([-a, i] for i, a in enumerate(A
            # it means stack[-1]'s next smaller(or equal) i
            while stack and stack[-1] < i:
                next_lower[stack.pop()] = i
            stack.append(i)

        higher, lower = [False] * n, [False] * n
        higher[-1] = lower[-1] = True
        ans = 1
        for i in range(n - 2, -1, -1):
            higher[i] = lower[next_higher[i]]
            lower[i] = higher[next_lower[i]]
            ans += higher[i]
        return ans
```

复杂度分析

令 N 为数组 A 的长度。

- 时间复杂度：$O(NlogN)$
- 空间复杂度：$O(N)$

有的同学好奇为什么不考虑 lower。类似：

```
ans = 1
for i in range(n - 2, -1, -1):
    higher[i] = lower[next_higher[i]]
    lower[i] = higher[next_lower[i]]
    ans += higher[i] or lower[i]
return ans
```

根本原因是题目**要求我们必须从奇数跳开始**，而不是能偶数跳开始。如果题目不限制奇数跳和偶数跳，你可以自己自由选择的话，就必须使用上面的代码啦。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(995. K 连续位的最小翻转次数)

https://leetcode-cn.com/problems/minimum-number-of-k-consecutive-bit-flips/

题目描述

```
在仅包含 0 和 1 的数组 A 中，一次 K 位翻转包括选择一个长度为 K 的（

返回所需的 K 位翻转的最小次数，以便数组没有值为 0 的元素。如果不可能，



示例 1：

输入：A = [0,1,0], K = 1
输出：2
解释：先翻转 A[0]，然后翻转 A[2]。


示例 2：

输入：A = [1,1,0], K = 2
输出：-1
解释：无论我们怎样翻转大小为 2 的子数组，我们都不能使数组变为 [1,1,1


示例 3：

输入：A = [0,0,0,1,0,1,1,0], K = 3
输出：3
解释：
翻转 A[0],A[1],A[2]: A变成 [1,1,1,1,0,1,1,0]
翻转 A[4],A[5],A[6]: A变成 [1,1,1,1,1,0,0,0]
翻转 A[5],A[6],A[7]: A变成 [1,1,1,1,1,1,1,1]



提示：

1 <= A.length <= 30000
1 <= K <= A.length
```

前置知识

- 连续子数组优化

公司

- 暂无

暴力解

思路

首先考虑暴力的解法。暴力的思路可以是从左到右遍历数组，如果碰到一个 0，我们以其为左端进行翻转。翻转的长度自然是以其开始长度为 K 的子数组了。由于是**以其为左端进行翻转**，因此如果遇到一个 0 ，我们必须执行翻转，否则就无法得到全 1 数组。由于翻转的顺序不影响最终结果，即如果最终答案是翻转以 i, j , k 为起点的子数组，那么先翻转谁后翻转谁都是一样的。因此采用从左往右遍历的方式是可以的。

概括一下：暴力的思路可以是从左到右遍历数组，如果碰到一个 0，我们以其为左端进行翻转，并修改当前位置开始的长度为 k 的子数组，同时计数器 + 1，最终如果数组不全为 0 则返回 -1 ，否则返回计数器的值。

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def minKBitFlips(self, A, K):
        N = len(A)
        ans = 0
        for i in range(N - K + 1):
            if A[i] == 1:
                continue
            for j in range(K):
                A[i + j] ^= 1
            ans += 1
        for i in range(N):
            if A[i] == 0:
                return -1
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n * k)$
- 空间复杂度：$O(1)$

连续子数组优化

思路

对于这种连续子数组的题目。一般优化思路就那么几种。我们来枚举一下：

- 前缀和 & 差分数组
- 滑动窗口
- 双端队列。比如 1696. 跳跃游戏 VI 和 239. 滑动窗口最大值 就是这种思路。

这三种技巧我都写过文章，如果不了解可以先看下。

对于这道题来说，我们可使用差分数组或者双端队列来优化。不管采用哪种，基本思路都差不多，你也可以对比下方代码看一下他们思路的一致性。 简单来说，他们的思路差不多，差别只是解决问题的使用的数据结构不同，因此 api 不同罢了。因此我并没有将二者作为两个解法。

对于差分数组来说，上面暴力解法内层有一个次数为 k 的循环。而如果使用差分数组只修改端点的值，就可轻松将时间复杂度优化到 $O(n)$。

对于双端队列来说，如果当前位置需要翻转，那么就将其入队。那如何判断当前位置的数字是多少呢？（可能由于前面数字的翻转，当前位置**被**翻转了若干次，可能不是以前的数字了）。由于被翻转偶数次等于没有翻转，被翻转奇数次效果一样，因此只需要记录被翻转次数的奇偶性即可。而这其实就是队列长度的奇偶性。因为**此时队列的长度就是当前数字被翻转的次数**。当然这要求你将长度已经大于 k 的从队列中移除。

关键点

- 连续子数组优化技巧

代码

- 语言支持：Python3

Python3 Code:

差分数组：

```python
class Solution:
    def minKBitFlips(self, A: List[int], K: int) -> int:
        n = len(A)
        diff = [0] * (n + 1)
        ans, cur = 0, 0
        for i in range(n):
            cur += diff[i]
            if cur % 2 == A[i]:
                if i + K > n:
                    return -1
                ans += 1
                cur += 1
                diff[i + K] -= 1
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

双端队列：

```python
class Solution:
    def minKBitFlips(self, A, K):
        N = len(A)
        q = collections.deque()
        ans = 0
        for i in range(N):
            if q and i >= q[0] + K:
                q.popleft()
            if len(q) % 2 == A[i]:
                if i + K > N:
                    return -1
                q.append(i)
                ans += 1
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(k)$

空间复杂度为 O(1) 的算法

思路

不管是使用差分数组和还是双端队列，其实都没有必要使用额外的数据结构，而是使用原来的数组 A，也就是**原地修改**。

比如，我们可以只记录双端队列的长度，要判断一个数字是否在队列里，只需要将其映射到题目数据范围内的另外一个数字即可。由于题目数据范围是 [0,1] 因此我们可以将其映射到 2。

关键点

- 原地修改

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def minKBitFlips(self, A, K):
        flips = ans = 0
        for i in range(len(A)):
            if i >= K and A[i - K] - 2 == 0:
                flips -= 1
            if (flips % 2) == A[i]:
                if i + K > len(A):
                    return -1
                A[i] = 2
                flips += 1
                ans += 1
        return ans
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址(1032. 字符流)

https://leetcode-cn.com/problems/stream-of-characters/

题目描述

```
按下述要求实现 StreamChecker 类：

StreamChecker(words)：构造函数，用给定的字词初始化数据结构。
query(letter)：如果存在某些 k >= 1，可以用查询的最后 k个字符（按从


示例：

StreamChecker streamChecker = new StreamChecker(["cd","f","
streamChecker.query('a');          // 返回 false
streamChecker.query('b');          // 返回 false
streamChecker.query('c');          // 返回 false
streamChecker.query('d');          // 返回 true，因为 'cd' 在
streamChecker.query('e');          // 返回 false
streamChecker.query('f');          // 返回 true，因为 'f' 在
streamChecker.query('g');          // 返回 false
streamChecker.query('h');          // 返回 false
streamChecker.query('i');          // 返回 false
streamChecker.query('j');          // 返回 false
streamChecker.query('k');          // 返回 false
streamChecker.query('l');          // 返回 true，因为 'kl' 在


提示：

1 <= words.length <= 2000
1 <= words[i].length <= 2000
字词只包含小写英文字母。
待查项只包含小写英文字母。
待查项最多 40000 个。
```

前置知识

- 前缀树

公司

- 字节

思路

很明显，我们需要将历史 query 的字符全部记录下来。

比如：

```
streamChecker.query("a"); // stream:  a
streamChecker.query("b"); // stream: ab
streamChecker.query("c"); // stream: abc
```

之后我们用拼接的单词在 words 中查询即可， 最简单的方式当然是每次 query 都去扫描一次，这种方式时间复杂度为 $O(m *n* q)$，其中 m 和 n 分别为 words 的长度和， words[i] 的平均长度，m 和 n 最大为 2000，q 是待查项，最大为 40000， 毫无疑问会超时。

我们可以采用构建 Trie 的形式，即以空间环时间， 其代码和常规的 Trie 类似，只需要将 search(word) 函数做一个简单修改即可，我们不需要检查整个 word 是否存在， 而已 word 的前缀存在即可。

> 提示：可以通过对 words 去重，来用空间换区时间。

具体算法：

- init 中 构建 Trie 和 双端队列 stream
- query 时，往 stream 的左边 append 即可。
- 调用 Trie 的 search（和常规的 search 稍有不同， 我上面已经讲了）

以上面的例子为例：

```
streamChecker.query("a"); // stream:  a
streamChecker.query("b"); // stream: ab
streamChecker.query("c"); // stream: abc
```

当 query("c") 的时候，我们需要查 abc，bc， c 三个是否**有一个**在 words 中。由于我们知道待查项的尾字符（在这个例子中尾字符是 c），因此从尾字符开始在前缀树中搜索，看是否能够搜索到 word 即可。这提示我们**前缀树倒序插入或者 stream 倒序插入**。

这里有两个小的点需要注意：

1. 如果用数组来存储， 由于每次都往数组头部插入一个元素，因此每次 query 操作的时间复杂度为 $O(N)$，其中 N 为截止当前执行 query 的次数，我们可以使用双端队列进行优化。

2. 由于不必 query 形成的查询全部命中。比如 stream 为 cba 的时候，找到单词 c， bc， abc 都是可以的。如果是找到 c，cb，cba 比较好吧，现在是反的。其实我们可以反序插入是，类似的技巧在211.add-and-search-word-data-structure-design 也有用到。

核心代码（Python）：

```
class StreamChecker:

    def __init__(self, words: List[str]):
        self.trie = Trie()
        self.stream = deque([])

        for word in set(words):
            self.trie.insert(word[::-1])

    def query(self, letter: str) -> bool:
        self.stream.appendleft(letter)
        return self.trie.search(self.stream)
```

关键点解析

- 前缀树模板
- 倒序插入

代码

- 语言支持: Python

Python Code:

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.Trie = {}

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                curr[w] = {}
            curr = curr[w]
        curr['#'] = 1

    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        curr = self.Trie
        for w in word:
            if w not in curr:
                return False
            if "#" in curr[w]:
                return True
            curr = curr[w]
        return False


class StreamChecker:

    def __init__(self, words: List[str]):
        self.trie = Trie()
        self.stream = deque([])

        for word in set(words):
            self.trie.insert(word[::-1])

    def query(self, letter: str) -> bool:
```

```
            self.stream.appendleft(letter)
            return self.trie.search(self.stream)
```

相关题目

- 0208.implement-trie-prefix-tree
- 0211.add-and-search-word-data-structure-design
- 0212.word-search-ii
- 0472.concatenated-words
- 0820.short-encoding-of-words

题目地址(1168. 水资源分配优化)

https://leetcode.com/problems/optimize-water-distribution-in-a-village/

题目描述

```
村庄内有n户人家，我们可以通过挖井或者建造水管向每家供水。

对于每户人家i，我们可以通过花费 wells[i] 直接在其房内挖水井，或者通过

请求出所有住户都能通水的最小花费。

示例1：


输入: n = 3, wells = [1,2,2], pipes = [[1,2,1],[2,3,1]]
输出: 3
解释:
The image shows the costs of connecting houses using pipes.
The best strategy is to build a well in the first house wit
提示：

1 <= n <= 10000
wells.length == n
0 <= wells[i] <= 10^5
1 <= pipes.length <= 10000
1 <= pipes[i][0], pipes[i][1] <= n
0 <= pipes[i][2] <= 10^5
pipes[i][0] != pipes[i][1]
```

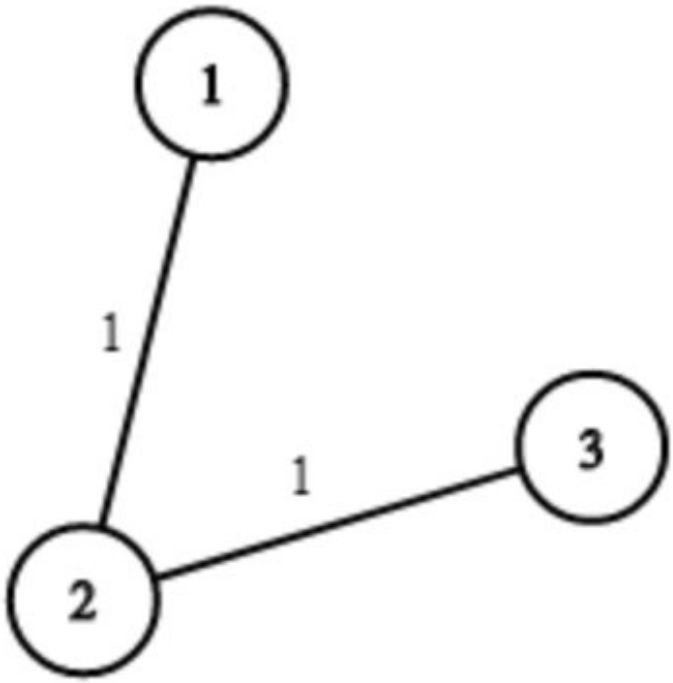
前置知识

- 图
- 最小生成树

公司

- 暂无

思路



题意，在每个城市打井需要一定的花费，也可以用其他城市的井水，城市之间建立连接管道需要一定的花费，怎么样安排可以花费最少的前灌溉所有城市。

这是一道连通所有点的最短路径/最小生成树问题，把城市看成图中的点，管道连接城市看成是连接两个点之间的边。这里打井的花费是直接在点上，而且并不是所有 点之间都有边连接，为了方便，我们可以假想一个点 `（root）0`，这里自身点的花费可以与 `0` 连接，花费可以是 `0-i` 之间的花费。这样我们就可以构建一个连通图包含所有的点和边。 那在一个连通图中求最短路径/最小生成树的问题.

参考延伸阅读中，维基百科针对这类题给出的几种解法。

解题步骤：

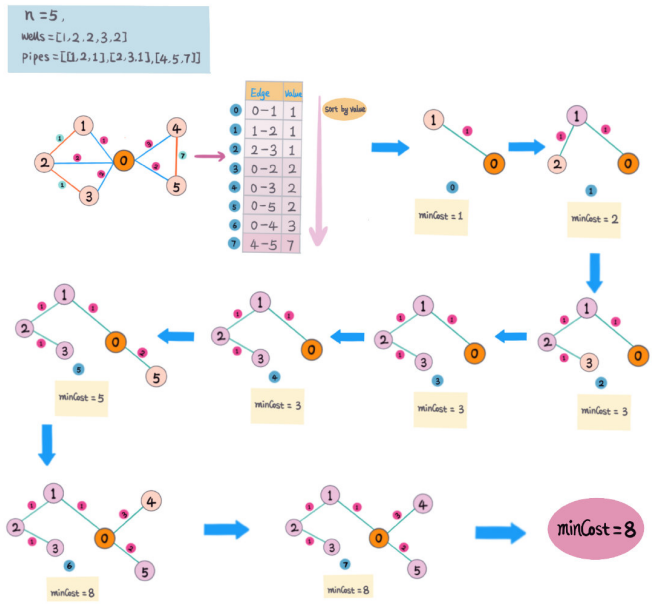
1. 创建 `POJO EdgeCost(node1, node2, cost) - 节点1 和 节点2 连接边的花费` 。
2. 假想一个 `root` 点 `0`，构建图
3. 连通所有节点和 `0`， `[0,i] - i 是节点 [1,n]`， `0-1` 是节点 `0` 和 `1` 的边，边的值是节点 `i` 上打井的花费 `wells[i]`；
4. 把打井花费和城市连接点转换成图的节点和边。
5. 对图的边的值排序（从小到大）
6. 遍历图的边，判断两个节点有没有连通 （ `Union-Find` ），
    - 已连通就跳过，继续访问下一条边
    - 没有连通，记录花费，连通节点
7. 若所有节点已连通，求得的最小路径即为最小花费，返回

8. 对于每次 `union` ，节点数 `n-1` ，如果 `n==0` 说明所有节点都已连通，可以提前退出，不需要继续访问剩余的边。

> 这里用加权Union-Find 判断两个节点是否连通，和连通未连通的节点。

举例： `n = 5, wells=[1,2,2,3,2], pipes=[[1,2,1],[2,3,1],[4,5,7]]`

如图：



从图中可以看到，最后所有的节点都是连通的。

复杂度分析

- 时间复杂度: `O(ElogE) - E 是图的边的个数`
- 空间复杂度: `O(E)`

> 一个图最多有 `n(n-1)/2 - n 是图中节点个数` 条边 （完全连通图）

关键点分析

1. 构建图，得出所有边
2. 对所有边排序
3. 遍历所有的边（从小到大）
4. 对于每条边，检查是否已经连通，若没有连通，加上边上的值，连通两个节点。若已连通，跳过。

代码 (`Java/Python3`)

Java code

```
class OptimizeWaterDistribution {
  public int minCostToSupplyWater(int n, int[] wells, int
    List<EdgeCost> costs = new ArrayList<>();
    for (int i = 1; i <= n; i++) {
      costs.add(new EdgeCost(0, i, wells[i - 1]));
    }
    for (int[] p : pipes) {
      costs.add(new EdgeCost(p[0], p[1], p[2]));
    }
    Collections.sort(costs);
    int minCosts = 0;
    UnionFind uf = new UnionFind(n);
    for (EdgeCost edge : costs) {
      int rootX = uf.find(edge.node1);
      int rootY = uf.find(edge.node2);
      if (rootX == rootY) continue;
      minCosts += edge.cost;
      uf.union(edge.node1, edge.node2);
      // for each union, we connnect one node
      n--;
      // if all nodes already connected, terminate early
      if (n == 0) {
        return minCosts;
      }
    }
    return minCosts;
  }

  class EdgeCost implements Comparable<EdgeCost> {
    int node1;
    int node2;
    int cost;
    public EdgeCost(int node1, int node2, int cost) {
      this.node1 = node1;
      this.node2 = node2;
      this.cost = cost;
    }

    @Override
    public int compareTo(EdgeCost o) {
      return this.cost - o.cost;
    }
  }

  class UnionFind {
    int[] parent;
    int[] rank;
    public UnionFind(int n) {
```

```
        parent = new int[n + 1];
        for (int i = 0; i <= n; i++) {
          parent[i] = i;
        }
        rank = new int[n + 1];
      }
      public int find(int x) {
        return x == parent[x] ? x : find(parent[x]);
      }
      public void union(int x, int y) {
        int px = find(x);
        int py = find(y);
        if (px == py) return;
        if (rank[px] >= rank[py]) {
          parent[py] = px;
          rank[px] += rank[py];
        } else {
          parent[px] = py;
          rank[py] += rank[px];
        }
      }
    }
  }
```

Pythong3 code

```
class Solution:
    def minCostToSupplyWater(self, n: int, wells: List[int]
        union_find = {i: i for i in range(n + 1)}

        def find(x):
            return x if x == union_find[x] else find(union_

        def union(x, y):
            px = find(x)
            py = find(y)
            union_find[px] = py

        graph_wells = [[cost, 0, i] for i, cost in enumerat
        graph_pipes = [[cost, i, j] for i, j, cost in pipes
        min_costs = 0
        for cost, x, y in sorted(graph_wells + graph_pipes)
            if find(x) == find(y):
                continue
            union(x, y)
            min_costs += cost
            n -= 1
            if n == 0:
                return min_costs
```

延伸阅读

1. 最短路径问题
2. Dijkstra算法
3. Floyd-Warshall算法
4. Bellman-Ford算法
5. Kruskal算法
6. Prim's 算法
7. 最小生成树

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(1178. 猜字谜)

https://leetcode-cn.com/problems/number-of-valid-words-for-each-puzzle/

题目描述

外国友人仿照中国字谜设计了一个英文版猜字谜小游戏，请你来猜猜看吧。

字谜的谜面 puzzle 按字符串形式给出，如果一个单词 word 符合下面两个条

单词 word 中包含谜面 puzzle 的第一个字母。
单词 word 中的每一个字母都可以在谜面 puzzle 中找到。
例如，如果字谜的谜面是 "abcdefg"，那么可以作为谜底的单词有 "faced",

返回一个答案数组 answer，数组中的每个元素 answer[i] 是在给出的单词列

示例：

输入：
words = ["aaaa","asas","able","ability","actt","actor","acc
puzzles = ["aboveyz","abrodyz","abslute","absoryz","actresz
输出：[1,1,3,2,4,0]
解释：
1 个单词可以作为 "aboveyz" 的谜底 : "aaaa"
1 个单词可以作为 "abrodyz" 的谜底 : "aaaa"
3 个单词可以作为 "abslute" 的谜底 : "aaaa", "asas", "able"
2 个单词可以作为 "absoryz" 的谜底 : "aaaa", "asas"
4 个单词可以作为 "actresz" 的谜底 : "aaaa", "asas", "actt", "a
没有单词可以作为 "gaswxyz" 的谜底，因为列表中的单词都不含字母 'g'。

提示：

1 <= words.length <= 10^5
4 <= words[i].length <= 50
1 <= puzzles.length <= 10^4
puzzles[i].length == 7
words[i][j], puzzles[i][j] 都是小写英文字母。
每个 puzzles[i] 所包含的字符都不重复。

前置知识

- 枚举子集
- 位运算
- 前缀树

公司

- 暂无

位运算

思路

朴素的想法是模拟。即遍历 puzzles 和 words 的所有组合，并判断是否满足条件，如果满足则计数器+1，最后返回计数器的值。

暴力法代码：

```
class Solution:
    def findNumOfValidWords(self, words: List[str], puzzles
        s_word = [set(word) for word in words]
        ans = []
        for puzzle in puzzles:
            cnt = 0
            for word in s_word:
                if puzzle[0] not in word:
                    continue
                flag = False
                for c in word:
                    if c not in puzzle:
                        flag = True
                        break
                if not flag:
                    cnt += 1
            ans.append(cnt)
        return ans
```

由于这种做法需要遍历 puzzles 和 words 的所有组合，因此时间复杂不低于是 $O(m * n)$，其中 m 和 n 分别为 words 和 puzzles 的长度。看下题目的约束条件：

```
1 <= words.length <= 10^5
1 <= puzzles.length <= 10^4
```

可知道 m * n 最大可以是 $10 ^ 9$ ，这显然是无法接受的。

> lucifer 小提示：小于等于 10 ^ 7 才可以哦

注意到题目的约束条件：

```
puzzles[i].length == 7
```

基本可以锁定为是**状态压缩**。力扣相关的题目很多，基本都是有一个约束条件的数据范围很小（比如 20 以内）。我们要做的通常就是**给这个小的变量做状态压缩**。

这道题的关键其实就是**单词 word 中的每一个字母都可以在谜面 puzzle 中找到**，所有的重复计算都是基于此产生的。这句话的含义其实就是 **word 中字符组成的集合是 puzzle 中组成的集合的子集**。

基于上面两条重要信息，我们可以初步锁定算法为：

- 用二进制表示 puzzle
- 枚举 puzzle 的所有子集（二进制子集枚举）
- 判断所有子集 j 是否在 words 中出现过。如果出现过，则将计数器累加出现的次数。这提示我们同样将 word 使用二进制进行存储。虽然 words[i] 的长度范围比较大（[4,50]），但我们关心的其实是**去重的子集**。注意到 words[i] 的取值范围是小写字符，因此这个范围不大于 26，使用 int 存储完全够了。

枚举二进制子集是一个常见的操作，竞赛中也不时出现，大家可以阅读相关内容，具体原理不再展开。

关键点

- 枚举子集算法

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def findNumOfValidWords(self, words: List[str], puzzles
        counts = collections.defaultdict(int)
        ans = [0] * len(puzzles)
        for word in words:
            bit = 0 # bit 是 word 的二进制表示
            for c in word:
                bit |= 1 << ord(c) - ord("a")
            counts[bit] += 1
        for i, puzzle in enumerate(puzzles):
            bit = 0 # bit 是 puzzle 的二进制表示
            for c in puzzle:
                bit |= 1 << ord(c) - ord("a")
            j = bit # j 是 bit 的子集
            # 倒序枚举 bit 的子集 j
            while j:
                # 单词 word 需要包含谜面的第一个字母
                if 1 << ord(puzzle[0]) - ord("a") & j:
                    ans[i] += counts[j]
                j = bit & (j - 1)
        return ans
```

复杂度分析

令 m 为 words 长度，w 为 word 的平均长度。n 为 puzzles 的长度，p 为 puzzle 的平均长度。

- 时间复杂度：$O(m*w* + *n*2^p)$
- 空间复杂度：$O(m)$

字典树

思路

看了官方的解答还提供了字典树的解法。于是我也用字典树实现了一遍。

之所以可以使用字典树求解是因为我们只关心：

- word 是否是 puzzle 的子集
- 如果是，则关心 word 出现的次数

但由于类似：words: ["abc", "acb", "bac"] 等的存在，使得判断的时间大大增加，如果进行一次排序，此时 words 为：["abc", "abc", "abc"]，而如果统计排序后相同 word 出现的次数，比如：

```
{
    "abc": 3
}
```

这就可以仅判断一次而不是多次了，这就减少了判断。而这**对于我们求的答案来说是等价的**。除此之外，word 中一个字符出现几次对我们来说是一样的。比如 words: ["abc", "aaaaabbbc"] 可以看成是 ["abc", "abc"] 这**对于我们求的答案来说是等价的**。

因此我们可以将其进行一次**排序并去重**。同理，我们需要对 puzzle 进行排序并去重。而由于题目规定了 puzzle 本身不含重复字符，因此只对 puzzle 进行排序也是可以的。

这种做法同样需要枚举 puzzle 的子集。伪代码：

```
def get_subset(puzzle, pos):
  # ...
  get_subset(next_with_puzzle_pos , pos + 1) # 选 pos
  get_subset(next_without_puzzle_pos, pos + 1) # 不选 pos
  # ...
```

由于第一个必选，因此上面的逻辑需要做一些微调，具体看下方代码区。

关键点

- 字典树的基本用法
- 递归枚举子集

代码

- 语言支持：Python3

Python3 Code:

```
class TrieNode:
    def __init__(self):
        self.count = 0
        self.children = {}


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        cur = self.root
        for c in word:
            if c not in cur.children:
                cur.children[c] = TrieNode()
            cur = cur.children[c]
        cur.count += 1


class Solution:
    def findNumOfValidWords(self, words: List[str], puzzles
        trie = Trie()
        for word in words:
            trie.insert(sorted(set(word)))

        def get_count(first_letter, cur, i, puzzle):
            if i == len(puzzle):
                return cur.count
            if not cur:
                return 0
            ans = 0
            # 这个判断成立的条件是 puzzle 中不存在重复的字符，  这
            if puzzle[i] != first_letter:
                ans += get_count(first_letter, cur, i + 1,
            if puzzle[i] in cur.children:
                ans += get_count(first_letter, cur.children
            return ans
```

复杂度分析

令 m 为 words 长度，w 为 word 的平均长度。n 为 puzzles 的长度，p 为 puzzle 的平均长度。

- 时间复杂度：$O(m*w* + *n*2^p)$
- 空间复杂度：$O(m*w + p)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址(1203. 项目管理)

https://leetcode-cn.com/problems/sort-items-by-groups-respecting-dependencies/

题目描述

公司共有 n 个项目和 m 个小组，每个项目要不无人接手，要不就由 m 个小

group[i] 表示第 i 个项目所属的小组，如果这个项目目前无人接手，那么 g

请你帮忙按要求安排这些项目的进度，并返回排序后的项目列表：

同一小组的项目，排序后在列表中彼此相邻。
项目之间存在一定的依赖关系，我们用一个列表 beforeItems 来表示，其中
如果存在多个解决方案，只需要返回其中任意一个即可。如果没有合适的解决方

示例 1：

Item	Group	Before
0	-1	
1	-1	6
2	1	5
3	0	6
4	0	3, 6
5	1	
6	0	
7	-1	

```
输入：n = 8, m = 2, group = [-1,-1,1,0,0,1,0,-1], beforeIter
输出：[6,3,4,1,5,2,0,7]
示例 2：

输入：n = 8, m = 2, group = [-1,-1,1,0,0,1,0,-1], beforeIter
输出：[]
解释：与示例 1 大致相同，但是在排序后的列表中，4 必须放在 6 的前面。


提示：

1 <= m <= n <= 3 * 104
group.length == beforeItems.length == n
-1 <= group[i] <= m - 1
0 <= beforeItems[i].length <= n - 1
0 <= beforeItems[i][j] <= n - 1
i != beforeItems[i][j]
beforeItems[i] 不含重复元素
```

前置知识

- 图论 - 拓扑排序
- BFS & DFS

公司

- 暂无

思路

首先这道题不简单。 题目隐藏了三个考点，参考了其他题解之后，发现他们思路挺不错的，但讲述的并不清楚，于是写下了这篇题解。

考点一 - 如何确定拓扑排序?

对于拓扑排序，我们可以使用 BFS 和 DFS 两种方式来解决。

使用 BFS 则从入度为 0 的开始（没有任何依赖），将其邻居（依赖）逐步加入队列，并将入度（依赖数目）减去 1，如果减到 0 了，说明没啥依赖了，将其入队处理。这种做法不需要使用 visited 数组，因为环的入度不可能为 0，也就不会入队，自然不会有死循环。

代码：

```
    def tp_sort(self, items, indegree, neighbors):
        q = collections.deque([])
        ans = []
        for item in items:
            if not indegree[item]:
                q.append(item)
        while q:
            cur = q.popleft()
            ans.append(cur)

            for neighbor in neighbors[cur]:
                indegree[neighbor] -= 1
                if not indegree[neighbor]:
                    q.append(neighbor)

        return ans
```

使用 DFS 可以从图的任意一点出发，基于深度优先遍历检测是否有环。如果有，则返回 []，如果没有，则直接将 path 返回即可。使用此方法需要 visited 数组。

代码：

```python
class Solution:
    def tp_sort(self, items: int, pres: List[List[int]]) ->
        res = []
        visited = [0] * items
        adjacent = [[] for _ in range(items)]

        def dfs(i):
            if visited[i] == 1:
                return False
            if visited[i] == 2:
                return True
            visited[i] = 1
            for j in adjacent[i]:
                if not dfs(j):
                    return False

            visited[i] = 2
            res.append(i)
            return True
        for cur, pre in pres:
            adjacent[cur].append(pre)
        for i in range(items):
            if not dfs(i):
                return []
        return res
```

相关题目：

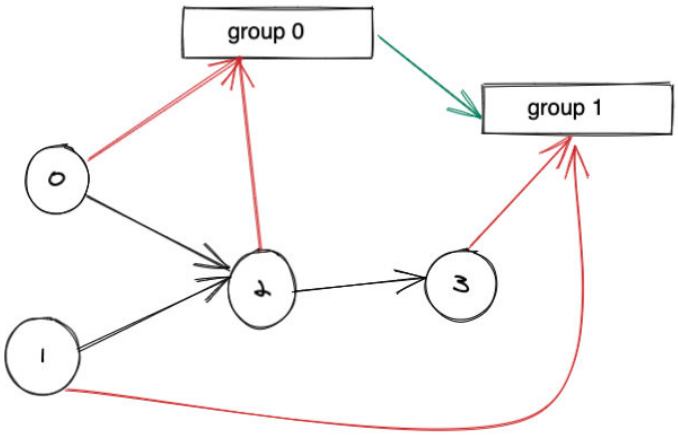
- 210. 课程表 II
- 207. 课程表

考点二 - 如何确定项目的依赖关系?

如下图：

- 圆圈表示的是项目
- 黑色线条表示项目的依赖关系
- 红色线条表示项目和组之间的依赖关系
- 绿色线条是项目之间的依赖关系

注意绿色线条不是题目给出的，而是需要我们自己生成。



生成绿色部分依赖关系的核心逻辑是**如果一个项目和这个项目的依赖（如果存在）需要不同的组来完成**，那么这两个组就拥有依赖关系。代码：

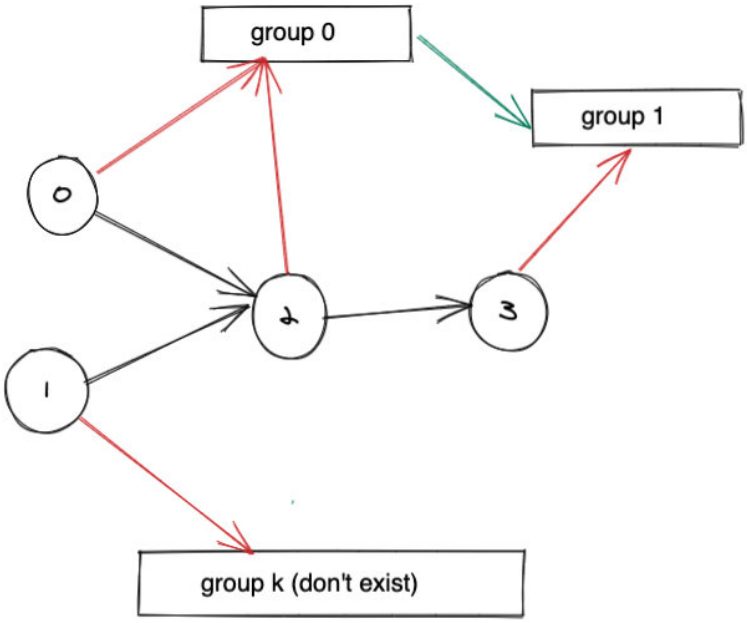
```
for pre in pres[project]:
    if group[pre] != group[project]:
        # 小组关系图
        group_indegree[group[project]] += 1
        group_neighbors[group[pre]].append(group[project])
    else:
        # 项目关系图
        # ...
```

pres 是题目中的 beforeItems，即项目的依赖关系。

考点三 - 无人负责的项目如何处理?

如果无组处理，意味着随便找一个组分配即可，这意味着其是图中入度为零的点。

一种方法是将这些无人处理的进行编号，只要给分别给它们一个不重复的 id 即可，注意这个 id 一定不能是已经存在的 id。由于原有的 group id 范围是 [0, m-1] 因此我们可以从 m 开始并逐个自增 1 来实现，详见代码。



代码

代码支持：Python3

Python3：

```
class Solution:
    def tp_sort(self, items, indegree, neighbors):
        q = collections.deque([])
        ans = []
        for item in items:
            if not indegree[item]:
                q.append(item)
        while q:
            cur = q.popleft()
            ans.append(cur)

            for neighbor in neighbors[cur]:
                indegree[neighbor] -= 1
                if not indegree[neighbor]:
                    q.append(neighbor)

        return ans

    def sortItems(self, n: int, m: int, group: List[int], p
        max_group_id = m
        for project in range(n):
            if group[project] == -1:
                group[project] = max_group_id
                max_group_id += 1

        project_indegree = collections.defaultdict(int)
        group_indegree = collections.defaultdict(int)
        project_neighbors = collections.defaultdict(list)
        group_neighbors = collections.defaultdict(list)
        group_projects = collections.defaultdict(list)

        for project in range(n):
            group_projects[group[project]].append(project)

            for pre in pres[project]:
                if group[pre] != group[project]:
                    # 小组关系图
                    group_indegree[group[project]] += 1
                    group_neighbors[group[pre]].append(grou
                else:
                    # 项目关系图
                    project_indegree[project] += 1
                    project_neighbors[pre].append(project)

        ans = []

        group_queue = self.tp_sort([i for i in range(max_gr
```

```
        if len(group_queue) != max_group_id:
            return []

        for group_id in group_queue:

            project_queue = self.tp_sort(group_projects[gr

            if len(project_queue) != len(group_projects[gr
                return []
            ans += project_queue

        return ans
```

复杂度分析

令 m 和 n 分别为图的边数和顶点数。

- 时间复杂度：$O(m + n)$
- 空间复杂度：$O(m + n)$

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址（1255. 得分最高的单词集合）

https://leetcode-cn.com/problems/maximum-score-words-formed-by-letters/

题目描述

你将会得到一份单词表 words，一个字母表 letters （可能会有重复字母）

请你帮忙计算玩家在单词拼写游戏中所能获得的「最高得分」：能够由 letter

单词拼写游戏的规则概述如下：

玩家需要用字母表 letters 里的字母来拼写单词表 words 中的单词。
可以只使用字母表 letters 中的部分字母，但是每个字母最多被使用一次。
单词表 words 中每个单词只能计分（使用）一次。
根据字母得分情况表score，字母 'a', 'b', 'c', ... , 'z' 对应的得分
本场游戏的「得分」是指：玩家所拼写出的单词集合里包含的所有字母的得分之和

示例 1：

输入：words = ["dog","cat","dad","good"], letters = ["a","a"
输出：23
解释：
字母得分为 a=1, c=9, d=5, g=3, o=2
使用给定的字母表 letters，我们可以拼写单词 "dad" (5+1+5)和 "good"
而单词 "dad" 和 "dog" 只能得到 21 分。
示例 2：

输入：words = ["xxxz","ax","bx","cx"], letters = ["z","a","b
输出：27
解释：
字母得分为 a=4, b=4, c=4, x=5, z=10
使用给定的字母表 letters，我们可以组成单词 "ax" (4+5)， "bx" (4+5
单词 "xxxz" 的得分仅为 25 。
示例 3：

输入：words = ["leetcode"], letters = ["l","e","t","c","o","
输出：0
解释：
字母 "e" 在字母表 letters 中只出现了一次，所以无法组成单词表 words

提示：

1 <= words.length <= 14
1 <= words[i].length <= 15
1 <= letters.length <= 100
letters[i].length == 1
score.length == 26
0 <= score[i] <= 10
words[i] 和 letters[i] 只包含小写的英文字母。

前置知识

- 回溯

公司

- 暂无

思路

题目的本质就是枚举所有的 words 组合，然后判断是否可以满足单词拼写的游戏规则，最后找出所有满足条件的最大分数即可。因此这道题可以用到 78. 子集 的代码。

由排列组合原理可知， 一个大小为 N 的集合的组合数是 2^N，因此这种解法的时间复杂度也大致是这个量级。

这道题比78. 子集 稍微复杂一点，不管是题目的数据输入还是限制条件都更复杂。

实际上， 这些限制条件影响的只是部分细节，我们仍然套用回溯的模板即可， 关于回溯模板可参考：回溯专题。

核心伪代码如下：

```
class Solution:
    def maxScoreWords(self, words, letters, score):
        ans = 0

        def dfs(start, 当前的分数, counter):
            if start > len(words): return
            ans = max(ans, cur)
            for j in 循环start之后的单词:
                if 如果当前单词加进去还满足游戏规则:
                    dfs(j + 1, 新的分数, 新的counter)

        dfs(0, 0, collections.Counter(letters))
        return ans
```

> 由于每次都新生成一个 counter，因此状态不需要回溯。

其中 collections.Counter(letters) 的功能是计数，比如['a', 'a', 'c', 'b']，会被处理为 { a: 2, b: 1, c: 1}。其功能是用于判断**当前单词加进去是否还满足游戏规则**。具体可以参考下方的代码区。

关键点

- 回溯模板
- 计数

代码

代码支持 Python3:

Python3 Code:

```
class Solution:
    def maxScoreWords(self, words, letters, score):
        self.ans = 0
        words_score = [sum(score[ord(c)-ord('a')] for c in
        words_counter = [collections.Counter(word) for wor

        def backtrack(start, cur, counter):
            if start > len(words):
                return
            self.ans = max(self.ans, cur)
            for j, w_counter in enumerate(words_counter[sta
                if all(n <= counter.get(c,0) for c,n in w_
                    backtrack(j+1, cur+words_score[j], cou

        backtrack(0, 0, collections.Counter(letters))
        return self.ans
```

复杂度分析

- 时间复杂度：$O(2^N)$，其中 N 为 words 的个数。
- 空间复杂度：$O(total)$，其中 total 为 words 中的字符总数。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大

家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(1345. 跳跃游戏 IV)

https://leetcode-cn.com/problems/jump-game-iv/

题目描述

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
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

前置知识

- BFS

思路

求最少的题目，考虑动态规划，贪心 和 BFS。

这道题没有想到贪心的做法，于是考虑到了动态规划。不过由于**arr[i] == arr[j] 且 i != j**也可以转移，因此这里涉及到了一个**连通性变更**的问题，代码会比较难写。

于是继续考虑 BFS。BFS 解题需要考虑三点：

- 初始点。 这里是 0
- 终点。这里是 n - 1，其中 n 为数组长度。
- 节点状态转移。这里是题目列举的三种情况。前两个非常简单，最后一个只需要建立一个 hashtable，将相同值的索引合并到一个 list 即可。具体请看代码。

这里我直接使用 BFS 的模板提交了，结果超时了。用例卡在了 [7,7,7,7,7,7.....] 无数个 7 上。

如果使用标准模板的 BFS，那么每一个 7 都会遍历到其他的所有 7，算法在这种情况下时间复杂度会退化到 $O(N^2)$。其实这里有一个上面讲的**连通性**的问题。如果 7 的 steps 求出来是 x，那么所有的 7 都是 x（不会比 7 大，也不会比 7 小），没有必要继续找了。因此一个剪枝就是遍历到 7 之后就将同值从 hashtable 中都清空。这个剪枝可将时间复杂度直接从 N^2 降低到 $O(N)$。

代码

代码支持： Python3

```python
class Solution:
    def minJumps(self, A: List[int]) -> int:
        dic = collections.defaultdict(list)
        n = len(A)

        for i, a in enumerate(A):
            dic[a].append(i)
        visited = set([0])
        q = collections.deque([0])
        steps = 0

        while q:
            for _ in range(len(q)):
                i = q.popleft()
                visited.add(i)
                if i == n - 1: return steps
                for neibor in dic[A[i]] + [i - 1, i + 1]:
                    if 0 <= neibor < n and neibor not in v
                        q.append(neibor)
                # 剪枝
                dic[A[i]] = []
            steps += 1
        return -1
```

复杂度分析

- 时间复杂度：$O(N)$，其中 N 为数组长度。
- 空间复杂度：$O(N)$，其中 N 为数组长度。

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

大家也可以关注我的公众号《力扣加加》获取更多更新鲜的 LeetCode 题解

题目地址（1449. 数位成本和为目标值的最大数字）

https://leetcode-cn.com/problems/form-largest-integer-with-digits-that-add-up-to-target/

题目描述

给你一个整数数组 cost 和一个整数 target 。请你返回满足如下规则可以得

给当前结果添加一个数位（i + 1）的成本为 cost[i] （cost 数组下标从 0
总成本必须恰好等于 target 。
添加的数位中没有数字 0 。
由于答案可能会很大，请你以字符串形式返回。

如果按照上述要求无法得到任何整数，请你返回 "0" 。

示例 1:

输入：cost = [4,3,2,5,6,7,2,5,5], target = 9
输出："7772"
解释：添加数位 '7' 的成本为 2 ，添加数位 '2' 的成本为 3 。所以 "77
 数字   成本
  1 ->   4
  2 ->   3
  3 ->   2
  4 ->   5
  5 ->   6
  6 ->   7
  7 ->   2
  8 ->   5
  9 ->   5
示例 2:

输入：cost = [7,6,5,5,5,6,8,7,8], target = 12
输出："85"
解释：添加数位 '8' 的成本是 7 ，添加数位 '5' 的成本是 5 。"85" 的成
示例 3:

输入：cost = [2,4,6,2,4,6,4,4,4], target = 5
输出："0"
解释：总成本是 target 的条件下，无法生成任何整数。
示例 4:

输入：cost = [6,10,15,40,40,40,40,40,40], target = 47
输出："32211"

提示:

cost.length == 9
1 <= cost[i] <= 5000
1 <= target <= 5000

前置知识

- 数组
- 动态规划
- 背包问题

公司

- 暂无

思路

由于数组可以重复选择，因此这是一个完全背包问题。

01 背包

对于 01 背包问题，我们的套路是：

```
for i in 0 to N:
    for j in 1 to V + 1:
        dp[j] = max(dp[j], dp[j - cost[i])
```

而一般我们为了处理边界问题，我们一般会这么写代码：

```
for i in 1 to N + 1:
    # 这里是倒序的，原因在于这里是01背包。
    for j in V to 0:
        dp[j] = max(dp[j], dp[j - cost[i - 1])
```

其中 dp[j] 表示只能选择前 i 个物品，背包容量为 j 的情况下，能够获得的最大价值。

> dp[j] 不是没 i 么？ 其实我这里 i 指的是 dp[j]当前所处的循环中的 i 值

完全背包问题

回到问题，我们这是完全背包问题：

```
for i in 1 to N + 1:
    # 这里不是倒序，原因是我们这里是完全背包问题
    for j in 1 to V + 1:
        dp[j] = max(dp[j], dp[j - cost[i - 1])
```

为什么 01 背包需要倒序，而完全背包则不可以

实际上，这是一个骚操作，我来详细给你讲一下。

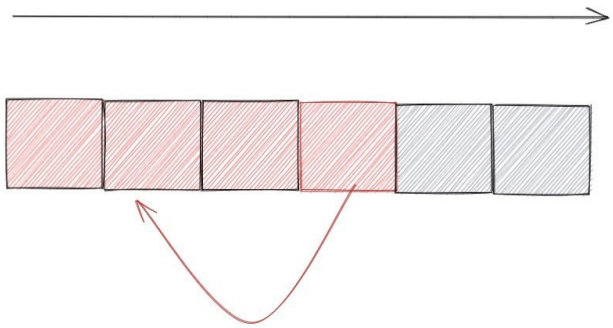
其实要回答这个问题，我要先将 01 背包和完全背包退化二维的情况。

对于 01 背包：

```
for i in 1 to N + 1:
    for j in V to 0:
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cost[i -
```

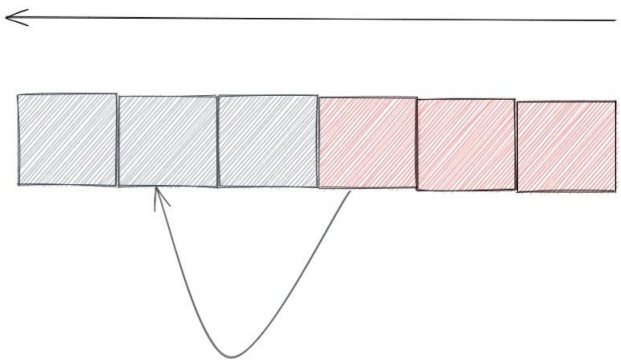
注意等号左边是 i，右边是 i - 1，这很好理解，因为 i 只能取一次嘛。

那么如果我们不降序遍历会怎么样呢？



如图橙色部分表示已经遍历的部分，而让我们去用[j - cost[i - 1]] 往前面回溯的时候，实际上回溯的是 dp[i]j - cost[i - 1]]，而不是 dp[i - 1]j - cost[i - 1]]。

如果是降序就可以了，如图：



这个明白的话，我们继续思考为什么完全背包就要不降序了呢？

我们还是像上面一样写出二维的代码：

```
for i in 1 to N + 1:
   for j in 1 to V + 1:
       dp[i][j] = max(dp[i - 1][j], dp[i][j - cost[i - 1])
```

由于 i 可以取无数次，那么正序遍历正好可以满足，如上图。

恰好装满 VS 可以不装满

题目有两种可能，一种是要求背包恰好装满，一种是可以不装满（只要不超过容量就行）。而本题是要求 `恰好装满` 的。而这两种情况仅仅影响我们 `dp数组初始化` 。

- 恰好装满。只需要初始化 dp[0] 为 0， 其他初始化为负数即可。
- 可以不装满。 只需要全部初始化为 0，即可，

原因很简单，我多次强调过 dp 数组本质上是记录了一个个自问题。 dp[0]是一个子问题，dp[1]是一个子问题。。。

有了上面的知识就不难理解了。 初始化的时候，我们还没有进行任何选择，那么也就是说 dp[0] = 0，因为我们可以通过什么都不选达到最大值 0。而 dp[1],dp[2]...则在当前什么都不选的情况下无法达成，也就是无解，因为为了区分，我们可以用负数来表示，当然你可以用任何可以区分的东西表示，比如 None。

回到本题

而这道题和普通的完全背包不一样，这个是选择一个组成的最大数。由小学数学知识 `一个数字的全排列中，按照数字降序排列是最大的` ，我这里用了一个骚操作，那就是 cost 从后往前遍历，因为后面表示的数字大。

代码

```
class Solution:
    def largestNumber(self, cost: List[int], target: int) -
        dp = [0] + [float('-inf')] * target
        for i in range(9, 0, -1):
            for j in range(1, target+1):
                if j >= cost[i - 1]:
                    dp[j] = max(dp[j], (dp[j-cost[i - 1]] >
        return str(dp[target]) if dp[target] > 0 else '0'
```

复杂度分析

- 时间复杂度：$O(target))$
- 空间复杂度：$O(target)$

扩展

最后贴几个我写过的背包问题，让大家看看历史是多么的相似。

(322. 硬币找零(完全背包问题))

> 这里内外循环和本题正好是反的，我只是为了"秀技"(好玩)，实际上在这里对答案并不影响。

Python Code:

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount + 1)
        dp[0] = 1

        for j in range(len(coins)):
            for i in range(1, amount + 1):
                if i >= coins[j]:
                    dp[i] += dp[i - coins[j]]

        return dp[-1]
```

(518. 零钱兑换 II)

> 这里内外循环和本题正好是反的，但是这里必须这么做，否则结果是不对的，具体可以点进去链接看我那个题解

所以这两层循环的位置起的实际作用是什么？ 代表的含义有什么不同?

本质上:

```
for i in 1 to N + 1:
    for j in V to 0:
        ...
```

这种情况选择物品 1 和物品 3（随便举的例子），是一种方式。选择物品 3 个物品 1（注意是有顺序的）是同一种方式。 **原因在于你是固定物品，去扫描容量**。

而:

```
for j in V to 0:
    for i in 1 to N + 1:
        ...
```

这种情况选择物品 1 和物品 3（随便举的例子），是一种方式。选择物品 3 个物品 1（注意是有顺序的）也是一种方式。**原因在于你是固定容量，去扫描物品**。

因此总的来说，如果你认为[1,3]和[3,1]是一种，那么就用方法 1 的遍历，否则用方法 2。

大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。 大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

题目地址(1494. 并行课程 II)

https://leetcode-cn.com/problems/parallel-courses-ii/

题目描述

给你一个整数 n 表示某所大学里课程的数目，编号为 1 到 n ，数组 depen

在一个学期中，你 最多 可以同时上 k 门课，前提是这些课的先修课在之前的

请你返回上完所有课最少需要多少个学期。题目保证一定存在一种上完所有课的

示例 1：

输入：n = 4, dependencies = [[2,1],[3,1],[1,4]], k = 2
输出：3
解释：上图展示了题目输入的图。在第一个学期中，我们可以上课程 2 和课程

示例 2：

输入：n = 5, dependencies = [[2,1],[3,1],[4,1],[1,5]], k = 2
输出：4
解释：上图展示了题目输入的图。一个最优方案是：第一学期上课程 2 和 3，

示例 3：

输入：n = 11, dependencies = [], k = 2
输出：6

提示：

1 <= n <= 15
1 <= k <= n
0 <= dependencies.length <= n * (n-1) / 2
dependencies[i].length == 2
1 <= xi, yi <= n
xi != yi
所有先修关系都是不同的，也就是说 dependencies[i] != dependencies
题目输入的图是个有向无环图。

前置知识

- 拓扑排序
- 位运算
- 动态规划

公司

- 暂无

思路

看了下 n 的取值范围是 [1, 15]，基本可以锁定为回溯或者状压 DP。

> 一般 20 以内都可以，具体的时间复杂度和数据规模的关系可从我的网站中的复杂度速查表中查看。

这道题是状压 DP，如果你对其不了解，可以看下我之前写的 状压 DP 是什么？这篇题解带你入门，这里一些细节不再赘述。

首先，我们需要用一个数据结构来存储课程之间的依赖关系。不妨使用 hashmap，这样可以在 $O(1)$ 的时间获取到一个课程的所有前置课程。

接下来，我们使用一个数组 studied 来表示已经学习的课程，其中 studied[i] 是一个布尔值，表示第 i 个课程是否已经学习。

假设我们的 studied 已经确认了，那么下一步我们可以继续学习哪些课程呢？这就需要用到前面的 hashmap 啦，不妨称其为 neighbors，neighbors 的 key 是课程 id，值是 studied 数组，含义是想学习课程 i 必须先把 studied 数组中为 true 的全部学了。那么如果 neighbors[j] 是当前已经学习的课程数组的子集，那么说明当前已经达到了学习课程 j 的条件。

我们可以不断枚举**当前已经学习的课程数组** 的值，并由此确定接下来可以学习的课程集合 sub。得到 sub 之后要做的就是枚举 sub 的子集啦。

如何枚举子集呢?

比如需要枚举一个集合 S 的所有子集，你会如何做?

1. 状态。我们可以用一个和 S 相同大小的数组 picked 记录每个数被选取的信息， 用 0 表示没有选取，用 1 表示选取。

比如 S 大小为 3， picked 数组 [1,1,0]，表示 S 中的第一项和第二项被选择（索引从 1 开始）。如果 S 的大小为 n，就需要用一个长度为 n 的数组来存储，那么就有 2^n 种状态。

由于数组的值**不是 0 就是 1，满足二值性**，因此更多时候我们会使用**一个数字** y 来表示状态，而不是上面的 picked 数组。其中 y 的**二进制位**对应上面提到的 picked 数组中的一项。

1. 不重不漏。

实际上，我们也可用另外一个数 x 来模拟集合 S。这样问题就转化为两个数（x 和 y）的位运算。

由于我们使用 1 表示被选取， 0 表示选取。因此 如果 x 对应位为 0，其实 y 也只能是 0，而如果 x 对应位是 1，y 却可能是 0 或者 1。也就是说 y 一定小于等于 x， 因此可以枚举所有小于等于 x 的数的二进制，并逐个**判断其是否真的是 x 的子集**。

令 n 为 x 的二进制位数，我们可以写出如下代码。

```
// 外层枚举所有小于等于 x 的数
ans = [];
for (i = 1; i < 1 << n; i++) {
  if ((x | i) === x) ans.push(i);
}
// ans 就是所有非空子集
```

这种算法的复杂度大约是 $O(4^n)$，也就是说和 x 成正比。这种算法 n 最多取到 12 左右。

这样做不重不漏么？答案是可以的。因为 (x | i) === x 就是 i 是 x 的子集的充要条件，当然你也可用 & ，即 (x | i) & i == i 来表示 i 是 x 的子集。

如果二进制你不好理解，其实你可以转化为十进制理解。比如我给你一个数 132，让你找 132 的子集，这里的子集我简单定义为当前位的数字是否小于等于原数字当前位的数字。这样我们就可以先从 1 枚举到 132，因为这些数潜在都可能是 132 的子集。如果我枚举了一个数字 030，由于 0 小于等于 1，3 小于等于 3，0 小于等于 2，因此 030 是 132 的子集。而如果我枚举了一个数字 040，由于 4 大于 3，因此 040 不是 132 的子集。

1. 效率。

上面的枚举方法虽然也可保证不重不漏，但是却不是最优的，这里介绍一种更好的枚举方法。

具体做法就是x_i和 x 进行&（与）运算。与运算可以**快速跳到下一个子集**。

```
ans = [];
// 外层枚举所有小于等于 x 的数
for (i = x; i != 0; i = (i - 1) & x) {
  ans.push(i);
}
// ans 就是所有非空子集
```

这样做不重不漏么？算法的关键在于 `i = (i - 1) & x` 。这个操作首先将 i - 1，从而把 i 最右边的 1 变成了 0，然后把这位之后的所有 0 变成了 1。经过这样的处理再与 x 求与，就保证了得到的结果是 x 的子集，并

且一定是所有子集中小于 i 的最大的一个。直观来看就是倒序枚举除了所有非空子集。

对于有 n 个 1 的二进制数字，需要 2^n 的时间复杂度。而有 n 个 1 的二进制数字有 $C(n,i)$ 个，所以这段代码的时间复杂度为 $\sum_{i=0}^{n} C(n,i)\times2^i$，大约是 $O(3^n)$。和上面一样，这种算法的时间复杂度也和 x 成正比。但是这种算法 n 最多取到 15 左右。

这种方法对题目有一定要求， 即：

1. 数据范围要合适，否则数字无法表示了。
2. 只能有两种状态，这样才可以用二进制位 0 和 1 进行模拟。

其实状态压缩没有什么神秘，只是 API 不一样罢了。

有了上面的铺垫就简单了。我们只需要枚举所有子集，对于每一个子集，我们考虑使用动态规划来转移状态。

```
dp[i | sub] = min(dp[i | sub], dp[i] + 1)
```

其中 i 为当前的学习的课程，sub 为当前可以学习的课程的子集。其中 i 和 sub 都是一个数字，每一位的 bit 为 0 表示无该课程，为 1 表示有该课程。

关键点

- 枚举
- 位运算的枚举子集优化

代码

- 语言支持：Python3

Python3 Code:

```
class Solution:
    def minNumberOfSemesters(self, n: int, dependencies: Li
        neighbors = collections.defaultdict(int)
        dp = [n] * (1 << n)

        for fr, to in dependencies:
            neighbors[to - 1] |= 1 << (fr - 1)
        dp[0] = 0  # 启动 dp
        for i in range(1 << n):
            can = 0
            for j in range(n):
                if (i & neighbors[j]) == neighbors[j]:
                    can |= 1 << j
            # 已经学过的不能学
            can &= ~i
            sub = can
            while sub:
                if bin(sub).count("1") <= k:
                    dp[i | sub] = min(dp[i | sub], dp[i] +
                sub = (sub - 1) & can
        return dp[(1 << n) - 1]
```

复杂度分析

令 n 为数组长度。

- 时间复杂度：$O(2^n)$
- 空间复杂度：$O(2^n)$

> 此题解由 力扣刷题插件 自动生成。

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 40K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（1521. 找到最接近目标值的函数值）

https://leetcode-cn.com/problems/find-a-value-of-a-mysterious-function-closest-to-target/

题目描述

```
func(arr, l, r) {
    if (r < l) {
        return -1000000000
    }
    ans = arr[l]
    for (i = l + 1; i <= r; i++) {
        ans = ans & arr[i]
    }
    return ans
}
```

```
Winston 构造了一个如上所示的函数 func 。他有一个整数数组 arr 和一个

请你返回 |func(arr, l, r) - target| 的最小值。

请注意， func 的输入参数 l 和 r 需要满足 0 <= l, r < arr.length



示例 1:

输入: arr = [9,12,3,7,15], target = 5
输出: 2
解释: 所有可能的 [l,r] 数对包括 [[0,0],[1,1],[2,2],[3,3],[4,4],
示例 2:

输入: arr = [1000000,1000000,1000000], target = 1
输出: 999999
解释: Winston 输入函数的所有可能 [l,r] 数对得到的函数值都为 100000
示例 3:

输入: arr = [1,2,4,8,16], target = 0
输出: 0


提示:

1 <= arr.length <= 10^5
1 <= arr[i] <= 10^6
0 <= target <= 10^7
```

前置知识

- 位运算
- 动态规划

公司

- 暂无

思路

首先我们要知道一个前提，那就是对于一个数组 arr，对 arr 的子数组 arr[l:r] 进行与操作满足单调性，也就是说 arr[l:r] >= arr[l+1:r] >= arr[l+2:r] ... 。其中的依据是**一个数异或另外一个数的结果一定不会比这两个数大**。

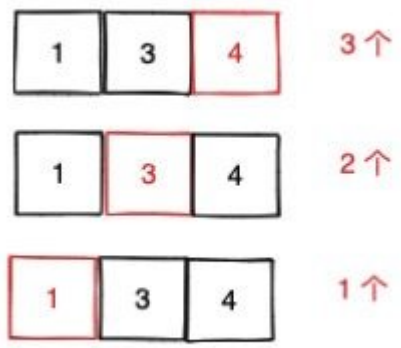
为了更好的理解本题。我们可以从一个简单的例子入手。

题目描述：

> 如果让你求一个数组的连续子数组总个数，你会如何求？其中连续指的是数组的

一种思路是总的连续子数组个数等于：以索引为 0 结尾的子数组个数 + 以索引为 1 结尾的子数组个数 + … + 以索引为 n - 1 结尾的子数组个数，这无疑是完备的。

> 关于这点不熟悉的，也可以看下我的 【西法带你学算法】一次搞定前缀和



我们也可以采用同样的思路进行枚举。

总的连续子数组按位与操作等于：以索引为 0 结尾的子数组按位与操作的结果, 以索引为 1 结尾的子数组按位与操作的结果 + … + 以索引为 n - 1 结尾的子数组按位与操作的结果，这无疑是完备的。

而题目我求的不是总和，而是与 target 最接近的，不过这不难，使用一个全局变量记录即可。

以索引为 i 结尾的子数组按位与操作的结果 sub[i] 其实就等于 `sub[i-1] & A[i]` 。这提示我们使用滚动数组来完成。而由于重复数字是没有意义的，因此可使用 hashset 来优化，而不是普通的数组，这样可以同时降低时间和空间复杂度。

> 关于滚动数组可以参考我之前写的动态规划章节

这种解法其实就是暴力求解，和动态规划没有啥区别。

关键点

- 识别出函数 func 满足某种单调性
- 采用合适的枚举方法

代码

代码支持：Python3

```
class Solution:
    def closestToTarget(self, A: List[int], target: int) ->
        seen = set()
        ans = float('inf')
        for a in A:
            seen.add(a)
            t = set()
            # 类似滚动数组 此时的 seen 相当于 sub[i-1]
            for b in seen:
                yu = a & b
                ans = min(ans, abs(yu - target))
                t.add(yu)
            # 此时的 t 就是 sub[i]，我们需要更新回 seen
            seen = t
        return ans
```

复杂度分析 令 N 为数组长度，C 为 seen 的大小。C 的大小和数据范围有关，在这里 C 不会超过 32。

- 时间复杂度：$O(N*C)$
- 空间复杂度：$O(C)$

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

题目地址（1526. 形成目标数组的子数组最少增加次数）

https://leetcode-cn.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array/

题目描述

```
给你一个整数数组 target 和一个数组 initial ，initial 数组与 targ

请你返回从 initial 得到  target 的最少操作次数，每次操作需遵循以下

在 initial 中选择 任意 子数组，并将子数组中每个元素增加 1 。
答案保证在 32 位有符号整数以内。



示例 1:

输入：target = [1,2,3,2,1]
输出：3
解释：我们需要至少 3 次操作从 intial 数组得到 target 数组。
[0,0,0,0,0] 将下标为 0 到 4 的元素（包含二者）加 1 。
[1,1,1,1,1] 将下标为 1 到 3 的元素（包含二者）加 1 。
[1,2,2,2,1] 将下表为 2 的元素增加 1 。
[1,2,3,2,1] 得到了目标数组。
示例 2:

输入：target = [3,1,1,2]
输出：4
解释：(initial)[0,0,0,0] -> [1,1,1,1] -> [1,1,1,2] -> [2,1,1
示例 3:

输入：target = [3,1,5,4,2]
输出：7
解释：(initial)[0,0,0,0,0] -> [1,1,1,1,1] -> [2,1,1,1,1] ->
                                    -> [3,1,2,2,2] -> [3,1,3,
示例 4:

输入：target = [1,1,1,1]
输出：1


提示：

1 <= target.length <= 10^5
1 <= target[i] <= 10^5
```

前置知识

- 差分与前缀和

公司

- 暂无

思路

首先我们要有前缀和以及差分的知识。这里简单讲述一下：

- 前缀和 pres：对于一个数组 A [1,2,3,4]，它的前缀和就是 [1,1+2,1+2+3,1+2+3+4]，也就是 [1,3,6,10]，也就是说前缀和 $pres[i] =\sum_{n=0}^{n=i}A[i]$
- 差分数组 d：对于一个数组 A [1,2,3,4]，它的差分数组就是 [1,2-1,3-2,4-3]，也就是 [1,1,1,1]，也就是说差分数组 $d[i] = A[i] - Ai-1$，$d[i] = Ai$

前缀和与差分数组互为逆运算。如何理解呢？这里的原因在于你对 A 的差分数组 d 求前缀和就是数组 A。前缀和对于求区间和有重大意义。而差分数组通常用于**先对数组的若干区间执行若干次增加或者减少操作**。仔细看这道题不就是**对数组若干区间执行 n 次增加操作**，让你返回从一个数组到另外一个数组的最少操作次数么？差分数组对两个数字的操作等价于原始数组区间操作，这样时间复杂度大大降低 O(N) -> O(1)。

题目要求**返回从 initial 得到 target 的最少操作次数**。这道题我们可以逆向思考**返回从 target 得到 initial 的最少操作次数**。

这有什么区别么？对问题求解有什么帮助？由于 initial 是全为 0 的数组，如果将其作为最终搜索状态则不需要对状态进行额外的判断。这句话可能比较难以理解，我举个例子你就懂了。比如我不反向思考，那么初始状态就是 initial ，最终搜索状态自然是 target ，假如我们现在搜索到一个状态 state.我们需要**逐个判断 state[i] 是否等于 target[i]**，如果全部都相等则说明搜索到了 target ，否则没有搜索到，我们继续搜索。而如果我们从 target 开始搜，最终状态就是 initial，我们只需要判断每一位是否都是 0 就好了。 这算是搜索问题的常用套路。

上面讲到了对差分数组求前缀和可以还原原数组，这是差分数组的性质决定的。这里还有一个特点是**如果差分数组是全 0 数组，比如[0, 0, 0, 0]，那么原数组也是[0, 0, 0, 0]**。因此将 target 的差分数组 d 变更为 全为 0 的数组就等价于 target 变更为 initaial。

如何将 target 变更为 initaial?

由于我们是反向操作，也就是说我们可执行的操作是 **-1**，反映在差分数组上就是在 d 的左端点 -1，右端点（可选）+1。如果没有对应的右端点+1 也是可以的。这相当于给原始数组的 [i,n-1] +1，其中 n 为 A 的长度。

如下是一种将 [3, -2, 0, 1] 变更为 [0, 0, 0, 0] 的可能序列。

```
[3, -2, 0, 1] -> [**2**, **-1**, 0, 1] -> [**1**, **0**, 0,
```

可以看出，上面需要进行四次区间操作，因此我们需要返回 4。

至此，我们的算法就比较明了了。

具体算法：

- 对 A 计算差分数组 d
- 遍历差分数组 d，对 d 中 大于 0 的求和。该和就是答案。

```
class Solution:
    def minNumberOperations(self, A: List[int]) -> int:
        d = [A[0]]
        ans = 0

        for i in range(1, len(A)):
            d.append(A[i] - A[i-1])
        for a in d:
            ans += max(0, a)
        return ans
```

复杂度分析 令 N 为数组长度。

- 时间复杂度：$O(N)$
- 空间复杂度：$O(N)$

实际上，我们没有必要真实地计算差分数组 d，而是边遍历边求，也不需要对 d 进行存储。具体见下方代码区。

关键点

- 逆向思考
- 使用差分减少时间复杂度

代码

代码支持：Python3

```
class Solution:
    def minNumberOperations(self, A: List[int]) -> int:
        ans = A[0]
        for i in range(1, len(A)):
            ans += max(0, A[i] - A[i-1])
        return ans
```

复杂度分析 令 N 为数组长度。

- 时间复杂度：$O(N)$

- 空间复杂度：$O(1)$

扩展

如果题目改为：给你一个数组 nums，以及 size 和 K。 其中 size 指的是你不能对区间大小为 size 的子数组执行+1 操作，而不是上面题目的**任意**子数组。K 指的是你只能进行 K 次 +1 操作，而不是上面题目的任意次。题目让你求的是**经过这样的 k 次+1 操作，数组 nums 的最小值最大可以达到多少**。

比如：

```
输入：
nums = [1, 4, 1, 1, 6]
size = 3
k = 2

解释：
将 [1, 4, 1] +1 得到 [2, 5, 2, 1, 6] ，对 [5, 2, 1] +1 得到 [
```

解决问题的关键有两点:

- 定义函数 possible(target)，其功能是**在 K 步之内，每次都只能对 size 大小的子数组+1，是否可以满足数组的最小值>=target**。
- 有了上面的铺垫。我们要找的其实就是 possible(target)为 true 的最大的 target。

这里有个关键点，那就是

- 如果 possible(target)为 true。那么 target 以下的都不用看了，肯定都满足。
- 如果 possible(target)为 false。那么 target 以上的都不用看了，肯定都不满足。

也就是说无论如何我们都能将解空间缩小一半，这提示我们使用二分法。结合前面的知识”我们要找的其实就是满足 possible(target) 的最大的 target“，可知道应该使用**最右二分**，如果对最右二分不熟悉的可以看下二分讲义

参考代码：

```
class Solution:
    def solve(self, A, size, K):
        N = len(A)

        def possible(target):
            # 差分数组 d
            d = [0] * N
            moves = a = 0
            for i in range(N):
                # a 相当于差分数组 d 的前缀和
                a += d[i]
                # 当前值和 target 的差距
                delta = target - (A[i] + a)
                # 大于 0 表示不到 target，我们必须需要进行 +1 操
                if delta > 0:
                    moves += delta
                    # 更新前缀和
                    a += delta
                    # 如果 i + size >= N 对应我上面提到的只修改
                    if i + size < N:
                        d[i + size] -= delta
            # 执行的+1操作小于等于K 说明可行
            return moves <= K
        # 定义解空间
        lo, hi = min(A), max(A) + K
        # 最右二分模板
        while lo <= hi:
            mi = (lo + hi) // 2
            if possible(mi):
                lo = mi + 1
            else:
                hi = mi - 1
        return hi
```

更多题解可以访问我的 LeetCode 题解仓库：
https://github.com/azl397985856/leetcode 。 目前已经 37K star 啦。

关注公众号力扣加加，努力用清晰直白的语言还原解题思路，并且有大量图解，手把手教你识别套路，高效刷题。

欢迎长按关注

题目地址（1649. 通过指令创建有序数组）

https://leetcode-cn.com/problems/create-sorted-array-through-instructions/

题目描述

给你一个整数数组 instructions ，你需要根据 instructions 中的元素

nums 中 严格小于 instructions[i] 的数字数目。
nums 中 严格大于 instructions[i] 的数字数目。
比方说，如果要将 3 插入到 nums = [1,2,3,5] ，那么插入操作的 代价

请你返回将 instructions 中所有元素依次插入 nums 后的 总最小代价 。

示例 1:

输入：instructions = [1,5,6,2]
输出：1
解释：一开始 nums = [] 。
插入 1 ，代价为 min(0, 0) = 0 ，现在 nums = [1] 。
插入 5 ，代价为 min(1, 0) = 0 ，现在 nums = [1,5] 。
插入 6 ，代价为 min(2, 0) = 0 ，现在 nums = [1,5,6] 。
插入 2 ，代价为 min(1, 2) = 1 ，现在 nums = [1,2,5,6] 。
总代价为 0 + 0 + 0 + 1 = 1 。
示例 2:

输入：instructions = [1,2,3,6,5,4]
输出：3
解释：一开始 nums = [] 。
插入 1 ，代价为 min(0, 0) = 0 ，现在 nums = [1] 。
插入 2 ，代价为 min(1, 0) = 0 ，现在 nums = [1,2] 。
插入 3 ，代价为 min(2, 0) = 0 ，现在 nums = [1,2,3] 。
插入 6 ，代价为 min(3, 0) = 0 ，现在 nums = [1,2,3,6] 。
插入 5 ，代价为 min(3, 1) = 1 ，现在 nums = [1,2,3,5,6] 。
插入 4 ，代价为 min(3, 2) = 2 ，现在 nums = [1,2,3,4,5,6] 。
总代价为 0 + 0 + 0 + 0 + 1 + 2 = 3 。
示例 3:

输入：instructions = [1,3,3,3,2,4,2,1,2]
输出：4
解释：一开始 nums = [] 。
插入 1 ，代价为 min(0, 0) = 0 ，现在 nums = [1] 。
插入 3 ，代价为 min(1, 0) = 0 ，现在 nums = [1,3] 。
插入 3 ，代价为 min(1, 0) = 0 ，现在 nums = [1,3,3] 。
插入 3 ，代价为 min(1, 0) = 0 ，现在 nums = [1,3,3,3] 。
插入 2 ，代价为 min(1, 3) = 1 ，现在 nums = [1,2,3,3,3] 。
插入 4 ，代价为 min(5, 0) = 0 ，现在 nums = [1,2,3,3,3,4] 。
插入 2 ，代价为 min(1, 4) = 1 ，现在 nums = [1,2,2,3,3,3,4] 。
插入 1 ，代价为 min(0, 6) = 0 ，现在 nums = [1,1,2,2,3,3,3,4]
插入 2 ，代价为 min(2, 4) = 2 ，现在 nums = [1,1,2,2,2,3,3,3,
总代价为 0 + 0 + 0 + 0 + 1 + 0 + 1 + 0 + 2 = 4 。

```
提示：

1 <= instructions.length <= 105
1 <= instructions[i] <= 105
```

前置知识

- 二分法
- 线段树

公司

- 暂无

二分法

思路

二分法的思路比较简单，直接模拟插入即可。每次只需要保证插入之后还是有序的，这样就可以通过二分查找，计算出**严格大于** 和 **严格小于** x 的数目了。

- 使用 `bisect.bisect_left(nums, instruction)` 可以计算出 instruction 如果插入到 nums ，instruction 在 nums 中的索引是。
- 使用 `bisect.bisect_right(nums, instruction)` 和 bisect_left 类似，只不过对于 nums 已经存在 instruction 了， bisect_left 会尝试插入到其左侧，bisect_right 则会尝试插入到其右侧。

根据 bisect_left 和 bisect_right，我们就可计算出 **严格大于** 和 **严格小于** instruction 的数目了。接下来，我们只需要模拟插入即可。

代码

代码支持：Python3

Python3 Code:

```python
class Solution:
    def createSortedArray(self, instructions: List[int]) ->
        mod = 10 ** 9 + 7
        nums = []
        ans = 0
        # eg: 1 2 2 3
        for instruction in instructions:
            l = bisect.bisect_left(nums, instruction)
            r = bisect.bisect_right(nums, instruction)
            nums[l:l] = [instruction]
            ans = (ans + min(l, len(nums) - r - 1)) % mod
        return ans
```

复杂度分析 令 N 为数组长度。

- 时间复杂度：遍历 instructions 需要 N 次，每次都需要插入数据，由于插入数组的时间复杂度是 $O(N)$。 因此总的时间复杂度为 $O(N^2)$
- 空间复杂度：$O(N)$

需要注意的是，如下代码会超时：

```
nums.insert(l, instruction)
```

也就是说必须使用切片语法才可以：

```
nums[l:l] = [instruction]
```

具体原因大家可以参考这个 stackoverflow 的回答

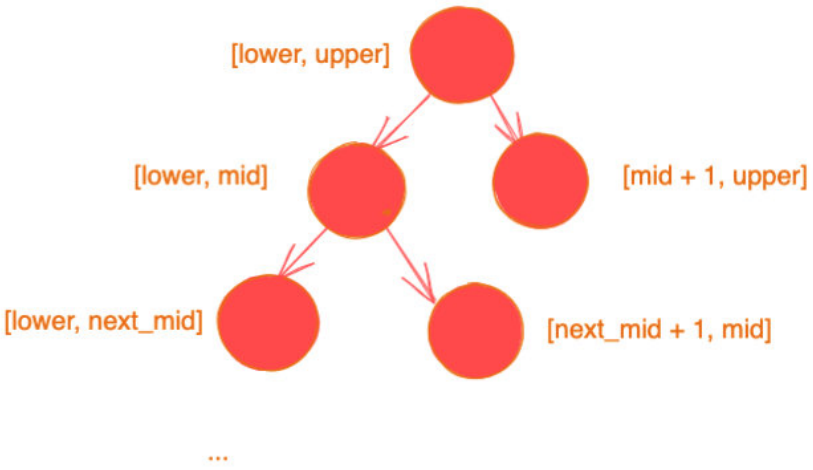
线段树（超时）

思路

这里我直接使用了**计数线段树的模板**。不懂线段树的可以先看下 线段树教程

我们可以维护一个 [lower,upper] 的一个线段树。线段树支持的操作：

- query(l, r): 查询 [l, r] 范围内的数的个数
- update(x): 将 x 更新到线段树



因此我们的目标其实就是 min(query(1, instruction - 1), query(instruction + 1, upper))，其中 upper 为 instructions 的最大树。

核心代码：

```
    upper = max(instructions)
    # 初始化线段树
    seg = SegmentTree(upper, 1)
    for instruction in instructions:
        # 进行两次查询
        l = seg.queryCount(1, instruction - 1)
        r = seg.queryCount(instruction + 1, upper)
        ans = (ans + min(l, r)) % mod
        # 进行一次更新
        seg.updateCount(instruction)
    return ans
```

代码

代码支持：Python3

Python3 Code:

```
class SegmentTree:
    def __init__(self, upper, lower):
        """
        data:传入的数组
        """
        self.lower = lower
        self.upper = upper
        #  申请4倍data长度的空间来存线段树节点
        self.tree = [0] * (4 * (upper - lower + 1))  # 索引

    # 本质就是一个自底向上的更新过程
    # 因此可以使用后序遍历，即在函数返回的时候更新父节点。
    def update(self, tree_index, l, r, index):
        """
        tree_index:某个根节点索引
        l, r : 此根节点代表区间的左右边界
        index : 更新的值的索引
        """
        if l > index or r < index:
            return
        self.tree[tree_index] += 1
        if l == r:
            return
        mid = (l + r) // 2
        left, right = tree_index * 2 + 1, tree_index * 2 +
        self.update(left, l, mid, index)
        self.update(right, mid + 1, r, index)

    def updateCount(self, index: int):
        self.update(0, self.lower, self.upper, index)

    def query(self, tree_index: int, l: int, r: int, ql: i
        """
        递归查询区间[ql,..,qr]的值
        tree_index : 某个根节点的索引
        l, r : 该节点表示的区间的左右边界
        ql, qr: 待查询区间的左右边界
        """
        if qr < l or ql > r:
            return 0
        # l 和 r 在 [ql, qr] 内
        if ql <= l and qr >= r:
            return self.tree[tree_index]
        mid = (l + r) // 2
        left, right = tree_index * 2 + 1, tree_index * 2 +
        return self.query(left, l, mid, ql, qr) + self.que

    def queryCount(self, ql: int, qr: int) -> int:
```

```
        """
        返回区间[ql,..,qr]的计数信息
        """
        return self.query(0, self.lower, self.upper, ql, qr

class Solution:
    def createSortedArray(self, instructions: List[int]) ->
        mod = 10 ** 9 + 7
        ans = 0
        # eg: 1 2 2 3
        upper = max(instructions)
        seg = SegmentTree(upper, 1)
        for instruction in instructions:
            l = seg.queryCount(1, instruction - 1)
            r = seg.queryCount(instruction + 1, upper)
            ans = (ans + min(l, r)) % mod
            seg.updateCount(instruction)
        return ans
```

复杂度分析 令 N 为数组长度。

由于线段树更新和查询的时间复杂度为 $O(log(upper - lower))$，其中 upper 为 instructions 最大值，lower 为 instructions 最小值。由于题目限制了 $1 <= instructions[i] <= 10^5$，因此最坏情况下 upper - lower 为 10^5。

线段树使用了 $4 * (upper - lower + 1)$ 的空间。

- 时间复杂度：$O(Nlog(upper-lower))$
- 空间复杂度：$O(upper- lower)$

题目地址（1671. 得到山形数组的最少删除次数）

https://leetcode-cn.com/problems/minimum-number-of-removals-to-make-mountain-array/

题目描述

```
我们定义 arr 是 山形数组 当且仅当它满足：

arr.length >= 3
存在某个下标 i （从 0 开始） 满足 0 < i < arr.length - 1 且：
arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
arr[i] > arr[i + 1] > ... > arr[arr.length - 1]
给你整数数组 nums ，请你返回将 nums 变成 山形状数组 的 最少 删除次数



示例 1：

输入：nums = [1,3,1]
输出：0
解释：数组本身就是山形数组，所以我们不需要删除任何元素。
示例 2：

输入：nums = [2,1,1,5,6,2,3,1]
输出：3
解释：一种方法是将下标为 0，1 和 5 的元素删除，剩余元素为 [1,5,6,3,
示例 3：

输入：nums = [4,3,2,1,1,2,3,1]
输出：4
提示：

输入：nums = [1,2,3,4,4,3,2,1]
输出：1


提示：

3 <= nums.length <= 1000
1 <= nums[i] <= 109
题目保证 nums 删除一些元素后一定能得到山形数组。
```

前置知识

- 最长上升子序列

思路

看了下数据范围 `3 <= nums.length <= 1000` 。直接莽过没问题。

这道题需要你有最长上升子序列的知识。如果你还不清楚，建议看下我之前写的文章 穿上衣服我就不认识你了？来聊聊最长上升子序列

有了这样的一个知识前提，我们可以枚举所有的山顶。那么

- 左侧需要删除的个数其实就是 L - LIS_LEFT，其中 L 为左侧长度，LIS_LEFT 为左侧的最长上升子序列长度。
- 右侧需要删除的个数其实就是 R - LDS_RIGHT，其中 R 为右侧长度，LDS_RIGHT 为右侧的最长下降子序列长度。

为了将逻辑统一为 **最长上升子序列长度**，我们可以将 R 翻转一次。

枚举山顶的时间复杂度为 $O(N)$，常规的 LIS 复杂度为 $O(N^2)$。

根据时间复杂度速查表：

数据规模	算法可接受时间复杂度
<= 10	O(n!)
<= 20	O(2^n)
<= 100	O(n^4)
<= 500	O(n^3)
<= 2500	O(n^2)
<= 10^6	O(nlogn)
<= 10^7	n

> 时间复杂度速查表可以在我的刷题插件中查到。刷题插件可以在我的公众号《力扣加加》回复插件获取。

本题的数据范围为 <= 1000。因此 N^3 无法通过。不过我们可以使用贪心求 LIS，时间复杂度为 N^2logN，勉强可以通过。关于贪心求解 LIS，上面的文章也有提到。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def minimumMountainRemovals(self, nums: List[int]) -> i
        n = len(nums)
        ans = n
        def LIS(A):
            d = []
            for a in A:
                i = bisect.bisect_left(d, a)
                if i < len(d):
                    d[i] = a
                elif not d or d[-1] < a:
                    d.append(a)
            return d.index(A[-1])

        for i in range(1, n-1):
            l, r = LIS(nums[:i+1]), LIS(nums[i:][::-1])
            if not l or not r: continue
            ans = min(ans, n - 1 - l - r)
        return ans
```

复杂度分析

令 N 为数组长度。

- 时间复杂度：$O(N^2logN)$
- 空间复杂度：$O(N)$

力扣的小伙伴可以关注我，这样就会第一时间收到我的动态啦~

以上就是本文的全部内容了。大家对此有何看法，欢迎给我留言，我有时间都会一一查看回答。更多算法套路可以访问我的 LeetCode 题解仓库：https://github.com/azl397985856/leetcode 。 目前已经 39K star 啦。大家也可以关注我的公众号《力扣加加》带你啃下算法这块硬骨头。

题目地址（1707. 与数组中元素的最大异或值）

https://leetcode-cn.com/problems/maximum-xor-with-an-element-from-array/

题目描述

```
给你一个由非负整数组成的数组 nums 。另有一个查询数组 queries ，其中

第 i 个查询的答案是 xi 和任何 nums 数组中不超过 mi 的元素按位异或（

返回一个整数数组 answer 作为查询的答案，其中 answer.length == que



示例 1：

输入：nums = [0,1,2,3,4], queries = [[3,1],[1,3],[5,6]]
输出：[3,3,7]
解释：
1) 0 和 1 是仅有的两个不超过 1 的整数。0 XOR 3 = 3 而 1 XOR 3 =
2) 1 XOR 2 = 3.
3) 5 XOR 2 = 7.
示例 2：

输入：nums = [5,2,4,6,6,3], queries = [[12,4],[8,1],[6,3]]
输出：[15,-1,5]


提示：

1 <= nums.length, queries.length <= 105
queries[i].length == 2
0 <= nums[j], xi, mi <= 109
```

前置知识

- 异或
- 位运算
- 剪枝
- 双指针

公司

- 暂无

思路

PS：使用 JS 可以平方复杂度直接莽过。不过这个数据范围平方意味着 $10^(10)$ 次运算，很难想象这是怎么 AC 的。

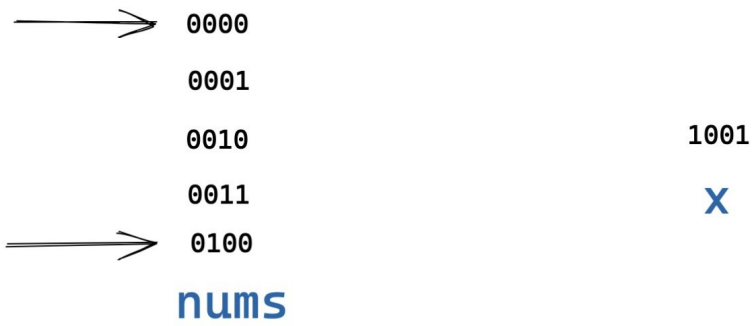
使用前缀树的思路和 字节跳动的算法面试题是什么难度？（第二弹） 第二题比较像，很多人的解法也是如此，我就不贴了。如果还是不懂得同学，建议先看下 421. 数组中两个数的最大异或值，基本就是一个前缀树的模板。

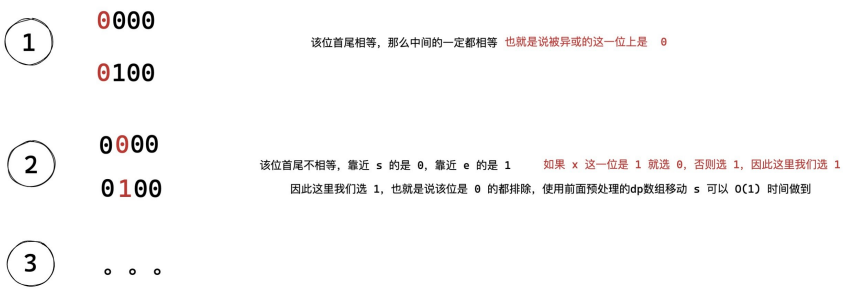
下面介绍一个 预处理 + 双指针的方法。

和 421. 数组中两个数的最大异或值 类似，核心一句话要记住。

不要关心 x 最后和 nums 中的谁异或了，**只关心最终异或的数的每一位分别是多少**。

以 nums[0,1,2,3,4], x 为 9 为例，给大家讲解一下核心原理。





具体算法：

- 首先对数据进行预处理，建立一个二维 dp 数组， dp[i][j] 是和 nums[j] 第 i 位相等的最小的数组下标。
- 为了使用双指针，我们需要对 nums 进行排序
- 接下来对每一个查询，我们调用 solve 函数计算最大的异或值。

- solve 函数内部使用双指针，比较头尾指针和 x 的异或结果。更新异或结果较小的那个即可。

代码

```
class Solution:
    def maximizeXor(self, nums: List[int], queries: List[Li
        def solve(x, m, s, e):
            if nums[0] > m: return -1
            max_v = 0
            for i in range(31, -1, -1):
                if nums[s] & (1<<i) == nums[e] & (1<<i):
                    max_v += nums[s] & (1<<i)
                elif nums[dp[i][e]] <= m and x ^ nums[s] <
                    max_v += nums[e] & (1<<i)
                    # 直接移动较小指针（s）到 dp[i][e]，其他不可
                    s = dp[i][e]
                else:
                    max_v += nums[s] & (1<<i)
                    # 直接移动较小指针（e）到 dp[i][e] - 1，其他
                    e = dp[i][e] - 1

            return max_v ^ x

        nums.sort()
        n = len(nums)
        #  dp[i][j] 是和 nums[j] 第 i 位相等的最小的数组下标
        dp = [[0 for _ in range(n)] for _ in range(32)]
        for i in range(32):
            for j in range(n):
                if j == 0 or (nums[j] & (1<<i)) != (nums[j-
                else: dp[i][j] = dp[i][j-1]
        return [solve(x, m, 0, n-1) for x,m in queries]
```

复杂度分析

- 时间复杂度：$O(max(NlogN, 32*Q))$，其中 Q 为 queries 长度,N 为 nums 长度。
- 空间复杂度：$O(32*N)$，其中 N 为 nums 长度。

后记

以上就是本电子书的全部内容了。如果你觉得这本书对你有用， 那么请将它分享给你身边的朋友，你的点赞和分享是我最大的动力。另外由于本人水平和精力有限，难免有不正确的地方，大家可以通过 github 的 pr 给我指正，感谢大家。

后期的话文章会第一时间在公众号和我的博客更新，并定期整理到这个电子书中来。因此你可以关注我的公众号或者博客， 也可以关注我的同步电子书的网站 西法的刷题秘籍 - 在线版 获得内容的更新。

如果想加入读者交流群， 在公众号回复 leetcode 即可，西法在群里等着你。

关注公众号《力扣加加》带你啃下算法这块硬骨头。

欢迎长按关注

lucifer 的博客地址：https://lucifer.ren/blog/