目录

开篇、刷题笔记阅读指南

准备工作：安装刷题全家桶

- labuladong 公众号/学习网站
- 配套 Chrome 刷题插件
- 配套 vscode 刷题插件
- 配套 Jebrains 刷题插件
- 番外：用算法打败算法
- 数据结构精品课
- 二叉树（递归）专题课
- 14 天刷题打卡挑战
- 面向新手的 Java 基础

第一章、基础数据结构

数组双指针

- 二分搜索
  - 34. 在排序数组中查找元素的第一个和最后一个位置
  - 704. 二分查找
  - 剑指 Offer 53 - I. 在排序数组中查找数字 I
  - 35. 搜索插入位置
  - 剑指 Offer II 068. 查找插入位置
  - 240. 搜索二维矩阵 II
  - 566. 重塑矩阵
  - 74. 搜索二维矩阵
  - 剑指 Offer 04. 二维数组中的查找
  - 519. 随机翻转矩阵
  - 354. 俄罗斯套娃信封问题
  - 392. 判断子序列
  - 658. 找到 K 个最接近的元素
  - 793. 阶乘函数后 K 个零
  - 852. 山脉数组的峰顶索引
  - 剑指 Offer II 069. 山峰数组的顶部
  - 1011. 在 D 天内送达包裹的能力
  - 875. 爱吃香蕉的珂珂
  - 剑指 Offer II 073. 狒狒吃香蕉
  - 1201. 丑数 III
  - 剑指 Offer 53 - II. 0～n-1中缺失的数字
- 滑动窗口
  - 3. 无重复字符的最长子串
  - 438. 找到字符串中所有字母异位词
  - 567. 字符串的排列

    - 76. 最小覆盖子串
    - 剑指 Offer 48. 最长不含重复字符的子字符串
    - 剑指 Offer II 014. 字符串中的变位词
    - 剑指 Offer II 015. 字符串中的所有变位词
    - 剑指 Offer II 016. 不含重复字符的最长子字符串
    - 剑指 Offer II 017. 含有所有字符的最短字符串
    - 155. 最小栈
    - 239. 滑动窗口最大值
    - 剑指 Offer 30. 包含min函数的栈
    - 剑指 Offer 59 - I. 滑动窗口的最大值
- 其他题目
    - 15. 三数之和
    - 18. 四数之和
    - 剑指 Offer II 007. 数组中和为 0 的三个数
    - 剑指 Offer 58 - I. 翻转单词顺序
    - 1260. 二维网格迁移
    - 151. 颠倒字符串中的单词
    - 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面
    - 11. 盛最多水的容器
    - 42. 接雨水
    - 26. 删除有序数组中的重复项
    - 27. 移除元素
    - 283. 移动零
    - 80. 删除有序数组中的重复项 II
    - 83. 删除排序链表中的重复元素
    - 剑指 Offer 57. 和为s的两个数字
    - 88. 合并两个有序数组
    - 977. 有序数组的平方
    - 986. 区间列表的交集
    - 82. 删除排序链表中的重复元素 II
    - 16. 最接近的三数之和
    - 259. 较小的三数之和
    - 360. 有序转化数组
    - 125. 验证回文串
    - 剑指 Offer II 018. 有效的回文
    - 870. 优势洗牌
    - 1099. 小于 K 的两数之和

链表双指针

- 2. 两数相加
- 445. 两数相加 II
- 67. 二进制求和
- 剑指 Offer II 025. 链表中的两数相加
- 141. 环形链表
- 142. 环形链表 II
- 160. 相交链表

- 19. 删除链表的倒数第 N 个结点
- 21. 合并两个有序链表
- 23. 合并 K 个升序链表
- 86. 分隔链表
- 876. 链表的中间结点
- 剑指 Offer 25. 合并两个排序的链表
- 剑指 Offer 52. 两个链表的第一个公共节点
- 剑指 Offer II 021. 删除链表的倒数第 n 个结点
- 剑指 Offer II 022. 链表中环的入口节点
- 剑指 Offer II 023. 两个链表的第一个重合节点
- 剑指 Offer II 078. 合并排序链表
- 1305. 两棵二叉搜索树中的所有元素
- 264. 丑数 II
- 313. 超级丑数
- 88. 合并两个有序数组
- 97. 交错字符串
- 977. 有序数组的平方
- 剑指 Offer 49. 丑数
- 剑指 Offer 18. 删除链表的节点
- 360. 有序转化数组
- 355. 设计推特
- 373. 查找和最小的 K 对数字
- 378. 有序矩阵中第 K 小的元素
- 剑指 Offer II 061. 和最小的 k 个数对
- 24. 两两交换链表中的节点
- 25. K 个一组翻转链表
- 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
- 92. 反转链表 II
- 1650. 二叉树的最近公共祖先 III
- 1257. 最小公共区域
- 234. 回文链表
- 剑指 Offer II 027. 回文链表
- 1201. 丑数 III

前缀和

- 303. 区域和检索 - 数组不可变
- 304. 二维区域和检索 - 矩阵不可变
- 剑指 Offer II 013. 二维子矩阵的和
- 1314. 矩阵区域和
- 325. 和等于 k 的最长子数组长度
- 327. 区间和的个数
- 1352. 最后 K 个数的乘积
- 剑指 Offer 66. 构建乘积数组

差分数组

- 1094. 拼车
- 1109. 航班预订统计
- 370. 区间加法

队列/栈算法

- 1541. 平衡括号字符串的最少插入次数
- 20. 有效的括号
- 921. 使括号有效的最少添加
- 32. 最长有效括号
- 71. 简化路径
- 150. 逆波兰表达式求值
- 剑指 Offer II 036. 后缀表达式
- 155. 最小栈
- 239. 滑动窗口最大值
- 剑指 Offer 30. 包含min函数的栈
- 剑指 Offer 59 - I. 滑动窗口的最大值
- 225. 用队列实现栈
- 232. 用栈实现队列
- 剑指 Offer 09. 用两个栈实现队列
- 445. 两数相加 II
- 剑指 Offer II 025. 链表中的两数相加
- 剑指 Offer 06. 从尾到头打印链表

二叉堆

- 23. 合并 K 个升序链表
- 313. 超级丑数
- 355. 设计推特
- 373. 查找和最小的 K 对数字
- 378. 有序矩阵中第 K 小的元素
- 剑指 Offer II 061. 和最小的 k 个数对
- 剑指 Offer II 078. 合并排序链表
- 215. 数组中的第 K 个最大元素
- 347. 前 K 个高频元素
- 703. 数据流中的第 K 大元素
- 剑指 Offer 40. 最小的k个数
- 剑指 Offer II 059. 数据流的第 K 大数值
- 剑指 Offer II 060. 出现频率最高的 k 个数字
- 剑指 Offer II 076. 数组中的第 k 大的数字
- 295. 数据流的中位数
- 剑指 Offer 41. 数据流中的中位数
- 692. 前K个高频单词
- 451. 根据字符出现频率排序
- 1834. 单线程 CPU
- 1845. 座位预约管理系统

数据结构设计

- 146. LRU 缓存机制
- 剑指 Offer II 031. 最近最少使用缓存
- 155. 最小栈
- 239. 滑动窗口最大值
- 剑指 Offer 30. 包含min函数的栈
- 剑指 Offer 59 - I. 滑动窗口的最大值
- 284. 顶端迭代器
- 341. 扁平化嵌套列表迭代器
- 355. 设计推特
- 380. O(1) 时间插入、删除和获取随机元素
- 519. 随机翻转矩阵
- 剑指 Offer II 030. 插入、删除和随机访问都是 O(1) 的容器
- 460. LFU 缓存
- 895. 最大频率栈
- 1845. 座位预约管理系统

第二章、进阶数据结构

二叉树

- 94. 二叉树的中序遍历
- 100. 相同的树
- 572. 另一棵树的子树
- 102. 二叉树的层序遍历
- 103. 二叉树的锯齿形层序遍历
- 107. 二叉树的层序遍历 II
- 1161. 最大层内元素和
- 1302. 层数最深叶子节点的和
- 1609. 奇偶树
- 637. 二叉树的层平均值
- 919. 完全二叉树插入器
- 958. 二叉树的完全性检验
- 剑指 Offer 32 - I. 从上到下打印二叉树
- 剑指 Offer 32 - II. 从上到下打印二叉树 II
- 剑指 Offer 32 - III. 从上到下打印二叉树 III
- 104. 二叉树的最大深度
- 1376. 通知所有员工所需的时间
- 144. 二叉树的前序遍历
- 543. 二叉树的直径
- 559. N 叉树的最大深度
- 865. 具有所有最深节点的最小子树
- 剑指 Offer 55 - I. 二叉树的深度
- 1123. 最深叶节点的最近公共祖先
- 105. 从前序与中序遍历序列构造二叉树
- 106. 从中序与后序遍历序列构造二叉树
- 654. 最大二叉树
- 889. 根据前序和后序遍历构造二叉树

- 剑指 Offer 07. 重建二叉树
- 111. 二叉树的最小深度
- 542. 01 矩阵
- 114. 二叉树展开为链表
- 116. 填充每个节点的下一个右侧节点指针
- 226. 翻转二叉树
- 897. 递增顺序搜索树
- 剑指 Offer 27. 二叉树的镜像
- 剑指 Offer II 052. 展平二叉搜索树
- 117. 填充每个节点的下一个右侧节点指针 II
- 145. 二叉树的后序遍历
- 222. 完全二叉树的节点个数
- 1644. 二叉树的最近公共祖先 II
- 1676. 二叉树的最近公共祖先 IV
- 235. 二叉搜索树的最近公共祖先
- 236. 二叉树的最近公共祖先
- 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先
- 剑指 Offer 68 - II. 二叉树的最近公共祖先
- 297. 二叉树的序列化与反序列化
- 449. 序列化和反序列化二叉搜索树
- 剑指 Offer 37. 序列化二叉树
- 剑指 Offer II 048. 序列化与反序列化二叉树
- 341. 扁平化嵌套列表迭代器
- 429. N 叉树的层序遍历
- 124. 二叉树中的最大路径和
- 1245. 树的直径
- 250. 统计同值子树
- 366. 寻找二叉树的叶子节点
- 687. 最长同值路径
- 814. 二叉树剪枝
- 979. 在二叉树中分配硬币
- 剑指 Offer II 047. 二叉树剪枝
- 剑指 Offer II 051. 节点之和最大的路径
- 1325. 删除给定值的叶子节点
- 582. 杀掉进程
- 589. N 叉树的前序遍历
- 590. N 叉树的后序遍历
- 652. 寻找重复的子树
- 998. 最大二叉树 II
- 965. 单值二叉树
- 1740. 找到二叉树中的距离
- 剑指 Offer 33. 二叉搜索树的后序遍历序列

二叉搜索树

- 95. 不同的二叉搜索树 II
- 96. 不同的二叉搜索树

- 255. 验证前序遍历序列二叉搜索树
- 450. 删除二叉搜索树中的节点
- 700. 二叉搜索树中的搜索
- 701. 二叉搜索树中的插入操作
- 98. 验证二叉搜索树
- 1038. 把二叉搜索树转换为累加树
- 230. 二叉搜索树中第 K 小的元素
- 538. 把二叉搜索树转换为累加树
- 剑指 Offer 54. 二叉搜索树的第k大节点
- 剑指 Offer II 054. 所有大于等于节点的值之和
- 501. 二叉搜索树中的众数
- 530. 二叉搜索树的最小绝对差
- 783. 二叉搜索树节点最小距离
- 270. 最接近的二叉搜索树值
- 285. 二叉搜索树中的中序后继
- 剑指 Offer II 053. 二叉搜索树中的中序后继
- 1373. 二叉搜索子树的最大键值和
- 剑指 Offer 33. 二叉搜索树的后序遍历序列

图论算法

- 图的遍历
  - 797. 所有可能的路径
- 二分图
  - 785. 判断二分图
  - 886. 可能的二分法
- 环检测/拓扑排序
  - 207. 课程表
  - 210. 课程表 II
  - 剑指 Offer II 113. 课程顺序
- 并查集算法
  - 130. 被围绕的区域
  - 990. 等式方程的可满足性
  - 765. 情侣牵手
  - 947. 移除最多的同行或同列石头
- 最小生成树
  - 1135. 最低成本联通所有城市
  - 1361. 验证二叉树
  - 1584. 连接所有点的最小费用
  - 261. 以图判树
- 最短路径
  - 1514. 概率最大的路径
  - 1631. 最小体力消耗路径
  - 743. 网络延迟时间

第三章、暴力搜索算法

回溯算法

- 17. 电话号码的字母组合
- 22. 括号生成
- 剑指 Offer II 085. 生成匹配的括号
- 37. 解数独
- 39. 组合总和
- 46. 全排列
- 77. 组合
- 78. 子集
- 剑指 Offer II 079. 所有子集
- 剑指 Offer II 080. 含有 k 个元素的组合
- 剑指 Offer II 081. 允许重复选择元素的组合
- 剑指 Offer II 083. 没有重复元素集合的全排列
- 51. N 皇后
- 104. 二叉树的最大深度
- 1376. 通知所有员工所需的时间
- 559. N 叉树的最大深度
- 865. 具有所有最深节点的最小子树
- 剑指 Offer 55 - I. 二叉树的深度
- 1123. 最深叶节点的最近公共祖先
- 293. 翻转游戏
- 294. 翻转游戏 II
- 491. 递增子序列
- 494. 目标和
- 剑指 Offer II 102. 加减的目标值
- 698. 划分为 k 个相等的子集
- 784. 字母大小写全排列
- 剑指 Offer 38. 字符串的排列

DFS 算法

- 79. 单词搜索
- 1020. 飞地的数量
- 1254. 统计封闭岛屿的数目
- 1905. 统计子岛屿
- 200. 岛屿数量
- 694. 不同的岛屿数量
- 695. 岛屿的最大面积
- 剑指 Offer II 105. 岛屿的最大面积
- 576. 出界的路径数
- 130. 被围绕的区域
- 剑指 Offer 13. 机器人的运动范围

BFS 算法

- 102. 二叉树的层序遍历
- 103. 二叉树的锯齿形层序遍历

- 107. 二叉树的层序遍历 II
- 1161. 最大层内元素和
- 1302. 层数最深叶子节点的和
- 1609. 奇偶树
- 637. 二叉树的层平均值
- 919. 完全二叉树插入器
- 958. 二叉树的完全性检验
- 剑指 Offer 32 - I. 从上到下打印二叉树
- 剑指 Offer 32 - II. 从上到下打印二叉树 II
- 剑指 Offer 32 - III. 从上到下打印二叉树 III
- 111. 二叉树的最小深度
- 542. 01 矩阵
- 752. 打开转盘锁
- 剑指 Offer II 109. 开密码锁
- 1926. 迷宫中离入口最近的出口
- 286. 墙与门
- 310. 最小高度树
- 429. N 叉树的层序遍历
- 582. 杀掉进程
- 773. 滑动谜题
- 1091. 二进制矩阵中的最短路径
- 490. 迷宫
- 505. 迷宫 II

第四章、动态规划算法

一维 DP

- 45. 跳跃游戏 II
- 55. 跳跃游戏
- 70. 爬楼梯
- 剑指 Offer 10- II. 青蛙跳台阶问题
- 198. 打家劫舍
- 213. 打家劫舍 II
- 337. 打家劫舍 III
- 剑指 Offer II 089. 房屋偷盗
- 剑指 Offer II 090. 环形房屋偷盗
- 300. 最长递增子序列
- 354. 俄罗斯套娃信封问题
- 53. 最大子序和
- 剑指 Offer 42. 连续子数组的最大和
- 152. 乘积最大子数组
- 209. 长度最小的子数组
- 918. 环形子数组的最大和
- 剑指 Offer II 008. 和大于等于 target 的最短子数组
- 279. 完全平方数
- 322. 零钱兑换

- 剑指 Offer II 103. 最少的硬币数目

二维 DP

- 10. 正则表达式匹配
- 44. 通配符匹配
- 剑指 Offer 19. 正则表达式匹配
- 62. 不同路径
- 剑指 Offer II 098. 路径的数目
- 64. 最小路径和
- 剑指 Offer 47. 礼物的最大价值
- 剑指 Offer II 099. 最小路径之和
- 1143. 最长公共子序列
- 72. 编辑距离
- 剑指 Offer II 095. 最长公共子序列
- 121. 买卖股票的最佳时机
- 122. 买卖股票的最佳时机 II
- 123. 买卖股票的最佳时机 III
- 188. 买卖股票的最佳时机 IV
- 309. 最佳买卖股票时机含冷冻期
- 714. 买卖股票的最佳时机含手续费
- 剑指 Offer 63. 股票的最大利润
- 174. 地下城游戏
- 312. 戳气球
- 416. 分割等和子集
- 剑指 Offer II 101. 分割等和子集
- 494. 目标和
- 剑指 Offer II 102. 加减的目标值
- 514. 自由之路
- 518. 零钱兑换 II
- 787. K 站中转内最便宜的航班
- 887. 鸡蛋掉落
- 931. 下降路径最小和
- 583. 两个字符串的删除操作
- 712. 两个字符串的最小 ASCII 删除和

背包问题

- 416. 分割等和子集
- 剑指 Offer II 101. 分割等和子集
- 494. 目标和
- 剑指 Offer II 102. 加减的目标值
- 518. 零钱兑换 II

第五章、其他经典算法

数学算法

- 面试题45. 把数组排成最小的数
- 面试题61. 扑克牌中的顺子
- 8. 字符串转换整数 (atoi)
- 面试题67. 把字符串转换成整数
- 9. 回文数
- 17. 电话号码的字母组合
- 50. Pow(x, n)
- 剑指 Offer 16. 数值的整数次方
- 67. 二进制求和
- 77. 组合
- 78. 子集
- 剑指 Offer II 079. 所有子集
- 剑指 Offer II 080. 含有 k 个元素的组合
- 134. 加油站
- 136. 只出现一次的数字
- 191. 位 1 的个数
- 231. 2 的幂
- 268. 丢失的数字
- 剑指 Offer 15. 二进制中1的个数
- 172. 阶乘后的零
- 793. 阶乘函数后 K 个零
- 204. 计数质数
- 264. 丑数 II
- 剑指 Offer 49. 丑数
- 1201. 丑数 III
- 263. 丑数
- 313. 超级丑数
- 279. 完全平方数
- 292. Nim 游戏
- 319. 灯泡开关
- 877. 石子游戏
- 295. 数据流的中位数
- 剑指 Offer 41. 数据流中的中位数
- 306. 累加数
- 372. 超级次方
- 382. 链表随机节点
- 398. 随机数索引
- 391. 完美矩形
- 400. 第 N 位数字
- 剑指 Offer 44. 数字序列中某一位的数字
- 509. 斐波那契数
- 70. 爬楼梯
- 剑指 Offer 10- I. 斐波那契数列
- 剑指 Offer 10- II. 青蛙跳台阶问题
- 645. 错误的集合
- 710. 黑名单中的随机数

区间问题

- 1288. 删除被覆盖区间
- 56. 合并区间
- 986. 区间列表的交集
- 剑指 Offer II 074. 合并区间
- 435. 无重叠区间
- 452. 用最少数量的箭引爆气球
- 1024. 视频拼接
- 451. 根据字符出现频率排序
- 1834. 单线程 CPU

开篇、刷题笔记阅读指南

在最开头先回答一个大家问的最多的问题：

东哥你太能写了，又是算法小抄，又是算法笔记，又是算法秘籍，又是纸质书，又是网站，又是课程，到底该怎么学？

好问题，随着我对算法教程的不断优化和完善，内容确实在一直更新，可能会给新来的读者造成困扰，所以在开头回答一下这个问题。

首先说一下《labuladong 的算法小抄》，这个名称是我在 2021～2022 年用来命名 PDF、纸质书、网站的，**所以无论是《算法小抄》的纸质书还是 PDF，你都可以理解为老版本，现在都不再建议学习了，因为有更好更全面的内容给你学**。

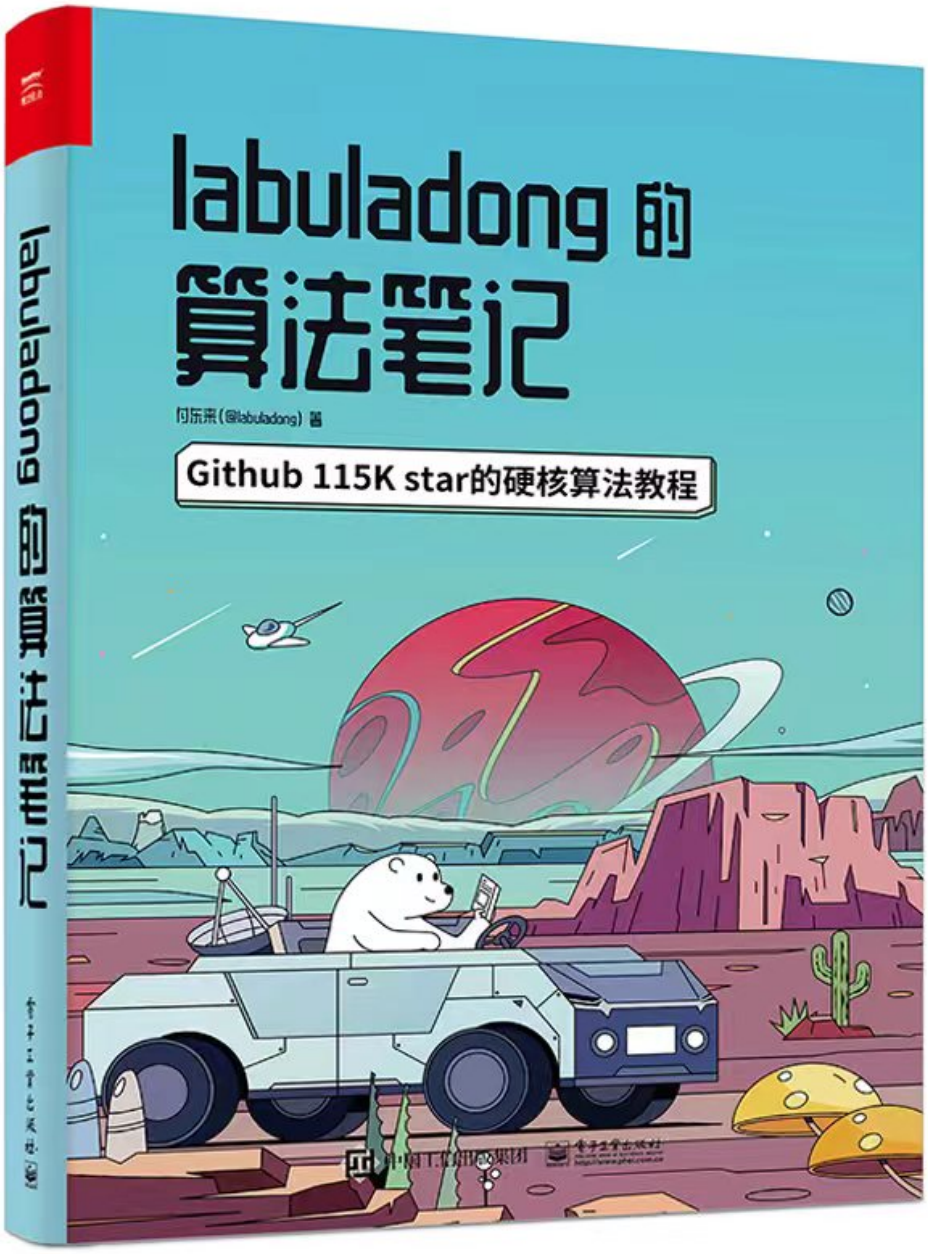
自 2024 年开始，你要学最新的内容有几个渠道：

最推荐的就是我的算法网站，最新地址是 https://labuladong.online/algo/，建议收藏。

因为网站的内容方便及时更新，而且我亲自开发很多实用的可交互功能，给你最丝滑完美的学习体验。我之前有推出过一些单独的付费课程，但是自 2024/3 开始，所有付费课程的内容都融合到了网站。所以我的唯一的付费项目就是 网站会员，购买后可以解锁我的所有内容和功能，没有其他付费项目了。

除了网站，如果你有学习 PDF 的需求，那么我有两本 PDF，一本是《labuladong 的算法秘籍》，另一本是《labuladong 的刷题笔记》。其中《labuladong 的算法秘籍》是从网站内容中精选出来的，可以理解成是一本教材，而《labuladong 的刷题笔记》是从刷题插件的题解思路中精选出来的，可以理解成是一本练习册。

除了网站，如果你有阅读纸质书的需求，我在 2023 年年底出的一本新书名为《labuladong 的算法笔记》，这本书可以理解为是《labuladong 的算法秘籍》对应的纸质书，其内容也是网站内容的精选。



那有读者肯定要说，既然纸质书对应的是《算法秘籍》PDF，为什么不直接把纸质书命名为《labuladong 的算法秘籍》，而是命名为《labuladong 的算法笔记》呢？这样不是徒增理解成本吗？

我肯定也和大家想的一样啊，《算法秘籍》PDF 火的一塌糊涂，纸质书如果和 PDF 同名，肯定可以带动纸质书的销售对吧，我何乐而不为呢？

其实根本原因是因为我国有相关规定，正规图书出版物名称中不能包含类似「秘籍」这样的字眼，没办法，当时给 PDF 起名的时候哪想得到这么多？所以最终决定给纸质书起名《算法笔记》，确实容易给不知情的读者造成困扰，所以就在这里阐明一下。

纸质书的更多介绍以及购买可以微信扫码查看：

就先啰嗦这么多，相信大家已经理清了各个名称之间的关系了，如果你没有特殊的需求，直接在我的网站 https://labuladong.online/algo/ 学习就可以了，这也是我最推荐的学习渠道。

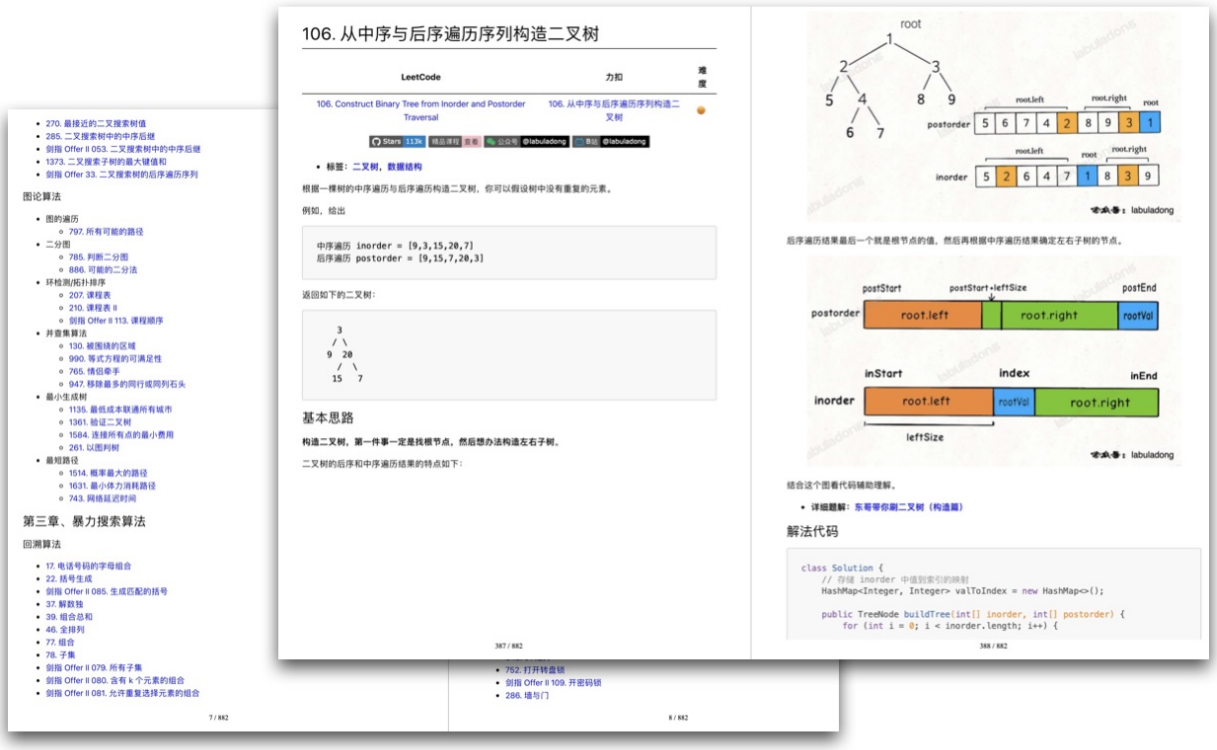
前言

我的 GitHub 算法仓库 目前已经快 100k star 了（如果你从我这学到了知识，希望你帮我点个 star），为了感谢大家一直以来的支持，我制作了刷题全家桶。

labuladong 的刷题全家桶 由学习网站、《算法秘籍》《刷题笔记》两本 PDF、Chrome/vscode/Jetbrain 三个不同平台的刷题插件组成，致力于为大家提供最丝滑的刷题体验。

这本 PDF 是《labuladong 的刷题笔记》：

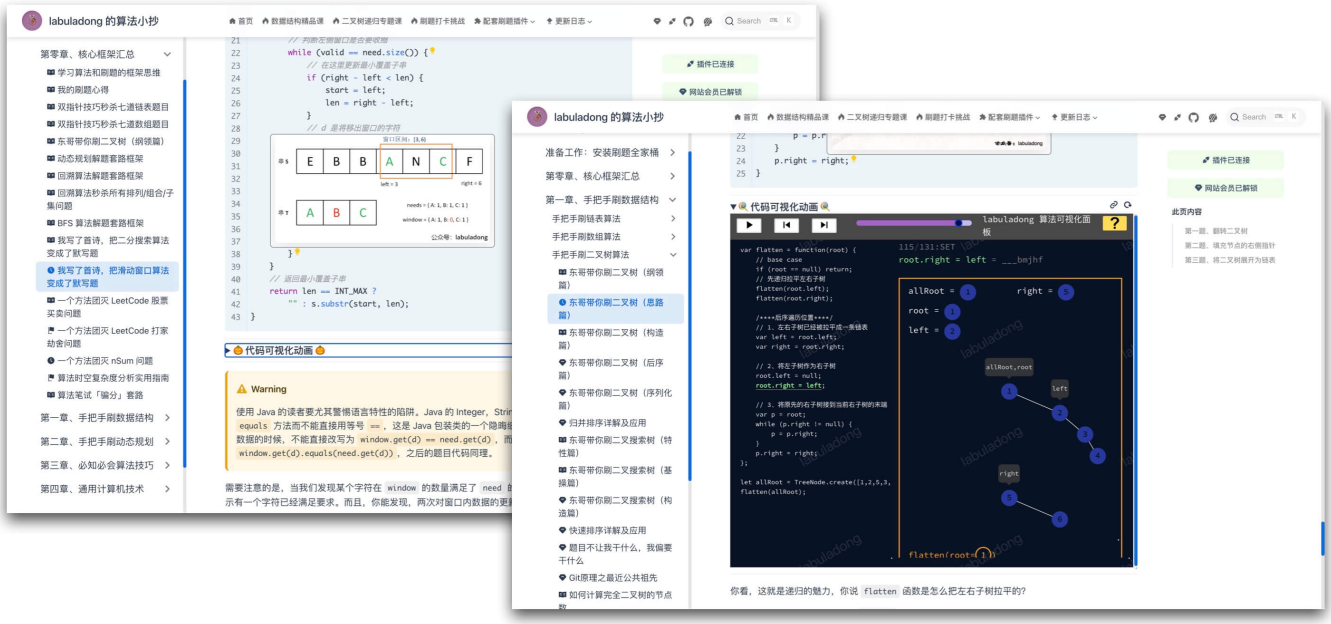
《labuladong 的算法笔记》：练习册，包含题目的简明思路，适合复习加深印象



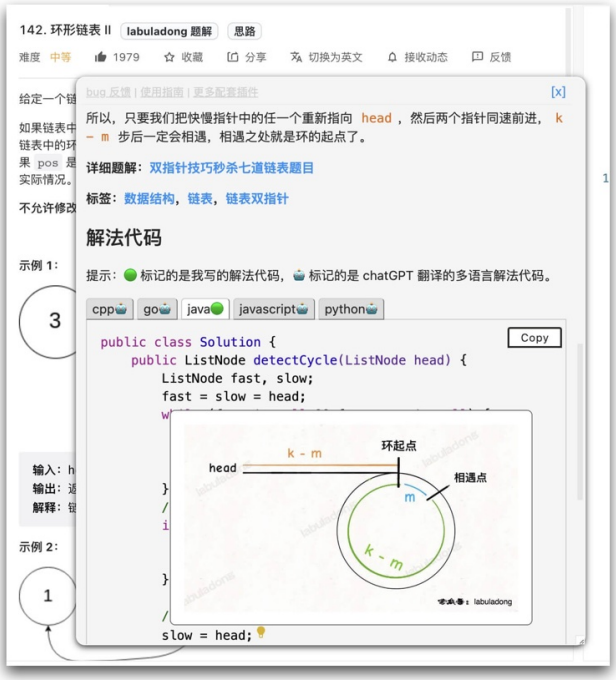
其他的配套工具我会在第一章逐一详细介绍：

《labuladong 的算法秘籍》：汇总所有常见算法框架，可跳转网页查看算法可视化面板

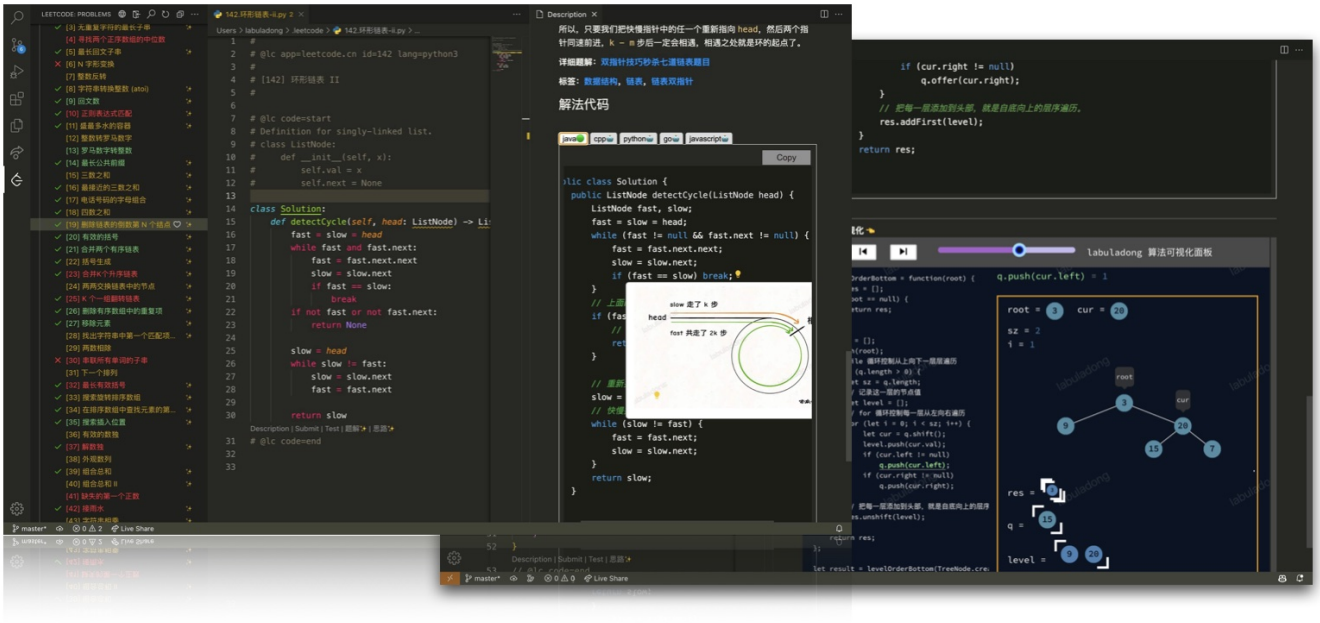
labuladong 的算法网站：汇总所有常见算法框架，支持算法可视化等实用功能



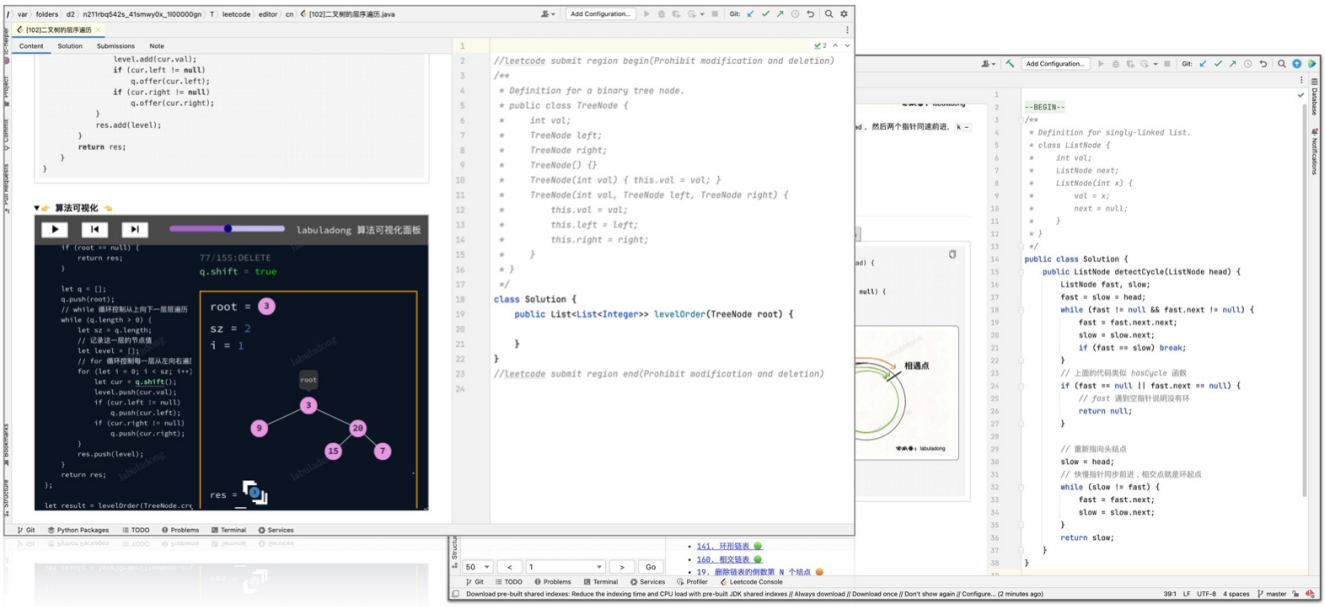
labuladong 的 Chrome 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



labuladong 的 vscode 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



labuladong 的 Jetbrains 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



公众号后台回复关键词「**全家桶**」即可获取下载和使用手册，而且你会发现 **PDF 和插件都有版本号，也就是说我一直在更新全家桶的内容**，修正错误或更新内容，所以你应该选择最新版本学习。

《算法秘籍》会详尽解析各种算法的原理和应用，可以理解成一本教材，而《刷题笔记》的灵感来源于「单词速记卡」的形式，其中只列出每到题目简明扼要的思路和参考解法，可以理解成一本练习册。**你可以在碎片化的时间翻看《刷题笔记》，像背单词一样背算法**，对于各种算法技巧，如果没事儿就看，哪有记不住的道理？或者，对于公众号的老读者，《算法笔记》中的技巧应该都了然于心了，那么这份《刷题笔记》可以作为一种测验手段，如果看了题目不能迅速想到解题思路，或者看了思路写不出代码，那就说明这块知识点掌握的不太好，需要重新复习巩固。

在阅读这本《刷题笔记》之前，你应该先读完《labuladong 的算法秘籍》，因为这本刷题笔记的主要作用是复习巩固算法秘籍中的各种算法技巧，帮助你高效复习。这本《刷题笔记》的目录结构和《算法秘籍》完全

相同，不同点在于本书是按照题目进行分类，每道题目只给出简明扼要的思路提示和参考答案。

扫码关注我的公众号，后台回复关键词「**全家桶**」即可免费下载两本 PDF 和配套刷题插件。

也建议关注我的微信视频号，我会不定期抽空直播，而且会在视频号积累学习算法的短视频，分享自己的学习经验：

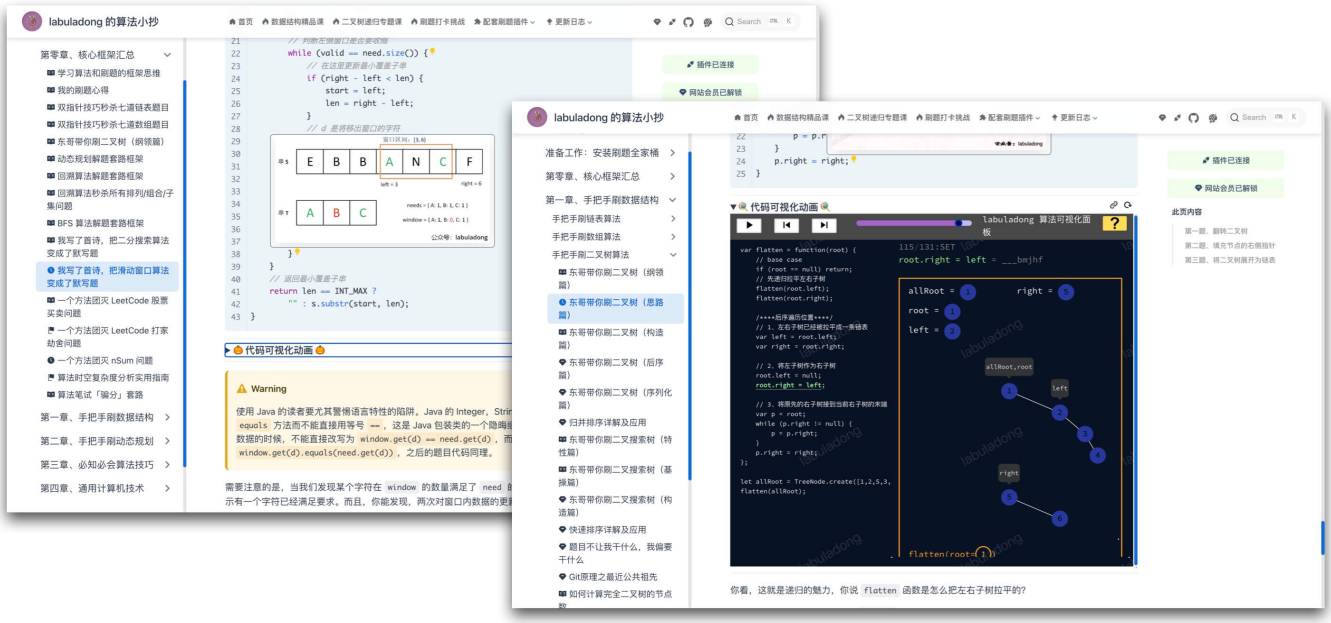
labuladong 的刷题全家桶总入口

> labuladong 的刷题全家桶由学习网站、《算法秘籍》《刷题笔记》两本 PDF、Chrome/vscode/Jetbrain 三个不同平台的刷题插件组成，致力于为大家提供最丝滑的刷题体验。公众号后台回复关键词「**全家桶**」即可获取下载和使用手册。

一、公众号/学习网站

内容当然是我的系列算法教程中最核心的部分，其他的插件和 PDF 教材都是为内容服务的。我的算法教程都发布在公众号/网站上，相信你会未来会在这里花费大量的学习时间，而不是仅仅加入收藏夹~

labuladong 的算法网站：汇总所有常见算法框架，支持算法可视化等实用功能



具体介绍：labuladong 的公众号/算法网站使用手册

二、PDF 教材

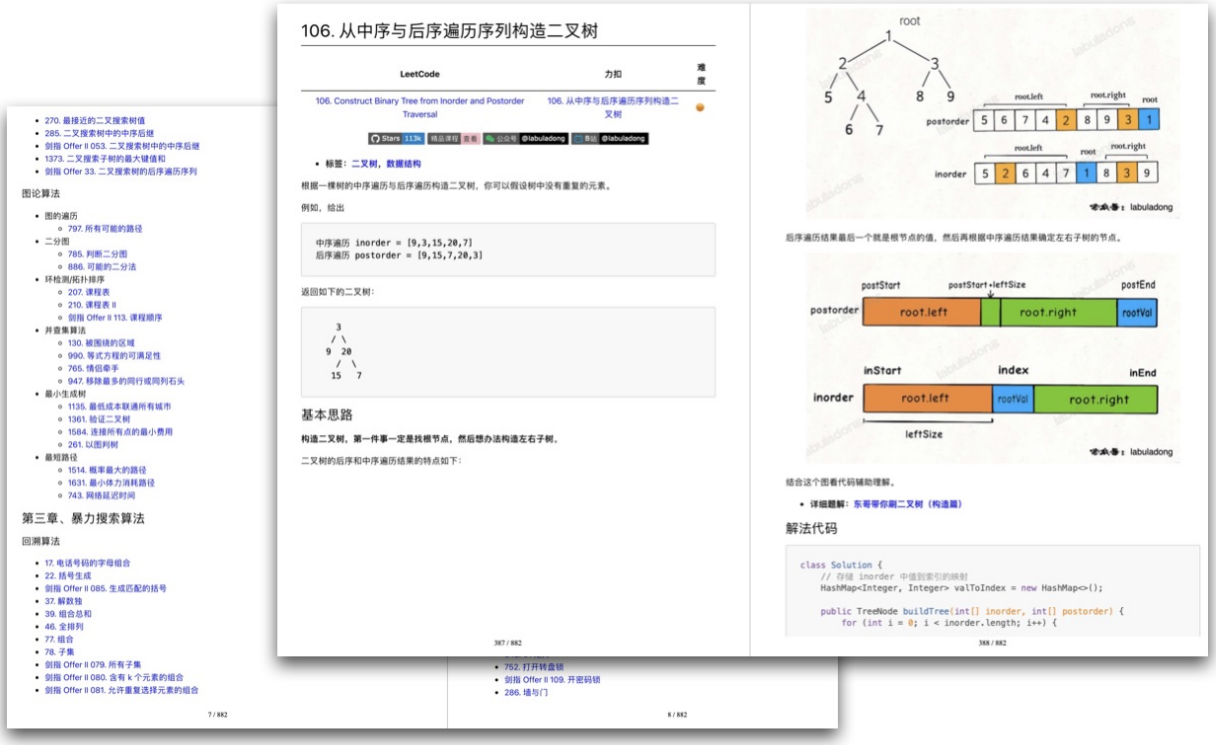
鉴于我积累的算法教程比较多，为了防止过于陡峭的学习曲线，我精心挑选了必知必会的算法技巧，设计了一套循序渐进的学习路线，并制作成 PDF 方便大家离线学习/记笔记。

PDF 共两本，一本《labuladong 的算法秘籍》类似教材，帮你系统掌握算法的知识体系：

《labuladong 的算法秘籍》：汇总所有常见算法框架，可跳转网页查看算法可视化面板

另一本《labuladong 的刷题笔记》类似练习册，助你遨游题海，随心所欲地运用所学的技巧解题：

《labuladong 的算法笔记》：练习册，包含题目的简明思路，适合复习加深印象



具体介绍：配套 PDF 算法教材使用手册

三、精品课程

如果你认可我的算法教程，可以购买我的付费课程，目前我制作完成了《数据结构精品课》 和《二叉树（递归）专题课》。另外，还有押金形式的刷题打卡挑战，完成打卡后押金将全额返还，打卡挑战适用于自己刷

题坚持不下来的小伙伴参加。

课程和打卡挑战都可以在我的教程网站购买和报名：

话说回来，我今天能拥有这么多读者，都靠大家捧场，大家能从我这学到东西我就很高兴，不差钱的买个课捧个钱场，缺钱的把我的公众号/网站推荐给身边的朋友捧个人场，都是对我的支持！

四、Chrome 插件

主要功能：Chrome 插件可以在中文版力扣或英文版 LeetCode 上快捷查看我的「题解」或「思路」，并添加了题目和算法技巧之间的引用关系，可以和我的网站/公众号/课程联动，给我的读者提供最丝滑的刷题体验。

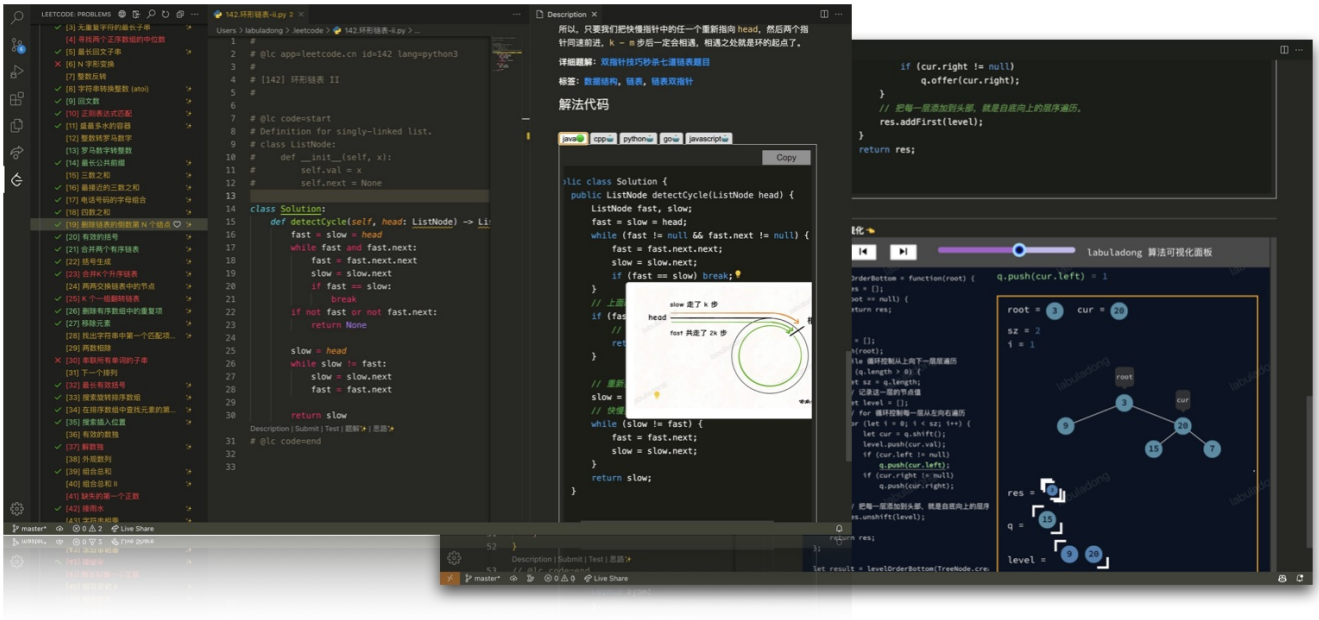
labuladong 的 Chrome 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化

安装使用手册：Chrome 插件安装使用手册

五、vscode 插件

主要功能：和 Chrome 插件功能基本相同，习惯在 vscode 上刷题的读者可以使用该插件。

labuladong 的 vscode 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化

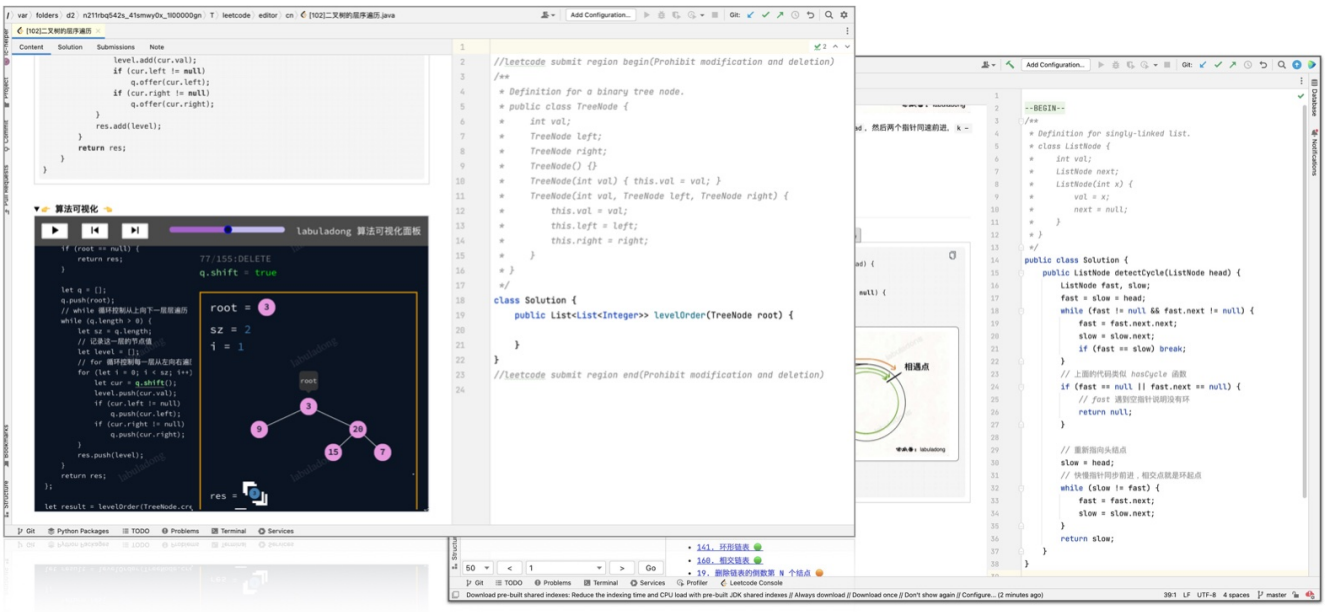


安装使用手册：vscode 插件安装使用手册

六、Jetbrains 插件

主要功能：和 Chrome 插件功能基本相同，习惯在 Jetbrains 家的 IDE（PyCharm/Intellij/Goland 等）上刷题的读者可以使用该插件。

labuladong 的 Jetbrains 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



安装使用手册：Jetbrains 插件安装使用手册

最后，把上面这些工具汇总成一张图：

祝大家学习愉快，在题海中自在遨游！

labuladong 公众号/网站

> 提示：labuladong 的刷题全家桶由学习网站、《算法秘籍》《刷题笔记》两本 PDF、Chrome/vscode/Jetbrain 三个不同平台的刷题插件组成，致力于为大家提供最丝滑的刷题体验。公众号后台回复关键词「**全家桶**」即可获取下载和使用手册。

很荣幸我的系列算法教程受到大家的认可，最开始我的教程都是发布在公众号上的，但考虑到公众号的诸多弊端，我把内容也同步到了我的算法网站上。

公众号的形式适合挤地铁时阅读，网站的形式适合~摸鱼时~，奥不是，是适合有大块时间专心学习时阅读，毕竟在电脑上可以边看文章边刷题，再结合我后面要介绍的刷题插件，体验会非常流畅。

先说下公众号，在公众号的菜单栏可以看到目录的入口以及各种插件的安装指南：

我的刷题网站同时部署在 GitHub Pages 和 Gitee Pages：

https://labuladong.gitee.io/algo/

https://labuladong.github.io/algo/

两个网站的内容都是一样的，主要是考虑到有的国内和海外的读者网络情况的差异，所以同时维护了这两个镜像站点，你可以根据自己的网络情况选择网速更快的站点学习。

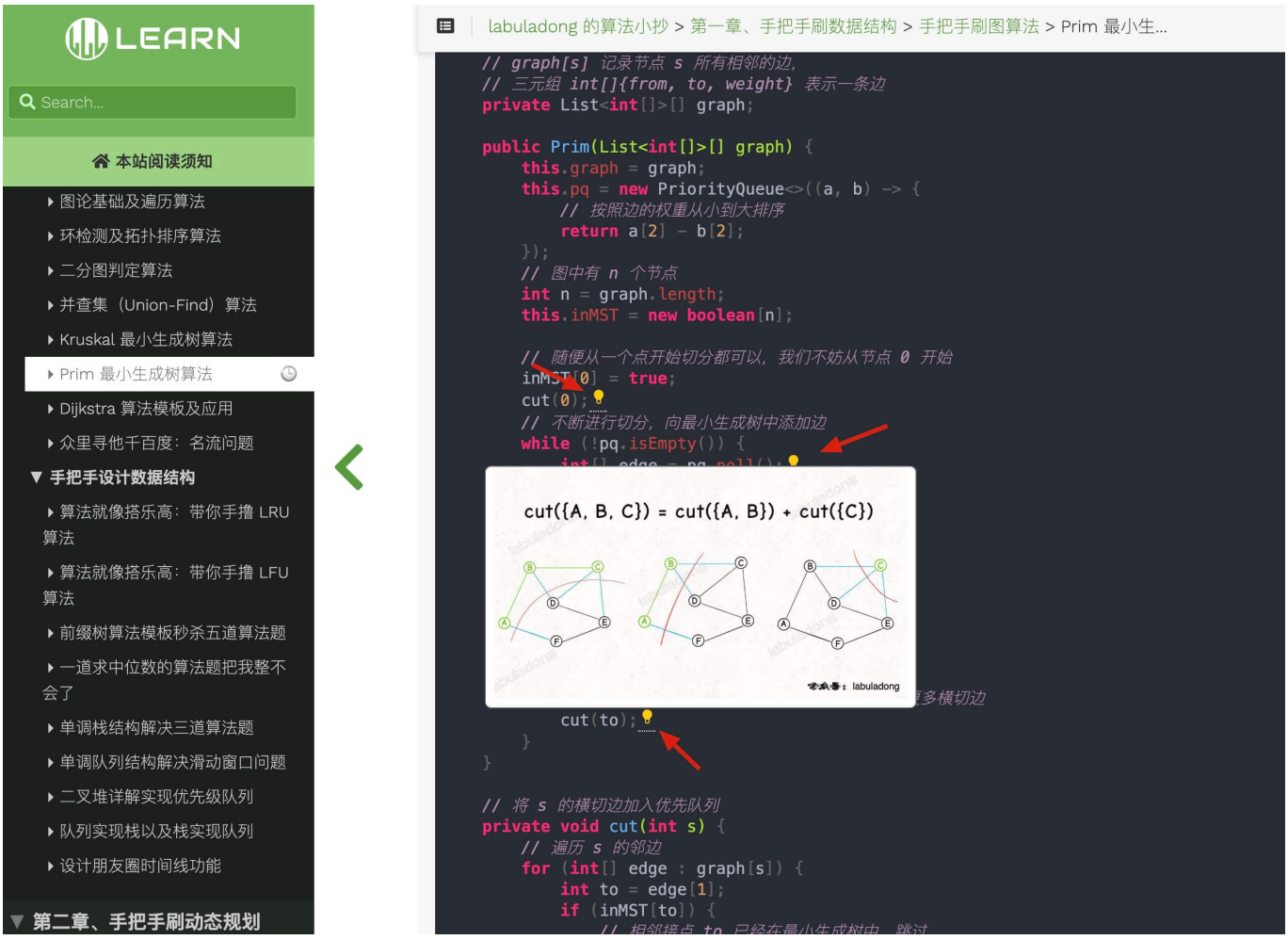
接下来列举一下我的刷题网站的实用功能：

0️⃣ 老读者知道，我以前的教程主要以 Java 语言为主，但现在有了 chatGPT 的加持，我的网站以及所有配套插件已经支持了所有主流编程语言，可以满足所有读者的需求：

java 🟢 | cpp 🤖 | python 🤖 | go 🤖 | javascript 🤖

```java
ListNode detectCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if (fast == slow) break;
    }
    // 上面的代码类似 hasCycle 函数
    if (fast == null || fast.next == null) {
        // fast 遇到空指针说明没有环
        return null;
    }

    // 重新指向头结点
    slow = head;
    // 快慢指针同步前进，相交点就是环起点
    while (slow != fast) {
        fast = fast.next;
        slow = slow.next;
    }
    return slow;
}
```

1️⃣ 有些较为复杂的代码块中包含小灯泡图标，鼠标移至小灯泡图标上就会弹出图片辅助理解：



2 我在每篇文章末尾添加了该文章相关的其他文章及其他算法题，你点击链接就可以跳转过去：

▼引用本文的文章

FloodFill算法详解及应用　东哥带你刷二叉树（纲领篇）

并查集（Union-Find）算法

▼引用本文的题目

安装 我的 Chrome 刷题插件 点开下列题目可直接查看解题思路：

LeetCode	力扣
1219. Path with Maximum Gold	1219. 黄金矿工
79. Word Search	79. 单词搜索
-	剑指 Offer 13. 机器人的运动范围
-	剑指 Offer II 105. 岛屿的最大面积

而且如果你安装了后文介绍的 Chrome 刷题插件，点击题目就可以直接查看我为该题目撰写的思路解析：

labuladong 的 Chrome 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化

3 还有一个贴心小功能，侧边栏中学完的文章会显示 ► 标记，未学完文章会显示 🕒 标记，全部学完的章节会显示 ► 标记。

4 我在每篇文章下都开放了评论区，欢迎大家友好讨论：

在评论区会有很多读者讨论问题，给出多语言的解法代码，甚至有读者会基于我的思路给出更高效的算法。

如果有的评论帮助到了你，请不要吝惜你的点赞，如果你想评论，请注意遵守基本礼仪，禁止嘲讽、阴阳怪气、刷屏、发学习无关的内容。我会定期查看并回复评论，发现违规者直接拉黑，希望大家共同维护良好的学习氛围。

5️⃣ 我的网站可以和我的系列刷题插件无缝衔接，提供最丝滑的刷题体验。具体可以查看 Chrome 插件安装使用手册/vscode 插件安装使用手册/Jetbrains 插件安装使用手册。

6️⃣ 根据网站/公众号的内容，我还制作了两本精品 PDF，详情见 配套 PDF 算法教材，这里就不详细介绍了。

最后，很多读者留言说我快把那些算法培训班干失业了，这话我可不敢当，我能有今天都靠大家捧场，大家能从我这学到东西我就很高兴，大家可以买个我的课支持下，或者把我的公众号/网站推荐个更多需要的朋友，感谢~

配套 Chrome 刷题插件

> labuladong 的刷题全家桶 由学习网站、《算法秘籍》《刷题笔记》两本 PDF、Chrome/vscode/Jetbrain 三个不同平台的刷题插件组成，致力于为大家提供最丝滑的刷题体验。公众号后台回复关键词「**全家桶**」即可获取下载和使用手册。

> 虽然我叫这款插件为「Chrome 插件」，但实际上插件并不仅限于 Chrome 浏览器安装使用。像 Edge 浏览器、360 浏览器这些使用 Chromium 内核的浏览器都可以安装，具体见安装指南。

视频介绍

最近录了一个视频介绍 Chrome 插件的主要功能，喜欢看视频的读者可以去 B 站观看：

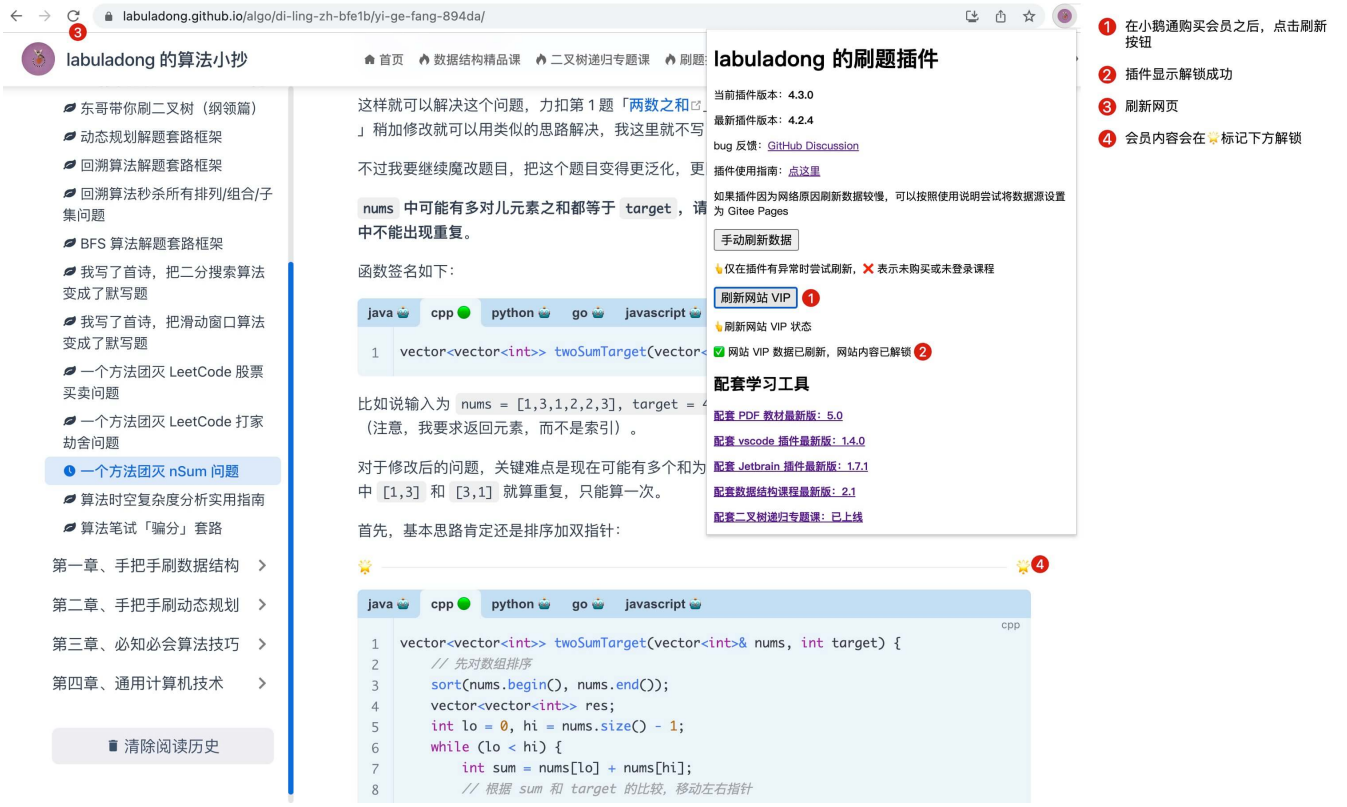
为什么要安装我的 Chrome 插件

建议我的读者都安装我的 Chrome 刷题插件，它的主要功能有两个：

1、在力扣和 LeetCode 的页面上添加「题解」和「思路」按钮，方便你跳转查看我的算法文章和解题思路：

labuladong 的 Chrome 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化

2、辅助 我的网站会员 在网页上学习，可以直接在网页上查看所有算法教程，享受代码图片注释、算法可视化动画等功能，不必跳转到小鹅通：



肯定有小伙伴问，力扣不是有题解和评论区吗？为啥还要安装你的这个插件呢？

首先，我建议你装插件，又没有说不让你学习别人的解法对吧，咱们多方学习，勤于思考，没毛病。

不过我的插件确实有它独特的价值。我先问一句，你是否有这样的困惑：看到新的算法技巧会大呼精妙，以为自己学会了，但下次换个题目，还是做不出来。

如果有，我告诉你为啥，因为人家给你看的只是最后的解法，一般不会给你讲清楚各个算法技巧之间的联系。

比方说，这个技巧 C 可能是技巧 B 和技巧 A 结合衍生出来的，人家只告诉你技巧 C 是什么，你直接从零开始接触 C 就会觉得这个思路不可能是普通人想出来的，甚至产生对算法的畏惧心理，更别说举一反三了。

但实际上，如果你能由浅入深，先了解 A 和 B，再接触 C，就不会觉得 C 有多了不起，你自己也可以推导出来，甚至你能随意变化 C 去解决更多新的的算法题。

我的文章和思路是经过精心设计浑然一体的，每道题的思路中会引用所有相关的技巧，插件可以保证你看到一道题，都可以寻根溯源由浅入深地学习到对应的算法技巧以及思维方法。

比如「丑数 II」这道题目：

乍一看这像是一道数学题，但实际上它掺杂了 高效寻找素数 中讲到的筛数法和 单链表的六大解题套路 讲到的单链表合并的技巧，我在插件中不仅给出了文章的链接，还给出了对应题目的链接，你可以去点击应题目的「思路」按钮对比题目之间的相似之处。

另外，很多小伙伴不是说看到题目没有思路，不知道从何下手吗？我会如实地分享我的思考过程，以及我这么想的依据是什么。虽然少数时候我的方法不是最优的，但我的方法一定是最「模板化」的。这种模板化的思考过程能够大幅提高初学者学习算法的效率：

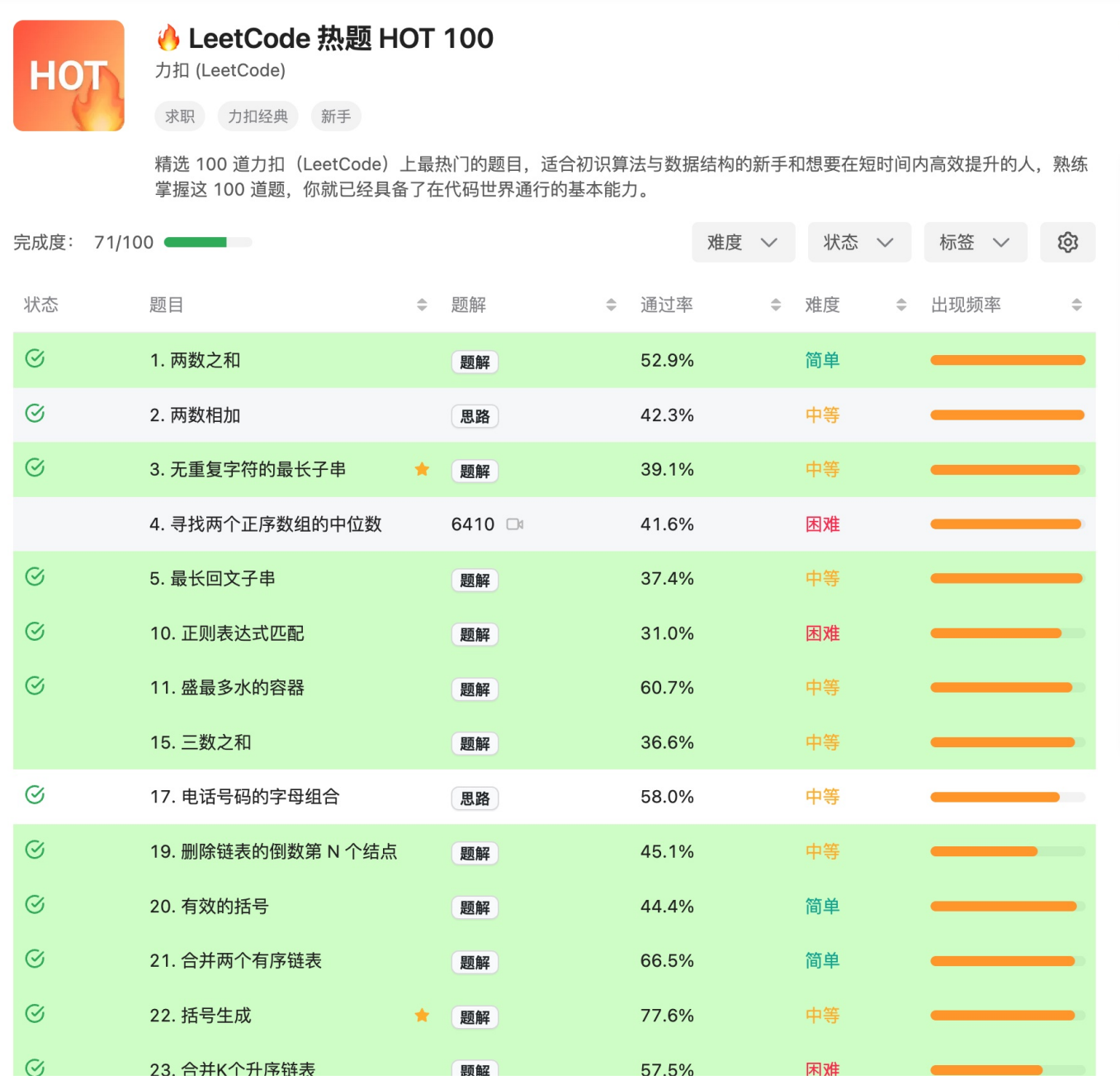
这就是我的刷题插件独有的价值，相当于我给你打造了一套完整的刷题工具链，如果你认可我算法教程的质量，那么使用我开发的配套工具，显然能给你提供最好的学习体验。

功能展示

总览

题目列表渲染

力扣 or LeetCode 的所有题目列表和题目详情页中会显示我的题解/思路按钮，支持所有常用编程语言：



点击「题解」按钮即可跳转到网站对应文章学习，点击「思路」按钮即可查看我精心撰写的简明解题思路和代码。

同时，我把公众号里所有我讲过的题目整理出一份列表，安装插件后访问可以看到效果：

力扣版	LeetCode 版
https://leetcode.cn/problem-list/59jEaTgw/	https://leetcode.com/list/9zwo3ww5/

思路/题解辅助刷题

题目详情页也会显示题解和思路按钮，可以直接复制带详细注释的代码，刷题非常方便：

代码图片注释

对于比较复杂的解法，代码中会包含图片注释，方便理解代码逻辑。鼠标移动到小灯泡图表即可查看图片注释：

889. 根据前序和后序遍历构造二叉树 labuladong 题解 思路

难度 中等 292

给定两个整数
无重复 值的
二叉树。

如果存在多个

示例 1：

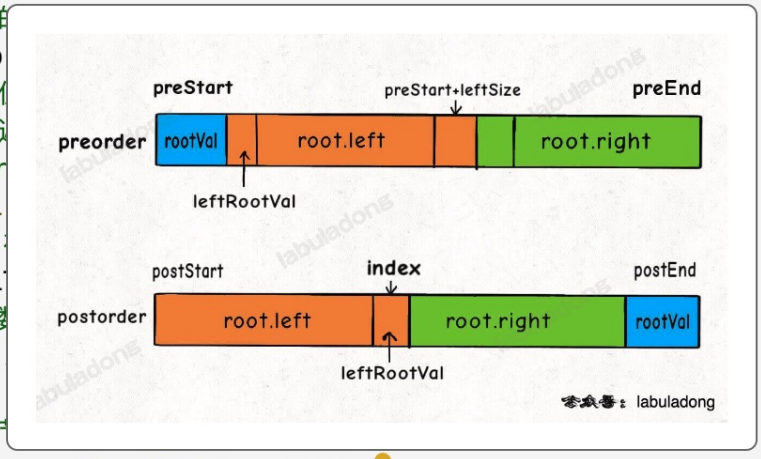
输入：pr
[4,5,2,
输出：[1

示例 2：

输入：preorder = [1], postorder = [1]
输出：[1]

bug 反馈 | 使用指南 | 更多配套插件 [x]

```
// root 节点对应的
int rootVal = p
// root.left 的个
// 通过前序和后序
// 确定 preorder
int leftRootVal
// leftRootVal
int index = val
// 左子树的元素个数
int leftSize =

// 先构造出当前根节
TreeNode root = new TreeNode(rootVal);
// 递归构造左右子树
// 根据左子树的根节点索引和元素个数推导左右子树的索引边界
root.left = build(preorder, preStart + 1, preStart + leftSize,
        postorder, postStart, index);
root.right = build(preorder, preStart + leftSize + 1, preEnd,
        postorder, index + 1, postEnd - 1);

return root;
}
```



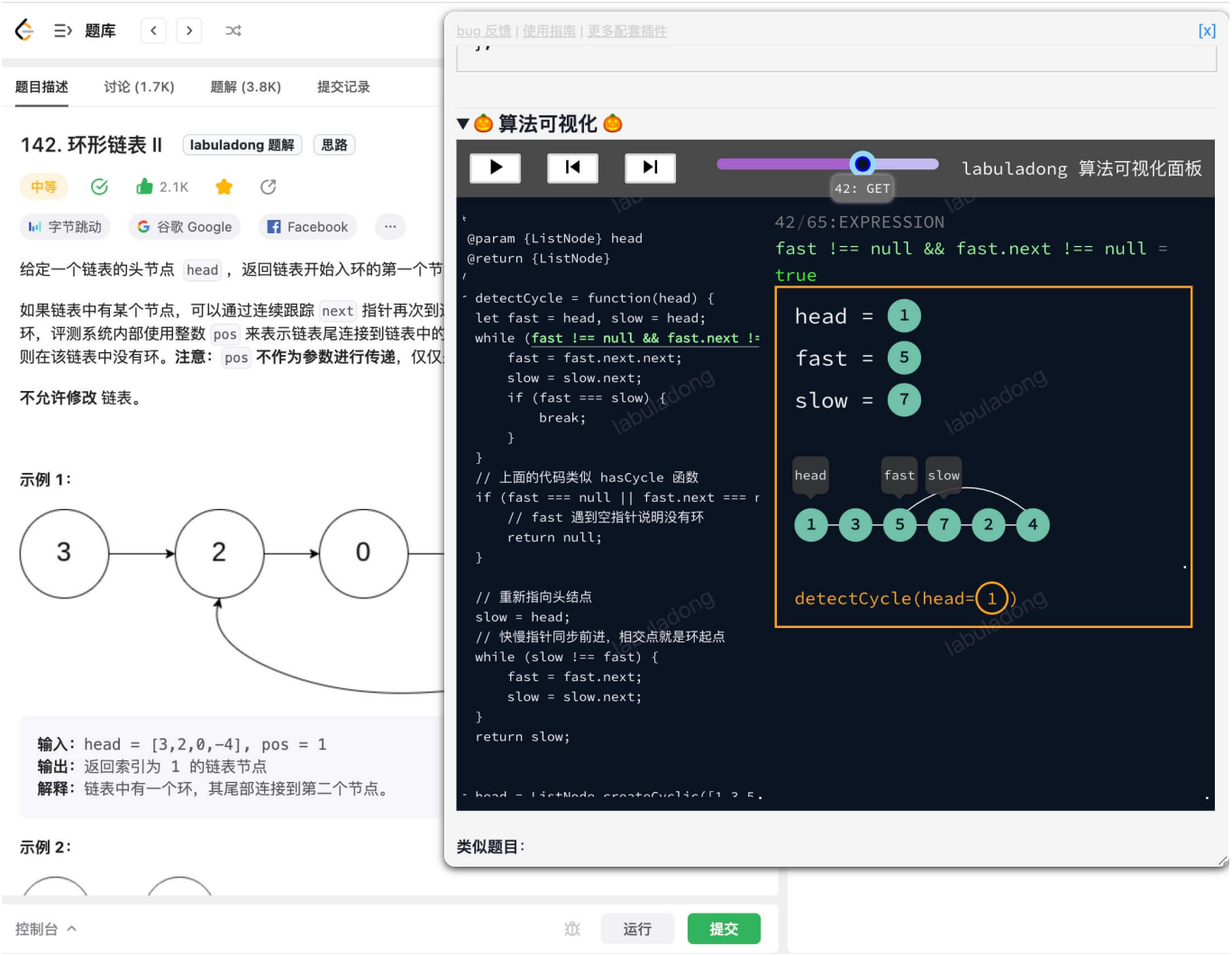
类似题目：

- 105. 从前序与中序遍历序列构造二叉树
- 106. 从中序与后序遍历序列构造二叉树

```
1 /**
2  * Definition for
3  * public class Tr
```

算法可视化面板

「思路」展开后，解法代码下方会显示一个可交互的算法可视化面板，帮助理解算法的执行过程：



可视化面板的具体功能请参考 算法可视化面板简介。

安装方式

Chrome 刷题插件不止支持 Chrome 浏览器，像 Edge 浏览器、360 浏览器这些使用 Chromium 内核的浏览器都可以通过离线安装 crx 文件的方式安装。

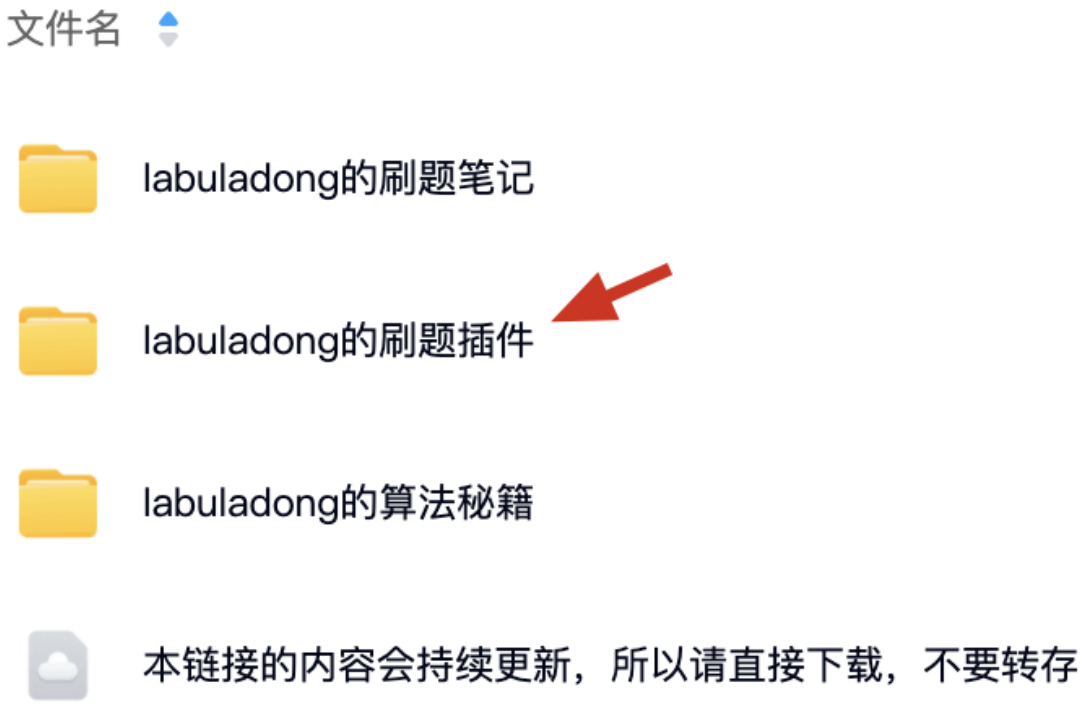
Chrome 浏览器用户有条件的话建议直接在 Chrome 商店下载：

https://chrome.google.com/webstore/detail/leetcode-helper-by-labula/elafhogmnaapleckojedgipgmidneccg

Edge 浏览器用户可以直接在 Edge 商店下载（国内也可以访问）：

https://microsoftedge.microsoft.com/addons/detail/leetcode-helper-by-labula/mgfjpejofejdbnillfolnnjbiefpokln

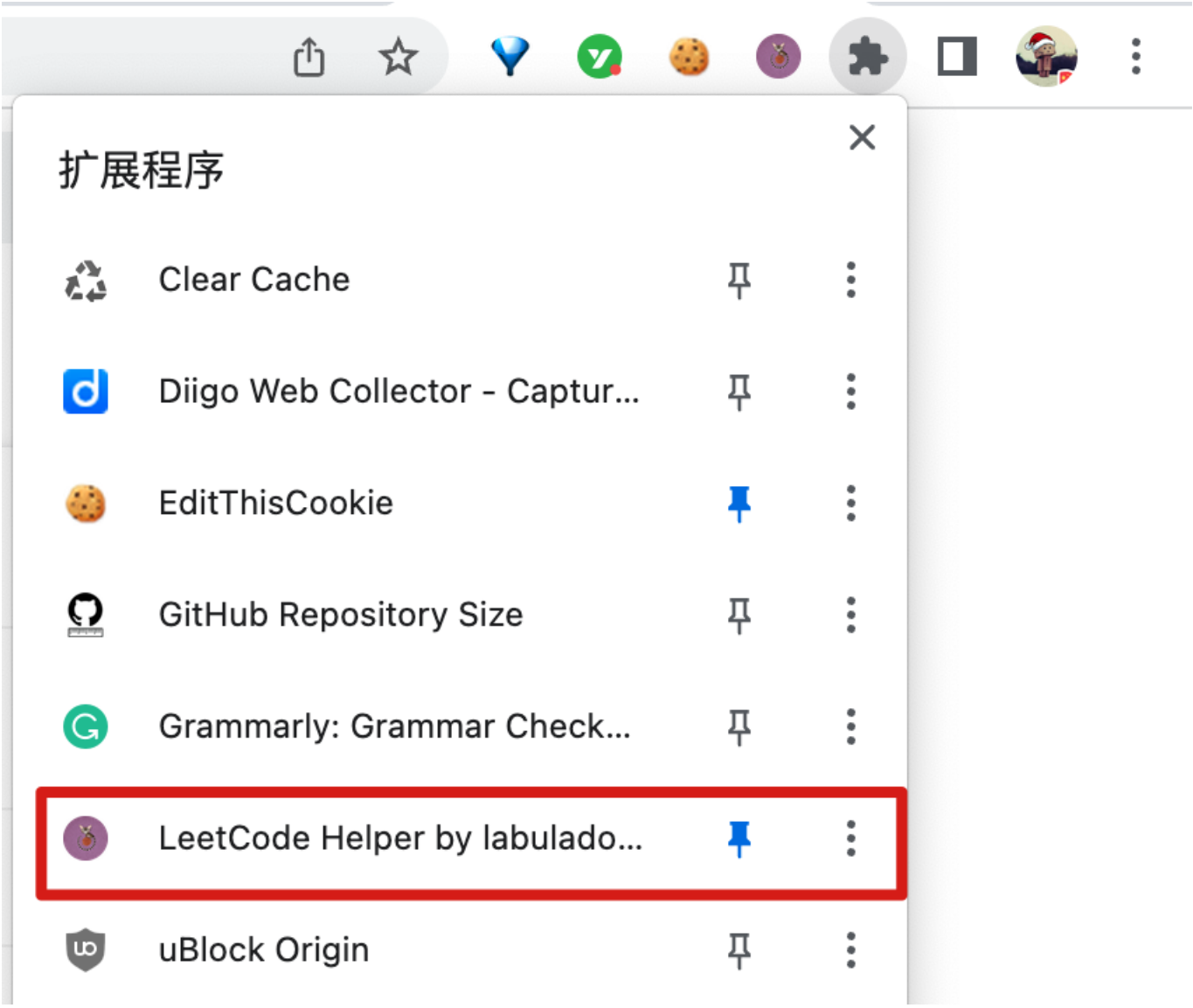
如果无法在线安装，也可以离线安装。在公众号后台回复关键词「插件」下载最新版本 crx 文件：



有了 crx 插件文件，手动安装的方法在网上一搜一大把，我就随便贴一个吧：

https://cloud.tencent.com/developer/article/1894180

安装成功后，可以在插件列表看到插件图标：



点击插件图标可以弹出插件弹窗，包含刷新数据的按钮和很多有用的链接：

更新方式

1、在 Chrome/Edge 商店安装的，会在新版本发布后自动更新。

2、通过 crx 文件离线安装的，我会在发布新版本之后更新百度网盘中的 crx 文件，所以你只需要重新在公众号后台回复关键词「插件」即可下载最新版本插件，再次安装即可。建议将插件固定在插件栏，有更新时会有 new 的提示字样。

> 任何时候，都推荐你使用最新版本的插件，因为我会不断优化插件的功能，修复 bug，所以如果你遇到了问题，也许更新到最新版本就能解决。

更新日志

详见 Chrome 插件更新日志。

插件配置方法

点击插件图标打开插件弹窗，下滑即可看到配置选项：

况，可以在新网站的 订单列表 查看。

插件配置

-> 题目列表的显示设置

有「题解」的题目 渲染为绿色

有「思路」的题目 不渲染为绿色

-> 快捷键设置（配置格式见这里）

在题目详情页打开/关闭「思路」弹窗

ctrl+i 确认

在题目详情页直接复制我的解法代码

ctrl+, 确认

其他配套学习工具

配套 PDF 教材最新版：5.2

配套 vscode 插件最新版：1.4.5

配套 Jetbrain 插件最新版：1.7.5

配套数据结构课程最新版：2.1

列表渲染设置可以选择是否把题目列表中有题解/思路的题目行渲染成绿色。

在题目详情页，可以使用自定义快捷键快速打开/关闭思路弹窗和复制解法代码。

常见问题的回复

安装插件后没有效果?

主要原因可能是网络问题，导致插件拉取初始数据时较慢。可以稍等一会儿，或者点击「手动刷新数据」按钮：

等待数据拉取完成后再尝试刷新网页，应该就能看到题解和思路按钮了。

无法成功刷新插件数据（显示❌ 标记）？

如果点击「手动刷新数据」也没有反应或全部显示❌ 标记，大概率是你的网络问题。「手动刷新数据」按钮右侧有一个下拉框，默认选项为 `Auto`，即插件会根据网速自动选择 GitHub 或 Gitee 拉取数据。

如果你的网络情况不稳定，可以尝试自己修改下拉框选项，选择 GitHub 或 Gitee，然后再点击「手动刷新数据」按钮，直到出现 ✅ 标记为止。

火狐浏览器可以安装吗？

不支持，因为火狐浏览器用的不是 chromium 内核。以前还有办法安装，但现在 chromium 内核接口升级了，所以除非火狐官方适配，否则无法安装 Chrome 插件。

macOS/Linux/Windows 都能安装吗？

是的，插件是寄生在浏览器里的，和操作系统无关。只要在你的操作系统安装 chromium 内核的浏览器，就能够正常安装使用插件。

插件会「读取和更改我在 xxx 网站上的数据」？

实际上插件不会读取你的任何账户信息，而是插件需要读取该网站的页面 HTML，结果 Chrome 浏览器给了这么一句很容易让人误会的提示语。

借此机会科普一下，Chrome/Edge 商店上架都要经过严格的审核的，如果有恶意插件，根本上不了架，要知道国外对隐私的保护可是比国内强得多。

另外，关于权限申请这块我是很规范的，仅仅申请几个必须的网站权限，所以如果更新的时候需要新权限，Chrome 都会提醒你；不规范的做法是直接用通配符 `https://*` 申请所有网站的权限，这样每次需要新权限的时候也不会提醒你，但回头你也搞不清这个插件会在哪些网站上干什么事情。

所以下载插件一定要从正规的渠道下载，我的刷题插件的正规下载渠道目前只有 Chrome/Edge 商店，或者通过公众号后台的链接下载 crx 离线安装，其他的渠道很可能有安全问题。

bug 反馈

请转到我的 GitHub Discussion，查看一些常见问题的解决方案，如确定遇到 bug，按规则截图反馈：

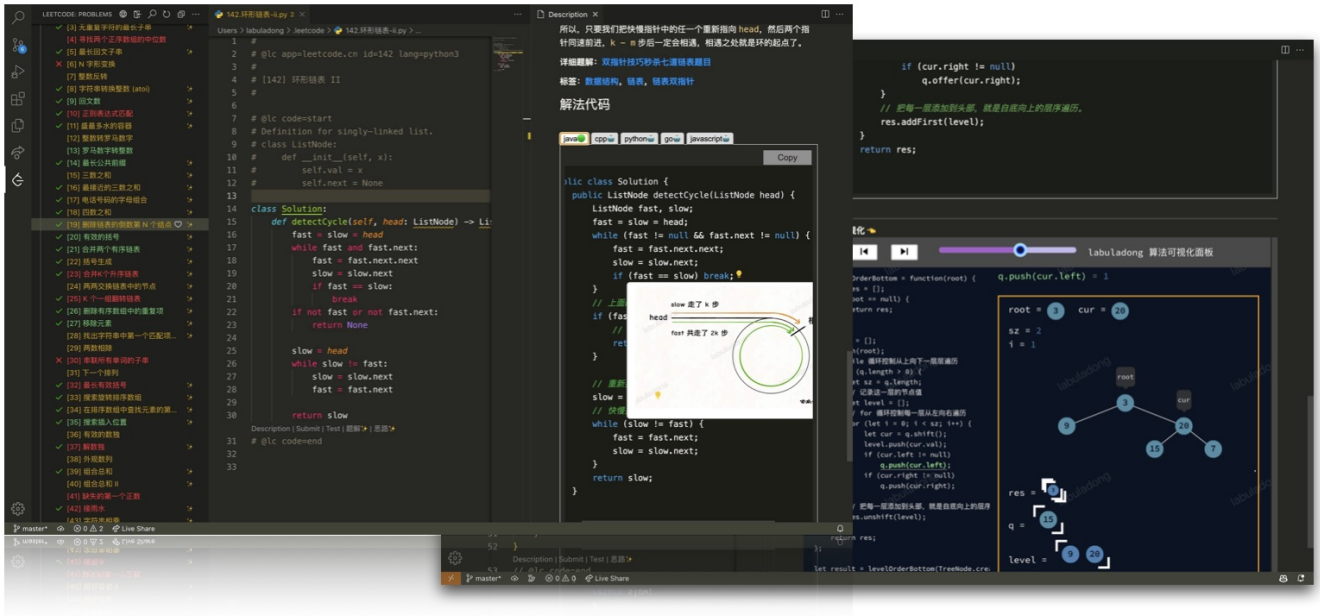
https://github.com/labuladong/fucking-algorithm/discussions/704

vscode 刷题插件

> labuladong 的刷题全家桶 由学习网站、《算法秘籍》《刷题笔记》两本 PDF、Chrome/vscode/Jetbrain 三个不同平台的刷题插件组成，致力于为大家提供最丝滑的刷题体验。公众号后台回复关键词「**全家桶**」即可获取下载和使用手册。

由于我的 Chrome 刷题插件 收到很多小伙伴的好评，不过有的小伙伴反馈不喜欢在网页刷题，所以我就开发了 vscode 刷题插件，提供和 Chrome 插件类似的功能，大家可以在 vscode 中刷题。

labuladong 的 vscode 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



视频介绍

最近录了一个视频介绍 vscode 插件的主要功能，喜欢看视频的读者可以去 B 站观看：

使用方法

基础功能

本插件是我基于 LeetCode 官方的开源插件改写的：

https://github.com/LeetCode-OpenSource/vscode-leetcode

所以本插件包含官方插件所有的功能，比如 LeetCode/力扣 账号的登录、题目搜索、代码的测试和提交等，中文文档见这里：

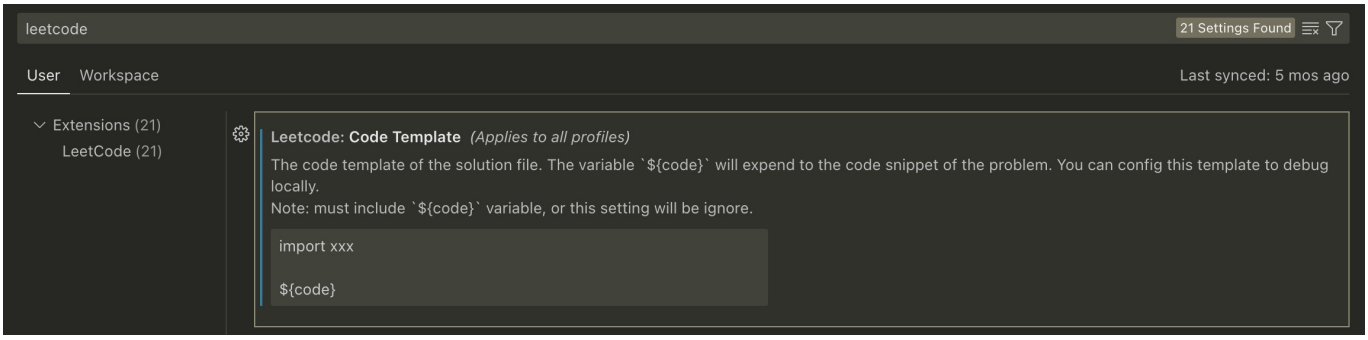
https://github.com/LeetCode-OpenSource/vscode-leetcode/blob/master/docs/README_zh-CN.md

官方插件的问题是，对英文版 LeetCode 适配比较好，但是对中文力扣的支持一般，而且好久都不维护了，连登录都会报错。

我把这些 bug 全部修复了，而且额外添加了一些好用的功能。

自定义代码模板

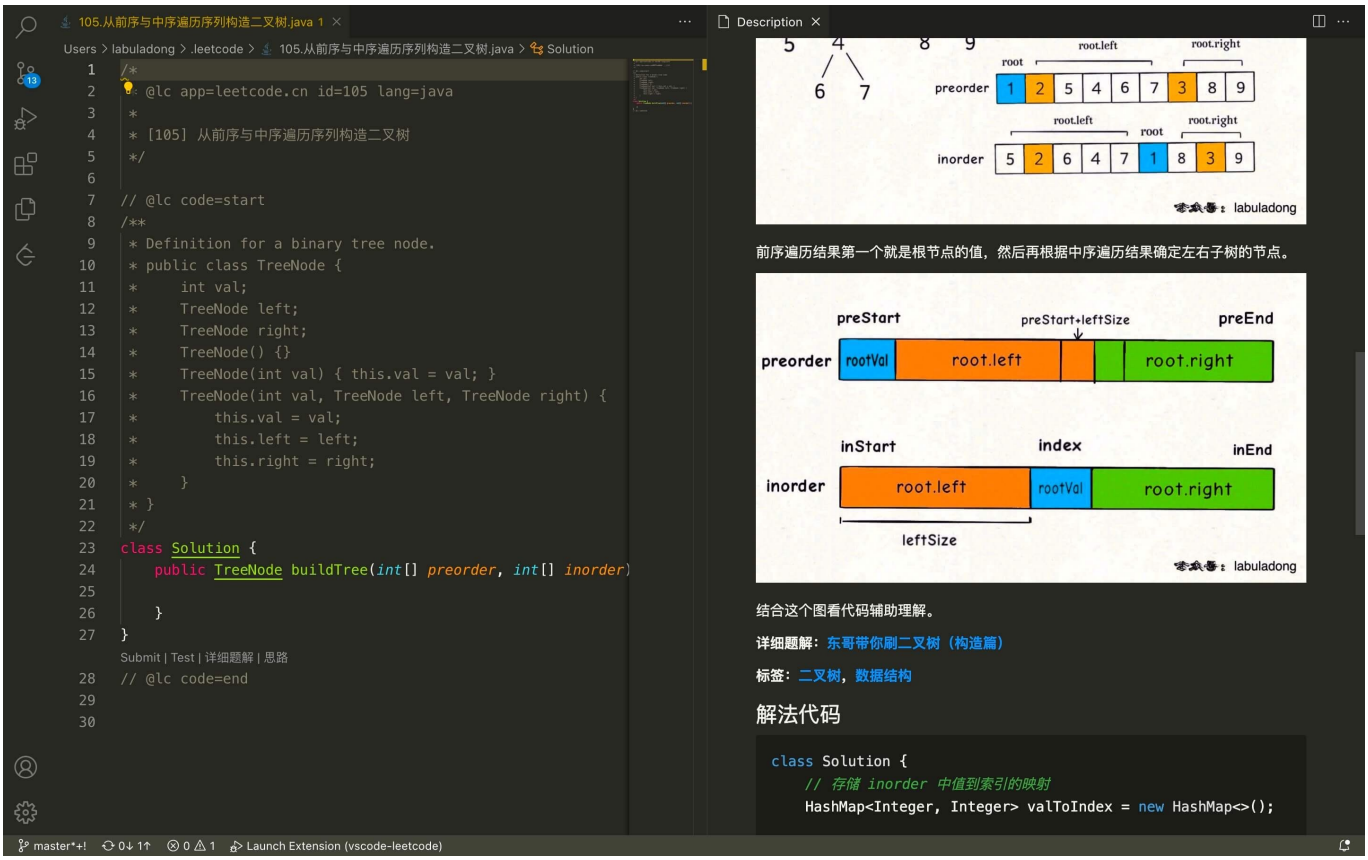
首先一个实用功能是可以配置代码模板，这样你就可以很方便地在本地编译和调试代码了：



题解/思路辅助刷题

另外一个重要的功能，是方便读者查看我的解题思路和详细题解。题目列表中带有 ✨ 标记的题目都是我在公众号讲解过的，可以查看题解或者思路：

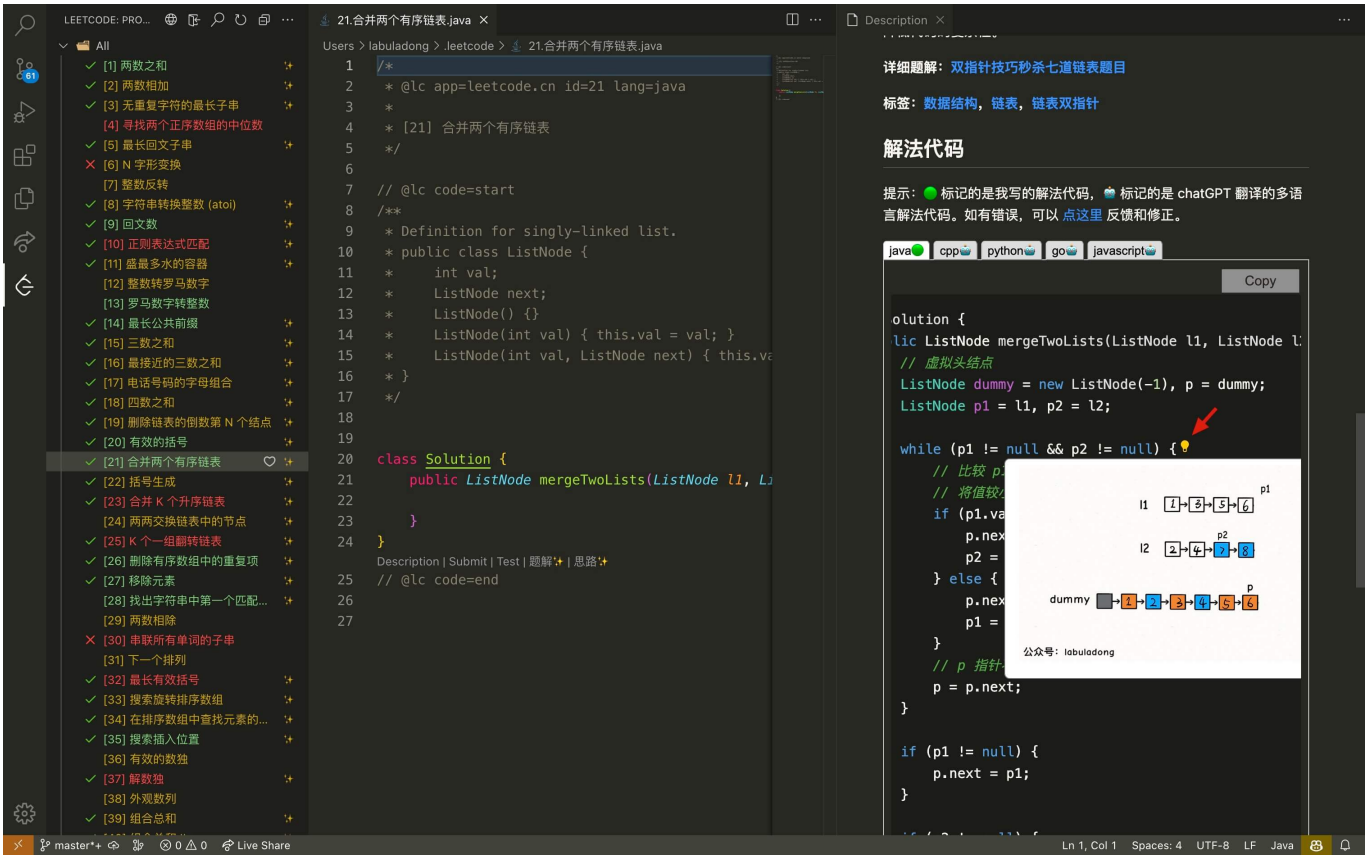
你可以一边写代码一边查看我的思路讲解：



我自己一直在坚持刷算法题并总结解题套路，所以插件中标 ✨ 的题目将会越来越多，Chrome 插件和 vscode 插件都会同步更新。

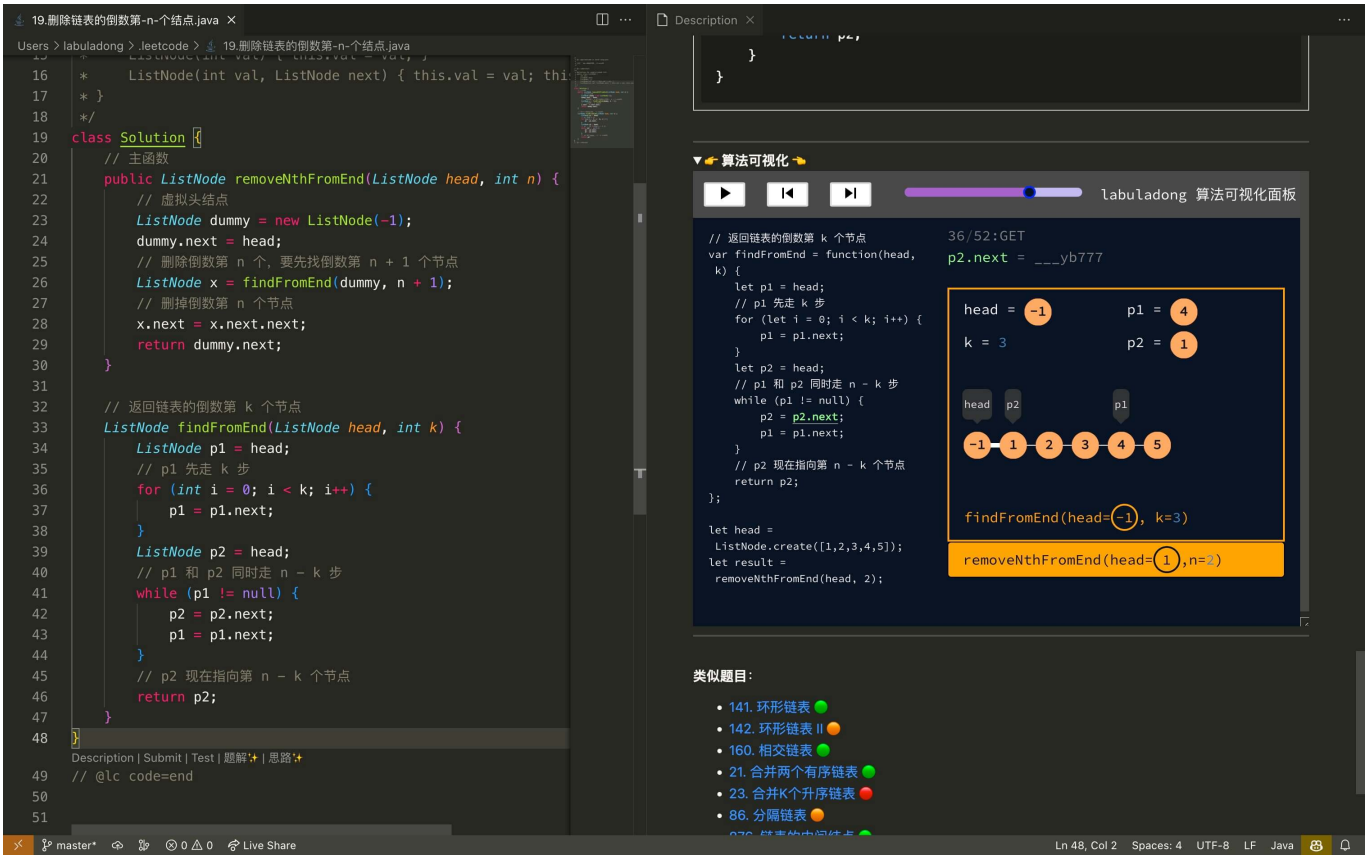
代码图片注释

「labuladong 思路」中的代码支持直接复制，且代码中的小灯泡图标会弹出图片辅助理解代码逻辑：



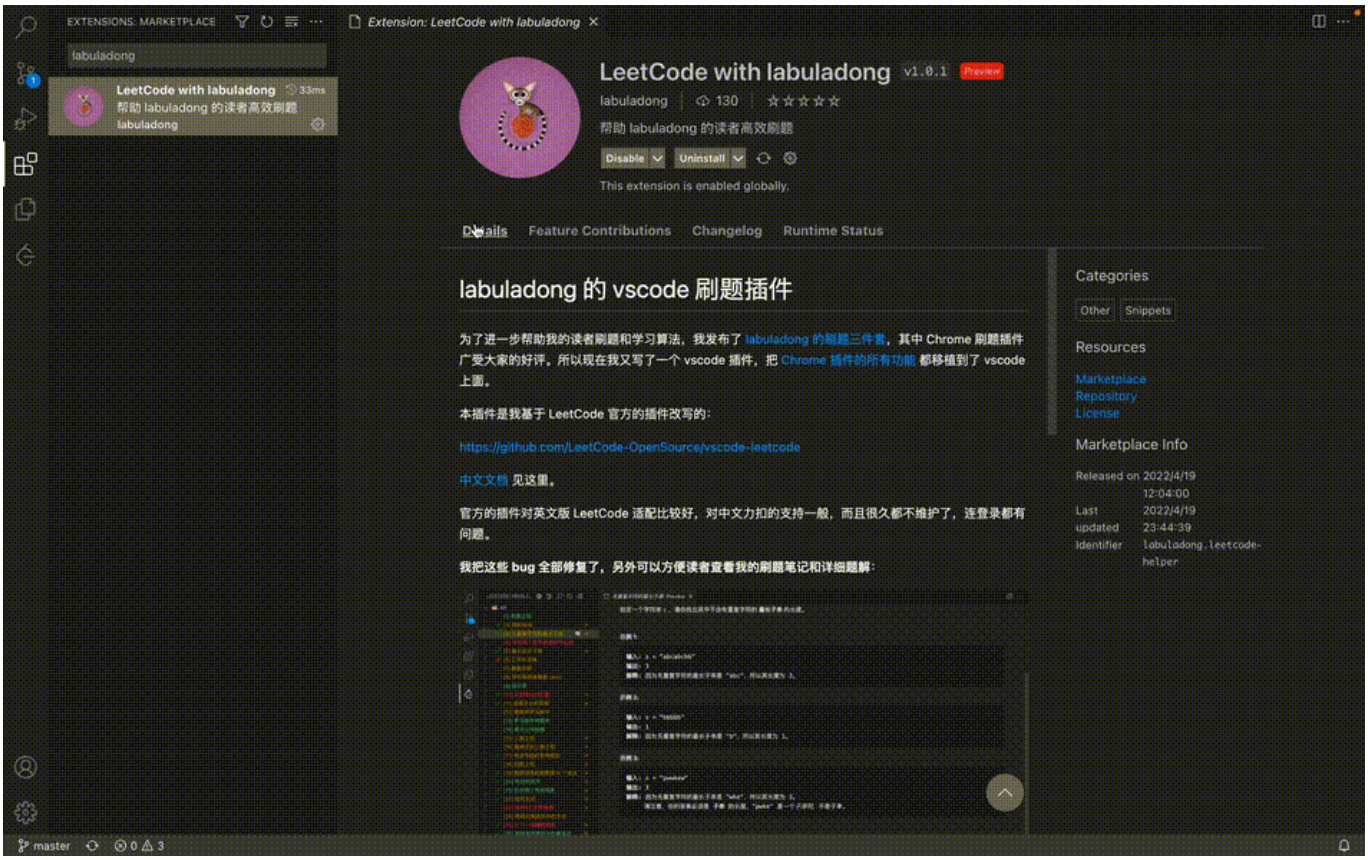
算法可视化面板

「labuladong 思路」展开后，解法代码下方会显示一个可交互的算法可视化面板，帮助理解算法的执行过程：



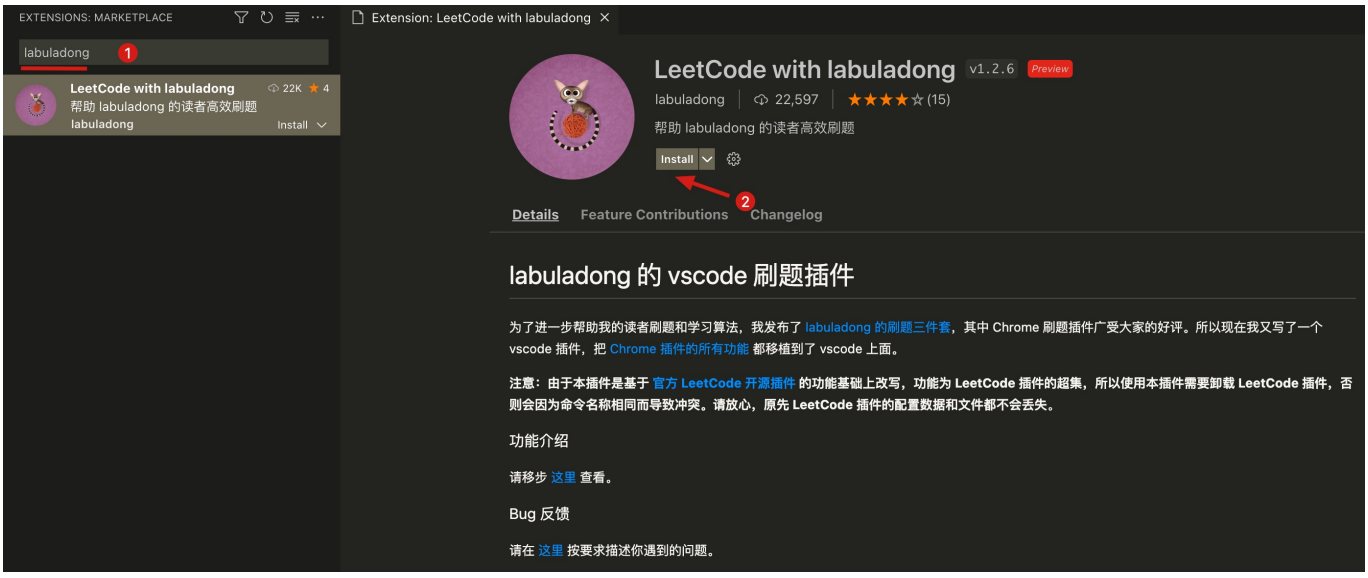
可视化面板的具体功能请参考 算法可视化面板简介。

用一个简短的 GIF 看下这个插件有多好用：



安装方法

微软插件商店在国内也可以正常访问，直接在 vscode 的插件商店中搜索关键词「labuladong」即可搜到插件，点击安装即可：



如果搜不到，可以从 vscode 插件商店的网页安装：

https://marketplace.visualstudio.com/items?itemName=labuladong.leetcode-helper

如果你已经安装了 LeetCode 官方插件，需要先卸载。因为本插件基于官方插件改写，同时开启的话会因为共用了相同的配置而产生冲突。请放心，因为我的插件功能是官方插件的超集，所以卸载原插件也完全不会影响之前的插件配置和代码文件。

更新方式

默认情况下，vscode 会自动检测和更新插件。

更新日志

详见 vscode 插件更新日志。

bug 反馈

如果你在使用过程中遇到问题，可以在我的 GitHub Discussion 中按要求进行反馈，我会定期处理和修复 bug：

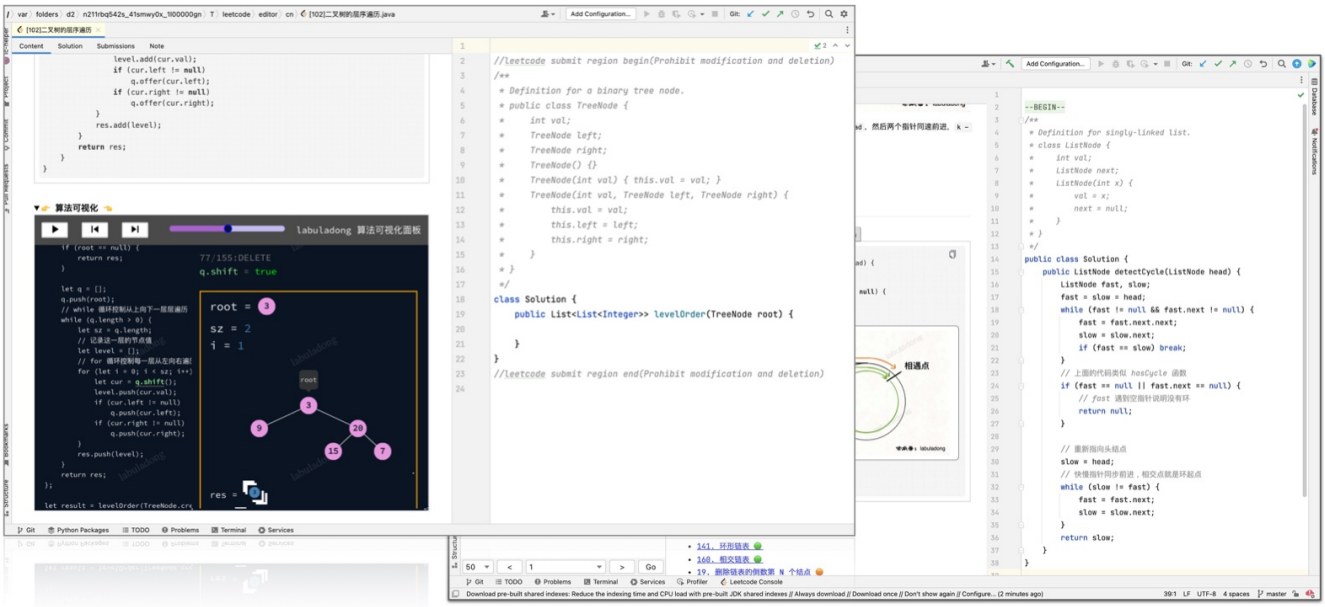
https://github.com/labuladong/fucking-algorithm/discussions/910

JetBrains 刷题插件

> labuladong 的刷题全家桶 由学习网站、《算法秘籍》《刷题笔记》两本 PDF、Chrome/vscode/Jetbrain 三个不同平台的刷题插件组成，致力于为大家提供最丝滑的刷题体验。公众号后台回复关键词「**全家桶**」即可获取下载和使用手册。

由于我的 Chrome 刷题插件 收到很多小伙伴的好评，不过有的小伙伴反馈不喜欢在网页刷题，所以我就开发了 JetBrains 刷题插件，提供和 Chrome 插件类似的功能，大家可以在 JetBrains 全家桶的所有 IDE（比如 Intellij，Pycharm 等等）中刷题。

labuladong 的 Jetbrains 刷题插件：添加「思路」「题解」按钮，支持算法代码的可视化



视频介绍

最近录了一个视频介绍 Jetbrains 插件的主要功能，喜欢看视频的读者可以去 B 站观看：

使用指南

基本功能

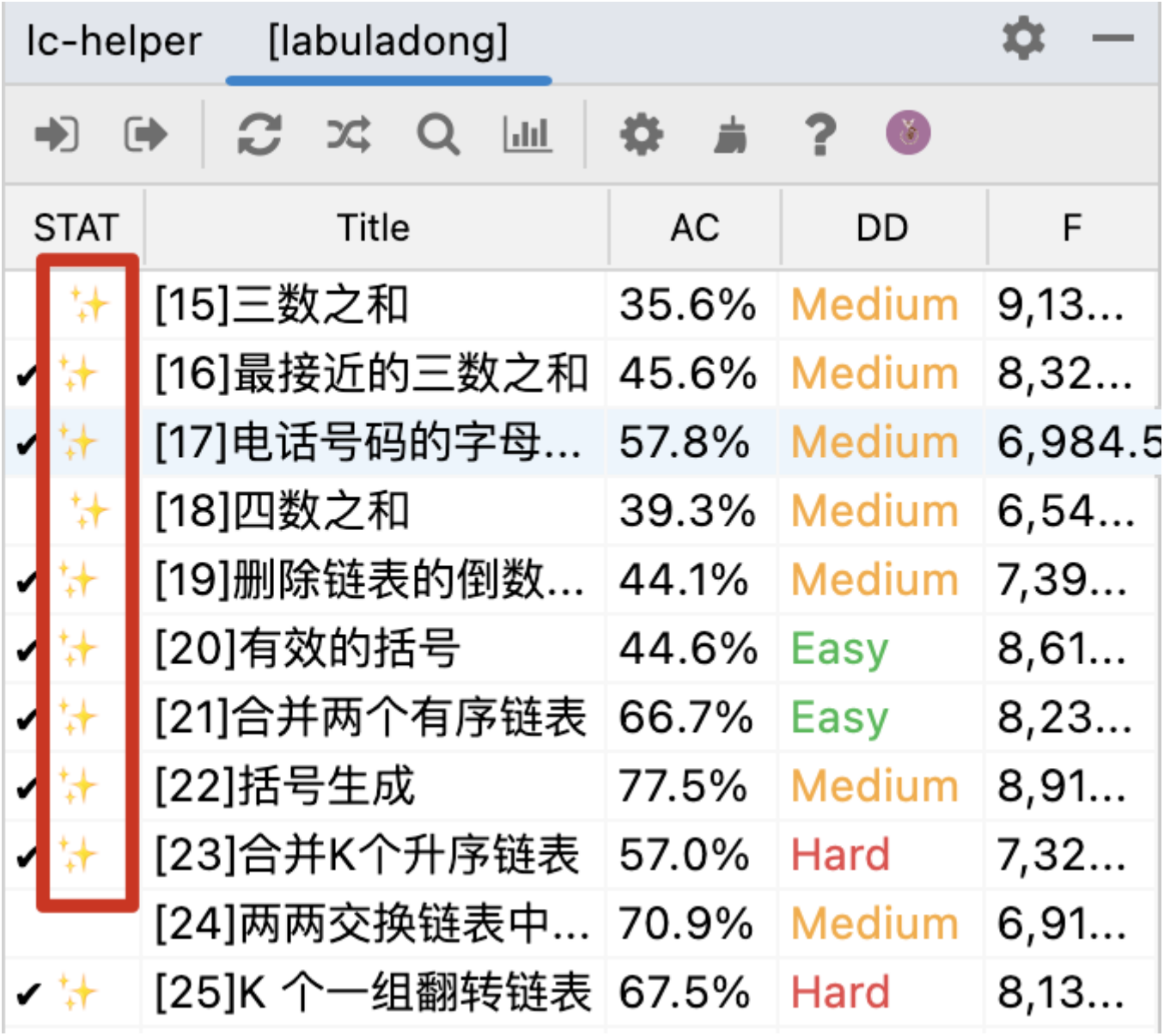
本插件基于开源插件 LeetCode Editor 开发，所以继承了该插件的所有功能，比如 LeetCode/力扣 账号的登录、题目搜索、代码的测试和提交、自定义代码模板等，说明文档如下：

https://github.com/shuzijun/leetcode-editor/blob/master/README_ZH.md

除了 LeetCode Editor 的基本功能，我还添加了很多实用的功能，下面简单介绍一下。

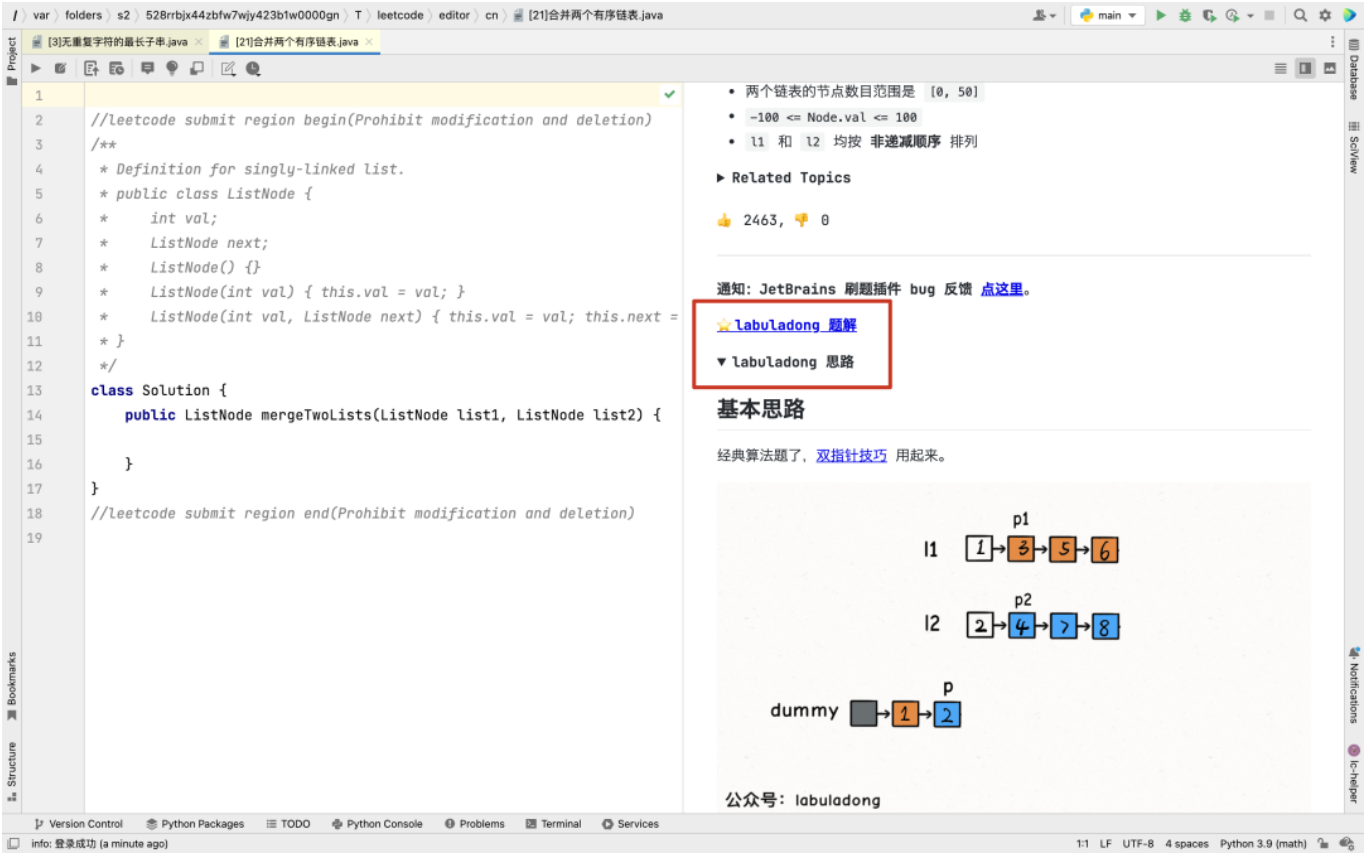
题解/思路辅助刷题

登录成功后，有我的题解或思路的题目会标记 ✨ ：



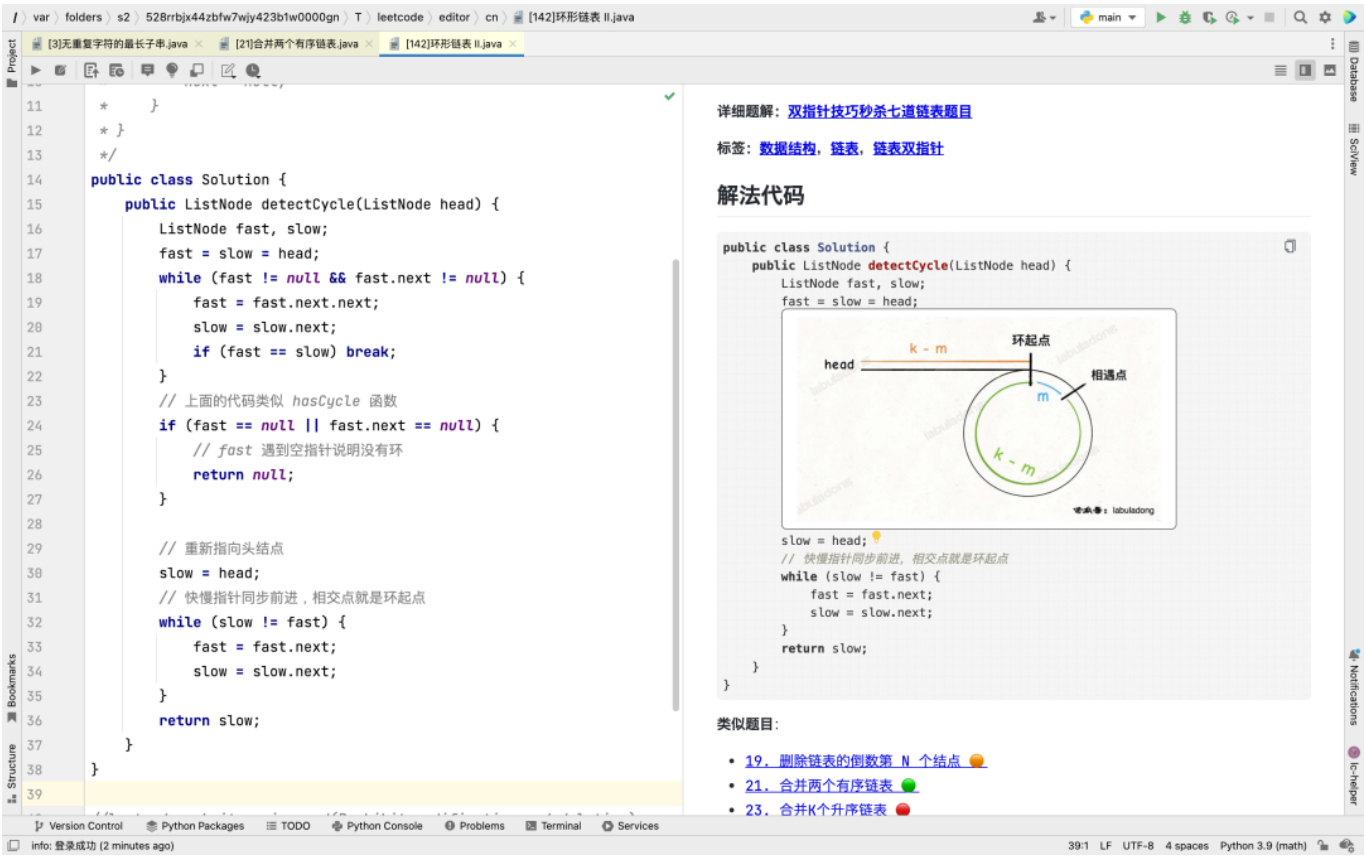
STAT	Title	AC	DD	F
✨	[15]三数之和	35.6%	Medium	9,13...
✔ ✨	[16]最接近的三数之和	45.6%	Medium	8,32...
✔ ✨	[17]电话号码的字母...	57.8%	Medium	6,984.5
✨	[18]四数之和	39.3%	Medium	6,54...
✔ ✨	[19]删除链表的倒数...	44.1%	Medium	7,39...
✔ ✨	[20]有效的括号	44.6%	Easy	8,61...
✔ ✨	[21]合并两个有序链表	66.7%	Easy	8,23...
✔ ✨	[22]括号生成	77.5%	Medium	8,91...
✔ ✨	[23]合并K个升序链表	57.0%	Hard	7,32...
	[24]两两交换链表中...	70.9%	Medium	6,91...
✔ ✨	[25]K 个一组翻转链表	67.5%	Hard	8,13...

点开带 ✨ 标记的题目详情页会显示「labuladong 题解」和「labuladong 思路」：



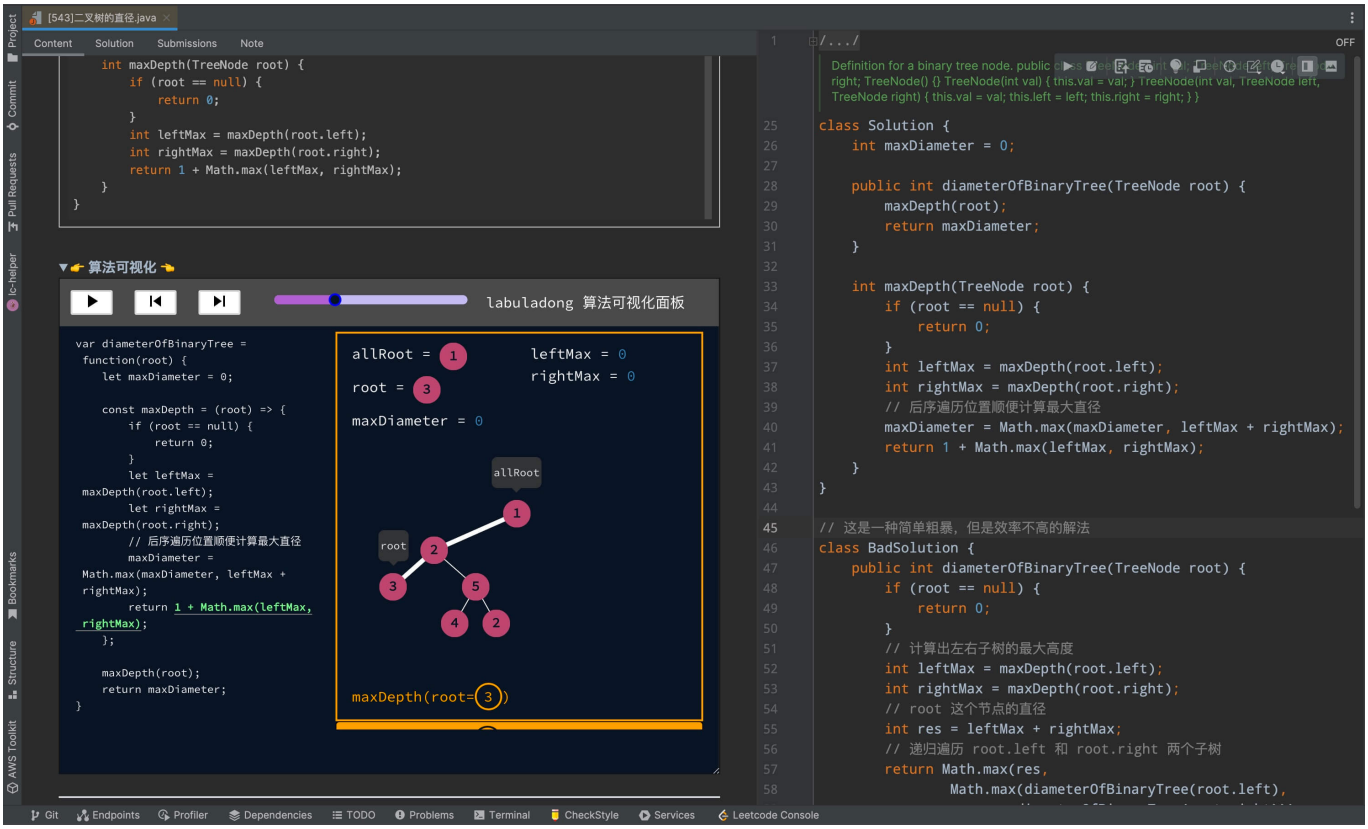
代码图片注释

「labuladong 思路」中的代码支持直接复制，且代码中的小灯泡图标会弹出图片辅助理解代码逻辑：



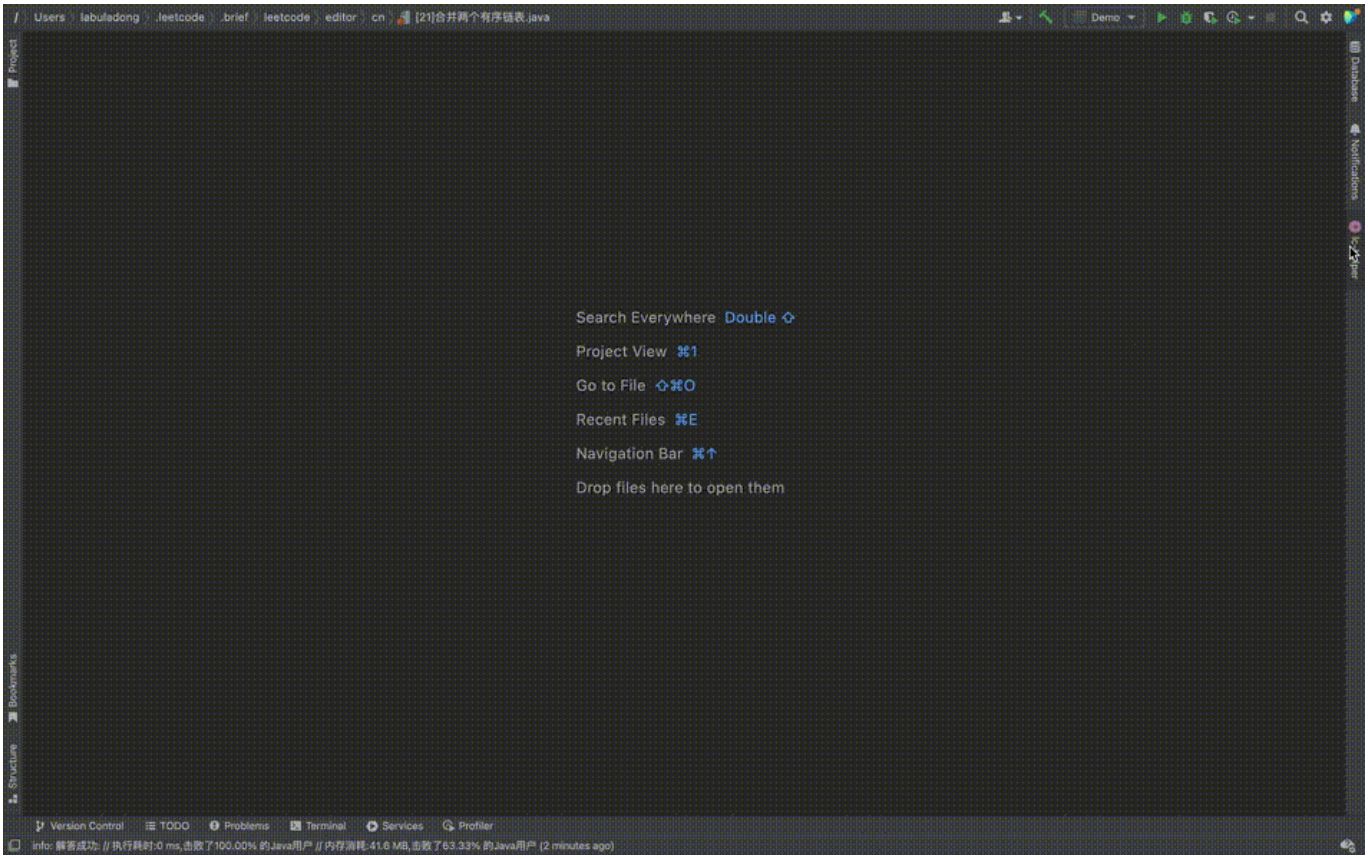
算法可视化面板

「labuladong 思路」展开后，解法代码下方会显示一个可交互的算法可视化面板，帮助理解算法的执行过程：



可视化面板的具体功能请参考 算法可视化面板简介。

用一个简短的 GIF 看下这个插件有多好用：



常见问题

如何本地调试代码

你写算法题的文件就是一个代码文件，在对应的 IDE 里面肯定是可以本地调试的。

比方说我在 Intellij 用 Java 刷题，除了题目给的 Solution 类外，我需要自己写一个类，比方说名为 Test，然后在其中添加 main 函数，然后构造测试用例，调用执行 Solution 类里面的算法函数，这样就可以本地调试了。

> 你可以把 Test 类、main 函数、`TreeNode, ListNode` 这种公用的代码实现统一放到 Code Template 里面，这样每次新打开题目的时候，这些代码就会自动添加到代码文件中，不用你每次手动添加了。

中文乱码?

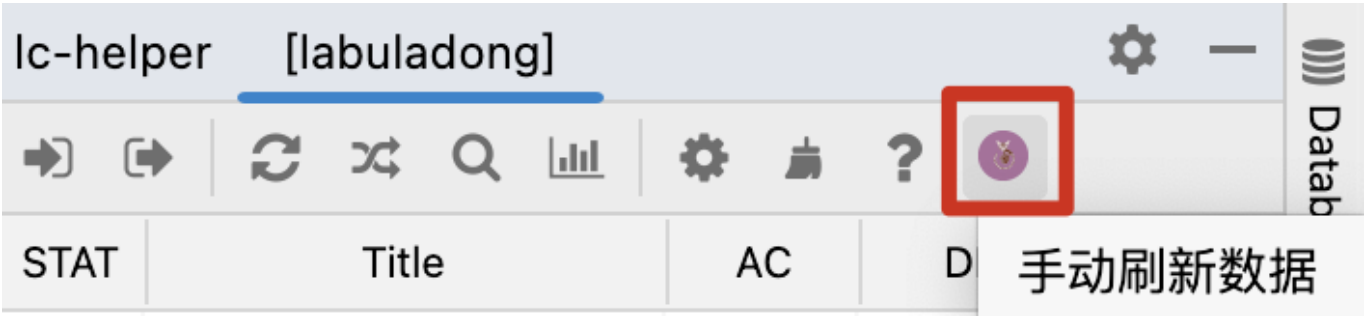
对于比较新的 IDE 版本和操作系统来说，一般不会出现这个问题。如果出现了，可以参考 这个帖子 修改 IDE 的编码为 utf-8。

题目页面会突然变成空白?

你的 IDE 是否已经安装使用了很久？只有 IDE 重度使用者才会遇到这个 bug，原因很难排查，大概是 IDE 升级过程中的某些缓存/内部配置的问题导致的。一个最简单的解决办法是：去官网重新下载最新版 IDE，覆盖安装本地的 IDE（不用担心，并不会覆盖已有的配置），这样一来所有问题都消失了。

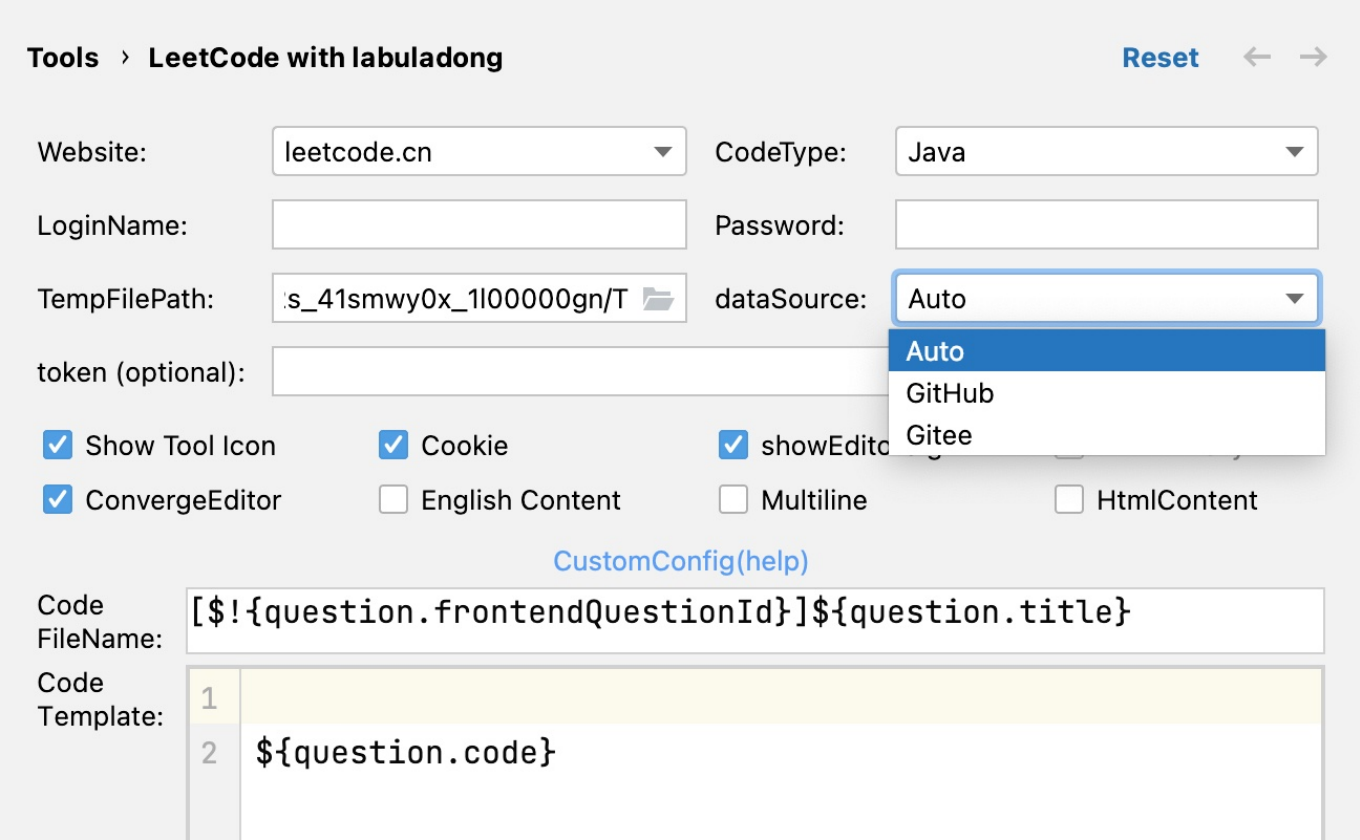
题目列表没有 ✨ 标记?

如果登录账号超过一分钟，题目列表还没有出现 ✨ 标记，可以点击工具栏最右侧的公众号图标手动刷新 labuladong 的题解数据：



等待十几秒左右就会看到通知栏显示「手动刷新 labuldong 数据成功」，✨ 标记就会出现。

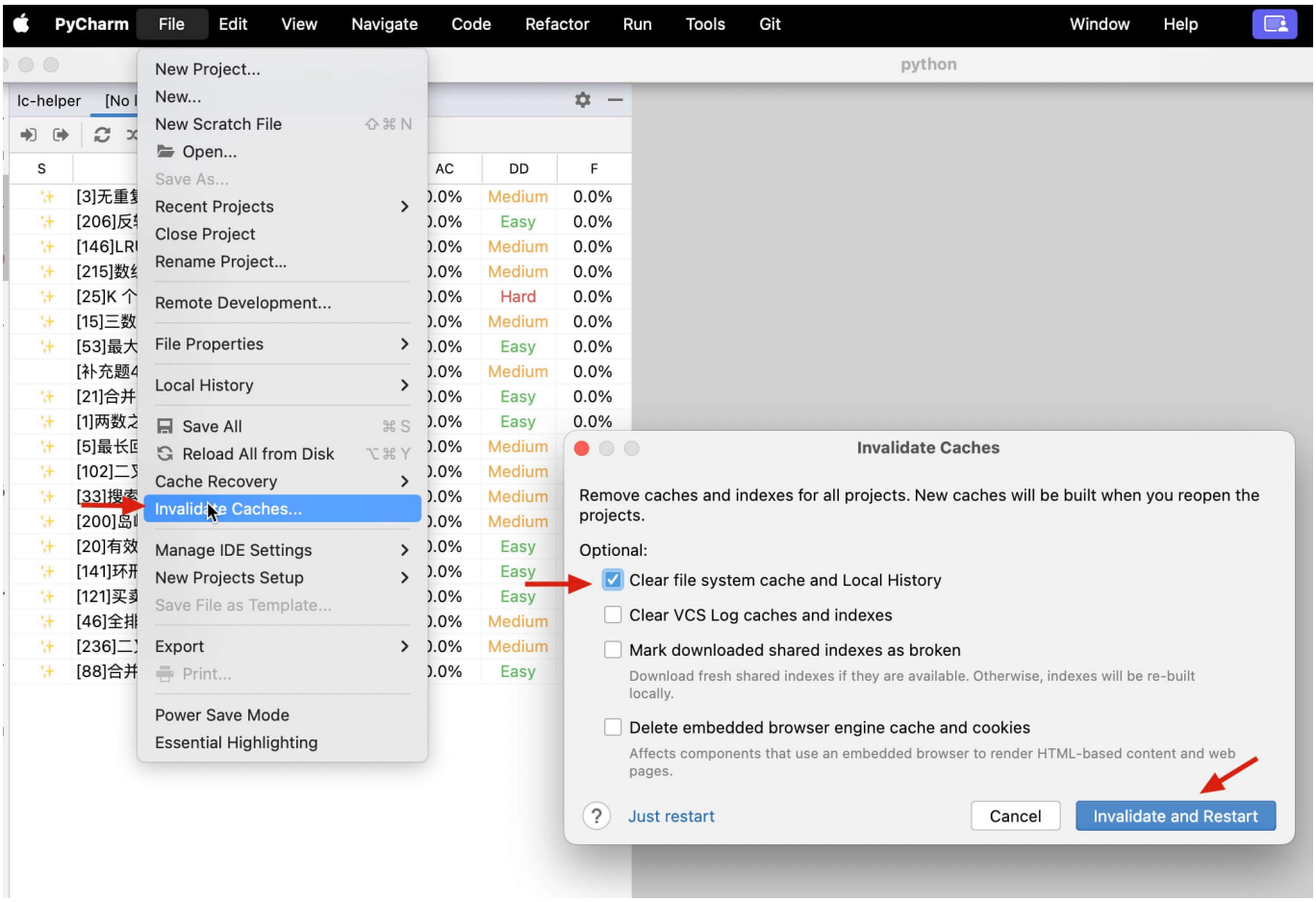
如果依然没有出现，可能是网络问题。因为默认优先去 GitHub 上拉取数据，国内访问可能不稳定，可以在设置中修改 `dataSource` 到国内的 Gitee 源：



有 ✨ 标记但是没有思路和题解按钮?

如果显示数据拉取成功，但是题目详情页没有「labuladong 题解」和「labuladong 思路」按钮，是因为 IDE 的文件系统缓存没有刷新。

你可以尝试手动清除 IDE 的缓存并重启 IDE：



没有代码自动补全?

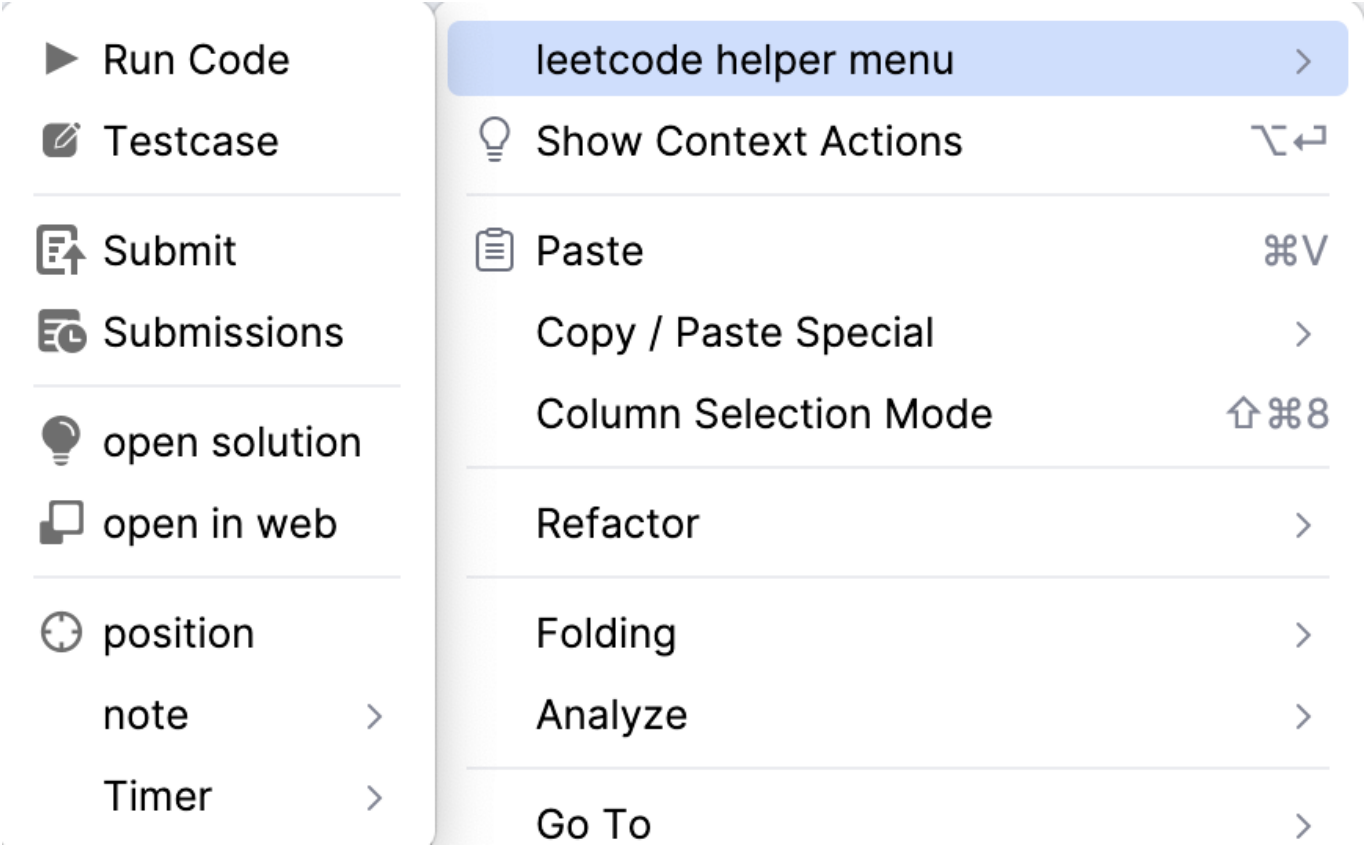
代码补全是 IDE 的基础功能，和插件无关。如果没有代码补全，可能是你的 IDE 没有配置好，或者是你的代码文件无法被 IDE 识别。

比如对于 Go 语言来说，需要在代码文件的第一行加上 `package main` 才能被 Goland IDE 识别，你可以使用 Code Template 功能来自动添加这一行。

类似的，再比如 Java 文件，需要在一个项目中才能被 IDE 识别。你可以把 `TempFilePath` 设置为你的项目路径，并设置 Code Template 自动添加 package 名称，这样代码文件就会被保存到项目中，IDE 就能识别并给出代码补全了。

没有代码提交按钮?

目前插件没有很好地适配新 UI，如果你使用的 Jetbrain 新 UI，可能找不到代码提交按钮。你可以使用右键进行代码提交：



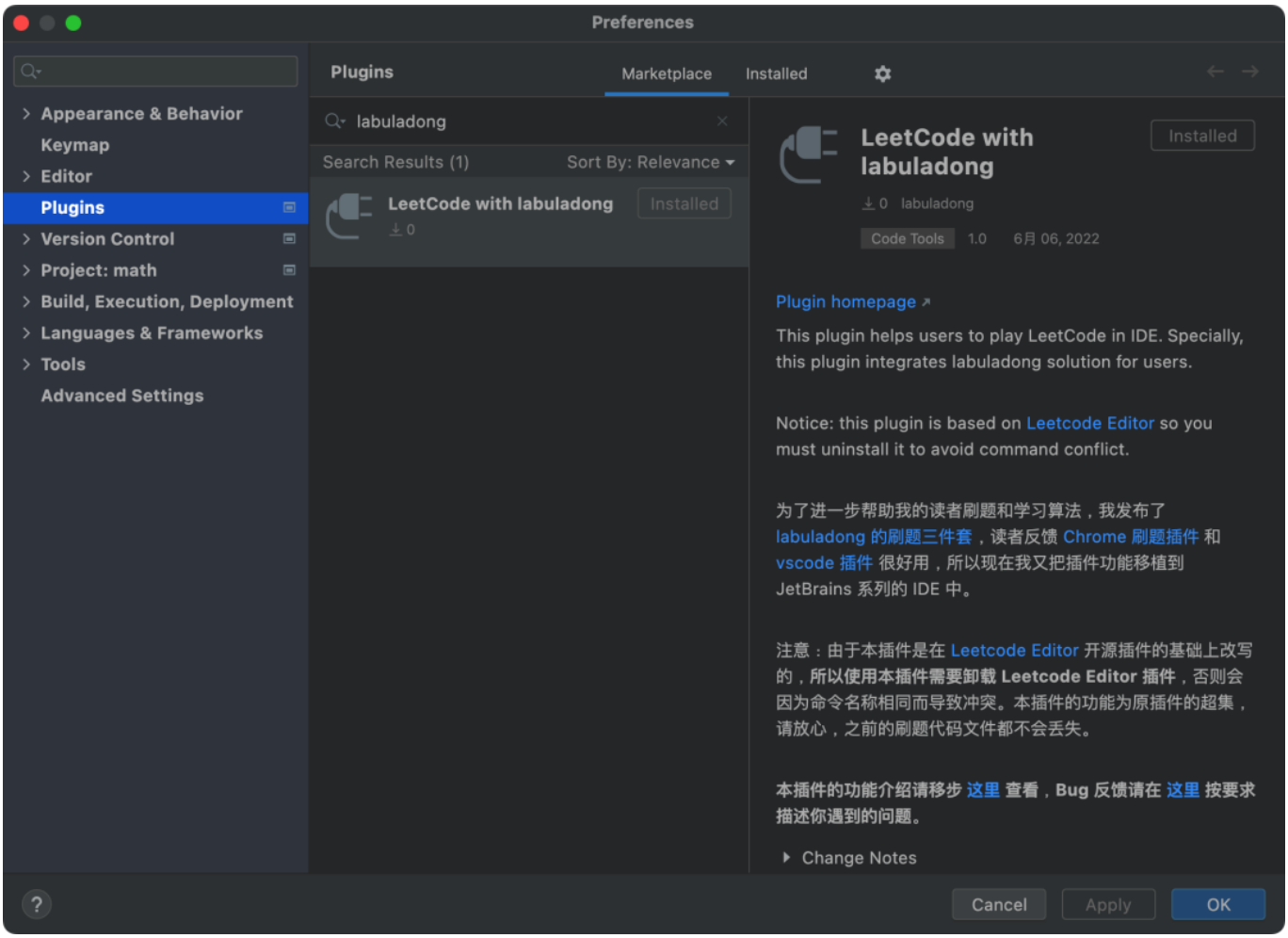
安装方式

注意：由于本插件基于 LeetCode Editor 开源插件改写：

https://github.com/shuzijun/leetcode-editor

所以如果你之前安装了 Leetcode Editor 插件，需要先将其卸载，否则会因为命令名称相同而导致冲突。本插件的功能为原插件的超集，请放心，之前的刷题代码文件都不会丢失。

我的插件全名为「**LeetCode with labuladong**」，在 JetBrains 系列 IDE 的插件商店中搜索关键词「labuladong」即可下载：



如果无法搜索到插件，可能是网络的问题。可以在 JetBrains 插件网页端按照指示下载安装：

https://plugins.jetbrains.com/plugin/19317-leetcode-with-labuladong

更新方式

JetBrains 家的 IDE 会自动检测更新，有更新时会有提示。建议及时更新最新版，以获得最流畅的体验。

更新日志

详见 Jetbrain 插件更新日志。

bug 反馈

如果你在使用中遇到问题，可以在这里按要求反馈 bug，我会定期回复和处理：

https://github.com/labuladong/fucking-algorithm/discussions/939

用算法打败算法

经常有读者问我学算法有什么用，我觉得算法是一种抽象的思维能力。现实中的很多问题只要稍加抽象，就能联想到算法题中的编程技巧，然后得心应手地解决它们。

就比如说 我的刷题全家桶 吧，从去年年底到目前已经有 20 多万次的下载：

这套 PDF 教程和刷题插件不仅沉淀了我这些年学算法的心得体会，而且还把算法的思维运用到了制作过程中。本文简单介绍一下我制作教程以及插件时用到的算法，看看算法如何辅助大家更顺畅地学习。

计算相关文章/题目

我在前文 近期的大更新 说到了近期我对网站、PDF 以及插件功能的更新，其中有几个很实用的功能：

每篇文章的末尾添加了「相关文章」和「相关题目」的功能：

▼引用本文的题目

安装 我的 Chrome 刷题插件 点开下列题目可直接查看解题思路：

LeetCode	力扣
793. Preimage Size of Factorial Zeroes Function	793. 阶乘函数后 K 个零
35. Search Insert Position	35. 搜索插入位置

▼引用本文的文章

- base case 和备忘录的初始值怎么定？
- 二分搜索怎么用？我又总结了套路
- 二分查找高效判定子序列
- 动态规划设计：最长递增子序列

每道题目的解法思路底下也添加了「相关题目」的功能：

```
                // 做选择
                track.add(nums[i]);
                used[i] = true;
                // 进入下一层决策树
                backtrack(nums, track, used);
                // 取消选择
                track.removeLast();
                used[i] = false;
            }
        }
    }
```

类似题目：

- 216. 组合总和 III 🟠
- 39. 组合总和 🟠
- 40. 组合总和 II 🟠
- 47. 全排列 II 🟠
- 51. N 皇后 🔴
- 77. 组合 🟠
- 78. 子集 🟠
- 90. 子集 II 🟠
- 剑指 Offer II 079. 所有子集 🟠

如果要实现这几个功能，我要维护三个映射：

1、文章到相关题目列表的映射。

2、文章到相关文章列表的映射。

3、题目到相关题目列表的映射。

我手动维护了一个文章到相关题目的映射：

```
Map<Article, List<Question>> map;
```

所以第 1 个功能可以直接实现，但其他两个功能怎么实现呢？

文章之间直接的引用肯定是一种相关关系，但如果多篇文章能够同时解决同一道题目，也可以说明这些文章之间存在相关关系。

因为我一般会在一篇文章里讲解多道题目，那么同一篇文章讲解的题目肯定有相关关系。同时，如果两篇文章相关，那么这两篇文章涉及的题目之间也应该是相关的。

想把上述思路写成代码，我们可以把上述的 `map` 转化成 图模型：

把「文章」和「题目」理解成一幅图中的两种节点，关联关系理解成图中的边，那么「文章」节点的相邻节点一定都是「题目」节点，「题目」节点的相邻节点一定都是「文章」节点，不可能有两个相同类型的节点相邻，这是一幅典型的 二分图。

一旦把问题抽象成图模型，就可以运用所有图论算法，我随便举几个例子：

1、「文章」节点的邻居节点就是相关题目，「题目」节点的相邻节点就是能够解决该题目的文章，这是 二分图 的特性。

2、可以利用 Union Find 并查集算法 判断图中的两个节点是否连通。

具体来说，可以在 O(1) 时间判断两道题是否相关、两篇文章是否相关、某道题目和某篇文章是否相关。

3、可以把图中的某个节点作为起点执行 DFS/BFS 遍历算法，所有可达的节点即为相关节点，且距离越近，相关性越高。

具体来说，给一个「题目」节点，我用 DFS/BFS 算法计算所有可达的节点中的「题目」节点就是相关题目；类似的，「文章」节点周围所有可达节点中的「文章」节点就是相关文章。

这样，之前提出的三个相关关系的映射就能算出来了。

计算合理的目录顺序

很多读者反馈我的 PDF 教程的难度设计由浅入深，读起来酣畅淋漓，那是因为我在目录的设计上也使用了算法辅助，让学习曲线变得尽可能平滑。

首先，我给每道题目都打了 tag，这样只需把对应 tag 的题目丢到对应的章节里就行了。比如我可以把「基本二分搜索」「二分搜索运用」等多个 tag 的题目都丢到二分搜索的章节。

但是同一章节的题目难度也应该循序渐进。所以，我维护了题目之间的依赖关系，并根据这个依赖关系生成了一幅以题目为节点，依赖关系为边的 有向图，并在这幅有向图中进行了 环检测和拓扑排序算法，保证做每道题之前已经具备相关的前置知识。

接下来遇到一个新问题：

之前我的插件和 PDF 支持了 手把手刷《剑指 offer》的功能，所以算法笔记中包含很多《剑指 offer》的题目。

但《剑指 offer》的一些题目和力扣主站的题目有重复，力扣本身也有一些题目是重复的，所以现在需要对每个章节的题目进行去重。

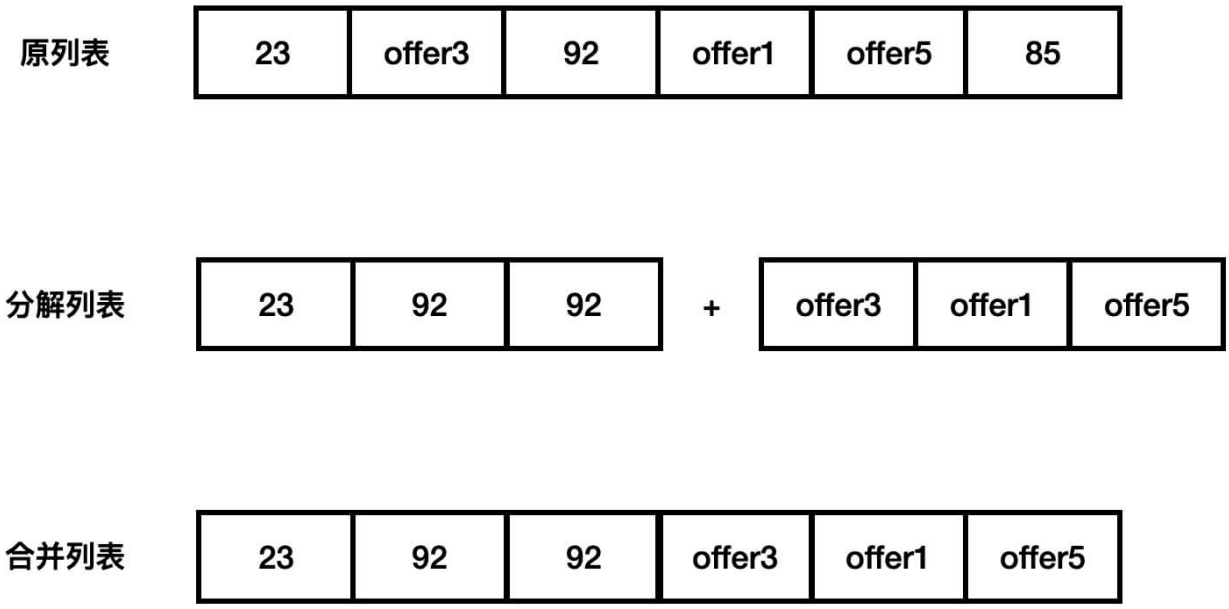
由于章节中题目的顺序是经过拓扑排序的，所以我们希望**去重时不改变题目之间的相对顺序**。

我们可以用简单直接的方法，不过既然是一套算法教程，那当然要秀一下算法技巧喽。所以这里可以用前文 数组双指针技巧 中的快慢指针进行数组的原地去重。

另外，我希望《剑指 offer》的题目和原题有所区分，比如对于同一 tag 下的题目，让主站的题目排在前面，《剑指 offer》的题目排在后面。

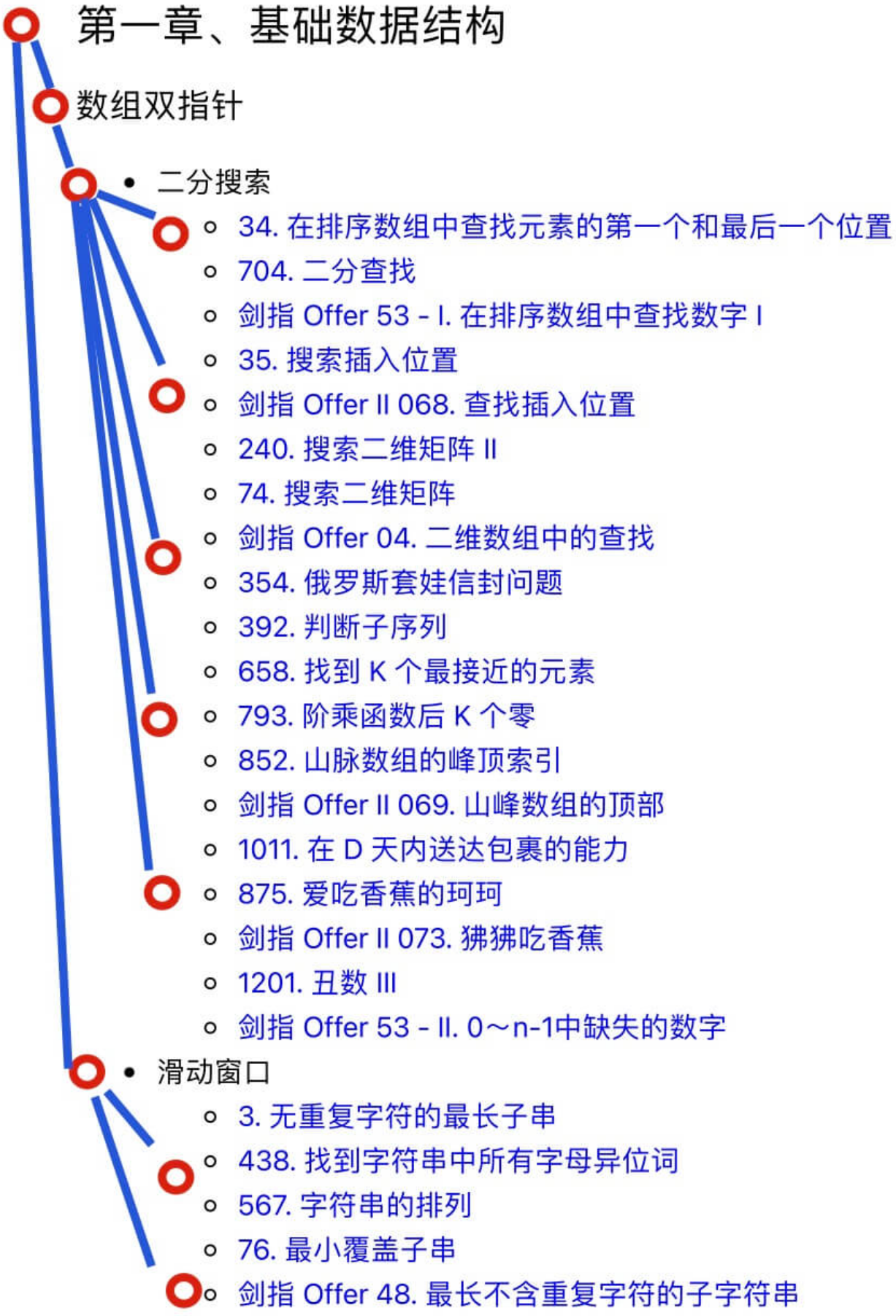
类似去重需求，因为已经进行了拓扑排序，所以重排不应该不改变主站和剑指 offer 题目之间的相对顺序。

这里可以用前文 链表双指针技巧 讲到的分解链表的技巧，把剑指 offer的题目分解出来再合并到末尾：

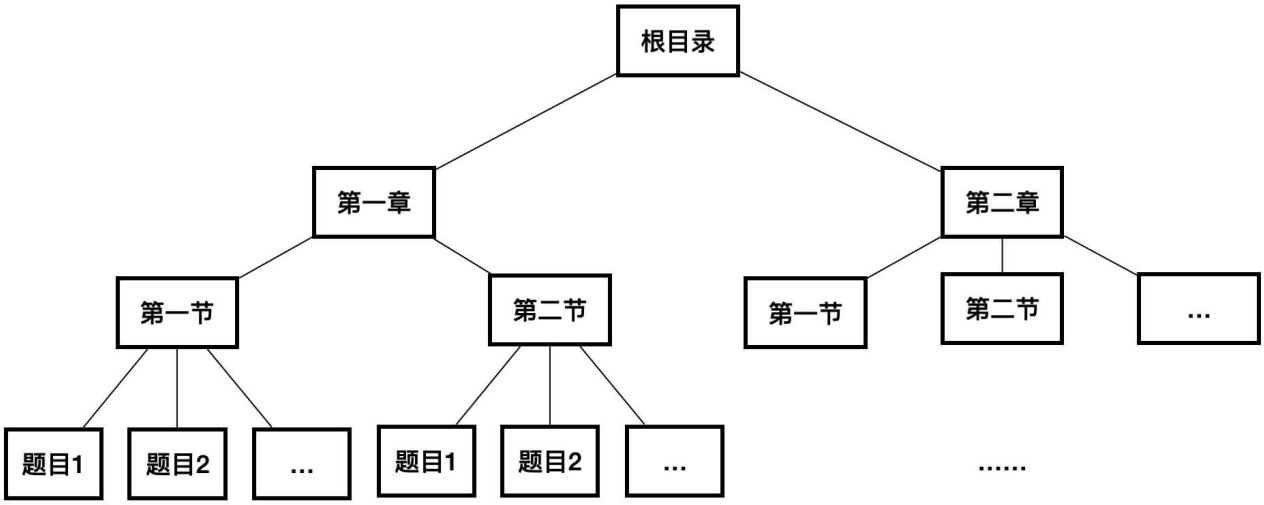


到这里，题目的顺序就定下来了。但由于目录包含章节、小节、文章等多个层级，需要一个算法来为章节名、小节名、题目名生成正确的缩进。

实际上目录是一个树形结构，叶子节点是每道题目的内容和思路，非叶子结点是章节、小节的名称：



从树的角度来看，节点的层数可以区别章节、小节和题目，而且注意**这棵树的前序遍历结果就是生成目录的顺序**：



但既然咱学了这么久算法，直接用前序遍历就显得格局低了。这里可以使用前文 嵌套迭代器 的思路，用迭代器的形式遍历多叉树：

迭代器中的元素可能是一个单独的元素（题目），也可能是一个迭代器（章节、小节）。初始迭代器中只有 5 个元素（因为共有 5 个章节），然后迭代器可以自动处理和展开每个列表元素，进而生成完整的 PDF 目录。

以上这些算法结合起来，也就得到了《算法笔记》的目录：

目录

开篇、刷题笔记阅读指南

第一章、基础数据结构

数组双指针

- 二分搜索
  - 34. 在排序数组中查找元素的第一个和最后一个位置
  - 704. 二分查找
  - 剑指 Offer 53 - I. 在排序数组中查找数字 I
  - 35. 搜索插入位置
  - 剑指 Offer II 068. 查找插入位置
  - 240. 搜索二维矩阵 II
  - 74. 搜索二维矩阵
  - 剑指 Offer 04. 二维数组中的查找
  - 354. 俄罗斯套娃信封问题
  - 392. 判断子序列
  - 658. 找到 K 个最接近的元素
  - 793. 阶乘函数后 K 个零
  - 852. 山脉数组的峰顶索引
  - 剑指 Offer II 069. 山峰数组的顶部
  - 1011. 在 D 天内送达包裹的能力
  - 875. 爱吃香蕉的珂珂
  - 剑指 Offer II 073. 狒狒吃香蕉
  - 1201. 丑数 III
  - 剑指 Offer 53 - II. 0～n-1中缺失的数字
- 滑动窗口
  - 3. 无重复字符的最长子串
  - 438. 找到字符串中所有字母异位词
  - 567. 字符串的排列
  - 76. 最小覆盖子串
  - 剑指 Offer 48. 最长不含重复字符的子字符串
  - 剑指 Offer II 014. 字符串中的变位词
  - 剑指 Offer II 015. 字符串中的所有变位词
  - 剑指 Offer II 016. 不含重复字符的最长子字符串
  - 剑指 Offer II 017. 含有所有字符的最短字符串
  - 239. 滑动窗口最大值
  - 26. 删除有序数组中的重复项
  - 27. 移除元素
  - 283. 移动零
  - 80. 删除有序数组中的重复项 II
  - 83. 删除排序链表中的重复元素
  - 剑指 Offer 57. 和为s的两个数字
  - 88. 合并两个有序数组
  - 977. 有序数组的平方
  - 1260. 二维网格迁移
  - 151. 颠倒字符串中的单词
  - 82. 删除排序链表中的重复元素 II
  - 剑指 Offer II 007. 数组中和为 0 的三个数
  - 15. 三数之和
  - 18. 四数之和
  - 870. 优势洗牌
  - 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面
  - 11. 盛最多水的容器
  - 42. 接雨水
  - 360. 有序转化数组

链表双指针

- 2. 两数相加
- 141. 环形链表
- 142. 环形链表 II
- 160. 相交链表
- 19. 删除链表的倒数第 N 个结点
- 21. 合并两个有序链表
- 23. 合并 K 个升序链表
- 86. 分隔链表
- 876. 链表的中间结点
- 剑指 Offer 25. 合并两个排序的链表
- 剑指 Offer 52. 两个链表的第一个公共节点
- 剑指 Offer II 021. 删除链表的倒数第 n 个结点
- 剑指 Offer II 022. 链表中环的入口节点
- 剑指 Offer II 023. 两个链表的第一个重合节点
- 剑指 Offer II 078. 合并排序链表
- 1305. 两棵二叉搜索树中的所有元素
- 264. 丑数 II
- 313. 超级丑数

《算法秘籍》也用了类似的算法优化了目录的组织顺序，我这里就不赘述了。

当然，PDF 中用到的算法也被运用在了 Chrome 刷题插件、vscode 刷题插件、JetBrains 刷题插件 中，致力于给大家营造丝滑的刷题学习体验，公众号后台回复关键词「**全家桶**」即可打包下载。

labuladong 的数据结构精品课 V2.1

> 在 2024/2/1 之前，我的课程都发布在小鹅通平台，但是为了给读者更丰富流畅的学习体验，自 2024/2/1 开始，我将课程全面迁移到新网站 labuladong.online，本课程的新地址为：
>
> https://labuladong.online/algo/ds-class/
>
> **未来我将不再使用小鹅通平台，之前已经在小鹅通购买课程的读者，请在 这里 按照说明操作，将你在小鹅通的课程权限同步到新网站。**
>
> 在新网站中，课程内容页将全线支持多语言解法、行内图片注释、可视化面板、可视化编辑器、刷题插件等配套工具，enjoy yourself！

本课程的内容

废话不多说，直接说大家最关心的点。本课程内容为常见数据结构的实现原理，用大量习题强化你对算法模板的运用。如果你想针对性提升写递归算法的功力，可以学习 二叉树递归专题课 进阶。

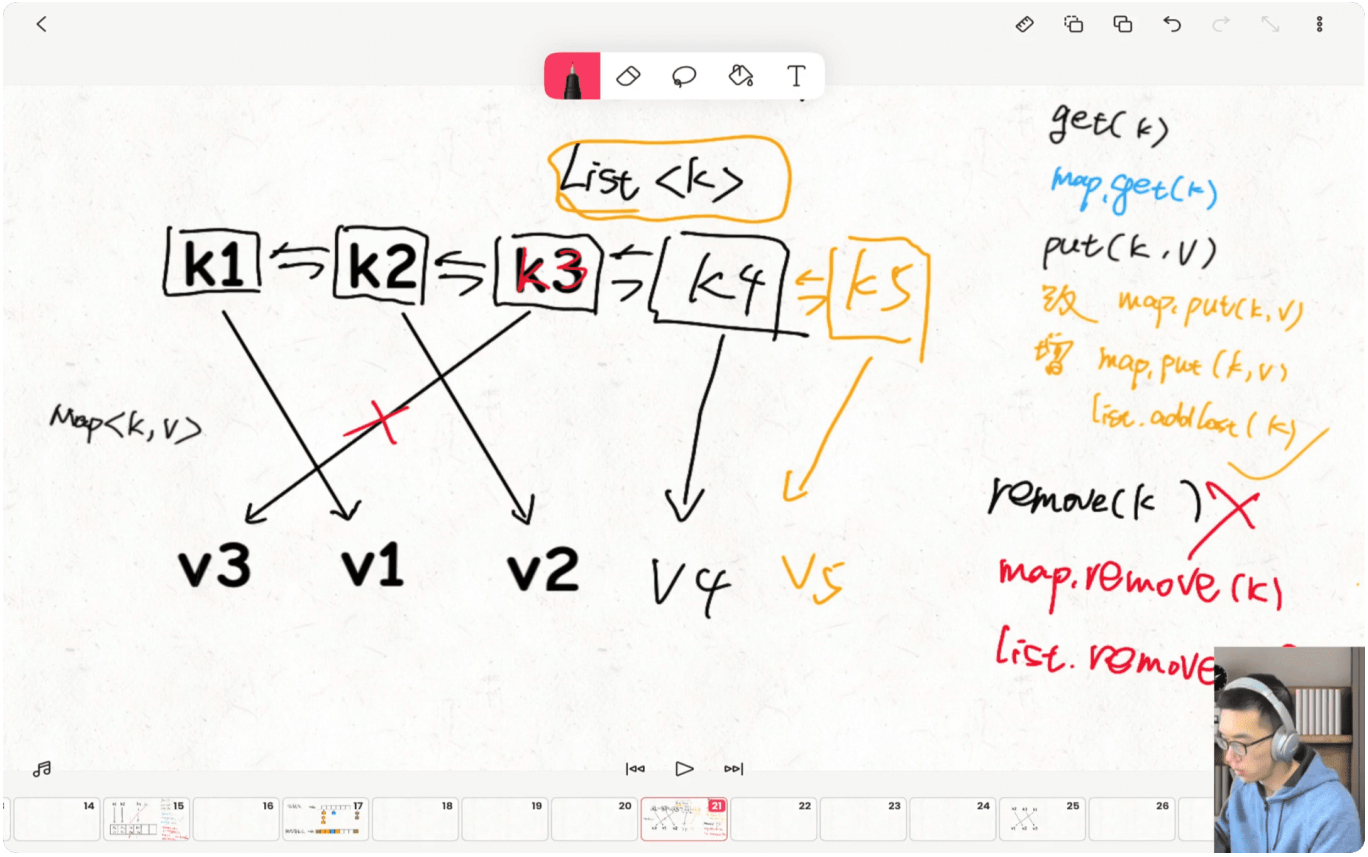
本课程主要分为两部分：第一部分是手把手带你实现常用的数据结构，第二部分手把手你套框架解决常见数据结构相关的经典习题。

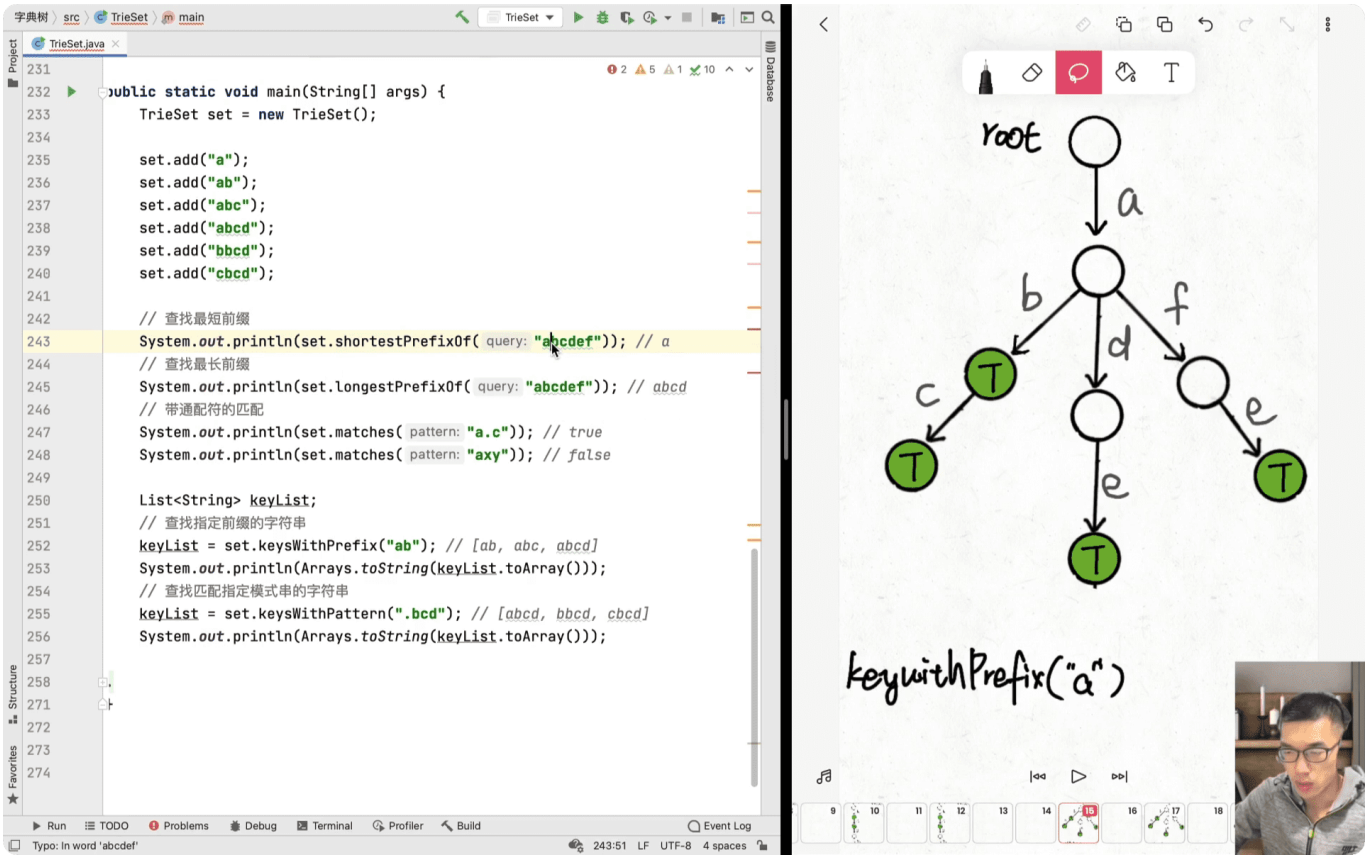
手把手实现数据结构的部分主要是视频，讲的会比较细，适合时间比较多或者基础比较薄弱的读者，比如跨行转码或者应届生。

如果你的时间不算充裕，希望快速提升算法基本功对付面试，可以选择性跳过视频部分的基础，着重学习标记为【强化练习】的习题部分。

我的课程中所有习题都是用框架化、流程化的思维模式手把手带你举一反三地解决这些问题，争取给你练出肌肉记忆。

我会在视频中边写代码边画图，方便大家理解数据结构底层的实现原理：





本课中主要用 Java 写代码，因为 Java 这种语言的语法非常规范，即便你之前没用过 Java 也不妨碍理解代码逻辑。而且我会在课程开头写一个简明的教程说明本课程所需的 Java 基础，所以无论你是否熟悉 Java 都可以学习本课程。

针对算法学习，我给出的方法是带你用完全统一的思维模式，甚至是完全统一的代码风格去不断地做题，手把手带你运用代码框架，以不变应万变。你可以说思维框架这种东西就像蹲马步，朴实无华且枯燥，但架不

住他就是比花拳绣腿有用啊，对不对？

比如我总结过 滑动窗口算法框架套路，其中总结了运用滑动窗口算法时需要问自己的三个问题。那么我在滑动窗口习题的这一节内容中就列举了几道经典题目，且每道题都是按照统一的流程和统一的代码风格求解：

1004. 最大连续1的个数 III

- **标签：数组，滑动窗口**

给定一个二进制数组 `nums` 和一个整数 `k`，如果最多可以把 `k` 个 `0` 翻转成 `1`，请返回数组中连续 `1` 的最大个数。

示例 1：

```
输入：nums = [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]
把 2 两个 0 翻转成 1，最长的值为 1 的子数组长度为 6。
```

基本思路

这题考察滑动窗口技巧，你维护一个窗口在 `nums` 上滑动，保证 `nums` 中的所有数字都被替换成 1，那么窗口可以达到的最大长度就是题目要的答案。

前文 滑动窗口框架 说过，使用滑动窗口算法需要搞清楚以下几个问题：

1、什么时候应该扩大窗口？

2、什么时候应该缩小窗口？

3、什么时候得到一个合法的答案？

针对本题，以上三个问题的答案是：

1、当可替换次数大于等于 0 时，扩大窗口，让进入窗口的 0 都变成 1，使得连续的 1 的长度尽可能大。

2、当可替换次数小于 0 时，缩小窗口，空余出可替换次数，以便继续扩大窗口。

3、只要可替换次数大于等于 0，窗口中的元素都会被替换成 1，也就是连续为 1 的子数组，我们想求的就是最大窗口长度。

713. 乘积小于 K 的子数组

- **标签：数组，滑动窗口**

给你一个整数数组 `nums` 和一个整数 `k`，请你返回子数组内所有元素的乘积严格小于 `k` 的连续子数组的数目。

示例 1：

```
输入：nums = [10,5,2,6], k = 100
输出：8
解释：8 个乘积小于 100 的子数组分别为：[10]、[5]、[2],、[6]、[10,5]、[5,2]、[2,6]、[5,2,6]。
需要注意的是 [10,5,2] 并不是乘积小于 100 的子数组。
```

基本思路

这题考察滑动窗口技巧，你维护一个窗口在 `nums` 上滑动，然后计算那些元素之积小于 `k` 的窗口个数即可。

前文 **滑动窗口框架** 说过，使用滑动窗口算法需要搞清楚以下几个问题：

1、什么时候应该扩大窗口？

2、什么时候应该缩小窗口？

3、什么时候得到一个合法的答案？

针对本题，以上三个问题的答案是：

1、当窗口元素之积小于 `k` 时，扩大窗口，让积更大一些。

2、当窗口元素之积大于 `k` 时，缩小窗口，让积更小一些。

3、当窗口元素之积小于 `k` 时，窗口内元素的所有子数组都是合法子数组。

注意：之所以本题可以用滑动窗口，关键是题目说了 `nums` 中的元素都是正数，这就保证了只要有元素加入窗口，积一定变大，只要有元素离开窗口，积一定变小。

另外，在我的 Chrome/vscode/Jetbrain 系列插件中也能解锁我对这些题目的讲解，结合课程中的视频和图文讲解，加上刷题插件的辅助，可以说这是一种全方位的丝滑学习体验。

当然，本课程的焦点在于数据结构的实现及其必知必会的算法技巧，不包含诸如动态规划等高级算法的内容。如果对这类进阶递归算法感兴趣，推荐你购买我的 二叉树递归专题课。

购买本课程后还将解锁我的全系刷题插件中的数据结构专属题解，解锁方法见课程第一章的准备工作。

labuladong 二叉树（递归）专题课

> 在 2024/2/1 之前，我的课程都发布在小鹅通平台，但是为了给读者更丰富流畅的学习体验，自 2024/2/1 开始，我将课程全面迁移到新网站 labuladong.online，本课程的新地址为：
>
> https://labuladong.online/algo/tree-class/
>
> **未来我将不再使用小鹅通平台，之前已经在小鹅通购买课程的读者，请在 这里 按照说明操作，将你在小鹅通的课程权限同步到新网站。**
>
> 在新网站中，课程内容页将全线支持多语言解法、行内图片注释、可视化面板、可视化编辑器、刷题插件等配套工具，enjoy yourself！

前言

经常有读者说那什么回溯算法啊，动态规划啊，分治算法啊乱七八糟的算法让他们头大，想让我出一个针对这些算法的专题课。但我认为真没这个必要，因为你说的这些算法的名字听起来高大上，实际上嘛，忽悠人的，因为他们全都是二叉树的「套壳」算法，你要是老盯着那个壳，那不晕头转向么？

所以我就先不讲那些高大上的算法，只讲二叉树，请你忘记那些高大上的算法技巧把这个课认真刷完，保你内功大涨，到时候再去看其他递归算法，就会融会贯通了。

本课程的内容

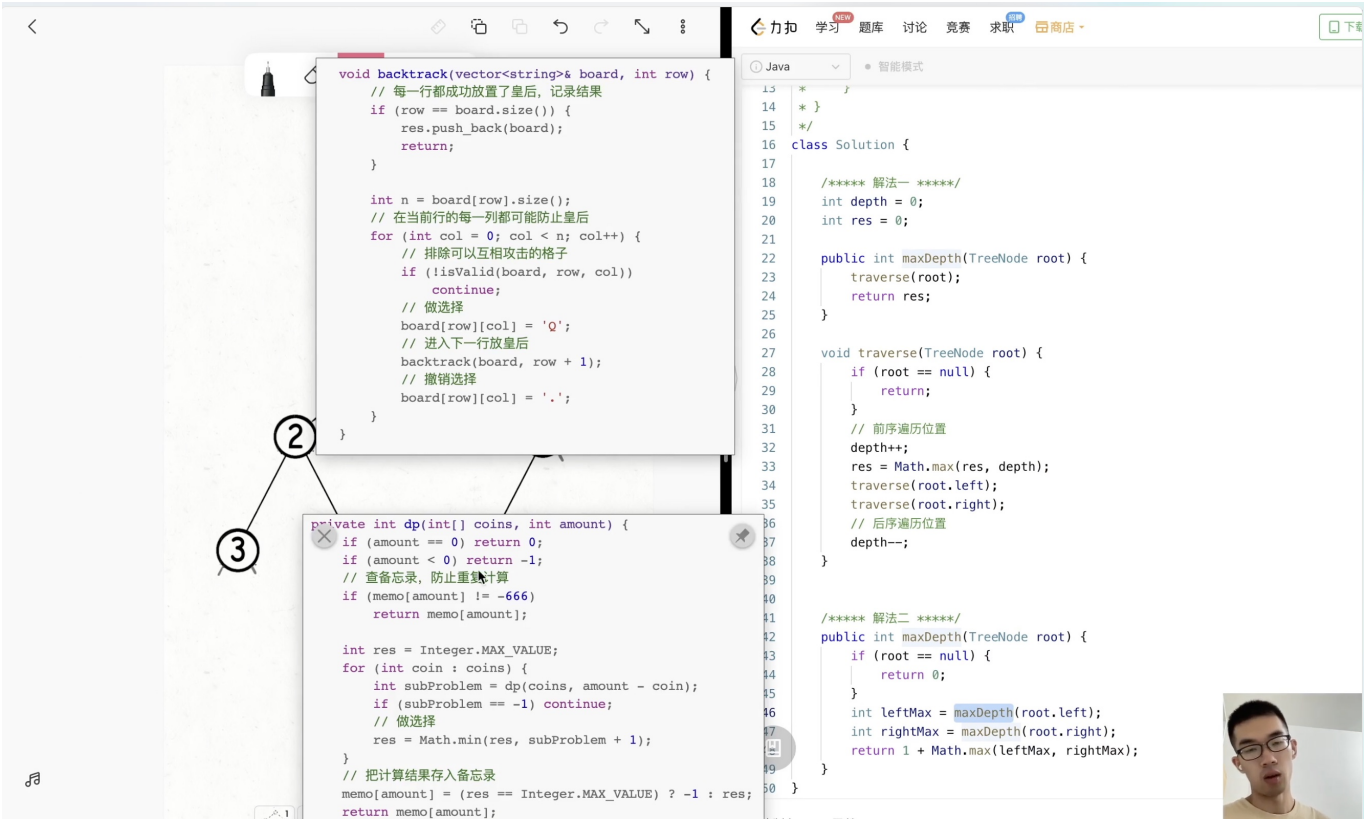
老读者都知道，我在 学习算法的框架思维 着重强调了二叉树题目的重要性，并提到二叉树算法是所有递归算法的根本，动态规划、回溯算法、图论算法等高级算法底层都是二叉树算法的思想。

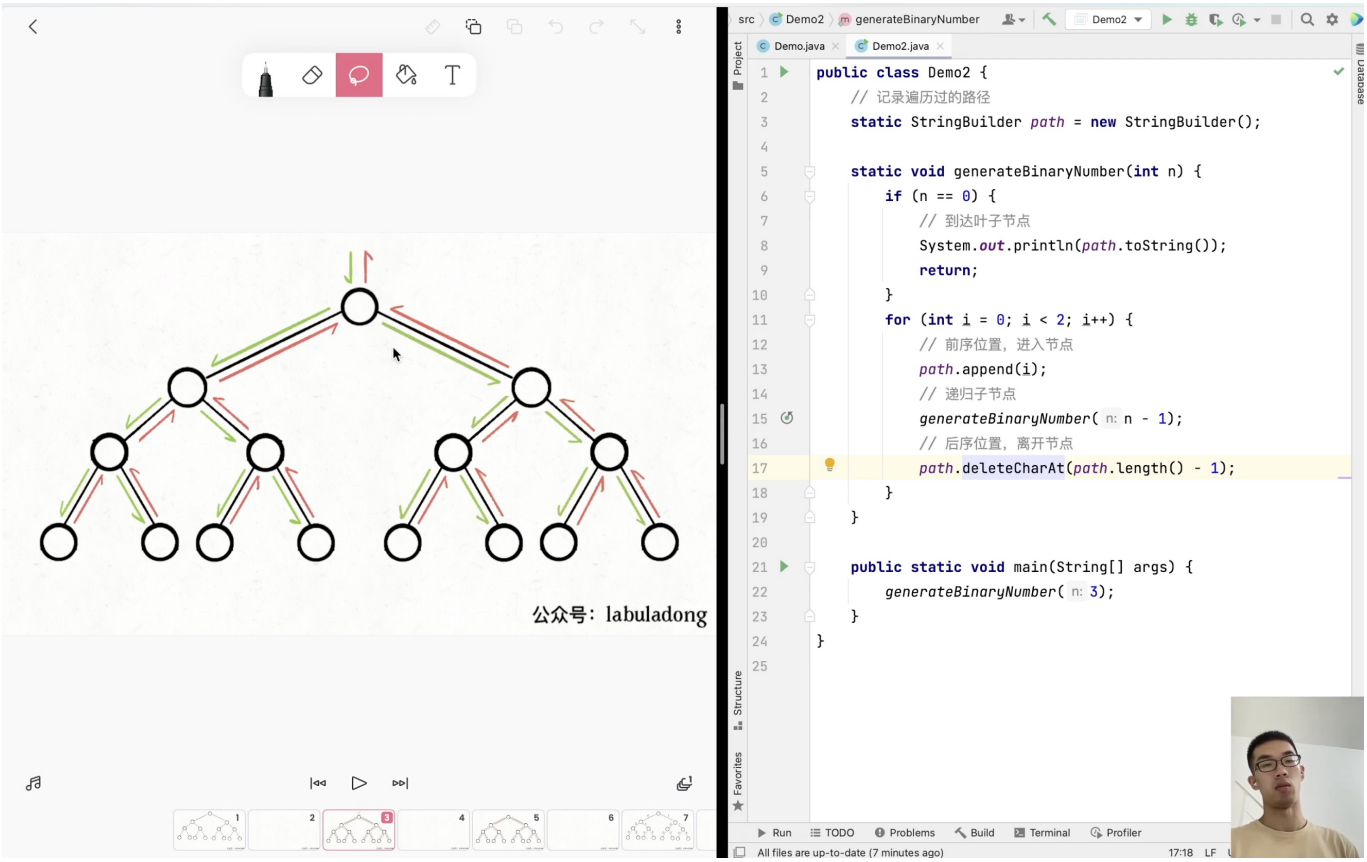
同时，我在 东哥手把手带你刷二叉树（纲领篇） 中提到，你甭管二叉树系列题目千变万化，说到底这些题目只有「遍历」和「分解问题」两种解题思路。

本专栏就手把手教你如何运用这两种思维模式，结合我的系列刷题插件，用一套模板搞定力扣上二叉树专栏中的 160 道精选二叉树题目。你可以安装 我的 Chrome 插件 后查看力扣的 二叉树题目列表，几乎所有题目都基于本课的内容做了讲解，学完本课后你应该可以把它们全部拿下。

本专栏包含三部分，如果时间充裕，可以按顺序学习；如果想快速提升，可以着重刷第二章和第三章的习题：

第一部分是视频课，我会通过画图和代码讲解的方式手把手带你通过二叉树深刻理解递归思维：





第二部分图文课程，我精选了几篇二叉树教程，细节拉满，手把手带你解析经典的二叉树题目套路。

第三部分是精选习题，我对力扣的二叉树题目进行了精选分类，用「遍历」和「分解问题」两种解题思路，手把手带你逐个击破所有题目。

[习题] 题海战术磨炼递归思维

图文 用「遍历」思维解决的经典例题 I

图文 用「遍历」思维解决的经典例题 II

图文 用「遍历」思维解决的经典例题 III

图文 用「分解问题」思维解决的经典例题 I

图文 用「分解问题」思维解决的经典例题 I

图文 同时用「遍历」和「分解问题」的思维解题

图文 需要利用后序位置的经典例题 I

图文 需要利用后序位置的经典例题 II

图文 需要利用后序位置的经典例题 III

图文 运用层序遍历的经典习题 I

图文 运用层序遍历的经典习题 II

图文 二叉搜索树经典例题 I

购买本课程后还将解锁我的全系刷题插件中的二叉树专属题解，解锁方法见课程第一章的准备工作。

30 天刷题打卡训练营

30 篇打卡内容都是选自网站上的算法教程，如果你时间充裕，且有那个动力坚持刷题学习，那直接刷网站的内容就可以了，没必要参与这个打卡挑战。

这个打卡主要针对的读者：

1、时间充裕，但是自制能力有限，容易半途而废。比如说你知道得开始刷题了，也有那个时间，但总是拖延。那么你可以借助打卡挑战的押金机制来督促自己迈出第一步。

2、时间不充裕，希望有一个外部的力量来适度逼迫一下自己坚持刷题，养成习惯。那么你也可以借助打卡挑战的押金机制来督促自己每日精进。

另外，因为打卡的内容只是网站上筛选出的部分文章，主要目的是帮你梳理出算法知识体系的一个框架，并培良好的学习习惯。所以即便你完成了本次 30 天打卡，还是需要继续保持精进的状态，学习网站上的其他内容并亲自动手实践运用，才能真正意义上搞定算法这个难关。

为什么要搞这个打卡挑战

经常有读者跟我说，我的公众号和网站总结的算法技巧已经非常详尽了，按理说只要照着刷一遍就可以。但问题是自己一个人坚持不下来，希望有一个刷题打卡的活动督促，克服惰性，养成每天刷题进步的习惯。

我觉得这个事情确实值得做，因为像刷题这种短期没有回报的事情，很多时候需要外部给一点动力，大家抱团去搞，有人指点，互相交流，才不容易放弃。所以我推出了一个 30 天刷题打卡训练营，先交押金，完成学习全额退押金，以此督促大家坚持学习。

老读者知道，我之前搞过二十多期的 14 天/ 21 天刷题打卡训练营，但是之前的打卡活动主要有如下两个问题：

问题一、有的参与者非常敷衍，直接复制我的思路，或者只粘贴代码，一个字都不改，完全没有自己的理解和总结。这样很难提高自己的算法能力，我就不明白，非要这样自欺欺人，又何必参加打卡活动呢？

问题二、每一期需要固定的起止时间，有些读者想参与，但是错过了本期打卡的报名时间，就只能等下一期才能报名，等待周期较长，不够灵活。

这次的 30 天刷题打卡训练营可以说是升级版，不仅更新了一部分内容，还针对这两个问题做了一些改进：

改进一、我会设置 100 字的最低字数限制，且学员之间可以查看彼此的打卡总结，希望好的打卡总结给大家起到好的带头作用。

改进二、现在打卡挑战不再有固定的起止时间，任何读者任何时候都可以报名，挑战周期为报名的当天往后 36 天（30 篇打卡内容每天一篇，另外加上 6 天的补卡机会）。

打卡方式

如何最高效地打卡学习呢？我给大家说一个秘诀：一边自己学，一边教别人，也就是一边输入一边输出，效果最好。

很多时候你觉得自己会了，实际上你不是真的会，否则就不会有那么多人说自己一上手就不会做题了。如果你试图把自己的理解表达出来，就会发现自己的知识盲点，从而进行更深入的思考，真正理解这个知识点。

所以我的打卡挑战不能仅仅是学习我的算法教程和做题，而且要有一定的总结输出，发送到网站下方的评论区作为每天的打卡内容。你只需要就当天的学习内容做 100 字以上的总结就可以，这个要求很容易完成吧，稍微认真写一写就够了。

对于打卡，重要的是对自己负责态度。如果遇到态度敷衍，直接复制粘贴我的代码或思路的情况，当天的打卡将被判定为无效，我会私信通知并给机会改正，以此来督促大家认真学习，对自己负责。

报名注意事项

参与我的打卡挑战是免费的，但如果完全没有门槛，大家都做看客了，效果不会好。所以我的打卡挑战是押金形式，劝退伸手党：**先交 299 元押金，按要求完成每日打卡（每天学习一篇文章，缺卡不超过 6 天），押金全额原路退还**。

在支付押金之前，请先阅读以下注意事项。

支付押金后，打卡就开始了，当天算作第一天打卡，所以付款后请立即完成第一天的打卡任务。第一天的任务很简单，就是带你走一遍打卡流程，预计花费 5 分钟，请务必认真阅读规则。

打卡的结算和刷新全部按照北京时间，且每天只允许打卡一次，北京时间 00:00 打卡次数刷新。

比方说，你完成 Day01 的打卡任务之后，不能立即查看 Day02 的打卡任务，因为当天的打卡次数已耗尽。需要等到北京时间第二天 00:00 才能解锁 Day02 的内容。

考虑到这个时间机制，不建议在北京时间 23:50 分之后报名参与打卡，因为如果你不能在 10 分钟内完成第一天的打卡任务的话，第一天就会被算作缺卡。

当然，我也考虑到这个情况，所以第一天的打卡任务很简单，几分钟即可完成，不过还是建议你等到北京时间 00:00 之后，或者在其他时段报名参与，避免不必要的缺卡。

Java 语言基础

> 我的网站解法代码现已支持了所有主流编程语言，包括 Java、Python、C++、JavaScript 和 Go，不过依然建议你简单了解一下 Java 语言，因为其他语言都是我用 chatGPT 从 Java 翻译过去的，如果偶尔遇到错误可能需要你对照理解 Java 的代码逻辑。

目前我的算法教程基本上都是用 Java 语言写的代码，主要有以下几个原因：

1、Java 是强类型语言。因为我们就是要学习运用各种数据结构写算法嘛，所以清楚地知道每个变量是什么类型非常重要，方便你 debug，也方便 IDE 进行语法检查。如果是 Python 这样的动态语言，每个变量的类型不明显，可能有碍大家的理解。

2、Java 这种语言中规中矩，没有什么花里胡哨的语言特性。即便你之前没学过 Java 语言，单看代码也能轻易地理解逻辑。所以如果你有其他比较熟悉的语言，完全可以把我的代码思路用自己的语言实现。

另外，抛开算法不谈，单就编程技术的学习来说，我也建议大家多学习几门语言，尤其是像 Java 这种使用非常广泛的语言。因为就我的经验而言，任何人的职业生涯都不可能只用一种编程语言的，一般都是换个工作就会换一套技术栈，所以主流编程语言最基本的语法要能看的懂，这对以后得工作也会有很大帮助。

当然了，对于一些不怎么熟悉 Java 且其他语言基础也比较差的小伙伴，初学时肯定会有一点吃力。因为算法本身就比较烧脑了，再加上对语言特性不太熟悉，很可能会因为一些语言特性的坑而导致算法错误，这当然是我们不希望看到的。

不过不用担心，我的所有解法代码都会刻意避开 Java 的高级语言特性，用最朴素的、最容易看懂的方式写代码。

下面我还会带大家过一遍学习我的算法教程所需的 Java 基本用法，以便不了解 Java 的小伙也能伴顺畅地学习我的文章。

首先，Java 的基本知识网上有很多，我就不重复造轮子了，我就随便贴一个中文教程吧，你如果有更喜欢的 Java 教程也可以看你喜欢的：

https://www.runoob.com/java/java-tutorial.html

我圈出的这一部分是 Java 的基本用法，你需要看一下。当然这个网站也有其他的 Java 基础知识，你有空也可以快速浏览一下：

下面我具体介绍一下我在教程中会用到的 Java 的标准库的数据结构以及一些可能踩坑的语言特性。

Java 标准库数据结构的基本用法

Java 中最基本的数据类型叫 Primitive Type（原始类型），比如 `int, char, long, boolean` 这种小写字母的关键字。大家应该都学过 C 语言，所以对于原始类型应该不陌生，因为 C 语言里几乎只有原始类型，连字符串都只能用 `char[]` 字符数组来表示，只能使用指针进行操作，使用起来非常不方便。

当然，作为高级编程语言，Java 提供了很多高级数据类型，都封装成了标准库中的**类**（Java 中类的名字都以大写字母开头），比如 Java 中的字符串就是 `String` 类，这个类有 `length(), substring()` 等方法，这样我们就不用管底层的操作细节，使用起来方便多了。我会在 数据结构精品课 会中带大家实现标准库中的常见数据结构类，深入理解其底层的实现细节，帮助大家更深入地理解常用数据结构。

以下是一些基本的 Java 类型。

1、数组。

初始化方法：

```
int m = 5, n = 10;

// 初始化一个大小为 10 的 int 数组
// 其中的值默认初始化为 0
int[] nums = new int[n]
```

```
// 初始化一个 m * n 的二维布尔数组
// 其中的元素默认初始化为 false
boolean[][] visited = new boolean[m][n];
```

作为原始类型，这就类似 C 语言的 `int` 数组，操作起来比较麻烦，所以我们更多地使用 `ArrayList` 动态数组，后面的视频课会详细讲解。

2、字符串 String

Java 的字符串处理起来挺麻烦的，因为它不支持用 `[]` 直接访问其中的字符，而且不能直接修改，要转化成 `char[]` 类型才能修改。主要说下 `String` 中我们会用到的一些特性：

```
String s1 = "hello world";
// 获取 s1[2] 那个字符 l
char c = s1.charAt(2);

char[] chars = s1.toCharArray();
chars[1] = 'a';
String s2 = new String(chars);
// 输出：hallo world
System.out.println(s2);

// 注意，一定要用 equals 方法判断字符串是否相同
if (s1.equals(s2)) {
    // s1 和 s2 相同
} else {
    // s1 和 s2 不相同
}

// 字符串可以用加号进行拼接
String s3 = s1 + "!";
// 输出：hello world!
System.out.println(s3);
```

Java 的字符串不能直接修改，要用 `toCharArray` 转化成 `char[]` 的数组进行修改，然后再转换回 `String` 类型。

另外，虽然字符串支持用 `+` 进行拼接，但是效率并不高，并不建议在 for 循环中使用。如果需要进行频繁的字符串拼接，推荐使用 `StringBuilder/StringBuffer`：

```
StringBuilder sb = new StringBuilder();

for (char c = 'a'; c <= 'f'; c++) {
    sb.append(c);
}

// append 方法支持拼接字符、字符串、数字等类型
sb.append('g').append("hij").append(123);
```

```
String res = sb.toString();
// 输出：abcdefghij123
System.out.println(res);
```

还有一个重要的问题就是字符串的相等性比较，这个问题涉及语言特性，简单说就是一定要用字符串的 `equals` 方法比较两个字符串是否相同，不要用 `==` 比较，否则可能出现不易察觉的 bug。

3、动态数组 ArrayList

`ArrayList` 相当于把 Java 内置的数组类型做了包装，初始化方法：

```
// 初始化一个存储 String 类型的动态数组
ArrayList<String> strings = new ArrayList<>();

// 初始化一个存储 int 类型的动态数组
ArrayList<Integer> nums = new ArrayList<>();
```

常用的方法如下（`E` 代表元素类型）：

```
// 判断数组是否为空
boolean isEmpty()

// 返回数组的元素个数
int size()

// 返回索引 index 的元素
E get(int index)

// 在数组尾部添加元素 e
boolean add(E e)
```

我们做算法题时这些最简单的方法就够用了，你应该看一眼就能明白。如果你对这些数据结构的底层实现感兴趣，我会在 数据结构精品课 中讲解。

4、双链表 LinkedList

`ArrayList` 列表底层是数组实现的，而 `LinkedList` 底层是双链表实现的，初始化方法也是类似的：

```
// 初始化一个存储 int 类型的双链表
LinkedList<Integer> nums = new LinkedList<>();

// 初始化一个存储 String 类型的双链表
LinkedList<String> strings = new LinkedList<>();
```

一般算法题中我们会用到以下方法（`E` 代表元素类型）：

```
// 判断链表是否为空
boolean isEmpty()

// 返回链表的元素个数
int size()

// 判断链表中是否存在元素 o
boolean contains(Object o)

// 在链表尾部添加元素 e
boolean add(E e)

// 在链表尾部添加元素 e
void addLast(E e)

// 在链表头部添加元素 e
void addFirst(E e)

// 删除链表头部第一个元素
E removeFirst()

// 删除链表尾部最后一个元素
E removeLast()
```

这些也都是最简单的方法，和 `ArrayList` 不同的是，我们更多地使用了 `LinkedList` 对于头部和尾部元素的操作，因为底层数据结构为链表，直接操作头尾的元素效率较高。其中只有 `contains` 方法的时间复杂度是 `O(N)`，因为必须遍历整个链表才能判断元素是否存在。

在 数据结构精品课 中我们会亲自实现一个 `MyLinkedList` 类，之后你就会理解每个方法的时间复杂度了。

5、哈希表 `HashMap`

哈希表是非常常用的数据结构，用来存储键值对映射，初始化方法：

```
// 整数映射到字符串的哈希表
HashMap<Integer, String> map = new HashMap<>();

// 字符串映射到数组的哈希表
HashMap<String, int[]> map = new HashMap<>();
```

在算法题中，我们只会用以下几个基本方法（`K` 代表键的类型，`V` 代表值的类型）：

```
// 判断哈希表中是否存在键 key
boolean containsKey(Object key)

// 获得键 key 对应的值，若 key 不存在，则返回 null
V get(Object key)

// 将 key, value 键值对存入哈希表
```

```
V put(K key, V value)

// 如果 key 存在，删除 key 并返回对应的值
V remove(Object key)
```

在 数据结构精品课 中，我会手把手带你实现 MyHashMap 类，深入理解和实现两种处理哈希冲突的方法。

6、哈希集合 HashSet

初始化方法：

```
// 新建一个存储 String 的哈希集合
Set<String> set = new HashSet<>();
```

我们可能在算法题中用到的方法（E 表示元素类型）：

```
// 如果 e 不存在，则添加 e 到哈希集合
boolean add(E e)

// 判断元素 o 是否存在于哈希集合中
boolean contains(Object o)

// 如果元素 o 存在，在删除元素 o
boolean remove(Object o)
```

等你实现了 MyHashMap，就知道 HashSet 底层和 HashMap 其实是一样的。

Java 类的一些基本用法

首先说一下泛型编程。泛型是 Java 提供的一种模板，能够将数据结构的实现和数据类型解耦。

比如说我们在使用 LinkedList 双链表的时候，可以随意设置其中的元素类型。不过需要注意，在泛型中只能使用类，不能使用原始类型：

```
// 装整数的双链表
LinkedList<Integer> list1 = new LinkedList<>();
// 报错，不能用 int 这种原始类型作为泛型
LinkedList<int> list2 = new LinkedList<>();

// 装字符串的双链表
LinkedList<String> list3 = new LinkedList<>();
```

我们在实现自己的数据结构类时，也需要使用泛型，以便我们的数据结构能够装任何类型。

我们在 数据结构精品课 中实现所有数据结构都需要使用泛型，比如 MyLinkedList，在 new 的时候传入 Integer 类型，那么这个 E 就会变成什么类型：

```
public class MyLinkedList<E> {

    // 在链表尾部添加元素
    public void addLast(E e) {
        // ...
    }
}

// 使用方法：
MyLinkedList<Integer> list1 = new MyLinkedList<>();
MyLinkedList<String> list2 = new MyLinkedList<>();
```

当然，某些特殊数据结构对存储的数据类型有要求。比如 TreeMap 这种数据结构，由于底层是用 BST（二叉搜索树）来存储键值对，所以 TreeMap 要求存入其中的键必须是「可比较的」，即对于任意两个键，必须能够知道它俩谁大谁小。

在 Java 中，可以给泛型变量添加 extends 来指定该泛型的某些特性：

```
// MyTreeMap 接收两个泛型 K 和 V，其中 K 必须实现 Comparable 接口
public class MyTreeMap<K extends Comparable<K>, V> {
    public V put(K key, V val) {
        // ...
    }
}

// 使用方法：
MyTreeMap<Integer, Integer> map1 = new MyTreeMap<>();
MyTreeMap<String, Integer> map2 = new MyTreeMap<>();
```

这个 Comparable 是 Java 标准库提供的一个接口，实现如下：

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

所以 K extends Comparable<K> 的意思就是说，泛型 K 实现了这个接口，即类型 K 有 compareTo 这个方法，这意味着两个 K 类型的数据可以比较大小。

另外再说下 Java 的接口，免得有些读者疑惑。我们有时候会看到这样的写法：

```
Queue<String> q = new LinkedList<>();
List<String> list = new LinkedList<>();
```

实际 new 出来的是 LinkedList 类型，但为什么用 Queue 类型和 List 类型接收呢？因为 Queue 和 List 都是 Java 标准库中的接口类型：

```
public interface Queue<E> extends Collection<E> {
    boolean offer(E e);
    E poll();
    E peek();
    // 还有若干方法...
}

public interface List<E> extends Collection<E> {
    // 若干方法...
}
```

所谓接口，就是一组方法的集合，只要一个类使用 implements 关键词申明自己实现了接口中的所有方法，那么就可以用这个接口的类型来接收这个类的实例化对象。

具体地，标准库提供的 LinkedList 类实现了 List 接口中的所有方法，且使用 implements 关键词申明自己实现了 List 接口：

```
// Deque 是 Queue 的子接口，包含了 Queue 接口的所有方法，所以实现了 Deque 就等于实现了 Queue 接口
// 还有若干接口，这里省略了
public class LinkedList<E> implements List<E>, Deque<E> {
    // ...

    // Queue 接口中声明的若干方法都会被实现
    boolean offer(E e) {...}
    E poll() {...}
    E peek() {...}
    // ...

    // List 接口的若干方法也都会被实现
    // ...
}
```

所以可以用 List 接口接收 LinkedList 类型，Queue 接口同理。

当然，编程语言的接口和继承往深里说有很多设计方法的学问，大家有兴趣可以自己搜索学习。因为我们当前的重点在数据结构和算法，所以掌握这些最最基本的知识完全足够你理解我写的 Java 算法代码了。

可视化面板使用方法和技巧

可视化面板编辑器已上线，可以在网页访问：

https://labuladong.online/algo-visualize/

另外，本站以及所有配套插件中嵌套的算法可视化面板也添加了「编辑」按钮，点击后即可直接修改并执行你的代码进行可视化。

目前可视化面板只支持 JavaScript 代码。如果你阅读可视化面板中的 JavaScript 代码有困难，我专门写了一个使用可视化面板的 极简 JavaScript 教程，帮助你 5 分钟上手。

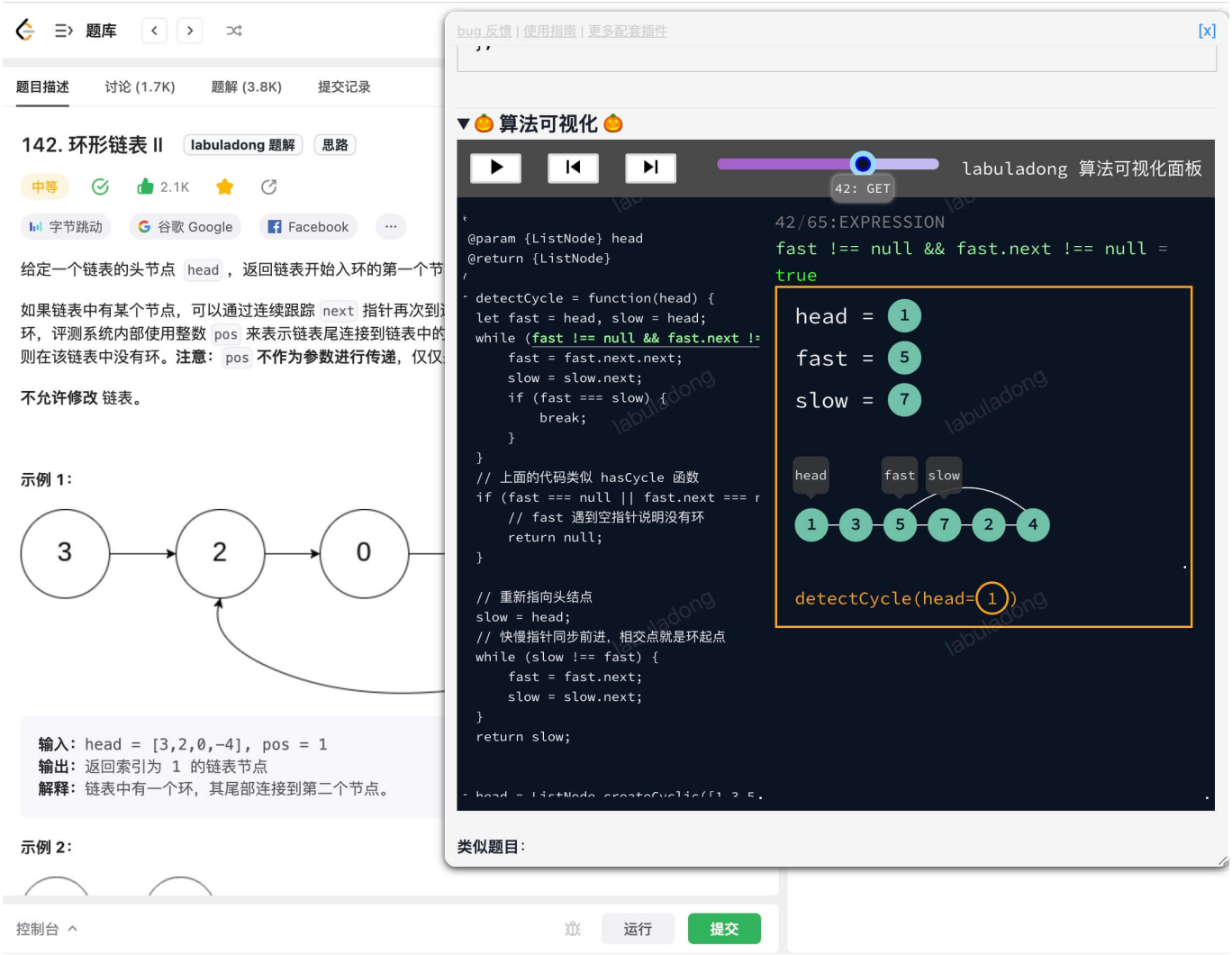
希望你能看到本文的最后，因为在最后一部分，我会结合具体的算法题场景教给你使用可视化面板的实用技巧，手把手带你利用这些技巧高效分析题目和算法的执行过程。等你认识到可视化面板的强大之处，你就会发现它能巨幅提升你对算法的理解深度。

我利用可视化面板的「讲解模式」手把手带大家体验一下可视化面板的主要功能。请你点击下方面板的按钮，全屏显示可视化面板，然后点击面板中的「讲解」按钮，开始体验吧。

我的所有算法代码都添加了动画可视化，且可视化面板全线支持我的配套刷题插件，包括 刷题网站、Chrome 插件、vscode 插件 和 Jetbrains 插件：

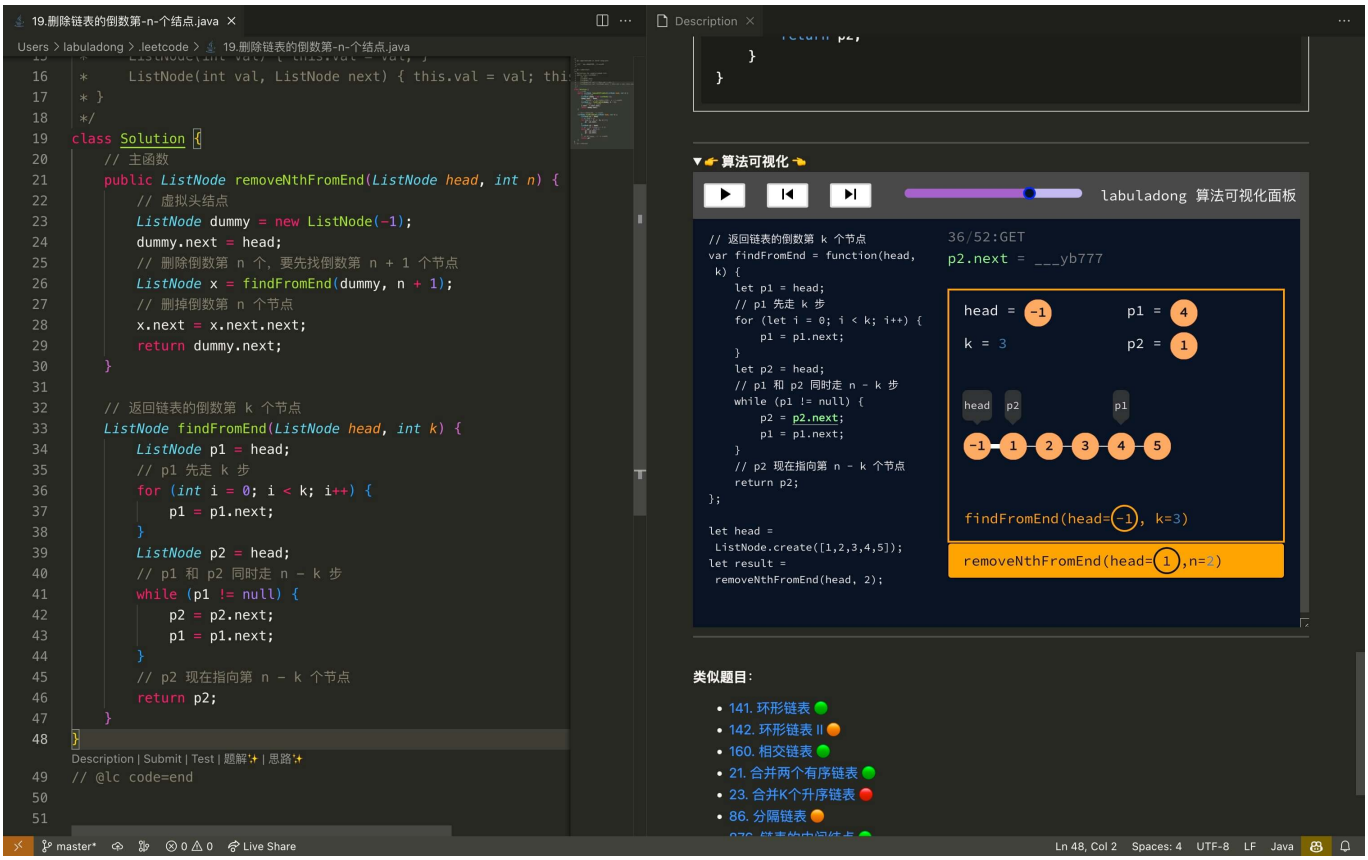
@slidestart

Chrome 刷题插件可视化面板

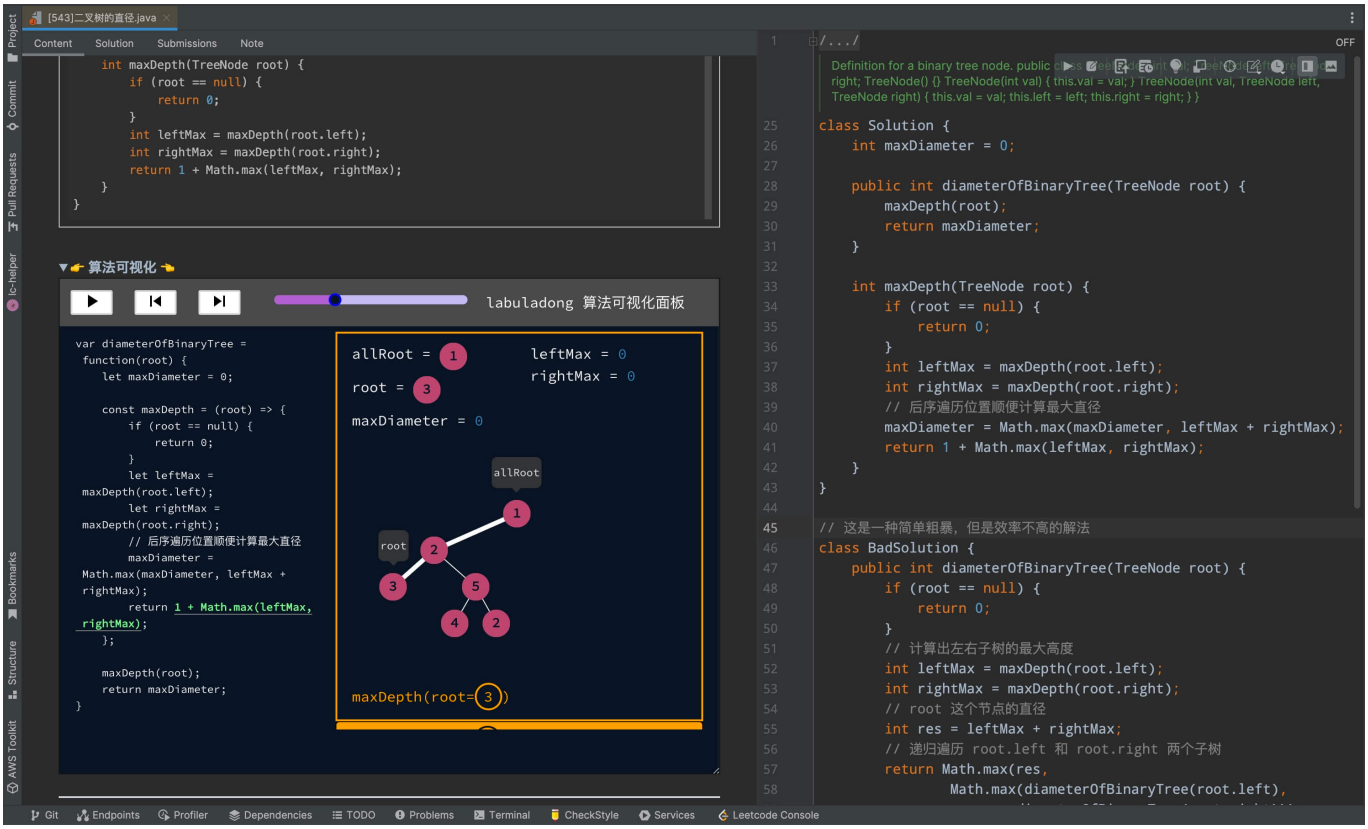


---

vscode 刷题插件可视化面板



Jetbrains 刷题插件可视化面板



@slideend

数据结构的可视化

简单展示一下吧。可视化面板不是一个简单的视频或者 GIF，而是交互式页面，你可以自己控制动画的播放，一步步查看代码的运行过程。我这里就展示一个简单的示例：检测单链表成环的算法。

> 环检测算法的核心逻辑就是快慢指针，慢指针 `slow` 每次只移动 1 步，快指针 `fast` 每次移动 2 步，如果单链表有环，最终它俩一定会相遇。
>
> 请你用鼠标不断点击可视化面板左侧代码中的 `if (slow === fast)` 这一行，就可以直观地看到快慢指针的追逐过程，直至它们相遇。

面板可以可视化数组、链表、二叉树、哈希表、哈希集合等常见数据结构，下面具体介绍一下如何创建这些数据结构。

标准库数据结构

类似数组、哈希表等 JavaScript 内置的数据结构，就正常初始化和操作它们即可，比如：

需要着重讲的是 力扣/LeetCode 中一些特殊的数据结构，比如单链表 `ListNode` 和二叉树 `TreeNode`，它们的基本定义和用法和力扣上一样，下面提供一些例子。

单链表

首先说一下单链表，单链表的每个节点有 `val, next` 属性，和力扣上的定义完全一致：

```
// 单链表节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function ListNode(val, next) {
    this.val = (val===undefined ? 0 : val)
    this.next = (next===undefined ? null : next)
}
```

你可以用 `ListNode.create` 方法快速创建一条单链表，这是一个简单的例子：

双链表

双链表的每个节点有 `val, prev, next` 属性，构造函数和力扣完全一致：

```
// 双链表节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function DoubleListNode(val, prev, next) {
    this.val = (val===undefined ? 0 : val)
    this.prev = (prev===undefined ? null : prev)
    this.next = (next===undefined ? null : next)
}
```

你可以用 `DoubleListNode.create` 方法快速创建一条双链表，或者手动组装双链表。这是一些简单的例子：

二叉树

再说一下二叉树，二叉树每个节点有 `val, left, right` 属性，构造函数和力扣保持一致：

```
// 二叉树节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function TreeNode(val, left, right) {
    this.val = (val===undefined ? 0 : val)
    this.left = (left===undefined ? null : left)
    this.right = (right===undefined ? null : right)
}
```

你可以用 `TreeNode.create` 方法快速创建一棵二叉树。注意我们用列表来表示二叉树，表示方式和 力扣题目中表示二叉树的方式 相同，这里举两个例子：

每个二叉树节点有 `val, left, right` 属性，和力扣上的定义完全一致。

多叉树

多叉树的每个节点有 `val, children` 属性，构造函数和力扣完全一致：

```
// 多叉树节点的定义
// 此类型已经内置，可以直接使用，不需要你再进行声明
function Node(val, children) {
    this.val = val;
    this.children = children;
}
```

你可以用 `Node.create` 来创建多叉树，大家主要注意一下多叉树的表示方法，我们还是用列表来表示多叉树，使用层序遍历的顺序，并用 `null` 来分割每组子节点。具体可以参考这道力扣题目 429. N 叉树的层序遍历 的示例。

每个多叉树节点有 `val, children` 属性，和力扣上的定义完全一致。

递归过程的可视化

递归算法是很多读者头痛的，我之前写过一篇 从树的角度理解一切递归算法，从理论上抽象出了递归算法最基本的思维模型和编写技巧。

简单来说就是：**把递归函数理解成递归树上的一个指针，回溯算法是在遍历一棵多叉树，并收集叶子节点的值；动态规划是在分解问题，用子问题的答案来推导原问题的答案。**

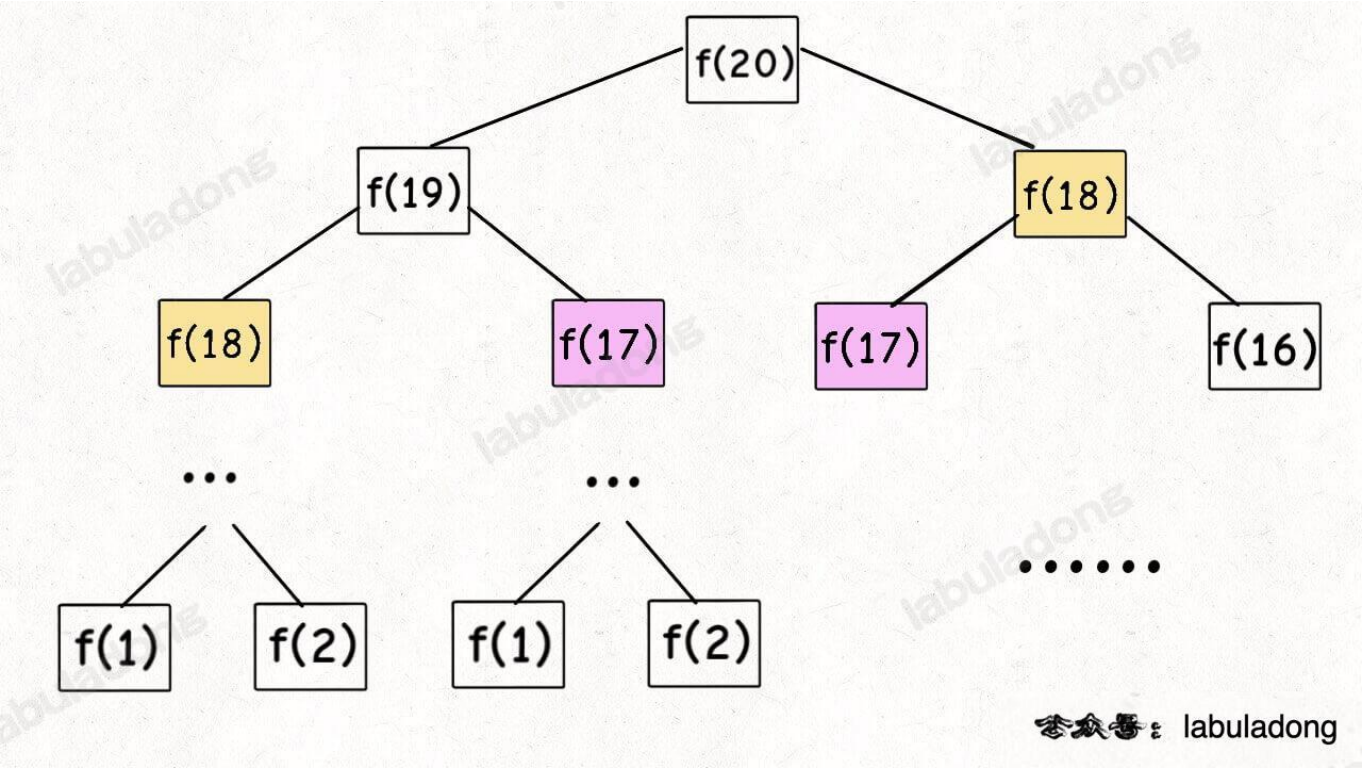
现在，我把文章中所阐述的思维模型融入了算法可视化面板，一定可以让你更直观地理解递归算法的执行过程。

对于递归算法，`@visualize status`，`@visualize choose` 和 `@visualize unchoose` 这几种注释可以帮到大忙，下面一一介绍。

`@visualize status` 举例

一句话，`@visualize status` 注释可以放在需要可视化的递归函数的上方，用来控制递归树上节点的值。

看个最简单的例子，我在讲解 动态规划算法核心框架 时画出了斐波那契问题的递归树，上层规模较大的问题逐渐被分解：



如何描述算法运行的过程？递归函数 `fib` 就好像递归树上的一个指针，它不断把原问题分解成子问题，当问题分解到 base case（叶子节点）时，开始逐层返回子问题的答案，推导出原问题的答案。

结合 `@visualize status` 注释，可以很直观地看到这个过程。下面这个面板是斐波那契算法运行的第 27 步，正在试图计算 `fib(3)` 的值：

`@visualize status` 注释写在 `fib` 函数上边，具体含义是：

1、对这个 `fib` 函数开启递归树可视化功能，每次递归调用会被可视化为递归树上的一个节点，函数参数中的 `n` 的值会作为状态显示在节点上。

2、`fib` 函数被视为一个遍历这棵递归树的指针，处于堆栈路径的树枝会加粗显示。

3、如果函数有返回值，那么当函数结束，计算出某个节点返回值时，鼠标移动到这个节点上，会显示该返回值。

1、定义函数时，必须使用 `var fib = function(n) { ... }` 的方式，不要使用 `function fib(n) { ... }` 或者箭头函数 `var fib = (n) => { ... }` 的方式，否则无法追踪递归过程。

2、`@visualize` 注释必须写在函数定义的上一行，否则无法追踪递归过程。

> 请你在步数栏输入 27 并敲回车，就可以跳转到第 27 步。把鼠标移动到节点 `(2)` 上，可以看到 `fib(2) = 1`，这说明 `fib(2)` 的值已经被计算出来了。

> 而处在递归路径上的 `(5),(4),(3)` 节点，它们的值还没被计算出来，你把鼠标移动上去也不会显示返回值。

> 请你尝试点击可视化面板的前进后退按钮，让算法向后运行几步，理解动态规划算法的递归树生长过程。

> 待到整棵递归树遍历完成之时，就是原问题 `fib(5)` 被计算出来之日，那时候整棵递归树上的所有节点都会显示返回值。
>
> 请你尝试拉动进度条到最后，看看这棵递归树的样子，并把鼠标移动到各个节点上，看看显示什么。

前面展示的斐波那契解法代码没有添加备忘录，是一个指数级时间复杂度的算法，现在我们来体验一下添加备忘录之后会对整棵递归树的生长产生什么影响。

> 👇 请你拉动进度条到算法的最后，看看递归树长什么样子。

> 👇 这是一个带备忘录优化的版本，请你拉动进度条到算法的最后，看看递归树长什么样子，和不带备忘录优化的递归树进行对比。

这样把整棵树可视化出来，你是不是就能很直观地理解动态规划通过备忘录消除重叠子问题的原理了？

`@visualize choose/unchoose` 举例

一句话，`@visualize choose/unchoose` 注释分别放在递归函数调用之前和之后，控制递归树树枝上的值。

我编一个简单的回溯算法的题目吧，比如让你写一个 `generateBinaryNumber` 函数，生成长度为 `n` 的二进制数，比如 `n = 3` 时，生成 `000, 001, 010, 011, 100, 101, 110, 111` 这 8 个二进制数。

这道题的解法代码也不复杂，我来展示一下如何用 `@visualize choose/unchoose` 注释来可视化回溯过程的：

你可以把鼠标移动到递归树上的任意节点，就可以显示出从根节点到该节点的路径上的所有信息。

你结合这棵递归树去理解递归算法的代码，是不是就很直观了？

场景练习题

下面我会结合具体的算法题目，向你提问，要求你利用可视化面板高效分析题目和算法的执行过程。我会给出问题的答案，你可以先自己思考，然后再看我的解答。

这部分主要用到的是可视化面板的「执行到指定代码位置」的功能：你可以点击代码的某一部分，然后算法就会执行到那里，并将执行过程可视化。**这个功能用得好，可以大幅提升你对算法的理解。**

热身

首先，我带大家熟悉一下「执行到指定代码位置」的功能，在之前展示的单链表成环算法中，我其实已经带你用过了，我让你不断点击代码中的 `if (slow === fast)` 这一行，观看快板指针的追逐过程：

为什么点击这部分就可以展示快慢指针的追逐呢？因为执行这部分代码时，`fast, slow` 指针刚刚移动完毕，所以点击这里，可以看到它们最完整的移动过程。

在下面的练习中，请你着重思考代码执行到某一部分时算法的状态，这样你就能更好地利用可视化面板，精确地控制算法的执行。

场景一：二叉树的前中后序遍历

> 上面这段代码是二叉树前中后序遍历，相信大家都不陌生。
>
> 请你点击播放按钮，查看这段代码的执行流程。播放时，上一步/下一步按钮将变成加速/减速按钮，用来调整播放速度。

你可以看到，二叉树将会被可视化出来，`root` 变量就好像一个指针，在二叉树上游走。我希望你能够使用「执行到指定代码位置」的功能，点击代码中的合适位置，精确控制 `root` 指针的移动顺序。

1、把面板复位，让 `root` 指针按照前序顺序访问整棵二叉树，即 `root` 指针按照 `1,2,4,3,5,6` 的顺序遍历二叉树。

2、把面板复位，`root` 指针按照中序顺序访问整棵二叉树，即 `root` 指针按照 `2,4,1,5,3,6` 的顺序遍历二叉树。

3、把面板复位，`root` 指针按照后序顺序访问整棵二叉树，即 `root` 指针按照 `4,2,5,6,3,1` 的顺序遍历二叉树。

这几个问题不难：

1、不断点击 `preorderResult.push(root.val)` 这部分代码。

2、不断点击 `inorderResult.push(root.val)` 这部分代码。

3、不断点击 `postorderResult.push(root.val)` 这部分代码。

场景二：观察递归树的生长

👇 这是刚才讲到的不带备忘录的斐波那契算法，现在请结合你对递归树的理解，点击代码中的合适位置，使得每点击一次，递归树就向下生长一个节点，直到递归树完全长成。

你可以不断点击代码中 `if (n < 2)` 这一行，观察递归树的生长过程。

为什么点击这部分就可以展示递归树的生长过程呢？因为这部分代码是递归函数的 base case，每次递归必然都会执行这个判断逻辑，而递归树的每个节点其实就对应着每次递归，所以点击这个 if 语句就可以看到递归树上节点的生长过程。

这是一个非常常用的技巧，可以用来观察递归过程。

场景三：快速获得一个穷举结果

👇 这是前文展示的全排列算法的可视化面板，在场景二中你观察了斐波那契问题的递归树生长，现在请你点击代码中的合适位置，观察全排列问题的递归树生长。

这个很简单，不断点击 `if (track.length === nums.length)` 这部分代码即可观察递归树的生长。

将面板复位。

现在，我不想看递归树一个一个节点地生长了，没什么意思，我希望你点击代码中的合适位置，每点击一次，递归树就生长出一条「树枝」。

这是个回溯算法中很常见的场景哦，因为每条「树枝」的末端是叶子节点，而叶子节点一般就存储这回溯算法穷举出来的一个结果。比如你看这道题，每条树枝末端的叶子节点就是一个排列结果。

不断点击 `res.push([...track])` 这部分代码，每次点击，回溯树都将生长出一条「树枝」。

因为叶子节点其实就是递归树结束生长的节点，即 `backtrack` 函数的 base case。那么我们只要点击 base case 中的代码，就可以直接跳到递归树的叶子节点，也就相当于生长出了一条树枝。

场景四：理解自顶向下/自底向上两种思维模式

我在 动态规划核心框架 中讲过，自顶向下用 `memo` 备忘录的递归解法和自底向上用 `dp` 数组的迭代解法本质上是一样的，`dp` 数组中的值和 `memo` 备忘录中的值完全一样，两种解法可以互相转化。

下面我将同时给你自顶向下的递归解法和自底向上的迭代解法，请你完成如下题目：

1、在自顶向下递归解法的面板上，点击代码的合适位置，使得每次点击都在 `memo` 中更新一个元素。

2、在自底向上迭代解法的面板上，点击代码的合适位置，使得每次点击都在 `dp` 中更新一个元素。

3、对比两个解法中 `dp` 数组和 `memo` 数组的更新，理解自顶向下的递归解法和自底向上的迭代解法本质上是相同的。

1、在自顶向下递归解法的面板上，每次点击 `memo[n] = dp(memo, n - 1) + dp(memo, n - 2)` 这部分代码，都会在 `memo` 数组中更新一个元素。

2、在自底向上迭代解法的面板上，每次点击 `dp[i] = dp[i - 1] + dp[i - 2]` 这部分代码，都会在 `dp` 数组中更新一个元素。

3、结合可视化面板应该不难理解，不过你会发现 `memo[1]` 和 `dp[1]` 的值不一样，那是因为我们把 `dp[1]` 作为 base case 设置了初始值，而递归解法中的 base case 是直接 return 的，并不需要设置在 `memo` 中。

可视化代码编辑器使用指南

可视化面板教程已统一汇总到 这里。

34. 在排序数组中查找元素的第一个和最后一个位置

LeetCode	力扣	难度
34. Find First and Last Position of Element in Sorted Array	34. 在排序数组中查找元素的第一个和最后一个位置	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索

给定一个按照升序排列的整数数组 `nums`，和一个目标值 `target`，找出给定目标值在数组中的开始位置和结束位置。如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

示例 1：

```
输入：nums = [5,7,7,8,8,10], target = 8
输出：[3,4]
```

示例 2：

```
输入：nums = [5,7,7,8,8,10], target = 6
输出：[-1,-1]
```

基本思路

> 本文有视频版：二分搜索核心框架套路

PS：这道题在《算法小抄》 的第 71 页。

二分搜索的难点就在于如何搜索左侧边界和右侧边界，代码的边界的控制非常考验你的微操，这也是很多人知道二分搜索原理但是很难写对代码的原因。

二分搜索框架详解 专门花了很大篇幅讨论如何写对二分搜索算法，总结来说：

写对二分搜索的关键在于搞清楚搜索边界，到底是开区间还是闭区间？到底应该往左侧收敛还是应该往右侧收敛？

深入的探讨请看详细题解。

- 详细题解：我写了首诗，把二分搜索算法变成了默写题

解法代码

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        return new int[]{left_bound(nums, target), right_bound(nums,
target)};
    }

    int left_bound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        // 搜索区间为 [left, right]
        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) {
                // 收缩右侧边界
                right = mid - 1;
            }
        }
        // 检查出界情况
        if (left >= nums.length || nums[left] != target) {
            return -1;
        }
        return left;
    }

    int right_bound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] == target) {
                // 这里改成收缩左侧边界即可
                left = mid + 1;
            }
        }
        // 这里改为检查 right 越界的情况，见下图
        if (right < 0 || nums[right] != target) {
            return -1;
        }
        return right;
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

- **类似题目**：
  - 704. 二分查找 🟢
  - 剑指 Offer 53 - I. 在排序数组中查找数字 I 🟢

704. 二分查找

LeetCode	力扣	难度
704. Binary Search	704. 二分查找	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二分搜索**

给定有 `n` 个元素的升序整型数组 `nums` 和一个目标值 `target`，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。

示例 1:

```
输入：nums = [-1,0,3,5,9,12], target = 9
输出：4
解释：9 出现在 nums 中并且下标为 4
```

示例 2:

```
输入：nums = [-1,0,3,5,9,12], target = 2
输出：-1
解释：2 不存在 nums 中因此返回 -1
```

基本思路

> 本文有视频版：二分搜索核心框架套路

PS：这道题在《算法小抄》 的第 71 页。

二分搜索的基本形式，不过并不实用，比如 `target` 重复出现多次，本算法得出的索引位置是不确定的。

更常见的二分搜索形式是搜索左侧边界和右侧边界，即对于 `target` 重复出现多次的情景，计算 `target` 的最小索引和最大索引。

这几种二分搜索的形式的详细探讨见详细题解。

- **详细题解：我写了首诗，把二分搜索算法变成了默写题**

解法代码

```
class Solution {
    public int search(int[] nums, int target) {
        int left = 0;
```

```
        int right = nums.length - 1; // 注意

        while(left <= right) {
            int mid = left + (right - left) / 2;
            if(nums[mid] == target)
                return mid;
            else if (nums[mid] < target)
                left = mid + 1; // 注意
            else if (nums[mid] > target)
                right = mid - 1; // 注意
        }
        return -1;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 34. 在排序数组中查找元素的第一个和最后一个位置 🟠
  - 剑指 Offer 53 - I. 在排序数组中查找数字 I 🟢

剑指 Offer 53 - I. 在排序数组中查找数字 I

LeetCode	力扣	难度
剑指Offer53-I. 在排序数组中查找数字 LCOF	剑指Offer53-I. 在排序数组中查找数字 I	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数组

统计一个数字在排序数组中出现的次数。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: 2
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: 0
```

基本思路

这道题考察二分搜索查找 `target` 的左右边界，和 34. 在排序数组中查找元素的第一个和最后一个位置 有些类似，用二分搜索找到左右边界的索引，就可以判断重复出现的次数了。

- 详细题解：我写了首诗，把二分搜索算法变成了默写题

解法代码

```
class Solution {
    public int search(int[] nums, int target) {
        int left_index = left_bound(nums, target);
        if (left_index == -1) {
            return 0;
        }
        int right_index = right_bound(nums, target);
        // 根据左右边界即可推导出元素出现的次数
        return right_index - left_index + 1;
    }

    int left_bound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        // 搜索区间为 [left, right]
```

```
        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) {
                // 收缩右侧边界
                right = mid - 1;
            }
        }
        // 检查出界情况
        if (left >= nums.length || nums[left] != target) {
            return -1;
        }
        return left;
    }

    int right_bound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] == target) {
                // 这里改成收缩左侧边界即可
                left = mid + 1;
            }
        }
        // 这里改为检查 right 越界的情况，见下图
        if (right < 0 || nums[right] != target) {
            return -1;
        }
        return right;
    }
}
```

- **类似题目**：
  - 34. 在排序数组中查找元素的第一个和最后一个位置
  - 704. 二分查找

35. 搜索插入位置

LeetCode	力扣	难度
35. Search Insert Position	35. 搜索插入位置	🟢

- **标签：二分搜索**

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请必须使用时间复杂度为 `O(log n)` 的算法。

示例 1:

```
输入: nums = [1,3,5,6], target = 5
输出: 2
```

示例 2:

```
输入: nums = [1,3,5,6], target = 2
输出: 1
```

基本思路

这道题就是考察搜索左侧边界的二分算法的细节理解，前文 二分搜索详解 着重讲了数组中存在目标元素重复的情况，没仔细讲目标元素不存在的情况。

当目标元素 `target` 不存在数组 `nums` 中时，搜索左侧边界的二分搜索的返回值可以做以下几种解读：

1、返回的这个值是 `nums` 中大于等于 `target` 的最小元素索引。

2、返回的这个值是 `target` 应该插入在 `nums` 中的索引位置。

3、返回的这个值是 `nums` 中小于 `target` 的元素个数。

比如在有序数组 `nums = [2,3,5,7]` 中搜索 `target = 4`，搜索左边界的二分算法会返回 2，你带入上面的说法，都是对的。

所以以上三种解读都是等价的，可以根据具体题目场景灵活运用，显然这里我们需要的是第二种。

解法代码

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        return left_bound(nums, target);
    }

    // 搜索左侧边界的二分算法
    int left_bound(int[] nums, int target) {
        if (nums.length == 0) return -1;
        int left = 0;
        int right = nums.length; // 注意

        while (left < right) { // 注意
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                right = mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid; // 注意
            }
        }
        return left;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 剑指 Offer II 068. 查找插入位置 🟢

剑指 Offer II 068. 查找插入位置

这道题和 35. 搜索插入位置 相同。

240. 搜索二维矩阵 II

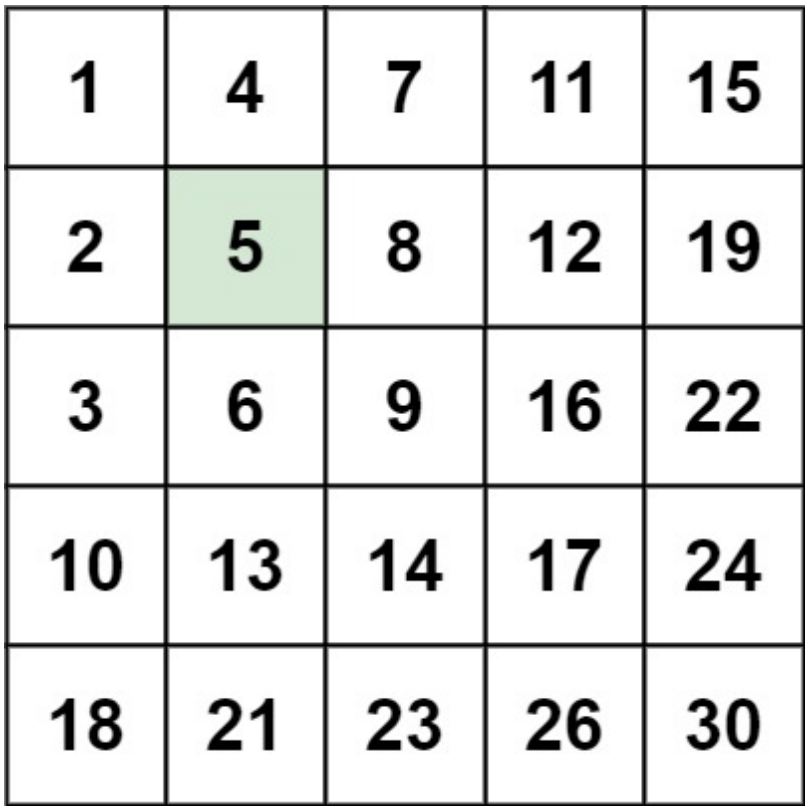
LeetCode	力扣	难度
240. Search a 2D Matrix II	240. 搜索二维矩阵 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组，数组双指针

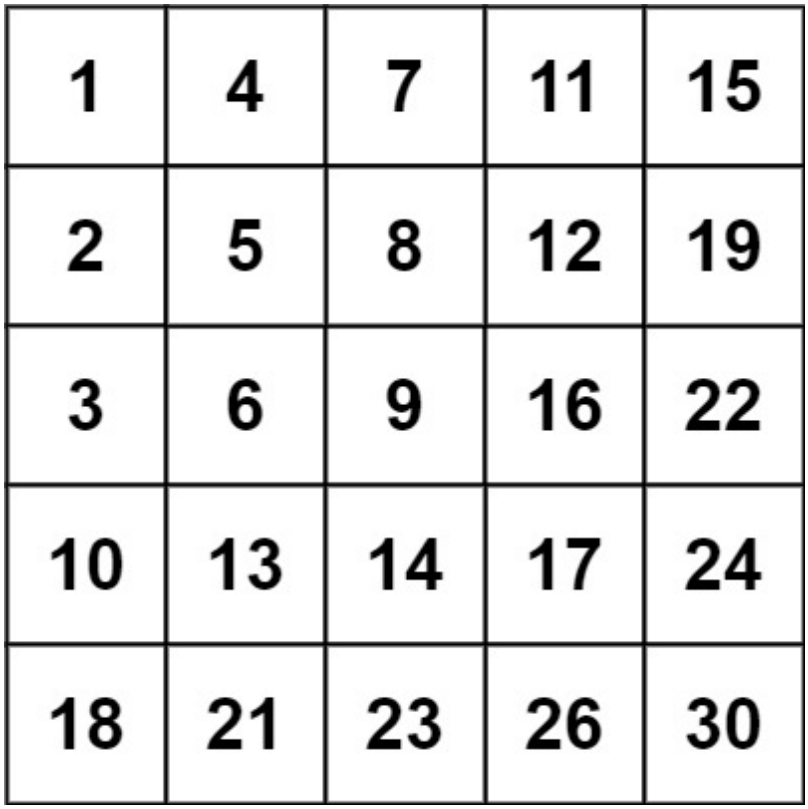
编写一个高效的算法来搜索 `m x n` 矩阵 `matrix` 中的一个目标值 `target`。该矩阵具有以下特性：每行的元素从左到右升序排列；每列的元素从上到下升序排列。

示例 1：




```
输入：matrix = [[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]], target = 5
输出：true
```

示例 2：




```
输入：matrix = [[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]], target = 20
输出：false
```

基本思路

作为 74. 搜索二维矩阵 的进阶题目，这道题不算 二分搜索算法，更像 一个方法秒杀所有 N 数之和问题，因为它们的思想上有些类似。

这道题说 `matrix` 从上到下递增，从左到右递增，显然左上角是最小元素，右下角是最大元素。**我们如果想高效在 `matrix` 中搜索一个元素，肯定需要从某个角开始**，比如说从左上角开始，然后每次只能向右或向下移动，不要走回头路。

如果真从左上角开始的话，就会发现无论向右还是向下走，元素大小都会增加，那么到底向右还是向下？不确定，那只好用类似 动态规划算法 的思路穷举了。

但实际上不用这么麻烦，我们不要从左上角开始，而是从右上角开始，规定只能向左或向下移动。

你注意，如果向左移动，元素在减小，如果向下移动，元素在增大，这样的话我们就可以根据当前位置的元素和 `target` 的相对大小来判断应该往哪移动，不断接近从而找到 `target` 的位置。

当然，如果你想从左下角开始，规定只能向右或向上移动也可以，具体看代码吧。

解法代码

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
```

```
        int m = matrix.length, n = matrix[0].length;
        // 初始化在右上角
        int i = 0, j = n - 1;
        while (i < m && j >= 0) {
            if (matrix[i][j] == target) {
                return true;
            }
            if (matrix[i][j] < target) {
                // 需要大一点，往下移动
                i++;
            } else {
                // 需要小一点，往左移动
                j--;
            }
        }
        // while 循环中没有找到，则 target 不存在
        return false;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 剑指 Offer 04. 二维数组中的查找 🟠

566. 重塑矩阵

LeetCode	力扣	难度
566. Reshape the Matrix	566. 重塑矩阵	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

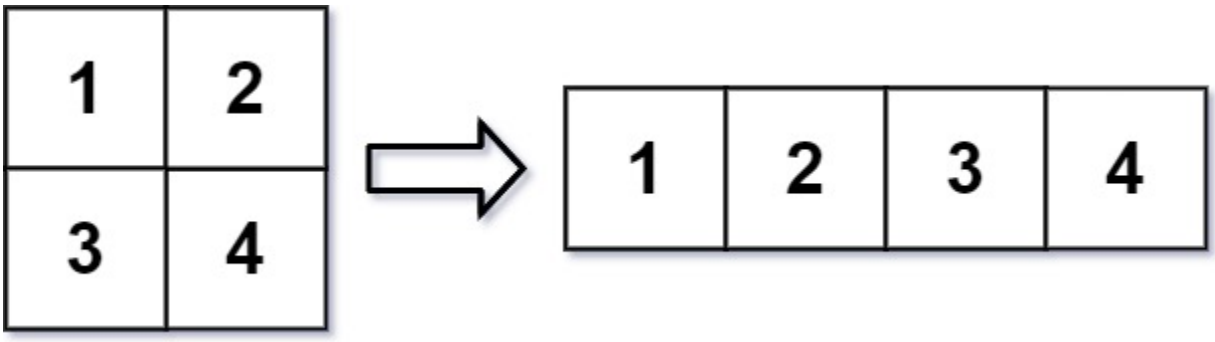
- **标签：二维矩阵，数组**

在 MATLAB 中，有一个非常有用的函数 `reshape`，它可以将一个 `m x n` 矩阵重塑为另一个大小不同（`r x c`）的新矩阵，但保留其原始数据。

给你一个由二维数组 `mat` 表示的 `m x n` 矩阵，以及两个正整数 `r` 和 `c`，分别表示想要的重构的矩阵的行数和列数。

重构后的矩阵需要将原始矩阵的所有元素以相同的 **行遍历顺序** 填充。如果具有给定参数的 `reshape` 操作是可行且合理的，则输出新的重塑矩阵；否则，输出原始矩阵。

示例 1：




```
输入：mat = [[1,2],[3,4]], r = 1, c = 4
输出：[[1,2,3,4]]
```

基本思路

这题不难，但指出了一个必知必会的算法技巧：**多维坐标之间的映射转换**。

我直接说结论：**任何多维数组都可以被映射到一维，所以甭管几维数组，你统一把多维的坐标转化成一维，然后再从一维坐标转化到多维**。

所以这道题，我们先把二维坐标转化成一维，然后再转化成不同 shape 的二维坐标即可。

我这里实现了通用的 `get/set` 函数，关于坐标维度转化的运用，还可以看下 74. 搜索二维矩阵。

解法代码

```java
class Solution {
    public int[][] matrixReshape(int[][] mat, int r, int c) {
        int m = mat.length, n = mat[0].length;
        // 如果想成功 reshape，元素个数应该相同
        if (r * c != m * n) {
            return mat;
        }

        int[][] res = new int[r][c];
        for (int i = 0; i < m * n; i++) {
            set(res, i, get(mat, i));
        }
        return res;
    }

    // 通过一维坐标访问二维数组中的元素
    int get(int[][] matrix, int index) {
        int m = matrix.length, n = matrix[0].length;
        // 计算二维中的横纵坐标
        int i = index / n, j = index % n;
        return matrix[i][j];
    }

    // 通过一维坐标设置二维数组中的元素
    void set(int[][] matrix, int index, int value) {
        int m = matrix.length, n = matrix[0].length;
        // 计算二维中的横纵坐标
        int i = index / n, j = index % n;
        matrix[i][j] = value;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 519. 随机翻转矩阵 🟠

74. 搜索二维矩阵

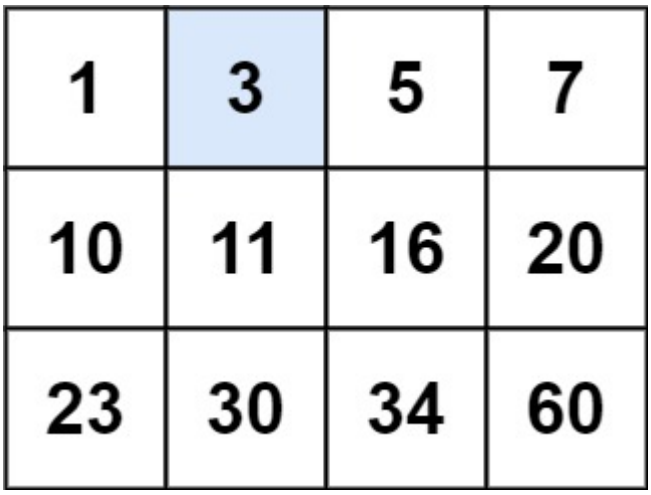
LeetCode	力扣	难度
74. Search a 2D Matrix	74. 搜索二维矩阵	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数组

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。该矩阵具有如下特性：每行中的整数从左到右按升序排列；每行的第一个整数大于前一行的最后一个整数。

示例 1：




```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
输出：true
```

基本思路

只要知道二维数组的的行数 `m` 和列数 `n`，二维数组的坐标 `(i, j)` 可以映射成一维的 `index = i * n + j`；反过来也可以通过一维 `index` 反解出二维坐标 `i = index / n, j = index % n`。

我们在前文 二分搜索框架详解 详细讲解了二分搜索的几种框架，本题可以实现一个 `get` 函数把二维数组 `matrix` 的元素访问抽象成在一维数组中访问元素，然后直接施展最基本的二分搜索即可。

解法代码

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int m = matrix.length, n = matrix[0].length;
        // 把二维数组映射到一维
        int left = 0, right = m * n - 1;
```

```
        // 前文讲的标准的二分搜索框架
        while(left <= right) {
            int mid = left + (right - left) / 2;
            if(get(matrix, mid) == target)
                return true;
            else if (get(matrix, mid) < target)
                left = mid + 1;
            else if (get(matrix, mid) > target)
                right = mid - 1;
        }
        return false;
    }

    // 通过一维坐标访问二维数组中的元素
    int get(int[][] matrix, int index) {
        int m = matrix.length, n = matrix[0].length;
        // 计算二维中的横纵坐标
        int i = index / n, j = index % n;
        return matrix[i][j];
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 240. 搜索二维矩阵 II 🟠
  - 566. 重塑矩阵 🟢
  - 剑指 Offer 04. 二维数组中的查找 🟠

剑指 Offer 04. 二维数组中的查找

这道题和 240. 搜索二维矩阵 II 相同。

519. 随机翻转矩阵

LeetCode	力扣	难度
519. Random Flip Matrix	519. 随机翻转矩阵	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：哈希表，数据结构，随机算法**

给你一个 `m x n` 的二元矩阵 `matrix`，且所有值被初始化为 `0`。请你设计一个算法，随机选取一个满足 `matrix[i][j] == 0` 的下标 `(i, j)`，并将它的值变为 `1`。所有满足 `matrix[i][j] == 0` 的下标 `(i, j)` 被选取的概率应当均等。

尽量最少调用内置的随机函数，并且优化时间和空间复杂度。

实现 `Solution` 类：

- `Solution(int m, int n)` 使用二元矩阵的大小 `m` 和 `n` 初始化该对象。
- `int[] flip()` 返回一个满足 `matrix[i][j] == 0` 的随机下标 `[i, j]`，并将其对应格子中的值变为 `1`。
- `void reset()` 将矩阵中所有的值重置为 `0`。

示例：

```
输入
["Solution", "flip", "flip", "flip", "reset", "flip"]
[[3, 1], [], [], [], [], []]
输出
[null, [1, 0], [2, 0], [0, 0], null, [2, 0]]

解释
Solution solution = new Solution(3, 1);
solution.flip();  // 返回 [1, 0]，此时返回 [0,0]、[1,0] 和 [2,0] 的概率应当相同
solution.flip();  // 返回 [2, 0]，因为 [1,0] 已经返回过了，此时返回 [2,0] 和
[0,0] 的概率应当相同
solution.flip();  // 返回 [0, 0]，根据前面已经返回过的下标，此时只能返回 [0,0]
solution.reset(); // 所有值都重置为 0，并可以再次选择下标返回
solution.flip();  // 返回 [2, 0]，此时返回 [0,0]、[1,0] 和 [2,0] 的概率应当相同
```

基本思路

这道题的需求看起来很简单，但说实话这题并不是初学者能够高效解决的，下面我来一步步拆解这道题。

首先，你需要直到对多维坐标进行「降维」的技巧，把二维坐标映射到一维，因为我们生成的随机数可以认为是一维坐标。这个技巧不难，可以先去做一下 566. 重塑矩阵。

其次，你需要看过我的这篇文章 给我常数时间，我可以删除/查找数组中的任意元素 并跟着做了 380. 常数时间插入、删除和获取随机元素，题目说要尽可能减少调用随机函数，就是在考你把数组元素换到尾部的技巧。

本题其实面对的是相同的问题，相当于是给你一个长度为 `m * n` 的数组，请你每次取出一个随机数并删除它。

所以解决本题也使用同样的思路，抽象出一个长度为 `m * n` 的数组，每次取出一个随机元素之后把这个元素置换到数组尾部删掉，同时用一个哈希表记录被置换元素的映射，保证每次随机不会得到重复元素。

具体的实现看代码吧，本质上其实和 380 是一模一样的。

解法代码

```
class Solution {

    int m, n;
    // 抽象一维数组的长度
    int len;
    // 已删除元素到尾部未删除元素的映射
    HashMap<Integer, Integer> deletedToExist;
    // 这个 deletedToExist 存的到底是什么呢？举个例子你就明白了：
    // 假设数组 [1,2,3,4,5]，我第一次随机选择了元素 2，
    // 然后我就会在 deletedToExist 记录一个映射 2 -> 5，并删除最后的元素 5
    // 现在数组是 [1,2,3,4]，我第二次随机选择又选到了 2，
    // 但不能重复选择同一个元素，所以我选择 5，因为 deletedToExist[2] = 5
    // 然后我再更新 deletedToExist，记录 2 -> 4 并删除最后的元素 4
    // 以此类推，这样就能保证随机选择到相同的元素时也能等概率地返回不同的元素

    Random random = new Random();

    public Solution(int m, int n) {
        this.m = m;
        this.n = n;
        this.len = m * n;
        this.deletedToExist = new HashMap<>();
    }

    public int[] flip() {
        int rand = random.nextInt(len);
        // 这个随机数可能已经被删掉了（刚才已经被选过）
        int res = rand;
        if (deletedToExist.containsKey(rand)) {
            res = deletedToExist.get(rand);
        }
        // 把 rand 置换到数组尾部
        int last = len - 1;
        // 尾部的那个元素也可能已经被删掉了
        if (deletedToExist.containsKey(last)) {
            last = deletedToExist.get(last);
        }
        deletedToExist.put(rand, last);
```

```
            // 把尾部的这个元素删掉
            len--;
            // 一维坐标转化成二维坐标
            return new int[]{res / n, res % n};
        }

        public void reset() {
            this.len = this.m * this.n;
            this.deletedToExist.clear();
        }
    }
```

354. 俄罗斯套娃信封问题

LeetCode	力扣	难度
354. Russian Doll Envelopes	354. 俄罗斯套娃信封问题	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：一维动态规划，二分搜索，动态规划

给你一个二维整数数组 `envelopes`，其中 `envelopes[i] = [wi, hi]`，表示第 `i` 个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算 **最多能有多少个**信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

```
输入：envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出：3
解释：最多信封的个数为 3，组合为：[2,3] => [5,4] => [6,7]。
```

基本思路

PS：这道题在《算法小抄》的第 104 页。

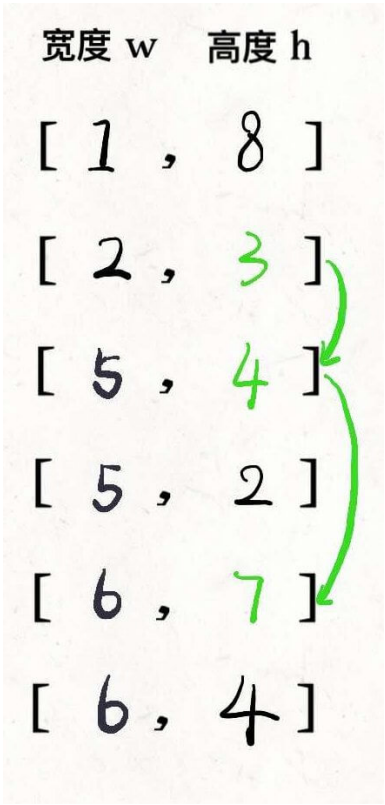
300. 最长递增子序列 在一维数组里面求元素的最长递增子序列，本题相当于在二维平面里面求最长递增子序列。

假设信封是由 `(w, h)` 这样的二维数对形式表示的，思路如下：

先对宽度 `w` 进行升序排序，如果遇到 `w` 相同的情况，则按照高度 `h` 降序排序。之后把所有的 `h` 作为一个数组，在这个数组上计算 LIS 的长度就是答案。

画个图理解一下，先对这些数对进行排序：

然后在 h 上寻找最长递增子序列：



- **详细题解：动态规划设计：最长递增子序列**

解法代码

```
class Solution {
    public int maxEnvelopes(int[][] envelopes) {
```

```
        int n = envelopes.length;
        // 按宽度升序排列，如果宽度一样，则按高度降序排列
        Arrays.sort(envelopes, new Comparator<int[]>()
        {
            public int compare(int[] a, int[] b) {
                return a[0] == b[0] ?
                    b[1] - a[1] : a[0] - b[0];
            }
        });
        // 对高度数组寻找 LIS
        int[] height = new int[n];
        for (int i = 0; i < n; i++)
            height[i] = envelopes[i][1];

        return lengthOfLIS(height);
    }

    /* 返回 nums 中 LIS 的长度 */
    public int lengthOfLIS(int[] nums) {
        int piles = 0, n = nums.length;
        int[] top = new int[n];
        for (int i = 0; i < n; i++) {
            // 要处理的扑克牌
            int poker = nums[i];
            int left = 0, right = piles;
            // 二分查找插入位置
            while (left < right) {
                int mid = (left + right) / 2;
                if (top[mid] >= poker)
                    right = mid;
                else
                    left = mid + 1;
            }
            if (left == piles) piles++;
            // 把这张牌放到牌堆顶
            top[left] = poker;
        }
        // 牌堆数就是 LIS 长度
        return piles;
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

- **类似题目**：
  - 300. 最长递增子序列 🟠

392. 判断子序列

LeetCode	力扣	难度
392. Is Subsequence	392. 判断子序列	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二分搜索，子序列**

给定字符串**s**和**t**，判断**s**是否为**t**的子序列。

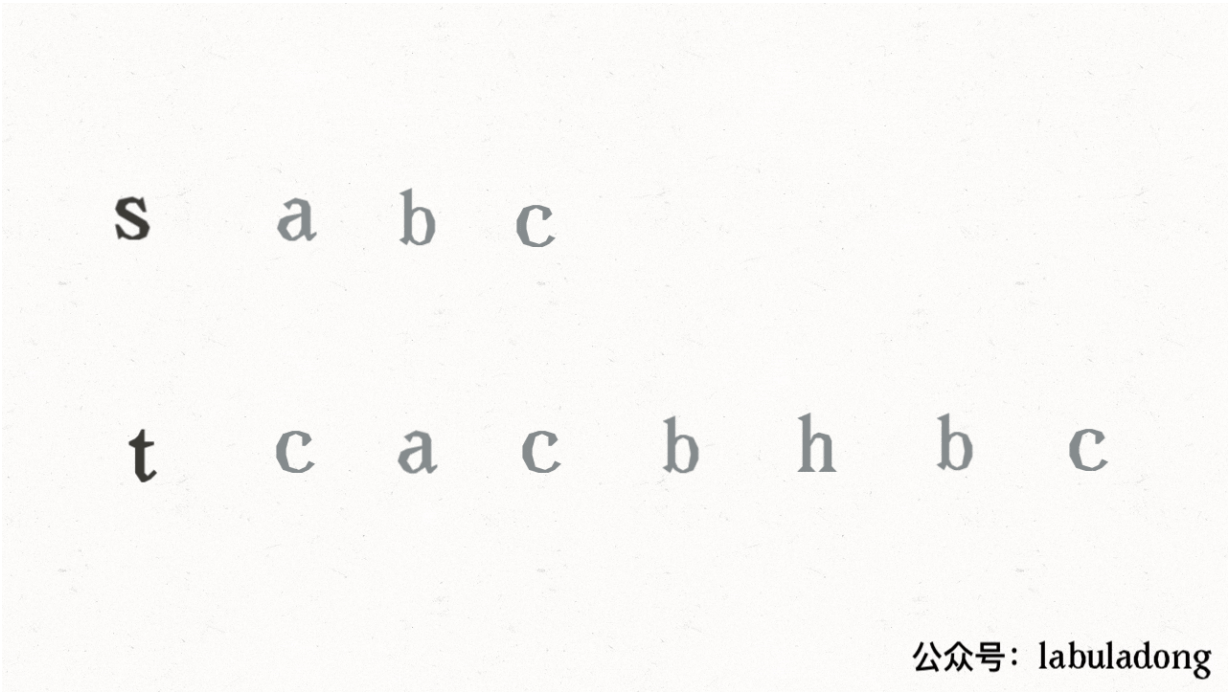
进阶：如果有大量输入的 `S`，称作 `S1, S2, ... , Sk` 其中 k > 10 亿，你需要依次检查它们是否为 `T` 的子序列。在这种情况下，你会怎样改变代码？

示例 1：

```
输入：s = "abc", t = "ahbgdc"
输出：true
```

基本思路

力扣上的这道题很简单，利用双指针 `i, j` 分别指向 `s, t`，一边前进一边匹配子序列。



但这题的进阶比较有难度，需要利用二分搜索技巧来判断子序列，见详细题解。

- **详细题解：二分查找高效判定子序列**

解法代码

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int i = 0, j = 0;
        while (i < s.length() && j < t.length()) {
            if (s.charAt(i) == t.charAt(j)) {
                i++;
            }
            j++;
        }
        return i == s.length();
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 792. 匹配子序列的单词数 🟠

658. 找到 K 个最接近的元素

LeetCode	力扣	难度
658. Find K Closest Elements	658. 找到 K 个最接近的元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数组双指针

给定一个 **排序好** 的数组 `arr`，两个整数 `k` 和 `x`，从数组中找到最靠近 `x`（两数之差最小）的 `k` 个数。返回的结果必须要是按升序排好的。

整数 `a` 比整数 `b` 更接近 `x` 需要满足：`|a - x| < |b - x|` 或者 `|a - x| == |b - x|` 且 `a < b`。

示例 1：

```
输入：arr = [1,2,3,4,5], k = 4, x = 3
输出：[1,2,3,4]
```

基本思路

我们就说一个最简单直接的方式：用 二分查找算法详解 中介绍的搜索左侧边界的二分查找算法找到 `x` 的位置，然后用 数组双指针技巧汇总 中解决 5. 最长回文子串 的从中间向两端的双指针算法找到这 `k` 个元素。

为什么是搜索左侧边界的二分搜索？可以仔细看下前文 二分查找算法详解，有提到左侧边界二分搜索的几种理解方式。

另外，因为题目要求返回的结果必须按升序排序，所以我们必须用 `LinkedList` 来从两端添加结果，使得结果有序。

解法代码

```
class Solution {
    public List<Integer> findClosestElements(int[] arr, int k, int x) {
        // 二分搜索找到 x 的位置
        int p = left_bound(arr, x);
        // 两端都开的区间 (left, right)
        int left = p - 1, right = p;
        LinkedList<Integer> res = new LinkedList<>();
        // 扩展区间，直到区间内包含 k 个元素
        while (right - left - 1 < k) {
            if (left == -1) {
                res.addLast(arr[right]);
                right++;
            } else if (right == arr.length) {
                res.addFirst(arr[left]);
```

```
                left--;
            } else if (x - arr[left] > arr[right] - x) {
                res.addLast(arr[right]);
                right++;
            } else {
                res.addFirst(arr[left]);
                left--;
            }
        }
        return res;
    }

    // 搜索左侧边界的二分搜索
    int left_bound(int[] nums, int target) {
        int left = 0;
        int right = nums.length;

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                right = mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid;
            }
        }
        return left;
    }
}
```

▶ 代码可视化动画（需安装 Chrome 插件）

793. 阶乘函数后 K 个零

LeetCode	力扣	难度
793. Preimage Size of Factorial Zeroes Function	793. 阶乘函数后 K 个零	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二分搜索，数学**

`f(x)` 是 `x!` 末尾是 0 的数量，`x! = 1 * 2 * 3 * ... * x`，且 `0! = 1`。

例如，`f(3) = 0`，因为 3! = 6 的末尾没有 0；而 `f(11) = 2`，因为 11! = 39916800 末端有 2 个 0。给定 `K`，找出多少个非负整数 `x`，能满足 `f(x) = K`。

示例 1：

```
输入：K = 0
输出：5
解释：0!, 1!, 2!, 3!, and 4! 均符合 K = 0 的条件。
```

基本思路

这题需要复用 172. 阶乘后的零 这道题的解法函数 `trailingZeroes`。

搜索有多少个 `n` 满足 `trailingZeroes(n) == K`，其实就是在问，满足条件的 `n` 最小是多少，最大是多少，最大值和最小值一减，就可以算出来有多少个 `n` 满足条件了，对吧？那不就是 二分查找 中「搜索左侧边界」和「搜索右侧边界」这两个事儿嘛？

观察题目给出的数据取值范围，`n` 可以在区间 `[0, LONG_MAX]` 中取值，寻找满足 `trailingZeroes(n) == K` 的左侧边界和右侧边界，相减即是答案。

- **详细题解：讲两道常考的阶乘算法题**

解法代码

```
class Solution {
    public int preimageSizeFZF(int K) {
        // 左边界和右边界之差 + 1 就是答案
        return (int)(right_bound(K) - left_bound(K) + 1);
    }

    // 逻辑不变，数据类型全部改成 long
    long trailingZeroes(long n) {
        long res = 0;
        for (long d = n; d / 5 > 0; d = d / 5) {
            res += d / 5;
```

```
        }
        return res;
    }

    /* 搜索 trailingZeroes(n) == K 的左侧边界 */
    long left_bound(int target) {
        long lo = 0, hi = Long.MAX_VALUE;
        while (lo < hi) {
            long mid = lo + (hi - lo) / 2;
            if (trailingZeroes(mid) < target) {
                lo = mid + 1;
            } else if (trailingZeroes(mid) > target) {
                hi = mid;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    /* 搜索 trailingZeroes(n) == K 的右侧边界 */
    long right_bound(int target) {
        long lo = 0, hi = Long.MAX_VALUE;
        while (lo < hi) {
            long mid = lo + (hi - lo) / 2;
            if (trailingZeroes(mid) < target) {
                lo = mid + 1;
            } else if (trailingZeroes(mid) > target) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
        }

        return lo - 1;
    }
}
```

- **类似题目**：
  - 172. 阶乘后的零 🟠

852. 山脉数组的峰顶索引

LeetCode	力扣	难度
852. Peak Index in a Mountain Array	852. 山脉数组的峰顶索引	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数组

符合下列属性的数组 arr 称为 **山脉数组**：

1、arr.length >= 3 2、存在 i（0 < i < arr.length - 1）使得：arr[0] < arr[1] < ... arr[i-1] < arr[i] 且 arr[i] > arr[i+1] > ... > arr[arr.length - 1]。

给你由整数组成的山脉数组 arr，返回任何满足 arr[0] < arr[1] < ... arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1] 的下标 i。

示例 1：

```
输入：arr = [0,1,0]
输出：1
```

基本思路

二分搜索框架详解 的经典应用，不过这道题和 162. 寻找峰值 差不多，直接把 162 题的解法复制过来即可通过。

解法代码

```
class Solution {
    public int peakIndexInMountainArray(int[] nums) {
        // 取两端都闭的二分搜索
        int left = 0, right = nums.length - 1;
        // 因为题目必然有解，所以设置 left == right 为结束条件
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[mid + 1]) {
                // mid 本身就是峰值或其左侧有一个峰值
                right = mid;
            } else {
                // mid 右侧有一个峰值
                left = mid + 1;
            }
        }
        return left;
```

```
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 剑指 Offer II 069. 山峰数组的顶部 🟢

剑指 Offer II 069. 山峰数组的顶部

这道题和 852. 山脉数组的峰顶索引 相同。

1011. 在 D 天内送达包裹的能力

LeetCode	力扣	难度
1011. Capacity To Ship Packages Within D Days	1011. 在 D 天内送达包裹的能力	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索

传送带上的第 `i` 个包裹的重量为 `weights[i]`，运输船每天都会来运输这些包裹，每天装载的包裹重量之和不能超过船的最大运载重量。如果要在 `D` 天内将所有包裹运输完毕，求运输船的最低运载能力。

示例 1：

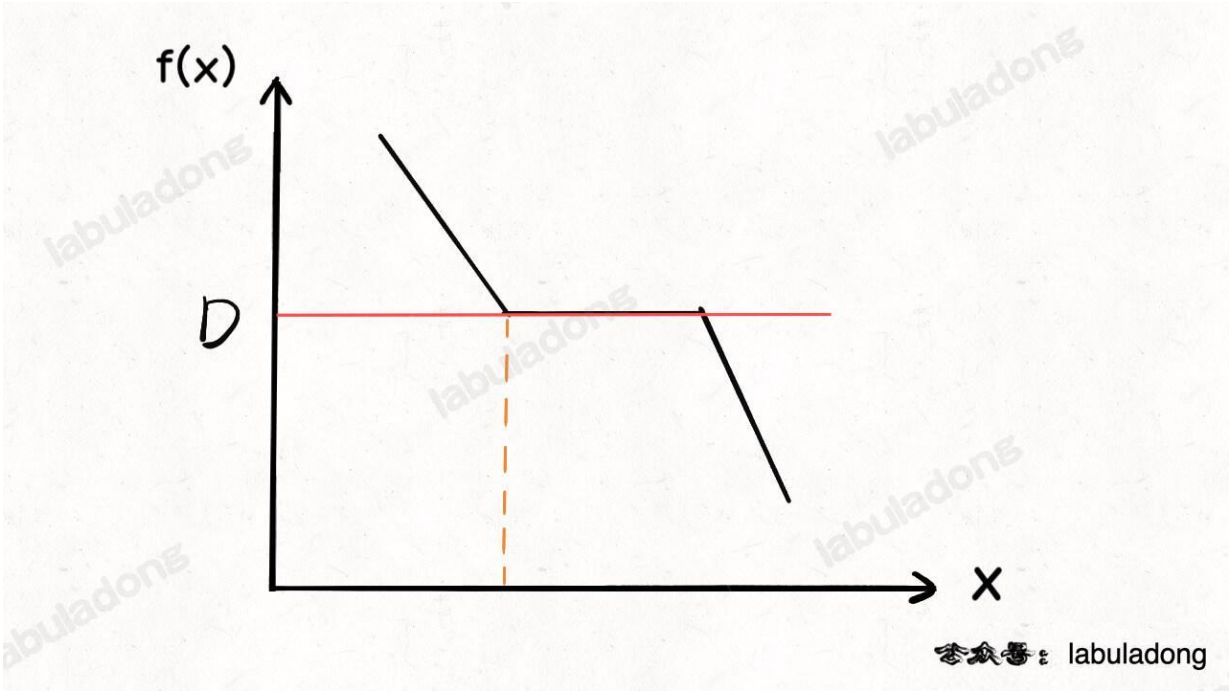
```
输入：weights = [1,2,3,4,5,6,7,8,9,10], D = 5
输出：15
解释：
船舶最低载重 15 就能够在 5 天内送达所有包裹，如下所示：
第 1 天：1, 2, 3, 4, 5
第 2 天：6, 7
第 3 天：8
第 4 天：9
第 5 天：10

请注意，货物必须按照给定的顺序装运，因此使用载重能力为 14 的船舶并将包装分成 (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) 是不允许的。
```

基本思路

PS：这道题在《算法小抄》 的第 360 页。

正如 二分搜索思路运用篇 所说，二分搜索的套路比较固定，如果遇到一个算法问题，能够确定 `x, f(x), target` 分别是什么，并写出单调函数 `f` 的代码，那么就可以运用二分搜索的思路求解。

船的运载能力就是自变量 `x`，运输天数和运载能力成反比，所以可以定义 `f(x)` 表示 `x` 的运载能力下需要的运输天数，`target` 显然就是运输天数 `D`，我们要在 `f(x) == D` 的约束下，算出船的最小载重 `x`。



关于本题二分搜索的具体思路见详细题解。

- **详细题解：实际二分搜索时的思维框架**

解法代码

```
class Solution {
    public int shipWithinDays(int[] weights, int days) {
        int left = 0;
        int right = 1;
        for (int w : weights) {
            left = Math.max(left, w);
            right += w;
        }

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (f(weights, mid) <= days) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }

    // 定义：当运载能力为 x 时，需要 f(x) 天运完所有货物
    // f(x) 随着 x 的增加单调递减
    int f(int[] weights, int x) {
        int days = 0;
        for (int i = 0; i < weights.length; ) {
            // 尽可能多装货物
```

```
            int cap = x;
            while (i < weights.length) {
                if (cap < weights[i]) break;
                else cap -= weights[i];
                i++;
            }
            days++;
        }
        return days;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 410. 分割数组的最大值 🔴
  - 875. 爱吃香蕉的珂珂 🟠
  - 剑指 Offer II 073. 狒狒吃香蕉 🟠

875. 爱吃香蕉的珂珂

LeetCode	力扣	难度
875. Koko Eating Bananas	875. 爱吃香蕉的珂珂	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索

珂珂喜欢吃香蕉。这里有 N 堆香蕉，第 i 堆中有 piles[i] 根香蕉。警卫已经离开了，将在 H 小时后回来。

珂珂可以决定她吃香蕉的速度 K（单位：根/小时）。每个小时，她将会选择一堆香蕉，从中吃掉 K 根。如果这堆香蕉少于 K 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。计算她可以在 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
```

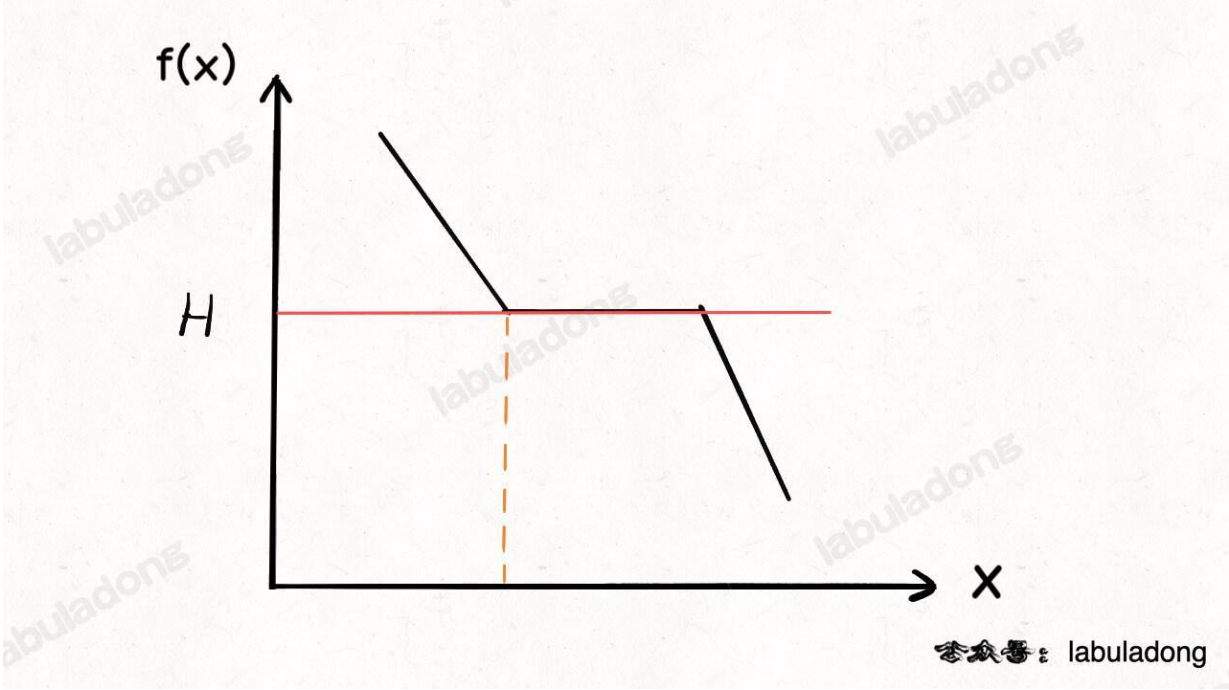
基本思路

PS：这道题在《算法小抄》 的第 360 页。

正如 二分搜索思路运用篇 所说，二分搜索的套路比较固定，如果遇到一个算法问题，能够确定 x, f(x), target 分别是什么，并写出单调函数 f 的代码，那么就可以运用二分搜索的思路求解。

这题珂珂吃香蕉的速度 K 就是自变量 x，吃完所有香蕉所需的时间就是单调函数 f(x)，target 就是吃香蕉的时间限制 H。我们需要调整 x，使得 f(x) 尽可能接近 target，也就是说，我们需要找到最小的 x，使得 f(x) <= target。

它们的关系如下图：



关于本题二分搜索的具体思路见详细题解。

- **详细题解：实际二分搜索时的思维框架**

解法代码

```
class Solution {
    public int minEatingSpeed(int[] piles, int H) {
        int left = 1;
        int right = 1000000000 + 1;

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (f(piles, mid) <= H) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }

    // 定义：速度为 x 时，需要 f(x) 小时吃完所有香蕉
    // f(x) 随着 x 的增加单调递减
    int f(int[] piles, int x) {
        int hours = 0;
        for (int i = 0; i < piles.length; i++) {
            hours += piles[i] / x;
            if (piles[i] % x > 0) {
                hours++;
            }
```

```
            }
            return hours;
        }
    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 1011. 在 D 天内送达包裹的能力 🟠
  - 410. 分割数组的最大值 🔴
  - 剑指 Offer II 073. 狒狒吃香蕉 🟠

剑指 Offer II 073. 狒狒吃香蕉

这道题和 875. 爱吃香蕉的珂珂 相同。

1201. 丑数 III

LeetCode	力扣	难度
1201. Ugly Number III	1201. 丑数 III	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数学，链表双指针

给你四个整数：`n` 、`a` 、`b` 、`c`，请你设计一个算法来找出第 `n` 个丑数。丑数是可以被 `a` **或** `b` **或** `c` 整除的 **正整数**。

示例 1：

```
输入：n = 3, a = 2, b = 3, c = 5
输出：4
解释：丑数序列为 2, 3, 4, 5, 6, 8, 9, 10... 其中第 3 个是 4。
```

基本思路

这道题和 264. 丑数 II 有些类似，你把第 264 题合并有序链表的解法照搬过来稍微改改就能解决这道题，代码我写在 `Solution2` 里面了。

但是注意题目给的数据规模，`a, b, c, n` 都是非常大的数字，如果用合并有序链表的思路，其复杂度是 O(N)，对于这么大的数据规模来说也是比较慢的，应该会超时，无法通过一些测试用例。

这道题的正确解法难度比较大，难点在于你要把一些数学知识和 二分搜索技巧 结合起来才能高效解决这个问题。

首先，根据 二分查找的实际运用 中讲到的二分搜索运用方法，我们可以抽象出一个单调递增的函数 `f`：

`f(num, a, b, c)` 计算 `[1..num]` 中，能够整除 `a` 或 `b` 或 `c` 的数字的个数，显然函数 `f` 的返回值是随着 `num` 的增加而增加的（单调递增）。

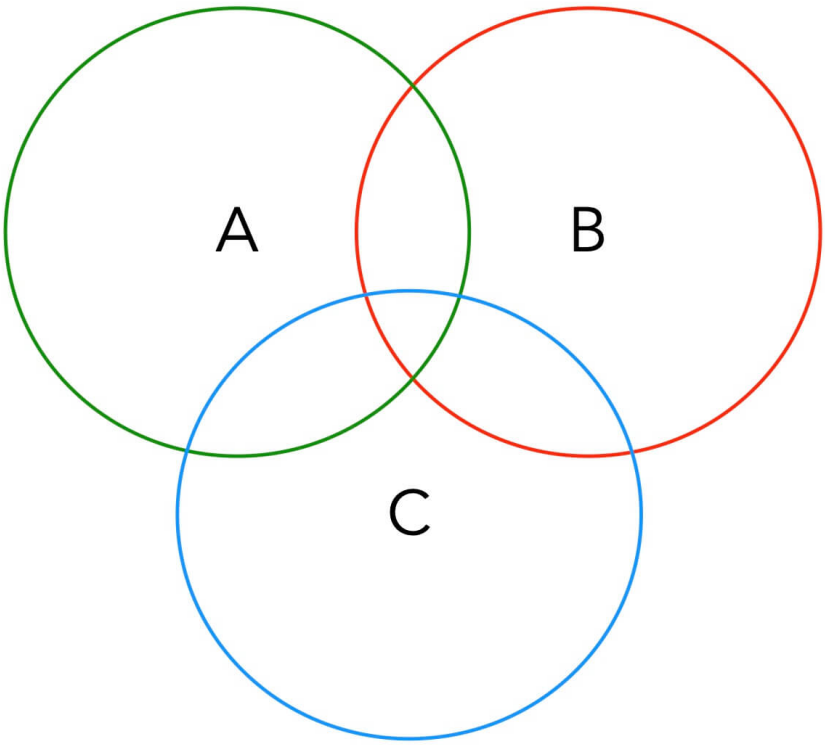
题目让我们求第 `n` 个能够整除 `a` 或 `b` 或 `c` 的数字是什么，也就是说我们要找到一个 `num`，使得 `f(num, a, b, c) == n`。

有了上述思路，就可以按照 二分查找的实际运用 中给出的模板运用二分搜索算法了。

关键说一下函数 `f` 怎么实现，这里面涉及集合论定理以及最小公因数、最小公倍数的计算方法。

首先，`[1..num]` 中，我把能够整除 `a` 的数字归为集合 `A`，能够整除 `b` 的数字归为集合 `B`，能够整除 `c` 的数字归为集合 `C`，那么 `len(A) = num / a, len(B) = num / b, len(C) = num / c`，这个很好理解。

但是 `f(num, a, b, c)` 的值肯定不是 `num / a + num / b + num / c` 这么简单，因为你注意有些数字可能可以被 `a, b, c` 中的两个数或三个数同时整除，如下图：



按照集合论的算法，这个集合中的元素应该是：A + B + C - A ∩ B - A ∩ C - B ∩ C + A ∩ B ∩ C。结合上图应该很好理解。

问题来了，`A, B, C` 三个集合的元素个数我们已经算出来了，但如何计算像 `A ∩ B` 这种交集的元素个数呢？

其实也很容易想明白，`A ∩ B` 的元素个数就是 `n / lcm(a, b)`，其中 `lcm` 是计算最小公倍数（Least Common Multiple）的函数。

类似的，`A ∩ B ∩ C` 的元素个数就是 `n / lcm(lcm(a, b), c)` 的值。

现在的问题是，最小公倍数怎么求？

直接记住定理吧：`lcm(a, b) = a * b / gcd(a, b)`，其中 `gcd` 是计算最大公因数（Greatest Common Divisor）的函数。

现在的问题是，最大公因数怎么求？

这应该是经典算法了，我们一般叫辗转相除算法（或者欧几里得算法）。

好了，套娃终于套完了，我们可以把上述思路翻译成解法，注意本题数据规模比较大，有时候需要用 `long` 类型防止 `int` 溢出，具体看我的代码实现以及注释吧。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
// 二分搜索 + 数学解法
class Solution {
    public int nthUglyNumber(int n, int a, int b, int c) {
```

```
        // 题目说本题结果在 [1, 2 * 10^9] 范围内，
        // 所以就按照这个范围初始化两端都闭的搜索区间
        int left = 1, right = (int) 2e9;
        // 搜索左侧边界的二分搜索
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (f(mid, a, b, c) < n) {
                // [1..mid] 中的元素个数不足 n，所以目标在右侧
                left = mid + 1;
            } else {
                // [1..mid] 中的元素个数大于 n，所以目标在左侧
                right = mid - 1;
            }
        }
        return left;
    }

    // 计算 [1..num] 之间有多少个能够被 a 或 b 或 c 整除的数字
    long f(int num, int a, int b, int c) {
        long setA = num / a, setB = num / b, setC = num / c;
        long setAB = num / lcm(a, b);
        long setAC = num / lcm(a, c);
        long setBC = num / lcm(b, c);
        long setABC = num / lcm(lcm(a, b), c);
        // 集合论定理：A + B + C - A ∩ B - A ∩ C - B ∩ C + A ∩ B ∩ C
        return setA + setB + setC - setAB - setAC - setBC + setABC;
    }

    // 计算最大公因数（辗转相除/欧几里得算法）
    long gcd(long a, long b) {
        if (a < b) {
            // 保证 a > b
            return gcd(b, a);
        }
        if (b == 0) {
            return a;
        }
        return gcd(b, a % b);
    }

    // 最小公倍数
    long lcm(long a, long b) {
        // 最小公倍数就是乘积除以最大公因数
        return a * b / gcd(a, b);
    }
}

// 用合并单链表的思路（超时）
class Solution2 {
    public int nthUglyNumber(int n, int a, int b, int c) {
        // 可以理解为三个有序链表的头结点的值
        long productA = a, productB = b, productC = c;
        // 可以理解为合并之后的有序链表上的指针
        int p = 1;
```

```
        long min = -666;

        // 开始合并三个有序链表，获取第 n 个节点的值
        while (p <= n) {
            // 取三个链表的最小结点
            min = Math.min(Math.min(productA, productB), productC);
            p++;
            // 前进最小结点对应链表的指针
            if (min == productA) {
                productA += a;
            }
            if (min == productB) {
                productB += b;
            }
            if (min == productC) {
                productC += c;
            }
        }
        return (int) min;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 263. 丑数 🟢
  - 264. 丑数 II 🟠
  - 313. 超级丑数 🟠
  - 剑指 Offer 49. 丑数 🟠

剑指 Offer 53 - II. 0～n-1中缺失的数字

LeetCode	力扣	难度
剑指Offer53-II. 缺失的数字 LCOF	剑指Offer53-II. 0～n-1中缺失的数字	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数组

一个长度为 n-1 的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围 0～n-1 之内。在范围 0～n-1 内的 n 个数字中有且只有一个数字不在该数组中，请找出这个数字。

示例 1:

```
输入: [0,1,3]
输出: 2
```

示例 2:

```
输入: [0,1,2,3,4,5,6,7,9]
输出: 8
```

基本思路

这道题考察 二分查找算法。常规的二分搜索让你在 `nums` 中搜索目标值 `target`，但这道题没有给你一个显式的 `target`，怎么办呢？

其实，二分搜索的关键在于，**你是否能够找到一些规律，能够在搜索区间中一次排除掉一半**。比如让你在 `nums` 中搜索 `target`，你可以通过判断 `nums[mid]` 和 `target` 的大小关系判断 `target` 在左边还是右边，一次排除半个数组。

所以这道题的关键是，你是否能够找到一些规律，能够判断缺失的元素在哪一边？

其实是有规律的，你可以观察 `nums[mid]` 和 `mid` 的关系，如果 `nums[mid]` 和 `mid` 相等，则缺失的元素在右半边，如果 `nums[mid]` 和 `mid` 不相等，则缺失的元素在左半边。

二分查找算法 中说到了二分搜索的几种形式，我就用搜索左侧边界的二分搜索定位缺失的元素位置。

解法代码

```
class Solution {
    public int missingNumber(int[] nums) {
        // 搜索左侧的二分搜索
        int left = 0, right = nums.length - 1;
```

```
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > mid) {
                // mid 和 nums[mid] 不对应，说明左边有元素缺失
                right = mid - 1;
            } else {
                // mid 和 nums[mid] 对应，说明元素缺失在右边
                left = mid + 1;
            }
        }
        return left;
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

3. 无重复字符的最长子串

LeetCode	力扣	难度
3. Longest Substring Without Repeating Characters	3. 无重复字符的最长子串	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：滑动窗口

给定一个字符串 `s`，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：s = "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。
```

基本思路

> 本文有视频版：滑动窗口算法核心模板框架

PS：这道题在《算法小抄》 的第 85 页。

这题比其他滑动窗口的题目简单，连 `need` 和 `valid` 都不需要，而且更新窗口内数据也只需要简单的更新计数器 `window` 即可。

当 `window[c]` 值大于 1 时，说明窗口中存在重复字符，不符合条件，就该移动 `left` 缩小窗口了。

另外，要在收缩窗口完成后更新 `res`，因为窗口收缩的 while 条件是存在重复元素，换句话说收缩完成后一定保证窗口中没有重复。

- 详细题解：我写了首诗，把滑动窗口算法变成了默写题

解法代码

```cpp
class Solution {
    public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> window;

        int left = 0, right = 0;
        int res = 0; // 记录结果
        while (right < s.size()) {
            char c = s[right];
            right++;
            // 进行窗口内数据的一系列更新
            window[c]++;
```

```
                // 判断左侧窗口是否要收缩
                while (window[c] > 1) {
                    char d = s[left];
                    left++;
                    // 进行窗口内数据的一系列更新
                    window[d]--;
                }
                // 在这里更新答案
                res = max(res, right - left);
            }
            return res;
        }
};
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 438. 找到字符串中所有字母异位词 🟠
  - 567. 字符串的排列 🟠
  - 76. 最小覆盖子串 🔴
  - 剑指 Offer 48. 最长不含重复字符的子字符串 🟠
  - 剑指 Offer II 014. 字符串中的变位词 🟠
  - 剑指 Offer II 015. 字符串中的所有变位词 🟠
  - 剑指 Offer II 016. 不含重复字符的最长子字符串 🟠
  - 剑指 Offer II 017. 含有所有字符的最短字符串 🔴

438. 找到字符串中所有字母异位词

LeetCode	力扣	难度
438. Find All Anagrams in a String	438. 找到字符串中所有字母异位词	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组双指针，滑动窗口

给定两个字符串 s 和 p，找到 s 中所有 p 的异位词子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词指由相同字母重排列形成的字符串（包括相同的字符串）。

示例 1:

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

示例 2:

```
输入: s = "abab", p = "ab"
输出: [0,1,2]
解释:
起始索引等于 0 的子串是 "ab", 它是 "ab" 的异位词。
起始索引等于 1 的子串是 "ba", 它是 "ab" 的异位词。
起始索引等于 2 的子串是 "ab", 它是 "ab" 的异位词。
```

基本思路

> 本文有视频版：滑动窗口算法核心模板框架

PS：这道题在《算法小抄》 的第 85 页。

这题和 567. 字符串的排列 一样，只不过找到一个合法异位词（排列）之后将它的起始索引加入结果列表即可。

滑动窗口算法难度略高，具体的算法原理以及算法模板见详细题解。

- 详细题解：我写了首诗，把滑动窗口算法变成了默写题

解法代码

```cpp
class Solution {
    public:
    vector<int> findAnagrams(string s, string t) {
        unordered_map<char, int> need, window;
        for (char c : t) need[c]++;

        int left = 0, right = 0;
        int valid = 0;
        vector<int> res; // 记录结果
        while (right < s.size()) {
            char c = s[right];
            right++;
            // 进行窗口内数据的一系列更新
            if (need.count(c)) {
                window[c]++;
                if (window[c] == need[c])
                    valid++;
            }
            // 判断左侧窗口是否要收缩
            while (right - left >= t.size()) {
                // 当窗口符合条件时，把起始索引加入 res
                if (valid == need.size())
                    res.push_back(left);
                char d = s[left];
                left++;
                // 进行窗口内数据的一系列更新
                if (need.count(d)) {
                    if (window[d] == need[d])
                        valid--;
                    window[d]--;
                }
            }
        }
        return res;
    }
};
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- 类似题目：
  - 3. 无重复字符的最长子串 🟠
  - 567. 字符串的排列 🟠
  - 76. 最小覆盖子串 🔴
  - 剑指 Offer 48. 最长不含重复字符的子字符串 🟠
  - 剑指 Offer II 014. 字符串中的变位词 🟠
  - 剑指 Offer II 015. 字符串中的所有变位词 🟠
  - 剑指 Offer II 016. 不含重复字符的最长子字符串 🟠
  - 剑指 Offer II 017. 含有所有字符的最短字符串 🔴

567. 字符串的排列

LeetCode	力扣	难度
567. Permutation in String	567. 字符串的排列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数组双指针，滑动窗口**

给你两个字符串 `s1` 和 `s2`，写一个函数来判断 `s2` 是否包含 `s1` 的排列（`s1` 的排列之一是 `s2` 的**子串**）。如果是，返回 `true`，否则返回 `false`。

示例 1：

```
输入：s1 = "ab" s2 = "eidbaooo"
输出：true
解释：s2 包含 s1 的排列之一 ("ba").
```

基本思路

> 本文有视频版：滑动窗口算法核心模板框架

PS：这道题在《算法小抄》 的第 85 页。

和子数组/子字符串相关的题目，很可能就是要考察滑动窗口算法，往这方面思考就行了。

这道题，相当于你一个 `S` 和一个 `T`，请问你 `S` 中是否存在一个子串，包含 `T` 中所有字符且不包含其他字符？

如果这样想的话就和 76. 最小覆盖子串 有些类似了。

一般来说滑动窗口算法难度略高，需要你掌握算法原理以及算法模板辅助，见详细题解吧。

- **详细题解：我写了首诗，把滑动窗口算法变成了默写题**

解法代码

```
class Solution {
    public:

    // 判断 s 中是否存在 t 的排列
    bool checkInclusion(string t, string s) {
        unordered_map<char, int> need, window;
        for (char c : t) need[c]++;

        int left = 0, right = 0;
        int valid = 0;
        while (right < s.size()) {
```

```
                char c = s[right];
                right++;
                // 进行窗口内数据的一系列更新
                if (need.count(c)) {
                    window[c]++;
                    if (window[c] == need[c])
                        valid++;
                }

                // 判断左侧窗口是否要收缩
                while (right - left >= t.size()) {
                    // 在这里判断是否找到了合法的子串
                    if (valid == need.size())
                        return true;
                    char d = s[left];
                    left++;
                    // 进行窗口内数据的一系列更新
                    if (need.count(d)) {
                        if (window[d] == need[d])
                            valid--;
                        window[d]--;
                    }
                }
            }
            // 未找到符合条件的子串
            return false;
        }
};
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 3. 无重复字符的最长子串 🟠
  - 438. 找到字符串中所有字母异位词 🟠
  - 76. 最小覆盖子串 🔴
  - 剑指 Offer 48. 最长不含重复字符的子字符串 🟠
  - 剑指 Offer II 014. 字符串中的变位词 🟠
  - 剑指 Offer II 015. 字符串中的所有变位词 🟠
  - 剑指 Offer II 016. 不含重复字符的最长子字符串 🟠
  - 剑指 Offer II 017. 含有所有字符的最短字符串 🔴

76. 最小覆盖子串

LeetCode	力扣	难度
76. Minimum Window Substring	76. 最小覆盖子串	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组双指针，滑动窗口

给你一个字符串 `s` 、一个字符串 `t`，返回 `s` 中涵盖 `t` 所有字符的最小子串；如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""`。

基本思路

> 本文有视频版：滑动窗口算法核心模板框架

PS：这道题在《算法小抄》 的第 85 页。

这题就是典型的滑动窗口类题目，一般来说难度略高，解法框架如下：

```
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
```

```
    }
}
```

具体的算法原理看详细题解吧，这里写不下。

- **详细题解：我写了首诗，把滑动窗口算法变成了默写题**

解法代码

```
class Solution {
    public:
    string minWindow(string s, string t) {
        unordered_map<char, int> need, window;
        for (char c : t) need[c]++;

        int left = 0, right = 0;
        int valid = 0;
        // 记录最小覆盖子串的起始索引及长度
        int start = 0, len = INT_MAX;
        while (right < s.size()) {
            // c 是将移入窗口的字符
            char c = s[right];
            // 右移窗口
            right++;
            // 进行窗口内数据的一系列更新
            if (need.count(c)) {
                window[c]++;
                if (window[c] == need[c])
                    valid++;
            }

            // 判断左侧窗口是否要收缩
            while (valid == need.size()) {
                // 在这里更新最小覆盖子串
                if (right - left < len) {
                    start = left;
                    len = right - left;
                }
                // d 是将移出窗口的字符
                char d = s[left];
                // 左移窗口
                left++;
                // 进行窗口内数据的一系列更新
                if (need.count(d)) {
                    if (window[d] == need[d])
                        valid--;
                    window[d]--;
                }
            }
        }
        // 返回最小覆盖子串
```

```
        return len == INT_MAX ?
               "" : s.substr(start, len);
    }
};
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 3. 无重复字符的最长子串 🟠
  - 438. 找到字符串中所有字母异位词 🟠
  - 567. 字符串的排列 🟠
  - 剑指 Offer 48. 最长不含重复字符的子字符串 🟠
  - 剑指 Offer II 014. 字符串中的变位词 🟠
  - 剑指 Offer II 015. 字符串中的所有变位词 🟠
  - 剑指 Offer II 016. 不含重复字符的最长子字符串 🟠
  - 剑指 Offer II 017. 含有所有字符的最短字符串 🔴

剑指 Offer 48. 最长不含重复字符的子字符串

这道题和 3. 无重复字符的最长子串 相同。

剑指 Offer II 014. 字符串中的变位词

这道题和 567. 字符串的排列 相同。

剑指 Offer II 015. 字符串中的所有变位词

这道题和 438. 找到字符串中所有字母异位词 相同。

剑指 Offer II 016. 不含重复字符的最长子字符串

这道题和 3. 无重复字符的最长子串 相同。

剑指 Offer II 017. 含有所有字符的最短字符串

这道题和 76. 最小覆盖子串 相同。

155. 最小栈

LeetCode	力扣	难度
155. Min Stack	155. 最小栈	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，栈，设计

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 `MinStack` 类：

- `MinStack()` 初始化堆栈对象。
- `void push(int val)` 将元素 val 推入堆栈。
- `void pop()` 删除堆栈顶部的元素。
- `int top()` 获取堆栈顶部的元素。
- `int getMin()` 获取堆栈中的最小元素。

示例 1:

```
输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

基本思路

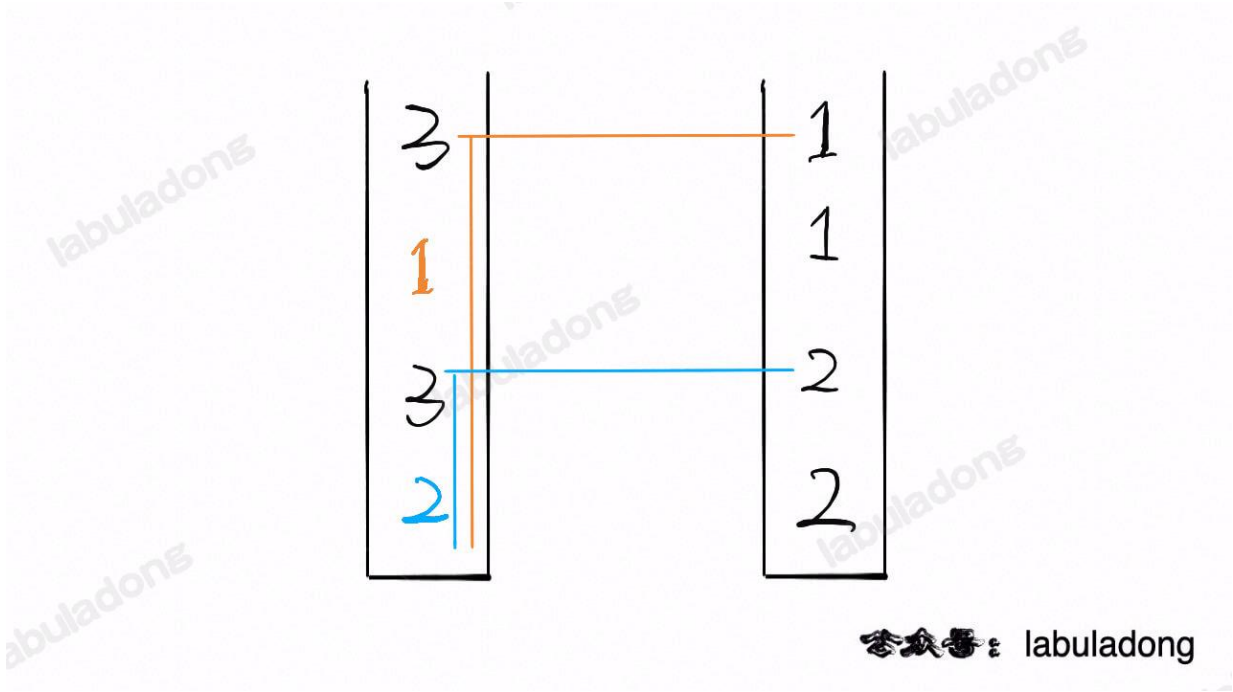
根据我们之前亲自动手实现的栈，我们知道栈是一种操作受限的数据结构，只能从栈顶插入或弹出元素，所以对于标准的栈来说，如果想实现本题的 `getMin` 方法，只能老老实实把所有元素弹出来然后找最小值。**想提高时间效率，那肯定要通过空间换时间的思路**。

不过在具体说解法之前，我想聊一下动态集合中维护最值的问题。这类问题看似简单，但实际上是个很棘手的问题。其实本题就是如下一个场景：

假设你有若干数字，你用一个 min 变量维护了其中的最小值，如果现在给这些数字中添加一个新数字，那么只要比较这个新数字和 min 的大小就可以得出最新的最小值。但如果现在从这些数字钟删除一个数字，你还能用 min 变量得到最小值吗？答案是不能，因为如果这个被删除的数字恰好是最小值，那么新的 min 变量应该更新为第二小的元素对吧，但是我没有记录第二小的元素是多少，所以只能把所有数字重新遍历一遍。

明确了难点再回到本题，就可以对症下药了。删除栈顶元素的时候，不确定新的最小值是多少，但楼下那哥们知道啊，他当时入栈时的最小值，就是现在的最小值呗。

所以这道题的关键就是，**每个元素入栈时，还要记下来当前栈中的最小值**。比方说，可以用一个额外的栈 minStk 来记录栈中每个元素入栈时的栈中的最小元素是多少，这样每次删除元素时就能快速得到剩余栈中的最小元素了。



当然，我们还可以做一些优化，减少 minStk 中存储的元素个数，我把原始解法和优化解法都写出来了，供参考。

> PS：这道题并不难，但我还是很细致地分析了，希望你深刻理解其中的难点。下一步可以做一下 239. 滑动窗口最大值，请仔细观察和思考，队列结构是如何解决这个难点的。

解法代码

```
// 原始思路
class MinStack1 {
    // 记录栈中的所有元素
    Stack<Integer> stk = new Stack<>();
    // 阶段性记录栈中的最小元素
    Stack<Integer> minStk = new Stack<>();

    public void push(int val) {
        stk.push(val);
        // 维护 minStk 栈顶为全栈最小元素
        if (minStk.isEmpty() || val <= minStk.peek()) {
            // 新插入的这个元素就是全栈最小的
```

```
            minStk.push(val);
        } else {
            // 插入的这个元素比较大
            minStk.push(minStk.peek());
        }
    }

    public void pop() {
        stk.pop();
        minStk.pop();
    }

    public int top() {
        return stk.peek();
    }

    public int getMin() {
        // minStk 栈顶为全栈最小元素
        return minStk.peek();
    }
}
// 优化版
class MinStack {
    // 记录栈中的所有元素
    Stack<Integer> stk = new Stack<>();
    // 阶段性记录栈中的最小元素
    Stack<Integer> minStk = new Stack<>();

    public void push(int val) {
        stk.push(val);
        // 维护 minStk 栈顶为全栈最小元素
        if (minStk.isEmpty() || val <= minStk.peek()) {
            // 新插入的这个元素就是全栈最小的
            minStk.push(val);
        }
    }

    public void pop() {
        // 注意 Java 的语言特性，比较 Integer 相等要用 equals 方法
        if (stk.peek().equals(minStk.peek())) {
            // 弹出的元素是全栈最小的
            minStk.pop();
        }
        stk.pop();
    }

    public int top() {
        return stk.peek();
    }

    public int getMin() {
        // minStk 栈顶为全栈最小元素
        return minStk.peek();
```

```
        }
    }
```

- **类似题目**：
  - 239. 滑动窗口最大值 🔴
  - 剑指 Offer 30. 包含min函数的栈 🟢
  - 剑指 Offer 59 - I. 滑动窗口的最大值 🔴

239. 滑动窗口最大值

LeetCode	力扣	难度
239. Sliding Window Maximum	239. 滑动窗口最大值	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，滑动窗口，队列

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧，返回滑动窗口中的最大值。

滑动窗口每次只向右移动一位，你只可以看到在滑动窗口内的 k 个数字。

示例 1：

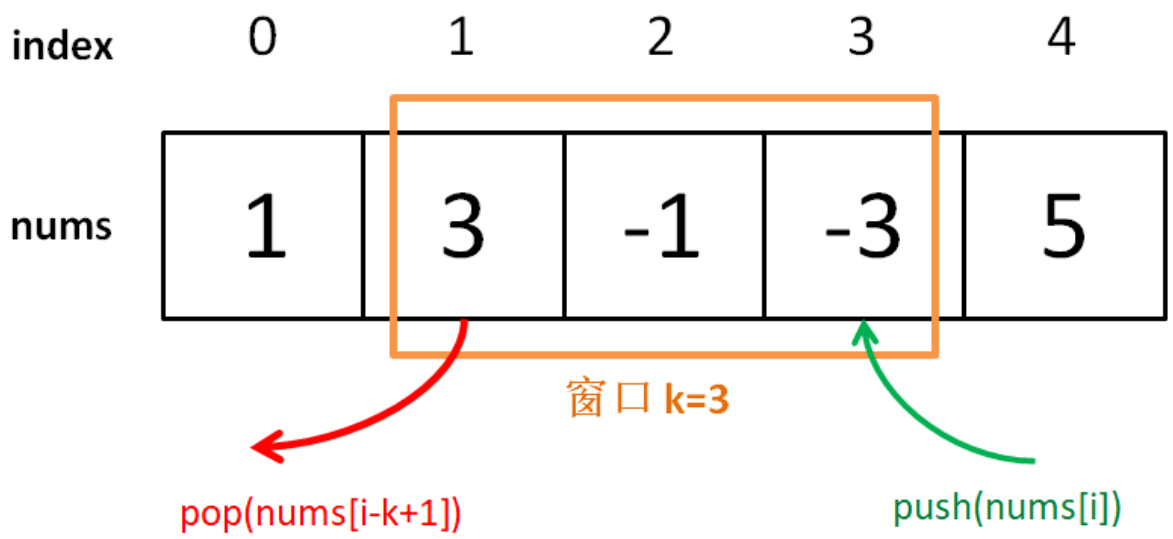
```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

基本思路

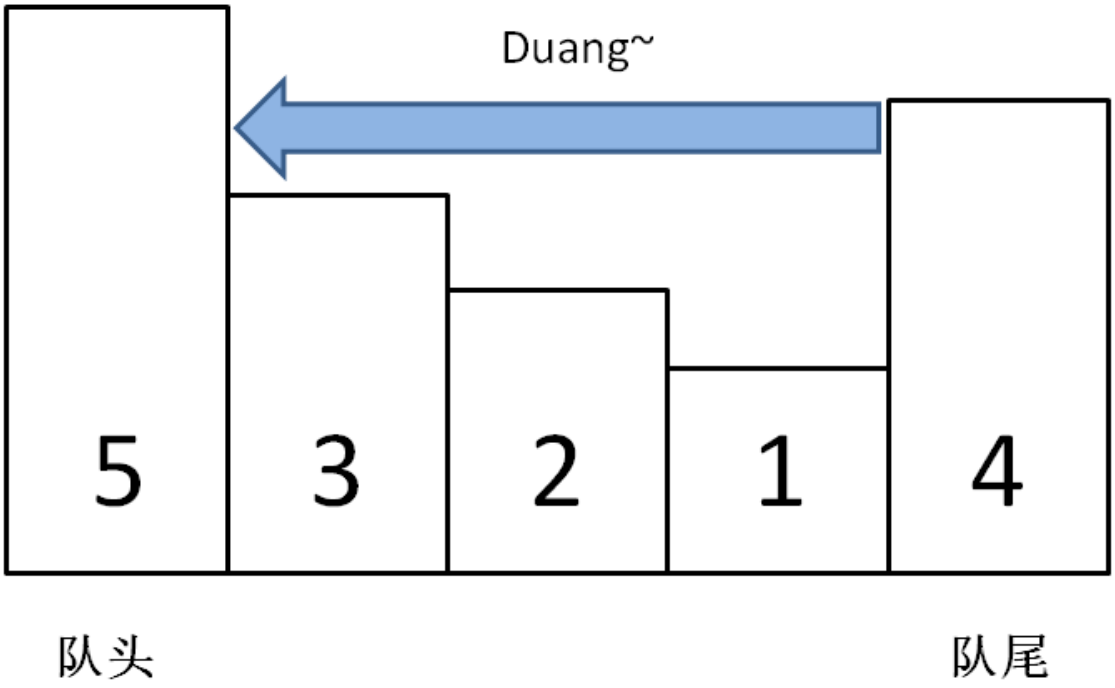
PS：这道题在《算法小抄》 的第 271 页。

> 我在 155. 最小栈 的思路中详细分析了动态集合中维护最值的「千古难题」，如果你没有做，可以先去做一下。我想请你结合这两道题仔细思考：队列和栈分别是如何解决最值维护问题的？

使用一个队列充当不断滑动的窗口，每次滑动记录其中的最大值：



如何在 `O(1)` 时间计算最大值，只需要一个特殊的数据结构「单调队列」，`push` 方法依然在队尾添加元素，但是要把前面比自己小的元素都删掉，直到遇到更大的元素才停止删除。



使用单调队列数据结构就能完成本题。

- **详细题解：单调队列结构解决滑动窗口问题**

解法代码

```
class Solution {
    /* 单调队列的实现 */
    class MonotonicQueue {
```

```
        LinkedList<Integer> q = new LinkedList<>();
        public void push(int n) {
            // 将小于 n 的元素全部删除
            while (!q.isEmpty() && q.getLast() < n) {
                q.pollLast();
            }
            // 然后将 n 加入尾部
            q.addLast(n);
        }

        public int max() {
            return q.getFirst();
        }

        public void pop(int n) {
            if (n == q.getFirst()) {
                q.pollFirst();
            }
        }
    }

    /* 解题函数的实现 */
    public int[] maxSlidingWindow(int[] nums, int k) {
        MonotonicQueue window = new MonotonicQueue();
        List<Integer> res = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            if (i < k - 1) {
                //先填满窗口的前 k - 1
                window.push(nums[i]);
            } else {
                // 窗口向前滑动，加入新数字
                window.push(nums[i]);
                // 记录当前窗口的最大值
                res.add(window.max());
                // 移出旧数字
                window.pop(nums[i - k + 1]);
            }
        }
        // 需要转成 int[] 数组再返回
        int[] arr = new int[res.size()];
        for (int i = 0; i < res.size(); i++) {
            arr[i] = res.get(i);
        }
        return arr;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：

- 155. 最小栈 🟠
- 剑指 Offer 30. 包含min函数的栈 🟢
- 剑指 Offer 59 - I. 滑动窗口的最大值 🔴
- 剑指 Offer 59 - II. 队列的最大值 🟠

剑指 Offer 30. 包含min函数的栈

这道题和 155. 最小栈 相同。

剑指 Offer 59 - I. 滑动窗口的最大值

这道题和 239. 滑动窗口最大值 相同。

15. 三数之和

LeetCode	力扣	难度
15. 3Sum	15. 三数之和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数组双指针，递归**

给你一个包含 `n` 个整数的数组 `nums`，判断 `nums` 中是否存在三个元素 `a，b，c`，使得 `a + b + c = 0`？

请你找出所有和为 `0` 且不重复的三元组。

示例 1：

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
```

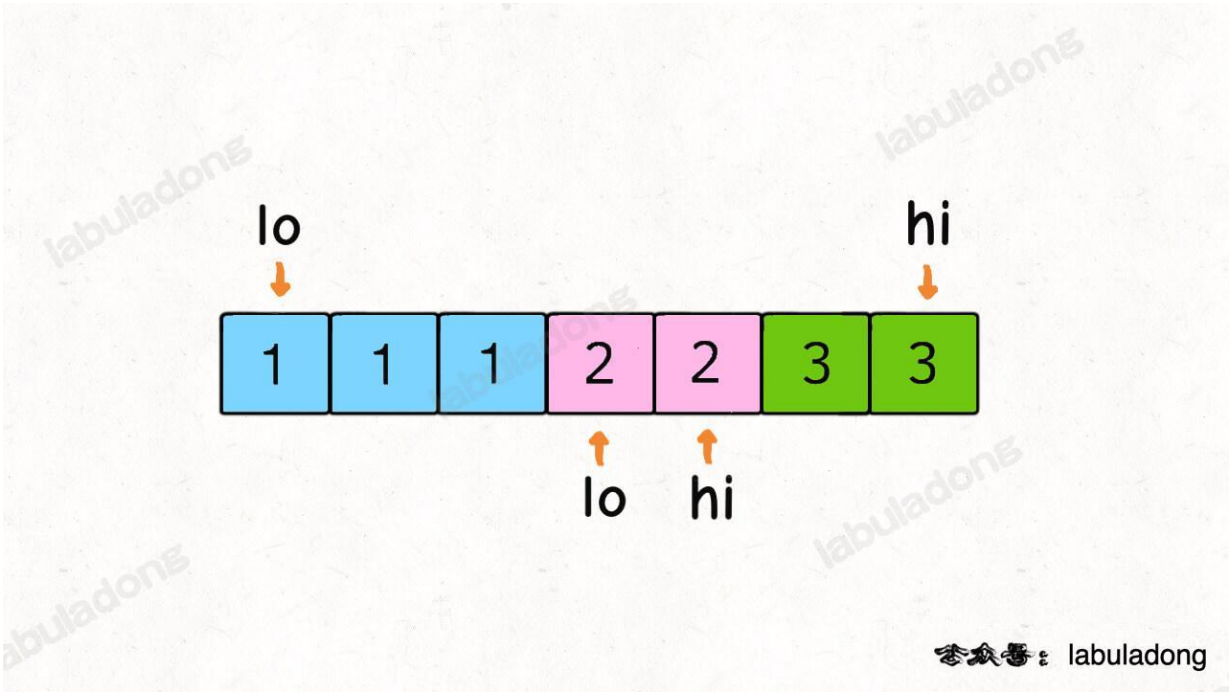
基本思路

PS：这道题在《算法小抄》 的第 319 页。

nSum 系列问题的核心思路就是**排序 + 双指针**。

先给数组从小到大排序，然后双指针 `lo` 和 `hi` 分别在数组开头和结尾，这样就可以控制 `nums[lo]` 和 `nums[hi]` 这两数之和的大小：

如果你想让它俩的和大一些，就让 `lo++`，如果你想让它俩的和小一些，就让 `hi--`。



基于两数之和可以得到一个万能函数 nSumTarget，扩展出 n 数之和的解法，具体分析见详细题解。

- **详细题解：一个方法团灭 nSum 问题**

解法代码

```
class Solution {
    public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        // n 为 3，从 nums[0] 开始计算和为 0 的三元组
        return nSumTarget(nums, 3, 0, 0);
    }

    /* 注意：调用这个函数之前一定要先给 nums 排序 */
    // n 填写想求的是几数之和，start 从哪个索引开始计算（一般填 0），target 填想凑出
的目标和
    vector<vector<int>> nSumTarget(
            vector<int>& nums, int n, int start, int target) {

        int sz = nums.size();
        vector<vector<int>> res;
        // 至少是 2Sum，且数组大小不应该小于 n
        if (n < 2 || sz < n) return res;
        // 2Sum 是 base case
        if (n == 2) {
            // 双指针那一套操作
            int lo = start, hi = sz - 1;
            while (lo < hi) {
                int sum = nums[lo] + nums[hi];
                int left = nums[lo], right = nums[hi];
                if (sum < target) {
                    while (lo < hi && nums[lo] == left) lo++;
                } else if (sum > target) {
                    while (lo < hi && nums[hi] == right) hi--;
                } else {
                    res.push_back({left, right});
                    while (lo < hi && nums[lo] == left) lo++;
                    while (lo < hi && nums[hi] == right) hi--;
                }
            }
        } else {
            // n > 2 时，递归计算 (n-1)Sum 的结果
            for (int i = start; i < sz; i++) {
                vector<vector<int>>
                        sub = nSumTarget(nums, n - 1, i + 1, target -
nums[i]);
                for (vector<int>& arr : sub) {
                    // (n-1)Sum 加上 nums[i] 就是 nSum
                    arr.push_back(nums[i]);
                    res.push_back(arr);
                }
```

```
                while (i < sz - 1 && nums[i] == nums[i + 1]) i++;
            }
        }
        return res;
    }
};
```

▶ 🦇 代码可视化动画（需安装 Chrome 插件）🦇

- **类似题目**：
  - 1. 两数之和 🟢
  - 167. 两数之和 II - 输入有序数组 🟠
  - 18. 四数之和 🟠
  - 剑指 Offer II 007. 数组中和为 0 的三个数 🟠

18. 四数之和

LeetCode	力扣	难度
18. 4Sum	18. 四数之和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数组双指针，递归**

给你一个由 `n` 个整数组成的数组 `nums`，和一个目标值 `target`。请你找出并返回满足下述全部条件且不重复的四元组 `[ nums[a], nums[b], nums[c], nums[d] ]`：

1、`0 <= a, b, c, d < n`。

2、`a`、`b`、`c` 和 `d` **互不相同**。

3、`nums[a] + nums[b] + nums[c] + nums[d] == target`。

你可以按**任意顺序**返回答案。

示例 1：

```
输入：nums = [1,0,-1,0,-2,2], target = 0
输出：[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

基本思路

PS：这道题在《算法小抄》 的第 319 页。

nSum 系列问题的核心思路就是**排序 + 双指针**。

先给数组从小到大排序，然后双指针 `lo` 和 `hi` 分别在数组开头和结尾，这样就可以控制 `nums[lo]` 和 `nums[hi]` 这两数之和的大小：

如果你想让它俩的和大一些，就让 `lo++`，如果你想让它俩的和小一些，就让 `hi--`。

基于两数之和可以得到一个万能函数 `nSumTarget`，扩展出 n 数之和的解法，具体分析见详细题解。

- **详细题解：一个方法团灭 nSum 问题**

解法代码

```
class Solution {
    public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        sort(nums.begin(), nums.end());
        // n 为 4，从 nums[0] 开始计算和为 target 的四元组
```

```
        return nSumTarget(nums, 4, 0, target);
    }

    /* 注意：调用这个函数之前一定要先给 nums 排序 */
    // n 填写想求的是几数之和，start 从哪个索引开始计算（一般填 0），target 填想凑出
的目标和
    vector<vector<int>> nSumTarget(
            vector<int>& nums, int n, int start, int target) {

        int sz = nums.size();
        vector<vector<int>> res;
        // 至少是 2Sum，且数组大小不应该小于 n
        if (n < 2 || sz < n) return res;
        // 2Sum 是 base case
        if (n == 2) {
            // 双指针那一套操作
            int lo = start, hi = sz - 1;
            while (lo < hi) {
                int sum = nums[lo] + nums[hi];
                int left = nums[lo], right = nums[hi];
                if (sum < target) {
                    while (lo < hi && nums[lo] == left) lo++;
                } else if (sum > target) {
                    while (lo < hi && nums[hi] == right) hi--;
                } else {
                    res.push_back({left, right});
                    while (lo < hi && nums[lo] == left) lo++;
                    while (lo < hi && nums[hi] == right) hi--;
                }
            }
        } else {
            // n > 2 时，递归计算 (n-1)Sum 的结果
            for (int i = start; i < sz; i++) {
                vector<vector<int>>
                        sub = nSumTarget(nums, n - 1, i + 1, target -
nums[i]);
                for (vector<int>& arr : sub) {
                    // (n-1)Sum 加上 nums[i] 就是 nSum
                    arr.push_back(nums[i]);
                    res.push_back(arr);
                }
                while (i < sz - 1 && nums[i] == nums[i + 1]) i++;
            }
        }
        return res;
    }
};
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- 类似题目：

- 1. 两数之和 🟢
- 15. 三数之和 🟠
- 167. 两数之和 II - 输入有序数组 🟠
- 剑指 Offer II 007. 数组中和为 0 的三个数 🟠

剑指 Offer II 007. 数组中和为 0 的三个数

这道题和 15. 三数之和 相同。

剑指 Offer 58 - I. 翻转单词顺序

这道题和 151. 颠倒字符串中的单词 相同。

1260. 二维网格迁移

LeetCode	力扣	难度
1260. Shift 2D Grid	1260. 二维网格迁移	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：dsvip，数组双指针**

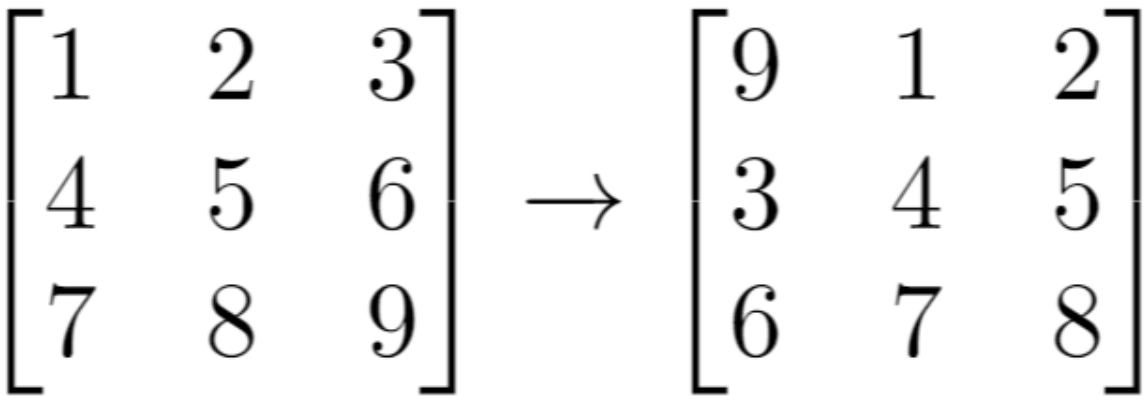
给你一个 `m` 行 `n` 列的二维网格 `grid` 和一个整数 `k`。你需要将 `grid` 迁移 `k` 次。

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][n - 1]` 的元素将会移动到 `grid[i + 1][0]`。
- 位于 `grid[m - 1][n - 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]]
```

基本思路

这道题有些像 151. 颠倒字符串中的单词，也要用到 二维数组的花式遍历 中讲到的多次翻转的技巧。

151 题让你把句子中的所有单词位置翻转，解法思路是先翻转整个句子，然后逐一翻转每个单词。

这道题是同样的思路：你可以写一个 `get` 方法和 `set` 方法把二维数组抽象成一维数组，然后题目就变成了让你将一个一维的数组平移 `k` 位，相当于把前 `mn - k` 个元素的位置和后 `k` 个元素的位置对调，也可以先把整个数组翻转，再分别翻转前 `mn - k` 个元素和后 `k` 个元素，得到的结果就是题目想要的。

解法代码

```
class Solution {
    public List<List<Integer>> shiftGrid(int[][] grid, int k) {
        // 把二维 grid 抽象成一维数组
        int m = grid.length, n = grid[0].length;
        int mn = m * n;
        k = k % mn;
        // 先把最后 k 个数翻转
        reverse(grid, mn - k, mn - 1);
        // 然后把前 mn - k 个数翻转
        reverse(grid, 0, mn - k - 1);
        // 最后把整体翻转
        reverse(grid, 0, mn - 1);

        // 转化成 Java List
        List<List<Integer>> res = new ArrayList<>();
        for (int[] row : grid) {
            List<Integer> rowList = new ArrayList<>();
            for (int e : row) {
                rowList.add(e);
            }
            res.add(rowList);
        }
        return res;
    }

    // 通过一维数组的索引访问二维数组的元素
    int get(int[][] grid, int index) {
        int n = grid[0].length;
        int i = index / n, j = index % n;
        return grid[i][j];
    }

    // 通过一维数组的索引修改二维数组的元素
    void set(int[][] grid, int index, int val) {
        int n = grid[0].length;
        int i = index / n, j = index % n;
        grid[i][j] = val;
    }

    // 翻转一维数组的索引区间元素
    void reverse(int[][] grid, int i, int j) {
        while (i < j) {
            int temp = get(grid, i);
            set(grid, i, get(grid, j));
            set(grid, j, temp);
            i++;
            j--;
        }
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

151. 颠倒字符串中的单词

LeetCode	力扣	难度
151. Reverse Words in a String	151. 翻转字符串里的单词	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：字符串，数组双指针**

给你一个字符串 `s`，颠倒字符串中 **单词** 的顺序。**单词** 是由非空格字符组成的字符串。`s` 中使用至少一个空格将字符串中的**单词** 分隔开。返回 **单词** 顺序颠倒且 **单词** 之间用单个空格连接的结果字符串。

注意：输入字符串 `s` 中可能会存在前导空格、尾随空格或者单词间的多个空格。返回的结果字符串中，单词间应当仅用单个空格分隔，且不包含任何额外的空格。

示例 1：

```
输入：s = "the sky is blue"
输出："blue is sky the"
```

基本思路

常规方法是用类似 `split` 再 `reverse` 最后 `join` 的方法得到结果，但更巧妙的方法是我在 二维数组的花式遍历 中讲到的：**先把整个字符串进行翻转，再把每个单词中的字母翻转**。

比如说，给你输入这样一个字符串：

```
s = "hello world labuladong"
```

那么我们先将整个字符串 `s` 反转：

```
s = "gnodalubal dlrow olleh"
```

然后将每个单词分别反转：

```
s = "labuladong world hello"
```

这样，就实现了原地反转所有单词顺序的目的。

整体的思路应该不难，就是细节比较恶心，直接看我写的代码吧。

- **详细题解：二维数组的花式遍历技巧**

解法代码

```
class Solution {
    public String reverseWords(String s) {
        StringBuilder sb = new StringBuilder();
        // 先清洗一下数据，把多于的空格都删掉
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c != ' ') {
                // 单词中的字母/数字
                sb.append(c);
            } else if (!sb.isEmpty() && sb.charAt(sb.length() - 1) != ' ')
{
                // 单词之间保留一个空格
                sb.append(' ');
            }
        }
        if (sb.isEmpty()) {
            return "";
        }
        // 末尾如果有空格，清除之
        if (sb.charAt(sb.length() - 1) == ' ') {
            sb.deleteCharAt(sb.length() - 1);
        }

        // 清洗之后的字符串
        char[] chars = sb.toString().toCharArray();
        int n = chars.length;
        // 进行单词的翻转，先整体翻转
        reverse(chars, 0, n - 1);
        // 再把每个单词翻转
        for (int i = 0; i < n; ) {
            for (int j = i; j < n; j++) {
                if (j + 1 == n || chars[j + 1] == ' ') {
                    // chars[i..j] 是一个单词，翻转之
                    reverse(chars, i, j);
                    // 把 i 置为下一个单词的首字母
                    i = j + 2;
                    break;
                }
            }
        }
        // 最后得到题目想要的结果
        return new String(chars);
    }

    // 翻转 arr[i..j]
    void reverse(char[] arr, int i, int j) {
        while (i < j) {
            char temp = arr[i];
```

```
                arr[i] = arr[j];
                arr[j] = temp;
                i++;
                j--;
            }
        }
    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 1260. 二维网格迁移 🟢
  - 48. 旋转图像 🟠
  - 54. 螺旋矩阵 🟠
  - 59. 螺旋矩阵 II 🟠
  - 61. 旋转链表 🟠
  - 剑指 Offer 29. 顺时针打印矩阵 🟢
  - 剑指 Offer 58 - I. 翻转单词顺序 🟢

剑指 Offer 21. 调整数组顺序使奇数位于偶数前面

LeetCode	力扣	难度
剑指Offer21. 调整数组顺序使奇数位于偶数前面 LCOF	剑指Offer21. 调整数组顺序使奇数位于偶数前面	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数组双指针**

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数在数组的前半部分，所有偶数在数组的后半部分。

示例：

```
输入：nums = [1,2,3,4]
输出：[1,3,2,4]
注：[3,1,2,4] 也是正确的答案之一。
```

基本思路

这题是前文 数组双指针技巧汇总 讲到的快慢指针技巧，可以复用 27. 移除元素（简单） 解法中的函数，只要稍微修改一下逻辑即可。

解法代码

```
class Solution {
    public int[] exchange(int[] nums) {
        // 维护 nums[0..slow) 都是奇数
        int fast = 0, slow = 0;
        while (fast < nums.length) {
            if (nums[fast] % 2 == 1) {
                // fast 遇到奇数，把 nums[fast] 换到 nums[slow]
                int temp = nums[slow];
                nums[slow] = nums[fast];
                nums[fast] = temp;
                slow++;
            }
            fast++;
        }
        return nums;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

---

11. 盛最多水的容器

LeetCode	力扣	难度
11. Container With Most Water	11. 盛最多水的容器	🟠

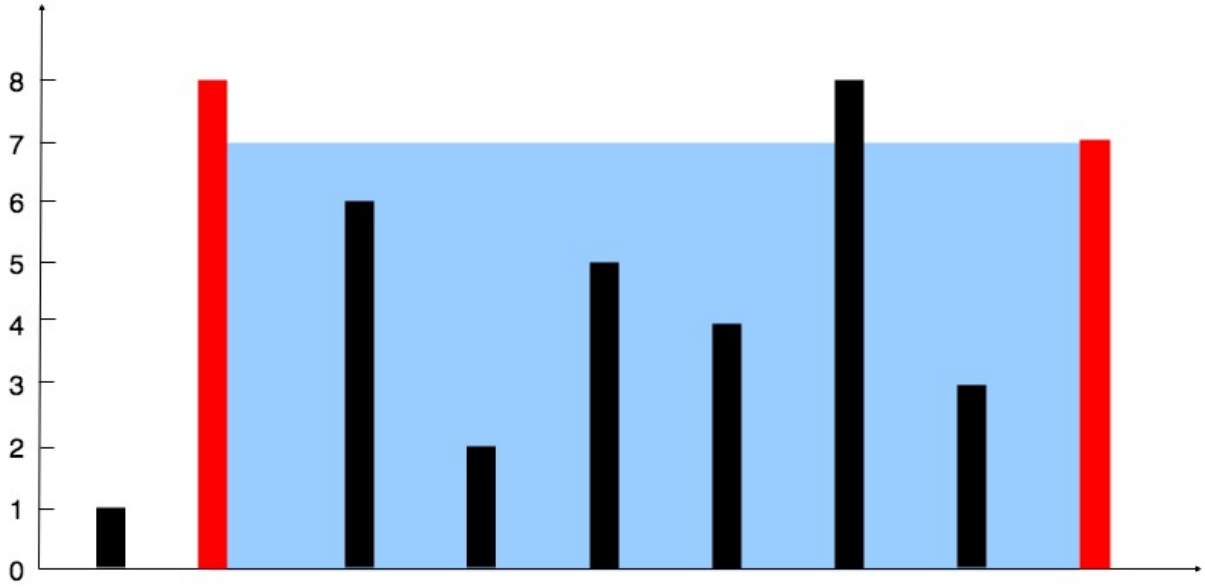
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组双指针

给你 `n` 个非负整数 `a1, a2, ..., an`，其中每个数代表坐标中的一个点 `(i, ai)`。在坐标内画 `n` 条垂直线，垂直线 `i` 的两个端点分别为 `(i, ai)` 和 `(i, 0)`。

找出其中的两条线，使得它们与 `x` 轴共同构成的容器可以容纳最多的水。

示例 1：



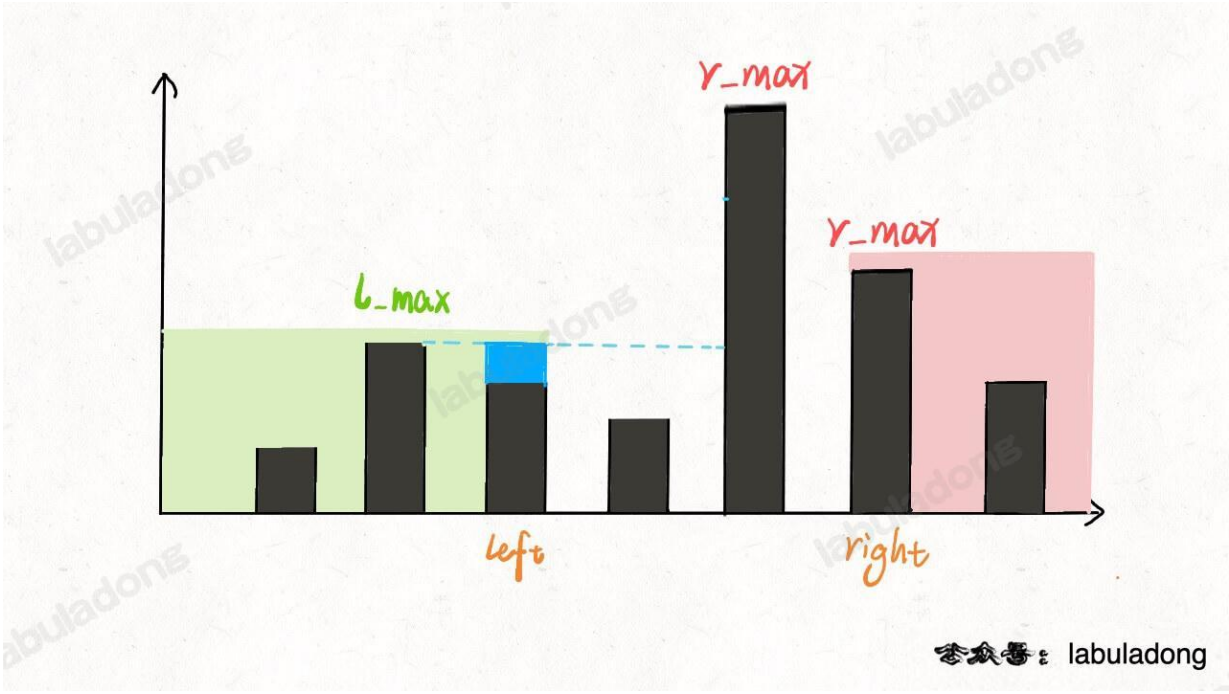

```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。
```

基本思路

这题前文 接雨水问题详解 讲过的 42. 接雨水 几乎一模一样。

区别在于：接雨水问题给出的类似一幅直方图，每个横坐标都有宽度，而本题给出的每个坐标是一条竖线，没有宽度。

接雨水问题更难一些，每个坐标对应的矩形通过左右的最大高度的最小值推算自己能装多少水：



本题可完全套用接雨水问题的思路，相对还更简单：

用 `left` 和 `right` 两个指针从两端向中心收缩，一边收缩一边计算 `[left, right]` 之间的矩形面积，取最大的面积值即是答案。

不过肯定有读者会问，下面这段 if 语句为什么要移动较低的一边：

```
// 双指针技巧，移动较低的一边
if (height[left] < height[right]) {
    left++;
} else {
    right--;
}
```

其实也好理解，因为矩形的高度是由 `min(height[left], height[right])` 即较低的一边决定的：

你如果移动较低的那一边，那条边可能会变高，使得矩形的高度变大，进而就「有可能」使得矩形的面积变大；相反，如果你去移动较高的那一边，矩形的高度是无论如何都不会变大的，所以不可能使矩形的面积变得更大。

- **详细题解：如何高效解决接雨水问题**

解法代码

```
class Solution {
    public int maxArea(int[] height) {
        int left = 0, right = height.length - 1;
        int res = 0;
        while (left < right) {
            // [left, right] 之间的矩形面积
            int cur_area = Math.min(height[left], height[right]) * (right
```

```
- left);
            res = Math.max(res, cur_area);
            // 双指针技巧，移动较低的一边
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }
        return res;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 42. 接雨水 🔴

42. 接雨水

LeetCode	力扣	难度
42. Trapping Rain Water	42. 接雨水	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组双指针

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：

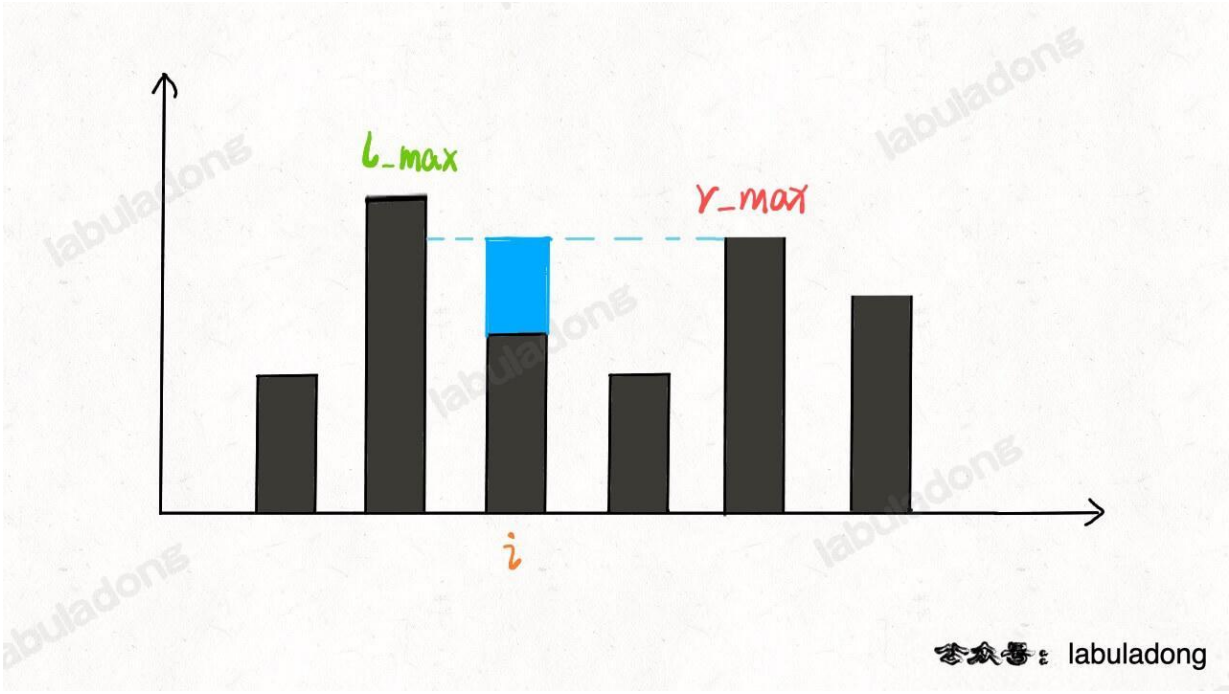
```
输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]
输出：6
解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。
```

基本思路

PS：这道题在《算法小抄》 的第 364 页。

对于任意一个位置 i，能够装的水为：

```
water[i] = min(
            # 左边最高的柱子
            max(height[0..i]),
            # 右边最高的柱子
            max(height[i..end])
        ) - height[i]
```



关键在于，如何能够快速计算出某一个位置左侧所有柱子的最大高度和右侧所有柱子的最大高度。

这道题的解法比较多样，可以预计算数组，可以用 双指技巧，可以用单调栈技巧，这里就说一个最简单的解法，用预计算的方式求解，优化暴力解法的时间复杂度，更多解法请看详细题解。

- **详细题解：如何高效解决接雨水问题**

解法代码

```
class Solution {
    public int trap(int[] height) {
        if (height.length == 0) {
            return 0;
        }
        int n = height.length;
        int res = 0;
        // 数组充当备忘录
        int[] l_max = new int[n];
        int[] r_max = new int[n];
        // 初始化 base case
        l_max[0] = height[0];
        r_max[n - 1] = height[n - 1];
        // 从左向右计算 l_max
        for (int i = 1; i < n; i++)
            l_max[i] = Math.max(height[i], l_max[i - 1]);
        // 从右向左计算 r_max
        for (int i = n - 2; i >= 0; i--)
            r_max[i] = Math.max(height[i], r_max[i + 1]);
        // 计算答案
        for (int i = 1; i < n - 1; i++)
            res += Math.min(l_max[i], r_max[i]) - height[i];
        return res;
```

```
        }
    }
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 11. 盛最多水的容器 🟠

26. 删除有序数组中的重复项

LeetCode	力扣	难度
26. Remove Duplicates from Sorted Array	26. 删除有序数组中的重复项	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组，数组双指针

给你一个有序数组 `nums`，请你**原地**删除重复出现的元素，使每个元素**只出现一次**，返回删除后数组的新长度。

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。

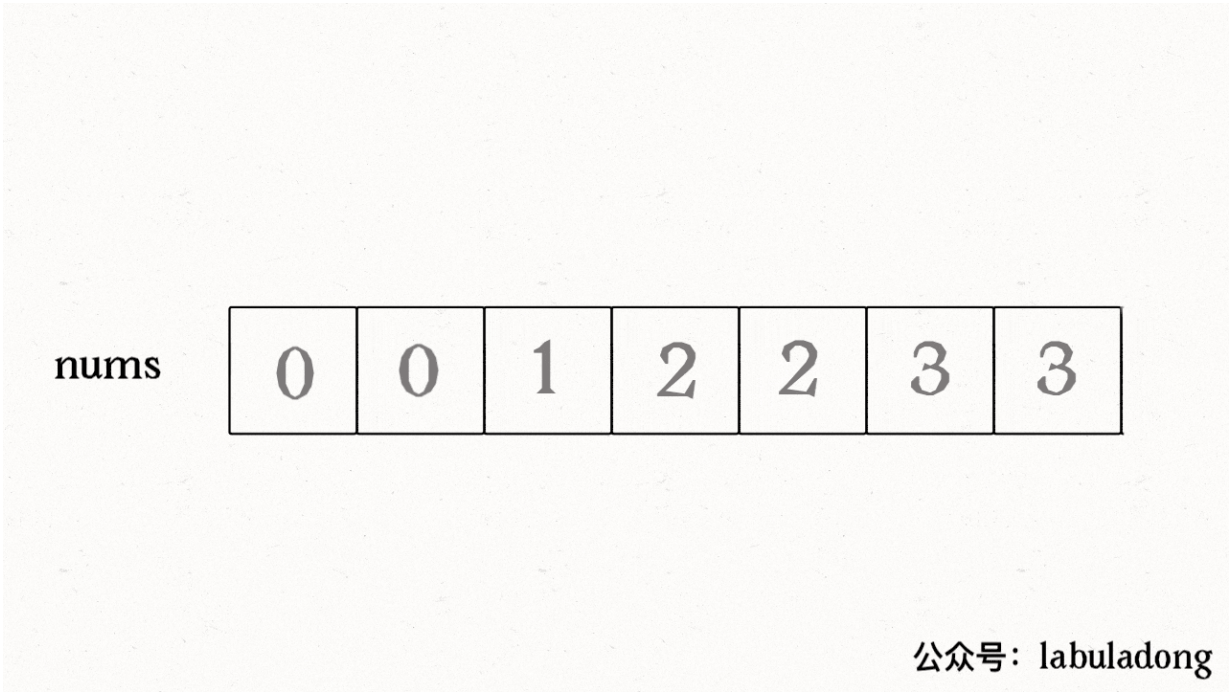
基本思路

> 本文有视频版：数组双指针技巧汇总

PS：这道题在《算法小抄》 的第 371 页。

有序序列去重的通用解法就是我们前文 双指针技巧 中的快慢指针技巧。

我们让慢指针 `slow` 走在后面，快指针 `fast` 走在前面探路，找到一个不重复的元素就告诉 `slow` 并让 `slow` 前进一步。这样当 `fast` 指针遍历完整个数组 `nums` 后，**`nums[0..slow]` 就是不重复元素**。



- 详细题解：双指针技巧秒杀七道数组题目

解法代码

```java
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        int slow = 0, fast = 0;
        while (fast < nums.length) {
            if (nums[fast] != nums[slow]) {
                slow++;
                // 维护 nums[0..slow] 无重复
                nums[slow] = nums[fast];
            }
            fast++;
        }
        // 数组长度为索引 + 1
        return slow + 1;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 167. 两数之和 II - 输入有序数组 🟠
  - 27. 移除元素 🟢
  - 283. 移动零 🟢
  - 344. 反转字符串 🟢
  - 5. 最长回文子串 🟠
  - 80. 删除有序数组中的重复项 II 🟠
  - 83. 删除排序链表中的重复元素 🟢
  - 剑指 Offer 57. 和为s的两个数字 🟢
  - 剑指 Offer II 006. 排序数组中两个数字之和 🟢

27. 移除元素

LeetCode	力扣	难度
27. Remove Element	27. 移除元素	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组，数组双指针

给你一个数组 `nums` 和一个值 `val`，你需要**原地**移除所有数值等于 `val` 的元素，并返回移除后数组的新长度。

你必须仅使用 `O(1)` 额外空间并**原地修改输入数组**。元素的顺序可以改变，你不需要考虑数组中超出新长度后面的元素。

基本思路

> 本文有视频版：数组双指针技巧汇总

类似 26. 删除有序数组中的重复项，需要使用 双指针技巧 中的快慢指针：

如果 `fast` 遇到需要去除的元素，则直接跳过，否则就告诉 `slow` 指针，并让 `slow` 前进一步。

- 详细题解：双指针技巧秒杀七道数组题目

解法代码

```
class Solution {
    public int removeElement(int[] nums, int val) {
        int fast = 0, slow = 0;
        while (fast < nums.length) {
            if (nums[fast] != val) {
                nums[slow] = nums[fast];
                slow++;
            }
            fast++;
        }
        return slow;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- 类似题目：
  - 167. 两数之和 II - 输入有序数组 🟠
  - 26. 删除有序数组中的重复项 🟢

- 283. 移动零 🟢
- 344. 反转字符串 🟢
- 5. 最长回文子串 🟠
- 83. 删除排序链表中的重复元素 🟢
- 剑指 Offer 21. 调整数组顺序使奇数位于偶数前面 🟢
- 剑指 Offer 57. 和为s的两个数字 🟢
- 剑指 Offer II 006. 排序数组中两个数字之和 🟢

283. 移动零

LeetCode	力扣	难度
283. Move Zeroes	283. 移动零	🟢

- **标签：数组，数组双指针**

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，必须在原数组上操作，同时保持非零元素的相对顺序。

示例：

```
输入：[0,1,0,3,12]
输出：[1,3,12,0,0]
```

基本思路

> 本文有视频版：数组双指针技巧汇总

可以直接复用 27. 移除元素 的解法，先移除所有 0，然后把最后的元素都置为 0，就相当于移动 0 的效果。

- **详细题解：双指针技巧秒杀七道数组题目**

解法代码

```
class Solution {
    public void moveZeroes(int[] nums) {
        // 去除 nums 中的所有 0
        // 返回去除 0 之后的数组长度
        int p = removeElement(nums, 0);
        // 将 p 之后的所有元素赋值为 0
        for (; p < nums.length; p++) {
            nums[p] = 0;
        }
    }

    // 双指针技巧，复用 [27. 移除元素] 的解法。
    int removeElement(int[] nums, int val) {
        int fast = 0, slow = 0;
        while (fast < nums.length) {
            if (nums[fast] != val) {
                nums[slow] = nums[fast];
                slow++;
            }
            fast++;
```

```
            }
            return slow;
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 167. 两数之和 II - 输入有序数组 🟠
  - 26. 删除有序数组中的重复项 🟢
  - 27. 移除元素 🟢
  - 344. 反转字符串 🟢
  - 5. 最长回文子串 🟠
  - 83. 删除排序链表中的重复元素 🟢
  - 剑指 Offer 57. 和为s的两个数字 🟢
  - 剑指 Offer II 006. 排序数组中两个数字之和 🟢

80. 删除有序数组中的重复项 II

LeetCode	力扣	难度
80. Remove Duplicates from Sorted Array II	80. 删除有序数组中的重复项 II	🟠

- 标签：数据结构，数组双指针

给你一个有序数组 nums，请你 **原地** 删除重复出现的元素，使每个元素 **最多出现两次**，返回删除后数组的新长度。

基本思路

这道题和前文 数组双指针技巧汇总 中讲的 26. 删除有序数组中的重复项 解法非常类似，只不过这道题说重复两次以上的元素才需要被去除。

本题解法依然使用快慢指针技巧，在之前的解法中添加一个 count 变量记录每个数字重复出现的次数，然后把 26 题的 if 判断额外复制粘贴一份就行了，直接看代码吧。

解法代码

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        // 快慢指针，维护 nums[0..slow] 为结果子数组
        int slow = 0, fast = 0;
        // 记录一个元素重复的次数
        int count = 0;
        while (fast < nums.length) {
            if (nums[fast] != nums[slow]) {
                // 此时，对于 nums[0..slow] 来说，nums[fast] 是一个新的元素，加进
来
                slow++;
                nums[slow] = nums[fast];
            } else if (slow < fast && count < 2) {
                // 此时，对于 nums[0..slow] 来说，nums[fast] 重复次数小于 2，也
加进来
                slow++;
                nums[slow] = nums[fast];
            }
            fast++;
            count++;
            if (fast < nums.length && nums[fast] != nums[fast - 1]) {
                // fast 遇到新的不同的元素时，重置 count
                count = 0;
```

```
                }
            }
            // 数组长度为索引 + 1
            return slow + 1;
        }
    }
```

▶ 代码可视化动画（需安装 Chrome 插件）

83. 删除排序链表中的重复元素

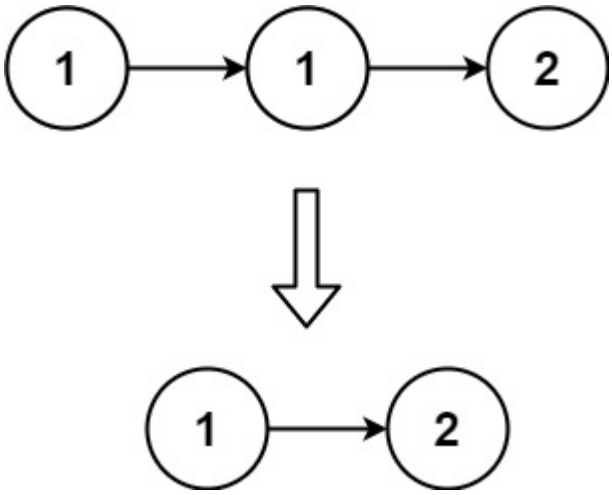
LeetCode	力扣	难度
83. Remove Duplicates from Sorted List	83. 删除排序链表中的重复元素	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：链表，链表双指针

存在一个按升序排列的链表，给你这个链表的头节点 head，请你删除所有重复的元素，使每个元素**只出现一次**，返回同样按升序排列的结果链表。

示例 1：



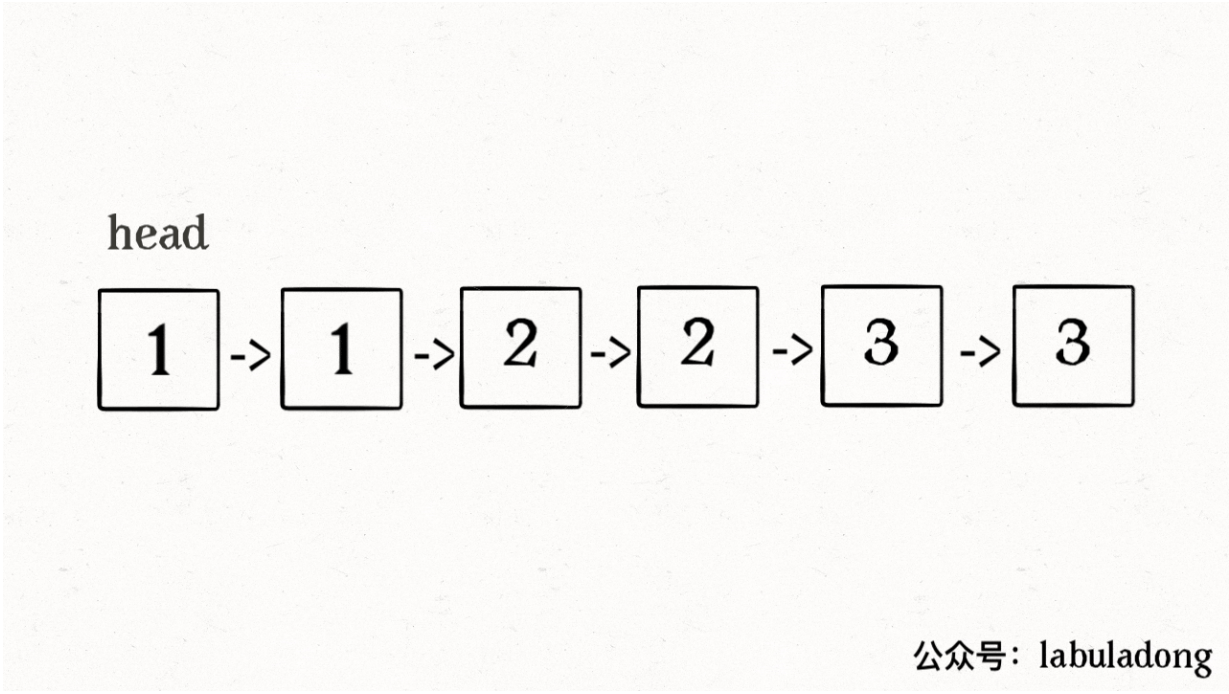

```
输入：head = [1,1,2]
输出：[1,2]
```

基本思路

> 本文有视频版：数组双指针技巧汇总

PS：这道题在《算法小抄》 的第 371 页。

思路和 26. 删除有序数组中的重复项 完全一样，唯一的区别是把数组赋值操作变成操作指针而已。



- 详细题解：双指针技巧秒杀七道数组题目

解法代码

```
class Solution {
    public deleteDuplicates(ListNode head) {
        if (head == null) return null;
        ListNode slow = head, fast = head;
        while (fast != null) {
            if (fast.val != slow.val) {
                // nums[slow] = nums[fast];
                slow.next = fast;
                // slow++;
                slow = slow.next;
            }
            // fast++
            fast = fast.next;
        }
        // 断开与后面重复元素的连接
        slow.next = null;
        return head;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- 类似题目：
  - 167. 两数之和 II - 输入有序数组 🟠
  - 26. 删除有序数组中的重复项 🟢
  - 27. 移除元素 🟢

- 283. 移动零 🟢
- 344. 反转字符串 🟢
- 5. 最长回文子串 🟠
- 82. 删除排序链表中的重复元素 II 🟠
- 剑指 Offer 57. 和为s的两个数字 🟢
- 剑指 Offer II 006. 排序数组中两个数字之和 🟢

剑指 Offer 57. 和为s的两个数字

LeetCode	力扣	难度
剑指Offer57. 和为s的两个数字 LCOF	剑指Offer57. 和为s的两个数字	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数组双指针**

输入一个递增排序的数组和一个数字 s，在数组中查找两个数，使得它们的和正好是 s。如果有多对数字的和等于 s，则输出任意一对即可。

示例 1：

```
输入：nums = [2,7,11,15], target = 9
输出：[2,7] 或者 [7,2]
```

基本思路

这道题和 1. 两数之和 基本一样，而且还说了 `nums` 是有序的，所以直接用 数组双指针技巧汇总 中讲的左右双指针解决即可。

- **详细题解：双指针技巧秒杀七道数组题目**

解法代码

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        // 左右双指针
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int sum = nums[left] + nums[right];
            if (sum < target) {
                // 让和大一点
                left++;
            } else if (sum > target) {
                // 让和小一点
                right--;
            } else {
                // 找到两个数
                return new int[]{nums[left], nums[right]};
            }
        }
        return null;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 167. 两数之和 II - 输入有序数组
  - 26. 删除有序数组中的重复项
  - 27. 移除元素
  - 283. 移动零
  - 344. 反转字符串
  - 5. 最长回文子串
  - 83. 删除排序链表中的重复元素
  - 剑指 Offer II 006. 排序数组中两个数字之和

88. 合并两个有序数组

LeetCode	力扣	难度
88. Merge Sorted Array	88. 合并两个有序数组	🟢

- 标签：数据结构，数组双指针

给你两个按 **非递减顺序** 排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。

请你 **合并** nums2 到 nums1 中，使合并后的数组同样按 **非递减顺序**排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0，应忽略。nums2 的长度为 n。

示例 1：

```
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]
解释：需要合并 [1,2,3] 和 [2,5,6]。
合并结果是 [1,2,2,3,5,6]，其中斜体加粗标注的为 nums1 中的元素。
```

基本思路

这道题很像前文 链表的双指针技巧汇总 中讲过的 21. 合并两个有序链表，这里让你合并两个有序数组。

对于单链表来说，我们直接用双指针从头开始合并即可，但对于数组来说会出问题。因为题目让我直接把结果存到 nums1 中，而 nums1 的开头有元素，如果我们无脑复制单链表的逻辑，会覆盖掉 nums1 的原始元素，导致错误。

但 nums1 后面是空的呀，所以这道题需要我们稍微变通一下：**将双指针初始化在数组的尾部，然后从后向前进行合并**，这样即便覆盖了 nums1 中的元素，这些元素也必然早就被用过了，不会影响答案的正确性。

解法代码

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        // 两个指针分别初始化在两个数组的最后一个元素（类似拉链两端的锯齿）
        int i = m - 1, j = n - 1;
        // 生成排序的结果（类似拉链的拉锁）
        int p = nums1.length - 1;
        // 从后向前生成结果数组，类似合并两个有序链表的逻辑
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
```

```
                nums1[p] = nums1[i];
                i--;
            } else {
                nums1[p] = nums2[j];
                j--;
            }
            p--;
        }
        // 可能其中一个数组的指针走到尽头了，而另一个还没走完
        // 因为我们本身就是在往 nums1 中放元素，所以只需考虑 nums2 是否剩元素即可
        while (j >= 0) {
            nums1[p] = nums2[j];
            j--;
            p--;
        }
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

- **类似题目**：
  - 977. 有序数组的平方 🟢

977. 有序数组的平方

LeetCode	力扣	难度
977. Squares of a Sorted Array	977. 有序数组的平方	🟢

- **标签：数组双指针**

给你一个按**非递减顺序**排序的整数数组 `nums`，返回**每个数字的平方**组成的新数组，要求也按**非递减顺序**排序。

示例 1：

```
输入：nums = [-4,-1,0,3,10]
输出：[0,1,9,16,100]
解释：平方后，数组变为 [16,1,0,9,100]
排序后，数组变为 [0,1,9,16,100]
```

基本思路

平方的特点是会把负数变成正数，所以一个负数和一个正数平方后的大小要根据绝对值来比较。

可以把元素 0 作为分界线，0 左侧的负数是一个有序数组 `nums1`，0 右侧的正数是另一个有序数组 `nums2`，那么这道题就和 88. 合并两个有序数组 差不多，在进一步就是前文 链表的双指针技巧汇总 讲过的 21. 合并两个有序链表 的变体。

所以，我们可以去寻找正负数的分界点，然后向左右扩展，执行合并有序数组的逻辑。不过还有个更好的办法，不用找正负分界点，而是直接将双指针分别初始化在 `nums` 的开头和结尾，相当于合并两个从大到小排序的数组，和 88 题类似。有了思路，直接看代码吧。

解法代码

```
class Solution {
    public int[] sortedSquares(int[] nums) {
        int n = nums.length;
        // 两个指针分别初始化在正负子数组绝对值最大的元素索引
        int i = 0, j = n - 1;
        // 得到的有序结果是降序的
        int p = n - 1;
        int[] res = new int[n];
        // 执行双指针合并有序数组的逻辑
        while (i <= j) {
            if (Math.abs(nums[i]) > Math.abs(nums[j])) {
                res[p] = nums[i] * nums[i];
                i++;
```

```
                } else {
                    res[p] = nums[j] * nums[j];
                    j--;
                }
                p--;
            }
            return res;
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 360. 有序转化数组 🟠

986. 区间列表的交集

LeetCode	力扣	难度
986. Interval List Intersections	986. 区间列表的交集	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：区间问题，数组双指针

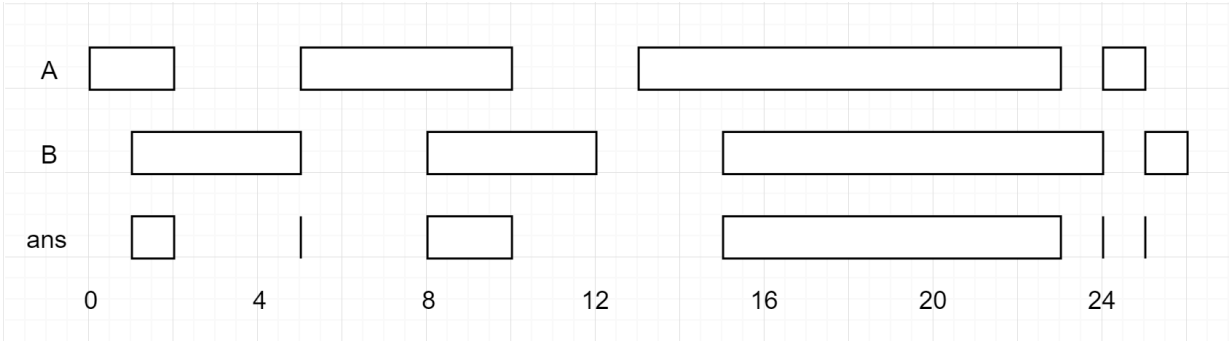
给定两个由一些 **闭区间** 组成的列表，`firstList` 和 `secondList`，其中 `firstList[i] = [starti, endi]` 而 `secondList[j] = [startj, endj]`。每个区间列表都是成对 **不相交** 的，并且 **已经排序**。

返回这 **两个区间列表的交集**。

形式上，**闭区间** `[a, b]`（其中 `a <= b`）表示实数 `x` 的集合，而 `a <= x <= b`。

两个闭区间的 **交集**是一组实数，要么为空集，要么为闭区间。例如，`[1, 3]` 和 `[2, 4]` 的交集为 `[2, 3]`。

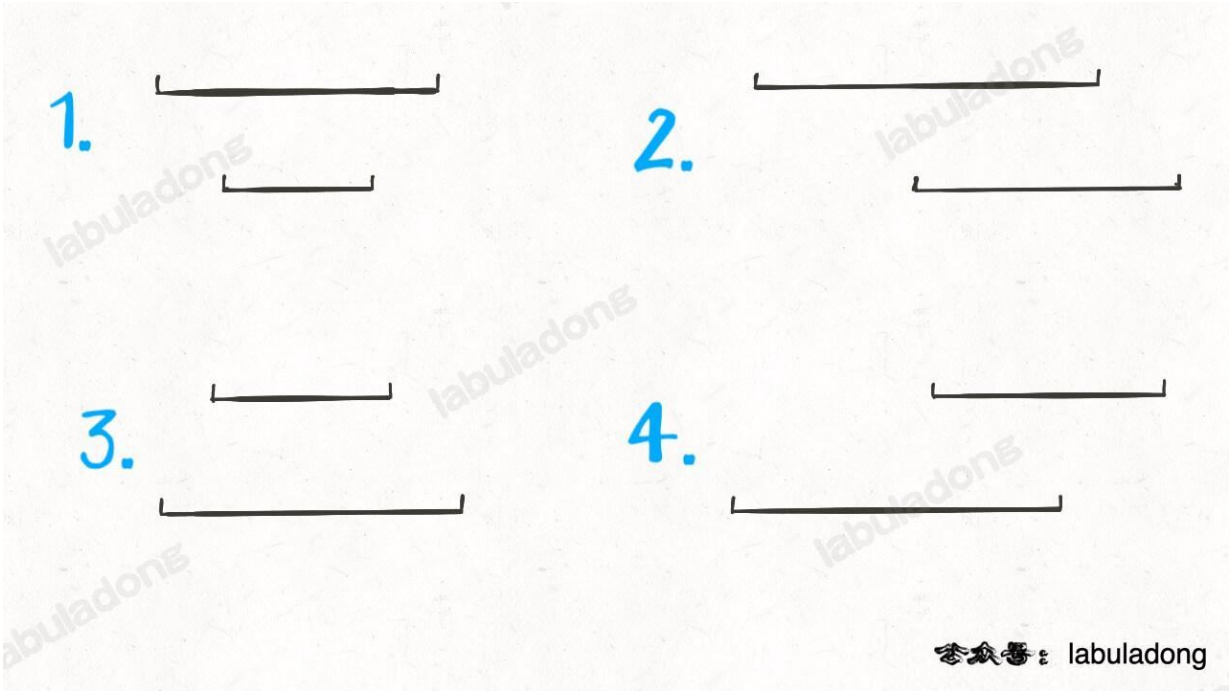
示例 1：



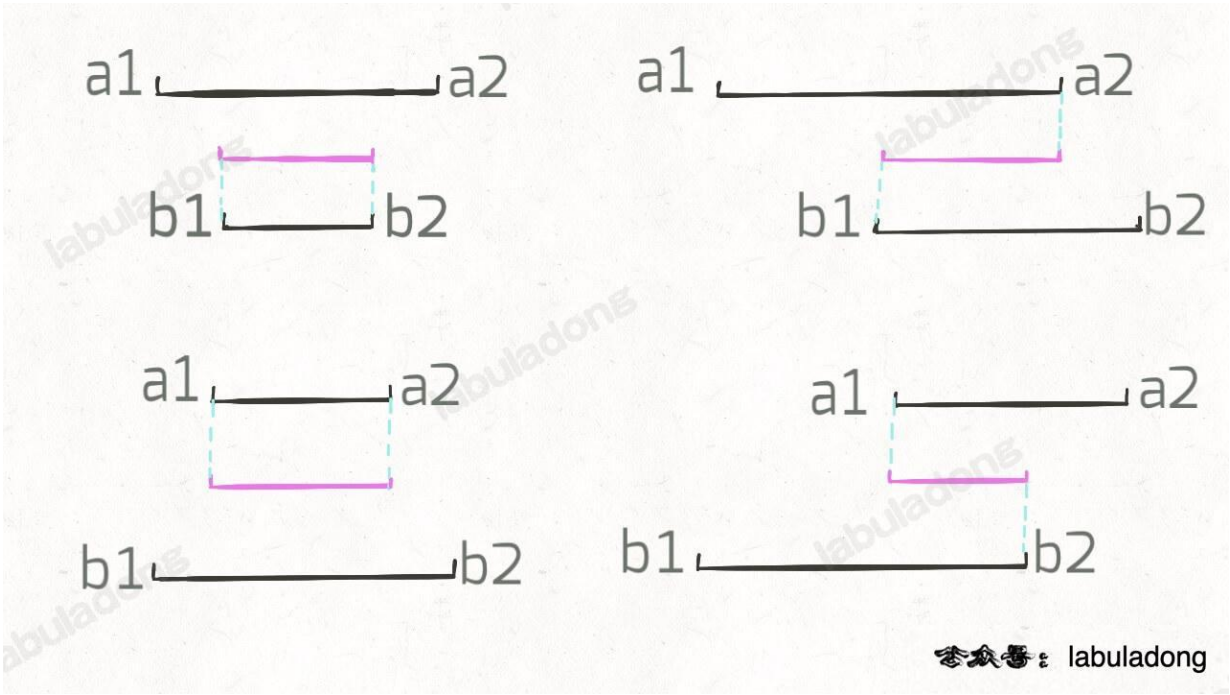

```
输入：firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]
输出：[[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

基本思路

我们用 `[a1, a2]` 和 `[b1, b2]` 表示在 `A` 和 `B` 中的两个区间，如果这两个区间有交集，需满足 `b2 >= a1 && a2 >= b1`，分下面四种情况：



根据上图可以发现规律，假设交集区间是 `[c1, c2]`，那么 `c1 = max(a1, b1), c2 = min(a2, b2)`：



这一点就是寻找交集的核心。

- **详细题解：一个方法解决三道区间问题**

解法代码

```
class Solution {
    public int[][] intervalIntersection(int[][] A, int[][] B) {
        List<int[]> res = new LinkedList<>();
        int i = 0, j = 0;
        while (i < A.length && j < B.length) {
            int a1 = A[i][0], a2 = A[i][1];
```

```
                int b1 = B[j][0], b2 = B[j][1];

                if (b2 >= a1 && a2 >= b1) {
                    res.add(new int[]{
                            Math.max(a1, b1), Math.min(a2, b2)
                    });
                }
                if (b2 < a2) {
                    j++;
                } else {
                    i++;
                }
            }
            return res.toArray(new int[0][0]);
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 1288. 删除被覆盖区间 🟠
  - 56. 合并区间 🟠
  - 剑指 Offer II 074. 合并区间 🟠

82. 删除排序链表中的重复元素 II

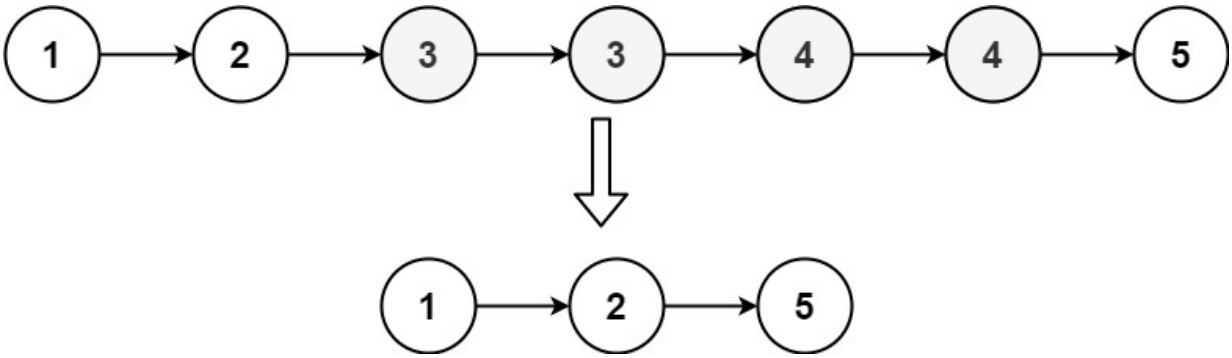
LeetCode	力扣	难度
82. Remove Duplicates from Sorted List II	82. 删除排序链表中的重复元素 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：dsvip，数据结构，链表双指针

给定一个已排序的链表的头 head，删除原始链表中所有重复数字的节点，只留下不同的数字，返回删除之后的排序链表。

示例 1：




```
输入：head = [1,2,3,3,4,4,5]
输出：[1,2,5]
```

基本思路

这道题是前文 链表的双指针技巧汇总 中讲的 83. 删除排序链表中的重复元素 的进阶版。如果只让你把多于的重复元素去掉，那么快慢指针可以搞定，但这道题要求你把存在重复的元素全都去掉，一个简单粗暴的解法就是借助像哈希表这样的数据结构记录哪些节点重复了，然后去掉它们。

不过这道题输入的链表是有序的，这意味着重复元素都靠在一起，其实不用额外的空间复杂度来辅助，用两个指针就可以达到去重的目的，只是细节有点多，直接结合代码的详细注释来看吧。

值得一提的是，这道题也可以用递归思维来做，虽然存在堆栈消耗空间复杂度，不过理解起来更容易，我也写出来供大家参考。

解法代码

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(-1);
        ListNode p = dummy, q = head;
        while (q != null) {
```

```
            if (q.next != null && q.val == q.next.val){
                // 发现重复节点，跳过这些重复节点
                while (q.next != null && q.val == q.next.val) {
                    q = q.next;
                }
                q = q.next;
                // 此时 q 跳过了这一段重复元素
                if (q == null) {
                    p.next = null;
                }
                // 不过下一段元素也可能重复，等下一轮 while 循环判断
            } else {
                // 不是重复节点，接到 dummy 后面
                p.next = q;
                p = p.next;
                q = q.next;
            }
        }
        return dummy.next;
    }
}

// 递归解法
class Solution2 {
    // 定义：输入一条单链表头结点，返回去重之后的单链表头结点
    public ListNode deleteDuplicates(ListNode head) {
        // base case
        if (head == null || head.next == null) {
            return head;
        }
        if (head.val != head.next.val) {
            // 如果头结点和身后节点的值不同，则对之后的链表去重即可
            head.next = deleteDuplicates(head.next);
            return head;
        }
        // 如果如果头结点和身后节点的值相同，则说明从 head 开始存在若干重复节点
        // 越过重复节点，找到 head 之后那个不重复的节点
        while (head.next != null && head.val == head.next.val) {
            head = head.next;
        }
        // 直接返回那个不重复节点开头的链表的去重结果，就把重复节点删掉了
        return deleteDuplicates(head.next);
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

16. 最接近的三数之和

LeetCode	力扣	难度
16. 3Sum Closest	16. 最接近的三数之和	🟠

- **标签：数组双指针**

给你一个长度为 `n` 的整数数组 `nums` 和一个目标值 `target`，请你从 `nums` 中选出三个整数，使它们的和与 `target` 最接近。

请你返回这三个数的和，题目保证每组输入只存在恰好一个解。

示例 1：

```
输入：nums = [-1,2,1,-4], target = 1
输出：2
解释：与 target 最接近的和是 2 (-1 + 2 + 1 = 2)。
```

基本思路

只要你做过 259. 较小的三数之和，这道题稍微改一下就应该能搞定了。

一样是先排序，然后固定第一个数，再去 `nums[start..]` 中寻找最接近 `target - delta` 的两数之和。

我写的解法其实可以合并成一个函数，但考虑到和前文 一个函数秒杀 nSum 问题 中代码的一致性，我还是把解法分成了两个函数来写。

解法代码

```
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        if (nums.length < 3) {
            return 0;
        }
        // 别忘了要先排序数组
        Arrays.sort(nums);
        // 记录三数之和与目标值的偏差
        int delta = Integer.MAX_VALUE;
        for (int i = 0; i < nums.length - 2; i++) {
            // 固定 nums[i] 为三数之和中的第一个数，
            // 然后对 nums[i+1..] 搜索接近 target - nums[i] 的两数之和
            int sum = nums[i] + twoSumClosest(nums, i + 1, target -
nums[i]);
            if (Math.abs(delta) > Math.abs(target - sum)) {
```

```
                delta = target - sum;
            }
        }
        return target - delta;
    }

    // 在 nums[start..] 搜索最接近 target 的两数之和
    int twoSumClosest(int[] nums, int start, int target) {
        int lo = start, hi = nums.length - 1;
        // 记录两数之和与目标值的偏差
        int delta = Integer.MAX_VALUE;
        while (lo < hi) {
            int sum = nums[lo] + nums[hi];
            if (Math.abs(delta) > Math.abs(target - sum)) {
                delta = target - sum;
            }
            if (sum < target) {
                lo++;
            } else {
                hi--;
            }
        }
        return target - delta;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

259. 较小的三数之和

LeetCode	力扣	难度
259. 3Sum Smaller	259. 较小的三数之和	🟠

- 标签：数组双指针

给定一个长度为 `n` 的整数数组和一个目标值 `target`，寻找能够使条件 `nums[i] + nums[j] + nums[k] < target` 成立的三元组 `i, j, k` 个数（`0 <= i < j < k < n`）。

示例 1：

```
输入：nums = [-2,0,1,3], target = 2
输出：2
解释：因为一共有两个三元组满足累加和小于 2:
     [-2,0,1]
     [-2,0,3]
```

基本思路

你需要先看一下前文 一个函数秒杀 nSum 问题，做一下 1099. 小于 K 的两数之和，然后来做本题就很容易了。

首先固定三数之和中的第一个数为 `nums[i]`，然后在剩余的子数组中搜索小于 `target - nums[i]` 的两数之和个数。

解法代码

```
class Solution {
    public int threeSumSmaller(int[] nums, int target) {
        if (nums.length < 3) {
            return 0;
        }
        // 别忘了要先排序数组
        Arrays.sort(nums);
        int res = 0;
        for (int i = 0; i < nums.length - 2; i++) {
            // 固定 nums[i] 为三数之和中的第一个数，
            // 然后对 nums[i+1..] 搜索小于 target - nums[i] 的两数之和个数
            res += twoSumSmaller(nums, i + 1, target - nums[i]);
        }
        return res;
    }
```

```
    // 在 nums[start..] 搜索小于 target 的两数之和个数
    int twoSumSmaller(int[] nums, int start, int target) {
        int lo = start, hi = nums.length - 1;
        int count = 0;
        while (lo < hi) {
            if (nums[lo] + nums[hi] < target) {
                // nums[lo] 和 nums[lo+1..hi]
                // 中的任一元素之和都小于 target
                count += hi - lo;
                lo++;
            } else {
                hi--;
            }
        }
        return count;
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

- **类似题目**：
  - 16. 最接近的三数之和 🟠

360. 有序转化数组

LeetCode	力扣	难度
360. Sort Transformed Array	360. 有序转化数组	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：dsvip，数据结构，数组双指针

给你一个已经**排好序**的整数数组 `nums` 和整数 `a, b, c`。对于数组中的每一个元素 `nums[i]`，计算函数值 `f(x) = ax2 + bx + c`，请按升序返回结果数组。

示例 1：

```
输入：nums = [-4,-2,2,4], a = 1, b = 3, c = 5
输出：[3,9,15,33]
```

基本思路

只要看过前文 链表的双指针技巧汇总 并做过 977. 有序数组的平方，应该有这道题的思路。

977 题其实就是这道题中 `a = 1, b = 0, c = 0` 的特殊情况，**所以这道题的关键也是在 `nums` 的开头和结尾设置 `i, j` 双指针相向而行，执行合并有序数组的逻辑**，只不过这里需要考虑的情况更多了一些罢了。

我们中学都学过这种二次函数，图像是一个抛物线，写个函数来表示：

```
int f(int x, int a, int b, int c) {
    return a*x*x + b*x + c;
}
```

`nums[i]` 就好像坐标系中 x 轴坐标，那么 `f(nums[i])` 之间的关系就取决于抛物线的对称轴位置以及抛物线的开口方向（`a` 的正负）。

如果 `nums` 中的元素全都落在抛物线的一侧，则这些元素本身就是有序递增或递减的，根据开口方向做判断就可以了，很容易处理。

关键是 `nums` 中的元素分布在在抛物线的两侧的情况，这就和 977 题的场景有些像，所以需要设置 `i, j` 双指针执行合并两个有序数组的逻辑了，当然还要考虑抛物线开口的方向。

有了上述思路，直接看代码吧。

解法代码

```java
class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
        // 双指针，相向而行，逼近对称轴
        int i = 0, j = nums.length - 1;
        // 如果开口朝上，越靠近对称轴函数值越小
        // 如果开口朝下，越靠近对称轴函数值越大
        int p = a > 0 ? nums.length - 1 : 0;
        int[] res = new int[nums.length];
        // 执行合并两个有序数组的逻辑
        while (i <= j) {
            int v1 = f(nums[i], a, b, c);
            int v2 = f(nums[j], a, b, c);
            if (a > 0) {
                // 如果开口朝上，越靠近对称轴函数值越小
                if (v1 > v2) {
                    res[p--] = v1;
                    i++;
                } else {
                    res[p--] = v2;
                    j--;
                }
            } else {
                // 如果开口朝下，越靠近对称轴函数值越大
                if (v1 > v2) {
                    res[p++] = v2;
                    j--;
                } else {
                    res[p++] = v1;
                    i++;
                }
            }
        }
        return res;
    }

    int f(int x, int a, int b, int c) {
        return a*x*x + b*x + c;
    }
}
```

▶ 代码可视化动画（需安装 Chrome 插件）

125. 验证回文串

LeetCode	力扣	难度
125. Valid Palindrome	125. 验证回文串	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回文串，数组双指针**

如果在将所有大写字符转换为小写字符、并移除所有非字母数字字符之后，短语正着读和反着读都一样。则可以认为该短语是一个**回文串**。

字母和数字都属于字母数字字符。

给你一个字符串 `s`，如果它是 **回文串**，返回 `true`；否则，返回 `false`。

示例 1：

```
输入：s = "A man, a plan, a canal: Panama"
输出：true
解释："amanaplanacanalpanama" 是回文串。
```

基本思路

这道题很简单，只要先把所有字符转化成小写，并过滤掉空格和标点这类字符，然后对剩下的字符执行 数组双指针技巧汇总 中提到的两端向中心的双指针算法即可。

解法代码

```
class Solution {
    public boolean isPalindrome(String s) {
        // 先把所有字符转化成小写，并过滤掉空格和标点这类字符
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isLetterOrDigit(c)) {
                sb.append(Character.toLowerCase(c));
            }
        }

        // 然后对剩下的这些目标字符执行双指针算法，判断回文串
        s = sb.toString();
        // 一左一右两个指针相向而行
        int left = 0, right = s.length() - 1;
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
```

```
                return false;
            }
            left++;
            right--;
        }
        return true;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 剑指 Offer II 018. 有效的回文 🟢

剑指 Offer II 018. 有效的回文

这道题和 125. 验证回文串 相同。

870. 优势洗牌

LeetCode	力扣	难度
870. Advantage Shuffle	870. 优势洗牌	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数组，数组双指针**

给定两个大小相等的数组 `A` 和 `B`，`A` 相对于 `B` 的**优势**可以用满足 `A[i] > B[i]` 的索引 `i` 的数目来描述。

请你返回 `A` 的**任意**排列，使其相对于 `B` 的优势最大化。

示例 1：

```
输入：A = [2,7,11,15], B = [1,10,4,11]
输出：[2,11,7,15]
```

基本思路

这题就像田忌赛马的情景，`nums1` 就是田忌的马，`nums2` 就是齐王的马，数组中的元素就是马的战斗力，你就是谋士孙膑，请你帮田忌安排赛马顺序，使胜场最多。

最优策略是将齐王和田忌的马按照战斗力排序，然后按照战斗力排名一一对比：

如果田忌的马能赢，那就比赛，如果赢不了，那就换个垫底的来送人头，保存实力。

具体分析见详细题解。

- **详细题解：田忌赛马背后的算法决策**

解法代码

```
class Solution {
    public int[] advantageCount(int[] nums1, int[] nums2) {
        int n = nums1.length;
        // 给 nums2 降序排序
        PriorityQueue<int[]> maxpq = new PriorityQueue<>(
                (int[] pair1, int[] pair2) -> {
                    return pair2[1] - pair1[1];
                }
        );
        for (int i = 0; i < n; i++) {
            maxpq.offer(new int[]{i, nums2[i]});
        }
        // 给 nums1 升序排序
        Arrays.sort(nums1);
```

```
        // nums1[left] 是最小值，nums1[right] 是最大值
        int left = 0, right = n - 1;
        int[] res = new int[n];

        while (!maxpq.isEmpty()) {
            int[] pair = maxpq.poll();
            // maxval 是 nums2 中的最大值，i 是对应索引
            int i = pair[0], maxval = pair[1];
            if (maxval < nums1[right]) {
                // 如果 nums1[right] 能胜过 maxval，那就自己上
                res[i] = nums1[right];
                right--;
            } else {
                // 否则用最小值混一下，养精蓄锐
                res[i] = nums1[left];
                left++;
            }
        }
        return res;
    }
}
```

1099. 小于 K 的两数之和

LeetCode	力扣	难度
1099. Two Sum Less Than K	1099. 小于 K 的两数之和	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数组双指针

给你一个整数数组 `nums` 和整数 `k`，返回最大和 `sum`，满足存在 `i < j` 使得 `nums[i] + nums[j] = sum` 且 `sum < k`。如果没有满足此等式的 `i, j` 存在，则返回 `-1`。

示例 1：

```
输入：nums = [34,23,1,24,75,33,54,8], k = 60
输出：58
解释：
34 和 24 相加得到 58，58 小于 60，满足题意。
```

示例 2：

```
输入：nums = [10,20,30], k = 15
输出：-1
解释：
我们无法找到和小于 15 的两个元素。
```

基本思路

如果你看过前文 一个函数秒杀 nSum 问题，那么这道题应该很简单：

先把数组排序，然后用左右指针技巧控制两数之和增大和减小，进而找到最接近 `k` 的两数之和。

解法代码

```java
class Solution {
    public int twoSumLessThanK(int[] nums, int k) {
        // 数组双指针一般都要先排序
        Arrays.sort(nums);
        // 左右指针技巧
        int lo = 0, hi = nums.length - 1;
        int sum = -1;
        while (lo < hi) {
            if (nums[lo] + nums[hi] < k) {
                // 比目标值 k 小，则右移左指针
```

```
                sum = Math.max(sum, nums[lo] + nums[hi]);
                lo++;
            } else {
                // 比目标值 k 大，则左移右指针
                hi--;
            }
        }
        return sum;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 259. 较小的三数之和 🟠

2. 两数相加

LeetCode	力扣	难度
2. Add Two Numbers	2. 两数相加	🟠

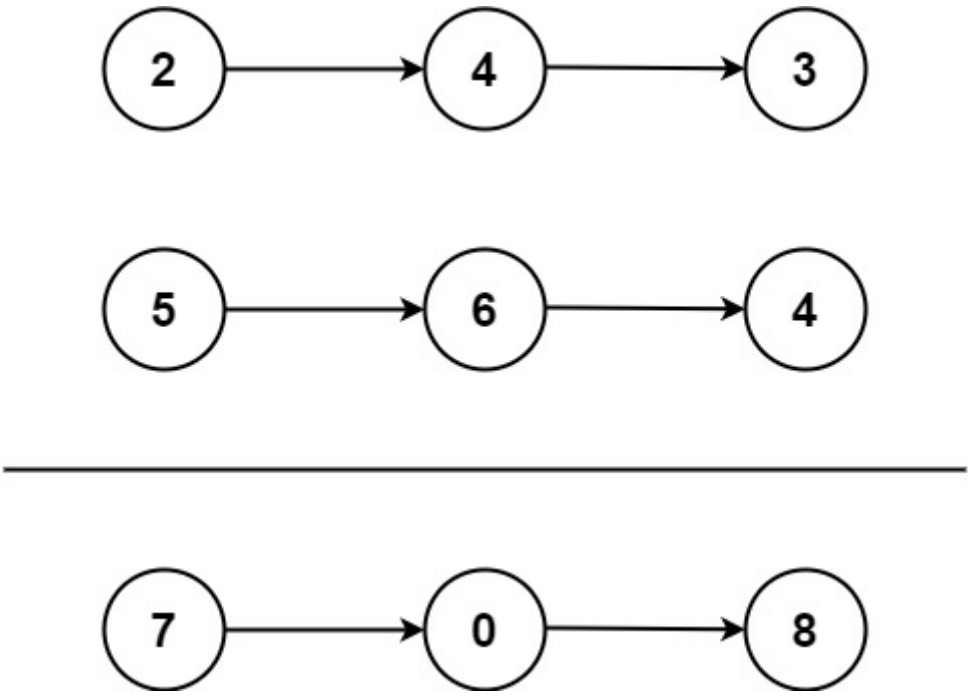
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表双指针

给你两个 **非空** 的链表，表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的，并且每个节点只能存储 **一位** 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表，你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

示例 1：




```
输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]
解释：342 + 465 = 807.
```

基本思路

逆序存储很友好了，直接遍历链表就是从个位开始的，符合我们计算加法的习惯顺序。如果是正序存储，那倒要费点脑筋了，可能需要 翻转链表 或者使用栈来辅助。

这道题主要考察 链表双指针技巧 和加法运算过程中对进位的处理。注意这个 `carry` 变量的处理，在我们手动模拟加法过程的时候会经常用到。

代码中还用到一个链表的算法题中是很常见的「虚拟头结点」技巧，也就是 dummy 节点。你可以试试，如果不使用 dummy 虚拟节点，代码会稍显复杂，而有了 dummy 节点这个占位符，可以避免处理初始的空指针情况，降低代码的复杂性。

解法代码

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        // 在两条链表上的指针
        ListNode p1 = l1, p2 = l2;
        // 虚拟头结点（构建新链表时的常用技巧）
        ListNode dummy = new ListNode(-1);
        // 指针 p 负责构建新链表
        ListNode p = dummy;
        // 记录进位
        int carry = 0;
        // 开始执行加法，两条链表走完且没有进位时才能结束循环
        while (p1 != null || p2 != null || carry > 0) {
            // 先加上上次的进位
            int val = carry;
            if (p1 != null) {
                val += p1.val;
                p1 = p1.next;
            }
            if (p2 != null) {
                val += p2.val;
                p2 = p2.next;
            }
            // 处理进位情况
            carry = val / 10;
            val = val % 10;
            // 构建新节点
            p.next = new ListNode(val);
            p = p.next;
        }
        // 返回结果链表的头结点（去除虚拟头结点）
        return dummy.next;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 445. 两数相加 II 🟠
  - 67. 二进制求和 🟢
  - 剑指 Offer II 025. 链表中的两数相加 🟠

445. 两数相加 II

LeetCode	力扣	难度
445. Add Two Numbers II	445. 两数相加 II	🟠

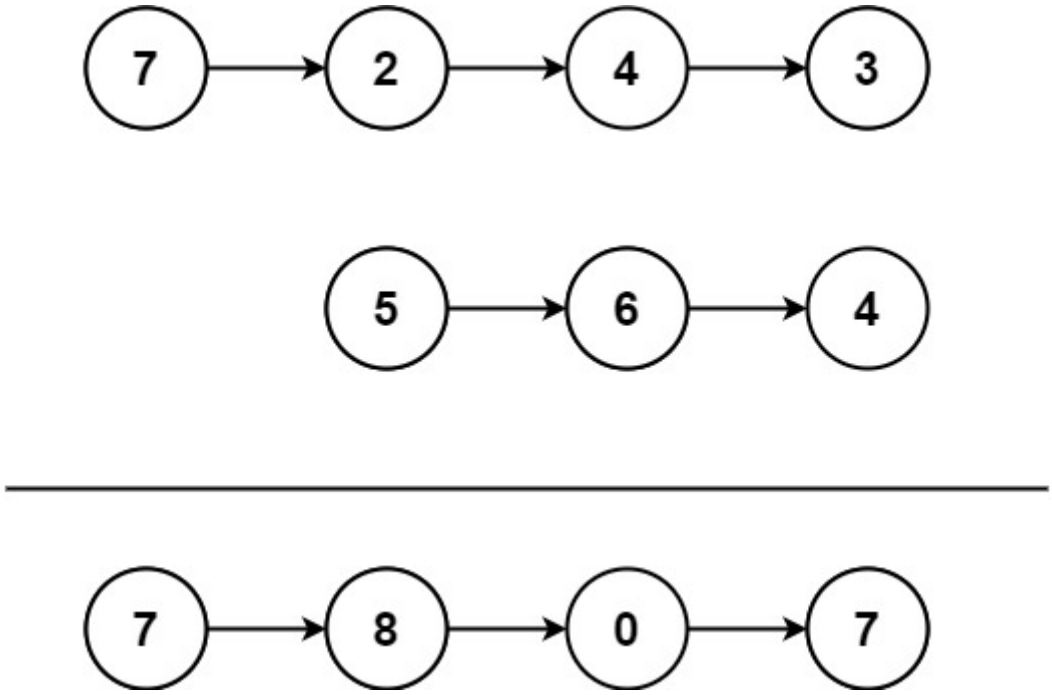
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：栈，链表

给你两个 **非空**链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例 1：




```
输入：l1 = [7,2,4,3], l2 = [5,6,4]
输出：[7,8,0,7]
```

示例 2：

```
输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[8,0,7]
```

示例 3：

```
输入：l1 = [0], l2 = [0]
输出：[0]
```

基本思路

这道题是 2. 两数相加 的进阶问题，我们模拟加法运算当然是从最低位开始加，这样才能正确的处理进位。但现在单链表的开头是最高位，那么最直接的想法就是先 翻转链表，这样就可以继续玩第 2 题那一套了，没什么难度。

不过本题也说了，如果不让你反转链表怎么办？其实也好办，我们可以利用栈这种先进后出的数据结构，把链表节点从头到尾放进栈中，再从栈拿出来就是从尾到头的顺序，相当于是反转链表的效果，然后又回到了第 2 题的加法逻辑。

还有一个需要注意的是，计算结果的高位也应该放在结果链表的左侧，也就是插入到 dummy 节点的后面。具体看代码吧。

解法代码

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        // 把链表元素转入栈中
        Stack<Integer> stk1 = new Stack<>();
        while (l1 != null) {
            stk1.push(l1.val);
            l1 = l1.next;
        }
        Stack<Integer> stk2 = new Stack<>();
        while (l2 != null) {
            stk2.push(l2.val);
            l2 = l2.next;
        }

        // 接下来基本上是复用我在第 2 题的代码逻辑
        // 注意新节点要直接插入到 dummy 后面

        // 虚拟头结点（构建新链表时的常用技巧）
        ListNode dummy = new ListNode(-1);

        // 记录进位
        int carry = 0;
        // 开始执行加法，两条链表走完且没有进位时才能结束循环
        while (!stk1.isEmpty() || !stk2.isEmpty() || carry > 0) {
            // 先加上上次的进位
            int val = carry;
            if (!stk1.isEmpty()) {
                val += stk1.pop();
            }
            if (!stk2.isEmpty()) {
                val += stk2.pop();
```

```
                }
                // 处理进位情况
                carry = val / 10;
                val = val % 10;
                // 构建新节点，直接接在 dummy 后面
                ListNode newNode = new ListNode(val);
                newNode.next = dummy.next;
                dummy.next = newNode;
            }
            // 返回结果链表的头结点（去除虚拟头结点）
            return dummy.next;
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 剑指 Offer II 025. 链表中的两数相加 🟠

67. 二进制求和

LeetCode	力扣	难度
67. Add Binary	67. 二进制求和	🟢

- **标签：位运算，数学**

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 **非空** 字符串且只包含数字 `1` 和 `0`。

示例 1:

```
输入: a = "11", b = "1"
输出: "100"
```

示例 2:

```
输入: a = "1010", b = "1011"
输出: "10101"
```

基本思路

这道题的关键就是让你用代码模拟加法的运算过程。**加法的模拟可以理解为是一个通用模式，需要你用一个 `carry` 变量记录并正确进位**，建议你先去做一下 2. 两数相加 再回来看这道题就手到擒来了。

解法代码

```
class Solution {
    public String addBinary(String a, String b) {
        // 先把输入的这两个二进制串反转，低位放在前面，方便处理进位
        a = new StringBuilder(a).reverse().toString();
        b = new StringBuilder(b).reverse().toString();
        // 存储结果
        StringBuilder sb = new StringBuilder();

        int m = a.length(), n = b.length();
        // carry 记录进位
        int carry = 0;
        int i = 0;

        // 开始类似 [2. 两数相加](#2) 的加法模拟逻辑
```

```
        // 只是这里运算的是二进制字符串
        while (i < Math.max(m, n) || carry > 0) {
            int val = carry;
            val += i < m ? (a.charAt(i) - '0') : 0;
            val += i < n ? (b.charAt(i) - '0') : 0;
            sb.append(val % 2);
            carry = val / 2;
            i++;
        }

        return sb.reverse().toString();
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

剑指 Offer II 025. 链表中的两数相加

这道题和 445. 两数相加 II 相同。

141. 环形链表

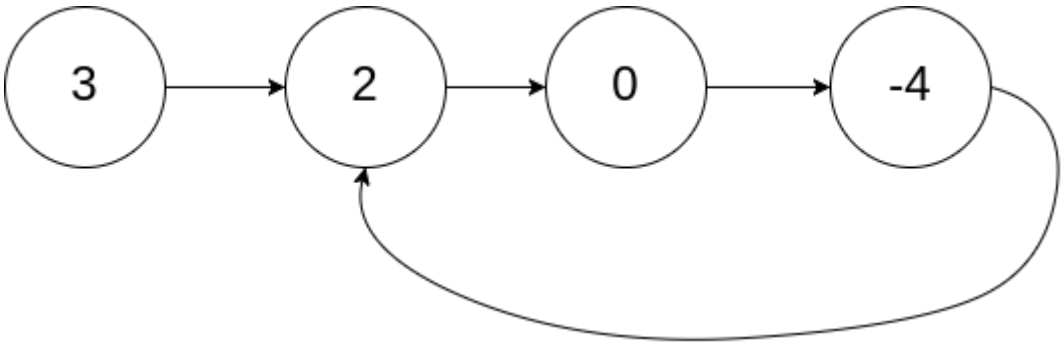
LeetCode	力扣	难度
141. Linked List Cycle	141. 环形链表	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表，链表双指针

给定一个链表，判断链表中是否有环，如果链表中存在环，则返回 `true`，否则返回 `false`。

示例 1：




```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

PS：这道题在《算法小抄》 的第 64 页。

经典题目了，要使用双指针技巧中的快慢指针，每当慢指针 `slow` 前进一步，快指针 `fast` 就前进两步。

如果 `fast` 最终遇到空指针，说明链表中没有环；如果 `fast` 最终和 `slow` 相遇，那肯定是 `fast` 超过了 `slow` 一圈，说明链表中含有环。

- 详细题解：双指针技巧秒杀七道链表题目

解法代码

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        // 快慢指针初始化指向 head
        ListNode slow = head, fast = head;
        // 快指针走到末尾时停止
```

```
        while (fast != null && fast.next != null) {
            // 慢指针走一步，快指针走两步
            slow = slow.next;
            fast = fast.next.next;
            // 快慢指针相遇，说明含有环
            if (slow == fast) {
                return true;
            }
        }
        // 不包含环
        return false;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 21. 合并两个有序链表 🟢
  - 23. 合并K个升序链表 🔴
  - 86. 分隔链表 🟠
  - 876. 链表的中间结点 🟢
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 078. 合并排序链表 🔴

142. 环形链表 II

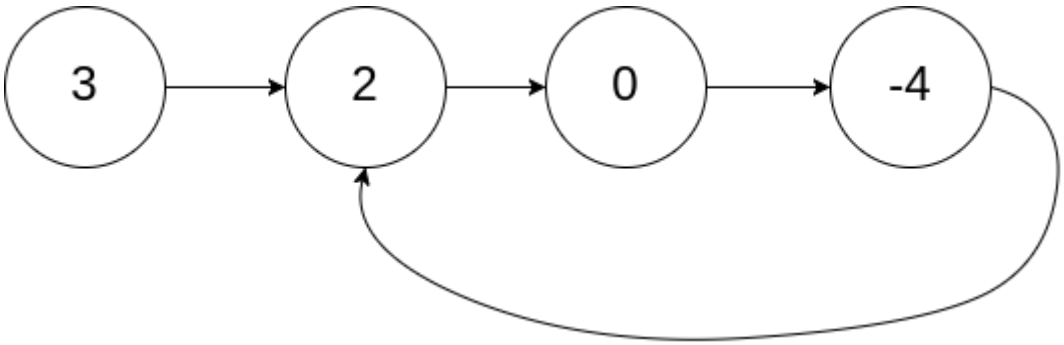
LeetCode	力扣	难度
142. Linked List Cycle II	142. 环形链表 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表，链表双指针

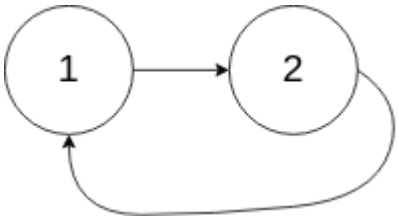
给定一个链表，返回链表开始入环的第一个节点，如果链表无环，则返回 `null`（**不允许修改给定的链表**）。

示例 1：




```
输入：head = [3,2,0,-4], pos = 1
输出：返回索引为 1 的链表节点
解释：链表中有一个环，其尾部连接到第二个节点。
```

示例 2：



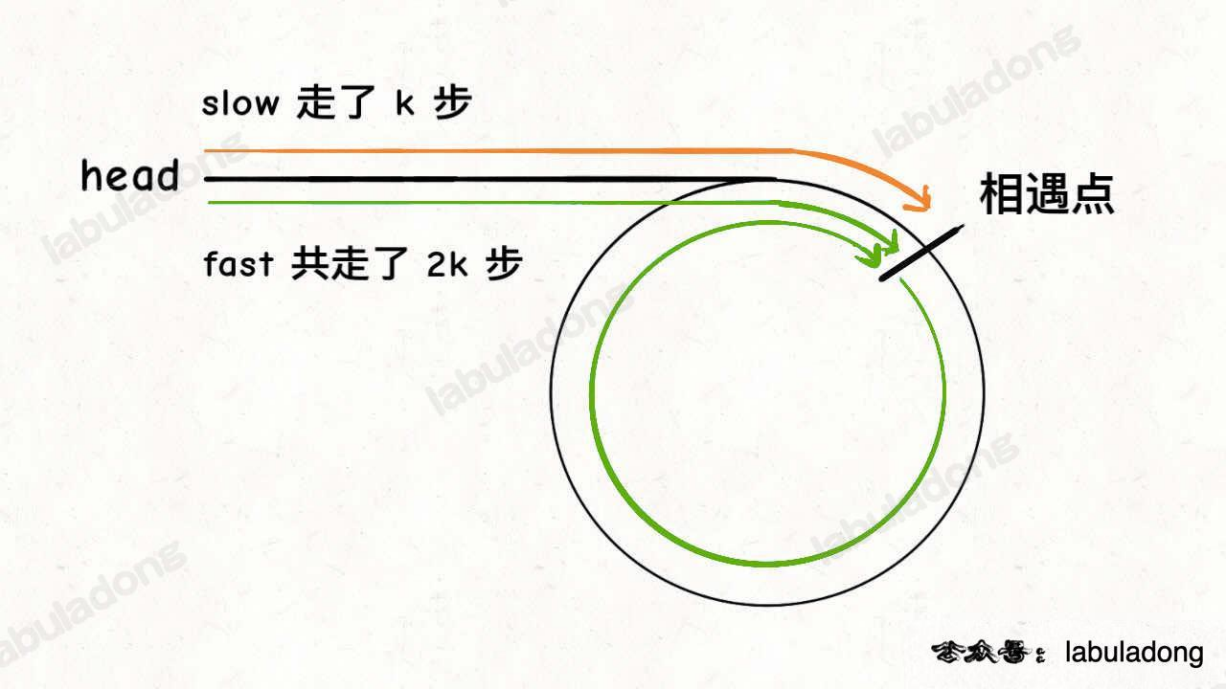

```
输入：head = [1,2], pos = 0
输出：返回索引为 0 的链表节点
解释：链表中有一个环，其尾部连接到第一个节点。
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

基于 141. 环形链表 的解法，直观地来说就是当快慢指针相遇时，让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。

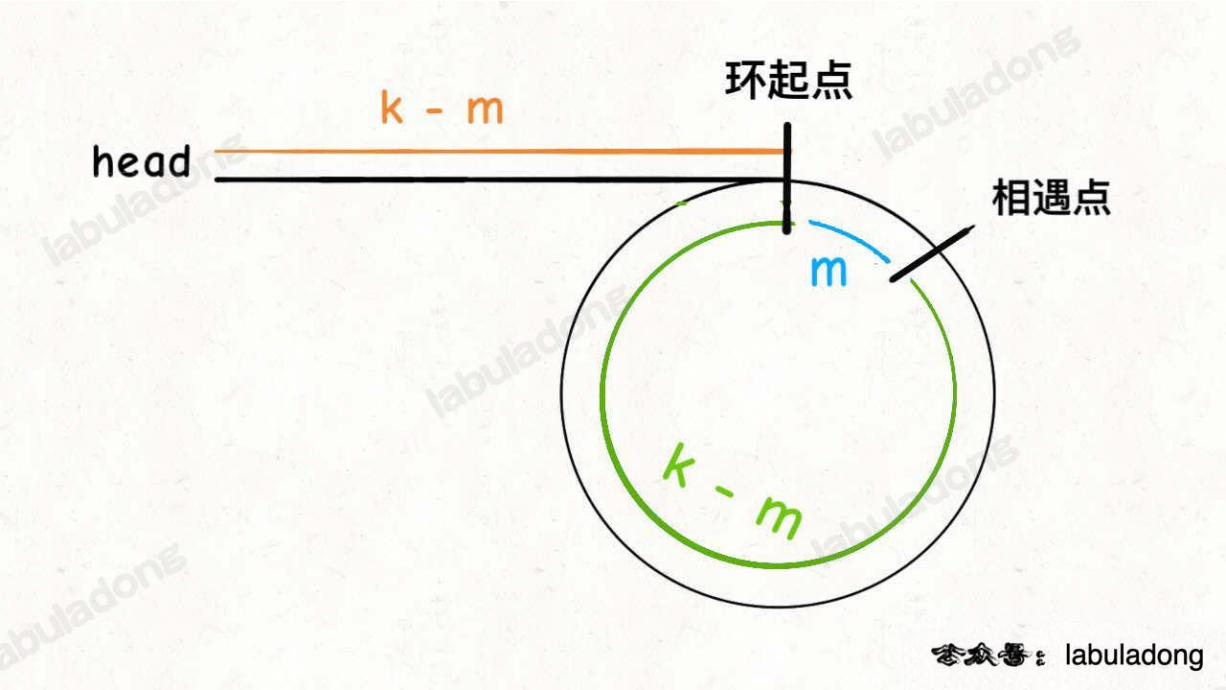
原理也简单说下吧，我们假设快慢指针相遇时，慢指针 `slow` 走了 `k` 步，那么快指针 `fast` 一定走了 `2k` 步：



`fast` 一定比 `slow` 多走了 `k` 步，这多走的 `k` 步其实就是 `fast` 指针在环里转圈圈，所以 `k` 的值就是环长度的「整数倍」。

假设相遇点距环的起点的距离为 `m`，那么结合上图的 `slow` 指针，环的起点距头结点 `head` 的距离为 `k - m`，也就是说如果从 `head` 前进 `k - m` 步就能到达环起点。

巧的是，如果从相遇点继续前进 `k - m` 步，也恰好到达环起点：



所以，只要我们把快慢指针中的任一个重新指向 `head`，然后两个指针同速前进，`k - m` 步后一定会相遇，相遇之处就是环的起点了。

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast, slow;
        fast = slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        // 上面的代码类似 hasCycle 函数
        if (fast == null || fast.next == null) {
            // fast 遇到空指针说明没有环
            return null;
        }

        // 重新指向头结点
        slow = head;
        // 快慢指针同步前进，相交点就是环起点
        while (slow != fast) {
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 141. 环形链表 🟢
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 21. 合并两个有序链表 🟢
  - 23. 合并K个升序链表 🔴
  - 86. 分隔链表 🟠
  - 876. 链表的中间结点 🟢
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 078. 合并排序链表 🔴

160. 相交链表

LeetCode	力扣	难度
160. Intersection of Two Linked Lists	160. 相交链表	🟢

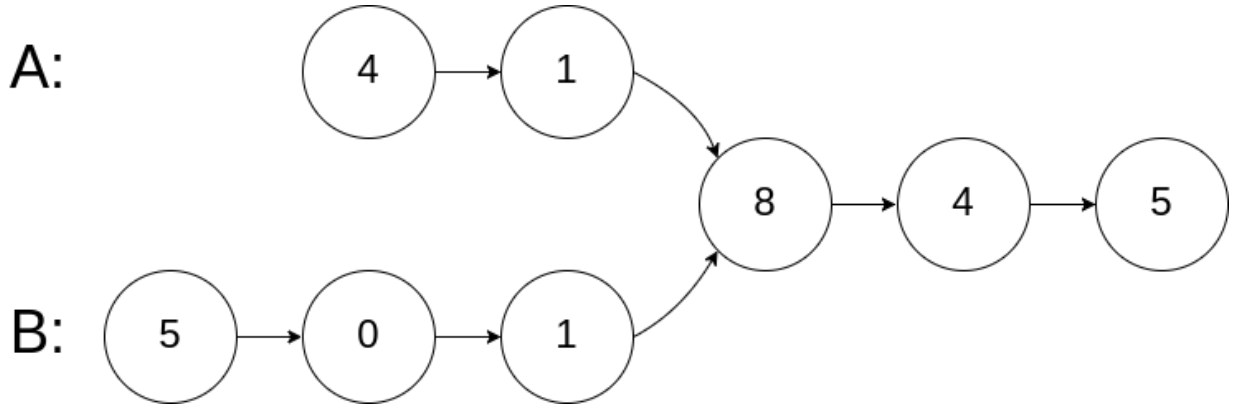
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表，链表双指针

给你两个单链表的头节点 `headA` 和 `headB`，请你找出并返回两个单链表相交的起始节点。如果两个链表没有交点，返回 `null`。

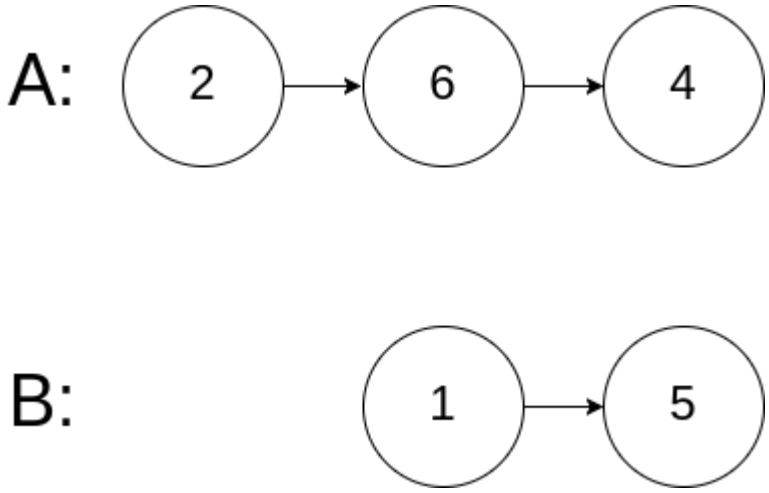
题目数据**保证**整个链式结构中不存在环，算法不能修改链表的原始结构。

示例 1：




```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Intersected at '8'
解释：相交节点的值为 8（注意，如果两个链表相交则不能为 0）。
从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。
在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

示例 2：



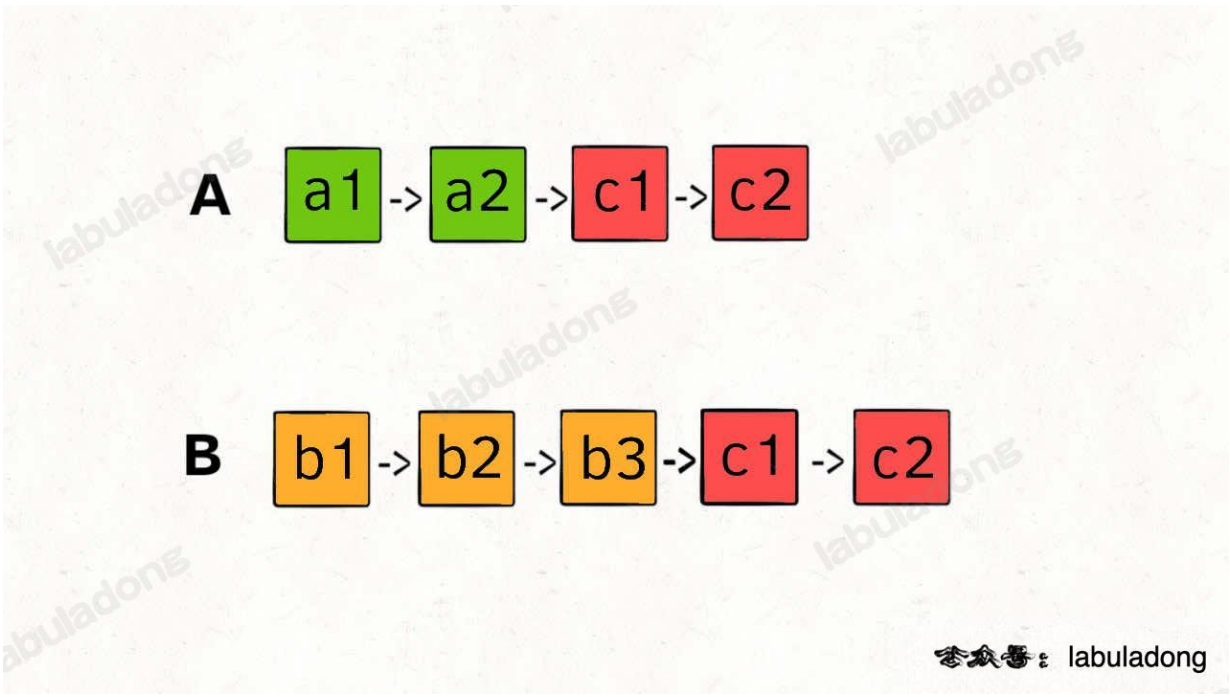

```
输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。
由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
这两个链表不相交，因此返回 null。
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

PS：这道题在《算法小抄》 的第 64 页。

这题难点在于，由于两条链表的长度可能不同，两条链表之间的节点无法对应：

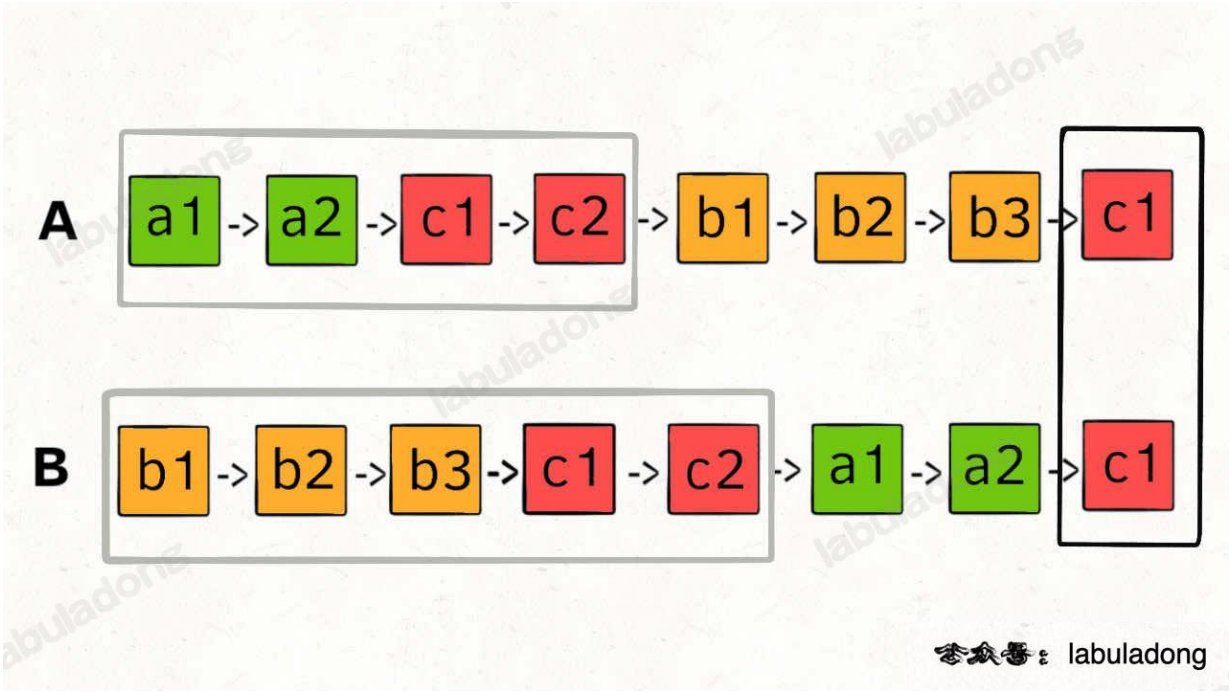


如果用两个指针 `p1` 和 `p2` 分别在两条链表上前进，并不能**同时**走到公共节点，也就无法得到相交节点 `c1`。

解决这个问题的关键是，通过某些方式，让 `p1` 和 `p2` 能够同时到达相交节点 `c1`。

如果用两个指针 p1 和 p2 分别在两条链表上前进，我们可以让 p1 遍历完链表 A 之后开始遍历链表 B，让 p2 遍历完链表 B 之后开始遍历链表 A，这样相当于「逻辑上」两条链表接在了一起。

如果这样进行拼接，就可以让 p1 和 p2 同时进入公共部分，也就是同时到达相交节点 c1：



另一种思路，先计算两条链表的长度，然后让 p1 和 p2 距离链表尾部的距离相同，然后齐头并进，

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        // p1 指向 A 链表头结点，p2 指向 B 链表头结点
        ListNode p1 = headA, p2 = headB;
        while (p1 != p2) {
            // p1 走一步，如果走到 A 链表末尾，转到 B 链表
            if (p1 == null) p1 = headB;
            else            p1 = p1.next;
            // p2 走一步，如果走到 B 链表末尾，转到 A 链表
            if (p2 == null) p2 = headA;
            else            p2 = p2.next;
        }
        return p1;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目：**
  - 141. 环形链表 🟢

- 142. 环形链表 II 🟠
- 1650. 二叉树的最近公共祖先 III 🟠
- 19. 删除链表的倒数第 N 个结点 🟠
- 21. 合并两个有序链表 🟢
- 23. 合并K个升序链表 🔴
- 86. 分隔链表 🟠
- 876. 链表的中间结点 🟢
- 剑指 Offer 22. 链表中倒数第k个节点 🟢
- 剑指 Offer 25. 合并两个排序的链表 🟢
- 剑指 Offer 52. 两个链表的第一个公共节点 🟢
- 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
- 剑指 Offer II 022. 链表中环的入口节点 🟠
- 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
- 剑指 Offer II 078. 合并排序链表 🔴

19. 删除链表的倒数第 N 个结点

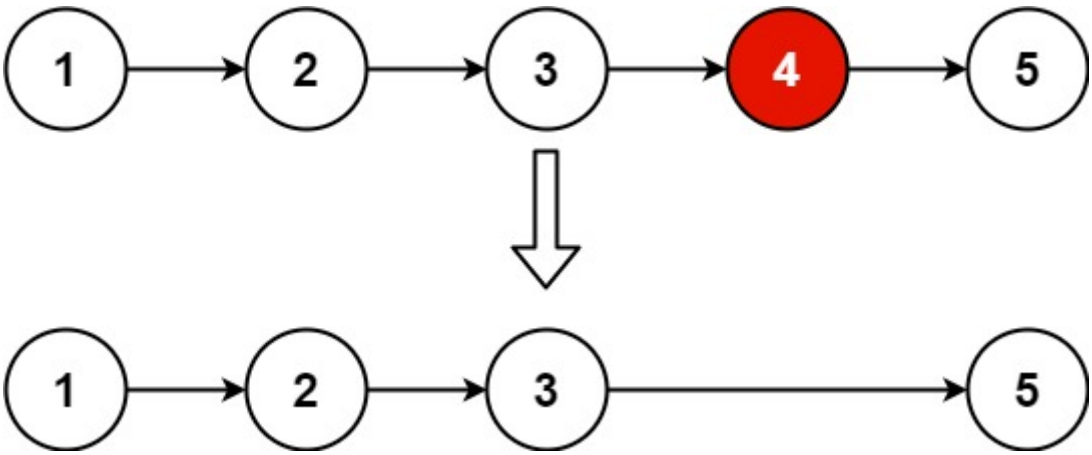
LeetCode	力扣	难度
19. Remove Nth Node From End of List	19. 删除链表的倒数第 N 个结点	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表，链表双指针

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

示例 1：




```
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]
```

基本思路

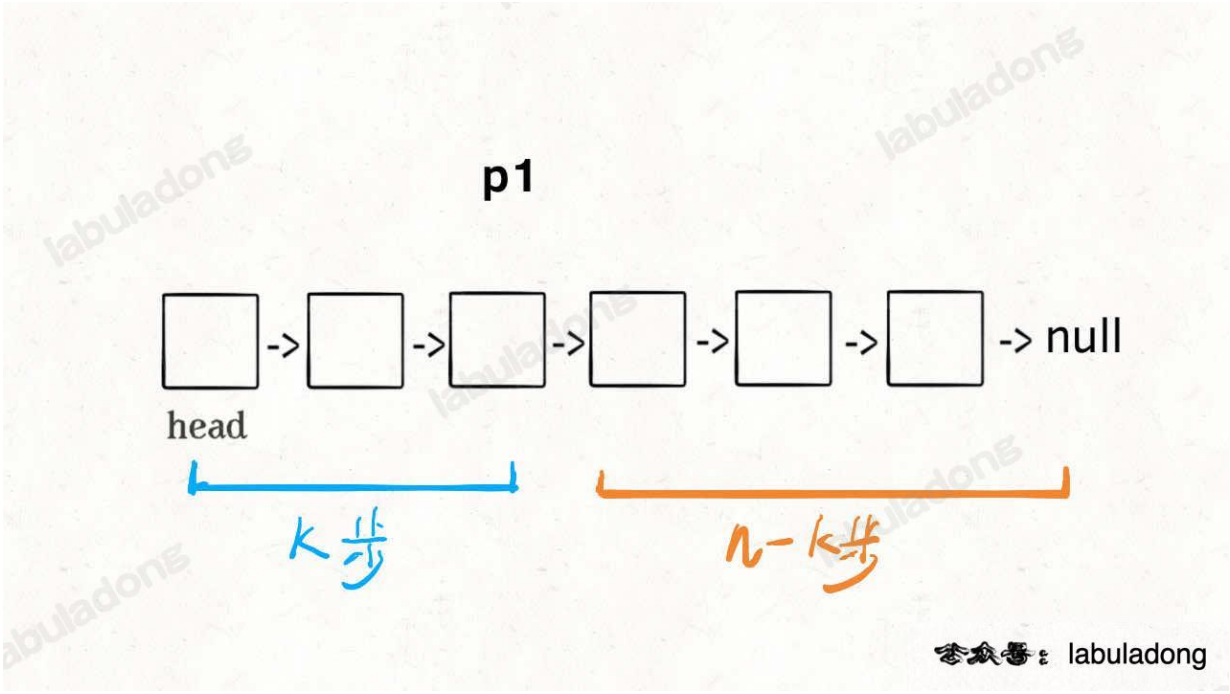
> 本文有视频版：链表双指针技巧全面汇总

PS：这道题在《算法小抄》 的第 64 页。

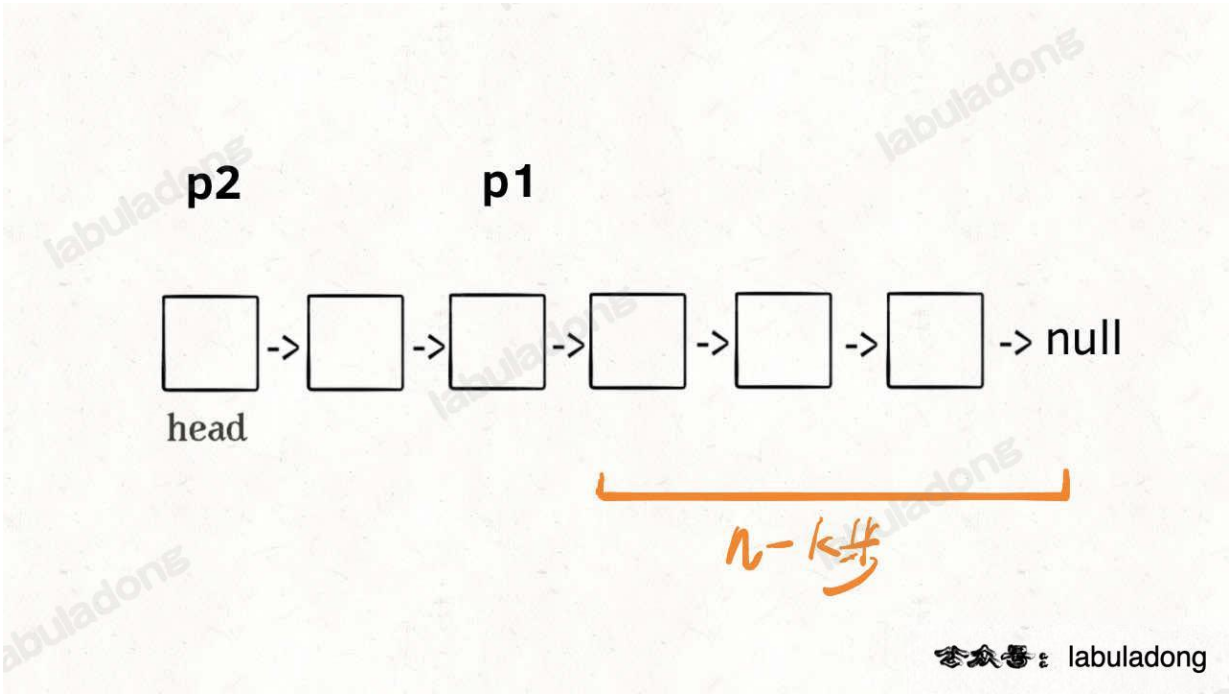
要删除倒数第 n 个节点，就得获得倒数第 n + 1 个节点的引用。

获取单链表的倒数第 k 个节点，就是想考察 双指针技巧 中快慢指针的运用，一般都会要求你**只遍历一次链表**，就算出倒数第 k 个节点。

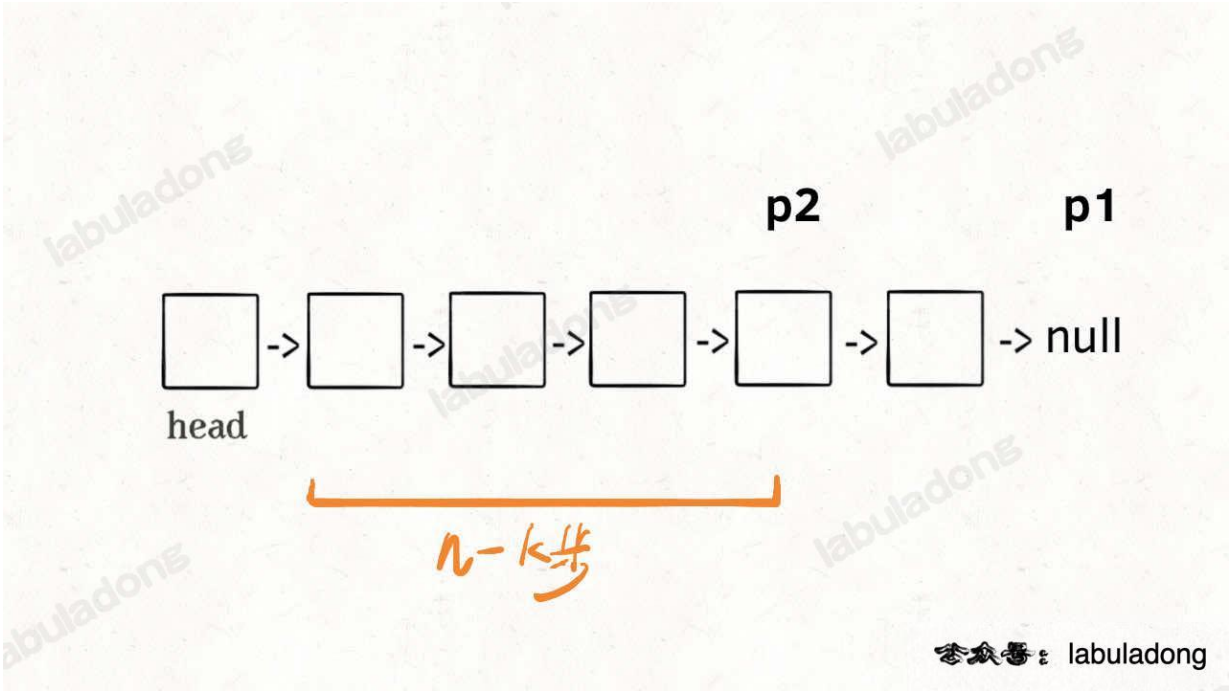
第一步，我们先让一个指针 p1 指向链表的头节点 head，然后走 k 步：



第二步，用一个指针 p2 指向链表头节点 head：



第三步，让 p1 和 p2 同时向前走，p1 走到链表末尾的空指针时走了 n - k 步，p2 也走了 n - k 步，也就是链表的倒数第 k 个节点：



这样，只遍历了一次链表，就获得了倒数第 k 个节点 p2。

解法中在链表头部接一个虚拟节点 dummy 是为了避免删除倒数第一个元素时出现空指针异常，在头部加入 dummy 节点并不影响尾部倒数第 k 个元素是什么。

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
class Solution {
    // 主函数
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // 虚拟头结点
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        // 删除倒数第 n 个，要先找倒数第 n + 1 个节点
        ListNode x = findFromEnd(dummy, n + 1);
        // 删掉倒数第 n 个节点
        x.next = x.next.next;
        return dummy.next;
    }

    // 返回链表的倒数第 k 个节点
    ListNode findFromEnd(ListNode head, int k) {
        ListNode p1 = head;
        // p1 先走 k 步
        for (int i = 0; i < k; i++) {
            p1 = p1.next;
        }
        ListNode p2 = head;
        // p1 和 p2 同时走 n - k 步
        while (p1 != null) {
            p2 = p2.next;
```

```
            p1 = p1.next;
        }
        // p2 现在指向第 n - k 个节点
        return p2;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 141. 环形链表 🟢
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 21. 合并两个有序链表 🟢
  - 23. 合并K个升序链表 🔴
  - 86. 分隔链表 🟠
  - 876. 链表的中间结点 🟢
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 078. 合并排序链表 🔴

21. 合并两个有序链表

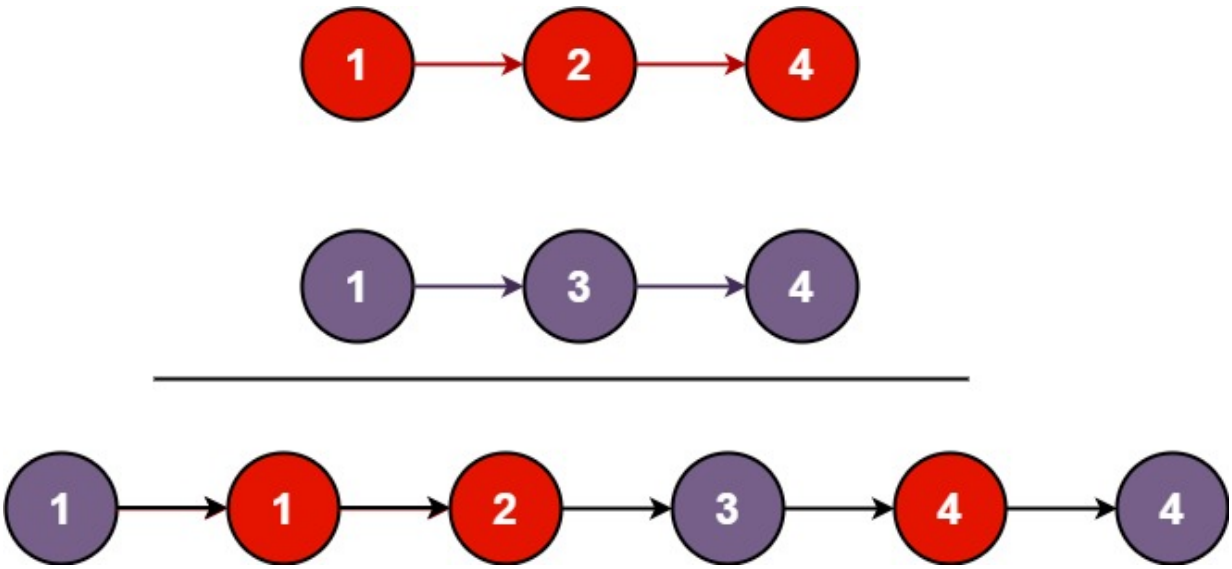
LeetCode	力扣	难度
21. Merge Two Sorted Lists	21. 合并两个有序链表	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表，链表双指针

输入两个升序链表，将它们合并为一个新的**升序**链表并返回。

示例 1：

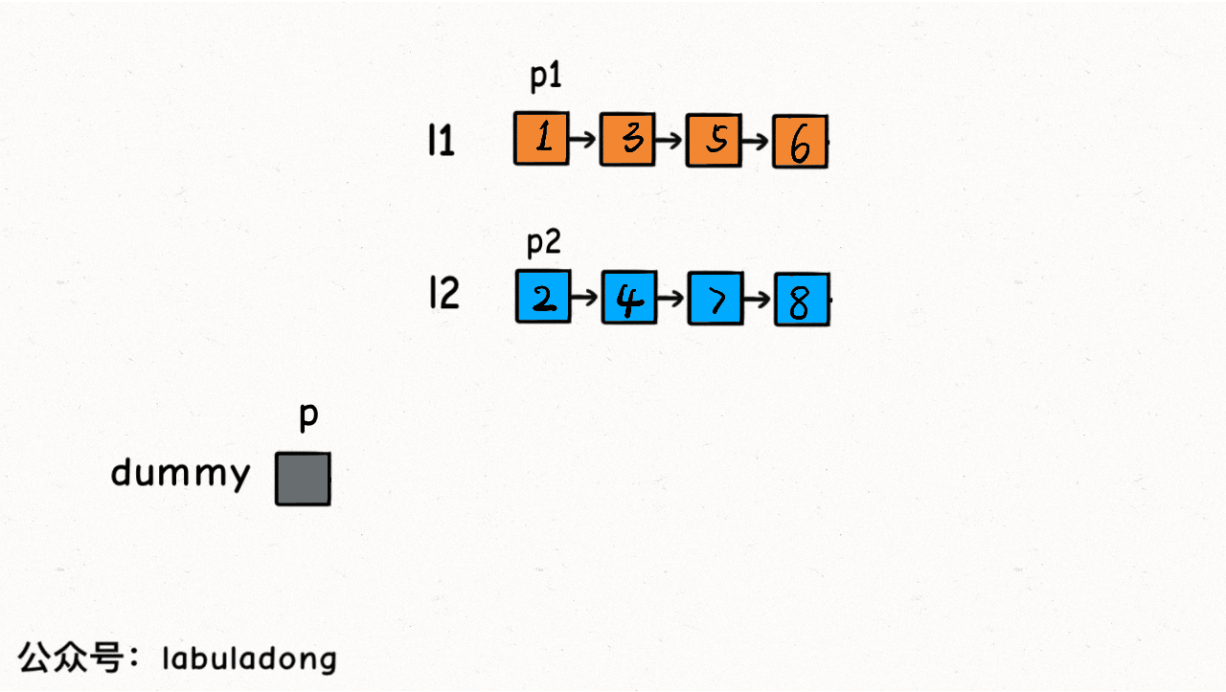



```
输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

经典算法题了，双指针技巧 用起来。



这个算法的逻辑类似于「拉拉链」，`l1, l2` 类似于拉链两侧的锯齿，指针 `p` 就好像拉链的拉索，将两个有序链表合并。

代码中还用到一个链表的算法题中是很常见的「虚拟头结点」技巧，也就是 `dummy` 节点，它相当于是个占位符，可以避免处理空指针的情况，降低代码的复杂性。

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 虚拟头结点
        ListNode dummy = new ListNode(-1), p = dummy;
        ListNode p1 = l1, p2 = l2;

        while (p1 != null && p2 != null) {
            // 比较 p1 和 p2 两个指针
            // 将值较小的的节点接到 p 指针
            if (p1.val > p2.val) {
                p.next = p2;
                p2 = p2.next;
            } else {
                p.next = p1;
                p1 = p1.next;
            }
            // p 指针不断前进
            p = p.next;
        }

        if (p1 != null) {
            p.next = p1;
        }
```

```
            if (p2 != null) {
                p.next = p2;
            }

            return dummy.next;
        }
    }
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1305. 两棵二叉搜索树中的所有元素 🟠
  - 141. 环形链表 🟢
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 23. 合并K个升序链表 🔴
  - 264. 丑数 II 🟠
  - 313. 超级丑数 🟠
  - 86. 分隔链表 🟠
  - 876. 链表的中间结点 🟢
  - 88. 合并两个有序数组 🟢
  - 97. 交错字符串 🟠
  - 977. 有序数组的平方 🟢
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 49. 丑数 🟠
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 078. 合并排序链表 🔴

23. 合并 K 个升序链表

LeetCode	力扣	难度
23. Merge k Sorted Lists	23. 合并K个升序链表	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数据结构，链表，链表双指针**

给你一个链表数组，每个链表都已经按升序排列，请你将这些链表合并成一个升序链表，返回合并后的链表。

示例 1：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

21. 合并两个有序链表 的延伸，利用 优先级队列（二叉堆） 进行节点排序即可。

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists.length == 0) return null;
        // 虚拟头结点
        ListNode dummy = new ListNode(-1);
        ListNode p = dummy;
        // 优先级队列，最小堆
        PriorityQueue<ListNode> pq = new PriorityQueue<>(
            lists.length, (a, b)->(a.val - b.val));
        // 将 k 个链表的头结点加入最小堆
        for (ListNode head : lists) {
```

```
                if (head != null)
                    pq.add(head);
            }

            while (!pq.isEmpty()) {
                // 获取最小节点，接到结果链表中
                ListNode node = pq.poll();
                p.next = node;
                if (node.next != null) {
                    pq.add(node.next);
                }
                // p 指针不断前进
                p = p.next;
            }
            return dummy.next;
        }
    }
```

- **类似题目**：
  - 141. 环形链表 🟢
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 21. 合并两个有序链表 🟢
  - 313. 超级丑数 🟠
  - 355. 设计推特 🟠
  - 373. 查找和最小的 K 对数字 🟠
  - 378. 有序矩阵中第 K 小的元素 🟠
  - 86. 分隔链表 🟠
  - 876. 链表的中间结点 🟢
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 061. 和最小的 k 个数对 🟠
  - 剑指 Offer II 078. 合并排序链表 🔴

86. 分隔链表

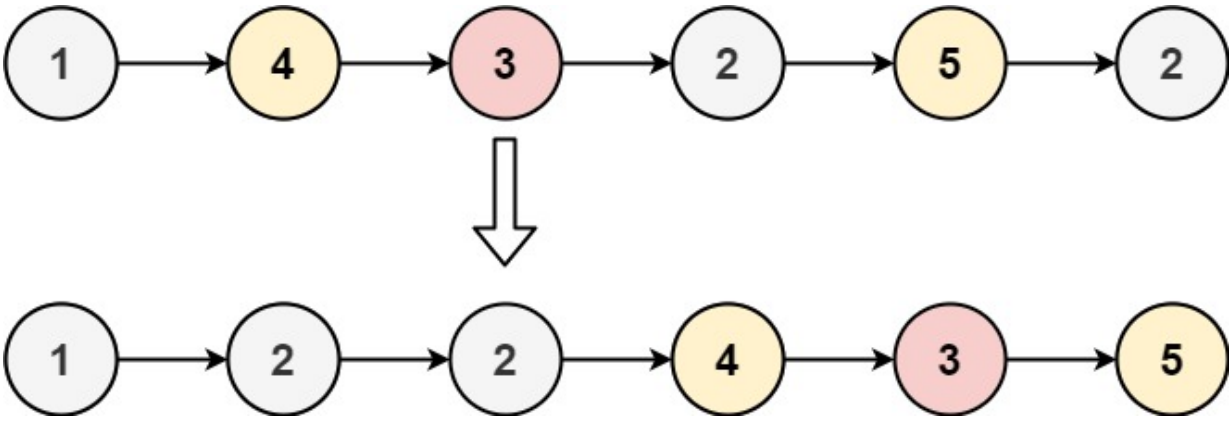
LeetCode	力扣	难度
86. Partition List	86. 分隔链表	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：dsvip，数据结构，链表双指针**

给你一个链表的头节点 head 和一个特定值 x，请你对链表进行分隔，使得所有**小于** x 的节点都出现在**大于或等于** x 的节点之前，你应当**保留**两个分区中每个节点的初始相对位置。

示例 1：




```
输入：head = [1,4,3,2,5,2], x = 3
输出：[1,2,2,4,3,5]
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

这道题很像 21. 合并两个有序链表，21 题让你合二为一，这里需要分解让你把原链表一分为二。

具体来说，我们可以把原链表分成两个小链表，一个链表中的元素大小都小于 x，另一个链表中的元素都大于等于 x，最后再把这两条链表接到一起，就得到了题目想要的结果。细节看代码吧，注意虚拟头结点的运用。

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
class Solution {
    public ListNode partition(ListNode head, int x) {
        // 存放小于 x 的链表的虚拟头结点
```

```
        ListNode dummy1 = new ListNode(-1);
        // 存放大于等于 x 的链表的虚拟头结点
        ListNode dummy2 = new ListNode(-1);
        // p1, p2 指针负责生成结果链表
        ListNode p1 = dummy1, p2 = dummy2;
        // p 负责遍历原链表，类似合并两个有序链表的逻辑
        // 这里是将一个链表分解成两个链表
        ListNode p = head;
        while (p != null) {
            if (p.val >= x) {
                p2.next = p;
                p2 = p2.next;
            } else {
                p1.next = p;
                p1 = p1.next;
            }
            // 断开原链表中的每个节点的 next 指针
            ListNode temp = p.next;
            p.next = null;
            p = temp;
        }
        // 链接两个链表
        p1.next = dummy2.next;

        return dummy1.next;
    }
}
```

肯定有读者对「断开原链表中的每个节点的 next 指针」这部分代码有疑问，借助我们的可视化面板就很容易看明白了，首先看下正确的写法：

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

如果你不断开原链表中的每个节点的 next 指针就会出错，因为结果链表中会包含一个环：

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

总的来说，如果我们需要把原链表的节点接到新链表上，而不是 new 新节点来组成新链表的话，那么断开节点和原链表之间的链接可能是必要的。那其实我们可以养成一个好习惯，但凡遇到这种情况，就把原链表的节点断开，这样就不会出错了。

- **类似题目**：
  - 141. 环形链表 🟢
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 21. 合并两个有序链表 🟢
  - 23. 合并K个升序链表 🔴
  - 876. 链表的中间结点 🟢

- 剑指 Offer 18. 删除链表的节点 🟢
- 剑指 Offer 22. 链表中倒数第k个节点 🟢
- 剑指 Offer 25. 合并两个排序的链表 🟢
- 剑指 Offer 52. 两个链表的第一个公共节点 🟢
- 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
- 剑指 Offer II 022. 链表中环的入口节点 🟠
- 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
- 剑指 Offer II 078. 合并排序链表 🔴

876. 链表的中间结点

LeetCode	力扣	难度
876. Middle of the Linked List	876. 链表的中间结点	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，链表，链表双指针

给定一个头结点为 head 的非空单链表，返回链表的中间结点；如果有两个中间结点，则返回第二个中间结点。

示例 1：

```
输入：[1,2,3,4,5]
输出：此列表中的结点 3（序列化形式：[3,4,5]）
返回的结点值为 3。（测评系统对该结点序列化表述是 [3,4,5]）。
注意，我们返回了一个 ListNode 类型的对象 ans，这样：
ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及
ans.next.next.next = NULL.
```

示例 2：

```
输入：[1,2,3,4,5,6]
输出：此列表中的结点 4（序列化形式：[4,5,6]）
由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

PS：这道题在《算法小抄》 的第 64 页。

如果想一次遍历就得到中间节点，也需要耍点小聪明，使用「快慢指针」的技巧：

我们让两个指针 slow 和 fast 分别指向链表头结点 head。

每当慢指针 slow 前进一步，快指针 fast 就前进两步，这样，当 fast 走到链表末尾时，slow 就指向了链表中点。

- 详细题解：双指针技巧秒杀七道链表题目

解法代码

```
class Solution {
    public ListNode middleNode(ListNode head) {
        // 快慢指针初始化指向 head
        ListNode slow = head, fast = head;
        // 快指针走到末尾时停止
        while (fast != null && fast.next != null) {
            // 慢指针走一步，快指针走两步
            slow = slow.next;
            fast = fast.next.next;
        }
        // 慢指针指向中点
        return slow;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- 类似题目：
  - 141. 环形链表 🟢
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 21. 合并两个有序链表 🟢
  - 23. 合并K个升序链表 🔴
  - 86. 分隔链表 🟠
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 078. 合并排序链表 🔴

剑指 Offer 25. 合并两个排序的链表

这道题和 21. 合并两个有序链表 相同。

剑指 Offer 52. 两个链表的第一个公共节点

这道题和 160. 相交链表 相同。

剑指 Offer II 021. 删除链表的倒数第 n 个结点

这道题和 19. 删除链表的倒数第 N 个结点 相同。

剑指 Offer II 022. 链表中环的入口节点

这道题和 142. 环形链表 II 相同。

剑指 Offer II 023. 两个链表的第一个重合节点

这道题和 160. 相交链表 相同。

剑指 Offer II 078. 合并排序链表

这道题和 23. 合并 K 个升序链表 相同。

1305. 两棵二叉搜索树中的所有元素

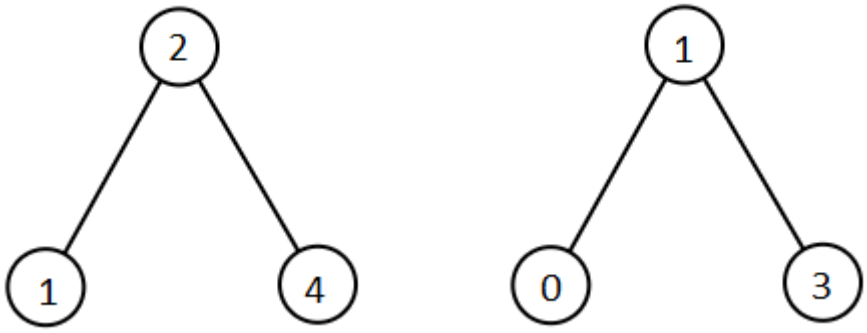
LeetCode	力扣	难度
1305. All Elements in Two Binary Search Trees	1305. 两棵二叉搜索树中的所有元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树vip

给你 `root1` 和 `root2` 这两棵二叉搜索树。请你返回一个列表，其中包含两棵树中的所有整数并按**升序**排序。

示例 1：




```
输入：root1 = [2,1,4], root2 = [1,0,3]
输出：[0,1,1,2,3,4]
```

基本思路

你可以直接中序遍历两个 BST 得到两个有序数组，然后把这两个有序数组合并，这个思路简单，但是空间复杂度会高一些。

比较好的办法是用 173. 二叉搜索树迭代器 中实现的 BST 迭代器，然后类似我们解决 21. 合并两个有序链表 中的逻辑操作这两个迭代器，获得合并的有序结果。

解法代码

```
class Solution {
    public List<Integer> getAllElements(TreeNode root1, TreeNode root2) {
        // BST 有序迭代器
        BSTIterator t1 = new BSTIterator(root1);
        BSTIterator t2 = new BSTIterator(root2);
        LinkedList<Integer> res = new LinkedList<>();
        // 类似合并有序链表的算法逻辑
```

```
        while (t1.hasNext() && t2.hasNext()) {
            if (t1.peek() > t2.peek()) {
                res.add(t2.next());
            } else {
                res.add(t1.next());
            }
        }
        // 如果有一棵 BST 还剩元素，添加到最后
        while (t1.hasNext()) {
            res.add(t1.next());
        }
        while (t2.hasNext()) {
            res.add(t2.next());
        }
        return res;
    }

}

// BST 有序迭代器
class BSTIterator {

    Stack<TreeNode> stk = new Stack<>();

    // 左侧树枝一撸到底
    private void pushLeftBranch(TreeNode p) {
        while (p != null) {
            stk.push(p);
            p = p.left;
        }
    }

    public BSTIterator(TreeNode root) {
        pushLeftBranch(root);
    }

    public int peek() {
        return stk.peek().val;
    }

    public int next() {
        TreeNode p = stk.pop();
        pushLeftBranch(p.right);
        return p.val;
    }

    public boolean hasNext() {
        return !stk.isEmpty();
    }
}
```

264. 丑数 II

LeetCode	力扣	难度
264. Ugly Number II	264. 丑数 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学，链表双指针**

给你一个整数 n，请你找出并返回第 n 个 **丑数**。**丑数** 就是只包含质因数 2、3 和/或 5 的正整数。

示例 1：

```
输入：n = 10
输出：12
解释：[1, 2, 3, 4, 5, 6, 8, 9, 10, 12] 是由前 10 个丑数组成的序列。
```

基本思路

这道题很精妙，你看着它好像是道数学题，实际上它却是一个合并多个有序链表的问题，同时用到了筛选素数的思路。

建议你先做一下 链表双指针技巧汇总 中讲到的 21. 合并两个有序链表（简单），然后再做一下 如何高效寻找素数 中讲的 204. 计数质数（简单），这样的话就能比较容易理解这道题的思路了。

类似 如何高效寻找素数 的思路：如果一个数 `x` 是丑数，那么 `x * 2, x * 3, x * 5` 都一定是丑数。

我们把所有丑数想象成一个从小到大排序的链表，就是这个样子：

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 8 -> ...
```

然后，我们可以把丑数分为三类：2 的倍数、3 的倍数、5 的倍数（按照题目的意思，1 算作特殊的丑数，放在开头），这三类丑数就好像三条有序链表，如下：

能被 2 整除的丑数：

```
1 -> 1*2 -> 2*2 -> 3*2 -> 4*2 -> 5*2 -> 6*2 -> 8*2 ->...
```

能被 3 整除的丑数：

```
1 -> 1*3 -> 2*3 -> 3*3 -> 4*3 -> 5*3 -> 6*3 -> 8*3 ->...
```

能被 5 整除的丑数：

```
1 -> 1*5 -> 2*5 -> 3*5 -> 4*5 -> 5*5 -> 6*5 -> 8*5 ->...
```

我们其实就是想把这三条「有序链表」合并在一起并去重，合并的结果就是丑数的序列。然后求合并后的这条有序链表中第 n 个元素是什么。所以这里就和 链表双指针技巧汇总 中讲到的合并 k 条有序链表的思路基本一样了。

具体思路看注释吧，你也可以对照我在 21. 合并两个有序链表（简单） 中给出的思路代码来看本题的思路代码，就能轻松看懂本题的解法代码了。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
class Solution {
    public int nthUglyNumber(int n) {
        // 可以理解为三个指向有序链表头结点的指针
        int p2 = 1, p3 = 1, p5 = 1;
        // 可以理解为三个有序链表的头节点的值
        int product2 = 1, product3 = 1, product5 = 1;
        // 可以理解为最终合并的有序链表（结果链表）
        int[] ugly = new int[n + 1];
        // 可以理解为结果链表上的指针
        int p = 1;

        // 开始合并三个有序链表
        while (p <= n) {
            // 取三个链表的最小结点
            int min = Math.min(Math.min(product2, product3), product5);
            // 接到结果链表上
            ugly[p] = min;
            p++;
            // 前进对应有序链表上的指针
            if (min == product2) {
                product2 = 2 * ugly[p2];
                p2++;
            }
            if (min == product3) {
                product3 = 3 * ugly[p3];
                p3++;
            }
            if (min == product5) {
                product5 = 5 * ugly[p5];
                p5++;
            }
        }
        // 返回第 n 个丑数
        return ugly[n];
```

```
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 1201. 丑数 III 🟠
  - 263. 丑数 🟢
  - 313. 超级丑数 🟠
  - 剑指 Offer 49. 丑数 🟠

313. 超级丑数

LeetCode	力扣	难度
313. Super Ugly Number	313. 超级丑数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉堆，链表双指针

超级丑数 是一个正整数，并满足其所有质因数都出现在质数数组 `primes` 中。给你一个整数 `n` 和一个整数数组 `primes`，返回第 `n` 个 **超级丑数**。

示例 1：

```
输入：n = 12, primes = [2,7,13,19]
输出：32
解释：给定长度为 4 的质数数组 primes = [2,7,13,19]，前 12 个超级丑数序列为：
[1,2,4,7,8,13,14,16,19,26,28,32]。
```

基本思路

这题是 264. 丑数 II 的进阶版，第 264 题是 21. 合并两个有序链表（简单） 的变体，而这道题是 23. 合并K个升序链表（困难） 的变体，我在 双指针技巧秒杀七道链表题目 中都讲过。

你一定要先做下 264 题，注意那里我们抽象出了三条链表，需要 `p2, p3, p5` 作为三条有序链表上的指针，同时需要 `product2, product3, product5` 记录指针所指节点的值，用 `min` 函数计算最小头结点。

这道题相当于输入了 `len(primes)` 条有序链表，我们不能用 `min` 函数计算最小头结点了，而是要用优先级队列来计算最小头结点，同时依然要维护链表指针、指针所指节点的值，我们把这些信息用一个三元组来保存。

结合第 23 题的解法逻辑，你应该能够看懂这道题的解法代码了。

- 详细题解：一文秒杀所有丑数系列问题

解法代码

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        // 优先队列中装三元组 int[] {product, prime, pi}
        // 其中 product 代表链表节点的值，prime 是计算下一个节点所需的质数因子，pi
代表链表上的指针
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            return a[0] - b[0];
        });
```

```
        // 把多条链表的头结点加入优先级队列
        for (int i = 0; i < primes.length; i++) {
            pq.offer(new int[]{ 1, primes[i], 1 });
        }

        // 可以理解为最终合并的有序链表（结果链表）
        int[] ugly = new int[n + 1];
        // 可以理解为结果链表上的指针
        int p = 1;

        while (p <= n) {
            // 取三个链表的最小结点
            int[] pair = pq.poll();
            int product = pair[0];
            int prime = pair[1];
            int index = pair[2];

            // 避免结果链表出现重复元素
            if (product != ugly[p - 1]) {
                // 接到结果链表上
                ugly[p] = product;
                p++;
            }

            // 生成下一个节点加入优先级队列
            int[] nextPair = new int[]{ugly[index] * prime, prime, index +
1};
            pq.offer(nextPair);
        }
        return ugly[n];
    }
}
```

- **类似题目**：
  - 1201. 丑数 III 🟠
  - 263. 丑数 🟢
  - 264. 丑数 II 🟠
  - 剑指 Offer 49. 丑数 🟠

88. 合并两个有序数组

LeetCode	力扣	难度
88. Merge Sorted Array	88. 合并两个有序数组	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，数组双指针

给你两个按 **非递减顺序** 排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。

请你 **合并** nums2 到 nums1 中，使合并后的数组同样按 **非递减顺序**排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 nums1 中。为了应对这种情况，nums1 的初始长度为 m + n，其中前 m 个元素表示应合并的元素，后 n 个元素为 0，应忽略。nums2 的长度为 n。

示例 1：

```
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]
解释：需要合并 [1,2,3] 和 [2,5,6]。
合并结果是 [1,2,2,3,5,6]，其中斜体加粗标注的为 nums1 中的元素。
```

基本思路

这道题很像前文 链表的双指针技巧汇总 中讲过的 21. 合并两个有序链表，这里让你合并两个有序数组。

对于单链表来说，我们直接用双指针从头开始合并即可，但对于数组来说会出问题。因为题目让我直接把结果存到 nums1 中，而 nums1 的开头有元素，如果我们无脑复制单链表的逻辑，会覆盖掉 nums1 的原始元素，导致错误。

但 nums1 后面是空的呀，所以这道题需要我们稍微变通一下：**将双指针初始化在数组的尾部，然后从后向前进行合并**，这样即便覆盖了 nums1 中的元素，这些元素也必然早就被用过了，不会影响答案的正确性。

解法代码

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        // 两个指针分别初始化在两个数组的最后一个元素（类似拉链两端的锯齿）
        int i = m - 1, j = n - 1;
        // 生成排序的结果（类似拉链的拉锁）
        int p = nums1.length - 1;
        // 从后向前生成结果数组，类似合并两个有序链表的逻辑
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
```

```
                nums1[p] = nums1[i];
                i--;
            } else {
                nums1[p] = nums2[j];
                j--;
            }
            p--;
        }
        // 可能其中一个数组的指针走到尽头了，而另一个还没走完
        // 因为我们本身就是在往 nums1 中放元素，所以只需考虑 nums2 是否剩元素即可
        while (j >= 0) {
            nums1[p] = nums2[j];
            j--;
            p--;
        }
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 977. 有序数组的平方 🟢

97. 交错字符串

LeetCode	力扣	难度
97. Interleaving String	97. 交错字符串	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：dpvip，动态规划，双指针

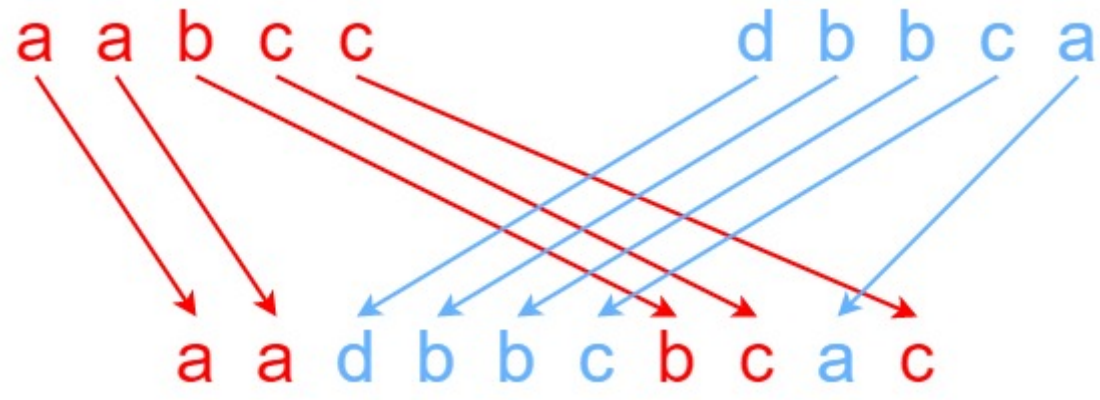
给定三个字符串 `s1, s2, s3`，请你帮忙验证 `s3` 是否是由 `s1` 和 `s2` **交错**组成的。

两个字符串 `s` 和 `t` **交错**的定义与过程如下，其中每个字符串都会被分割成若干**非空**子字符串：

- `s = s1 + s2 + ... + sn`
- `t = t1 + t2 + ... + tm`
- `|n - m| <= 1`
- **交错** 是 `s1 + t1 + s2 + t2 + s3 + t3 + ...` 或者 `t1 + s1 + t2 + s2 + t3 + s3 + ...`

注意： `a + b` 意味着字符串 `a` 和 `b` 连接。

示例 1：




```
输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
输出：true
```

基本思路

如果你看过前文 单链表六大解题套路 中讲解的 21. 合并两个有序链表 就会发现，题目巴拉巴拉说了一大堆，**实则就是一个使用双指针技巧合并两个字符串的过程**。

双指针的大致逻辑如下：

```
int i = 0, j = 0;
for (int k = 0; k < s3.length; k++) {
```

```
        if (s1[i] == s3[k]) {
            i++;
        } else if (s2[j] == s3[k]) {
            j++;
        }
    }
    assert i == s1.length() && j == s2.length();
```

但本题跟普通的数组/链表双指针技巧不同的是，这里需要穷举所有情况。比如 `s1[i], s2[j]` 都能匹配 `s3[k]` 的时候，到底应该让谁来匹配，才能完全合并出 `s3` 呢？

回答这个问题很简单，我不知道让谁来，那就都来试一遍好了，前文 经典动态规划：最长公共子序列 和 经典动态规划：编辑距离 都处理过类似的情况。

所以本题肯定需要一个递归函数来穷举双指针的匹配过程，然后用一个备忘录消除递归过程中的重叠子问题，也就能写出自顶向下的递归的动态规划解法了。

解法代码

```
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {
        int m = s1.length(), n = s2.length();
        // 如果长度对不上，必然不可能
        if (m + n != s3.length()) {
            return false;
        }
        // 备忘录，其中 -1 代表未计算，0 代表 false，1 代表 true
        memo = new int[m + 1][n + 1];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }

        return dp(s1, 0, s2, 0, s3);
    }

    int[][] memo;

    // 定义：计算 s1[i..] 和 s2[j..] 是否能组合出 s3[i+j..]
    boolean dp(String s1, int i, String s2, int j, String s3) {
        int k = i + j;
        // base case，s3 构造完成
        if (k == s3.length()) {
            return true;
        }
        // 查备忘录，如果已经计算过，直接返回
        if (memo[i][j] != -1) {
            return memo[i][j] == 1 ? true : false;
        }

        boolean res = false;
        // 如果，s1[i] 可以匹配 s3[k]，那么填入 s1[i] 试一下
```

```
        if (i < s1.length() && s1.charAt(i) == s3.charAt(k)) {
            res = dp(s1, i + 1, s2, j, s3);
        }
        // 如果，s1[i] 匹配不了，s2[j] 可以匹配，那么填入 s2[j] 试一下
        if (j < s2.length() && s2.charAt(j) == s3.charAt(k)) {
            res = res || dp(s1, i, s2, j + 1, s3);
        }
        // 如果 s1[i] 和 s2[j] 都匹配不了，则返回 false
        // 将结果存入备忘录
        memo[i][j] = res == true ? 1 : 0;

        return res;
    }
}
```

977. 有序数组的平方

LeetCode	力扣	难度
977. Squares of a Sorted Array	977. 有序数组的平方	🟢

- 标签：数组双指针

给你一个按**非递减顺序**排序的整数数组 nums，返回**每个数字的平方**组成的新数组，要求也按**非递减顺序**排序。

示例 1：

```
输入：nums = [-4,-1,0,3,10]
输出：[0,1,9,16,100]
解释：平方后，数组变为 [16,1,0,9,100]
排序后，数组变为 [0,1,9,16,100]
```

基本思路

平方的特点是会把负数变成正数，所以一个负数和一个正数平方后的大小要根据绝对值来比较。

可以把元素 0 作为分界线，0 左侧的负数是一个有序数组 nums1，0 右侧的正数是另一个有序数组 nums2，那么这道题就和 88. 合并两个有序数组 差不多，在进一步就是前文 链表的双指针技巧汇总 讲过的 21. 合并两个有序链表 的变体。

所以，我们可以去寻找正负数的分界点，然后向左右扩展，执行合并有序数组的逻辑。不过还有个更好的办法，不用找正负分界点，而是直接将双指针分别初始化在 nums 的开头和结尾，相当于合并两个从大到小排序的数组，和 88 题类似。有了思路，直接看代码吧。

解法代码

```
class Solution {
    public int[] sortedSquares(int[] nums) {
        int n = nums.length;
        // 两个指针分别初始化在正负子数组绝对值最大的元素索引
        int i = 0, j = n - 1;
        // 得到的有序结果是降序的
        int p = n - 1;
        int[] res = new int[n];
        // 执行双指针合并有序数组的逻辑
        while (i <= j) {
            if (Math.abs(nums[i]) > Math.abs(nums[j])) {
                res[p] = nums[i] * nums[i];
                i++;
```

```
                } else {
                    res[p] = nums[j] * nums[j];
                    j--;
                }
                p--;
            }
            return res;
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 360. 有序转化数组 🟠

剑指 Offer 49. 丑数

这道题和 264. 丑数 II 相同。

剑指 Offer 18. 删除链表的节点

LeetCode	力扣	难度
剑指Offer18. 删除链表的节点 LCOF	剑指Offer18. 删除链表的节点	🟢

- 标签：链表双指针

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。返回删除后的链表的头节点。

示例 1:

```
输入：head = [4,5,1,9], val = 5
输出：[4,1,9]
解释：给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

基本思路

这道题常规的思路是通过操作指针来删除值为 `val` 的节点，不过处理起来可能稍微有点麻烦，我们可以使用链表双指针技巧汇总 中讲到的分解链表的思路来解决这道题（你可以先去做下 86. 分隔链表）。

你可以认为这道题是把原链表分解成「值为 `val`」和「值不为 `val`」的两条链表，就可以复用 86 题的思路了。

解法代码

```
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        // 存放删除 val 的链表
        ListNode dummy = new ListNode(-1);
        // q 指针负责生成结果链表
        ListNode q = dummy;
        // p 负责遍历原链表
        ListNode p = head;
        while (p != null) {
            if (p.val != val) {
                // 把值不为 val 的节点接到结果链表上
                q.next = p;
                q = q.next;
            }
            // 断开原链表中的每个节点的 next 指针
            ListNode temp = p.next;
            p.next = null;
            p = temp;
```

```
            }

            return dummy.next;
        }
    }
```

360. 有序转化数组

LeetCode	力扣	难度
360. Sort Transformed Array	360. 有序转化数组	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：dsvip，数据结构，数组双指针

给你一个已经**排好序**的整数数组 `nums` 和整数 `a, b, c`。对于数组中的每一个元素 `nums[i]`，计算函数值 `f(x) = ax2 + bx + c`，请按升序返回结果数组。

示例 1：

```
输入：nums = [-4,-2,2,4], a = 1, b = 3, c = 5
输出：[3,9,15,33]
```

基本思路

只要看过前文 链表的双指针技巧汇总 并做过 977. 有序数组的平方，应该有这道题的思路。

977 题其实就是这道题中 `a = 1, b = 0, c = 0` 的特殊情况，**所以这道题的关键也是在 `nums` 的开头和结尾设置 `i, j` 双指针相向而行，执行合并有序数组的逻辑**，只不过这里需要考虑的情况更多了一些罢了。

我们中学都学过这种二次函数，图像是一个抛物线，写个函数来表示：

```
int f(int x, int a, int b, int c) {
    return a*x*x + b*x + c;
}
```

`nums[i]` 就好像坐标系中 x 轴坐标，那么 `f(nums[i])` 之间的关系就取决于抛物线的对称轴位置以及抛物线的开口方向（`a` 的正负）。

如果 `nums` 中的元素全都落在抛物线的一侧，则这些元素本身就是有序递增或递减的，根据开口方向做判断就可以了，很容易处理。

关键是 `nums` 中的元素分布在在抛物线的两侧的情况，这就和 977 题的场景有些像，所以需要设置 `i, j` 双指针执行合并两个有序数组的逻辑了，当然还要考虑抛物线开口的方向。

有了上述思路，直接看代码吧。

解法代码

```
class Solution {
    public int[] sortTransformedArray(int[] nums, int a, int b, int c) {
        // 双指针，相向而行，逼近对称轴
        int i = 0, j = nums.length - 1;
        // 如果开口朝上，越靠近对称轴函数值越小
        // 如果开口朝下，越靠近对称轴函数值越大
        int p = a > 0 ? nums.length - 1 : 0;
        int[] res = new int[nums.length];
        // 执行合并两个有序数组的逻辑
        while (i <= j) {
            int v1 = f(nums[i], a, b, c);
            int v2 = f(nums[j], a, b, c);
            if (a > 0) {
                // 如果开口朝上，越靠近对称轴函数值越小
                if (v1 > v2) {
                    res[p--] = v1;
                    i++;
                } else {
                    res[p--] = v2;
                    j--;
                }
            } else {
                // 如果开口朝下，越靠近对称轴函数值越大
                if (v1 > v2) {
                    res[p++] = v2;
                    j--;
                } else {
                    res[p++] = v1;
                    i++;
                }
            }
        }
        return res;
    }

    int f(int x, int a, int b, int c) {
        return a*x*x + b*x + c;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

355. 设计推特

LeetCode	力扣	难度
355. Design Twitter	355. 设计推特	🟠

- 标签：数据结构，设计

设计一个简化版的推特 (Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近 10 条推文。

实现 Twitter 类：

- void postTweet(int userId, int tweetId) 根据给定的 tweetId 和 userId 创建一条新推文。每次调用此函数都会使用一个不同的 tweetId。
- List<Integer> getNewsFeed(int userId) 检索当前用户新闻推送中最近 10 条推文的 ID。新闻推送中的每一项都必须是由用户关注的人或者是用户自己发布的推文。推文必须**按照时间顺序由最近到最远排序**。
- void follow(int followerId, int followeeId) ID 为 followerId 的用户开始关注 ID 为 followeeId 的用户。
- void unfollow(int followerId, int followeeId) ID 为 followerId 的用户不再关注 ID 为 followeeId 的用户。

示例：

```
输入
["Twitter", "postTweet", "getNewsFeed", "follow", "postTweet",
"getNewsFeed", "unfollow", "getNewsFeed"]
[[], [1, 5], [1], [1, 2], [2, 6], [1], [1, 2], [1]]
输出
[null, null, [5], null, null, [6, 5], null, [5]]

解释
Twitter twitter = new Twitter();
twitter.postTweet(1, 5); // 用户 1 发送了一条新推文 (用户 id = 1, 推文 id = 5)
twitter.getNewsFeed(1);  // 用户 1 的获取推文应当返回一个列表，其中包含一个 id 为
5 的推文
twitter.follow(1, 2);    // 用户 1 关注了用户 2
twitter.postTweet(2, 6); // 用户 2 发送了一个新推文 (推文 id = 6)
twitter.getNewsFeed(1);  // 用户 1 的获取推文应当返回一个列表，其中包含两个推文，id
分别为 -> [6, 5]。推文 id 6 应当在推文 id 5 之前，因为它是在 5 之后发送的
twitter.unfollow(1, 2);  // 用户 1 取消关注了用户 2
twitter.getNewsFeed(1);  // 用户 1 获取推文应当返回一个列表，其中包含一个 id 为 5
的推文。因为用户 1 已经不再关注用户 2
```

基本思路

这道题比较经典，在特定场景下让你设计算法。其难点在于 `getNewsFeed` 方法，要按照时间线顺序展示所有关注用户的推文，这个方法要用到我在 单链表的六大解题套路 解决 23. 合并K个升序链表 的合并多个有序链表的技巧：

你把一个用户发布的所有推文做成一条有序链表（靠近头部的推文是最近发布的），那么只要合并关注用户的推文链表，即可获得按照时间线顺序排序的信息流。

具体看代码吧，我注释比较详细。

- **详细题解：设计朋友圈时间线功能**

解法代码

```
class Twitter {
    // 全局时间戳
    int globalTime = 0;
    // 记录用户 ID 到用户示例的映射
    HashMap<Integer, User> idToUser = new HashMap<>();

    // Tweet 类
    class Tweet {
        private int id;
        // 时间戳用于对信息流按照时间排序
        private int timestamp;
        // 指向下一条 tweet，类似单链表结构
        private Tweet next;

        public Tweet(int id) {
            this.id = id;
            // 新建一条 tweet 时记录并更新时间戳
            this.timestamp = globalTime++;
        }

        public int getId() {
            return id;
        }

        public int getTimestamp() {
            return timestamp;
        }

        public Tweet getNext() {
            return next;
        }

        public void setNext(Tweet next) {
            this.next = next;
        }
    }
```

```
    // 用户类
    class User {
        // 记录该用户的 id 以及发布的 tweet
        private int id;
        private Tweet tweetHead;
        // 记录该用户的关注者
        private HashSet<User> followedUserSet;

        public User(int id) {
            this.id = id;
            this.tweetHead = null;
            this.followedUserSet = new HashSet<>();
        }

        public int getId() {
            return id;
        }

        public Tweet getTweetHead() {
            return tweetHead;
        }

        public HashSet<User> getFollowedUserSet() {
            return followedUserSet;
        }

        public boolean equals(User other) {
            return this.id == other.id;
        }

        // 关注其他人
        public void follow(User other) {
            followedUserSet.add(other);
        }

        // 取关其他人
        public void unfollow(User other) {
            followedUserSet.remove(other);
        }

        // 发布一条 tweet
        public void post(Tweet tweet) {
            // 把新发布的 tweet 作为链表头节点
            tweet.setNext(tweetHead);
            tweetHead = tweet;
        }
    }

    public void postTweet(int userId, int tweetId) {
        // 如果这个用户还不存在，新建用户
        if (!idToUser.containsKey(userId)) {
            idToUser.put(userId, new User(userId));
        }
        User user = idToUser.get(userId);
```

```
        user.post(new Tweet(tweetId));
    }

    public List<Integer> getNewsFeed(int userId) {
        List<Integer> res = new LinkedList<>();
        if (!idToUser.containsKey(userId)) {
            return res;
        }
        // 获取该用户关注的用户列表
        User user = idToUser.get(userId);
        Set<User> followedUserSet = user.getFollowedUserSet();
        // 每个用户的 tweet 是一条按时间排序的链表
        // 现在执行合并多条有序链表的逻辑，找出时间线中的最近 10 条动态
        PriorityQueue<Tweet> pq = new PriorityQueue<>((a, b) -> {
            // 按照每条 tweet 的发布时间降序排序（最近发布的排在事件流前面）
            return b.timestamp - a.timestamp;
        });
        // 该用户自己的 tweet 也在时间线内
        if (user.getTweetHead() != null) {
            pq.offer(user.getTweetHead());
        }
        for (User other : followedUserSet) {
            if (other.getTweetHead() != null) {
                pq.offer(other.tweetHead);
            }
        }
        // 合并多条有序链表
        int count = 0;
        while (!pq.isEmpty() && count < 10) {
            Tweet tweet = pq.poll();
            res.add(tweet.getId());
            if (tweet.getNext() != null) {
                pq.offer(tweet.getNext());
            }
            count++;
        }
        return res;
    }

    public void follow(int followerId, int followeeId) {
        // 如果用户还不存在，则新建用户
        if (!idToUser.containsKey(followerId)) {
            idToUser.put(followerId, new User(followerId));
        }
        if (!idToUser.containsKey(followeeId)) {
            idToUser.put(followeeId, new User(followeeId));
        }

        User follower = idToUser.get(followerId);
        User followee = idToUser.get(followeeId);
        // 关注者关注被关注者
        follower.follow(followee);
    }
```

```
    public void unfollow(int followerId, int followeeId) {
        if (!idToUser.containsKey(followerId) ||
!idToUser.containsKey(followeeId)) {
            return;
        }
        User follower = idToUser.get(followerId);
        User followee = idToUser.get(followeeId);
        // 关注者取关被关注者
        follower.unfollow(followee);
    }
}
```

373. 查找和最小的 K 对数字

LeetCode	力扣	难度
373. Find K Pairs with Smallest Sums	373. 查找和最小的K对数字	🟠

- **标签：二叉堆，链表双指针**

给定两个以 **升序排列** 的整数数组 `nums1` 和 `nums2`, 以及一个整数 `k`。

定义一对值 `(u,v)`，其中第一个元素来自 `nums1`，第二个元素来自 `nums2`。

请找到**和**最小的 `k` 个数对 `(u1,v1)`, `(u2,v2)` ... `(uk,vk)`。

示例 1:

```
输入: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
输出: [1,2],[1,4],[1,6]
解释: 返回序列中的前 3 对数:
     [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

基本思路

这道题其实是前文 单链表的六大解题套路 中讲过的 23. 合并K个升序链表 的变体。

怎么把这道题变成合并多个有序链表呢？就比如说题目输入的用例：

```
nums1 = [1,7,11], nums2 = [2,4,6]
```

组合出的所有数对儿这就可以抽象成三个有序链表：

```
[1, 2] -> [1, 4] -> [1, 6]
[7, 2] -> [7, 4] -> [7, 6]
[11, 2] -> [11, 4] -> [11, 6]
```

这三个链表中每个元素（数对之和）是递增的，所以就可以按照 23. 合并K个升序链表 的思路来合并，取出前 `k` 个作为答案即可。

解法代码

```
class Solution {
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2,
int k) {
        // 存储三元组 (num1[i], nums2[i], i)
        // i 记录 nums2 元素的索引位置，用于生成下一个节点
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            // 按照数对的元素和升序排序
            return (a[0] + a[1]) - (b[0] + b[1]);
        });
        // 按照 23 题的逻辑初始化优先级队列
        for (int i = 0; i < nums1.length; i++) {
            pq.offer(new int[]{nums1[i], nums2[0], 0});
        }

        List<List<Integer>> res = new ArrayList<>();
        // 执行合并多个有序链表的逻辑
        while (!pq.isEmpty() && k > 0) {
            int[] cur = pq.poll();
            k--;
            // 链表中的下一个节点加入优先级队列
            int next_index = cur[2] + 1;
            if (next_index < nums2.length) {
                pq.add(new int[]{cur[0], nums2[next_index], next_index});
            }

            List<Integer> pair = new ArrayList<>();
            pair.add(cur[0]);
            pair.add(cur[1]);
            res.add(pair);
        }
        return res;
    }
}
```

- **类似题目**：
  - 剑指 Offer II 061. 和最小的 k 个数对 🟠

378. 有序矩阵中第 K 小的元素

LeetCode	力扣	难度
378. Kth Smallest Element in a Sorted Matrix	378. 有序矩阵中第 K 小的元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，链表双指针**

给你一个 `n x n` 矩阵 `matrix`，其中每行和每列元素均按升序排序，找到矩阵中第 `k` 小的元素。
请注意，它是 **排序后** 的第 `k` 小元素，而不是第 `k` 个 **不同**的元素。

你必须找到一个内存复杂度优于 `O(n2)` 的解决方案。

示例 1：

```
输入：matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
输出：13
解释：矩阵中的元素为 [1,5,9,10,11,12,13,13,15]，第 8 小元素是 13
```

基本思路

这道题其实是前文 单链表的六大解题套路 中讲过的 23. 合并K个升序链表 的变体。

矩阵中的每一行都是排好序的，就好比多条有序链表，你用优先级队列施展合并多条有序链表的逻辑就能找到第 `k` 小的元素了。

解法代码

```
class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        // 存储二元组 (matrix[i][j], i, j)
        // i, j 记录当前元素的索引位置，用于生成下一个节点
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            // 按照元素大小升序排序
            return a[0] - b[0];
        });


        // 初始化优先级队列，把每一行的第一个元素装进去
        for (int i = 0; i < matrix.length; i++) {
            pq.offer(new int[]{matrix[i][0], i, 0});
        }

        int res = -1;
        // 执行合并多个有序链表的逻辑，找到第 k 小的元素
```

```
        while (!pq.isEmpty() && k > 0) {
            int[] cur = pq.poll();
            res = cur[0];
            k--;
            // 链表中的下一个节点加入优先级队列
            int i = cur[1], j = cur[2];
            if (j + 1 < matrix[i].length) {
                pq.add(new int[]{matrix[i][j + 1], i, j + 1});
            }
        }
        return res;
    }
}
```

剑指 Offer II 061. 和最小的 k 个数对

这道题和 373. 查找和最小的 K 对数字 相同。

24. 两两交换链表中的节点

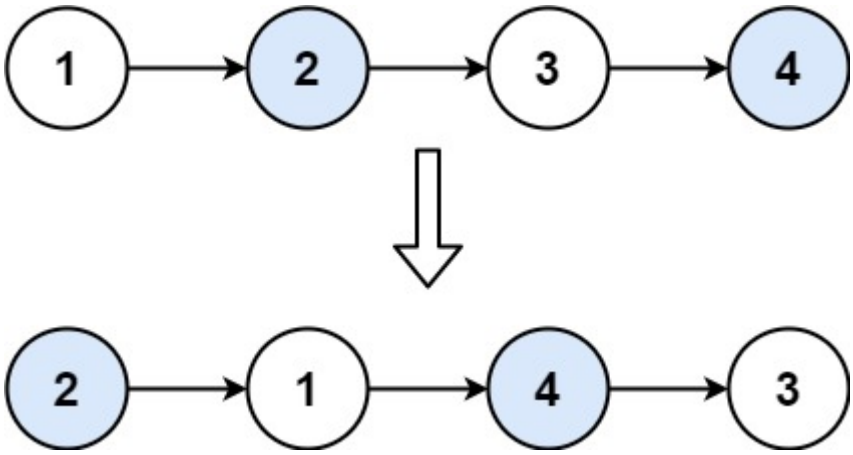
LeetCode	力扣	难度
24. Swap Nodes in Pairs	24. 两两交换链表中的节点	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：单链表，递归**

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（只能进行节点交换）。

示例 1：




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

基本思路

这道题不难，常规方法就是维护多个指针，遍历一遍链表顺便把每两个节点翻转。不过迭代的思路虽然直接，但细节问题会比较多，写起来麻烦。

所以我直接用递归的方式来写，只要搞明白递归函数的定义，然后利用这个定义就可以完成这道题。

其实前文 如何 k 个一组反转链表 中讲过的 25. K 个一组翻转链表 就是这道题的进阶版，你可以去做一做。

解法代码

```
class Solution {
    // 定义：输入以 head 开头的单链表，将这个单链表中的每两个元素翻转，
    // 返回翻转后的链表头结点
    public ListNode swapPairs(ListNode head) {
        if (head == null || head.next == null) {
            return head;
```

```
        }
        ListNode first = head;
        ListNode second = head.next;
        ListNode others = head.next.next;
        // 先把前两个元素翻转
        second.next = first;
        // 利用递归定义，将剩下的链表节点两两翻转，接到后面
        first.next = swapPairs(others);
        // 现在整个链表都成功翻转了，返回新的头结点
        return second;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

25. K 个一组翻转链表

LeetCode	力扣	难度
25. Reverse Nodes in k-Group	25. K 个一组翻转链表	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

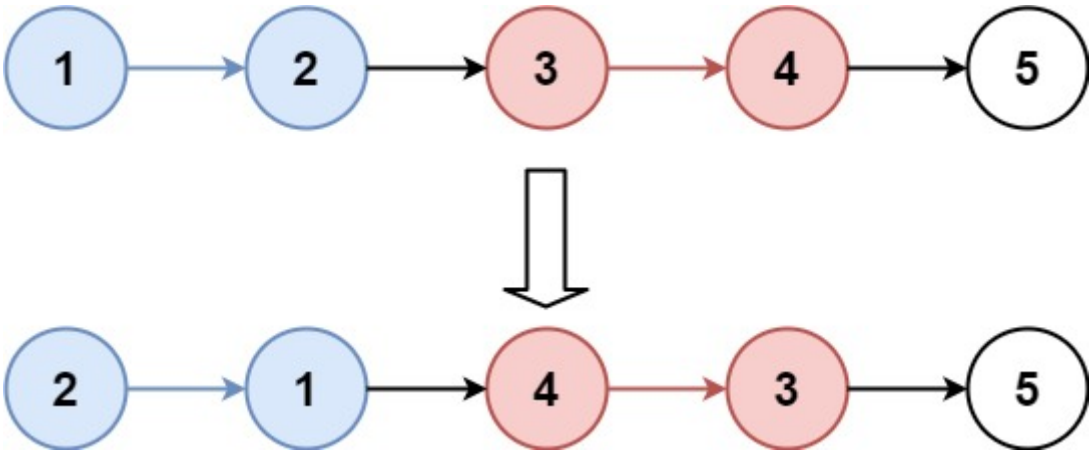
- 标签：数据结构，链表，链表双指针

给你一个链表，请你对每 *k* 个节点一组进行翻转，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。

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。

示例 1：




```
输入：head = [1,2,3,4,5], k = 2
输出：[2,1,4,3,5]
```

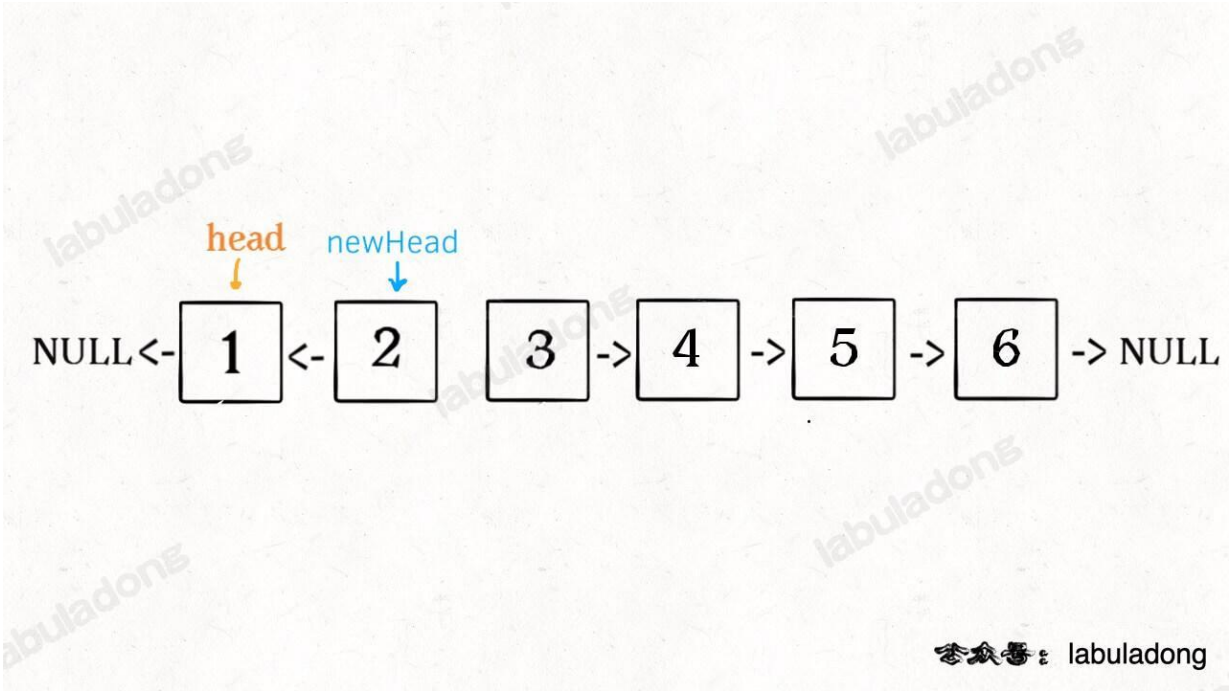
基本思路

PS：这道题在《算法小抄》 的第 289 页。

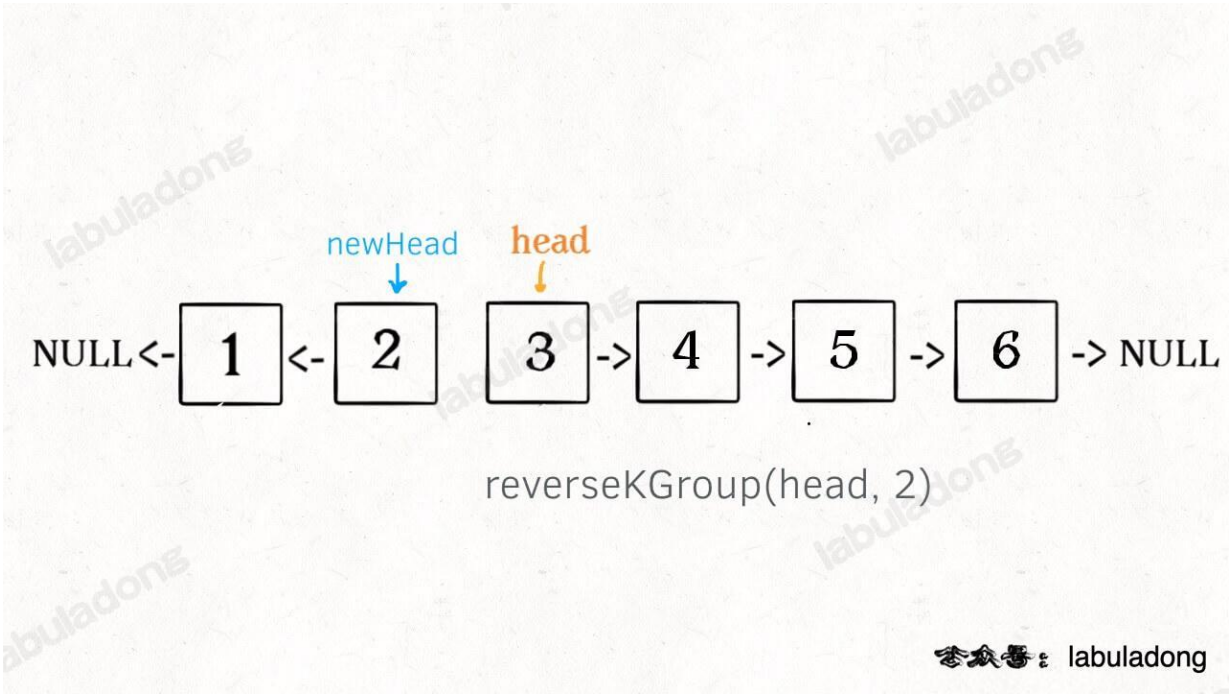
输入 `head`，`reverseKGroup` 函数能够把以 `head` 为头的这条链表进行翻转。

我们要充分利用这个递归函数的定义，把原问题分解成规模更小的子问题进行求解。

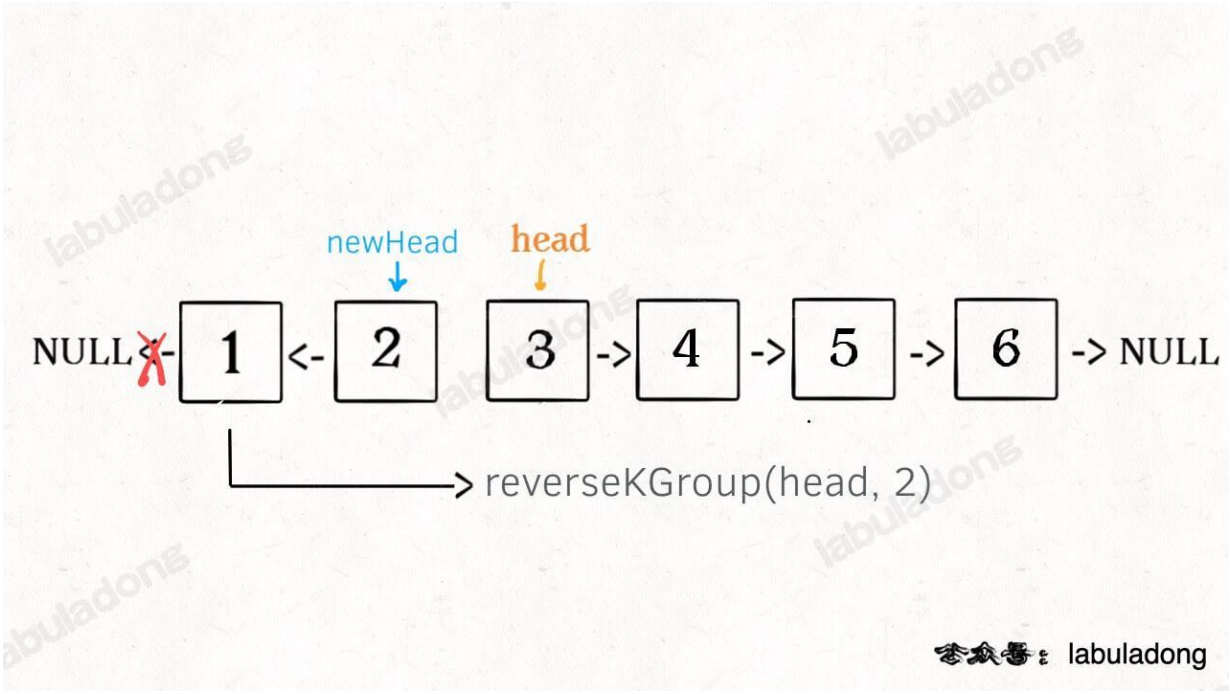
1、先反转以 `head` 开头的 `k` 个元素。



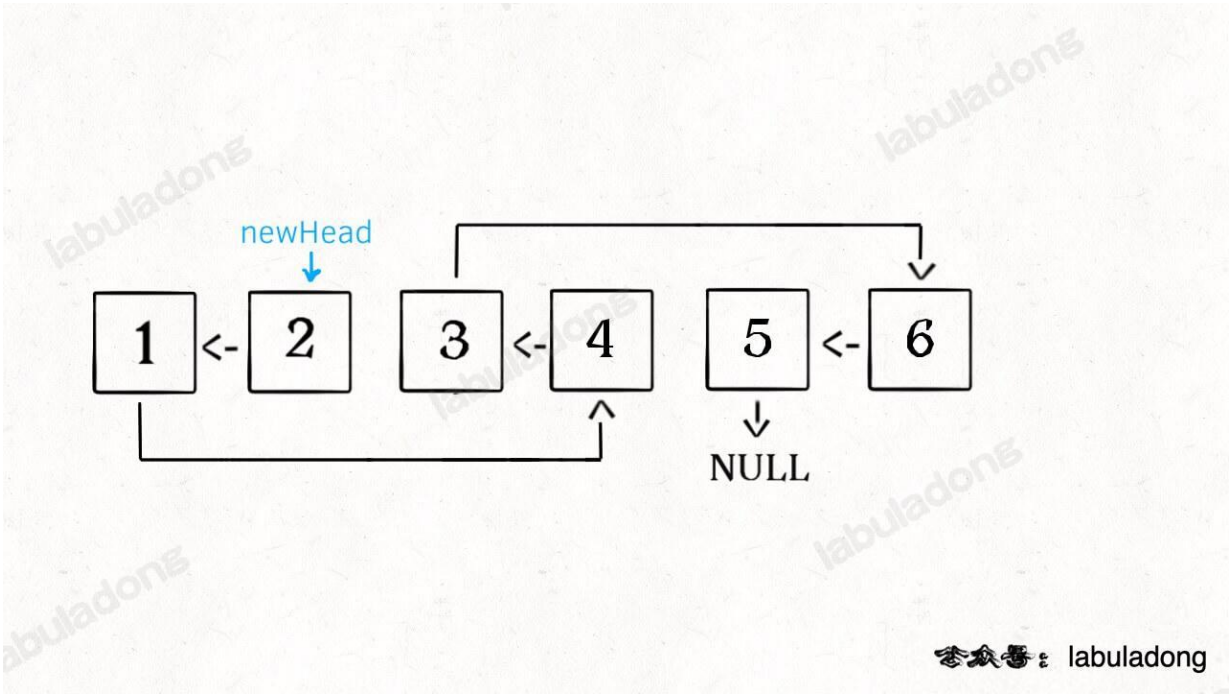
2、将第 `k + 1` 个元素作为 `head` 递归调用 `reverseKGroup` 函数。



3、将上述两个过程的结果连接起来。



最后函数递归完成之后就是这个结果，完全符合题意：



- **详细题解：如何 K 个一组反转链表**

解法代码

```
class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null) return null;
        // 区间 [a, b) 包含 k 个待反转元素
        ListNode a, b;
        a = b = head;
        for (int i = 0; i < k; i++) {
            // 不足 k 个，不需要反转，base case
```

```
            if (b == null) return head;
            b = b.next;
        }
        // 反转前 k 个元素
        ListNode newHead = reverse(a, b);
        // 递归反转后续链表并连接起来
        a.next = reverseKGroup(b, k);
        return newHead;
    }

    /* 反转区间 [a, b) 的元素，注意是左闭右开 */
    ListNode reverse(ListNode a, ListNode b) {
        ListNode pre, cur, nxt;
        pre = null;
        cur = a;
        nxt = a;
        // while 终止的条件改一下就行了
        while (cur != b) {
            nxt = cur.next;
            cur.next = pre;
            pre = cur;
            cur = nxt;
        }
        // 返回反转后的头结点
        return pre;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 24. 两两交换链表中的节点 🟠

82. 删除排序链表中的重复元素 II

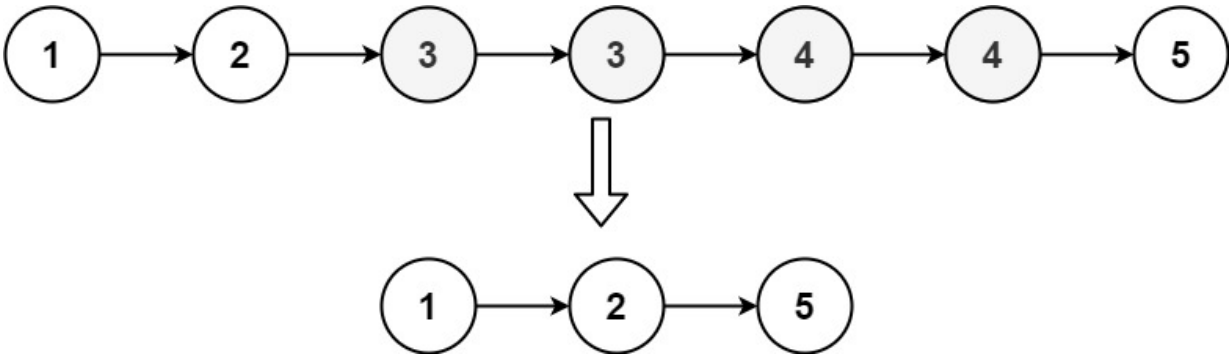
LeetCode	力扣	难度
82. Remove Duplicates from Sorted List II	82. 删除排序链表中的重复元素 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：dsvip，数据结构，链表双指针

给定一个已排序的链表的头 head，删除原始链表中所有重复数字的节点，只留下不同的数字，返回删除之后的排序链表。

示例 1：




```
输入：head = [1,2,3,3,4,4,5]
输出：[1,2,5]
```

基本思路

这道题是前文 链表的双指针技巧汇总 中讲的 83. 删除排序链表中的重复元素 的进阶版。如果只让你把多于的重复元素去掉，那么快慢指针可以搞定，但这道题要求你把存在重复的元素全都去掉，一个简单粗暴的解法就是借助像哈希表这样的数据结构记录哪些节点重复了，然后去掉它们。

不过这道题输入的链表是有序的，这意味着重复元素都靠在一起，其实不用额外的空间复杂度来辅助，用两个指针就可以达到去重的目的，只是细节有点多，直接结合代码的详细注释来看吧。

值得一提的是，这道题也可以用递归思维来做，虽然存在堆栈消耗空间复杂度，不过理解起来更容易，我也写出来供大家参考。

解法代码

```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode dummy = new ListNode(-1);
        ListNode p = dummy, q = head;
        while (q != null) {
```

```
            if (q.next != null && q.val == q.next.val){
                // 发现重复节点，跳过这些重复节点
                while (q.next != null && q.val == q.next.val) {
                    q = q.next;
                }
                q = q.next;
                // 此时 q 跳过了这一段重复元素
                if (q == null) {
                    p.next = null;
                }
                // 不过下一段元素也可能重复，等下一轮 while 循环判断
            } else {
                // 不是重复节点，接到 dummy 后面
                p.next = q;
                p = p.next;
                q = q.next;
            }
        }
        return dummy.next;
    }
}

// 递归解法
class Solution2 {
    // 定义：输入一条单链表头结点，返回去重之后的单链表头结点
    public ListNode deleteDuplicates(ListNode head) {
        // base case
        if (head == null || head.next == null) {
            return head;
        }
        if (head.val != head.next.val) {
            // 如果头结点和身后节点的值不同，则对之后的链表去重即可
            head.next = deleteDuplicates(head.next);
            return head;
        }
        // 如果如果头结点和身后节点的值相同，则说明从 head 开始存在若干重复节点
        // 越过重复节点，找到 head 之后那个不重复的节点
        while (head.next != null && head.val == head.next.val) {
            head = head.next;
        }
        // 直接返回那个不重复节点开头的链表的去重结果，就把重复节点删掉了
        return deleteDuplicates(head.next);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

83. 删除排序链表中的重复元素

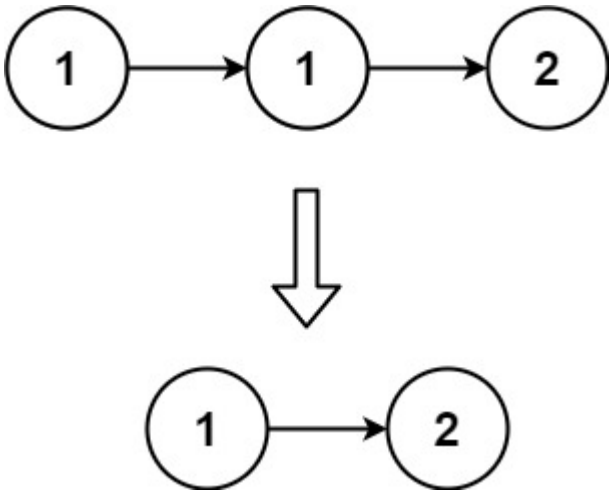
LeetCode	力扣	难度
83. Remove Duplicates from Sorted List	83. 删除排序链表中的重复元素	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：链表，链表双指针

存在一个按升序排列的链表，给你这个链表的头节点 head，请你删除所有重复的元素，使每个元素**只出现一次**，返回同样按升序排列的结果链表。

示例 1：



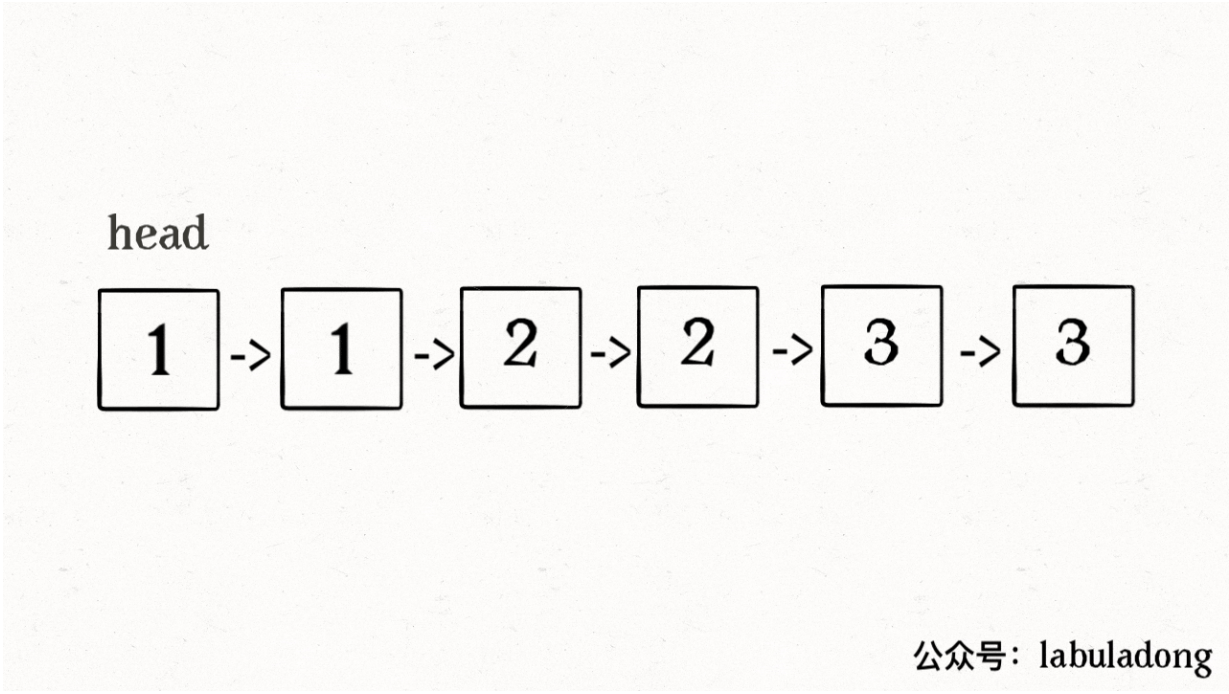

```
输入：head = [1,1,2]
输出：[1,2]
```

基本思路

> 本文有视频版：数组双指针技巧汇总

PS：这道题在《算法小抄》 的第 371 页。

思路和 26. 删除有序数组中的重复项 完全一样，唯一的区别是把数组赋值操作变成操作指针而已。



- 详细题解：双指针技巧秒杀七道数组题目

解法代码

```
class Solution {
    public deleteDuplicates(ListNode head) {
        if (head == null) return null;
        ListNode slow = head, fast = head;
        while (fast != null) {
            if (fast.val != slow.val) {
                // nums[slow] = nums[fast];
                slow.next = fast;
                // slow++;
                slow = slow.next;
            }
            // fast++
            fast = fast.next;
        }
        // 断开与后面重复元素的连接
        slow.next = null;
        return head;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- 类似题目：
  - 167. 两数之和 II - 输入有序数组 🟠
  - 26. 删除有序数组中的重复项 🟢
  - 27. 移除元素 🟢

- 283. 移动零 🟢
- 344. 反转字符串 🟢
- 5. 最长回文子串 🟠
- 82. 删除排序链表中的重复元素 II 🟠
- 剑指 Offer 57. 和为s的两个数字 🟢
- 剑指 Offer II 006. 排序数组中两个数字之和 🟢

92. 反转链表 II

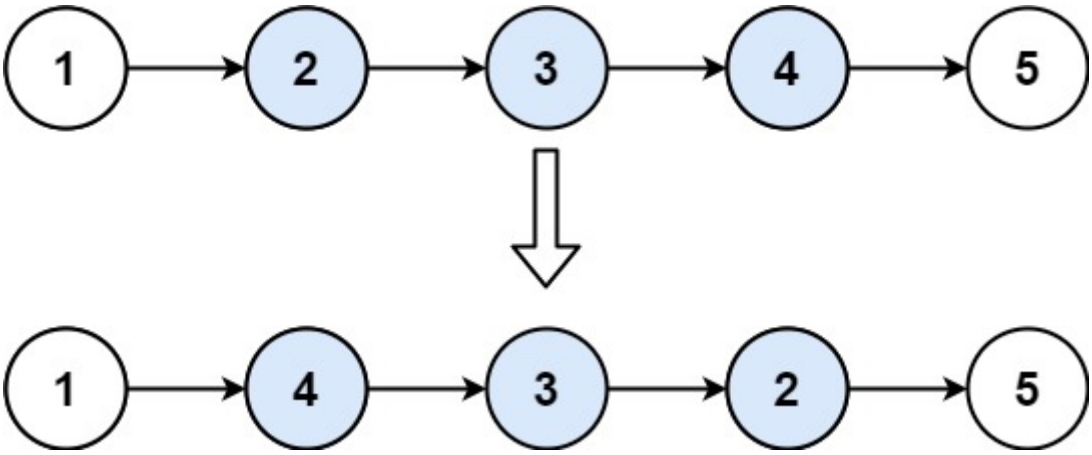
LeetCode	力扣	难度
92. Reverse Linked List II	92. 反转链表 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，递归，链表，链表双指针

给你单链表的头指针 `head` 和两个整数 `left` 和 `right`，其中 `left <= right`，请你反转从位置 `left` 到位置 `right` 的链表节点，返回**反转后的链表**。

示例 1：




```
输入：head = [1,2,3,4,5], left = 2, right = 4
输出：[1,4,3,2,5]
```

基本思路

PS：这道题在《算法小抄》 的第 283 页。

迭代解法很简单，用一个 for 循环即可，但这道题经常用来考察递归思维，让你用纯递归的形式来解决，这里就给出递归解法的思路。

要想真正理解递归操作链表的代码思路，需要从递归翻转整条链表的算法开始，推导出递归翻转前 `N` 个节点的算法，最后改写出递归翻转区间内的节点的解法代码。

关键点还是要明确递归函数的定义，由于内容和图比较多，这里就不写了，请看详细题解。

- 详细题解：递归魔法：反转单链表

解法代码

```java
class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        // base case
        if (m == 1) {
            return reverseN(head, n);
        }
        // 前进到反转的起点触发 base case
        head.next = reverseBetween(head.next, m - 1, n - 1);
        return head;
    }

    ListNode successor = null; // 后驱节点
    // 反转以 head 为起点的 n 个节点，返回新的头结点
    ListNode reverseN(ListNode head, int n) {
        if (n == 1) {
            // 记录第 n + 1 个节点
            successor = head.next;
            return head;
        }
        // 以 head.next 为起点，需要反转前 n - 1 个节点
        ListNode last = reverseN(head.next, n - 1);

        head.next.next = head;
        // 让反转之后的 head 节点和后面的节点连起来
        head.next = successor;
        return last;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 206. 反转链表 🟢
  - 剑指 Offer 24. 反转链表 🟢
  - 剑指 Offer II 024. 反转链表 🟢

1650. 二叉树的最近公共祖先 III

LeetCode	力扣	难度
1650. Lowest Common Ancestor of a Binary Tree III	1650. 二叉树的最近公共祖先 III	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip，链表双指针**

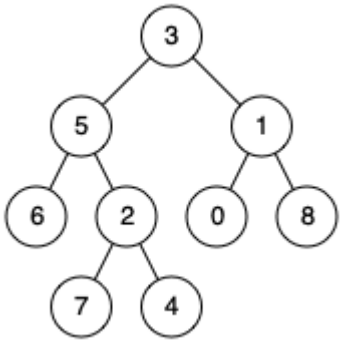
给定一棵二叉树中的两个节点 p 和 q，返回它们的最近公共祖先节点（LCA）。

每个节点都包含其父节点的引用（指针）。Node 的定义如下：

```
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node parent;
}
```

根据维基百科中对最近公共祖先节点的定义：“两个节点 p 和 q 在二叉树 T 中的最近公共祖先节点是后代节点中既包括 p 又包括 q 的最深节点（我们允许**一个节点为自身的一个后代节点**）”。一个节点 x 的后代节点是节点 x 到某一叶节点间的路径中的节点 y。

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 1 的最近公共祖先是 3。
```

基本思路

这道题需要你看过 单链表的六大解题套路总结 中 160. 相交链表 的解法。

你就把二叉树节点当成链表节点，`parent` 指针就是链表节点的 `next` 指针，最近公共祖先就是两个链表的交点，这道题就解决了。

- **详细题解：Git原理之最近公共祖先**

解法代码

```
class Solution {
    public Node lowestCommonAncestor(Node p, Node q) {
        // 施展链表双指针技巧
        Node a = p, b = q;
        while (a != b) {
            // a 走一步，如果走到根节点，转到 q 节点
            if (a == null) a = q;
            else a = a.parent;
            // b 走一步，如果走到根节点，转到 p 节点
            if (b == null) b = p;
            else b = b.parent;
        }
        return a;
    }
}
```

- **类似题目**：
  - 1257. 最小公共区域 🟠
  - 1644. 二叉树的最近公共祖先 II 🟠
  - 1676. 二叉树的最近公共祖先 IV 🟠
  - 235. 二叉搜索树的最近公共祖先 🟠
  - 236. 二叉树的最近公共祖先 🟠
  - 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先 🟢
  - 剑指 Offer 68 - II. 二叉树的最近公共祖先 🟢

1257. 最小公共区域

LeetCode	力扣	难度
1257. Smallest Common Region	1257. 最小公共区域	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip，链表双指针

给你一些区域列表 regions，每个列表的第一个区域都包含这个列表内所有其他区域。

很自然地，如果区域 X 包含区域 Y，那么区域 X 比区域 Y 大。给定两个区域 region1 和 region2，找到同时包含这两个区域的**最小**区域。

如果区域列表中 r1 包含 r2 和 r3，那么数据保证 r2 不会包含 r3 。数据同样保证最小公共区域一定存在。

示例 1：

```
输入：
regions = [["Earth","North America","South America"],
["North America","United States","Canada"],
["United States","New York","Boston"],
["Canada","Ontario","Quebec"],
["South America","Brazil"]],
region1 = "Quebec",
region2 = "New York"
输出："North America"
```

基本思路

这道题有意思，很显然是一个多叉树最近公共祖先的问题。建议看下这篇文章 二叉树最近公共祖先 中讲到的 1650. 二叉树的最近公共祖先 III。

我们可以用题目输入的 regions 结合哈希表构建多叉树，且节点指向关系是从子节点指向父节点，方便我们根据 region1, region2 回溯最近的公共祖先。构建出来之后可以直接套用 1650 的解题思路，利用 单链表的双指针技巧汇总 中讲的寻找单链表交点的算法找到答案。

解法代码

```
class Solution {
    HashMap<String, String> tree = new HashMap<>();

    public String findSmallestRegion(List<List<String>> regions, String
region1, String region2) {
        // 构建反向链接的多叉树，key 为子节点，value 为父节点
        for (List<String> list : regions) {
```

```
                String parent = list.get(0);
                for (String child : list.subList(1, list.size())) {
                    tree.put(child, parent);
                }
            }

            // 转化为 1650. 二叉树的最近公共祖先 III 的解法
            return lowestCommonAncestor(region1, region2);
        }

        String lowestCommonAncestor(String p, String q) {
            // 施展链表双指针技巧
            String a = p, b = q;
            while (!a.equals(b)) {
                // a 走一步，如果走到根节点，转到 q 节点
                a = tree.getOrDefault(a, q);
                // b 走一步，如果走到根节点，转到 p 节点
                b = tree.getOrDefault(b, p);
            }
            return a;
        }
    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

234. 回文链表

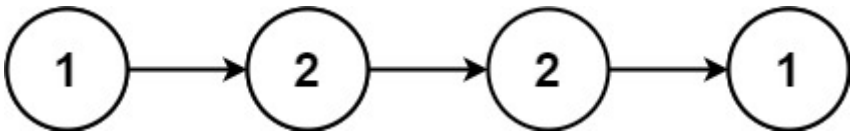
LeetCode	力扣	难度
234. Palindrome Linked List	234. 回文链表	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回文问题，数据结构，链表，链表双指针**

给你一个单链表的头节点 `head`，请你判断该链表是否为回文链表。如果是，返回 `true`；否则返回 `false`。

示例 1：




```
输入：head = [1,2,2,1]
输出：true
```

基本思路

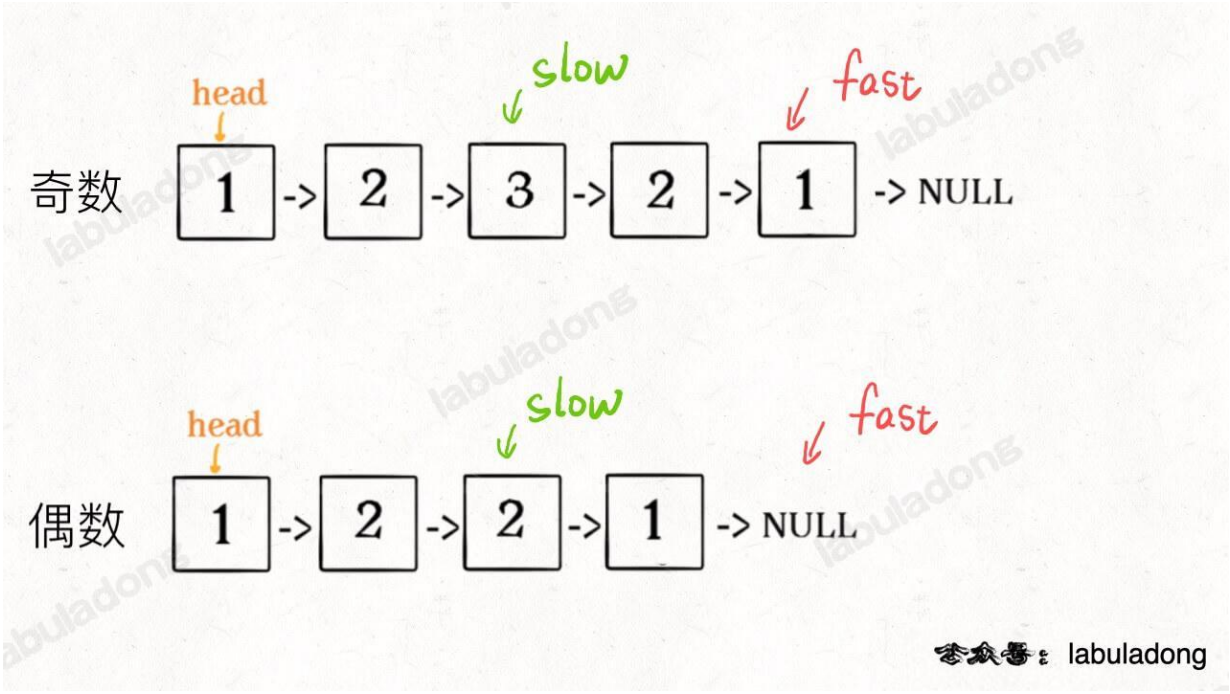
PS：这道题在《算法小抄》 的第 277 页。

这道题的关键在于，单链表无法倒着遍历，无法使用双指针技巧。

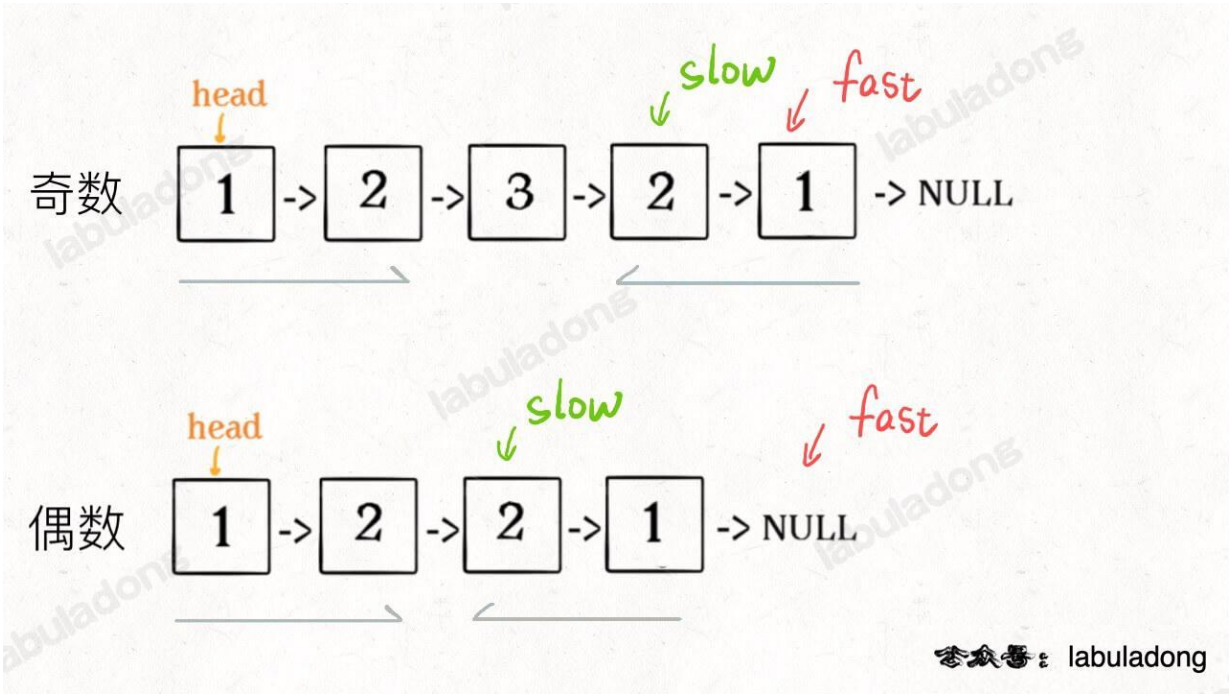
那么最简单的办法就是，把原始链表反转存入一条新的链表，然后比较这两条链表是否相同。

更聪明一些的办法是借助双指针算法：

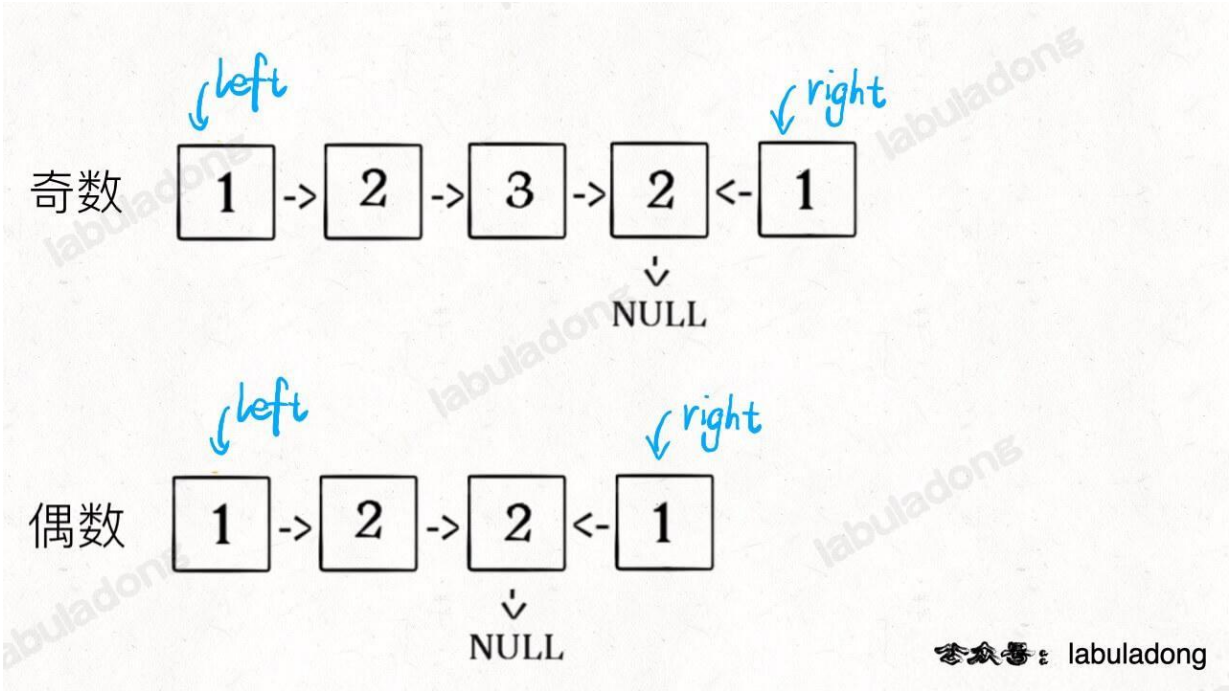
1、先通过 双指针技巧 中的快慢指针来找到链表的中点：



2、如果 `fast` 指针没有指向 `null`，说明链表长度为奇数，`slow` 还要再前进一步：



3、从 `slow` 开始反转后面的链表，现在就可以开始比较回文串了：



- **详细题解：如何判断回文链表**

解法代码

```
class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode slow, fast;
        slow = fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        if (fast != null)
            slow = slow.next;

        ListNode left = head;
        ListNode right = reverse(slow);
        while (right != null) {
            if (left.val != right.val)
                return false;
            left = left.next;
            right = right.next;
        }

        return true;
    }

    ListNode reverse(ListNode head) {
        ListNode pre = null, cur = head;
        while (cur != null) {
            ListNode next = cur.next;
            cur.next = pre;
```

```
                pre = cur;
                cur = next;
            }
            return pre;
        }
    }
```

▶ 🐏 代码可视化动画（需安装 Chrome 插件）🐏

- **类似题目**：
  - 剑指 Offer II 027. 回文链表 🟢

剑指 Offer II 027. 回文链表

这道题和 234. 回文链表 相同。

1201. 丑数 III

LeetCode	力扣	难度
1201. Ugly Number III	1201. 丑数 III	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数学，链表双指针

给你四个整数：`n` 、`a` 、`b` 、`c`，请你设计一个算法来找出第 `n` 个丑数。丑数是可以被 `a` **或** `b` **或** `c` 整除的 **正整数**。

示例 1：

```
输入：n = 3, a = 2, b = 3, c = 5
输出：4
解释：丑数序列为 2, 3, 4, 5, 6, 8, 9, 10... 其中第 3 个是 4。
```

基本思路

这道题和 264. 丑数 II 有些类似，你把第 264 题合并有序链表的解法照搬过来稍微改改就能解决这道题，代码我写在 `Solution2` 里面了。

但是注意题目给的数据规模，`a, b, c, n` 都是非常大的数字，如果用合并有序链表的思路，其复杂度是 O(N)，对于这么大的数据规模来说也是比较慢的，应该会超时，无法通过一些测试用例。

这道题的正确解法难度比较大，难点在于你要把一些数学知识和 二分搜索技巧 结合起来才能高效解决这个问题。

首先，根据 二分查找的实际运用 中讲到的二分搜索运用方法，我们可以抽象出一个单调递增的函数 `f`：

`f(num, a, b, c)` 计算 `[1..num]` 中，能够整除 `a` 或 `b` 或 `c` 的数字的个数，显然函数 `f` 的返回值是随着 `num` 的增加而增加的（单调递增）。

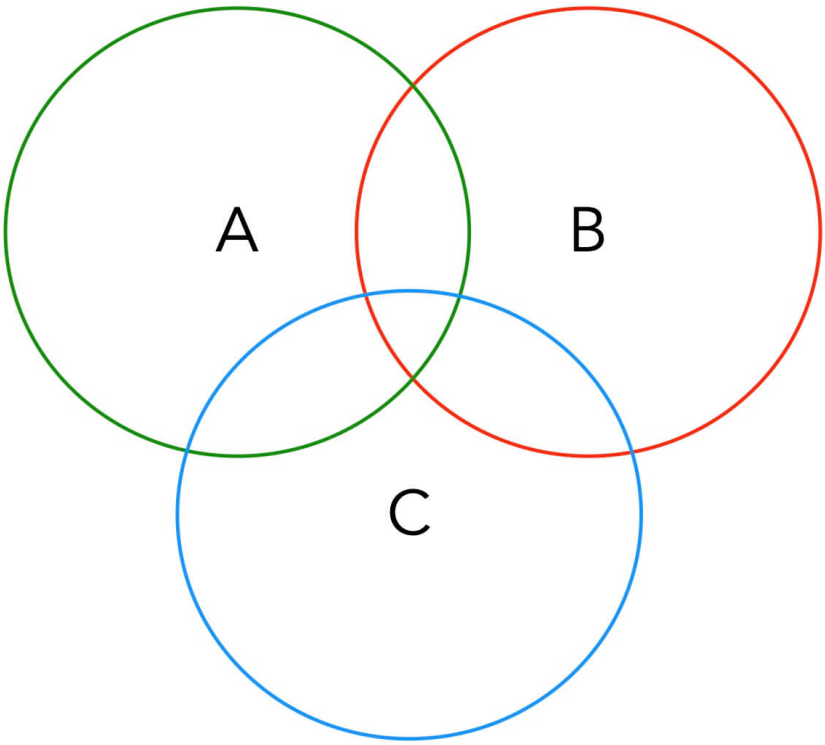
题目让我们求第 `n` 个能够整除 `a` 或 `b` 或 `c` 的数字是什么，也就是说我们要找到一个 `num`，使得 `f(num, a, b, c) == n`。

有了上述思路，就可以按照 二分查找的实际运用 中给出的模板运用二分搜索算法了。

关键说一下函数 `f` 怎么实现，这里面涉及集合论定理以及最小公因数、最小公倍数的计算方法。

首先，`[1..num]` 中，我把能够整除 `a` 的数字归为集合 `A`，能够整除 `b` 的数字归为集合 `B`，能够整除 `c` 的数字归为集合 `C`，那么 `len(A) = num / a, len(B) = num / b, len(C) = num / c`，这个很好理解。

但是 `f(num, a, b, c)` 的值肯定不是 `num / a + num / b + num / c` 这么简单，因为你注意有些数字可能可以被 `a, b, c` 中的两个数或三个数同时整除，如下图：



按照集合论的算法，这个集合中的元素应该是：A + B + C - A ∩ B - A ∩ C - B ∩ C + A ∩ B ∩ C。结合上图应该很好理解。

问题来了，`A, B, C` 三个集合的元素个数我们已经算出来了，但如何计算像 `A ∩ B` 这种交集的元素个数呢？

其实也很容易想明白，`A ∩ B` 的元素个数就是 `n / lcm(a, b)`，其中 `lcm` 是计算最小公倍数（Least Common Multiple）的函数。

类似的，`A ∩ B ∩ C` 的元素个数就是 `n / lcm(lcm(a, b), c)` 的值。

现在的问题是，最小公倍数怎么求？

直接记住定理吧：`lcm(a, b) = a * b / gcd(a, b)`，其中 `gcd` 是计算最大公因数（Greatest Common Divisor）的函数。

现在的问题是，最大公因数怎么求？

这应该是经典算法了，我们一般叫辗转相除算法（或者欧几里得算法）。

好了，套娃终于套完了，我们可以把上述思路翻译成解法，注意本题数据规模比较大，有时候需要用 `long` 类型防止 `int` 溢出，具体看我的代码实现以及注释吧。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
// 二分搜索 + 数学解法
class Solution {
    public int nthUglyNumber(int n, int a, int b, int c) {
```

```
        // 题目说本题结果在 [1, 2 * 10^9] 范围内，
        // 所以就按照这个范围初始化两端都闭的搜索区间
        int left = 1, right = (int) 2e9;
        // 搜索左侧边界的二分搜索
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (f(mid, a, b, c) < n) {
                // [1..mid] 中的元素个数不足 n，所以目标在右侧
                left = mid + 1;
            } else {
                // [1..mid] 中的元素个数大于 n，所以目标在左侧
                right = mid - 1;
            }
        }
        return left;
    }

    // 计算 [1..num] 之间有多少个能够被 a 或 b 或 c 整除的数字
    long f(int num, int a, int b, int c) {
        long setA = num / a, setB = num / b, setC = num / c;
        long setAB = num / lcm(a, b);
        long setAC = num / lcm(a, c);
        long setBC = num / lcm(b, c);
        long setABC = num / lcm(lcm(a, b), c);
        // 集合论定理：A + B + C - A ∩ B - A ∩ C - B ∩ C + A ∩ B ∩ C
        return setA + setB + setC - setAB - setAC - setBC + setABC;
    }

    // 计算最大公因数（辗转相除/欧几里得算法）
    long gcd(long a, long b) {
        if (a < b) {
            // 保证 a > b
            return gcd(b, a);
        }
        if (b == 0) {
            return a;
        }
        return gcd(b, a % b);
    }

    // 最小公倍数
    long lcm(long a, long b) {
        // 最小公倍数就是乘积除以最大公因数
        return a * b / gcd(a, b);
    }
}

// 用合并单链表的思路（超时）
class Solution2 {
    public int nthUglyNumber(int n, int a, int b, int c) {
        // 可以理解为三个有序链表的头结点的值
        long productA = a, productB = b, productC = c;
        // 可以理解为合并之后的有序链表上的指针
        int p = 1;
```

```
        long min = -666;

        // 开始合并三个有序链表，获取第 n 个节点的值
        while (p <= n) {
            // 取三个链表的最小结点
            min = Math.min(Math.min(productA, productB), productC);
            p++;
            // 前进最小结点对应链表的指针
            if (min == productA) {
                productA += a;
            }
            if (min == productB) {
                productB += b;
            }
            if (min == productC) {
                productC += c;
            }
        }
        return (int) min;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 263. 丑数 🟢
  - 264. 丑数 II 🟠
  - 313. 超级丑数 🟠
  - 剑指 Offer 49. 丑数 🟠

303. 区域和检索 - 数组不可变

LeetCode	力扣	难度
303. Range Sum Query - Immutable	303. 区域和检索 - 数组不可变	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：前缀和**

给你输入一个整数数组 `nums`，请你实现 `NumArray` 类：

1、`NumArray(int[] nums)` 使用数组 `nums` 初始化对象

2、`int sumRange(int i, int j)` 返回数组 `nums` 从索引 `i` 到 `j`（`i ≤ j`）范围内元素的总和，包含 `i`, `j` 两点（也就是 `sum(nums[i], nums[i + 1], ... , nums[j])`）

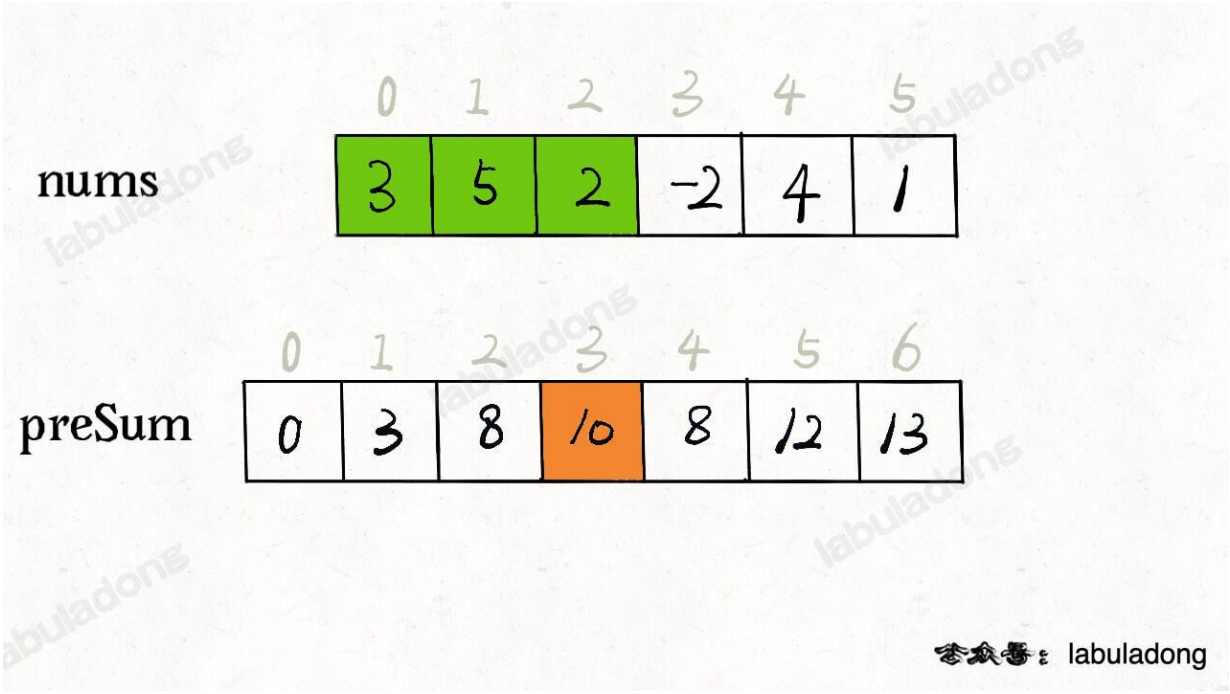
示例：

```
输入：
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
输出：
[null, 1, -1, -3]

解释：
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return 1 ((-2) + 0 + 3)
numArray.sumRange(2, 5); // return -1 (3 + (-5) + 2 + (-1))
numArray.sumRange(0, 5); // return -3 ((-2) + 0 + 3 + (-5) + 2 + (-1))
```

基本思路

> 本文有视频版：前缀和/差分数组技巧精讲

标准的前缀和问题，核心思路是用一个新的数组 `preSum` 记录 `nums[0..i-1]` 的累加和，看图 10 = 3 + 5 + 2：



看这个 `preSum` 数组，如果我想求索引区间 `[1, 4]` 内的所有元素之和，就可以通过 `preSum[5] - preSum[1]` 得出。

这样，`sumRange` 函数仅仅需要做一次减法运算，避免了每次进行 for 循环调用，最坏时间复杂度为常数 `O(1)`。

- **详细题解：小而美的算法技巧：前缀和数组**

解法代码

```
class NumArray {
    // 前缀和数组
    private int[] preSum;

    /* 输入一个数组，构造前缀和 */
    public NumArray(int[] nums) {
        // preSum[0] = 0，便于计算累加和
        preSum = new int[nums.length + 1];
        // 计算 nums 的累加和
        for (int i = 1; i < preSum.length; i++) {
            preSum[i] = preSum[i - 1] + nums[i - 1];
        }
    }

    /* 查询闭区间 [left, right] 的累加和 */
    public int sumRange(int left, int right) {
        return preSum[right + 1] - preSum[left];
    }
}
```

- **类似题目：**
  - 304. 二维区域和检索 - 矩阵不可变 🟠

- 剑指 Offer II 013. 二维子矩阵的和 🟠

304. 二维区域和检索 - 矩阵不可变

LeetCode	力扣	难度
304. Range Sum Query 2D - Immutable	304. 二维区域和检索 - 矩阵不可变	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：前缀和**

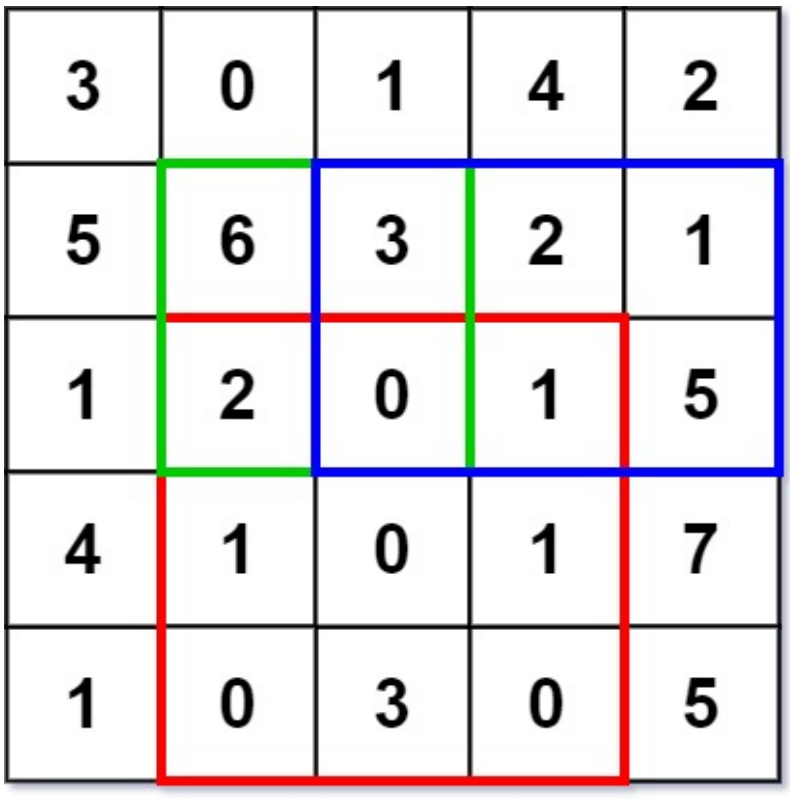
给定一个二维矩阵 `matrix`，其中的一个子矩阵用其**左上角**坐标 `(row1, col1)` 和**右下角**坐标 `(row2, col2)` 来表示。

请你实现 `NumMatrix` 类：

1、`NumMatrix(int[][] matrix)` 给定整数矩阵 `matrix` 进行初始化

2、`int sumRegion(int row1, int col1, int row2, int col2)` 返回**左上角** `(row1, col1)`，**右下角** `(row2, col2)` 所描述的子矩阵的元素**总和**。

示例 1：



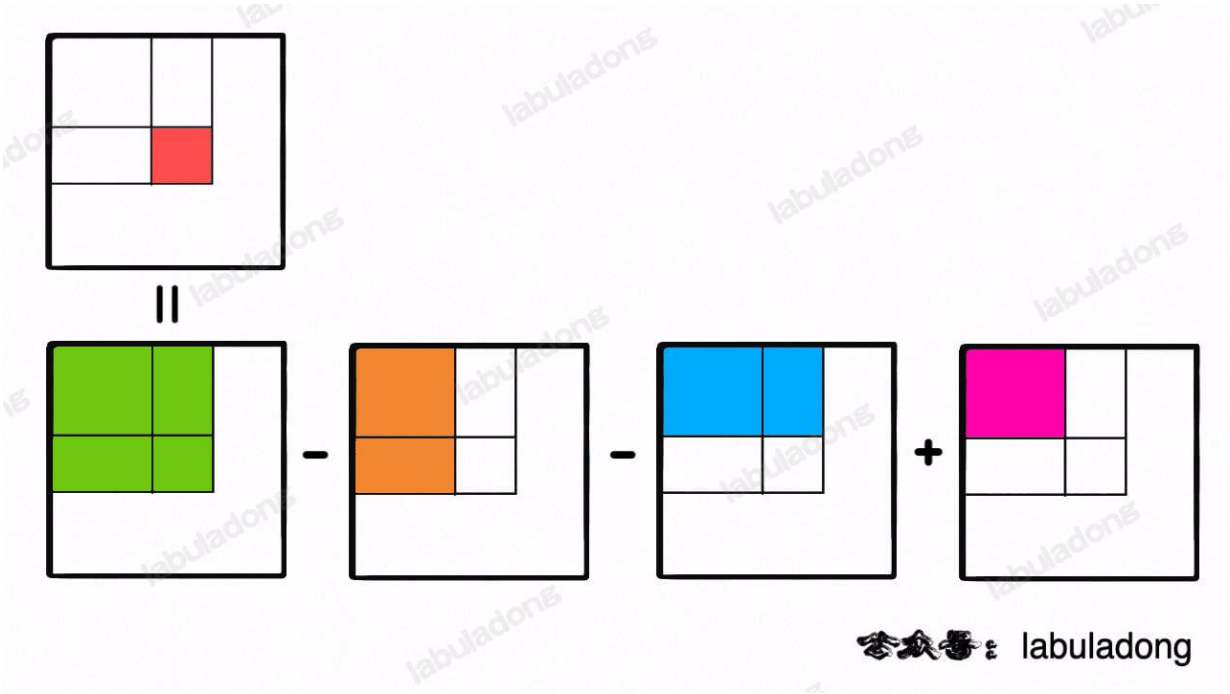

```
输入：
["NumMatrix","sumRegion","sumRegion","sumRegion"]
[[[[3,0,1,4,2],[5,6,3,2,1],[1,2,0,1,5],[4,1,0,1,7],[1,0,3,0,5]]],
[2,1,4,3],[1,1,2,2],[1,2,2,4]]
输出：
[null, 8, 11, 12]
```

```
解释:
NumMatrix numMatrix = new NumMatrix([[3,0,1,4,2],[5,6,3,2,1],[1,2,0,1,5],
[4,1,0,1,7],[1,0,3,0,5]]);
numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)
numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)
```

基本思路

> 本文有视频版：前缀和/差分数组技巧精讲

这题的思路和 303. 区域和检索 - 数组不可变 中一维数组中的前缀和问题是非常类似的，如下图：



如果我想计算红色的这个子矩阵的元素之和，可以用绿色矩阵减去蓝色矩阵减去橙色矩阵最后加上粉色矩阵，而绿蓝橙粉这四个矩阵有一个共同的特点，就是左上角就是 `(0, 0)` 原点。

那么我们可以维护一个二维 `preSum` 数组，专门记录以原点为顶点的矩阵的元素之和，就可以用几次加减运算算出任何一个子矩阵的元素和。

- **详细题解：小而美的算法技巧：前缀和数组**

解法代码

```
class NumMatrix {
    // preSum[i][j] 记录矩阵 [0, 0, i-1, j-1] 的元素和
    private int[][] preSum;

    public NumMatrix(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        if (m == 0 || n == 0) return;
        // 构造前缀和矩阵
```

```
        preSum = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // 计算每个矩阵 [0, 0, i, j] 的元素和
                preSum[i][j] = preSum[i-1][j] + preSum[i][j-1] + matrix[i
- 1][j - 1] - preSum[i-1][j-1];
            }
        }
    }

    // 计算子矩阵 [x1, y1, x2, y2] 的元素和
    public int sumRegion(int x1, int y1, int x2, int y2) {
        // 目标矩阵之和由四个相邻矩阵运算获得
        return preSum[x2+1][y2+1] - preSum[x1][y2+1] - preSum[x2+1][y1] +
preSum[x1][y1];
    }
}
```

- **类似题目**：
  - 1314. 矩阵区域和 🟠
  - 303. 区域和检索 - 数组不可变 🟢
  - 剑指 Offer II 013. 二维子矩阵的和 🟠

剑指 Offer II 013. 二维子矩阵的和

这道题和 304. 二维区域和检索 - 矩阵不可变 相同。

1314. 矩阵区域和

LeetCode	力扣	难度
1314. Matrix Block Sum	1314. 矩阵区域和	🟠

- **标签：前缀和，数据结构，数组**

给你一个 `m x n` 的矩阵 `mat` 和一个整数 `k`，请你返回一个矩阵 `answer`，其中每个 `answer[i][j]` 是所有满足下述条件的元素 `mat[r][c]` 的和：

```
i - k <= r <= i + k, j - k <= c <= j + k 且 (r, c) 在矩阵内。
```

示例 1：

```
输入：mat = [[1,2,3],[4,5,6],[7,8,9]], k = 1
输出：[[12,21,16],[27,45,33],[24,39,28]]
```

基本思路

这道题可以直接套用前文 前缀和数组技巧 中讲 304. 二维区域和检索 时实现的 `NumMatrix` 类，没什么难度。主要注意下通过 `min, max` 函数优雅避免索引越界的技巧，这个还是蛮常用的。

解法代码

```
class Solution {
    public int[][] matrixBlockSum(int[][] mat, int k) {
        int m = mat.length, n = mat[0].length;
        NumMatrix numMatrix = new NumMatrix(mat);
        int[][] res = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // 左上角的坐标
                int x1 = Math.max(i - k, 0);
                int y1 = Math.max(j - k, 0);
                // 右下角坐标
                int x2 = Math.min(i + k, m - 1);
                int y2 = Math.min(j + k, n - 1);

                res[i][j] = numMatrix.sumRegion(x1, y1, x2, y2);
            }
        }
        return res;
```

```
    }
}

class NumMatrix {
    // 定义：preSum[i][j] 记录 matrix 中子矩阵 [0, 0, i-1, j-1] 的元素和
    private int[][] preSum;

    public NumMatrix(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        if (m == 0 || n == 0) return;
        // 构造前缀和矩阵
        preSum = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // 计算每个矩阵 [0, 0, i, j] 的元素和
                preSum[i][j] = preSum[i - 1][j] + preSum[i][j - 1] +
matrix[i - 1][j - 1] - preSum[i - 1][j - 1];
            }
        }
    }

    // 计算子矩阵 [x1, y1, x2, y2] 的元素和
    public int sumRegion(int x1, int y1, int x2, int y2) {
        // 目标矩阵之和由四个相邻矩阵运算获得
        return preSum[x2 + 1][y2 + 1] - preSum[x1][y2 + 1] - preSum[x2 +
1][y1] + preSum[x1][y1];
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

325. 和等于 k 的最长子数组长度

LeetCode	力扣	难度
325. Maximum Size Subarray Sum Equals k	325. 和等于 k 的最长子数组长度	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：前缀和，数组**

给定一个数组 `nums` 和一个目标值 `k`，找到和等于 *k*的最长连续子数组长度。如果不存在任意一个符合要求的子数组，则返回 `0`。

示例 1:

```
输入: nums = [1,-1,5,-2,3], k = 3
输出: 4
解释: 子数组 [1, -1, 5, -2] 和等于 3，且长度最长。
```

基本思路

这道题就是标准的前缀和技巧 + 哈希表的组合，建议你先做一下 525. 连续数组 和 560. 和为 K 的子数组，我在那两道题的思路写的比较详细。

本题让你寻找长度最长的和为 `k` 的子数组，翻译一下就是：

寻找 `i, j` 使得 `preSum[i] - preSum[j] == k` 且 `i - j` 尽可能的大。

另外，`preSum[i] - preSum[j] == k` 其实就是 `preSum[j] == preSum[i] - k`。

所以我们使用一个哈希表，记录 `preSum[i]` 的值以及这个前缀和第一次出现的索引，就可以迅速判断 `preSum[i]` 是否符合条件并计算最长子数组长度了。具体看代码吧。

解法代码

```
class Solution {
    public static int maxSubArrayLen(int[] nums, int k) {
        int n = nums.length;
        // preSum 中的值 -> 对应的最小索引
        // 比如 preSum = [2,4,1,3,4], preSumToIndex[4] = 1
        HashMap<Integer, Integer> preSumToIndex = new HashMap<>();
        int maxLen = 0;
        // 前缀和数组（在这道题中可以优化为一个变量）
        int preSum = 0;
        // base case，这样索引相减的时候可以算出正确的子数组长度
        preSumToIndex.put(0, -1);
        for (int i = 0; i < n; i++) {
```

```
                // 计算前缀和，维护 preSum = sum(nums[0..i])
                preSum += nums[i];
                // 确保 preSumToIndex 中记录的索引是第一次出现的位置
                preSumToIndex.putIfAbsent(preSum, i);
                int need = preSum[i] - k;
                if (preSumToIndex.containsKey(need)) {
                    int j = preSumToIndex.get(need);
                    // nums[j..i] 是和为 k 的子数组
                    maxLen = Math.max(maxLen, i - j);
                }
            }
            return maxLen;
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

327. 区间和的个数

LeetCode	力扣	难度
327. Count of Range Sum	327. 区间和的个数	🔴

- **标签：前缀和，双指针，归并排序**

给你一个整数数组 `nums` 以及两个整数 `lower` 和 `upper`。求数组中，值位于范围 `[lower, upper]`（包含 `lower` 和 `upper`）之内的 **区间和的个数**。

区间和 `S(i, j)` 表示在 `nums` 中，位置从 `i` 到 `j` 的元素之和，包含 `i` 和 `j`。

示例 1：

```
输入：nums = [-2,5,-1], lower = -2, upper = 2
输出：3
解释：存在三个区间：[0,0]、[2,2] 和 [0,2]，对应的区间和分别是：-2 、-1 、2。
```

基本思路

这道题难度非常大，建议你先阅读前文 小而美的算法技巧：前缀和数组 以及 归并排序详解及应用，并完成 315. 计算右侧小于当前元素的个数（困难）。

然后，你就会发现，这道题和 315. 计算右侧小于当前元素的个数（困难） 非常类似：

315 题让你计算每个元素之后比它小的元素个数，即求出一个 `count` 数组，使得 `count[i] = COUNT(nums[j], j > i and nums[j] < nums[i])`。

这道题，你可以先对原数组求一下前缀和数组 `preSum`，然后去 `preSum` 中求一个 `count` 数组，使得 `count[i] = COUNT(nums[j], j > i and lower <= preSum[j] - nums[i] <= upper)`，然后 `SUM(count)` 就是题目想要的结果。

那么思路也是在归并排序的过程中夹带点私货，可以对比第 315 题直接看解法代码。

- **详细题解：归并排序详解及应用**

解法代码

```
class Solution {
    int lower, upper;

    public int countRangeSum(int[] nums, int lower, int upper) {
        this.lower = lower;
        this.upper = upper;
```

```
        long[] preSum = new long[nums.length + 1];
        for (int i = 0; i < nums.length; i++) {
            preSum[i + 1] = (long) nums[i] + preSum[i];
        }
        sort(preSum);
        return count;
    }

    // 用于辅助合并有序数组
    private long[] temp;
    private int count = 0;

    public void sort(long[] nums) {
        // 先给辅助数组开辟内存空间
        temp = new long[nums.length];
        // 排序整个数组（原地修改）
        sort(nums, 0, nums.length - 1);
    }

    // 定义：将子数组 nums[lo..hi] 进行排序
    private void sort(long[] nums, int lo, int hi) {
        if (lo == hi) {
            // 单个元素不用排序
            return;
        }
        // 这样写是为了防止溢出，效果等同于 (hi + lo) / 2
        int mid = lo + (hi - lo) / 2;
        // 先对左半部分数组 nums[lo..mid] 排序
        sort(nums, lo, mid);
        // 再对右半部分数组 nums[mid+1..hi] 排序
        sort(nums, mid + 1, hi);
        // 将两部分有序数组合并成一个有序数组
        merge(nums, lo, mid, hi);
    }

    // 将 nums[lo..mid] 和 nums[mid+1..hi] 这两个有序数组合并成一个有序数组
    private void merge(long[] nums, int lo, int mid, int hi) {
        // 先把 nums[lo..hi] 复制到辅助数组中
        // 以便合并后的结果能够直接存入 nums
        for (int i = lo; i <= hi; i++) {
            temp[i] = nums[i];
        }

        // 这段代码会超时
        // for (int i = lo; i <= mid; i++) {
        //     // 在区间 [mid + 1, hi] 中寻找 lower <= delta <= upper 的元素
        //     for (int k = mid + 1; k <= hi; k++) {
        //         long delta = nums[k] - nums[i];
        //         if (delta <= upper && delta >= lower) {
        //             count++;
        //         }
        //     }
        // }
```

```
        // 进行效率优化
        // 维护左闭右开区间 [start, end) 中的元素落在 [lower, upper] 中
        int start = mid + 1, end = mid + 1;
        for (int i = lo; i <= mid; i++) {
            while (start <= hi && nums[start] - nums[i] < lower) {
                start++;
            }
            while (end <= hi && nums[end] - nums[i] <= upper) {
                end++;
            }
            count += end - start;
        }

        // 数组双指针技巧，合并两个有序数组
        int i = lo, j = mid + 1;
        for (int p = lo; p <= hi; p++) {
            if (i == mid + 1) {
                // 左半边数组已全部被合并
                nums[p] = temp[j++];
            } else if (j == hi + 1) {
                // 右半边数组已全部被合并
                nums[p] = temp[i++];
            } else if (temp[i] > temp[j]) {
                nums[p] = temp[j++];
            } else {
                nums[p] = temp[i++];
            }
        }
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 315. 计算右侧小于当前元素的个数 🔴
  - 493. 翻转对 🔴
  - 912. 排序数组 🟠

1352. 最后 K 个数的乘积

LeetCode	力扣	难度
1352. Product of the Last K Numbers	1352. 最后 K 个数的乘积	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：前缀和**

请你实现一个「数字乘积类」`ProductOfNumbers`，要求支持下述两种方法：

1. `add(int num)` 将数字 `num` 添加到当前数字列表的最后面。

2. `getProduct(int k)` 返回当前数字列表中，最后 `k` 个数字的乘积。

你可以假设当前列表中始终 **至少** 包含 `k` 个数字。

示例：

```
输入：
["ProductOfNumbers","add","add","add","add","add","getProduct","getProduct
","getProduct","add","getProduct"]
[[],[3],[0],[2],[5],[4],[2],[3],[4],[8],[2]]

输出：
[null,null,null,null,null,null,20,40,0,null,32]

解释：
ProductOfNumbers productOfNumbers = new ProductOfNumbers();
productOfNumbers.add(3);        // [3]
productOfNumbers.add(0);        // [3,0]
productOfNumbers.add(2);        // [3,0,2]
productOfNumbers.add(5);        // [3,0,2,5]
productOfNumbers.add(4);        // [3,0,2,5,4]
productOfNumbers.getProduct(2); // 返回 20。最后 2 个数字的乘积是 5 * 4 = 20
productOfNumbers.getProduct(3); // 返回 40。最后 3 个数字的乘积是 2 * 5 * 4 =
40
productOfNumbers.getProduct(4); // 返回  0。最后 4 个数字的乘积是 0 * 2 * 5 * 4
= 0
productOfNumbers.add(8);        // [3,0,2,5,4,8]
productOfNumbers.getProduct(2); // 返回 32。最后 2 个数字的乘积是 4 * 8 = 32
```

基本思路

如果你看过前文 小而美的算法技巧：前缀和数组 这道题就不难，前缀和和前缀积很类似，只不过乘积中如果有 0 需要特殊处理。

解法代码

```
class ProductOfNumbers {
    // 前缀积数组
    // preProduct[i] / preProduct[j] 就是 [i, j] 之间的元素积
    ArrayList<Integer> preProduct = new ArrayList<>();

    public ProductOfNumbers() {
        // 初始化放一个 1，便于计算后续添加元素的乘积
        preProduct.add(1);
    }

    public void add(int num) {
        if (num == 0) {
            // 如果添加的元素是 0，则前面的元素积都废了
            preProduct.clear();
            preProduct.add(1);
            return;
        }
        int n = preProduct.size();
        // 前缀积数组中每个元素
        preProduct.add(preProduct.get(n - 1) * num);
    }

    public int getProduct(int k) {
        int n = preProduct.size();
        if (k > n - 1) {
            // 不足 k 个元素，是因为最后 k 个元素存在 0
            return 0;
        }
        // 计算最后 k 个元素积
        return preProduct.get(n - 1) / preProduct.get(n - k - 1);
    }
}
```

剑指 Offer 66. 构建乘积数组

LeetCode	力扣	难度
剑指Offer66. 构建乘积数组 LCOF	剑指Offer66. 构建乘积数组	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：前缀和，数组**

给定一个数组 `A[0,1,…,n-1]`，请构建一个数组 `B[0,1,…,n-1]`，其中 `B[i]` 的值是数组 `A` 中除了下标 `i` 以外的元素的积，即 `B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]`。不能使用除法。

示例:

```
输入：[1,2,3,4,5]
输出：[120,60,40,30,24]
```

基本思路

这道题和 238. 除自身以外数组的乘积 一模一样，可以看下 238 的思路或者直接看解法代码吧。

解法代码

```
class Solution {
    public int[] constructArr(int[] nums) {
        int n = nums.length;
        if (n == 0) {
            return new int[0];
        }
        // 从左到右的前缀积，prefix[i] 是 nums[0..i] 的元素积
        int[] prefix = new int[n];
        prefix[0] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            prefix[i] = prefix[i - 1] * nums[i];
        }
        // 从右到左的前缀积，suffix[i] 是 nums[i..n-1] 的元素积
        int[] suffix = new int[n];
        suffix[n - 1] = nums[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            suffix[i] = suffix[i + 1] * nums[i];
        }
        // 结果数组
        int[] res = new int[n];
        res[0] = suffix[1];
        res[n - 1] = prefix[n - 2];
        for (int i = 1; i < n - 1; i++) {
```

```
            // 除了 nums[i] 自己的元素积就是 nums[i] 左侧和右侧所有元素之积
            res[i] = prefix[i - 1] * suffix[i + 1];
        }
        return res;
    }
}
```

▶ 代码可视化动画（需安装 Chrome 插件）

1094. 拼车

LeetCode	力扣	难度
1094. Car Pooling	1094. 拼车	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：差分数组**

你是一个开公交车的司机，公交车的最大载客量为 `capacity`，沿途要经过若干车站，给你一份乘客行程表 `int[][] trips`，其中 `trips[i] = [num, start, end]` 代表着有 `num` 个旅客要从站点 `start` 上车，到站点 `end` 下车，请你计算是否能够一次把所有旅客运送完毕（不能超过最大载客量 `capacity`）。

示例 1：

```
输入：trips = [[2,1,5],[3,3,7]], capacity = 4
输出：false
```

基本思路

相信你已经能够联想到差分数组技巧了：**`trips[i]` 代表着一组区间操作，旅客的上车和下车就相当于数组的区间加减；只要结果数组中的元素都小于 `capacity`，就说明可以不超载运输所有旅客。**

这题还有一个细节，一批乘客从站点 `trip[1]` 上车，到站点 `trip[2]` 下车，呆在车上的站点应该是 `[trip[1], trip[2] - 1]`，这是需要被操作的索引区间。

- **详细题解：小而美的算法技巧：差分数组**

解法代码

```
class Solution {
    public boolean carPooling(int[][] trips, int capacity) {
        // 最多有 1000 个车站
        int[] nums = new int[1001];
        // 构造差分解法
        Difference df = new Difference(nums);

        for (int[] trip : trips) {
            // 乘客数量
            int val = trip[0];
            // 第 trip[1] 站乘客上车
            int i = trip[1];
            // 第 trip[2] 站乘客已经下车，
            // 即乘客在车上的区间是 [trip[1], trip[2] - 1]
            int j = trip[2] - 1;
            // 进行区间操作
```

```
            df.increment(i, j, val);
        }

        int[] res = df.result();

        // 客车自始至终都不应该超载
        for (int i = 0; i < res.length; i++) {
            if (capacity < res[i]) {
                return false;
            }
        }
        return true;
    }

    // 差分数组工具类
    class Difference {
        // 差分数组
        private int[] diff;

        /* 输入一个初始数组，区间操作将在这个数组上进行 */
        public Difference(int[] nums) {
            assert nums.length > 0;
            diff = new int[nums.length];
            // 根据初始数组构造差分数组
            diff[0] = nums[0];
            for (int i = 1; i < nums.length; i++) {
                diff[i] = nums[i] - nums[i - 1];
            }
        }

        /* 给闭区间 [i, j] 增加 val（可以是负数）*/
        public void increment(int i, int j, int val) {
            diff[i] += val;
            if (j + 1 < diff.length) {
                diff[j + 1] -= val;
            }
        }

        /* 返回结果数组 */
        public int[] result() {
            int[] res = new int[diff.length];
            // 根据差分数组构造结果数组
            res[0] = diff[0];
            for (int i = 1; i < diff.length; i++) {
                res[i] = res[i - 1] + diff[i];
            }
            return res;
        }
    }

}
```

- **类似题目**：

- 1109. 航班预订统计 🟠
- 370. 区间加法 🟠

1109. 航班预订统计

LeetCode	力扣	难度
1109. Corporate Flight Bookings	1109. 航班预订统计	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：差分数组，数组**

这里有 `n` 个航班，它们分别从 `1` 到 `n` 进行编号。有一份航班预订表 `bookings`，表中第 `i` 条预订记录 `bookings[i] = [firsti, lasti, seatsi]` 意味着在从 `firsti` 到 `lasti`（**包含** `firsti` 和 `lasti`）的**每个航班**上预订了 `seatsi` 个座位。

请你返回一个长度为 `n` 的数组 `answer`，里面的元素是每个航班预定的座位总数。

示例 1：

```
输入：bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
输出：[10,55,45,25,25]
解释：
航班编号         1   2   3   4   5
预订记录 1：     10  10
预订记录 2：         20  20
预订记录 3：         25  25  25  25
总座位数：       10  55  45  25  25
因此，answer = [10,55,45,25,25]
```

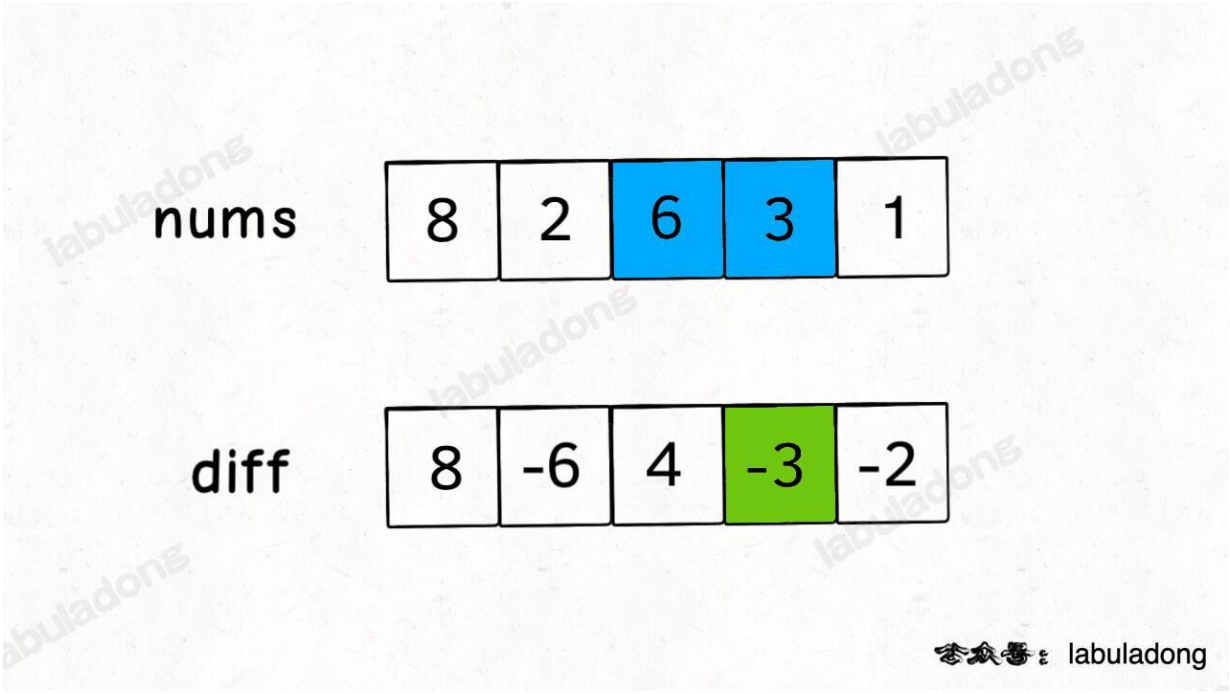
基本思路

这题考察差分数组技巧，差分数组技巧适用于频繁对数组区间进行增减的场景。

核心原理：

1、构造差分数组：

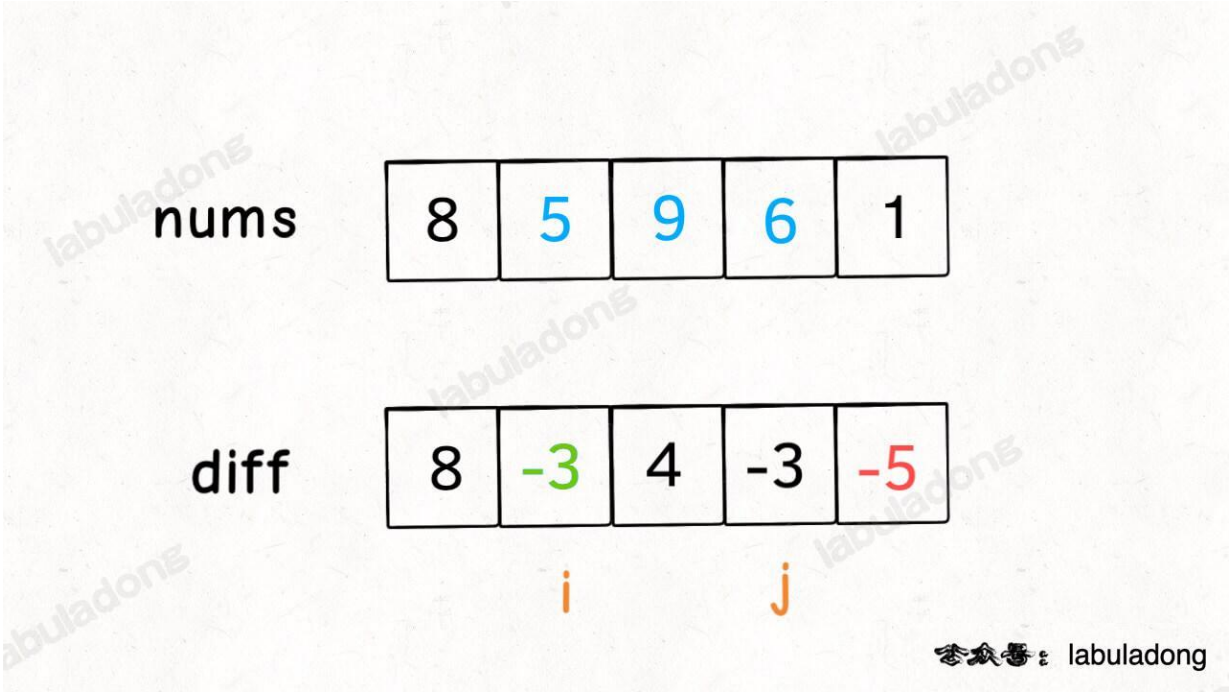
```
int[] diff = new int[nums.length];
// 构造差分数组
diff[0] = nums[0];
for (int i = 1; i < nums.length; i++) {
    diff[i] = nums[i] - nums[i - 1];
}
```



2、还原原始数组：

```
int[] res = new int[diff.length];
// 根据差分数组构造结果数组
res[0] = diff[0];
for (int i = 1; i < diff.length; i++) {
    res[i] = res[i - 1] + diff[i];
}
```

2、进行区间增减，如果你想对区间 `nums[i..j]` 的元素全部加 3，那么只需要让 `diff[i] += 3`，然后再让 `diff[j+1] -= 3` 即可：



本题就相当于给你输入一个长度为 n 的数组 nums，其中所有元素都是 0，然后让你进行一系列区间加减操作，可以套用差分数组技巧。

- **详细题解：小而美的算法技巧：差分数组**

解法代码

```
class Solution {
    public int[] corpFlightBookings(int[][] bookings, int n) {
        // nums 初始化为全 0
        int[] nums = new int[n];
        // 构造差分解法
        Difference df = new Difference(nums);

        for (int[] booking : bookings) {
            // 注意转成数组索引要减一哦
            int i = booking[0] - 1;
            int j = booking[1] - 1;
            int val = booking[2];
            // 对区间 nums[i..j] 增加 val
            df.increment(i, j, val);
        }
        // 返回最终的结果数组
        return df.result();
    }

    class Difference {
        // 差分数组
        private int[] diff;

        public Difference(int[] nums) {
            assert nums.length > 0;
            diff = new int[nums.length];
            // 构造差分数组
            diff[0] = nums[0];
            for (int i = 1; i < nums.length; i++) {
                diff[i] = nums[i] - nums[i - 1];
            }
        }

        /* 给闭区间 [i, j] 增加 val（可以是负数）*/
        public void increment(int i, int j, int val) {
            diff[i] += val;
            if (j + 1 < diff.length) {
                diff[j + 1] -= val;
            }
        }

        public int[] result() {
            int[] res = new int[diff.length];
            // 根据差分数组构造结果数组
            res[0] = diff[0];
```

```
                for (int i = 1; i < diff.length; i++) {
                    res[i] = res[i - 1] + diff[i];
                }
                return res;
            }
        }

    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 1094. 拼车 🟠
  - 370. 区间加法 🟠

370. 区间加法

LeetCode	力扣	难度
370. Range Addition	370. 区间加法	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：差分数组**

假设你有一个长度为 n 的数组，初始情况下所有的数字均为 0，你将会被给出 k 个更新的操作。

其中，每个操作会被表示为一个三元组：`[startIndex, endIndex, inc]`，你需要将子数组 `A[startIndex ... endIndex]`（包括 `startIndex` 和 `endIndex`）增加 `inc`。

请你返回 k 次操作后的数组。

示例 1：

```
输入: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]]
输出: [-2,0,3,5,3]

解释:
初始状态:
[0,0,0,0,0]

进行了操作 [1,3,2] 后的状态:
[0,2,2,2,0]

进行了操作 [2,4,3] 后的状态:
[0,2,5,5,3]

进行了操作 [0,2,-2] 后的状态:
[-2,0,3,5,3]
```

基本思路

这题是标准的差分数组技巧，基本原理见 1109. 航班预订统计，或见详细题解。

解法代码直接复用差分算法类即可。

- **详细题解：小而美的算法技巧：差分数组**

解法代码

```
class Solution {
    public int[] getModifiedArray(int length, int[][] updates) {
```

```
        // nums 初始化为全 0
        int[] nums = new int[length];
        // 构造差分解法
        Difference df = new Difference(nums);
        for (int[] update : updates) {
            int i = update[0];
            int j = update[1];
            int val = update[2];
            df.increment(i, j, val);
        }
        return df.result();
    }

    class Difference {
        // 差分数组
        private int[] diff;

        public Difference(int[] nums) {
            assert nums.length > 0;
            diff = new int[nums.length];
            // 构造差分数组
            diff[0] = nums[0];
            for (int i = 1; i < nums.length; i++) {
                diff[i] = nums[i] - nums[i - 1];
            }
        }

        /* 给闭区间 [i, j] 增加 val（可以是负数）*/
        public void increment(int i, int j, int val) {
            diff[i] += val;
            if (j + 1 < diff.length) {
                diff[j + 1] -= val;
            }
        }

        public int[] result() {
            int[] res = new int[diff.length];
            // 根据差分数组构造结果数组
            res[0] = diff[0];
            for (int i = 1; i < diff.length; i++) {
                res[i] = res[i - 1] + diff[i];
            }
            return res;
        }
    }

}
```

- **类似题目**：
  - 1094. 拼车 🟠
  - 1109. 航班预订统计 🟠

1541. 平衡括号字符串的最少插入次数

LeetCode	力扣	难度
1541. Minimum Insertions to Balance a Parentheses String	1541. 平衡括号字符串的最少插入次数	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：括号问题**

给你一个括号字符串 `s`，它只包含字符 `'('` 和 `')'`。一个括号字符串被称为平衡的当它满足：

1、任何左括号 `'('` 必须对应两个连续的右括号 `'))'`。

2、左括号 `'('` 必须在对应的连续两个右括号 `'))'` 之前。

比方说 `"())"`，`"())(())))"` 和 `"(())())))"` 都是平衡的，`")()"`，`"()))"` 和 `"(()))"` 都是不平衡的。

你可以在任意位置插入字符 `'('` 和 `')'`，请你返回让 `s` 平衡的最少插入次数。

示例 1：

```
输入：s = "(()))"
输出：1
解释：第二个左括号有与之匹配的两个右括号，但是第一个左括号只有一个右括号。我们需要在字符串结尾额外增加一个 ')' 使字符串变成平衡字符串 "(())))"。
```

基本思路

遍历字符串，通过一个 `need` 变量记录对右括号的需求数，根据 `need` 的变化来判断是否需要插入。

类似 921. 使括号有效的最少添加，当 `need == -1` 时，意味着我们遇到一个多余的右括号，显然需要插入一个左括号。

另外，当遇到左括号时，若对右括号的需求量为奇数，需要插入 1 个右括号，因为一个左括号需要两个右括号嘛，右括号的需求必须是偶数，这一点也是本题的难点。

- **详细题解：如何解决括号相关的问题**

解法代码

```
class Solution {
    public int minInsertions(String s) {
        int res = 0, need = 0;
```

```
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                need += 2;
                if (need % 2 == 1) {
                    res++;
                    need--;
                }
            }

            if (s.charAt(i) == ')') {
                need--;
                if (need == -1) {
                    res++;
                    need = 1;
                }
            }
        }

        return res + need;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 20. 有效的括号 🟢
  - 921. 使括号有效的最少添加 🟠

20. 有效的括号

LeetCode	力扣	难度
20. Valid Parentheses	20. 有效的括号	🟢

- **标签：括号问题，栈**

给定一个只包括 `'('`，`')'`，`'{'`，`'}'`，`'['`，`']'` 的字符串 `s`，判断字符串是否有效。

有效字符串需满足：

1、 左括号必须用相同类型的右括号闭合。

2、 左括号必须以正确的顺序闭合。

示例 1：

```
输入：s = "([)]"
输出：false
```

示例 2：

```
输入：s = "()[]{}"
输出：true
```

基本思路

栈是一种先进后出的数据结构，处理括号问题的时候尤其有用。

遇到左括号就入栈，遇到右括号就去栈中寻找最近的左括号，看是否匹配。

- **详细题解：如何解决括号相关的问题**

解法代码

```
class Solution {
    public boolean isValid(String str) {
        Stack<Character> left = new Stack<>();
        for (char c : str.toCharArray()) {
            if (c == '(' || c == '{' || c == '[')
                left.push(c);
            else // 字符 c 是右括号
                if (!left.isEmpty() && leftOf(c) == left.peek())
```

```
                    left.pop();
                else
                    // 和最近的左括号不匹配
                    return false;
            }
            // 是否所有的左括号都被匹配了
            return left.isEmpty();
        }

        char leftOf(char c) {
            if (c == '}') return '{';
            if (c == ')') return '(';
            return '[';
        }
    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 1541. 平衡括号字符串的最少插入次数 🟠
  - 921. 使括号有效的最少添加 🟠

921. 使括号有效的最少添加

LeetCode	力扣	难度
921. Minimum Add to Make Parentheses Valid	921. 使括号有效的最少添加	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：括号问题**

输入一个括号字符串，返回使其成为合法括号串所需添加的最少括号数。

示例 1：

```
输入："())"
输出：1
解释：添加 1 个左括号成为 (())
```

示例 2： 输入："(((" 输出：3 解释：添加 3 个左括号成为 ()()()

基本思路

核心思路是以左括号为基准，通过维护对右括号的需求数 need，来计算最小的插入次数。

- **详细题解：如何解决括号相关的问题**

解法代码

```
class Solution {
    public int minAddToMakeValid(String s) {
        // res 记录插入次数
        int res = 0;
        // need 变量记录右括号的需求量
        int need = 0;

        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                // 对右括号的需求 + 1
                need++;
            }

            if (s.charAt(i) == ')') {
                // 对右括号的需求 - 1
                need--;

                if (need == -1) {
                    need = 0;
                    // 需插入一个左括号
```

```
                    res++;
                }
            }
        }

        return res + need;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1541. 平衡括号字符串的最少插入次数 🟠
  - 20. 有效的括号 🟢

32. 最长有效括号

LeetCode	力扣	难度
32. Longest Valid Parentheses	32. 最长有效括号	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：括号问题，栈**

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

```
输入：s = "(()(()()("
输出：4
解释：最长有效括号子串是 "()()"
```

基本思路

如果你看过前文 手把手解决三道括号相关的算法题，就知道一般判断括号串是否合法的算法如下：

```
Stack<Integer> stk = new Stack<>();
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '(') {
        // 遇到左括号，记录索引
        stk.push(i);
    } else {
        // 遇到右括号
        if (!stk.isEmpty()) {
            // 配对的左括号对应索引，[leftIndex, i] 是一个合法括号子串
            int leftIndex = stk.pop();
            // 这个合法括号子串的长度
            int len = 1 + i - leftIndex;
        } else {
            // 没有配对的左括号
        }
    }
}
```

但如果多个合法括号子串连在一起，会形成一个更长的合法括号子串，而上述算法无法适配这种情况。

所以需要一个 dp 数组，记录 leftIndex 相邻合法括号子串的长度，才能得出题目想要的正确结果。

解法代码

```java
class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stk = new Stack<>();
        // dp[i] 的定义：记录以 s[i-1] 结尾的最长合法括号子串长度
        int[] dp = new int[s.length() + 1];
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                // 遇到左括号，记录索引
                stk.push(i);
                // 左括号不可能是合法括号子串的结尾
                dp[i + 1] = 0;
            } else {
                // 遇到右括号
                if (!stk.isEmpty()) {
                    // 配对的左括号对应索引
                    int leftIndex = stk.pop();
                    // 以这个右括号结尾的最长子串长度
                    int len = 1 + i - leftIndex + dp[leftIndex];
                    dp[i + 1] = len;
                } else {
                    // 没有配对的左括号
                    dp[i + 1] = 0;
                }
            }
        }
        // 计算最长子串的长度
        int res = 0;
        for (int i = 0; i < dp.length; i++) {
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

71. 简化路径

LeetCode	力扣	难度
71. Simplify Path	71. 简化路径	🟠

- **标签：数据结构，栈**

给你一个字符串 `path`，表示指向某一文件或目录的 Unix 风格 **绝对路径**（以 `'/'` 开头），请你将其转化为更加简洁的规范路径。

在 Unix 风格的文件系统中，一个点（`.`）表示当前目录本身；此外，两个点（`..`） 表示将目录切换到上一级（指向父目录）；两者都可以是复杂相对路径的组成部分。任意多个连续的斜杠（即，`'//'`）都被视为单个斜杠 `'/'`。对于此问题，任何其他格式的点（例如，`'...'`）均被视为文件/目录名称。

请注意，返回的 **规范路径** 必须遵循下述格式：

- 始终以斜杠 `'/'` 开头。
- 两个目录名之间必须只有一个斜杠 `'/'`。
- 最后一个目录名（如果存在）**不能** 以 `'/'` 结尾。
- 此外，规范路径仅包含从根目录到目标文件或目录的路径上的目录（即，不含 `'.'` 或 `'..'`）。

返回简化后得到的 **规范路径**。

示例 1：

```
输入：path = "/home/"
输出："/home"
解释：注意，最后一个目录名后面没有斜杠。
```

基本思路

这题很简单，利用栈先进后出的特性处理上级目录 `..`，最后组装化简后的路径即可。

解法代码

```
class Solution {
    public String simplifyPath(String path) {
        String[] parts = path.split("/");
        Stack<String> stk = new Stack<>();
        // 借助栈计算最终的文件夹路径
        for (String part : parts) {
            if (part.isEmpty() || part.equals(".")) {
                continue;
            }
```

```
            if (part.equals("..")) {
                if (!stk.isEmpty()) stk.pop();
                continue;
            }
            stk.push(part);
        }
        // 栈中存储的文件夹组成路径
        String res = "";
        while (!stk.isEmpty()) {
            res = "/" + stk.pop() + res;
        }
        return res.isEmpty() ? "/" : res;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

150. 逆波兰表达式求值

LeetCode	力扣	难度
150. Evaluate Reverse Polish Notation	150. 逆波兰表达式求值	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，栈

根据逆波兰表示法，求表达式的值。

有效的算符包括 +、-、*、/。每个运算对象可以是整数，也可以是另一个逆波兰表达式，两个整数之间的除法只保留整数部分。

可以保证给定的逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
输入：tokens = ["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：tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
输出：22
解释：该算式转化为常见的中缀算术表达式为：
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

基本思路

逆波兰表达式发明出来就是为了方便计算机运用「栈」进行表达式运算的，其运算规则如下：

按顺序遍历逆波兰表达式中的字符，如果是数字，则放入栈；如果是运算符，则将栈顶的两个元素拿出来进行运算，再将结果放入栈。对于减法和除法，运算顺序别搞反了，栈顶第二个数是被除（减）数。

所以这题很简单，直接按照运算规则借助栈计算表达式结果即可。

解法代码

```java
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stk = new Stack<>();
        for (String token : tokens) {
            if ("+-*/".contains(token)) {
                // 是个运算符，从栈顶拿出两个数字进行运算，运算结果入栈
                int a = stk.pop(), b = stk.pop();
                switch (token) {
                    case "+":
                        stk.push(a + b);
                        break;
                    case "*":
                        stk.push(a * b);
                        break;
                    // 对于减法和除法，顺序别搞反了，第二个数是被除（减）数
                    case "-":
                        stk.push(b - a);
                        break;
                    case "/":
                        stk.push(b / a);
                        break;
                }
            } else {
                // 是个数字，直接入栈即可
                stk.push(Integer.parseInt(token));
            }
        }
        // 最后栈中剩下一个数字，即是计算结果
        return stk.pop();
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 剑指 Offer II 036. 后缀表达式 🟠

剑指 Offer II 036. 后缀表达式

这道题和 150. 逆波兰表达式求值 相同。

155. 最小栈

LeetCode	力扣	难度
155. Min Stack	155. 最小栈	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，栈，设计

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 `MinStack` 类：

- `MinStack()` 初始化堆栈对象。
- `void push(int val)` 将元素 val 推入堆栈。
- `void pop()` 删除堆栈顶部的元素。
- `int top()` 获取堆栈顶部的元素。
- `int getMin()` 获取堆栈中的最小元素。

示例 1:

```
输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

基本思路

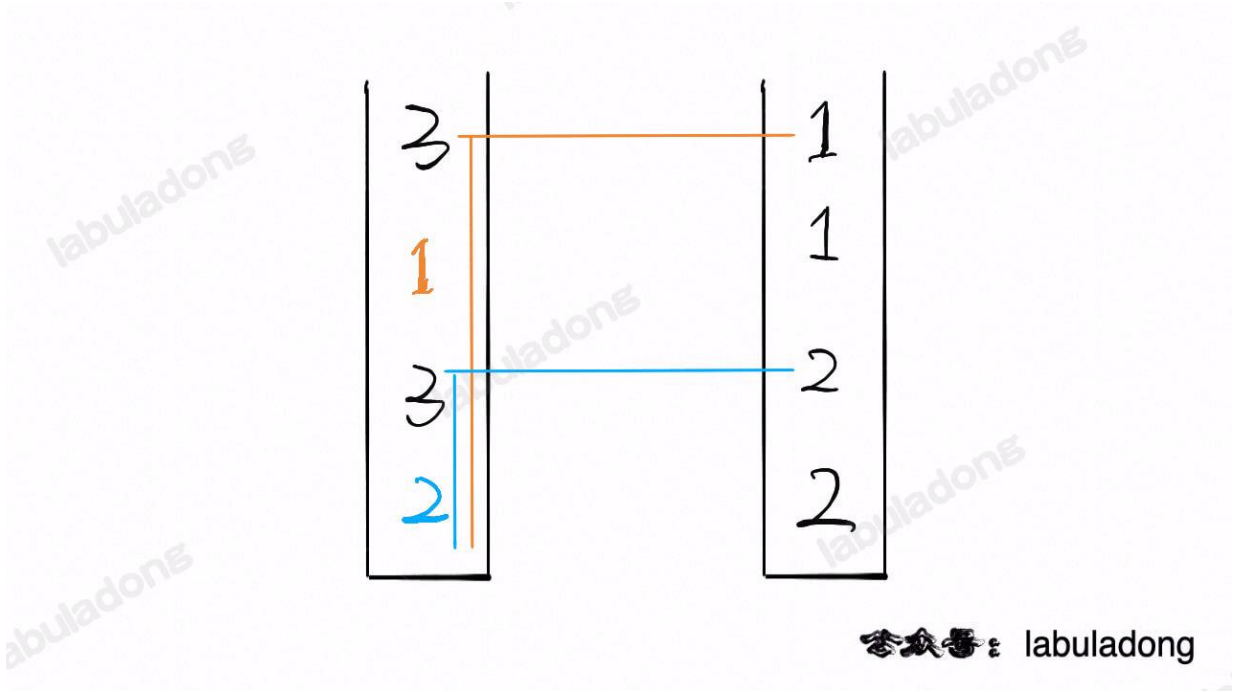
根据我们之前亲自动手实现的栈，我们知道栈是一种操作受限的数据结构，只能从栈顶插入或弹出元素，所以对于标准的栈来说，如果想实现本题的 `getMin` 方法，只能老老实实把所有元素弹出来然后找最小值。**想提高时间效率，那肯定要通过空间换时间的思路**。

不过在具体说解法之前，我想聊一下动态集合中维护最值的问题。这类问题看似简单，但实际上是个很棘手的问题。其实本题就是如下一个场景：

假设你有若干数字，你用一个 min 变量维护了其中的最小值，如果现在给这些数字中添加一个新数字，那么只要比较这个新数字和 min 的大小就可以得出最新的最小值。但如果现在从这些数字钟删除一个数字，你还能用 min 变量得到最小值吗？答案是不能，因为如果这个被删除的数字恰好是最小值，那么新的 min 变量应该更新为第二小的元素对吧，但是我没有记录第二小的元素是多少，所以只能把所有数字重新遍历一遍。

明确了难点再回到本题，就可以对症下药了。删除栈顶元素的时候，不确定新的最小值是多少，但楼下那哥们知道啊，他当时入栈时的最小值，就是现在的最小值呗。

所以这道题的关键就是，**每个元素入栈时，还要记下来当前栈中的最小值**。比方说，可以用一个额外的栈 minStk 来记录栈中每个元素入栈时的栈中的最小元素是多少，这样每次删除元素时就能快速得到剩余栈中的最小元素了。



当然，我们还可以做一些优化，减少 minStk 中存储的元素个数，我把原始解法和优化解法都写出来了，供参考。

> PS：这道题并不难，但我还是很细致地分析了，希望你深刻理解其中的难点。下一步可以做一下 239. 滑动窗口最大值，请仔细观察和思考，队列结构是如何解决这个难点的。

解法代码

```
// 原始思路
class MinStack1 {
    // 记录栈中的所有元素
    Stack<Integer> stk = new Stack<>();
    // 阶段性记录栈中的最小元素
    Stack<Integer> minStk = new Stack<>();

    public void push(int val) {
        stk.push(val);
        // 维护 minStk 栈顶为全栈最小元素
        if (minStk.isEmpty() || val <= minStk.peek()) {
            // 新插入的这个元素就是全栈最小的
```

```
            minStk.push(val);
        } else {
            // 插入的这个元素比较大
            minStk.push(minStk.peek());
        }
    }

    public void pop() {
        stk.pop();
        minStk.pop();
    }

    public int top() {
        return stk.peek();
    }

    public int getMin() {
        // minStk 栈顶为全栈最小元素
        return minStk.peek();
    }
}
// 优化版
class MinStack {
    // 记录栈中的所有元素
    Stack<Integer> stk = new Stack<>();
    // 阶段性记录栈中的最小元素
    Stack<Integer> minStk = new Stack<>();

    public void push(int val) {
        stk.push(val);
        // 维护 minStk 栈顶为全栈最小元素
        if (minStk.isEmpty() || val <= minStk.peek()) {
            // 新插入的这个元素就是全栈最小的
            minStk.push(val);
        }
    }

    public void pop() {
        // 注意 Java 的语言特性，比较 Integer 相等要用 equals 方法
        if (stk.peek().equals(minStk.peek())) {
            // 弹出的元素是全栈最小的
            minStk.pop();
        }
        stk.pop();
    }

    public int top() {
        return stk.peek();
    }

    public int getMin() {
        // minStk 栈顶为全栈最小元素
        return minStk.peek();
```

```
        }
    }
```

- **类似题目**：
    - 239. 滑动窗口最大值 🔴
    - 剑指 Offer 30. 包含min函数的栈 🟢
    - 剑指 Offer 59 - I. 滑动窗口的最大值 🔴

239. 滑动窗口最大值

LeetCode	力扣	难度
239. Sliding Window Maximum	239. 滑动窗口最大值	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，滑动窗口，队列

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧，返回滑动窗口中的最大值。

滑动窗口每次只向右移动一位，你只可以看到在滑动窗口内的 k 个数字。

示例 1：

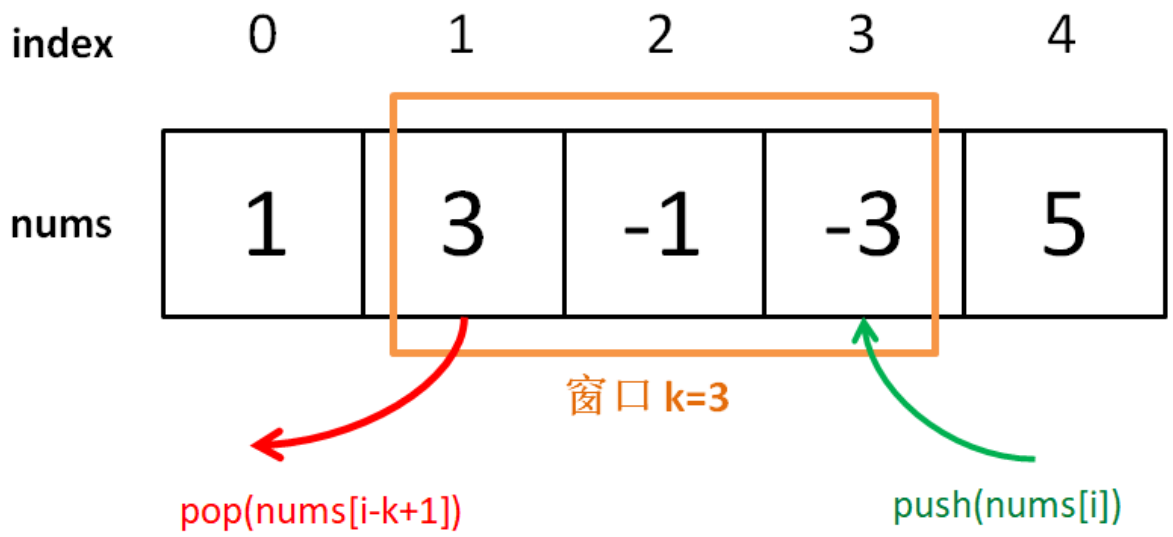
```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

基本思路

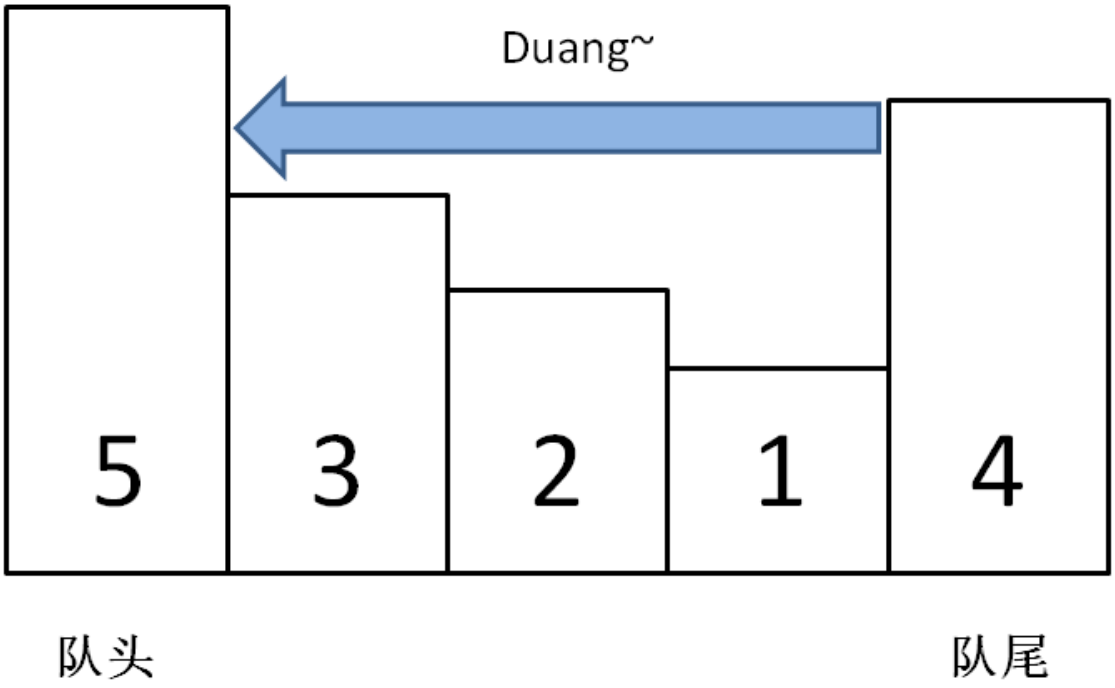
PS：这道题在《算法小抄》 的第 271 页。

> 我在 155. 最小栈 的思路中详细分析了动态集合中维护最值的「千古难题」，如果你没有做，可以先去做一下。我想请你结合这两道题仔细思考：队列和栈分别是如何解决最值维护问题的？

使用一个队列充当不断滑动的窗口，每次滑动记录其中的最大值：



如何在 O(1) 时间计算最大值，只需要一个特殊的数据结构「单调队列」，push 方法依然在队尾添加元素，但是要把前面比自己小的元素都删掉，直到遇到更大的元素才停止删除。



使用单调队列数据结构就能完成本题。

- **详细题解：单调队列结构解决滑动窗口问题**

解法代码

```
class Solution {
    /* 单调队列的实现 */
    class MonotonicQueue {
```

```
        LinkedList<Integer> q = new LinkedList<>();
        public void push(int n) {
            // 将小于 n 的元素全部删除
            while (!q.isEmpty() && q.getLast() < n) {
                q.pollLast();
            }
            // 然后将 n 加入尾部
            q.addLast(n);
        }

        public int max() {
            return q.getFirst();
        }

        public void pop(int n) {
            if (n == q.getFirst()) {
                q.pollFirst();
            }
        }
    }

    /* 解题函数的实现 */
    public int[] maxSlidingWindow(int[] nums, int k) {
        MonotonicQueue window = new MonotonicQueue();
        List<Integer> res = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            if (i < k - 1) {
                //先填满窗口的前 k - 1
                window.push(nums[i]);
            } else {
                // 窗口向前滑动，加入新数字
                window.push(nums[i]);
                // 记录当前窗口的最大值
                res.add(window.max());
                // 移出旧数字
                window.pop(nums[i - k + 1]);
            }
        }
        // 需要转成 int[] 数组再返回
        int[] arr = new int[res.size()];
        for (int i = 0; i < res.size(); i++) {
            arr[i] = res.get(i);
        }
        return arr;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：

- 155. 最小栈 🟠
- 剑指 Offer 30. 包含min函数的栈 🟢
- 剑指 Offer 59 - I. 滑动窗口的最大值 🔴
- 剑指 Offer 59 - II. 队列的最大值 🟠

剑指 Offer 30. 包含min函数的栈

这道题和 155. 最小栈 相同。

剑指 Offer 59 - I. 滑动窗口的最大值

这道题和 239. 滑动窗口最大值 相同。

225. 用队列实现栈

LeetCode	力扣	难度
225. Implement Stack using Queues	225. 用队列实现栈	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，栈，队列

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（`push` 、 `top` 、 `pop` 和 `empty`）。

实现 `MyStack` 类：

1、 `void push(int x)` 将元素 x 压入栈顶。

2、 `int pop()` 移除并返回栈顶元素。

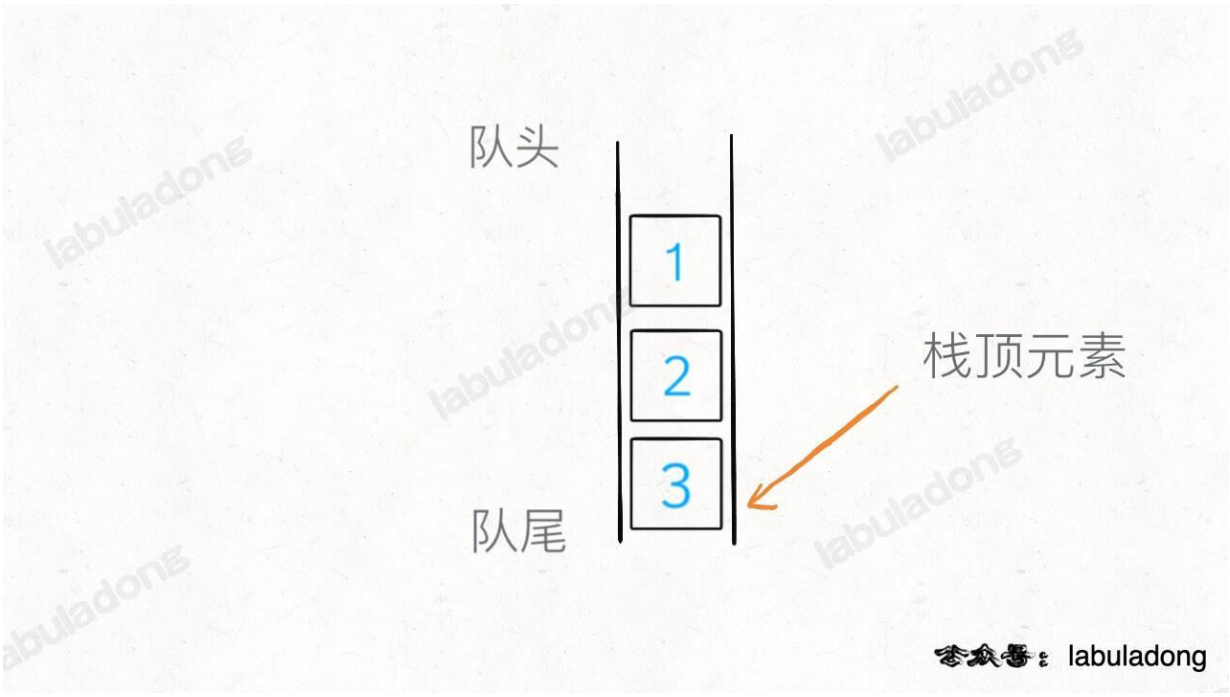
3、 `int top()` 返回栈顶元素。

4、 `boolean empty()` 如果栈是空的，返回 `true`；否则，返回 `false`。

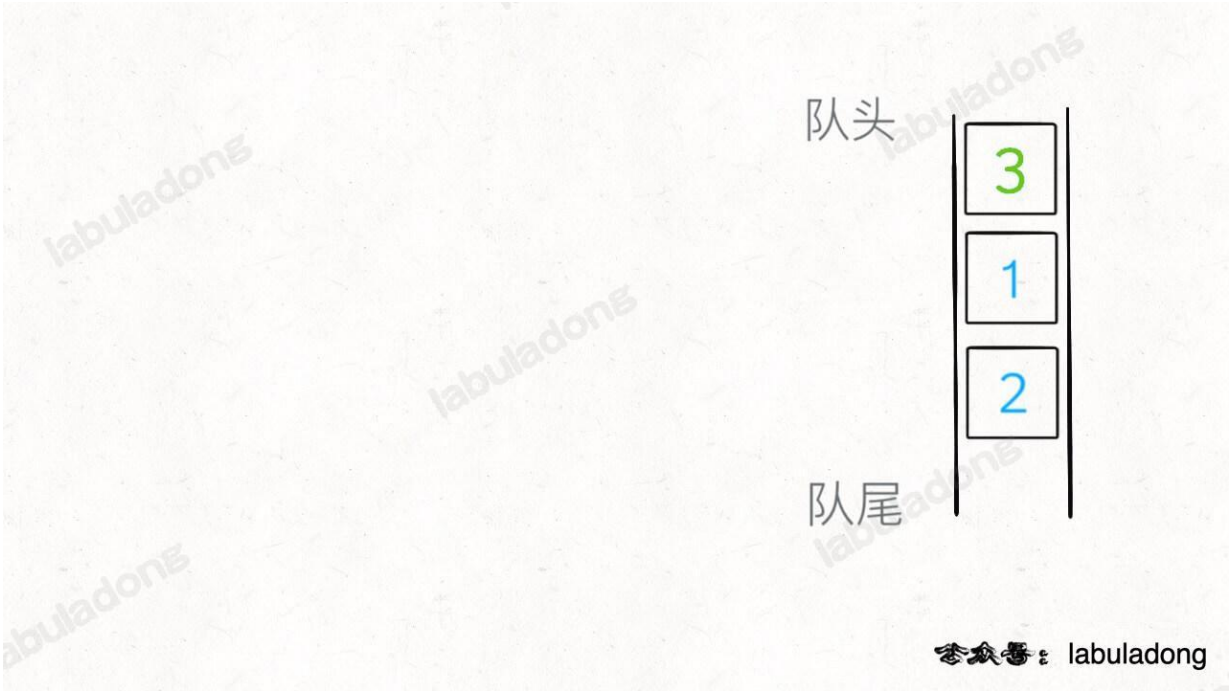
基本思路

用队列实现栈就比较简单粗暴了，只需要一个队列作为底层数据结构。

底层队列只能向队尾添加元素，所以栈的 `pop` API 相当于要从队尾取元素：



那就把队尾元素前面的所有元素重新塞到队尾，让队尾元素排到队头，这样就可以取出了：



- **详细题解：队列实现栈以及栈实现队列**

解法代码

```
class MyStack {
    Queue<Integer> q = new LinkedList<>();
    int top_elem = 0;

    /**
     * 添加元素到栈顶
     */
    public void push(int x) {
        // x 是队列的队尾，是栈的栈顶
        q.offer(x);
        top_elem = x;
    }

    /**
     * 返回栈顶元素
     */
    public int top() {
        return top_elem;
    }

    /**
     * 删除栈顶的元素并返回
     */
    public int pop() {
        int size = q.size();
        // 留下队尾 2 个元素
        while (size > 2) {
            q.offer(q.poll());
            size--;
```

```
            }
            // 记录新的队尾元素
            top_elem = q.peek();
            q.offer(q.poll());
            // 删除之前的队尾元素
            return q.poll();
        }

        /**
         * 判断栈是否为空
         */
        public boolean empty() {
            return q.isEmpty();
        }
}
```

- **类似题目**：
  - 232. 用栈实现队列 🟢
  - 剑指 Offer 09. 用两个栈实现队列 🟢

232. 用栈实现队列

LeetCode	力扣	难度
232. Implement Queue using Stacks	232. 用栈实现队列	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，栈，队列

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（`push` 、 `pop` 、 `peek` 、 `empty`）：

实现 `MyQueue` 类：

1、 `void push(int x)` 将元素 x 推到队列的末尾

2、 `int pop()` 从队列的开头移除并返回元素

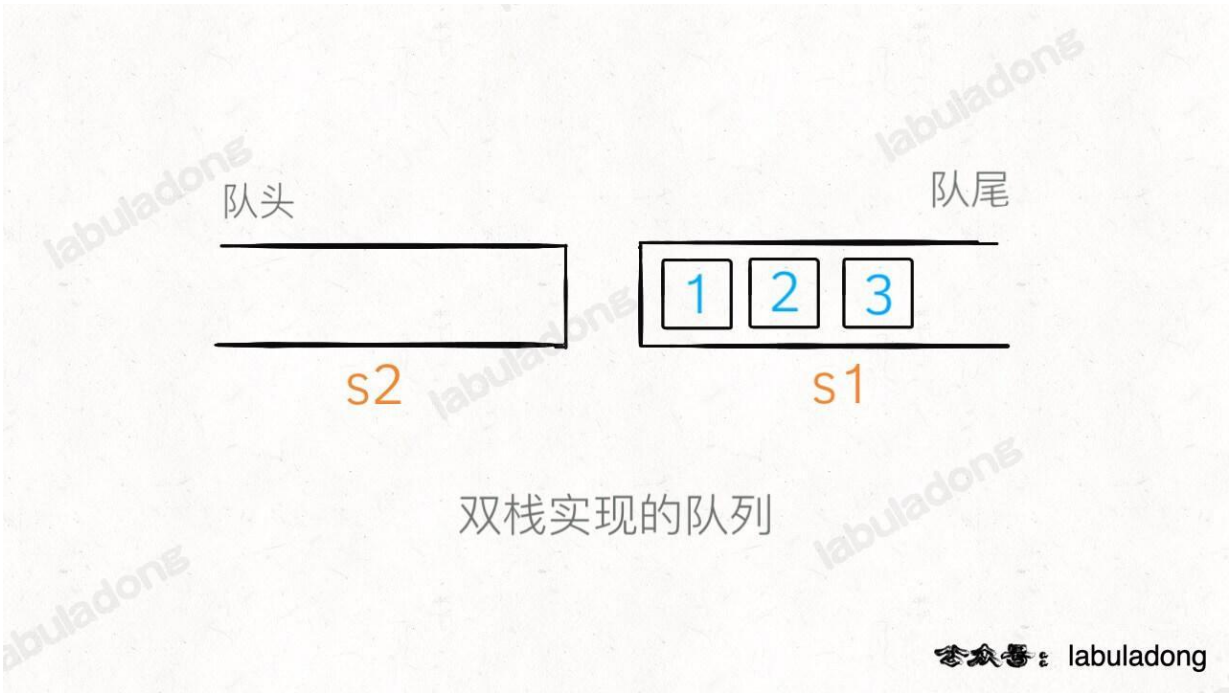
3、 `int peek()` 返回队列开头的元素

4、 `boolean empty()` 如果队列为空，返回 `true`；否则，返回 `false`

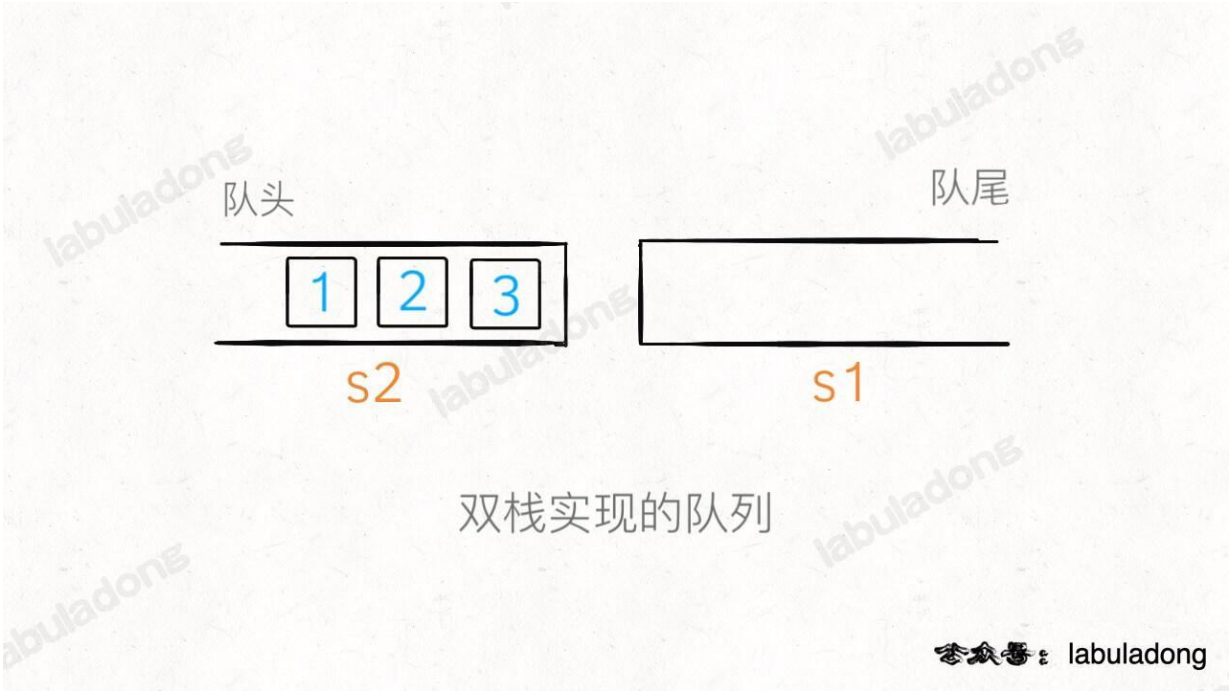
基本思路

我们使用两个栈 `s1, s2` 就能实现一个队列的功能。

当调用 `push` 让元素入队时，只要把元素压入 `s1` 即可：



使用 `peek` 或 `pop` 操作队头的元素时，若 `s2` 为空，可以把 `s1` 的所有元素取出再添加进 `s2`，**这时候 `s2` 中元素就是先进先出顺序了**：



- **详细题解：队列实现栈以及栈实现队列**

解法代码

```
class MyQueue {
    private Stack<Integer> s1, s2;

    public MyQueue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }

    /**
     * 添加元素到队尾
     */
    public void push(int x) {
        s1.push(x);
    }

    /**
     * 删除队头的元素并返回
     */
    public int pop() {
        // 先调用 peek 保证 s2 非空
        peek();
        return s2.pop();
    }

    /**
     * 返回队头元素
     */
    public int peek() {
        if (s2.isEmpty())
```

```
            // 把 s1 元素压入 s2
            while (!s1.isEmpty())
                s2.push(s1.pop());
        return s2.peek();
    }

    /**
     * 判断队列是否为空
     */
    public boolean empty() {
        return s1.isEmpty() && s2.isEmpty();
    }
}
```

- **类似题目**：
  - 225. 用队列实现栈 🟢
  - 剑指 Offer 09. 用两个栈实现队列 🟢

剑指 Offer 09. 用两个栈实现队列

这道题和 232. 用栈实现队列 相同。

445. 两数相加 II

LeetCode	力扣	难度
445. Add Two Numbers II	445. 两数相加 II	🟠

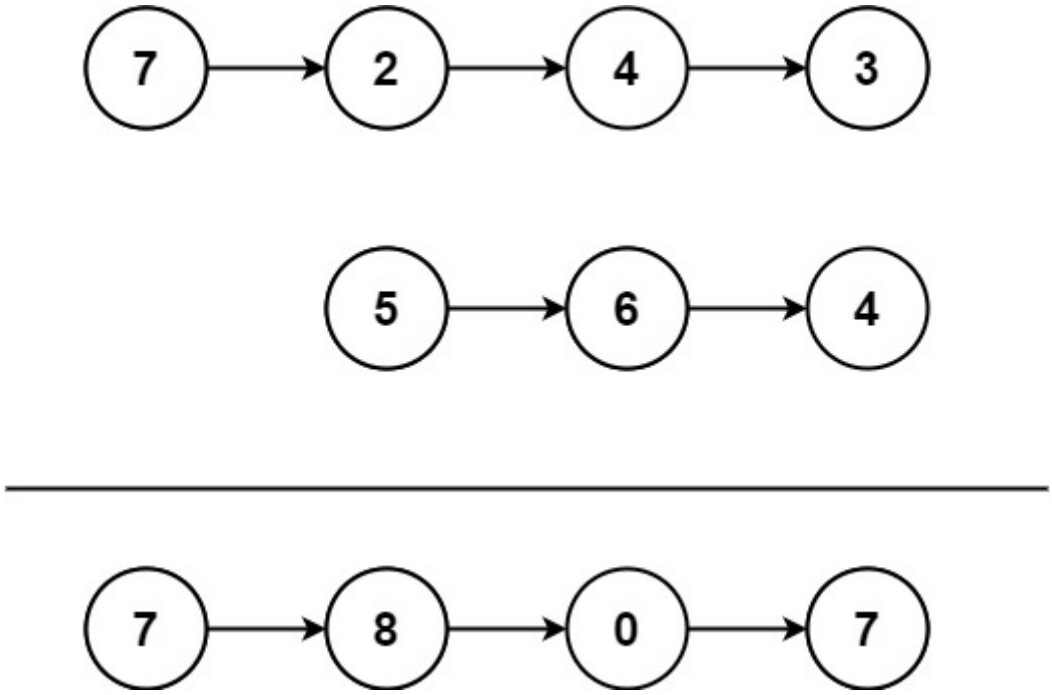
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：栈，链表

给你两个 **非空**链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例 1：




```
输入：l1 = [7,2,4,3], l2 = [5,6,4]
输出：[7,8,0,7]
```

示例 2：

```
输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[8,0,7]
```

示例 3：

```
输入：l1 = [0], l2 = [0]
输出：[0]
```

基本思路

这道题是 2. 两数相加 的进阶问题，我们模拟加法运算当然是从最低位开始加，这样才能正确的处理进位。但现在单链表的开头是最高位，那么最直接的想法就是先 翻转链表，这样就可以继续玩第 2 题那一套了，没什么难度。

不过本题也说了，如果不让你反转链表怎么办？其实也好办，我们可以利用栈这种先进后出的数据结构，把链表节点从头到尾放进栈中，再从栈拿出来就是从尾到头的顺序，相当于是反转链表的效果，然后又回到了第 2 题的加法逻辑。

还有一个需要注意的是，计算结果的高位也应该放在结果链表的左侧，也就是插入到 dummy 节点的后面。具体看代码吧。

解法代码

```
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        // 把链表元素转入栈中
        Stack<Integer> stk1 = new Stack<>();
        while (l1 != null) {
            stk1.push(l1.val);
            l1 = l1.next;
        }
        Stack<Integer> stk2 = new Stack<>();
        while (l2 != null) {
            stk2.push(l2.val);
            l2 = l2.next;
        }

        // 接下来基本上是复用我在第 2 题的代码逻辑
        // 注意新节点要直接插入到 dummy 后面

        // 虚拟头结点（构建新链表时的常用技巧）
        ListNode dummy = new ListNode(-1);

        // 记录进位
        int carry = 0;
        // 开始执行加法，两条链表走完且没有进位时才能结束循环
        while (!stk1.isEmpty() || !stk2.isEmpty() || carry > 0) {
            // 先加上上次的进位
            int val = carry;
            if (!stk1.isEmpty()) {
                val += stk1.pop();
            }
            if (!stk2.isEmpty()) {
                val += stk2.pop();
```

```
                }
                // 处理进位情况
                carry = val / 10;
                val = val % 10;
                // 构建新节点，直接接在 dummy 后面
                ListNode newNode = new ListNode(val);
                newNode.next = dummy.next;
                dummy.next = newNode;
            }
            // 返回结果链表的头结点（去除虚拟头结点）
            return dummy.next;
        }
    }
```

▶ 🌟 **代码可视化动画（需安装 Chrome 插件）**🌟

- **类似题目**：
  - 剑指 Offer II 025. 链表中的两数相加 🟠

剑指 Offer II 025. 链表中的两数相加

这道题和 445. 两数相加 II 相同。

剑指 Offer 06. 从尾到头打印链表

LeetCode	力扣	难度
剑指Offer06. 从尾到头打印链表 LCOF	剑指Offer06. 从尾到头打印链表	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：栈，链表

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

```
输入：head = [1,3,2]
输出：[2,3,1]
```

基本思路

这题解法很多，比如我们可以借助「栈」这种先进后出的结构来得到链表的倒序遍历结果，但是本文使用 东哥手把手带你刷二叉树（纲领篇） 讲到的后序遍历技巧来写代码。

递归函数本质上就是让操作系统帮我们维护了递归栈，和栈的解法效率上完全相同，但是这样写代码有助于我们深入理解递归的思维。

解法代码

```
// 用「遍历」的思路写递归解法
class Solution {
    public int[] reversePrint(ListNode head) {
        traverse(head);
        return res;
    }

    // 记录链表长度
    int len = 0;
    // 结果数组
    int[] res;
    // 结果数组中的指针
    int p = 0;

    // 递归遍历单链表
    void traverse(ListNode head) {
        if (head == null) {
            // 到达链表尾部，此时知道了链表的总长度
            // 创建结果数组
            res = new int[len];
```

```
            return;
        }
        len++;
        traverse(head.next);
        // 后序位置，可以倒序操作链表
        res[p] = head.val;
        p++;
    }



    // 用「分解问题」的思路写递归解法
    // 因为 Java 的 int[] 数组不支持 add 相关的操作，所以我们把返回值修改成 List
    // 定义：输入一个单链表，返回该链表翻转的值，示例 1->2->3
    List<Integer> reversePrint2(ListNode head) {
        // base case
        if (head == null) {
            return new LinkedList<>();
        }

        // 把子链表翻转的结果算出来，示例 [3,2]
        List<Integer> subProblem = reversePrint2(head.next);
        // 把 head 的值接到子链表的翻转结果的尾部，示例 [3,2,1]
        subProblem.add(head.val);
        return subProblem;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

23. 合并 K 个升序链表

LeetCode	力扣	难度
23. Merge k Sorted Lists	23. 合并K个升序链表	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数据结构，链表，链表双指针**

给你一个链表数组，每个链表都已经按升序排列，请你将这些链表合并成一个升序链表，返回合并后的链表。

示例 1：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

基本思路

> 本文有视频版：链表双指针技巧全面汇总

21. 合并两个有序链表 的延伸，利用 优先级队列（二叉堆） 进行节点排序即可。

- **详细题解：双指针技巧秒杀七道链表题目**

解法代码

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists.length == 0) return null;
        // 虚拟头结点
        ListNode dummy = new ListNode(-1);
        ListNode p = dummy;
        // 优先级队列，最小堆
        PriorityQueue<ListNode> pq = new PriorityQueue<>(
            lists.length, (a, b)->(a.val - b.val));
        // 将 k 个链表的头结点加入最小堆
        for (ListNode head : lists) {
```

```
                if (head != null)
                    pq.add(head);
            }

            while (!pq.isEmpty()) {
                // 获取最小节点，接到结果链表中
                ListNode node = pq.poll();
                p.next = node;
                if (node.next != null) {
                    pq.add(node.next);
                }
                // p 指针不断前进
                p = p.next;
            }
            return dummy.next;
        }
    }
```

- **类似题目**：
  - 141. 环形链表 🟢
  - 142. 环形链表 II 🟠
  - 160. 相交链表 🟢
  - 19. 删除链表的倒数第 N 个结点 🟠
  - 21. 合并两个有序链表 🟢
  - 313. 超级丑数 🟠
  - 355. 设计推特 🟠
  - 373. 查找和最小的 K 对数字 🟠
  - 378. 有序矩阵中第 K 小的元素 🟠
  - 86. 分隔链表 🟠
  - 876. 链表的中间结点 🟢
  - 剑指 Offer 22. 链表中倒数第k个节点 🟢
  - 剑指 Offer 25. 合并两个排序的链表 🟢
  - 剑指 Offer 52. 两个链表的第一个公共节点 🟢
  - 剑指 Offer II 021. 删除链表的倒数第 n 个结点 🟠
  - 剑指 Offer II 022. 链表中环的入口节点 🟠
  - 剑指 Offer II 023. 两个链表的第一个重合节点 🟢
  - 剑指 Offer II 061. 和最小的 k 个数对 🟠
  - 剑指 Offer II 078. 合并排序链表 🔴

313. 超级丑数

LeetCode	力扣	难度
313. Super Ugly Number	313. 超级丑数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，链表双指针**

超级丑数 是一个正整数，并满足其所有质因数都出现在质数数组 `primes` 中。给你一个整数 `n` 和一个整数数组 `primes`，返回第 `n` 个 **超级丑数**。

示例 1：

```
输入：n = 12, primes = [2,7,13,19]
输出：32
解释：给定长度为 4 的质数数组 primes = [2,7,13,19]，前 12 个超级丑数序列为：
[1,2,4,7,8,13,14,16,19,26,28,32]。
```

基本思路

这题是 264. 丑数 II 的进阶版，第 264 题是 21. 合并两个有序链表（简单） 的变体，而这道题是 23. 合并K个升序链表（困难） 的变体，我在 双指针技巧秒杀七道链表题目 中都讲过。

你一定要先做下 264 题，注意那里我们抽象出了三条链表，需要 `p2, p3, p5` 作为三条有序链表上的指针，同时需要 `product2, product3, product5` 记录指针所指节点的值，用 `min` 函数计算最小头结点。

这道题相当于输入了 `len(primes)` 条有序链表，我们不能用 `min` 函数计算最小头结点了，而是要用优先级队列来计算最小头结点，同时依然要维护链表指针、指针所指节点的值，我们把这些信息用一个三元组来保存。

结合第 23 题的解法逻辑，你应该能够看懂这道题的解法代码了。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        // 优先队列中装三元组 int[] {product, prime, pi}
        // 其中 product 代表链表节点的值，prime 是计算下一个节点所需的质数因子，pi
代表链表上的指针
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            return a[0] - b[0];
        });
```

```
        // 把多条链表的头结点加入优先级队列
        for (int i = 0; i < primes.length; i++) {
            pq.offer(new int[]{ 1, primes[i], 1 });
        }

        // 可以理解为最终合并的有序链表（结果链表）
        int[] ugly = new int[n + 1];
        // 可以理解为结果链表上的指针
        int p = 1;

        while (p <= n) {
            // 取三个链表的最小结点
            int[] pair = pq.poll();
            int product = pair[0];
            int prime = pair[1];
            int index = pair[2];

            // 避免结果链表出现重复元素
            if (product != ugly[p - 1]) {
                // 接到结果链表上
                ugly[p] = product;
                p++;
            }

            // 生成下一个节点加入优先级队列
            int[] nextPair = new int[]{ugly[index] * prime, prime, index +
1};
            pq.offer(nextPair);
        }
        return ugly[n];
    }
}
```

- **类似题目**：
  - 1201. 丑数 III 🟠
  - 263. 丑数 🟢
  - 264. 丑数 II 🟠
  - 剑指 Offer 49. 丑数 🟠

355. 设计推特

LeetCode	力扣	难度
355. Design Twitter	355. 设计推特	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，设计

设计一个简化版的推特 (Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近 10 条推文。

实现 Twitter 类：

- void postTweet(int userId, int tweetId) 根据给定的 tweetId 和 userId 创建一条新推文。每次调用此函数都会使用一个不同的 tweetId。
- List<Integer> getNewsFeed(int userId) 检索当前用户新闻推送中最近 10 条推文的 ID。新闻推送中的每一项都必须是由用户关注的人或者是用户自己发布的推文。推文必须**按照时间顺序由最近到最远排序**。
- void follow(int followerId, int followeeId) ID 为 followerId 的用户开始关注 ID 为 followeeId 的用户。
- void unfollow(int followerId, int followeeId) ID 为 followerId 的用户不再关注 ID 为 followeeId 的用户。

示例：

```
输入
["Twitter", "postTweet", "getNewsFeed", "follow", "postTweet",
"getNewsFeed", "unfollow", "getNewsFeed"]
[[], [1, 5], [1], [1, 2], [2, 6], [1], [1, 2], [1]]
输出
[null, null, [5], null, null, [6, 5], null, [5]]

解释
Twitter twitter = new Twitter();
twitter.postTweet(1, 5); // 用户 1 发送了一条新推文 (用户 id = 1, 推文 id = 5)
twitter.getNewsFeed(1);  // 用户 1 的获取推文应当返回一个列表，其中包含一个 id 为
5 的推文
twitter.follow(1, 2);    // 用户 1 关注了用户 2
twitter.postTweet(2, 6); // 用户 2 发送了一个新推文 (推文 id = 6)
twitter.getNewsFeed(1);  // 用户 1 的获取推文应当返回一个列表，其中包含两个推文，id
分别为 -> [6, 5]。推文 id 6 应当在推文 id 5 之前，因为它是在 5 之后发送的
twitter.unfollow(1, 2);  // 用户 1 取消关注了用户 2
twitter.getNewsFeed(1);  // 用户 1 获取推文应当返回一个列表，其中包含一个 id 为 5
的推文。因为用户 1 已经不再关注用户 2
```

基本思路

这道题比较经典，在特定场景下让你设计算法。其难点在于 `getNewsFeed` 方法，要按照时间线顺序展示所有关注用户的推文，这个方法要用到我在 单链表的六大解题套路 解决 23. 合并K个升序链表 的合并多个有序链表的技巧：

你把一个用户发布的所有推文做成一条有序链表（靠近头部的推文是最近发布的），那么只要合并关注用户的推文链表，即可获得按照时间线顺序排序的信息流。

具体看代码吧，我注释比较详细。

- **详细题解：设计朋友圈时间线功能**

解法代码

```
class Twitter {
    // 全局时间戳
    int globalTime = 0;
    // 记录用户 ID 到用户示例的映射
    HashMap<Integer, User> idToUser = new HashMap<>();

    // Tweet 类
    class Tweet {
        private int id;
        // 时间戳用于对信息流按照时间排序
        private int timestamp;
        // 指向下一条 tweet，类似单链表结构
        private Tweet next;

        public Tweet(int id) {
            this.id = id;
            // 新建一条 tweet 时记录并更新时间戳
            this.timestamp = globalTime++;
        }

        public int getId() {
            return id;
        }

        public int getTimestamp() {
            return timestamp;
        }

        public Tweet getNext() {
            return next;
        }

        public void setNext(Tweet next) {
            this.next = next;
        }
    }
```

```
    // 用户类
    class User {
        // 记录该用户的 id 以及发布的 tweet
        private int id;
        private Tweet tweetHead;
        // 记录该用户的关注者
        private HashSet<User> followedUserSet;

        public User(int id) {
            this.id = id;
            this.tweetHead = null;
            this.followedUserSet = new HashSet<>();
        }

        public int getId() {
            return id;
        }

        public Tweet getTweetHead() {
            return tweetHead;
        }

        public HashSet<User> getFollowedUserSet() {
            return followedUserSet;
        }

        public boolean equals(User other) {
            return this.id == other.id;
        }

        // 关注其他人
        public void follow(User other) {
            followedUserSet.add(other);
        }

        // 取关其他人
        public void unfollow(User other) {
            followedUserSet.remove(other);
        }

        // 发布一条 tweet
        public void post(Tweet tweet) {
            // 把新发布的 tweet 作为链表头节点
            tweet.setNext(tweetHead);
            tweetHead = tweet;
        }
    }

    public void postTweet(int userId, int tweetId) {
        // 如果这个用户还不存在，新建用户
        if (!idToUser.containsKey(userId)) {
            idToUser.put(userId, new User(userId));
        }
        User user = idToUser.get(userId);
```

```
        user.post(new Tweet(tweetId));
    }

    public List<Integer> getNewsFeed(int userId) {
        List<Integer> res = new LinkedList<>();
        if (!idToUser.containsKey(userId)) {
            return res;
        }
        // 获取该用户关注的用户列表
        User user = idToUser.get(userId);
        Set<User> followedUserSet = user.getFollowedUserSet();
        // 每个用户的 tweet 是一条按时间排序的链表
        // 现在执行合并多条有序链表的逻辑，找出时间线中的最近 10 条动态
        PriorityQueue<Tweet> pq = new PriorityQueue<>((a, b) -> {
            // 按照每条 tweet 的发布时间降序排序（最近发布的排在事件流前面）
            return b.timestamp - a.timestamp;
        });
        // 该用户自己的 tweet 也在时间线内
        if (user.getTweetHead() != null) {
            pq.offer(user.getTweetHead());
        }
        for (User other : followedUserSet) {
            if (other.getTweetHead() != null) {
                pq.offer(other.tweetHead);
            }
        }
        // 合并多条有序链表
        int count = 0;
        while (!pq.isEmpty() && count < 10) {
            Tweet tweet = pq.poll();
            res.add(tweet.getId());
            if (tweet.getNext() != null) {
                pq.offer(tweet.getNext());
            }
            count++;
        }
        return res;
    }

    public void follow(int followerId, int followeeId) {
        // 如果用户还不存在，则新建用户
        if (!idToUser.containsKey(followerId)) {
            idToUser.put(followerId, new User(followerId));
        }
        if (!idToUser.containsKey(followeeId)) {
            idToUser.put(followeeId, new User(followeeId));
        }

        User follower = idToUser.get(followerId);
        User followee = idToUser.get(followeeId);
        // 关注者关注被关注者
        follower.follow(followee);
    }
```

```
    public void unfollow(int followerId, int followeeId) {
        if (!idToUser.containsKey(followerId) ||
!idToUser.containsKey(followeeId)) {
            return;
        }
        User follower = idToUser.get(followerId);
        User followee = idToUser.get(followeeId);
        // 关注者取关被关注者
        follower.unfollow(followee);
    }
}
```

373. 查找和最小的 K 对数字

LeetCode	力扣	难度
373. Find K Pairs with Smallest Sums	373. 查找和最小的K对数字	🟠

- **标签：二叉堆，链表双指针**

给定两个以 **升序排列** 的整数数组 `nums1` 和 `nums2`, 以及一个整数 `k`。

定义一对值 `(u,v)`，其中第一个元素来自 `nums1`，第二个元素来自 `nums2`。

请找到**和**最小的 `k` 个数对 `(u1,v1)`, `(u2,v2)` ... `(uk,vk)`。

示例 1:

```
输入: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
输出: [1,2],[1,4],[1,6]
解释: 返回序列中的前 3 对数:
     [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
```

基本思路

这道题其实是前文 单链表的六大解题套路 中讲过的 23. 合并K个升序链表 的变体。

怎么把这道题变成合并多个有序链表呢？就比如说题目输入的用例：

```
nums1 = [1,7,11], nums2 = [2,4,6]
```

组合出的所有数对儿这就可以抽象成三个有序链表：

```
[1, 2] -> [1, 4] -> [1, 6]
[7, 2] -> [7, 4] -> [7, 6]
[11, 2] -> [11, 4] -> [11, 6]
```

这三个链表中每个元素（数对之和）是递增的，所以就可以按照 23. 合并K个升序链表 的思路来合并，取出前 `k` 个作为答案即可。

解法代码

```
class Solution {
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2,
int k) {
        // 存储三元组 (num1[i], nums2[i], i)
        // i 记录 nums2 元素的索引位置，用于生成下一个节点
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            // 按照数对的元素和升序排序
            return (a[0] + a[1]) - (b[0] + b[1]);
        });
        // 按照 23 题的逻辑初始化优先级队列
        for (int i = 0; i < nums1.length; i++) {
            pq.offer(new int[]{nums1[i], nums2[0], 0});
        }

        List<List<Integer>> res = new ArrayList<>();
        // 执行合并多个有序链表的逻辑
        while (!pq.isEmpty() && k > 0) {
            int[] cur = pq.poll();
            k--;
            // 链表中的下一个节点加入优先级队列
            int next_index = cur[2] + 1;
            if (next_index < nums2.length) {
                pq.add(new int[]{cur[0], nums2[next_index], next_index});
            }

            List<Integer> pair = new ArrayList<>();
            pair.add(cur[0]);
            pair.add(cur[1]);
            res.add(pair);
        }
        return res;
    }
}
```

- **类似题目**：
  - 剑指 Offer II 061. 和最小的 k 个数对 🟠

378. 有序矩阵中第 K 小的元素

LeetCode	力扣	难度
378. Kth Smallest Element in a Sorted Matrix	378. 有序矩阵中第 K 小的元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，链表双指针**

给你一个 `n x n` 矩阵 `matrix`，其中每行和每列元素均按升序排序，找到矩阵中第 `k` 小的元素。
请注意，它是 **排序后** 的第 `k` 小元素，而不是第 `k` 个 **不同**的元素。

你必须找到一个内存复杂度优于 `O(n2)` 的解决方案。

示例 1：

```
输入：matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
输出：13
解释：矩阵中的元素为 [1,5,9,10,11,12,13,13,15]，第 8 小元素是 13
```

基本思路

这道题其实是前文 单链表的六大解题套路 中讲过的 23. 合并K个升序链表 的变体。

矩阵中的每一行都是排好序的，就好比多条有序链表，你用优先级队列施展合并多条有序链表的逻辑就能找到第 `k` 小的元素了。

解法代码

```
class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        // 存储二元组 (matrix[i][j], i, j)
        // i, j 记录当前元素的索引位置，用于生成下一个节点
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            // 按照元素大小升序排序
            return a[0] - b[0];
        });


        // 初始化优先级队列，把每一行的第一个元素装进去
        for (int i = 0; i < matrix.length; i++) {
            pq.offer(new int[]{matrix[i][0], i, 0});
        }

        int res = -1;
        // 执行合并多个有序链表的逻辑，找到第 k 小的元素
```

```
        while (!pq.isEmpty() && k > 0) {
            int[] cur = pq.poll();
            res = cur[0];
            k--;
            // 链表中的下一个节点加入优先级队列
            int i = cur[1], j = cur[2];
            if (j + 1 < matrix[i].length) {
                pq.add(new int[]{matrix[i][j + 1], i, j + 1});
            }
        }
        return res;
    }
}
```

剑指 Offer II 061. 和最小的 k 个数对

这道题和 373. 查找和最小的 K 对数字 相同。

剑指 Offer II 078. 合并排序链表

这道题和 23. 合并 K 个升序链表 相同。

215. 数组中的第 K 个最大元素

LeetCode	力扣	难度
215. Kth Largest Element in an Array	215. 数组中的第K个最大元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，快速选择算法，数组**

给定整数数组 `nums` 和整数 `k`，请返回数组中第 `k` 个最大的元素。

请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

示例 1:

```
输入：[3,2,1,5,6,4] 和 k = 2
输出：5
```

示例 2:

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

基本思路

二叉堆的解法比较简单，实际写算法题的时候，推荐大家写这种解法。

可以把小顶堆 `pq` 理解成一个筛子，较大的元素会沉淀下去，较小的元素会浮上来；当堆大小超过 `k` 的时候，我们就删掉堆顶的元素，因为这些元素比较小，而我们想要的是前 `k` 个最大元素嘛。

当 `nums` 中的所有元素都过了一遍之后，筛子里面留下的就是最大的 `k` 个元素，而堆顶元素是堆中最小的元素，也就是「第 `k` 个最大的元素」。

二叉堆插入和删除的时间复杂度和堆中的元素个数有关，在这里我们堆的大小不会超过 `k`，所以插入和删除元素的复杂度是 `O(logK)`，再套一层 for 循环，总的时间复杂度就是 `O(NlogK)`。

当然，这道题可以有效率更高的解法叫「快速选择算法」，只需要 `O(N)` 的时间复杂度。

快速选择算法不用借助二叉堆结构，而是稍微改造了快速排序的算法思路，有兴趣的读者可以看详细题解。

- **详细题解：快速排序详解及应用**

解法代码

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        // 小顶堆，堆顶是最小元素
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int e : nums) {
            // 每个元素都要过一遍二叉堆
            pq.offer(e);
            // 堆中元素多于 k 个时，删除堆顶元素
            if (pq.size() > k) {
                pq.poll();
            }
        }
        // pq 中剩下的是 nums 中 k 个最大元素，
        // 堆顶是最小的那个，即第 k 个最大元素
        return pq.peek();
    }
}
```

- **类似题目**：
  - 347. 前 K 个高频元素 🟠
  - 703. 数据流中的第 K 大元素 🟢
  - 912. 排序数组 🟠
  - 剑指 Offer 40. 最小的k个数 🟢
  - 剑指 Offer II 059. 数据流的第 K 大数值 🟢
  - 剑指 Offer II 060. 出现频率最高的 k 个数字 🟠
  - 剑指 Offer II 076. 数组中的第 k 大的数字 🟠

347. 前 K 个高频元素

LeetCode	力扣	难度
347. Top K Frequent Elements	347. 前 K 个高频元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，哈希表，快速选择**

给你一个整数数组 `nums` 和一个整数 `k`，请你返回其中出现频率前 `k` 高的元素。你可以按 **任意顺序** 返回答案。

示例 1:

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

基本思路

首先，肯定要用一个 `valToFreq` 哈希表把每个元素出现的频率计算出来。

然后，这道题就变成了 215. 数组中的第 K 个最大元素，只不过第 215 题让你求数组中元素值 `e` 排在第 `k` 大的那个元素，这道题让你求数组中元素值 `valToFreq[e]` 排在前 `k` 个的元素。

我在 快速排序详解及运用 中讲过第 215 题，可以用 优先级队列 或者快速选择算法解决这道题。这里稍微改一下优先级队列的比较函数，或者改一下快速选择算法中的逻辑即可。

这里我再加一种解法，用计数排序的方式找到前 `k` 个高频元素，见代码。

解法代码

```
// 用优先级队列解决这道题
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        // nums 中的元素 -> 该元素出现的频率
        HashMap<Integer, Integer> valToFreq = new HashMap<>();
        for (int v : nums) {
            valToFreq.put(v, valToFreq.getOrDefault(v, 0) + 1);
        }

        PriorityQueue<Map.Entry<Integer, Integer>>
                pq = new PriorityQueue<>((entry1, entry2) -> {
            // 队列按照键值对中的值（元素出现频率）从小到大排序
            return entry1.getValue().compareTo(entry2.getValue());
        });
```

```
        for (Map.Entry<Integer, Integer> entry : valToFreq.entrySet()) {
            pq.offer(entry);
            if (pq.size() > k) {
                // 弹出最小元素，维护队列内是 k 个频率最大的元素
                pq.poll();
            }
        }

        int[] res = new int[k];
        for (int i = k - 1; i >= 0; i--) {
            // res 数组中存储前 k 个最大元素
            res[i] = pq.poll().getKey();
        }

        return res;
    }
}

// 用计数排序的方法解决这道题
class Solution2 {
    public int[] topKFrequent(int[] nums, int k) {
        // nums 中的元素 -> 该元素出现的频率
        HashMap<Integer, Integer> valToFreq = new HashMap<>();
        for (int v : nums) {
            valToFreq.put(v, valToFreq.getOrDefault(v, 0) + 1);
        }

        // 频率 -> 这个频率有哪些元素
        ArrayList<Integer>[] freqToVals = new ArrayList[nums.length + 1];
        for (int val : valToFreq.keySet()) {
            int freq = valToFreq.get(val);
            if (freqToVals[freq] == null) {
                freqToVals[freq] = new ArrayList<>();
            }
            freqToVals[freq].add(val);
        }

        int[] res = new int[k];
        int p = 0;
        // freqToVals 从后往前存储着出现最多的元素
        for (int i = freqToVals.length - 1; i > 0; i--) {
            ArrayList<Integer> valList = freqToVals[i];
            if (valList == null) continue;
            for (int j = 0; j < valList.size(); j++) {
                // 将出现次数最多的 k 个元素装入 res
                res[p] = valList.get(j);
                p++;
                if (p == k) {
                    return res;
                }
            }
        }

        return null;
```

```
        }
    }
```

- **类似题目**：
    - 692. 前K个高频单词 🟠
    - 剑指 Offer II 060. 出现频率最高的 k 个数字 🟠

703. 数据流中的第 K 大元素

LeetCode	力扣	难度
703. Kth Largest Element in a Stream	703. 数据流中的第 K 大元素	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数据结构**

设计一个找到数据流中第 `k` 大元素的类（class）。注意是排序后的第 `k` 大元素，不是第 `k` 个不同的元素。

请实现 `KthLargest` 类：

1、`KthLargest(int k, int[] nums)` 使用整数 `k` 和整数流 `nums` 初始化对象。

2、`int add(int val)` 将 `val` 插入数据流 `nums` 后，返回当前数据流中第 `k` 大的元素。

示例：

```
输入：
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
输出：
[null, 4, 5, 5, 8, 8]

解释：
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3);   // return 4
kthLargest.add(5);   // return 5
kthLargest.add(10);  // return 5
kthLargest.add(9);   // return 8
kthLargest.add(4);   // return 8
```

基本思路

这题考察优先级队列的使用，可以先做下这道类似的题目 215. 数组中的第 K 个最大元素。

优先级队列的实现原理详见 图文详解二叉堆，实现优先级队列。

解法代码

```
class KthLargest {

    private int k;
    // 默认是小顶堆
    private PriorityQueue<Integer> pq = new PriorityQueue<>();
```

```
    public KthLargest(int k, int[] nums) {
        // 将 nums 装入小顶堆，保留下前 k 大的元素
        for (int e : nums) {
            pq.offer(e);
            if (pq.size() > k) {
                pq.poll();
            }
        }
        this.k = k;
    }

    public int add(int val) {
        // 维护小顶堆只保留前 k 大的元素
        pq.offer(val);
        if (pq.size() > k) {
            pq.poll();
        }
        // 堆顶就是第 k 大元素（即倒数第 k 小的元素）
        return pq.peek();
    }
}
```

- **类似题目**：
  - 剑指 Offer II 059. 数据流的第 K 大数值 🟢

剑指 Offer 40. 最小的k个数

LeetCode	力扣	难度
剑指Offer40. 最小的k个数 LCOF	剑指Offer40. 最小的k个数	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：快速排序**

输入整数数组 `arr`，找出其中最小的 `k` 个数。例如，输入 4、5、1、6、2、7、3、8 这 8 个数字，则最小的 4 个数字是 1、2、3、4。

示例 1：

```
输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]
```

基本思路

这道题和 215. 数组中的第 K 个最大元素 很像，建议你先做 215 题，我专门给那道题写了一篇详细的题解。

这道题有很多解题思路，第一个最简单的思路就是排序，然后把前 `k` 个元素拿出来就行了，时间复杂度是 `O(NlogN)`，`N` 表示 `arr` 的元素个数。

第二个思路是利用 二叉堆，时间复杂度是 `O(NlogK)`，第三个思路是利用 快速选择算法，时间复杂度是 `O(N)`，效率最高。

这里我写一下后两种思路吧，其实就是把 215 题的解法稍微改一改就行了，具体思路解析见 215 题和 快速选择算法 的图文解析。

解法代码

```
// 二叉堆的解法思路
class Solution1 {
    public int[] getLeastNumbers(int[] arr, int k) {
        // 大顶堆，堆顶是最大元素
        PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> {
            return b - a;
        });
        for (int e : arr) {
            // 每个元素都要过一遍二叉堆
            pq.offer(e);
            // 堆中元素多于 k 个时，删除堆顶元素
            if (pq.size() > k) {
                pq.poll();
            }
```

```
        }
        // pq 中剩下的是 arr 中最小的 k 个元素
        int[] res = new int[k];
        int i = 0;
        while (!pq.isEmpty()) {
            res[i] = pq.poll();
            i++;
        }
        return res;
    }
}

// 快速选择的解法思路
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        int[] res = new int[k];
        // 注意此题的 k 是元素个数而不是索引，所以和索引 p 做比较时要 - 1
        // 首先随机打乱数组
        shuffle(arr);
        int lo = 0, hi = arr.length - 1;
        // 现在开始寻找第 k 大的元素
        while (lo <= hi) {
            // 在 arr[lo..hi] 中选一个分界点
            int p = partition(arr, lo, hi);
            if (p < k - 1) {
                // 第 k 大的元素在 arr[p+1..hi] 中
                lo = p + 1;
            } else if (p > k - 1) {
                // 第 k 大的元素在 arr[lo..p-1] 中
                hi = p - 1;
            } else {
                // arr[p] 就是第 k 大元素，又因为快速排序的性质，
                // arr[p] 左边的元素都比 arr[p] 小，所以现在 arr[0..k] 就是我们要
找的答案
                for (int i = 0; i < k; i++) {
                    res[i] = arr[i];
                }
                return res;
            }
        }
        return res;
    }

    // 对 nums[lo..hi] 进行切分
    private static int partition(int[] nums, int lo, int hi) {
        int pivot = nums[lo];
        // 关于区间的边界控制需格外小心，稍有不慎就会出错
        // 我这里把 i, j 定义为开区间，同时定义：
        // [lo, i) <= pivot；(j, hi] > pivot
        // 之后都要正确维护这个边界区间的定义
        int i = lo + 1, j = hi;
        // 当 i > j 时结束循环，以保证区间 [lo, hi] 都被覆盖
        while (i <= j) {
            while (i < hi && nums[i] <= pivot) {
```

```
                i++;
                // 此 while 结束时恰好 nums[i] > pivot
            }
            while (j > lo && nums[j] > pivot) {
                j--;
                // 此 while 结束时恰好 nums[j] <= pivot
            }

            if (i >= j) {
                break;
            }
            // 此时 [lo, i) <= pivot && (j, hi] > pivot
            // 交换 nums[j] 和 nums[i]
            swap(nums, i, j);
            // 此时 [lo, i] <= pivot && [j, hi] > pivot
        }
        // 最后将 pivot 放到合适的位置，即 pivot 左边元素较小，右边元素较大
        swap(nums, lo, j);
        return j;
    }

    // 洗牌算法，将输入的数组随机打乱
    private static void shuffle(int[] nums) {
        Random rand = new Random();
        int n = nums.length;
        for (int i = 0 ; i < n; i++) {
            // 生成 [i, n - 1] 的随机数
            int r = i + rand.nextInt(n - i);
            swap(nums, i, r);
        }
    }

    // 原地交换数组中的两个元素
    private static void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

剑指 Offer II 059. 数据流的第 K 大数值

这道题和 703. 数据流中的第 K 大元素 相同。

剑指 Offer II 060. 出现频率最高的 k 个数字

这道题和 347. 前 K 个高频元素 相同。

剑指 Offer II 076. 数组中的第 k 大的数字

这道题和 215. 数组中的第 K 个最大元素 相同。

295. 数据流的中位数

LeetCode	力扣	难度
295. Find Median from Data Stream	295. 数据流的中位数	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数学**

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

设计一个支持以下两种操作的数据结构：

1、`void addNum(int num)` 从数据流中添加一个整数到数据结构中。

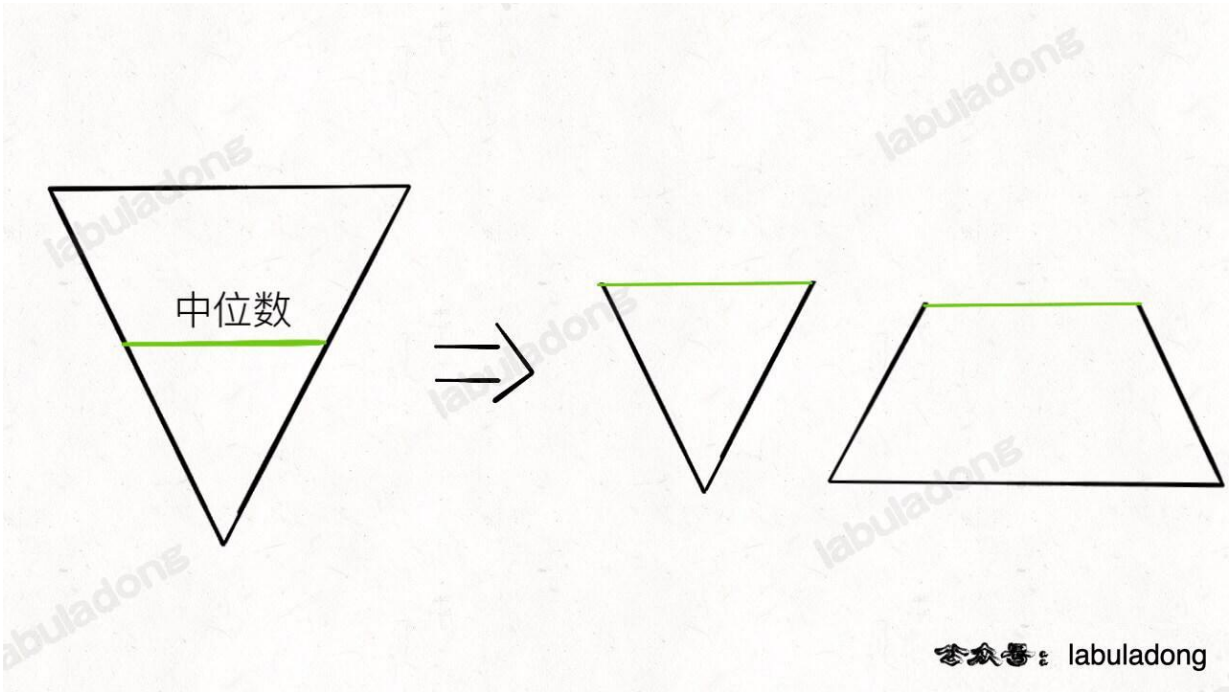
2、`double findMedian()` 返回目前所有元素的中位数。

示例：

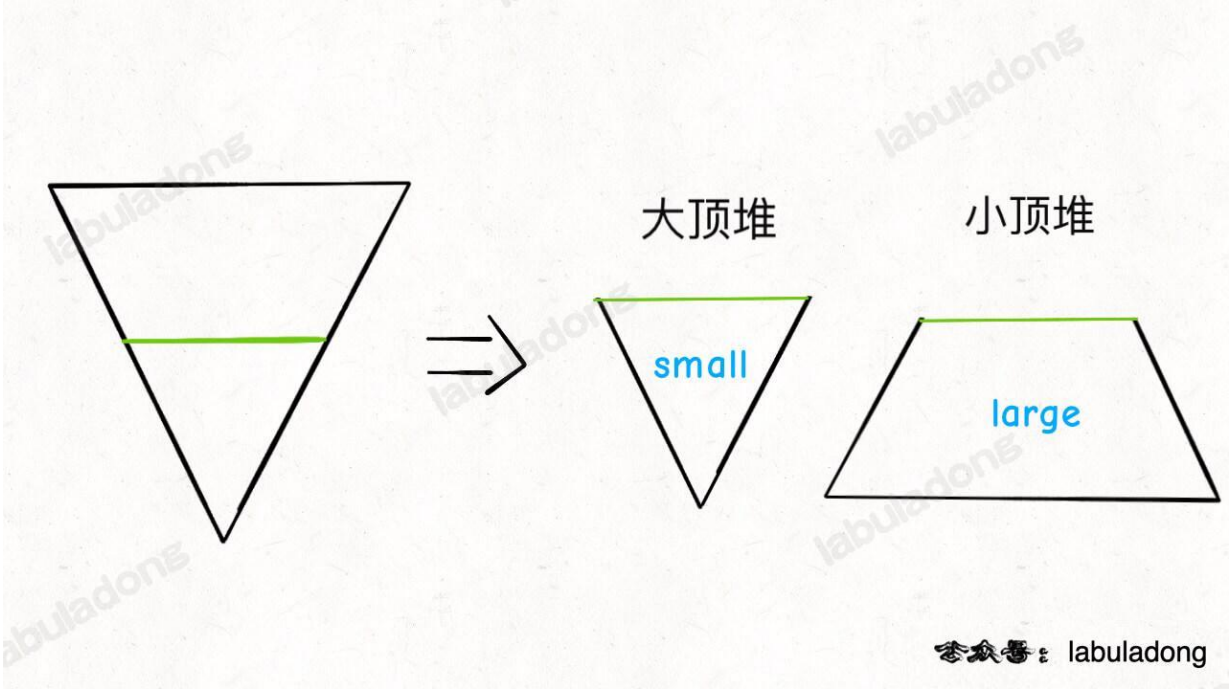
```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

基本思路

本题的核心思路是使用两个优先级队列。



小的倒三角就是个大顶堆，梯形就是个小顶堆，中位数可以通过它们的堆顶元素算出来：



- **详细题解：一道求中位数的算法题把我整不会了**

解法代码

```
class MedianFinder {
    private PriorityQueue<Integer> large;
    private PriorityQueue<Integer> small;

    public MedianFinder() {
        // 小顶堆
        large = new PriorityQueue<>();
        // 大顶堆
        small = new PriorityQueue<>((a, b) -> {
            return b - a;
        });
    }

    public double findMedian() {
        // 如果元素不一样多，多的那个堆的堆顶元素就是中位数
        if (large.size() < small.size()) {
            return small.peek();
        } else if (large.size() > small.size()) {
            return large.peek();
        }
        // 如果元素一样多，两个堆堆顶元素的平均数是中位数
        return (large.peek() + small.peek()) / 2.0;
    }

    public void addNum(int num) {
        if (small.size() >= large.size()) {
            small.offer(num);
```

```
                large.offer(small.poll());
            } else {
                large.offer(num);
                small.offer(large.poll());
            }
        }
    }
```

- **类似题目**：
    - 剑指 Offer 41. 数据流中的中位数 🔴

剑指 Offer 41. 数据流中的中位数

这道题和 295. 数据流的中位数 相同。

692. 前K个高频单词

LeetCode	力扣	难度
692. Top K Frequent Words	692. 前K个高频单词	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，哈希表**

给定一个单词列表 `words` 和一个整数 `k`，返回前 `k` 个出现次数最多的单词。返回的答案应该按单词出现频率由高到低排序，如果不同的单词有相同出现频率，**按字典顺序** 排序。

示例 1：

```
输入: words = ["i", "love", "leetcode", "i", "love", "coding"], k = 2
输出: ["i", "love"]
解析: "i" 和 "love" 为出现次数最多的两个单词，均为 2 次。
    注意，按字母顺序 "i" 在 "love" 之前。
```

基本思路

这道题可以认为是 347. 前 K 个高频元素 的进阶版。整体思路还哈希表计数，然后用 优先级队列 维护出现频率最高的 `k` 个单词。

只是我们需要注意题目要求，在 `PriorityQueue` 的比较器中正确处理频率相同的单词的字典序。

解法代码

```
class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        // 字符串 -> 该字符串出现的频率
        HashMap<String, Integer> wordToFreq = new HashMap<>();
        for (String word : words) {
            wordToFreq.put(word, wordToFreq.getOrDefault(word, 0) + 1);
        }

        PriorityQueue<Map.Entry<String, Integer>> pq = new PriorityQueue<>
(
                (entry1, entry2) -> {
                    if (entry1.getValue().equals(entry2.getValue())) {
                        // 如果出现频率相同，按照字符串字典序排序
                        return entry2.getKey().compareTo(entry1.getKey());
                    }
                    // 队列按照字符串出现频率从小到大排序
                    return entry1.getValue().compareTo(entry2.getValue());
                });
```

```
        // 按照字符串频率升序排序
        for (Map.Entry<String, Integer> entry : wordToFreq.entrySet()) {
            pq.offer(entry);
            if (pq.size() > k) {
                // 维护出现频率最多的 k 个单词
                pq.poll();
            }
        }

        // 把出现次数最多的 k 个字符串返回
        LinkedList<String> res = new LinkedList<>();
        while (!pq.isEmpty()) {
            res.addFirst(pq.poll().getKey());
        }
        return res;
    }
}
```

451. 根据字符出现频率排序

LeetCode	力扣	难度
451. Sort Characters By Frequency	451. 根据字符出现频率排序	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，排序**

给定一个字符串 `s`，根据字符出现的 **频率** 对其进行 **降序排序**。一个字符出现的 **频率** 是它出现在字符串中的次数，**返回已排序的字符串**。如果有多个答案，返回其中任何一个。

示例 1:

```
输入: s = "tree"
输出: "eert"
解释: 'e' 出现两次，'r' 和 't' 都只出现一次。
因此 'e' 必须出现在 'r' 和 't' 之前。此外，"eetr" 也是一个有效的答案。
```

基本思路

做这道题肯定需要计算每个字符出现的频率，然后你可以用很多种其他方法把字符按照频率排序。我这里用优先级队列来实现排序的效果，详细看代码。

解法代码

```
class Solution {
    public String frequencySort(String s) {
        char[] chars = s.toCharArray();
        // s 中的字符 -> 该字符出现的频率
        HashMap<Character, Integer> charToFreq = new HashMap<>();
        for (char ch : chars) {
            charToFreq.put(ch, charToFreq.getOrDefault(ch, 0) + 1);
        }

        PriorityQueue<Map.Entry<Character, Integer>>
                pq = new PriorityQueue<>((entry1, entry2) -> {
            // 队列按照键值对中的值（字符出现频率）从大到小排序
            return entry2.getValue().compareTo(entry1.getValue());
        });

        // 按照字符频率排序
        for (Map.Entry<Character, Integer> entry : charToFreq.entrySet())
{
            pq.offer(entry);
        }
```

```
        StringBuilder sb = new StringBuilder();
        while (!pq.isEmpty()) {
            // 把频率最高的字符排在前面
            Map.Entry<Character, Integer> entry = pq.poll();
            String part =
String.valueOf(entry.getKey()).repeat(entry.getValue());
            sb.append(part);
        }

        return sb.toString();
    }
}
```

1834. 单线程 CPU

LeetCode	力扣	难度
1834. Single-Threaded CPU	1834. 单线程 CPU	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，排序**

给你一个二维数组 `tasks`，用于表示 `n` 项从 `0` 到 `n - 1` 编号的任务。其中 `tasks[i] = [enqueueTime_i, processingTime_i]` 意味着第 `i` 项任务将会于 `enqueueTime_i` 时进入任务队列，需要 `processingTime_i` 的时长完成执行。

现有一个单线程 CPU，同一时间只能执行 **最多一项** 任务，该 CPU 将会按照下述方式运行：

- 如果 CPU 空闲，且任务队列中没有需要执行的任务，则 CPU 保持空闲状态。
- 如果 CPU 空闲，但任务队列中有需要执行的任务，则 CPU 将会选择**执行时间最短**的任务开始执行。如果多个任务具有同样的最短执行时间，则选择下标最小的任务开始执行。
- 一旦某项任务开始执行，CPU 在 **执行完整个任务** 前都不会停止。
- CPU 可以在完成一项任务后，立即开始执行一项新任务。

返回 CPU 处理任务的顺序。

示例 1：

```
输入：tasks = [[1,2],[2,4],[3,2],[4,1]]
输出：[0,2,3,1]
解释：事件按下述流程运行：
- time = 1，任务 0 进入任务队列，可执行任务项 = {0}
- 同样在 time = 1，空闲状态的 CPU 开始执行任务 0，可执行任务项 = {}
- time = 2，任务 1 进入任务队列，可执行任务项 = {1}
- time = 3，任务 2 进入任务队列，可执行任务项 = {1, 2}
- 同样在 time = 3，CPU 完成任务 0 并开始执行队列中用时最短的任务 2，可执行任务项 =
{1}
- time = 4，任务 3 进入任务队列，可执行任务项 = {1, 3}
- time = 5，CPU 完成任务 2 并开始执行队列中用时最短的任务 3，可执行任务项 = {1}
- time = 6，CPU 完成任务 3 并开始执行任务 1，可执行任务项 = {}
- time = 10，CPU 完成任务 1 并进入空闲状态
```

基本思路

这题的难度不算大，就是有些复杂，**难点在于你要同时控制三个变量（开始时间、处理时间、索引）的有序性，而且这三个变量还有优先级**：

首先应该考虑开始时间，因为只要到了开始时间，任务才进入可执行状态；

其次应该考虑任务的处理时间，在所有可以执行的任务中优先选择处理时间最短的；

如果存在处理时间相同的任务，那么优先选择索引最小的。

所以这道题的思路是：

先根据任务「开始时间」排序，维护一个时间线变量 now 来判断哪些任务到了可执行状态，然后借助一个优先级队列 pq 对「处理时间」和「索引」进行动态排序。

利用优先级队列动态排序是有必要的，因为每完成一个任务，时间线 now 就要更新，进而产生新的可执行任务。

解法代码

```
class Solution {
    public int[] getOrder(int[][] tasks) {
        int n = tasks.length;
        // 把原始索引也添加上，方便后面排序用
        ArrayList<int[]> triples = new ArrayList<>();
        for (int i = 0; i < tasks.length; i++) {
            triples.add(new int[]{tasks[i][0], tasks[i][1], i});
        }
        // 数组先按照任务的开始时间排序
        triples.sort((a, b) -> {
            return a[0] - b[0];
        });

        // 按照任务的处理时间排序，如果处理时间相同，按照原始索引排序
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            if (a[1] != b[1]) {
                // 比较处理时间
                return a[1] - b[1];
            }
            // 比较原始索引
            return a[2] - b[2];
        });

        ArrayList<Integer> res = new ArrayList<>();
        // 记录完成任务的时间线
        int now = 0;
        int i = 0;
        while (res.size() < n) {
            if (!pq.isEmpty()) {
                // 完成队列中的一个任务
                int[] triple = pq.poll();
                res.add(triple[2]);
                // 每完成一个任务，就要推进时间线
                now += triple[1];
            } else if (i < n && triples.get(i)[0] > now) {
                // 队列为空可能因为还没到开始时间，
                // 直接把时间线推进到最近任务的开始时间
                now = triples.get(i)[0];
            }
```

```
                // 由于时间线的推进，会产生可以开始执行的任务
                for (; i < n && triples.get(i)[0] <= now; i++) {
                    pq.offer(triples.get(i));
                }
            }

            // Java 语言特性，将 List 转化成 int[] 格式
            int[] arr = new int[n];
            for (int j = 0; j < n; j++) {
                arr[j] = res.get(j);
            }
            return arr;
        }
    }
```

1845. 座位预约管理系统

LeetCode	力扣	难度
1845. Seat Reservation Manager	1845. 座位预约管理系统	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数据结构，设计**

请你设计一个管理 `n` 个座位预约的系统，座位编号从 `1` 到 `n`。

请你实现 `SeatManager` 类：

- `SeatManager(int n)` 初始化一个 `SeatManager` 对象，它管理从 `1` 到 `n` 编号的 `n` 个座位。所有座位初始都是可预约的。
- `int reserve()` 返回可以预约座位的**最小编号**，此座位变为不可预约。
- `void unreserve(int seatNumber)` 将给定编号 `seatNumber` 对应的座位变成可以预约。

示例 1：

```
输入：
["SeatManager", "reserve", "reserve", "unreserve", "reserve", "reserve",
"reserve", "reserve", "unreserve"]
[[5], [], [], [2], [], [], [], [], [5]]
输出：
[null, 1, 2, null, 2, 3, 4, 5, null]

解释：
SeatManager seatManager = new SeatManager(5); // 初始化 SeatManager，有 5 个
座位。
seatManager.reserve();    // 所有座位都可以预约，所以返回最小编号的座位，也就是 1。
seatManager.reserve();    // 可以预约的座位为 [2,3,4,5]，返回最小编号的座位，也就
是 2。
seatManager.unreserve(2); // 将座位 2 变为可以预约，现在可预约的座位为
[2,3,4,5]。
seatManager.reserve();    // 可以预约的座位为 [2,3,4,5]，返回最小编号的座位，也就
是 2。
seatManager.reserve();    // 可以预约的座位为 [3,4,5]，返回最小编号的座位，也就是
3。
seatManager.reserve();    // 可以预约的座位为 [4,5]，返回最小编号的座位，也就是
4。
seatManager.reserve();    // 唯一可以预约的是座位 5，所以返回 5。
seatManager.unreserve(5); // 将座位 5 变为可以预约，现在可预约的座位为 [5]。
```

基本思路

这题是 379. 电话目录管理系统 的进阶版，那一道题返回的空闲号码可以随意，而这道题要求返回**最小**的座位编号。

其实很思路是一样的，只是这里需要用到能够按照元素大小自动排序的数据结构 优先级队列（二叉堆），直接看代码吧。

解法代码

```
class SeatManager {
    // 利用优先级队列自动排序，队头的元素就是最小的
    PriorityQueue<Integer> pq = new PriorityQueue<>();

    public SeatManager(int n) {
        // 初始化所有空闲座位
        for (int i = 1; i <= n; i++) {
            pq.offer(i);
        }
    }

    public int reserve() {
        // 拿出队头元素（最小）
        return pq.poll();
    }

    public void unreserve(int i) {
        pq.offer(i);
    }
}
```

146. LRU 缓存机制

LeetCode	力扣	难度
146. LRU Cache	146. LRU 缓存	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数据结构，设计**

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：

1、 `LRUCache(int capacity)` 以正整数作为容量 `capacity` 初始化 LRU 缓存。

2、 `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`。

3、 `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，则插入该键值对。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。

示例：

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get",
"get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
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=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=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

基本思路

PS：这道题在《算法小抄》 的第 216 页。

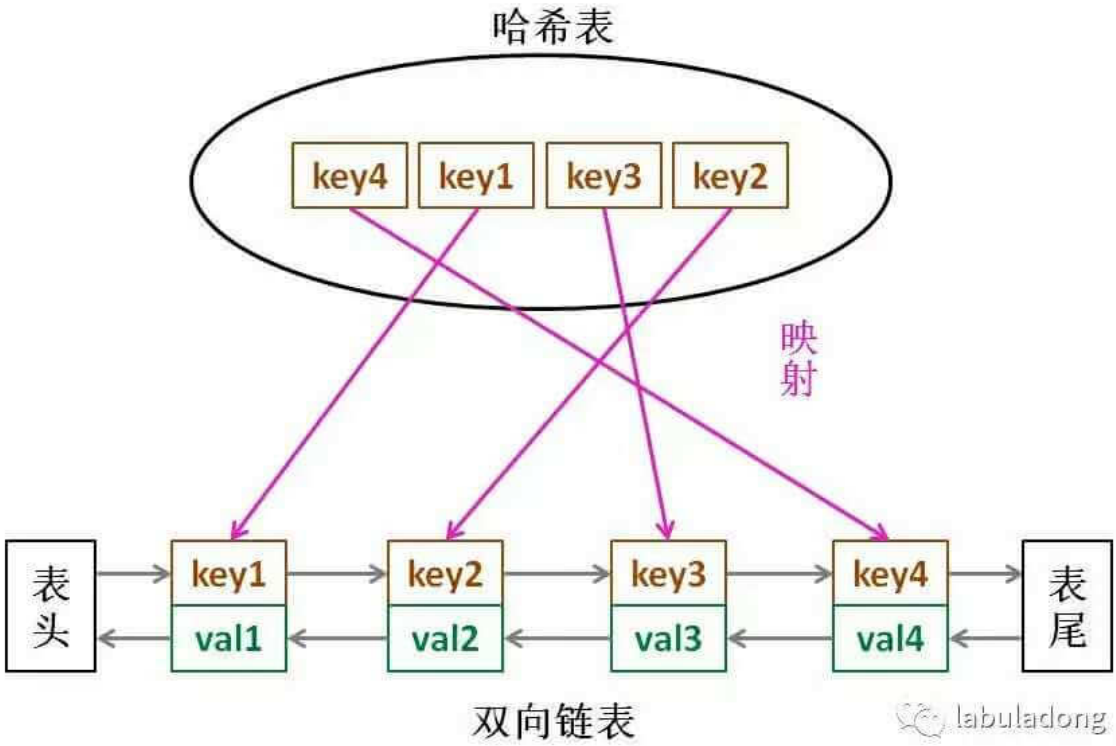
要让 `put` 和 `get` 方法的时间复杂度为 `O(1)`，我们可以总结出 `cache` 这个数据结构必要的条件：

1、显然 cache 中的元素必须有时序，以区分最近使用的和久未使用的数据，当容量满了之后要删除最久未使用的那个元素腾位置。

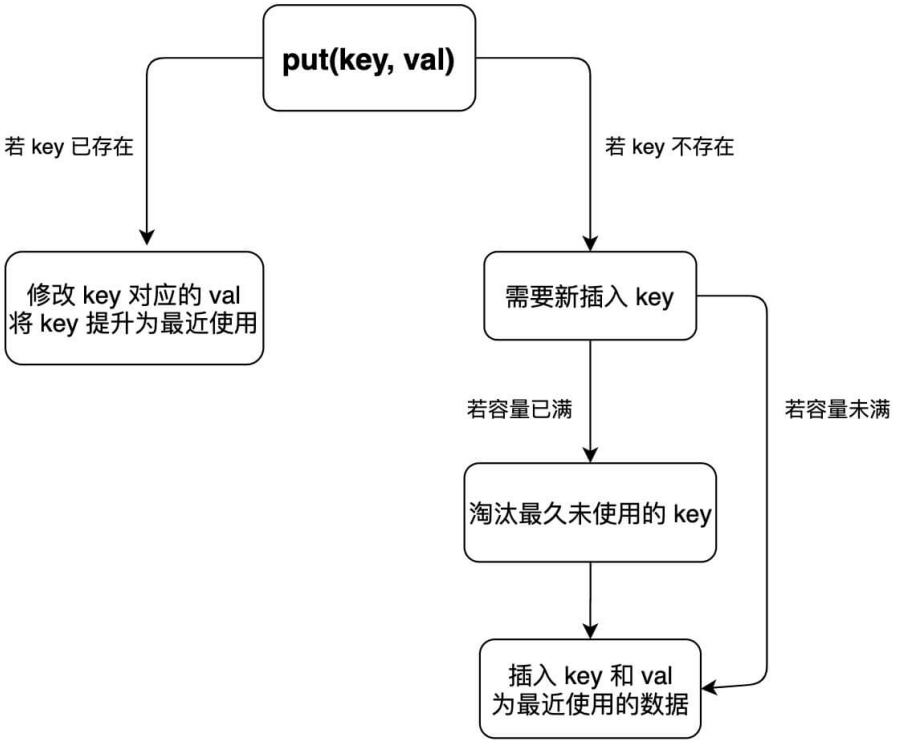
2、我们要在 cache 中快速找某个 key 是否已存在并得到对应的 val；

3、每次访问 cache 中的某个 key，需要将这个元素变为最近使用的，也就是说 cache 要支持在任意位置快速插入和删除元素。

哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢，所以结合二者的长处，可以形成一种新的数据结构：哈希链表 LinkedHashMap：



至于 put 和 get 的具体逻辑，可以画出这样一个流程图：



根据上述逻辑写代码即可。

- **详细题解：算法就像搭乐高：带你手撸 LRU 算法**

解法代码

```
class LRUCache {
    int cap;
    LinkedHashMap<Integer, Integer> cache = new LinkedHashMap<>();
    public LRUCache(int capacity) {
        this.cap = capacity;
    }

    public int get(int key) {
        if (!cache.containsKey(key)) {
            return -1;
        }
        // 将 key 变为最近使用
        makeRecently(key);
        return cache.get(key);
    }

    public void put(int key, int val) {
        if (cache.containsKey(key)) {
            // 修改 key 的值
            cache.put(key, val);
            // 将 key 变为最近使用
            makeRecently(key);
            return;
        }
```

```
        if (cache.size() >= this.cap) {
            // 链表头部就是最久未使用的 key
            int oldestKey = cache.keySet().iterator().next();
            cache.remove(oldestKey);
        }
        // 将新的 key 添加链表尾部
        cache.put(key, val);
    }

    private void makeRecently(int key) {
        int val = cache.get(key);
        // 删除 key，重新插入到队尾
        cache.remove(key);
        cache.put(key, val);
    }
}
```

- **类似题目**：
  - 剑指 Offer II 031. 最近最少使用缓存 🟠

剑指 Offer II 031. 最近最少使用缓存

这道题和 146. LRU 缓存机制 相同。

155. 最小栈

LeetCode	力扣	难度
155. Min Stack	155. 最小栈	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，栈，设计

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 `MinStack` 类：

- `MinStack()` 初始化堆栈对象。
- `void push(int val)` 将元素 val 推入堆栈。
- `void pop()` 删除堆栈顶部的元素。
- `int top()` 获取堆栈顶部的元素。
- `int getMin()` 获取堆栈中的最小元素。

示例 1:

```
输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

基本思路

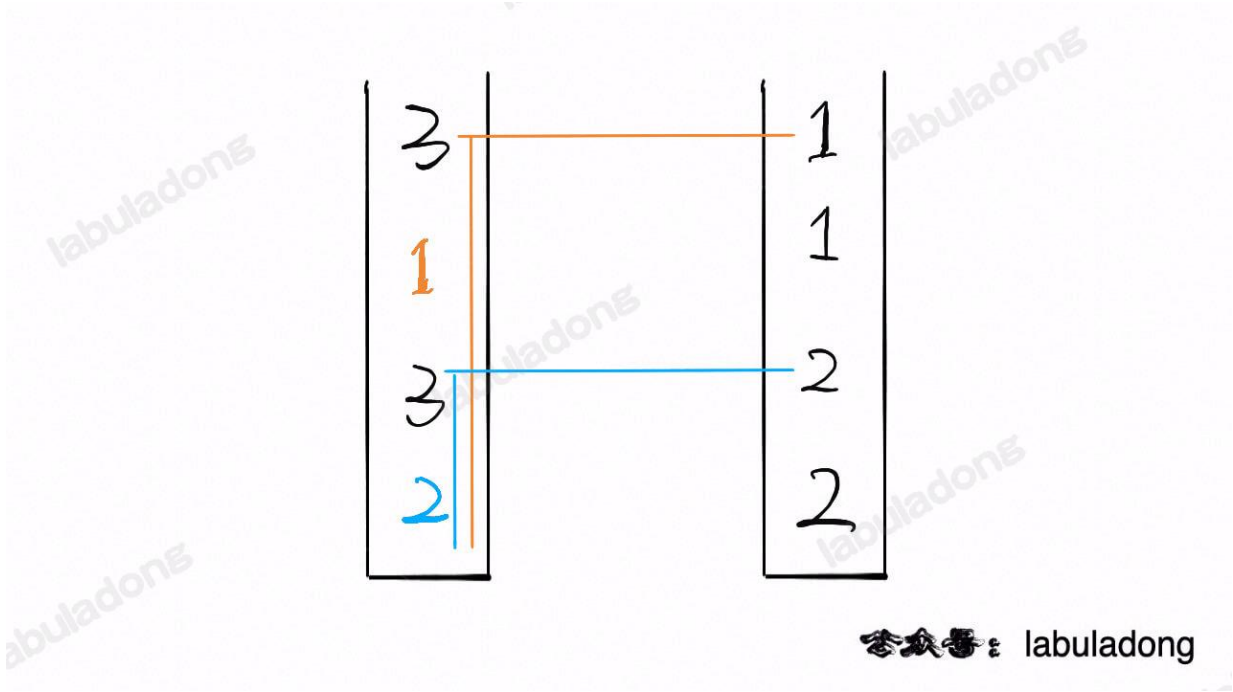
根据我们之前亲自动手实现的栈，我们知道栈是一种操作受限的数据结构，只能从栈顶插入或弹出元素，所以对于标准的栈来说，如果想实现本题的 `getMin` 方法，只能老老实实把所有元素弹出来然后找最小值。**想提高时间效率，那肯定要通过空间换时间的思路**。

不过在具体说解法之前，我想聊一下动态集合中维护最值的问题。这类问题看似简单，但实际上是个很棘手的问题。其实本题就是如下一个场景：

假设你有若干数字，你用一个 min 变量维护了其中的最小值，如果现在给这些数字中添加一个新数字，那么只要比较这个新数字和 min 的大小就可以得出最新的最小值。但如果现在从这些数字钟删除一个数字，你还能用 min 变量得到最小值吗？答案是不能，因为如果这个被删除的数字恰好是最小值，那么新的 min 变量应该更新为第二小的元素对吧，但是我没有记录第二小的元素是多少，所以只能把所有数字重新遍历一遍。

明确了难点再回到本题，就可以对症下药了。删除栈顶元素的时候，不确定新的最小值是多少，但楼下那哥们知道啊，他当时入栈时的最小值，就是现在的最小值呗。

所以这道题的关键就是，**每个元素入栈时，还要记下来当前栈中的最小值**。比方说，可以用一个额外的栈 minStk 来记录栈中每个元素入栈时的栈中的最小元素是多少，这样每次删除元素时就能快速得到剩余栈中的最小元素了。



当然，我们还可以做一些优化，减少 minStk 中存储的元素个数，我把原始解法和优化解法都写出来了，供参考。

> PS：这道题并不难，但我还是很细致地分析了，希望你深刻理解其中的难点。下一步可以做一下 239. 滑动窗口最大值，请仔细观察和思考，队列结构是如何解决这个难点的。

解法代码

```
// 原始思路
class MinStack1 {
    // 记录栈中的所有元素
    Stack<Integer> stk = new Stack<>();
    // 阶段性记录栈中的最小元素
    Stack<Integer> minStk = new Stack<>();

    public void push(int val) {
        stk.push(val);
        // 维护 minStk 栈顶为全栈最小元素
        if (minStk.isEmpty() || val <= minStk.peek()) {
            // 新插入的这个元素就是全栈最小的
```

```
            minStk.push(val);
        } else {
            // 插入的这个元素比较大
            minStk.push(minStk.peek());
        }
    }

    public void pop() {
        stk.pop();
        minStk.pop();
    }

    public int top() {
        return stk.peek();
    }

    public int getMin() {
        // minStk 栈顶为全栈最小元素
        return minStk.peek();
    }
}
// 优化版
class MinStack {
    // 记录栈中的所有元素
    Stack<Integer> stk = new Stack<>();
    // 阶段性记录栈中的最小元素
    Stack<Integer> minStk = new Stack<>();

    public void push(int val) {
        stk.push(val);
        // 维护 minStk 栈顶为全栈最小元素
        if (minStk.isEmpty() || val <= minStk.peek()) {
            // 新插入的这个元素就是全栈最小的
            minStk.push(val);
        }
    }

    public void pop() {
        // 注意 Java 的语言特性，比较 Integer 相等要用 equals 方法
        if (stk.peek().equals(minStk.peek())) {
            // 弹出的元素是全栈最小的
            minStk.pop();
        }
        stk.pop();
    }

    public int top() {
        return stk.peek();
    }

    public int getMin() {
        // minStk 栈顶为全栈最小元素
        return minStk.peek();
```

```
        }
    }
```

- **类似题目**：
    - 239. 滑动窗口最大值 🔴
    - 剑指 Offer 30. 包含min函数的栈 🟢
    - 剑指 Offer 59 - I. 滑动窗口的最大值 🔴

239. 滑动窗口最大值

LeetCode	力扣	难度
239. Sliding Window Maximum	239. 滑动窗口最大值	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，滑动窗口，队列

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧，返回滑动窗口中的最大值。

滑动窗口每次只向右移动一位，你只可以看到在滑动窗口内的 k 个数字。

示例 1：

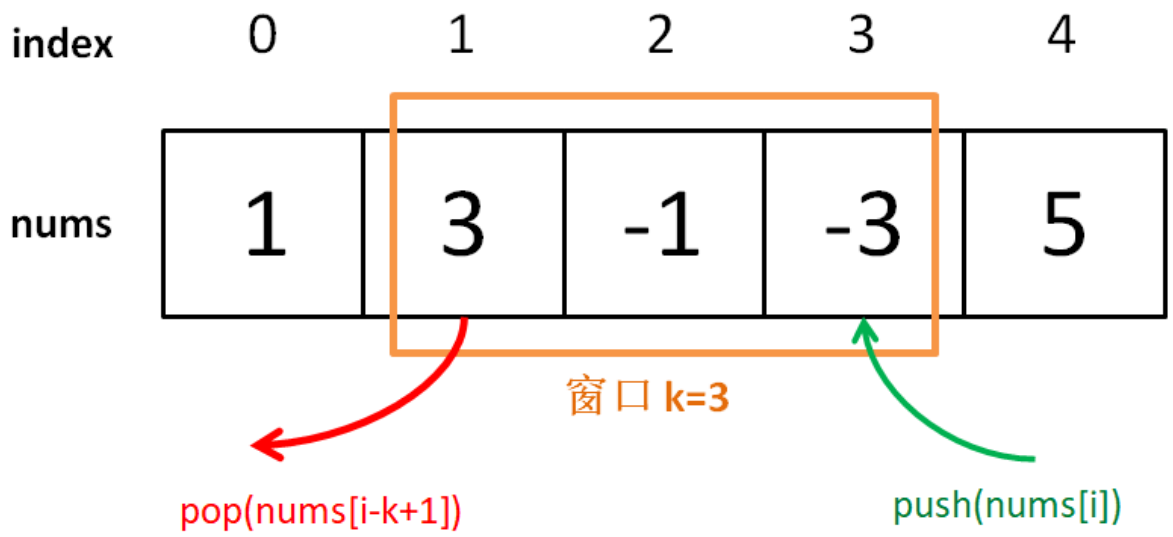
```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

基本思路

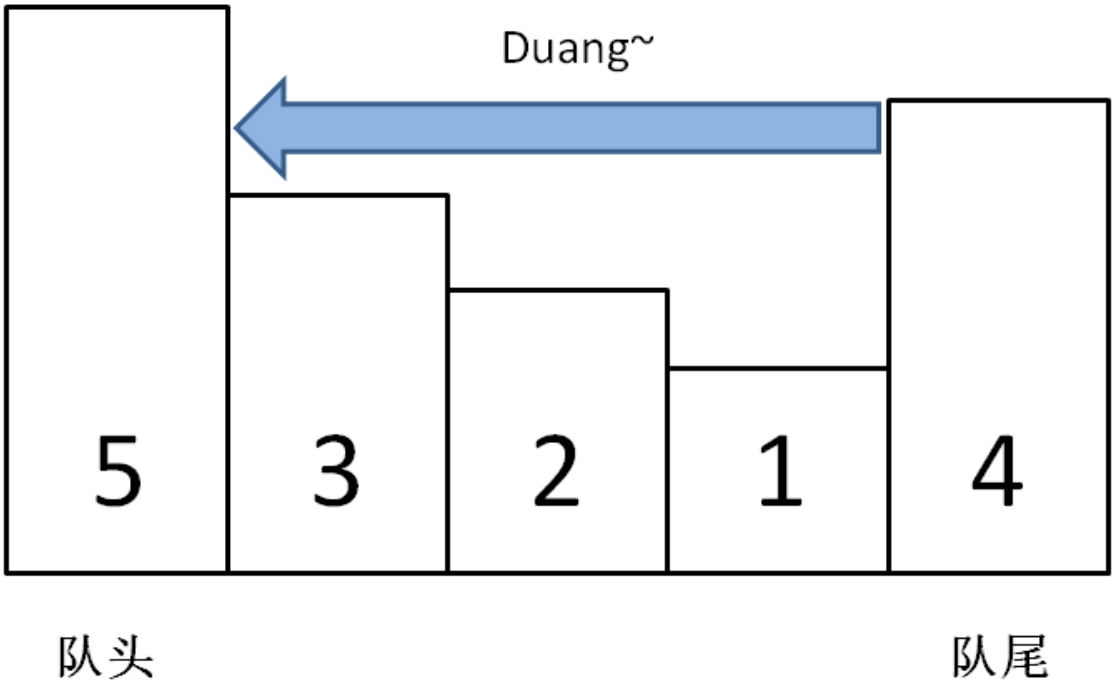
PS：这道题在《算法小抄》 的第 271 页。

> 我在 155. 最小栈 的思路中详细分析了动态集合中维护最值的「千古难题」，如果你没有做，可以先去做一下。我想请你结合这两道题仔细思考：队列和栈分别是如何解决最值维护问题的？

使用一个队列充当不断滑动的窗口，每次滑动记录其中的最大值：



如何在 O(1) 时间计算最大值，只需要一个特殊的数据结构「单调队列」，push 方法依然在队尾添加元素，但是要把前面比自己小的元素都删掉，直到遇到更大的元素才停止删除。



使用单调队列数据结构就能完成本题。

- **详细题解：单调队列结构解决滑动窗口问题**

解法代码

```
class Solution {
    /* 单调队列的实现 */
    class MonotonicQueue {
```

```
        LinkedList<Integer> q = new LinkedList<>();
        public void push(int n) {
            // 将小于 n 的元素全部删除
            while (!q.isEmpty() && q.getLast() < n) {
                q.pollLast();
            }
            // 然后将 n 加入尾部
            q.addLast(n);
        }

        public int max() {
            return q.getFirst();
        }

        public void pop(int n) {
            if (n == q.getFirst()) {
                q.pollFirst();
            }
        }
    }

    /* 解题函数的实现 */
    public int[] maxSlidingWindow(int[] nums, int k) {
        MonotonicQueue window = new MonotonicQueue();
        List<Integer> res = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            if (i < k - 1) {
                //先填满窗口的前 k - 1
                window.push(nums[i]);
            } else {
                // 窗口向前滑动，加入新数字
                window.push(nums[i]);
                // 记录当前窗口的最大值
                res.add(window.max());
                // 移出旧数字
                window.pop(nums[i - k + 1]);
            }
        }
        // 需要转成 int[] 数组再返回
        int[] arr = new int[res.size()];
        for (int i = 0; i < res.size(); i++) {
            arr[i] = res.get(i);
        }
        return arr;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：

- 155. 最小栈 🟠
- 剑指 Offer 30. 包含min函数的栈 🟢
- 剑指 Offer 59 - I. 滑动窗口的最大值 🔴
- 剑指 Offer 59 - II. 队列的最大值 🟠

剑指 Offer 30. 包含min函数的栈

这道题和 155. 最小栈 相同。

剑指 Offer 59 - I. 滑动窗口的最大值

这道题和 239. 滑动窗口最大值 相同。

284. 顶端迭代器

LeetCode	力扣	难度
284. Peeking Iterator	284. 窥探迭代器	🟠

- **标签：设计**

请你在设计一个迭代器，在集成现有迭代器拥有的 `hasNext` 和 `next` 操作的基础上，还额外支持 `peek` 操作。

实现 `PeekingIterator` 类：

- `PeekingIterator(Iterator<int> nums)` 使用指定整数迭代器 `nums` 初始化迭代器。
- `int next()` 返回数组中的下一个元素，并将指针移动到下个元素处。
- `bool hasNext()` 如果数组中存在下一个元素，返回 `true`；否则，返回 `false`。
- `int peek()` 返回数组中的下一个元素，但 **不** 移动指针。

注意：每种语言可能有不同的构造函数和迭代器 `Iterator`，但均支持 `int next()` 和 `boolean hasNext()` 函数。

示例 1：

```
输入：
["PeekingIterator", "next", "peek", "next", "next", "hasNext"]
[[[1, 2, 3]], [], [], [], [], []]
输出：
[null, 1, 2, 2, 3, false]

解释：
PeekingIterator peekingIterator = new PeekingIterator([1, 2, 3]); //
[1,2,3]
peekingIterator.next();    // 返回 1，指针移动到下一个元素 [1,2,3]
peekingIterator.peek();    // 返回 2，指针未发生移动 [1,2,3]
peekingIterator.next();    // 返回 2，指针移动到下一个元素 [1,2,3]
peekingIterator.next();    // 返回 3，指针移动到下一个元素 [1,2,3]
peekingIterator.hasNext(); // 返回 False
```

基本思路

这个题需要你了解「迭代器」的接口特性，用 Java 来做比较好，建议你先做下 251. 展开二维向量 了解下迭代器是什么。

你看构造函数的入参是 `Iterator<Integer>` 类型，就是说输入一个装 `Integer` 的迭代器，且你只能调用 `hasNext` 方法判断迭代器内是否还有元素和 `next` 方法从迭代器中拿出下一个元素。

这道题的关键是让你额外实现一个 peek 方法，这个 peek 返回迭代器中的下一个元素，但不能删除这个元素。注意 next 方法是返回**并删除**迭代器中的下一个元素。

所以我们可以把迭代器的第一个元素缓存起来，以实现 peek 方法，直接看代码吧。

解法代码

```
// Java Iterator interface reference:
// https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

class PeekingIterator implements Iterator<Integer> {
    private Iterator<Integer> iter;
    // 把迭代器的下一个元素提前拿出来并缓存起来
    private Integer nextElem;

    public PeekingIterator(Iterator<Integer> iterator) {
        this.iter = iterator;
        this.nextElem = iterator.next();
    }

    public Integer peek() {
        return nextElem;
    }


    @Override
    public Integer next() {
        Integer res = nextElem;
        // 更新 nextElem
        if (iter.hasNext()) {
            nextElem = iter.next();
        } else {
            nextElem = null;
        }
        return res;
    }

    @Override
    public boolean hasNext() {
        return nextElem != null;
    }
}
```

341. 扁平化嵌套列表迭代器

LeetCode	力扣	难度
341. Flatten Nested List Iterator	341. 扁平化嵌套列表迭代器	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构，设计

给你一个嵌套的整数列表 `nestedList`。每个元素要么是一个整数，要么是一个列表；该列表的元素也可能是整数或者是其他列表。请你实现一个迭代器将其扁平化，使之能够遍历这个列表中的所有整数。

实现扁平迭代器类 `NestedIterator`：

1、`NestedIterator(List<NestedInteger> nestedList)` 用嵌套列表 `nestedList` 初始化迭代器。

2、`int next()` 返回嵌套列表的下一个整数。

3、`boolean hasNext()` 如果仍然存在待迭代的整数，返回 `true`；否则，返回 `false`。

你的代码将会用下述伪代码检测：

```
initialize iterator with nestedList
res = []
while iterator.hasNext()
    append iterator.next() to the end of res
return res
```

如果 `res` 与预期的扁平化列表匹配，那么你的代码将会被判为正确。

示例 1：

```
输入：nestedList = [[1,1],2,[1,1]]
输出：[1,1,2,1,1]
解释：通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是：
[1,1,2,1,1]。
```

基本思路

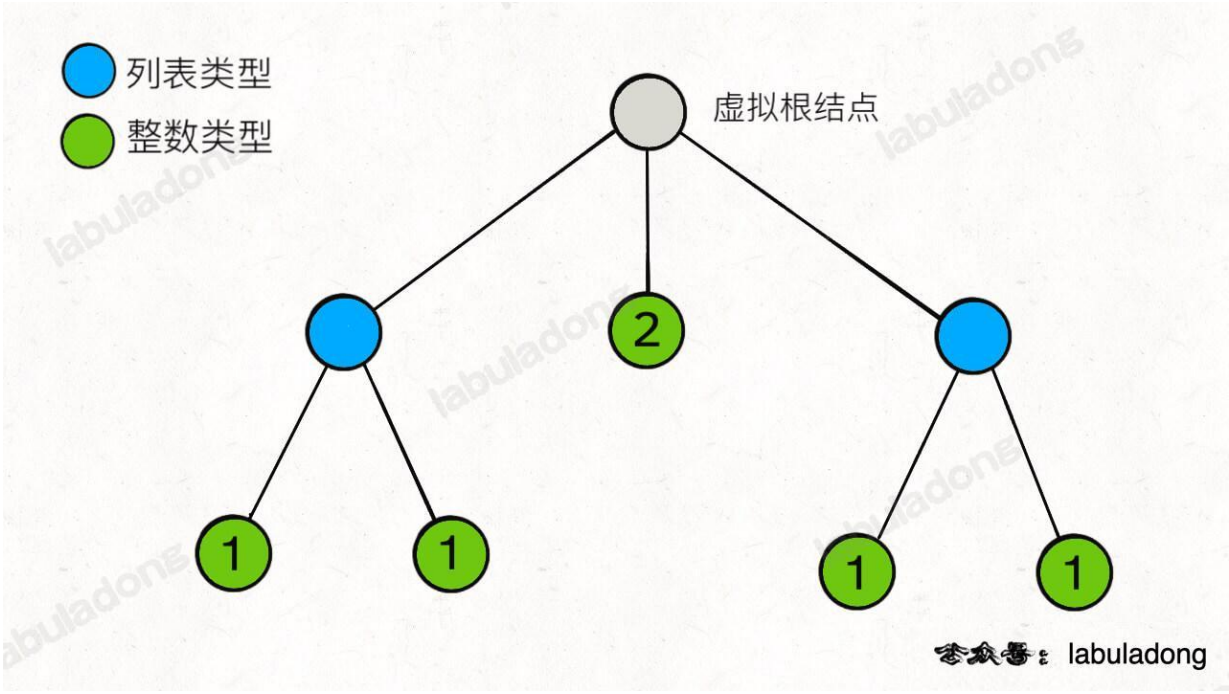
PS：这道题在《算法小抄》 的第 345 页。

题目专门说不要尝试实现或者猜测 `NestedInteger` 的实现，那我们就立即实现一下 `NestedInteger` 的结构：

```
public class NestedInteger {
    private Integer val;
    private List<NestedInteger> list;

    public NestedInteger(Integer val) {
        this.val = val;
        this.list = null;
    }
    public NestedInteger(List<NestedInteger> list) {
        this.list = list;
        this.val = null;
    }

    // 如果其中存的是一个整数，则返回 true，否则返回 false
    public boolean isInteger() {
        return val != null;
    }

    // 如果其中存的是一个整数，则返回这个整数，否则返回 null
    public Integer getInteger() {
        return this.val;
    }

    // 如果其中存的是一个列表，则返回这个列表，否则返回 null
    public List<NestedInteger> getList() {
        return this.list;
    }
}
```

根据 学习数据结构和算法的框架思维，发现这玩意儿竟然就是个多叉树的结构：

```
class NestedInteger {
    Integer val;
    List<NestedInteger> list;
}

// 基本的 N 叉树节点
class TreeNode {
    int val;
    TreeNode[] children;
}
```



所以，把一个 `NestedInteger` 扁平化就等价于遍历一棵 N 叉树的所有「叶子节点」。

用迭代器的方式来实现这个功能，就是**调用 `hasNext` 时，如果 `nestedList` 的第一个元素是列表类型，则不断展开这个元素，直到第一个元素是整数类型。**

- **详细题解：题目不让我干什么，我偏要干什么**

解法代码

```
public class NestedIterator implements Iterator<Integer> {
    private LinkedList<NestedInteger> list;

    public NestedIterator(List<NestedInteger> nestedList) {
        // 不直接用 nestedList 的引用，是因为不能确定它的底层实现
        // 必须保证是 LinkedList，否则下面的 addFirst 会很低效
        list = new LinkedList<>(nestedList);
    }

    public Integer next() {
        // hasNext 方法保证了第一个元素一定是整数类型
        return list.remove(0).getInteger();
    }

    public boolean hasNext() {
        // 循环拆分列表元素，直到列表第一个元素是整数类型
        while (!list.isEmpty() && !list.get(0).isInteger()) {
            // 当列表开头第一个元素是列表类型时，进入循环
            List<NestedInteger> first = list.remove(0).getList();
            // 将第一个列表打平并按顺序添加到开头
            for (int i = first.size() - 1; i >= 0; i--) {
                list.addFirst(first.get(i));
            }
        }
```

```
            return !list.isEmpty();
        }
    }
```

355. 设计推特

LeetCode	力扣	难度
355. Design Twitter	355. 设计推特	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，设计

设计一个简化版的推特 (Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近 10 条推文。

实现 Twitter 类：

- void postTweet(int userId, int tweetId) 根据给定的 tweetId 和 userId 创建一条新推文。每次调用此函数都会使用一个不同的 tweetId。
- List<Integer> getNewsFeed(int userId) 检索当前用户新闻推送中最近 10 条推文的 ID。新闻推送中的每一项都必须是由用户关注的人或者是用户自己发布的推文。推文必须**按照时间顺序由最近到最远排序**。
- void follow(int followerId, int followeeId) ID 为 followerId 的用户开始关注 ID 为 followeeId 的用户。
- void unfollow(int followerId, int followeeId) ID 为 followerId 的用户不再关注 ID 为 followeeId 的用户。

示例：

```
输入
["Twitter", "postTweet", "getNewsFeed", "follow", "postTweet",
"getNewsFeed", "unfollow", "getNewsFeed"]
[[], [1, 5], [1], [1, 2], [2, 6], [1], [1, 2], [1]]
输出
[null, null, [5], null, null, [6, 5], null, [5]]

解释
Twitter twitter = new Twitter();
twitter.postTweet(1, 5); // 用户 1 发送了一条新推文 (用户 id = 1, 推文 id = 5)
twitter.getNewsFeed(1);  // 用户 1 的获取推文应当返回一个列表，其中包含一个 id 为
5 的推文
twitter.follow(1, 2);    // 用户 1 关注了用户 2
twitter.postTweet(2, 6); // 用户 2 发送了一个新推文 (推文 id = 6)
twitter.getNewsFeed(1);  // 用户 1 的获取推文应当返回一个列表，其中包含两个推文，id
分别为 -> [6, 5]。推文 id 6 应当在推文 id 5 之前，因为它是在 5 之后发送的
twitter.unfollow(1, 2);  // 用户 1 取消关注了用户 2
twitter.getNewsFeed(1);  // 用户 1 获取推文应当返回一个列表，其中包含一个 id 为 5
的推文。因为用户 1 已经不再关注用户 2
```

基本思路

这道题比较经典，在特定场景下让你设计算法。其难点在于 `getNewsFeed` 方法，要按照时间线顺序展示所有关注用户的推文，这个方法要用到我在 单链表的六大解题套路 解决 23. 合并K个升序链表 的合并多个有序链表的技巧：

你把一个用户发布的所有推文做成一条有序链表（靠近头部的推文是最近发布的），那么只要合并关注用户的推文链表，即可获得按照时间线顺序排序的信息流。

具体看代码吧，我注释比较详细。

- **详细题解：设计朋友圈时间线功能**

解法代码

```
class Twitter {
    // 全局时间戳
    int globalTime = 0;
    // 记录用户 ID 到用户示例的映射
    HashMap<Integer, User> idToUser = new HashMap<>();

    // Tweet 类
    class Tweet {
        private int id;
        // 时间戳用于对信息流按照时间排序
        private int timestamp;
        // 指向下一条 tweet，类似单链表结构
        private Tweet next;

        public Tweet(int id) {
            this.id = id;
            // 新建一条 tweet 时记录并更新时间戳
            this.timestamp = globalTime++;
        }

        public int getId() {
            return id;
        }

        public int getTimestamp() {
            return timestamp;
        }

        public Tweet getNext() {
            return next;
        }

        public void setNext(Tweet next) {
            this.next = next;
        }
    }
```

```
// 用户类
class User {
    // 记录该用户的 id 以及发布的 tweet
    private int id;
    private Tweet tweetHead;
    // 记录该用户的关注者
    private HashSet<User> followedUserSet;

    public User(int id) {
        this.id = id;
        this.tweetHead = null;
        this.followedUserSet = new HashSet<>();
    }

    public int getId() {
        return id;
    }

    public Tweet getTweetHead() {
        return tweetHead;
    }

    public HashSet<User> getFollowedUserSet() {
        return followedUserSet;
    }

    public boolean equals(User other) {
        return this.id == other.id;
    }

    // 关注其他人
    public void follow(User other) {
        followedUserSet.add(other);
    }

    // 取关其他人
    public void unfollow(User other) {
        followedUserSet.remove(other);
    }

    // 发布一条 tweet
    public void post(Tweet tweet) {
        // 把新发布的 tweet 作为链表头节点
        tweet.setNext(tweetHead);
        tweetHead = tweet;
    }
}

public void postTweet(int userId, int tweetId) {
    // 如果这个用户还不存在，新建用户
    if (!idToUser.containsKey(userId)) {
        idToUser.put(userId, new User(userId));
    }
    User user = idToUser.get(userId);
```

```
        user.post(new Tweet(tweetId));
    }

    public List<Integer> getNewsFeed(int userId) {
        List<Integer> res = new LinkedList<>();
        if (!idToUser.containsKey(userId)) {
            return res;
        }
        // 获取该用户关注的用户列表
        User user = idToUser.get(userId);
        Set<User> followedUserSet = user.getFollowedUserSet();
        // 每个用户的 tweet 是一条按时间排序的链表
        // 现在执行合并多条有序链表的逻辑，找出时间线中的最近 10 条动态
        PriorityQueue<Tweet> pq = new PriorityQueue<>((a, b) -> {
            // 按照每条 tweet 的发布时间降序排序（最近发布的排在事件流前面）
            return b.timestamp - a.timestamp;
        });
        // 该用户自己的 tweet 也在时间线内
        if (user.getTweetHead() != null) {
            pq.offer(user.getTweetHead());
        }
        for (User other : followedUserSet) {
            if (other.getTweetHead() != null) {
                pq.offer(other.tweetHead);
            }
        }
        // 合并多条有序链表
        int count = 0;
        while (!pq.isEmpty() && count < 10) {
            Tweet tweet = pq.poll();
            res.add(tweet.getId());
            if (tweet.getNext() != null) {
                pq.offer(tweet.getNext());
            }
            count++;
        }
        return res;
    }

    public void follow(int followerId, int followeeId) {
        // 如果用户还不存在，则新建用户
        if (!idToUser.containsKey(followerId)) {
            idToUser.put(followerId, new User(followerId));
        }
        if (!idToUser.containsKey(followeeId)) {
            idToUser.put(followeeId, new User(followeeId));
        }

        User follower = idToUser.get(followerId);
        User followee = idToUser.get(followeeId);
        // 关注者关注被关注者
        follower.follow(followee);
    }
```

```
    public void unfollow(int followerId, int followeeId) {
        if (!idToUser.containsKey(followerId) ||
!idToUser.containsKey(followeeId)) {
            return;
        }
        User follower = idToUser.get(followerId);
        User followee = idToUser.get(followeeId);
        // 关注者取关被关注者
        follower.unfollow(followee);
    }
}
```

380. O(1) 时间插入、删除和获取随机元素

LeetCode	力扣	难度
380. Insert Delete GetRandom O(1)	380. O(1) 时间插入、删除和获取随机元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：哈希表，数据结构，数组，设计，随机算法**

实现 `RandomizedSet` 类：

1、 `RandomizedSet()` 初始化 `RandomizedSet` 对象

2、 `bool insert(int val)` 当元素 `val` 不存在时，向集合中插入该项，并返回 `true`；否则，返回 `false`。

3、 `bool remove(int val)` 当元素 `val` 存在时，从集合中移除该项，并返回 `true`；否则，返回 `false`。

4、 `int 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。

你必须实现类的所有函数，并满足每个函数的**平均**时间复杂度为 `O(1)`。

示例：

```
输入：
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove",
"insert", "getRandom"]
[[], [1], [2], [2], [], [1], [2], []]
输出：
[null, true, false, true, 2, true, false, 2]

解释：
RandomizedSet randomizedSet = new RandomizedSet();
randomizedSet.insert(1); // 向集合中插入 1。返回 true 表示 1 被成功地插入。
randomizedSet.remove(2); // 返回 false，表示集合中不存在 2。
randomizedSet.insert(2); // 向集合中插入 2。返回 true。集合现在包含 [1,2]。
randomizedSet.getRandom(); // getRandom 应随机返回 1 或 2。
randomizedSet.remove(1); // 从集合中移除 1，返回 true。集合现在包含 [2]。
randomizedSet.insert(2); // 2 已在集合中，所以返回 false。
randomizedSet.getRandom(); // 由于 2 是集合中唯一的数字，getRandom 总是返回 2。
```

基本思路

对于一个标准的 `HashSet`，你能否在 `O(1)` 的时间内实现 `getRandom` 函数？

其实是不能的，因为根据刚才说到的底层实现，元素是被哈希函数「分散」到整个数组里面的，更别说还有拉链法等等解决哈希冲突的机制，基本做不到 O(1) 时间等概率随机获取元素。

换句话说，对于 getRandom 方法，如果想「等概率」且「在 O(1) 的时间」取出元素，一定要满足：

底层用数组实现，且数组必须是紧凑的，这样我们就可以直接生成随机数作为索引，选取随机元素。

但如果用数组存储元素的话，常规的插入，删除的时间复杂度又不可能是 O(1)。

然而，对数组尾部进行插入和删除操作不会涉及数据搬移，时间复杂度是 O(1)。

所以，如果我们想在 O(1) 的时间删除数组中的某一个元素 val，可以先把这个元素交换到数组的尾部，然后再 pop 掉。

- **详细题解：常数时间删除/查找数组中的任意元素**

解法代码

```
class RandomizedSet {
    public:
    // 存储元素的值
    vector<int> nums;
    // 记录每个元素对应在 nums 中的索引
    unordered_map<int,int> valToIndex;

    bool insert(int val) {
        // 若 val 已存在，不用再插入
        if (valToIndex.count(val)) {
            return false;
        }
        // 若 val 不存在，插入到 nums 尾部，
        // 并记录 val 对应的索引值
        valToIndex[val] = nums.size();
        nums.push_back(val);
        return true;
    }

    bool remove(int val) {
        // 若 val 不存在，不用再删除
        if (!valToIndex.count(val)) {
            return false;
        }
        // 先拿到 val 的索引
        int index = valToIndex[val];
        // 将最后一个元素对应的索引修改为 index
        valToIndex[nums.back()] = index;
        // 交换 val 和最后一个元素
        swap(nums[index], nums.back());
        // 在数组中删除元素 val
        nums.pop_back();
        // 删除元素 val 对应的索引
        valToIndex.erase(val);
        return true;
```

```
    }

    int getRandom() {
        // 随机获取 nums 中的一个元素
        return nums[rand() % nums.size()];
    }
};
```

- **类似题目**：
  - 519. 随机翻转矩阵 🟠
  - 710. 黑名单中的随机数 🔴
  - 剑指 Offer II 030. 插入、删除和随机访问都是 O(1) 的容器 🟠

519. 随机翻转矩阵

LeetCode	力扣	难度
519. Random Flip Matrix	519. 随机翻转矩阵	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：哈希表，数据结构，随机算法**

给你一个 `m x n` 的二元矩阵 `matrix`，且所有值被初始化为 `0`。请你设计一个算法，随机选取一个满足 `matrix[i][j] == 0` 的下标 `(i, j)`，并将它的值变为 `1`。所有满足 `matrix[i][j] == 0` 的下标 `(i, j)` 被选取的概率应当均等。

尽量最少调用内置的随机函数，并且优化时间和空间复杂度。

实现 `Solution` 类：

- `Solution(int m, int n)` 使用二元矩阵的大小 `m` 和 `n` 初始化该对象。
- `int[] flip()` 返回一个满足 `matrix[i][j] == 0` 的随机下标 `[i, j]`，并将其对应格子中的值变为 `1`。
- `void reset()` 将矩阵中所有的值重置为 `0`。

示例：

```
输入
["Solution", "flip", "flip", "flip", "reset", "flip"]
[[3, 1], [], [], [], [], []]
输出
[null, [1, 0], [2, 0], [0, 0], null, [2, 0]]

解释
Solution solution = new Solution(3, 1);
solution.flip();  // 返回 [1, 0]，此时返回 [0,0]、[1,0] 和 [2,0] 的概率应当相同
solution.flip();  // 返回 [2, 0]，因为 [1,0] 已经返回过了，此时返回 [2,0] 和
[0,0] 的概率应当相同
solution.flip();  // 返回 [0, 0]，根据前面已经返回过的下标，此时只能返回 [0,0]
solution.reset(); // 所有值都重置为 0，并可以再次选择下标返回
solution.flip();  // 返回 [2, 0]，此时返回 [0,0]、[1,0] 和 [2,0] 的概率应当相同
```

基本思路

这道题的需求看起来很简单，但说实话这题并不是初学者能够高效解决的，下面我来一步步拆解这道题。

首先，你需要直到对多维坐标进行「降维」的技巧，把二维坐标映射到一维，因为我们生成的随机数可以认为是一维坐标。这个技巧不难，可以先去做一下 566. 重塑矩阵。

其次，你需要看过我的这篇文章 给我常数时间，我可以删除/查找数组中的任意元素 并跟着做了 380. 常数时间插入、删除和获取随机元素，题目说要尽可能减少调用随机函数，就是在考你把数组元素换到尾部的技巧。

本题其实面对的是相同的问题，相当于是给你一个长度为 `m * n` 的数组，请你每次取出一个随机数并删除它。

所以解决本题也使用同样的思路，抽象出一个长度为 `m * n` 的数组，每次取出一个随机元素之后把这个元素置换到数组尾部删掉，同时用一个哈希表记录被置换元素的映射，保证每次随机不会得到重复元素。

具体的实现看代码吧，本质上其实和 380 是一模一样的。

解法代码

```
class Solution {

    int m, n;
    // 抽象一维数组的长度
    int len;
    // 已删除元素到尾部未删除元素的映射
    HashMap<Integer, Integer> deletedToExist;
    // 这个 deletedToExist 存的到底是什么呢？举个例子你就明白了：
    // 假设数组 [1,2,3,4,5]，我第一次随机选择了元素 2，
    // 然后我就会在 deletedToExist 记录一个映射 2 -> 5，并删除最后的元素 5
    // 现在数组是 [1,2,3,4]，我第二次随机选择又选到了 2，
    // 但不能重复选择同一个元素，所以我选择 5，因为 deletedToExist[2] = 5
    // 然后我再更新 deletedToExist，记录 2 -> 4 并删除最后的元素 4
    // 以此类推，这样就能保证随机选择到相同的元素时也能等概率地返回不同的元素

    Random random = new Random();

    public Solution(int m, int n) {
        this.m = m;
        this.n = n;
        this.len = m * n;
        this.deletedToExist = new HashMap<>();
    }

    public int[] flip() {
        int rand = random.nextInt(len);
        // 这个随机数可能已经被删掉了（刚才已经被选过）
        int res = rand;
        if (deletedToExist.containsKey(rand)) {
            res = deletedToExist.get(rand);
        }
        // 把 rand 置换到数组尾部
        int last = len - 1;
        // 尾部的那个元素也可能已经被删掉了
        if (deletedToExist.containsKey(last)) {
            last = deletedToExist.get(last);
        }
        deletedToExist.put(rand, last);
```

```
            // 把尾部的这个元素删掉
            len--;
            // 一维坐标转化成二维坐标
            return new int[]{res / n, res % n};
        }

        public void reset() {
            this.len = this.m * this.n;
            this.deletedToExist.clear();
        }
    }
```

剑指 Offer II 030. 插入、删除和随机访问都是 O(1) 的容器

这道题和 380. O(1) 时间插入、删除和获取随机元素 相同。

460. LFU 缓存

LeetCode	力扣	难度
460. LFU Cache	460. LFU 缓存	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，设计

请你为最不经常使用缓存算法（LFU） 设计并实现数据结构。

实现 `LFUCache` 类：

1、 `LFUCache(int capacity)` 用数据结构的容量 `capacity` 初始化对象。

2、 `int get(int key)` 如果键存在于缓存中，则获取键的值，否则返回 -1。

3、 `void put(int key, int value)` 如果键已存在，则变更其值；如果键不存在，插入键值对。当缓存达到其容量时，则应该在插入新键值对之前，删除最不经常使用的键值对；若存在两个或更多个键具有相同使用频率时，应该去除**最近最久未使用**的键。

注：「键值对的使用次数」就是自插入该键以来对其调用 `get` 和 `put` 函数的次数之和，使用次数会在对应键被移除后置为 0。

示例：

```
输入：
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get",
"get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
输出：
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

解释：
// cnt(x) = 键 x 的使用计数
// cache=[] 将显示最后一次使用的顺序（最左边的元素是最近的）
LFUCache lFUCache = new LFUCache(2);
lFUCache.put(1, 1);   // cache=[1,_], cnt(1)=1
lFUCache.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lFUCache.get(1);      // 返回 1
                      // cache=[1,2], cnt(2)=1, cnt(1)=2
lFUCache.put(3, 3);   // 去除键 2，因为 cnt(2)=1，使用计数最小
                      // cache=[3,1], cnt(3)=1, cnt(1)=2
lFUCache.get(2);      // 返回 -1（未找到）
lFUCache.get(3);      // 返回 3
                      // cache=[3,1], cnt(3)=2, cnt(1)=2
lFUCache.put(4, 4);   // 去除键 1，1 和 3 的 cnt 相同，但 1 最久未使用
                      // cache=[4,3], cnt(4)=1, cnt(3)=2
```

```
lFUCache.get(1);      // 返回 -1（未找到）
lFUCache.get(3);      // 返回 3
                      // cache=[3,4], cnt(4)=1, cnt(3)=3
lFUCache.get(4);      // 返回 4
                      // cache=[3,4], cnt(4)=2, cnt(3)=3
```

基本思路

PS：这道题在《算法小抄》 的第 227 页。

总结下题目的要求：

1、调用 `get(key)` 方法时，要返回该 `key` 对应的 `val`。

2、只要用 `get` 或者 `put` 方法访问一次某个 `key`，该 `key` 的 `freq` 就要加一。

3、如果在容量满了的时候进行插入，则需要将 `freq` 最小的 `key` 删除，如果最小的 `freq` 对应多个 `key`，则删除其中最旧的那一个。

具体思路略微复杂，请查看详细题解。

- **详细题解：算法就像搭乐高：带你手撸 LFU 算法**

解法代码

```
class LFUCache {

    // key 到 val 的映射，我们后文称为 KV 表
    HashMap<Integer, Integer> keyToVal;
    // key 到 freq 的映射，我们后文称为 KF 表
    HashMap<Integer, Integer> keyToFreq;
    // freq 到 key 列表的映射，我们后文称为 FK 表
    HashMap<Integer, LinkedHashSet<Integer>> freqToKeys;
    // 记录最小的频次
    int minFreq;
    // 记录 LFU 缓存的最大容量
    int cap;

    public LFUCache(int capacity) {
        keyToVal = new HashMap<>();
        keyToFreq = new HashMap<>();
        freqToKeys = new HashMap<>();
        this.cap = capacity;
        this.minFreq = 0;
    }

    public int get(int key) {
        if (!keyToVal.containsKey(key)) {
            return -1;
        }
        // 增加 key 对应的 freq
```

```
        increaseFreq(key);
        return keyToVal.get(key);
    }

    public void put(int key, int val) {
        if (this.cap <= 0) return;

        /* 若 key 已存在，修改对应的 val 即可 */
        if (keyToVal.containsKey(key)) {
            keyToVal.put(key, val);
            // key 对应的 freq 加一
            increaseFreq(key);
            return;
        }

        /* key 不存在，需要插入 */
        /* 容量已满的话需要淘汰一个 freq 最小的 key */
        if (this.cap <= keyToVal.size()) {
            removeMinFreqKey();
        }

        /* 插入 key 和 val，对应的 freq 为 1 */
        // 插入 KV 表
        keyToVal.put(key, val);
        // 插入 KF 表
        keyToFreq.put(key, 1);
        // 插入 FK 表
        freqToKeys.putIfAbsent(1, new LinkedHashSet<>());
        freqToKeys.get(1).add(key);
        // 插入新 key 后最小的 freq 肯定是 1
        this.minFreq = 1;
    }

    private void increaseFreq(int key) {
        int freq = keyToFreq.get(key);
        /* 更新 KF 表 */
        keyToFreq.put(key, freq + 1);
        /* 更新 FK 表 */
        // 将 key 从 freq 对应的列表中删除
        freqToKeys.get(freq).remove(key);
        // 将 key 加入 freq + 1 对应的列表中
        freqToKeys.putIfAbsent(freq + 1, new LinkedHashSet<>());
        freqToKeys.get(freq + 1).add(key);
        // 如果 freq 对应的列表空了，移除这个 freq
        if (freqToKeys.get(freq).isEmpty()) {
            freqToKeys.remove(freq);
            // 如果这个 freq 恰好是 minFreq，更新 minFreq
            if (freq == this.minFreq) {
                this.minFreq++;
            }
        }
    }

    private void removeMinFreqKey() {
```

```
            // freq 最小的 key 列表
            LinkedHashSet<Integer> keyList = freqToKeys.get(this.minFreq);
            // 其中最先被插入的那个 key 就是该被淘汰的 key
            int deletedKey = keyList.iterator().next();
            /* 更新 FK 表 */
            keyList.remove(deletedKey);
            if (keyList.isEmpty()) {
                freqToKeys.remove(this.minFreq);
                // 问：这里需要更新 minFreq 的值吗？
            }
            /* 更新 KV 表 */
            keyToVal.remove(deletedKey);
            /* 更新 KF 表 */
            keyToFreq.remove(deletedKey);
        }
    }
```

895. 最大频率栈

LeetCode	力扣	难度
895. Maximum Frequency Stack	895. 最大频率栈	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数据结构，设计

实现 FreqStack，模拟类似栈的数据结构的操作的一个类。

FreqStack 有两个方法：

1、push(int x)，将整数 x 推入栈中。

2、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","pop","pop","
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]。
```

基本思路

我们仔细思考一下 push 和 pop 方法，难点如下：

1、每次 pop 时，必须要知道频率最高的元素是什么。

2、如果频率最高的元素有多个，还得知道哪个是最近 push 进来的元素是哪个。

为了实现上述难点，我们要做到以下几点：

1、肯定要有一个变量 `maxFreq` 记录当前栈中最高的频率是多少。

2、我们得知道一个频率 `freq` 对应的元素有哪些，且这些元素要有时间顺序。

3、随着 `pop` 的调用，每个 `val` 对应的频率会变化，所以还得维持一个映射记录每个 `val` 对应的 `freq`。

算法执行过程如下 GIF 所示：

- **详细题解：数据结构设计：最大栈**

解法代码

```
class FreqStack {
    // 记录 FreqStack 中元素的最大频率
    int maxFreq = 0;
    // 记录 FreqStack 中每个 val 对应的出现频率，后文就称为 VF 表
    HashMap<Integer, Integer> valToFreq = new HashMap<>();
    // 记录频率 freq 对应的 val 列表，后文就称为 FV 表
    HashMap<Integer, Stack<Integer>> freqToVals = new HashMap<>();

    public void push(int val) {
        // 修改 VF 表：val 对应的 freq 加一
        int freq = valToFreq.getOrDefault(val, 0) + 1;
        valToFreq.put(val, freq);
        // 修改 FV 表：在 freq 对应的列表加上 val
        freqToVals.putIfAbsent(freq, new Stack<>());
        freqToVals.get(freq).push(val);
        // 更新 maxFreq
        maxFreq = Math.max(maxFreq, freq);
    }

    public int pop() {
```

```
        // 修改 FV 表：pop 出一个 maxFreq 对应的元素 v
        Stack<Integer> vals = freqToVals.get(maxFreq);
        int v = vals.pop();
        // 修改 VF 表：v 对应的 freq 减一
        int freq = valToFreq.get(v) - 1;
        valToFreq.put(v, freq);
        // 更新 maxFreq
        if (vals.isEmpty()) {
            // 如果 maxFreq 对应的元素空了
            maxFreq--;
        }
        return v;
    }
}
```

1845. 座位预约管理系统

LeetCode	力扣	难度
1845. Seat Reservation Manager	1845. 座位预约管理系统	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数据结构，设计**

请你设计一个管理 `n` 个座位预约的系统，座位编号从 `1` 到 `n`。

请你实现 `SeatManager` 类：

- `SeatManager(int n)` 初始化一个 `SeatManager` 对象，它管理从 `1` 到 `n` 编号的 `n` 个座位。所有座位初始都是可预约的。
- `int reserve()` 返回可以预约座位的**最小编号**，此座位变为不可预约。
- `void unreserve(int seatNumber)` 将给定编号 `seatNumber` 对应的座位变成可以预约。

示例 1：

```
输入：
["SeatManager", "reserve", "reserve", "unreserve", "reserve", "reserve",
"reserve", "reserve", "unreserve"]
[[5], [], [], [2], [], [], [], [], [5]]
输出：
[null, 1, 2, null, 2, 3, 4, 5, null]

解释：
SeatManager seatManager = new SeatManager(5); // 初始化 SeatManager，有 5 个
座位。
seatManager.reserve();    // 所有座位都可以预约，所以返回最小编号的座位，也就是 1。
seatManager.reserve();    // 可以预约的座位为 [2,3,4,5]，返回最小编号的座位，也就
是 2。
seatManager.unreserve(2); // 将座位 2 变为可以预约，现在可预约的座位为
[2,3,4,5]。
seatManager.reserve();    // 可以预约的座位为 [2,3,4,5]，返回最小编号的座位，也就
是 2。
seatManager.reserve();    // 可以预约的座位为 [3,4,5]，返回最小编号的座位，也就是
3。
seatManager.reserve();    // 可以预约的座位为 [4,5]，返回最小编号的座位，也就是
4。
seatManager.reserve();    // 唯一可以预约的是座位 5，所以返回 5。
seatManager.unreserve(5); // 将座位 5 变为可以预约，现在可预约的座位为 [5]。
```

基本思路

这题是 379. 电话目录管理系统 的进阶版，那一道题返回的空闲号码可以随意，而这道题要求返回**最小**的座位编号。

其实很思路是一样的，只是这里需要用到能够按照元素大小自动排序的数据结构 优先级队列（二叉堆），直接看代码吧。

解法代码

```
class SeatManager {
    // 利用优先级队列自动排序，队头的元素就是最小的
    PriorityQueue<Integer> pq = new PriorityQueue<>();

    public SeatManager(int n) {
        // 初始化所有空闲座位
        for (int i = 1; i <= n; i++) {
            pq.offer(i);
        }
    }

    public int reserve() {
        // 拿出队头元素（最小）
        return pq.poll();
    }

    public void unreserve(int i) {
        pq.offer(i);
    }
}
```

94. 二叉树的中序遍历

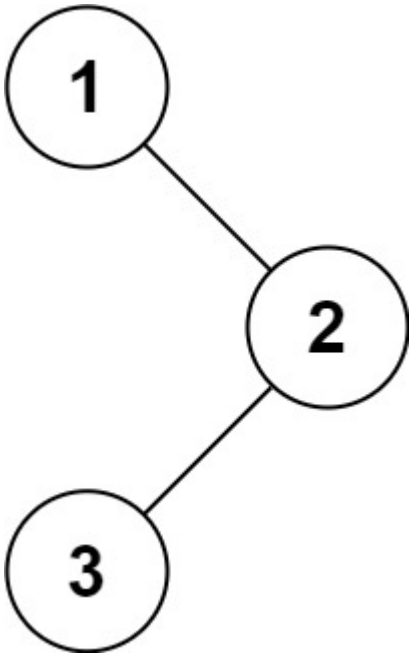
LeetCode	力扣	难度
94. Binary Tree Inorder Traversal	94. 二叉树的中序遍历	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给定一个二叉树的根节点 `root`，返回它的 **中序** 遍历。

示例 1：

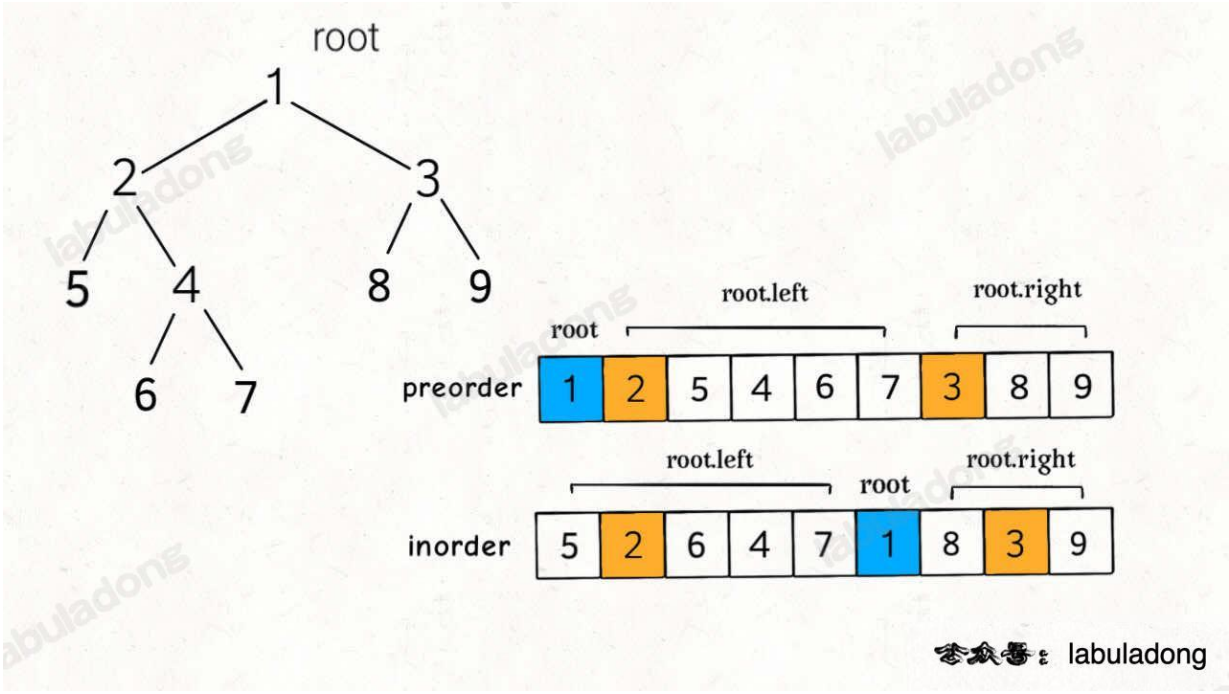



```
输入：root = [1,null,2,3]
输出：[1,3,2]
```

基本思路

不要瞧不起二叉树的遍历问题，前文 我的刷题经验总结 说过，二叉树的遍历代码是动态规划和回溯算法的祖宗。

动态规划思路的核心在于明确并利用函数的定义分解问题，中序遍历结果的特点是 `root.val` 在中间，左右子树在两侧：



回溯算法的核心很简单，就是 `traverse` 函数遍历二叉树。

本题就分别用两种不同的思路来写代码，注意体会两种思路的区别所在。

解法代码

```
class Solution {
    /* 动态规划思路 */
    // 定义：输入一个节点，返回以该节点为根的二叉树的中序遍历结果
    public List<Integer> inorderTraversal(TreeNode root) {
        LinkedList<Integer> res = new LinkedList<>();
        if (root == null) {
            return res;
        }
        res.addAll(inorderTraversal(root.left));
        res.add(root.val);
        res.addAll(inorderTraversal(root.right));
        return res;
    }

    /* 回溯算法思路 */
    LinkedList<Integer> res = new LinkedList<>();

    // 返回前序遍历结果
    public List<Integer> inorderTraversal2(TreeNode root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
```

```
            }
            traverse(root.left);
            // 中序遍历位置
            res.add(root.val);
            traverse(root.right);
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

100. 相同的树

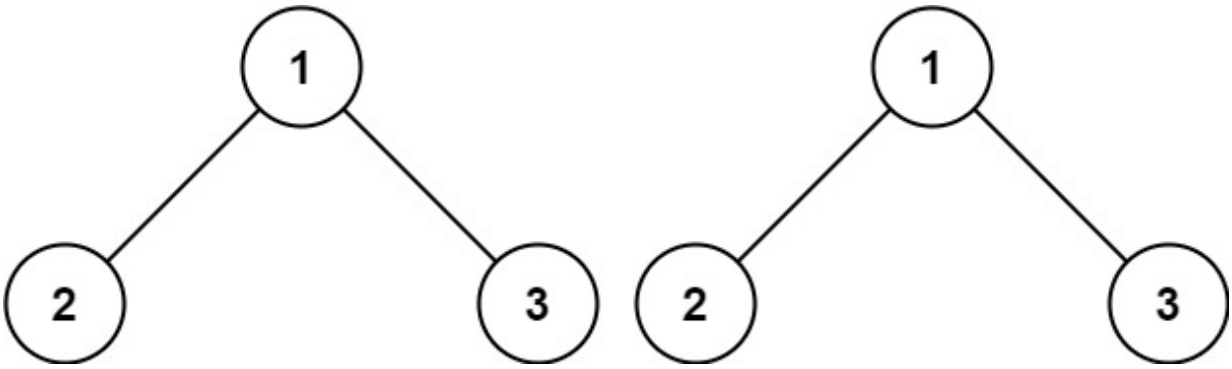
LeetCode	力扣	难度
100. Same Tree	100. 相同的树	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给你两棵二叉树的根节点 p 和 q，编写一个函数来检验这两棵树是否相同。如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

示例 1：




```
输入：p = [1,2,3], q = [1,2,3]
输出：true
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维模式。

判断两棵树是否相同，可以分解为判断根节点是否相同，然后判断左右子树的节点是否都相同。

解法代码

```
class Solution {
    // 定义：输入两个根节点，返回以它们为根的两棵二叉树是否相同
    public boolean isSameTree(TreeNode p, TreeNode q) {
        // 判断一对节点是否相同
        if (p == null && q == null) {
            return true;
        }
        if (p == null || q == null) {
            return false;
        }
```

```
            if (p.val != q.val) {
                return false;
            }
            // 判断其他节点是否相同
            return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
        }
    }
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 572. 另一棵树的子树 🟢

572. 另一棵树的子树

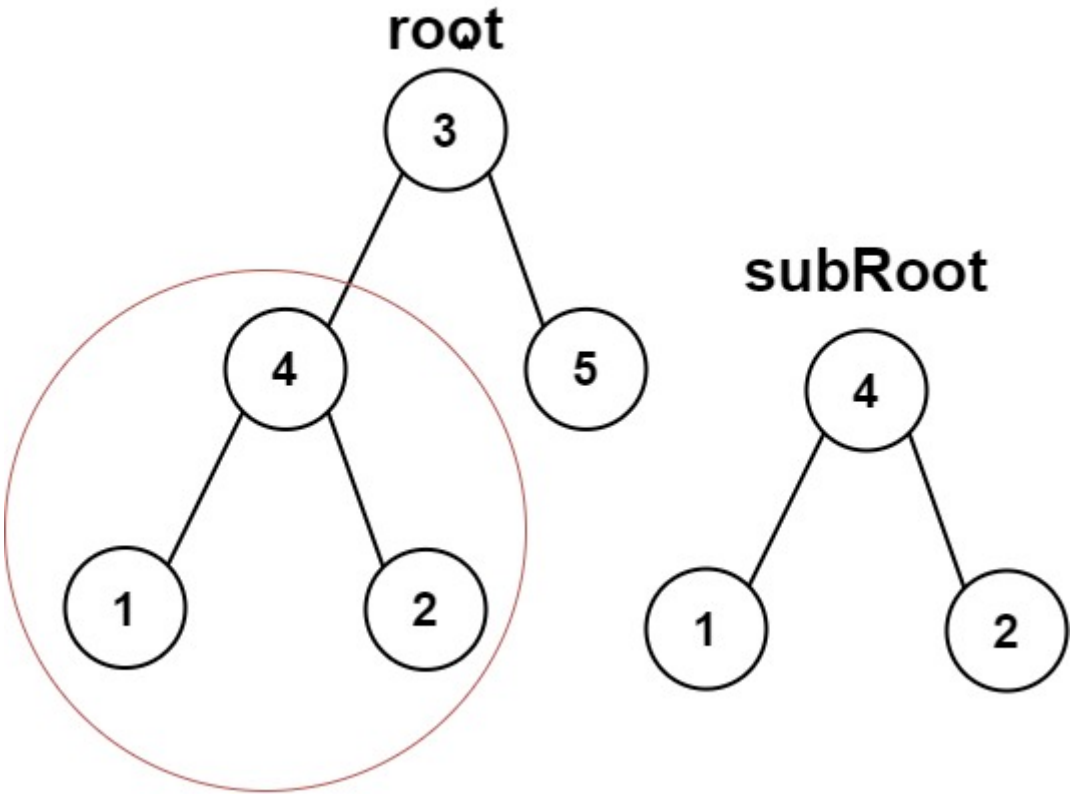
LeetCode	力扣	难度
572. Subtree of Another Tree	572. 另一棵树的子树	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给你两棵二叉树 `root` 和 `subRoot`。检验 `root` 中是否包含和 `subRoot` 具有相同结构和节点值的子树。如果存在，返回 `true`；否则，返回 `false`。

示例 1：




```
输入：root = [3,4,5,1,2], subRoot = [4,1,2]
输出：true
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

遍历以 `root` 为根的这棵二叉树的所有节点，用 100. 相同的树 中的 `isSame` 函数判断以该节点为根的子树是否和以 `subRoot` 为根的那棵树相同。

解法代码

```
class Solution {
    public boolean isSubtree(TreeNode root, TreeNode subRoot) {
        if (root == null) {
            return subRoot == null;
        }
        // 判断以 root 为根的二叉树是否和 subRoot 相同
        if (isSameTree(root, subRoot)) {
            return true;
        }
        // 去左右子树中判断是否有和 subRoot 相同的子树
        return isSubtree(root.left, subRoot) || isSubtree(root.right,
subRoot);
    }

    public boolean isSameTree(TreeNode p, TreeNode q) {
        // 判断一对节点是否相同
        if (p == null && q == null) {
            return true;
        }
        if (p == null || q == null) {
            return false;
        }
        if (p.val != q.val) {
            return false;
        }
        // 判断其他节点是否相同
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

102. 二叉树的层序遍历

LeetCode	力扣	难度
102. Binary Tree Level Order Traversal	102. 二叉树的层序遍历	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

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，从左到右访问所有节点）。

示例：
二叉树：`[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

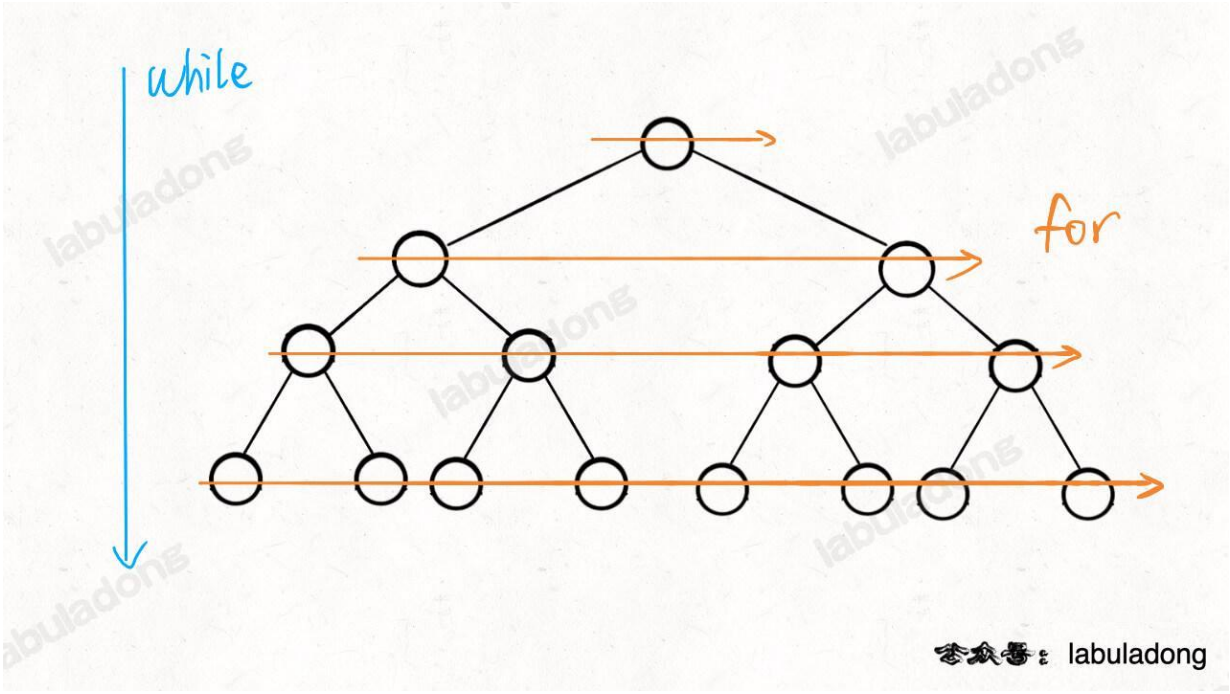
返回其层序遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

基本思路

前文 BFS 算法框架 就是由二叉树的层序遍历演变出来的。

下面是层序遍历的一般写法，通过一个 while 循环控制从上向下一层层遍历，for 循环控制每一层从左向右遍历：



解法代码

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录这一层的节点值
            List<Integer> level = new LinkedList<>();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                level.add(cur.val);
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            res.add(level);
        }
        return res;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 103. 二叉树的锯齿形层序遍历 🟠
  - 107. 二叉树的层序遍历 II 🟠
  - 1161. 最大层内元素和 🟠
  - 1302. 层数最深叶子节点的和 🟠
  - 1609. 奇偶树 🟠
  - 637. 二叉树的层平均值 🟢
  - 919. 完全二叉树插入器 🟠
  - 958. 二叉树的完全性检验 🟠
  - 剑指 Offer 32 - I. 从上到下打印二叉树 🟠
  - 剑指 Offer 32 - II. 从上到下打印二叉树 II 🟢
  - 剑指 Offer 32 - III. 从上到下打印二叉树 III 🟠

103. 二叉树的锯齿形层序遍历

LeetCode	力扣	难度
103. Binary Tree Zigzag Level Order Traversal	103. 二叉树的锯齿形层序遍历	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个二叉树，返回其节点值的锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：
给定二叉树 `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回锯齿形层序遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

基本思路

这题和 102. 二叉树的层序遍历 几乎是一样的，只要用一个布尔变量 `flag` 控制遍历方向即可。

解法代码

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
```

```
        // 为 true 时向右，false 时向左
        boolean flag = true;

        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录这一层的节点值
            LinkedList<Integer> level = new LinkedList<>();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                // 实现 z 字形遍历
                if (flag) {
                    level.addLast(cur.val);
                } else {
                    level.addFirst(cur.val);
                }
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            // 切换方向
            flag = !flag;
            res.add(level);
        }
        return res;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 1609. 奇偶树 🟠
  - 剑指 Offer 32 - III. 从上到下打印二叉树 III 🟠

107. 二叉树的层序遍历 II

LeetCode	力扣	难度
107. Binary Tree Level Order Traversal II	107. 二叉树的层序遍历 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个二叉树，返回其节点值自底向上的层序遍历（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）。

例如：
给定二叉树 `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其自底向上的层序遍历为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

基本思路

这题和 102. 二叉树的层序遍历 几乎是一样的，自顶向下的层序遍历反过来就行了。

解法代码

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        LinkedList<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
```

```
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录这一层的节点值
            List<Integer> level = new LinkedList<>();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                level.add(cur.val);
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            // 把每一层添加到头部，就是自底向上的层序遍历。
            res.addFirst(level);
        }
        return res;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

1161. 最大层内元素和

LeetCode	力扣	难度
1161. Maximum Level Sum of a Binary Tree	1161. 最大层内元素和	🟠

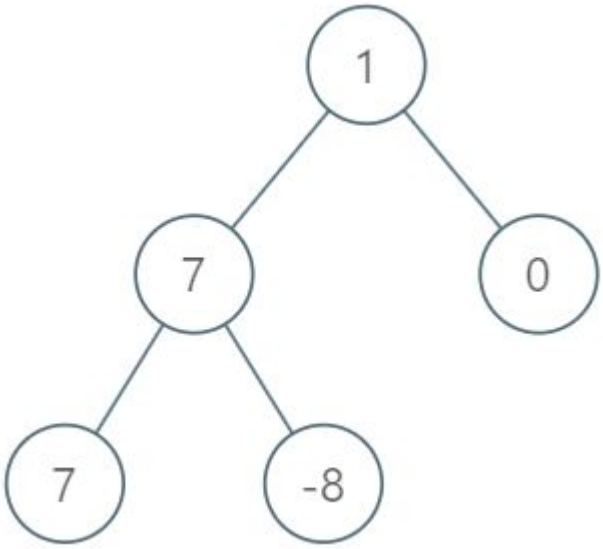
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树，二叉树vip

给你一个二叉树的根节点 `root`。设根节点位于二叉树的第 1 层，而根节点的子节点位于第 2 层，依此类推。

请你找出层内元素之和**最大**的那几层（可能只有一层）的层号，并返回其中**最小**的那个。

示例 1：




```
输入：root = [1,7,0,7,-8,null,null]
输出：2
解释：
第 1 层各元素之和为 1,
第 2 层各元素之和为 7 + 0 = 7,
第 3 层各元素之和为 7 + -8 = -1,
所以我们返回第 2 层的层号，它的层内元素之和最大。
```

基本思路

把 102. 二叉树的层序遍历 给出的层序遍历框架稍微变通即可解决这题。

解法代码

```java
class Solution {
    public int maxLevelSum(TreeNode root) {
        if (root == null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // 记录 BFS 走到的层数
        int depth = 1;
        // 记录元素和最大的那一行和最大元素和
        int res = 0, maxSum = Integer.MIN_VALUE;

        while (!q.isEmpty()) {
            int sz = q.size();
            int levelSum = 0;
            // 遍历这一层
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                levelSum += cur.val;

                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            if (levelSum > maxSum) {
                // 更新最大元素和
                res = depth;
                maxSum = levelSum;
            }
            depth++;
        }
        return res;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

1302. 层数最深叶子节点的和

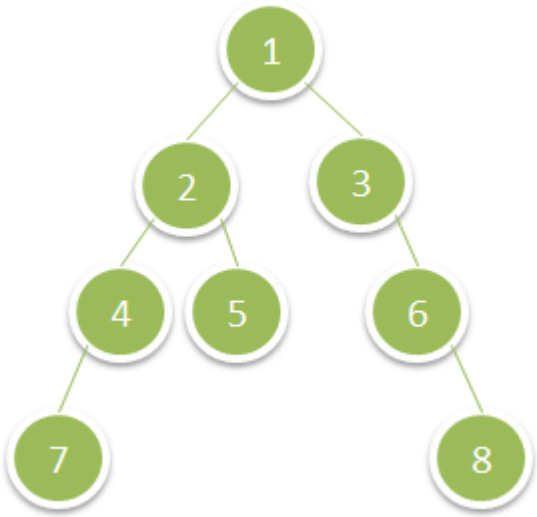
LeetCode	力扣	难度
1302. Deepest Leaves Sum	1302. 层数最深叶子节点的和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树，二叉树vip

给你一棵二叉树的根节点 `root`，请你返回 **层数最深的叶子节点的和**。

示例 1：




```
输入：root = [1,2,3,4,5,null,6,7,null,null,null,null,8]
输出：15
```

基本思路

这题用 DFS 或者 BFS 都可以，我就用 BFS 层序遍历算法吧，层序遍历算法参见 102. 层序遍历二叉树，这题只要把最后一层的节点值累加起来就行了。

解法代码

```
class Solution {
    public int deepestLeavesSum(TreeNode root) {
        if (root == null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);

        int sum = 0;
        while (!q.isEmpty()) {
```

```
                sum = 0;
                int sz = q.size();
                for (int i = 0; i < sz; i++) {
                    TreeNode cur = q.poll();
                    // 累加一层的节点之和
                    sum += cur.val;
                    if (cur.left != null)
                        q.offer(cur.left);
                    if (cur.right != null)
                        q.offer(cur.right);
                }
            }
            // 现在就是最后一层的节点值和
            return sum;
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

1609. 奇偶树

LeetCode	力扣	难度
1609. Even Odd Tree	1609. 奇偶树	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

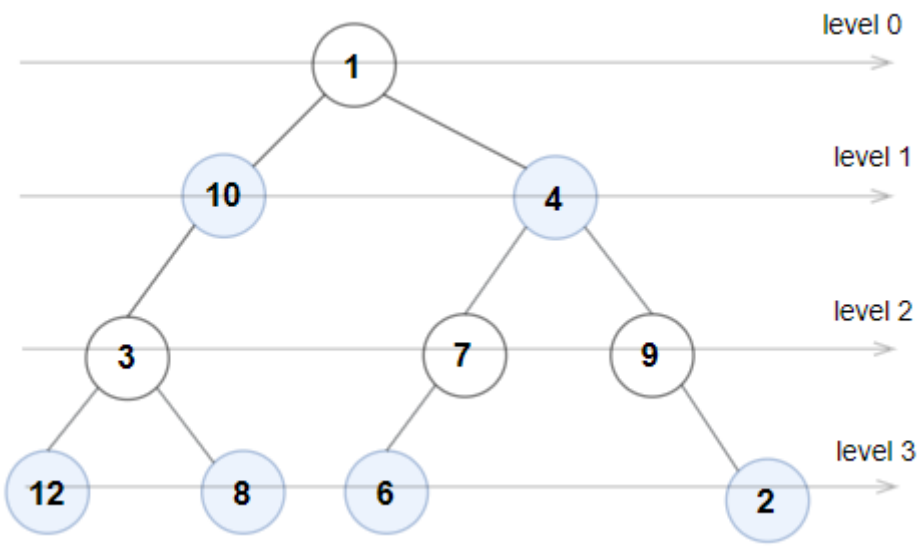
- **标签：二叉树，二叉树vip**

如果一棵二叉树满足下述几个条件，则可以称为 **奇偶树**：

- 二叉树根节点所在层下标为 `0`，根的子节点所在层下标为 `1`，根的孙节点所在层下标为 `2`，依此类推。
- **偶数下标**层上的所有节点的值都是**奇**整数，从左到右按顺序**严格递增**
- **奇数下标**层上的所有节点的值都是**偶**整数，从左到右按顺序**严格递减**

给你二叉树的根节点，如果二叉树为**奇偶树**，则返回 `true`，否则返回 `false`。

示例 1：




```
输入：root = [1,10,4,3,null,7,9,12,8,6,null,null,2]
输出：true
解释：每一层的节点值分别是：
0 层：[1]
1 层：[10,4]
2 层：[3,7,9]
3 层：[12,8,6,2]
由于 0 层和 2 层上的节点值都是奇数且严格递增，而 1 层和 3 层上的节点值都是偶数且严格递减，因此这是一棵奇偶树。
```

基本思路

这道题主要考察二叉树的层序遍历，你可以先做一下 102. 二叉树的层序遍历 和 103. 二叉树的锯齿形层序遍历 这两道题，然后再做这道题。具体思路可看解法代码的注释。

解法代码

```
class Solution {
    public boolean isEvenOddTree(TreeNode root) {
        if (root == null) {
            return true;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // 记录奇偶层数
        boolean even = true;
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录前一个节点，便于判断是否递增/递减
            int prev = even ? Integer.MIN_VALUE : Integer.MAX_VALUE;
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                if (even) {
                    // 偶数层
                    if (prev >= cur.val || cur.val % 2 == 0) {
                        return false;
                    }
                } else {
                    // 奇数层
                    if (prev <= cur.val || cur.val % 2 == 1) {
                        return false;
                    }
                }
                prev = cur.val;

                if (cur.left != null) {
                    q.offer(cur.left);
                }
                if (cur.right != null) {
                    q.offer(cur.right);
                }
            }
            // 奇偶层数切换
            even = !even;
        }
        return true;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

637. 二叉树的层平均值

LeetCode	力扣	难度
637. Average of Levels in Binary Tree	637. 二叉树的层平均值	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给定一个非空二叉树，返回一个由每层节点平均值组成的数组。

示例 1：

```
输入：
    3
   / \
  9  20
    /  \
   15   7
输出：[3, 14.5, 11]
解释：
第 0 层的平均值是 3 ,  第 1 层是 14.5 , 第 2 层是 11。因此返回 [3, 14.5, 11]。
```

基本思路

标准的二叉树层序遍历，把 102. 层序遍历二叉树 的代码稍微改一改就行了。

解法代码

```
class Solution {
    public List<Double> averageOfLevels(TreeNode root) {
        List<Double> res = new LinkedList<>();
        if (root == null) return res;

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            int size = q.size();
            // 记录当前层所有节点之和
            double sum = 0;
            for (int i = 0; i < size; i++) {
                TreeNode cur = q.poll();
                if (cur.left != null) {
                    q.offer(cur.left);
                }
                if (cur.right != null) {
```

```
                    q.offer(cur.right);
                }
                sum += cur.val;
            }
            // 记录当前行的平均值
            res.add(1.0 * sum / size);
        }

        return res;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

919. 完全二叉树插入器

LeetCode	力扣	难度
919. Complete Binary Tree Inserter	919. 完全二叉树插入器	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树，二叉树vip

完全二叉树是每一层（除最后一层外）都是完全填充（即节点数达到最大）的，并且所有的节点都尽可能地集中在左侧。

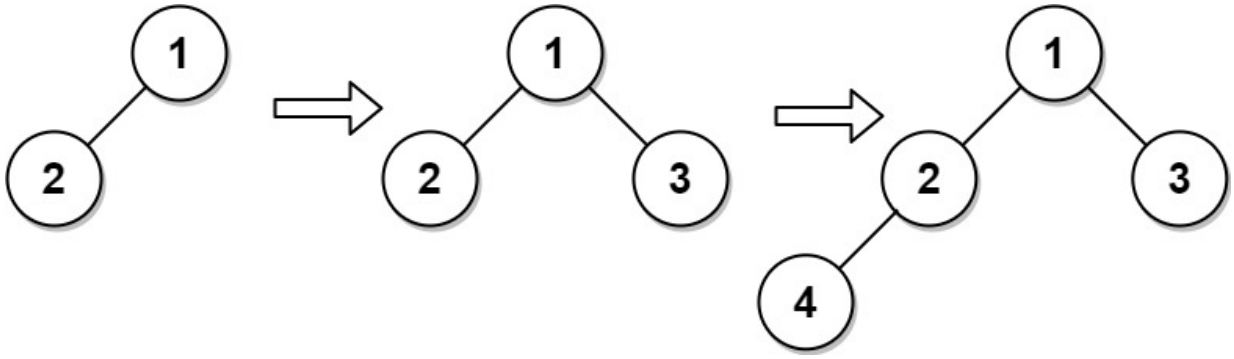
设计一个用完全二叉树初始化的数据结构 `CBTInserter`，它支持以下几种操作：

1、`CBTInserter(TreeNode root)` 使用头节点为 `root` 的给定树初始化该数据结构；

2、`CBTInserter.insert(int v)` 向树中插入一个新节点，节点类型为 `TreeNode`，值为 `v`。使树保持完全二叉树的状态，**并返回插入的新节点的父节点的值**；

3、`CBTInserter.get_root()` 将返回树的头节点。

示例 1：




```
输入：inputs = ["CBTInserter","insert","get_root"], inputs = [[[1]],[2],[]]
输出：[null,1,[1,2]]
```

基本思路

这道题考察二叉树的层序遍历，你需要先做 102. 二叉树的层序遍历 再做这道题，用队列维护底部可以进行插入的节点即可。

解法代码

```
class CBTInserter {
    // 这个队列只记录完全二叉树底部可以进行插入的节点
    private Queue<TreeNode> q = new LinkedList<>();
```

```
    private TreeNode root;

    public CBTInserter(TreeNode root) {
        this.root = root;
        // 进行普通的 BFS，目的是找到底部可插入的节点
        Queue<TreeNode> temp = new LinkedList<>();
        temp.offer(root);
        while (!temp.isEmpty()) {
            TreeNode cur = temp.poll();
            if (cur.left != null) {
                temp.offer(cur.left);
            }
            if (cur.right != null) {
                temp.offer(cur.right);
            }
            if (cur.right == null || cur.left == null) {
                // 找到完全二叉树底部可以进行插入的节点
                q.offer(cur);
            }
        }
    }

    public int insert(int val) {
        TreeNode node = new TreeNode(val);
        TreeNode cur = q.peek();
        // 进行插入
        if (cur.left == null) {
            cur.left = node;
        } else if (cur.right == null) {
            cur.right = node;
            q.poll();
        }
        // 新节点的左右节点也是可以插入的
        q.offer(node);
        return cur.val;
    }

    public TreeNode get_root() {
        return root;
    }
}
```

958. 二叉树的完全性检验

LeetCode	力扣	难度
958. Check Completeness of a Binary Tree	958. 二叉树的完全性检验	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

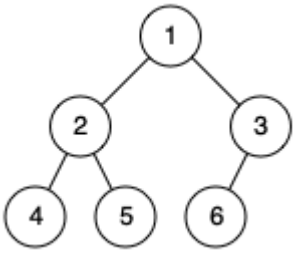
- **标签：二叉树，二叉树vip**

给定一个二叉树，确定它是否是一个**完全二叉树**。

百度百科中对完全二叉树的定义如下：

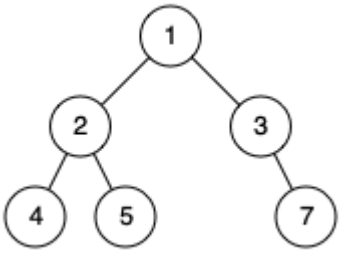
若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。

示例 1：




```
输入：[1,2,3,4,5,6]
输出：true
解释：最后一层前的每一层都是满的（即，结点值为 {1} 和 {2,3} 的两层），且最后一层中的所有结点（{4,5,6}）都尽可能地向左。
```

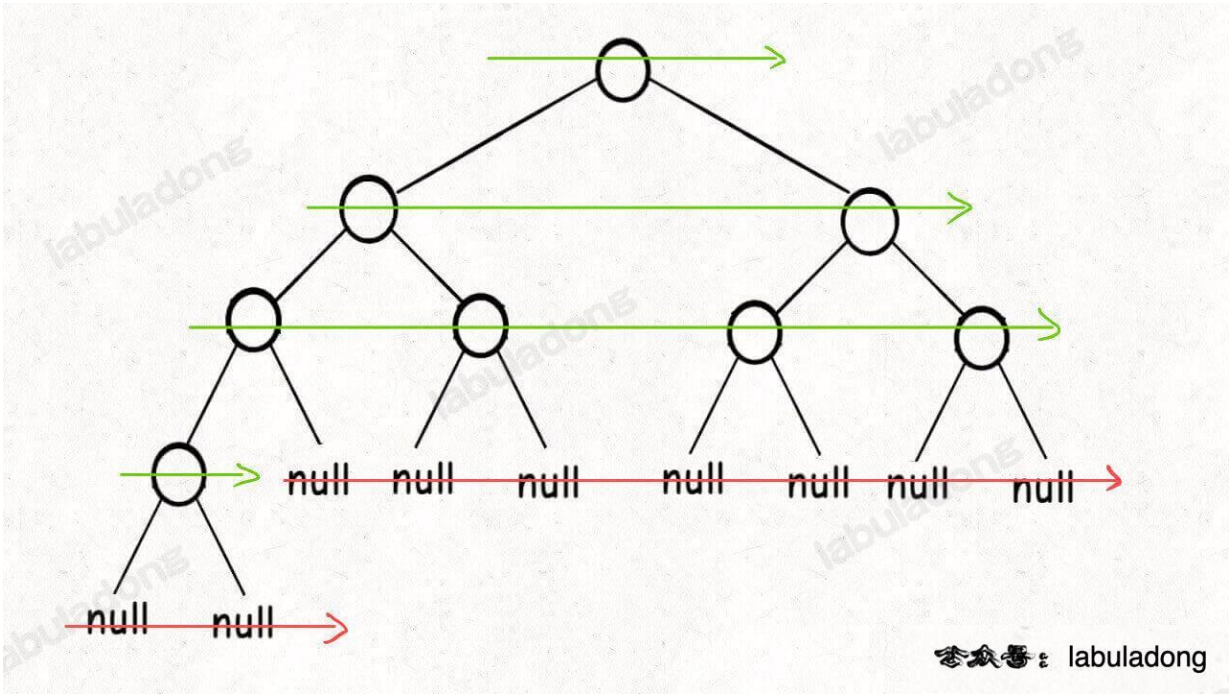
示例 2：




```
输入：[1,2,3,4,5,null,7]
输出：false
解释：值为 7 的结点没有尽可能靠向左侧。
```

基本思路

这题的关键是对完全二叉树特性的理解，**如果按照 BFS 层序遍历的方式遍历完全二叉树，队列最后留下的应该都是空指针**：



所以可以用 102. 二叉树的层序遍历 给出的层序遍历框架解决这题。

解法代码

```
class Solution {
    public boolean isCompleteTree(TreeNode root) {
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // 遍历完所有非空节点时变成 true
        boolean end = false;
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                if (cur == null) {
                    // 第一次遇到 null 时 end 变成 true
                    // 如果之后的所有节点都是 null，则说明是完全二叉树
                    end = true;
                } else {
                    if (end) {
                        // end 为 true 时遇到非空节点说明不是完全二叉树
                        return false;
                    }
                    // 将下一层节点放入队列，不用判断是否非空
                    q.offer(cur.left);
                    q.offer(cur.right);
```

```
                }
            }
        }
        return true;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

剑指 Offer 32 - I. 从上到下打印二叉树

这道题和 102. 二叉树的层序遍历 相同。

剑指 Offer 32 - II. 从上到下打印二叉树 II

这道题和 102. 二叉树的层序遍历 相同。

剑指 Offer 32 - III. 从上到下打印二叉树 III

这道题和 103. 二叉树的锯齿形层序遍历 相同。

104. 二叉树的最大深度

LeetCode	力扣	难度
104. Maximum Depth of Binary Tree	104. 二叉树的最大深度	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，动态规划，回溯算法**

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，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。

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

我的刷题经验总结 说过，二叉树问题虽然简单，但是暗含了动态规划和回溯算法等高级算法的思想。

下面提供两种思路的解法代码。

- **详细题解：东哥带你刷二叉树（纲领篇）**

解法代码

```
/***** 解法一，回溯算法思路 *****/
class Solution {

    int depth = 0;
    int res = 0;

    public int maxDepth(TreeNode root) {
        traverse(root);
        return res;
    }
```

```
    // 遍历二叉树
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        // 前序遍历位置
        depth++;
        // 遍历的过程中记录最大深度
        res = Math.max(res, depth);
        traverse(root.left);
        traverse(root.right);
        // 后序遍历位置
        depth--;
    }
}

/***** 解法二，动态规划思路 *****/
class Solution2 {
    // 定义：输入一个节点，返回以该节点为根的二叉树的最大深度
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 根据左右子树的最大深度推出原二叉树的最大深度
        return 1 + Math.max(leftMax, rightMax);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1376. 通知所有员工所需的时间 🟠
  - 144. 二叉树的前序遍历 🟢
  - 543. 二叉树的直径 🟢
  - 559. N 叉树的最大深度 🟢
  - 865. 具有所有最深节点的最小子树 🟠
  - 剑指 Offer 55 - I. 二叉树的深度 🟢

1376. 通知所有员工所需的时间

LeetCode	力扣	难度
1376. Time Needed to Inform All Employees	1376. 通知所有员工所需的时间	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip

公司里有 `n` 名员工，每个员工的 ID 都是独一无二的，编号从 `0` 到 `n - 1`。公司的总负责人通过 `headID` 进行标识。

在 `manager` 数组中，每个员工都有一个直属负责人，其中 `manager[i]` 是第 `i` 名员工的直属负责人。对于总负责人，`manager[headID] = -1`。题目保证从属关系可以用树结构显示。

公司总负责人想要向公司所有员工通告一条紧急消息。他将会首先通知他的直属下属们，然后由这些下属通知他们的下属，直到所有的员工都得知这条紧急消息。

第 `i` 名员工需要 `informTime[i]` 分钟来通知它的所有直属下属（也就是说在 `informTime[i]` 分钟后，他的所有直属下属都可以开始传播这一消息）。

返回通知所有员工这一紧急消息所需要的 **分钟数**。

示例 1：

```
输入：n = 1, headID = 0, manager = [-1], informTime = [0]
输出：0
解释：公司总负责人是该公司的唯一一名员工。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维模式。

我想这道题不用多说，其实稍微变通一下，这道题就是让你求多叉树的最大深度，只不过标准的 104. 二叉树的最大深度 每个节点的高度默认是 1，而这道题每个节点的高度记录在 `informTime` 中。

所以我们需要先将题目的输入数据格式转化成多叉树的形式，然后写一个函数计算多叉树的最大深度，具体看代码注释吧。

解法代码

```
class Solution {
    // 多叉树结构，key 是父节点，value 是子节点列表
    HashMap<Integer, List<Integer>> tree = new HashMap<>();
    int[] informTime;
```

```
    public int numOfMinutes(int n, int headID, int[] manager, int[]
informTime) {
        // 构造多叉树结构
        for (int i = 0; i < manager.length; i++) {
            int parent = manager[i];
            if (!tree.containsKey(parent)) {
                tree.put(parent, new ArrayList<>());
            }
            tree.get(parent).add(i);
        }
        this.informTime = informTime;
        return maxDepth(headID);
    }

    // 计算多叉树的最大深度
    int maxDepth(int root) {
        if (!tree.containsKey(root)) {
            // 叶子节点的深度为 0
            return 0;
        }

        int subMaxDepth = 0;
        for (int child : tree.get(root)) {
            subMaxDepth = Math.max(subMaxDepth, maxDepth(child));
        }
        // 以当前节点为根的多叉树的最大深度等于：
        // 当前节点的树枝长度 + 所有子树中最大深度的最大值
        return informTime[root] + subMaxDepth;
    }
}
```

144. 二叉树的前序遍历

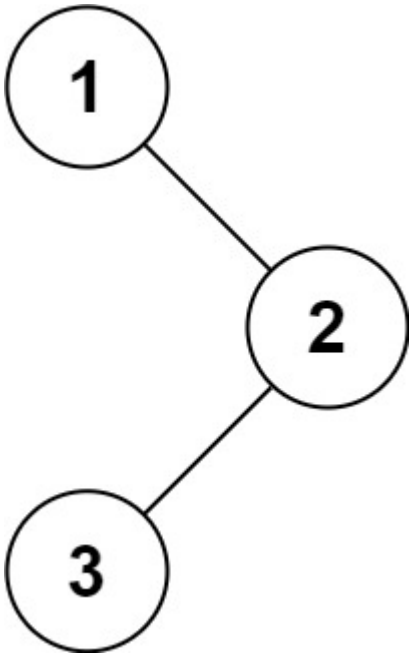
LeetCode	力扣	难度
144. Binary Tree Preorder Traversal	144. 二叉树的前序遍历	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给你二叉树的根节点 `root`，返回它节点值的**前序**遍历。

示例 1：




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

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

不要瞧不起二叉树的前中后序遍历。

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，分别代表回溯算法和动态规划的底层思想。

本题用两种思维模式来解答，注意体会其中思维方式的差异。

- 详细题解：东哥带你刷二叉树（纲领篇）

解法代码

```
class Solution {
    /* 动态规划思路 */
    // 定义：输入一个节点，返回以该节点为根的二叉树的前序遍历结果
    public List<Integer> preorderTraversal(TreeNode root) {
        LinkedList<Integer> res = new LinkedList<>();
        if (root == null) {
            return res;
        }
        // 前序遍历结果特点：第一个是根节点的值，接着是左子树，最后是右子树
        res.add(root.val);
        res.addAll(preorderTraversal(root.left));
        res.addAll(preorderTraversal(root.right));
        return res;
    }

    /* 回溯算法思路 */
    LinkedList<Integer> res = new LinkedList<>();

    // 返回前序遍历结果
    public List<Integer> preorderTraversal2(TreeNode root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        // 前序遍历位置
        res.add(root.val);
        traverse(root.left);
        traverse(root.right);
    }
}
```

- **类似题目**：
  - 104. 二叉树的最大深度 🟢
  - 543. 二叉树的直径 🟢
  - 剑指 Offer 55 - I. 二叉树的深度 🟢

543. 二叉树的直径

LeetCode	力扣	难度
543. Diameter of Binary Tree	543. 二叉树的直径	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，后序遍历**

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例：

给定二叉树

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

返回 **3**, 直径是路径 [4,2,1,3] 或者 [5,2,1,3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

所谓二叉树的直径，就是左右子树的最大深度之和，那么直接的想法是对每个节点计算左右子树的最大高度，得出每个节点的直径，从而得出最大的那个直径。

但是由于 `maxDepth` 也是递归函数，所以上述方式时间复杂度较高。

这题类似 366. 寻找二叉树的叶子节点，需要灵活运用二叉树的后序遍历，在 `maxDepth` 的后序遍历位置顺便计算最大直径。

- **详细题解：东哥带你刷二叉树（纲领篇）**

解法代码

```
class Solution {
    int maxDiameter = 0;

    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
```

```
        return maxDiameter;
    }

    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 后序遍历位置顺便计算最大直径
        maxDiameter = Math.max(maxDiameter, leftMax + rightMax);
        return 1 + Math.max(leftMax, rightMax);
    }
}

// 这是一种简单粗暴，但是效率不高的解法
class BadSolution {
    public int diameterOfBinaryTree(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 计算出左右子树的最大高度
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // root 这个节点的直径
        int res = leftMax + rightMax;
        // 递归遍历 root.left 和 root.right 两个子树
        return Math.max(res,
                Math.max(diameterOfBinaryTree(root.left),
                        diameterOfBinaryTree(root.right)));
    }

    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        return 1 + Math.max(leftMax, rightMax);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 104. 二叉树的最大深度 🟢
  - 124. 二叉树中的最大路径和 🔴
  - 1245. 树的直径 🟠
  - 144. 二叉树的前序遍历 🟢
  - 250. 统计同值子树 🟠
  - 366. 寻找二叉树的叶子节点 🟠

- 687. 最长同值路径 🟠
- 814. 二叉树剪枝 🟠
- 979. 在二叉树中分配硬币 🟠
- 剑指 Offer 55 - I. 二叉树的深度 🟢
- 剑指 Offer II 047. 二叉树剪枝 🟠
- 剑指 Offer II 051. 节点之和最大的路径 🔴

559. N 叉树的最大深度

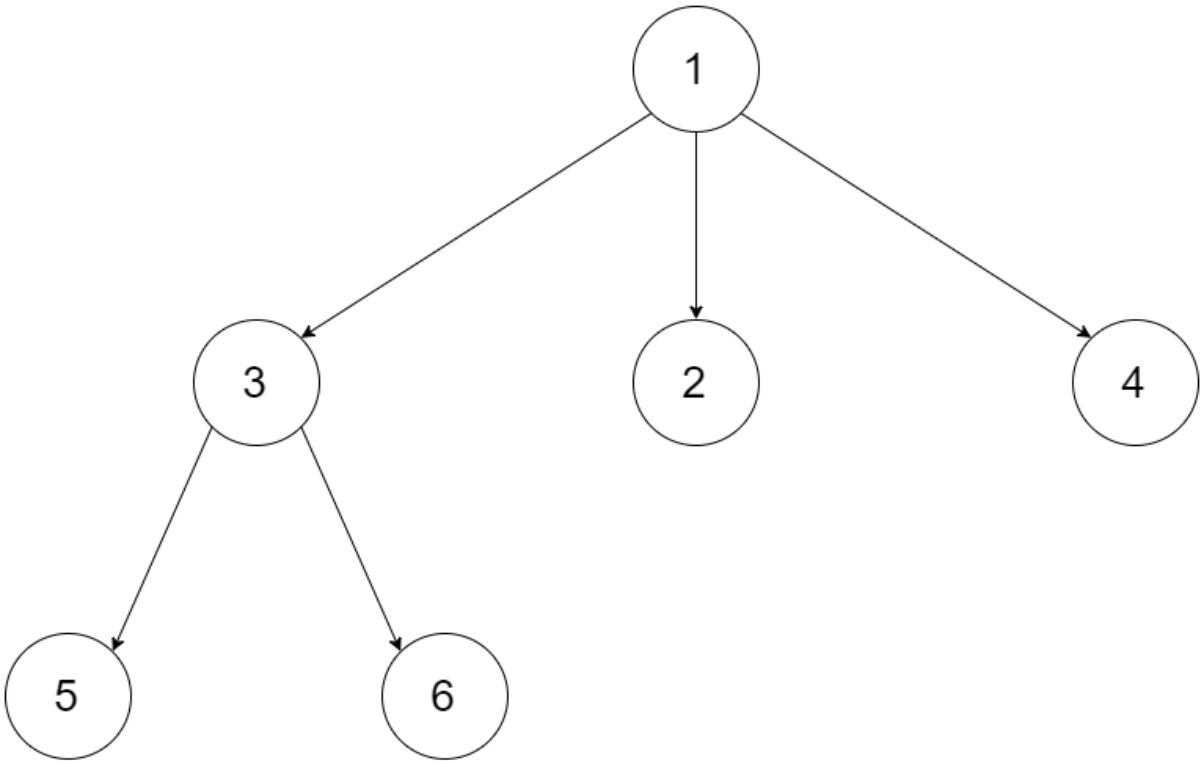
LeetCode	力扣	难度
559. Maximum Depth of N-ary Tree	559. N 叉树的最大深度	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给定一个 N 叉树，找到其最大深度。最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

示例 1：




```
输入：root = [1,null,3,2,4,null,5,6]
输出：3
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题可以同时使用两种思维模式，我把两种解法都写一下。

可以对照 104. 二叉树的最大深度 题的解法。

解法代码

```java
// 分解问题的思路
class Solution {
    public int maxDepth(Node root) {
        if (root == null) {
            return 0;
        }
        int subTreeMaxDepth = 0;
        for (Node child : root.children) {
            subTreeMaxDepth = Math.max(subTreeMaxDepth, maxDepth(child));
        }
        return 1 + subTreeMaxDepth;
    }
}

// 遍历的思路
class Solution2 {
    public int maxDepth(Node root) {
        traverse(root);
        return res;
    }

    // 记录递归遍历到的深度
    int depth = 0;
    // 记录最大的深度
    int res = 0;

    void traverse(Node root) {
        if (root == null) {
            return;
        }
        // 前序遍历位置
        depth++;
        res = Math.max(res, depth);

        for (Node child : root.children) {
            traverse(child);
        }
        // 后序遍历位置
        depth--;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

865. 具有所有最深节点的最小子树

LeetCode	力扣	难度
865. Smallest Subtree with all the Deepest Nodes	865. 具有所有最深节点的最小子树	

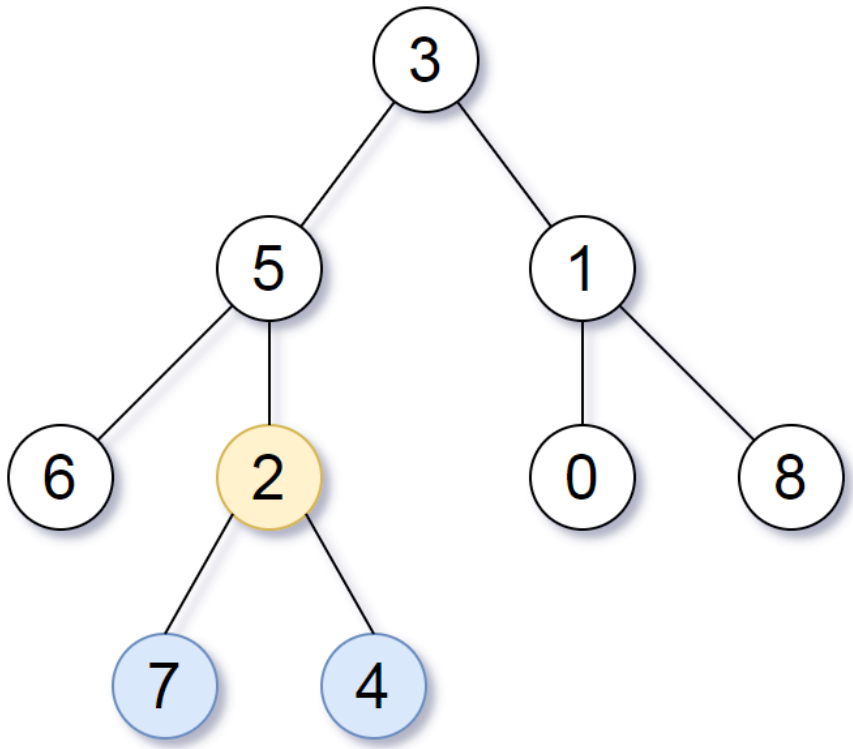
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树vip

给定一个根为 root 的二叉树，每个节点的深度是**该节点到根的最短距离**。

返回能满足**以该节点为根的子树中包含所有最深的节点**这一条件的具有最大深度的节点。

示例 1：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4]
输出：[2,7,4]
解释：
我们返回值为 2 的节点，在图中用黄色标记。
在图中用蓝色标记的是树的最深的节点。
注意，节点 5、3 和 2 包含树中最深的节点，但节点 2 的子树最小，因此我们返回它。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维，而且涉及处理子树，需要用后序遍历。

说到底，这道题就是让你求那些「最深」的叶子节点的最近公共祖先，可以看下前文 二叉树最近公共祖先。

你想想，一个节点需要知道哪些信息，才能确定自己是最深叶子节点的最近公共祖先?

它需要知道自己的左右子树的最大深度：如果左右子树一样深，那么当前节点就是最近公共祖先；如果左右子树不一样深，那么最深叶子节点的最近公共祖先肯定在左右子树上。

所以我们新建一个 `Result` 类，存放左右子树的最大深度及叶子节点的最近公共祖先节点，其余逻辑类似 104. 二叉树的最大深度。

解法代码

```
class Solution {
    class Result {
        public TreeNode node;
        public int depth;

        public Result(TreeNode node, int depth) {
            // 记录最近公共祖先节点 node
            this.node = node;
            // 记录以 node 为根的二叉树最大深度
            this.depth = depth;
        }
    }

    public TreeNode subtreeWithAllDeepest(TreeNode root) {
        Result res = maxDepth(root);
        return res.node;
    }

    // 定义：输入一棵二叉树，返回该二叉树的最大深度以及最深叶子节点的最近公共祖先节点
    Result maxDepth(TreeNode root) {
        if (root == null) {
            return new Result(null, 0);
        }
        Result left = maxDepth(root.left);
        Result right = maxDepth(root.right);
        if (left.depth == right.depth) {
            // 当左右子树的最大深度相同时，这个根节点是新的最近公共祖先
            // 以当前 root 节点为根的子树深度是子树深度 + 1
            return new Result(root, left.depth + 1);
        }
        // 左右子树的深度不同，则最近公共祖先在 depth 较大的一边
        Result res = left.depth > right.depth ? left : right;
        // 正确维护二叉树的最大深度
        res.depth++;

        return res;
```

```
        }
    }
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 1123. 最深叶节点的最近公共祖先 🟠

剑指 Offer 55 - I. 二叉树的深度

这道题和 104. 二叉树的最大深度 相同。

1123. 最深叶节点的最近公共祖先

LeetCode	力扣	难度
1123. Lowest Common Ancestor of Deepest Leaves	1123. 最深叶节点的最近公共祖先	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给你一个有根节点的二叉树，找到它最深的叶节点的最近公共祖先。

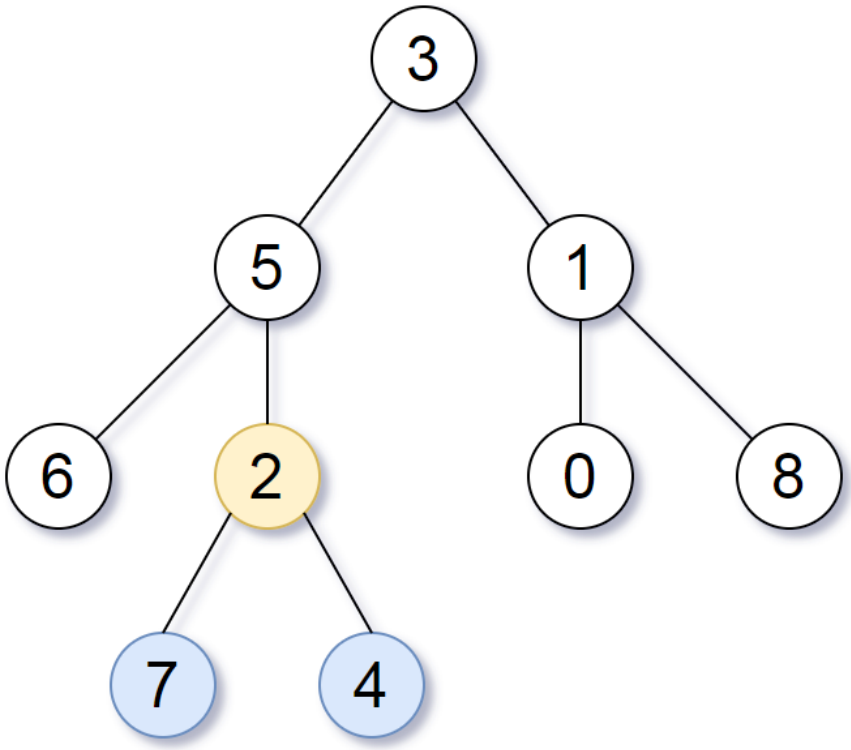
回想一下：

1、**叶节点**是二叉树中没有子节点的节点

2、树的根节点的**深度**为 `0`，如果某一节点的深度为 `d`，那它的子节点的深度就是 `d+1`

3、如果我们假定 `A` 是一组节点 `S` 的**最近公共祖先**，`S` 中的每个节点都在以 `A` 为根节点的子树中，且 `A` 的深度达到此条件下可能的最大值。

示例 1：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4]
输出：[2,7,4]
解释：
我们返回值为 2 的节点，在图中用黄色标记。
```

在图中用蓝色标记的是树的最深的节点。
注意，节点 6、0 和 8 也是叶节点，但是它们的深度是 2，而节点 7 和 4 的深度是 3。

基本思路

这题和 865. 具有所有最深节点的最小子树 一模一样，思路见那道题。

解法代码

```
class Solution {
    class Result {
        public TreeNode node;
        public int depth;

        public Result(TreeNode node, int depth) {
            // 记录最近公共祖先节点 node
            this.node = node;
            // 记录以 node 为根的二叉树最大深度
            this.depth = depth;
        }
    }

    public TreeNode lcaDeepestLeaves(TreeNode root) {
        Result res = maxDepth(root);
        return res.node;
    }

    // 定义：输入一棵二叉树，返回该二叉树的最大深度以及最深叶子节点的最近公共祖先节点
    Result maxDepth(TreeNode root) {
        if (root == null) {
            return new Result(null, 0);
        }
        Result left = maxDepth(root.left);
        Result right = maxDepth(root.right);
        if (left.depth == right.depth) {
            // 当左右子树的最大深度相同时，这个根节点是新的最近公共祖先
            return new Result(root, left.depth + 1);
        }
        // 左右子树的深度不同，则最近公共祖先在 depth 较大的一边
        Result res = left.depth > right.depth ? left : right;
        // 正确维护二叉树的最大深度
        res.depth++;

        return res;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

105. 从前序与中序遍历序列构造二叉树

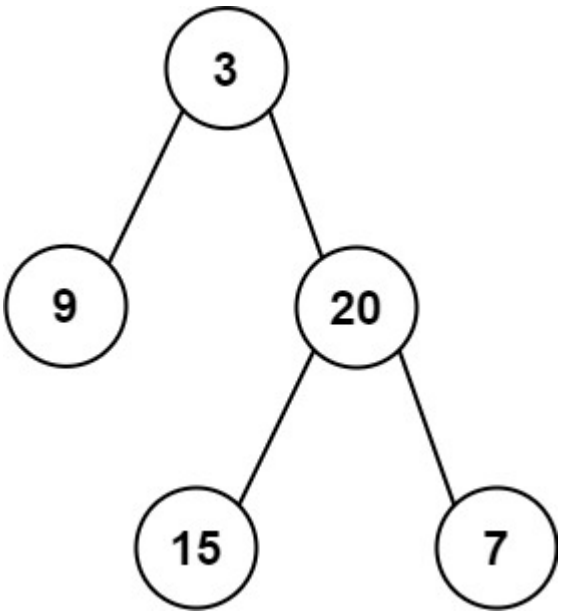
LeetCode	力扣	难度
105. Construct Binary Tree from Preorder and Inorder Traversal	105. 从前序与中序遍历序列构造二叉树	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构

给定一棵树的前序遍历结果 `preorder` 与中序遍历结果 `inorder`，请构造二叉树并返回其根节点。

示例 1:

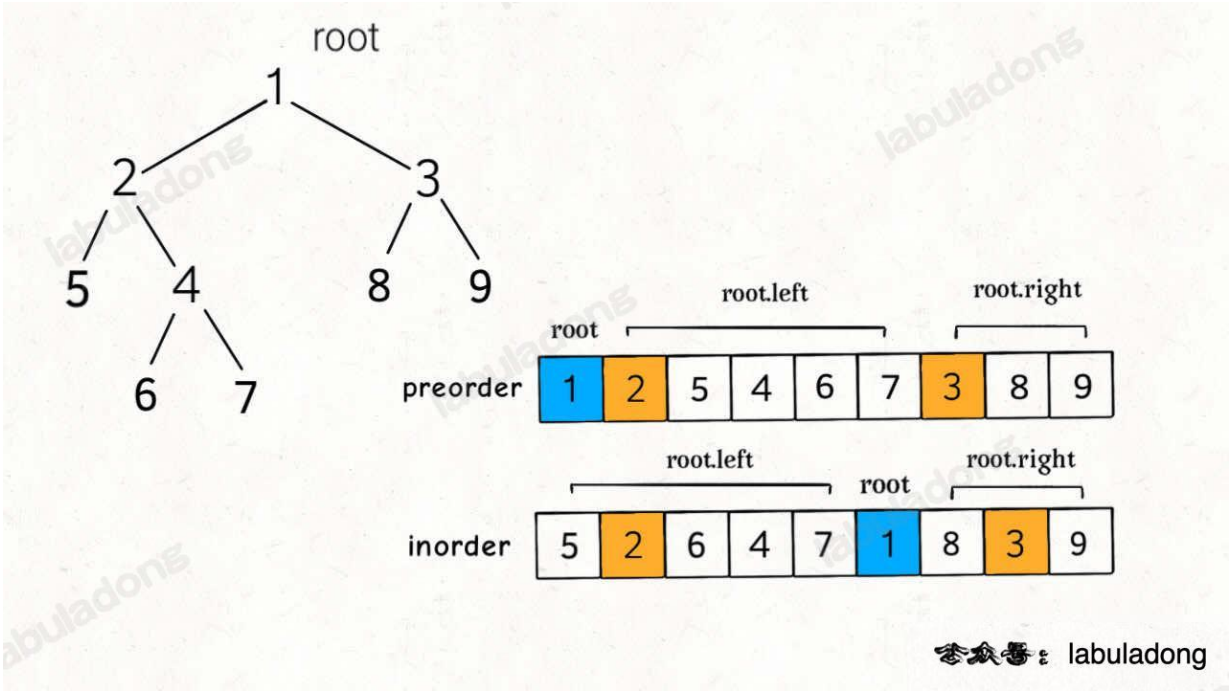



```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

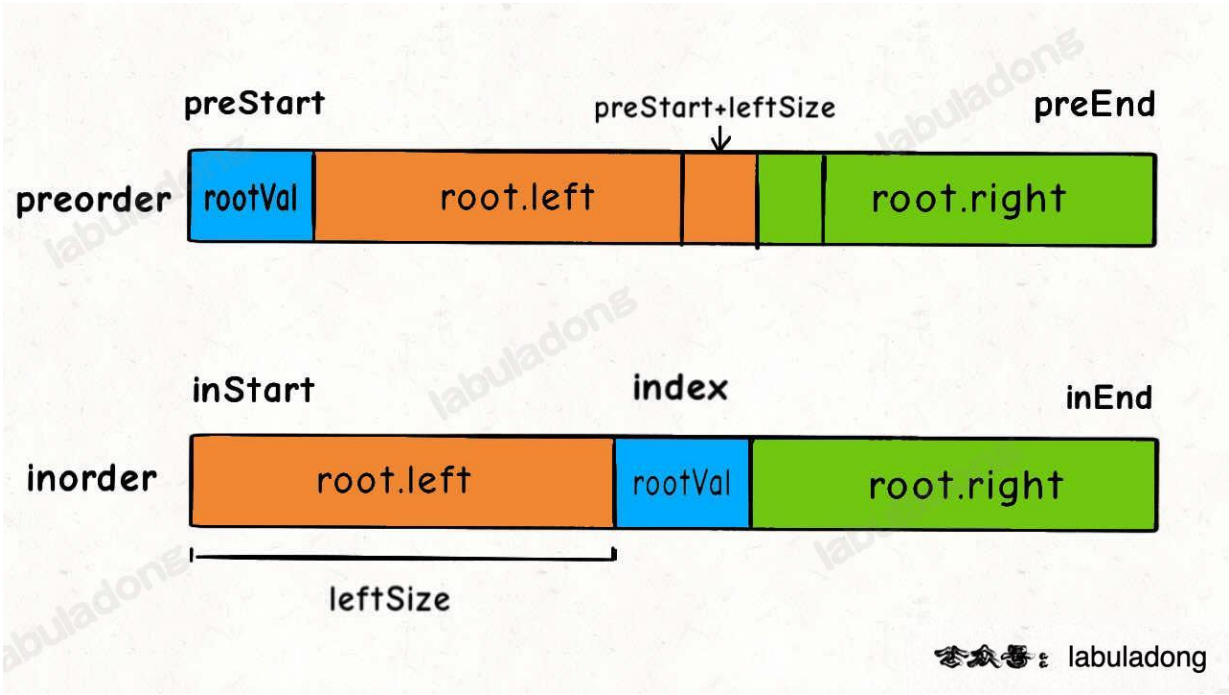
基本思路

构造二叉树，第一件事一定是找根节点，然后想办法构造左右子树。

二叉树的前序和中序遍历结果的特点如下：



前序遍历结果第一个就是根节点的值，然后再根据中序遍历结果确定左右子树的节点。



结合这个图看代码辅助理解。

- **详细题解：东哥带你刷二叉树（构造篇）**

解法代码

```
class Solution {
    // 存储 inorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        for (int i = 0; i < inorder.length; i++) {
```

```
            valToIndex.put(inorder[i], i);
        }
        return build(preorder, 0, preorder.length - 1,
                     inorder, 0, inorder.length - 1);
    }

    /*
       定义：前序遍历数组为 preorder[preStart..preEnd]，
       中序遍历数组为 inorder[inStart..inEnd]，
       构造这个二叉树并返回该二叉树的根节点
    */
    TreeNode build(int[] preorder, int preStart, int preEnd,
                   int[] inorder, int inStart, int inEnd) {
        if (preStart > preEnd) {
            return null;
        }

        // root 节点对应的值就是前序遍历数组的第一个元素
        int rootVal = preorder[preStart];
        // rootVal 在中序遍历数组中的索引
        int index = valToIndex.get(rootVal);

        int leftSize = index - inStart;

        // 先构造出当前根节点
        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        root.left = build(preorder, preStart + 1, preStart + leftSize,
                inorder, inStart, index - 1);

        root.right = build(preorder, preStart + leftSize + 1, preEnd,
                inorder, index + 1, inEnd);
        return root;
    }
}
```

- **类似题目**：
  - 106. 从中序与后序遍历序列构造二叉树 🟠
  - 654. 最大二叉树 🟠
  - 889. 根据前序和后序遍历构造二叉树 🟠
  - 剑指 Offer 07. 重建二叉树 🟠

106. 从中序与后序遍历序列构造二叉树

LeetCode	力扣	难度
106. Construct Binary Tree from Inorder and Postorder Traversal	106. 从中序与后序遍历序列构造二叉树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，数据结构**

根据一棵树的中序遍历与后序遍历构造二叉树，你可以假设树中没有重复的元素。

例如，给出

```
中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
```

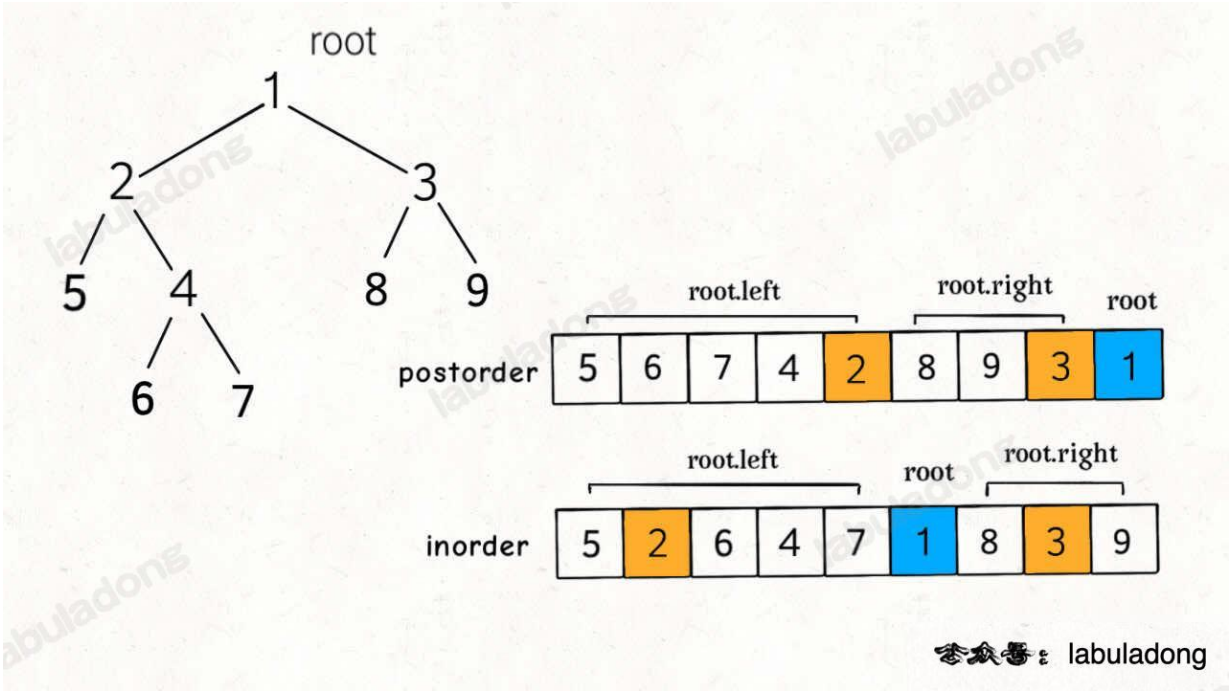
返回如下的二叉树：

```
    3
   / \
  9  20
    /  \
   15   7
```

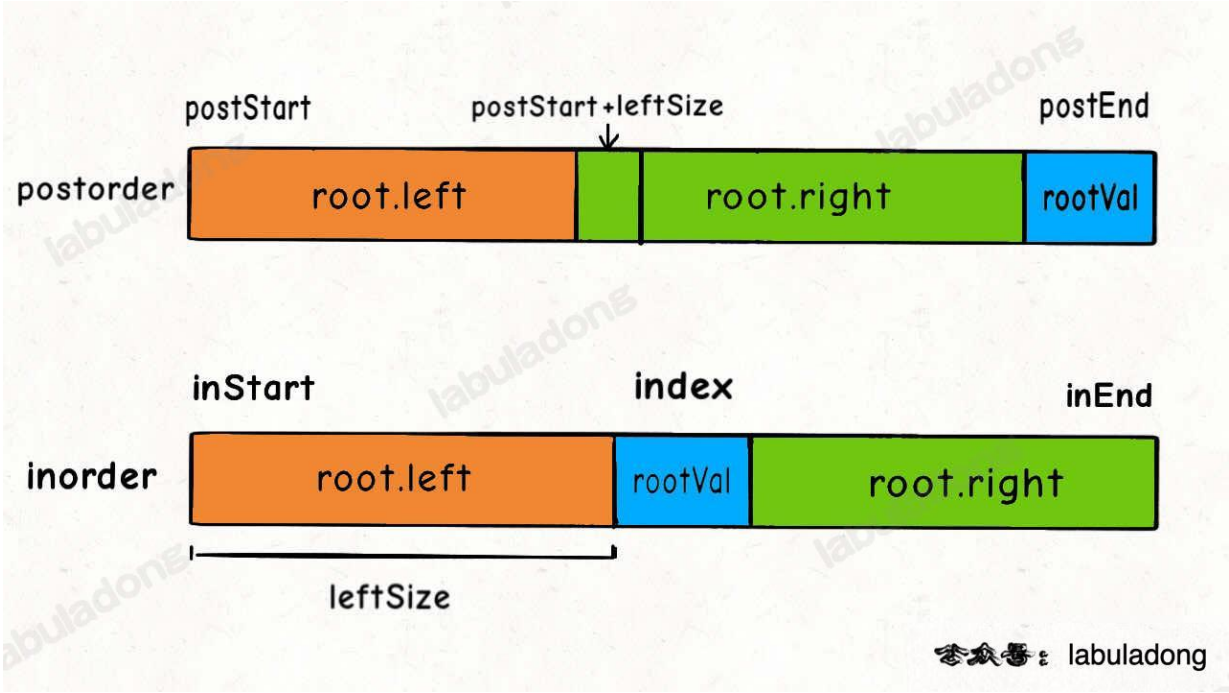
基本思路

构造二叉树，第一件事一定是找根节点，然后想办法构造左右子树。

二叉树的后序和中序遍历结果的特点如下：



后序遍历结果最后一个就是根节点的值，然后再根据中序遍历结果确定左右子树的节点。



结合这个图看代码辅助理解。

- **详细题解：东哥带你刷二叉树（构造篇）**

解法代码

```
class Solution {
    // 存储 inorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode buildTree(int[] inorder, int[] postorder) {
        for (int i = 0; i < inorder.length; i++) {
```

```
            valToIndex.put(inorder[i], i);
        }
        return build(inorder, 0, inorder.length - 1,
                     postorder, 0, postorder.length - 1);
    }

    /*
       定义：
       中序遍历数组为 inorder[inStart..inEnd]，
       后序遍历数组为 postorder[postStart..postEnd]，
       构造这个二叉树并返回该二叉树的根节点
    */
    TreeNode build(int[] inorder, int inStart, int inEnd,
                   int[] postorder, int postStart, int postEnd) {

        if (inStart > inEnd) {
            return null;
        }
        // root 节点对应的值就是后序遍历数组的最后一个元素
        int rootVal = postorder[postEnd];
        // rootVal 在中序遍历数组中的索引
        int index = valToIndex.get(rootVal);
        // 左子树的节点个数
        int leftSize = index - inStart;
        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        root.left = build(inorder, inStart, index - 1,
                          postorder, postStart, postStart + leftSize - 1);

        root.right = build(inorder, index + 1, inEnd,
                           postorder, postStart + leftSize, postEnd - 1);
        return root;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 105. 从前序与中序遍历序列构造二叉树 🟠
  - 654. 最大二叉树 🟠
  - 889. 根据前序和后序遍历构造二叉树 🟠
  - 剑指 Offer 07. 重建二叉树 🟠

654. 最大二叉树

LeetCode	力扣	难度
654. Maximum Binary Tree	654. 最大二叉树	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构

给定一个不含重复元素的整数数组 nums，一个以此数组直接递归构建的**最大二叉树**定义如下：

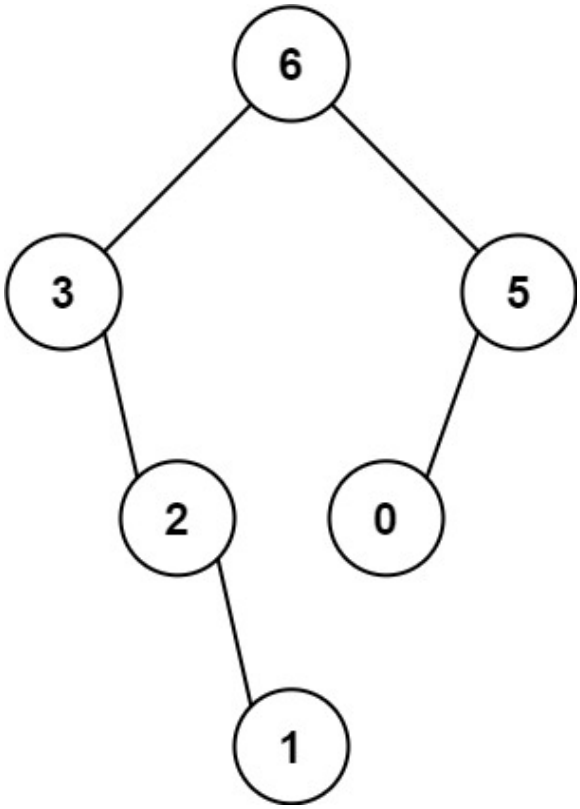
1、二叉树的根是数组 nums 中的最大元素。

2、左子树是通过数组中**最大元素左边的部分**递归构造出的最大二叉树。

3、右子树是通过数组中**最大元素右边的部分**递归构造出的最大二叉树。

返回由给定数组 nums 构建的**最大二叉树**。

示例 1：




```
输入：nums = [3,2,1,6,0,5]
输出：[6,3,5,null,2,0,null,null,1]
解释：递归调用如下所示：
- [3,2,1,6,0,5] 中的最大值是 6，左边部分是 [3,2,1]，右边部分是 [0,5]。
    - [3,2,1] 中的最大值是 3，左边部分是 []，右边部分是 [2,1]。
        - 空数组，无子节点。
```

```
        - [2,1] 中的最大值是 2，左边部分是 []，右边部分是 [1]。
            - 空数组，无子节点。
            - 只有一个元素，所以子节点是一个值为 1 的节点。
    - [0,5] 中的最大值是 5，左边部分是 [0]，右边部分是 []。
        - 只有一个元素，所以子节点是一个值为 0 的节点。
        - 空数组，无子节点。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归算法可以分两类，一类是遍历二叉树的类型，一类是分解子问题的类型。

前者较简单，只要运用二叉树的递归遍历框架即可；后者的关键在于明确递归函数的定义，然后利用这个定义。

这题是后者，函数 `build` 的定义是根据输入的数组构造最大二叉树，那么只要我先要找到根节点，然后让 `build` 函数递归生成左右子树即可。

- **详细题解：东哥带你刷二叉树（构造篇）**

解法代码

```
class Solution {
    /* 主函数 */
    public TreeNode constructMaximumBinaryTree(int[] nums) {
        return build(nums, 0, nums.length - 1);
    }

    /* 定义：将 nums[lo..hi] 构造成符合条件的树，返回根节点 */
    TreeNode build(int[] nums, int lo, int hi) {
        // base case
        if (lo > hi) {
            return null;
        }

        // 找到数组中的最大值和对应的索引
        int index = -1, maxVal = Integer.MIN_VALUE;
        for (int i = lo; i <= hi; i++) {
            if (maxVal < nums[i]) {
                index = i;
                maxVal = nums[i];
            }
        }

        TreeNode root = new TreeNode(maxVal);
        // 递归调用构造左右子树
        root.left = build(nums, lo, index - 1);
        root.right = build(nums, index + 1, hi);

        return root;
```

```
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 105. 从前序与中序遍历序列构造二叉树 🟠
  - 106. 从中序与后序遍历序列构造二叉树 🟠
  - 889. 根据前序和后序遍历构造二叉树 🟠
  - 998. 最大二叉树 II 🟠
  - 剑指 Offer 07. 重建二叉树 🟠

889. 根据前序和后序遍历构造二叉树

LeetCode	力扣	难度
889. Construct Binary Tree from Preorder and Postorder Traversal	889. 根据前序和后序遍历构造二叉树	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树**

返回与给定的前序和后序遍历匹配的任何二叉树。

`pre` 和 `post` 遍历中的值是不同的正整数。

示例：

```
输入：pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
输出：[1,2,3,4,5,6,7]
```

基本思路

做这道题之前，建议你先看一下 东哥手把手帮你刷通二叉树|第二期，做一下 105. 从前序与中序遍历序列构造二叉树（中等） 和 106. 从中序与后序遍历序列构造二叉树（中等） 这两道题。

这道题让用后序遍历和前序遍历结果还原二叉树，和前两道题有一个本质的区别：

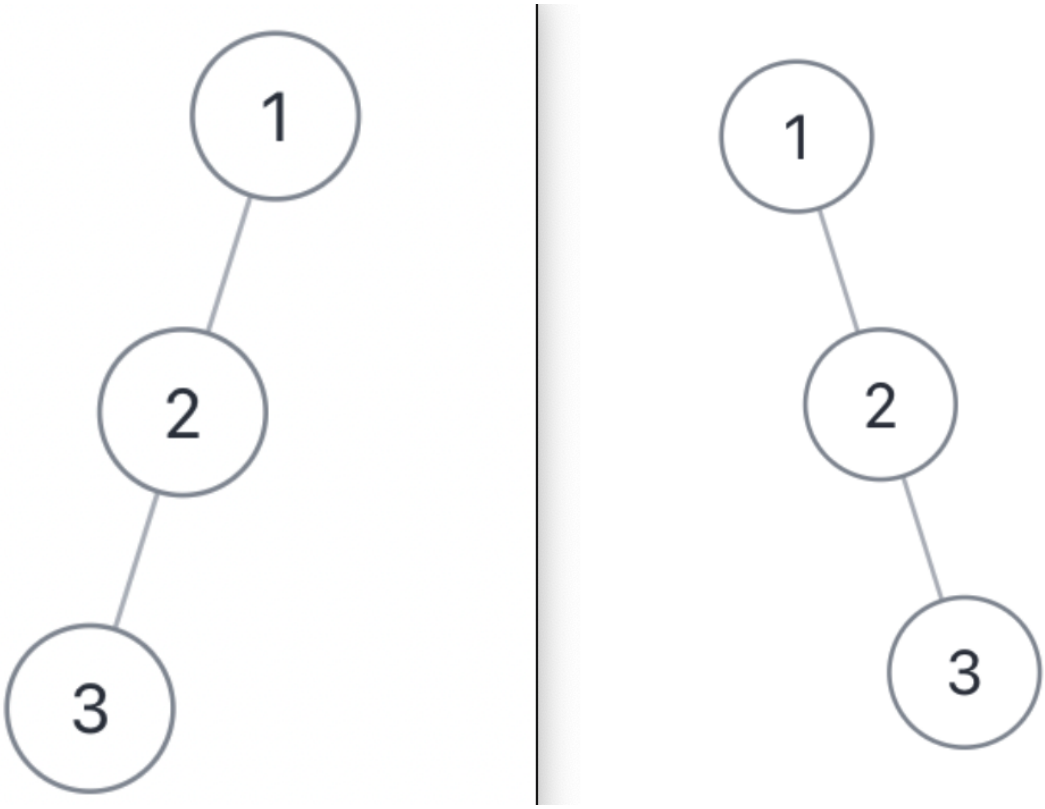
通过前序中序，或者后序中序遍历结果可以确定一棵原始二叉树，但是通过前序后序遍历结果无法确定原始二叉树。题目也说了，如果有多种结果，你可以返回任意一种。

为什么呢？我们说过，构建二叉树的套路很简单，先找到根节点，然后找到并递归构造左右子树即可。

前两道题，可以通过前序或者后序遍历结果找到根节点，然后根据中序遍历结果确定左右子树。

这道题，你可以确定根节点，但是无法确切的知道左右子树有哪些节点。

举个例子，下面这两棵树结构不同，但是它们的前序遍历和后序遍历结果是相同的：

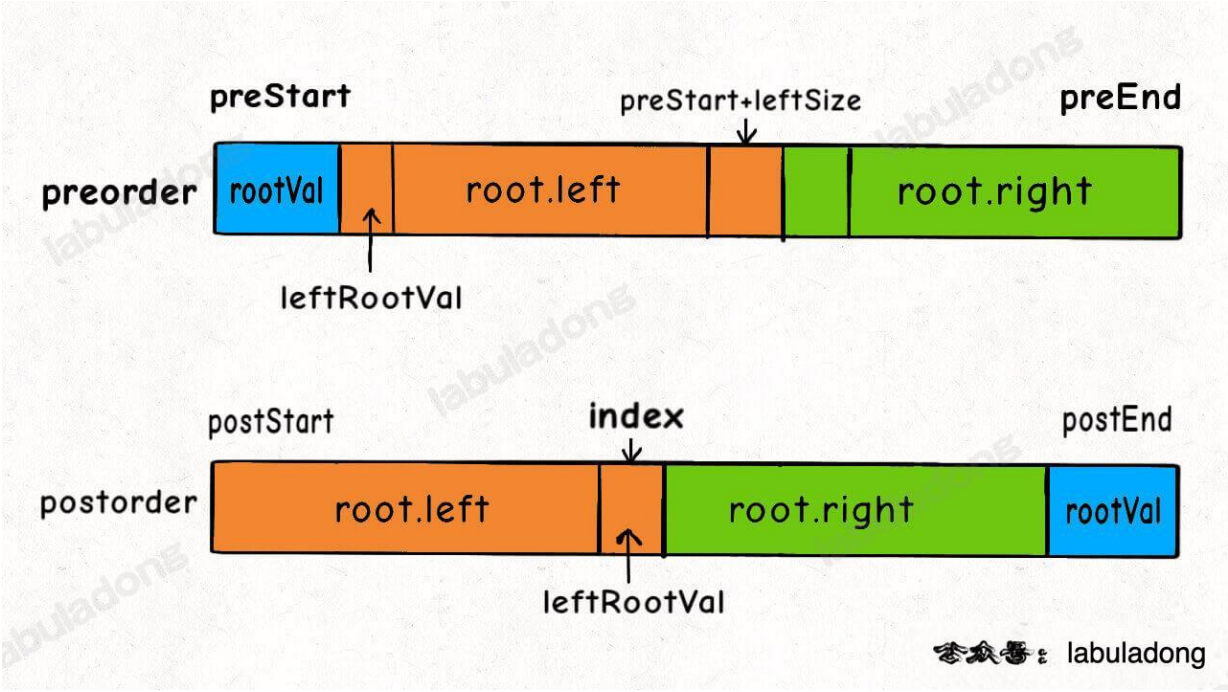


不过话说回来，用后序遍历和前序遍历结果还原二叉树，解法逻辑上和前两道题差别不大，也是通过控制左右子树的索引来构建：

1、首先把前序遍历结果的第一个元素或者后序遍历结果的最后一个元素确定为根节点的值。

2、然后把前序遍历结果的第二个元素作为左子树的根节点的值。

3、在后序遍历结果中寻找左子树根节点的值，从而确定了左子树的索引边界，进而确定右子树的索引边界，递归构造左右子树即可。



- **详细题解：东哥带你刷二叉树（构造篇）**

解法代码

```
class Solution {
    // 存储 postorder 中值到索引的映射
    HashMap<Integer, Integer> valToIndex = new HashMap<>();

    public TreeNode constructFromPrePost(int[] preorder, int[] postorder)
{
        for (int i = 0; i < postorder.length; i++) {
            valToIndex.put(postorder[i], i);
        }
        return build(preorder, 0, preorder.length - 1,
                     postorder, 0, postorder.length - 1);
    }

    // 定义：根据 preorder[preStart..preEnd] 和 postorder[postStart..postEnd]
    // 构建二叉树，并返回根节点。
    TreeNode build(int[] preorder, int preStart, int preEnd,
                   int[] postorder, int postStart, int postEnd) {
        if (preStart > preEnd) {
            return null;
        }
        if (preStart == preEnd) {
            return new TreeNode(preorder[preStart]);
        }

        // root 节点对应的值就是前序遍历数组的第一个元素
        int rootVal = preorder[preStart];
        // root.left 的值是前序遍历第二个元素
        // 通过前序和后序遍历构造二叉树的关键在于通过左子树的根节点
        // 确定 preorder 和 postorder 中左右子树的元素区间
        int leftRootVal = preorder[preStart + 1];
        // leftRootVal 在后序遍历数组中的索引
        int index = valToIndex.get(leftRootVal);
        // 左子树的元素个数
        int leftSize = index - postStart + 1;

        // 先构造出当前根节点
        TreeNode root = new TreeNode(rootVal);
        // 递归构造左右子树
        // 根据左子树的根节点索引和元素个数推导左右子树的索引边界
        root.left = build(preorder, preStart + 1, preStart + leftSize,
                postorder, postStart, index);
        root.right = build(preorder, preStart + leftSize + 1, preEnd,
                postorder, index + 1, postEnd - 1);

        return root;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 105. 从前序与中序遍历序列构造二叉树 🟠
  - 106. 从中序与后序遍历序列构造二叉树 🟠
  - 654. 最大二叉树 🟠
  - 剑指 Offer 07. 重建二叉树 🟠

剑指 Offer 07. 重建二叉树

这道题和 105. 从前序与中序遍历序列构造二叉树 相同。

111. 二叉树的最小深度

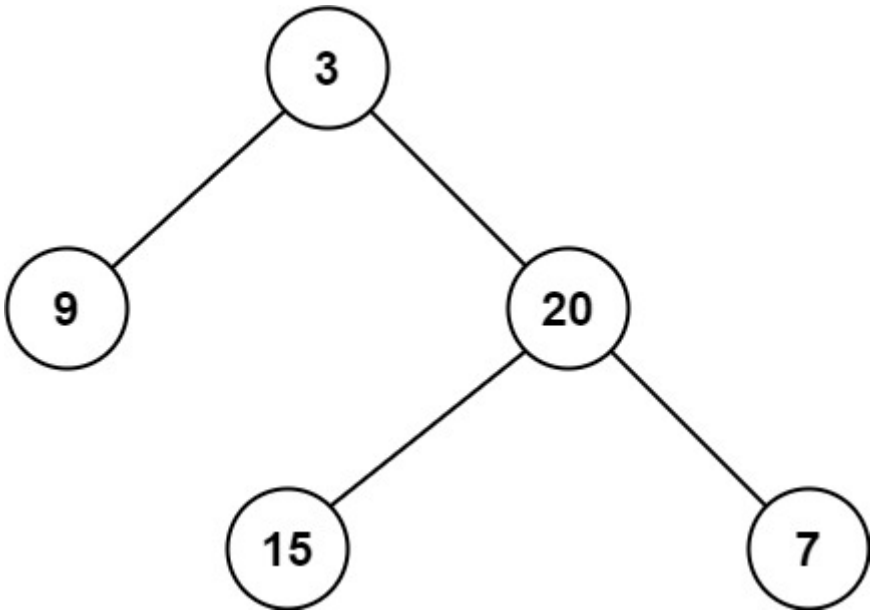
LeetCode	力扣	难度
111. Minimum Depth of Binary Tree	111. 二叉树的最小深度	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个二叉树，找出其最小深度，最小深度是从根节点到最近叶子节点（没有子节点的节点）的最短路径上的节点数量。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：2
```

基本思路

> 本文有视频版：BFS 算法核心框架套路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题可以同时运用两种思维，而且还可以用 BFS 层序遍历的方式求解，我会给出三种解法。

希望大家借助这道题仔细理解二叉树和 BFS 算法、动态规划算法、回溯算法 的联系。

- 详细题解：BFS 算法解题套路框架

解法代码

```java
// 「迭代」的递归思路
class Solution {
    private int minDepth = Integer.MAX_VALUE;
    private int currentDepth = 0;

    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        traverse(root);
        return minDepth;
    }

    private void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        // 做选择：在进入节点时增加当前深度
        currentDepth++;

        // 如果当前节点是叶子节点，更新最小深度
        if (root.left == null && root.right == null) {
            minDepth = Math.min(minDepth, currentDepth);
        }

        traverse(root.left);
        traverse(root.right);

        // 撤销选择：在离开节点时减少当前深度
        currentDepth--;
    }
}

// 「分解问题」的递归思路
class Solution2 {
    public int minDepth(TreeNode root) {
        // 基本情况：如果节点为空，返回深度为0
        if (root == null) {
            return 0;
        }

        // 递归计算左子树的最小深度
        int leftDepth = minDepth(root.left);
        // 递归计算右子树的最小深度
        int rightDepth = minDepth(root.right);

        // 特殊情况处理：如果左子树为空，返回右子树的深度加1
        if (leftDepth == 0) {
            return rightDepth + 1;
        }
        // 特殊情况处理：如果右子树为空，返回左子树的深度加1
        if (rightDepth == 0) {
```

```
            return leftDepth + 1;
        }

        // 计算并返回最小深度：左右子树深度的最小值加1
        return Math.min(leftDepth, rightDepth) + 1;
    }
}


// BFS 的思路
class Solution3 {
    public int minDepth(TreeNode root) {
        if (root == null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // root 本身就是一层，depth 初始化为 1
        int depth = 1;

        while (!q.isEmpty()) {
            int sz = q.size();
            /* 遍历当前层的节点 */
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                /* 判断是否到达叶子结点 */
                if (cur.left == null && cur.right == null)
                    return depth;
                /* 将下一层节点加入队列 */
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            /* 这里增加步数 */
            depth++;
        }
        return depth;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- 类似题目：
  - 542. 01 矩阵 🟠
  - 752. 打开转盘锁 🟠
  - 剑指 Offer II 109. 开密码锁 🟠

542. 01 矩阵

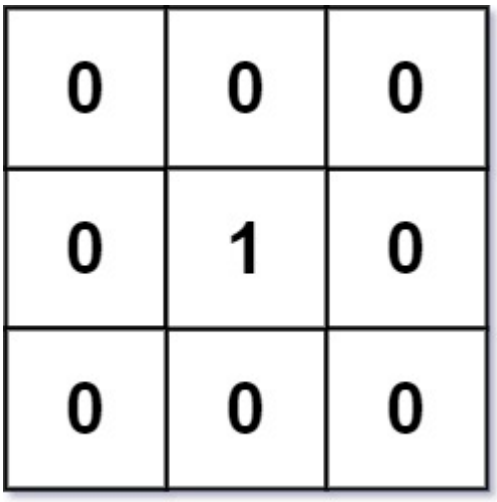
LeetCode	力扣	难度
542. 01 Matrix	542. 01 矩阵	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：**BFS 算法**

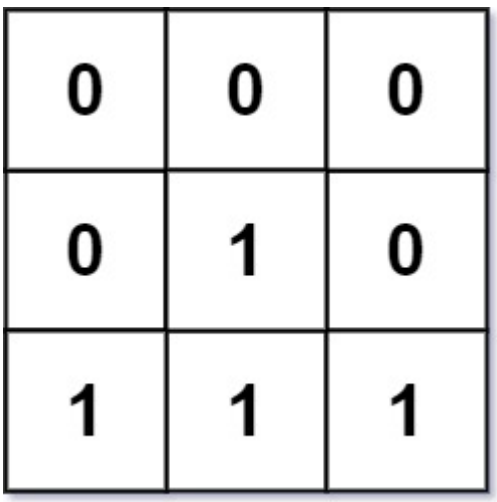
给定一个由 `0` 和 `1` 组成的矩阵 `mat`，请输出一个大小相同的矩阵，其中每一个格子是 `mat` 中对应位置元素到最近的 `0` 的距离。两个相邻元素间的距离为 `1`。

示例 1：




```
输入：mat = [[0,0,0],[0,1,0],[0,0,0]]
输出：[[0,0,0],[0,1,0],[0,0,0]]
```

示例 2：



```
输入：mat = [[0,0,0],[0,1,0],[1,1,1]]
输出：[[0,0,0],[0,1,0],[1,2,1]]
```

基本思路

我在第一眼看到这个题目时有两个思路方向：一是 动态规划，二是 BFS 算法。

何以见得可以用动态规划思路解决呢？

首先这是个求最值的问题，而且我感觉 `mat[x][y]` 离 0 最近的距离肯定能根据它周围元素 `mat[x+1][y], mat[x][y+1], mat[x-1][y], mat[x][y-1]` 离 0 最近的距离推算出来。这就说明可以比较容易地得到状态转移方程，那么写出动态规划解法也就没什么困难了。

何以见得可以用 BFS 算法思路解决呢？

因为这题描述的场景让我想到 111. 二叉树的最小深度 这类在二叉树中计算最小距离的题目，以及像 1091. 二进制矩阵中的最短路径 这种计算二维矩阵从左上角到右下角最短距离的题目。

再者，如果我把二维矩阵视为一幅「图」，那么最短距离肯定可以用图论中的最短距离算法计算出来。我在 Dijkstra 最短路径算法 中就讲到了，Dijkstra 算法就是 BFS 算法在加权图中的升级版。所以这题肯定能用普通 BFS 算法解决。

我这里就使用 BFS 算法来求解吧，使用 BFS 算法模板 中给出的框架改写一下就出来了，解法代码供参考。

解法代码

```
class Solution {
    public int[][] updateMatrix(int[][] mat) {
        int m = mat.length, n = mat[0].length;
        // 记录答案的结果数组
        int[][] res = new int[m][n];
        // 初始化全部填充特殊值 -1，代表未计算，
        // 待会可以用来判断坐标是否已经计算过，避免重复遍历
        for (int[] row : res) {
            Arrays.fill(row, -1);
        }

        Queue<int[]> q = new LinkedList<>();
        // 初始化队列，把那些值为 0 的坐标放到队列里
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (mat[i][j] == 0) {
                    q.offer(new int[]{i, j});
                    res[i][j] = 0;
                }
            }
        }
        // 执行 BFS 算法框架，从值为 0 的坐标开始向四周扩散
        int[][] dirs = new int[][]{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
```

```
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int x = cur[0], y = cur[1];
            // 向四周扩散
            for (int i = 0; i < 4; i++) {
                int nextX = x + dirs[i][0];
                int nextY = y + dirs[i][1];
                // 确保相邻的这个坐标没有越界且之前未被计算过
                if (nextX >= 0 && nextX < m && nextY >= 0 && nextY < n
                        && res[nextX][nextY] == -1) {
                    q.offer(new int[]{nextX, nextY});
                    // 从 mat[x][y] 走到 mat[nextX][nextY] 需要一步
                    res[nextX][nextY] = res[x][y] + 1;
                }
            }
        }

        return res;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

114. 二叉树展开为链表

LeetCode	力扣	难度
114. Flatten Binary Tree to Linked List	114. 二叉树展开为链表	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

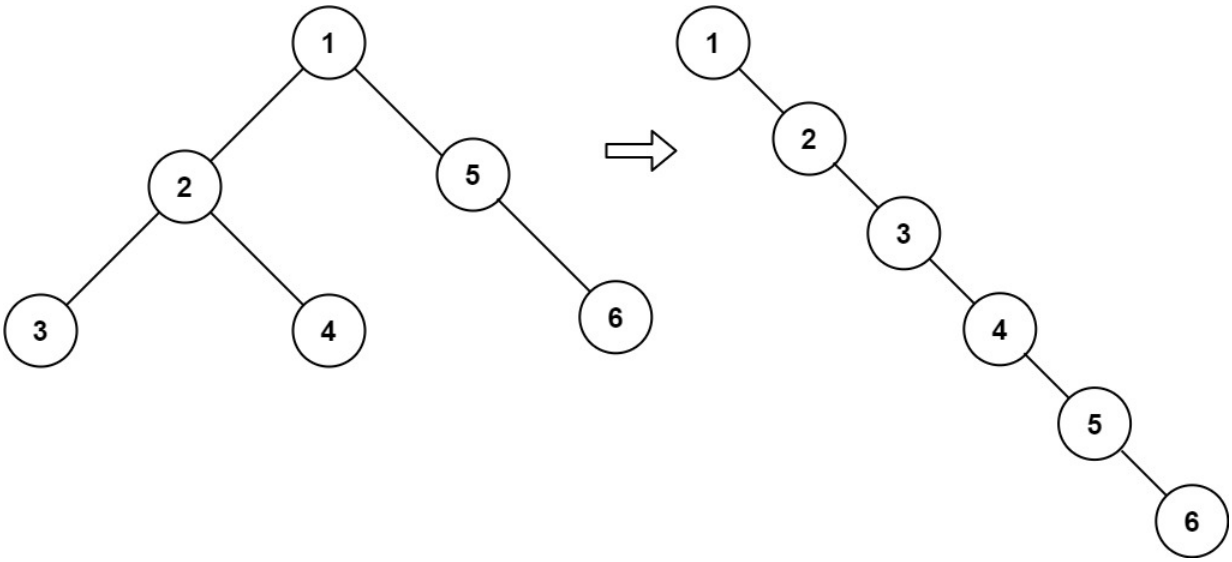
- 标签：二叉树，数据结构

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

1、展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。

2、展开后的单链表应该与二叉树 先序遍历 顺序相同。

示例 1：




```
输入：root = [1,2,5,3,4,null,6]
输出：[1,null,2,null,3,null,4,null,5,null,6]
```

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维。

前者较简单，只要运用二叉树的递归遍历框架即可；后者的关键在于明确递归函数的定义，然后利用这个定义，这题就属于后者，`flatten` 函数的定义如下：

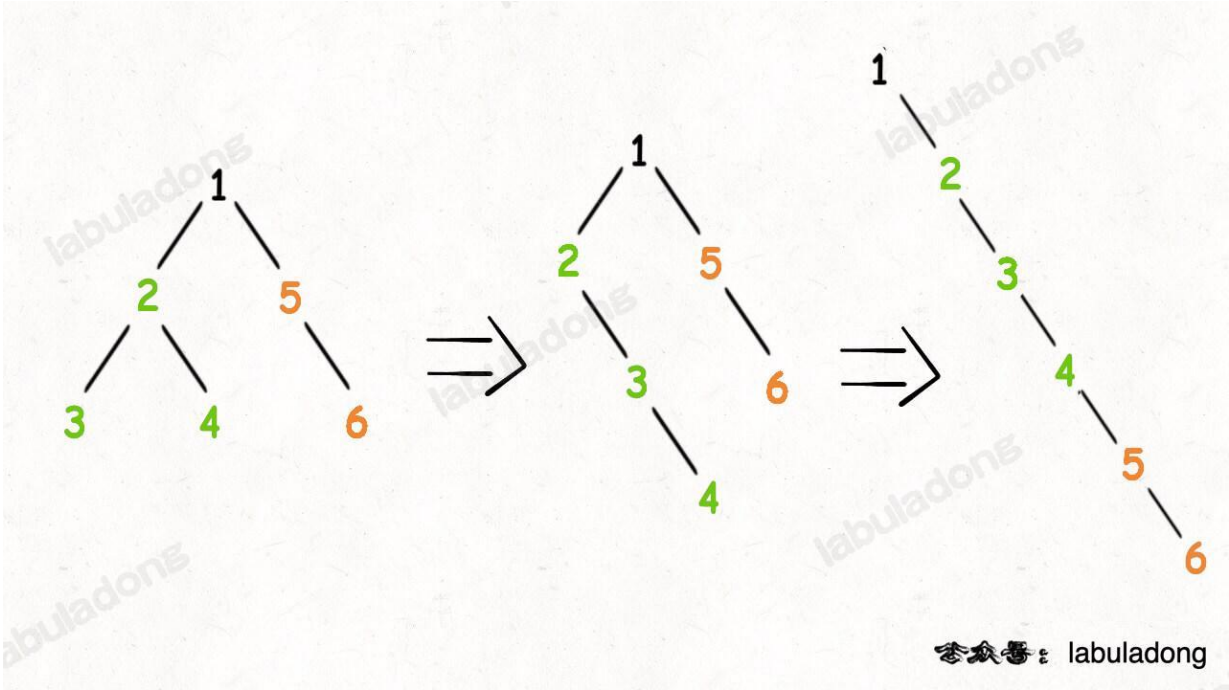
给 `flatten` 函数输入一个节点 `root`，那么以 `root` 为根的二叉树就会被拉平为一条链表。

如何利用这个定义来完成算法？你想想怎么把以 root 为根的二叉树拉平为一条链表？

很简单，以下流程：

1、将 root 的左子树和右子树拉平。

2、将 root 的右子树接到左子树下方，然后将整个左子树作为右子树。



至于如何把 root 的左右子树拉平，不用你操心，flatten 函数的定义就是这样，交给他做就行了。

把上面的逻辑翻译成代码，即可解决本题。

- **详细题解：东哥带你刷二叉树（思路篇）**

解法代码

```
class Solution {
    // 定义：将以 root 为根的树拉平为链表
    public void flatten(TreeNode root) {
        // base case
        if (root == null) return;
        // 先递归拉平左右子树
        flatten(root.left);
        flatten(root.right);

        /****后序遍历位置****/
        // 1、左右子树已经被拉平成一条链表
        TreeNode left = root.left;
        TreeNode right = root.right;

        // 2、将左子树作为右子树
        root.left = null;
        root.right = left;
```

```
            // 3、将原先的右子树接到当前右子树的末端
            TreeNode p = root;
            while (p.right != null) {
                p = p.right;
            }
            p.right = right;
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 116. 填充每个节点的下一个右侧节点指针 🟠
  - 226. 翻转二叉树 🟢
  - 897. 递增顺序搜索树 🟢
  - 剑指 Offer 27. 二叉树的镜像 🟢
  - 剑指 Offer II 052. 展平二叉搜索树 🟢

116. 填充每个节点的下一个右侧节点指针

LeetCode	力扣	难度
116. Populating Next Right Pointers in Each Node	116. 填充每个节点的下一个右侧节点指针	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构

给定一个**完美二叉树**，其所有叶子节点都在同一层，每个父节点都有两个子节点。二叉树定义如下：

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。初始状态下，所有 next 指针都被设置为 NULL。

示例：

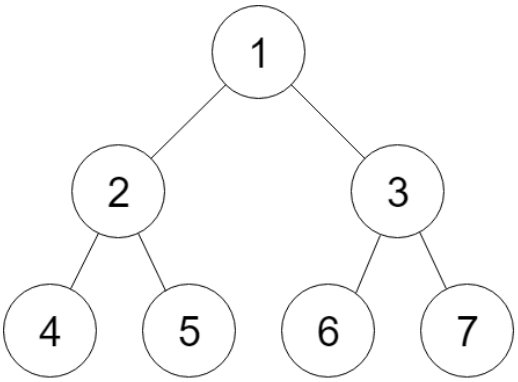


Figure A

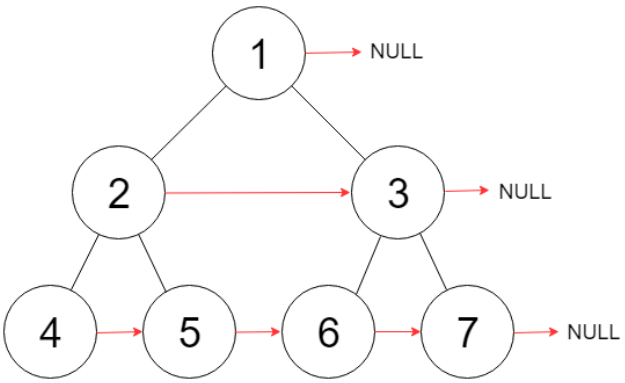


Figure B

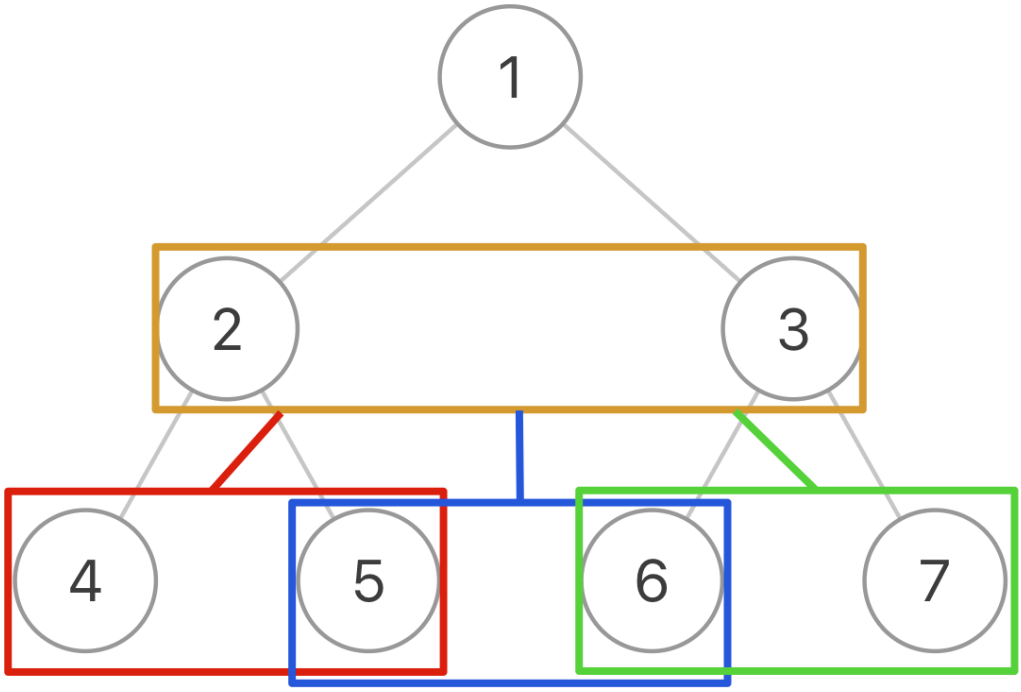
```
输入：root = [1,2,3,4,5,6,7]
输出：[1,#,2,3,#,4,5,6,7,#]
解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化的输出按层序遍历排列，同一层节点由 next 指针连接，'#' 标志着每一层的结束。
```

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

你可以把二叉树的相邻节点抽象成一个「三叉树节点」，这样二叉树就变成了一棵「三叉树」，然后你去遍历这棵三叉树，把每个「三叉树节点」中的两个节点连接就行了：



- **详细题解：东哥带你刷二叉树（思路篇）**

解法代码

```
class Solution {
    // 主函数
    public Node connect(Node root) {
        if (root == null) return null;
        // 遍历「三叉树」，连接相邻节点
        traverse(root.left, root.right);
        return root;
    }

    // 三叉树遍历框架
    void traverse(Node node1, Node node2) {
        if (node1 == null || node2 == null) {
            return;
        }
        /**** 前序位置 ****/
        // 将传入的两个节点穿起来
        node1.next = node2;

        // 连接相同父节点的两个子节点
        traverse(node1.left, node1.right);
```

```
            traverse(node2.left, node2.right);
            // 连接跨越父节点的两个子节点
            traverse(node1.right, node2.left);
        }
    }
```

- **类似题目**：
  - 114. 二叉树展开为链表 🟠
  - 117. 填充每个节点的下一个右侧节点指针 II 🟠
  - 226. 翻转二叉树 🟢
  - 剑指 Offer 27. 二叉树的镜像 🟢

226. 翻转二叉树

LeetCode	力扣	难度
226. Invert Binary Tree	226. 翻转二叉树	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，数据结构**

翻转一棵二叉树。

示例：

输入：

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

输出：

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题可以同时使用两种思维模式。

如何翻转二叉树？其实就是把二叉树上的每个节点的左右子节点都交换一下，我同时给出两种思维模式下的解法供你对比。

- **详细题解：东哥带你刷二叉树（思路篇）**

解法代码

```java
// 「遍历」的思路
class Solution {
    // 主函数
    public TreeNode invertTree(TreeNode root) {
        // 遍历二叉树，交换每个节点的子节点
        traverse(root);
        return root;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        /**** 前序位置 ****/
        // 每一个节点需要做的事就是交换它的左右子节点
        TreeNode tmp = root.left;
        root.left = root.right;
        root.right = tmp;

        // 遍历框架，去遍历左右子树的节点
        traverse(root.left);
        traverse(root.right);
    }
}

// 「分解问题」的思路
class Solution2 {
    // 定义：将以 root 为根的这棵二叉树翻转，返回翻转后的二叉树的根节点
    TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }
        // 利用函数定义，先翻转左右子树
        TreeNode left = invertTree(root.left);
        TreeNode right = invertTree(root.right);

        // 然后交换左右子节点
        root.left = right;
        root.right = left;

        // 和定义逻辑自恰：以 root 为根的这棵二叉树已经被翻转，返回 root
        return root;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：

- 114. 二叉树展开为链表 🟠
- 116. 填充每个节点的下一个右侧节点指针 🟠
- 剑指 Offer 27. 二叉树的镜像 🟢

897. 递增顺序搜索树

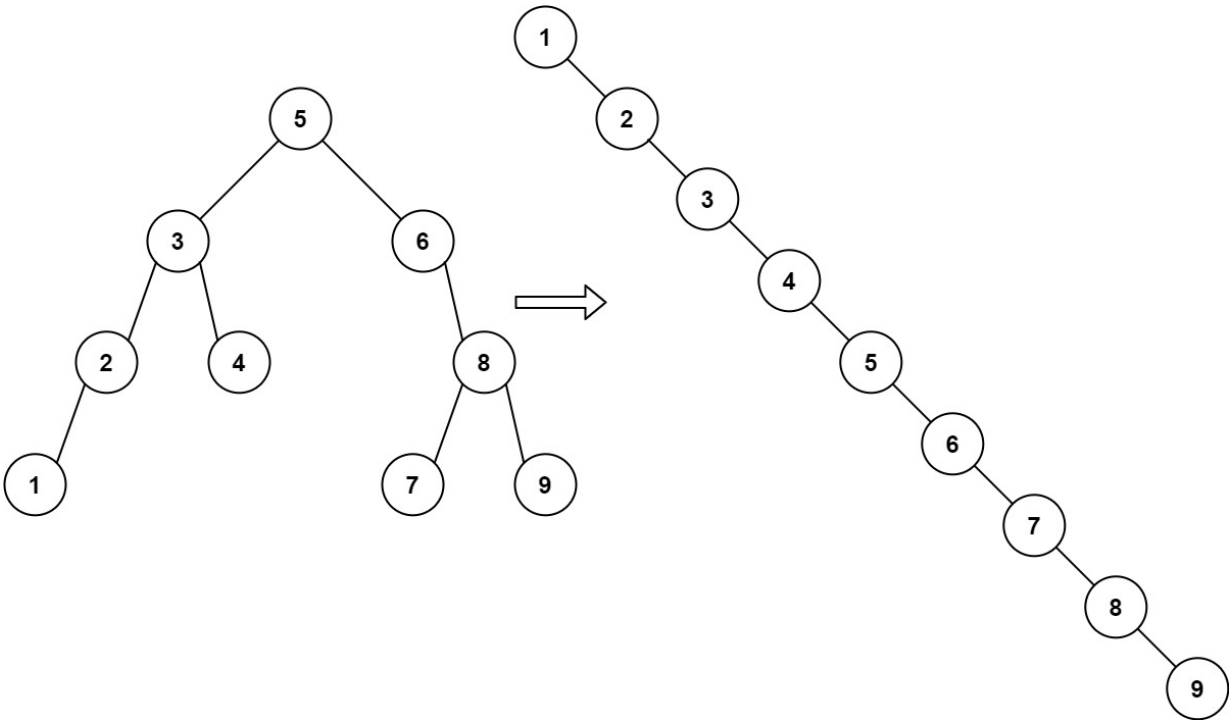
LeetCode	力扣	难度
897. Increasing Order Search Tree	897. 递增顺序搜索树	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树vip

给你一棵二叉搜索树，请你**按中序遍历**将其重新排列为一棵递增顺序搜索树，使树中最左边的节点成为树的根节点，并且每个节点没有左子节点，只有一个右子节点。

示例 1：




```
输入：root = [5,3,6,2,4,null,8,1,null,null,null,7,9]
输出：[1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题可以同时用到两种思维模式。

「遍历」的话很简单，你对 BST 做中序遍历，其结果就是有序的，重新构造出题目要求的这个类似链表的二叉树即可。

「分解问题」的思路也不难，你只要做过 114. 二叉树展开为链表 这道题，稍微改下解法就可以解决这道题了，明确 `increasingBST` 的定义，然后利用这个定义进行操作即可。

这里我给出分解问题的解法思路。

解法代码

```
class Solution {
    // 输入一棵 BST，返回一个有序「链表」
    public TreeNode increasingBST(TreeNode root) {
        if (root == null) {
            return null;
        }
        // 先把左右子树拉平
        TreeNode left = increasingBST(root.left);
        root.left = null;
        TreeNode right = increasingBST(root.right);
        root.right = right;
        // 左子树为空的话，就不用处理了
        if (left == null) {
            return root;
        }
        // 左子树非空，需要把根节点和右子树接到左子树末尾
        TreeNode p = left;
        while (p != null && p.right != null) {
            p = p.right;
        }
        p.right = root;

        return left;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 剑指 Offer II 052. 展平二叉搜索树 🟢

剑指 Offer 27. 二叉树的镜像

这道题和 226. 翻转二叉树 相同。

剑指 Offer II 052. 展平二叉搜索树

这道题和 897. 递增顺序搜索树 相同。

117. 填充每个节点的下一个右侧节点指针 II

LeetCode	力扣	难度
117. Populating Next Right Pointers in Each Node II	117. 填充每个节点的下一个右侧节点指针 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

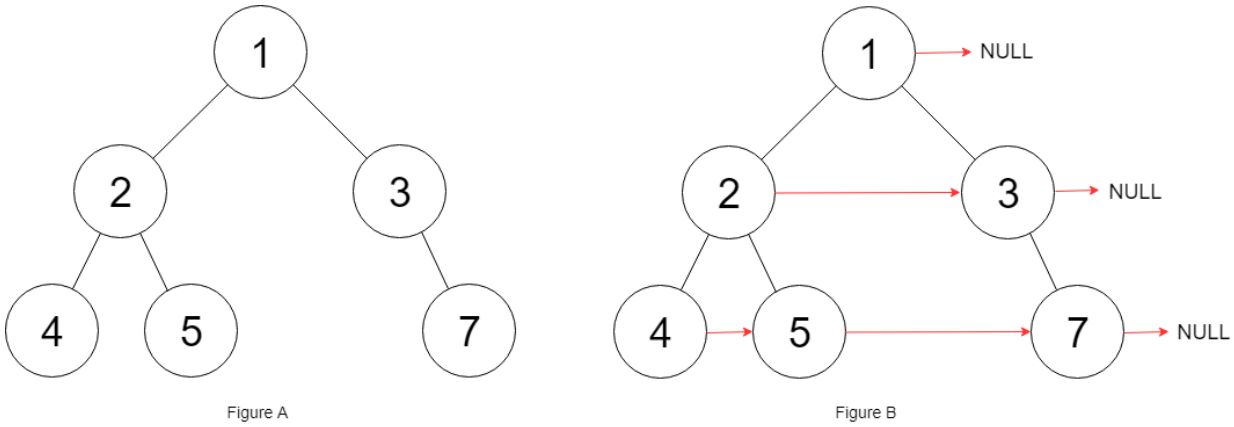
- 标签：BFS 算法，二叉树，二叉树vip

给定一个二叉树

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 NULL。初始状态下，所有 next 指针都被设置为 NULL。

示例：




```
输入：root = [1,2,3,4,5,null,7]
输出：[1,#,2,3,#,4,5,7,#]
解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化输出按层序遍历顺序（由 next 指针连接），'#' 表示每层的末尾。
```

基本思路

前文 我的算法学习经验 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式。

但这题和 116. 填充每个节点的下一个右侧节点指针 还不一样，输入的不是完全二叉树，所以不好直接用递归。

这题用 BFS 算法 进行层序遍历比较直观，在 for 循环，无非就是想办法遍历所有节点，然后把这个节点和相邻节点连起来罢了。

当然，还有效率更高的方式，就是直接操作指针，不过略有些难懂，暂时不写。

解法代码

```
class Solution {
    public Node connect(Node root) {
        if (root == null) {
            return null;
        }
        // 二叉树层序遍历框架
        Queue<Node> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            int sz = q.size();
            // 遍历一层
            Node pre = null;
            for (int i = 0; i < sz; i++) {
                Node cur = q.poll();
                // 链接当前层所有节点的 next 指针
                if (pre != null) {
                    pre.next = cur;
                }
                pre = cur;
                // 将下一层节点装入队列
                if (cur.left != null) {
                    q.offer(cur.left);
                }
                if (cur.right != null) {
                    q.offer(cur.right);
                }
            }
        }
        return root;
    }
}
```

145. 二叉树的后序遍历

LeetCode	力扣	难度
145. Binary Tree Postorder Traversal	145. 二叉树的后序遍历	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树**

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]
```

基本思路

不要瞧不起二叉树的前中后序遍历。

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，分别代表回溯算法和动态规划的底层思想。

本题用两种思维模式来解答，注意体会其中思维方式的差异。

解法代码

```
class Solution {
    /* 动态规划思路 */
    // 定义：输入一个节点，返回以该节点为根的二叉树的后序遍历结果
    public List<Integer> postorderTraversal(TreeNode root) {
        LinkedList<Integer> res = new LinkedList<>();
        if (root == null) {
            return res;
        }
        // 后序遍历结果特点：先是左子树，接着是右子树，最后是根节点的值
        res.addAll(postorderTraversal(root.left));
        res.addAll(postorderTraversal(root.right));
        res.add(root.val);
        return res;
```

```
    }

    /* 回溯算法思路 */
    LinkedList<Integer> res = new LinkedList<>();

    // 返回后序遍历结果
    public List<Integer> postorderTraversal2(TreeNode root) {
        traverse(root);
        return res;
    }

    // 二叉树遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        traverse(root.left);
        traverse(root.right);
        // 后序遍历位置
        res.add(root.val);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

222. 完全二叉树的节点个数

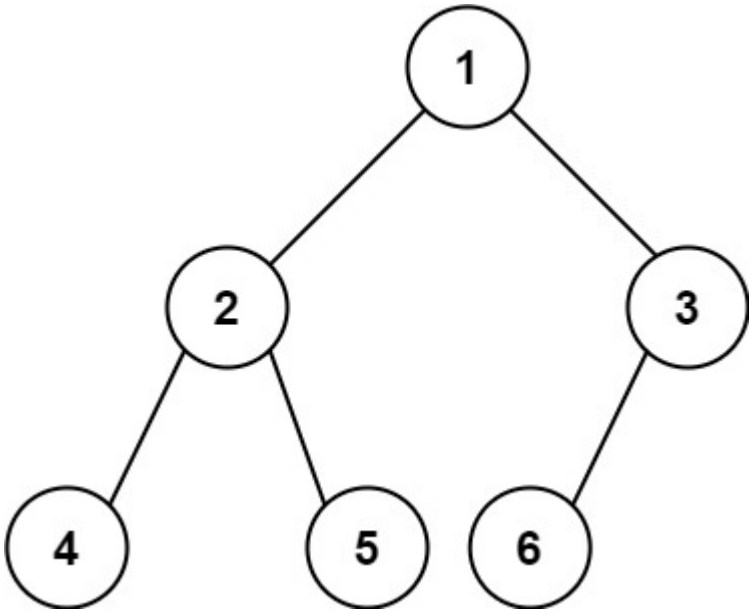
LeetCode	力扣	难度
222. Count Complete Tree Nodes	222. 完全二叉树的节点个数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构

给你一棵**完全二叉树**的根节点 `root`，求出该树的节点个数（完全二叉树的定义）。

示例 1：

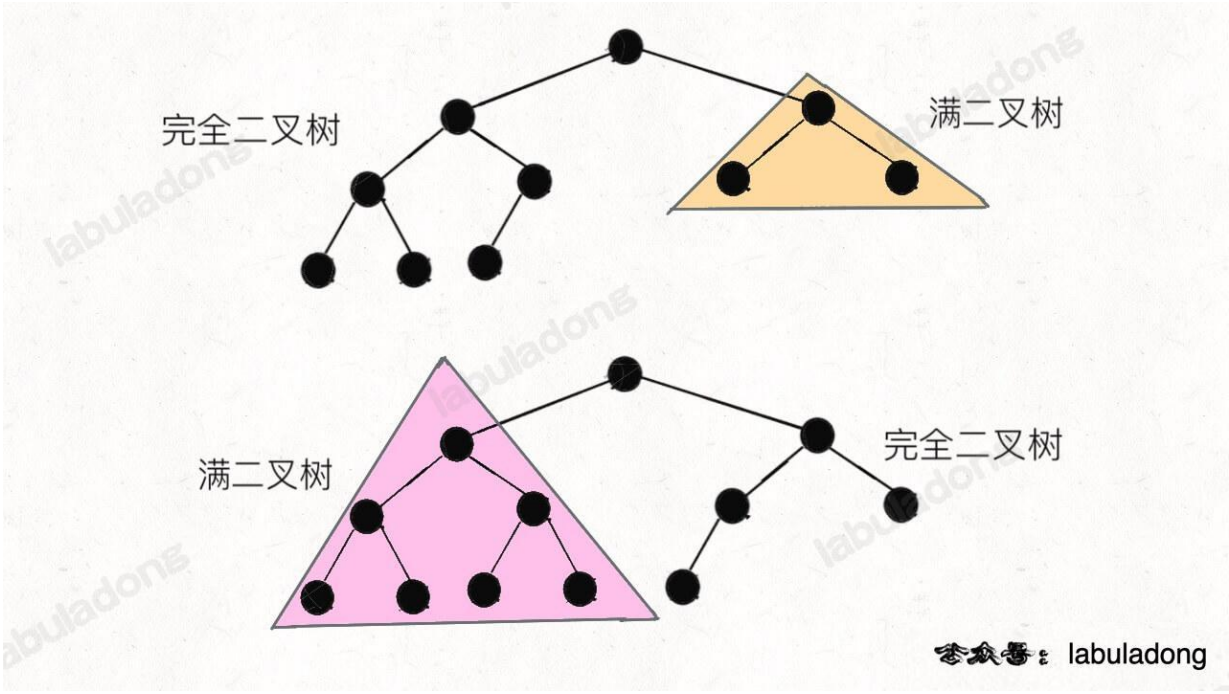



```
输入：root = [1,2,3,4,5,6]
输出：6
```

基本思路

PS：这道题在《算法小抄》 的第 243 页。

一棵完全二叉树的两棵子树，至少有一棵是满二叉树：



计算满二叉树的节点个数不用一个个节点去数，可以直接通过树高算出来，这也是这道题提高效率的关键点。

- **详细题解：如何计算完全二叉树的节点数**

解法代码

```
class Solution {
    public int countNodes(TreeNode root) {
        TreeNode l = root, r = root;
        // 记录左、右子树的高度
        int hl = 0, hr = 0;
        while (l != null) {
            l = l.left;
            hl++;
        }
        while (r != null) {
            r = r.right;
            hr++;
        }
        // 如果左右子树的高度相同，则是一棵满二叉树
        if (hl == hr) {
            return (int) Math.pow(2, hl) - 1;
        }
        // 如果左右高度不同，则按照普通二叉树的逻辑计算
        return 1 + countNodes(root.left) + countNodes(root.right);
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

1644. 二叉树的最近公共祖先 II

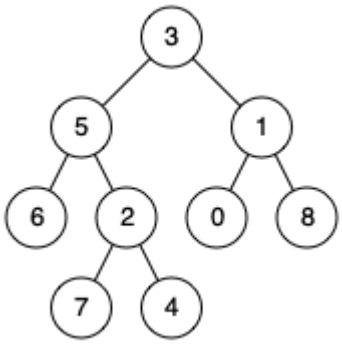
LeetCode	力扣	难度
1644. Lowest Common Ancestor of a Binary Tree II	1644. 二叉树的最近公共祖先 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给定一棵二叉树的根节点 `root`，树中的每个节点值都是互不相同的，返回给定节点 `p` 和 `q` 的最近公共祖先（LCA）节点。如果 `p` 或 `q` 之一**不存在**于该二叉树中，返回 `null`。

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 1 的共同祖先节点是 3。
```

基本思路

这题只要把 235. 二叉搜索树的最近公共祖先 的解法稍微改一下就行了。

235. 二叉搜索树的最近公共祖先 说 `p` 和 `q` 必然存在二叉树中，而这道题中 `p, q` 可能不存在，所以需要遍历整棵二叉树才能判断公共祖先是否存在。

所以可以用变量 `foundP` 和 `foundQ` 记录 `p` 和 `q` 是否存在。

- **详细题解：Git原理之最近公共祖先**

解法代码

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        TreeNode res = LCA(root, p, q);
```

```
        if (foundP && foundQ) {
            return res;
        }
        return null;
    }

    boolean foundP = false, foundQ = false;

    // 定义：输入一棵二叉树，返回这棵二叉树中 `p` 和 `q` 的最近公共祖先。
    TreeNode LCA(TreeNode root, TreeNode p, TreeNode q) {
        // base case
        if (root == null) return null;

        TreeNode left = LCA(root.left, p, q);
        TreeNode right = LCA(root.right, p, q);

        if (root == p || root == q) {
            if (root == p) foundP = true;
            if (root == q) foundQ = true;
            return root;
        }
        // 情况 1
        if (left != null && right != null) {
            return root;
        }
        // 情况 2
        if (left == null && right == null) {
            return null;
        }
        // 情况 3
        return left == null ? right : left;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1650. 二叉树的最近公共祖先 III 🟠
  - 1676. 二叉树的最近公共祖先 IV 🟠
  - 235. 二叉搜索树的最近公共祖先 🟠
  - 236. 二叉树的最近公共祖先 🟠
  - 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先 🟢
  - 剑指 Offer 68 - II. 二叉树的最近公共祖先 🟢

1676. 二叉树的最近公共祖先 IV

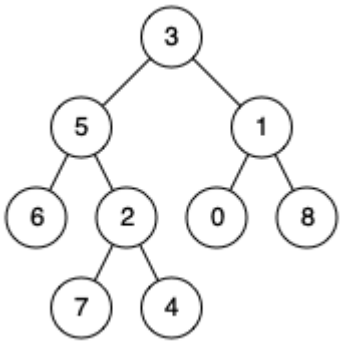
LeetCode	力扣	难度
1676. Lowest Common Ancestor of a Binary Tree IV	1676. 二叉树的最近公共祖先 IV	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给定一棵二叉树的根节点 `root` 和 `TreeNode` 类对象的数组 `nodes`，返回 `nodes` 中所有节点的最近公共祖先（LCA）。数组中所有节点都存在于该二叉树中，且二叉树中所有节点的值都是互不相同的。

示例 1:




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], nodes = [4,7]
输出：2
解释: 节点 4 和 7 的最近公共祖先是 2。
```

基本思路

236. 二叉树的最近公共祖先 让你算两个节点的最近公共祖先，现在让你算多个节点的最近公共祖先。把这些节点装到 `HashSet` 里面方便迅速判断就行了，其他的逻辑和上一道题一模一样。

- **详细题解：Git原理之最近公共祖先**

解法代码

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode[] nodes)
{
        HashSet<Integer> set = new HashSet<>();
        // 转化到集合中
        for (TreeNode node : nodes) {
            set.add(node.val);
        }
```

```
        return LCA(root, set);
    }

    private TreeNode LCA(TreeNode root, HashSet<Integer> set) {
        // base case
        if (root == null) return null;
        if (set.contains(root.val)) return root;

        TreeNode left = LCA(root.left, set);
        TreeNode right = LCA(root.right, set);
        // 情况 1
        if (left != null && right != null) {
            return root;
        }
        // 情况 2
        if (left == null && right == null) {
            return null;
        }
        // 情况 3
        return left == null ? right : left;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 1644. 二叉树的最近公共祖先 II 🟠
  - 1650. 二叉树的最近公共祖先 III 🟠
  - 235. 二叉搜索树的最近公共祖先 🟠
  - 236. 二叉树的最近公共祖先 🟠
  - 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先 🟢
  - 剑指 Offer 68 - II. 二叉树的最近公共祖先 🟢

235. 二叉搜索树的最近公共祖先

LeetCode	力扣	难度
235. Lowest Common Ancestor of a Binary Search Tree	235. 二叉搜索树的最近公共祖先	

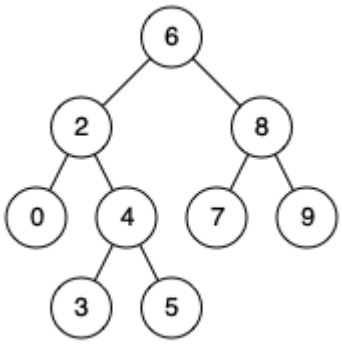
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给定一个二叉搜索树，找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：对于有根树的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。

示例 1:




```
输入：root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出：6
解释：节点 2 和节点 8 的最近公共祖先是 6。
```

基本思路

比较经典的是 236. 二叉树的最近公共祖先，我在 这篇文章 讲了那道题的解法。

如果在 BST 中寻找最近公共祖先，反而容易很多，主要利用 BST 左小右大（左子树所有节点都比当前节点小，右子树所有节点都比当前节点大）的特点即可。

1、如果 p 和 q 都比当前节点小，那么显然 p 和 q 都在左子树，那么 LCA 在左子树。

2、如果 p 和 q 都比当前节点大，那么显然 p 和 q 都在右子树，那么 LCA 在右子树。

3、一旦发现 p 和 q 在当前节点的两侧，说明当前节点就是 LCA。

- **详细题解：Git原理之最近公共祖先**

解法代码

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        if (root == null) return null;
        if (p.val > q.val) {
            // 保证 p.val <= q.val，便于后续情况讨论
            return lowestCommonAncestor(root, q, p);
        }
        if (root.val >= p.val && root.val <= q.val) {
            // p <= root <= q
            // 即 p 和 q 分别在 root 的左右子树，那么 root 就是 LCA
            return root;
        }
        if (root.val > q.val) {
            // p 和 q 都在 root 的左子树，那么 LCA 在左子树
            return lowestCommonAncestor(root.left, p, q);
        } else {
            // p 和 q 都在 root 的右子树，那么 LCA 在右子树
            return lowestCommonAncestor(root.right, p, q);
        }
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1644. 二叉树的最近公共祖先 II 🟠
  - 1650. 二叉树的最近公共祖先 III 🟠
  - 1676. 二叉树的最近公共祖先 IV 🟠
  - 236. 二叉树的最近公共祖先 🟠
  - 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先 🟢
  - 剑指 Offer 68 - II. 二叉树的最近公共祖先 🟢

236. 二叉树的最近公共祖先

LeetCode	力扣	难度
236. Lowest Common Ancestor of a Binary Tree	236. 二叉树的最近公共祖先	

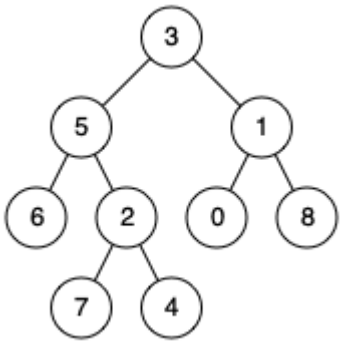
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树**

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：对于有根树的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。

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。
```

基本思路

经典问题了，先给出递归函数的定义：给该函数输入三个参数 `root`，`p`，`q`，它会返回一个节点：

情况 1，如果 `p` 和 `q` 都在以 `root` 为根的树中，函数返回的即使 `p` 和 `q` 的最近公共祖先节点。

情况 2，那如果 `p` 和 `q` 都不在以 `root` 为根的树中怎么办呢？函数理所当然地返回 `null` 呗。

情况 3，那如果 `p` 和 `q` 只有一个存在于 `root` 为根的树中呢？函数就会返回那个节点。

根据这个定义，分情况讨论：

情况 1，如果 `p` 和 `q` 都在以 `root` 为根的树中，那么 `left` 和 `right` 一定分别是 `p` 和 `q`（从 base case 看出来的）。

情况 2，如果 `p` 和 `q` 都不在以 `root` 为根的树中，直接返回 `null`。

情况 3，如果 p 和 q 只有一个存在于 root 为根的树中，函数返回该节点。

- **详细题解：Git原理之最近公共祖先**

解法代码

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        // base case
        if (root == null) return null;
        if (root == p || root == q) return root;

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        // 情况 1
        if (left != null && right != null) {
            return root;
        }
        // 情况 2
        if (left == null && right == null) {
            return null;
        }
        // 情况 3
        return left == null ? right : left;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目：**
  - 1644. 二叉树的最近公共祖先 II 🟠
  - 1650. 二叉树的最近公共祖先 III 🟠
  - 1676. 二叉树的最近公共祖先 IV 🟠
  - 235. 二叉搜索树的最近公共祖先 🟠
  - 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先 🟢
  - 剑指 Offer 68 - II. 二叉树的最近公共祖先 🟢

剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

这道题和 235. 二叉搜索树的最近公共祖先 相同。

剑指 Offer 68 - II. 二叉树的最近公共祖先

这道题和 236. 二叉树的最近公共祖先 相同。

297. 二叉树的序列化与反序列化

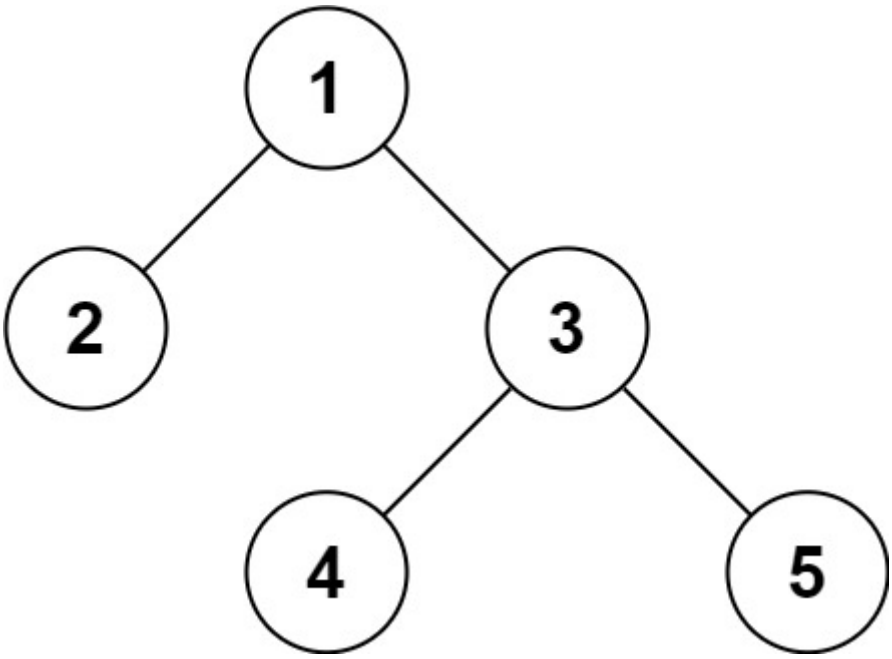
LeetCode	力扣	难度
297. Serialize and Deserialize Binary Tree	297. 二叉树的序列化与反序列化	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构，递归

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化/反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

示例 1：



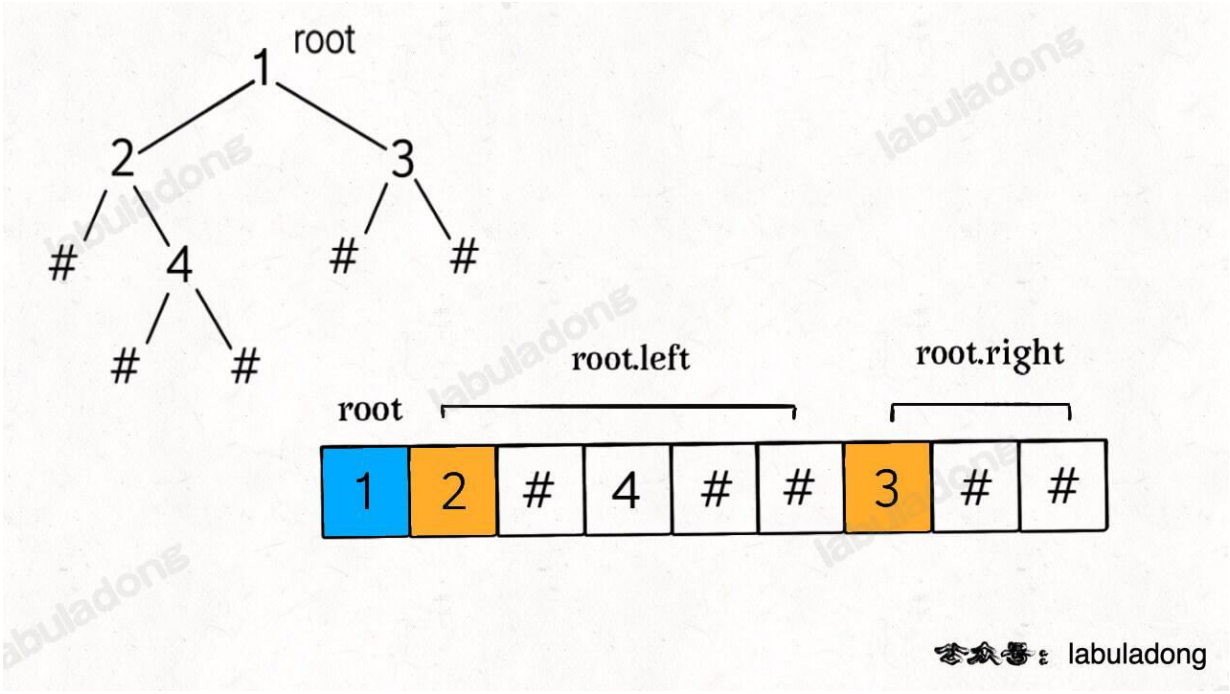

```
输入：root = [1,2,3,null,null,4,5]
输出：[1,2,3,null,null,4,5]
```

基本思路

PS：这道题在《算法小抄》 的第 247 页。

序列化问题其实就是遍历问题，你能遍历，顺手把遍历的结果转化成字符串的形式，不就是序列化了么？

这里我就简单说说用前序遍历的思路，前序遍历的特点是根节点在开头，然后接着左子树的前序遍历结果，然后接着右子树的前序遍历结果：



所以如果按照前序遍历顺序进行序列化，反序列化的时候，就知道第一个元素是根节点的值，然后递归调用反序列化左右子树，接到根节点上即可，上述思路翻译成代码即可解决本题。

当然，这题也可以尝试使用二叉树的中序、后序、层序的遍历方式来做，具体可看详细题解。

- **详细题解：东哥带你刷二叉树（序列化篇）**

解法代码

```
public class Codec {
    String SEP = ",";
    String NULL = "#";

    /* 主函数，将二叉树序列化为字符串 */
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serialize(root, sb);
        return sb.toString();
    }

    /* 辅助函数，将二叉树存入 StringBuilder */
    void serialize(TreeNode root, StringBuilder sb) {
        if (root == null) {
            sb.append(NULL).append(SEP);
            return;
        }

        /******前序遍历位置******/
        sb.append(root.val).append(SEP);
        /************************/

        serialize(root.left, sb);
        serialize(root.right, sb);
```

```
    }

    /* 主函数，将字符串反序列化为二叉树结构 */
    public TreeNode deserialize(String data) {
        // 将字符串转化成列表
        LinkedList<String> nodes = new LinkedList<>();
        for (String s : data.split(SEP)) {
            nodes.addLast(s);
        }
        return deserialize(nodes);
    }

    /* 辅助函数，通过 nodes 列表构造二叉树 */
    TreeNode deserialize(LinkedList<String> nodes) {
        if (nodes.isEmpty()) return null;

        /******前序遍历位置******/
        // 列表最左侧就是根节点
        String first = nodes.removeFirst();
        if (first.equals(NULL)) return null;
        TreeNode root = new TreeNode(Integer.parseInt(first));
        /**********************/

        root.left = deserialize(nodes);
        root.right = deserialize(nodes);

        return root;
    }
}
```

- **类似题目**：
  - 449. 序列化和反序列化二叉搜索树 🟠
  - 剑指 Offer 37. 序列化二叉树 🔴
  - 剑指 Offer II 048. 序列化与反序列化二叉树 🔴

449. 序列化和反序列化二叉搜索树

LeetCode	力扣	难度
449. Serialize and Deserialize BST	449. 序列化和反序列化二叉搜索树	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

设计一个算法来序列化和反序列化 **二叉搜索树**。对序列化/反序列化算法的工作方式没有限制，只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。**编码的字符串应尽可能紧凑。**

示例 1：

```
输入：root = [2,1,3]
输出：[2,1,3]
```

基本思路

在做本题之前，你需要先看前文 二叉树的序列化和反序列化的不同方式，然后就可以很容易理解本题的思路了。

二叉树的构造算法通用思路很简单，无非就是构造根节点，然后递归构造左右子树，最后把它们接起来，**关键在于如何找到根节点和左右子树的节点，不同的序列化方法，找根节点的方式也不同。**

本题当然可以直接复制 297. 二叉树的序列化和反序列化 的代码过来，但是显然没有利用到 BST 左小右大的特性，复杂度会更高。

对比 297 题普通二叉树的序列化，利用 BST 左小右大的特性主要可以避免序列化空指针，利用 `min, max` 边界来划定一棵子树的边界，从而提升算法效率。

解法代码

```
public class Codec {
    // 分隔符，区分每个节点的值
    private final static String SEP = ",";

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serialize(root, sb);
        return sb.toString();
    }

    private void serialize(TreeNode root, StringBuilder sb) {
```

```
        if (root == null) {
            return;
        }
        // 前序遍历位置进行序列化
        sb.append(root.val).append(SEP);
        serialize(root.left, sb);
        serialize(root.right, sb);
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if (data.isEmpty()) return null;
        // 转化成前序遍历结果
        LinkedList<Integer> inorder = new LinkedList<>();
        for (String s : data.split(SEP)) {
            inorder.offer(Integer.parseInt(s));
        }
        return deserialize(inorder, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }

    // 定义：将 nodes 中值在闭区间 [min, max] 的节点构造成一棵 BST
    private TreeNode deserialize(LinkedList<Integer> nodes, int min, int
max) {
        if (nodes.isEmpty()) return null;
        // 前序遍历位置进行反序列化
        // 前序遍历结果第一个节点是根节点
        int rootVal = nodes.getFirst();
        if (rootVal > max || rootVal < min) {
            // 超过闭区间 [min, max]，则返回空指针
            return null;
        }
        nodes.removeFirst();
        // 生成 root 节点
        TreeNode root = new TreeNode(rootVal);
        // 构建左右子树
        // BST 左子树都比根节点小，右子树都比根节点大
        root.left = deserialize(nodes, min, rootVal);
        root.right = deserialize(nodes, rootVal, max);

        return root;
    }
}
```

剑指 Offer 37. 序列化二叉树

这道题和 297. 二叉树的序列化与反序列化 相同。

剑指 Offer II 048. 序列化与反序列化二叉树

这道题和 297. 二叉树的序列化与反序列化 相同。

341. 扁平化嵌套列表迭代器

LeetCode	力扣	难度
341. Flatten Nested List Iterator	341. 扁平化嵌套列表迭代器	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，数据结构，设计

给你一个嵌套的整数列表 `nestedList`。每个元素要么是一个整数，要么是一个列表；该列表的元素也可能是整数或者是其他列表。请你实现一个迭代器将其扁平化，使之能够遍历这个列表中的所有整数。

实现扁平迭代器类 `NestedIterator`：

1、`NestedIterator(List<NestedInteger> nestedList)` 用嵌套列表 `nestedList` 初始化迭代器。

2、`int next()` 返回嵌套列表的下一个整数。

3、`boolean hasNext()` 如果仍然存在待迭代的整数，返回 `true`；否则，返回 `false`。

你的代码将会用下述伪代码检测：

```
initialize iterator with nestedList
res = []
while iterator.hasNext()
    append iterator.next() to the end of res
return res
```

如果 `res` 与预期的扁平化列表匹配，那么你的代码将会被判为正确。

示例 1：

```
输入：nestedList = [[1,1],2,[1,1]]
输出：[1,1,2,1,1]
解释：通过重复调用 next 直到 hasNext 返回 false，next 返回的元素的顺序应该是：
[1,1,2,1,1]。
```

基本思路

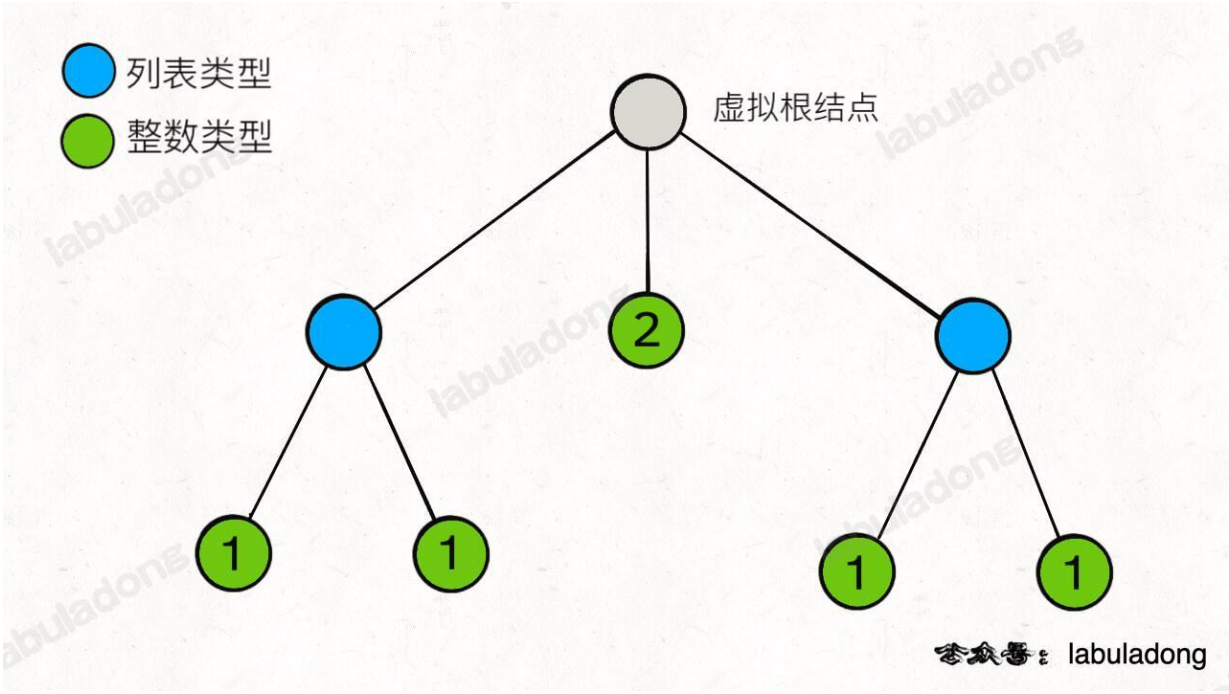
PS：这道题在《算法小抄》 的第 345 页。

题目专门说不要尝试实现或者猜测 `NestedInteger` 的实现，那我们就立即实现一下 `NestedInteger` 的结构：

```
public class NestedInteger {
    private Integer val;
    private List<NestedInteger> list;

    public NestedInteger(Integer val) {
        this.val = val;
        this.list = null;
    }
    public NestedInteger(List<NestedInteger> list) {
        this.list = list;
        this.val = null;
    }

    // 如果其中存的是一个整数，则返回 true，否则返回 false
    public boolean isInteger() {
        return val != null;
    }

    // 如果其中存的是一个整数，则返回这个整数，否则返回 null
    public Integer getInteger() {
        return this.val;
    }

    // 如果其中存的是一个列表，则返回这个列表，否则返回 null
    public List<NestedInteger> getList() {
        return this.list;
    }
}
```

根据 学习数据结构和算法的框架思维，发现这玩意儿竟然就是个多叉树的结构：

```
class NestedInteger {
    Integer val;
    List<NestedInteger> list;
}

// 基本的 N 叉树节点
class TreeNode {
    int val;
    TreeNode[] children;
}
```



所以，把一个 `NestedInteger` 扁平化就等价于遍历一棵 N 叉树的所有「叶子节点」。

用迭代器的方式来实现这个功能，就是**调用 `hasNext` 时，如果 `nestedList` 的第一个元素是列表类型，则不断展开这个元素，直到第一个元素是整数类型。**

- **详细题解：题目不让我干什么，我偏要干什么**

解法代码

```
public class NestedIterator implements Iterator<Integer> {
    private LinkedList<NestedInteger> list;

    public NestedIterator(List<NestedInteger> nestedList) {
        // 不直接用 nestedList 的引用，是因为不能确定它的底层实现
        // 必须保证是 LinkedList，否则下面的 addFirst 会很低效
        list = new LinkedList<>(nestedList);
    }

    public Integer next() {
        // hasNext 方法保证了第一个元素一定是整数类型
        return list.remove(0).getInteger();
    }

    public boolean hasNext() {
        // 循环拆分列表元素，直到列表第一个元素是整数类型
        while (!list.isEmpty() && !list.get(0).isInteger()) {
            // 当列表开头第一个元素是列表类型时，进入循环
            List<NestedInteger> first = list.remove(0).getList();
            // 将第一个列表打平并按顺序添加到开头
            for (int i = first.size() - 1; i >= 0; i--) {
                list.addFirst(first.get(i));
            }
        }
```

```
            return !list.isEmpty();
        }
    }
```

429. N 叉树的层序遍历

LeetCode	力扣	难度
429. N-ary Tree Level Order Traversal	429. N 叉树的层序遍历	

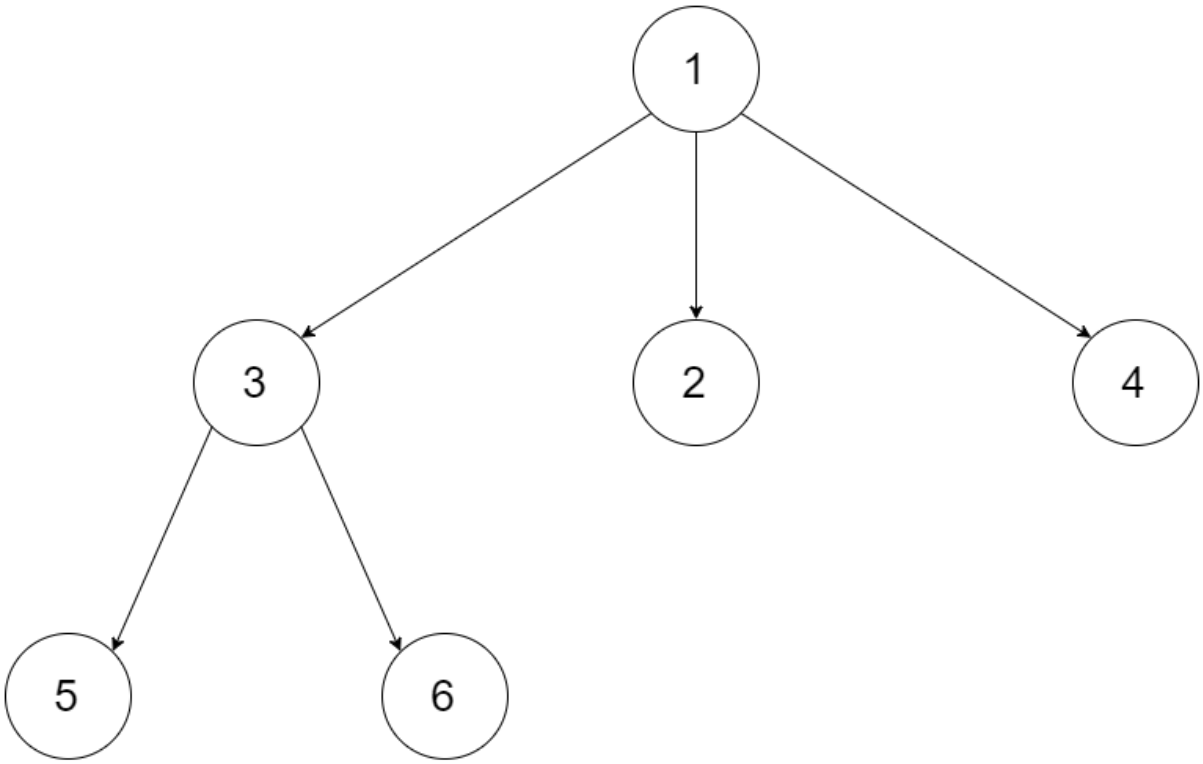
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个 N 叉树，返回其节点值的*层序遍历*。（即从左到右，逐层遍历）。

树的序列化输入是用层序遍历，每组子节点都由 null 值分隔（参见示例）。

示例 1：




```
输入：root = [1,null,3,2,4,null,5,6]
输出：[[1],[3,2,4],[5,6]]
```

基本思路

解法代码

```
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> res = new LinkedList<>();
```

```
        if (root == null) {
            return res;
        }

        // 以下是标准的 BFS 算法遍历框架
        Queue<Node> q = new LinkedList<>();
        q.offer(root);

        while (!q.isEmpty()) {
            int sz = q.size();
            List<Integer> level = new LinkedList<>();
            for (int i = 0; i < sz; i++) {
                Node cur = q.poll();
                level.add(cur.val);
                // 多叉树可能有多个子节点，将所有子节点都加入队列
                for (Node child : cur.children) {
                    q.offer(child);
                }
            }
            // 当前层遍历完毕
            res.add(level);
        }
        return res;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

124. 二叉树中的最大路径和

LeetCode	力扣	难度
124. Binary Tree Maximum Path Sum	124. 二叉树中的最大路径和	🔴

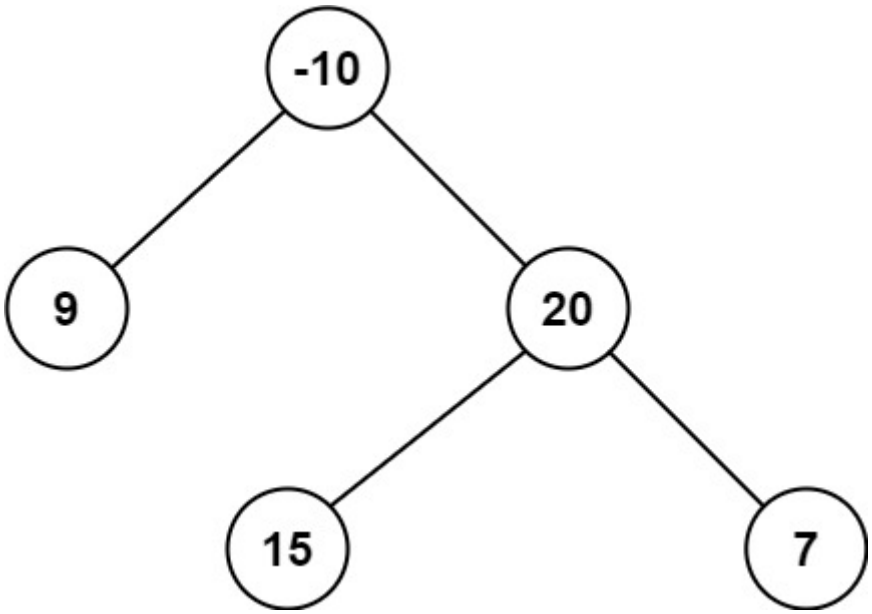
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip，后序遍历

路径被定义为一条从树中**任意**节点出发，达到任意节点的序列。同一个节点在一条路径序列中**至多出现一次**。该路径**至少包含一个**节点，且不一定经过根节点。**路径和**是路径中各节点值的总和。

给你一个二叉树的根节点 `root`，返回其 **最大路径和**。

示例 1：



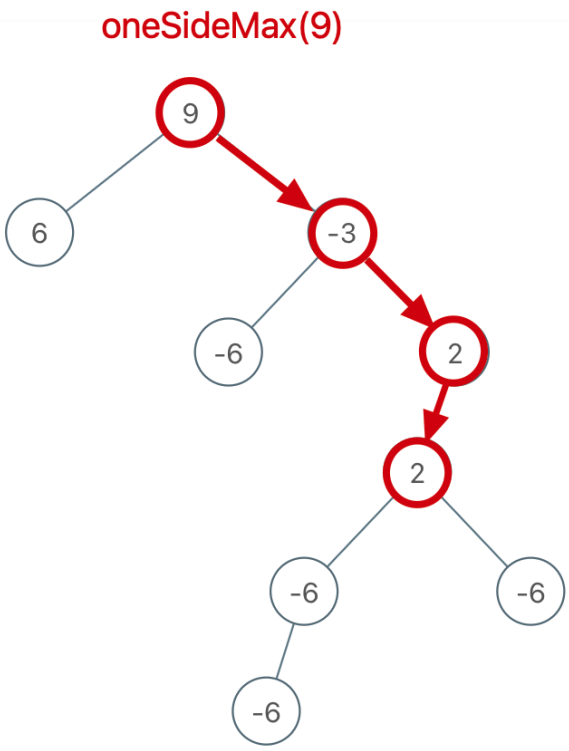

```
输入：root = [-10,9,20,null,null,15,7]
输出：42
解释：最优路径是 15 -> 20 -> 7，路径和为 15 + 20 + 7 = 42
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维。

这题需要巧用二叉树的后序遍历，可以先去做一下 543. 二叉树的直径 和 366. 寻找二叉树的叶子节点。

`oneSideMax` 函数和上述几道题中都用到的 `maxDepth` 函数非常类似，只不过 `maxDepth` 计算最大深度，`oneSideMax` 计算「单边」最大路径和：



然后在后序遍历的时候顺便计算题目要求的最大路径和。

解法代码

```
class Solution {
    int res = Integer.MIN_VALUE;

    public int maxPathSum(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 计算单边路径和时顺便计算最大路径和
        oneSideMax(root);
        return res;
    }

    // 定义：计算从根节点 root 为起点的最大单边路径和
    int oneSideMax(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMaxSum = Math.max(0, oneSideMax(root.left));
        int rightMaxSum = Math.max(0, oneSideMax(root.right));
        // 后序遍历位置，顺便更新最大路径和
        int pathMaxSum = root.val + leftMaxSum + rightMaxSum;
        res = Math.max(res, pathMaxSum);
        // 实现函数定义，左右子树的最大单边路径和加上根节点的值
        // 就是从根节点 root 为起点的最大单边路径和
        return Math.max(leftMaxSum, rightMaxSum) + root.val;
```

```
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 剑指 Offer II 051. 节点之和最大的路径 🔴

1245. 树的直径

LeetCode	力扣	难度
1245. Tree Diameter	1245. 树的直径	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

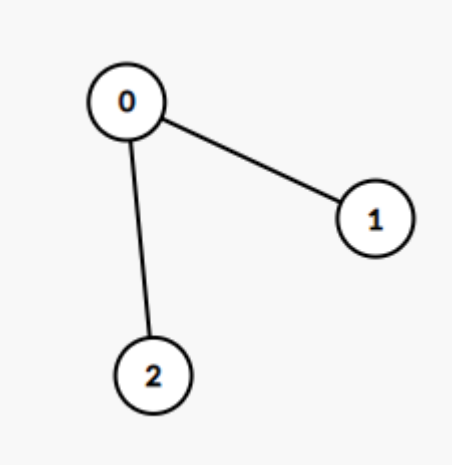
- **标签：二叉树，二叉树vip**

给你这棵「无向树」，请你测算并返回它的「直径」：这棵树上最长简单路径的**边数**。

我们用一个由所有「边」组成的数组 `edges` 来表示一棵无向树，其中 `edges[i] = [u, v]` 表示节点 `u` 和 `v` 之间的双向边。

树上的节点都已经用 `{0, 1, ..., edges.length}` 中的数做了标记，每个节点上的标记都是独一无二的。

示例 1：




```
输入：edges = [[0,1],[0,2]]
输出：2
解释：
这棵树上最长的路径是 1 - 0 - 2，边数为 2。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维模式，同时要利用后序位置来计算答案。

看到这道题，我的第一反应就是去复制 543. 二叉树的直径 的代码，我估计只要让二叉树算法适配本题的多叉树就能把这道题解决了。

但代码改了一半我突然发现，这道题给的输入是若干条「无向边」，换句话说，虽然题目保证输入的是一棵多叉树（不会出现环），但是我不知道哪个是根节点啊？

543 题解法的关键是在 maxDepth 函数的后序位置做点手脚，但如果我连根节点都无法确定，那么所谓树的深度又从何谈起呢？

我又转念一想，根节点有那么重要么？既然题目保证输入的是棵树，那么我随便拎出一个节点不都可以作为根节点吗？

从树的角度来说，把不同的节点作为根节点会产生不同的最大深度，但这并不重要。因为我们算的是「直径」，也就是两个最大深度之和，这个结果是固定的。

综上，只要魔改一下第 543 的代码即可解决这道题：

1、用 邻接表的形式 构建多叉树，以任意一个节点为根节点。

2、计算多叉树的最大深度，在后序位置计算整棵树的最大直径。

结合 543. 二叉树的直径，直接看代码应该不难理解了。

解法代码

```
class Solution {
    // 记录最大直径的长度
    int maxDiameter = 0;

    // 用无向图邻接表的形式存储的一棵多叉树
    HashMap<Integer, List<Integer>> tree = new HashMap<>();
    HashSet<Integer> visited = new HashSet<>();

    public int treeDiameter(int[][] edges) {
        if (edges.length == 0) {
            return 0;
        }
        for (int[] edge : edges) {
            int a = edge[0], b = edge[1];
            if (!tree.containsKey(a)) {
                tree.put(a, new ArrayList<>());
            }
            if (!tree.containsKey(b)) {
                tree.put(b, new ArrayList<>());
            }
            tree.get(a).add(b);
            tree.get(b).add(a);
        }

        // 以任意一个节点为根节点，开始计算最大深度
        maxDepth(edges[0][0]);
        return maxDiameter;
    }

    int maxDepth(int root) {
        if (visited.contains(root)) {
            // 已经遍历过，不要再遍历
            return 0;
```

```
            }
            visited.add(root);

            int firstMaxDepth = 0, secondMaxDepth = 0;
            for (int child : tree.get(root)) {
                int chlidDepth = maxDepth(child);
                if (chlidDepth >= firstMaxDepth) {
                    secondMaxDepth = firstMaxDepth;
                    firstMaxDepth = chlidDepth;
                } else if (chlidDepth > secondMaxDepth) {
                    secondMaxDepth = chlidDepth;
                }
            }
            // 后序位置，顺便计算最大直径
            // 穿过当前节点 root 的最大直径，是子树中最大深度最大的那两个之和
            int myDiameter = firstMaxDepth + secondMaxDepth;
            maxDiameter = Math.max(maxDiameter, myDiameter);

            return 1 + firstMaxDepth;
        }
    }
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

250. 统计同值子树

LeetCode	力扣	难度
250. Count Univalue Subtrees	250. 统计同值子树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip，后序遍历**

给定一个二叉树，统计该二叉树数值相同的子树个数。同值子树是指该子树的所有节点都拥有相同的值。

示例：

```
输入：root = [5,1,5,5,5,null,5]

              5
             / \
            1   5
           / \   \
          5   5   5

输出：4
```

基本思路

这题的难点在于，一个节点如何知道以自己为根的二叉树中的所有节点的值是否全部相同？难道要对每个节点的子树都进行一次遍历去比较吗？

显然不用，这就是前文 手把手刷二叉树总结篇 说的的后序位置的妙用：

如果一个节点想知道子树的信息，那么需要合理地定义递归函数的返回值，然后再后序位置接收子树传递来的返回值。

你可以先去做一下 110. 平衡二叉树 和 543. 二叉树的直径 理解后序位置的特殊性。

我把详细的思路写在解法代码的注释中。

解法代码

```
class Solution {
    public int countUnivalSubtrees(TreeNode root) {
        if (root == null) {
            // 先保证 root 不为空
            return 0;
        }
        getUnivalue(root);
```

```
        return res;
    }

    int res = 0;

    // 定义：输入一棵二叉树，如果这棵二叉树的所有节点值都相同，则返回它们的值，
    // 如果这棵二叉树的所有节点的值不是相同的，则返回 -1001。
    // （因为题目说节点的正常取值为 [-1000, 1000]，所以 -1001 是个特殊值）
    int getUnivalue(TreeNode root) {
        // 先算出左右子树的值是否全部相同
        int left = root.left == null ? root.val : getUnivalue(root.left);
        int right = root.right == null ? root.val :
getUnivalue(root.right);
        // 如果有任何一棵子树的值不相同，那么以 root 为根的这棵树的值肯定不可能全部相
同
        if (left == -1001 || right == -1001) {
            return -1001;
        }
        // 如果左右子树的值都相同，且等于 root.val，
        // 则说明以 root 为根的二叉树是一棵所有节点都相同的二叉树
        if (left == right && root.val == left) {
            // 给全局变量 res 加一
            res++;
            return root.val;
        }
        // 否则，以 root 为根的二叉树不是一棵所有节点都相同的二叉树
        return -1001;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

366. 寻找二叉树的叶子节点

LeetCode	力扣	难度
366. Find Leaves of Binary Tree	366. 寻找二叉树的叶子节点	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip，后序遍历**

给你一棵二叉树，请按以下要求的顺序收集它的全部节点：

1、依次从左到右，每次收集并删除所有的叶子节点。

2、重复如上过程直到整棵树为空。

示例:

```
输入：[1,2,3,4,5]

          1
         / \
        2   3
       / \
      4   5

输出：[[4,5,3],[2],[1]]
```

解释:

1. 删除叶子节点 [4,5,3]，得到如下树结构：

```
      1
     /
    2
```

2. 现在删去叶子节点 [2]，得到如下树结构：

```
      1
```

3. 现在删去叶子节点 [1]，得到空树：

```
      []
```

基本思路

这题有一点难度，需要你深入理解后序遍历特性，可以先看下这篇 美团面试官：你对后序遍历一无所知，最好也做一下这题 543. 二叉树的直径。

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要混用两种思维。

我们正常说的二叉树高度都是从上到下，从根节点到叶子结点递增的，**而这题可以需要思考，把二叉树的高度理解成从叶子节点到根节点从下到上递增的，那么把相同高度的节点分到一起就是题目想要的答案**。

那么，如何知道一个节点距离叶子节点的高度呢?

首先，我们需要实现一个函数 `maxDepth` 计算这棵二叉树的最大高度；然后，我们在 `maxDepth` 函数的后序遍历位置就可以获取当前节点距离叶子节点的高度。

解法代码

```
class Solution {
    ArrayList<List<Integer>> res = new ArrayList<>();

    public List<List<Integer>> findLeaves(TreeNode root) {
        maxDepth(root);
        return res;
    }

    // 定义：输入节点 root，返回以 root 为根的树的最大深度
    int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 当前节点距离叶子节点的高度（最大深度）
        int h = Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
        // 后序遍历位置
        if (res.size() < h) {
            res.add(new ArrayList<>());
        }
        // 把所有相同高度的叶子节点放在一起
        res.get(h - 1).add(root.val);
        return h;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- 类似题目：
  - 124. 二叉树中的最大路径和 🔴
  - 543. 二叉树的直径 🟢

- 剑指 Offer II 051. 节点之和最大的路径 🔴

687. 最长同值路径

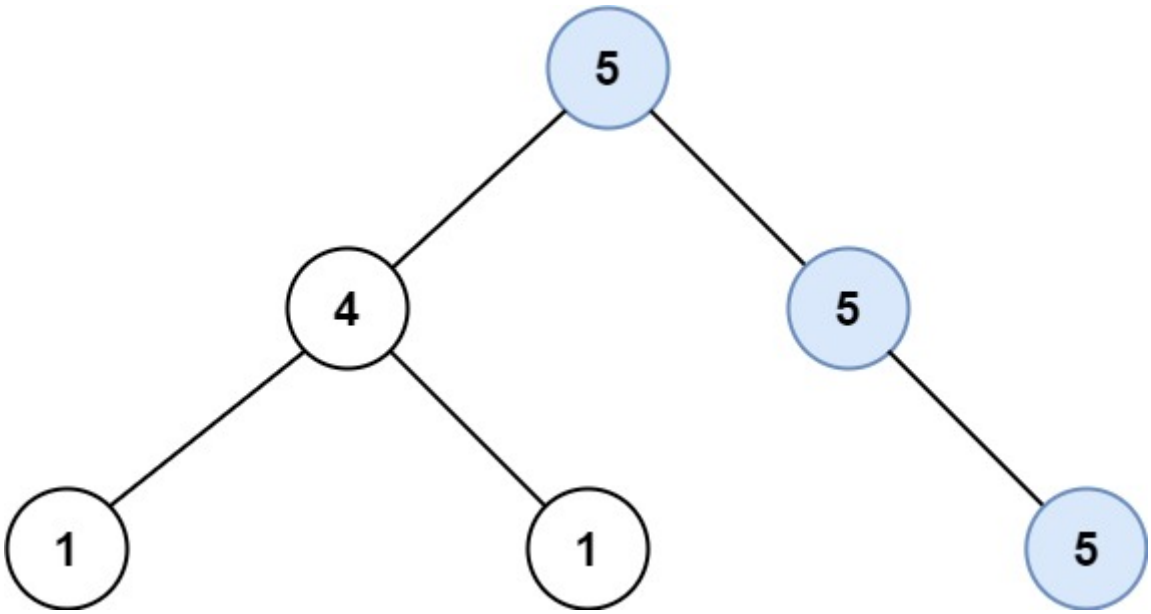
LeetCode	力扣	难度
687. Longest Univalue Path	687. 最长同值路径	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip，后序遍历

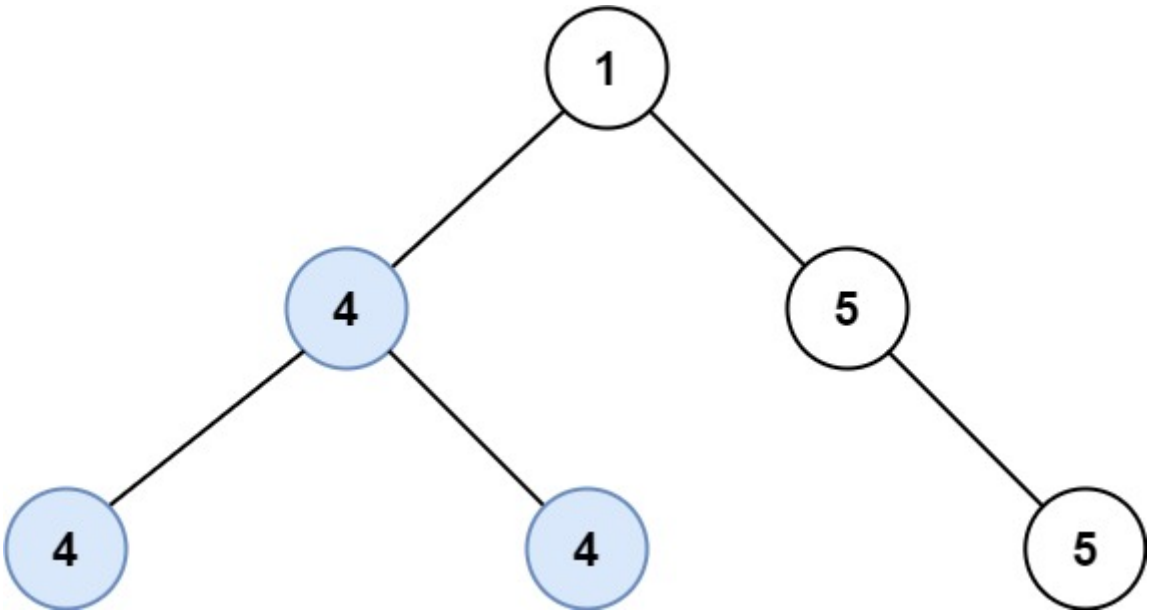
给定一个二叉树，找到最长的路径，这个路径中的每个节点具有相同值。两个节点之间的路径长度由它们之间的边数表示，这条路径可以经过也可以不经过根节点。

示例 1:




```
输入: root = [5,4,5,1,1,5]
输出: 2
```

示例 2:




```
输入: root = [1,4,5,4,4,5]
输出: 2
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维，而且这类题目需要利用二叉树的后序遍历。

做这题之前，我建议你先做 543. 二叉树的直径 题并进行对比，把那道题的最大深度函数 `maxDepth` 的定义带入到这道题中，`maxLen` 相当于求值为 `parentVal` 的节点的最大深度。配合代码注释就立马明白了。

解法代码

```
class Solution {
    int res = 0;
    public int longestUnivaluePath(TreeNode root) {
        if (root == null) {
            return 0;
        }
        // 在后序遍历的位置更新 res
        maxLen(root, root.val);
        return res;
    }

    // 定义：计算以 root 为根的这棵二叉树中，从 root 开始值为 parentVal 的最长树枝长
度
    private int maxLen(TreeNode root, int parentVal) {
        if (root == null) {
            return 0;
        }
        // 利用函数定义，计算左右子树值为 root.val 的最长树枝长度
        int leftLen = maxLen(root.left, root.val);
```

```
            int rightLen = maxLen(root.right, root.val);

            // 后序遍历位置顺便更新全局变量
            // 同值路径就是左右同值树枝长度之和
            res = Math.max(res, leftLen + rightLen);
            // 如果 root 本身和上级值不同，那么整棵子树都不可能有同值树枝
            if (root.val != parentVal) {
                return 0;
            }
            // 实现函数的定义：
            // 以 root 为根的二叉树从 root 开始值为 parentVal 的最长树枝长度
            // 等于左右子树的最长树枝长度的最大值加上 root 节点本身
            return  1 + Math.max(leftLen, rightLen);
        }
    }
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 814. 二叉树剪枝 🟠
  - 剑指 Offer II 047. 二叉树剪枝 🟠

814. 二叉树剪枝

LeetCode	力扣	难度
814. Binary Tree Pruning	814. 二叉树剪枝	

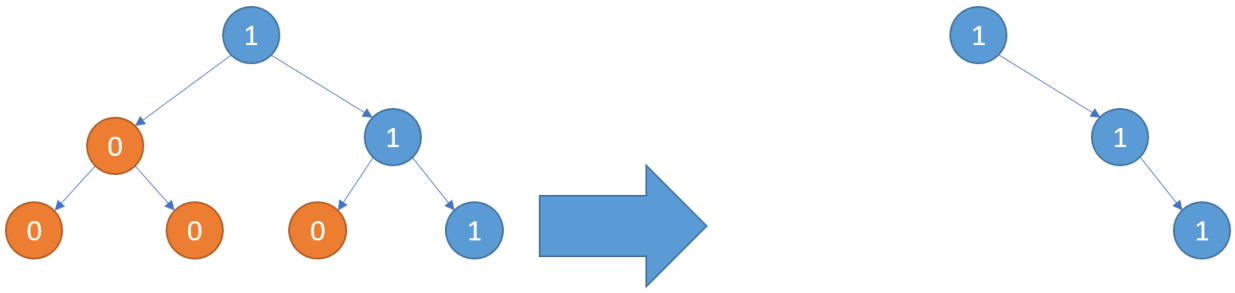
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip

给你二叉树的根结点 `root`，此外树的每个结点的值要么是 `0`，要么是 `1`。

请你对该二叉树进行剪枝，使得返回的二叉树的不存在值为 0 的叶子节点。

示例 1：




```
输入：root = [1,0,1,0,0,0,1]
输出：[1,null,1,null,1]
```

基本思路

建议先做一下 543. 二叉树的直径 和 687. 最长同值路径，理解后序遍历位置的特殊性。

这道题的难点在于要一直剪枝，直到没有值为 0 的叶子节点为止，只有从后序遍历位置自底向上处理才能获得最高的效率。

解法代码

```
class Solution {
    // 定义：输入一棵二叉树，返回的二叉树叶子节点都是 1
    public TreeNode pruneTree(TreeNode root) {
        if (root == null) {
            return null;
        }
        // 二叉树递归框架
        root.left = pruneTree(root.left);
        root.right = pruneTree(root.right);

        // 后序遍历位置，判断自己是否是值为 0 的叶子节点
```

```
            if (root.val == 0 && root.left == null && root.right == null) {
                // 返回值会被父节点接收，相当于把自己删掉了
                return null;
            }
            // 如果不是，正常返回
            return root;
        }
    }
```

▶ 🍭 **代码可视化动画（需安装 Chrome 插件）**🍭

- **类似题目**：
  - 1325. 删除给定值的叶子节点 🟠
  - 剑指 Offer II 047. 二叉树剪枝 🟠

979. 在二叉树中分配硬币

LeetCode	力扣	难度
979. Distribute Coins in Binary Tree	979. 在二叉树中分配硬币	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

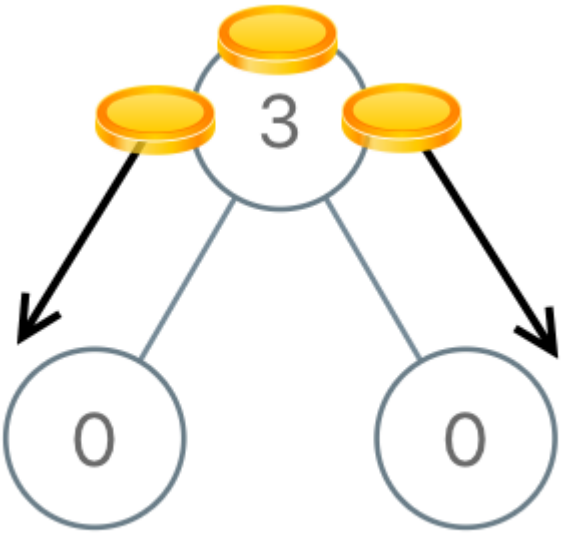
- **标签：二叉树，二叉树vip**

给定一个有 `N` 个结点的二叉树的根结点 `root`，树中的每个结点上都对应有 `node.val` 枚硬币，并且总共有 `N` 枚硬币。

在一次移动中，我们可以选择两个相邻的结点，然后将一枚硬币从其中一个结点移动到另一个结点(移动可以是从父结点到子结点，或者从子结点移动到父结点)。

返回使每个结点上只有一枚硬币所需的最少移动次数。

示例 1：




```
输入：[3,0,0]
输出：2
解释：从树的根结点开始，我们将一枚硬币移到它的左子结点上，一枚硬币移到它的右子结点上。
```

基本思路

做这道题之前，你应该先做一下 543. 二叉树的直径，并且认真理解 手把手刷二叉树总结篇 中说到的后序遍历的用法。

硬币的移动规则看似很复杂，因为一个节点可能需要移出硬币，也可能移入硬币，还要求移动次数最少。实际上我们需要观察规律，做一些等价。

如果一个节点的硬币个数是 `x`，无论是移出还是移入，把该节点的硬币个数变成 1 的最少移动次数必然是 `abs(x - 1)`。

发现这个规律后，开始二叉树的通用解题思路：**假想你现在站在某个根节点上，你如何知道把当前这棵子树的所有节点配平所需的最小移动次数？**

那就很简单了，你假想多余的和缺少的硬币都移动到根节点去配平，当然根节点本身也要配平，所以整棵树配平的移动次数就是：

```
// 让当前这棵树平衡所需的移动次数
Math.abs(left) + Math.abs(right) + (root.val - 1);
```

现在你去看代码就能理解了。

解法代码

```
class Solution {
    public int distributeCoins(TreeNode root) {
        getRest(root);
        return res;
    }

    int res = 0;

    // 定义：输入一棵二叉树，返回这棵二叉树中多出的硬币个数，返回负数代表缺少硬币
    int getRest(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int left = getRest(root.left);
        int right = getRest(root.right);
        // 让当前这棵树平衡所需的移动次数
        res += Math.abs(left) + Math.abs(right) + (root.val - 1);
        // 实现函数的定义
        return left + right + (root.val - 1);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

剑指 Offer II 047. 二叉树剪枝

这道题和 814. 二叉树剪枝 相同。

剑指 Offer II 051. 节点之和最大的路径

这道题和 124. 二叉树中的最大路径和 相同。

1325. 删除给定值的叶子节点

LeetCode	力扣	难度
1325. Delete Leaves With a Given Value	1325. 删除给定值的叶子节点	🟠

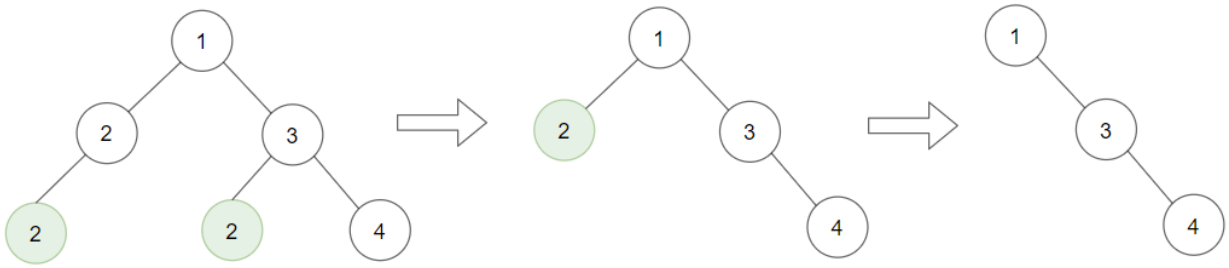
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

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 `target` 的**叶子节点**。

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。

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），它们会被删除，得到中间的图。
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，从而得到最右边的图。
```

基本思路

删除指定值的叶子节点，其实就是遍历所有的叶子节点，然后判断是否需要删除；删除叶子节点也很简单，return null 让父节点接收即可。

难点在于他这个删除操作是循环的，一直删到叶子结点不存在 `target` 为止。这里要用到前文 手把手刷二叉树总结篇 说过的后序位置的妙用了：

一个节点要在后序位置接收左右子树的返回值，才能知道自己的叶子节点是否都被删掉了，以此判断自己是不是变成了叶子节点。

这个考点在 814. 二叉树剪枝 中也有体现，没做过的读者建议去做一下，解法的关键点在于利用后序遍历特点，在后序遍历位置每个节点可以直到自己是否需要被删除。

解法代码

```
class Solution {
    public TreeNode removeLeafNodes(TreeNode root, int target) {
        if (root == null) return null;
        // 二叉树递归框架
        // 如果左右子节点需要被删除，先递归删除它们
        root.left = removeLeafNodes(root.left, target);
        root.right = removeLeafNodes(root.right, target);
        // 后序遍历位置，此时节点 root 直到自己是否需要被删除
        if (root.val == target && root.left == null && root.right == null)
{
            return null;
        }
        return root;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

582. 杀掉进程

LeetCode	力扣	难度
582. Kill Process	582. 杀掉进程	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

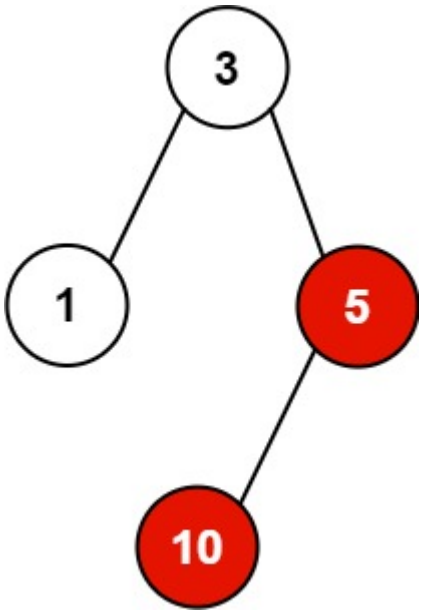
系统中存在 `n` 个进程，形成一个有根树结构。给你两个整数数组 `pid` 和 `ppid`，其中 `pid[i]` 是第 `i` 个进程的 ID，`ppid[i]` 是第 `i` 个进程的父进程 ID。

每一个进程只有 **一个父进程**，但是可能会有 **一个或者多个子进程**。只有一个进程的 `ppid[i] = 0`，意味着这个进程 **没有父进程**。

当一个进程 **被杀掉** 的时候，它所有的子进程和后代进程都要被杀掉。

给你一个整数 `kill` 表示要杀掉进程的 ID，返回杀掉该进程后的所有进程 ID 的列表。可以按 **任意顺序** 返回答案。

示例 1：




```
输入：pid = [1,3,10,5], ppid = [3,0,5,3], kill = 5
输出：[5,10]
解释：涂为红色的进程是应该被杀掉的进程。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维模式。

我觉得这个题非常有趣，先说解法吧，其实不算难。

如果是一棵常规的二叉树，让你从中摘除一棵子树，你会不会？应该很简单对吧，遍历一遍找到子树的根节点，然后从整棵树中摘除就行了。

但本题的难点是我们只有子节点到父节点的引用，和常规二叉树不同，这样就无法用常规的二叉树遍历算法从根节点去遍历整棵树了。

怎么办？很简单，把它转化成常规的二叉树不就行了？好，那怎么转换呢？

这也是我说这题有趣的原因，如果你看过 图论算法基础 和 Union-Find 算法详解 这两篇文章，你就会会心一笑。

首先，题目给你输入的这种树的形式，就是 [Union-Find 算法]](https://labuladong.github.io/article/fname.html?fname=UnionFind算法详解) 中对树的存储形式，其目的是快速计算某个节点的根节点，你如果忘了的话可以去复习一下。

其次，题目给你输入的这两个数组其实就是若干条有向边（`pid[i]->ppid[i]`），那么我们就可以用 图论算法基础 中介绍的通用方式（邻接表/邻接矩阵）去存储。

说了这么多，思路就呼之欲出了：

先把题目输入的这棵树转化成常规形式（父节点指向子节点），然后执行遍历算法（DFS/BFS 都可以）去遍历删除以目标节点 `kill` 为根的整棵子树。

解法代码

```
class Solution {
    public List<Integer> killProcess(List<Integer> pid, List<Integer>
ppid, int kill) {
        // 构建多叉树，key 为父节点，value 为所有子节点的列表
        HashMap<Integer, List<Integer>> tree = new HashMap<>();
        for (int i = 0; i < pid.size(); i++) {
            int child = pid.get(i);
            int parent = ppid.get(i);
            tree.putIfAbsent(parent, new ArrayList<>());
            tree.get(parent).add(child);
        }

        List<Integer> res = new LinkedList<>();
        // 我这里用 BFS 算法遍历子树，删除以 kill 为根的所有子节点
        Queue<Integer> q = new LinkedList<>();
        q.offer(kill);
        while (!q.isEmpty()) {
            int cur = q.poll();
            res.add(cur);
            if (tree.containsKey(cur)) {
                // 所有子节点入队
                q.addAll(tree.get(cur));
            }
        }
        return res;
```

```
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

589. N 叉树的前序遍历

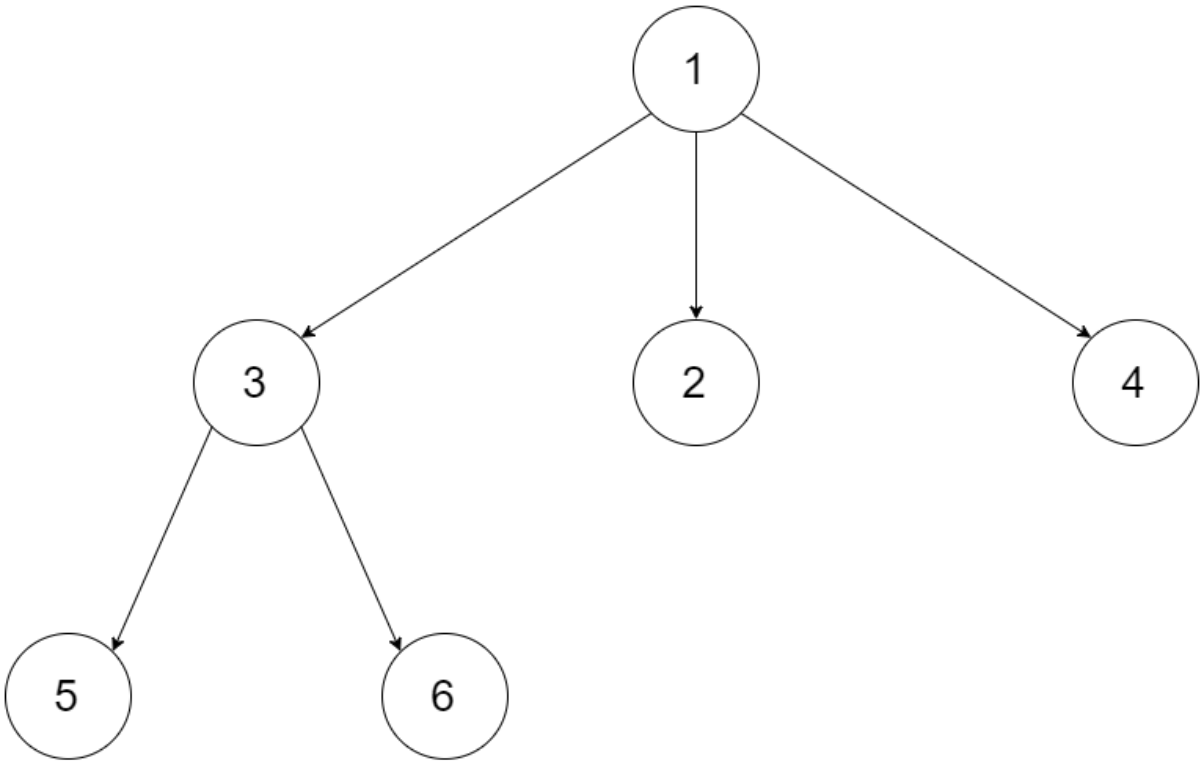
LeetCode	力扣	难度
589. N-ary Tree Preorder Traversal	589. N 叉树的前序遍历	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给定一个 N 叉树，返回其节点值的 **前序遍历**。

示例 1：




```
输入：root = [1,null,3,2,4,null,5,6]
输出：[1,3,5,6,2,4]
```

基本思路

按照 学习数据结构和算法的框架思维 给出的二叉树遍历框架就能推导出多叉树遍历框架了。

解法代码

```
class Solution {
    public List<Integer> preorder(Node root) {
        traverse(root);
```

```
        return res;
    }

    List<Integer> res = new LinkedList<>();

    void traverse(Node root) {
        if (root == null) {
            return;
        }
        // 前序遍历位置
        res.add(root.val);
        for (Node child : root.children) {
            traverse(child);
        }
        // 后序遍历位置
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

590. N 叉树的后序遍历

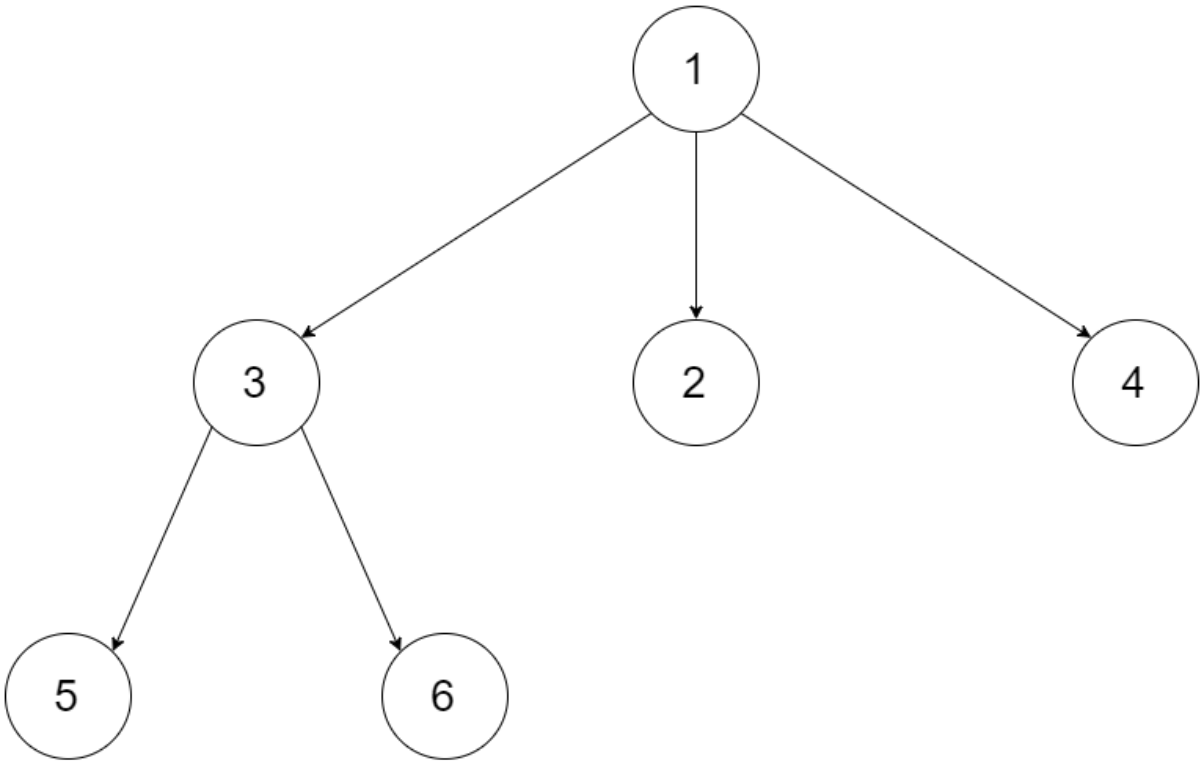
LeetCode	力扣	难度
590. N-ary Tree Postorder Traversal	590. N 叉树的后序遍历	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给定一个 N 叉树，返回其节点值的 **后序遍历**。

示例 1：




```
输入：root = [1,null,3,2,4,null,5,6]
输出：[5,6,3,2,4,1]
```

基本思路

按照 学习数据结构和算法的框架思维 给出的二叉树遍历框架就能推导出多叉树遍历框架了。

解法代码

```
class Solution {
    public List<Integer> postorder(Node root) {
        traverse(root);
```

```
        return res;
    }

    List<Integer> res = new LinkedList<>();

    void traverse(Node root) {
        if (root == null) {
            return;
        }
        // 前序遍历位置
        for (Node child : root.children) {
            traverse(child);
        }
        // 后序遍历位置
        res.add(root.val);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

652. 寻找重复的子树

LeetCode	力扣	难度
652. Find Duplicate Subtrees	652. 寻找重复的子树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树**

给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。

两棵树重复是指它们具有相同的结构以及相同的结点值。

示例 1：

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

下面是两个重复的子树：

```
     2
    /
   4
```

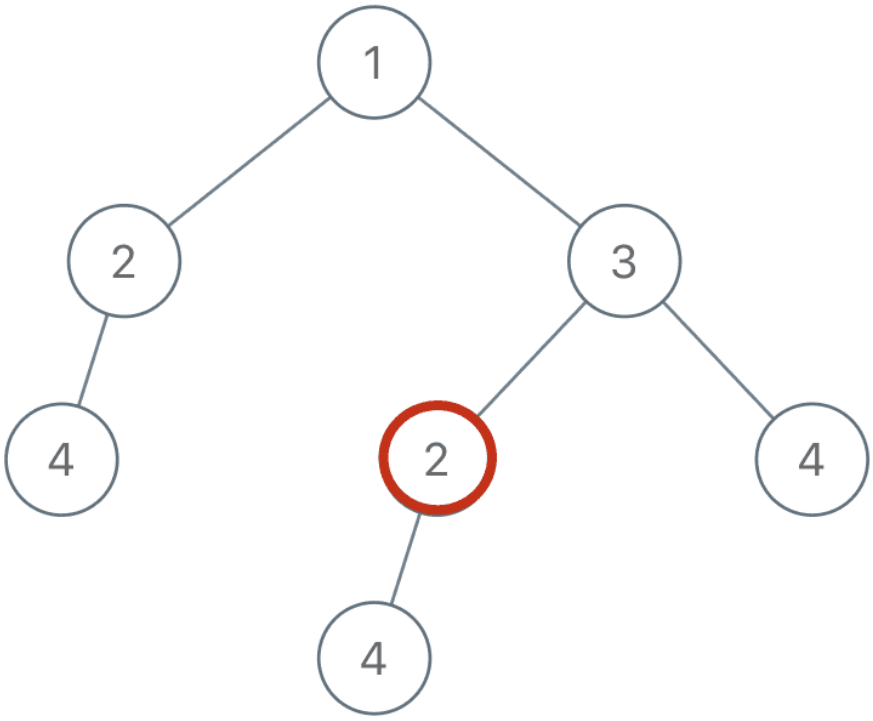
和

```
   4
```

因此，你需要以列表的形式返回上述重复子树的根结点。

基本思路

比如说，你站在图中这个节点 2 上：



如果你想知道以自己为根的子树是不是重复的，是否应该被加入结果列表中，你需要知道什么信息？

你需要知道以下两点：

1、以我为根的这棵二叉树（子树）长啥样？

2、以其他节点为根的子树都长啥样？

这就叫知己知彼嘛，我得知道自己长啥样，还得知道别人长啥样，然后才能知道有没有人跟我重复，对不对？

我怎么知道自己以我为根的二叉树长啥样？前文 序列化和反序列化二叉树 其实写过了，二叉树的前序/中序/后序遍历结果可以描述二叉树的结构。

我咋知道其他子树长啥样？每个节点都把以自己为根的子树的样子存到一个外部的数据结构里即可，按照这个思路看代码就不难理解了。

这道题的事件复杂度还是比较容易分析的，二叉树有 N 个节点，遍历需要 O(N) 时间；遍历每个节点时需要序列化子树所有节点，所需时间粗略算作 O(N)，那么总的复杂度就是它俩相乘，即 O(N^2)。

- **详细题解：东哥带你刷二叉树（后序篇）**

解法代码

```
class Solution {
    // 记录所有子树以及出现的次数
    HashMap<String, Integer> memo = new HashMap<>();
    // 记录重复的子树根节点
    LinkedList<TreeNode> res = new LinkedList<>();
```

```java
    /* 主函数 */
    public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
        traverse(root);
        return res;
    }

    String traverse(TreeNode root) {
        if (root == null) {
            return "#";
        }

        String left = traverse(root.left);
        String right = traverse(root.right);

        String subTree = left + "," + right + "," + root.val;

        int freq = memo.getOrDefault(subTree, 0);
        // 多次重复也只会被加入结果集一次
        if (freq == 1) {
            res.add(root);
        }
        // 给子树对应的出现次数加一
        memo.put(subTree, freq + 1);
        return subTree;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

998. 最大二叉树 II

LeetCode	力扣	难度
998. Maximum Binary Tree II	998. 最大二叉树 II	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

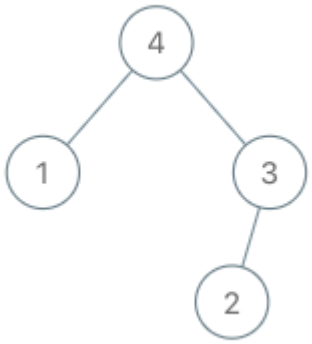
最大树定义：一个树，其中每个节点的值都大于其子树中的任何其他值。就像 之前的问题 那样，给定的树是从列表 `A`（`root = Construct(A)`）递归地使用下述 `Construct(A)` 例程构造的：

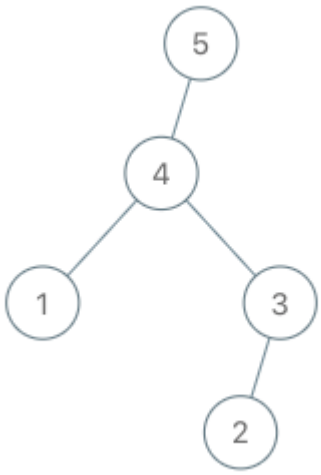
- 如果 `A` 为空，返回 `null`
- 否则，令 `A[i]` 作为 A 的最大元素。创建一个值为 `A[i]` 的根节点 `root`
- `root` 的左子树将被构建为 `Construct([A[0], A[1], ..., A[i-1]])`
- `root` 的右子树将被构建为 `Construct([A[i+1], A[i+2], ..., A[A.length - 1]])`
- 返回 `root`

请注意，我们没有直接给定 `A`，只有一个根节点 `root = Construct(A)`。

假设 `B` 是 `A` 的副本，并在末尾附加值 `val`，题目数据保证 `B` 中的值是不同的，请你返回 `Construct(B)` 的根节点。

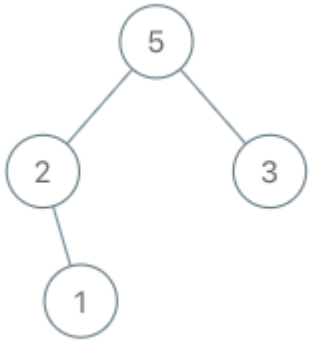
示例 1：

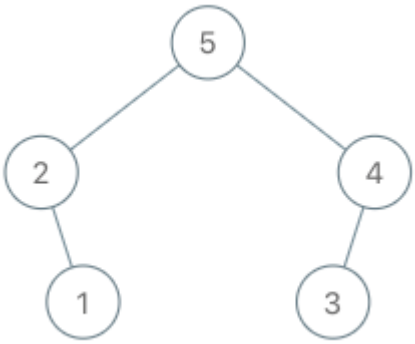





```
输入：root = [4,1,3,null,null,2], val = 5
输出：[5,4,null,1,3,null,null,2]
解释：A = [1,4,2,3], B = [1,4,2,3,5]
```

示例 2：





**

```
输入：root = [5,2,3,null,1], val = 4
输出：[5,2,4,null,1,3]
解释：A = [2,1,5,3], B = [2,1,5,3,4]
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维。

做这道题之前，你一定要去做一下 654. 最大二叉树 这道题，知道了构建最大二叉树的逻辑就很容易解决这道题了。

新增的 `val` 是添加在原始数组的最后的，根据构建最大二叉树的逻辑，正常来说最后的这个值一定是在右子树的，可以对右子树递归调用 `insertIntoMaxTree` 插入进去。

但是一种特殊情况是 `val` 比原始数组中的所有元素都大，那么根据构建最大二叉树的逻辑，原来的这棵树应该成为 `val` 节点的左子树。

解法代码

```
class Solution {
    public TreeNode insertIntoMaxTree(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }
        if (root.val < val) {
            // 如果 val 是整棵树最大的，那么原来的这棵树应该是 val 节点的左子树，
            // 因为 val 节点是接在原始数组 a 的最后一个元素
            TreeNode temp = root;
            root = new TreeNode(val);
            root.left = temp;
        } else {
            // 如果 val 不是最大的，那么就应该在右子树上，
            // 因为 val 节点是接在原始数组 a 的最后一个元素
            root.right = insertIntoMaxTree(root.right, val);
        }
        return root;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

965. 单值二叉树

LeetCode	力扣	难度
965. Univalued Binary Tree	965. 单值二叉树	

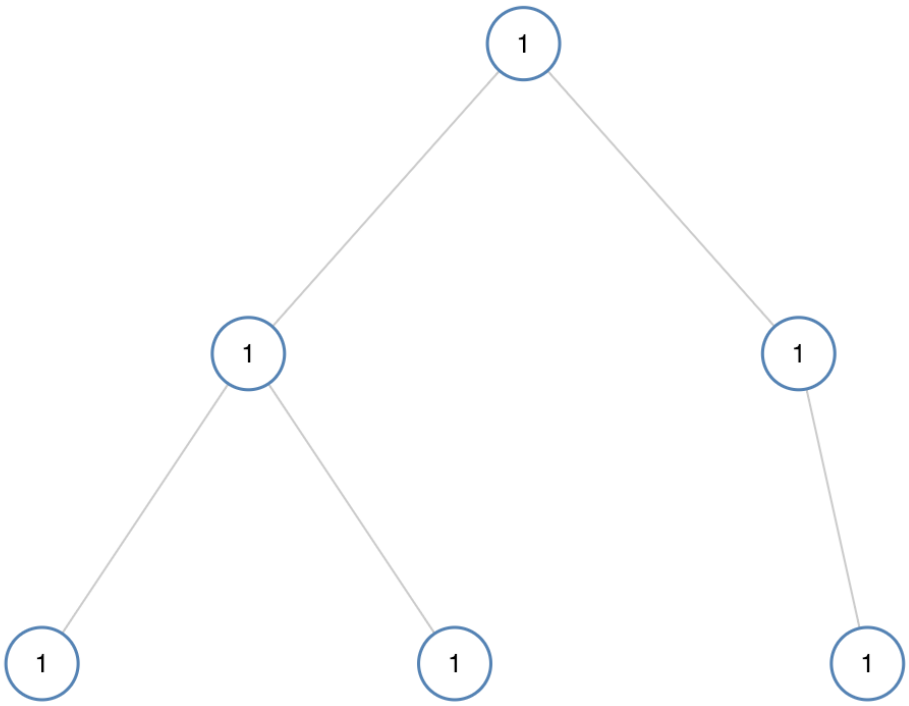
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

如果二叉树每个节点都具有相同的值，那么该二叉树就是**单值**二叉树。

只有给定的树是单值二叉树时，才返回 `true`；否则返回 `false`。

示例 1：




```
输入：[1,1,1,1,1,null,1]
输出：true
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

用 `traverse` 遍历框架遍历一遍二叉树即可得出答案。

解法代码

```
class Solution {
    public boolean isUnivalTree(TreeNode root) {
        if (root == null) {
            return true;
        }
        prev = root.val;
        traverse(root);
        return isUnival;
    }

    int prev;
    boolean isUnival = true;

    void traverse(TreeNode root) {
        if (root == null || !isUnival) {
            return;
        }
        if (root.val != prev) {
            isUnival = false;
            return;
        }
        traverse(root.left);
        traverse(root.right);
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

1740. 找到二叉树中的距离

LeetCode	力扣	难度
1740. Find Distance in a Binary Tree	1740. 找到二叉树中的距离	🟠

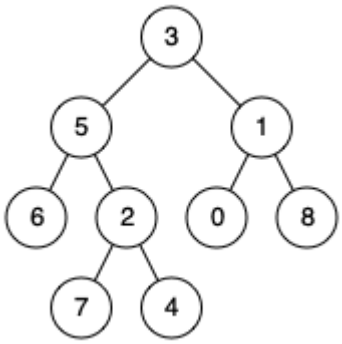
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给定一棵二叉树的根节点 root 以及两个整数 p 和 q，返回该二叉树中值为 p 的结点与值为 q 的结点间的 **距离**。

两个结点间的 **距离** 就是从一个结点到另一个结点的路径上边的数目。

示例 1：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 0
输出：3
解释：在 5 和 0 之间有 3 条边：5-3-1-0
```

基本思路

这道题很明显是最近公共祖先问题（LCA）的拓展，原版的 LCA 问题是让我们返回最近公共祖先节点，现在相当于是在问 p 和 q 到最近公共祖先的树枝的距离之和。

不过说实话，这题的难度还是比较大的，需要你认真阅读我写的 一文秒杀所有二叉树最近公共祖先问题，理解 LCA 问题解法逐步演进的原理，然后再来看这道题。

具体看代码注释吧，最好结合 东哥手把手带你刷二叉树（纲领篇） 体悟一下二叉树的遍历顺序，毕竟递归算法不是线性执行的，只要搞明白代码的执行顺序就可以更深刻地理解递归。

解法代码

```
class Solution {
    public int findDistance(TreeNode root, int p, int q) {
```

```
        lca(root, p, q);
        return res;
    }


    boolean found = false;
    int res = 0;

    // 定义：当子树中不包含 p 或 q 时，返回 0；
    // 当子树中仅包含 p 或 q 中的一个时，返回 root 到 p 或 q 的距离；
    // 当子树中同时包含 p 和 q 时，返回一个无意义的值（答案会被存在外部变量 res 中）
    int lca(TreeNode root, int p, int q) {
        if (found) {
            // found 为 true 时答案已经被记录在全局 res 中
            // 递归函数的返回值已不需要了，此时返回什么值都无所谓
            return 123;
        }
        if (root == null) {
            return 0;
        }

        int left = lca(root.left, p, q);
        int right = lca(root.right, p, q);

        if (left == 0 && right == 0) {
            // root 的左右子树都不包含 p 或 q
            if (root.val == p || root.val == q) {
                return 1;
            }
            return 0;
        }

        if (left != 0 && right != 0 && !found) {
            // 当前节点 root 就是 p, q 的最近公共祖先节点
            // 答案已经算出来了，更新全局变量
            res = left + right;
            // 这个递归函数的返回值已经不重要了，让它终止递归
            found = true;
            return 456;
        }

        // 此时 left 和 right 有一个为 0，即只找到了一个节点
        // branch 就是到该节点的距离
        int branch = left == 0 ? right : left;

        if (root.val == p || root.val == q) {
            res = branch;
        }

        return branch + 1;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

剑指 Offer 33. 二叉搜索树的后序遍历序列

LeetCode	力扣	难度
剑指Offer33. 二叉搜索树的后序遍历序列 LCOF	剑指Offer33. 二叉搜索树的后序遍历序列	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉搜索树，二叉树**

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 `true`，否则返回 `false`。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```
     5
    / \
   2   6
  / \
 1   3
```

示例 1：

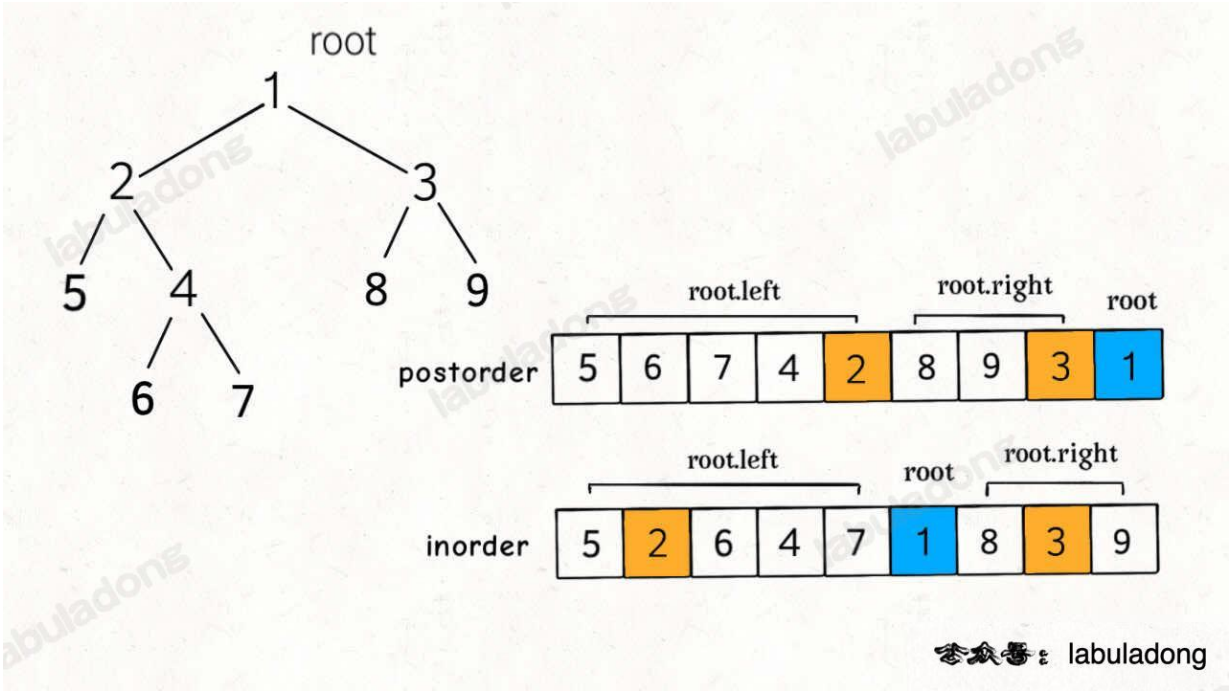
```
输入：[1,6,3,2,5]
输出：false
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维模式。

建议你先看一下 东哥手把手带你刷二叉树（构造篇） 了解二叉树前中后序遍历结果的特点，再看一下 手把手带你刷二叉搜索树（第二期） 了解 BST 的特性和基本操作方法，再来看这道题，解法就呼之欲出了。

后序遍历结果的特点是：左边一部分是左子树，右边一部分是右子树，最后一个元素是根节点。



而 BST 的特点是左子树元素都小于根节点，右子树元素都大于根节点。

所以结合上述规律，就可以得到解决本题的思路：

1、先找到根节点元素

2、根据根节点元素找到左子树元素，递归检查左子树是否是 BST

3、根据根节点元素找到右子树元素，递归检查右子树是否是 BST

解法代码

```
class Solution {
    public boolean verifyPostorder(int[] postorder) {
        return check(postorder, 0, postorder.length - 1);
    }

    // 定义：检查 postorder[i..j] 是否是一个合法的 BST
    boolean check(int[] postorder, int i, int j) {
        if (i >= j) {
            return true;
        }
        // 根节点的值是后序遍历结果的最后一个元素
        int root = postorder[j];

        // postorder[i..left) 是左子树，应该都小于 root
        int left = i;
        while (left < j && postorder[left] < root) {
            left++;
        }

        // postorder[left..j) 是右子树，应该都大于 root
        int right = left;
```

```
        while (right < j && postorder[right] > root) {
            right++;
        }
        if (right != j) {
            return false;
        }
        // 递归检查左子树 [i..left) 和右子树 [left..j) 也符合 BST 的性质
        return check(postorder, i, left - 1) && check(postorder, left, j -
1);
    }

}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

95. 不同的二叉搜索树 II

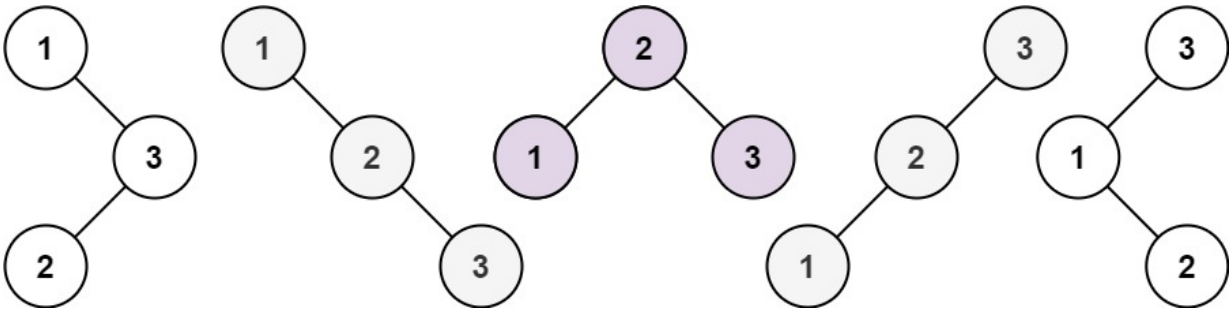
LeetCode	力扣	难度
95. Unique Binary Search Trees II	95. 不同的二叉搜索树 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，数据结构

给你一个整数 n，请你生成并返回所有由 n 个节点组成且节点值从 1 到 n 互不相同的不同**二叉搜索树**，可以按**任意顺序**返回答案。

示例 1：




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

基本思路

类似 96. 不同的二叉搜索树，这题的思路也是类似的，想要构造出所有合法 BST，分以下三步：

1、穷举 root 节点的所有可能。

2、递归构造出左右子树的所有合法 BST。

3、给 root 节点穷举所有左右子树的组合。

- 详细题解：东哥带你刷二叉搜索树（构造篇）

解法代码

```
class Solution {
    /* 主函数 */
    public List<TreeNode> generateTrees(int n) {
        if (n == 0) return new LinkedList<>();
        // 构造闭区间 [1, n] 组成的 BST
        return build(1, n);
```

```
    }

    /* 构造闭区间 [lo, hi] 组成的 BST */
    List<TreeNode> build(int lo, int hi) {
        List<TreeNode> res = new LinkedList<>();
        // base case
        if (lo > hi) {
            res.add(null);
            return res;
        }

        // 1、穷举 root 节点的所有可能。
        for (int i = lo; i <= hi; i++) {
            // 2、递归构造出左右子树的所有合法 BST。
            List<TreeNode> leftTree = build(lo, i - 1);
            List<TreeNode> rightTree = build(i + 1, hi);
            // 3、给 root 节点穷举所有左右子树的组合。
            for (TreeNode left : leftTree) {
                for (TreeNode right : rightTree) {
                    // i 作为根节点 root 的值
                    TreeNode root = new TreeNode(i);
                    root.left = left;
                    root.right = right;
                    res.add(root);
                }
            }
        }
        return res;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 96. 不同的二叉搜索树 🟠

96. 不同的二叉搜索树

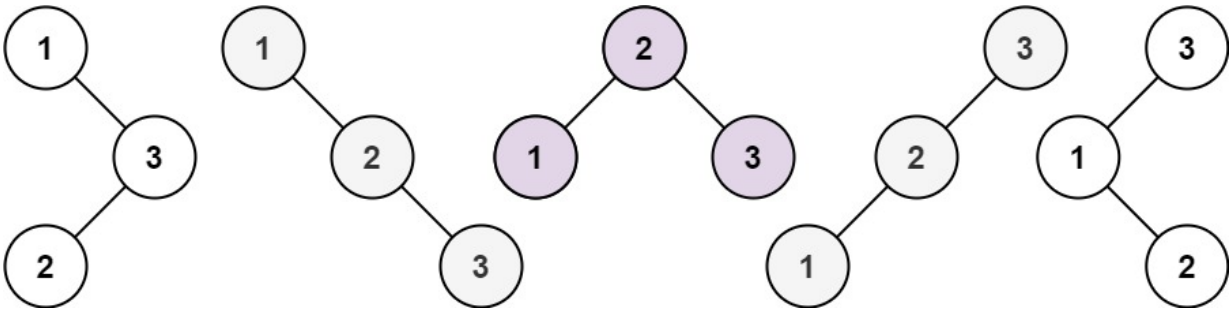
LeetCode	力扣	难度
96. Unique Binary Search Trees	96. 不同的二叉搜索树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，数据结构

给你一个整数 n，求恰由 n 个节点组成且节点值从 1 到 n 互不相同的**二叉搜索树**有多少种？返回满足题意的二叉搜索树的种数。

示例 1：

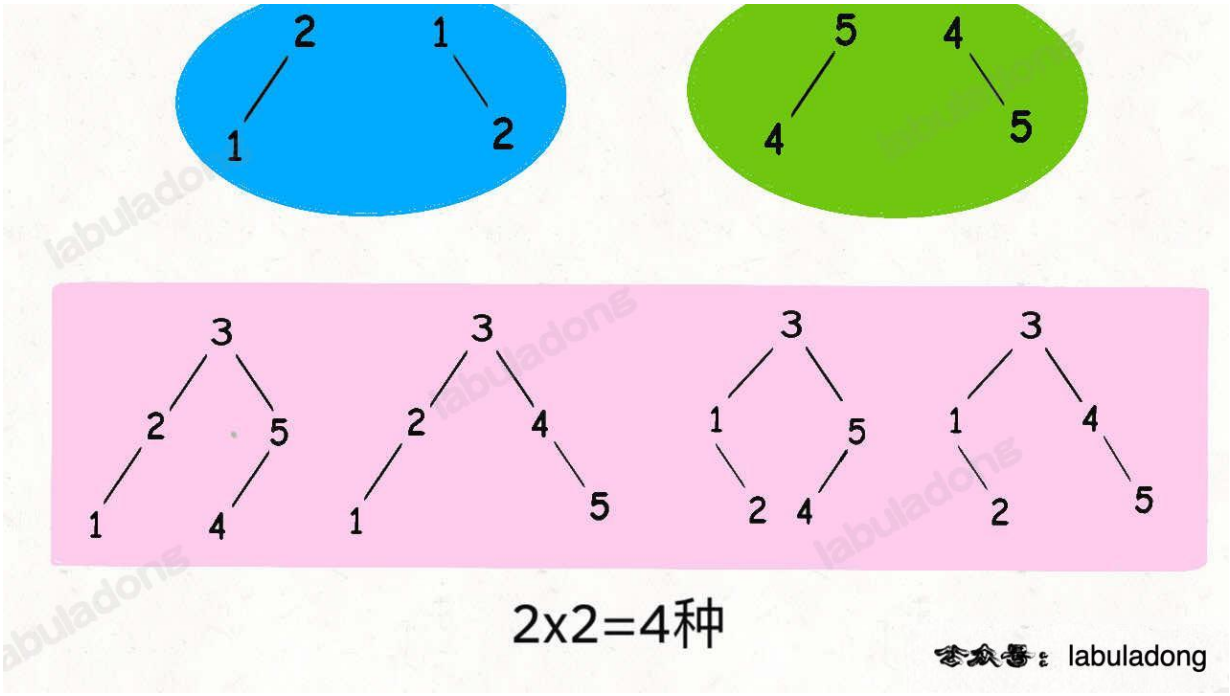



```
输入：n = 3
输出：5
```

基本思路

假设给算法输入 `n = 5`，也就是说用 `{1,2,3,4,5}` 这些数字去构造 BST。

如果固定 3 作为根节点，左子树节点就是 `{1,2}` 的组合，右子树就是 `{4,5}` 的组合：



那么 {1,2} 和 {4,5} 的组合有多少种呢？只要合理定义递归函数，这些可以交给递归函数去做。

另外，这题存在重叠子问题，可以通过备忘录的方式消除冗余计算。

- **详细题解：东哥带你刷二叉搜索树（构造篇）**

解法代码

```
class Solution {
    // 备忘录
    int[][] memo;

    int numTrees(int n) {
        // 备忘录的值初始化为 0
        memo = new int[n + 1][n + 1];
        return count(1, n);
    }

    int count(int lo, int hi) {
        if (lo > hi) return 1;
        // 查备忘录
        if (memo[lo][hi] != 0) {
            return memo[lo][hi];
        }

        int res = 0;
        for (int mid = lo; mid <= hi; mid++) {
            int left = count(lo, mid - 1);
            int right = count(mid + 1, hi);
            res += left * right;
        }
        // 将结果存入备忘录
        memo[lo][hi] = res;
```

```
            return res;
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 95. 不同的二叉搜索树 II 🟠

255. 验证前序遍历序列二叉搜索树

LeetCode	力扣	难度
255. Verify Preorder Sequence in Binary Search Tree	255. 验证前序遍历序列二叉搜索树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树vip

给定一个整数数组，你需要验证它是否是一个二叉搜索树正确的先序遍历序列。

示例 1：

```
     5
    / \
   2   6
  / \
 1   3

输入: [5,2,6,1,3]
输出: false
```

基本思路

二叉树的构造/检验问题遵循固定的套路，说白了就是先找到根节点，再处理左右子树。这题的解法不止一个，我给出一个效率并不是最优，但比较符合通用套路的解法。

所以本题只要把 98. 验证二叉搜索树 和 1008. 前序遍历构造二叉搜索树 的解法逻辑相结合，即可解决这道题。

你可以先把那两道题做一下再回过来做这道题。

解法代码

```
class Solution {
    public boolean verifyPreorder(int[] preorder) {
        int n = preorder.length;
        return check(preorder, 0, n - 1, Integer.MIN_VALUE,
Integer.MAX_VALUE);
    }

    // 定义：检查 preorder[start..end] 是否能够组成一棵值在 [min, max] 中的 BST
    boolean check(int[] preorder, int start, int end, int min, int max) {
        if (start > end) {
            return true;
        }
```

```
        // 前序遍历结果中 root.val 在第一位
        int rootVal = preorder[start];
        if (rootVal < min || rootVal > max) {
            return false;
        }

        // 让 p 移动到右子树的开头
        int p = start + 1;
        while (p <= end && preorder[p] < rootVal) {
            p++;
        }
        // 递归检查左右子树是否是 BST
        return check(preorder, start + 1, p - 1, min, rootVal)
                && check(preorder, p, end, rootVal, max);
    }
}
```

▶ 代码可视化动画（需安装 Chrome 插件）

450. 删除二叉搜索树中的节点

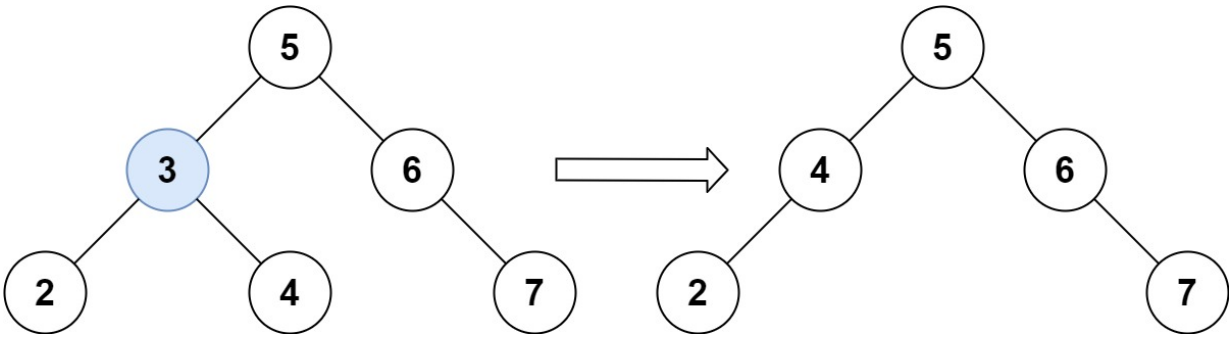
LeetCode	力扣	难度
450. Delete Node in a BST	450. 删除二叉搜索树中的节点	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，数据结构

给定一个二叉搜索树的根节点 `root` 和一个值 `key`，删除二叉搜索树中的 `key` 对应的节点，并保证二叉搜索树的性质不变，返回删除后的二叉搜索树的根节点（有可能被更新）。

示例 1:



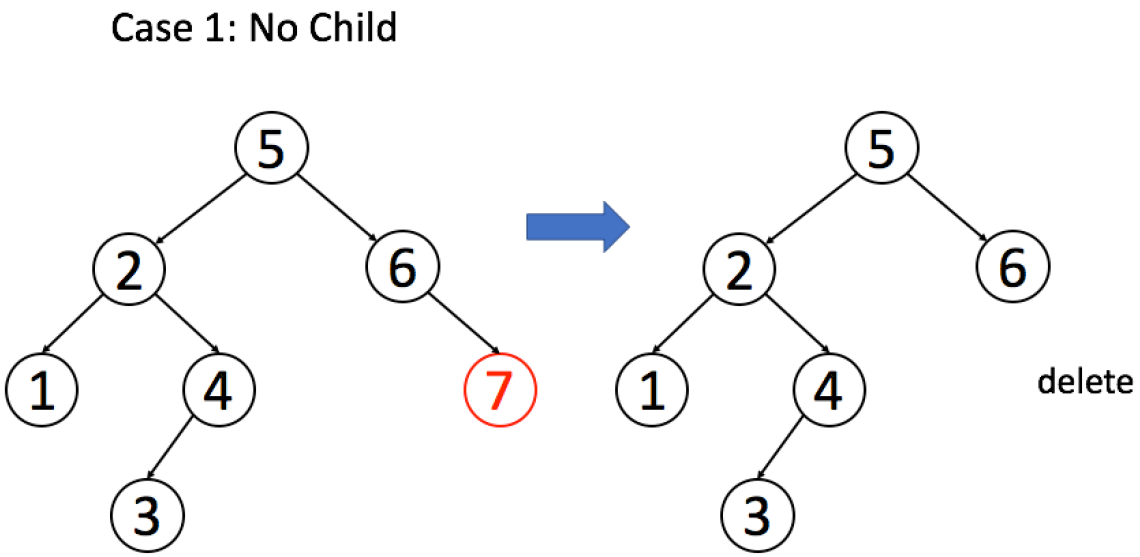

```
输入：root = [5,3,6,2,4,null,7], key = 3
输出：[5,4,6,2,null,null,7]
解释：给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。
一个正确的答案是 [5,4,6,2,null,null,7], 如下图所示。
另一个正确答案是 [5,2,6,null,4,null,7]。
```

基本思路

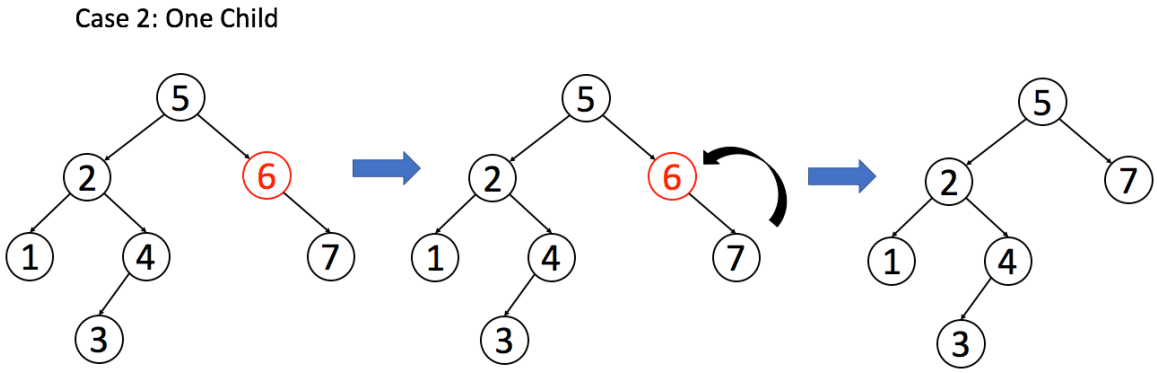
PS：这道题在《算法小抄》 的第 235 页。

删除比插入和搜索都要复杂一些，分三种情况：

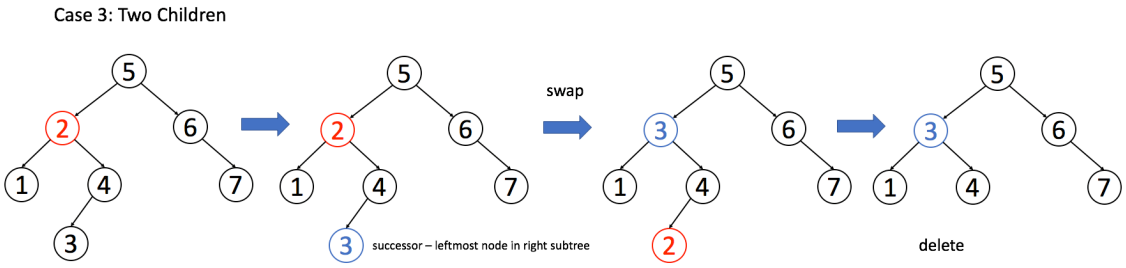
情况 1：A 恰好是末端节点，两个子节点都为空，那么它可以当场去世了：



情况 2：A 只有一个非空子节点，那么它要让这个孩子接替自己的位置：



情况 3：A 有两个子节点，麻烦了，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点或者右子树中最小的那个节点来接替自己，我的解法是用右子树中最小节点来替换：



- **详细题解：东哥带你刷二叉搜索树（基操篇）**

解法代码

```
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) return null;
        if (root.val == key) {
```

```
                // 这两个 if 把情况 1 和 2 都正确处理了
                if (root.left == null) return root.right;
                if (root.right == null) return root.left;
                // 处理情况 3
                // 获得右子树最小的节点
                TreeNode minNode = getMin(root.right);
                // 删除右子树最小的节点
                root.right = deleteNode(root.right, minNode.val);
                // 用右子树最小的节点替换 root 节点
                minNode.left = root.left;
                minNode.right = root.right;
                root = minNode;
            } else if (root.val > key) {
                root.left = deleteNode(root.left, key);
            } else if (root.val < key) {
                root.right = deleteNode(root.right, key);
            }
            return root;
        }

        TreeNode getMin(TreeNode node) {
            // BST 最左边的就是最小的
            while (node.left != null) node = node.left;
            return node;
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 700. 二叉搜索树中的搜索 🟢
  - 701. 二叉搜索树中的插入操作 🟠
  - 98. 验证二叉搜索树 🟠

700. 二叉搜索树中的搜索

LeetCode	力扣	难度
700. Search in a Binary Search Tree	700. 二叉搜索树中的搜索	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉搜索树，数据结构**

给定二叉搜索树（BST）的根节点和一个目标值。你需要在 BST 中找到节点值等于目标值的节点并返回，如果节点不存在，则返回 NULL。

例如，

```
给定二叉搜索树：

        4
      / \
     2   7
    / \
   1   3

目标值：2
```

你应该返回节点 2。

基本思路

PS：这道题在《算法小抄》 的第 235 页。

利用 BST 左小右大的特性，可以避免搜索整棵二叉树去寻找元素，从而提升效率。

- **详细题解：东哥带你刷二叉搜索树（基操篇）**

解法代码

```
class Solution {
    public TreeNode searchBST(TreeNode root, int target) {
        if (root == null) {
            return null;
        }
        // 去左子树搜索
        if (root.val > target) {
            return searchBST(root.left, target);
        }
        // 去右子树搜索
```

```
            if (root.val < target) {
                return searchBST(root.right, target);
            }
            return root;
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 270. 最接近的二叉搜索树值 🟢
  - 285. 二叉搜索树中的中序后继 🟠
  - 450. 删除二叉搜索树中的节点 🟠
  - 701. 二叉搜索树中的插入操作 🟠
  - 98. 验证二叉搜索树 🟠
  - 剑指 Offer II 053. 二叉搜索树中的中序后继 🟠

701. 二叉搜索树中的插入操作

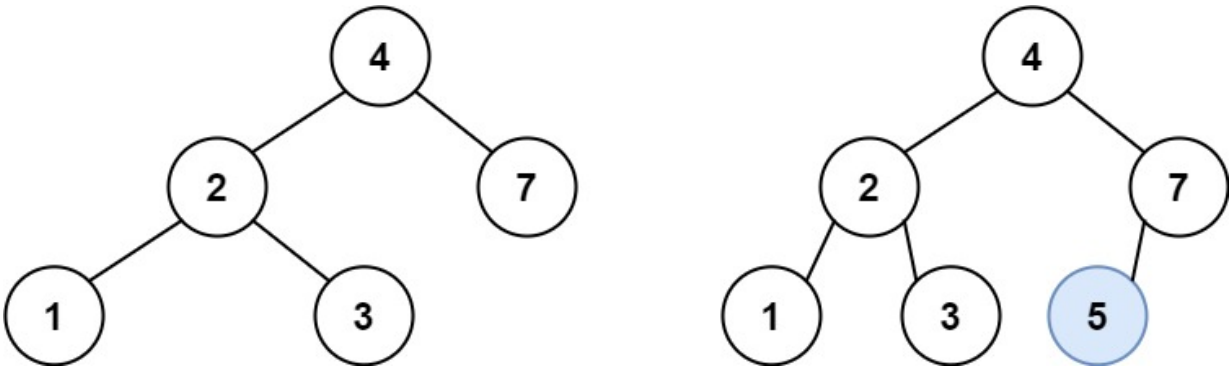
LeetCode	力扣	难度
701. Insert into a Binary Search Tree	701. 二叉搜索树中的插入操作	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

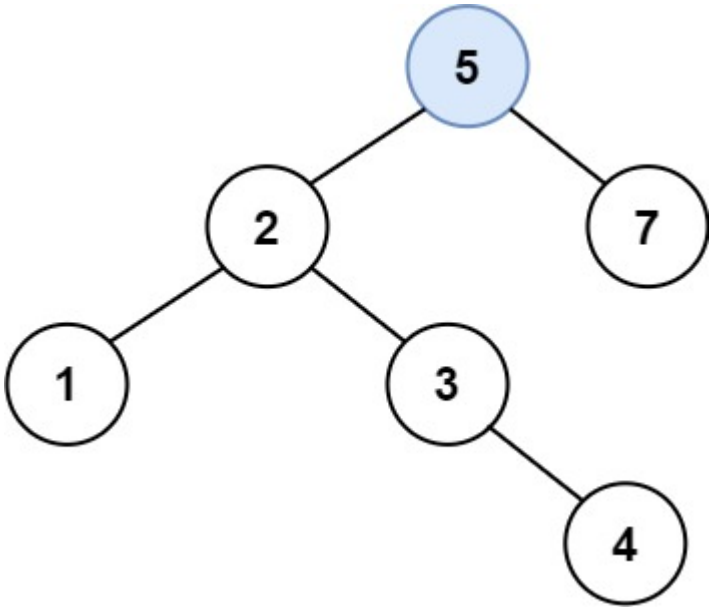
- 标签：二叉搜索树，数据结构

给定二叉搜索树（BST）的根节点和要插入树中的值（不会插入 BST 已存在的值），将值插入二叉搜索树，返回插入后二叉搜索树的根节点。

示例 1：




```
输入：root = [4,2,7,1,3], val = 5
输出：[4,2,7,1,3,5]
解释：另一个满足题目要求可以通过的树是：
```



基本思路

PS：这道题在《算法小抄》 的第 235 页。

如果要递归地插入或者删除二叉树节点，递归函数一定要有返回值，而且返回值要被正确的接收。

插入的过程可以分两部分：

1、寻找正确的插入位置，类似 700. 二叉搜索树中的搜索。

2、把元素插进去，这就要把新节点以返回值的方式接到父节点上。

- **详细题解：东哥带你刷二叉搜索树（基操篇）**

解法代码

```
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        // 找到空位置插入新节点
        if (root == null) return new TreeNode(val);
        // if (root.val == val)
        //     BST 中一般不会插入已存在元素
        if (root.val < val)
            root.right = insertIntoBST(root.right, val);
        if (root.val > val)
            root.left = insertIntoBST(root.left, val);
        return root;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 450. 删除二叉搜索树中的节点 🟠
  - 700. 二叉搜索树中的搜索 🟢
  - 98. 验证二叉搜索树 🟠

98. 验证二叉搜索树

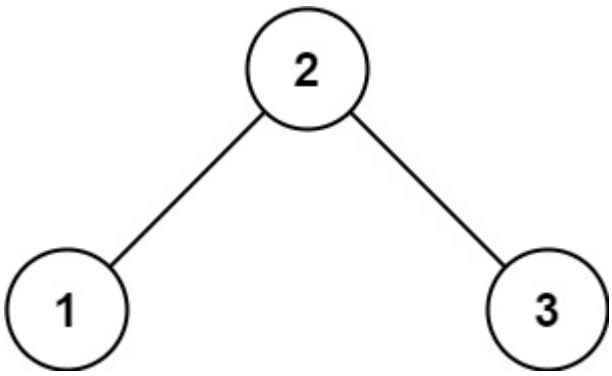
LeetCode	力扣	难度
98. Validate Binary Search Tree	98. 验证二叉搜索树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉搜索树，数据结构**

给你一个二叉树的根节点 `root`，判断其是否是一个有效的二叉搜索树。

示例 1：




```
输入：root = [2,1,3]
输出：true
```

基本思路

PS：这道题在《算法小抄》 的第 235 页。

初学者做这题很容易有误区：BST 不是左小右大么，那我只要检查 `root.val > root.left.val` 且 `root.val < root.right.val` 不就行了？

这样是不对的，因为 BST 左小右大的特性是指 `root.val` 要比左子树的所有节点都更大，要比右子树的所有节点都小，你只检查左右两个子节点当然是不够的。

正确解法是通过使用辅助函数，增加函数参数列表，在参数中携带额外信息，将这种约束传递给子树的所有节点，这也是二叉搜索树算法的一个小技巧吧。

- **详细题解：东哥带你刷二叉搜索树（基操篇）**

解法代码

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, null, null);
```

```
    }

    /* 限定以 root 为根的子树节点必须满足 max.val > root.val > min.val */
    boolean isValidBST(TreeNode root, TreeNode min, TreeNode max) {
        // base case
        if (root == null) return true;
        // 若 root.val 不符合 max 和 min 的限制，说明不是合法 BST
        if (min != null && root.val <= min.val) return false;
        if (max != null && root.val >= max.val) return false;
        // 限定左子树的最大值是 root.val，右子树的最小值是 root.val
        return isValidBST(root.left, min, root)
                && isValidBST(root.right, root, max);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 255. 验证前序遍历序列二叉搜索树 🟠
  - 450. 删除二叉搜索树中的节点 🟠
  - 700. 二叉搜索树中的搜索 🟢
  - 701. 二叉搜索树中的插入操作 🟠

1038. 把二叉搜索树转换为累加树

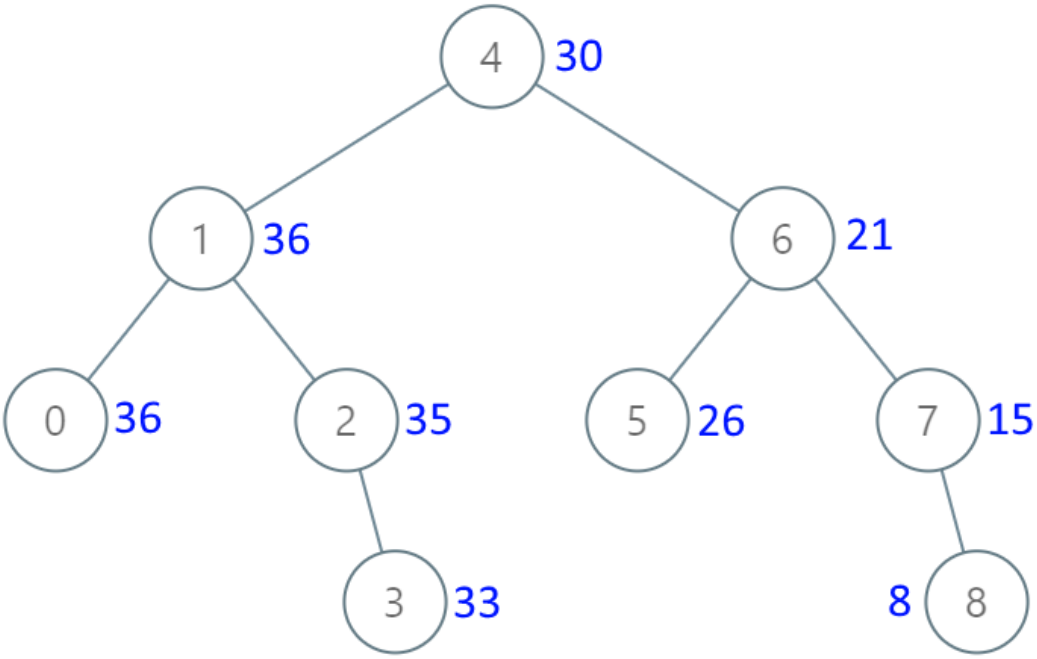
LeetCode	力扣	难度
1038. Binary Search Tree to Greater Sum Tree	1038. 把二叉搜索树转换为累加树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉搜索树**

给定一个二叉搜索树，请将它的每个节点的值替换成树中大于或者等于该节点值的所有节点值之和。

示例 1：




```
输入：[4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出：[30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

基本思路

和第 538. 把二叉搜索树转换为累加树 一模一样，这里就不多解释了。

- **详细题解：东哥带你刷二叉搜索树（特性篇）**

解法代码

```
class Solution {
    public TreeNode bstToGst(TreeNode root) {
        traverse(root);
        return root;
    }

    // 记录累加和
    int sum = 0;
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        traverse(root.right);
        // 维护累加和
        sum += root.val;
        // 将 BST 转化成累加树
        root.val = sum;
        traverse(root.left);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 230. 二叉搜索树中第K小的元素 🟠
  - 538. 把二叉搜索树转换为累加树 🟠
  - 剑指 Offer II 054. 所有大于等于节点的值之和 🟠

230. 二叉搜索树中第 K 小的元素

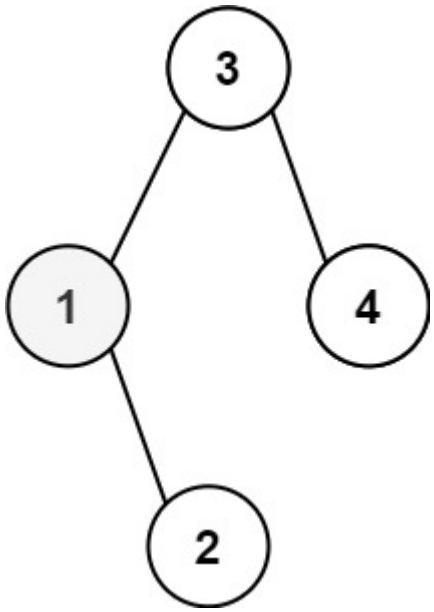
LeetCode	力扣	难度
230. Kth Smallest Element in a BST	230. 二叉搜索树中第K小的元素	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，数据结构

给定一个二叉搜索树的根节点 `root`，和一个整数 `k`，请你设计一个算法查找其中第 `k` 个最小元素（从 1 开始计数）。

示例 1：




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

基本思路

BST 的中序遍历结果是有序的（升序），所以用一个外部变量记录中序遍历结果第 `k` 个元素即是第 `k` 小的元素。

- 详细题解：东哥带你刷二叉搜索树（特性篇）

解法代码

```
class Solution {
    public int kthSmallest(TreeNode root, int k) {
        // 利用 BST 的中序遍历特性
```

```
        traverse(root, k);
        return res;
    }

    // 记录结果
    int res = 0;
    // 记录当前元素的排名
    int rank = 0;
    void traverse(TreeNode root, int k) {
        if (root == null) {
            return;
        }
        traverse(root.left, k);
        /* 中序遍历代码位置 */
        rank++;
        if (k == rank) {
            // 找到第 k 小的元素
            res = root.val;
            return;
        }
        /*****************/
        traverse(root.right, k);
    }
}
```

▶ 🦇 代码可视化动画（需安装 Chrome 插件）🦇

- **类似题目**：
  - 1038. 从二叉搜索树到更大和树 🟠
  - 538. 把二叉搜索树转换为累加树 🟠
  - 剑指 Offer 54. 二叉搜索树的第k大节点 🟢
  - 剑指 Offer II 054. 所有大于等于节点的值之和 🟠

538. 把二叉搜索树转换为累加树

LeetCode	力扣	难度
538. Convert BST to Greater Tree	538. 把二叉搜索树转换为累加树	🟠

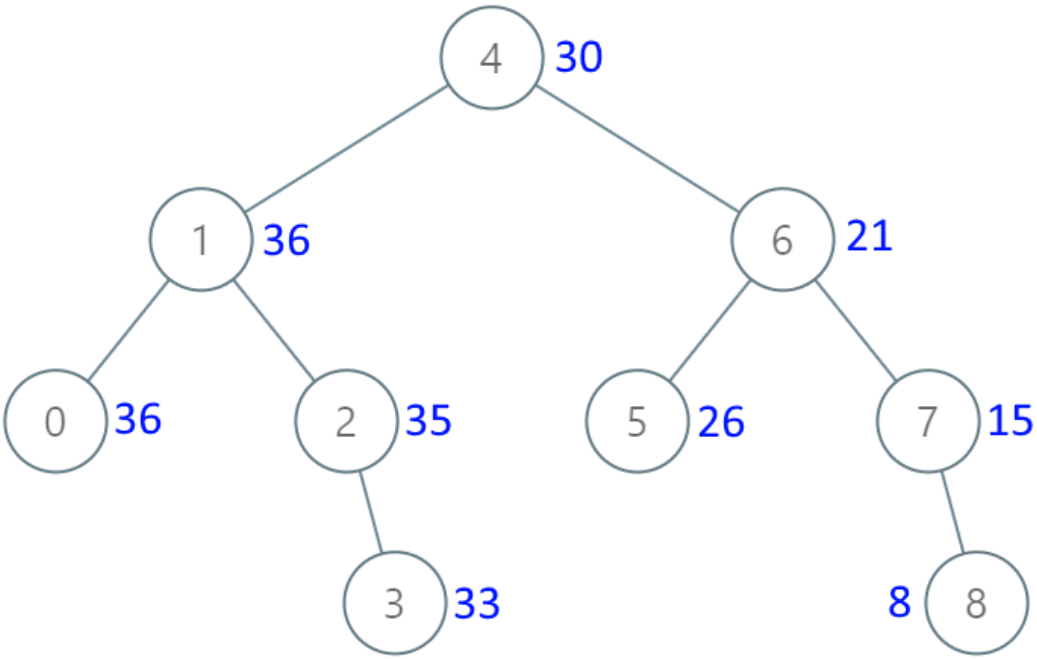
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，数据结构

给出二叉搜索树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater SumTree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

PS：本题和 1038. 把二叉搜索树转换为累加树 相同。

示例 1：




```
输入：[4,1,6,0,2,5,7,null,null,null,3,null,null,null,8]
输出：[30,36,21,36,35,26,15,null,null,null,33,null,null,null,8]
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

维护一个外部累加变量 `sum`，在遍历 BST 的过程中增加 `sum`，同时把 `sum` 赋值给 BST 中的每一个节点，就将 BST 转化成累加树了。

但是注意顺序，正常的中序遍历顺序是先左子树后右子树，这里需要反过来，先右子树后左子树。

- **详细题解：东哥带你刷二叉搜索树（特性篇）**

解法代码

```
class Solution {
    public TreeNode convertBST(TreeNode root) {
        traverse(root);
        return root;
    }

    // 记录累加和
    int sum = 0;
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        traverse(root.right);
        // 维护累加和
        sum += root.val;
        // 将 BST 转化成累加树
        root.val = sum;
        traverse(root.left);
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 1038. 从二叉搜索树到更大和树 🟠
  - 230. 二叉搜索树中第K小的元素 🟠
  - 剑指 Offer II 054. 所有大于等于节点的值之和 🟠

剑指 Offer 54. 二叉搜索树的第k大节点

LeetCode	力扣	难度
剑指Offer54. 二叉搜索树的第k大节点 LCOF	剑指Offer54. 二叉搜索树的第k大节点	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉搜索树**

给定一棵二叉搜索树，请找出其中第 k 大的节点的值。

示例 1:

```
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 4
```

基本思路

这道题很像 230. 二叉搜索树中第 K 小的元素，只不过 230 题求第 k 小的值，这里求第 k 大的值。

你可以看下我在写的 230 题的思路，本题也可以利用 BST 的中序遍历计算第 k 大的元素。只不过常规的中序遍历得到的顺序是从小到大的，而我们想得到从大到小的顺序。

这也很简单，只要把中序遍历框架反过来，先递归遍历右子树，再递归遍历左子树，即可获得降序结果：

```
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    traverse(root.right);
    // 中序位置
    traverse(root.left);
}
```

所以只要把 230 题的解法稍加修改即可解决本题。

解法代码

```
class Solution {
    public int kthLargest(TreeNode root, int k) {
        // 利用 BST 的中序遍历特性
        traverse(root, k);
        return res;
    }

    // 记录结果
    int res = 0;
    // 记录当前元素的排名
    int rank = 0;
    void traverse(TreeNode root, int k) {
        if (root == null) {
            return;
        }
        traverse(root.right, k);
        /* 中序遍历代码位置 */
        rank++;
        if (k == rank) {
            // 找到第 k 大的元素
            res = root.val;
            return;
        }
        /*****************/
        traverse(root.left, k);
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

剑指 Offer II 054. 所有大于等于节点的值之和

这道题和 538. 把二叉搜索树转换为累加树 相同。

501. 二叉搜索树中的众数

LeetCode	力扣	难度
501. Find Mode in Binary Search Tree	501. 二叉搜索树中的众数	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树

给定一个有相同值的二叉搜索树（BST），找出 BST 中的所有众数（出现频率最高的元素）。

例如给定 BST `[1,null,2,2]`，返回 `[2]`。

```
   1
    \
     2
    /
   2
```

提示：如果众数超过 1 个，不需考虑输出顺序

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

BST 的中序遍历有序，在中序遍历的位置做一些判断逻辑和操作有序数组差不多，很容易找出众数。

解法代码

```
class Solution {
    ArrayList<Integer> mode = new ArrayList<>();
    TreeNode prev = null;
    // 当前元素的重复次数
    int curCount = 0;
    // 全局的最长相同序列长度
    int maxCount = 0;

    public int[] findMode(TreeNode root) {
        // 执行中序遍历
        traverse(root);

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

```
        return res;
    }

    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        traverse(root.left);

        // 中序遍历位置
        if (prev == null) {
            // 初始化
            curCount = 1;
            maxCount = 1;
            mode.add(root.val);
        } else {
            if (root.val == prev.val) {
                // root.val 重复的情况
                curCount++;
                if (curCount == maxCount) {
                    // root.val 是众数
                    mode.add(root.val);
                } else if (curCount > maxCount) {
                    // 更新众数
                    mode.clear();
                    maxCount = curCount;
                    mode.add(root.val);
                }
            }

            if (root.val != prev.val) {
                // root.val 不重复的情况
                curCount = 1;
                if (curCount == maxCount) {
                    mode.add(root.val);
                }
            }
        }
        // 别忘了更新 prev
        prev = root;

        traverse(root.right);
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

530. 二叉搜索树的最小绝对差

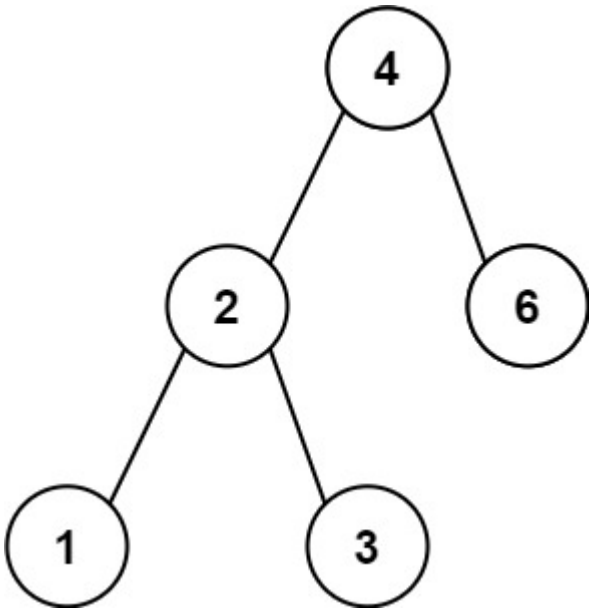
LeetCode	力扣	难度
530. Minimum Absolute Difference in BST	530. 二叉搜索树的最小绝对差	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树

给你一个二叉搜索树的根节点 `root`，返回**树中任意两不同节点值之间的最小差值**。差值是一个正数，其数值等于两值之差的绝对值。

示例 1：




```
输入：root = [4,2,6,1,3]
输出：1
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

记住，看到一棵 BST，你就把它理解是一个有序数组就行了，只是遍历方式不同罢了。所以这道题的思路就很简单，用中序遍历遍历一遍 BST 的所有节点得到有序结果，然后在遍历过程中计算最小差值即可。

解法代码

```
class Solution {
    public int getMinimumDifference(TreeNode root) {
```

```
        traverse(root);
        return res;
    }

    TreeNode prev = null;
    int res = Integer.MAX_VALUE;

    // 遍历函数
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }
        traverse(root.left);
        // 中序遍历位置
        if (prev != null) {
            res = Math.min(res, root.val - prev.val);
        }
        prev = root;
        traverse(root.right);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 783. 二叉搜索树节点最小距离 🟢

783. 二叉搜索树节点最小距离

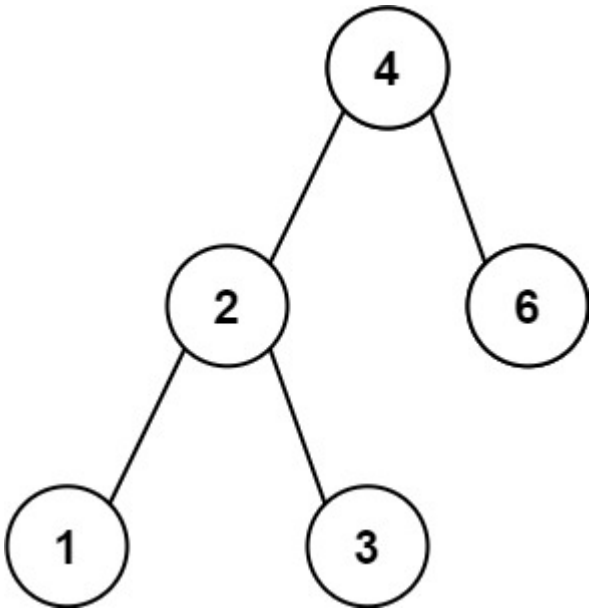
LeetCode	力扣	难度
783. Minimum Distance Between BST Nodes	783. 二叉搜索树节点最小距离	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树

给你一个二叉搜索树的根节点 `root`，返回**树中任意两不同节点值之间的最小差值**。差值是一个正数，其数值等于两值之差的绝对值。

示例 1：




```
输入：root = [4,2,6,1,3]
输出：1
```

基本思路

这题和 530. 二叉搜索树的最小绝对差 完全一样，可去那道题看下思路。

解法代码

```
class Solution {
    public int minDiffInBST(TreeNode root) {
        traverse(root);
        return res;
    }
```

```
        TreeNode prev = null;
        int res = Integer.MAX_VALUE;

        // 遍历函数
        void traverse(TreeNode root) {
            if (root == null) {
                return;
            }
            traverse(root.left);
            // 中序遍历位置
            if (prev != null) {
                res = Math.min(res, root.val - prev.val);
            }
            prev = root;
            traverse(root.right);
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

270. 最接近的二叉搜索树值

LeetCode	力扣	难度
270. Closest Binary Search Tree Value	270. 最接近的二叉搜索树值	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树vip

给定一个不为空的二叉搜索树和一个目标值 `target`，请在该二叉搜索树中找到最接近目标值 `target` 的数值。如果有多个答案，返回最小的那个。

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维。

我们遍历 BST，施展 700. 二叉搜索树中的搜索 的解法在 BST 中搜索 `target`，甭管能不能搜索到，一边搜索一边更新最接近 `target` 的值即可。

2023 Update：

原先这道题的题目中没有「如果有多个答案，返回最小的那个」这句话，这个要求应该是后来加上的。我一开始写的解法代码是这样的，大家请看：

```
class Solution {
    int res = 0;

    public int closestValue(TreeNode root, double target) {
        res = root.val;
        traverse(root, target);
        return res;
    }

    // 遍历函数，在 BST 中搜索 target
    void traverse(TreeNode root, double target) {
        if (root == null) {
            return;
        }
        // 一边搜索一边更新离 target 最近的值
        if (Math.abs(root.val - target) < Math.abs(res - target)) {
            res = root.val;
        }
        // 根据 target 和 root.val 的相对大小决定去左右子树搜索
        if (root.val < target) {
            traverse(root.right, target);
        } else {
            traverse(root.left, target);
        }
```

```
        }
    }
```

这段代码逻辑很简单，就是一边遍历 BST 一边寻找离 `target` 最近的值，不过题目改了之后肯定无法满足「如果有多个答案，返回最小的那个」这个要求的，但其实做到也不难，稍微修改下代码即可。

我想邀请你先思考一下，基于上述代码，如何用最少的修改，来满足「如果有多个答案，返回最小的那个」这个需求？你可以先不往下看，自己想想，动手试试。

下面，我带你分析一下这个需求的难点在哪里。比方说 BST 中节点的值为 `3, 2, 5`，`target = 4`，其实节点 `3, 5` 都距离 `4` 最近，但我们需要返回 `3`。

按照上述代码的逻辑，由于 `Math.abs(root.val - target) < Math.abs(res - target)` 这里使用的是小于号，也就是说当差值相等的时候不会进入更新 `res` 变量的 if 语句。

所以如果我们能确保 BST 中的节点从小到大执行这个更新 `res` 的 if 语句，那么最终的 `res` 记录的就是符合条件的最小的节点值。用上面的例子来说，就会返回 `3` 而不是 `5`。

那么如何才能确保从小到大执行这个更新 `res` 的 if 语句呢？提醒到这里你应该有些想法了吧，可以自己想一想，也许会有多种不同的思路，我把我的思路写到下面的代码中了。

解法代码

```
class Solution {
    int res = Integer.MAX_VALUE;

    public int closestValue(TreeNode root, double target) {
        traverse(root, target);
        return res;
    }

    // 遍历函数，在 BST 中搜索 target
    // 我们在中序位置写 if 判断逻辑，这样就可以从小到大执行，保证最终结果是值最小的
    void traverse(TreeNode root, double target) {
        if (root == null) {
            return;
        }
        // 根据 target 和 root.val 的相对大小决定去左右子树搜索
        if (root.val < target) {
            // 中序位置（左子树遍历结束，准备遍历右子树时）
            if (Math.abs(root.val - target) < Math.abs(res - target)) {
                res = root.val;
            }

            // 如果 target 比 root 大，那么 root 的左子树差值肯定更大，直接遍历右子
树
            traverse(root.right, target);
        } else {
            // 如果 target 比 root 小，那么 root 的右子树差值肯定更大，直接遍历左子
树
            traverse(root.left, target);
```

```
            // 中序位置（左子树遍历结束，准备遍历右子树时）
            if (Math.abs(root.val - target) < Math.abs(res - target)) {
                res = root.val;
            }
        }
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

285. 二叉搜索树中的中序后继

LeetCode	力扣	难度
285. Inorder Successor in BST	285. 二叉搜索树中的中序后继	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树，二叉树vip

给定一棵二叉搜索树和其中的一个节点 `p`，找到该节点在树中的中序后继。如果节点没有中序后继，请返回 `null`。

节点 `p` 的后继是值比 `p.val` 大的节点中键值最小的节点。

示例 1：

```
输入：root = [2,1,3], p = 1
输出：2
解释：这里 1 的中序后继是 2。请注意 p 和返回值都应是 TreeNode 类型。
```

示例 2：

```
输入：root = [5,3,6,2,4,null,null,1], p = 6
输出：null
解释：因为给出的节点没有中序后继，所以答案就返回 null 了。
```

基本思路

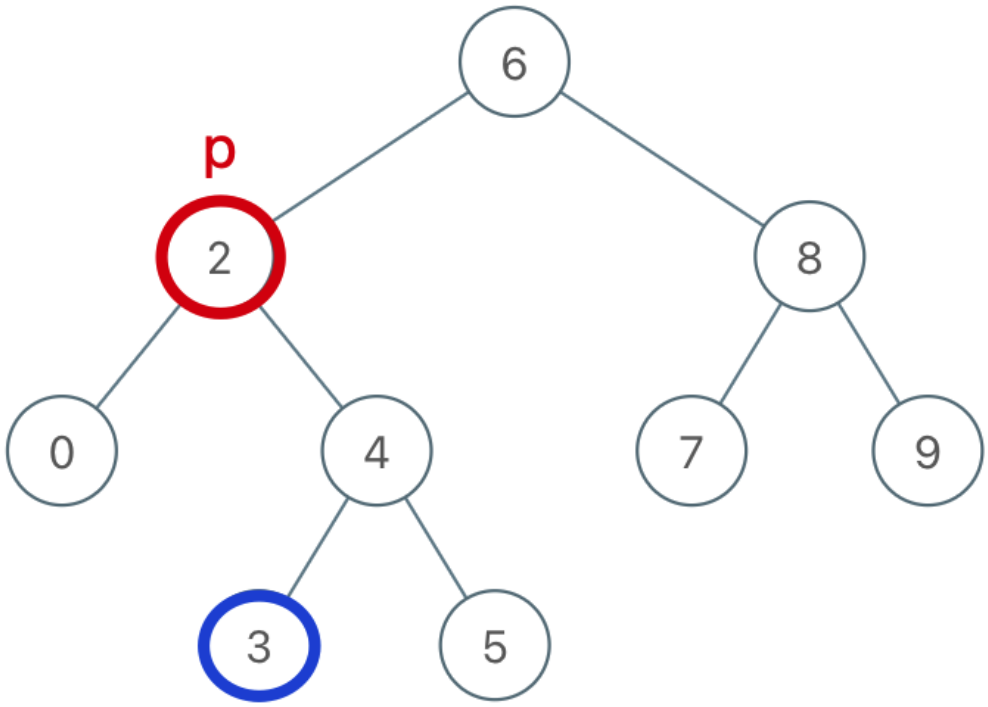
一个拍脑袋的解法就是去中序遍历所有节点获得有序结果，从而找到 successor，但这样时间复杂度是 O(N)，格局就低了。一般来说 BST 的算法都希望我们的复杂度保持在 O(logN)。

这题可以先参考 700. 二叉搜索树中的搜索 的解法，先写出在 BST 中搜索 `p.val` 的算法：

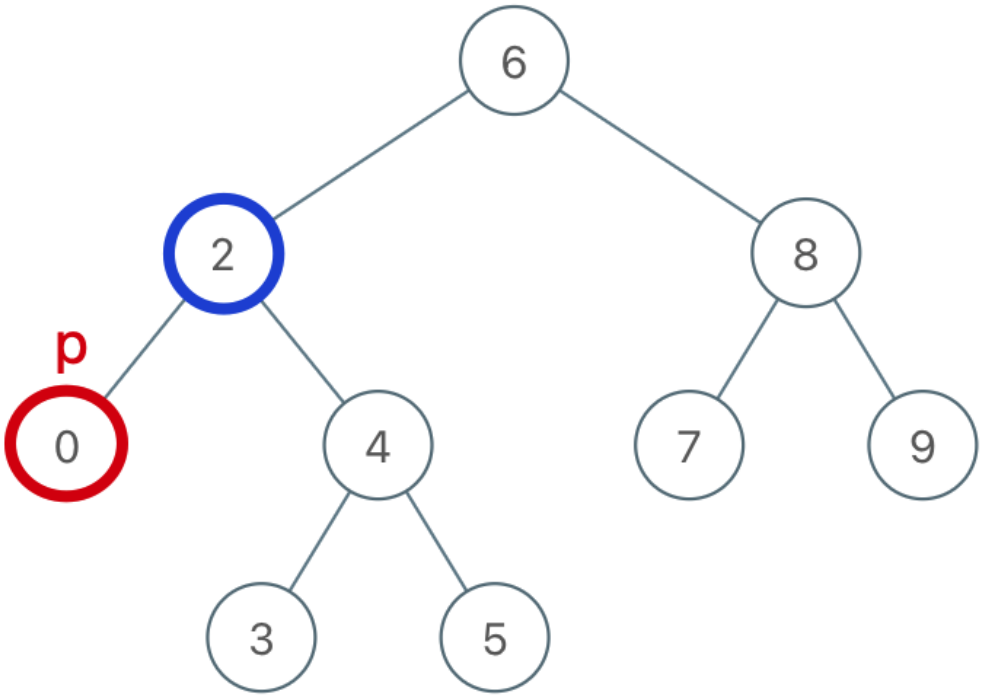
```
TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    if (root == null) {
        return null;
    }
    TreeNode target = null;

    if (root.val > p.val) {
```

```
            target = inorderSuccessor(root.left, p);
        }
        if (root.val < p.val) {
            target = inorderSuccessor(root.right, p);
        }
        if (root.val == p.val) {
            target = root;
        }

        return target;
    }
```

好，现在你搜索到了 `p` 这个节点，然后怎么去找它的 successor？显然比 `p` 大的最小元素就是 `p.right` 子树中最左侧的那个元素。



那如果 `p.right` 为空怎么办？此时比 `p` 大的最小元素就是 `p` 的父节点，我们要想办法通知 `p` 的父节点，让他老人家把自己作为结果返回。



依据以上思路，就可以写出解法代码。如果你对二叉树的递归有更深入的理解，还可以进一步精简代码（大佬们装逼专用），不过从复杂度都是一样的。

解法代码

```
class Solution {
    public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
        if (root == null) {
            return null;
        }

        TreeNode successor = null;

        if (root.val > p.val) {
            // 父节点收到 null 的话说明自己是 successor
            successor = inorderSuccessor(root.left, p);
            if (successor == null) {
                successor = root;
            }
        }
        if (root.val < p.val) {
            successor = inorderSuccessor(root.right, p);
        }

        if (root.val == p.val) {
            // 我是目标节点，我的 successor 要么是右子树的最小节点，要么是父节点
            successor = getMinNode(root.right);
        }

        return successor;
```

```
    }

    // BST 中最左侧的节点就是最小节点
    private TreeNode getMinNode(TreeNode p) {
        while (p != null && p.left != null) {
            p = p.left;
        }
        return p;
    }

    // 精简代码装逼版解法
    public TreeNode inorderSuccessor_Opt(TreeNode root, TreeNode p) {
        if (root == null) {
            return null;
        }

        if (root.val > p.val) {
            TreeNode successor = inorderSuccessor(root.left, p);
            return successor == null ? root : successor;
        }
        // root.val == p.val || root.val < p.val
        return inorderSuccessor(root.right, p);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 剑指 Offer II 053. 二叉搜索树中的中序后继 🟠

剑指 Offer II 053. 二叉搜索树中的中序后继

这道题和 285. 二叉搜索树中的中序后继 相同。

1373. 二叉搜索子树的最大键值和

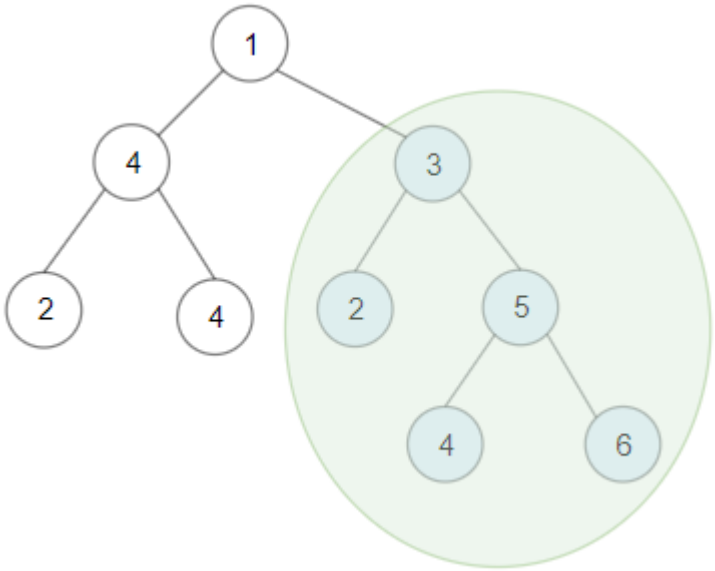
LeetCode	力扣	难度
1373. Maximum Sum BST in Binary Tree	1373. 二叉搜索子树的最大键值和	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树

给你一棵以 `root` 为根的**普通二叉树**，请你返回这棵普通二叉树中所有二叉搜索子树的最大节点和。

示例 1：




```
输入：root = [1,4,3,2,4,2,5,null,null,null,null,null,null,4,6]
输出：20
解释：以节点 3 为根的子树是二叉搜索树，且节点之和是所有二叉搜索子树中最大的。
```

基本思路

二叉树相关题目最核心的思路是明确当前节点需要做的事情是什么，那么我们想计算子树中 BST 的最大和，站在当前节点的视角，需要做什么呢？

1、我肯定得知道左右子树是不是合法的 BST，如果这俩儿子有一个不是 BST，以我为根的这棵树肯定不会是 BST，对吧。

2、如果左右子树都是合法的 BST，我得瞅瞅左右子树加上自己还是不是合法的 BST 了。因为按照 BST 的定义，当前节点的值应该大于左子树的最大值，小于右子树的最小值，否则就破坏了 BST 的性质。

3、因为题目要计算最大的节点之和，如果左右子树加上我自己还是一棵合法的 BST，也就是说以我为根的整棵树是一棵 BST，那我需要知道我们这棵 BST 的所有节点值之和是多少，方便和别的 BST 争个高下，对

吧。

简单说就是要知道以下具体信息：

1、左右子树是否是 BST。

2、左子树的最大值和右子树的最小值。

3、左右子树的节点值之和。

想要获得子树的信息，就要用到前文 手把手刷二叉树总结篇 说过的后序位置的妙用了。

我们定义一个 `findMaxMinSum` 函数，`findMaxMinSum(root)` 返回一个大小为 4 的 int 数组，我们暂且称它为 `res`，其中：

`res[0]` 记录以 `root` 为根的二叉树是否是 BST，若为 1 则说明是 BST，若为 0 则说明不是 BST；

`res[1]` 记录以 `root` 为根的二叉树所有节点中的最小值；

`res[2]` 记录以 `root` 为根的二叉树所有节点中的最大值；

`res[3]` 记录以 `root` 为根的二叉树所有节点值之和。

按照上述思路理解代码。

- **详细题解：后序遍历的妙用**

解法代码

```
class Solution {
    // 全局变量，记录 BST 最大节点之和
    int maxSum = 0;

    /* 主函数 */
    public int maxSumBST(TreeNode root) {
        traverse(root);
        return maxSum;
    }

    int[] findMaxMinSum(TreeNode root) {
        // base case
        if (root == null) {
            return new int[] {
                    1, Integer.MAX_VALUE, Integer.MIN_VALUE, 0
            };
        }

        // 递归计算左右子树
        int[] left = findMaxMinSum(root.left);
        int[] right = findMaxMinSum(root.right);

        /*******后序遍历位置*******/
        int[] res = new int[4];
```

```
            // 这个 if 在判断以 root 为根的二叉树是不是 BST
            if (left[0] == 1 && right[0] == 1 &&
                    root.val > left[2] && root.val < right[1]) {
                // 以 root 为根的二叉树是 BST
                res[0] = 1;
                // 计算以 root 为根的这棵 BST 的最小值
                res[1] = Math.min(left[1], root.val);
                // 计算以 root 为根的这棵 BST 的最大值
                res[2] = Math.max(right[2], root.val);
                // 计算以 root 为根的这棵 BST 所有节点之和
                res[3] = left[3] + right[3] + root.val;
                // 更新全局变量
                maxSum = Math.max(maxSum, res[3]);
            } else {
                // 以 root 为根的二叉树不是 BST
                res[0] = 0;
                // 其他的值都没必要计算了，因为用不到
            }
            /*************************/

            return res;
        }
    }
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

剑指 Offer 33. 二叉搜索树的后序遍历序列

LeetCode	力扣	难度
剑指Offer33. 二叉搜索树的后序遍历序列 LCOF	剑指Offer33. 二叉搜索树的后序遍历序列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉搜索树，二叉树**

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 `true`，否则返回 `false`。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```
     5
    / \
   2   6
  / \
 1   3
```

示例 1：

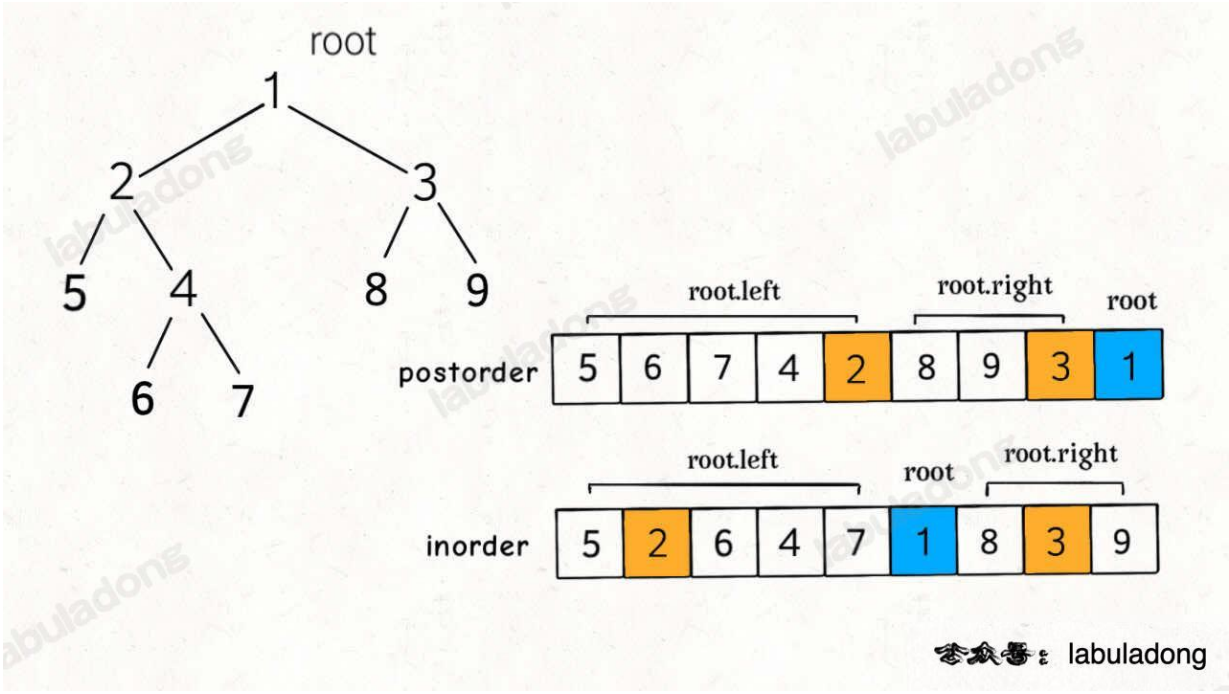
```
输入：[1,6,3,2,5]
输出：false
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维模式。

建议你先看一下 东哥手把手带你刷二叉树（构造篇） 了解二叉树前中后序遍历结果的特点，再看一下 手把手带你刷二叉搜索树（第二期） 了解 BST 的特性和基本操作方法，再来看这道题，解法就呼之欲出了。

后序遍历结果的特点是：左边一部分是左子树，右边一部分是右子树，最后一个元素是根节点。



而 BST 的特点是左子树元素都小于根节点，右子树元素都大于根节点。

所以结合上述规律，就可以得到解决本题的思路：

1、先找到根节点元素

2、根据根节点元素找到左子树元素，递归检查左子树是否是 BST

3、根据根节点元素找到右子树元素，递归检查右子树是否是 BST

解法代码

```
class Solution {
    public boolean verifyPostorder(int[] postorder) {
        return check(postorder, 0, postorder.length - 1);
    }

    // 定义：检查 postorder[i..j] 是否是一个合法的 BST
    boolean check(int[] postorder, int i, int j) {
        if (i >= j) {
            return true;
        }
        // 根节点的值是后序遍历结果的最后一个元素
        int root = postorder[j];

        // postorder[i..left) 是左子树，应该都小于 root
        int left = i;
        while (left < j && postorder[left] < root) {
            left++;
        }

        // postorder[left..j) 是右子树，应该都大于 root
        int right = left;
```

```
        while (right < j && postorder[right] > root) {
            right++;
        }
        if (right != j) {
            return false;
        }
        // 递归检查左子树 [i..left) 和右子树 [left..j) 也符合 BST 的性质
        return check(postorder, i, left - 1) && check(postorder, left, j -
1);
    }

}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

797. 所有可能的路径

LeetCode	力扣	难度
797. All Paths From Source to Target	797. 所有可能的路径	🟠

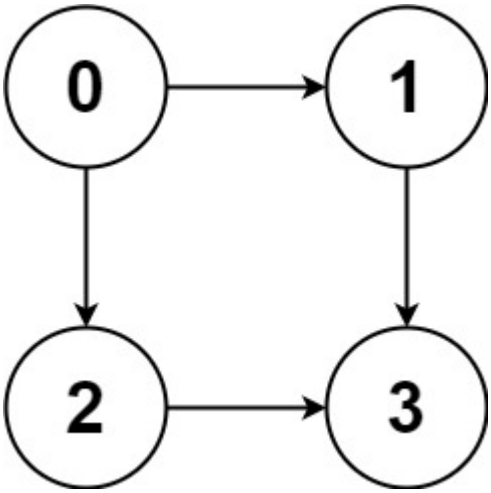
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：图论算法，数据结构**

给你一个有 `n` 个节点的**有向无环图（DAG）**，`graph[i]` 记录着第节点 `i` 能够到达的节点。

请你找出所有从节点 `0` 到节点 `n-1` 的路径并输出（**不要求按特定顺序**）。

示例 1：




```
输入：graph = [[1,2],[3],[3],[]]
输出：[[0,1,3],[0,2,3]]
解释：有两条路径 0 -> 1 -> 3 和 0 -> 2 -> 3
```

基本思路

> 本文有视频版：图论基础及遍历算法

解法很简单，以 `0` 为起点遍历图，同时记录遍历过的路径，当遍历到终点时将路径记录下来即可。

既然输入的图是无环的，我们就不需要 `visited` 数组辅助了，可以直接套用 图的遍历框架。

- **详细题解：图论基础及遍历算法**

解法代码

```
class Solution {
    // 记录所有路径
```

```
    List<List<Integer>> res = new LinkedList<>();

    public List<List<Integer>> allPathsSourceTarget(int[][] graph) {
        LinkedList<Integer> path = new LinkedList<>();
        traverse(graph, 0, path);
        return res;
    }

    /* 图的遍历框架 */
    void traverse(int[][] graph, int s, LinkedList<Integer> path) {

        // 添加节点 s 到路径
        path.addLast(s);

        int n = graph.length;
        if (s == n - 1) {
            // 到达终点
            res.add(new LinkedList<>(path));
            path.removeLast();
            return;
        }

        // 递归每个相邻节点
        for (int v : graph[s]) {
            traverse(graph, v, path);
        }

        // 从路径移出节点 s
        path.removeLast();
    }
}
```

▶ 🎮 代码可视化动画（需安装 Chrome 插件）🎮

- 类似题目：
  - 剑指 Offer II 110. 所有路径 🟠

785. 判断二分图

LeetCode	力扣	难度
785. Is Graph Bipartite?	785. 判断二分图	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二分图，图论算法**

存在一个 **无向图**，图中有 `n` 个节点。其中每个节点都有一个介于 `0` 到 `n - 1` 之间的唯一编号。给你一个二维数组 `graph`，其中 `graph[u]` 是一个节点数组，由节点 `u` 的邻接节点组成。形式上，对于 `graph[u]` 中的每个 `v`，都存在一条位于节点 `u` 和节点 `v` 之间的无向边。该无向图同时具有以下属性：

1、不存在自环（`graph[u]` 不包含 `u`）。

2、不存在平行边（`graph[u]` 不包含重复值）。

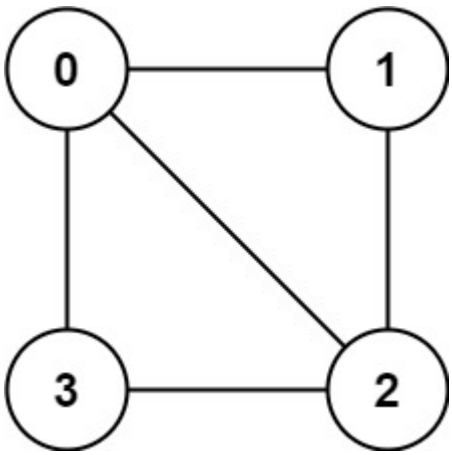
3、如果 `v` 在 `graph[u]` 内，那么 `u` 也应该在 `graph[v]` 内（该图是无向图）

4、这个图可能不是连通图，也就是说两个节点 `u` 和 `v` 之间可能不存在一条连通彼此的路径。

二分图 定义：如果能将一个图的节点集合分割成两个独立的子集 `A` 和 `B`，并使图中的每一条边的两个节点一个来自 `A` 集合，一个来自 `B` 集合，就将这个图称为 **二分图**。

如果图是二分图，返回 `true`；否则，返回 `false`。

示例 1：




```
输入：graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
输出：false
解释：不能将节点分割成两个独立的子集，以使每条边都连通一个子集中的一个节点与另一个子集中的一个节点。
```

基本思路

> 本文有视频版：二分图判定算法及应用

二分图判定问题等同于图论的「双色问题」：

给你一幅「图」，请你用两种颜色将图中的所有顶点着色，且使得任意一条边的两个端点的颜色都不相同，你能做到吗？

如果能成功对整幅图染色，则说明这是一幅二分图，否则就不是二分图。

思路也很简单，遍历一遍图，一边遍历一边染色，看看能不能用两种颜色给所有节点染色，且相邻节点的颜色都不相同。

- **详细题解：二分图判定算法**

解法代码

```
class Solution {

    // 记录图是否符合二分图性质
    private boolean ok = true;
    // 记录图中节点的颜色，false 和 true 代表两种不同颜色
    private boolean[] color;
    // 记录图中节点是否被访问过
    private boolean[] visited;

    // 主函数，输入邻接表，判断是否是二分图
    public boolean isBipartite(int[][] graph) {
        int n = graph.length;
        color = new boolean[n];
        visited = new boolean[n];
        // 因为图不一定是联通的，可能存在多个子图
        // 所以要把每个节点都作为起点进行一次遍历
        // 如果发现任何一个子图不是二分图，整幅图都不算二分图
        for (int v = 0; v < n; v++) {
            if (!visited[v]) {
                traverse(graph, v);
            }
        }
        return ok;
    }

    // DFS 遍历框架
    private void traverse(int[][] graph, int v) {
        // 如果已经确定不是二分图了，就不用浪费时间再递归遍历了
        if (!ok) return;

        visited[v] = true;
        for (int w : graph[v]) {
            if (!visited[w]) {
                // 相邻节点 w 没有被访问过
                // 那么应该给节点 w 涂上和节点 v 不同的颜色
                color[w] = !color[v];
```

```
                // 继续遍历 w
                traverse(graph, w);
            } else {
                // 相邻节点 w 已经被访问过
                // 根据 v 和 w 的颜色判断是否是二分图
                if (color[w] == color[v]) {
                    // 若相同，则此图不是二分图
                    ok = false;
                }
            }
        }
    }

}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 886. 可能的二分法 🟠
  - 剑指 Offer II 106. 二分图 🟠

886. 可能的二分法

LeetCode	力扣	难度
886. Possible Bipartition	886. 可能的二分法	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二分图，图论算法**

给定一组 `N` 人（编号为 `1, 2, ..., N`），我们想把每个人分进**任意**大小的两组。

每个人都可能不喜欢其他人，那么他们不应该属于同一组。

形式上，如果 `dislikes[i] = [a, b]`，表示不允许将编号为 `a` 和 `b` 的人归入同一组。

当可以用这种方法将所有人分进两组时，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]
```

基本思路

> 本文有视频版：二分图判定算法及应用

和 785. 判断二分图 一样，其实这题考察的就是二分图的判定：

如果你把每个人看做图中的节点，相互讨厌的关系看做图中的边，那么 `dislikes` 数组就可以构成一幅图；

又因为题目说互相讨厌的人不能放在同一组里，相当于图中的所有相邻节点都要放进两个不同的组；

那就回到了「双色问题」，如果能够用两种颜色着色所有节点，且相邻节点颜色都不同，那么你按照颜色把这些节点分成两组不就行了嘛。

所以解法就出来了，我们把 `dislikes` 构造成一幅图，然后执行二分图的判定算法即可。

- **详细题解：二分图判定算法**

解法代码

```
class Solution {

    private boolean ok = true;
    private boolean[] color;
    private boolean[] visited;
```

```
    public boolean possibleBipartition(int n, int[][] dislikes) {
        // 图节点编号从 1 开始
        color = new boolean[n + 1];
        visited = new boolean[n + 1];
        // 转化成邻接表表示图结构
        List<Integer>[] graph = buildGraph(n, dislikes);

        for (int v = 1; v <= n; v++) {
            if (!visited[v]) {
                traverse(graph, v);
            }
        }
        return ok;
    }

    // 建图函数
    private List<Integer>[] buildGraph(int n, int[][] dislikes) {
        // 图节点编号为 1...n
        List<Integer>[] graph = new LinkedList[n + 1];
        for (int i = 1; i <= n; i++) {
            graph[i] = new LinkedList<>();
        }
        for (int[] edge : dislikes) {
            int v = edge[1];
            int w = edge[0];
            // 「无向图」相当于「双向图」
            // v -> w
            graph[v].add(w);
            // w -> v
            graph[w].add(v);
        }
        return graph;
    }

    // 和之前判定二分图的 traverse 函数完全相同
    private void traverse(List<Integer>[] graph, int v) {
        if (!ok) return;
        visited[v] = true;
        for (int w : graph[v]) {
            if (!visited[w]) {
                color[w] = !color[v];
                traverse(graph, w);
            } else {
                if (color[w] == color[v]) {
                    ok = false;
                }
            }
        }
    }

}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目：**
  - 785. 判断二分图 🟠
  - 剑指 Offer II 106. 二分图 🟠

207. 课程表

LeetCode	力扣	难度
207. Course Schedule	207. 课程表	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：图论算法，数据结构，环检测**

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

在选修某些课程之前需要一些先修课程。先修课程按数组 `prerequisites` 给出，其中 `prerequisites[i] = [ai, bi]`，表示如果要学习课程 `ai` 则**必须**先学习课程 `bi`。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true`；否则，返回 `false`。

示例 1：

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

示例 2：

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

基本思路

> 本文有视频版：拓扑排序详解及应用

只要会遍历图结构，就可以判断环了。

利用布尔数组 `onPath`，如果遍历过程中发现下一个即将遍历的节点已经被标记为 true，说明遇到了环（可以联想贪吃蛇咬到自己的场景）。

我给出 DFS 遍历的解法，其实本题也可以用 BFS 算法解决，稍微有些技巧，可以看详细题解。

- **详细题解：环检测及拓扑排序算法**

解法代码

```
class Solution {
    // 记录一次 traverse 递归经过的节点
    boolean[] onPath;
    // 记录遍历过的节点，防止走回头路
    boolean[] visited;
    // 记录图中是否有环
    boolean hasCycle = false;

    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);

        visited = new boolean[numCourses];
        onPath = new boolean[numCourses];

        for (int i = 0; i < numCourses; i++) {
            // 遍历图中的所有节点
            traverse(graph, i);
        }
        // 只要没有循环依赖可以完成所有课程
        return !hasCycle;
    }

    void traverse(List<Integer>[] graph, int s) {
        if (onPath[s]) {
            // 出现环
            hasCycle = true;
        }

        if (visited[s] || hasCycle) {
            // 如果已经找到了环，也不用再遍历了
            return;
        }
        // 前序遍历代码位置
        visited[s] = true;
        onPath[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
        // 后序遍历代码位置
        onPath[s] = false;
    }

    List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
        // 图中共有 numCourses 个节点
        List<Integer>[] graph = new LinkedList[numCourses];
        for (int i = 0; i < numCourses; i++) {
            graph[i] = new LinkedList<>();
        }
        for (int[] edge : prerequisites) {
            int from = edge[1];
            int to = edge[0];
            // 修完课程 from 才能修课程 to
            // 在图中添加一条从 from 指向 to 的有向边
```

```
                graph[from].add(to);
            }
            return graph;
        }
    }
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 210. 课程表 II 🟠
  - 剑指 Offer II 113. 课程顺序 🟠

210. 课程表 II

LeetCode	力扣	难度
210. Course Schedule II	210. 课程表 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：图论算法，拓扑排序，数据结构**

现在你总共有 `numCourses` 门课需要选，记为 `0` 到 `numCourses - 1`。给你一个数组 `prerequisites`，其中 `prerequisites[i] = [ai, bi]`，表示在选修课程 `ai` 前**必须**先选修 `bi`。

返回你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序，你只要返回**任意一种**就可以了。如果不可能完成所有课程，返回**一个空数组**。

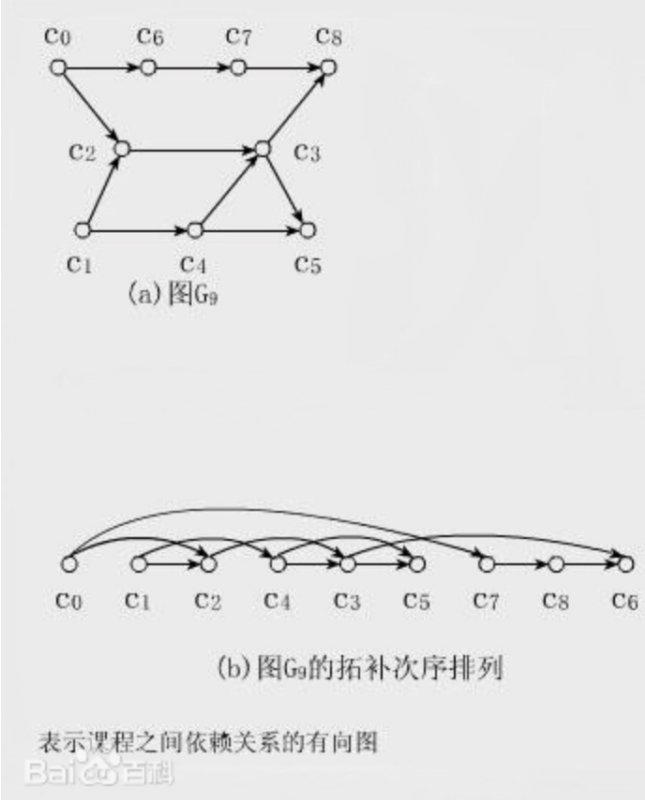
示例 1：

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

基本思路

> 本文有视频版：拓扑排序详解及应用

直观地说，拓扑排序就是让你把一幅无环图「拉平」，而且这个「拉平」的图里面，所有箭头方向都是一致的：



在进行拓扑排序之前，首先要确保图中无环，这就依赖 207. 课程表 中讲的环检测算法。

拓扑排序可以使用 DFS 算法，图的后序遍历结果进行反转就是拓扑排序结果。

另外，也可以用 BFS 算法借助每个节点的入度进行拓扑排序，这里就用 BFS 算法来解决。

DFS 解法和算法执行过程详解请看详细题解。

- **详细题解：环检测及拓扑排序算法**

解法代码

```
class Solution {
    // 主函数
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        // 建图，和环检测算法相同
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);
        // 计算入度，和环检测算法相同
        int[] indegree = new int[numCourses];
        for (int[] edge : prerequisites) {
            int from = edge[1], to = edge[0];
            indegree[to]++;
        }

        // 根据入度初始化队列中的节点，和环检测算法相同
        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                q.offer(i);
            }
        }
```

```
        // 记录拓扑排序结果
        int[] res = new int[numCourses];
        // 记录遍历节点的顺序（索引）
        int count = 0;
        // 开始执行 BFS 算法
        while (!q.isEmpty()) {
            int cur = q.poll();
            // 弹出节点的顺序即为拓扑排序结果
            res[count] = cur;
            count++;
            for (int next : graph[cur]) {
                indegree[next]--;
                if (indegree[next] == 0) {
                    q.offer(next);
                }
            }
        }

        if (count != numCourses) {
            // 存在环，拓扑排序不存在
            return new int[]{};
        }

        return res;
    }

    // 建图函数
    List<Integer>[] buildGraph(int numCourses, int[][] prerequisites) {
        // 图中共有 numCourses 个节点
        List<Integer>[] graph = new LinkedList[numCourses];
        for (int i = 0; i < numCourses; i++) {
            graph[i] = new LinkedList<>();
        }
        for (int[] edge : prerequisites) {
            int from = edge[1], to = edge[0];
            // 修完课程 from 才能修课程 to
            // 在图中添加一条从 from 指向 to 的有向边
            graph[from].add(to);
        }
        return graph;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 207. 课程表 🟠
  - 剑指 Offer II 113. 课程顺序 🟠

剑指 Offer II 113. 课程顺序

这道题和 210. 课程表 II 相同。

130. 被围绕的区域

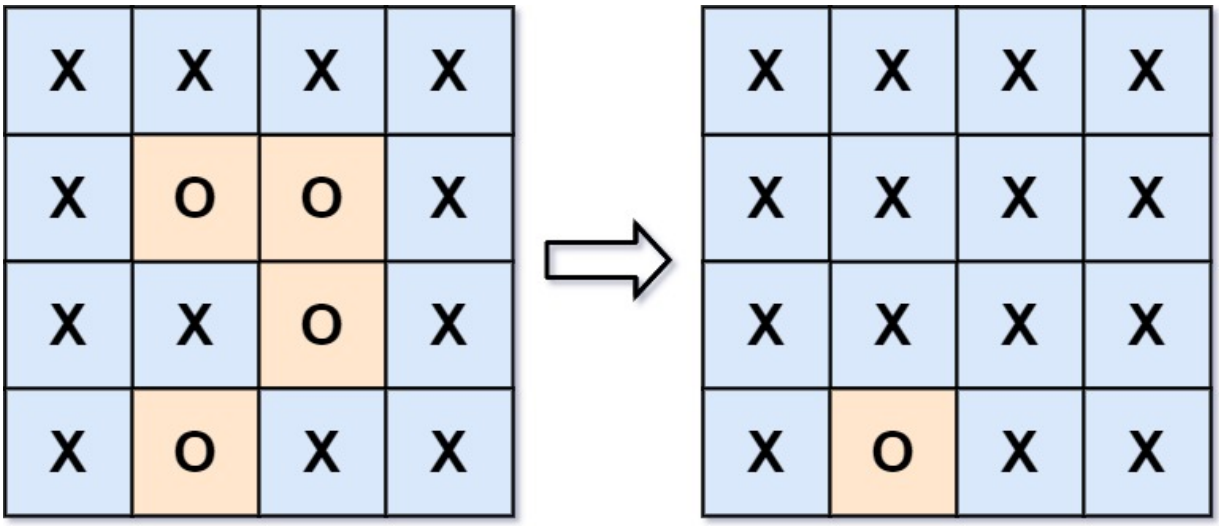
LeetCode	力扣	难度
130. Surrounded Regions	130. 被围绕的区域	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：DFS 算法，并查集算法

给你一个 `m x n` 的矩阵 `board`，由若干字符 `'X'` 和 `'O'` 组成，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 1：



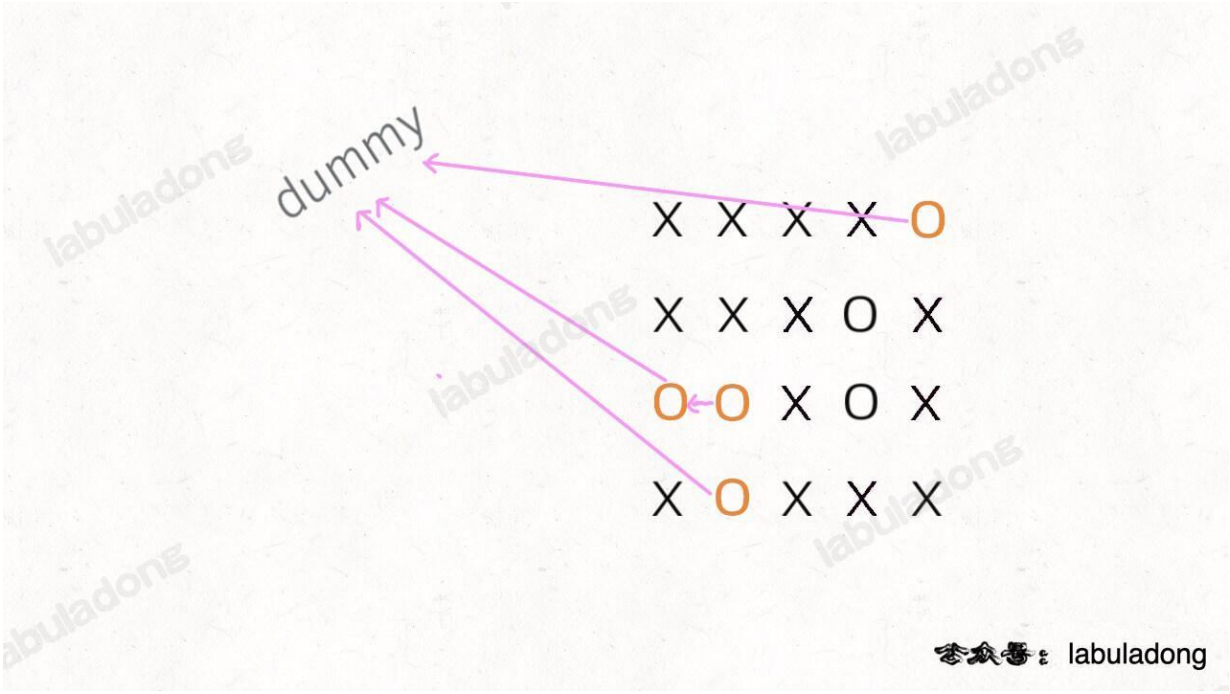

```
输入：board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],
["X","O","X","X"]]
输出：[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],
["X","O","X","X"]]
解释：被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。
```

基本思路

PS：这道题在《算法小抄》 的第 396 页。

这题和 1254. 统计封闭岛屿的数目 几乎完全一样，常规做法就是 DFS，那我们这里就讲一个另类的解法，看看并查集算法如何解决这道题。

我们可以把所有靠边的 `O` 和一个虚拟节点 `dummy` 进行连通：



然后再遍历整个 board，那些和 dummy 不连通的 O 就是被围绕的区域，需要被替换。

- **详细题解：并查集（Union-Find）算法**

解法代码

```
class Solution {
    public void solve(char[][] board) {
        if (board.length == 0) return;

        int m = board.length;
        int n = board[0].length;
        // 给 dummy 留一个额外位置
        UF uf = new UF(m * n + 1);
        int dummy = m * n;
        // 将首列和末列的 O 与 dummy 连通
        for (int i = 0; i < m; i++) {
            if (board[i][0] == 'O')
                uf.union(i * n, dummy);
            if (board[i][n - 1] == 'O')
                uf.union(i * n + n - 1, dummy);
        }
        // 将首行和末行的 O 与 dummy 连通
        for (int j = 0; j < n; j++) {
            if (board[0][j] == 'O')
                uf.union(j, dummy);
            if (board[m - 1][j] == 'O')
                uf.union(n * (m - 1) + j, dummy);
        }
        // 方向数组 d 是上下左右搜索的常用手法
        int[][] d = new int[][]{{1, 0}, {0, 1}, {0, -1}, {-1, 0}};
        for (int i = 1; i < m - 1; i++)
            for (int j = 1; j < n - 1; j++)
```

```
                if (board[i][j] == 'O')
                    // 将此 O 与上下左右的 O 连通
                    for (int k = 0; k < 4; k++) {
                        int x = i + d[k][0];
                        int y = j + d[k][1];
                        if (board[x][y] == 'O')
                            uf.union(x * n + y, i * n + j);
                    }
            // 所有不和 dummy 连通的 O，都要被替换
            for (int i = 1; i < m - 1; i++)
                for (int j = 1; j < n - 1; j++)
                    if (!uf.connected(dummy, i * n + j))
                        board[i][j] = 'X';
        }
    }

    class UF {
        // 记录连通分量个数
        private int count;
        // 存储若干棵树
        private int[] parent;
        // 记录树的“重量”
        private int[] size;

        public UF(int n) {
            this.count = n;
            parent = new int[n];
            size = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                size[i] = 1;
            }
        }

        /* 将 p 和 q 连通 */
        public void union(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            if (rootP == rootQ)
                return;

            // 小树接到大树下面，较平衡
            if (size[rootP] > size[rootQ]) {
                parent[rootQ] = rootP;
                size[rootP] += size[rootQ];
            } else {
                parent[rootP] = rootQ;
                size[rootQ] += size[rootP];
            }
            count--;
        }

        /* 判断 p 和 q 是否互相连通 */
        public boolean connected(int p, int q) {
```

```
        int rootP = find(p);
        int rootQ = find(q);
        // 处于同一棵树上的节点，相互连通
        return rootP == rootQ;
    }

    /* 返回节点 x 的根节点 */
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    public int count() {
        return count;
    }
}
```

- **类似题目**：
  - 323. 无向图中连通分量的数目 🟠
  - 990. 等式方程的可满足性 🟠

990. 等式方程的可满足性

LeetCode	力扣	难度
990. Satisfiability of Equality Equations	990. 等式方程的可满足性	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：并查集算法**

给定一个由表示变量之间关系的字符串方程组成的数组，每个字符串方程 `equations[i]` 的长度为 `4`，并采用两种不同的形式之一：`"a==b"` 或 `"a!=b"`。在这里，a 和 b 是小写字母（不一定不同），表示单字母变量名。

只有当可以将整数分配给变量名，以便满足所有给定的方程时才返回 `true`，否则返回 `false`。

示例 1：

```
输入：["a==b","b!=a"]
输出：false
解释：如果我们指定，a = 1 且 b = 1，那么可以满足第一个方程，但无法满足第二个方程。没有办法分配变量同时满足这两个方程。
```

基本思路

PS：这道题在《算法小抄》 的第 396 页。

本题是前文 Union Find 并查集算法 的应用。

解题核心思想是，**将 `equations` 中的算式根据 `==` 和 `!=` 分成两部分，先处理 `==` 算式，使得他们通过相等关系各自勾结成门派（连通分量）；然后处理 `!=` 算式，检查不等关系是否破坏了相等关系的连通性。**

- **详细题解：并查集（Union-Find）算法**

解法代码

```
class Solution {
    public boolean equationsPossible(String[] equations) {
        // 26 个英文字母
        UF uf = new UF(26);
        // 先让相等的字母形成连通分量
        for (String eq : equations) {
            if (eq.charAt(1) == '=') {
                char x = eq.charAt(0);
                char y = eq.charAt(3);
                uf.union(x - 'a', y - 'a');
            }
```

```
        }
        // 检查不等关系是否打破相等关系的连通性
        for (String eq : equations) {
            if (eq.charAt(1) == '!') {
                char x = eq.charAt(0);
                char y = eq.charAt(3);
                // 如果相等关系成立，就是逻辑冲突
                if (uf.connected(x - 'a', y - 'a'))
                    return false;
            }
        }
        return true;
    }
}

class UF {
    // 记录连通分量个数
    private int count;
    // 存储若干棵树
    private int[] parent;
    // 记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    /* 将 p 和 q 连通 */
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        count--;
    }

    /* 判断 p 和 q 是否互相连通 */
    public boolean connected(int p, int q) {
        int rootP = find(p);
```

```
        int rootQ = find(q);
        // 处于同一棵树上的节点，相互连通
        return rootP == rootQ;
    }

    /* 返回节点 x 的根节点 */
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    public int count() {
        return count;
    }
}
```

- **类似题目**：
  - 130. 被围绕的区域 🟠
  - 323. 无向图中连通分量的数目 🟠

765. 情侣牵手

LeetCode	力扣	难度
765. Couples Holding Hands	765. 情侣牵手	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：并查集算法**

n 对情侣坐在连续排列的 2n 个座位上，想要牵到对方的手。

人和座位由一个整数数组 row 表示，其中 row[i] 是坐在第 i 个座位上的人的**ID**。情侣们按顺序编号，第一对是 (0, 1)，第二对是 (2, 3)，以此类推，最后一对是 (2n-2, 2n-1)。

返回 **最少交换座位的次数，以便每对情侣可以并肩坐在一起**。每次交换可选择任意两人，让他们站起来交换座位。

示例 1:

```
输入: row = [0,2,1,3]
输出: 1
解释: 只需要交换 row[1]和 row[2]的位置即可。
```

基本思路

这道题的思路比较巧妙。

首先，每个人都有一个 person_id，person_id 相邻的两个人被认为是一对情侣，即 (0, 1), (2, 3) ...

那如果我想给「每对情侣」给一个 couple_id，从 0 到 n - 1，如何分配？

可以把 person_id / 2 作为 couple_id，因为每对情侣的 person_id 相邻，除以二向下取整之后结果相同：

```
0 / 2 == 1 / 2 == 0, 2 / 2 == 3 / 2 == 1 ...
```

在理想情况下，每对情侣坐在一起，结合 Union Find 算法 将 couple_id 相同的两个节点相连，最终应该有 n 个联通分量。

但实际上并不是每对情侣都坐在正确的位置上，所以会有多对情侣进入了同一个连通分量，导致连通分量的数量变少，n 和连通分量只差就是需要交换的次数，可以自行画图理解。

解法代码

```
class Solution {
    public int minSwapsCouples(int[] row) {
        int n = row.length;
        UF uf = new UF(n);
        for (int i = 0; i < n; i += 2) {
            // 将两人的 couple_id 进行连接
            uf.union(row[i] / 2, row[i + 1] / 2);
        }
        // 和连通分量的差即为需要交换的次数
        return n - uf.count();
    }
}

// 并查集算法模板
class UF {
    // 记录连通分量个数
    private int count;
    // 存储若干棵树
    private int[] parent;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    /* 将 p 和 q 连通 */
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        parent[rootQ] = rootP;
        count--;
    }

    /* 判断 p 和 q 是否互相连通 */
    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        // 处于同一棵树上的节点，相互连通
        return rootP == rootQ;
    }

    /* 返回节点 x 的根节点 */
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
```

```
            }
            return x;
        }

        public int count() {
            return count;
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

947. 移除最多的同行或同列石头

LeetCode	力扣	难度
947. Most Stones Removed with Same Row or Column	947. 移除最多的同行或同列石头	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维矩阵，并查集算法**

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石头。

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 i 块石头的位置，返回 **可以移除的石子** 的最大数量。

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。
```

基本思路

只要你看过前文 Union-Find 算法详解，这道题就很简单。

我们可以把每一块石头看做图中的一个节点，同一行的节点之间互相连通，同一列的节点之间也互相连通，这样，整幅图中就会有若干连通分量：

按照题目消除石头的规则，最优消除的策略可以消除每个连通分量中的大部分石头，最终无论如何会剩下一个石头。

所以可以消除掉的石头个数就是石头总数和连通分量个数之差。具体解法看代码吧。

解法代码

```
class Solution {
    public int removeStones(int[][] stones) {
```

```
        int n = stones.length;

        // 一维坐标 -> 节点 ID
        HashMap<Integer, Integer> codeToId = new HashMap<>();
        for (int i = 0; i < n; i++) {
            codeToId.put(encode(stones[i]), i);
        }

        // 记录每一行每一列有哪些节点
        HashMap<Integer, List<Integer>> colIndexToCodes = new HashMap<>();
        HashMap<Integer, List<Integer>> rowIndexToCodes = new HashMap<>();
        for (int[] point : stones) {
            int x = point[0], y = point[1];
            rowIndexToCodes.putIfAbsent(x, new ArrayList<>());
            colIndexToCodes.putIfAbsent(y, new ArrayList<>());
            rowIndexToCodes.get(x).add(encode(point));
            colIndexToCodes.get(y).add(encode(point));
        }

        // 启动 union find 算法
        UF uf = new UF(n);

        // 把每一列的节点连通
        for (int index : colIndexToCodes.keySet()) {
            List<Integer> col = colIndexToCodes.get(index);
            int firstId = codeToId.get(col.get(0));
            for (int i = 1; i < col.size(); i++) {
                int otherId = codeToId.get(col.get(i));
                uf.union(firstId, otherId);
            }
        }

        // 把每一行的节点连通
        for (int index : rowIndexToCodes.keySet()) {
            List<Integer> row = rowIndexToCodes.get(index);
            int firstId = codeToId.get(row.get(0));
            for (int i = 1; i < row.size(); i++) {
                int otherId = codeToId.get(row.get(i));
                uf.union(firstId, otherId);
            }
        }
        // 石头总数 - 连通分量数量就是被消除的石头个数
        return n - uf.count();
    }

    // 将二维坐标转化成一维索引
    int encode(int[] point) {
        return point[0] * 10000 + point[1];
    }

}

// union find 算法模板
class UF {
```

```
    // 连通分量个数
    private int count;
    // 存储每个节点的父节点
    private int[] parent;

    // n 为图中节点的个数
    public UF(int n) {
        this.count = n;
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    // 将节点 p 和节点 q 连通
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);

        if (rootP == rootQ) {
            return;
        }

        parent[rootQ] = rootP;
        // 两个连通分量合并成一个连通分量
        count--;
    }

    // 判断节点 p 和节点 q 是否连通
    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

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

    // 返回图中的连通分量个数
    public int count() {
        return count;
    }
}
```

1135. 最低成本联通所有城市

LeetCode	力扣	难度
1135. Connecting Cities With Minimum Cost	1135. 最低成本联通所有城市	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

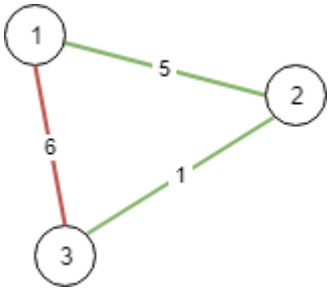
- **标签：图论算法，并查集算法，最小生成树**

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

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

计算使得每对城市都连通的最小成本。如果根据已知条件无法完成该项任务，则请你返回 -1。

示例 1：




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

基本思路

每座城市相当于图中的节点，连通城市的成本相当于边的权重，连通所有城市的最小成本即是最小生成树的权重之和。

Kruskal 最小生成树算法的逻辑如下：

将所有边按照权重从小到大排序，从权重最小的边开始遍历，如果这条边和 `mst` 中的其它边不会形成环，则这条边是最小生成树的一部分，将它加入 `mst` 集合；否则，这条边不是最小生成树的一部分，不要把它加入 `mst` 集合。

- **详细题解：Kruskal 最小生成树算法**

解法代码

```java
class Solution {
    public int minimumCost(int n, int[][] connections) {
        // 城市编号为 1...n，所以初始化大小为 n + 1
        UF uf = new UF(n + 1);
        // 对所有边按照权重从小到大排序
        Arrays.sort(connections, (a, b) -> (a[2] - b[2]));
        // 记录最小生成树的权重之和
        int mst = 0;
        for (int[] edge : connections) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];
            // 若这条边会产生环，则不能加入 mst
            if (uf.connected(u, v)) {
                continue;
            }
            // 若这条边不会产生环，则属于最小生成树
            mst += weight;
            uf.union(u, v);
        }
        // 保证所有节点都被连通
        // 按理说 uf.count() == 1 说明所有节点被连通
        // 但因为节点 0 没有被使用，所以 0 会额外占用一个连通分量
        return uf.count() == 2 ? mst : -1;
    }

    class UF {
        // 连通分量个数
        private int count;
        // 存储一棵树
        private int[] parent;
        // 记录树的「重量」
        private int[] size;

        // n 为图中节点的个数
        public UF(int n) {
            this.count = n;
            parent = new int[n];
            size = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                size[i] = 1;
            }
        }

        // 将节点 p 和节点 q 连通
        public void union(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            if (rootP == rootQ)
                return;

            // 小树接到大树下面，较平衡
```

```
            if (size[rootP] > size[rootQ]) {
                parent[rootQ] = rootP;
                size[rootP] += size[rootQ];
            } else {
                parent[rootP] = rootQ;
                size[rootQ] += size[rootP];
            }
            // 两个连通分量合并成一个连通分量
            count--;
        }

        // 判断节点 p 和节点 q 是否连通
        public boolean connected(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            return rootP == rootQ;
        }

        // 返回节点 x 的连通分量根节点
        private int find(int x) {
            while (parent[x] != x) {
                // 进行路径压缩
                parent[x] = parent[parent[x]];
                x = parent[x];
            }
            return x;
        }

        // 返回图中的连通分量个数
        public int count() {
            return count;
        }
    }
}
```

- **类似题目**：
  - 1584. 连接所有点的最小费用 🟠
  - 261. 以图判树 🟠

1361. 验证二叉树

LeetCode	力扣	难度
1361. Validate Binary Tree Nodes	1361. 验证二叉树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip，并查集算法**

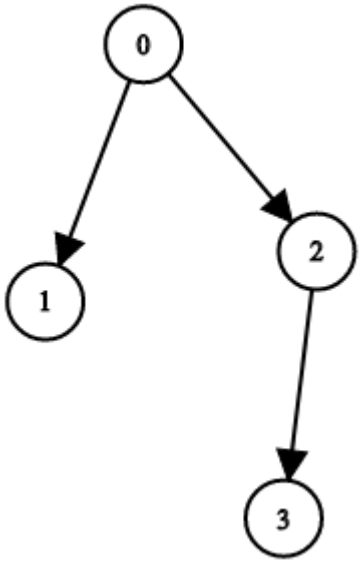
二叉树上有 `n` 个节点，按从 `0` 到 `n - 1` 编号，其中节点 `i` 的两个子节点分别是 `leftChild[i]` 和 `rightChild[i]`。

只有 **所有** 节点能够形成且 **只** 形成**一颗** 有效的二叉树时，返回 `true`；否则返回 `false`。

如果节点 `i` 没有左子节点，那么 `leftChild[i]` 就等于 `-1`。右子节点也符合该规则。

注意：节点没有值，本问题中仅仅使用节点编号。

示例 1：




```
输入：n = 4, leftChild = [1,-1,3,-1], rightChild = [2,-1,-1,-1]
输出：true
```

基本思路

看到这道题我就想到 261. 以图判树，你可以先去做这道题，理解树和图的关键区别之后再来做这道题。

我们解决 261 题的思路是用 Union-Find 并查集算法，但这道题的不同之处在于：

这里的每条边相当于是有方向的，而标准的并查集算法是处理无向图的，如果直接套用的话会出问题。

比如说，一个正常二叉树的节点不可能有两个入度（两个父节点），单纯的并查集算法只能检查是否成环，无法检查每个节点到底有多少入度。

不过我们可以用额外的代码来检查每个节点的入度是否合法，最后用并查集算法检测是否成环，从而判断二叉树是否合法。

除了并查集算法，我们还可以用二叉树的遍历函数来检查是否成环，两种思路我都写了解法代码，具体细节见代码注释。

解法代码

```
// 用 Union Find 算法判断
class Solution {
    public boolean validateBinaryTreeNodes(int n, int[] leftChild, int[]
rightChild) {
        // 记录每个节点的入度
        int[] indegree = new int[n];
        for (int i = 0; i < n; i++) {
            if (leftChild[i] != -1) {
                indegree[leftChild[i]]++;
            }
            if (rightChild[i] != -1) {
                indegree[rightChild[i]]++;
            }
        }
        // 按道理应该有且只有根节点的入度为 0，
        // 其他节点的入度都必须为 1
        int root = -1;
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                if (root != -1) {
                    // 有多个入度为 0 的节点
                    return false;
                }
                root = i;
            } else if (indegree[i] != 1) {
                // 除了根节点外其他节点的入度都必须为 1
                return false;
            }
        }

        // 如果没有根节点，那肯定不是合法二叉树
        if (root == -1) {
            return false;
        }

        // 启动 Union-Find 并查集算法，
        // 保证树中只有一个联通分量且不成环
        UF uf = new UF(n);
        for (int i = 0; i < n; i++) {
            int left = leftChild[i];
            int right = rightChild[i];
```

```
            if (left != -1) {
                if (uf.connected(i, left)) {
                    // 成环
                    return false;
                }
                uf.union(i, left);
            }
            if (right != -1) {
                if (uf.connected(i, right)) {
                    // 成环
                    return false;
                }
                uf.union(i, right);
            }
        }
        // 要保证只有一个连通分量
        return uf.count() == 1;
    }

    class UF {
        // 记录连通分量个数
        private int count;
        // 存储若干棵树
        private int[] parent;
        // 记录树的“重量”
        private int[] size;

        public UF(int n) {
            this.count = n;
            parent = new int[n];
            size = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                size[i] = 1;
            }
        }

        /* 将 p 和 q 连通 */
        public void union(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            if (rootP == rootQ)
                return;

            // 小树接到大树下面，较平衡
            if (size[rootP] > size[rootQ]) {
                parent[rootQ] = rootP;
                size[rootP] += size[rootQ];
            } else {
                parent[rootP] = rootQ;
                size[rootQ] += size[rootP];
            }
            count--;
```

```
        }

        /* 判断 p 和 q 是否互相连通 */
        public boolean connected(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            // 处于同一棵树上的节点，相互连通
            return rootP == rootQ;
        }

        /* 返回节点 x 的根节点 */
        private int find(int x) {
            while (parent[x] != x) {
                // 进行路径压缩
                parent[x] = parent[parent[x]];
                x = parent[x];
            }
            return x;
        }

        public int count() {
            return count;
        }
    }
}

// 用 DFS 遍历算法判断
class Solution2 {
    public boolean validateBinaryTreeNodes(int n, int[] leftChild, int[]
rightChild) {
        // 记录每个节点的入度
        int[] indegree = new int[n];
        for (int i = 0; i < n; i++) {
            if (leftChild[i] != -1) {
                indegree[leftChild[i]]++;
            }
            if (rightChild[i] != -1) {
                indegree[rightChild[i]]++;
            }
        }
        // 按道理应该有且只有根节点的入度为 0，
        // 其他节点的入度都必须为 1
        int root = -1;
        for (int i = 0; i < n; i++) {
            if (indegree[i] == 0) {
                if (root != -1) {
                    // 有多个入度为 0 的节点
                    return false;
                }
                root = i;
            } else if (indegree[i] != 1) {
                // 除了根节点外其他节点的入度都必须为 1
                return false;
            }
```

```
        }

        // 如果没有根节点，那肯定不是合法二叉树
        if (root == -1) {
            return false;
        }

        // 为了凸显二叉树遍历框架，我把这些都作为全局变量
        this.leftChild = leftChild;
        this.rightChild = rightChild;
        this.visited = new boolean[n];

        // 用二叉树遍历框架进行遍历，
        // 保证树中只有一个联通分量且不成环
        traverse(root);
        // 遍历过程中发现成环了，说明肯定不是二叉树
        if (hasCycle) {
            return false;
        }
        // 如果一次遍历没有经过所有节点，也说明不是二叉树
        for (int i = 0; i < n; i++) {
            if (visited[i] != true) {
                return false;
            }
        }
        // 能通过上面的检测，判定是一棵二叉树
        return true;
    }

    int[] leftChild, rightChild;
    // 记录遍历过的节点，防止成环
    boolean[] visited;
    boolean hasCycle = false;

    // 二叉树遍历函数
    void traverse(int root) {
        if (root == -1 || hasCycle) {
            return;
        }
        // 走了回头路，说明成环了
        if (visited[root]) {
            hasCycle = true;
            return;
        }
        visited[root] = true;

        traverse(leftChild[root]);
        traverse(rightChild[root]);
    }
}
```

1584. 连接所有点的最小费用

LeetCode	力扣	难度
1584. Min Cost to Connect All Points	1584. 连接所有点的最小费用	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

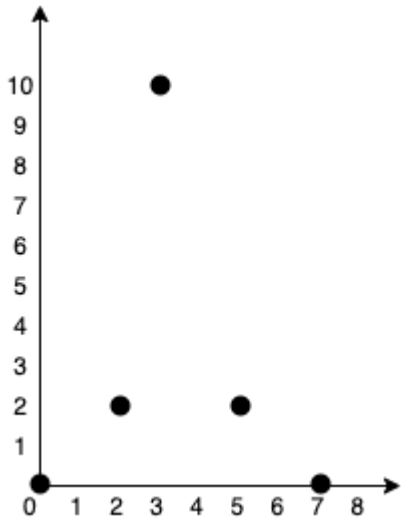
- **标签：图论算法，并查集算法，最小生成树**

给你一个 `points` 数组，表示 2D 平面上的一些点，其中 `points[i] = [xi, yi]`。

连接点 `[xi, yi]` 和点 `[xj, yj]` 的费用为它们之间的**曼哈顿距离**：`|xi - xj| + |yi - yj|`，其中 `|val|` 表示 `val` 的绝对值。

请你返回将所有点连接的最小总费用。只有任意两点之间**有且仅有**一条简单路径时，才认为所有点都已连接。

示例 1：




```
输入：points = [[0,0],[2,2],[3,10],[5,2],[7,0]]
输出：20
解释：

我们可以按照上图所示连接所有点得到最小总费用，总费用为 20。
注意到任意两个点之间只有唯一一条路径互相到达。
```

基本思路

很显然这也是一个标准的最小生成树问题：每个点就是无向加权图中的节点，边的权重就是曼哈顿距离，连接所有点的最小费用就是最小生成树的权重和。

所以解法思路就是先生成所有的边以及权重，然后对这些边执行 Kruskal 算法即可。

这道题做了一个小的变通：每个坐标点是一个二元组，那么按理说应该用五元组表示一条带权重的边，但这样的话不便执行 Union-Find 算法；所以我们用 `points` 数组中的索引代表每个坐标点，这样就可以直接复用之前的 Kruskal 算法逻辑了。

- **详细题解：Kruskal 最小生成树算法**

解法代码

```
class Solution {
    public int minCostConnectPoints(int[][] points) {
        int n = points.length;
        // 生成所有边及权重
        List<int[]> edges = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int xi = points[i][0], yi = points[i][1];
                int xj = points[j][0], yj = points[j][1];
                // 用坐标点在 points 中的索引表示坐标点
                edges.add(new int[]{
                        i, j, Math.abs(xi - xj) + Math.abs(yi - yj)
                });
            }
        }
        // 将边按照权重从小到大排序
        Collections.sort(edges, (a, b) -> {
            return a[2] - b[2];
        });
        // 执行 Kruskal 算法
        int mst = 0;
        UF uf = new UF(n);
        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];
            // 若这条边会产生环，则不能加入 mst
            if (uf.connected(u, v)) {
                continue;
            }
            // 若这条边不会产生环，则属于最小生成树
            mst += weight;
            uf.union(u, v);
        }
        return mst;
    }

    class UF {
        // 连通分量个数
        private int count;
        // 存储一棵树
        private int[] parent;
        // 记录树的「重量」
        private int[] size;
```

```
        // n 为图中节点的个数
        public UF(int n) {
            this.count = n;
            parent = new int[n];
            size = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                size[i] = 1;
            }
        }

        // 将节点 p 和节点 q 连通
        public void union(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            if (rootP == rootQ)
                return;

            // 小树接到大树下面，较平衡
            if (size[rootP] > size[rootQ]) {
                parent[rootQ] = rootP;
                size[rootP] += size[rootQ];
            } else {
                parent[rootP] = rootQ;
                size[rootQ] += size[rootP];
            }
            // 两个连通分量合并成一个连通分量
            count--;
        }

        // 判断节点 p 和节点 q 是否连通
        public boolean connected(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            return rootP == rootQ;
        }

        // 返回节点 x 的连通分量根节点
        private int find(int x) {
            while (parent[x] != x) {
                // 进行路径压缩
                parent[x] = parent[parent[x]];
                x = parent[x];
            }
            return x;
        }

        // 返回图中的连通分量个数
        public int count() {
            return count;
        }
    }
}
```

- **类似题目**：
  - 1135. 最低成本联通所有城市 🟠
  - 261. 以图判树 🟠

261. 以图判树

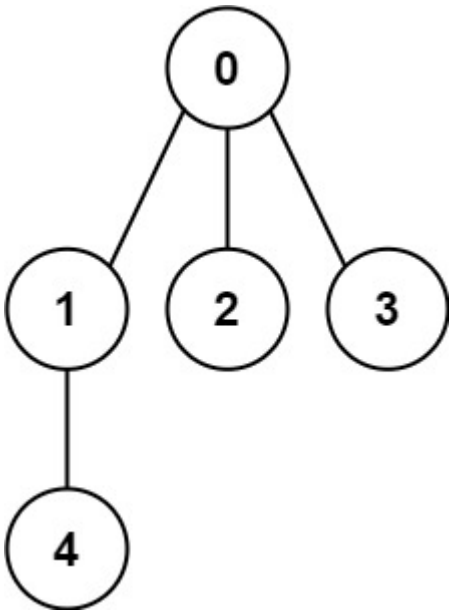
LeetCode	力扣	难度
261. Graph Valid Tree	261. 以图判树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：图论算法，并查集算法，最小生成树**

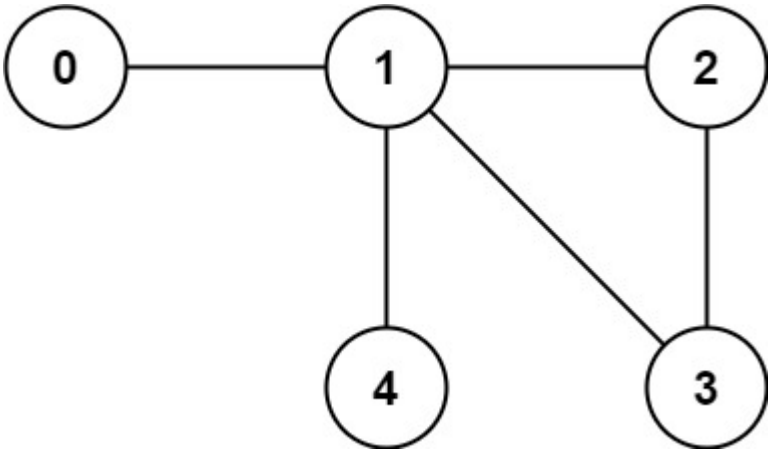
给定从 `0` 到 `n-1` 标号的 `n` 个结点，和一个无向边列表（每条边以结点对来表示），请编写一个函数用来判断这些边是否能够形成一个合法有效的树结构。

示例 1：




```
输入: n = 5, 边列表 edges = [[0,1], [0,2], [0,3], [1,4]]
输出: true
```

示例 2：

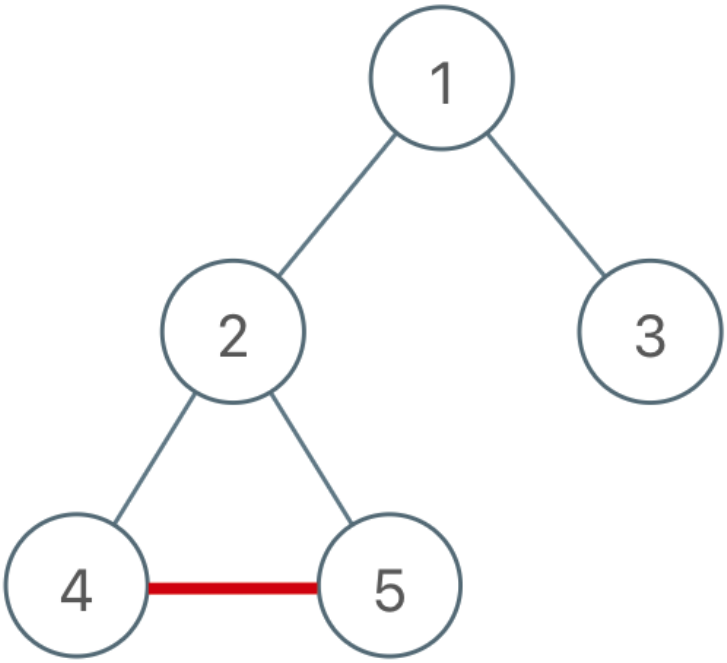


```
输入: n = 5, 边列表 edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]
输出: false
```

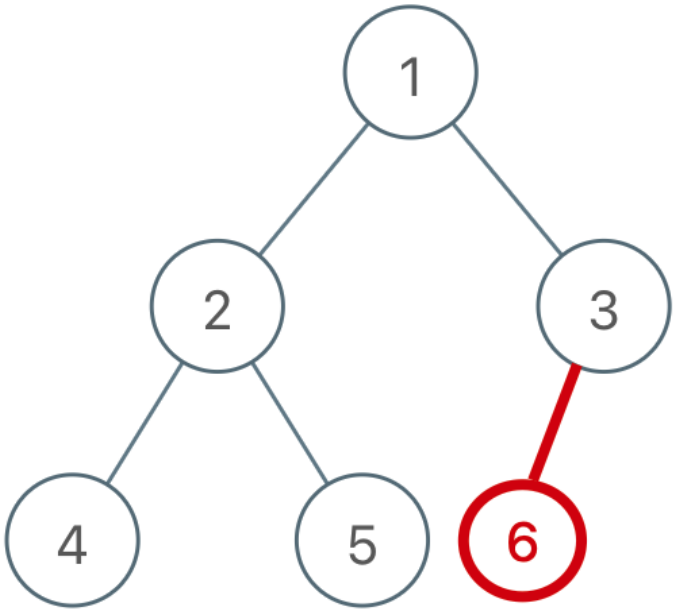
基本思路

对于这道题，我们可以思考一下，什么情况下加入一条边会使得树变成图（出现环）？

显然，像下面这样添加边会出现环：



而这样添加边则不会出现环：



总结一下规律就是：

对于添加的这条边，如果该边的两个节点本来就在同一连通分量里，那么添加这条边会产生环；反之，如果该边的两个节点不在同一连通分量里，则添加这条边不会产生环。

而判断两个节点是否连通（是否在同一个连通分量中）就是 Union-Find 并查集算法 的拿手绝活。

- **详细题解：Kruskal 最小生成树算法**

解法代码

```
class Solution {
    public boolean validTree(int n, int[][] edges) {
        // 初始化 0...n-1 共 n 个节点
        UF uf = new UF(n);
        // 遍历所有边，将组成边的两个节点进行连接
        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            // 若两个节点已经在同一连通分量中，会产生环
            if (uf.connected(u, v)) {
                return false;
            }
            // 这条边不会产生环，可以是树的一部分
            uf.union(u, v);
        }
        // 要保证最后只形成了一棵树，即只有一个连通分量
        return uf.count() == 1;
    }
```

```
class UF {
    // 连通分量个数
    private int count;
    // 存储一棵树
    private int[] parent;
    // 记录树的「重量」
    private int[] size;

    // n 为图中节点的个数
    public UF(int n) {
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    // 将节点 p 和节点 q 连通
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        }
        // 两个连通分量合并成一个连通分量
        count--;
    }

    // 判断节点 p 和节点 q 是否连通
    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    // 返回节点 x 的连通分量根节点
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }
```

```
        // 返回图中的连通分量个数
        public int count() {
            return count;
        }
    }
}
```

- **类似题目**：
  - 1135. 最低成本联通所有城市 🟠
  - 1361. 验证二叉树 🟠
  - 1584. 连接所有点的最小费用 🟠

1514. 概率最大的路径

LeetCode	力扣	难度
1514. Path with Maximum Probability	1514. 概率最大的路径	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

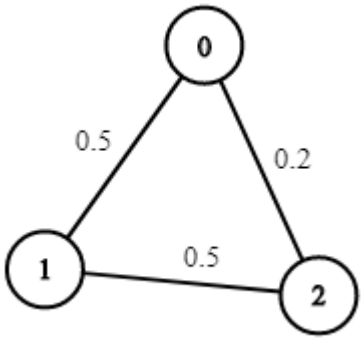
- **标签：Dijkstra 算法，图论算法，最短路径算法**

给你一个由 `n` 个节点（下标从 0 开始）组成的无向加权图，该图由一个描述边的列表组成，其中 `edges[i] = [a, b]` 表示连接节点 `a` 和 `b` 的一条无向边，且该边遍历成功的概率为 `succProb[i]`。

指定两个节点分别作为起点 `start` 和终点 `end`，请你找出从起点到终点成功概率最大的路径，并返回其成功概率。

如果不存在从 `start` 到 `end` 的路径，请**返回 0**。只要答案与标准答案的误差不超过 **1e-5**，就会被视作正确答案。

示例 1：




```
输入：n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start = 0, end = 2
输出：0.25000
解释：从起点到终点有两条路径，其中一条的成功概率为 0.2，而另一条为 0.5 * 0.5 = 0.25
```

基本思路

虽然这题让计算最大值，但是也可以用 Dijkstra 算法模板，由于 Dijkstra 算法背景知识较多，请看详细题解。

- **详细题解：Dijkstra 算法模板及应用**

解法代码

```
class Solution {
    public double maxProbability(int n, int[][] edges, double[] succProb,
int start, int end) {
        List<double[]>[] graph = new LinkedList[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new LinkedList<>();
        }
        // 构造无向图
        for (int i = 0; i < edges.length; i++) {
            int from = edges[i][0];
            int to = edges[i][1];
            double weight = succProb[i];
            // 无向图其实就是双向图
            graph[from].add(new double[]{(double)to, weight});
            graph[to].add(new double[]{(double)from, weight});
        }


        return dijkstra(start, end, graph);
    }

    class State {
        // 图节点的 id
        int id;
        // 从 start 节点到当前节点的距离
        double distFromStart;

        State(int id, double distFromStart) {
            this.id = id;
            this.distFromStart = distFromStart;
        }
    }

    double dijkstra(int start, int end, List<double[]>[] graph) {
        // 图中节点的个数
        int V = graph.length;
        // 记录最短路径的权重，你可以理解为 dp table
        // 定义：distTo[i] 的值就是节点 start 到达节点 i 的最短路径权重
        double[] distTo = new double[V];
        // dp table 初始化为正无穷
        Arrays.fill(distTo, -1);
        // base case，start 到 start 的最短距离就是 0
        distTo[start] = 1;

        // 优先级队列，distFromStart 较小的排在前面
        Queue<State> pq = new PriorityQueue<>((a, b) -> {
            return Double.compare(b.distFromStart, a.distFromStart);
        });
        // 从起点 start 开始进行 BFS
        pq.offer(new State(start, 1));

        while (!pq.isEmpty()) {
            State curState = pq.poll();
```

```
                int curNodeID = curState.id;
                double curDistFromStart = curState.distFromStart;

                // 在这里加一个判断就行了，其他代码不用改
                if (curNodeID == end) {
                    return curDistFromStart;
                }

                if (curDistFromStart < distTo[curNodeID]) {
                    // 已经有一条更短的路径到达 curNode 节点了
                    continue;
                }
                // 将 curNode 的相邻节点装入队列
                for (double[] neighbor : graph[curNodeID]) {
                    int nextNodeID = (int)neighbor[0];
                    // 看看从 curNode 达到 nextNode 的距离是否会更短
                    double distToNextNode = distTo[curNodeID] * neighbor[1];
                    if (distTo[nextNodeID] < distToNextNode) {
                        // 更新 dp table
                        distTo[nextNodeID] = distToNextNode;
                        // 将这个节点以及距离放入队列
                        pq.offer(new State(nextNodeID, distToNextNode));
                    }
                }
            }
            return 0.0;
        }
    }
```

- **类似题目**：
  - 1631. 最小体力消耗路径 🟠
  - 743. 网络延迟时间 🟠

1631. 最小体力消耗路径

LeetCode	力扣	难度
1631. Path With Minimum Effort	1631. 最小体力消耗路径	🟠

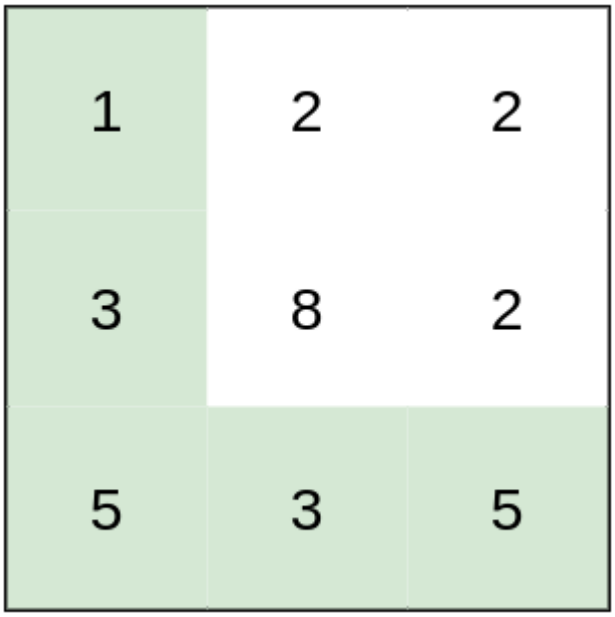
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：Dijkstra 算法，图论算法，最短路径算法

你准备参加一场远足活动。给你一个二维 `rows x columns` 的地图 `heights`，其中 `heights[row][col]` 表示格子 `(row, col)` 的高度。一开始你在最左上角的格子 `(0, 0)`，且你希望去最右下角的格子 `(rows-1, columns-1)`（注意下标从**0**开始编号）。你每次可以往**上**，**下**，**左**，**右** 四个方向之一移动，你想要找到耗费**体力**最小的一条路径。

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**决定的。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，因为另一条路径差值最大值为 3。
```

基本思路

如果你把二维数组中每个 `(x, y)` 坐标看做一个节点，它的上下左右坐标就是相邻节点，它对应的值和相邻坐标对应的值之差的绝对值就是题目说的「体力消耗」，你就可以理解为边的权重。

这样就可以使用 Dijkstra 算法求解了，只不过这道题中评判一条路径是长还是短的标准不再是路径经过的权重总和，而是路径经过的权重最大值。

Dijkstra 算法模板的背景知识较多，请看详细题解。

- **详细题解：Dijkstra 算法模板及应用**

解法代码

```
class Solution {
    // Dijkstra 算法，计算 (0, 0) 到 (m - 1, n - 1) 的最小体力消耗
    public int minimumEffortPath(int[][] heights) {
        int m = heights.length, n = heights[0].length;
        // 定义：从 (0, 0) 到 (i, j) 的最小体力消耗是 effortTo[i][j]
        int[][] effortTo = new int[m][n];
        // dp table 初始化为正无穷
        for (int i = 0; i < m; i++) {
            Arrays.fill(effortTo[i], Integer.MAX_VALUE);
        }
        // base case，起点到起点的最小消耗就是 0
        effortTo[0][0] = 0;

        // 优先级队列，effortFromStart 较小的排在前面
        Queue<State> pq = new PriorityQueue<>((a, b) -> {
            return a.effortFromStart - b.effortFromStart;
        });

        // 从起点 (0, 0) 开始进行 BFS
        pq.offer(new State(0, 0, 0));

        while (!pq.isEmpty()) {
            State curState = pq.poll();
            int curX = curState.x;
            int curY = curState.y;
            int curEffortFromStart = curState.effortFromStart;

            // 到达终点提前结束
            if (curX == m - 1 && curY == n - 1) {
                return curEffortFromStart;
            }

            if (curEffortFromStart > effortTo[curX][curY]) {
                continue;
            }
            // 将 (curX, curY) 的相邻坐标装入队列
            for (int[] neighbor : adj(heights, curX, curY)) {
                int nextX = neighbor[0];
                int nextY = neighbor[1];
                // 计算从 (curX, curY) 达到 (nextX, nextY) 的消耗
                int effortToNextNode = Math.max(
                        effortTo[curX][curY],
                        Math.abs(heights[curX][curY] - heights[nextX]
```

```
[nextY])
                );
                // 更新 dp table
                if (effortTo[nextX][nextY] > effortToNextNode) {
                    effortTo[nextX][nextY] = effortToNextNode;
                    pq.offer(new State(nextX, nextY, effortToNextNode));
                }
            }
        }
        // 正常情况不会达到这个 return
        return -1;
    }

    // 方向数组，上下左右的坐标偏移量
    int[][] dirs = new int[][]{{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // 返回坐标 (x, y) 的上下左右相邻坐标
    List<int[]> adj(int[][] matrix, int x, int y) {
        int m = matrix.length, n = matrix[0].length;
        // 存储相邻节点
        List<int[]> neighbors = new ArrayList<>();
        for (int[] dir : dirs) {
            int nx = x + dir[0];
            int ny = y + dir[1];
            if (nx >= m || nx < 0 || ny >= n || ny < 0) {
                // 索引越界
                continue;
            }
            neighbors.add(new int[]{nx, ny});
        }
        return neighbors;
    }

    class State {
        // 矩阵中的一个位置
        int x, y;
        // 从起点 (0, 0) 到当前位置的最小体力消耗（距离）
        int effortFromStart;

        State(int x, int y, int effortFromStart) {
            this.x = x;
            this.y = y;
            this.effortFromStart = effortFromStart;
        }
    }
}
```

- **类似题目**：
  - 1514. 概率最大的路径 🟠
  - 743. 网络延迟时间 🟠

743. 网络延迟时间

LeetCode	力扣	难度
743. Network Delay Time	743. 网络延迟时间	🟠

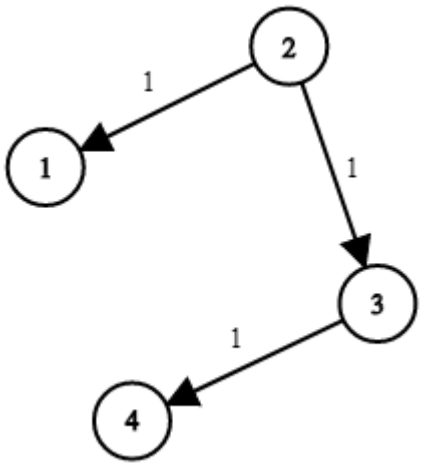
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：Dijkstra 算法，图论算法，最短路径算法**

有 `n` 个网络节点，标记为 `1` 到 `n`，给你一个列表 `times`，表示信号经过**有向**边的传递时间。`times[i] = (ui, vi, wi)`，其中 `ui` 是源节点，`vi` 是目标节点，`wi` 是一个信号从源节点传递到目标节点的时间。

现在，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？如果不能使所有节点收到信号，返回 `-1`。

示例 1：




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

基本思路

典型的 Dijkstra 算法应用场景，把延迟看做边的权重，最小延迟就是最小权重路径。

Dijkstra 算法模板的背景知识较多，请看详细题解。

- **详细题解：Dijkstra 算法模板及应用**

解法代码

```
class Solution {
    public int networkDelayTime(int[][] times, int n, int k) {
        // 节点编号是从 1 开始的，所以要一个大小为 n + 1 的邻接表
```

```
        List<int[]>[] graph = new LinkedList[n + 1];
        for (int i = 1; i <= n; i++) {
            graph[i] = new LinkedList<>();
        }
        // 构造图
        for (int[] edge : times) {
            int from = edge[0];
            int to = edge[1];
            int weight = edge[2];
            // from -> List<(to, weight)>
            // 邻接表存储图结构，同时存储权重信息
            graph[from].add(new int[]{to, weight});
        }
        // 启动 dijkstra 算法计算以节点 k 为起点到其他节点的最短路径
        int[] distTo = dijkstra(k, graph);

        // 找到最长的那一条最短路径
        int res = 0;
        for (int i = 1; i < distTo.length; i++) {
            if (distTo[i] == Integer.MAX_VALUE) {
                // 有节点不可达，返回 -1
                return -1;
            }
            res = Math.max(res, distTo[i]);
        }
        return res;
    }

    class State {
        // 图节点的 id
        int id;
        // 从 start 节点到当前节点的距离
        int distFromStart;

        State(int id, int distFromStart) {
            this.id = id;
            this.distFromStart = distFromStart;
        }
    }

    // 输入一个起点 start，计算从 start 到其他节点的最短距离
    int[] dijkstra(int start, List<int[]>[] graph) {
        // 定义：distTo[i] 的值就是起点 start 到达节点 i 的最短路径权重
        int[] distTo = new int[graph.length];
        Arrays.fill(distTo, Integer.MAX_VALUE);
        // base case，start 到 start 的最短距离就是 0
        distTo[start] = 0;

        // 优先级队列，distFromStart 较小的排在前面
        Queue<State> pq = new PriorityQueue<>((a, b) -> {
            return a.distFromStart - b.distFromStart;
        });
        // 从起点 start 开始进行 BFS
        pq.offer(new State(start, 0));
```

```
        while (!pq.isEmpty()) {
            State curState = pq.poll();
            int curNodeID = curState.id;
            int curDistFromStart = curState.distFromStart;

            if (curDistFromStart > distTo[curNodeID]) {
                continue;
            }

            // 将 curNode 的相邻节点装入队列
            for (int[] neighbor : graph[curNodeID]) {
                int nextNodeID = neighbor[0];
                int distToNextNode = distTo[curNodeID] + neighbor[1];
                // 更新 dp table
                if (distTo[nextNodeID] > distToNextNode) {
                    distTo[nextNodeID] = distToNextNode;
                    pq.offer(new State(nextNodeID, distToNextNode));
                }
            }
        }
        return distTo;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 1514. 概率最大的路径 🟠
  - 1631. 最小体力消耗路径 🟠

17. 电话号码的字母组合

LeetCode	力扣	难度
17. Letter Combinations of a Phone Number	17. 电话号码的字母组合	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：回溯算法，数学

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。答案可以按**任意顺序** 返回。

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 1：

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

基本思路

你需要先看前文 回溯算法详解 和 回溯算法团灭子集、排列、组合问题，然后看这道题就很简单了，无非是回溯算法的运用而已。

组合问题本质上就是遍历一棵回溯树，套用回溯算法代码框架即可。

解法代码

```
class Solution {
    // 每个数字到字母的映射
    String[] mapping = new String[] {
            "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
"wxyz"
    };

    List<String> res = new LinkedList<>();

    public List<String> letterCombinations(String digits) {
```

```
        if (digits.isEmpty()) {
            return res;
        }
        // 从 digits[0] 开始进行回溯
        backtrack(digits, 0, new StringBuilder());
        return res;
    }

    // 回溯算法主函数
    void backtrack(String digits, int start, StringBuilder sb) {
        if (sb.length() == digits.length()) {
            // 到达回溯树底部
            res.add(sb.toString());
            return;
        }
        // 回溯算法框架
        for (int i = start; i < digits.length(); i++) {
            int digit = digits.charAt(i) - '0';
            for (char c : mapping[digit].toCharArray()) {
                // 做选择
                sb.append(c);
                // 递归下一层回溯树
                backtrack(digits, i + 1, sb);
                // 撤销选择
                sb.deleteCharAt(sb.length() - 1);
            }
        }
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

22. 括号生成

LeetCode	力扣	难度
22. Generate Parentheses	22. 括号生成	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法**

数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且**有效的**括号组合。

有效括号组合需满足：左括号必须以正确的顺序闭合。

示例 1：

```
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

基本思路

PS：这道题在《算法小抄》 的第 306 页。

本题可以改写为：

现在有 `2n` 个位置，每个位置可以放置字符 `(` 或者 `)`，组成的所有括号组合中，有多少个是合法的？

这就是典型的回溯算法提醒，暴力穷举就行了。

不过为了减少不必要的穷举，我们要知道合法括号串有以下性质：

1、一个「合法」括号组合的左括号数量一定等于右括号数量，这个很好理解。

2、对于一个「合法」的括号字符串组合 `p`，必然对于任何 `0 <= i < len(p)` 都有：子串 `p[0..i]` 中左括号的数量都大于或等于右括号的数量。

因为从左往右算的话，肯定是左括号多嘛，到最后左右括号数量相等，说明这个括号组合是合法的。

用 `left` 记录还可以使用多少个左括号，用 `right` 记录还可以使用多少个右括号，就可以直接套用 回溯算法套路框架 了。

- **详细题解：回溯算法最佳实践：括号生成**

解法代码

```
class Solution {
    public:
    vector<string> generateParenthesis(int n) {
```

```
        if (n == 0) return {};
        // 记录所有合法的括号组合
        vector<string> res;
        // 回溯过程中的路径
        string track;
        // 可用的左括号和右括号数量初始化为 n
        backtrack(n, n, track, res);
        return res;
    }

    // 可用的左括号数量为 left 个，可用的右括号数量为 rgiht 个
    void backtrack(int left, int right,
                string& track, vector<string>& res) {
        // 若左括号剩下的多，说明不合法
        if (right < left) return;
        // 数量小于 0 肯定是不合法的
        if (left < 0 || right < 0) return;
        // 当所有括号都恰好用完时，得到一个合法的括号组合
        if (left == 0 && right == 0) {
            res.push_back(track);
            return;
        }

        // 尝试放一个左括号
        track.push_back('('); // 选择
        backtrack(left - 1, right, track, res);
        track.pop_back(); // 撤消选择

        // 尝试放一个右括号
        track.push_back(')'); // 选择
        backtrack(left, right - 1, track, res);
        track.pop_back(); // 撤消选择
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 剑指 Offer II 085. 生成匹配的括号 🟠

剑指 Offer II 085. 生成匹配的括号

这道题和 22. 括号生成 相同。

37. 解数独

LeetCode	力扣	难度
37. Sudoku Solver	37. 解数独	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：回溯算法

编写一个程序，通过填充空格来解决数独问题，数独的解法需**遵循如下规则**：

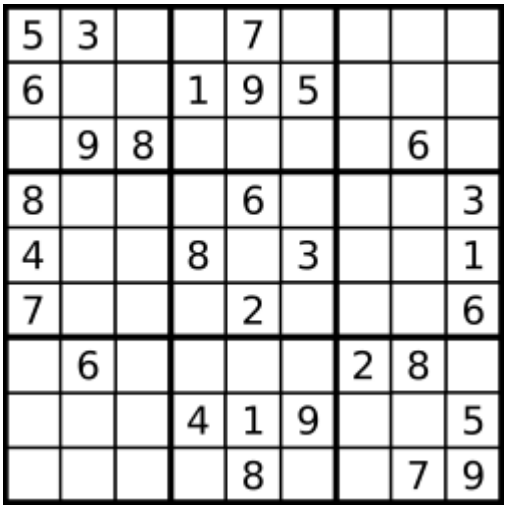
1、数字 1-9 在每一行只能出现一次。

2、数字 1-9 在每一列只能出现一次。

3、数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

数独部分空格内已填入了数字，空白格用 '.' 表示。

示例：




```
输入：board = [["5","3",".",".","7",".",".",".","."],
["6",".",".","1","9","5",".",".","."],
[".","9","8",".",".",".",".","6","."],
["8",".",".",".","6",".",".",".","3"],
["4",".",".","8",".","3",".",".","1"],
["7",".",".",".","2",".",".",".","6"],
[".","6",".",".",".",".","2","8","."],
[".",".",".","4","1","9",".",".","5"],
[".",".",".",".","8",".",".","7","9"]]
输出：[["5","3","4","6","7","8","9","1","2"],
["6","7","2","1","9","5","3","4","8"],
["1","9","8","3","4","2","5","6","7"],
["8","5","9","7","6","1","4","2","3"],
["4","2","6","8","5","3","7","9","1"],
```

```
["7","1","3","9","2","4","8","5","6"],
["9","6","1","5","3","7","2","8","4"],
["2","8","7","4","1","9","6","3","5"],
["3","4","5","2","8","6","1","7","9"]]
```

解释：输入的数独如上图所示，唯一有效的解决方案如下所示：

基本思路

算法的核心思路非常非常的简单，就是穷举：

对每一个空着的格子穷举 1 到 9，如果遇到不合法的数字（在同一行或同一列或同一个 3×3 的区域中存在相同的数字）则跳过，如果找到一个合法的数字，则继续穷举下一个空格子。

- **详细题解：回溯算法最佳实践：解数独**

解法代码

```
class Solution {
    public void solveSudoku(char[][] board) {
        backtrack(board, 0, 0);
    }

    boolean backtrack(char[][] board, int i, int j) {
        int m = 9, n = 9;
        if (j == n) {
            // 穷举到最后一列的话就换到下一行重新开始。
            return backtrack(board, i + 1, 0);
        }
        if (i == m) {
            // 找到一个可行解，触发 base case
            return true;
        }

        if (board[i][j] != '.') {
            // 如果有预设数字，不用我们穷举
            return backtrack(board, i, j + 1);
        }
```

```
        for (char ch = '1'; ch <= '9'; ch++) {
            // 如果遇到不合法的数字，就跳过
            if (!isValid(board, i, j, ch))
                continue;

            board[i][j] = ch;
            // 如果找到一个可行解，立即结束
            if (backtrack(board, i, j + 1)) {
                return true;
            }
            board[i][j] = '.';
        }
        // 穷举完 1~9，依然没有找到可行解，此路不通
        return false;
    }

    // 判断 board[i][j] 是否可以填入 n
    boolean isValid(char[][] board, int r, int c, char n) {
        for (int i = 0; i < 9; i++) {
            // 判断行是否存在重复
            if (board[r][i] == n) return false;
            // 判断列是否存在重复
            if (board[i][c] == n) return false;
            // 判断 3 x 3 方框是否存在重复
            if (board[(r/3)*3 + i/3][(c/3)*3 + i%3] == n)
                return false;
        }
        return true;
    }
}
```

39. 组合总和

LeetCode	力扣	难度
39. Combination Sum	39. 组合总和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：回溯算法

给定一个**无重复元素**的正整数数组 candidates 和一个正整数 target，找出 candidates 中所有可以使数字和为目标数 target 的唯一组合。

提示：candidates 中的数字可以无限制重复被选取。如果至少一个所选数字数量不同，则两种组合是唯一的。

示例 1：

```
输入：candidates = [2,3,6,7], target = 7
输出：[[7],[2,2,3]]
```

示例 2：

```
输入：candidates = [2,3,5], target = 8
输出：[[2,2,2,2],[2,3,3],[3,5]]
```

基本思路

> 本文有视频版：回溯算法秒杀所有排列/组合/子集问题

你需要先看前文 回溯算法详解 和 回溯算法团灭子集、排列、组合问题，然后看这道题就很简单了，无非是回溯算法的运用而已。

这道题的关键在于 candidates 中的元素可以复用多次，体现在代码中是下面这段：

```
void backtrack(int[] candidates, int start, int target, int sum) {
    // 回溯算法框架
    for (int i = start; i < candidates.length; i++) {
        // 选择 candidates[i]
        backtrack(candidates, i, target, sum);
        // 撤销选择 candidates[i]
    }
}
```

对比 回溯算法团灭子集、排列、组合问题 中不能重复使用元素的标准组合问题：

```
void backtrack(int[] candidates, int start, int target, int sum) {
    // 回溯算法框架
    for (int i = start; i < candidates.length; i++) {
        // 选择 candidates[i]
        backtrack(candidates, i + 1, target, sum);
        // 撤销选择 candidates[i]
    }
}
```

体会到控制是否重复使用元素的关键了吗？

- **详细题解：回溯算法秒杀所有排列/组合/子集问题**

解法代码

```
class Solution {
    List<List<Integer>> res = new LinkedList<>();

    public List<List<Integer>> combinationSum(int[] candidates, int
target) {
        if (candidates.length == 0) {
            return res;
        }
        backtrack(candidates, 0, target, 0);
        return res;
    }

    // 记录回溯的路径
    LinkedList<Integer> track = new LinkedList<>();

    // 回溯算法主函数
    void backtrack(int[] candidates, int start, int target, int sum) {
        if (sum == target) {
            // 找到目标和
            res.add(new LinkedList<>(track));
            return;
        }

        if (sum > target) {
            // 超过目标和，直接结束
            return;
        }

        // 回溯算法框架
        for (int i = start; i < candidates.length; i++) {
            // 选择 candidates[i]
            track.add(candidates[i]);
            sum += candidates[i];
```

```
                // 递归遍历下一层回溯树
                backtrack(candidates, i, target, sum);
                // 撤销选择 candidates[i]
                sum -= candidates[i];
                track.removeLast();
            }
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 216. 组合总和 III 🟠
  - 40. 组合总和 II 🟠
  - 46. 全排列 🟠
  - 47. 全排列 II 🟠
  - 77. 组合 🟠
  - 78. 子集 🟠
  - 90. 子集 II 🟠
  - 剑指 Offer II 079. 所有子集 🟠
  - 剑指 Offer II 080. 含有 k 个元素的组合 🟠
  - 剑指 Offer II 081. 允许重复选择元素的组合 🟠
  - 剑指 Offer II 082. 含有重复元素集合的组合 🟠
  - 剑指 Offer II 083. 没有重复元素集合的全排列 🟠
  - 剑指 Offer II 084. 含有重复元素集合的全排列 🟠

46. 全排列

LeetCode	力扣	难度
46. Permutations	46. 全排列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法**

给定一个**不含重复数字**的数组 `nums`，返回其所有全排列，你可以按任意顺序返回答案。

示例 1：

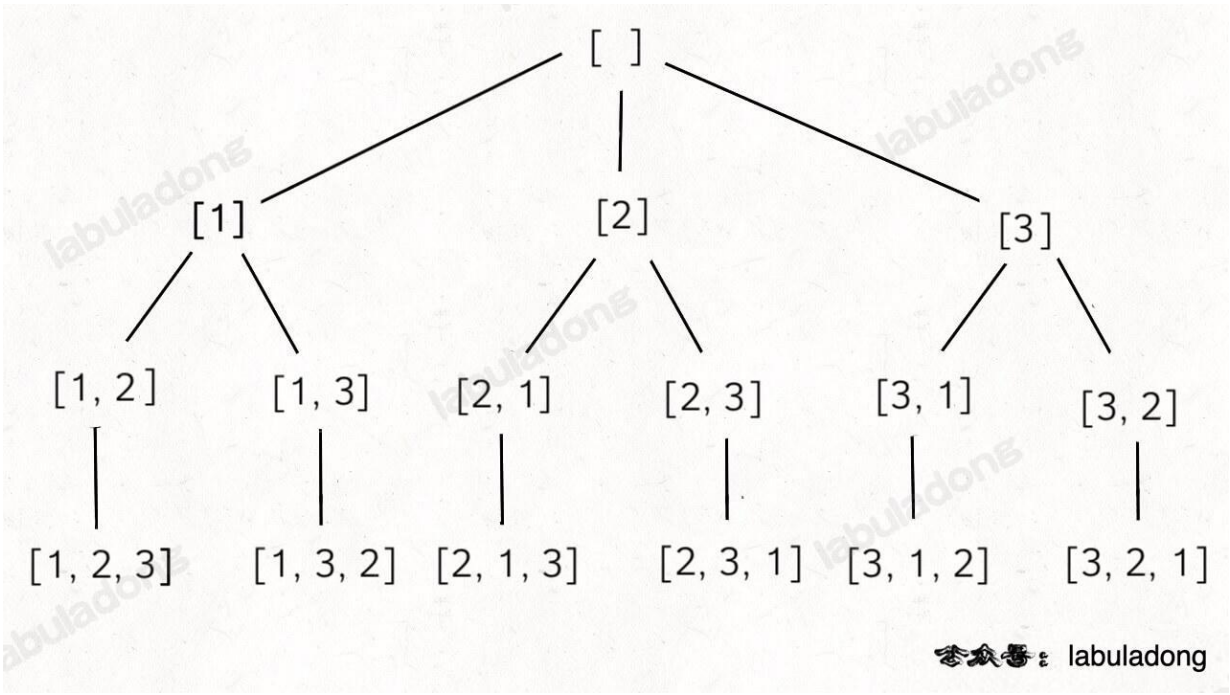
```
输入：nums = [1,2,3]
输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

基本思路

> 本文有视频版：回溯算法秒杀所有排列/组合/子集问题

PS：这道题在《算法小抄》 的第 43 页。

回溯算法详解 中就是拿这个问题来解释回溯模板的，首先画出回溯树来看一看：



写代码遍历这棵回溯树即可。

- **详细题解：回溯算法秒杀所有排列/组合/子集问题**

解法代码

```java
class Solution {

    List<List<Integer>> res = new LinkedList<>();

    /* 主函数，输入一组不重复的数字，返回它们的全排列 */
    List<List<Integer>> permute(int[] nums) {
        // 记录「路径」
        LinkedList<Integer> track = new LinkedList<>();
        //  「路径」中的元素会被标记为 true，避免重复使用
        boolean[] used = new boolean[nums.length];

        backtrack(nums, track, used);
        return res;
    }

    // 路径：记录在 track 中
    // 选择列表：nums 中不存在于 track 的那些元素（used[i] 为 false）
    // 结束条件：nums 中的元素全都在 track 中出现
    void backtrack(int[] nums, LinkedList<Integer> track, boolean[] used)
{
        // 触发结束条件
        if (track.size() == nums.length) {
            res.add(new LinkedList(track));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            // 排除不合法的选择
            if (used[i]) {
                // nums[i] 已经在 track 中，跳过
                continue;
            }
            // 做选择
            track.add(nums[i]);
            used[i] = true;
            // 进入下一层决策树
            backtrack(nums, track, used);
            // 取消选择
            track.removeLast();
            used[i] = false;
        }
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 216. 组合总和 III 🟠
  - 39. 组合总和 🟠
  - 40. 组合总和 II 🟠

- 47. 全排列 II 🟠
- 51. N 皇后 🔴
- 77. 组合 🟠
- 78. 子集 🟠
- 90. 子集 II 🟠
- 剑指 Offer II 079. 所有子集 🟠
- 剑指 Offer II 080. 含有 k 个元素的组合 🟠
- 剑指 Offer II 081. 允许重复选择元素的组合 🟠
- 剑指 Offer II 082. 含有重复元素集合的组合 🟠
- 剑指 Offer II 083. 没有重复元素集合的全排列 🟠
- 剑指 Offer II 084. 含有重复元素集合的全排列 🟠

77. 组合

LeetCode	力扣	难度
77. Combinations	77. 组合	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，数学**

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。你可以按**任何顺序**返回答案。

示例 1：

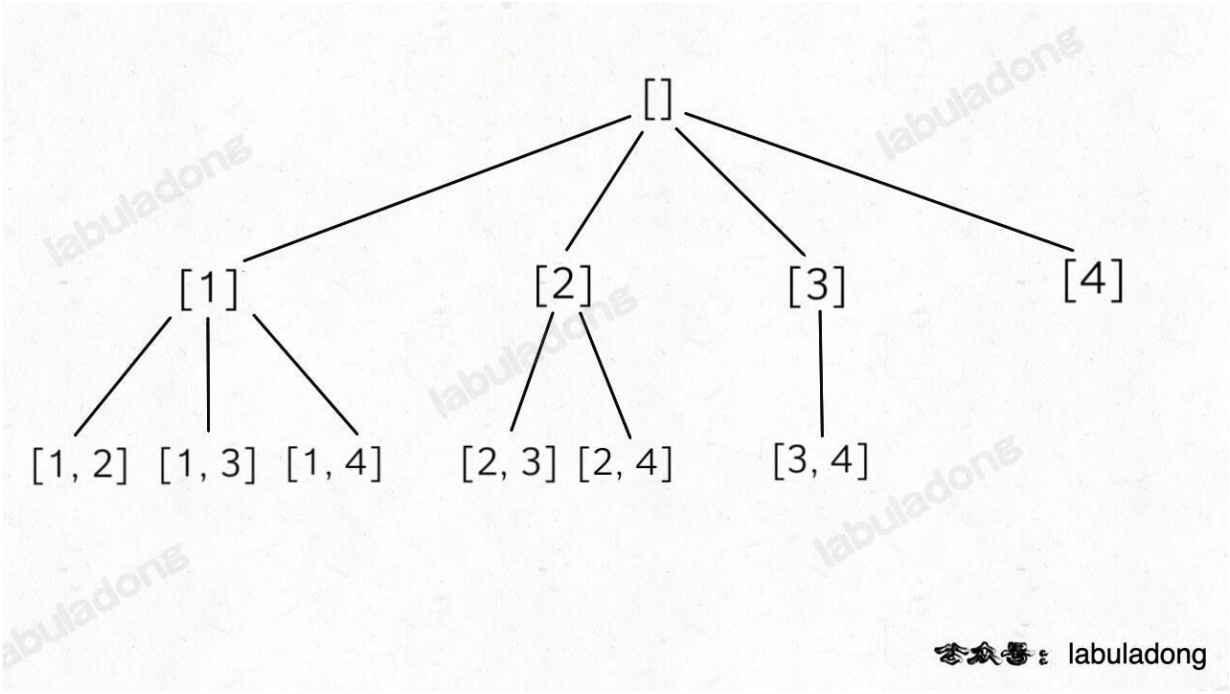
```
输入：n = 4, k = 2
输出：
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

基本思路

> 本文有视频版：回溯算法秒杀所有排列/组合/子集问题

PS：这道题在《算法小抄》 的第 293 页。

这也是典型的回溯算法，`k` 限制了树的高度，`n` 限制了树的宽度，继续套我们以前讲过的 回溯算法模板框架 就行了：



- **详细题解：回溯算法秒杀所有排列/组合/子集问题**

解法代码

```
class Solution {
    public:

    vector<vector<int>>res;
    vector<vector<int>> combine(int n, int k) {
        if (k <= 0 || n <= 0) return res;
        vector<int> track;
        backtrack(n, k, 1, track);
        return res;
    }

    void backtrack(int n, int k, int start, vector<int>& track) {
        // 到达树的底部
        if (k == track.size()) {
            res.push_back(track);
            return;
        }
        // 注意 i 从 start 开始递增
        for (int i = start; i <= n; i++) {
            // 做选择
            track.push_back(i);
            backtrack(n, k, i + 1, track);
            // 撤销选择
            track.pop_back();
        }
    }
};
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目：**
  - 216. 组合总和 III 🟠
  - 39. 组合总和 🟠
  - 40. 组合总和 II 🟠
  - 46. 全排列 🟠
  - 47. 全排列 II 🟠
  - 78. 子集 🟠
  - 90. 子集 II 🟠
  - 剑指 Offer II 079. 所有子集 🟠
  - 剑指 Offer II 080. 含有 k 个元素的组合 🟠
  - 剑指 Offer II 081. 允许重复选择元素的组合 🟠
  - 剑指 Offer II 082. 含有重复元素集合的组合 🟠
  - 剑指 Offer II 083. 没有重复元素集合的全排列 🟠
  - 剑指 Offer II 084. 含有重复元素集合的全排列 🟠

78. 子集

LeetCode	力扣	难度
78. Subsets	78. 子集	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，数学**

给你一个整数数组 `nums`，数组中的元素**互不相同**，返回该数组所有可能的子集（幂集）。

解集**不能**包含重复的子集。你可以按**任意顺序**返回解集。

示例 1：

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

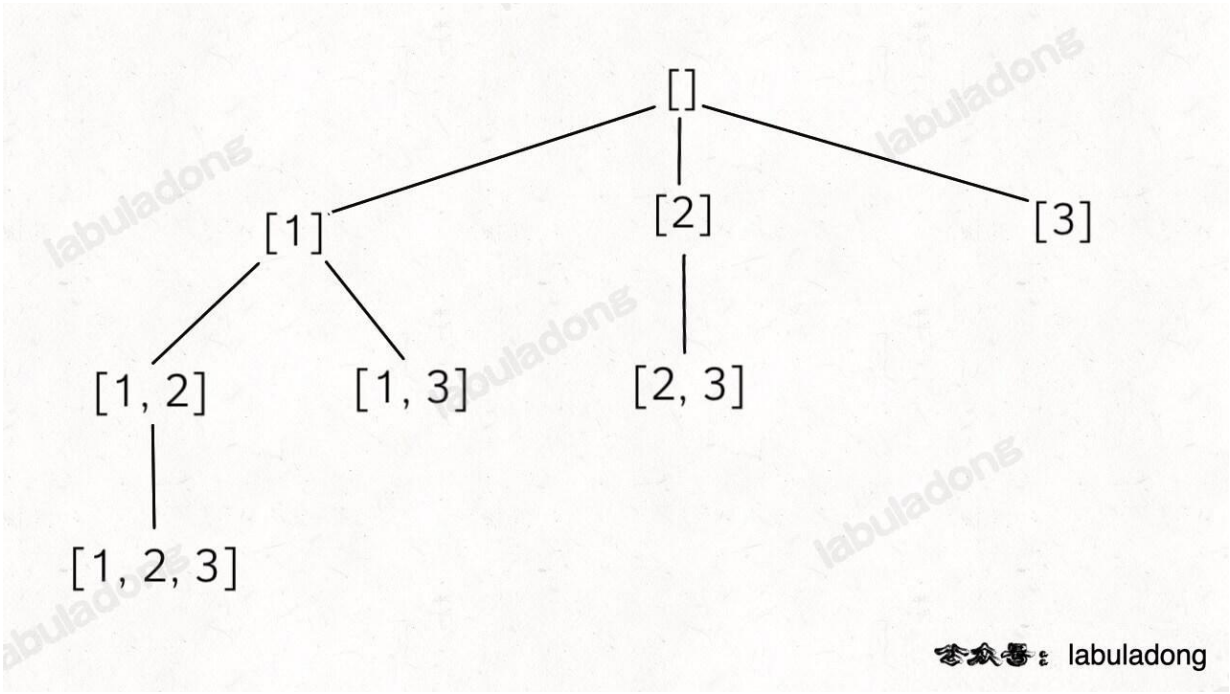
基本思路

> 本文有视频版：回溯算法秒杀所有排列/组合/子集问题

PS：这道题在《算法小抄》 的第 293 页。

有两种方法解决这道题，这里主要说回溯算法思路，因为比较通用，可以套前文 回溯算法详解 写过回溯算法模板。

本质上子集问题就是遍历这样用一棵回溯树：



- **详细题解：回溯算法秒杀所有排列/组合/子集问题**

解法代码

```
class Solution {
    public:
    vector<vector<int>> res;
    vector<vector<int>> subsets(vector<int>& nums) {
        // 记录走过的路径
        vector<int> track;
        backtrack(nums, 0, track);
        return res;
    }

    void backtrack(vector<int>& nums, int start, vector<int>& track) {
        res.push_back(track);
        for (int i = start; i < nums.size(); i++) {
            // 做选择
            track.push_back(nums[i]);
            // 回溯
            backtrack(nums, i + 1, track);
            // 撤销选择
            track.pop_back();
        }
    }
};
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目：**
  - 216. 组合总和 III 🟠
  - 39. 组合总和 🟠
  - 40. 组合总和 II 🟠
  - 46. 全排列 🟠
  - 47. 全排列 II 🟠
  - 77. 组合 🟠
  - 90. 子集 II 🟠
  - 剑指 Offer II 079. 所有子集 🟠
  - 剑指 Offer II 080. 含有 k 个元素的组合 🟠
  - 剑指 Offer II 081. 允许重复选择元素的组合 🟠
  - 剑指 Offer II 082. 含有重复元素集合的组合 🟠
  - 剑指 Offer II 083. 没有重复元素集合的全排列 🟠
  - 剑指 Offer II 084. 含有重复元素集合的全排列 🟠

剑指 Offer II 079. 所有子集

这道题和 78. 子集 相同。

剑指 Offer II 080. 含有 k 个元素的组合

这道题和 77. 组合 相同。

剑指 Offer II 081. 允许重复选择元素的组合

这道题和 39. 组合总和 相同。

剑指 Offer II 083. 没有重复元素集合的全排列

这道题和 46. 全排列 相同。

51. N 皇后

LeetCode	力扣	难度
51. N-Queens	51. N 皇后	🔴

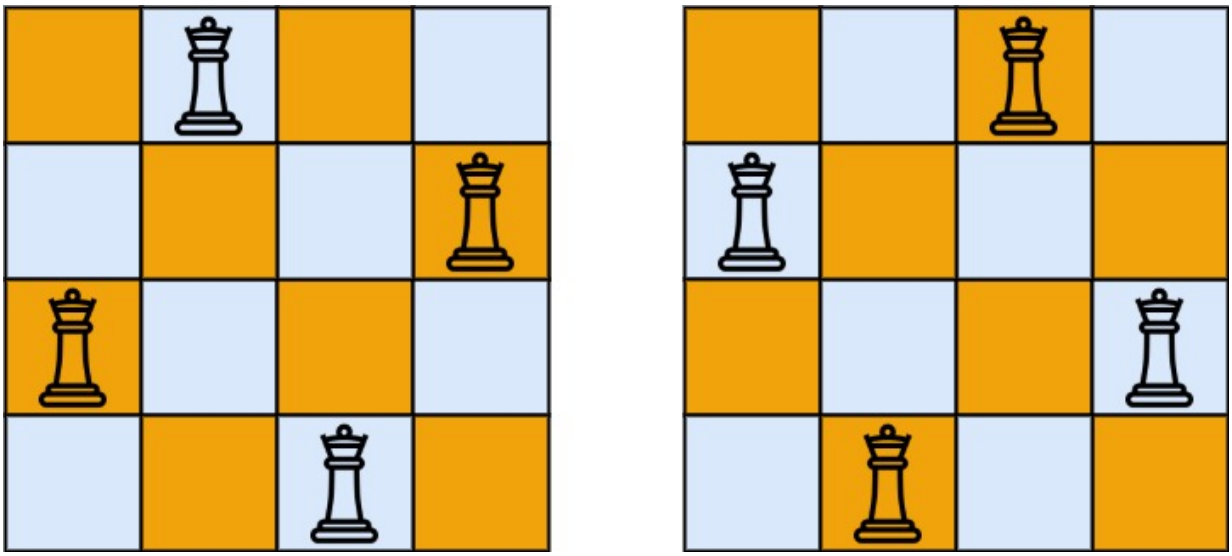
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：回溯算法

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。现在输入一个整数 n，请你返回所有不同的 **n 皇后问题** 的解决方案。

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

示例 1：



```
输入：n = 4
输出：[[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
解释：如上图所示，4 皇后问题存在两个不同的解法。
```

基本思路

PS：这道题在《算法小抄》 的第 43 页。

视频讲解回溯算法原理：回溯算法框架套路详解

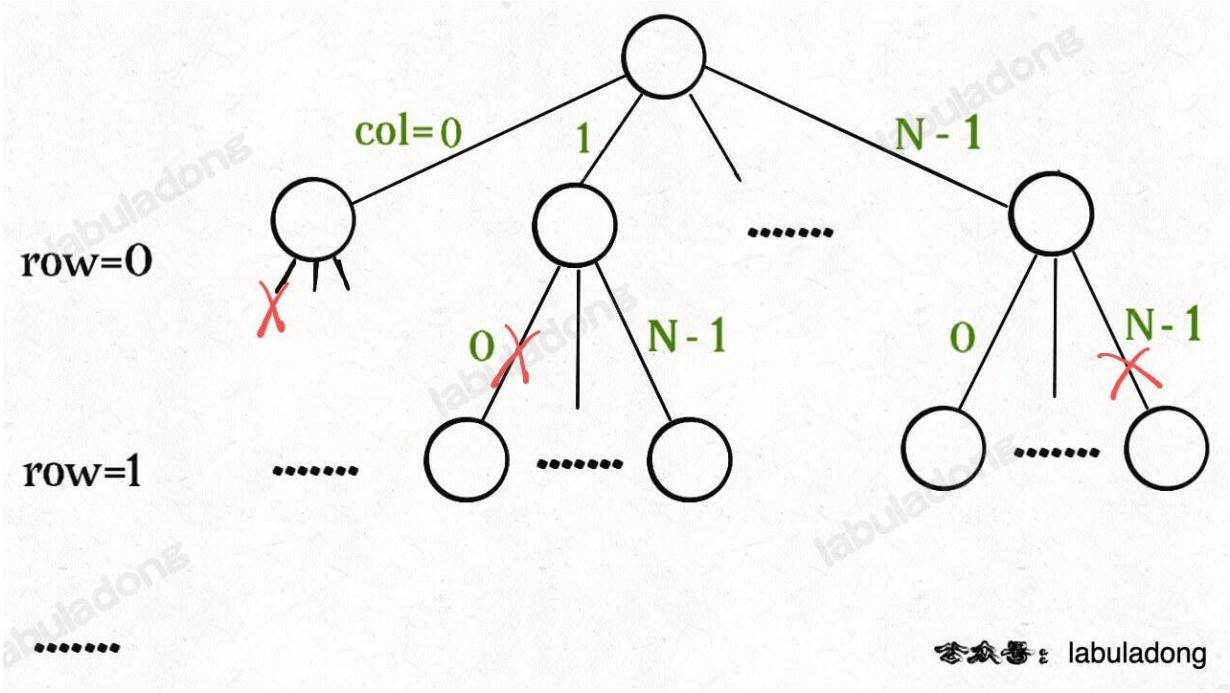
N 皇后问题就是一个决策问题：对于每一行，我应该选择在哪一列防止皇后呢？

这就是典型的回溯算法题目，回溯算法的框架如下：

```
result = []
def backtrack(路径, 选择列表):
```

```
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

回溯算法框架就是遍历决策树的过程：



关于回溯算法的详细讲解可以看 46. 全排列 或者详细题解。

- **详细题解：回溯算法解题套路框架**

解法代码

```
class Solution {
    public:
    vector<vector<string>> res;

    /* 输入棋盘边长 n，返回所有合法的放置 */
    vector<vector<string>> solveNQueens(int n) {
        // '.' 表示空，'Q' 表示皇后，初始化空棋盘。
        vector<string> board(n, string(n, '.'));
        backtrack(board, 0);
        return res;
    }

    // 路径：board 中小于 row 的那些行都已经成功放置了皇后
    // 选择列表：第 row 行的所有列都是放置皇后的选择
    // 结束条件：row 超过 board 的最后一行
```

```
    void backtrack(vector<string>& board, int row) {
        // 触发结束条件
        if (row == board.size()) {
            res.push_back(board);
            return;
        }

        int n = board[row].size();
        for (int col = 0; col < n; col++) {
            // 排除不合法选择
            if (!isValid(board, row, col)) {
                continue;
            }
            // 做选择
            board[row][col] = 'Q';
            // 进入下一行决策
            backtrack(board, row + 1);
            // 撤销选择
            board[row][col] = '.';
        }
    }

    /* 是否可以在 board[row][col] 放置皇后? */
    bool isValid(vector<string>& board, int row, int col) {
        int n = board.size();
        // 检查列是否有皇后互相冲突
        for (int i = 0; i <= row; i++) {
            if (board[i][col] == 'Q')
                return false;
        }
        // 检查右上方是否有皇后互相冲突
        for (int i = row - 1, j = col + 1;
             i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q')
                return false;
        }
        // 检查左上方是否有皇后互相冲突
        for (int i = row - 1, j = col - 1;
             i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q')
                return false;
        }
        return true;
    }
};
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- 类似题目：
  - 46. 全排列🟠
  - 52. N皇后 II🔴

- 剑指 Offer II 083. 没有重复元素集合的全排列 🟠

104. 二叉树的最大深度

LeetCode	力扣	难度
104. Maximum Depth of Binary Tree	104. 二叉树的最大深度	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，动态规划，回溯算法**

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，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。

基本思路

> 本文有视频版：二叉树/递归的框架思维（纲领篇）

我的刷题经验总结 说过，二叉树问题虽然简单，但是暗含了动态规划和回溯算法等高级算法的思想。

下面提供两种思路的解法代码。

- **详细题解：东哥带你刷二叉树（纲领篇）**

解法代码

```
/***** 解法一，回溯算法思路 *****/
class Solution {

    int depth = 0;
    int res = 0;

    public int maxDepth(TreeNode root) {
        traverse(root);
        return res;
    }
```

```
    // 遍历二叉树
    void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        // 前序遍历位置
        depth++;
        // 遍历的过程中记录最大深度
        res = Math.max(res, depth);
        traverse(root.left);
        traverse(root.right);
        // 后序遍历位置
        depth--;
    }
}

/***** 解法二，动态规划思路 *****/
class Solution2 {
    // 定义：输入一个节点，返回以该节点为根的二叉树的最大深度
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);
        // 根据左右子树的最大深度推出原二叉树的最大深度
        return 1 + Math.max(leftMax, rightMax);
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1376. 通知所有员工所需的时间 🟠
  - 144. 二叉树的前序遍历 🟢
  - 543. 二叉树的直径 🟢
  - 559. N 叉树的最大深度 🟢
  - 865. 具有所有最深节点的最小子树 🟠
  - 剑指 Offer 55 - I. 二叉树的深度 🟢

1376. 通知所有员工所需的时间

LeetCode	力扣	难度
1376. Time Needed to Inform All Employees	1376. 通知所有员工所需的时间	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip

公司里有 `n` 名员工，每个员工的 ID 都是独一无二的，编号从 `0` 到 `n - 1`。公司的总负责人通过 `headID` 进行标识。

在 `manager` 数组中，每个员工都有一个直属负责人，其中 `manager[i]` 是第 `i` 名员工的直属负责人。对于总负责人，`manager[headID] = -1`。题目保证从属关系可以用树结构显示。

公司总负责人想要向公司所有员工通告一条紧急消息。他将会首先通知他的直属下属们，然后由这些下属通知他们的下属，直到所有的员工都得知这条紧急消息。

第 `i` 名员工需要 `informTime[i]` 分钟来通知它的所有直属下属（也就是说在 `informTime[i]` 分钟后，他的所有直属下属都可以开始传播这一消息）。

返回通知所有员工这一紧急消息所需要的 **分钟数**。

示例 1：

```
输入：n = 1, headID = 0, manager = [-1], informTime = [0]
输出：0
解释：公司总负责人是该公司的唯一一名员工。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维模式。

我想这道题不用多说，其实稍微变通一下，这道题就是让你求多叉树的最大深度，只不过标准的 104. 二叉树的最大深度 每个节点的高度默认是 1，而这道题每个节点的高度记录在 `informTime` 中。

所以我们需要先将题目的输入数据格式转化成多叉树的形式，然后写一个函数计算多叉树的最大深度，具体看代码注释吧。

解法代码

```
class Solution {
    // 多叉树结构，key 是父节点，value 是子节点列表
    HashMap<Integer, List<Integer>> tree = new HashMap<>();
    int[] informTime;
```

```
    public int numOfMinutes(int n, int headID, int[] manager, int[]
informTime) {
        // 构造多叉树结构
        for (int i = 0; i < manager.length; i++) {
            int parent = manager[i];
            if (!tree.containsKey(parent)) {
                tree.put(parent, new ArrayList<>());
            }
            tree.get(parent).add(i);
        }
        this.informTime = informTime;
        return maxDepth(headID);
    }

    // 计算多叉树的最大深度
    int maxDepth(int root) {
        if (!tree.containsKey(root)) {
            // 叶子节点的深度为 0
            return 0;
        }

        int subMaxDepth = 0;
        for (int child : tree.get(root)) {
            subMaxDepth = Math.max(subMaxDepth, maxDepth(child));
        }
        // 以当前节点为根的多叉树的最大深度等于：
        // 当前节点的树枝长度 + 所有子树中最大深度的最大值
        return informTime[root] + subMaxDepth;
    }
}
```

559. N 叉树的最大深度

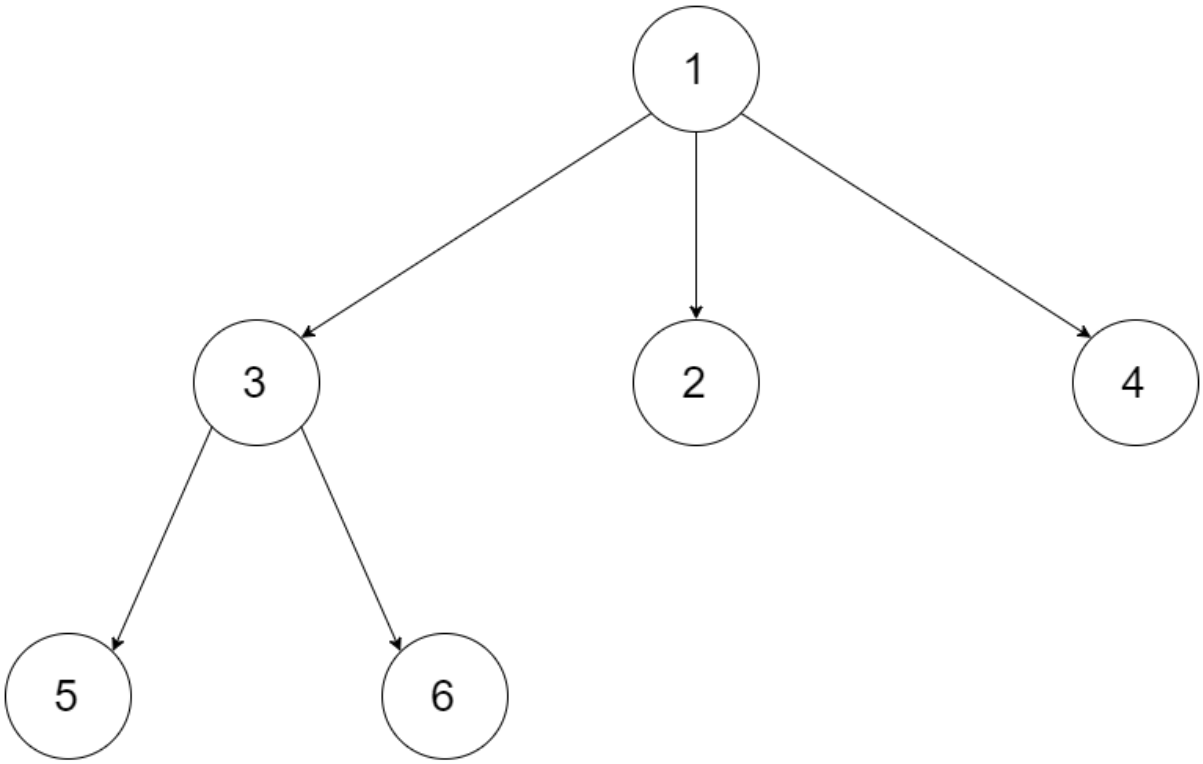
LeetCode	力扣	难度
559. Maximum Depth of N-ary Tree	559. N 叉树的最大深度	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树

给定一个 N 叉树，找到其最大深度。最大深度是指从根节点到最远叶子节点的最长路径上的节点总数。

示例 1：




```
输入：root = [1,null,3,2,4,null,5,6]
输出：3
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题可以同时使用两种思维模式，我把两种解法都写一下。

可以对照 104. 二叉树的最大深度 题的解法。

解法代码

```java
// 分解问题的思路
class Solution {
    public int maxDepth(Node root) {
        if (root == null) {
            return 0;
        }
        int subTreeMaxDepth = 0;
        for (Node child : root.children) {
            subTreeMaxDepth = Math.max(subTreeMaxDepth, maxDepth(child));
        }
        return 1 + subTreeMaxDepth;
    }
}

// 遍历的思路
class Solution2 {
    public int maxDepth(Node root) {
        traverse(root);
        return res;
    }

    // 记录递归遍历到的深度
    int depth = 0;
    // 记录最大的深度
    int res = 0;

    void traverse(Node root) {
        if (root == null) {
            return;
        }
        // 前序遍历位置
        depth++;
        res = Math.max(res, depth);

        for (Node child : root.children) {
            traverse(child);
        }
        // 后序遍历位置
        depth--;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

865. 具有所有最深节点的最小子树

LeetCode	力扣	难度
865. Smallest Subtree with all the Deepest Nodes	865. 具有所有最深节点的最小子树	

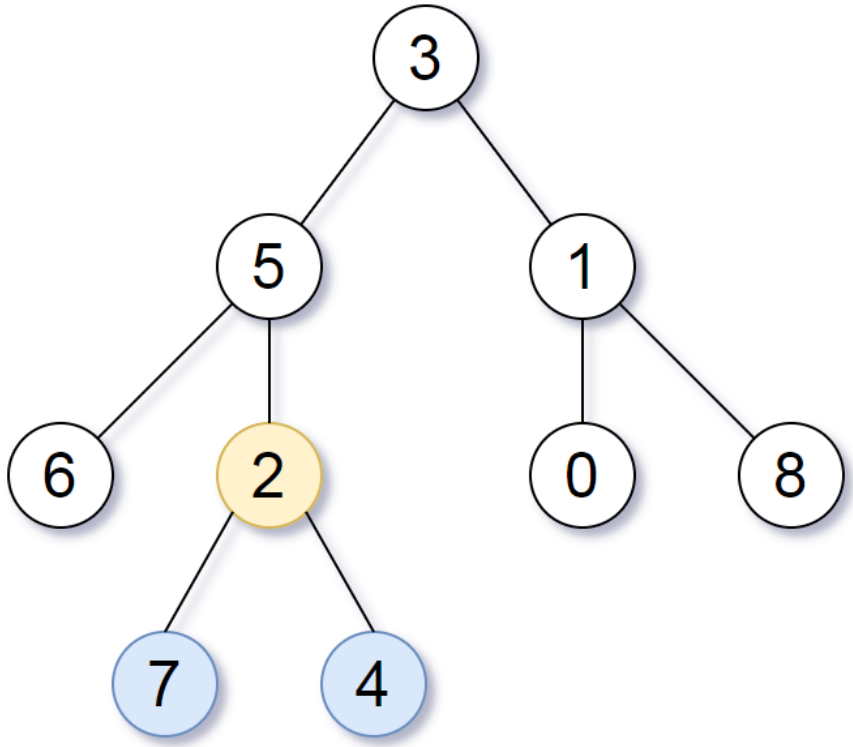
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉搜索树，二叉树vip

给定一个根为 root 的二叉树，每个节点的深度是**该节点到根的最短距离**。

返回能满足**以该节点为根的子树中包含所有最深的节点**这一条件的具有最大深度的节点。

示例 1：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4]
输出：[2,7,4]
解释：
我们返回值为 2 的节点，在图中用黄色标记。
在图中用蓝色标记的是树的最深的节点。
注意，节点 5、3 和 2 包含树中最深的节点，但节点 2 的子树最小，因此我们返回它。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「分解问题」的思维，而且涉及处理子树，需要用后序遍历。

说到底，这道题就是让你求那些「最深」的叶子节点的最近公共祖先，可以看下前文 二叉树最近公共祖先。

你想想，一个节点需要知道哪些信息，才能确定自己是最深叶子节点的最近公共祖先?

它需要知道自己的左右子树的最大深度：如果左右子树一样深，那么当前节点就是最近公共祖先；如果左右子树不一样深，那么最深叶子节点的最近公共祖先肯定在左右子树上。

所以我们新建一个 `Result` 类，存放左右子树的最大深度及叶子节点的最近公共祖先节点，其余逻辑类似 104. 二叉树的最大深度。

解法代码

```
class Solution {
    class Result {
        public TreeNode node;
        public int depth;

        public Result(TreeNode node, int depth) {
            // 记录最近公共祖先节点 node
            this.node = node;
            // 记录以 node 为根的二叉树最大深度
            this.depth = depth;
        }
    }

    public TreeNode subtreeWithAllDeepest(TreeNode root) {
        Result res = maxDepth(root);
        return res.node;
    }

    // 定义：输入一棵二叉树，返回该二叉树的最大深度以及最深叶子节点的最近公共祖先节点
    Result maxDepth(TreeNode root) {
        if (root == null) {
            return new Result(null, 0);
        }
        Result left = maxDepth(root.left);
        Result right = maxDepth(root.right);
        if (left.depth == right.depth) {
            // 当左右子树的最大深度相同时，这个根节点是新的最近公共祖先
            // 以当前 root 节点为根的子树深度是子树深度 + 1
            return new Result(root, left.depth + 1);
        }
        // 左右子树的深度不同，则最近公共祖先在 depth 较大的一边
        Result res = left.depth > right.depth ? left : right;
        // 正确维护二叉树的最大深度
        res.depth++;

        return res;
```

```
        }
    }
```

▶ 🐭 代码可视化动画（需安装 Chrome 插件）🐭

- **类似题目**：
  - 1123. 最深叶节点的最近公共祖先 🟠

剑指 Offer 55 - I. 二叉树的深度

这道题和 104. 二叉树的最大深度 相同。

1123. 最深叶节点的最近公共祖先

LeetCode	力扣	难度
1123. Lowest Common Ancestor of Deepest Leaves	1123. 最深叶节点的最近公共祖先	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉树，二叉树vip**

给你一个有根节点的二叉树，找到它最深的叶节点的最近公共祖先。

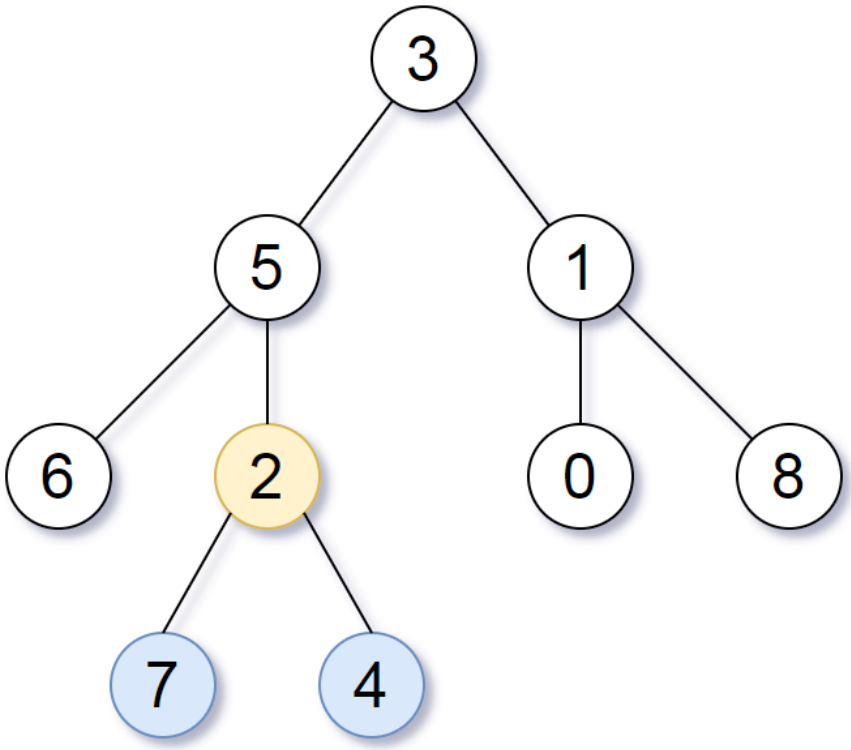
回想一下：

1、**叶节点**是二叉树中没有子节点的节点

2、树的根节点的**深度**为 `0`，如果某一节点的深度为 `d`，那它的子节点的深度就是 `d+1`

3、如果我们假定 `A` 是一组节点 `S` 的**最近公共祖先**，`S` 中的每个节点都在以 `A` 为根节点的子树中，且 `A` 的深度达到此条件下可能的最大值。

示例 1：




```
输入：root = [3,5,1,6,2,0,8,null,null,7,4]
输出：[2,7,4]
解释：
我们返回值为 2 的节点，在图中用黄色标记。
```

在图中用蓝色标记的是树的最深的节点。
注意，节点 6、0 和 8 也是叶节点，但是它们的深度是 2，而节点 7 和 4 的深度是 3。

基本思路

这题和 865. 具有所有最深节点的最小子树 一模一样，思路见那道题。

解法代码

```
class Solution {
    class Result {
        public TreeNode node;
        public int depth;

        public Result(TreeNode node, int depth) {
            // 记录最近公共祖先节点 node
            this.node = node;
            // 记录以 node 为根的二叉树最大深度
            this.depth = depth;
        }
    }

    public TreeNode lcaDeepestLeaves(TreeNode root) {
        Result res = maxDepth(root);
        return res.node;
    }

    // 定义：输入一棵二叉树，返回该二叉树的最大深度以及最深叶子节点的最近公共祖先节点
    Result maxDepth(TreeNode root) {
        if (root == null) {
            return new Result(null, 0);
        }
        Result left = maxDepth(root.left);
        Result right = maxDepth(root.right);
        if (left.depth == right.depth) {
            // 当左右子树的最大深度相同时，这个根节点是新的最近公共祖先
            return new Result(root, left.depth + 1);
        }
        // 左右子树的深度不同，则最近公共祖先在 depth 较大的一边
        Result res = left.depth > right.depth ? left : right;
        // 正确维护二叉树的最大深度
        res.depth++;

        return res;
    }
}
```

▶ 代码可视化动画（需安装 Chrome 插件）

293. 翻转游戏

LeetCode	力扣	难度
293. Flip Game	293. 翻转游戏	🟢

- 标签：回溯算法，字符串

你和朋友玩一个叫做「翻转游戏」的游戏。游戏规则如下：

给你一个字符串 `currentState`，其中只含 `'+'` 和 `'-'`。你和朋友轮流将 **连续** 的两个 `"++"` 反转成 `"--"`。当一方无法进行有效的翻转时便意味着游戏结束，则另一方获胜。

计算并返回 **一次有效操作** 后，字符串 `currentState` 所有的可能状态，返回结果可以按 **任意顺序**排列。如果不存在可能的有效操作，请返回一个空列表 `[]`。

示例 1：

```
输入：currentState = "++++"
输出：["--++","+--+","++--"]
```

基本思路

这道题很简单，我们会用这道题的解法去解决进阶题目 294. 翻转游戏 II。

解法代码

```
class Solution {
    public List<String> generatePossibleNextMoves(String currentState) {
        List<String> res = new ArrayList<>();
        char[] arr = currentState.toCharArray();
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] == '+' && arr[i - 1] == '+') {
                // 做选择
                arr[i] = '-';
                arr[i - 1] = '-';
                res.add(new String(arr));
                // 撤销选择
                arr[i] = '+';
                arr[i - 1] = '+';
            }
        }
        return res;
    }
}
```

294. 翻转游戏 II

LeetCode	力扣	难度
294. Flip Game II	294. 翻转游戏 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，字符串**

你和朋友玩一个叫做「翻转游戏」的游戏。游戏规则如下：

给你一个字符串 `currentState`，其中只含 `'+'` 和 `'-'`。你和朋友轮流将 **连续** 的两个 `"++"` 反转成 `"--"`。当一方无法进行有效的翻转时便意味着游戏结束，则另一方获胜。默认每个人都会采取最优策略。

请你写出一个函数来判定起始玩家 **是否存在必胜的方案**：如果存在，返回 `true`；否则，返回 `false`。

示例 1：

```
输入：currentState = "++++"
输出：true
解释：起始玩家可将中间的 "++" 翻转变为 "+--+" 从而得胜。
```

基本思路

看到两个人互相博弈，而且还说每个人会使用最优策略，你是不是就感觉有点无从下手了？其实那些都是障眼法，算法的本质 说过，计算机唯一会做的就是穷举，它那个所谓最优策略，也是穷举出来的，所以你不要把这种问题想的太复杂，暴力穷举就完事儿了。

这题要用递归进行暴力穷举，一切暴力穷举皆出自二叉树，这里要用到 手把手刷二叉树总结篇 说过的后序位置的妙用，因此要采取「分解问题」的思路。

为啥要用到后序位置，因为对于一个字符串局面，比方说 `++--++`，当前的先手是否可以赢，需要穷举所有可能可能才能知道。换句话说，你要遍历完所有子树才能知道结果，这就必须是后序位置。

一个递归树胜过千言万语，我直接用 算法可视化面板 把递归树画出来，你啥都明白了：

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

在这个可视化的代码中，我故意把两个剪枝逻辑注释掉了，这样你可以看到完整的递归树。你理解了穷举过程后，可以自己尝试把这两个剪枝逻辑恢复，看看剪枝对递归树的优化。

解法代码

```
class Solution {
```

```
    // 直接把 293 的代码 copy 过来，生成所有可能的下一步
    List<String> generatePossibleNextMoves(String currentState) {
        List<String> res = new ArrayList<>();
        char[] arr = currentState.toCharArray();
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] == '+' && arr[i - 1] == '+') {
                // 做选择
                arr[i] = '-';
                arr[i - 1] = '-';
                res.add(new String(arr));
                // 撤销选择
                arr[i] = '+';
                arr[i - 1] = '+';
            }
        }
        return res;
    }

    // 备忘录
    Map<String, Boolean> memo = new HashMap<>();

    public boolean canWin(String currentState) {
        // 记入备忘录
        memo.clear();
        return dp(currentState);
    }

    // 定义：输入字符串 s，返回在此局面下先手是否可能赢
    boolean dp(String s) {
        if (memo.containsKey(s)) {
            // 之前遇到过这种局面，直接返回结果
            return memo.get(s);
        }
        boolean res = false;
        List<String> nextStates = generatePossibleNextMoves(s);
        for (String nextState : nextStates) {
            // 只要有一个是 false，说明当前玩家赢了
            boolean win = dp(nextState);
            // 后序位置，做判断
            if (!win) {
                res = true;
                break;
            }
        }
        // 记入备忘录
        memo.put(s, res);
        return res;
    }
}
```

- **类似题目**：
  - 293. 翻转游戏 🟢

491. 递增子序列

LeetCode	力扣	难度
491. Increasing Subsequences	491. 递增子序列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，排列组合**

给你一个整数数组 `nums`，找出并返回所有该数组中不同的递增子序列，递增子序列中**至少有两个元素**，你可以按 **任意顺序** 返回答案。

数组中可能含有重复元素，如出现两个整数相等，也可以视作递增序列的一种特殊情况。

示例 1：

```
输入：nums = [4,6,7,7]
输出：[[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]
```

基本思路

其实这道题就是前文 一文秒杀所有排列组合子集问题 中讲过的「元素可重不可复选」的组合问题。

只不过这里又有一个额外的条件：组合中所有元素都必须是递增顺序。我们只需要添加一个 `track.getLast() > nums[i]` 条件即可。

另外，一文秒杀所有排列组合子集问题 中是利用排序的方式防止重复使用相同元素的，但这道题不能改变 `nums` 的原始顺序，所以不能用排序的方式，而是用 `used` 集合来避免重复使用相同元素。

综上，只要把 40. 组合总和 II 的解法稍改一下即可完成这道题。

解法代码

```
class Solution {
    public List<List<Integer>> findSubsequences(int[] nums) {
        if (nums.length == 0) {
            return res;
        }
        backtrack(nums, 0);
        return res;
    }

    List<List<Integer>> res = new LinkedList<>();
    // 记录回溯的路径
    LinkedList<Integer> track = new LinkedList<>();
```

```
    // 回溯算法主函数
    void backtrack(int[] nums, int start) {
        if (track.size() >= 2) {
            // 找到一个合法答案
            res.add(new LinkedList<>(track));
        }
        // 用哈希集合防止重复选择相同元素
        HashSet<Integer> used = new HashSet<>();
        // 回溯算法标准框架
        for (int i = start; i < nums.length; i++) {
            // 保证集合中元素都是递增顺序
            if (!track.isEmpty() && track.getLast() > nums[i]) {
                continue;
            }
            // 保证不要重复使用相同的元素
            if (used.contains(nums[i])) {
                continue;
            }
            // 选择 nums[i]
            used.add(nums[i]);
            track.add(nums[i]);
            // 递归遍历下一层回溯树
            backtrack(nums, i + 1);
            // 撤销选择 nums[i]
            track.removeLast();
        }
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

494. 目标和

LeetCode	力扣	难度
494. Target Sum	494. 目标和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，回溯算法，背包问题**

给你一个整数数组 `nums` 和一个整数 `target`，向数组中的每个整数前添加 `'+'` 或 `'-'`，然后串联起所有整数，可以构造一个**表达式**。

例如，`nums = [2, 1]`，可以在 `2` 之前添加 `'+'`，在 `1` 之前添加 `'-'`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同**表达式**的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 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
+1 + 1 + 1 + 1 - 1 = 3
```

基本思路

PS：这道题在《算法小抄》 的第 207 页。

这题有多种解法，可以用回溯算法剪枝求解，也可以用转化成背包问题求解，这里用前者吧，容易理解一些，背包问题解法可以查看详细题解。

对于每一个 1，要么加正号，要么加负号，把所有情况穷举出来，即可计算结果。

- **详细题解：目标和问题：背包问题的变体**

解法代码

```
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        if (nums.length == 0) return 0;
        return dp(nums, 0, target);
    }
```

```
    // 备忘录
    HashMap<String, Integer> memo = new HashMap<>();

    int dp(int[] nums, int i, int remain) {
        // base case
        if (i == nums.length) {
            if (remain == 0) return 1;
            return 0;
        }
        // 把它俩转成字符串才能作为哈希表的键
        String key = i + "," + remain;
        // 避免重复计算
        if (memo.containsKey(key)) {
            return memo.get(key);
        }
        // 还是穷举
        int result = dp(nums, i + 1, remain - nums[i]) + dp(nums, i + 1,
remain + nums[i]);
        // 记入备忘录
        memo.put(key, result);
        return result;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 剑指 Offer II 102. 加减的目标值 🟠

剑指 Offer II 102. 加减的目标值

这道题和 494. 目标和 相同。

698. 划分为 k 个相等的子集

LeetCode	力扣	难度
698. Partition to K Equal Sum Subsets	698. 划分为k个相等的子集	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法**

给定一个整数数组 `nums` 和一个正整数 `k`，找出是否有可能把这个数组分成 `k` 个非空子集，其总和都相等。

示例 1：

```
输入：nums = [4, 3, 2, 3, 5, 2, 1], k = 4
输出：True
说明：有可能将其分成 4 个子集 {5}, {1,4}, {2,3}, {2,3} 等于总和。
```

基本思路

回溯算法是笔试中最好用的算法，只要你没什么思路，就用回溯算法暴力求解，即便不能通过所有测试用例，多少能过一点。

这道题的解法其实就是暴力穷举所有的子集划分方式，看看有没有符合题意的划分方法。详细题解讲解了两种穷举思路，分别是以数字的角度和子集的角度进行穷举，这里只讲后者，因为效率较高。

以桶的视角进行穷举，每个桶需要遍历 `nums` 中的所有数字，决定是否把当前数字装进桶中；当装满一个桶之后，还要装下一个桶，直到所有桶都装满为止。

按照这个逻辑，结合 回溯算法框架，就能写出 `backtrack` 函数了。

当然，单纯暴力穷举会出现冗余计算，所以我们需要借助备忘录进行剪枝。如果你想进一步提高算法效率，还需要运用位图技巧优化空间复杂度，建议看详细题解。

- **详细题解：回溯算法最佳实践：集合划分**

解法代码

```
class Solution {
    public boolean canPartitionKSubsets(int[] nums, int k) {
        // 排除一些基本情况
        if (k > nums.length) return false;
        int sum = 0;
        for (int v : nums) sum += v;
        if (sum % k != 0) return false;

        int used = 0; // 使用位图技巧
```

```
        int target = sum / k;
        // k 号桶初始什么都没装，从 nums[0] 开始做选择
        return backtrack(k, 0, nums, 0, used, target);
    }

    HashMap<Integer, Boolean> memo = new HashMap<>();

    boolean backtrack(int k, int bucket,
                      int[] nums, int start, int used, int target) {
        // base case
        if (k == 0) {
            // 所有桶都被装满了，而且 nums 一定全部用完了
            return true;
        }
        if (bucket == target) {
            // 装满了当前桶，递归穷举下一个桶的选择
            // 让下一个桶从 nums[0] 开始选数字
            boolean res = backtrack(k - 1, 0, nums, 0, used, target);
            // 缓存结果
            memo.put(used, res);
            return res;
        }

        if (memo.containsKey(used)) {
            // 避免冗余计算
            return memo.get(used);
        }

        for (int i = start; i < nums.length; i++) {
            // 剪枝
            if (((used >> i) & 1) == 1) { // 判断第 i 位是否是 1
                // nums[i] 已经被装入别的桶中
                continue;
            }
            if (nums[i] + bucket > target) {
                continue;
            }
            // 做选择
            used |= 1 << i; // 将第 i 位置为 1
            bucket += nums[i];
            // 递归穷举下一个数字是否装入当前桶
            if (backtrack(k, bucket, nums, i + 1, used, target)) {
                return true;
            }
            // 撤销选择
            used ^= 1 << i; // 将第 i 位置为 0
            bucket -= nums[i];
        }

        return false;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目：**
  - 46. 全排列 🟠
  - 78. 子集 🟠
  - 剑指 Offer II 079. 所有子集 🟠
  - 剑指 Offer II 083. 没有重复元素集合的全排列 🟠

784. 字母大小写全排列

LeetCode	力扣	难度
784. Letter Case Permutation	784. 字母大小写全排列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：回溯算法

给定一个字符串 s ，通过将字符串 s 中的每个字母转变大小写，我们可以获得一个新的字符串。返回所有可能得到的字符串集合。以 **任意顺序** 返回输出。

示例 1：

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

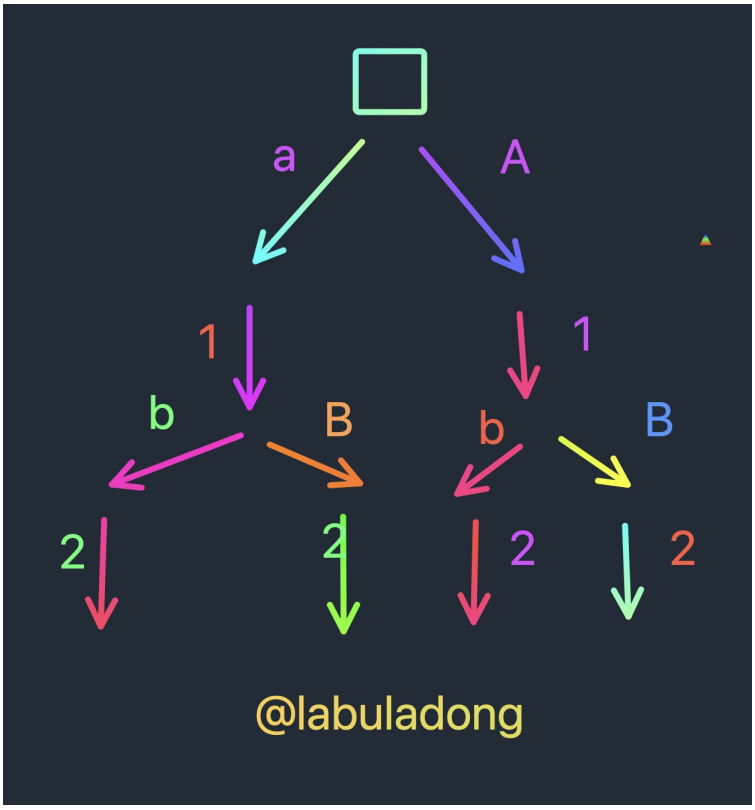
基本思路

这是标准的回溯算法，可以通过前文讲的 回溯算法框架 求解。

如何穷举呢？对于每一个字符，有如下选择：

1、如果该字符是字母，则可以是大写，也可以是小写。

2、如果该字符是数字，则只能是它本身。



有了这个思路，直接翻译成代码即可。

解法代码

```
class Solution {
    public List<String> letterCasePermutation(String s) {
        backtrack(s, 0);
        return res;
    }

    StringBuilder track = new StringBuilder();
    List<String> res = new LinkedList<>();

    void backtrack(String s, int index) {
        if (index == s.length()) {
            res.add(track.toString());
            return;
        }

        if ('0' <= s.charAt(index) && s.charAt(index) <= '9') {
            // s[index] 是数字
            // 做选择
            track.append(s.charAt(index));
            backtrack(s, index + 1);
            // 撤销选择
            track.deleteCharAt(track.length() - 1);
        } else {
            // s[index] 是字母

            // 小写字母，做选择
            track.append(Character.toLowerCase(s.charAt(index)));
            backtrack(s, index + 1);
            // 撤销选择
            track.deleteCharAt(track.length() - 1);

            // 大写字母，做选择
            track.append(Character.toUpperCase(s.charAt(index)));
            backtrack(s, index + 1);
            // 撤销选择
            track.deleteCharAt(track.length() - 1);
        }
    }
}
```

剑指 Offer 38. 字符串的排列

LeetCode	力扣	难度
剑指Offer38. 字符串的排列 LCOF	剑指Offer38. 字符串的排列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，排列组合**

输入一个字符串，打印出该字符串中字符的所有排列。你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

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

基本思路

这就是 一文秒杀所有排列组合子集问题 中讲的「元素可重不可复选」的排列问题，前文我使用 47. 全排列 II（中等） 举的例子，你把 47 题的解法代码稍微改一改就可以解决这道题了。

解法代码

```
class Solution {
    public String[] permutation(String s) {
        permuteUnique(s.toCharArray());
        String[] arr = new String[res.size()];
        for (int i = 0; i < res.size(); i++) {
            arr[i] = res.get(i);
        }
        return arr;
    }

    List<String> res = new ArrayList<>();
    StringBuilder track = new StringBuilder();
    boolean[] used;

    public List<String> permuteUnique(char[] nums) {
        // 先排序，让相同的元素靠在一起
        Arrays.sort(nums);
        used = new boolean[nums.length];
        backtrack(nums);
        return res;
    }
```

```
    void backtrack(char[] nums) {
        if (track.length() == nums.length) {
            res.add(track.toString());
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (used[i]) {
                continue;
            }
            // 新添加的剪枝逻辑，固定相同的元素在排列中的相对位置
            if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
                continue;
            }
            track.append(nums[i]);
            used[i] = true;
            backtrack(nums);
            track.deleteCharAt(track.length() - 1);
            used[i] = false;
        }
    }
}
```

79. 单词搜索

LeetCode	力扣	难度
79. Word Search	79. 单词搜索	🟠

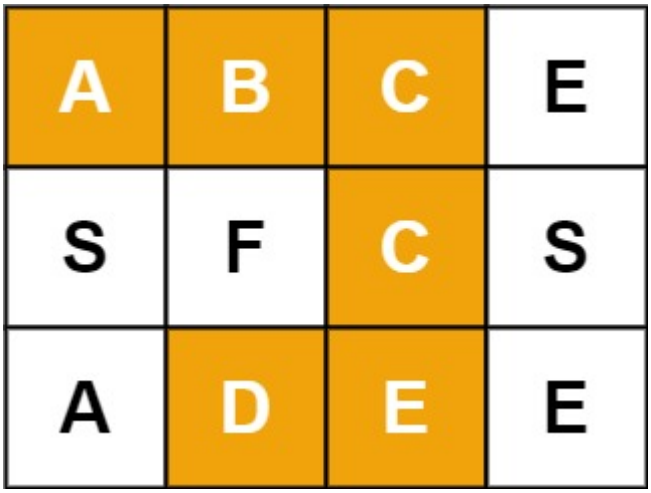
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：DFS 算法

给定一个 `m x n` 二维字符网格 `board` 和一个字符串单词 `word`。如果 `word` 存在于网格中，返回 `true`；否则，返回 `false`。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1：




```
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word
= "ABCCED"
输出：true
```

基本思路

这道题和 一文秒杀所有岛屿题目 中讲到的题目非常类似，用 DFS 算法暴力搜索就行了。

注意我们要对已经匹配过的字符做标记，比如用 图论算法基础 中讲到的 `visited` 数组，或者使用其他方法标记 `board` 中已经匹配过的字符。

如果不做标记的话会出现错误的结果，比如这个测试用例：

```
[["A","B","C","E"],
 ["S","F","C","S"],
 ["A","D","E","E"]]
```

你如果在其中搜索 "ABCB"，按道理不应该搜到，但如果你不对已经匹配过的字符做标记的话，算法可能匹配了第一行前三列的 "ABC" 之后又回头匹配了第二列的 "B"，导致出现错误的结果。

我就用加负号的方式标记已经匹配过的字符，来避免走回头路，具体看代码吧。

解法代码

```
class Solution {

    boolean found = false;

    public boolean exist(char[][] board, String word) {
        int m = board.length, n = board[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dfs(board, i, j, word, 0);
                if (found) {
                    return true;
                }
            }
        }
        return false;
    }

    // 从 (i, j) 开始向四周搜索，试图匹配 word[p..]
    void dfs(char[][] board, int i, int j, String word, int p) {
        if (p == word.length()) {
            // 整个 word 已经被匹配完，找到了一个答案
            found = true;
            return;
        }
        if (found) {
            // 已经找到了一个答案，不用再搜索了
            return;
        }
        int m = board.length, n = board[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n) {
            return;
        }
        if (board[i][j] != word.charAt(p)) {
            return;
        }

        // 已经匹配过的字符，我们给它添一个负号作为标记，避免走回头路
        board[i][j] = (char)(-board[i][j]);
        // word[p] 被 board[i][j] 匹配，开始向四周搜索 word[p+1..]
        dfs(board, i + 1, j, word, p + 1);
        dfs(board, i, j + 1, word, p + 1);
        dfs(board, i - 1, j, word, p + 1);
        dfs(board, i, j - 1, word, p + 1);
```

```
            board[i][j] = (char)(-board[i][j]);
        }
    }
```

▶ 代码可视化动画（需安装 Chrome 插件）

1020. 飞地的数量

LeetCode	力扣	难度
1020. Number of Enclaves	1020. 飞地的数量	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：DFS 算法，二维矩阵

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）。一次移动是指在从一个陆地单元格走到另一个（上下左右）陆地单元格，或走出网格的边界。

计算网格中**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。
```

基本思路

这题属于岛屿系列问题，岛屿系列问题的基本思路框架是 200. 岛屿数量 这道题，没看过的先看这篇。

这道题和 1254. 统计封闭岛屿的数目 基本一样，只是后者让你算封闭岛屿的数量，这题让你算这些封闭岛屿的陆地总数，稍微改改代码就行了。

注意这题中 1 代表陆地，0 代表海水。

- 详细题解：一文秒杀所有岛屿题目

解法代码

```
class Solution {
    public int numEnclaves(int[][] grid) {
        int m = grid.length, n = grid[0].length;

        for (int i = 0; i < m; i++) {
            dfs(grid, i, 0);
            dfs(grid, i, n - 1);
        }

        for (int j = 0; j < n; j++) {
            dfs(grid, 0, j);
```

```
                dfs(grid, m - 1, j);
            }

            int res = 0;
            for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                    if (grid[i][j] == 1) {
                        res += 1;
                    }
                }
            }
            return res;
        }

        void dfs(int[][] grid, int i, int j) {
            int m = grid.length, n = grid[0].length;
            if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == 0) {
                return;
            }

            grid[i][j] = 0;

            dfs(grid, i + 1, j);
            dfs(grid, i, j + 1);
            dfs(grid, i - 1, j);
            dfs(grid, i, j - 1);
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 1254. 统计封闭岛屿的数目 🟠
  - 1905. 统计子岛屿 🟠
  - 200. 岛屿数量 🟠
  - 694. 不同岛屿的数量 🟠
  - 695. 岛屿的最大面积 🟠
  - 剑指 Offer II 105. 岛屿的最大面积 🟠

1254. 统计封闭岛屿的数目

LeetCode	力扣	难度
1254. Number of Closed Islands	1254. 统计封闭岛屿的数目	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

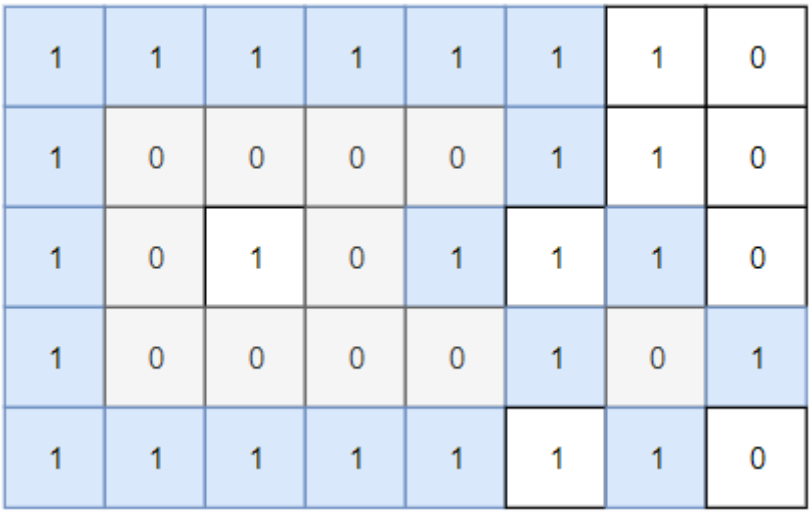
- 标签：DFS 算法，二维矩阵

有一个二维矩阵 `grid`，每个位置要么是陆地（记号为 `0`）要么是水域（记号为 `1`）。

我们从一块陆地出发，每次可以往上下左右 4 个方向相邻区域走，能走到的所有陆地区域，我们将其称为一座「**岛屿**」。如果一座岛屿**完全**由水域包围，即陆地边缘上下左右所有相邻区域都是水域，那么我们将其称为「**封闭岛屿**」。

请计算封闭岛屿的数目（靠边的岛屿不算封闭的）。

示例 1：




```
输入：grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],
[1,0,0,0,0,1,0,1],[1,1,1,1,1,1,1,0]]
输出：2
解释：
灰色区域的岛屿是封闭岛屿，因为这座岛屿完全被水域包围（即被 1 区域包围）。
```

基本思路

岛屿系列问题的基本思路框架是 200. 岛屿数量 这道题，没看过的先看这篇。

如何判断「封闭岛屿」呢？其实很简单，把 200. 岛屿数量 中那些靠边的岛屿排除掉，剩下的不就是「封闭岛屿」了吗？

有了这个思路，就可以直接写出代码了，注意这题规定 `0` 表示陆地，用 `1` 表示海水。

- **详细题解：一文秒杀所有岛屿题目**

解法代码

```
class Solution {
     // 主函数：计算封闭岛屿的数量
    public int closedIsland(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        for (int j = 0; j < n; j++) {
            // 把靠上边的岛屿淹掉
            dfs(grid, 0, j);
            // 把靠下边的岛屿淹掉
            dfs(grid, m - 1, j);
        }
        for (int i = 0; i < m; i++) {
            // 把靠左边的岛屿淹掉
            dfs(grid, i, 0);
            // 把靠右边的岛屿淹掉
            dfs(grid, i, n - 1);
        }
        // 遍历 grid，剩下的岛屿都是封闭岛屿
        int res = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 0) {
                    res++;
                    dfs(grid, i, j);
                }
            }
        }
        return res;
    }

    // 从 (i, j) 开始，将与之相邻的陆地都变成海水
    void dfs(int[][] grid, int i, int j) {
        int m = grid.length, n = grid[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n) {
            return;
        }
        if (grid[i][j] == 1) {
            // 已经是海水了
            return;
        }
        // 将 (i, j) 变成海水
        grid[i][j] = 1;
        // 淹没上下左右的陆地
        dfs(grid, i + 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i - 1, j);
        dfs(grid, i, j - 1);
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目：**
  - 1020. 飞地的数量 🟠
  - 130. 被围绕的区域 🟠
  - 1905. 统计子岛屿 🟠
  - 200. 岛屿数量 🟠
  - 694. 不同岛屿的数量 🟠
  - 695. 岛屿的最大面积 🟠
  - 剑指 Offer II 105. 岛屿的最大面积 🟠

1905. 统计子岛屿

LeetCode	力扣	难度
1905. Count Sub Islands	1905. 统计子岛屿	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

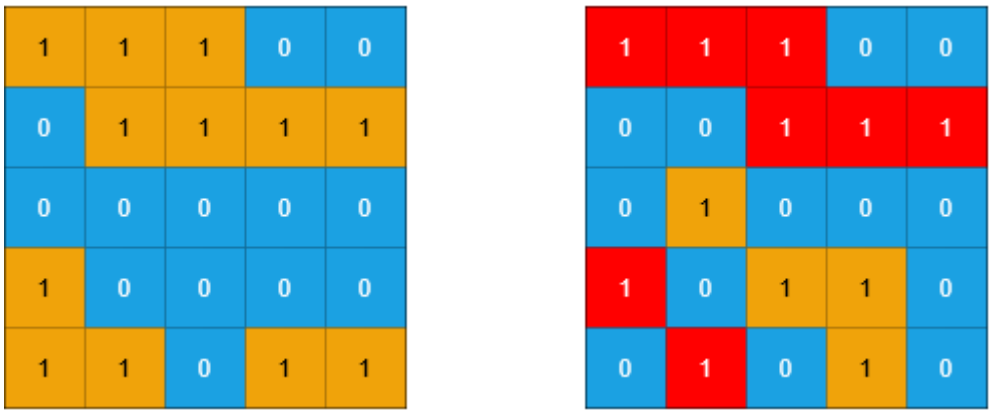
- 标签：DFS 算法，二维矩阵

给你两个 `m x n` 的二进制矩阵 `grid1` 和 `grid2`，它们只包含 `0`（表示水域）和 `1`（表示陆地）。一个**岛屿**是由**四个方向**（水平或者竖直）上相邻的 `1` 组成的区域。任何矩阵以外的区域都视为水域。

如果 `grid2` 的一个岛屿，被 `grid1` 的一个岛屿**完全**包含，也就是说 `grid2` 中该岛屿的每一个格子都被 `grid1` 中同一个岛屿完全包含，那么我们称 `grid2` 中的这个岛屿为**子岛屿**。

请你返回 `grid2` 中**子岛屿**的数目。

示例 1：




```
输入：grid1 = [[1,1,1,0,0],[0,1,1,1,1],[0,0,0,0,0],[1,0,0,0,0],[1,1,0,1,1]],
grid2 = [[1,1,1,0,0],[0,0,1,1,1],[0,1,0,0,0],[1,0,1,1,0],[0,1,0,1,0]]
输出：3
解释：如上图所示，左边为 grid1，右边为 grid2。
grid2 中标红的 1 区域是子岛屿，总共有 3 个子岛屿。
```

基本思路

这题属于岛屿系列问题，岛屿系列问题的基本思路框架是 200. 岛屿数量 这道题，没看过的先看这篇。

这道题的关键在于，如何快速判断子岛屿？

什么情况下 `grid2` 中的一个岛屿 `B` 是 `grid1` 中的一个岛屿 `A` 的子岛？

当岛屿 `B` 中所有陆地在岛屿 `A` 中也是陆地的时候，岛屿 `B` 是岛屿 `A` 的子岛。

反过来说，如果岛屿 `B` 中存在一片陆地，在岛屿 `A` 的对应位置是海水，那么岛屿 `B` 就不是岛屿 `A` 的子岛。

那么，我们只要遍历 grid2 中的所有岛屿，把那些不可能是子岛的岛屿排除掉，剩下的就是子岛。

- **详细题解：一文秒杀所有岛屿题目**

解法代码

```
class Solution {
    public int countSubIslands(int[][] grid1, int[][] grid2) {
        int m = grid1.length, n = grid1[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid1[i][j] == 0 && grid2[i][j] == 1) {
                    // 这个岛屿肯定不是子岛，淹掉
                    dfs(grid2, i, j);
                }
            }
        }
        // 现在 grid2 中剩下的岛屿都是子岛，计算岛屿数量
        int res = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid2[i][j] == 1) {
                    res++;
                    dfs(grid2, i, j);
                }
            }
        }
        return res;
    }

    // 从 (i, j) 开始，将与之相邻的陆地都变成海水
    void dfs(int[][] grid, int i, int j) {
        int m = grid.length, n = grid[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n) {
            return;
        }
        if (grid[i][j] == 0) {
            return;
        }

        grid[i][j] = 0;
        dfs(grid, i + 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i - 1, j);
        dfs(grid, i, j - 1);
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 1020. 飞地的数量 🟠
  - 1254. 统计封闭岛屿的数目 🟠
  - 200. 岛屿数量 🟠
  - 694. 不同岛屿的数量 🟠
  - 695. 岛屿的最大面积 🟠
  - 剑指 Offer II 105. 岛屿的最大面积 🟠

200. 岛屿数量

LeetCode	力扣	难度
200. Number of Islands	200. 岛屿数量	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：DFS 算法，二维矩阵**

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

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。此外，你可以假设该网格的四条边均被水包围。

示例 1：

```
输入：grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
输出：1
```

基本思路

岛屿系列问题可以用 DFS/BFS 算法或者 Union-Find 并查集算法 来解决。

用 DFS 算法解决岛屿题目是最常见的，每次遇到一个岛屿中的陆地，就用 DFS 算法吧这个岛屿「淹掉」。

如何使用 DFS 算法遍历二维数组？你把二维数组中的每个格子看做「图」中的一个节点，这个节点和周围的四个节点连通，这样二维矩阵就被抽象成了一幅网状的「图」。

为什么每次遇到岛屿，都要用 DFS 算法把岛屿「淹了」呢？主要是为了省事，避免维护 `visited` 数组。

图算法遍历基础 说了，遍历图是需要 `visited` 数组记录遍历过的节点防止走回头路。

因为 `dfs` 函数遍历到值为 `0` 的位置会直接返回，所以只要把经过的位置都设置为 `0`，就可以起到不走回头路的作用。

- **详细题解：一文秒杀所有岛屿题目**

解法代码

```
class Solution {
    // 主函数，计算岛屿数量
```

```
    public int numIslands(char[][] grid) {
        int res = 0;
        int m = grid.length, n = grid[0].length;
        // 遍历 grid
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    // 每发现一个岛屿，岛屿数量加一
                    res++;
                    // 然后使用 DFS 将岛屿淹了
                    dfs(grid, i, j);
                }
            }
        }
        return res;
    }

    // 从 (i, j) 开始，将与之相邻的陆地都变成海水
    void dfs(char[][] grid, int i, int j) {
        int m = grid.length, n = grid[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n) {
            // 超出索引边界
            return;
        }
        if (grid[i][j] == '0') {
            // 已经是海水了
            return;
        }
        // 将 (i, j) 变成海水
        grid[i][j] = '0';
        // 淹没上下左右的陆地
        dfs(grid, i + 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i - 1, j);
        dfs(grid, i, j - 1);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 1020. 飞地的数量 🟠
  - 1254. 统计封闭岛屿的数目 🟠
  - 1905. 统计子岛屿 🟠
  - 694. 不同岛屿的数量 🟠
  - 695. 岛屿的最大面积 🟠
  - 剑指 Offer II 105. 岛屿的最大面积 🟠

694. 不同的岛屿数量

LeetCode	力扣	难度
694. Number of Distinct Islands	694. 不同岛屿的数量	🟠

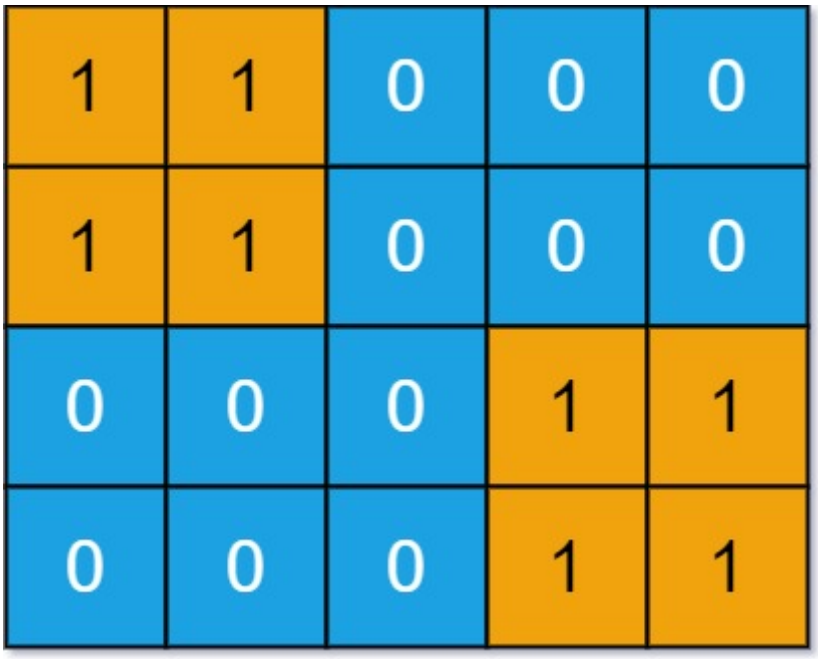
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：DFS 算法，二维矩阵**

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

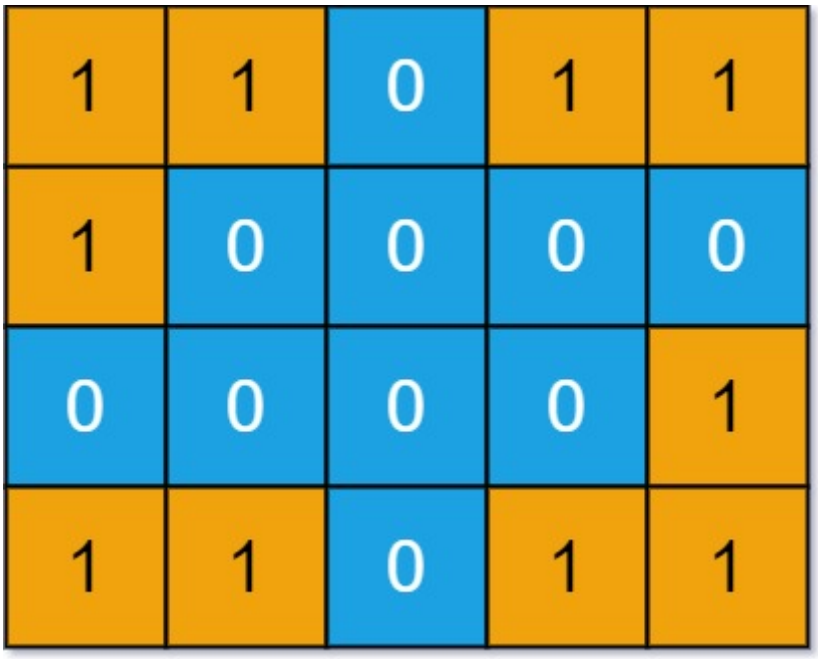
岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。此外，你可以假设该网格的四条边均被水包围。

示例 1：



给定上图，返回结果 1。

示例 2：

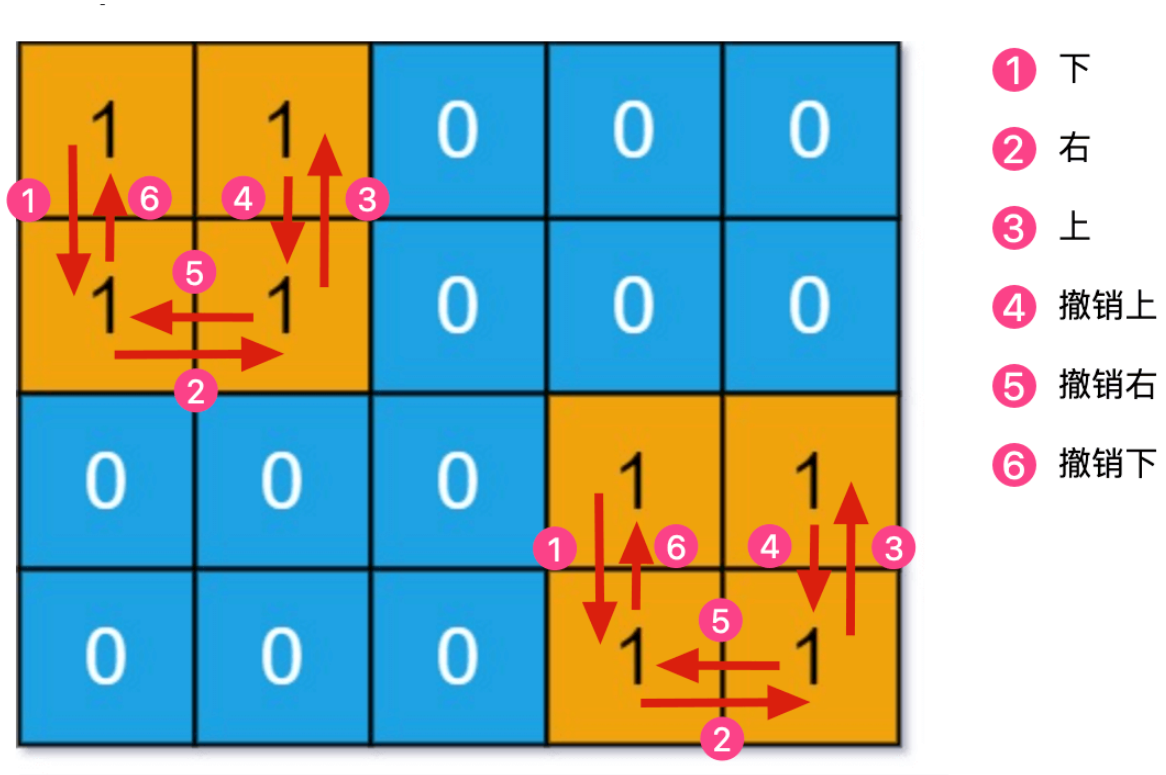


给定上图，返回结果 3。

基本思路

如果想把岛屿转化成字符串，说白了就是序列化，序列化说白了就是遍历嘛，前文 二叉树的序列化和反序列化 讲了二叉树和字符串互转，这里也是类似的，对于形状相同的岛屿，如果从同一起点出发，`dfs` 函数遍历的顺序肯定是一样的。

所以，遍历顺序从某种意义上说就可以用来描述岛屿的形状，比如下图这两个岛屿：



假设它们的遍历顺序是：

> 下，右，上，撤销上，撤销右，撤销下

如果我用分别用 `1, 2, 3, 4` 代表上下左右，用 `-1, -2, -3, -4` 代表上下左右的撤销，那么可以这样表示它们的遍历顺序：

```
2, 4, 1, -1, -4, -2
```

这就相当于是岛屿序列化的结果，只要每次使用 `dfs` 遍历岛屿的时候生成这串数字进行比较，就可以计算到底有多少个不同的岛屿了。

- **详细题解：一文秒杀所有岛屿题目**

解法代码

```
class Solution {
    public int numDistinctIslands(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        // 记录所有岛屿的序列化结果
        HashSet<String> islands = new HashSet<>();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    // 淹掉这个岛屿，同时存储岛屿的序列化结果
                    StringBuilder sb = new StringBuilder();
                    // 初始的方向可以随便写，不影响正确性
                    dfs(grid, i, j, sb, 666);
                    islands.add(sb.toString());
                }
            }
        }
        // 不相同的岛屿数量
        return islands.size();
    }

    private void dfs(int[][] grid, int i, int j, StringBuilder sb, int
dir) {
        int m = grid.length, n = grid[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n
            || grid[i][j] == 0) {
            return;
        }
        // 前序遍历位置：进入 (i, j)
        grid[i][j] = 0;
        sb.append(dir).append(',');

        dfs(grid, i - 1, j, sb, 1); // 上
        dfs(grid, i + 1, j, sb, 2); // 下
        dfs(grid, i, j - 1, sb, 3); // 左
        dfs(grid, i, j + 1, sb, 4); // 右

        // 后序遍历位置：离开 (i, j)
        sb.append(-dir).append(',');
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目：**
  - 1020. 飞地的数量 🟠
  - 1254. 统计封闭岛屿的数目 🟠
  - 1905. 统计子岛屿 🟠
  - 200. 岛屿数量 🟠
  - 695. 岛屿的最大面积 🟠
  - 剑指 Offer II 105. 岛屿的最大面积 🟠

695. 岛屿的最大面积

LeetCode	力扣	难度
695. Max Area of Island	695. 岛屿的最大面积	🟠

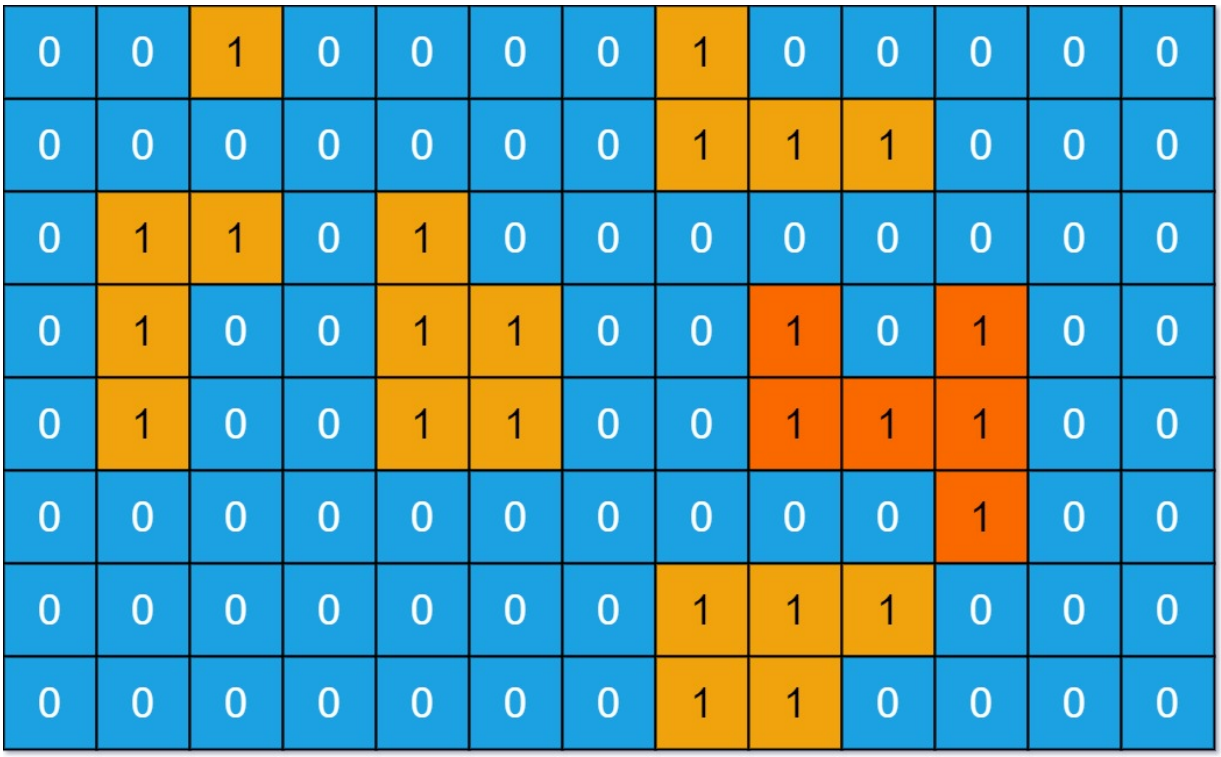
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：DFS 算法，二维矩阵

给你一个大小为 `m x n` 的二维矩阵 `grid`，其中**岛屿**是由一些相邻的 `1`（代表土地）构成的组合，这里的「相邻」要求两个 `1` 必须在**水平或者竖直的四个方向上**相邻。你可以假设 `grid` 的四个边缘都被 `0`（代表水）包围着。

岛屿的面积是岛上值为 `1` 的单元格的数目，请你计算并返回 `grid` 中最大的岛屿面积。如果没有岛屿，则返回面积为 `0`。

示例 1：




```
输入：grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],
[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],
[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],
[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]
输出：6
解释：答案不应该是 11，因为岛屿只能包含水平或垂直这四个方向上的 1。
```

基本思路

这题属于岛屿系列问题，岛屿系列问题的基本思路框架是 200. 岛屿数量 这道题，没看过的先看这篇。

这题的大体思路和 200. 岛屿数量 完全一样，只不过 `dfs` 函数淹没岛屿的同时，还应该想办法记录这个岛屿的面积。

我们可以给 `dfs` 函数设置返回值，记录每次淹没的陆地的个数，直接看解法吧。

- **详细题解：一文秒杀所有岛屿题目**

解法代码

```
class Solution {
    public int maxAreaOfIsland(int[][] grid) {
        // 记录岛屿的最大面积
        int res = 0;
        int m = grid.length, n = grid[0].length;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    // 淹没岛屿，并更新最大岛屿面积
                    res = Math.max(res, dfs(grid, i, j));
                }
            }
        }
        return res;
    }

    // 淹没与 (i, j) 相邻的陆地，并返回淹没的陆地面积
    int dfs(int[][] grid, int i, int j) {
        int m = grid.length, n = grid[0].length;
        if (i < 0 || j < 0 || i >= m || j >= n) {
            // 超出索引边界
            return 0;
        }
        if (grid[i][j] == 0) {
            // 已经是海水了
            return 0;
        }
        // 将 (i, j) 变成海水
        grid[i][j] = 0;

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

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1020. 飞地的数量 🟠
  - 1254. 统计封闭岛屿的数目 🟠
  - 1905. 统计子岛屿 🟠
  - 200. 岛屿数量 🟠
  - 694. 不同岛屿的数量 🟠
  - 剑指 Offer II 105. 岛屿的最大面积 🟠

剑指 Offer II 105. 岛屿的最大面积

这道题和 695. 岛屿的最大面积 相同。

576. 出界的路径数

LeetCode	力扣	难度
576. Out of Boundary Paths	576. 出界的路径数	🟠

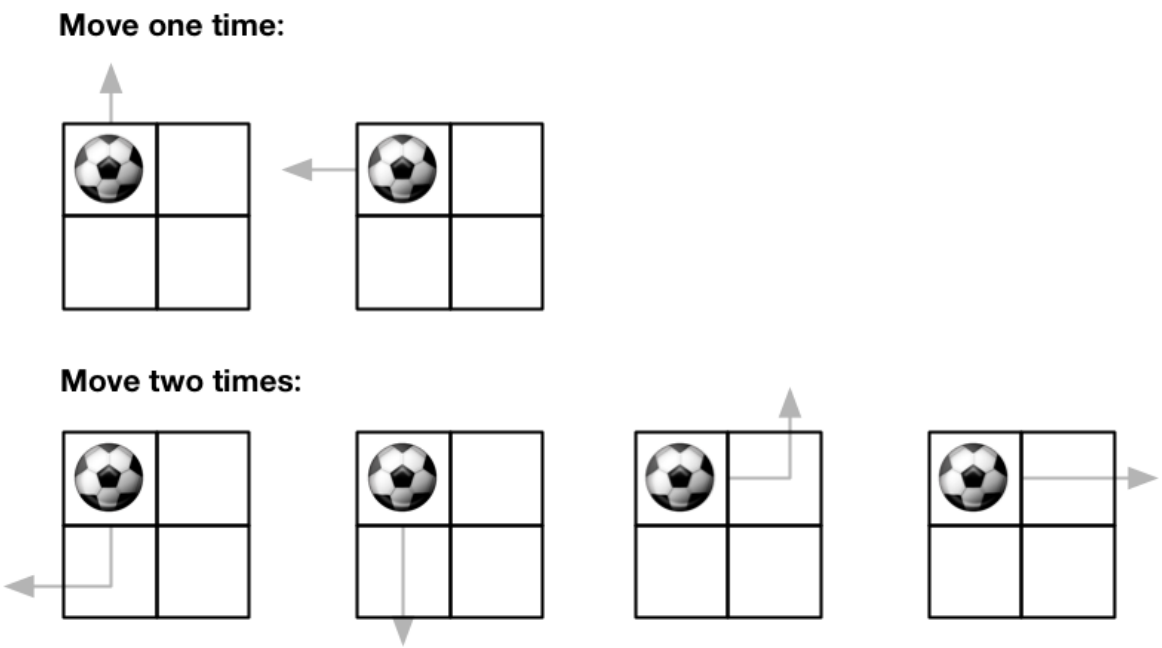
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：DFS 算法，动态规划

给你一个大小为 `m x n` 的网格和一个球。球的起始坐标为 `[startRow, startColumn]`。你可以将球移到在四个方向上相邻的单元格内（可以穿过网格边界到达网格之外）。你**最多** 可以移动 `maxMove` 次球。

给你五个整数 `m`、`n`、`maxMove`、`startRow` 以及 `startColumn`，找出并返回可以将球移出边界的路径数量。因为答案可能非常大，返回对 `10^9 + 7` **取余** 后的结果。

示例 1：




```
输入：m = 2, n = 2, maxMove = 2, startRow = 0, startColumn = 0
输出：6
```

基本思路

关于动态规划的解题步骤和思维方法见前文 动态规划核心套路 和 动态规划答疑篇，这里就不赘述了，直接给出最关键的状态转移方程。

这道题有三个状态：横坐标、纵坐标、可以移动的次数，所以我们可以用一个三维 `dp` 数组或者一个带有三个参数的 `dp` 函数记录这些状态，比如我定义一个 `dp` 函数：

```
// 定义：从 (i, j) 出发，最大移动次数为 maxMove，出界的路径数量为 dp(maxMove, i,
j)
int dp(int maxMove, int i, int j);
```

那么状态转移关系就是：

在 maxMove 步之内从 (i, j) 踢出界的路径数量等于在 maxMove - 1 步之内从 (i, j) 的相邻位置踢出界的路径数量之和。

这样就可以写出递归的暴力解了，然后按部就班加个备忘录消除冗余计算，即可解决这道题。

解法代码

```
class Solution {
    int[][][] memo;
    int m, n;

    public int findPaths(int m, int n, int maxMove, int startRow, int
startColumn) {
        this.m = m;
        this.n = n;
        memo = new int[m][n][maxMove + 1];
        // 备忘录初始化为特殊值 -1
        for (int[][] plane : memo) {
            for (int[] row : plane) {
                Arrays.fill(row, -1);
            }
        }
        // 按照定义，计算从 (startRow, startColumn) 开始，最大移动次数为 maxMove
的出界路径数量
        return dp(maxMove, startRow, startColumn);
    }

    // 定义：从 (i, j) 出发，最大移动次数为 maxMove，出界的路径数量为 dp(maxMove,
i, j)
    int dp(int maxMove, int i, int j) {
        // base case，找到一条出界路径
        if (i < 0 || j < 0 || i >= m || j >= n) {
            return 1;
        }
        // 避免冗余计算
        if (memo[i][j][maxMove] != -1) {
            return memo[i][j][maxMove];
        }
        // 无法在有限的步数内出界
        if (maxMove == 0) {
            return 0;
        }
        // 状态转移关系：
        // 在 maxMove 步之内从 (i, j) 踢出界的路径数量等于
```

```
            // 在 maxMove - 1 步之内从 (i, j) 的相邻位置踢出界的路径数量之和
            long res = 0;
            res += dp(maxMove - 1, i, j + 1);
            res += dp(maxMove - 1, i + 1, j);
            res += dp(maxMove - 1, i - 1, j);
            res += dp(maxMove - 1, i, j - 1);

            memo[i][j][maxMove] = (int) (res % 1000000007);
            return memo[i][j][maxMove];
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

130. 被围绕的区域

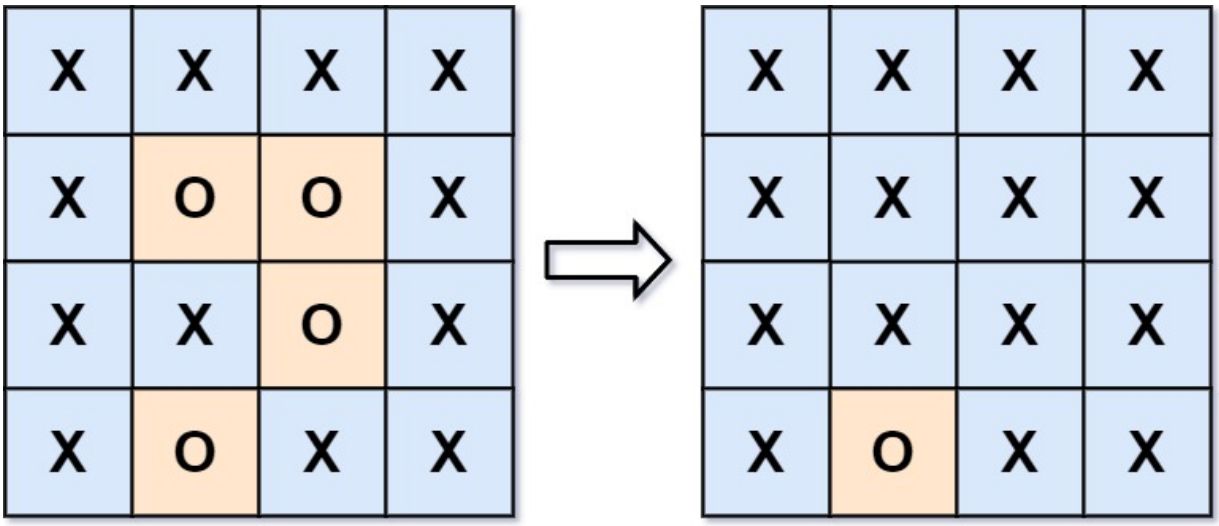
LeetCode	力扣	难度
130. Surrounded Regions	130. 被围绕的区域	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：DFS 算法，并查集算法

给你一个 `m x n` 的矩阵 `board`，由若干字符 `'X'` 和 `'O'` 组成，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 1：



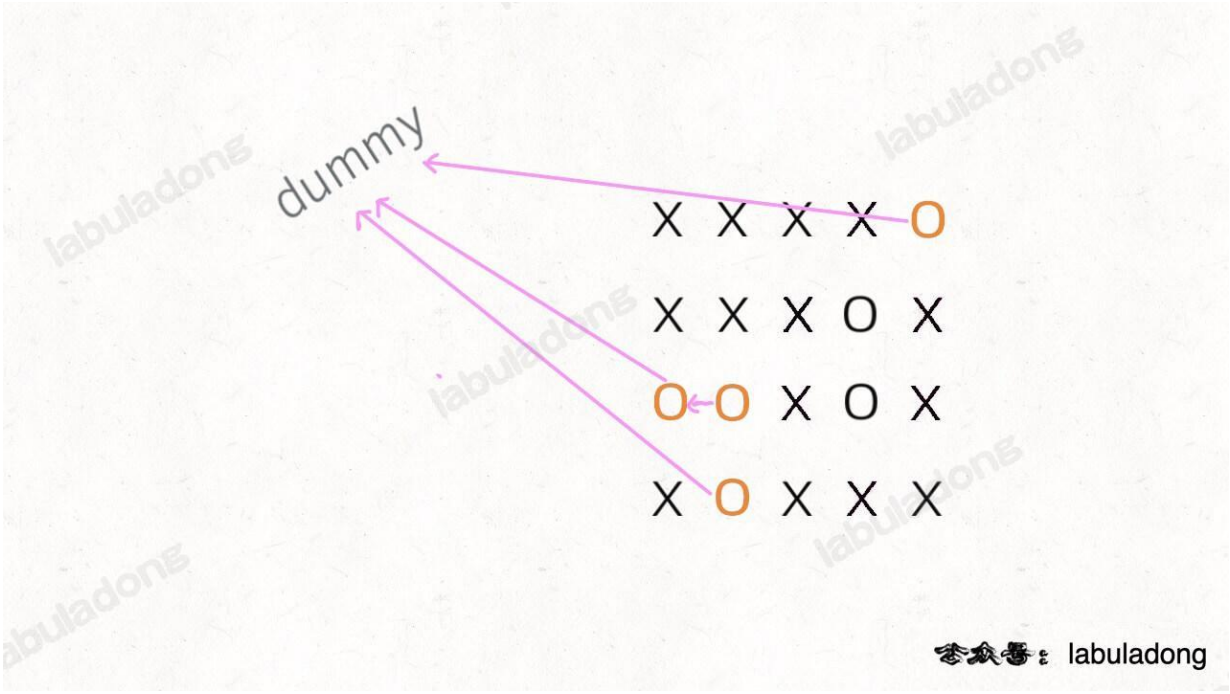

```
输入：board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],
["X","O","X","X"]]
输出：[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],
["X","O","X","X"]]
解释：被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。
```

基本思路

PS：这道题在《算法小抄》 的第 396 页。

这题和 1254. 统计封闭岛屿的数目 几乎完全一样，常规做法就是 DFS，那我们这里就讲一个另类的解法，看看并查集算法如何解决这道题。

我们可以把所有靠边的 `O` 和一个虚拟节点 `dummy` 进行连通：



然后再遍历整个 board，那些和 dummy 不连通的 O 就是被围绕的区域，需要被替换。

- **详细题解：并查集（Union-Find）算法**

解法代码

```
class Solution {
    public void solve(char[][] board) {
        if (board.length == 0) return;

        int m = board.length;
        int n = board[0].length;
        // 给 dummy 留一个额外位置
        UF uf = new UF(m * n + 1);
        int dummy = m * n;
        // 将首列和末列的 O 与 dummy 连通
        for (int i = 0; i < m; i++) {
            if (board[i][0] == 'O')
                uf.union(i * n, dummy);
            if (board[i][n - 1] == 'O')
                uf.union(i * n + n - 1, dummy);
        }
        // 将首行和末行的 O 与 dummy 连通
        for (int j = 0; j < n; j++) {
            if (board[0][j] == 'O')
                uf.union(j, dummy);
            if (board[m - 1][j] == 'O')
                uf.union(n * (m - 1) + j, dummy);
        }
        // 方向数组 d 是上下左右搜索的常用手法
        int[][] d = new int[][]{{1, 0}, {0, 1}, {0, -1}, {-1, 0}};
        for (int i = 1; i < m - 1; i++)
            for (int j = 1; j < n - 1; j++)
```

```
                if (board[i][j] == 'O')
                    // 将此 O 与上下左右的 O 连通
                    for (int k = 0; k < 4; k++) {
                        int x = i + d[k][0];
                        int y = j + d[k][1];
                        if (board[x][y] == 'O')
                            uf.union(x * n + y, i * n + j);
                    }
            // 所有不和 dummy 连通的 O，都要被替换
            for (int i = 1; i < m - 1; i++)
                for (int j = 1; j < n - 1; j++)
                    if (!uf.connected(dummy, i * n + j))
                        board[i][j] = 'X';
        }
    }

    class UF {
        // 记录连通分量个数
        private int count;
        // 存储若干棵树
        private int[] parent;
        // 记录树的“重量”
        private int[] size;

        public UF(int n) {
            this.count = n;
            parent = new int[n];
            size = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                size[i] = 1;
            }
        }

        /* 将 p 和 q 连通 */
        public void union(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            if (rootP == rootQ)
                return;

            // 小树接到大树下面，较平衡
            if (size[rootP] > size[rootQ]) {
                parent[rootQ] = rootP;
                size[rootP] += size[rootQ];
            } else {
                parent[rootP] = rootQ;
                size[rootQ] += size[rootP];
            }
            count--;
        }

        /* 判断 p 和 q 是否互相连通 */
        public boolean connected(int p, int q) {
```

```
        int rootP = find(p);
        int rootQ = find(q);
        // 处于同一棵树上的节点，相互连通
        return rootP == rootQ;
    }

    /* 返回节点 x 的根节点 */
    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    public int count() {
        return count;
    }
}
```

- **类似题目**：
  - 323. 无向图中连通分量的数目 🟠
  - 990. 等式方程的可满足性 🟠

剑指 Offer 13. 机器人的运动范围

LeetCode	力扣	难度
剑指Offer13. 机器人的运动范围 LCOF	剑指Offer13. 机器人的运动范围	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：DFS 算法，二维矩阵**

地上有一个 m 行n 列的方格，从坐标 `[0,0]` 到坐标 `[m-1,n-1]`。一个机器人从坐标 `[0, 0]` 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也不能进入行坐标和列坐标的数位之和大于 k 的格子。例如，当 k 为18 时，机器人能够进入方格 [35, 37]，因为 3+5+3+7=18。但它不能进入方格 [35,38]，因为 3+5+3+8=19。请问该机器人能够到达多少个格子？

示例 1：

```
输入：m = 2, n = 3, k = 1
输出：3
```

基本思路

用一个标准的 DFS 遍历就可以了，类似的题目可以参见 DFS 算法秒杀岛屿系列题目。

解法代码

```
class Solution {
    public int movingCount(int m, int n, int k) {
        boolean[][] visited = new boolean[m][n];
        dfs(m, n, k, 0, 0, visited);
        return res;
    }

    // 记录合法坐标数
    int res = 0;

    public void dfs(int m, int n, int k, int i, int j, boolean[][]
visited) {
        if (i < 0 || j < 0 || i >= m || j >= n) {
            // 超出索引边界
            return;
        }

        if (i / 10 + i % 10 + j / 10 + j % 10 > k) {
            // 坐标和超出 k 的限制
            return;
        }
```

```
        if (visited[i][j]) {
            // 之前已经访问过当前坐标
            return;
        }

        // 走到一个合法坐标
        res++;
        visited[i][j] = true;

        // DFS 遍历上下左右
        dfs(m, n, k, i + 1, j, visited);
        dfs(m, n, k, i, j + 1, visited);
        dfs(m, n, k, i - 1, j, visited);
        dfs(m, n, k, i, j - 1, visited);
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

102. 二叉树的层序遍历

LeetCode	力扣	难度
102. Binary Tree Level Order Traversal	102. 二叉树的层序遍历	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

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，从左到右访问所有节点）。

示例：
二叉树：`[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

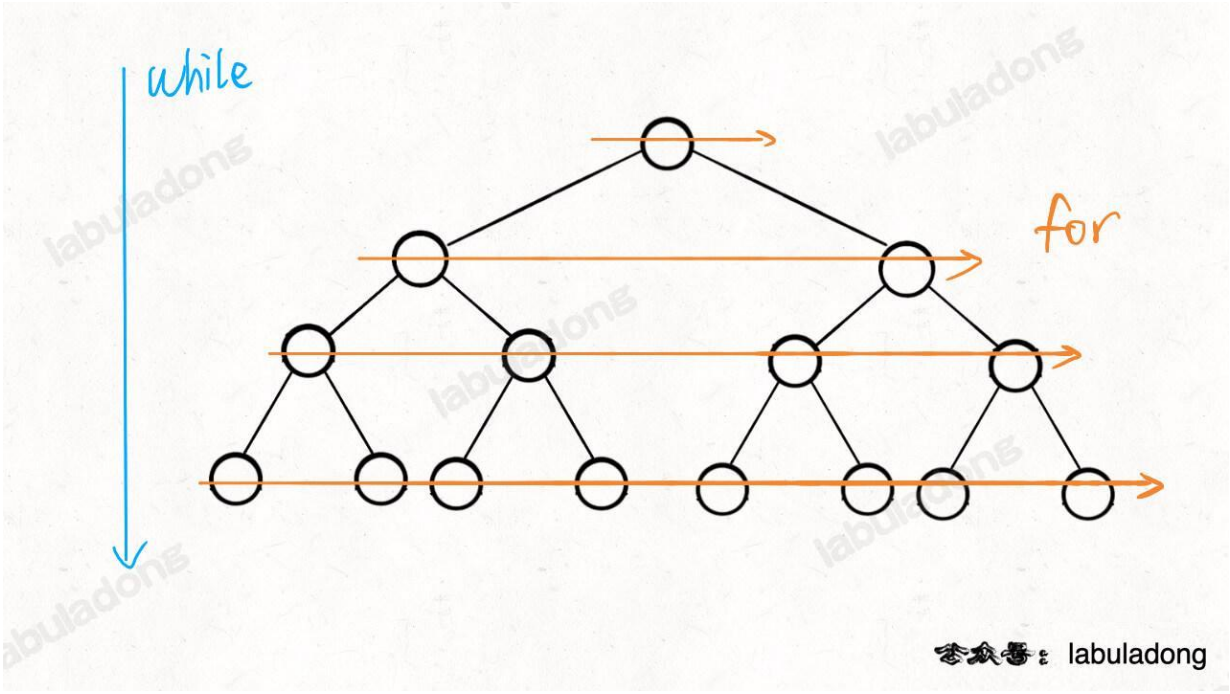
返回其层序遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

基本思路

前文 BFS 算法框架 就是由二叉树的层序遍历演变出来的。

下面是层序遍历的一般写法，通过一个 while 循环控制从上向下一层层遍历，for 循环控制每一层从左向右遍历：



解法代码

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录这一层的节点值
            List<Integer> level = new LinkedList<>();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                level.add(cur.val);
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            res.add(level);
        }
        return res;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 103. 二叉树的锯齿形层序遍历 🟠
  - 107. 二叉树的层序遍历 II 🟠
  - 1161. 最大层内元素和 🟠
  - 1302. 层数最深叶子节点的和 🟠
  - 1609. 奇偶树 🟠
  - 637. 二叉树的层平均值 🟢
  - 919. 完全二叉树插入器 🟠
  - 958. 二叉树的完全性检验 🟠
  - 剑指 Offer 32 - I. 从上到下打印二叉树 🟠
  - 剑指 Offer 32 - II. 从上到下打印二叉树 II 🟢
  - 剑指 Offer 32 - III. 从上到下打印二叉树 III 🟠

103. 二叉树的锯齿形层序遍历

LeetCode	力扣	难度
103. Binary Tree Zigzag Level Order Traversal	103. 二叉树的锯齿形层序遍历	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个二叉树，返回其节点值的锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：
给定二叉树 `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回锯齿形层序遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

基本思路

这题和 102. 二叉树的层序遍历 几乎是一样的，只要用一个布尔变量 `flag` 控制遍历方向即可。

解法代码

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
```

```
        // 为 true 时向右，false 时向左
        boolean flag = true;

        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录这一层的节点值
            LinkedList<Integer> level = new LinkedList<>();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                // 实现 z 字形遍历
                if (flag) {
                    level.addLast(cur.val);
                } else {
                    level.addFirst(cur.val);
                }
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            // 切换方向
            flag = !flag;
            res.add(level);
        }
        return res;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 1609. 奇偶树 🟠
  - 剑指 Offer 32 - III. 从上到下打印二叉树 III 🟠

107. 二叉树的层序遍历 II

LeetCode	力扣	难度
107. Binary Tree Level Order Traversal II	107. 二叉树的层序遍历 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个二叉树，返回其节点值自底向上的层序遍历（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）。

例如：
给定二叉树 `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其自底向上的层序遍历为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

基本思路

这题和 102. 二叉树的层序遍历 几乎是一样的，自顶向下的层序遍历反过来就行了。

解法代码

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        LinkedList<List<Integer>> res = new LinkedList<>();
        if (root == null) {
            return res;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
```

```
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录这一层的节点值
            List<Integer> level = new LinkedList<>();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                level.add(cur.val);
                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            // 把每一层添加到头部，就是自底向上的层序遍历。
            res.addFirst(level);
        }
        return res;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

1161. 最大层内元素和

LeetCode	力扣	难度
1161. Maximum Level Sum of a Binary Tree	1161. 最大层内元素和	🟠

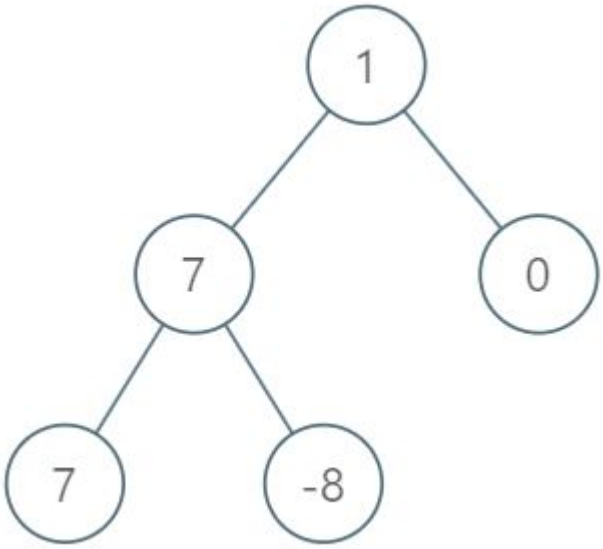
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树，二叉树vip

给你一个二叉树的根节点 `root`。设根节点位于二叉树的第 1 层，而根节点的子节点位于第 2 层，依此类推。

请你找出层内元素之和**最大**的那几层（可能只有一层）的层号，并返回其中**最小**的那个。

示例 1：




```
输入：root = [1,7,0,7,-8,null,null]
输出：2
解释：
第 1 层各元素之和为 1,
第 2 层各元素之和为 7 + 0 = 7,
第 3 层各元素之和为 7 + -8 = -1,
所以我们返回第 2 层的层号，它的层内元素之和最大。
```

基本思路

把 102. 二叉树的层序遍历 给出的层序遍历框架稍微变通即可解决这题。

解法代码

```java
class Solution {
    public int maxLevelSum(TreeNode root) {
        if (root == null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // 记录 BFS 走到的层数
        int depth = 1;
        // 记录元素和最大的那一行和最大元素和
        int res = 0, maxSum = Integer.MIN_VALUE;

        while (!q.isEmpty()) {
            int sz = q.size();
            int levelSum = 0;
            // 遍历这一层
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                levelSum += cur.val;

                if (cur.left != null)
                    q.offer(cur.left);
                if (cur.right != null)
                    q.offer(cur.right);
            }
            if (levelSum > maxSum) {
                // 更新最大元素和
                res = depth;
                maxSum = levelSum;
            }
            depth++;
        }
        return res;
    }
}
```

▶ 😽 代码可视化动画（需安装 Chrome 插件）😽

1302. 层数最深叶子节点的和

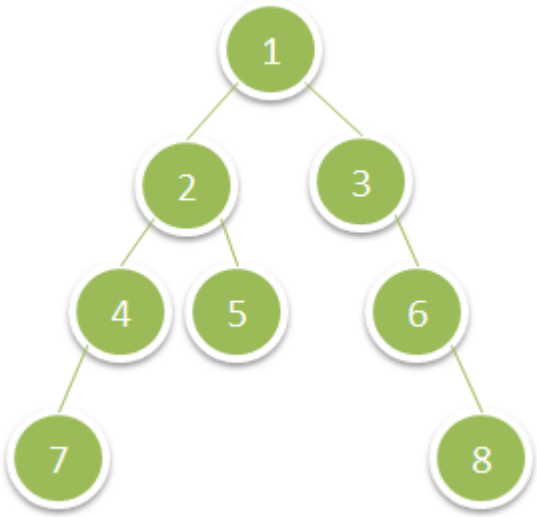
LeetCode	力扣	难度
1302. Deepest Leaves Sum	1302. 层数最深叶子节点的和	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树，二叉树vip

给你一棵二叉树的根节点 `root`，请你返回 **层数最深的叶子节点的和**。

示例 1：




```
输入：root = [1,2,3,4,5,null,6,7,null,null,null,null,8]
输出：15
```

基本思路

这题用 DFS 或者 BFS 都可以，我就用 BFS 层序遍历算法吧，层序遍历算法参见 102. 层序遍历二叉树，这题只要把最后一层的节点值累加起来就行了。

解法代码

```
class Solution {
    public int deepestLeavesSum(TreeNode root) {
        if (root == null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);

        int sum = 0;
        while (!q.isEmpty()) {
```

```
                sum = 0;
                int sz = q.size();
                for (int i = 0; i < sz; i++) {
                    TreeNode cur = q.poll();
                    // 累加一层的节点之和
                    sum += cur.val;
                    if (cur.left != null)
                        q.offer(cur.left);
                    if (cur.right != null)
                        q.offer(cur.right);
                }
            }
            // 现在就是最后一层的节点值和
            return sum;
        }
    }
```

▶ 代码可视化动画（需安装 Chrome 插件）

1609. 奇偶树

LeetCode	力扣	难度
1609. Even Odd Tree	1609. 奇偶树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

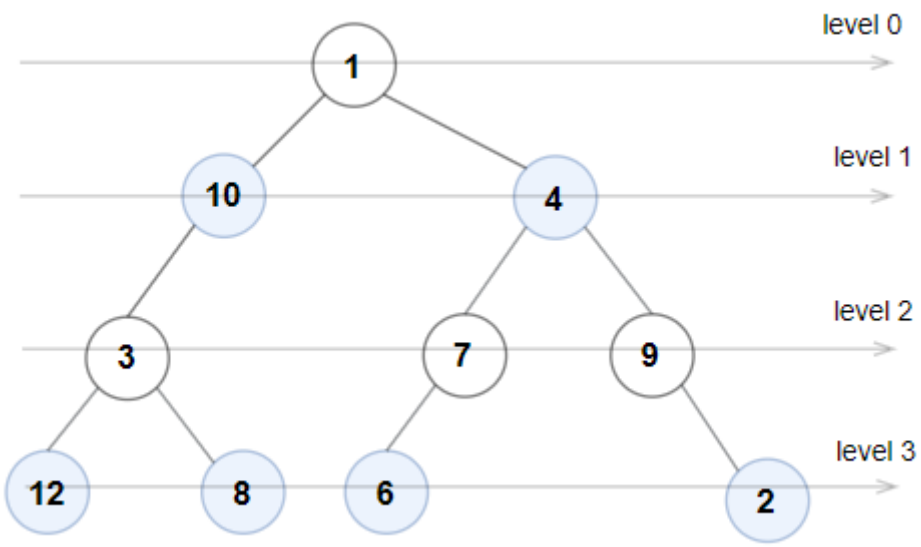
- 标签：二叉树，二叉树vip

如果一棵二叉树满足下述几个条件，则可以称为 **奇偶树**：

- 二叉树根节点所在层下标为 `0`，根的子节点所在层下标为 `1`，根的孙节点所在层下标为 `2`，依此类推。
- **偶数下标**层上的所有节点的值都是**奇**整数，从左到右按顺序**严格递增**
- **奇数下标**层上的所有节点的值都是**偶**整数，从左到右按顺序**严格递减**

给你二叉树的根节点，如果二叉树为**奇偶树**，则返回 `true`，否则返回 `false`。

示例 1：




```
输入：root = [1,10,4,3,null,7,9,12,8,6,null,null,2]
输出：true
解释：每一层的节点值分别是：
0 层：[1]
1 层：[10,4]
2 层：[3,7,9]
3 层：[12,8,6,2]
由于 0 层和 2 层上的节点值都是奇数且严格递增，而 1 层和 3 层上的节点值都是偶数且严格递减，因此这是一棵奇偶树。
```

基本思路

这道题主要考察二叉树的层序遍历，你可以先做一下 102. 二叉树的层序遍历 和 103. 二叉树的锯齿形层序遍历 这两道题，然后再做这道题。具体思路可看解法代码的注释。

解法代码

```
class Solution {
    public boolean isEvenOddTree(TreeNode root) {
        if (root == null) {
            return true;
        }

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // 记录奇偶层数
        boolean even = true;
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // 记录前一个节点，便于判断是否递增/递减
            int prev = even ? Integer.MIN_VALUE : Integer.MAX_VALUE;
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                if (even) {
                    // 偶数层
                    if (prev >= cur.val || cur.val % 2 == 0) {
                        return false;
                    }
                } else {
                    // 奇数层
                    if (prev <= cur.val || cur.val % 2 == 1) {
                        return false;
                    }
                }
                prev = cur.val;

                if (cur.left != null) {
                    q.offer(cur.left);
                }
                if (cur.right != null) {
                    q.offer(cur.right);
                }
            }
            // 奇偶层数切换
            even = !even;
        }
        return true;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

---

637. 二叉树的层平均值

LeetCode	力扣	难度
637. Average of Levels in Binary Tree	637. 二叉树的层平均值	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二叉树，二叉树vip

给定一个非空二叉树，返回一个由每层节点平均值组成的数组。

示例 1：

```
输入：
    3
   / \
  9  20
    /  \
   15   7
输出：[3, 14.5, 11]
解释：
第 0 层的平均值是 3 ,  第 1 层是 14.5 , 第 2 层是 11。因此返回 [3, 14.5, 11]。
```

基本思路

标准的二叉树层序遍历，把 102. 层序遍历二叉树 的代码稍微改一改就行了。

解法代码

```
class Solution {
    public List<Double> averageOfLevels(TreeNode root) {
        List<Double> res = new LinkedList<>();
        if (root == null) return res;

        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while (!q.isEmpty()) {
            int size = q.size();
            // 记录当前层所有节点之和
            double sum = 0;
            for (int i = 0; i < size; i++) {
                TreeNode cur = q.poll();
                if (cur.left != null) {
                    q.offer(cur.left);
                }
                if (cur.right != null) {
```

```
                    q.offer(cur.right);
                }
                sum += cur.val;
            }
            // 记录当前行的平均值
            res.add(1.0 * sum / size);
        }

        return res;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

919. 完全二叉树插入器

LeetCode	力扣	难度
919. Complete Binary Tree Inserter	919. 完全二叉树插入器	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树，二叉树vip

完全二叉树是每一层（除最后一层外）都是完全填充（即节点数达到最大）的，并且所有的节点都尽可能地集中在左侧。

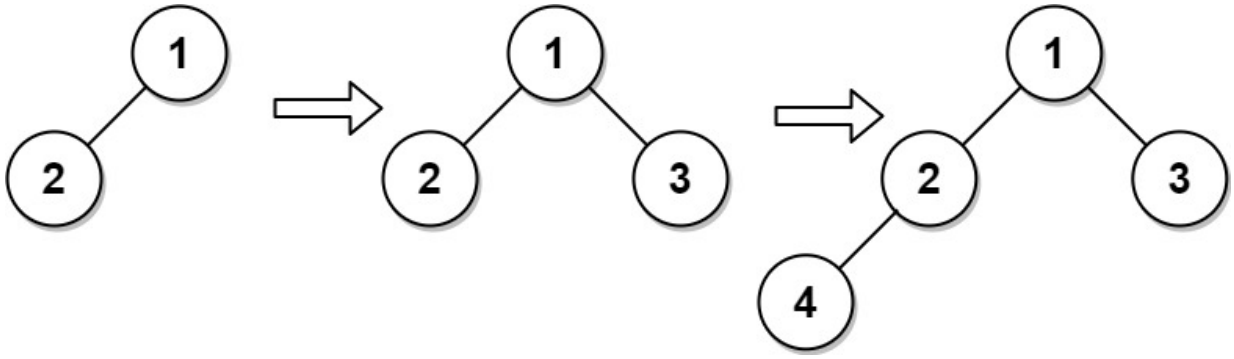
设计一个用完全二叉树初始化的数据结构 `CBTInserter`，它支持以下几种操作：

1、`CBTInserter(TreeNode root)` 使用头节点为 `root` 的给定树初始化该数据结构；

2、`CBTInserter.insert(int v)` 向树中插入一个新节点，节点类型为 `TreeNode`，值为 `v`。使树保持完全二叉树的状态，**并返回插入的新节点的父节点的值**；

3、`CBTInserter.get_root()` 将返回树的头节点。

示例 1：




```
输入：inputs = ["CBTInserter","insert","get_root"], inputs = [[[1]],[2],[]]
输出：[null,1,[1,2]]
```

基本思路

这道题考察二叉树的层序遍历，你需要先做 102. 二叉树的层序遍历 再做这道题，用队列维护底部可以进行插入的节点即可。

解法代码

```
class CBTInserter {
    // 这个队列只记录完全二叉树底部可以进行插入的节点
    private Queue<TreeNode> q = new LinkedList<>();
```

```
    private TreeNode root;

    public CBTInserter(TreeNode root) {
        this.root = root;
        // 进行普通的 BFS，目的是找到底部可插入的节点
        Queue<TreeNode> temp = new LinkedList<>();
        temp.offer(root);
        while (!temp.isEmpty()) {
            TreeNode cur = temp.poll();
            if (cur.left != null) {
                temp.offer(cur.left);
            }
            if (cur.right != null) {
                temp.offer(cur.right);
            }
            if (cur.right == null || cur.left == null) {
                // 找到完全二叉树底部可以进行插入的节点
                q.offer(cur);
            }
        }
    }

    public int insert(int val) {
        TreeNode node = new TreeNode(val);
        TreeNode cur = q.peek();
        // 进行插入
        if (cur.left == null) {
            cur.left = node;
        } else if (cur.right == null) {
            cur.right = node;
            q.poll();
        }
        // 新节点的左右节点也是可以插入的
        q.offer(node);
        return cur.val;
    }

    public TreeNode get_root() {
        return root;
    }
}
```

958. 二叉树的完全性检验

LeetCode	力扣	难度
958. Check Completeness of a Binary Tree	958. 二叉树的完全性检验	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

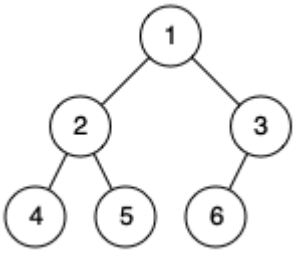
- **标签：二叉树，二叉树vip**

给定一个二叉树，确定它是否是一个**完全二叉树**。

百度百科中对完全二叉树的定义如下：

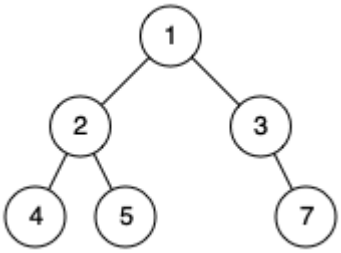
若设二叉树的深度为 h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。

示例 1：




```
输入：[1,2,3,4,5,6]
输出：true
解释：最后一层前的每一层都是满的（即，结点值为 {1} 和 {2,3} 的两层），且最后一层中的所有结点（{4,5,6}）都尽可能地向左。
```

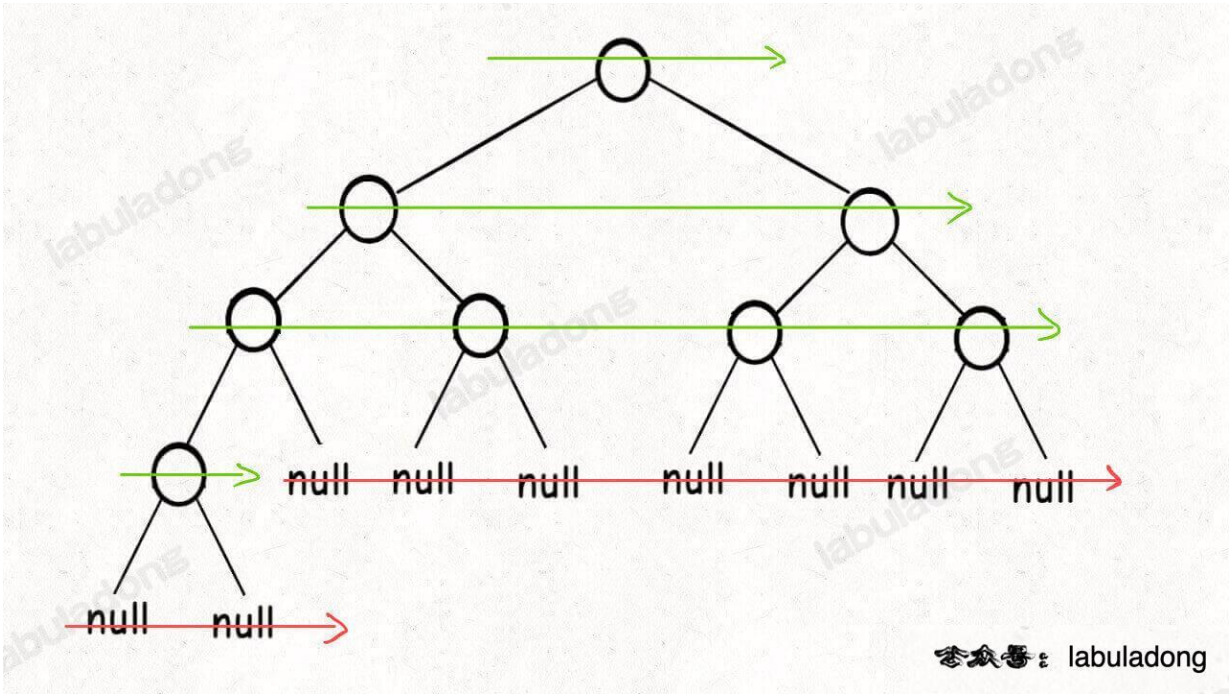
示例 2：




```
输入：[1,2,3,4,5,null,7]
输出：false
解释：值为 7 的结点没有尽可能靠向左侧。
```

基本思路

这题的关键是对完全二叉树特性的理解，**如果按照 BFS 层序遍历的方式遍历完全二叉树，队列最后留下的应该都是空指针**：



所以可以用 102. 二叉树的层序遍历 给出的层序遍历框架解决这题。

解法代码

```
class Solution {
    public boolean isCompleteTree(TreeNode root) {
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        // 遍历完所有非空节点时变成 true
        boolean end = false;
        // while 循环控制从上向下一层层遍历
        while (!q.isEmpty()) {
            int sz = q.size();
            // for 循环控制每一层从左向右遍历
            for (int i = 0; i < sz; i++) {
                TreeNode cur = q.poll();
                if (cur == null) {
                    // 第一次遇到 null 时 end 变成 true
                    // 如果之后的所有节点都是 null，则说明是完全二叉树
                    end = true;
                } else {
                    if (end) {
                        // end 为 true 时遇到非空节点说明不是完全二叉树
                        return false;
                    }
                    // 将下一层节点放入队列，不用判断是否非空
                    q.offer(cur.left);
                    q.offer(cur.right);
```

```
                }
            }
        }
        return true;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

剑指 Offer 32 - I. 从上到下打印二叉树

这道题和 102. 二叉树的层序遍历 相同。

剑指 Offer 32 - II. 从上到下打印二叉树 II

这道题和 102. 二叉树的层序遍历 相同。

剑指 Offer 32 - III. 从上到下打印二叉树 III

这道题和 103. 二叉树的锯齿形层序遍历 相同。

111. 二叉树的最小深度

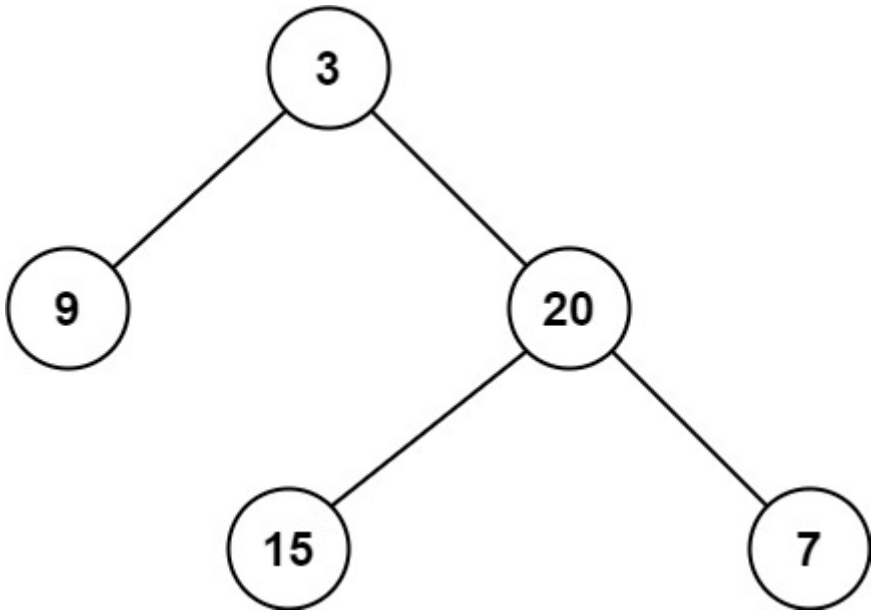
LeetCode	力扣	难度
111. Minimum Depth of Binary Tree	111. 二叉树的最小深度	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个二叉树，找出其最小深度，最小深度是从根节点到最近叶子节点（没有子节点的节点）的最短路径上的节点数量。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：2
```

基本思路

> 本文有视频版：BFS 算法核心框架套路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题可以同时运用两种思维，而且还可以用 BFS 层序遍历的方式求解，我会给出三种解法。

希望大家借助这道题仔细理解二叉树和 BFS 算法、动态规划算法、回溯算法 的联系。

- 详细题解：BFS 算法解题套路框架

解法代码

```java
// 「迭代」的递归思路
class Solution {
    private int minDepth = Integer.MAX_VALUE;
    private int currentDepth = 0;

    public int minDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        traverse(root);
        return minDepth;
    }

    private void traverse(TreeNode root) {
        if (root == null) {
            return;
        }

        // 做选择：在进入节点时增加当前深度
        currentDepth++;

        // 如果当前节点是叶子节点，更新最小深度
        if (root.left == null && root.right == null) {
            minDepth = Math.min(minDepth, currentDepth);
        }

        traverse(root.left);
        traverse(root.right);

        // 撤销选择：在离开节点时减少当前深度
        currentDepth--;
    }
}

// 「分解问题」的递归思路
class Solution2 {
    public int minDepth(TreeNode root) {
        // 基本情况：如果节点为空，返回深度为0
        if (root == null) {
            return 0;
        }

        // 递归计算左子树的最小深度
        int leftDepth = minDepth(root.left);
        // 递归计算右子树的最小深度
        int rightDepth = minDepth(root.right);

        // 特殊情况处理：如果左子树为空，返回右子树的深度加1
        if (leftDepth == 0) {
            return rightDepth + 1;
        }
        // 特殊情况处理：如果右子树为空，返回左子树的深度加1
        if (rightDepth == 0) {
```

```
                return leftDepth + 1;
            }

            // 计算并返回最小深度：左右子树深度的最小值加1
            return Math.min(leftDepth, rightDepth) + 1;
        }
    }


    // BFS 的思路
    class Solution3 {
        public int minDepth(TreeNode root) {
            if (root == null) return 0;
            Queue<TreeNode> q = new LinkedList<>();
            q.offer(root);
            // root 本身就是一层，depth 初始化为 1
            int depth = 1;

            while (!q.isEmpty()) {
                int sz = q.size();
                /* 遍历当前层的节点 */
                for (int i = 0; i < sz; i++) {
                    TreeNode cur = q.poll();
                    /* 判断是否到达叶子结点 */
                    if (cur.left == null && cur.right == null)
                        return depth;
                    /* 将下一层节点加入队列 */
                    if (cur.left != null)
                        q.offer(cur.left);
                    if (cur.right != null)
                        q.offer(cur.right);
                }
                /* 这里增加步数 */
                depth++;
            }
            return depth;
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 542. 01 矩阵 🟠
  - 752. 打开转盘锁 🟠
  - 剑指 Offer II 109. 开密码锁 🟠

542. 01 矩阵

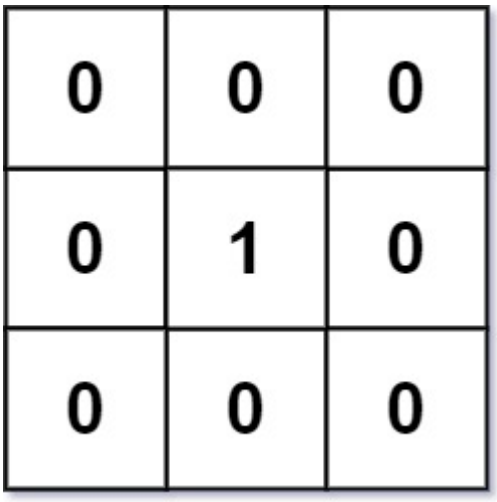
LeetCode	力扣	难度
542. 01 Matrix	542. 01 矩阵	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：**BFS 算法**

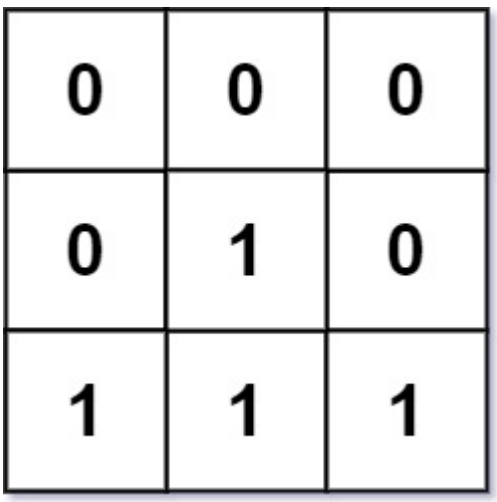
给定一个由 `0` 和 `1` 组成的矩阵 `mat`，请输出一个大小相同的矩阵，其中每一个格子是 `mat` 中对应位置元素到最近的 `0` 的距离。两个相邻元素间的距离为 `1`。

示例 1：




```
输入：mat = [[0,0,0],[0,1,0],[0,0,0]]
输出：[[0,0,0],[0,1,0],[0,0,0]]
```

示例 2：



```
输入：mat = [[0,0,0],[0,1,0],[1,1,1]]
输出：[[0,0,0],[0,1,0],[1,2,1]]
```

基本思路

我在第一眼看到这个题目时有两个思路方向：一是 动态规划，二是 BFS 算法。

何以见得可以用动态规划思路解决呢？

首先这是个求最值的问题，而且我感觉 `mat[x][y]` 离 0 最近的距离肯定能根据它周围元素 `mat[x+1][y], mat[x][y+1], mat[x-1][y], mat[x][y-1]` 离 0 最近的距离推算出来。这就说明可以比较容易地得到状态转移方程，那么写出动态规划解法也就没什么困难了。

何以见得可以用 BFS 算法思路解决呢？

因为这题描述的场景让我想到 111. 二叉树的最小深度 这类在二叉树中计算最小距离的题目，以及像 1091. 二进制矩阵中的最短路径 这种计算二维矩阵从左上角到右下角最短距离的题目。

再者，如果我把二维矩阵视为一幅「图」，那么最短距离肯定可以用图论中的最短距离算法计算出来。我在 Dijkstra 最短路径算法 中就讲到了，Dijkstra 算法就是 BFS 算法在加权图中的升级版。所以这题肯定能用普通 BFS 算法解决。

我这里就使用 BFS 算法来求解吧，使用 BFS 算法模板 中给出的框架改写一下就出来了，解法代码供参考。

解法代码

```
class Solution {
    public int[][] updateMatrix(int[][] mat) {
        int m = mat.length, n = mat[0].length;
        // 记录答案的结果数组
        int[][] res = new int[m][n];
        // 初始化全部填充特殊值 -1，代表未计算，
        // 待会可以用来判断坐标是否已经计算过，避免重复遍历
        for (int[] row : res) {
            Arrays.fill(row, -1);
        }

        Queue<int[]> q = new LinkedList<>();
        // 初始化队列，把那些值为 0 的坐标放到队列里
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (mat[i][j] == 0) {
                    q.offer(new int[]{i, j});
                    res[i][j] = 0;
                }
            }
        }
        // 执行 BFS 算法框架，从值为 0 的坐标开始向四周扩散
        int[][] dirs = new int[][]{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
```

```
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int x = cur[0], y = cur[1];
            // 向四周扩散
            for (int i = 0; i < 4; i++) {
                int nextX = x + dirs[i][0];
                int nextY = y + dirs[i][1];
                // 确保相邻的这个坐标没有越界且之前未被计算过
                if (nextX >= 0 && nextX < m && nextY >= 0 && nextY < n
                        && res[nextX][nextY] == -1) {
                    q.offer(new int[]{nextX, nextY});
                    // 从 mat[x][y] 走到 mat[nextX][nextY] 需要一步
                    res[nextX][nextY] = res[x][y] + 1;
                }
            }
        }

        return res;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

752. 打开转盘锁

LeetCode	力扣	难度
752. Open the Lock	752. 打开转盘锁	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有 10 个数字：`'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'`。每个拨轮可以自由旋转：例如把 `'9'` 变为 `'0'`，`'0'` 变为 `'9'`。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 `'0000'`，一个代表四个拨轮的数字的字符串。

列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 `target` 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回 `-1`。

示例 1:

```
输入：deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出：6
解释：
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" ->
"0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
因为当拨动到 "0102" 时这个锁就会被锁定。
```

基本思路

> 本文有视频版：BFS 算法核心框架套路

PS：这道题在《算法小抄》 的第 53 页。

本质上就是穷举，在避开 `deadends` 密码的前提下，对四位密码的每一位进行 0~9 的穷举。

根据 BFS 算法的性质，第一次拨出 `target` 时的旋转次数就是最少的，直接套 BFS 算法框架 即可。

另外，针对这道题的场景，还可以使用「双向 BFS」技巧进行优化，见详细题解。

- 详细题解：BFS 算法解题套路框架

解法代码

```
class Solution {
    public int openLock(String[] deadends, String target) {
        // 记录需要跳过的死亡密码
        Set<String> deads = new HashSet<>();
        for (String s : deadends) deads.add(s);
        // 记录已经穷举过的密码，防止走回头路
        Set<String> visited = new HashSet<>();
        Queue<String> q = new LinkedList<>();
        // 从起点开始启动广度优先搜索
        int step = 0;
        q.offer("0000");
        visited.add("0000");

        while (!q.isEmpty()) {
            int sz = q.size();
            /* 将当前队列中的所有节点向周围扩散 */
            for (int i = 0; i < sz; i++) {
                String cur = q.poll();

                /* 判断是否到达终点 */
                if (deads.contains(cur))
                    continue;
                if (cur.equals(target))
                    return step;

                /* 将一个节点的未遍历相邻节点加入队列 */
                for (int j = 0; j < 4; j++) {
                    String up = plusOne(cur, j);
                    if (!visited.contains(up)) {
                        q.offer(up);
                        visited.add(up);
                    }
                    String down = minusOne(cur, j);
                    if (!visited.contains(down)) {
                        q.offer(down);
                        visited.add(down);
                    }
                }
            }
            /* 在这里增加步数 */
            step++;
        }
        // 如果穷举完都没找到目标密码，那就是找不到了
        return -1;
    }

    // 将 s[j] 向上拨动一次
    String plusOne(String s, int j) {
        char[] ch = s.toCharArray();
        if (ch[j] == '9')
            ch[j] = '0';
        else
            ch[j] += 1;
```

```
            return new String(ch);
        }

        // 将 s[i] 向下拨动一次
        String minusOne(String s, int j) {
            char[] ch = s.toCharArray();
            if (ch[j] == '0')
                ch[j] = '9';
            else
                ch[j] -= 1;
            return new String(ch);
        }
    }
```

- **类似题目**：
  - 111. 二叉树的最小深度 🟢
  - 剑指 Offer II 109. 开密码锁 🟠

剑指 Offer II 109. 开密码锁

这道题和 752. 打开转盘锁 相同。

1926. 迷宫中离入口最近的出口

LeetCode	力扣	难度
1926. Nearest Exit from Entrance in Maze	1926. 迷宫中离入口最近的出口	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

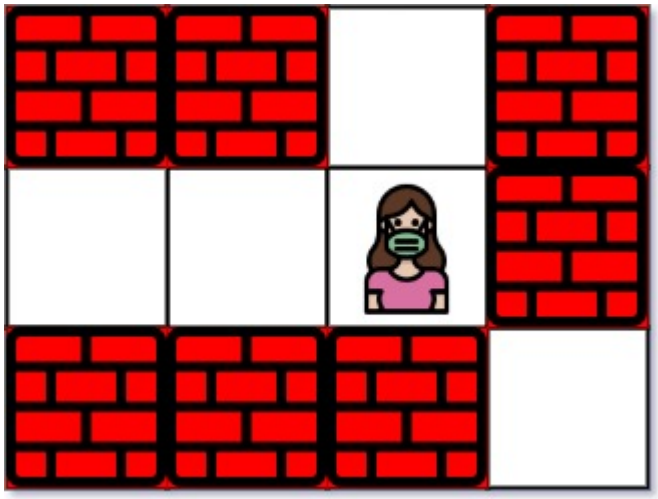
- 标签：BFS 算法

给你一个 `m x n` 的迷宫矩阵 `maze` （**下标从 0 开始**），矩阵中有空格子（用 `'.'` 表示）和墙（用 `'+'` 表示）。同时给你迷宫的入口 `entrance` ，用 `entrance = [entrancerow, entrancecol]` 表示你一开始所在格子的行和列。

每一步操作，你可以往 **上**，**下**，**左** 或者**右** 移动一个格子。你不能进入墙所在的格子，你也不能离开迷宫。你的目标是找到离 `entrance` **最近** 的出口。**出口** 的含义是 `maze` **边界** 上的 **空格子**。`entrance` 格子 **不算** 出口。

请你返回从 `entrance` 到最近出口的最短路径的**步数** ，如果不存在这样的路径，请你返回 `-1` 。

示例 1：



```
输入：maze = [["+","+",".","+"],[".",".",".","+"],["+","+","+","."]],
entrance = [1,2]
输出：1
解释：总共有 3 个出口，分别位于 (1,0)，(0,2) 和 (2,3)。
一开始，你在入口格子 (1,2) 处。
- 你可以往左移动 2 步到达 (1,0)。
- 你可以往上移动 1 步到达 (0,2)。
从入口处没法到达 (2,3)。
所以，最近的出口是 (0,2)，距离为 1 步。
```

基本思路

这道题非常简单，就是标准的 BFS 算法，只要套用 BFS 算法模板框架 就可以了，直接看代码吧。另外我建议做一下 286. 墙与门，比这道题有意思一些。

解法代码

```
class Solution {
    public int nearestExit(char[][] maze, int[] entrance) {
        int m = maze.length;
        int n = maze[0].length;
        int[][] dirs = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

        // BFS 算法的队列和 visited 数组
        Queue<int[]> queue = new LinkedList<>();
        boolean[][] visited = new boolean[m][n];
        queue.offer(entrance);
        visited[entrance[0]][entrance[1]] = true;
        // 启动 BFS 算法从 entrance 开始像四周扩散
        int step = 0;
        while (!queue.isEmpty()) {
            int sz = queue.size();
            step++;
            // 扩散当前队列中的所有节点
            for (int i = 0; i < sz; i++) {
                int[] cur = queue.poll();
                // 每个节点都会尝试向上下左右四个方向扩展一步
                for (int[] dir : dirs) {
                    int x = cur[0] + dir[0];
                    int y = cur[1] + dir[1];
                    if (x < 0 || x >= m || y < 0 || y >= n
                            || visited[x][y] || maze[x][y] == '+') {
                        continue;
                    }
                    if (x == 0 || x == m - 1 || y == 0 || y == n - 1) {
                        // 走到边界（出口）
                        return step;
                    }
                    visited[x][y] = true;
                    queue.offer(new int[]{x, y});
                }
            }
        }
        return -1;
    }
}
```

- **类似题目**：
  - 490. 迷宫 🟠

286. 墙与门

LeetCode	力扣	难度
286. Walls and Gates	286. 墙与门	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法

你被给定一个 `m × n` 的二维网格 `rooms` ，网格中有以下三种可能的初始化值：

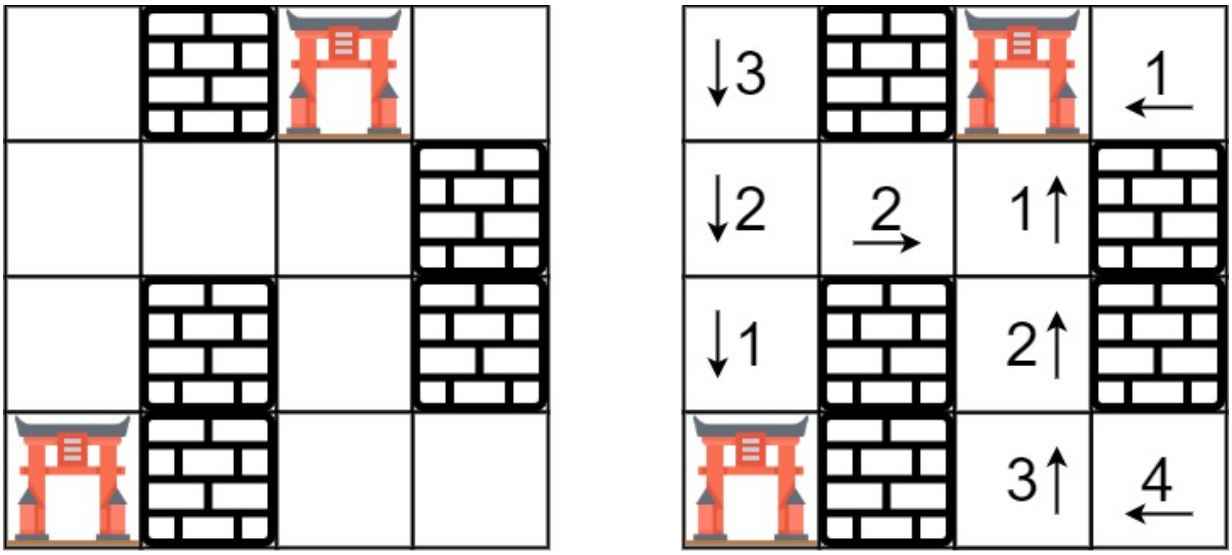
1、`-1` 表示墙或是障碍物

2、`0` 表示一扇门

3、`INF` 无限表示一个空的房间。然后，我们用 `2^31 - 1 = 2147483647` 代表 `INF`。你可以认为通往门的距离总是小于 `2147483647` 的。

你要给每个空房间位上填上该房间到 **最近门的距离** ，如果无法到达门，则填 `INF` 即可。

示例 1：




```
输入：rooms = [[2147483647,-1,0,2147483647],
[2147483647,2147483647,2147483647,-1],[2147483647,-1,2147483647,-1],
[0,-1,2147483647,2147483647]]
输出：[[3,-1,0,1],[2,2,1,-1],[1,-1,2,-1],[0,-1,3,4]]
```

基本思路

这道题看起来有些困难，比如有的读者可能会想到 Dijkstra 算法 这样比较复杂的最短路径算法，或者类似 毕业旅行省钱算法 这种利用动态规划算法在约束之下求最小值的题型。但实际上这道题并不难，只需要简单的 BFS 算法框架 就能解决。

我在 BFS 算法框架套路详解 中介绍过，BFS 算法的特色是一旦遍历到一个点，那么 BFS 执行的步数就是最近距离。

本题中可以理解为每个「门」都是一个起点，所以要把所有门的坐标都放到队列里开始执行 BFS 算法，当第一次遍历到「房间」时，所走的步数就是该房间到最近的门的距离。具体看代码吧。

解法代码

```
class Solution {

    public void wallsAndGates(int[][] rooms) {
        final int ROOM = Integer.MAX_VALUE;
        final int GATE = 0;

        int m = rooms.length;
        int n = rooms[0].length;

        Queue<int[]> q = new LinkedList<>();
        // 把所有的门放入队列，作为 BFS 遍历的起点
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (rooms[i][j] == GATE) {
                    q.add(new int[]{i, j});
                }
            }
        }

        int[][] dirs = new int[][]{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
        // 开始执行 BFS 算法，根据 BFS 算法的特性，第一次遍历到新节点所走的步数就是最
短距离
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int x = cur[0], y = cur[1];
            for (int[] d : dirs) {
                int next_x = x + d[0], next_y = y + d[1];
                if (next_x < 0 || next_y < 0 || next_x >= m || next_y >= n
                        || rooms[next_x][next_y] != ROOM) {
                    // 这个位置是障碍物或者已经计算出最短距离了，跳过
                    continue;
                }
                // 计算出 (next_x, next_y) 达到最近的门的距离
                rooms[next_x][next_y] = rooms[x][y] + 1;
                q.offer(new int[]{next_x, next_y});
            }
        }
    }
}
```

- **类似题目**：
  - 1926. 迷宫中离入口最近的出口 🟠

310. 最小高度树

LeetCode	力扣	难度
310. Minimum Height Trees	310. 最小高度树	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，图论算法

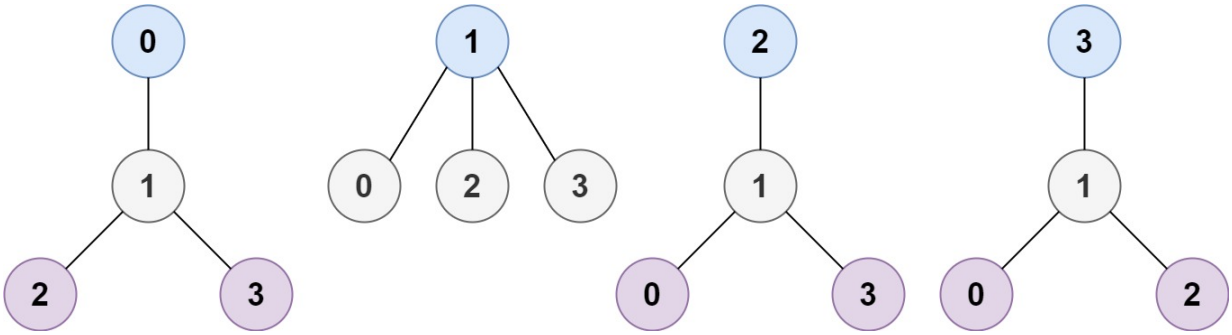
树是一个无向图，其中任何两个顶点只通过一条路径连接。换句话说，一个任何没有简单环路的连通图都是一棵树。

给你一棵包含 `n` 个节点的树，标记为 `0` 到 `n - 1`。给定数字 `n` 和一个有 `n - 1` 条无向边的 `edges` 列表（每一个边都是一对标签），其中 `edges[i] = [ai, bi]` 表示树中节点 `ai` 和 `bi` 之间存在一条无向边。

可选择树中任何一个节点作为根。当选择节点 `x` 作为根节点时，设结果树的高度为 `h`。在所有可能的树中，具有最小高度的树（即，`min(h)`）被称为**最小高度树**。

请你找到所有的 **最小高度树** 并按 **任意顺序**返回它们的根节点标签列表。树的 **高度** 是指根节点和叶子节点之间最长向下路径上边的数量。

示例 1：




```
输入：n = 4, edges = [[1,0],[1,2],[1,3]]
输出：[1]
解释：如图所示，当根是标签为 1 的节点时，树的高度是 1，这是唯一的最小高度树。
```

基本思路

我认为这道题非常有意思，关于图的题目，我已经写了如下的算法：

1、图论算法基础

2、二分图判定算法

3、环检测和拓扑排序算法

4、Dijkstra 最短路径算法

5、Union Find 并查集算法

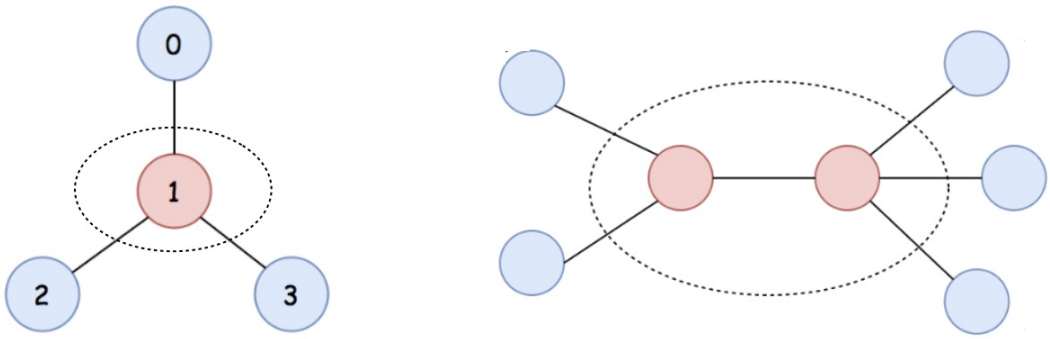
6、Kruskal 最小生成树算法

这道题肯定要用到 图论算法基础 中讲的图的编程表示（邻接表/邻接矩阵），感觉和 拓扑排序算法 和 Kruskal 最小生成树算法 又有点关系。

实际上这道题用标准的 BFS 算法框架 就可以解决，只是需要一些改变。我直接说解题思路，很容易理解：

1、首先，我们按照 图论算法基础 中所讲的，把题目输入的 edges 转换成邻接表。

2、然后，我们从叶子节点开始，一层一层地删除叶子节点（每删除一层叶子节点，就会产生新的叶子节点），直到剩下的节点数小于等于 2 个为止。之所以是 2 个而不是 1 个，是因为如果输入的这幅图两边完全对称，可能出现两个节点都可以作为根节点的情况。



3、最后剩下的这些节点，就是我们要找的最小高度树的根节点。

如何一层一层删除所有叶子节点呢？只要用 BFS 算法，借助一个队列就可以了，具体实现看代码吧。

肯定有读者会问，这种题怎么能想出来呢？实际上这种题可以理解为类似冒泡排序这种套路题，找最近叶子节点就从根开始 BFS，找根节点的话就从叶子开始 BFS，你记住这种处理方式就好了，一般不会有什么变体。

解法代码

```
class Solution {
    public List<Integer> findMinHeightTrees(int n, int[][] edges) {
        // 1、构建邻接表
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            graph.add(new ArrayList<Integer>());
        }
        for (int[] edge : edges) {
            // 无向图，等同于双向图
            graph.get(edge[0]).add(edge[1]);
            graph.get(edge[1]).add(edge[0]);
        }

        // 2、找到所有的叶子节点
        List<Integer> leaves = new ArrayList<>();
```

```
        for (int i = 0; i < n; i++) {
            if (graph.get(i).size() == 1) {
                leaves.add(i);
            }
        }

        // 3、不断删除叶子节点，直到剩下的节点数小于等于 2 个
        int remainNodeNum = n;
        while (remainNodeNum > 2) {
            // 删除当前叶子节点，计算新的叶子节点
            remainNodeNum -= leaves.size();
            List<Integer> newLeaves = new ArrayList<>();
            for (int leaf : leaves) {
                // 将被删除的叶子节点的邻接节点的度减 1
                int neighbor = graph.get(leaf).get(0);
                graph.get(neighbor).remove(leaf);
                // 如果邻接节点的度为 1，说明它也变成了叶子节点
                if (graph.get(neighbor).size() == 1) {
                    newLeaves.add(neighbor);
                }
            }

            // 最后剩下的节点就是根节点
            return leaves;
        }
    }
```

429. N 叉树的层序遍历

LeetCode	力扣	难度
429. N-ary Tree Level Order Traversal	429. N 叉树的层序遍历	🟠

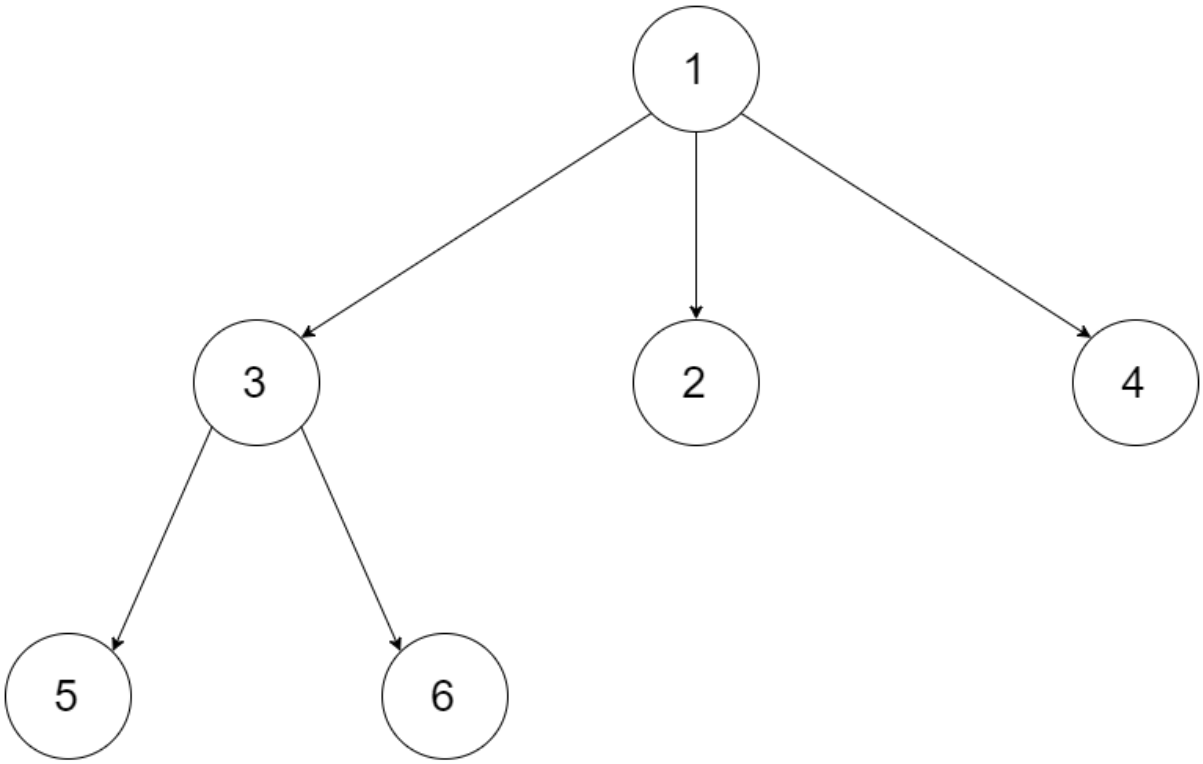
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

给定一个 N 叉树，返回其节点值的*层序遍历*。（即从左到右，逐层遍历）。

树的序列化输入是用层序遍历，每组子节点都由 null 值分隔（参见示例）。

示例 1：




```
输入：root = [1,null,3,2,4,null,5,6]
输出：[[1],[3,2,4],[5,6]]
```

基本思路

解法代码

```
class Solution {
    public List<List<Integer>> levelOrder(Node root) {
        List<List<Integer>> res = new LinkedList<>();
```

```
        if (root == null) {
            return res;
        }

        // 以下是标准的 BFS 算法遍历框架
        Queue<Node> q = new LinkedList<>();
        q.offer(root);

        while (!q.isEmpty()) {
            int sz = q.size();
            List<Integer> level = new LinkedList<>();
            for (int i = 0; i < sz; i++) {
                Node cur = q.poll();
                level.add(cur.val);
                // 多叉树可能有多个子节点，将所有子节点都加入队列
                for (Node child : cur.children) {
                    q.offer(child);
                }
            }
            // 当前层遍历完毕
            res.add(level);
        }
        return res;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

582. 杀掉进程

LeetCode	力扣	难度
582. Kill Process	582. 杀掉进程	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：BFS 算法，二叉树

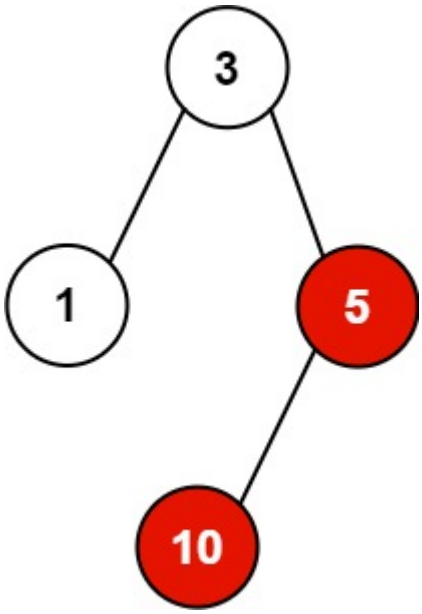
系统中存在 `n` 个进程，形成一个有根树结构。给你两个整数数组 `pid` 和 `ppid`，其中 `pid[i]` 是第 `i` 个进程的 ID，`ppid[i]` 是第 `i` 个进程的父进程 ID。

每一个进程只有 **一个父进程**，但是可能会有 **一个或者多个子进程**。只有一个进程的 `ppid[i] = 0`，意味着这个进程 **没有父进程**。

当一个进程 **被杀掉** 的时候，它所有的子进程和后代进程都要被杀掉。

给你一个整数 `kill` 表示要杀掉进程的 ID，返回杀掉该进程后的所有进程 ID 的列表。可以按 **任意顺序** 返回答案。

示例 1：




```
输入：pid = [1,3,10,5], ppid = [3,0,5,3], kill = 5
输出：[5,10]
解释：涂为红色的进程是应该被杀掉的进程。
```

基本思路

前文 手把手刷二叉树总结篇 说过二叉树的递归分为「遍历」和「分解问题」两种思维模式，这道题需要用到「遍历」的思维模式。

我觉得这个题非常有趣，先说解法吧，其实不算难。

如果是一棵常规的二叉树，让你从中摘除一棵子树，你会不会？应该很简单对吧，遍历一遍找到子树的根节点，然后从整棵树中摘除就行了。

但本题的难点是我们只有子节点到父节点的引用，和常规二叉树不同，这样就无法用常规的二叉树遍历算法从根节点去遍历整棵树了。

怎么办？很简单，把它转化成常规的二叉树不就行了？好，那怎么转换呢？

这也是我说这题有趣的原因，如果你看过 图论算法基础 和 Union-Find 算法详解 这两篇文章，你就会会心一笑。

首先，题目给你输入的这种树的形式，就是 [Union-Find 算法]](https://labuladong.github.io/article/fname.html?fname=UnionFind算法详解) 中对树的存储形式，其目的是快速计算某个节点的根节点，你如果忘了的话可以去复习一下。

其次，题目给你输入的这两个数组其实就是若干条有向边（`pid[i]->ppid[i]`），那么我们就可以用 图论算法基础 中介绍的通用方式（邻接表/邻接矩阵）去存储。

说了这么多，思路就呼之欲出了：

先把题目输入的这棵树转化成常规形式（父节点指向子节点），然后执行遍历算法（DFS/BFS 都可以）去遍历删除以目标节点 `kill` 为根的整棵子树。

解法代码

```
class Solution {
    public List<Integer> killProcess(List<Integer> pid, List<Integer>
ppid, int kill) {
        // 构建多叉树，key 为父节点，value 为所有子节点的列表
        HashMap<Integer, List<Integer>> tree = new HashMap<>();
        for (int i = 0; i < pid.size(); i++) {
            int child = pid.get(i);
            int parent = ppid.get(i);
            tree.putIfAbsent(parent, new ArrayList<>());
            tree.get(parent).add(child);
        }

        List<Integer> res = new LinkedList<>();
        // 我这里用 BFS 算法遍历子树，删除以 kill 为根的所有子节点
        Queue<Integer> q = new LinkedList<>();
        q.offer(kill);
        while (!q.isEmpty()) {
            int cur = q.poll();
            res.add(cur);
            if (tree.containsKey(cur)) {
                // 所有子节点入队
                q.addAll(tree.get(cur));
            }
        }
        return res;
```

```
        }
    }
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

773. 滑动谜题

LeetCode	力扣	难度
773. Sliding Puzzle	773. 滑动谜题	🔴

- **标签：BFS 算法，字符串**

在一个 2 x 3 的棋盘 `board` 上有 5 块卡片，用数字 `1~5` 来表示，以及一块空缺用 `0` 来表示。一次「移动」定义为选择 `0` 与一个相邻的数字（上下左右）进行交换。

进行若干次移动，使得 `board` 的结果是 `[[1,2,3],[4,5,0]]`，则谜板被解开。

给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能通过移动解开谜板，则返回 -1。

示例：

```
输入：board = [[1,2,3],[4,0,5]]
输出：1
解释：交换 0 和 5，1 步完成

输入：board = [[1,2,3],[5,4,0]]
输出：-1
解释：没有办法完成谜板

输入：board = [[4,1,2],[5,0,3]]
输出：5
解释：
最少完成谜板的最少移动次数是 5，
一种移动路径：
尚未移动：[[4,1,2],[5,0,3]]
移动 1 次：[[4,1,2],[0,5,3]]
移动 2 次：[[0,1,2],[4,5,3]]
移动 3 次：[[1,0,2],[4,5,3]]
移动 4 次：[[1,2,0],[4,5,3]]
移动 5 次：[[1,2,3],[4,5,0]]

输入：board = [[3,2,4],[1,5,0]]
输出：14
```

基本思路

PS：这道题在《算法小抄》 的第 310 页。

这题可以用 BFS 算法解决。BFS 算法并不只是一个寻路算法，而是一种暴力搜索算法，只要涉及暴力穷举的问题，BFS 就可以用，而且可以最快地穷举出答案，关于 BFS 算法原理可以看 BFS 算法框架。

- **详细题解：如何用 BFS 算法秒杀各种智力题**

解法代码

```
class Solution {
    public int slidingPuzzle(int[][] board) {
        int m = 2, n = 3;
        StringBuilder sb = new StringBuilder();
        String target = "123450";
        // 将 2x3 的数组转化成字符串作为 BFS 的起点
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                sb.append(board[i][j]);
            }
        }
        String start = sb.toString();

        // 记录一维字符串的相邻索引
        int[][] neighbor = new int[][]{
                {1, 3},
                {0, 4, 2},
                {1, 5},
                {0, 4},
                {3, 1, 5},
                {4, 2}
        };

        /******* BFS 算法框架开始 *******/
        Queue<String> q = new LinkedList<>();
        HashSet<String> visited = new HashSet<>();
        // 从起点开始 BFS 搜索
        q.offer(start);
        visited.add(start);

        int step = 0;
        while (!q.isEmpty()) {
            int sz = q.size();
            for (int i = 0; i < sz; i++) {
                String cur = q.poll();
                // 判断是否达到目标局面
                if (target.equals(cur)) {
                    return step;
                }
                // 找到数字 0 的索引
                int idx = 0;
                for (; cur.charAt(idx) != '0'; idx++) ;
                // 将数字 0 和相邻的数字交换位置
                for (int adj : neighbor[idx]) {
                    String new_board = swap(cur.toCharArray(), adj, idx);
```

```
                    // 防止走回头路
                    if (!visited.contains(new_board)) {
                        q.offer(new_board);
                        visited.add(new_board);
                    }
                }
            }
            step++;
        }
        /******* BFS 算法框架结束 *******/
        return -1;
    }

    private String swap(char[] chars, int i, int j) {
        char temp = chars[i];
        chars[i] = chars[j];
        chars[j] = temp;
        return new String(chars);
    }

}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

1091. 二进制矩阵中的最短路径

LeetCode	力扣	难度
1091. Shortest Path in Binary Matrix	1091. 二进制矩阵中的最短路径	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：BFS 算法，二维矩阵**

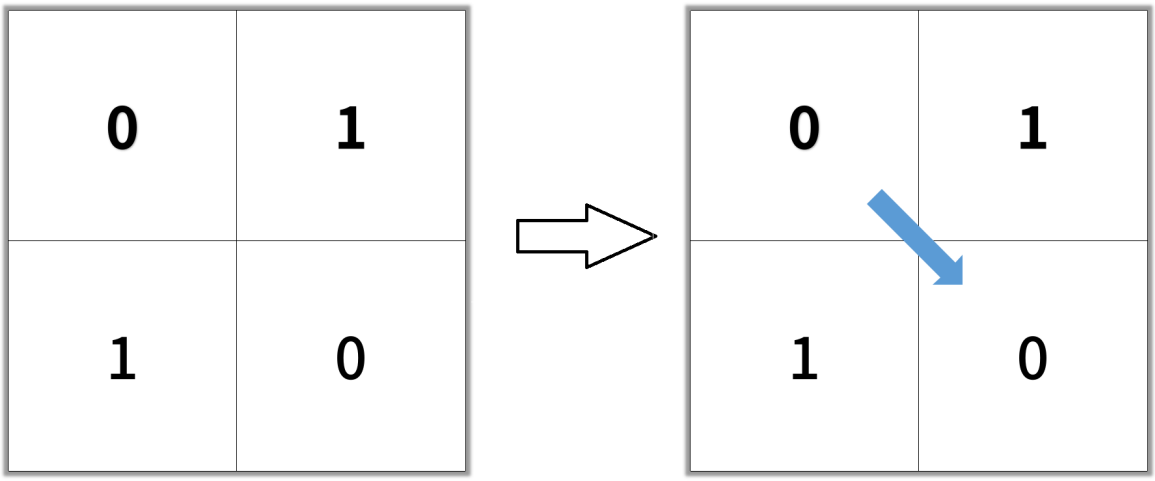
给你一个 `n x n` 的二进制矩阵 `grid` 中，返回矩阵中最短 **畅通路径**的长度。如果不存在这样的路径，返回 `-1`。

二进制矩阵中的 畅通路径 是一条从 **左上角** 单元格（即，`(0, 0)`）到右下角 单元格（即，`(n - 1, n - 1)`）的路径，该路径同时满足下述要求：

- 路径途经的所有单元格都的值都是 `0`。- 路径中所有相邻的单元格应当在 **8 个方向之一** 上连通（即，相邻两单元之间彼此不同且共享一条边或者一个角）。

畅通路径的长度 是该路径途经的单元格总数。

示例 1：




```
输入：grid = [[0,1],[1,0]]
输出：2
```

基本思路

这道题的思路应该很明显，BFS 算法 肯定可以解决。

一般我们二维矩阵相关的题目只允许上下左右移动，这里还允许斜着移动，只要稍微改一改 BFS 算法框架 中的方向数组 `dirs` 即可。

解法代码

```
class Solution {
    public int shortestPathBinaryMatrix(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        if (grid[0][0] == 1 || grid[m - 1][n - 1] == 1) {
            return -1;
        }

        Queue<int[]> q = new LinkedList<>();
        // 需要记录走过的路径，避免死循环
        boolean[][] visited = new boolean[m][n];

        // 初始化队列，从 (0, 0) 出发
        q.offer(new int[]{0, 0});
        visited[0][0] = true;
        int pathLen = 1;

        // 执行 BFS 算法框架，从值为 0 的坐标开始向八个方向扩散
        int[][] dirs = new int[][]{
                {0, 1}, {0, -1}, {1, 0}, {-1, 0},
                {1, 1}, {1, -1}, {-1, 1}, {-1, -1}
        };
        while (!q.isEmpty()) {
            int sz = q.size();
            for (int __ = 0; __ < sz; __++) {
                int[] cur = q.poll();
                int x = cur[0], y = cur[1];
                if (x == m - 1 && y == n - 1) {
                    return pathLen;
                }
                // 向八个方向扩散
                for (int i = 0; i < 8; i++) {
                    int nextX = x + dirs[i][0];
                    int nextY = y + dirs[i][1];
                    // 确保相邻的这个坐标没有越界且值为 0 且之前没有走过
                    if (nextX >= 0 && nextX < m && nextY >= 0 && nextY < n
                            && grid[nextX][nextY] == 0 && !visited[nextX]
[nextY]) {
                        q.offer(new int[]{nextX, nextY});
                        visited[nextX][nextY] = true;
                    }
                }
            }
            pathLen++;
        }
        return -1;
    }
}
```

- **类似题目**：

- 542. 01 矩阵 🟠

490. 迷宫

LeetCode	力扣	难度
490. The Maze	490. 迷宫	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

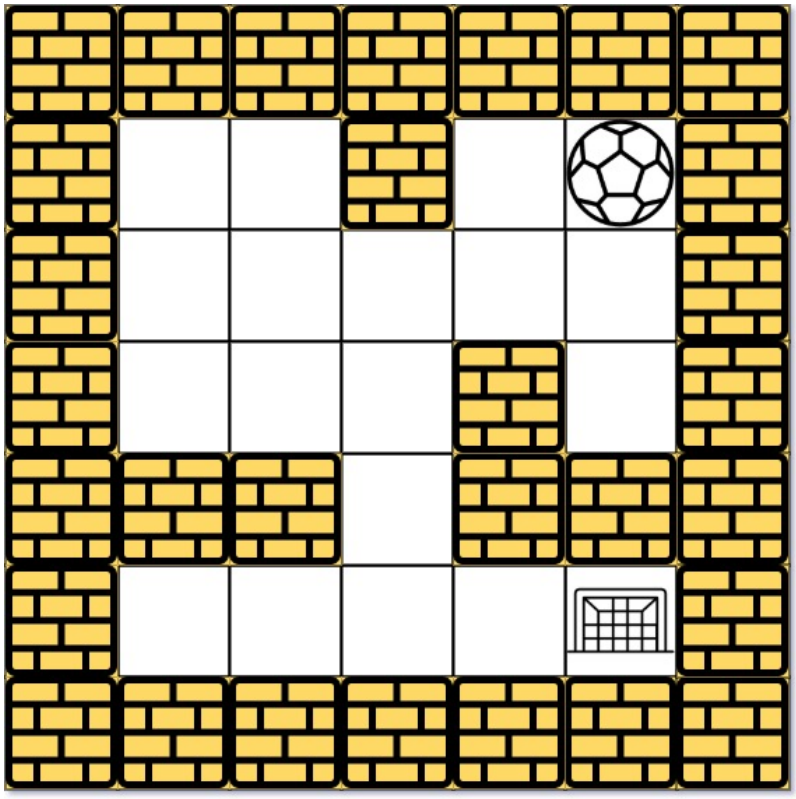
- 标签：BFS 算法

由空地（用 `0` 表示）和墙（用 `1` 表示）组成的迷宫 `maze` 中有一个球。球可以途经空地向 **上**、**下**、**左**、**右**四个方向滚动，且在遇到墙壁前不会停止滚动。当球停下时，可以选择向下一个方向滚动。

给你一个大小为 `m x n` 的迷宫 `maze`，以及球的初始位置 `start` 和目的地 `destination`，其中 `start = [startrow, startcol]` 且 `destination = [destinationrow, destinationcol]`。请你判断球能否在目的地停下：如果可以，返回 `true`；否则，返回 `false`。

你可以 **假定迷宫的边缘都是墙壁**（参考示例）。

示例 1：



```
输入：maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],
start = [0,4], destination = [4,4]
输出：true
解释：一种可能的路径是 ： 左 -> 下 -> 左 -> 下 -> 右 -> 下 -> 右。
```

基本思路

这道题看起来和 1926. 迷宫中离入口最近的出口 这种标准的二维数组的最短路径算法不太一样对吧？但实际上换汤不换药，只是这道题向四周扩散时是一直走到墙，而不是只走一步。

直接套用 BFS 算法框架 即可，只需要稍微修改一下四面扩散的逻辑即可，具体看代码吧。

解法代码

```
class Solution {
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        int m = maze.length, n = maze[0].length;
        // 方向数组，方便上下左右移动
        int[][] dirs = {{0,1},{0,-1},{1,0},{-1,0}};

        // BFS 算法的队列和 visited 数组
        Queue<int[]> q = new LinkedList<>();
        boolean[][] visited = new boolean[m][n];
        // 加入起点
        q.offer(start);
        visited[start[0]][start[1]] = true;
        // 启动 BFS 算法框架
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            // 向四个方向扩展
            for (int[] dir : dirs) {
                int x = cur[0], y = cur[1];
                // 和其他题目不同的是，这里一直走到墙，而不是只走一步
                while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] ==
0) {
                    x += dir[0];
                    y += dir[1];
                }
                x -= dir[0];
                y -= dir[1];
                // 判断是否到达终点
                if (x == destination[0] && y == destination[1]) {
                    return true;
                }
                if (!visited[x][y]) {
                    visited[x][y] = true;
                    q.offer(new int[]{x, y});
                }
            }
        }
        return false;
    }
}
```

- **类似题目**：
  - 505. 迷宫 II 🟠

505. 迷宫 II

LeetCode	力扣	难度
505. The Maze II	505. 迷宫 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

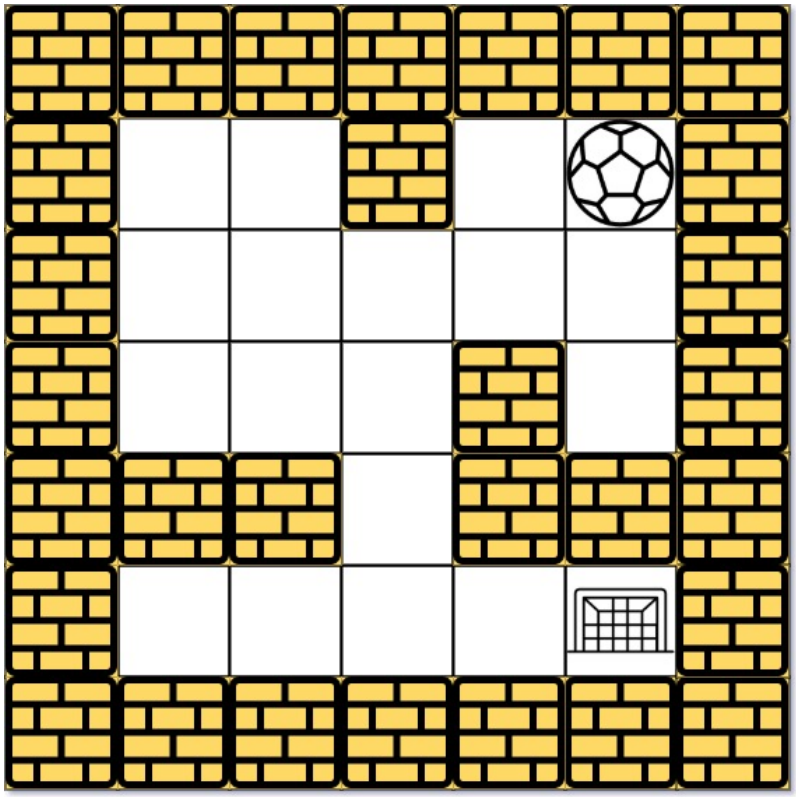
- 标签：BFS 算法

迷宫中有一个球，它有空地 (表示为 `0`) 和墙 (表示为 `1`)。球可以**向上**、**向下**、**向左**或**向右**滚过空地，但直到撞上墙之前它都不会停止滚动。当球停止时，它才可以选择下一个滚动方向。

给定 `m × n` 的**迷宫**(`maze`)，球的起始位置 (`start = [startrow, startcol]`) 和目的地 (`destination = [destinationrow, destinationcol]`)，返回球在目的地 (`destination`) 停止的最短**距离**。如果球不能在**目的地**(`destination`) 停止，返回 `-1`。

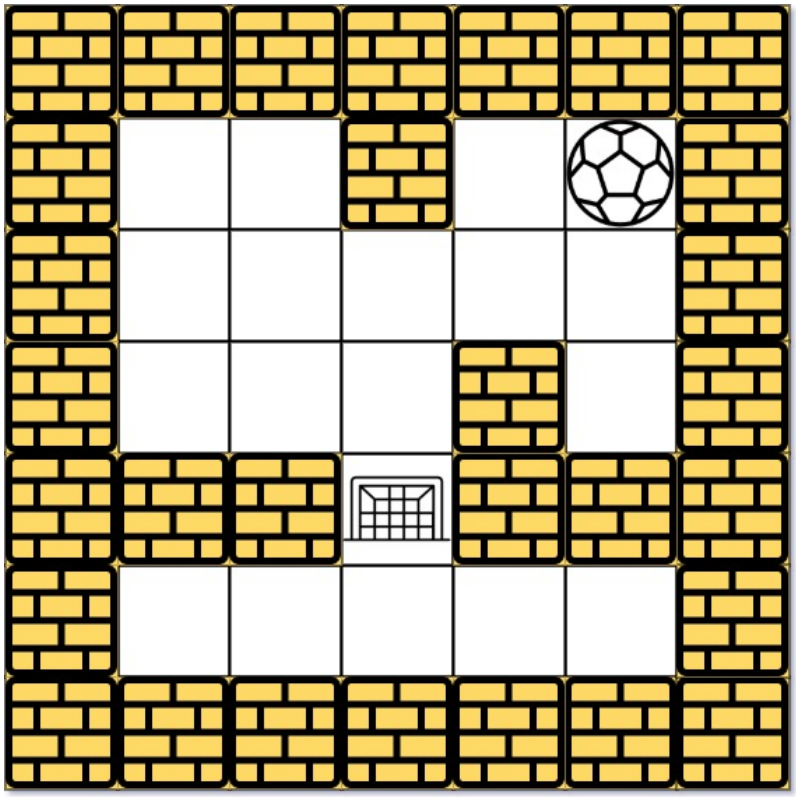
距离是指球从起始位置 (不包括) 到终点 (包括) 所经过的**空地**数。你可以假设**迷宫的边界都是墙**(见例子)。

示例 1:



```
输入: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],
start = [0,4], destination = [4,4]
输出: 12
解析: 一条最短路径 : left -> down -> left -> down -> right -> down -> right。
             总距离为 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12。
```

示例 2:



```
输入：maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]],
start = [0,4], destination = [3,2]
输出：-1
解析：球不可能在目的地停下来。注意，你可以经过目的地，但不能在那里停下来。
```

示例 3:

```
输入：maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]],
start = [4,3], destination = [0,1]
输出：-1
```

基本思路

这道题是 490. 迷宫 的升级版，这里要求的是最短路径，而不是判断是否存在路径。不过依然很简单，只要在向四周扩散的时候记录一下步数即可，一套 BFS 算法框架 就出来了，具体看代码吧。

解法代码

```
class Solution {
    public int shortestDistance(int[][] maze, int[] start, int[]
destination) {
        int m = maze.length, n = maze[0].length;
```

```
        // 方向数组，方便上下左右移动
        int[][] dirs = {{0,1},{0,-1},{1,0},{-1,0}};

        // BFS 算法的队列和 visited 数组
        Queue<int[]> q = new LinkedList<>();
        boolean[][] visited = new boolean[m][n];
        // 加入起点
        q.offer(start);
        visited[start[0]][start[1]] = true;
        int step = 0;
        // 启动 BFS 算法框架
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            // 向四个方向扩展
            for (int[] dir : dirs) {
                int x = cur[0], y = cur[1];
                // 和其他题目不同的是，这里一直走到墙，而不是只走一步，同时要记录走过
的步数
                while (x >= 0 && x < m && y >= 0 && y < n && maze[x][y] ==
0) {
                    x += dir[0];
                    y += dir[1];
                    step++;
                }
                x -= dir[0];
                y -= dir[1];
                step--;
                // 判断是否到达终点
                if (x == destination[0] && y == destination[1]) {
                    return step;
                }
                if (!visited[x][y]) {
                    visited[x][y] = true;
                    q.offer(new int[]{x, y});
                }
            }
        }
        return -1;
    }
}
```

45. 跳跃游戏 II

LeetCode	力扣	难度
45. Jump Game II	45. 跳跃游戏 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划，贪心算法**

给你一个非负整数数组 `nums`，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，请你使用最少的跳跃次数到达数组的最后一个位置（假设你总是可以到达数组的最后一个位置）。

示例 1:

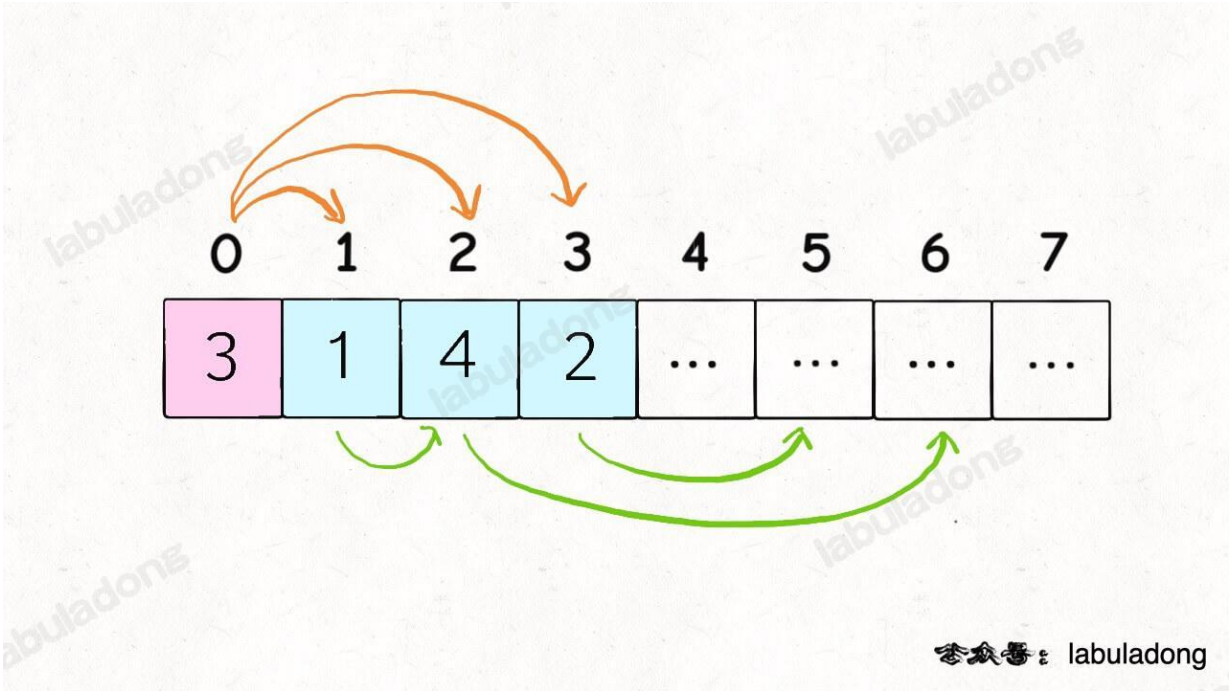
```
输入：nums = [2,3,1,1,4]
输出：2
解释：跳到最后一个位置的最小跳跃数是 2。
     从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
```

基本思路

PS：这道题在《算法小抄》 的第 376 页。

常规的思维就是暴力穷举，把所有可行的跳跃方案都穷举出来，计算步数最少的。穷举的过程会有重叠子问题，用备忘录消除一下，就成了自顶向下的动态规划。

不过直观地想一想，似乎不需要穷举所有方案，只需要判断哪一个选择最具有「潜力」即可，这就是贪心思想来做，比动态规划效率更高。



比如上图这种情况，我们站在索引 0 的位置，可以向前跳 1，2 或 3 步，你说应该选择跳多少呢？

显然应该跳 2 步调到索引 2，因为 `nums[2]` 的可跳跃区域涵盖了索引区间 `[3..6]`，比其他的都大。

这就是思路，我们用 `i` 和 `end` 标记了可以选择的跳跃步数，`farthest` 标记了所有选择 `[i..end]` 中能够跳到的最远距离，`jumps` 记录跳跃次数。

- **详细题解：如何运用贪心思想玩跳跃游戏**

解法代码

```
class Solution {
    public int jump(int[] nums) {
        int n = nums.length;
        int end = 0, farthest = 0;
        int jumps = 0;
        for (int i = 0; i < n - 1; i++) {
            farthest = Math.max(nums[i] + i, farthest);
            if (end == i) {
                jumps++;
                end = farthest;
            }
        }
        return jumps;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- 类似题目：
  - 55. 跳跃游戏 🟠

55. 跳跃游戏

LeetCode	力扣	难度
55. Jump Game	55. 跳跃游戏	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划，贪心算法**

给定一个非负整数数组 nums，你最初位于数组的**第一个下标**，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下标。

示例 1：

```
输入：nums = [2,3,1,1,4]
输出：true
解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。
```

示例 2：

```
输入：nums = [3,2,1,0,4]
输出：false
解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远不可能到达最后一个下标。
```

基本思路

PS：这道题在《算法小抄》的第 376 页。

这道题表面上不是求最值，但是可以改一改：

请问通过题目中的跳跃规则，最多能跳多远？如果能够越过最后一格，返回 true，否则返回 false。

所以解题关键在于求出能够跳到的最远距离。

- **详细题解：如何运用贪心思想玩跳跃游戏**

解法代码

```
class Solution {
    public boolean canJump(int[] nums) {
        int n = nums.length;
        int farthest = 0;
        for (int i = 0; i < n - 1; i++) {
```

```
                // 不断计算能跳到的最远距离
                farthest = Math.max(farthest, i + nums[i]);
                // 可能碰到了 0，卡住跳不动了
                if (farthest <= i) {
                    return false;
                }
            }
            return farthest >= n - 1;
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 45. 跳跃游戏 II 🟠

70. 爬楼梯

LeetCode	力扣	难度
70. Climbing Stairs	70. 爬楼梯	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划**

假设你正站在第 0 层楼，需要爬 n（n 是正整数）级台阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶，你有多少种不同的方法可以爬到楼顶呢?

示例 1：

```
输入：3
输出：3
解释：有 3 种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

基本思路

这题属于最基本的动态规划，建议先看下前文 动态规划框架详解。

这题很像 509. 斐波那契数：爬到第 n 级台阶的方法个数等于爬到 n - 1 的方法个数和爬到 n - 2 的方法个数之和。

解法代码

```
class Solution {
    // 备忘录
    int[] memo;

    public int climbStairs(int n) {
        memo = new int[n + 1];
        return dp(n);
    }

    // 定义：爬到第 n 级台阶的方法个数为 dp(n)
    int dp(int n) {
        // base case
        if (n <= 2) {
            return n;
        }
```

```
        if (memo[n] > 0) {
            return memo[n];
        }
        // 状态转移方程：
        // 爬到第 n 级台阶的方法个数等于爬到 n - 1 的方法个数和爬到 n - 2 的方法个数
之和。
        memo[n] = dp(n - 1) + dp(n - 2);
        return memo[n];
    }
}
```

▶ 代码可视化动画（需安装 Chrome 插件）

- **类似题目**：
  - 剑指 Offer 10- II. 青蛙跳台阶问题 🟢

剑指 Offer 10- II. 青蛙跳台阶问题

这道题和 70. 爬楼梯 相同。

198. 打家劫舍

LeetCode	力扣	难度
198. House Robber	198. 打家劫舍	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划**

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警**。

给定一个代表每个房屋存放金额的非负整数数组，计算你**不触动警报装置的情况下**，一夜之内能够偷窃到的最高金额。

示例 1：

```
输入：[1,2,3,1]
输出：4
解释：偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
     偷窃到的最高金额 = 1 + 3 = 4。
```

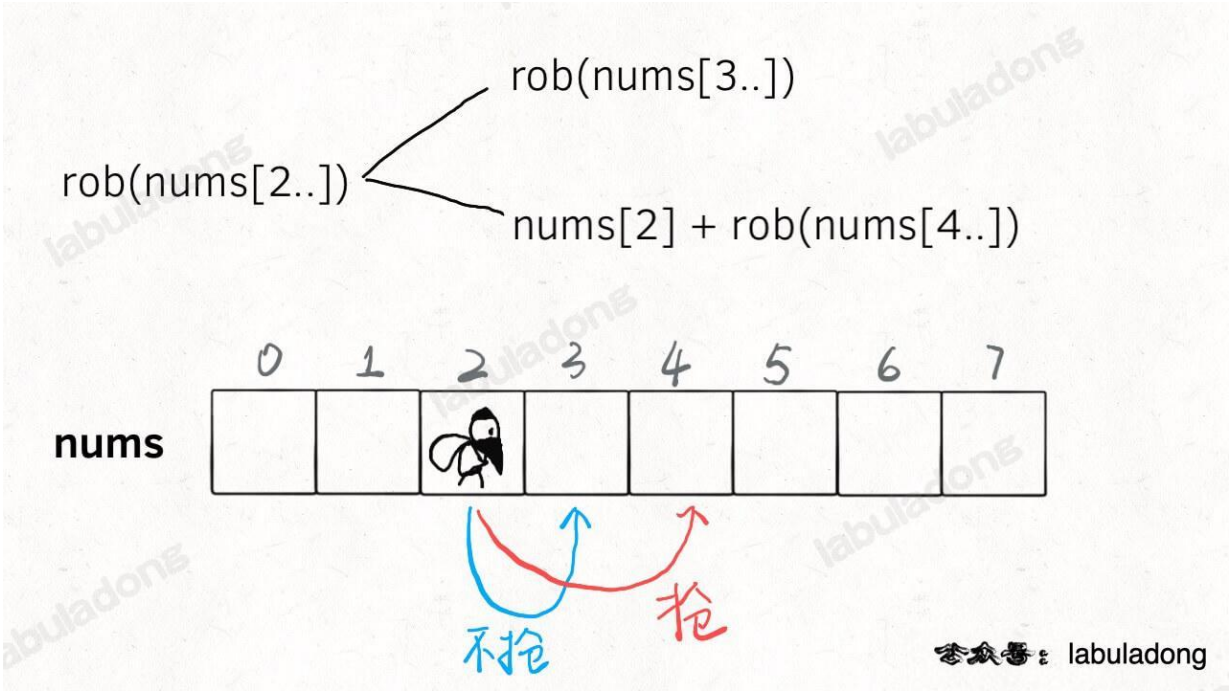
基本思路

PS：这道题在《算法小抄》的第 201 页。

假想你就是这个强盗，从左到右走过这一排房子，在每间房子前都有两种**选择**：抢或者不抢。

当你走过了最后一间房子后，你就没得抢了，能抢到的钱显然是 0（**base case**）。

以上已经明确了「状态」和「选择」：**你面前房子的索引就是状态，抢和不抢就是选择。**



状态转移方程：

```
int res = Math.max(
    // 不抢，去下家
    dp(nums, start + 1),
    // 抢，去下下家
    nums[start] + dp(nums, start + 2)
);
```

打家劫舍系列问题还可以进一步优化，见文章详解，这里只给出最通用的框架性解法。

- **详细题解：一个方法团灭 LeetCode 打家劫舍问题**

解法代码

```
class Solution {
    // 备忘录
    private int[] memo;
    // 主函数
    public int rob(int[] nums) {
        // 初始化备忘录
        memo = new int[nums.length];
        Arrays.fill(memo, -1);
        // 强盗从第 0 间房子开始抢劫
        return dp(nums, 0);
    }

    // 返回 dp[start..] 能抢到的最大值
    private int dp(int[] nums, int start) {
        if (start >= nums.length) {
            return 0;
```

```
        }
        // 避免重复计算
        if (memo[start] != -1) return memo[start];

        int res = Math.max(dp(nums, start + 1),
                nums[start] + dp(nums, start + 2));
        // 记入备忘录
        memo[start] = res;
        return res;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 213. 打家劫舍 II 🟠
  - 337. 打家劫舍 III 🟠
  - 剑指 Offer II 089. 房屋偷盗 🟠
  - 剑指 Offer II 090. 环形房屋偷盗 🟠

213. 打家劫舍 II

LeetCode	力扣	难度
213. House Robber II	213. 打家劫舍 II	🟠

- 标签：一维动态规划，动态规划

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都**围成一圈**，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警**。

给定一个代表每个房屋存放金额的非负整数数组，计算你**在不触动警报装置的情况下**，今晚能够偷窃到的最高金额。

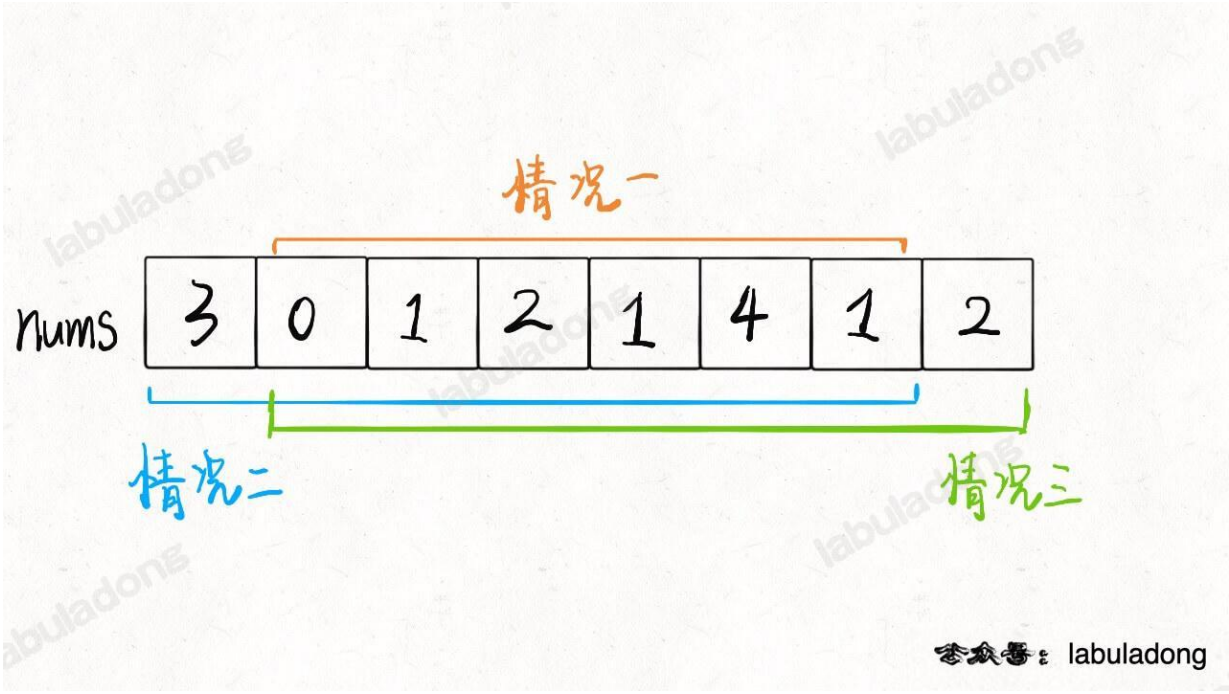
示例 1：

```
输入：nums = [2,3,2]
输出：3
解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。
```

基本思路

PS：这道题在《算法小抄》的第 201 页。

首先，首尾房间不能同时被抢，那么只可能有三种不同情况：要么都不被抢；要么第一间房子被抢最后一间不抢；要么最后一间房子被抢第一间不抢。



这三种情况哪个结果最大，就是最终答案。其实，情况一的结果肯定最小，我们只要比较情况二和情况三就行了，**因为这两种情况对于房子的选择余地比情况一大，房子里的钱数都是非负数，所以选择余地大，最优决策结果肯定不会小**。

把 打家劫舍 I 的解法稍加改造即可。

- **详细题解：一个方法团灭 LeetCode 打家劫舍问题**

解法代码

```
class Solution {

    public int rob(int[] nums) {
        int n = nums.length;
        if (n == 1) return nums[0];

        int[] memo1 = new int[n];
        int[] memo2 = new int[n];
        Arrays.fill(memo1, -1);
        Arrays.fill(memo2, -1);
        // 两次调用使用两个不同的备忘录
        return Math.max(
                dp(nums, 0, n - 2, memo1),
                dp(nums, 1, n - 1, memo2)
        );
    }

    // 定义：计算闭区间 [start,end] 的最优结果
    int dp(int[] nums, int start, int end, int[] memo) {
        if (start > end) {
            return 0;
        }
```

```
        if (memo[start] != -1) {
            return memo[start];
        }
        // 状态转移方程
        int res = Math.max(
                dp(nums, start + 2, end, memo) + nums[start],
                dp(nums, start + 1, end, memo)
        );

        memo[start] = res;
        return res;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 198. 打家劫舍 🟠
  - 337. 打家劫舍 III 🟠
  - 剑指 Offer II 089. 房屋偷盗 🟠
  - 剑指 Offer II 090. 环形房屋偷盗 🟠

337. 打家劫舍 III

LeetCode	力扣	难度
337. House Robber III	337. 打家劫舍 III	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划**

198. 打家劫舍（简单） 和 213. 打家劫舍 II（中等） 的扩展。这次房屋是放在二叉树上，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.
```

基本思路

PS：这道题在《算法小抄》 的第 201 页。

这题 打家劫舍 I 和 打家劫舍 II 的思路完全一样，稍微改写一下就出来了。

- **详细题解：一个方法团灭 LeetCode 打家劫舍问题**

解法代码

```
class Solution {
    Map<TreeNode, Integer> memo = new HashMap<>();

    public int rob(TreeNode root) {
        if (root == null) return 0;
        // 利用备忘录消除重叠子问题
        if (memo.containsKey(root))
            return memo.get(root);
        // 抢，然后去下下家
        int do_it = root.val
                + (root.left == null ?
```

```
                0 : rob(root.left.left) + rob(root.left.right))
                + (root.right == null ?
                0 : rob(root.right.left) + rob(root.right.right));
        // 不抢，然后去下家
        int not_do = rob(root.left) + rob(root.right);

        int res = Math.max(do_it, not_do);
        memo.put(root, res);
        return res;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 198. 打家劫舍 🟠
  - 213. 打家劫舍 II 🟠
  - 剑指 Offer II 089. 房屋偷盗 🟠
  - 剑指 Offer II 090. 环形房屋偷盗 🟠

剑指 Offer II 089. 房屋偷盗

这道题和 198. 打家劫舍 相同。

剑指 Offer II 090. 环形房屋偷盗

这道题和 213. 打家劫舍 II 相同。

300. 最长递增子序列

LeetCode	力扣	难度
300. Longest Increasing Subsequence	300. 最长递增子序列	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划，子序列**

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，例如 `[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

示例 1：

```
输入：nums = [10,9,2,5,3,7,101,18]
输出：4
解释：最长递增子序列是 [2,3,7,101]，因此长度为 4。
```

基本思路

PS：这道题在《算法小抄》 的第 96 页。

`dp` 数组的定义：`dp[i]` 表示以 `nums[i]` 这个数结尾的最长递增子序列的长度。

那么 `dp` 数组中最大的那个值就是最长的递增子序列长度。

- **详细题解：动态规划设计：最长递增子序列**

解法代码

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        // dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度
        int[] dp = new int[nums.length];
        // base case：dp 数组全都初始化为 1
        Arrays.fill(dp, 1);

        for (int i = 0; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j])
                    dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }

        int res = 0;
        for (int i = 0; i < dp.length; i++) {
```

```
                res = Math.max(res, dp[i]);
            }
            return res;
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 354. 俄罗斯套娃信封问题 🔴
  - 53. 最大子数组和 🟠
  - 剑指 Offer 42. 连续子数组的最大和 🟢

354. 俄罗斯套娃信封问题

LeetCode	力扣	难度
354. Russian Doll Envelopes	354. 俄罗斯套娃信封问题	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：一维动态规划，二分搜索，动态规划

给你一个二维整数数组 `envelopes`，其中 `envelopes[i] = [wi, hi]`，表示第 `i` 个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算 **最多能有多少个**信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

```
输入：envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出：3
解释：最多信封的个数为 3，组合为：[2,3] => [5,4] => [6,7]。
```

基本思路

PS：这道题在《算法小抄》的第 104 页。

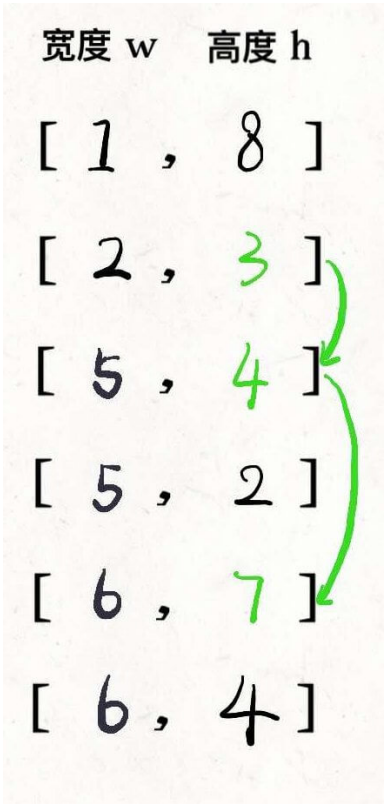
300. 最长递增子序列 在一维数组里面求元素的最长递增子序列，本题相当于在二维平面里面求最长递增子序列。

假设信封是由 `(w, h)` 这样的二维数对形式表示的，思路如下：

先对宽度 `w` 进行升序排序，如果遇到 `w` 相同的情况，则按照高度 `h` 降序排序。之后把所有的 `h` 作为一个数组，在这个数组上计算 LIS 的长度就是答案。

画个图理解一下，先对这些数对进行排序：

然后在 h 上寻找最长递增子序列：



- **详细题解：动态规划设计：最长递增子序列**

解法代码

```
class Solution {
    public int maxEnvelopes(int[][] envelopes) {
```

```
        int n = envelopes.length;
        // 按宽度升序排列，如果宽度一样，则按高度降序排列
        Arrays.sort(envelopes, new Comparator<int[]>()
        {
            public int compare(int[] a, int[] b) {
                return a[0] == b[0] ?
                    b[1] - a[1] : a[0] - b[0];
            }
        });
        // 对高度数组寻找 LIS
        int[] height = new int[n];
        for (int i = 0; i < n; i++)
            height[i] = envelopes[i][1];

        return lengthOfLIS(height);
    }

    /* 返回 nums 中 LIS 的长度 */
    public int lengthOfLIS(int[] nums) {
        int piles = 0, n = nums.length;
        int[] top = new int[n];
        for (int i = 0; i < n; i++) {
            // 要处理的扑克牌
            int poker = nums[i];
            int left = 0, right = piles;
            // 二分查找插入位置
            while (left < right) {
                int mid = (left + right) / 2;
                if (top[mid] >= poker)
                    right = mid;
                else
                    left = mid + 1;
            }
            if (left == piles) piles++;
            // 把这张牌放到牌堆顶
            top[left] = poker;
        }
        // 牌堆数就是 LIS 长度
        return piles;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 300. 最长递增子序列 🟠

53. 最大子序和

LeetCode	力扣	难度
53. Maximum Subarray	53. 最大子数组和	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：一维动态规划，动态规划，数组

给定一个整数数组 nums，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例 1：

```
输入：nums = [-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。
```

基本思路

PS：这道题在《算法小抄》 的第 108 页。

这题类似 300. 最长递增子序列，dp 数组的含义：

以 nums[i] 为结尾的「最大子数组和」为 dp[i]。

dp[i] 有两种「选择」，要么与前面的相邻子数组连接，形成一个和更大的子数组；要么不与前面的子数组连接，自成一派，自己作为一个子数组。

在这两种选择中择优，就可以计算出最大子数组，而且空间复杂度还有优化空间，见详细题解。

- 详细题解：动态规划设计：最大子数组

解法代码

```
class Solution {
    public int maxSubArray(int[] nums) {
        int n = nums.length;
        if (n == 0) return 0;
        int[] dp = new int[n];
        // base case
        // 第一个元素前面没有子数组
        dp[0] = nums[0];
        // 状态转移方程
        for (int i = 1; i < n; i++) {
            dp[i] = Math.max(nums[i], nums[i] + dp[i - 1]);
        }
        // 得到 nums 的最大子数组
```

```
            int res = Integer.MIN_VALUE;
            for (int i = 0; i < n; i++) {
                res = Math.max(res, dp[i]);
            }
            return res;
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 152. 乘积最大子数组 🟠
  - 209. 长度最小的子数组 🟠
  - 918. 环形子数组的最大和 🟠
  - 剑指 Offer 42. 连续子数组的最大和 🟢
  - 剑指 Offer II 008. 和大于等于 target 的最短子数组 🟠

剑指 Offer 42. 连续子数组的最大和

这道题和 53. 最大子序和 相同。

152. 乘积最大子数组

LeetCode	力扣	难度
152. Maximum Product Subarray	152. 乘积最大子数组	🟠

- **标签：dpvip，动态规划**

给你一个整数数组 `nums`，请你找出数组中乘积最大的非空连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。**子数组** 是数组的连续子序列。

示例 1:

```
输入: nums = [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
```

基本思路

这道题和 53. 最大子数组和 有点像，那道题定义的 `dp` 数组是：`dp[i]` 记录以 `nums[i]` 为结尾的「最大子数组和」，从而写出状态转移方程。

这道题可以采用类似的思路，但需要注意的是，在 53 题中，**子数组 `nums[0..i]` 的最大元素和是由 `nums[0..i-1]` 的最大元素和推导出的**，但本题变成子数组的乘积则不一定。

比如 `nums[i] = -1`，`nums[0..i-1]` 子数组的**最大**元素乘积为 10，那么我能不能说 `nums[0..i]` 的最大元素乘积为 `max(-1, -1 * 10) = -1` 呢？

其实不行，因为可能`nums[0..i-1]` 子数组的**最小**元素乘积为 -6，那么 `nums[0..i]` 的最大元素乘积应该为 max(-1, -1 * 10, -1 * -6) = 6`。

所以这道题和 53 题的最大区别在于，要同时维护「以 `nums[i]` 结尾的最大子数组」和「以 `nums[i]` 结尾的最小子数组」，以便适配 `nums[i]` 可能为负的情况。

解法代码

```
class Solution {
    public int maxProduct(int[] nums) {
        int n = nums.length;

        // 定义：以 nums[i] 结尾的子数组，乘积最小为 dp1[i]
        int[] dp1 = new int[n];
        // 定义：以 nums[i] 结尾的子数组，乘积最大为 dp2[i]
        int[] dp2 = new int[n];
```

```
        // base case
        dp1[0] = nums[0];
        dp2[0] = nums[0];

        // 状态转移方程
        for (int i = 1; i < n; i++) {
            dp1[i] = min(dp1[i - 1] * nums[i], dp2[i - 1] * nums[i],
nums[i]);
            dp2[i] = max(dp1[i - 1] * nums[i], dp2[i - 1] * nums[i],
nums[i]);
        }

        // 遍历所有子数组的最大乘积，求最大值
        int res = Integer.MIN_VALUE;
        for (int i = 0; i < n; i++) {
            res = Math.max(res, dp2[i]);
        }

        return res;
    }

    int min(int a, int b, int c) {
        return Math.min(Math.min(a, b), c);
    }

    int max(int a, int b, int c) {
        return Math.max(Math.max(a, b), c);
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

209. 长度最小的子数组

LeetCode	力扣	难度
209. Minimum Size Subarray Sum	209. 长度最小的子数组	🟠

- **标签：dsvip，滑动窗口**

给定一个含有 `n` 个正整数的数组和一个正整数 `target`。找出该数组中满足其和 `≥ target` 的长度最小的**连续子数组**并返回其长度。如果不存在符合条件的子数组，返回 `0`。

示例 1：

```
输入：target = 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。
```

基本思路

这题是标准的滑动窗口算法，你只要看过前文 滑动窗口算法框架 就能轻松搞定。

不过需要强调的是，**题目说了 `nums` 数组中的元素都是正数，有了这个前提才能使用滑动窗口算法**，因为窗口扩大时窗口内元素之和必然增大，窗口缩小时窗口内元素之和必然减小。

如果 `nums` 数组中包含负数，则窗口扩大时元素和不见得就增大，窗口缩小时元素和不见得就减小，这种情况就不能单纯使用滑动窗口技巧了，可能需要混合动态规划和单调队列来做。

比如你可以去试试 1425. 带限制的子序列和，862. 和至少为 K 的最短子数组，53. 最大子序和。

解法代码

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int left = 0, right = 0;
        // 维护窗口内元素之和
        int windowSum = 0;
        int res = Integer.MAX_VALUE;

        while (right < nums.length) {
            // 扩大窗口
            windowSum += nums[right];
            right++;
            while (windowSum >= target && left < right) {
                // 已经达到 target，缩小窗口，同时更新答案
                res = Math.min(res, right - left);
```

```
                windowSum -= nums[left];
                left++;
            }
        }
        return res == Integer.MAX_VALUE ? 0 : res;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 剑指 Offer II 008. 和大于等于 target 的最短子数组 🟠

918. 环形子数组的最大和

LeetCode	力扣	难度
918. Maximum Sum Circular Subarray	918. 环形子数组的最大和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：dsvip，单调队列，数据结构，环形数组**

给定一个长度为 n 的**环形整数数组** nums，返回 nums 的非空**子数组**的最大和。

环形数组意味着数组的末端将会与开头相连呈环状。形式上，nums[i] 的下一个元素是 nums[(i + 1) % n]，nums[i] 的前一个元素是 nums[(i - 1 + n) % n]。

示例 1：

```
输入：nums = [1,-2,3,-2]
输出：3
解释：从子数组 [3] 得到最大和 3
```

基本思路

诚然，这道题有很巧妙的方法，你可以搜索一下「Kadane 算法」来解决这道题。不过为了举一反三地运用我之前讲过的算法技巧，我就结合 单调队列结构 和前文 前缀和技巧 写一个更通用的解法。

首先，这道题和 53. 最大子序和 非常类似，区别在于本题的数组是环形的，所以上一题的动态规划思路和前缀和思路都不能直接套用过来。不过前文 单调栈结构详解 中讲过处理环形数组的方法，其实就是把原数组大小扩大一倍，这样就能模拟出环形的效果了。

那么本题也可以把 nums 数组扩大一倍，计算前缀和数组 preSum，借助一个定长为 nums.length 的单调队列来计算环形数组中的最大子数组和。具体实现直接看代码吧。

> PS：MonotonicQueue 的通用实现见 单调队列设计与实现

解法代码

```
class Solution {
    public int maxSubarraySumCircular(int[] nums) {
        int n = nums.length;
        // 模拟环状的 nums 数组
        int[] preSum = new int[2 * n + 1];
        preSum[0] = 0;
        // 计算环状 nums 的前缀和
        for (int i = 1; i < preSum.length; i++) {
            preSum[i] = preSum[i - 1] + nums[(i - 1) % n];
        }
```

```
        // 记录答案
        int maxSum = Integer.MIN_VALUE;
        // 维护一个滑动窗口，以便根据窗口中的最小值计算最大子数组和
        MonotonicQueue<Integer> window = new MonotonicQueue<>();
        window.push(0);
        for (int i = 1; i < preSum.length; i++) {
            maxSum = Math.max(maxSum, preSum[i] - window.min());
            // 维护窗口的大小为 nums 数组的大小
            if (window.size() == n) {
                window.pop();
            }
            window.push(preSum[i]);
        }

        return maxSum;
    }
}

/*
单调队列的通用实现，可以高效维护最大值和最小值
由于我这里考虑了泛型和通用性，所以提交的性能会略差，你可自行精简
*/
class MonotonicQueue<E extends Comparable<E>> {}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

剑指 Offer II 008. 和大于等于 target 的最短子数组

这道题和 209. 长度最小的子数组 相同。

279. 完全平方数

LeetCode	力扣	难度
279. Perfect Squares	279. 完全平方数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：动态规划，数学

给你一个整数 n，返回**和为 n 的完全平方数的最少数量**。**完全平方数**是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

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

基本思路

这道题有一些比较有技巧性的数学方法，我这里不探讨数学，就用通用的动态规划思路解决：

题目问和为 n 的平方数的最小数量，那么我可以根据和为 n-1x1, n-2x2, n-3x3... 的平方数的最小数量推导出来。

如果这个思路你绕不过来，我再给你变化下就更容易理解了，这个问题完全可以变化成 动态规划详解 中讲的 322. 零钱兑换 问题：

给你一个目标金额 n，和一个若干硬币的面额 coins = 1,4,9,16...，问你最少需要几枚硬币凑出这个金额（因为包含面值为 1 的硬币，所以不存在凑不出来的情况）。

这样变个说法理解题目，是不是就豁然开朗了？具体解法看代码吧，复杂度是 O(N*sqrt(N))，也是不错的。

解法代码

```
class Solution {
    public int numSquares(int n) {
        // 定义：和为 i 的平方数的最小数量是 dp[i]
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        // base case
        dp[0] = 0;
        // 状态转移方程
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j * j <= i; j++) {
```

```
                // i - j * j 只要再加一个平方数 j * j 即可凑出 i
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }
        return dp[n];
    }
}
```

▶ 😋 代码可视化动画（需安装 Chrome 插件）😋

322. 零钱兑换

LeetCode	力扣	难度
322. Coin Change	322. 零钱兑换	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划，最短路径算法**

给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。

计算并返回可以凑成总金额所需的**最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 `-1`（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
```

基本思路

> 本文有视频版：动态规划框架套路详解

PS：这道题在《算法小抄》 的第 31 页。

1、**确定 base case**，显然目标金额 `amount` 为 0 时算法返回 0，因为不需要任何硬币就已经凑出目标金额了。

2、**确定「状态」，也就是原问题和子问题中会变化的变量**。由于硬币数量无限，硬币的面额也是题目给定的，只有目标金额会不断地向 base case 靠近，所以唯一的「状态」就是目标金额 `amount`。

3、**确定「选择」，也就是导致「状态」产生变化的行为**。目标金额为什么变化呢，因为你在选择硬币，你每选择一枚硬币，就相当于减少了目标金额。所以说所有硬币的面值，就是你的「选择」。

4、**明确 `dp` 函数/数组的定义**：输入一个目标金额 `n`，返回凑出目标金额 `n` 的最少硬币数量。

按照 `dp` 函数的定义描述「选择」，得到最终答案 `dp(amount)`。

- **详细题解：动态规划解题套路框架**

解法代码

```
class Solution {
    int[] memo;

    public int coinChange(int[] coins, int amount) {
```

```
        memo = new int[amount + 1];
        // dp 数组全都初始化为特殊值
        Arrays.fill(memo, -666);
        return dp(coins, amount);
    }

    int dp(int[] coins, int amount) {
        if (amount == 0) return 0;
        if (amount < 0) return -1;
        // 查备忘录，防止重复计算
        if (memo[amount] != -666)
            return memo[amount];

        int res = Integer.MAX_VALUE;
        for (int coin : coins) {
            // 计算子问题的结果
            int subProblem = dp(coins, amount - coin);
            // 子问题无解则跳过
            if (subProblem == -1) continue;
            // 在子问题中选择最优解，然后加一
            res = Math.min(res, subProblem + 1);
        }
        // 把计算结果存入备忘录
        memo[amount] = (res == Integer.MAX_VALUE) ? -1 : res;
        return memo[amount];
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 279. 完全平方数 🟠
  - 509. 斐波那契数 🟢
  - 剑指 Offer II 103. 最少的硬币数目 🟠

剑指 Offer II 103. 最少的硬币数目

这道题和 322. 零钱兑换 相同。

10. 正则表达式匹配

LeetCode	力扣	难度
10. Regular Expression Matching	10. 正则表达式匹配	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，字符串**

给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配，`'.'` 匹配任意单个字符 `'*'` 匹配零个或多个前面的那一个元素。

算法返回 `p` 是否可以匹配**整个**字符串 `s`。

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。
```

基本思路

PS：这道题在《算法小抄》 的第 155 页。

`s` 和 `p` 相互匹配的过程大致是，两个指针 `i, j` 分别在 `s` 和 `p` 上移动，如果最后两个指针都能移动到字符串的末尾，那么就匹配成功，反之则匹配失败。

正则表达算法问题只需要把住一个基本点：看 `s[i]` 和 `p[j]` 两个字符是否匹配，一切逻辑围绕匹配/不匹配两种情况展开即可。

动态规划算法的核心就是「状态」和「选择」，**「状态」无非就是 `i` 和 `j` 两个指针的位置，「选择」就是模式串的 `p[j]` 选择匹配几个字符。**

`dp` 函数的定义如下：

若 `dp(s, i, p, j) = true`，则表示 `s[i..]` 可以匹配 `p[j..]`；若 `dp(s, i, p, j) = false`，则表示 `s[i..]` 无法匹配 `p[j..]`。

- **详细题解：经典动态规划：正则表达式**

解法代码

```
class Solution {
    public:
    // 备忘录
    vector<vector<int>> memo;
```

```
    bool isMatch(string s, string p) {
        int m = s.size(), n = p.size();
        memo = vector<vector<int>>(m, vector<int>(n, -1));
        // 指针 i, j 从索引 0 开始移动
        return dp(s, 0, p, 0);
    }

    /* 计算 p[j..] 是否匹配 s[i..] */
    bool dp(string& s, int i, string& p, int j) {
        int m = s.size(), n = p.size();
        // base case
        if (j == n) {
            return i == m;
        }
        if (i == m) {
            if ((n - j) % 2 == 1) {
                return false;
            }
            for (; j + 1 < n; j += 2) {
                if (p[j + 1] != '*') {
                    return false;
                }
            }
            return true;
        }

        // 查备忘录，防止重复计算
        if (memo[i][j] != -1) {
            return memo[i][j];
        }

        bool res = false;

        if (s[i] == p[j] || p[j] == '.') {
            if (j < n - 1 && p[j + 1] == '*') {
                res = dp(s, i, p, j + 2)
                        || dp(s, i + 1, p, j);
            } else {
                res = dp(s, i + 1, p, j + 1);
            }
        } else {
            if (j < n - 1 && p[j + 1] == '*') {
                res = dp(s, i, p, j + 2);
            } else {
                res = false;
            }
        }
        // 将当前结果记入备忘录
        memo[i][j] = res;
        return res;
    }
};
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- 类似题目：
  - 44. 通配符匹配 🔴
  - 剑指 Offer 19. 正则表达式匹配 🔴

44. 通配符匹配

LeetCode	力扣	难度
44. Wildcard Matching	44. 通配符匹配	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：动态规划

给定一个字符串 (s) 和一个模式串 (p)，实现一个支持 '?' 和 '*' 的通配符匹配。

```
'?' 可以匹配任何单个字符。
'*' 可以匹配任意字符串（包括空字符串）。
```

两个字符串**完全匹配**才算匹配成功。

基本思路

这道题和 10. 正则表达式匹配 几乎一样，这里的 ? 通配符就是第 10 题的 .，这里的 * 就是第 10 题的 .* 组合，所以一个投机取巧的解法就是把本题中的通配符转化成第 10 题的形式，然后直接套用第 10 题的解法。

如果不投机取巧，照着第 10 题的解法逻辑也可以写出本题的解法，唯一需要注意的是本题的测试用例可能出现很多 * 连续出现的情况，很容易看出**连续多个 * 和一个 * 的通配效果是一样的**，所以我们可以提前删除连续的 * 以便提升一些效率。

解法代码

```
class Solution {
    public:
    // 备忘录，-1 代表还未计算，0 代表 false，1 代表 true
    vector<vector<int>> memo;

    bool isMatch(string s, string p) {
        if (p.empty()) {
            return s.empty();
        }
        // 将 p 中相邻的 * 去除，以提升效率
        string pp = remove_adj_star(p);
        int m = s.size(), n = pp.size();
        // 备忘录初始化为 -1
        memo = vector<vector<int>>(m, vector<int>(n, -1));
        // 执行自顶向下带备忘录的动态规划
        return dp(s, 0, pp, 0);
    }

    // 删除相邻的 * 号，返回删除后的字符
```

```
    string remove_adj_star(string p) {
        if (p.empty()) {
            return "";
        }
        string pp;
        pp.push_back(p[0]);
        for (int i = 1; i < p.size(); i++) {
            if (p[i] == '*' && p[i - 1] == '*') {
                continue;
            }
            pp.push_back(p[i]);
        }
        return pp;
    }

    // 定义：判断 s[i..] 是否能被 p[j..] 匹配
    bool dp(string& s, int i, string& p, int j) {
        // base case
        if (j == p.size() && i == s.size()) {
            return true;
        }
        if (i == s.size()) {
            for (int k = j; k < p.size(); k++) {
                if (p[k] != '*') {
                    return false;
                }
            }
            return true;
        }
        if (j == p.size()) {
            return false;
        }
        if (memo[i][j] != -1) {
            return bool(memo[i][j]);
        }

        bool res = false;
        if (s[i] == p[j] || p[j] == '?') {
            // s[i] 和 p[j] 完成匹配
            res = dp(s, i + 1, p, j + 1);
        } else if (p[j] == '*') {
            // s[i] 和 p[j] 不匹配，但 p[j] 是通配符 *
            // 可以匹配 0 个或多个 s 中的字符，
            // 只要有一种情况能够完成匹配即可
            res = dp(s, i + 1, p, j)
                    || dp(s, i, p, j + 1);
        }
        // 将 s[i] 和 p[j] 的匹配结果存储备忘录
        memo[i][j] = res;

        return res;
    }
};
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

剑指 Offer 19. 正则表达式匹配

这道题和 10. 正则表达式匹配 相同。

62. 不同路径

LeetCode	力扣	难度
62. Unique Paths	62. 不同路径	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，二维矩阵，动态规划

一个机器人位于一个 `m x n` 网格的左上角（起始点在下图中标记为 Start）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径?

示例 1：

```
输入：m = 3, n = 7
输出：28
```

示例 2：

```
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 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下
```

基本思路

如果你看过前文 动态规划框架详解，就知道这道题是非常基本的动态规划问题。

对 `dp` 数组的定义和状态转移方程如下：

```
public int uniquePaths(int m, int n) {
    return dp(m - 1, n - 1);
}

// 定义：从 (0, 0) 到 (x, y) 有 dp(x, y) 条路径
int dp(int x, int y) {
    if (x == 0 && y == 0) {
        return 1;
    }
    if (x < 0 || y < 0) {
        return 0;
    }
    // 状态转移方程：
    // 到达 (x, y) 的路径数等于到达 (x - 1, y) 和 (x, y - 1) 路径数之和
    return dp(x - 1, y) + dp(x, y - 1);
}
```

添加备忘录或者改写为自底向上的迭代解法即可降低上述暴力解法的时间复杂度。

解法代码

```
class Solution {
    // 备忘录
    int[][] memo;

    public int uniquePaths(int m, int n) {
        memo = new int[m][n];
        return dp(m - 1, n - 1);
    }

    // 定义：从 (0, 0) 到 (x, y) 有 dp(x, y) 条路径
    int dp(int x, int y) {
        // base case
        if (x == 0 && y == 0) {
            return 1;
        }
        if (x < 0 || y < 0) {
            return 0;
        }
        // 避免冗余计算
        if (memo[x][y] > 0) {
            return memo[x][y];
        }
        // 状态转移方程：
        // 到达 (x, y) 的路径数等于到达 (x - 1, y) 和 (x, y - 1) 路径数之和
        memo[x][y] = dp(x - 1, y) + dp(x, y - 1);
        return memo[x][y];
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 剑指 Offer II 098. 路径的数目 🟠

剑指 Offer II 098. 路径的数目

这道题和 62. 不同路径 相同。

64. 最小路径和

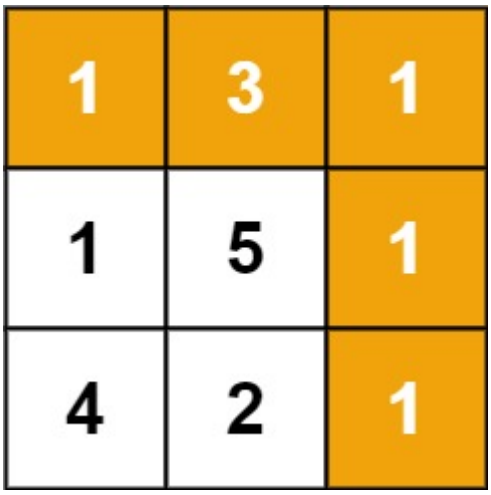
LeetCode	力扣	难度
64. Minimum Path Sum	64. 最小路径和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，二维矩阵，动态规划

给定一个包含非负整数的 `m x n` 网格 `grid`，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小（每次只能向下或者向右移动一步）。

示例 1：




```
输入：grid = [[1,3,1],[1,5,1],[4,2,1]]
输出：7
解释：因为路径 1→3→1→1→1 的总和最小。
```

基本思路

一般来说，让你在二维矩阵中求最优化问题（最大值或者最小值），肯定需要递归 + 备忘录，也就是动态规划技巧。

`dp` 函数的定义：**从左上角位置 `(0, 0)` 走到位置 `(i, j)` 的最小路径和为 `dp(grid, i, j)`**。

这样，`dp(grid, i, j)` 的值由 `dp(grid, i - 1, j)` 和 `dp(grid, i, j - 1)` 的值转移而来：

```
dp(grid, i, j) = Math.min(
    dp(grid, i - 1, j),
    dp(grid, i, j - 1)
) + grid[i][j];
```

- **详细题解：动态规划之最小路径和**

解法代码

```
class Solution {
    int[][] memo;

    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        // 构造备忘录，初始值全部设为 -1
        memo = new int[m][n];
        for (int[] row : memo)
            Arrays.fill(row, -1);

        return dp(grid, m - 1, n - 1);
    }

    int dp(int[][] grid, int i, int j) {
        // base case
        if (i == 0 && j == 0) {
            return grid[0][0];
        }
        if (i < 0 || j < 0) {
            return Integer.MAX_VALUE;
        }
        // 避免重复计算
        if (memo[i][j] != -1) {
            return memo[i][j];
        }
        // 将计算结果记入备忘录
        memo[i][j] = Math.min(
                dp(grid, i - 1, j),
                dp(grid, i, j - 1)
        ) + grid[i][j];

        return memo[i][j];
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目：**
  - 剑指 Offer 47. 礼物的最大价值 🟠
  - 剑指 Offer II 099. 最小路径之和 🟠

剑指 Offer 47. 礼物的最大价值

LeetCode	力扣	难度
剑指Offer47. 礼物的最大价值 LCOF	剑指Offer47. 礼物的最大价值	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：动态规划

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

示例 1:

```
输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 12
解释: 路径 1→3→5→2→1 可以拿到最多价值的礼物
```

基本思路

这题就是前文 最小路径和问题 中讲的 64. 最小路径和（中等），几乎完全一样，你只需要把那道题解法中的 `min` 改成 `max` 即可解决这道题。

解法代码

```
class Solution {
    public int maxValue(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int[][] dp = new int[m][n];

        /**** base case ****/
        dp[0][0] = grid[0][0];
        for (int i = 1; i < m; i++)
            dp[i][0] = dp[i - 1][0] + grid[i][0];

        for (int j = 1; j < n; j++)
            dp[0][j] = dp[0][j - 1] + grid[0][j];

        // 状态转移
```

```
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = Math.max(
                    dp[i - 1][j],
                    dp[i][j - 1]
                ) + grid[i][j];
            }
        }
        return dp[m - 1][n - 1];
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

剑指 Offer II 099. 最小路径之和

这道题和 64. 最小路径和 相同。

1143. 最长公共子序列

LeetCode	力扣	难度
1143. Longest Common Subsequence	1143. 最长公共子序列	🟠

- **标签：二维动态规划，动态规划，子序列**

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。如果不存在**公共子序列**，返回 `0`。

子序列的定义：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。

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。

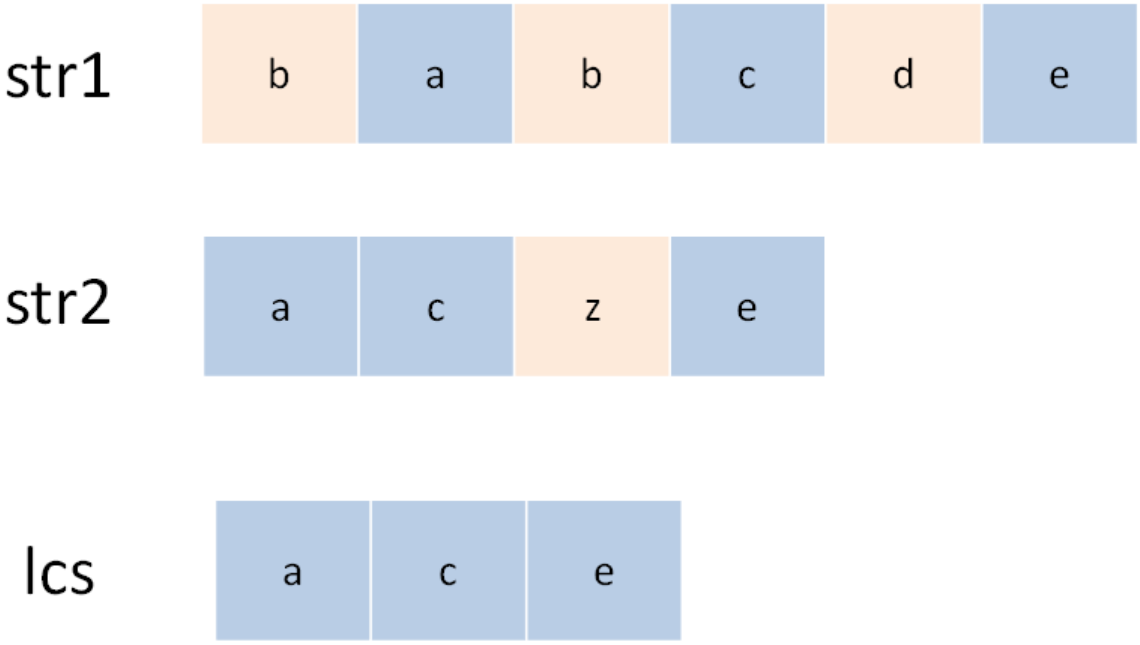
示例 1：

```
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。
```

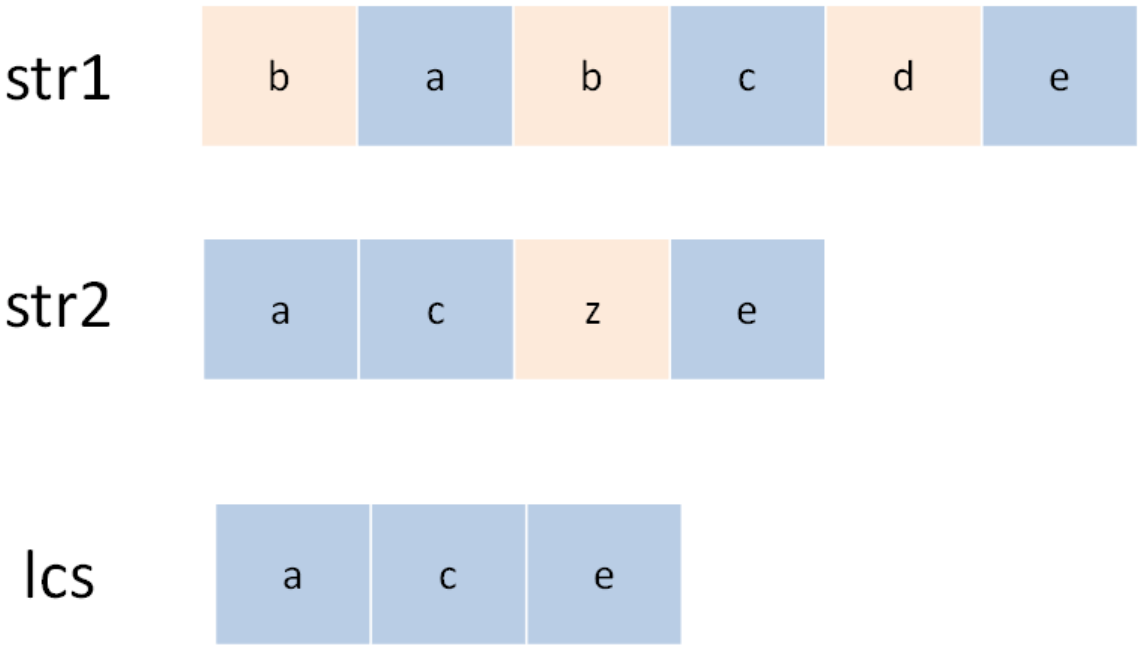
基本思路

PS：这道题在《算法小抄》 的第 117 页。

和 72. 编辑距离 同为经典的双字符串动态规划问题。用两个指针 `i, j` 在两个字符串上游走，这就是「状态」，字符串中的每个字符都有两种「选择」，要么在 `lcs` 中，要么不在。



`dp[i][j]` 的含义是：对于 `s1[1..i]` 和 `s2[1..j]`，它们的 LCS 长度是 `dp[i][j]`。



- **详细题解：经典动态规划：最长公共子序列**

解法代码

```
class Solution {
    public int longestCommonSubsequence(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        // 定义：s1[0..i-1] 和 s2[0..j-1] 的 lcs 长度为 dp[i][j]
        int[][] dp = new int[m + 1][n + 1];
        // 目标：s1[0..m-1] 和 s2[0..n-1] 的 lcs 长度，即 dp[m][n]
        // base case: dp[0][..] = dp[..][0] = 0
```

```
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // 现在 i 和 j 从 1 开始，所以要减一
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    // s1[i-1] 和 s2[j-1] 必然在 lcs 中
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    // s1[i-1] 和 s2[j-1] 至少有一个不在 lcs 中
                    dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]);
                }
            }
        }

        return dp[m][n];
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 583. 两个字符串的删除操作 🟠
  - 712. 两个字符串的最小ASCII删除和 🟠
  - 剑指 Offer II 095. 最长公共子序列 🟠

72. 编辑距离

LeetCode	力扣	难度
72. Edit Distance	72. 编辑距离	🔴

- **标签：二维动态规划，动态规划**

给你两个单词 `word1` 和 `word2`，请你计算出将 `word1` 转换成 `word2` 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

1、插入一个字符

2、删除一个字符

3、替换一个字符

示例 1：

```
输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse（将 'h' 替换为 'r'）
rorse -> rose（删除 'r'）
rose -> ros（删除 'e'）
```

示例 2：

```
输入：word1 = "intention", word2 = "execution"
输出：5
解释：
intention -> inention（删除 't'）
inention -> enention（将 'i' 替换为 'e'）
enention -> exention（将 'n' 替换为 'x'）
exention -> exection（将 'n' 替换为 'c'）
exection -> execution（插入 'u'）
```

基本思路

> 本文有视频版：编辑距离详解动态规划

PS：这道题在《算法小抄》 的第 123 页。

解决两个字符串的动态规划问题，一般都是用两个指针 `i, j` 分别指向两个字符串的最后，然后一步步往前走，缩小问题的规模。

对于每对儿字符 `s1[i]` 和 `s2[j]`，可以有四种操作：

```
if s1[i] == s2[j]:
    啥都别做（skip）
    i, j 同时向前移动
else:
    三选一：
        插入（insert）
        删除（delete）
        替换（replace）
```

那么「状态」就是指针 `i, j` 的位置，「选择」就是上述的四种操作。

如果使用自底向上的迭代解法，这样定义 `dp` 数组：`dp[i-1][j-1]` 存储 `s1[0..i]` 和 `s2[0..j]` 的最小编辑距离。`dp` 数组索引至少是 0，所以索引会偏移一位。

然后把上述四种选择用 `dp` 函数表示出来，就可以得出最后答案了。

- **详细题解：经典动态规划：编辑距离**

解法代码

```
class Solution {
    public int minDistance(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        int[][] dp = new int[m + 1][n + 1];
        // base case
        for (int i = 1; i <= m; i++)
            dp[i][0] = i;
        for (int j = 1; j <= n; j++)
            dp[0][j] = j;
        // 自底向上求解
        for (int i = 1; i <= m; i++)
            for (int j = 1; j <= n; j++)
                if (s1.charAt(i - 1) == s2.charAt(j - 1))
                    dp[i][j] = dp[i - 1][j - 1];
                else
                    dp[i][j] = min(
                        dp[i - 1][j] + 1,
                        dp[i][j - 1] + 1,
                        dp[i - 1][j - 1] + 1
                    );
        // 储存着整个 s1 和 s2 的最小编辑距离
        return dp[m][n];
    }

    int min(int a, int b, int c) {
```

```
            return Math.min(a, Math.min(b, c));
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 1143. 最长公共子序列 🟠
  - 剑指 Offer II 095. 最长公共子序列 🟠

剑指 Offer II 095. 最长公共子序列

这道题和 1143. 最长公共子序列 相同。

121. 买卖股票的最佳时机

LeetCode	力扣	难度
121. Best Time to Buy and Sell Stock	121. 买卖股票的最佳时机	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，动态规划

给定一个数组 `prices`，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择**某一天**买入这只股票，并选择在**未来的某一个不同的日子**卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0`。

示例 1：

```
输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
```

基本思路

提示：股票系列问题有共通性，但难度较大，初次接触此类问题的话很难看懂下述思路，建议直接看 详细题解。

股票系列问题状态定义：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
```

股票系列问题通用状态转移方程：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
              max( 今天选择 rest,         今天选择 sell       )

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
              max( 今天选择 rest,         今天选择 buy         )
```

通用 base case：

```
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity
```

特化到 `k = 1` 的情况，状态转移方程和 base case 如下：

```
状态转移方程：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], -prices[i])

base case：
dp[i][0] = 0;
dp[i][1] = -prices[i];
```

详细思路解析和空间复杂度优化的解法见详细题解。

- **详细题解：一个方法团灭 LeetCode 股票买卖问题**

解法代码

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int[][] dp = new int[n][2];
        for (int i = 0; i < n; i++) {
            if (i - 1 == -1) {
                // base case
                dp[i][0] = 0;
                dp[i][1] = -prices[i];
                continue;
            }
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i - 1][1], -prices[i]);
        }
        return dp[n - 1][0];
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目：**
  - 122. 买卖股票的最佳时机 II 🟠
  - 123. 买卖股票的最佳时机 III 🔴
  - 188. 买卖股票的最佳时机 IV 🔴

- 309. 最佳买卖股票时机含冷冻期 🟠
- 714. 买卖股票的最佳时机含手续费 🟠
- 剑指 Offer 63. 股票的最大利润 🟠

122. 买卖股票的最佳时机 II

LeetCode	力扣	难度
122. Best Time to Buy and Sell Stock II	122. 买卖股票的最佳时机 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划**

给定一个数组 `prices`，其中 `prices[i]` 是一支给定股票第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入：prices = [7,1,5,3,6,4]
输出：7
解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5-1 = 4。
     随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6-3 = 3。
```

基本思路

提示：股票系列问题有共通性，但难度较大，初次接触此类问题的话很难看懂下述思路，建议直接看 详细题解。

股票系列问题状态定义：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
```

股票系列问题通用状态转移方程和 base case：

```
状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

base case：
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity
```

特化到 k 无限制的情况，状态转移方程如下：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
```

详细思路解析和空间复杂度优化的解法见详细题解。

- **详细题解：一个方法团灭 LeetCode 股票买卖问题**

解法代码

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int[][] dp = new int[n][2];
        for (int i = 0; i < n; i++) {
            if (i - 1 == -1) {
                // base case
                dp[i][0] = 0;
                dp[i][1] = -prices[i];
                continue;
            }
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
        }
        return dp[n - 1][0];
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 121. 买卖股票的最佳时机 🟢
  - 123. 买卖股票的最佳时机 III 🔴
  - 188. 买卖股票的最佳时机 IV 🔴
  - 309. 最佳买卖股票时机含冷冻期 🟠
  - 714. 买卖股票的最佳时机含手续费 🟠
  - 剑指 Offer 63. 股票的最大利润 🟠

123. 买卖股票的最佳时机 III

LeetCode	力扣	难度
123. Best Time to Buy and Sell Stock III	123. 买卖股票的最佳时机 III	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：三维动态规划，动态规划**

给定一个数组，它的第 `i` 个元素是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成**两笔**交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入：prices = [3,3,5,0,0,3,1,4]
输出：6
解释：在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3。
     随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天（股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3。
```

基本思路

提示：股票系列问题有共通性，但难度较大，初次接触此类问题的话很难看懂下述思路，建议直接看 详细题解。

股票系列问题状态定义：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
```

股票系列问题通用状态转移方程和 base case：

```
状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

base case：
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity
```

之前的几道股票问题，状态 k 都被化简掉了，这道题无法化简 k 的限制，所以就要加一层 for 循环穷举这个状态。

详细思路解析和空间复杂度优化的解法见详细题解。

- **详细题解：一个方法团灭 LeetCode 股票买卖问题**

解法代码

```
class Solution {
    public int maxProfit(int[] prices) {
        int max_k = 2, n = prices.length;
        int[][][] dp = new int[n][max_k + 1][2];
        for (int i = 0; i < n; i++) {
            for (int k = max_k; k >= 1; k--) {
                if (i - 1 == -1) {
                    // 处理 base case
                    dp[i][k][0] = 0;
                    dp[i][k][1] = -prices[i];
                    continue;
                }
                dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] +
prices[i]);
                dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] -
prices[i]);
            }
        }
        // 穷举了 n × max_k × 2 个状态，正确。
        return dp[n - 1][max_k][0];
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 121. 买卖股票的最佳时机 🟢
  - 122. 买卖股票的最佳时机 II 🟠
  - 188. 买卖股票的最佳时机 IV 🔴
  - 309. 最佳买卖股票时机含冷冻期 🟠
  - 714. 买卖股票的最佳时机含手续费 🟠
  - 剑指 Offer 63. 股票的最大利润 🟠

188. 买卖股票的最佳时机 IV

LeetCode	力扣	难度
188. Best Time to Buy and Sell Stock IV	188. 买卖股票的最佳时机 IV	🔴

- 标签：三维动态规划，动态规划

给定一个整数数组 `prices`，它的第 `i` 个元素 `prices[i]` 是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 `k` 笔交易。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

```
输入：k = 2, prices = [2,4,1]
输出：2
解释：在第 1 天（股票价格 = 2）的时候买入，在第 2 天（股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-2 = 2。
```

基本思路

提示：股票系列问题有共通性，但难度较大，初次接触此类问题的话很难看懂下述思路，建议直接看 详细题解。

股票系列问题状态定义：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
```

股票系列问题通用状态转移方程和 base case：

```
状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

base case：
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity
```

这题算是股票问题的终极形态，理论上把上面的状态转移方程实现就行了，但一个关键点在于限制 k 的大小，否则会出现内存超限的错误。

一次交易由买入和卖出构成，至少需要两天，所以说有效的限制 k 应该不超过 n/2，如果超过，就没有约束作用了，相当于 k = +infinity，这是 122. 买卖股票的最佳时机 II 解决过的。

详细思路解析和空间复杂度优化的解法见详细题解。

- **详细题解：一个方法团灭 LeetCode 股票买卖问题**

解法代码

```
class Solution {
    public int maxProfit(int max_k, int[] prices) {
        int n = prices.length;
        if (n <= 0) {
            return 0;
        }
        if (max_k > n / 2) {
            // 交易次数 k 没有限制的情况
            return maxProfit_k_inf(prices);
        }

        // base case:
        // dp[-1][...][0] = dp[...][0][0] = 0
        // dp[-1][...][1] = dp[...][0][1] = -infinity
        int[][][] dp = new int[n][max_k + 1][2];
        // k = 0 时的 base case
        for (int i = 0; i < n; i++) {
            dp[i][0][1] = Integer.MIN_VALUE;
            dp[i][0][0] = 0;
        }

        for (int i = 0; i < n; i++)
            for (int k = max_k; k >= 1; k--) {
                if (i - 1 == -1) {
                    // 处理 i = -1 时的 base case
                    dp[i][k][0] = 0;
                    dp[i][k][1] = -prices[i];
                    continue;
                }
                // 状态转移方程
                dp[i][k][0] = Math.max(dp[i - 1][k][0], dp[i - 1][k][1] +
prices[i]);
                dp[i][k][1] = Math.max(dp[i - 1][k][1], dp[i - 1][k - 1]
[0] - prices[i]);
            }
        return dp[n - 1][max_k][0];
    }

    // 第 122 题，k 无限的解法
    private int maxProfit_k_inf(int[] prices) {
```

```
        int n = prices.length;
        int[][] dp = new int[n][2];
        for (int i = 0; i < n; i++) {
            if (i - 1 == -1) {
                // base case
                dp[i][0] = 0;
                dp[i][1] = -prices[i];
                continue;
            }
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);
        }
        return dp[n - 1][0];
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 121. 买卖股票的最佳时机 🟢
  - 122. 买卖股票的最佳时机 II 🟠
  - 123. 买卖股票的最佳时机 III 🔴
  - 309. 最佳买卖股票时机含冷冻期 🟠
  - 714. 买卖股票的最佳时机含手续费 🟠
  - 剑指 Offer 63. 股票的最大利润 🟠

309. 最佳买卖股票时机含冷冻期

LeetCode	力扣	难度
309. Best Time to Buy and Sell Stock with Cooldown	309. 最佳买卖股票时机含冷冻期	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，动态规划

给定一个整数数组，其中第 `i` 个元素代表了第 `i` 天的股票价格。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

1、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）。

2、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]
```

基本思路

提示：股票系列问题有共通性，但难度较大，初次接触此类问题的话很难看懂下述思路，建议直接看 详细题解。

股票系列问题状态定义：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
```

股票系列问题通用状态转移方程和 base case：

```
状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

base case：
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity
```

特化到 k 无限制且包含手续费的情况，只需稍微修改 122. 买卖股票的最佳时机 II，每次 sell 之后要等一天才能继续交易。

只要把这个特点融入上一题的状态转移方程即可：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1。
```

当然，由于 i - 2 也可能小于 0，所以再添加一个 i - 2 < 0 的 base case，根据状态转移方程推出 base case 的具体逻辑。

详细思路解析和空间复杂度优化的解法见详细题解。

- **详细题解：一个方法团灭 LeetCode 股票买卖问题**

解法代码

```
class Solution {
    public int maxProfit(int[] prices) {
        int n = prices.length;
        int[][] dp = new int[n][2];
        for (int i = 0; i < n; i++) {
            if (i - 1 == -1) {
                // base case 1
                dp[i][0] = 0;
                dp[i][1] = -prices[i];
                continue;
            }
            if (i - 2 == -1) {
                // base case 2
                dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] +
prices[i]);
                // i - 2 小于 0 时根据状态转移方程推出对应 base case
                dp[i][1] = Math.max(dp[i - 1][1], -prices[i]);
                //   dp[i][1]
                // = max(dp[i-1][1], dp[-1][0] - prices[i])
                // = max(dp[i-1][1], 0 - prices[i])
                // = max(dp[i-1][1], -prices[i])
                continue;
            }
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i - 1][1], dp[i - 2][0] - prices[i]);
        }
        return dp[n - 1][0];
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 121. 买卖股票的最佳时机 🟢
  - 122. 买卖股票的最佳时机 II 🟠
  - 123. 买卖股票的最佳时机 III 🔴
  - 188. 买卖股票的最佳时机 IV 🔴
  - 714. 买卖股票的最佳时机含手续费 🟠
  - 剑指 Offer 63. 股票的最大利润 🟠

714. 买卖股票的最佳时机含手续费

LeetCode	力扣	难度
714. Best Time to Buy and Sell Stock with Transaction Fee	714. 买卖股票的最佳时机含手续费	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：三维动态规划，动态规划

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；整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

请你计算获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1：

```
输入：prices = [1, 3, 2, 8, 4, 9], fee = 2
输出：8
解释：能够达到的最大利润：
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润：((8 - 1) - 2) + ((9 - 4) - 2) = 8
```

基本思路

提示：股票系列问题有共通性，但难度较大，初次接触此类问题的话很难看懂下述思路，建议直接看 详细题解。

股票系列问题状态定义：

```
dp[i][k][0 or 1]
0 <= i <= n - 1, 1 <= k <= K
n 为天数，大 K 为交易数的上限，0 和 1 代表是否持有股票。
```

股票系列问题通用状态转移方程和 base case：

```
状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

base case：
dp[-1][...][0] = dp[...][0][0] = 0
dp[-1][...][1] = dp[...][0][1] = -infinity
```

特化到 k 无限制且包含手续费的情况，只需稍微修改 122. 买卖股票的最佳时机 II，手续费可以认为是买入价变贵了或者卖出价变便宜了。

状态转移方程如下：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)
解释：相当于买入股票的价格升高了。
```

注意状态转移方程改变后 base case 也要做出对应改变，详细思路解析和空间复杂度优化的解法见详细题解。

- **详细题解：一个方法团灭 LeetCode 股票买卖问题**

解法代码

```
class Solution {
    public int maxProfit(int[] prices, int fee) {
        int n = prices.length;
        int[][] dp = new int[n][2];
        for (int i = 0; i < n; i++) {
            if (i - 1 == -1) {
                // base case
                dp[i][0] = 0;
                dp[i][1] = -prices[i] - fee;
                //   dp[i][1]
                // = max(dp[i - 1][1], dp[i - 1][0] - prices[i] - fee)
                // = max(dp[-1][1], dp[-1][0] - prices[i] - fee)
                // = max(-inf, 0 - prices[i] - fee)
                // = -prices[i] - fee
                continue;
            }
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);
            dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i] -
fee);
        }
        return dp[n - 1][0];
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 121. 买卖股票的最佳时机 🟢
  - 122. 买卖股票的最佳时机 II 🟠
  - 123. 买卖股票的最佳时机 III 🔴
  - 188. 买卖股票的最佳时机 IV 🔴
  - 309. 最佳买卖股票时机含冷冻期 🟠
  - 剑指 Offer 63. 股票的最大利润 🟠

剑指 Offer 63. 股票的最大利润

这道题和 121. 买卖股票的最佳时机 相同。

174. 地下城游戏

LeetCode	力扣	难度
174. Dungeon Game	174. 地下城游戏	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，二维矩阵，动态规划**

地下城是由 M x N 个房间组成的二维网格，骑士（**K**）最初被安置在左上角的房间里，他必须拯救关在地下城的右下角公主（**P**）。

骑士的初始生命值为一个正整数，如果他的生命值在某一时刻降至 0 或以下，他会立即死亡。

每个矩阵元素代表地下城房间，如果房间的值为*负整数*，则表示骑士将遇到怪物损失生命值；如果房间的值为为*0*则说明什么都不会发生，如果房间的值为*正整数*，则表示骑士将增加生命值。

为了尽快到达公主，骑士决定每次只向右或向下移动一步，**编写一个函数来计算确保骑士能够拯救到公主所需的最低初始生命值。**

例如，考虑到如下布局的地下城，如果骑士遵循最佳路径 `右 -> 右 -> 下 -> 下`，则骑士的初始生命值至少为**7**。

基本思路

`dp` 函数的定义：**从 `grid[i][j]` 到达终点（右下角）所需的最少生命值是 `dp(grid, i, j)`**。

我们想求 `dp(0, 0)`，那就应该试图通过 `dp(i, j+1)` 和 `dp(i+1, j)` 推导出 `dp(i, j)`，这样才能不断逼近 base case，正确进行状态转移。

状态转移方程：

```
int res = min(
    dp(i + 1, j),
    dp(i, j + 1)
) - grid[i][j];

dp(i, j) = res <= 0 ? 1 : res;
```

- **详细题解：动态规划帮我通关了《魔塔》**

解法代码

```
class Solution {
```

```
    public int calculateMinimumHP(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        // 备忘录中都初始化为 -1
        memo = new int[m][n];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }

        return dp(grid, 0, 0);
    }

    // 备忘录，消除重叠子问题
    int[][] memo;

    /* 定义：从 (i, j) 到达右下角，需要的初始血量至少是多少 */
    int dp(int[][] grid, int i, int j) {
        int m = grid.length;
        int n = grid[0].length;
        // base case
        if (i == m - 1 && j == n - 1) {
            return grid[i][j] >= 0 ? 1 : -grid[i][j] + 1;
        }
        if (i == m || j == n) {
            return Integer.MAX_VALUE;
        }
        // 避免重复计算
        if (memo[i][j] != -1) {
            return memo[i][j];
        }
        // 状态转移逻辑
        int res = Math.min(
                dp(grid, i, j + 1),
                dp(grid, i + 1, j)
        ) - grid[i][j];
        // 骑士的生命值至少为 1
        memo[i][j] = res <= 0 ? 1 : res;

        return memo[i][j];
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

312. 戳气球

LeetCode	力扣	难度
312. Burst Balloons	312. 戳气球	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划**

有 `n` 个气球，编号为 `0` 到 `n - 1`，每个气球上都标有一个数字，这些数字存在数组 `nums` 中。

现在要求你戳破所有的气球。戳破第 `i` 个气球，你可以获得 `nums[i - 1] * nums[i] * nums[i + 1]` 枚硬币。这里的 `i - 1` 和 `i + 1` 代表和 `i` 相邻的两个气球的序号。如果 `i - 1` 或 `i + 1` 超出了数组的边界，那么就当它是一个数字为 `1` 的气球。

求所能获得硬币的最大数量。

示例 1：

```
输入：nums = [3,1,5,8]
输出：167
解释：
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8   +  1*3*8  + 1*8*1 = 167
```

基本思路

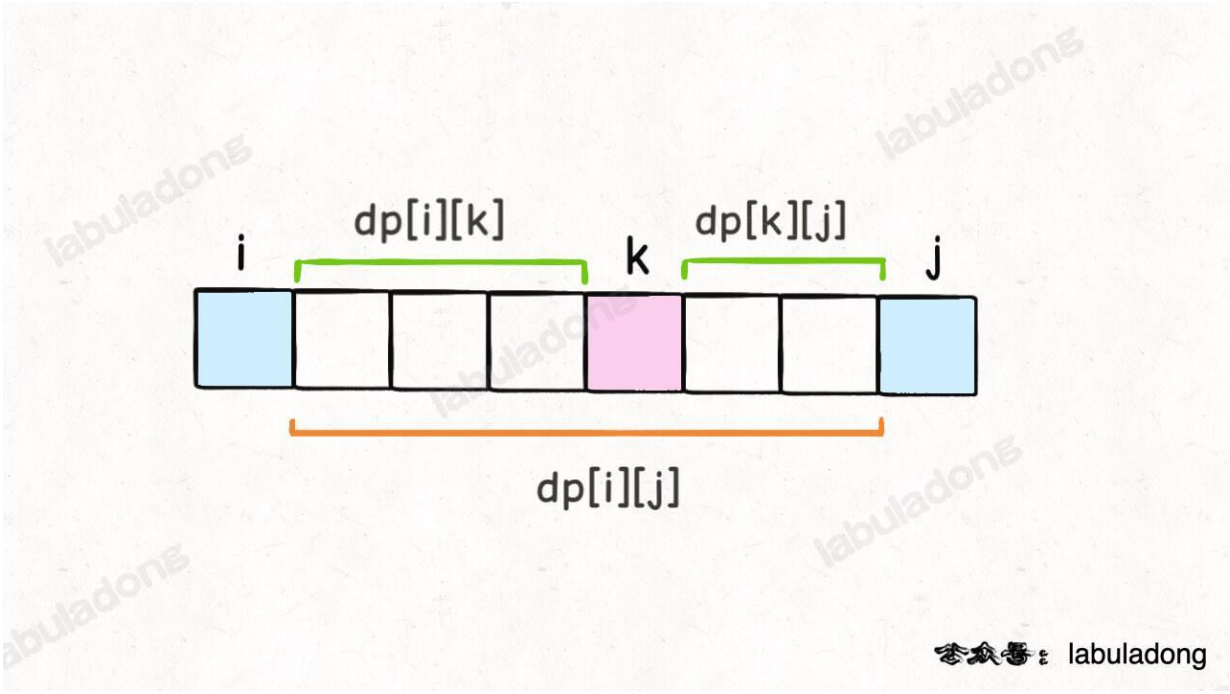
PS：这道题在《算法小抄》 的第 181 页。

这题比较难，建议看详细题解。

`dp` 数组的含义：`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，可以获得的最高分数为 `x`。

状态转移方程：

```
dp[i][j] = dp[i][k] + dp[k][j]
         + points[i]*points[k]*points[j]
```



- **详细题解：经典动态规划：戳气球**

解法代码

```
class Solution {
    public int maxCoins(int[] nums) {
        int n = nums.length;
        // 添加两侧的虚拟气球
        int[] points = new int[n + 2];
        points[0] = points[n + 1] = 1;
        for (int i = 1; i <= n; i++) {
            points[i] = nums[i - 1];
        }
        // base case 已经都被初始化为 0
        int[][] dp = new int[n + 2][n + 2];
        // 开始状态转移
        // i 应该从下往上
        for (int i = n; i >= 0; i--) {
            // j 应该从左往右
            for (int j = i + 1; j < n + 2; j++) {
                // 最后戳破的气球是哪个？
                for (int k = i + 1; k < j; k++) {
                    // 择优做选择
                    dp[i][j] = Math.max(
                            dp[i][j],
                            dp[i][k] + dp[k][j] + points[i] * points[j] *
points[k]
                    );
                }
            }
        }
        return dp[0][n + 1];
```

```
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

416. 分割等和子集

LeetCode	力扣	难度
416. Partition Equal Subset Sum	416. 分割等和子集	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，背包问题**

给你一个**只包含正整数的非空数组** `nums`。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11]。
```

基本思路

PS：这道题在《算法小抄》 的第 192 页。

对于这个问题，我们可以先对集合求和，得出 `sum`，然后把问题转化为背包问题：

给一个可装载重量为 `sum / 2` 的背包和 `N` 个物品，每个物品的重量为 `nums[i]`。现在让你装物品，是否存在一种装法，能够恰好将背包装满？

第一步要明确两点，「状态」和「选择」，状态就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」。

`dp` 数组的定义：`dp[i][j] = x` 表示，对于前 `i` 个物品，当前背包的容量为 `j` 时，若 `x` 为 `true`，则说明可以恰好将背包装满，若 `x` 为 `false`，则说明不能恰好将背包装满。

根据 `dp` 数组含义，可以根据「选择」对 `dp[i][j]` 得到以下状态转移：

如果不把 `nums[i]` 算入子集，**或者说你不把这第 `i` 个物品装入背包**，那么是否能够恰好装满背包，取决于上一个状态 `dp[i-1][j]`，继承之前的结果。

如果把 `nums[i]` 算入子集，**或者说你把这第 `i` 个物品装入了背包**，那么是否能够恰好装满背包，取决于状态 `dp[i-1][j-nums[i-1]]`。

- **详细题解：经典动态规划：子集背包问题**

解法代码

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int sum = 0;
        for (int num : nums) sum += num;
        // 和为奇数时，不可能划分成两个和相等的集合
        if (sum % 2 != 0) return false;
        int n = nums.length;
        sum = sum / 2;
        boolean[][] dp = new boolean[n + 1][sum + 1];
        // base case
        for (int i = 0; i <= n; i++)
            dp[i][0] = true;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                if (j - nums[i - 1] < 0) {
                    // 背包容量不足，不能装入第 i 个物品
                    dp[i][j] = dp[i - 1][j];
                } else {
                    // 装入或不装入背包
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
                }
            }
        }
        return dp[n][sum];
    }
}
```

- **类似题目**：
  - 剑指 Offer II 101. 分割等和子集 🟢

剑指 Offer II 101. 分割等和子集

这道题和 416. 分割等和子集 相同。

494. 目标和

LeetCode	力扣	难度
494. Target Sum	494. 目标和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，回溯算法，背包问题**

给你一个整数数组 `nums` 和一个整数 `target`，向数组中的每个整数前添加 `'+'` 或 `'-'`，然后串联起所有整数，可以构造一个**表达式**。

例如，`nums = [2, 1]`，可以在 `2` 之前添加 `'+'`，在 `1` 之前添加 `'-'`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同**表达式**的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 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
+1 + 1 + 1 + 1 - 1 = 3
```

基本思路

PS：这道题在《算法小抄》 的第 207 页。

这题有多种解法，可以用回溯算法剪枝求解，也可以用转化成背包问题求解，这里用前者吧，容易理解一些，背包问题解法可以查看详细题解。

对于每一个 1，要么加正号，要么加负号，把所有情况穷举出来，即可计算结果。

- **详细题解：目标和问题：背包问题的变体**

解法代码

```
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        if (nums.length == 0) return 0;
        return dp(nums, 0, target);
    }
```

```
    // 备忘录
    HashMap<String, Integer> memo = new HashMap<>();

    int dp(int[] nums, int i, int remain) {
        // base case
        if (i == nums.length) {
            if (remain == 0) return 1;
            return 0;
        }
        // 把它俩转成字符串才能作为哈希表的键
        String key = i + "," + remain;
        // 避免重复计算
        if (memo.containsKey(key)) {
            return memo.get(key);
        }
        // 还是穷举
        int result = dp(nums, i + 1, remain - nums[i]) + dp(nums, i + 1,
remain + nums[i]);
        // 记入备忘录
        memo.put(key, result);
        return result;
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

- **类似题目**：
  - 剑指 Offer II 102. 加减的目标值 🟠

剑指 Offer II 102. 加减的目标值

这道题和 494. 目标和 相同。

514. 自由之路

LeetCode	力扣	难度
514. Freedom Trail	514. 自由之路	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，动态规划

给定一个字符串 **ring**，表示刻在外环上的编码；给定另一个字符串 **key**，表示需要拼写的关键词。你需要算出能够拼写关键词中所有字符的**最少**步数。

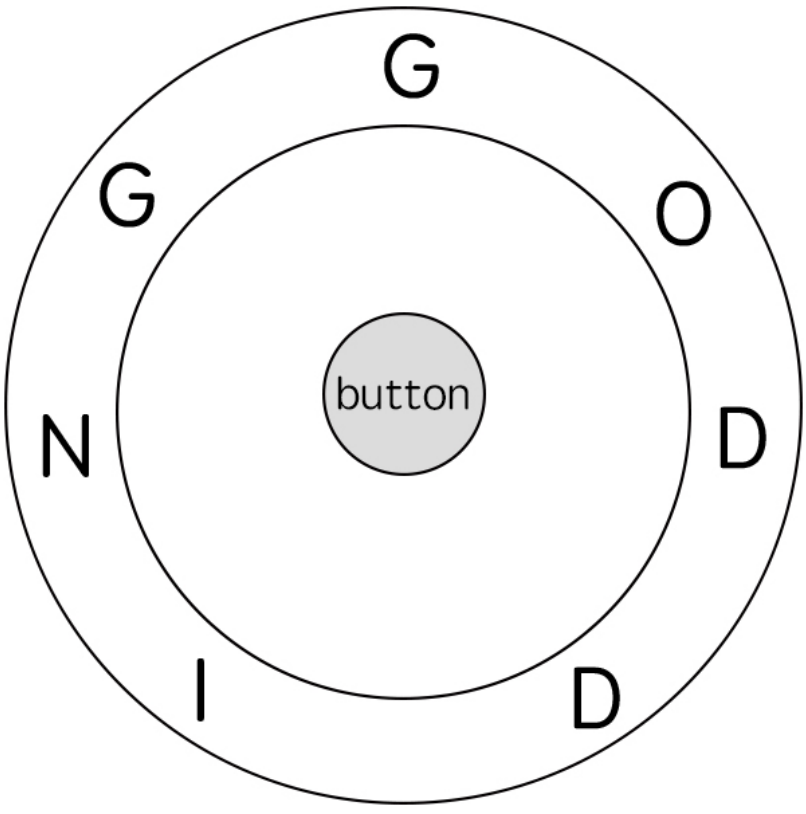
最初，**ring**的第一个字符与 12:00 方向对齐。你需要顺时针或逆时针旋转**ring**以使 **key**的一个字符在 12:00 方向对齐，然后按下中心按钮，以此逐个拼写完 **key**中的所有字符。

旋转 **ring** 拼出 key 字符 **key[i]** 的过程中：

1、你可以将 **ring**顺时针或逆时针旋转**一个位置**，计为 1 步。旋转的最终目的是将字符串**ring**的一个字符与 12:00 方向对齐，并且这个字符必须等于字符 **key[i]**。

2、如果字符 **key[i]** 已经对齐到 12:00 方向，你需要按下中心按钮进行拼写，这也将算作 **1 步**。按完之后，你可以开始拼写 **key**的下一个字符（下一阶段），直至完成所有拼写。

示例：




```
输入: ring = "godding", key = "gd"
输出: 4
```

解释：
对于 key 的第一个字符 'g'，已经在正确的位置，我们只需要 1 步来拼写这个字符。
对于 key 的第二个字符 'd'，我们需要逆时针旋转 ring "godding" 2 步使它变成 "ddinggo"。
当然，我们还需要 1 步进行拼写。
因此最终的输出是 4。

基本思路

dp 函数的定义如下：**当圆盘指针指向 ring[i] 时，输入字符串 key[j..] 至少需要 dp(ring, i, key, j) 次操作。**

根据这个定义，题目其实就是想计算 dp(ring, 0, key, 0) 的值，base case 就是 dp(ring, i, key, len(key)) = 0。

- **详细题解：动态规划帮我通关了《辐射4》**

解法代码

```
class Solution {
    public:
    // 字符 -> 索引列表
    unordered_map<char, vector<int>> charToIndex;
    // 备忘录
    vector<vector<int>> memo;

    /* 主函数 */
    int findRotateSteps(string ring, string key) {
        int m = ring.size();
        int n = key.size();
        // 备忘录全部初始化为 0
        memo.resize(m, vector<int>(n, 0));
        // 记录圆环上字符到索引的映射
        for (int i = 0; i < ring.size(); i++) {
            charToIndex[ring[i]].push_back(i);
        }
        // 圆盘指针最初指向 12 点钟方向，
        // 从第一个字符开始输入 key
        return dp(ring, 0, key, 0);
    }

    // 计算圆盘指针在 ring[i]，输入 key[j..] 的最少操作数
    int dp(string& ring, int i, string& key, int j) {
        // base case 完成输入
        if (j == key.size()) return 0;
        // 查找备忘录，避免重叠子问题
        if (memo[i][j] != 0) return memo[i][j];

        int n = ring.size();
        // 做选择
        int res = INT_MAX;
```

```
            // ring 上可能有多个字符 key[j]
            for (int k : charToIndex[key[j]]) {
                // 拨动指针的次数
                int delta = abs(k - i);
                // 选择顺时针还是逆时针
                delta = min(delta, n - delta);
                // 将指针拨到 ring[k]，继续输入 key[j+1..]
                int subProblem = dp(ring, k, key, j + 1);
                // 选择「整体」操作次数最少的
                // 加一是因为按动按钮也是一次操作
                res = min(res, 1 + delta + subProblem);
            }
            // 将结果存入备忘录
            memo[i][j] = res;
            return res;
        }
    };
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

518. 零钱兑换 II

LeetCode	力扣	难度
518. Coin Change 2	518. 零钱兑换 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，背包问题**

给你一个整数数组 `coins` 表示不同面额的硬币（硬币个数无限），另给一个整数 `amount` 表示总金额。请你计算并返回可以凑成总金额的硬币组合数，如果任何硬币组合都无法凑出总金额，返回 0。

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
```

基本思路

PS：这道题在《算法小抄》 的第 196 页。

我们可以把这个问题转化为背包问题的描述形式：

有一个背包，最大容量为 `amount`，有一系列物品 `coins`，每个物品的重量为 `coins[i]`，**每个物品的数量无限**。请问有多少种方法，能够把背包恰好装满？

第一步要明确两点，「状态」和「选择」，状态有两个，就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」。

`dp[i][j]` 的定义：若只使用前 `i` 个物品（可以重复使用），当背包容量为 `j` 时，有 `dp[i][j]` 种方法可以装满背包。

最终想得到的答案是 `dp[N][amount]`，其中 `N` 为 `coins` 数组的大小。

如果你不把这第 `i` 个物品装入背包，也就是说你不使用 `coins[i]` 这个面值的硬币，那么凑出面额 `j` 的方法数 `dp[i][j]` 应该等于 `dp[i-1][j]`，继承之前的结果。

如果你把这第 `i` 个物品装入了背包，也就是说你使用 `coins[i]` 这个面值的硬币，那么 `dp[i][j]` 应该等于 `dp[i][j-coins[i-1]]`。

- **详细题解：经典动态规划：完全背包问题**

解法代码

```
class Solution {
    public int change(int amount, int[] coins) {
        int n = coins.length;
        int[][] dp = new int[n + 1][amount + 1];
        // base case
        for (int i = 0; i <= n; i++)
            dp[i][0] = 1;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= amount; j++)
                if (j - coins[i-1] >= 0)
                    dp[i][j] = dp[i - 1][j]
                            + dp[i][j - coins[i-1]];
                else
                    dp[i][j] = dp[i - 1][j];
        }
        return dp[n][amount];
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

787. K 站中转内最便宜的航班

LeetCode	力扣	难度
787. Cheapest Flights Within K Stops	787. K 站中转内最便宜的航班	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，图论算法，最短路径算法**

有 `n` 个城市通过一些航班连接。给你一个数组 `flights`，其中 `flights[i] = [fromi, toi, pricei]`，表示该航班都从城市 `fromi` 开始，以价格 `pricei` 抵达 `toi`。

现在给定所有的城市和航班，以及出发城市 `src` 和目的地 `dst`，你的任务是找到出一条最多经过 `k` 站中转的路线，使得从 `src` 到 `dst` 的**价格最便宜**，并返回该价格。如果不存在这样的路线，则输出 `-1`。

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。
```

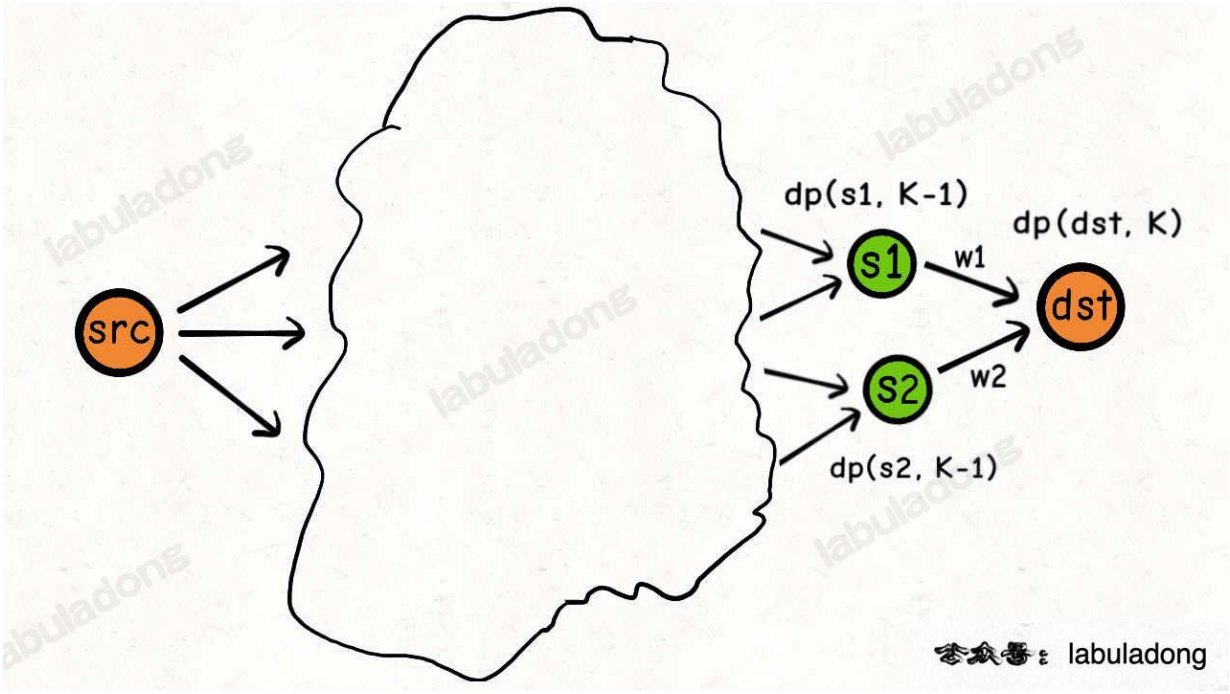
基本思路

`dp` 函数的定义：**从起点 `src` 出发，`k` 步之内（一步就是一条边）到达节点 `s` 的最小路径权重为 `dp(s, k)`**。

这样，题目想求的最小机票开销就可以用 `dp(dst, K+1)` 来表示。

状态转移方程：

```
dp(dst, k) = min(
    dp(s1, k - 1) + w1,
    dp(s2, k - 1) + w2
)
```



- **详细题解：旅游省钱大法：加权最短路径**

解法代码

```
class Solution {
    HashMap<Integer, List<int[]>> indegree;
    int src, dst;
    // 备忘录
    int[][] memo;

    public int findCheapestPrice(int n, int[][] flights, int src, int dst,
int K) {
        // 将中转站个数转化成边的条数
        K++;
        this.src = src;
        this.dst = dst;
        // 初始化备忘录，全部填一个特殊值
        memo = new int[n][K + 1];
        for (int[] row : memo) {
            Arrays.fill(row, -888);
        }

        indegree = new HashMap<>();
        for (int[] f : flights) {
            int from = f[0];
            int to = f[1];
            int price = f[2];
            // 记录谁指向该节点，以及之间的权重
            indegree.putIfAbsent(to, new LinkedList<>());
            indegree.get(to).add(new int[]{from, price});
        }

        return dp(dst, K);
```

```
        }

        // 定义：从 src 出发，k 步之内到达 s 的最短路径权重
        int dp(int s, int k) {
            // base case
            if (s == src) {
                return 0;
            }
            if (k == 0) {
                return -1;
            }
            // 查备忘录，防止冗余计算
            if (memo[s][k] != -888) {
                return memo[s][k];
            }

            // 初始化为最大值，方便等会取最小值
            int res = Integer.MAX_VALUE;
            if (indegree.containsKey(s)) {
                // 当 s 有入度节点时，分解为子问题
                for (int[] v : indegree.get(s)) {
                    int from = v[0];
                    int price = v[1];
                    // 从 src 到达相邻的入度节点所需的最短路径权重
                    int subProblem = dp(from, k - 1);
                    // 跳过无解的情况
                    if (subProblem != -1) {
                        res = Math.min(res, subProblem + price);
                    }
                }
            }
            // 存入备忘录
            memo[s][k] = res == Integer.MAX_VALUE ? -1 : res;
            return memo[s][k];
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

887. 鸡蛋掉落

LeetCode	力扣	难度
887. Super Egg Drop	887. 鸡蛋掉落	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划**

给你 `k` 枚相同的鸡蛋，并可以使用一栋从第 `1` 层到第 `n` 层共有 `n` 层楼的建筑。

已知存在楼层 `f`，满足 `0 <= f <= n`，任何从**高于** `f` 的楼层落下的鸡蛋都会碎，从 `f` 楼层或比它低的楼层落下的鸡蛋都不会破。

每次操作，你可以取一枚没有碎的鸡蛋并把它从任一楼层 `x` 扔下（满足 `1 <= x <= n`）。如果鸡蛋碎了，你就不能再次使用它。如果某枚鸡蛋扔下后没有摔碎，则可以在之后的操作中**重复使用**这枚鸡蛋。

请你计算并返回**确定 `f` 的最小操作次数**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，肯定能得出 f = 0。
否则，鸡蛋从 2 楼掉落。如果它碎了，肯定能得出 f = 1。
如果它没碎，那么肯定能得出 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。
```

基本思路

PS：这道题在《算法小抄》 的第 168 页。

这道经典题目的难度比较大，甚至连题目都不容易理解正确，建议看详细题解，有多种解法和优化手段。

`dp` 数组的定义：`dp[k][m] = n` 表示，当前有 `k` 个鸡蛋，可以尝试扔 `m` 次鸡蛋，这个条件下最坏情况下最多能确切测试一栋 `n` 层的楼

- **详细题解：经典动态规划：高楼扔鸡蛋**

解法代码

```
class Solution {
    public int superEggDrop(int K, int N) {
        // m 最多不会超过 N 次（线性扫描）
        int[][] dp = new int[K + 1][N + 1];
```

```
        // base case:
        // dp[0][..] = 0
        // dp[..][0] = 0
        // Java 默认初始化数组都为 0
        int m = 0;
        while (dp[K][m] < N) {
            m++;
            for (int k = 1; k <= K; k++)
                dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1;
        }
        return m;
    }
}
```

▶ 😻 代码可视化动画（需安装 Chrome 插件）😻

931. 下降路径最小和

LeetCode	力扣	难度
931. Minimum Falling Path Sum	931. 下降路径最小和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，动态规划

给你一个 `n x n` 的整数数组 `matrix`，请你找出并返回 `matrix` 的**下降路径**的**最小和**。

下降路径可以从第一行中的任何元素开始，并从每一行中选择一个元素。在下一行选择的元素和当前行所选元素最多相隔一列（即位于正下方或者沿对角线向左或者向右的第一个元素，可类比俄罗斯方块）。具体来说，位置 `(row, col)` 的下一个元素应当是 `(row + 1, col - 1)`、`(row + 1, col)` 或者 `(row + 1, col + 1)`。

示例 1：

```
输入：matrix = [[2,1,3],[6,5,4],[7,8,9]]
输出：13
解释：下面是两条和最小的下降路径，用加粗+斜体标注：
[[2,1,3],      [[2,1,3],
 [6,5,4],       [6,5,4],
 [7,8,9]]       [7,8,9]]
```

基本思路

对于 `matrix[i][j]`，只有可能从 `matrix[i-1][j],matrix[i-1][j-1],matrix[i-1][j+1]` 这三个位置转移过来。

`dp` 函数的定义：从第一行（`matrix[0][..]`）向下落，落到位置 `matrix[i][j]` 的最小路径和为 `dp(matrix,i, j)`，因此答案就是：

```
min(
    dp(matrix, i - 1, j),
    dp(matrix, i - 1, j - 1),
    dp(matrix, i - 1, j + 1)
)
```

- 详细题解：base case 和备忘录的初始值怎么定？

解法代码

```java
class Solution {
    public int minFallingPathSum(int[][] matrix) {
        int n = matrix.length;
        int res = Integer.MAX_VALUE;
        // 备忘录里的值初始化为 66666
        memo = new int[n][n];
        for (int i = 0; i < n; i++) {
            Arrays.fill(memo[i], 66666);
        }
        // 终点可能在 matrix[n-1] 的任意一列
        for (int j = 0; j < n; j++) {
            res = Math.min(res, dp(matrix, n - 1, j));
        }
        return res;
    }

    // 备忘录
    int[][] memo;

    int dp(int[][] matrix, int i, int j) {
        // 1、索引合法性检查
        if (i < 0 || j < 0 ||
                i >= matrix.length ||
                j >= matrix[0].length) {

            return 99999;
        }
        // 2、base case
        if (i == 0) {
            return matrix[0][j];
        }
        // 3、查找备忘录，防止重复计算
        if (memo[i][j] != 66666) {
            return memo[i][j];
        }
        // 进行状态转移
        memo[i][j] = matrix[i][j] + min(
                dp(matrix, i - 1, j),
                dp(matrix, i - 1, j - 1),
                dp(matrix, i - 1, j + 1)
        );

        return memo[i][j];
    }
    int min(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

583. 两个字符串的删除操作

LeetCode	力扣	难度
583. Delete Operation for Two Strings	583. 两个字符串的删除操作	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，子序列**

给定两个单词 *word1* 和 *word2*，找到使得 *word1* 和 *word2* 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

示例：

```
输入："sea", "eat"
输出：2
解释：第一步将 "sea" 变为 "ea"，第二步将 "eat" 变为 "ea"
```

基本思路

怎么样让两个字符串相同？直接全删成空串，肯定是相等了，但是题目又要求删除次数要尽可能少。

那怎么删？就是删成最长公共子序列嘛，换句话说，只要计算出最长公共子序列的长度，就能算出最少的删除次数了。

前文 1143. 最长公共子序列问题 讲了计算最长公共子序列的方法，这里就不展开了。

- **详细题解：经典动态规划：最长公共子序列**

解法代码

```
class Solution {
    public int minDistance(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        // 复用前文计算 lcs 长度的函数
        int lcs = longestCommonSubsequence(s1, s2);
        return m - lcs + n - lcs;
    }

    // 计算最长公共子序列的长度
    int longestCommonSubsequence(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        // 定义：s1[0..i-1] 和 s2[0..j-1] 的 lcs 长度为 dp[i][j]
        int[][] dp = new int[m + 1][n + 1];

        for (int i = 1; i <= m; i++) {
```

```
                for (int j = 1; j <= n; j++) {
                    // 现在 i 和 j 从 1 开始，所以要减一
                    if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                        // s1[i-1] 和 s2[j-1] 必然在 lcs 中
                        dp[i][j] = 1 + dp[i - 1][j - 1];
                    } else {
                        // s1[i-1] 和 s2[j-1] 至少有一个不在 lcs 中
                        dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]);
                    }
                }
            }
            return dp[m][n];
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 1143. 最长公共子序列 🟠
  - 712. 两个字符串的最小ASCII删除和 🟠
  - 剑指 Offer II 095. 最长公共子序列 🟠

712. 两个字符串的最小 ASCII 删除和

LeetCode	力扣	难度
712. Minimum ASCII Delete Sum for Two Strings	712. 两个字符串的最小ASCII删除和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划**

给定两个字符串 `s1, s2`，找到使两个字符串相等所需删除字符的 ASCII 值的最小和。

示例 1:

```
输入: s1 = "sea", s2 = "eat"
输出: 231
解释: 在 "sea" 中删除 "s" 并将 "s" 的值（115）加入总和。
在 "eat" 中删除 "t" 并将 116 加入总和。
结束时，两个字符串相等，115 + 116 = 231 就是符合条件的最小和。
```

示例 2:

```
输入: s1 = "delete", s2 = "leet"
输出: 403
解释: 在 "delete" 中删除 "dee" 字符串变成 "let"，
将 100[d]+101[e]+101[e] 加入总和。在 "leet" 中删除 "e" 将 101[e] 加入总和。
结束时，两个字符串都等于 "let"，结果即为 100+101+101+101 = 403。
如果改为将两个字符串转换为 "lee" 或 "eet"，我们会得到 433 或 417 的结果，比答案更大。
```

基本思路

这题本质上是考察最长公共子序列问题的解法，把 1143. 最长公共子序列问题 的解法代码稍微改一下就 OK 了。

- **详细题解：经典动态规划：最长公共子序列**

解法代码

```
class Solution {

    // 备忘录
    int memo[][];

    /* 主函数 */
    public int minimumDeleteSum(String s1, String s2) {
```

```
        int m = s1.length(), n = s2.length();
        // 备忘录值为 -1 代表未曾计算
        memo = new int[m][n];
        for (int[] row : memo)
            Arrays.fill(row, -1);

        return dp(s1, 0, s2, 0);
    }

    // 定义：将 s1[i..] 和 s2[j..] 删除成相同字符串，
    // 最小的 ASCII 码之和为 dp(s1, i, s2, j)。
    int dp(String s1, int i, String s2, int j) {
        int res = 0;
        // base case
        if (i == s1.length()) {
            // 如果 s1 到头了，那么 s2 剩下的都得删除
            for (; j < s2.length(); j++)
                res += s2.charAt(j);
            return res;
        }
        if (j == s2.length()) {
            // 如果 s2 到头了，那么 s1 剩下的都得删除
            for (; i < s1.length(); i++)
                res += s1.charAt(i);
            return res;
        }

        if (memo[i][j] != -1) {
            return memo[i][j];
        }

        if (s1.charAt(i) == s2.charAt(j)) {
            // s1[i] 和 s2[j] 都是在 lcs 中的，不用删除
            memo[i][j] = dp(s1, i + 1, s2, j + 1);
        } else {
            // s1[i] 和 s2[j] 至少有一个不在 lcs 中，删一个
            memo[i][j] = Math.min(
                    s1.charAt(i) + dp(s1, i + 1, s2, j),
                    s2.charAt(j) + dp(s1, i, s2, j + 1)
            );
        }
        return memo[i][j];
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 1143. 最长公共子序列 🟠
  - 583. 两个字符串的删除操作 🟠
  - 剑指 Offer II 095. 最长公共子序列 🟠

416. 分割等和子集

LeetCode	力扣	难度
416. Partition Equal Subset Sum	416. 分割等和子集	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，背包问题**

给你一个**只包含正整数的非空数组** `nums`。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11]。
```

基本思路

PS：这道题在《算法小抄》 的第 192 页。

对于这个问题，我们可以先对集合求和，得出 `sum`，然后把问题转化为背包问题：

给一个可装载重量为 `sum / 2` 的背包和 `N` 个物品，每个物品的重量为 `nums[i]`。现在让你装物品，是否存在一种装法，能够恰好将背包装满？

第一步要明确两点，「状态」和「选择」，状态就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」。

`dp` 数组的定义：`dp[i][j] = x` 表示，对于前 `i` 个物品，当前背包的容量为 `j` 时，若 `x` 为 `true`，则说明可以恰好将背包装满，若 `x` 为 `false`，则说明不能恰好将背包装满。

根据 `dp` 数组含义，可以根据「选择」对 `dp[i][j]` 得到以下状态转移：

如果不把 `nums[i]` 算入子集，**或者说你不把这第 `i` 个物品装入背包**，那么是否能够恰好装满背包，取决于上一个状态 `dp[i-1][j]`，继承之前的结果。

如果把 `nums[i]` 算入子集，**或者说你把这第 `i` 个物品装入了背包**，那么是否能够恰好装满背包，取决于状态 `dp[i-1][j-nums[i-1]]`。

- **详细题解：经典动态规划：子集背包问题**

解法代码

```java
class Solution {
    public boolean canPartition(int[] nums) {
        int sum = 0;
        for (int num : nums) sum += num;
        // 和为奇数时，不可能划分成两个和相等的集合
        if (sum % 2 != 0) return false;
        int n = nums.length;
        sum = sum / 2;
        boolean[][] dp = new boolean[n + 1][sum + 1];
        // base case
        for (int i = 0; i <= n; i++)
            dp[i][0] = true;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= sum; j++) {
                if (j - nums[i - 1] < 0) {
                    // 背包容量不足，不能装入第 i 个物品
                    dp[i][j] = dp[i - 1][j];
                } else {
                    // 装入或不装入背包
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
                }
            }
        }
        return dp[n][sum];
    }
}
```

- **类似题目**：
  - 剑指 Offer II 101. 分割等和子集 🟢

剑指 Offer II 101. 分割等和子集

这道题和 416. 分割等和子集 相同。

494. 目标和

LeetCode	力扣	难度
494. Target Sum	494. 目标和	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二维动态规划，动态规划，回溯算法，背包问题**

给你一个整数数组 `nums` 和一个整数 `target`，向数组中的每个整数前添加 `'+'` 或 `'-'`，然后串联起所有整数，可以构造一个**表达式**。

例如，`nums = [2, 1]`，可以在 `2` 之前添加 `'+'`，在 `1` 之前添加 `'-'`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同**表达式**的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 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
+1 + 1 + 1 + 1 - 1 = 3
```

基本思路

PS：这道题在《算法小抄》 的第 207 页。

这题有多种解法，可以用回溯算法剪枝求解，也可以用转化成背包问题求解，这里用前者吧，容易理解一些，背包问题解法可以查看详细题解。

对于每一个 1，要么加正号，要么加负号，把所有情况穷举出来，即可计算结果。

- **详细题解：目标和问题：背包问题的变体**

解法代码

```
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        if (nums.length == 0) return 0;
        return dp(nums, 0, target);
    }
```

```
    // 备忘录
    HashMap<String, Integer> memo = new HashMap<>();

    int dp(int[] nums, int i, int remain) {
        // base case
        if (i == nums.length) {
            if (remain == 0) return 1;
            return 0;
        }
        // 把它俩转成字符串才能作为哈希表的键
        String key = i + "," + remain;
        // 避免重复计算
        if (memo.containsKey(key)) {
            return memo.get(key);
        }
        // 还是穷举
        int result = dp(nums, i + 1, remain - nums[i]) + dp(nums, i + 1,
remain + nums[i]);
        // 记入备忘录
        memo.put(key, result);
        return result;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 剑指 Offer II 102. 加减的目标值 🟠

剑指 Offer II 102. 加减的目标值

这道题和 494. 目标和 相同。

518. 零钱兑换 II

LeetCode	力扣	难度
518. Coin Change 2	518. 零钱兑换 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二维动态规划，动态规划，背包问题

给你一个整数数组 `coins` 表示不同面额的硬币（硬币个数无限），另给一个整数 `amount` 表示总金额。请你计算并返回可以凑成总金额的硬币组合数，如果任何硬币组合都无法凑出总金额，返回 0。

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
```

基本思路

PS：这道题在《算法小抄》 的第 196 页。

我们可以把这个问题转化为背包问题的描述形式：

有一个背包，最大容量为 `amount`，有一系列物品 `coins`，每个物品的重量为 `coins[i]`，**每个物品的数量无限**。请问有多少种方法，能够把背包恰好装满？

第一步要明确两点，「状态」和「选择」，状态有两个，就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」。

`dp[i][j]` 的定义：若只使用前 `i` 个物品（可以重复使用），当背包容量为 `j` 时，有 `dp[i][j]` 种方法可以装满背包。

最终想得到的答案是 `dp[N][amount]`，其中 `N` 为 `coins` 数组的大小。

如果你不把这第 `i` 个物品装入背包，也就是说你不使用 `coins[i]` 这个面值的硬币，那么凑出面额 `j` 的方法数 `dp[i][j]` 应该等于 `dp[i-1][j]`，继承之前的结果。

如果你把这第 `i` 个物品装入了背包，也就是说你使用 `coins[i]` 这个面值的硬币，那么 `dp[i][j]` 应该等于 `dp[i][j-coins[i-1]]`。

- 详细题解：经典动态规划：完全背包问题

解法代码

```
class Solution {
    public int change(int amount, int[] coins) {
        int n = coins.length;
        int[][] dp = new int[n + 1][amount + 1];
        // base case
        for (int i = 0; i <= n; i++)
            dp[i][0] = 1;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= amount; j++)
                if (j - coins[i-1] >= 0)
                    dp[i][j] = dp[i - 1][j]
                            + dp[i][j - coins[i-1]];
                else
                    dp[i][j] = dp[i - 1][j];
        }
        return dp[n][amount];
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

面试题45. 把数组排成最小的数

LeetCode	力扣	难度
面试题45. 把数组排成最小的数 LCOF	面试题45. 把数组排成最小的数	🟠

- 标签：数学

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

示例 1:

```
输入: [10,2]
输出: "102"
```

示例 2:

```
输入: [3,30,34,5,9]
输出: "3033459"
```

基本思路

这个题比较简单，想让结果最小，肯定是让最高位的数字尽可能小嘛。所以我们只要按照特殊规则对整个数组排序就可以了：

1、最高位最小的那个数字排在前面。

2、如果最高位的数字相同，那么次高位的数字较小的排在前面，如果次高位数字也相同，就比较次次高位，以此类推。

一个取巧的办法是把 `nums` 中的元素转换成字符串，然后通过字符串拼接的方式进行大小比较，具体看代码吧。

解法代码

```
class Solution {
    public String minNumber(int[] nums) {
        int n = nums.length;
        String[] strs = new String[n];
        for (int i = 0; i < n; i++) {
            strs[i] = Integer.toString(nums[i]);
        }
        Arrays.sort(strs, (s1, s2) -> {
```

```
                // 看看那种拼接方式得到的数字更小，排前面
                // 不用转成 int 类型，因为字符串的比较算法和正整数的比较算法是一样的
                // 而且拼接字符串比较长，会导致 int 类型溢出
                return (s1 + s2).compareTo(s2 + s1);
            });

            return String.join("", strs);
        }
    }
```

▶ 代码可视化动画（需安装 Chrome 插件）

面试题61. 扑克牌中的顺子

LeetCode	力扣	难度
面试题61. 扑克牌中的顺子 LCOF	面试题61. 扑克牌中的顺子	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

从**若干副扑克牌**中随机抽 5 张牌，判断是不是一个顺子，即这 5 张牌是不是连续的。2～10 为数字本身，A 为 1，J 为 11，Q 为 12，K 为 13，而大、小王为 0，可以看成任意数字。A 不能视为 14。

示例 1:

```
输入: [1,2,3,4,5]
输出: True
```

示例 2:

```
输入: [0,0,1,2,5]
输出: True
```

基本思路

这道题看起来不难，我想到的最直接的方式是排序，因为排序之后很容易判定顺子（元素递增 1）。

不过我按照这个思路写了下解法，发现大小王的万能匹配稍微有些难处理。所以仔细想了下，这题可以通过计算 5 个元素中的最大值和最小值来判断是否递增。

只要最大值和最小值的差等于 4，就说明是 5 个递增的元素。当然，由于大小王用 0 表示，所以 0 不应该参与最大最小值的选取。另外，顺子中不应该出现重复元素。

如果结合以上思路，就会发现其实不需要排序了，直接遍历一遍 `nums` 数组找最大最小值，然后再用一个 `HashSet` 判断重复元素就行了。具体看代码吧。

解法代码

```
class Solution {
    public boolean isStraight(int[] nums) {
        Set<Integer> numSet = new HashSet<>();
        int max = Integer.MIN_VALUE, min = Integer.MAX_VALUE;
        for (int num : nums) {
            if (num == 0) {
                // 小王不用管
```

```
                continue;
            }
            if (numSet.contains(num)) {
                // 出现重复，不可能是顺子
                return false;
            }
            numSet.add(num);

            max = Math.max(max, num);
            min = Math.min(min, num);
        }
        // 考虑到大小王，只要差小于 4 就一定可以凑出顺子
        return max - min <= 4;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

8. 字符串转换整数 (atoi)

LeetCode	力扣	难度
8. String to Integer (atoi)	8. 字符串转换整数 (atoi)	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数（类似 C/C++ 中的 `atoi` 函数）。

函数 `myAtoi(string s)` 的算法如下：

1、读入字符串并丢弃无用的前导空格

2、检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。确定最终结果是负数还是正数。如果两者都不存在，则假定结果为正。

3、读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。

4、将前面步骤读入的这些数字转换为整数（即，"123" - > 123，"0032" - > 32）。如果没有读入数字，则整数为 `0`。必要时更改符号（从步骤 2 开始）。

5、如果整数数超过 32 位有符号整数范围，需要截断这个整数，使其保持在这个范围内。如果大于 INT_MAX，则返回 INT_MAX，如果小于 INT_MIN，则返回 INT_MIN。

6、返回整数作为最终结果。

基本思路

这道题说实话没有什么难度，无非就是处理数字、符号、空格和 int 溢出的细节问题，具体看代码吧，把每一步的注释写清楚就不容易在细节上出错了。

解法代码

```
class Solution {
    public int myAtoi(String str) {
        int n = str.length();
        int i = 0;
        // 记录正负号
        int sign = 1;
        // 用 long 避免 int 溢出
        long res = 0;
        // 跳过前导空格
        while (i < n && str.charAt(i) == ' ') {
            i++;
        }
        if (i == n) {
```

```
            return 0;
        }

        // 记录符号位
        if (str.charAt(i) == '-') {
            sign = -1;
            i++;
        } else if (str.charAt(i) == '+') {
            i++;
        }
        if (i == n) {
            return 0;
        }

        // 统计数字位
        while (i < n && '0' <= str.charAt(i) && str.charAt(i) <= '9') {
            res = res * 10 + str.charAt(i) - '0';
            if (res > Integer.MAX_VALUE) {
                break;
            }
            i++;
        }
        // 如果溢出，强转成 int 就会和真实值不同
        if ((int) res != res) {
            return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
        }
        return (int) res * sign;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 面试题67. 把字符串转换成整数 🟠

面试题67. 把字符串转换成整数

LeetCode	力扣	难度
面试题67. 把字符串转换成整数 LCOF	面试题67. 把字符串转换成整数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数（类似 C/C++ 中的 `atoi` 函数）。

函数 `myAtoi(string s)` 的算法如下：

1、读入字符串并丢弃无用的前导空格

2、检查下一个字符（假设还未到字符末尾）为正还是负号，读取该字符（如果有）。确定最终结果是负数还是正数。如果两者都不存在，则假定结果为正。

3、读入下一个字符，直到到达下一个非数字字符或到达输入的结尾。字符串的其余部分将被忽略。

4、将前面步骤读入的这些数字转换为整数（即，"123" - > 123，"0032" - > 32）。如果没有读入数字，则整数为 0。必要时更改符号（从步骤 2 开始）。

5、如果整数数超过 32 位有符号整数范围，需要截断这个整数，使其保持在这个范围内。如果大于 INT_MAX，则返回 INT_MAX，如果小于 INT_MIN，则返回 INT_MIN。

6、返回整数作为最终结果。

基本思路

这道题和 8. 字符串转换整数 (atoi) 相同。

这道题说实话没有什么难度，无非就是处理数字、符号、空格和 int 溢出的细节问题，具体看代码吧，把每一步的注释写清楚就不容易在细节上出错了。

解法代码

```
class Solution {
    public int myAtoi(String str) {
        int n = str.length();
        int i = 0;
        // 记录正负号
        int sign = 1;
        // 用 long 避免 int 溢出
        long res = 0;
        // 跳过前导空格
        while (i < n && str.charAt(i) == ' ') {
            i++;
        }
        if (i == n) {
            return 0;
        }

        // 记录符号位
        if (str.charAt(i) == '-') {
            sign = -1;
            i++;
        } else if (str.charAt(i) == '+') {
            i++;
        }
        if (i == n) {
            return 0;
        }

        // 统计数字位
```

```
        while (i < n && '0' <= str.charAt(i) && str.charAt(i) <= '9') {
            res = res * 10 + str.charAt(i) - '0';
            if (res > Integer.MAX_VALUE) {
                break;
            }
            i++;
        }
        // 如果溢出，强转成 int 就会和真实值不同
        if ((int) res != res) {
            return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
        }
        return (int) res * sign;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 面试题67. 把字符串转换成整数 🟠

9. 回文数

LeetCode	力扣	难度
9. Palindrome Number	9. 回文数	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

给你一个整数 x，如果 x 是一个回文整数，返回 true；否则，返回 false。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。例如，121 是回文，而 123 不是。

示例 1：

```
输入：x = 121
输出：true
```

示例 2：

```
输入：x = -121
输出：false
解释：从左向右读，为 -121。从右向左读，为 121-。因此它不是一个回文数。
```

示例 3：

```
输入：x = 10
输出：false
解释：从右向左读，为 01。因此它不是一个回文数。
```

基本思路

如果让你判断回文串应该很简单，我在 数组双指针技巧汇总 中讲过。

操作数字没办法像操作字符串那么简单粗暴，但只要你要知道我在 Rabin Karp 算法详解 中讲到的从最高位开始生成数字的技巧，就能轻松解决这个问题：

```
String s = "8264";
int number = 0;
for (int i = 0; i < s.size(); i++) {
    // 将字符转化成数字
    number = 10 * number + (s[i] - '0');
```

```
    print(number);
}
// 打印输出:
// 8
// 82
// 826
// 8264
```

你**从后往前**把 x 的每一位拿出来，用这个技巧生成一个数字 y，如果 y 和 x 相等，则说明 x 是回文数字。

如何**从后往前**拿出一个数字的每一位？和 10 求余数就行了呗。看代码吧。

解法代码

```
class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0) {
            return false;
        }
        int temp = x;
        // y 是 x 翻转后的数字
        int y = 0;
        while (temp > 0) {
            int last_num = temp % 10;
            temp = temp / 10;
            // 从最高位生成数字的技巧
            y = y * 10 + last_num;
        }
        return y == x;
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

17. 电话号码的字母组合

LeetCode	力扣	难度
17. Letter Combinations of a Phone Number	17. 电话号码的字母组合	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：回溯算法，数学

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。答案可以按**任意顺序** 返回。

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 1：

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

基本思路

你需要先看前文 回溯算法详解 和 回溯算法团灭子集、排列、组合问题，然后看这道题就很简单了，无非是回溯算法的运用而已。

组合问题本质上就是遍历一棵回溯树，套用回溯算法代码框架即可。

解法代码

```
class Solution {
    // 每个数字到字母的映射
    String[] mapping = new String[] {
            "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
"wxyz"
    };

    List<String> res = new LinkedList<>();

    public List<String> letterCombinations(String digits) {
```

```
        if (digits.isEmpty()) {
            return res;
        }
        // 从 digits[0] 开始进行回溯
        backtrack(digits, 0, new StringBuilder());
        return res;
    }

    // 回溯算法主函数
    void backtrack(String digits, int start, StringBuilder sb) {
        if (sb.length() == digits.length()) {
            // 到达回溯树底部
            res.add(sb.toString());
            return;
        }
        // 回溯算法框架
        for (int i = start; i < digits.length(); i++) {
            int digit = digits.charAt(i) - '0';
            for (char c : mapping[digit].toCharArray()) {
                // 做选择
                sb.append(c);
                // 递归下一层回溯树
                backtrack(digits, i + 1, sb);
                // 撤销选择
                sb.deleteCharAt(sb.length() - 1);
            }
        }
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

50. Pow(x, n)

LeetCode	力扣	难度
50. Pow(x, n)	50. Pow(x, n)	🟠

- 标签：数学

实现 pow(x,n)，即计算 x 的整数 n 次幂函数（即 x^n）。

示例 1：

```
输入：x = 2.00000, n = 10
输出：1024.00000
```

基本思路

幂运算是经典的数学运算技巧了，建议你看下前文 如何高效进行模幂运算 就能很容易理解解法代码里的思想了。这道题唯一有点恶心的就是 k 的取值范围特别大，不能直接加符号，否则会造成整型溢出，具体解法看代码吧。

解法代码

```
class Solution {
    public double myPow(double a, int k) {
        if (k == 0) return 1;

        if (k == Integer.MIN_VALUE) {
            // 把 k 是 INT_MIN 的情况单独拿出来处理
            // 避免 -k 整型溢出
            return myPow(1 / a, -(k + 1)) / a;
        }

        if (k < 0) {
            return myPow(1 / a, -k);
        }

        if (k % 2 == 1) {
            // k 是奇数
            return (a * myPow(a, k - 1));
        } else {
            // k 是偶数
            double sub = myPow(a, k / 2);
            return (sub * sub);
        }
```

```
        }
    }
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
    - 剑指 Offer 16. 数值的整数次方 🟠

剑指 Offer 16. 数值的整数次方

这道题和 50. Pow(x, n) 相同。

67. 二进制求和

LeetCode	力扣	难度
67. Add Binary	67. 二进制求和	🟢

- **标签：位运算，数学**

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 **非空** 字符串且只包含数字 `1` 和 `0`。

示例 1:

```
输入: a = "11", b = "1"
输出: "100"
```

示例 2:

```
输入: a = "1010", b = "1011"
输出: "10101"
```

基本思路

这道题的关键就是让你用代码模拟加法的运算过程。**加法的模拟可以理解为是一个通用模式，需要你用一个 `carry` 变量记录并正确进位**，建议你先去做一下 2. 两数相加 再回来看这道题就手到擒来了。

解法代码

```
class Solution {
    public String addBinary(String a, String b) {
        // 先把输入的这两个二进制串反转，低位放在前面，方便处理进位
        a = new StringBuilder(a).reverse().toString();
        b = new StringBuilder(b).reverse().toString();
        // 存储结果
        StringBuilder sb = new StringBuilder();

        int m = a.length(), n = b.length();
        // carry 记录进位
        int carry = 0;
        int i = 0;

        // 开始类似 [2. 两数相加](#2) 的加法模拟逻辑
```

```
        // 只是这里运算的是二进制字符串
        while (i < Math.max(m, n) || carry > 0) {
            int val = carry;
            val += i < m ? (a.charAt(i) - '0') : 0;
            val += i < n ? (b.charAt(i) - '0') : 0;
            sb.append(val % 2);
            carry = val / 2;
            i++;
        }

        return sb.reverse().toString();
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

77. 组合

LeetCode	力扣	难度
77. Combinations	77. 组合	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，数学**

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。你可以按**任何顺序**返回答案。

示例 1：

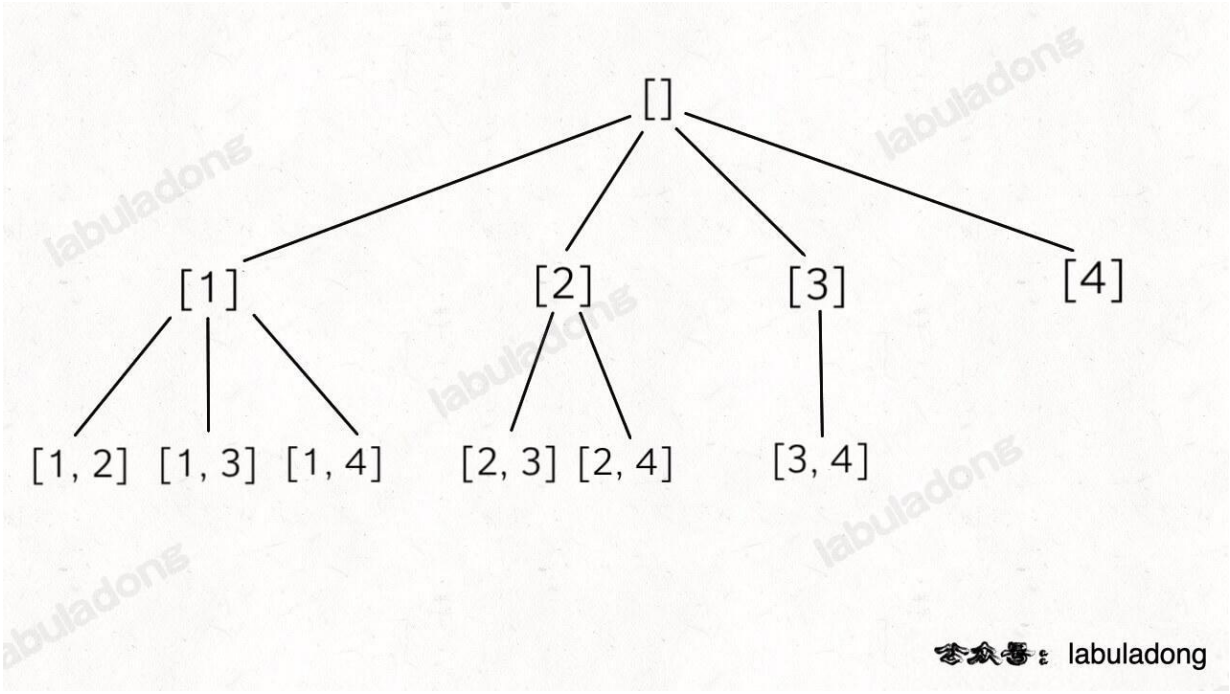
```
输入：n = 4, k = 2
输出：
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

基本思路

> 本文有视频版：回溯算法秒杀所有排列/组合/子集问题

PS：这道题在《算法小抄》 的第 293 页。

这也是典型的回溯算法，`k` 限制了树的高度，`n` 限制了树的宽度，继续套我们以前讲过的 回溯算法模板框架 就行了：



- **详细题解：回溯算法秒杀所有排列/组合/子集问题**

解法代码

```
class Solution {
    public:

    vector<vector<int>>res;
    vector<vector<int>> combine(int n, int k) {
        if (k <= 0 || n <= 0) return res;
        vector<int> track;
        backtrack(n, k, 1, track);
        return res;
    }

    void backtrack(int n, int k, int start, vector<int>& track) {
        // 到达树的底部
        if (k == track.size()) {
            res.push_back(track);
            return;
        }
        // 注意 i 从 start 开始递增
        for (int i = start; i <= n; i++) {
            // 做选择
            track.push_back(i);
            backtrack(n, k, i + 1, track);
            // 撤销选择
            track.pop_back();
        }
    }
};
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目：**
  - 216. 组合总和 III 🟠
  - 39. 组合总和 🟠
  - 40. 组合总和 II 🟠
  - 46. 全排列 🟠
  - 47. 全排列 II 🟠
  - 78. 子集 🟠
  - 90. 子集 II 🟠
  - 剑指 Offer II 079. 所有子集 🟠
  - 剑指 Offer II 080. 含有 k 个元素的组合 🟠
  - 剑指 Offer II 081. 允许重复选择元素的组合 🟠
  - 剑指 Offer II 082. 含有重复元素集合的组合 🟠
  - 剑指 Offer II 083. 没有重复元素集合的全排列 🟠
  - 剑指 Offer II 084. 含有重复元素集合的全排列 🟠

78. 子集

LeetCode	力扣	难度
78. Subsets	78. 子集	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：回溯算法，数学**

给你一个整数数组 `nums`，数组中的元素**互不相同**，返回该数组所有可能的子集（幂集）。

解集**不能**包含重复的子集。你可以按**任意顺序**返回解集。

示例 1：

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

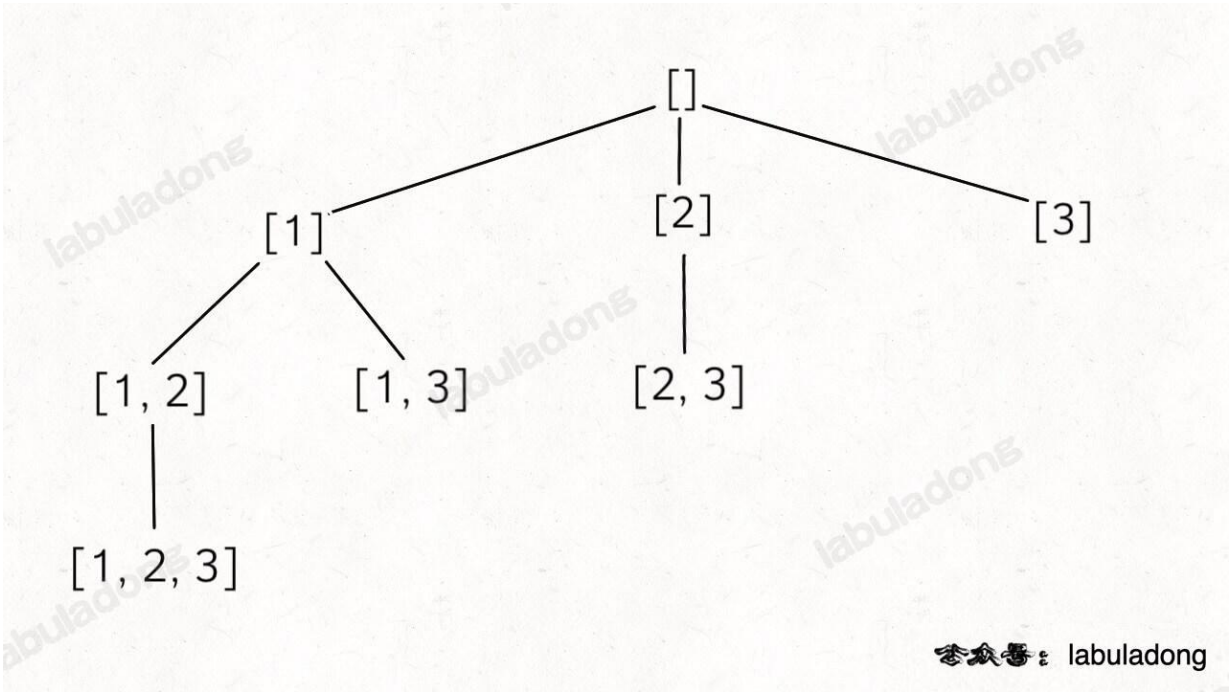
基本思路

> 本文有视频版：回溯算法秒杀所有排列/组合/子集问题

PS：这道题在《算法小抄》 的第 293 页。

有两种方法解决这道题，这里主要说回溯算法思路，因为比较通用，可以套前文 回溯算法详解 写过回溯算法模板。

本质上子集问题就是遍历这样用一棵回溯树：



- **详细题解：回溯算法秒杀所有排列/组合/子集问题**

解法代码

```
class Solution {
    public:
    vector<vector<int>> res;
    vector<vector<int>> subsets(vector<int>& nums) {
        // 记录走过的路径
        vector<int> track;
        backtrack(nums, 0, track);
        return res;
    }

    void backtrack(vector<int>& nums, int start, vector<int>& track) {
        res.push_back(track);
        for (int i = start; i < nums.size(); i++) {
            // 做选择
            track.push_back(nums[i]);
            // 回溯
            backtrack(nums, i + 1, track);
            // 撤销选择
            track.pop_back();
        }
    }
};
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 216. 组合总和 III 🟠
  - 39. 组合总和 🟠
  - 40. 组合总和 II 🟠
  - 46. 全排列 🟠
  - 47. 全排列 II 🟠
  - 77. 组合 🟠
  - 90. 子集 II 🟠
  - 剑指 Offer II 079. 所有子集 🟠
  - 剑指 Offer II 080. 含有 k 个元素的组合 🟠
  - 剑指 Offer II 081. 允许重复选择元素的组合 🟠
  - 剑指 Offer II 082. 含有重复元素集合的组合 🟠
  - 剑指 Offer II 083. 没有重复元素集合的全排列 🟠
  - 剑指 Offer II 084. 含有重复元素集合的全排列 🟠

剑指 Offer II 079. 所有子集

这道题和 78. 子集 相同。

剑指 Offer II 080. 含有 k 个元素的组合

这道题和 77. 组合 相同。

134. 加油站

LeetCode	力扣	难度
134. Gas Station	134. 加油站	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学，贪心算法**

在一条环路上有 *N* 个加油站，其中第 *i* 个加油站有汽油 `gas[i]` 升。你有一辆油箱容量无限的的汽车，从第 *i* 个加油站开往第 *i+1* 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

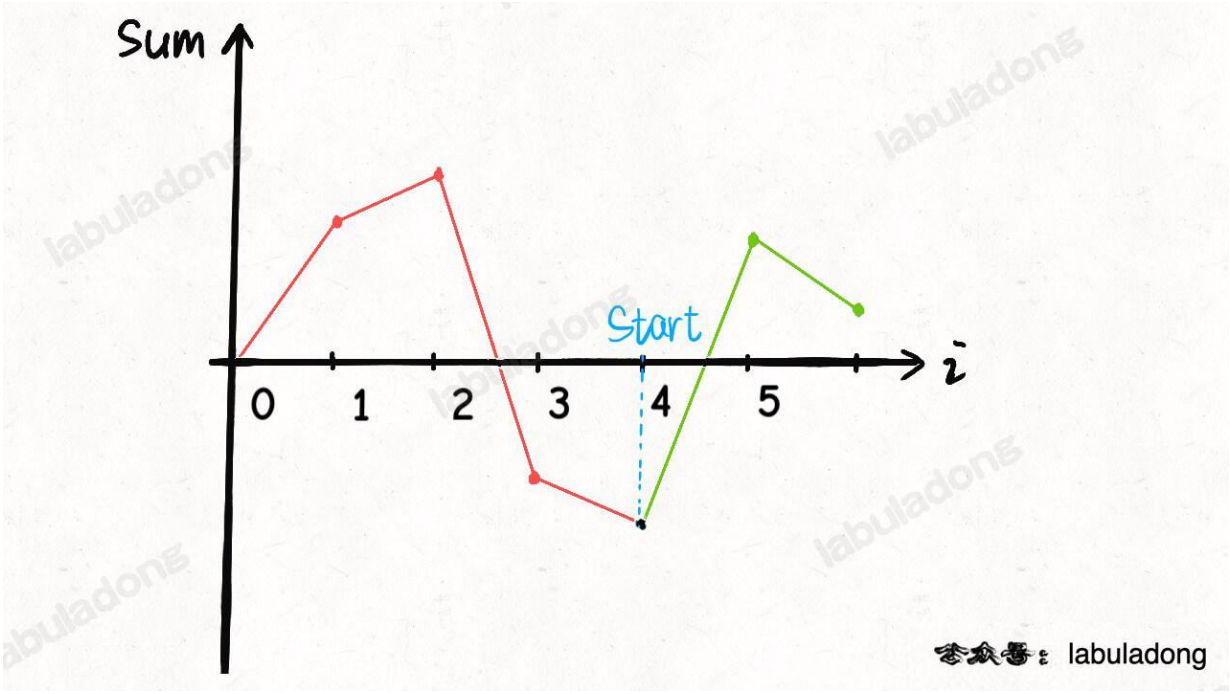
如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1（如果题目有解，该答案即为唯一答案）。

示例 1:

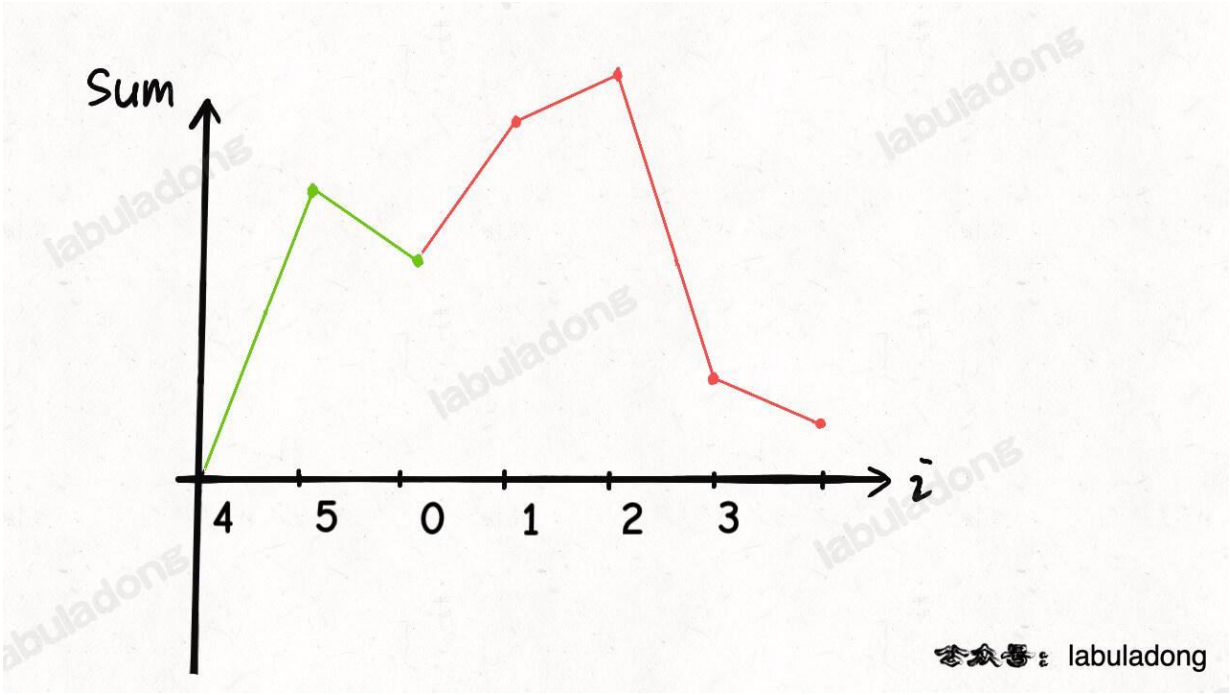
```
输入:
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]

输出: 3

解释:
从 3 号加油站（索引为 3 处）出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油
开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油
开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油
开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油
开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。
因此，3 可为起始索引。
```

基本思路

这题可以通过观察图像或者贪心算法解决，这里就说图像法，对贪心算法有兴趣的读者请看详细题解。



`sum` 代表路途中油箱的油量，如果把这个「最低点」作为起点，即把这个点作为坐标轴原点，就相当于把图像「最大限度」向上平移了：



如果经过平移后图像全部在 x 轴以上，就说明可以行使一周。

- **详细题解：老司机加油算法**

解法代码

```
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int n = gas.length;
        // 相当于图像中的坐标点和最低点
        int sum = 0, minSum = 0;
```

```
        int start = 0;
        for (int i = 0; i < n; i++) {
            sum += gas[i] - cost[i];
            if (sum < minSum) {
                // 经过第 i 个站点后，使 sum 到达新低
                // 所以站点 i + 1 就是最低点（起点）
                start = i + 1;
                minSum = sum;
            }
        }
        if (sum < 0) {
            // 总油量小于总的消耗，无解
            return -1;
        }
        // 环形数组特性
        return start == n ? 0 : start;
    }
}
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

136. 只出现一次的数字

LeetCode	力扣	难度
136. Single Number	136. 只出现一次的数字	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：位运算，数学**

给定一个**非空**整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

基本思路

这里就可以运用异或运算的性质：

一个数和它本身做异或运算结果为 0，即 `a ^ a = 0`；一个数和 0 做异或运算的结果为它本身，即 `a ^ 0 = a`。

对于这道题目，我们只要把所有数字进行异或，成对儿的数字就会变成 0，落单的数字和 0 做异或还是它本身，所以最后异或的结果就是只出现一次的元素。

- **详细题解：常用的位操作**

解法代码

```
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for (int n : nums) {
            res ^= n;
        }
        return res;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目：**
  - 191. 位1的个数 🟢
  - 231. 2 的幂 🟢
  - 268. 丢失的数字 🟢
  - 剑指 Offer 15. 二进制中1的个数 🟢

191. 位 1 的个数

LeetCode	力扣	难度
191. Number of 1 Bits	191. 位1的个数	

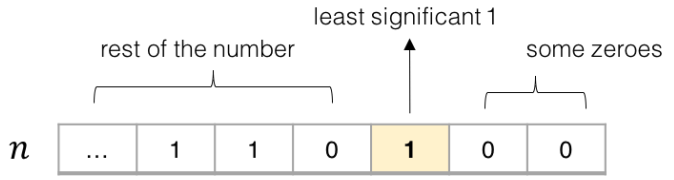
Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

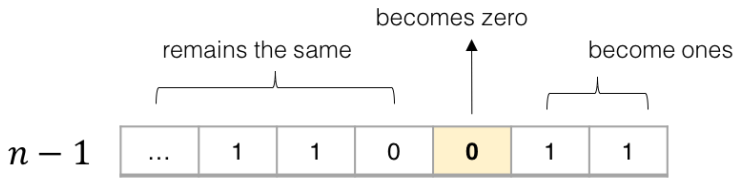
- **标签：位运算，数学**

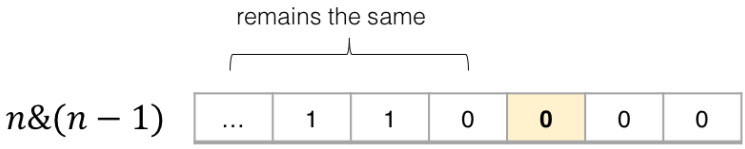
编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 1 的个数（也被称为汉明重量）。

基本思路

`n & (n-1)` 这个操作是算法中常见的，作用是消除数字 `n` 的二进制表示中的最后一个 1：







不断消除数字 `n` 中的 1，直到 `n` 变为 0。

- **详细题解：常用的位操作**

解法代码

```
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int res = 0;
        while (n != 0) {
```

```
            n = n & (n - 1);
            res++;
        }
        return res;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 136. 只出现一次的数字 🟢
  - 231. 2 的幂 🟢
  - 268. 丢失的数字 🟢
  - 剑指 Offer 15. 二进制中1的个数 🟢

231. 2 的幂

LeetCode	力扣	难度
231. Power of Two	231. 2 的幂	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：位运算，数学**

给你一个整数 `n`，请你判断该整数是否是 2 的指数。如果是，返回 `true`；否则返回 `false`。

示例 1：

```
输入：n = 16
输出：true
解释：2^4 = 1
```

基本思路

一个数如果是 2 的指数，那么它的二进制表示一定只含有一个 1。

位运算 `n&(n-1)` 在算法中挺常见的，作用是消除数字 `n` 的二进制表示中的最后一个 1，用这个技巧可以判断 2 的指数。

- **详细题解：常用的位操作**

解法代码

```
class Solution {
    public boolean isPowerOfTwo(int n) {
        if (n <= 0) return false;
        return (n & (n - 1)) == 0;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目：**
  - 136. 只出现一次的数字 🟢
  - 191. 位1的个数 🟢
  - 268. 丢失的数字 🟢
  - 剑指 Offer 15. 二进制中1的个数 🟢

268. 丢失的数字

LeetCode	力扣	难度
268. Missing Number	268. 丢失的数字	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：位运算，数学

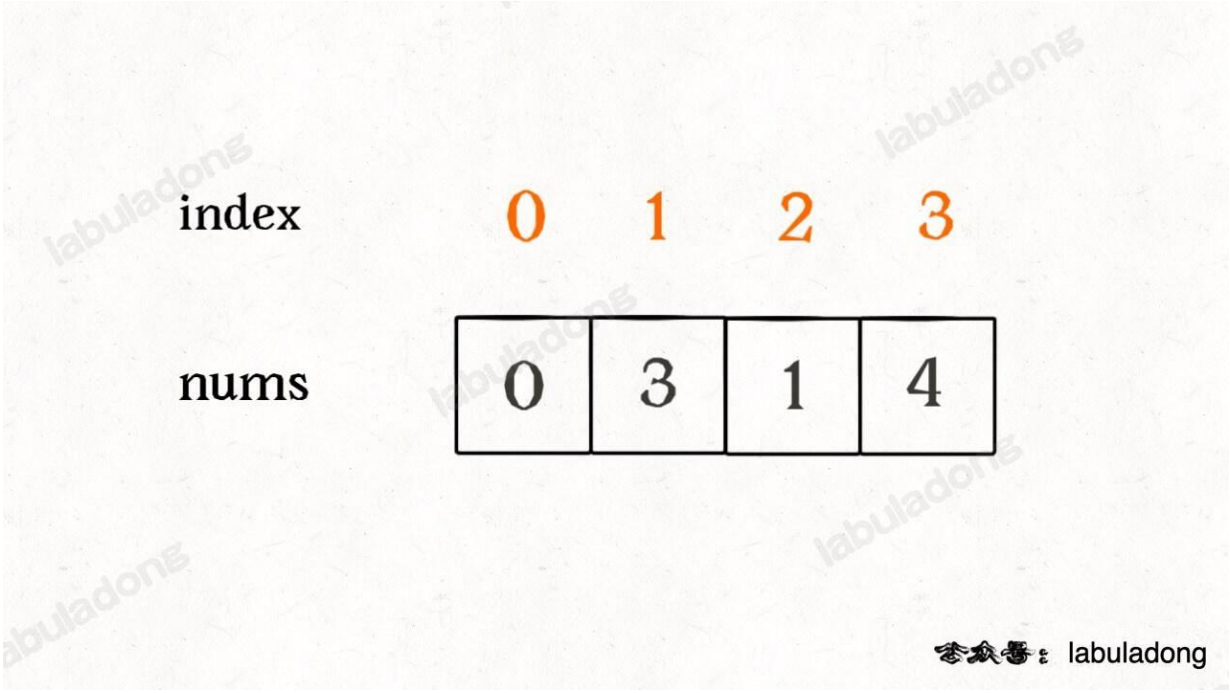
给定一个包含 [0, n] 中 n 个数的数组 nums，找出 [0, n] 这个范围内没有出现在数组中的那个数。

示例 1：

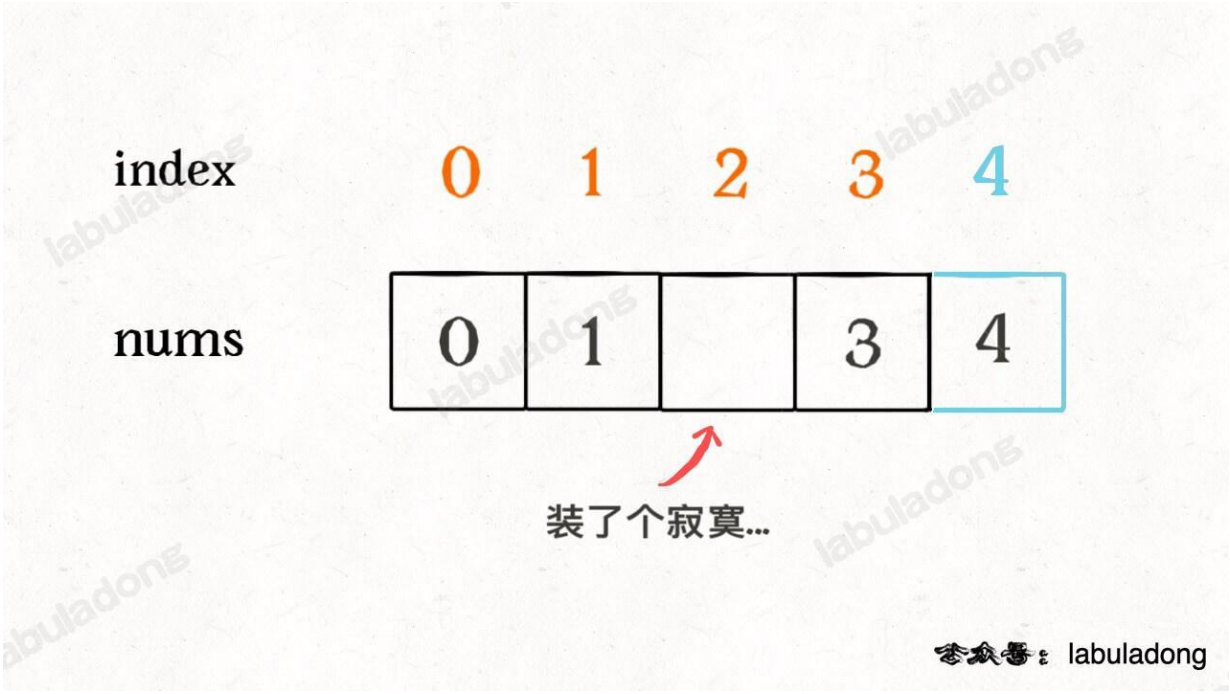
```
输入：nums = [3,0,1]
输出：2
解释：n = 3，因为有 3 个数字，所以所有的数字都在范围 [0,3] 内。2 是丢失的数字，因为它没有出现在 nums 中。
```

基本思路

假设 nums = [0,3,1,4]：



我们先把索引补一位，然后让每个元素和自己相等的索引相对应：



这样做了之后，就可以发现除了缺失元素之外，所有的索引和元素都组成一对儿了，现在如果把这个落单的索引 2 找出来，也就找到了缺失的那个元素。

如何找？只要把所有的元素和索引做异或运算，成对儿的数字都会消为 0，只有这个落单的元素会剩下。

- **详细题解：常用的位操作**

解法代码

```
class Solution {
    public int missingNumber(int[] nums) {
        int n = nums.length;
        int res = 0;
        // 先和新补的索引异或一下
        res ^= n;
        // 和其他的元素、索引做异或
        for (int i = 0; i < n; i++)
            res ^= i ^ nums[i];
        return res;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目：**
  - 136. 只出现一次的数字 🟢
  - 191. 位1的个数 🟢
  - 231. 2 的幂 🟢
  - 剑指 Offer 15. 二进制中1的个数 🟢

剑指 Offer 15. 二进制中1的个数

这道题和 191. 位 1 的个数 相同。

172. 阶乘后的零

LeetCode	力扣	难度
172. Factorial Trailing Zeroes	172. 阶乘后的零	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

给定一个整数 `n`，返回 `n!` 结果中尾随零的数量。

提示：`n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1`

示例 1：

```
输入：n = 3
输出：0
解释：3! = 6，不含尾随 0
```

基本思路

首先，两个数相乘结果末尾有 0，一定是因为两个数中有因子 2 和 5，也就是说，问题转化为：`n!` 最多可以分解出多少个因子 2 和 5？

最多可以分解出多少个因子 2 和 5，主要取决于能分解出几个因子 5，因为每个偶数都能分解出因子 2，因子 2 肯定比因子 5 多得多。

那么，问题转化为：`n!` 最多可以分解出多少个因子 5？难点在于像 25，50，125 这样的数，可以提供不止一个因子 5，不能漏数了。

这样，我们假设 `n = 125`，来算一算 `125!` 的结果末尾有几个 0：

首先，125 / 5 = 25，这一步就是计算有多少个像 5，15，20，25 这些 5 的倍数，它们一定可以提供一个因子 5。

但是，这些足够吗？刚才说了，像 25，50，75 这些 25 的倍数，可以提供两个因子 5，那么我们再计算出 `125!` 中有 125 / 25 = 5 个 25 的倍数，它们每人可以额外再提供一个因子 5。

够了吗？我们发现 125 = 5 x 5 x 5，像 125，250 这些 125 的倍数，可以提供 3 个因子 5，那么我们还得再计算出 `125!` 中有 125 / 125 = 1 个 125 的倍数，它还可以额外再提供一个因子 5。

这下应该够了，`125!` 最多可以分解出 25 + 5 + 1 = 31 个因子 5，也就是说阶乘结果的末尾有 31 个 0。

- **详细题解：讲两道常考的阶乘算法题**

解法代码

```
class Solution {
    public int trailingZeroes(int n) {
        int res = 0;
        long divisor = 5;
        while (divisor <= n) {
            res += n / divisor;
            divisor *= 5;
        }
        return res;
    }
}
```

▶ 🦇 代码可视化动画（需安装 Chrome 插件）🦇

- **类似题目**：
  - 793. 阶乘函数后 K 个零 🔴

793. 阶乘函数后 K 个零

LeetCode	力扣	难度
793. Preimage Size of Factorial Zeroes Function	793. 阶乘函数后 K 个零	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二分搜索，数学**

`f(x)` 是 `x!` 末尾是 0 的数量，`x! = 1 * 2 * 3 * ... * x`，且 `0! = 1`。

例如，`f(3) = 0`，因为 3! = 6 的末尾没有 0；而 `f(11) = 2`，因为 11! = 39916800 末端有 2 个 0。给定 `K`，找出多少个非负整数 `x`，能满足 `f(x) = K`。

示例 1：

```
输入：K = 0
输出：5
解释：0!, 1!, 2!, 3!, and 4! 均符合 K = 0 的条件。
```

基本思路

这题需要复用 172. 阶乘后的零 这道题的解法函数 `trailingZeroes`。

搜索有多少个 `n` 满足 `trailingZeroes(n) == K`，其实就是在问，满足条件的 `n` 最小是多少，最大是多少，最大值和最小值一减，就可以算出来有多少个 `n` 满足条件了，对吧？那不就是 二分查找 中「搜索左侧边界」和「搜索右侧边界」这两个事儿嘛？

观察题目给出的数据取值范围，`n` 可以在区间 `[0, LONG_MAX]` 中取值，寻找满足 `trailingZeroes(n) == K` 的左侧边界和右侧边界，相减即是答案。

- **详细题解：讲两道常考的阶乘算法题**

解法代码

```
class Solution {
    public int preimageSizeFZF(int K) {
        // 左边界和右边界之差 + 1 就是答案
        return (int)(right_bound(K) - left_bound(K) + 1);
    }

    // 逻辑不变，数据类型全部改成 long
    long trailingZeroes(long n) {
        long res = 0;
        for (long d = n; d / 5 > 0; d = d / 5) {
            res += d / 5;
```

```
        }
        return res;
    }

    /* 搜索 trailingZeroes(n) == K 的左侧边界 */
    long left_bound(int target) {
        long lo = 0, hi = Long.MAX_VALUE;
        while (lo < hi) {
            long mid = lo + (hi - lo) / 2;
            if (trailingZeroes(mid) < target) {
                lo = mid + 1;
            } else if (trailingZeroes(mid) > target) {
                hi = mid;
            } else {
                hi = mid;
            }
        }
        return lo;
    }

    /* 搜索 trailingZeroes(n) == K 的右侧边界 */
    long right_bound(int target) {
        long lo = 0, hi = Long.MAX_VALUE;
        while (lo < hi) {
            long mid = lo + (hi - lo) / 2;
            if (trailingZeroes(mid) < target) {
                lo = mid + 1;
            } else if (trailingZeroes(mid) > target) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
        }

        return lo - 1;
    }
}
```

- **类似题目**：
  - 172. 阶乘后的零 🟠

204. 计数质数

LeetCode	力扣	难度
204. Count Primes	204. 计数质数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

统计所有小于非负整数 n 的质数的数量。

示例 1：

```
输入：n = 10
输出：4
解释：小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7。
```

基本思路

PS：这道题在《算法小抄》 的第 351 页。

筛数法是常见的计算素数的算法。

因为判断一个数字是否是素数的时间成本较高，所以我们不要一个个判断每个数字是否是素数，而是用排除法，把所有非素数都排除，剩下的就是素数。

- **详细题解：如何高效寻找素数**

解法代码

```
class Solution {
    public int countPrimes(int n) {
        boolean[] isPrime = new boolean[n];
        Arrays.fill(isPrime, true);
        for (int i = 2; i * i < n; i++)
            if (isPrime[i])
                for (int j = i * i; j < n; j += i)
                    isPrime[j] = false;

        int count = 0;
        for (int i = 2; i < n; i++)
            if (isPrime[i]) count++;

        return count;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- 类似题目：
  - 264. 丑数 II 🟠
  - 剑指 Offer 49. 丑数 🟠

264. 丑数 II

LeetCode	力扣	难度
264. Ugly Number II	264. 丑数 II	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学，链表双指针**

给你一个整数 n，请你找出并返回第 n 个 **丑数**。**丑数** 就是只包含质因数 2、3 和/或 5 的正整数。

示例 1：

```
输入：n = 10
输出：12
解释：[1, 2, 3, 4, 5, 6, 8, 9, 10, 12] 是由前 10 个丑数组成的序列。
```

基本思路

这道题很精妙，你看着它好像是道数学题，实际上它却是一个合并多个有序链表的问题，同时用到了筛选素数的思路。

建议你先做一下 链表双指针技巧汇总 中讲到的 21. 合并两个有序链表（简单），然后再做一下 如何高效寻找素数 中讲的 204. 计数质数（简单），这样的话就能比较容易理解这道题的思路了。

类似 如何高效寻找素数 的思路：如果一个数 `x` 是丑数，那么 `x * 2, x * 3, x * 5` 都一定是丑数。

我们把所有丑数想象成一个从小到大排序的链表，就是这个样子：

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 8 -> ...
```

然后，我们可以把丑数分为三类：2 的倍数、3 的倍数、5 的倍数（按照题目的意思，1 算作特殊的丑数，放在开头），这三类丑数就好像三条有序链表，如下：

能被 2 整除的丑数：

```
1 -> 1*2 -> 2*2 -> 3*2 -> 4*2 -> 5*2 -> 6*2 -> 8*2 ->...
```

能被 3 整除的丑数：

```
1 -> 1*3 -> 2*3 -> 3*3 -> 4*3 -> 5*3 -> 6*3 -> 8*3 ->...
```

能被 5 整除的丑数：

```
1 -> 1*5 -> 2*5 -> 3*5 -> 4*5 -> 5*5 -> 6*5 -> 8*5 ->...
```

我们其实就是想把这三条「有序链表」合并在一起并去重，合并的结果就是丑数的序列。然后求合并后的这条有序链表中第 n 个元素是什么。所以这里就和 链表双指针技巧汇总 中讲到的合并 k 条有序链表的思路基本一样了。

具体思路看注释吧，你也可以对照我在 21. 合并两个有序链表（简单） 中给出的思路代码来看本题的思路代码，就能轻松看懂本题的解法代码了。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
class Solution {
    public int nthUglyNumber(int n) {
        // 可以理解为三个指向有序链表头结点的指针
        int p2 = 1, p3 = 1, p5 = 1;
        // 可以理解为三个有序链表的头节点的值
        int product2 = 1, product3 = 1, product5 = 1;
        // 可以理解为最终合并的有序链表（结果链表）
        int[] ugly = new int[n + 1];
        // 可以理解为结果链表上的指针
        int p = 1;

        // 开始合并三个有序链表
        while (p <= n) {
            // 取三个链表的最小结点
            int min = Math.min(Math.min(product2, product3), product5);
            // 接到结果链表上
            ugly[p] = min;
            p++;
            // 前进对应有序链表上的指针
            if (min == product2) {
                product2 = 2 * ugly[p2];
                p2++;
            }
            if (min == product3) {
                product3 = 3 * ugly[p3];
                p3++;
            }
            if (min == product5) {
                product5 = 5 * ugly[p5];
                p5++;
            }
        }
        // 返回第 n 个丑数
        return ugly[n];
```

```
        }
    }
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 1201. 丑数 III 🟠
  - 263. 丑数 🟢
  - 313. 超级丑数 🟠
  - 剑指 Offer 49. 丑数 🟠

剑指 Offer 49. 丑数

这道题和 264. 丑数 II 相同。

1201. 丑数 III

LeetCode	力扣	难度
1201. Ugly Number III	1201. 丑数 III	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：二分搜索，数学，链表双指针

给你四个整数：`n` 、`a` 、`b` 、`c`，请你设计一个算法来找出第 `n` 个丑数。丑数是可以被 `a` **或** `b` **或** `c` 整除的 **正整数**。

示例 1：

```
输入：n = 3, a = 2, b = 3, c = 5
输出：4
解释：丑数序列为 2, 3, 4, 5, 6, 8, 9, 10... 其中第 3 个是 4。
```

基本思路

这道题和 264. 丑数 II 有些类似，你把第 264 题合并有序链表的解法照搬过来稍微改改就能解决这道题，代码我写在 `Solution2` 里面了。

但是注意题目给的数据规模，`a, b, c, n` 都是非常大的数字，如果用合并有序链表的思路，其复杂度是 O(N)，对于这么大的数据规模来说也是比较慢的，应该会超时，无法通过一些测试用例。

这道题的正确解法难度比较大，难点在于你要把一些数学知识和 二分搜索技巧 结合起来才能高效解决这个问题。

首先，根据 二分查找的实际运用 中讲到的二分搜索运用方法，我们可以抽象出一个单调递增的函数 `f`：

`f(num, a, b, c)` 计算 `[1..num]` 中，能够整除 `a` 或 `b` 或 `c` 的数字的个数，显然函数 `f` 的返回值是随着 `num` 的增加而增加的（单调递增）。

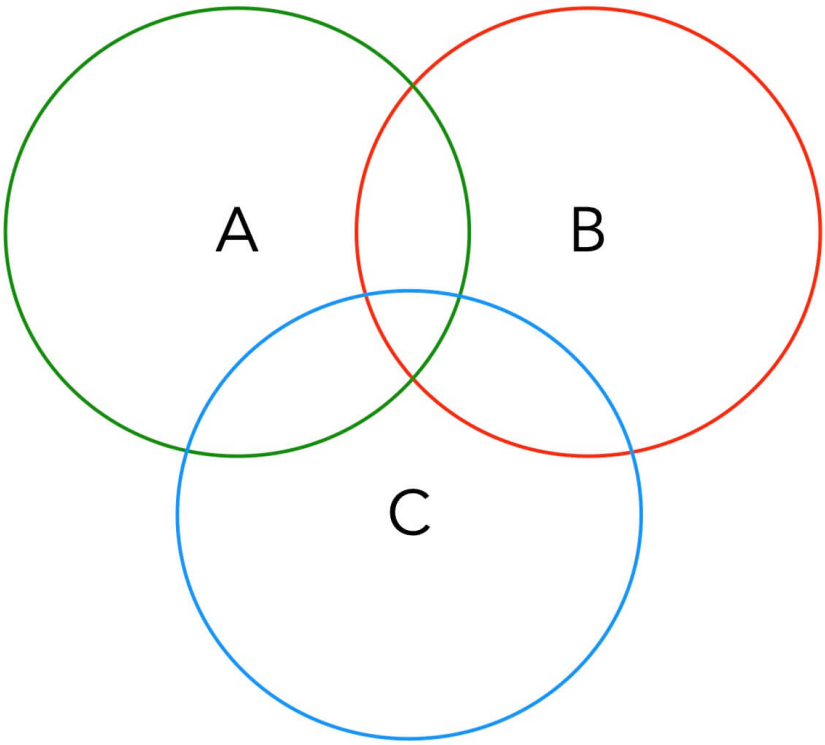
题目让我们求第 `n` 个能够整除 `a` 或 `b` 或 `c` 的数字是什么，也就是说我们要找到一个 `num`，使得 `f(num, a, b, c) == n`。

有了上述思路，就可以按照 二分查找的实际运用 中给出的模板运用二分搜索算法了。

关键说一下函数 `f` 怎么实现，这里面涉及集合论定理以及最小公因数、最小公倍数的计算方法。

首先，`[1..num]` 中，我把能够整除 `a` 的数字归为集合 `A`，能够整除 `b` 的数字归为集合 `B`，能够整除 `c` 的数字归为集合 `C`，那么 `len(A) = num / a, len(B) = num / b, len(C) = num / c`，这个很好理解。

但是 `f(num, a, b, c)` 的值肯定不是 `num / a + num / b + num / c` 这么简单，因为你注意有些数字可能可以被 `a, b, c` 中的两个数或三个数同时整除，如下图：



按照集合论的算法，这个集合中的元素应该是：A + B + C - A ∩ B - A ∩ C - B ∩ C + A ∩ B ∩ C。结合上图应该很好理解。

问题来了，`A, B, C` 三个集合的元素个数我们已经算出来了，但如何计算像 `A ∩ B` 这种交集的元素个数呢？

其实也很容易想明白，`A ∩ B` 的元素个数就是 `n / lcm(a, b)`，其中 `lcm` 是计算最小公倍数（Least Common Multiple）的函数。

类似的，`A ∩ B ∩ C` 的元素个数就是 `n / lcm(lcm(a, b), c)` 的值。

现在的问题是，最小公倍数怎么求？

直接记住定理吧：`lcm(a, b) = a * b / gcd(a, b)`，其中 `gcd` 是计算最大公因数（Greatest Common Divisor）的函数。

现在的问题是，最大公因数怎么求？

这应该是经典算法了，我们一般叫辗转相除算法（或者欧几里得算法）。

好了，套娃终于套完了，我们可以把上述思路翻译成解法，注意本题数据规模比较大，有时候需要用 `long` 类型防止 `int` 溢出，具体看我的代码实现以及注释吧。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
// 二分搜索 + 数学解法
class Solution {
    public int nthUglyNumber(int n, int a, int b, int c) {
```

```
        // 题目说本题结果在 [1, 2 * 10^9] 范围内，
        // 所以就按照这个范围初始化两端都闭的搜索区间
        int left = 1, right = (int) 2e9;
        // 搜索左侧边界的二分搜索
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (f(mid, a, b, c) < n) {
                // [1..mid] 中的元素个数不足 n，所以目标在右侧
                left = mid + 1;
            } else {
                // [1..mid] 中的元素个数大于 n，所以目标在左侧
                right = mid - 1;
            }
        }
        return left;
    }

    // 计算 [1..num] 之间有多少个能够被 a 或 b 或 c 整除的数字
    long f(int num, int a, int b, int c) {
        long setA = num / a, setB = num / b, setC = num / c;
        long setAB = num / lcm(a, b);
        long setAC = num / lcm(a, c);
        long setBC = num / lcm(b, c);
        long setABC = num / lcm(lcm(a, b), c);
        // 集合论定理：A + B + C - A ∩ B - A ∩ C - B ∩ C + A ∩ B ∩ C
        return setA + setB + setC - setAB - setAC - setBC + setABC;
    }

    // 计算最大公因数（辗转相除/欧几里得算法）
    long gcd(long a, long b) {
        if (a < b) {
            // 保证 a > b
            return gcd(b, a);
        }
        if (b == 0) {
            return a;
        }
        return gcd(b, a % b);
    }

    // 最小公倍数
    long lcm(long a, long b) {
        // 最小公倍数就是乘积除以最大公因数
        return a * b / gcd(a, b);
    }
}

// 用合并单链表的思路（超时）
class Solution2 {
    public int nthUglyNumber(int n, int a, int b, int c) {
        // 可以理解为三个有序链表的头结点的值
        long productA = a, productB = b, productC = c;
        // 可以理解为合并之后的有序链表上的指针
        int p = 1;
```

```
        long min = -666;

        // 开始合并三个有序链表，获取第 n 个节点的值
        while (p <= n) {
            // 取三个链表的最小结点
            min = Math.min(Math.min(productA, productB), productC);
            p++;
            // 前进最小结点对应链表的指针
            if (min == productA) {
                productA += a;
            }
            if (min == productB) {
                productB += b;
            }
            if (min == productC) {
                productC += c;
            }
        }
        return (int) min;
    }
}
```

▶ 🍭 代码可视化动画（需安装 Chrome 插件）🍭

- **类似题目**：
  - 263. 丑数 🟢
  - 264. 丑数 II 🟠
  - 313. 超级丑数 🟠
  - 剑指 Offer 49. 丑数 🟠

263. 丑数

LeetCode	力扣	难度
263. Ugly Number	263. 丑数	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

丑数 就是只包含质因数 2、3 和 5 的正整数。给你一个整数 n，请你判断 n 是否为 **丑数**。如果是，返回 true；否则，返回 false。

示例 1：

```
输入：n = 6
输出：true
解释：6 = 2 × 3
```

基本思路

这道题其实很简单，主要考察你算术基本定理（正整数唯一分解定理），即：**任意一个大于 1 的自然数，要么它本身就是质数，要么它可以分解为若干质数的乘积。**

那么题目所说的丑数当然也可以被分解成若干质数的乘积，且这些质数只包括 2, 3, 5。那么解题思路就很显然了，只要判断 n 是不是只有这三种质因子即可。

- 详细题解：一文秒杀所有丑数系列问题

解法代码

```
class Solution {
    public boolean isUgly(int n) {
        if (n <= 0) return false;
        // 如果 n 是丑数，分解因子应该只有 2, 3, 5
        while (n % 2 == 0) n /= 2;
        while (n % 3 == 0) n /= 3;
        while (n % 5 == 0) n /= 5;
        return n == 1;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- 类似题目：

- 1201. 丑数 III 🟠
- 264. 丑数 II 🟠
- 313. 超级丑数 🟠
- 剑指 Offer 49. 丑数 🟠

313. 超级丑数

LeetCode	力扣	难度
313. Super Ugly Number	313. 超级丑数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，链表双指针**

超级丑数 是一个正整数，并满足其所有质因数都出现在质数数组 `primes` 中。给你一个整数 `n` 和一个整数数组 `primes`，返回第 `n` 个 **超级丑数**。

示例 1：

```
输入：n = 12, primes = [2,7,13,19]
输出：32
解释：给定长度为 4 的质数数组 primes = [2,7,13,19]，前 12 个超级丑数序列为：
[1,2,4,7,8,13,14,16,19,26,28,32]。
```

基本思路

这题是 264. 丑数 II 的进阶版，第 264 题是 21. 合并两个有序链表（简单） 的变体，而这道题是 23. 合并K个升序链表（困难） 的变体，我在 双指针技巧秒杀七道链表题目 中都讲过。

你一定要先做下 264 题，注意那里我们抽象出了三条链表，需要 `p2, p3, p5` 作为三条有序链表上的指针，同时需要 `product2, product3, product5` 记录指针所指节点的值，用 `min` 函数计算最小头结点。

这道题相当于输入了 `len(primes)` 条有序链表，我们不能用 `min` 函数计算最小头结点了，而是要用优先级队列来计算最小头结点，同时依然要维护链表指针、指针所指节点的值，我们把这些信息用一个三元组来保存。

结合第 23 题的解法逻辑，你应该能够看懂这道题的解法代码了。

- **详细题解：一文秒杀所有丑数系列问题**

解法代码

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        // 优先队列中装三元组 int[] {product, prime, pi}
        // 其中 product 代表链表节点的值，prime 是计算下一个节点所需的质数因子，pi
代表链表上的指针
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            return a[0] - b[0];
        });
```

```
        // 把多条链表的头结点加入优先级队列
        for (int i = 0; i < primes.length; i++) {
            pq.offer(new int[]{ 1, primes[i], 1 });
        }

        // 可以理解为最终合并的有序链表（结果链表）
        int[] ugly = new int[n + 1];
        // 可以理解为结果链表上的指针
        int p = 1;

        while (p <= n) {
            // 取三个链表的最小结点
            int[] pair = pq.poll();
            int product = pair[0];
            int prime = pair[1];
            int index = pair[2];

            // 避免结果链表出现重复元素
            if (product != ugly[p - 1]) {
                // 接到结果链表上
                ugly[p] = product;
                p++;
            }

            // 生成下一个节点加入优先级队列
            int[] nextPair = new int[]{ugly[index] * prime, prime, index +
1};
            pq.offer(nextPair);
        }
        return ugly[n];
    }
}
```

- **类似题目**：
  - 1201. 丑数 III 🟠
  - 263. 丑数 🟢
  - 264. 丑数 II 🟠
  - 剑指 Offer 49. 丑数 🟠

279. 完全平方数

LeetCode	力扣	难度
279. Perfect Squares	279. 完全平方数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：动态规划，数学

给你一个整数 n，返回**和为 n 的完全平方数的最少数量**。**完全平方数**是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

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

基本思路

这道题有一些比较有技巧性的数学方法，我这里不探讨数学，就用通用的动态规划思路解决：

题目问和为 n 的平方数的最小数量，那么我可以根据和为 n-1x1, n-2x2, n-3x3... 的平方数的最小数量推导出来。

如果这个思路你绕不过来，我再给你变化下就更容易理解了，这个问题完全可以变化成 动态规划详解 中讲的 322. 零钱兑换 问题：

给你一个目标金额 n，和一个若干硬币的面额 coins = 1,4,9,16...，问你最少需要几枚硬币凑出这个金额（因为包含面值为 1 的硬币，所以不存在凑不出来的情况）。

这样变个说法理解题目，是不是就豁然开朗了？具体解法看代码吧，复杂度是 O(N*sqrt(N))，也是不错的。

解法代码

```
class Solution {
    public int numSquares(int n) {
        // 定义：和为 i 的平方数的最小数量是 dp[i]
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        // base case
        dp[0] = 0;
        // 状态转移方程
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j * j <= i; j++) {
```

```
                // i - j * j 只要再加一个平方数 j * j 即可凑出 i
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }
        return dp[n];
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

292. Nim 游戏

LeetCode	力扣	难度
292. Nim Game	292. Nim 游戏	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

你和你的朋友，两个人一起玩 Nim 游戏：

桌子上有一堆石头，你们轮流进行自己的回合，你作为先手；每一回合，轮到的人拿掉 1 ~ 3 块石头，拿掉最后一块石头的人就是获胜者。

假设你们每一步都是最优解，请编写一个函数，来判断你是否可以在给定石头数量为 `n` 的情况下赢得游戏。如果可以赢，返回 `true`；否则，返回 `false`。

示例 1：

```
输入：n = 4
输出：false
解释：如果堆中有 4 块石头，那么你永远不会赢得比赛；
      因为无论你拿走 1 块、2 块 还是 3 块石头，最后一块石头总是会被你的朋友拿走。
```

基本思路

PS：这道题在《算法小抄》 的第 414 页。

我们解决这种问题的思路一般都是反着思考：

如果我能赢，那么最后轮到我取石子的时候必须要剩下 1~3 颗石子，这样我才能一把拿完。

如何营造这样的一个局面呢？显然，如果对手拿的时候只剩 4 颗石子，那么无论他怎么拿，总会剩下 1~3 颗石子，我就能赢。

如何逼迫对手面对 4 颗石子呢？要想办法，让我选择的时候还有 5~7 颗石子，这样的话我就有把握让对方不得不面对 4 颗石子。

如何营造 5~7 颗石子的局面呢？让对手面对 8 颗石子，无论他怎么拿，都会给我剩下 5~7 颗，我就能赢。

这样一直循环下去，我们发现只要踩到 4 的倍数，就落入了圈套，永远逃不出 4 的倍数，而且一定会输。

- **详细题解：一行代码就能解决的算法题**

解法代码

```
class Solution {
    public boolean canWinNim(int n) {
        // 如果上来就踩到 4 的倍数，那就认输吧
        // 否则，可以把对方控制在 4 的倍数，必胜
        return n % 4 != 0;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 319. 灯泡开关 🟠
  - 877. 石子游戏 🟠

319. 灯泡开关

LeetCode	力扣	难度
319. Bulb Switcher	319. 灯泡开关	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

初始时有 `n` 个灯泡处于关闭状态。

对某个灯泡切换开关意味着：如果灯泡状态为关闭，那该灯泡就会被开启；而灯泡状态为开启，那该灯泡就会被关闭。

第 1 轮，每个灯泡切换一次开关，即打开所有的灯泡。

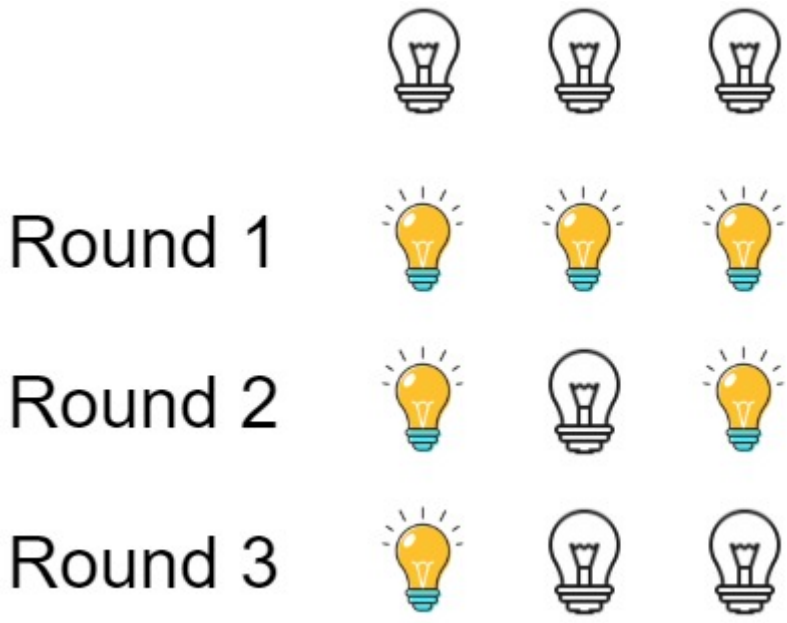
第 2 轮，每两个灯泡切换一次开关。即每两个灯泡关闭一个。

第 3 轮，每三个灯泡切换一次开关，以此类推……

第 `i` 轮，每 `i` 个灯泡切换一次开关，而第 `n` 轮你只切换最后一个灯泡的开关。

计算 `n` 轮后有多少个亮着的灯泡。

示例 1：




```
输入：n = 3
输出：1
解释：
初始时，灯泡状态 [关闭，关闭，关闭].
第一轮后，灯泡状态 [开启，开启，开启].
```

第二轮后，灯泡状态 [开启，关闭，开启].
第三轮后，灯泡状态 [开启，关闭，关闭].

你应该返回 1，因为只有一个灯泡还亮着。

基本思路

PS：这道题在《算法小抄》 的第 414 页。

因为电灯一开始都是关闭的，所以某一盏灯最后如果是点亮的，必然要被按奇数次开关。

我们假设只有 16 盏灯，对于第十六盏灯会被按几次？

被按的次数就是 16 不同因子的个数，因为 `16 = 1*16 = 2*8 = 4*4`，其中因子 4 重复出现，所以第 16 盏灯会被按 5 次，奇数次。

一个正整数 `n` 的不同因子有几个？就是 `n` 的平方根向下取整，也就是这个问题的答案。

- **详细题解：一行代码就能解决的算法题**

解法代码

```
class Solution {
    public int bulbSwitch(int n) {
        return (int)Math.sqrt(n);
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 292. Nim 游戏 🟢
  - 877. 石子游戏 🟠

877. 石子游戏

LeetCode	力扣	难度
877. Stone Game	877. 石子游戏	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

甲和乙用几堆石子在做游戏。**偶数堆石子**排成一行，每堆都有正整数颗石子 `piles[i]`。

从甲先开始，玩家轮流从这行石子的开头或末尾处取走整堆石头，直到没有更多的石子堆为止，此时手中石子最多的玩家获胜。**石子的总数是奇数**，所以没有平局。

假设甲和乙都发挥出最佳水平，当甲赢得比赛时返回 `true`，当乙赢得比赛时返回 `false`。

示例：

```
输入：[5,3,4,5]
输出：true
解释：
甲先开始，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]。
如果乙拿走前 3 颗，那么剩下的是 [4,5]，甲拿走后 5 颗赢得 10 分。
如果乙拿走后 5 颗，那么剩下的是 [3,4]，甲拿走后 4 颗赢得 9 分。
这表明，取前 5 颗石子对甲来说是一个胜利的举动，所以我们返回 true。
```

基本思路

PS：这道题在《算法小抄》 的第 414 页。

这个条件下先手必胜。

如果我们把这四堆石头按索引的奇偶分为两组，即第 1、3 堆和第 2、4 堆，那么这两组石头的数量一定不同，也就是说一堆多一堆少。因为石头的总数是奇数，不能被平分。

而作为第一个拿石头的人，你可以控制自己拿到所有偶数堆，或者所有的奇数堆。

你最开始可以选择第 1 堆或第 4 堆。如果你想要偶数堆，你就拿第 4 堆，这样留给对手的选择只有第 1、3 堆，他不管怎么拿，第 2 堆又会暴露出来，你就可以拿。同理，如果你想拿奇数堆，你就拿第 1 堆，留给对手的只有第 2、4 堆，他不管怎么拿，第 3 堆又给你暴露出来了。

也就是说，你可以在第一步就观察好，奇数堆的石头总数多，还是偶数堆的石头总数多，然后步步为营，就一切尽在掌控之中了。知道了这个漏洞，可以整一整不知情的同学了。

当然，「总共有偶数堆石子」和「石子总数为奇数」是先手必胜的前提条件，如果题目更具一般性，没有这两个条件，就属于标准的博弈问题，应该使用动态规划算法来解决了，详见 动态规划之博弈问题。

- **详细题解：一行代码就能解决的算法题**

解法代码

```
class Solution {
    public boolean stoneGame(int[] piles) {
        return true;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目：**
  - 292. Nim 游戏 🟢
  - 319. 灯泡开关 🟠

295. 数据流的中位数

LeetCode	力扣	难度
295. Find Median from Data Stream	295. 数据流的中位数	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，数学**

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

设计一个支持以下两种操作的数据结构：

1、`void addNum(int num)` 从数据流中添加一个整数到数据结构中。

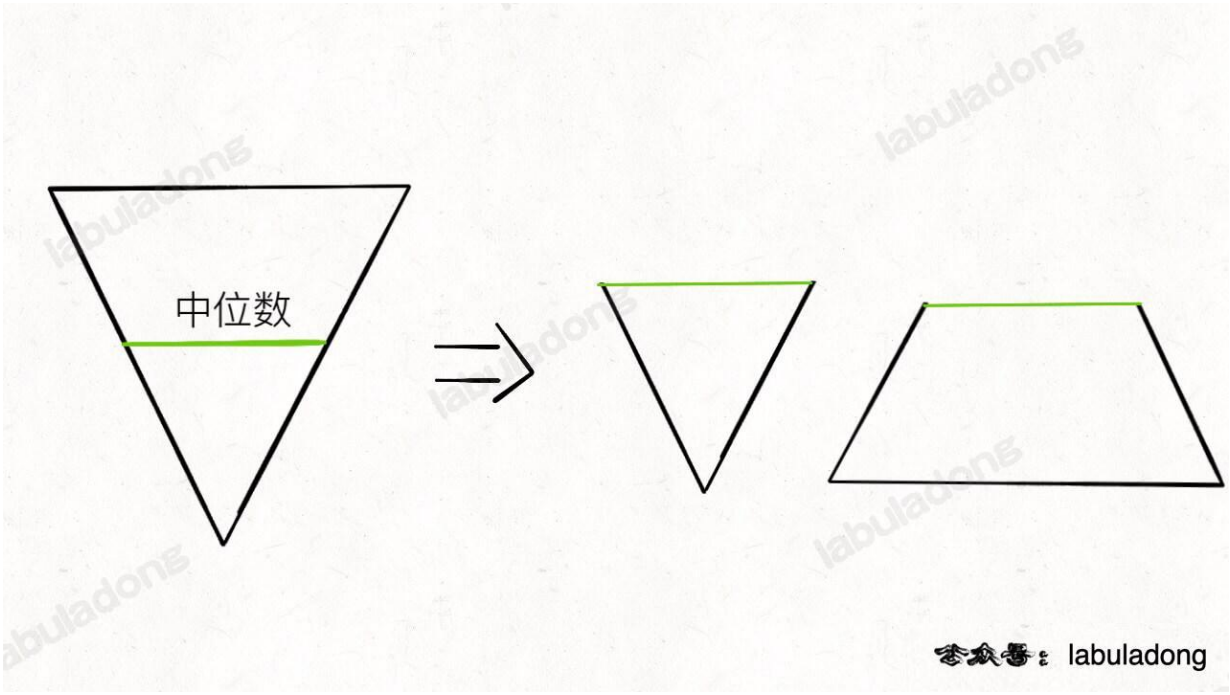
2、`double findMedian()` 返回目前所有元素的中位数。

示例：

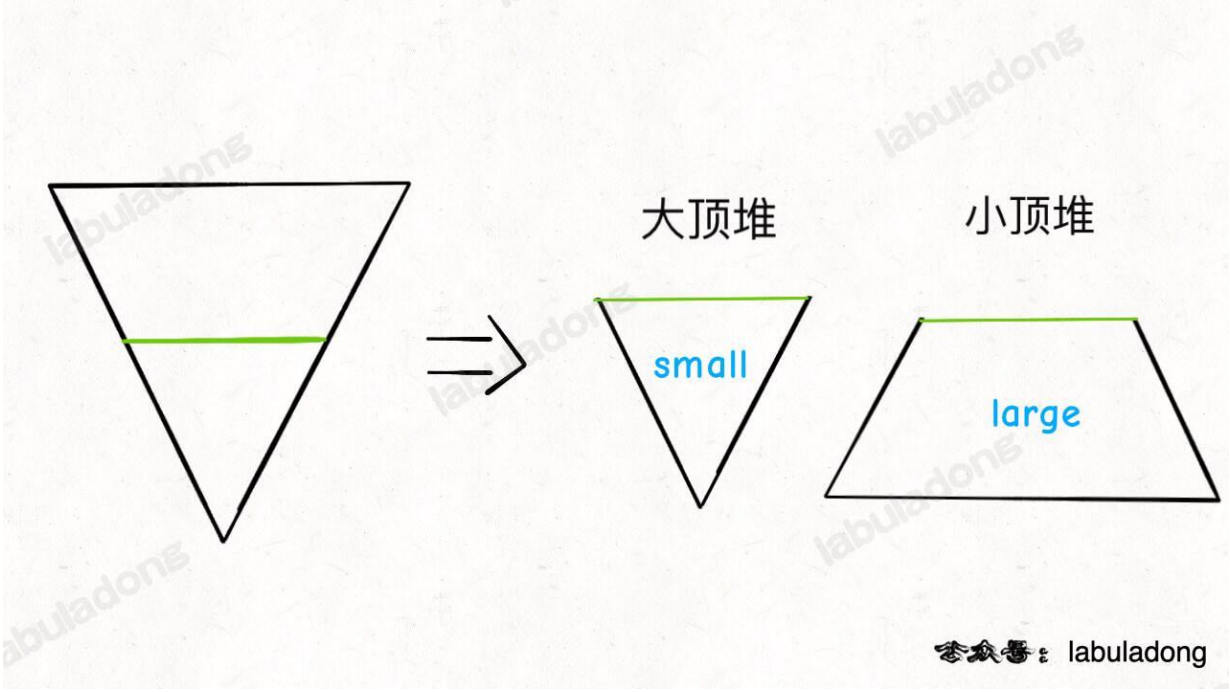
```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

基本思路

本题的核心思路是使用两个优先级队列。



小的倒三角就是个大顶堆，梯形就是个小顶堆，中位数可以通过它们的堆顶元素算出来：



- **详细题解：一道求中位数的算法题把我整不会了**

解法代码

```
class MedianFinder {
    private PriorityQueue<Integer> large;
    private PriorityQueue<Integer> small;

    public MedianFinder() {
        // 小顶堆
        large = new PriorityQueue<>();
        // 大顶堆
        small = new PriorityQueue<>((a, b) -> {
            return b - a;
        });
    }

    public double findMedian() {
        // 如果元素不一样多，多的那个堆的堆顶元素就是中位数
        if (large.size() < small.size()) {
            return small.peek();
        } else if (large.size() > small.size()) {
            return large.peek();
        }
        // 如果元素一样多，两个堆堆顶元素的平均数是中位数
        return (large.peek() + small.peek()) / 2.0;
    }

    public void addNum(int num) {
        if (small.size() >= large.size()) {
            small.offer(num);
```

```
                large.offer(small.poll());
            } else {
                large.offer(num);
                small.offer(large.poll());
            }
        }
    }
```

- **类似题目**：
    - 剑指 Offer 41. 数据流中的中位数 🔴

剑指 Offer 41. 数据流中的中位数

这道题和 295. 数据流的中位数 相同。

306. 累加数

LeetCode	力扣	难度
306. Additive Number	306. 累加数	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学，穷举

累加数 是一个字符串，组成它的数字可以形成累加序列。

一个有效的 **累加序列** 必须 **至少** 包含 3 个数。除了最开始的两个数以外，序列中的每个后续数字必须是它之前两个数字之和。

给你一个只包含数字 `'0'-'9'` 的字符串，编写一个算法来判断给定输入是否是**累加数**。如果是，返回 `true`；否则，返回 `false`。

说明：累加序列里的数，除数字 0 之外，**不会** 以 0 开头，所以不会出现 `1, 2, 03` 或者 `1, 02, 3` 的情况。

示例 1：

```
输入："112358"
输出：true
解释：累加序列为：1, 1, 2, 3, 5, 8。1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
```

示例 2：

```
输入："199100199"
输出：true
解释：累加序列为：1, 99, 100, 199。1 + 99 = 100, 99 + 100 = 199
```

基本思路

这道题不算难，要先观察，由于这个「累加」关系是传导的，所以只要我们确定了第一个和第二个数字，后面的数字其实就确定了。

比如说，如果我们确定了第一个数字是 1，第二个数字是 2，那么后面的数字就一定是 3, 5, 8, 13,...

基于这个特性，我们用一个嵌套 for 循环就可以穷举出前两个数字的所有可能了，然后实现一个 `isValid` 函数来验证后面的数字是否满足累加数的性质即可。

解法代码

```
class Solution {
    public boolean isAdditiveNumber(String num) {
        // 穷举前两个数字
        int n = num.length();
        for (int i = 1; i <= n; i++) {
            for (int j = i + 1; j <= n; j++) {
                String first = num.substring(0, i);
                String second = num.substring(i, j);
                if (isValid(num, first, second)) {
                    return true;
                }
            }
        }
        return false;
    }

    // 定义：num 前两个数字分别是 first 和 second，判断 num 是否满足累加数的性质
    boolean isValid(String num, String first, String second) {
        if (first.startsWith("0") && first.length() > 1
                || second.startsWith("0") && second.length() > 1) {
            // 0 开头的数字，只能是 0 本身
            return false;
        }
        String sumStr = strAdd(first, second);
        String next = num.substring(first.length() + second.length());
        if (!next.startsWith(sumStr)) {
            // 不满足累加数的性质
            return false;
        }
        if (next.equals(sumStr)) {
            // 已经匹配完整个字符串
            return true;
        }
        // 根据递归函数的定义，继续匹配后面的三个数字，我这里用递归的方式去比较，因为容易写
        // 你也可以改用迭代写法，一样的
        return isValid(num.substring(first.length()), second, sumStr);
    }

    // 模拟加法竖式运算，具体可以看下这道题
    // https://leetcode.cn/problems/add-strings/
    String strAdd(String a, String b) {
        int n = a.length(), m = b.length();
        int i = n - 1, j = m - 1, add = 0;
        StringBuilder builder = new StringBuilder();
        while (i >= 0 || j >= 0 || add != 0) {
            int x = i >= 0 ? a.charAt(i) - '0' : 0;
            int y = j >= 0 ? b.charAt(j) - '0' : 0;
            int result = x + y + add;
            builder.append(result % 10);
            add = result / 10;
            i--;
            j--;
```

```
            }
            return builder.reverse().toString();
        }
    }
```

372. 超级次方

LeetCode	力扣	难度
372. Super Pow	372. 超级次方	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

你的任务是计算 a^b 对 1337 取模，a 是一个正整数，b 是一个非常大的正整数且会以数组形式给出。

示例 1：

```
输入：a = 2, b = [3]
输出：8
```

基本思路

PS：这道题在《算法小抄》 的第 355 页。

利用指数的性质，显然：

$$
\begin{aligned}
& a^{[1,5,6,4]} \\
=& a^4 \times a^{[1,5,6,0]} \\
=& a^4 \times \left(a^{[1,5,6]}\right)^{10}
\end{aligned}
$$

我们的老读者肯定已经敏感地意识到了，这就是递归的标志，因为问题的规模缩小了：

```
    superPow(a, [1,5,6,4])
=>  superPow(a, [1,5,6])
```

把上述逻辑翻译成代码即可。

由于结果很大，题目要求求模，那么关于求模运算，这里有必要强调一个推论：

```
(a * b) % k = (a % k)(b % k) % k
```

也就是说，对乘法的结果求模，等价于先对每个因子都求模，然后对因子相乘的结果再求模。证明见详细题解。

- **详细题解：如何高效进行模幂运算**

解法代码

```
class Solution {
    public:

    int base = 1337;

    // 计算 a 的 k 次方然后与 base 求模的结果
    int mypow(int a, int k) {
        // 对因子求模
        a %= base;
        int res = 1;
        for (int _ = 0; _ < k; _++) {
            // 这里有乘法，是潜在的溢出点
            res *= a;
            // 对乘法结果求模
            res %= base;
        }
        return res;
    }

    int superPow(int a, vector<int>& b) {
        if (b.empty()) return 1;
        int last = b.back();
        b.pop_back();

        int part1 = mypow(a, last);
        int part2 = mypow(superPow(a, b), 10);
        // 每次乘法都要求模
        return (part1 * part2) % base;
    }
};
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

382. 链表随机节点

LeetCode	力扣	难度
382. Linked List Random Node	382. 链表随机节点	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学，水塘抽样算法，随机算法**

给定一个单链表，随机选择链表的一个节点，并返回相应的节点值。保证每个节点**被选的概率一样**。

进阶：如果链表十分大且长度未知，如何解决这个问题？你能否使用常数级空间复杂度实现？

示例：

```
// 初始化一个单链表 [1,2,3].
ListNode head = new ListNode(1);
head.next = new ListNode(2);
head.next.next = new ListNode(3);
Solution solution = new Solution(head);

// getRandom() 方法应随机返回 1,2,3 中的一个，保证每个元素被返回的概率相等。
solution.getRandom();
```

基本思路

这题属于数学题，如何在长度未知的序列（数据流）中**随机**选择一个元素出来？

结论：当你遇到第 `i` 个元素时，应该有 `1/i` 的概率选择该元素，`1 - 1/i` 的概率保持原有的选择。

证明请看详细题解。

- **详细题解：谈谈游戏中的随机算法**

解法代码

```
class Solution {

    ListNode head;
    Random r = new Random();

    public Solution(ListNode head) {
        this.head = head;
    }

    /* 返回链表中一个随机节点的值 */
    int getRandom() {
```

```
        int i = 0, res = 0;
        ListNode p = head;
        // while 循环遍历链表
        while (p != null) {
            i++;
            // 生成一个 [0, i) 之间的整数
            // 这个整数等于 0 的概率就是 1/i
            if (0 == r.nextInt(i)) {
                res = p.val;
            }
            p = p.next;
        }
        return res;
    }
}
```

- **类似题目**：
  - 384. 打乱数组 🟠
  - 398. 随机数索引 🟠

398. 随机数索引

LeetCode	力扣	难度
398. Random Pick Index	398. 随机数索引	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学，水塘抽样算法，随机算法**

给定一个可能含有重复元素的整数数组，要求随机输出给定的数字的索引，你可以假设给定的数字一定存在于数组中。

注意：数组大小可能非常大。使用太多额外空间的解决方案将不会通过测试。

示例：

```
int[] nums = new int[] {1,2,3,3,3};
Solution solution = new Solution(nums);

// pick(3) 应该返回索引 2, 3 或者 4。每个索引的返回概率应该相等。
solution.pick(3);

// pick(1) 应该返回 0。因为只有 nums[0]等于 1。
solution.pick(1);
```

基本思路

这题按理说可以使用 HashMap 来做，存储元素到索引列表的映射，然后随机从列表中取出一个元素，但是似乎这题对空间复杂度的要求较高，这个简单直接的方式会超过内存限制。

所以我们只好用时间换空间，每次 `pick` 都遍历一遍 `nums` 数组，用水塘抽样算法从中随机选出一个索引。

水塘抽样算法就是解决如何在长度未知的序列（数据流）中**随机**选择一个元素的数学技巧，类似 382. 链表随机节点。

结论：当你遇到第 `i` 个元素时，应该有 `1/i` 的概率选择该元素，`1 - 1/i` 的概率保持原有的选择。数学证明请看详细题解。

- **详细题解：谈谈游戏中的随机算法**

解法代码

```
class Solution {
    int[] nums;
    Random rand;
```

```
    public Solution(int[] nums) {
        this.nums = nums;
        this.rand = new Random();
    }

    public int pick(int target) {
        int count = 0, res = -1;
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] != target) {
                continue;
            }
            count++;
            if (rand.nextInt(count) == 0) {
                res = i;
            }
        }

        return res;
    }
}
```

- **类似题目**：
  - 382. 链表随机节点 🟠
  - 384. 打乱数组 🟠

391. 完美矩形

LeetCode	力扣	难度
391. Perfect Rectangle	391. 完美矩形	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

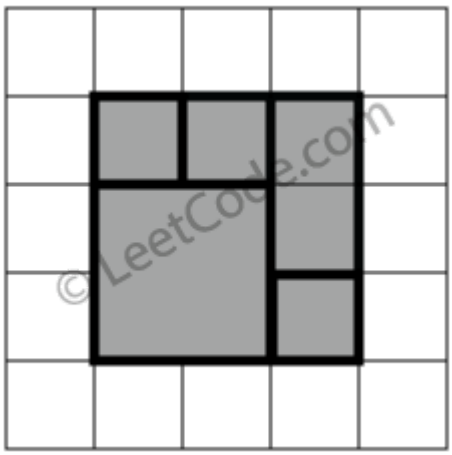
我们有 N 个矩形（N > 0），如果它们能够精确地覆盖一个矩形区域，我们称之为「完美矩形」。

每个矩形用左下角的点和右上角的点的坐标来表示。例如，一个单位正方形可以表示为 [1,1,2,2]，左下角的点的坐标为 (1, 1) 以及右上角的点的坐标为 (2, 2)。

判断这 N 个矩形是否能够构成完美矩形。

示例 1:

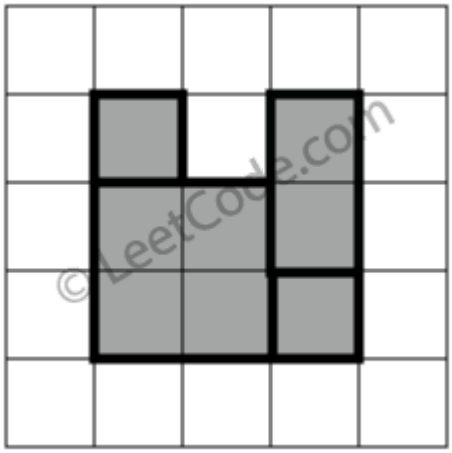
```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [3,2,4,4],
  [1,3,2,4],
  [2,3,3,4]
]

返回 true。5 个矩形一起可以精确地覆盖一个矩形区域。
```



示例 2:

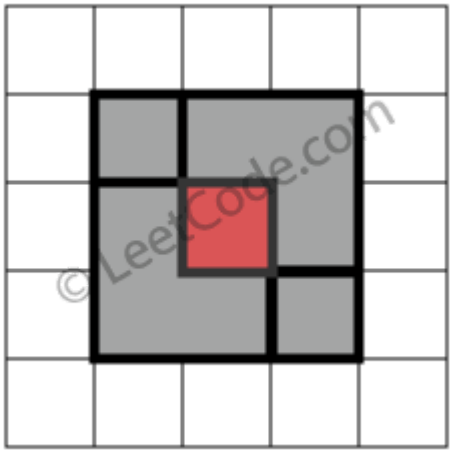
```
rectangles = [
  [1,1,3,3],
```

```
  [3,1,4,2],
  [1,3,2,4],
  [3,2,4,4]
]

返回 false。图形顶端留有间隔，无法覆盖成一个矩形。
```



示例 3:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [2,2,4,4]
]

返回 false。因为中间有相交区域，虽然形成了矩形，但不是精确覆盖。
```



基本思路

想判断最终形成的图形是否是完美矩形，需要从「面积」和「顶点」两个角度来处理。

第一步，计算出完美矩形的「理论坐标」，即所有小矩形中最靠左下角的顶点坐标和最靠右上角的顶点坐标。

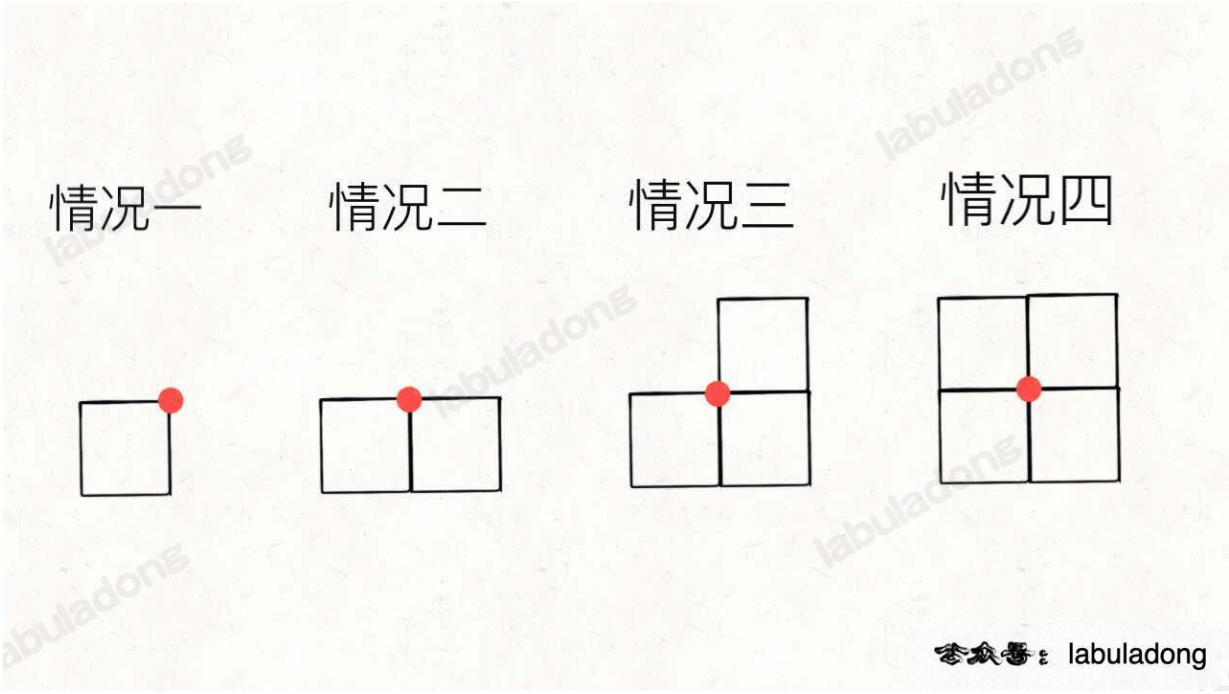
如果所有小矩形的面积之和不等于这个完美矩形的理论面积，那么说明最终形成的图形肯定存在空缺或者重叠，肯定不是完美矩形。

如果小矩形面积之和与理论面积相同，也不一定就是完美矩形，比如你假想一个完美矩形，然后我在它中间挖掉一个小矩形，把这个小矩形向下平移一个单位。这样小矩形的面积之和没变，但显然已经不是完美矩形了。

所以第二步，我们需要从「顶点」的维度来辅助判断，显然完美矩形只应该有四个顶点，如果不是四个，就不是完美矩形。

那么如何判断一个点是否是矩形的顶点？

当某一个点同时是 2 个或者 4 个小矩形的顶点时，该点最终不是顶点；当某一个点同时是 1 个或者 3 个小矩形的顶点时，该点最终是一个顶点，如下图：



按照「面积」和「顶点」两个角度来写代码即可。

- **详细题解：如何判定完美矩形**

解法代码

```
class Solution:
    def isRectangleCover(self, rectangles: List[List[int]]) -> bool:
        X1, Y1 = float('inf'), float('inf')
        X2, Y2 = -float('inf'), -float('inf')

        points = set()
        actual_area = 0
        for x1, y1, x2, y2 in rectangles:
            # 计算完美矩形的理论顶点坐标
```

```
            X1, Y1 = min(X1, x1), min(Y1, y1)
            X2, Y2 = max(X2, x2), max(Y2, y2)
            # 累加小矩形的面积
            actual_area += (x2 - x1) * (y2 - y1)
            # 记录最终形成的图形中的顶点
            p1, p2 = (x1, y1), (x1, y2)
            p3, p4 = (x2, y1), (x2, y2)
            for p in [p1, p2, p3, p4]:
                if p in points: points.remove(p)
                else:           points.add(p)
        # 判断面积是否相同
        expected_area = (X2 - X1) * (Y2 - Y1)
        if actual_area != expected_area:
            return False
        # 判断最终留下的顶点个数是否为 4
        if len(points) != 4:        return False
        # 判断留下的 4 个顶点是否是完美矩形的顶点
        if (X1, Y1) not in points: return False
        if (X1, Y2) not in points: return False
        if (X2, Y1) not in points: return False
        if (X2, Y2) not in points: return False
        # 面积和顶点都对应，说明矩形符合题意
        return True
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

400. 第 N 位数字

LeetCode	力扣	难度
400. Nth Digit	400. 第 N 位数字	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

给你一个整数 n，请你在无限的整数序列 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...] 中找出并返回第 n 位上的数字。

示例 1：

```
输入：n = 3
输出：3
```

基本思路

从我的个人刷题经验来看，遇到这类题目最咱首先想到的最简单有效的方式应该是找规律。

一位数有几个？1~9 共 9 * 1 = 9 个。共几位？共 1 * 9 = 9 位。

二位数有几个？10~99 共 9 * 10 = 90 个。共几位？共 2 * 90 = 180 位。

三位数有几个？100~999 共 9 * 100 = 900 个。共几位？共 3 * 900 = 2700 位。

以此类推，我们可以通过这个规律推断第 n 位的数字到底是什么。所以这道题的难点在于如何把上述规律写成算法代码，具体看我的实现吧。

解法代码

```
class Solution {
    public int findNthDigit(int n) {
        // 位数（一位数，两位数...）
        int digit = 1;
        // 1,10,100, 1000 这样的后缀
        long base = 1;

        while (n > 9 * base * digit) {
            n -= 9 * base * digit;
            base *= 10;
            digit++;
        }

        // 此时假设 base = 1000，那么说明 n 是 100~999 中的某个三位数的某一位
        // 哪个三位数呢？这样算：
```

```
            long val = base + (n - 1) / digit;
            // 是这个三位数的第几位呢？这样算：
            int index = (n - 1) % digit;

            // 怎么把 val 的第 index 这一位数字抠出来呢？可以转化成字符串来算：
            return ("" + val).charAt(index) - '0';
        }
    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 剑指 Offer 44. 数字序列中某一位的数字 🟠

剑指 Offer 44. 数字序列中某一位的数字

LeetCode	力扣	难度
剑指Offer44. 数字序列中某一位的数字 LCOF	剑指Offer44. 数字序列中某一位的数字	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学

数字以 0123456789101112131415… 的格式序列化到一个字符序列中。在这个序列中，第 5 位（从下标 0 开始计数）是 5，第 13 位是 1，第 19 位是 4，等等。

请写一个函数，求任意第 n 位对应的数字。

示例 1：

```
输入：n = 3
输出：3
```

基本思路

这道题和 400. 第 N 位数字 基本相同，只是这里的 n 是从 0 开始的，但数字序列也是从 0 开始的，所以只要特殊处理一下 0 就可以。具体的思路可以看我在第 400 题中写的思路提示，这里就不多说了。

解法代码

```
class Solution {
    public int findNthDigit(int n) {
        if (n == 0) {
            return 0;
        }
        return findNthDigit_100(n);
    }

    // 第 400 题的解法代码
    int findNthDigit_400(int n) {
        // 位数（一位数，两位数...）
        int digit = 1;
        // 1,10,100, 1000 这样的后缀
        long base = 1;

        while (n > 9 * base * digit) {
            n -= 9 * base * digit;
            base *= 10;
            digit++;
        }
```

```
        // 此时假设 base = 1000，那么说明 n 是 100~999 中的某个三位数的某一位
        // 哪个三位数呢？这样算：
        long val = base + (n - 1) / digit;
        // 是这个三位数的第几位呢？这样算：
        int index = (n - 1) % digit;

        // 怎么把 val 的第 index 这一位数字抠出来呢？这样算：
        return ("" + val).charAt(index) - '0';
    }
}
```

509. 斐波那契数

LeetCode	力扣	难度
509. Fibonacci Number	509. 斐波那契数	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：数学**

斐波那契数，通常用 `F(n)` 表示，形成的序列称为 **斐波那契数列**。该数列由 `0` 和 `1` 开始，后面的每一项数字都是前面两项数字的和。也就是：

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), 其中 n > 1
```

给你 `n`，请计算 `F(n)`。

示例 1：

```
输入：2
输出：1
解释：F(2) = F(1) + F(0) = 1 + 0 = 1
```

基本思路

> 本文有视频版：动态规划框架套路详解

PS：这道题在《算法小抄》 的第 31 页。

这题本身肯定是没有难度的，但是斐波那契数列可以帮你由浅入深理解动态规划算法的原理，建议阅读详细题解。

- **详细题解：动态规划解题套路框架**

解法代码

```
class Solution {
    public int fib(int n) {
        if (n == 0 || n == 1) {
            // base case
            return n;
        }
        // 分别代表 dp[i - 1] 和 dp[i - 2]
        int dp_i_1 = 1, dp_i_2 = 0;
```

```
        for (int i = 2; i <= n; i++) {
            // dp[i] = dp[i - 1] + dp[i - 2];
            int dp_i = dp_i_1 + dp_i_2;
            dp_i_2 = dp_i_1;
            dp_i_1 = dp_i;
        }
        return dp_i_1;
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 322. 零钱兑换 🟠
  - 70. 爬楼梯 🟢
  - 剑指 Offer 10- I. 斐波那契数列 🟢
  - 剑指 Offer 10- II. 青蛙跳台阶问题 🟢
  - 剑指 Offer II 103. 最少的硬币数目 🟠

70. 爬楼梯

LeetCode	力扣	难度
70. Climbing Stairs	70. 爬楼梯	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：一维动态规划，动态规划**

假设你正站在第 0 层楼，需要爬 n（n 是正整数）级台阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶，你有多少种不同的方法可以爬到楼顶呢?

示例 1：

```
输入：3
输出：3
解释：有 3 种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

基本思路

这题属于最基本的动态规划，建议先看下前文 动态规划框架详解。

这题很像 509. 斐波那契数：爬到第 n 级台阶的方法个数等于爬到 n - 1 的方法个数和爬到 n - 2 的方法个数之和。

解法代码

```
class Solution {
    // 备忘录
    int[] memo;

    public int climbStairs(int n) {
        memo = new int[n + 1];
        return dp(n);
    }

    // 定义：爬到第 n 级台阶的方法个数为 dp(n)
    int dp(int n) {
        // base case
        if (n <= 2) {
            return n;
        }
```

```
        if (memo[n] > 0) {
            return memo[n];
        }
        // 状态转移方程：
        // 爬到第 n 级台阶的方法个数等于爬到 n - 1 的方法个数和爬到 n - 2 的方法个数
之和。
        memo[n] = dp(n - 1) + dp(n - 2);
        return memo[n];
    }
}
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 剑指 Offer 10- II. 青蛙跳台阶问题 🟢

剑指 Offer 10- I. 斐波那契数列

LeetCode	力扣	难度
剑指Offer10-I. 斐波那契数列 LCOF	剑指Offer10-I. 斐波那契数列	🟢

- **标签：动态规划，数学**

写一个函数，输入 n，求斐波那契（Fibonacci）数列的第 n 项（即 F(N)）。斐波那契数列的定义如下：

```
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), 其中 N > 1.
```

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

```
输入：n = 2
输出：1
```

基本思路

> 本文有视频版：动态规划框架套路详解

PS：这道题在《算法小抄》 的第 31 页。

这题本身肯定是没有难度的，但是斐波那契数列可以帮你由浅入深理解动态规划算法的原理，建议阅读详细题解。

另外，剑指 offer 的这道斐波那契和原题 509. 斐波那契数 还是有点不一样的，就是要求你对结果求模。

因为结果会让 int 溢出，所以要么你用 long 类型，要么你一边求和一边求模。

加法求模的一个运算法则如下：

```
(a + b) % Q = (a % Q + b % Q) % Q
```

这个等式很容易推导，而且也可以运用到乘法上，具体的推导思路可以查看我的这篇文章 如何高效进行模幂运算。

- **详细题解：动态规划解题套路框架**

解法代码

```
class Solution {
    public int fib(int n) {
        if (n == 0 || n == 1) {
            // base case
            return n;
        }
        int Q = 1000000007;
        // 分别代表 dp[i - 1] 和 dp[i - 2]
        int dp_i_1 = 1, dp_i_2 = 0;
        for (int i = 2; i <= n; i++) {
            // dp[i] = dp[i - 1] + dp[i - 2];
            int dp_i = (dp_i_1 % Q + dp_i_2 % Q) % Q;
            dp_i_2 = dp_i_1;
            dp_i_1 = dp_i;
        }
        return dp_i_1;
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

剑指 Offer 10- II. 青蛙跳台阶问题

这道题和 70. 爬楼梯 相同。

645. 错误的集合

LeetCode	力扣	难度
645. Set Mismatch	645. 错误的集合	🟢

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学，数组

集合 S 包含从 1 到 n 的整数。不幸的是，因为数据错误，导致集合里面某一个数字复制了成了集合里面的另外一个数字的值，导致集合**丢失了一个数字并且有一个数字重复**。

给定一个数组 nums 代表了集合 S 发生错误后的结果，请你找出重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。

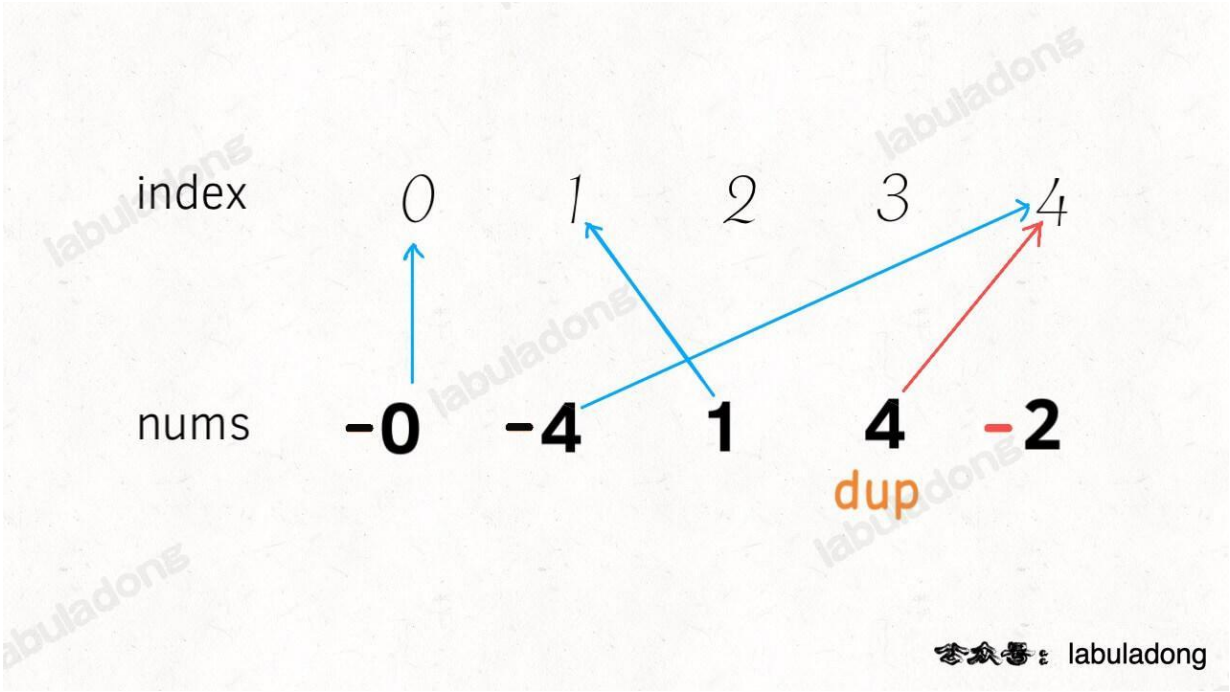
示例 1：

```
输入：nums = [1,2,2,4]
输出：[2,3]
```

基本思路

这题的核心思路是将每个索引对应的元素变成负数，以表示这个索引被对应过一次。

如果出现重复元素 4，直观结果就是，索引 4 所对应的元素已经是负数了：



对于缺失元素 3，直观结果就是，索引 3 所对应的元素是正数：

依据这个特点，就能找到缺失和重复的元素了。

- **详细题解：如何同时寻找缺失和重复的元素**

解法代码

```
class Solution {
    public int[] findErrorNums(int[] nums) {
        int n = nums.length;
        int dup = -1;
        for (int i = 0; i < n; i++) {
            // 现在的元素是从 1 开始的
            int index = Math.abs(nums[i]) - 1;
            if (nums[index] < 0)
                dup = Math.abs(nums[i]);
            else
                nums[index] *= -1;
        }

        int missing = -1;
        for (int i = 0; i < n; i++)
            if (nums[i] > 0)
                // 将索引转换成元素
                missing = i + 1;

        return new int[]{dup, missing};
    }
}
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

710. 黑名单中的随机数

LeetCode	力扣	难度
710. Random Pick with Blacklist	710. 黑名单中的随机数	🔴

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：数学，数组，随机算法

给定一个包含 `[0, n)` 中不重复整数的黑名单 `blacklist`，写一个函数从 `[0, n)` 中返回一个**不在** `blacklist` 中的随机整数。

对它进行优化使其尽量少调用系统方法 `Math.random()`。

基本思路

本题考察如下两点：

1、如果想高效地，等概率地随机获取元素，就要使用数组作为底层容器。

2、如果既要保持数组元素的紧凑性，又想从数组中间删除元素，那么可以把待删除元素换到最后，然后 pop 掉末尾的元素，这样时间复杂度就是 O(1) 了。当然，这样做的代价是我们需要额外的哈希表记录值到索引的映射。

- 详细题解：常数时间删除/查找数组中的任意元素

解法代码

```
class Solution {
public:
    int sz;
    unordered_map<int, int> mapping;

    Solution(int N, vector<int>& blacklist) {
        sz = N - blacklist.size();
        for (int b : blacklist) {
            mapping[b] = 666;
        }

        int last = N - 1;
        for (int b : blacklist) {
            // 如果 b 已经在区间 [sz, N)
            // 可以直接忽略
            if (b >= sz) {
                continue;
            }
            while (mapping.count(last)) {
                last--;
            }
```

```
            mapping[b] = last;
            last--;
        }
    }

    int pick() {
        // 随机选取一个索引
        int index = rand() % sz;
        // 这个索引命中了黑名单，
        // 需要被映射到其他位置
        if (mapping.count(index)) {
            return mapping[index];
        }
        // 若没命中黑名单，则直接返回
        return index;
    }
};
```

- **类似题目**：
  - 380. O(1) 时间插入、删除和获取随机元素 🟠
  - 剑指 Offer II 030. 插入、删除和随机访问都是 O(1) 的容器 🟠

1288. 删除被覆盖区间

LeetCode	力扣	难度
1288. Remove Covered Intervals	1288. 删除被覆盖区间	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：区间问题，排序**

给你一个区间列表，请你删除列表中被其他区间所覆盖的区间。

只有当 `c <= a` 且 `b <= d` 时，我们才认为区间 `[a,b)` 被区间 `[c,d)` 覆盖。

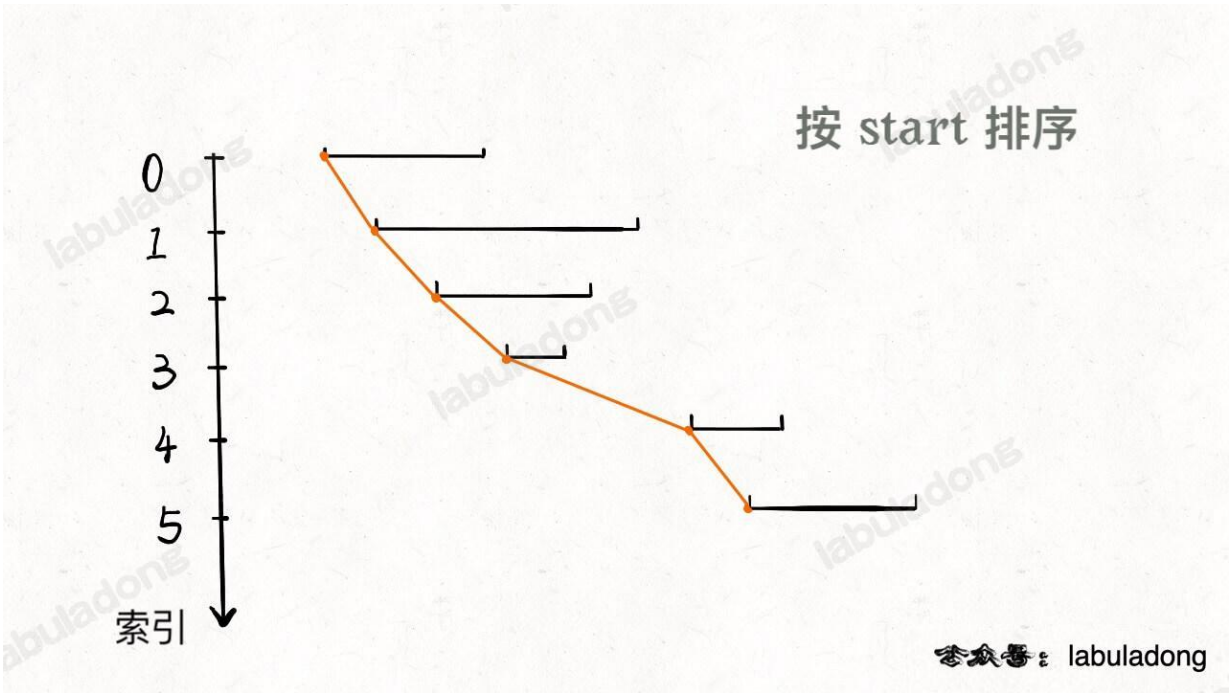
在完成所有删除操作后，请你返回列表中剩余区间的数目。

示例：

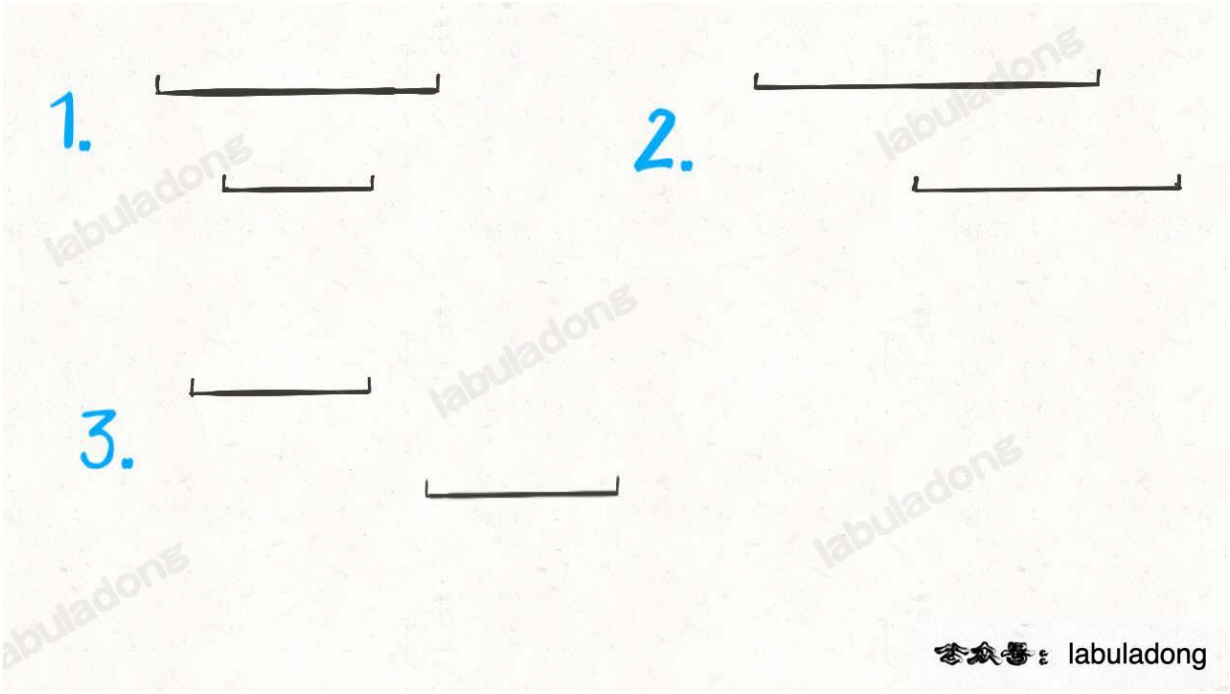
```
输入：intervals = [[1,4],[3,6],[2,8]]
输出：2
解释：区间 [3,6] 被区间 [2,8] 覆盖，所以它被删除了。
```

基本思路

按照区间的起点进行升序排序：



排序之后，两个相邻区间可能有如下三种情况：



对于情况一，找到了覆盖区间。

对于情况二，两个区间可以合并，成一个大区间。

对于情况三，两个区间完全不相交。

依据几种情况，就可以写出代码了。

- **详细题解：一个方法解决三道区间问题**

解法代码

```
class Solution {
    public int removeCoveredIntervals(int[][] intervals) {
        // 按照起点升序排列，起点相同时降序排列
        Arrays.sort(intervals, (a, b) -> {
            if (a[0] == b[0]) {
                return b[1] - a[1];
            }
            return a[0] - b[0];
        });

        // 记录合并区间的起点和终点
        int left = intervals[0][0];
        int right = intervals[0][1];

        int res = 0;
        for (int i = 1; i < intervals.length; i++) {
            int[] intv = intervals[i];
            // 情况一，找到覆盖区间
            if (left <= intv[0] && right >= intv[1]) {
                res++;
            }
```

```
                // 情况二，找到相交区间，合并
                if (right >= intv[0] && right <= intv[1]) {
                    right = intv[1];
                }
                // 情况三，完全不相交，更新起点和终点
                if (right < intv[0]) {
                    left = intv[0];
                    right = intv[1];
                }
            }

            return intervals.length - res;
        }
    }
```

▶ 🎃 代码可视化动画（需安装 Chrome 插件）🎃

- **类似题目**：
  - 56. 合并区间 🟠
  - 986. 区间列表的交集 🟠
  - 剑指 Offer II 074. 合并区间 🟠

56. 合并区间

LeetCode	力扣	难度
56. Merge Intervals	56. 合并区间	🟠

- **标签：区间问题，排序**

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`，请你将所有重叠的区间合并后返回。

示例 1：

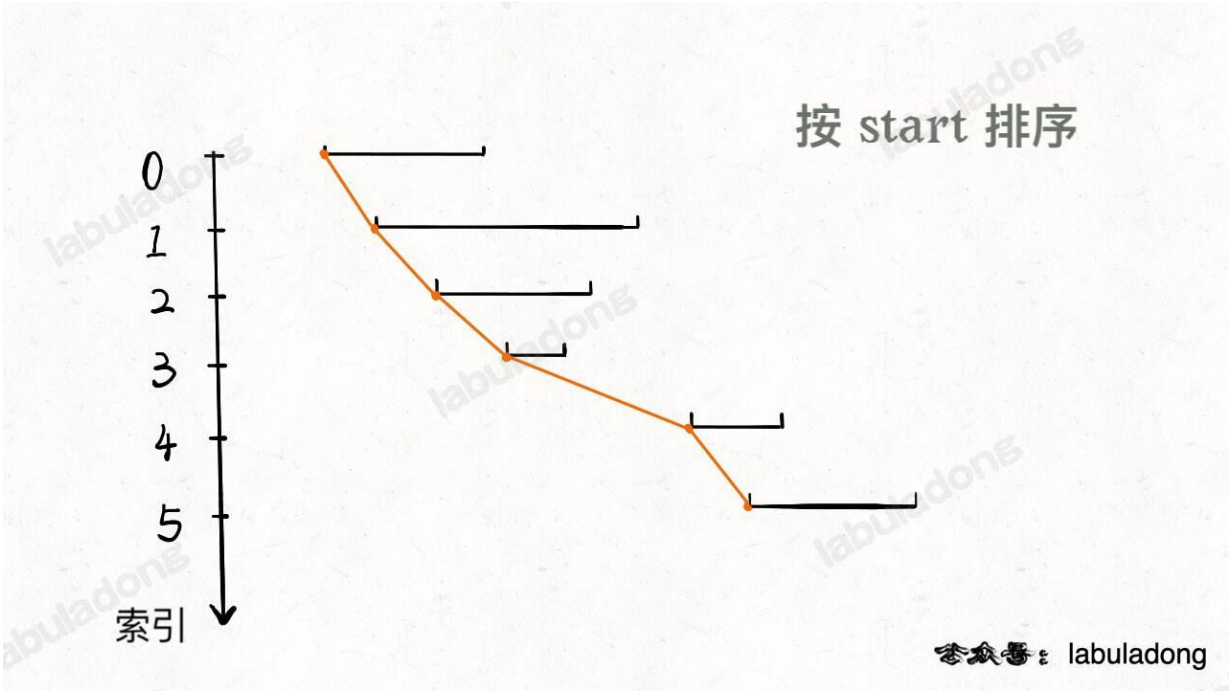
```
输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

示例 2：

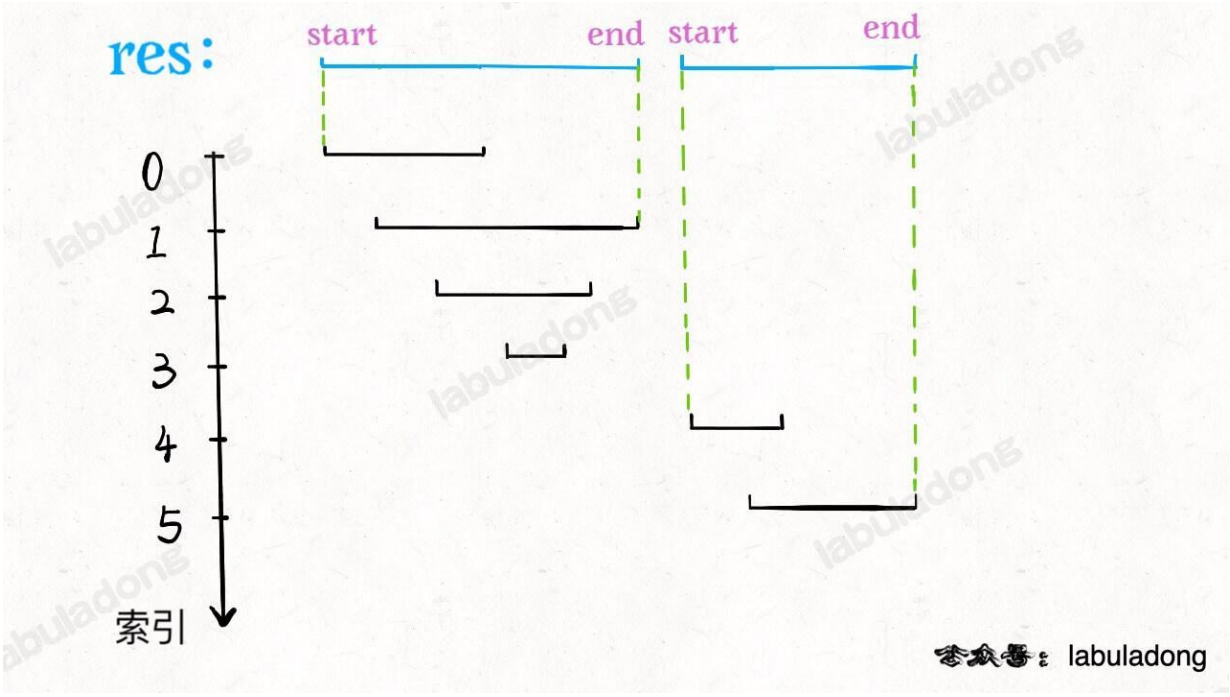
```
输入：intervals = [[1,4],[4,5]]
输出：[[1,5]]
解释：区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

基本思路

一个区间可以表示为 `[start, end]`，先按区间的 `start` 排序：



显然，对于几个相交区间合并后的结果区间 `x`，`x.start` 一定是这些相交区间中 `start` 最小的，`x.end` 一定是这些相交区间中 `end` 最大的：



由于已经排了序，`x.start` 很好确定，求 `x.end` 也很容易，可以类比在数组中找最大值的过程。

- **详细题解：一个方法解决三道区间问题**

解法代码

```
class Solution {
    public int[][] merge(int[][] intervals) {
        LinkedList<int[]> res = new LinkedList<>();
        // 按区间的 start 升序排列
        Arrays.sort(intervals, (a, b) -> {
```

```
            return a[0] - b[0];
        });

        res.add(intervals[0]);
        for (int i = 1; i < intervals.length; i++) {
            int[] curr = intervals[i];
            // res 中最后一个元素的引用
            int[] last = res.getLast();
            if (curr[0] <= last[1]) {
                last[1] = Math.max(last[1], curr[1]);
            } else {
                // 处理下一个待合并区间
                res.add(curr);
            }
        }
        return res.toArray(new int[0][0]);
    }
}
```

▶ 🥳 代码可视化动画（需安装 Chrome 插件）🥳

- **类似题目**：
  - 1288. 删除被覆盖区间 🟠
  - 986. 区间列表的交集 🟠
  - 剑指 Offer II 074. 合并区间 🟠

986. 区间列表的交集

LeetCode	力扣	难度
986. Interval List Intersections	986. 区间列表的交集	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- 标签：区间问题，数组双指针

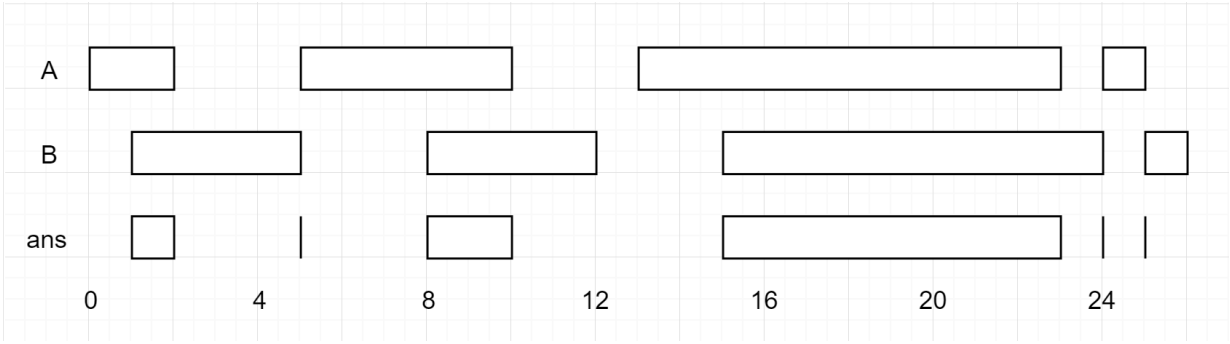
给定两个由一些 **闭区间** 组成的列表，`firstList` 和 `secondList`，其中 `firstList[i] = [starti, endi]` 而 `secondList[j] = [startj, endj]`。每个区间列表都是成对 **不相交** 的，并且 **已经排序**。

返回这 **两个区间列表的交集**。

形式上，**闭区间** `[a, b]`（其中 `a <= b`）表示实数 `x` 的集合，而 `a <= x <= b`。

两个闭区间的 **交集**是一组实数，要么为空集，要么为闭区间。例如，`[1, 3]` 和 `[2, 4]` 的交集为 `[2, 3]`。

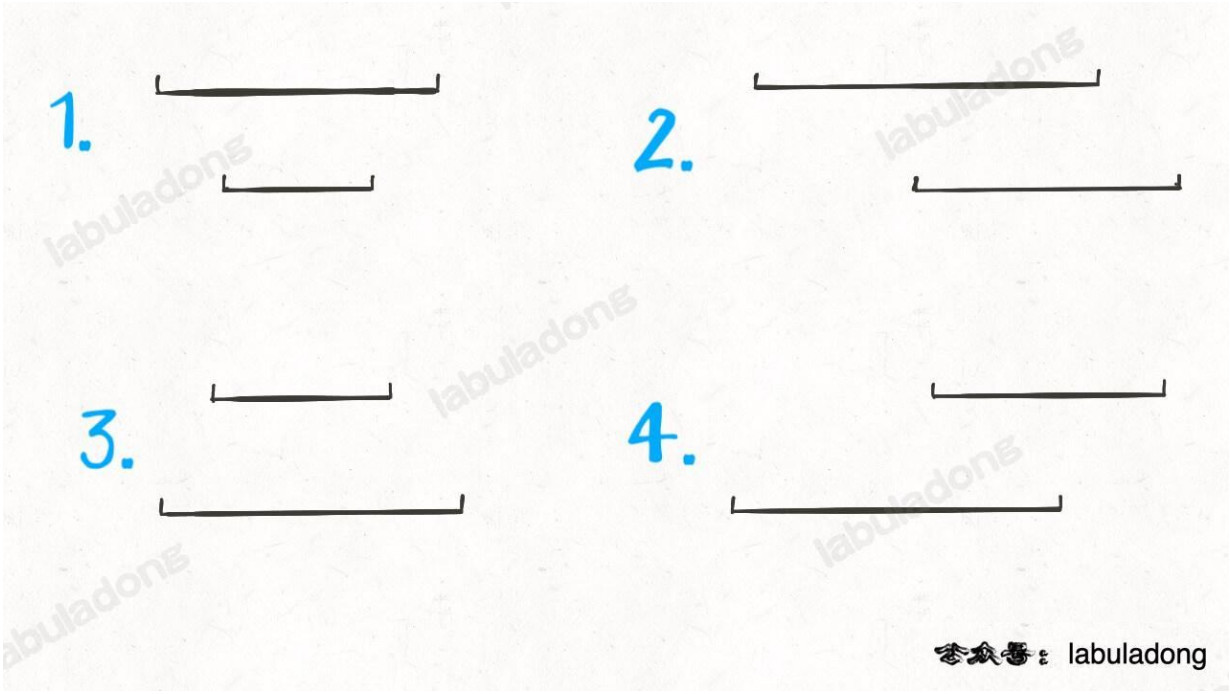
示例 1：



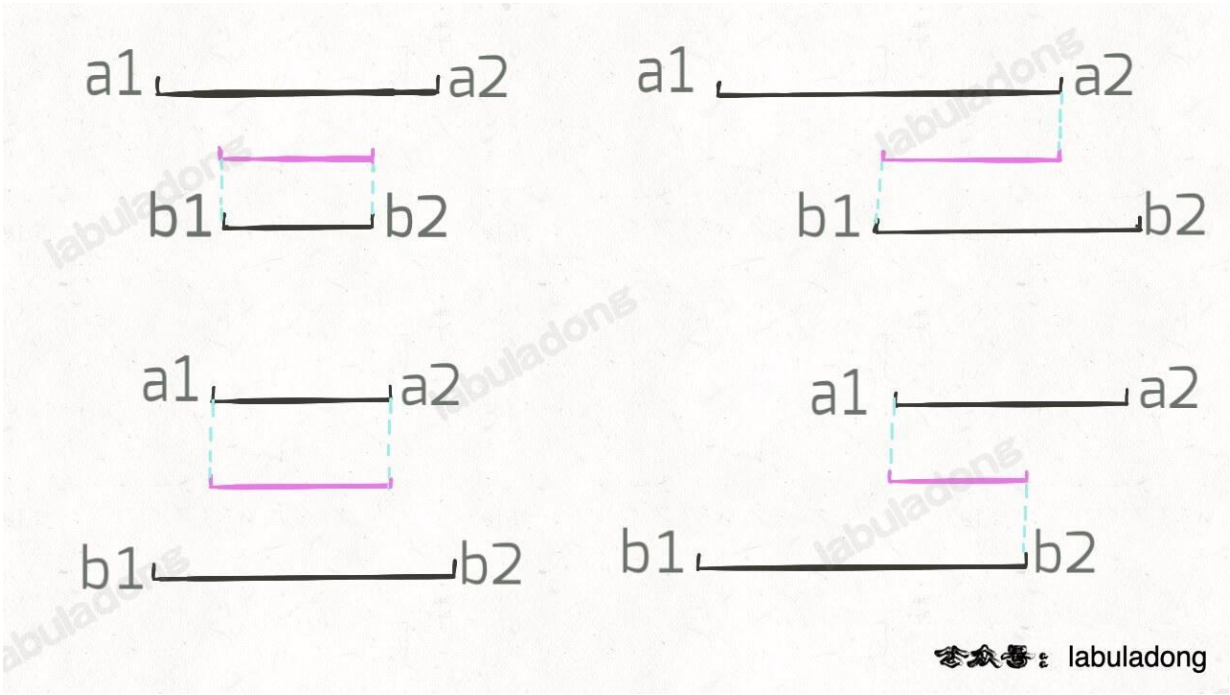

```
输入：firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]
输出：[[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

基本思路

我们用 `[a1, a2]` 和 `[b1, b2]` 表示在 `A` 和 `B` 中的两个区间，如果这两个区间有交集，需满足 `b2 >= a1 && a2 >= b1`，分下面四种情况：



根据上图可以发现规律，假设交集区间是 `[c1, c2]`，那么 `c1 = max(a1, b1), c2 = min(a2, b2)`：



这一点就是寻找交集的核心。

- **详细题解：一个方法解决三道区间问题**

解法代码

```
class Solution {
    public int[][] intervalIntersection(int[][] A, int[][] B) {
        List<int[]> res = new LinkedList<>();
        int i = 0, j = 0;
        while (i < A.length && j < B.length) {
            int a1 = A[i][0], a2 = A[i][1];
```

```
                int b1 = B[j][0], b2 = B[j][1];

                if (b2 >= a1 && a2 >= b1) {
                    res.add(new int[]{
                            Math.max(a1, b1), Math.min(a2, b2)
                    });
                }
                if (b2 < a2) {
                    j++;
                } else {
                    i++;
                }
            }
            return res.toArray(new int[0][0]);
        }
    }
```

▶ 🌈 代码可视化动画（需安装 Chrome 插件）🌈

- **类似题目**：
  - 1288. 删除被覆盖区间 🟠
  - 56. 合并区间 🟠
  - 剑指 Offer II 074. 合并区间 🟠

剑指 Offer II 074. 合并区间

这道题和 56. 合并区间 相同。

435. 无重叠区间

LeetCode	力扣	难度
435. Non-overlapping Intervals	435. 无重叠区间	

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：区间问题，排序**

给定一个区间的集合，计算需要移除区间的最小数量，使剩余区间互不重叠。

基本思路

PS：这道题在《算法小抄》 的第 381 页。

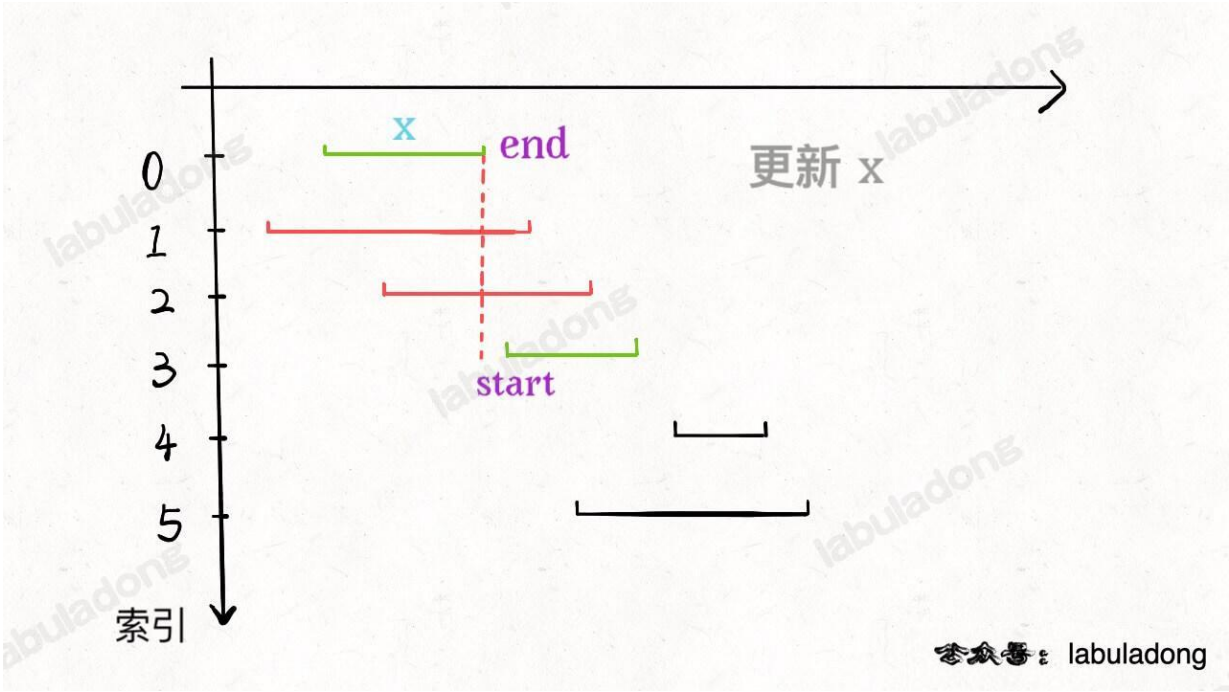
区间调度问题是让你计算若干区间中最多有几个互不相交的区间，这道题是区间调度问题的一个简单变体。

区间调度问题思路可以分为以下三步：

1、从区间集合 `intvs` 中选择一个区间 `x`，这个 `x` 是在当前所有区间中**结束最早的**（`end` 最小）。

2、把所有与 `x` 区间相交的区间从区间集合 `intvs` 中删除。

3、重复步骤 1 和 2，直到 `intvs` 为空为止。之前选出的那些 `x` 就是最大不相交子集。



- **详细题解：贪心算法之区间调度问题**

解法代码

```java
class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        int n = intervals.length;
        return n - intervalSchedule(intervals);
    }

    // 区间调度算法，算出 intvs 中最多有几个互不相交的区间
    int intervalSchedule(int[][] intvs) {
        if (intvs.length == 0) return 0;
        // 按 end 升序排序
        Arrays.sort(intvs, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                return a[1] - b[1];
            }
        });
        // 至少有一个区间不相交
        int count = 1;
        // 排序后，第一个区间就是 x
        int x_end = intvs[0][1];
        for (int[] interval : intvs) {
            int start = interval[0];
            if (start >= x_end) {
                // 找到下一个选择的区间了
                count++;
                x_end = interval[1];
            }
        }
        return count;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

- **类似题目**：
  - 452. 用最少数量的箭引爆气球 🟠

452. 用最少数量的箭引爆气球

LeetCode	力扣	难度
452. Minimum Number of Arrows to Burst Balloons	452. 用最少数量的箭引爆气球	🟠

- **标签：区间问题，排序**

在二维空间中有许多圆形的气球，一个气球在 x 轴上的投影为一个坐标区间 `[start, end]`。

一支弓箭可以沿着 x 轴从不同点垂直向上射出，如果在坐标 `x` 处射出一支箭，所有 `start <= x <= end` 的气球都会被射爆。

给你一个数组 `points`，其中 `points[i] = [start_i, end_i]` 表示第 `i` 个气球的位置，可以射出的弓箭的数量没有限制，计算引爆所有气球所必须射出的最小弓箭数。

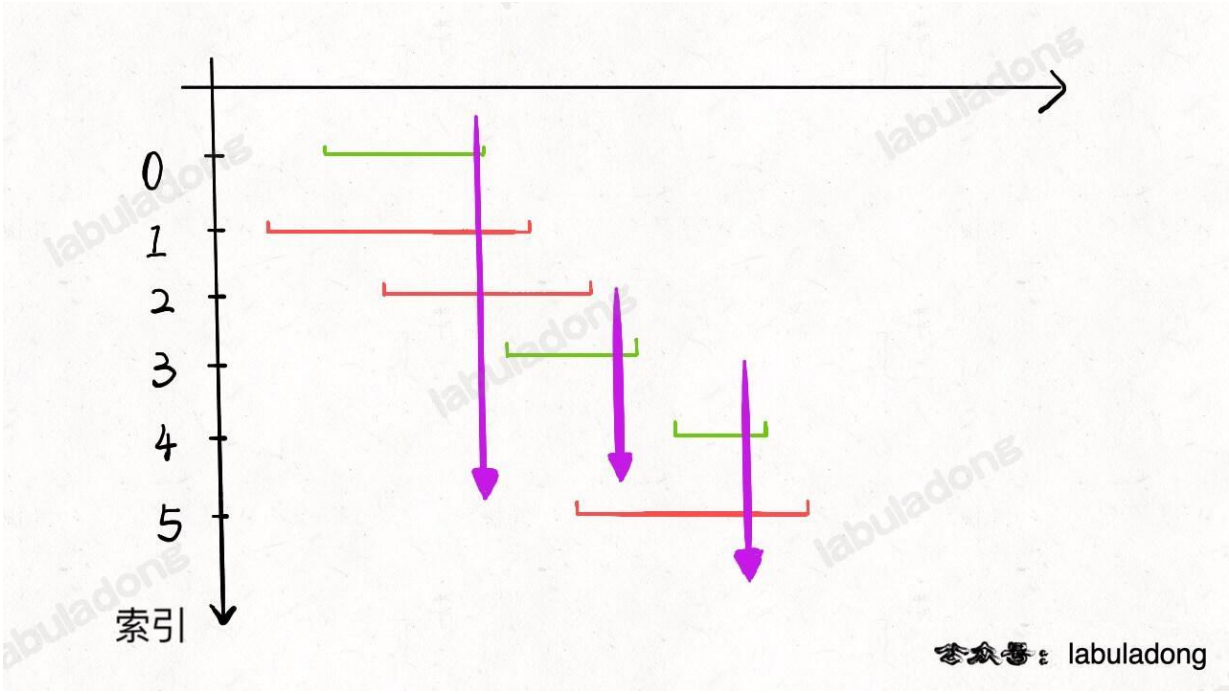
示例 1：

```
输入：points = [[10,16],[2,8],[1,6],[7,12]]
输出：2
解释：对于该样例，x = 6 可以射爆 [2,8],[1,6] 两个气球，以及 x = 11 射爆另外两个气球
```

基本思路

PS：这道题在《算法小抄》 的第 381 页。

区间调度问题是让你计算若干区间中最多有几个互不相交的区间，这道题是区间调度问题的一个简单变体，需要的箭头数量就是不重叠区间的数量。



区间调度问题思路可以分为以下三步：

1、从区间集合 intvs 中选择一个区间 x，这个 x 是在当前所有区间中**结束最早的**（end 最小）。

2、把所有与 x 区间相交的区间从区间集合 intvs 中删除。

3、重复步骤 1 和 2，直到 intvs 为空为止。之前选出的那些 x 就是最大不相交子集。

- **详细题解：贪心算法之区间调度问题**

解法代码

```
class Solution {
    // 区间调度问题
    public int findMinArrowShots(int[][] intvs) {
        if (intvs.length == 0) return 0;
        // 按 end 升序排序
        Arrays.sort(intvs, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                return a[1] - b[1];
            }
        });
        // 至少有一个区间不相交
        int count = 1;
        // 排序后，第一个区间就是 x
        int x_end = intvs[0][1];
        for (int[] interval : intvs) {
            int start = interval[0];
            // 把 >= 改成 > 就行了
            if (start > x_end) {
                count++;
                x_end = interval[1];
            }
```

```
            }
            return count;
        }
    }
```

▶ 👾 代码可视化动画（需安装 Chrome 插件）👾

- **类似题目**：
  - 435. 无重叠区间 🟠

1024. 视频拼接

LeetCode	力扣	难度
1024. Video Stitching	1024. 视频拼接	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：区间问题，排序，贪心算法**

你将会获得一系列视频片段，这些片段来自于一项持续时长为 `T` 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

视频片段 `clips[i]` 都用区间进行表示：开始于 `clips[i][0]` 并于 `clips[i][1]` 结束。我们甚至可以对这些片段自由地再剪辑，例如片段 `[0, 7]` 可以剪切成 `[0, 1] + [1, 3] + [3, 7]` 三部分。

我们需要将这些片段进行再剪辑，并将剪辑后的内容拼接成覆盖整个运动过程的片段（`[0, T]`）。返回所需片段的最小数目，如果无法完成该任务，则返回 `-1`。

示例 1：

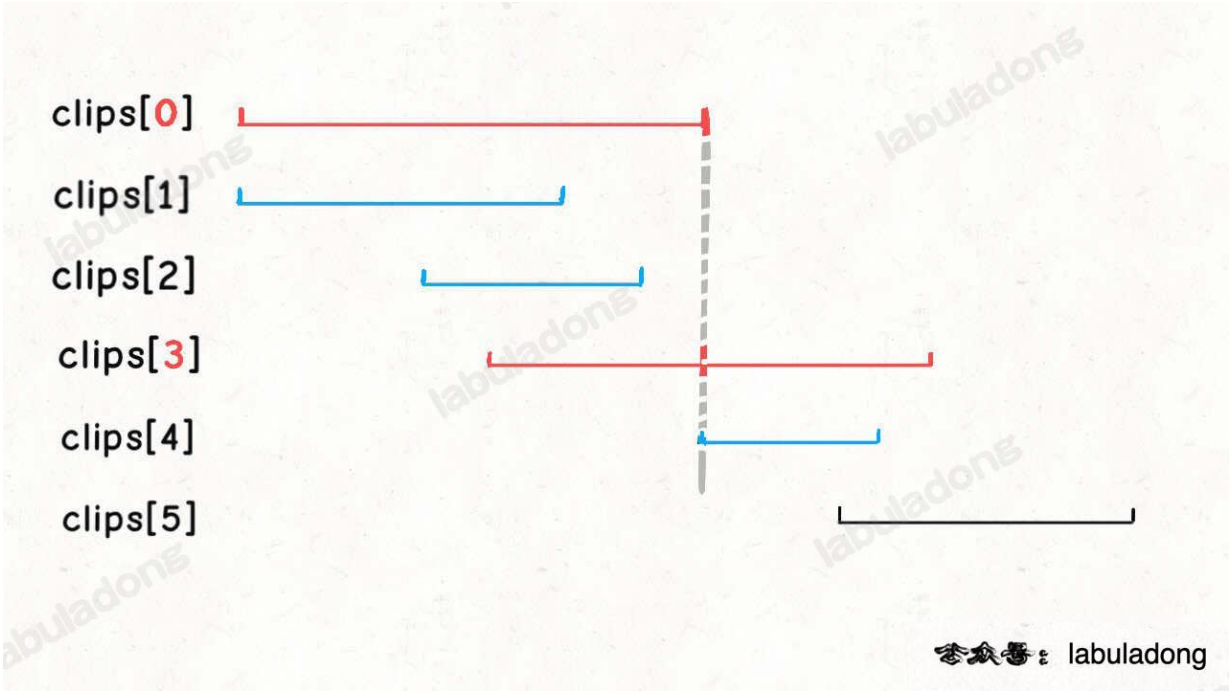
```
输入：clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], T = 10
输出：3
解释：
我们选中 [0,2], [8,10], [1,9] 这三个片段。
然后，按下面的方案重制比赛片段：
将 [1,9] 再剪辑为 [1,2] + [2,8] + [8,9]。
现在我们手上有 [0,2] + [2,8] + [8,10]，而这些涵盖了整场比赛 [0, 10]。
```

基本思路

我做这道题的思路是先按照起点升序排序，如果起点相同的话按照终点降序排序，主要考虑到这道题的以下两个特点：

1、要用若干短视频凑出完成视频 `[0, T]`，至少得有一个短视频的起点是 0。

2、如果有几个短视频的起点都相同，那么一定应该选择那个最长（终点最大）的视频。



排序之后，从第一个区间开始选，每当选中一个区间 `x`（图中红色的区间），我们会比较所有起点小于 `x.start` 的区间，根据贪心策略，它们中终点最大的那个区间就是下一个会被选中的区间，以此类推。

- **详细题解：剪视频剪出一个贪心算法**

解法代码

```
class Solution {
    public int videoStitching(int[][] clips, int T) {
        if (T == 0) return 0;
        // 按起点升序排列，起点相同的降序排列
        // PS：其实起点相同的不用降序排列也可以，不过我觉得这样更清晰
        Arrays.sort(clips, (a, b) -> {
            if (a[0] == b[0]) {
                return b[1] - a[1];
            }
            return a[0] - b[0];
        });
        // 记录选择的短视频个数
        int res = 0;

        int curEnd = 0, nextEnd = 0;
        int i = 0, n = clips.length;
        while (i < n && clips[i][0] <= curEnd) {
            // 在第 res 个视频的区间内贪心选择下一个视频
            while (i < n && clips[i][0] <= curEnd) {
                nextEnd = Math.max(nextEnd, clips[i][1]);
                i++;
            }
            // 找到下一个视频，更新 curEnd
            res++;
            curEnd = nextEnd;
            if (curEnd >= T) {
```

```
                // 已经可以拼出区间 [0, T]
                return res;
            }
        }
        // 无法连续拼出区间 [0, T]
        return -1;
    }
}
```

▶ 🌟 代码可视化动画（需安装 Chrome 插件）🌟

451. 根据字符出现频率排序

LeetCode	力扣	难度
451. Sort Characters By Frequency	451. 根据字符出现频率排序	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，排序**

给定一个字符串 `s`，根据字符出现的 **频率** 对其进行 **降序排序**。一个字符出现的 **频率** 是它出现在字符串中的次数，**返回已排序的字符串**。如果有多个答案，返回其中任何一个。

示例 1:

```
输入: s = "tree"
输出: "eert"
解释: 'e' 出现两次，'r' 和 't' 都只出现一次。
因此 'e' 必须出现在 'r' 和 't' 之前。此外，"eetr" 也是一个有效的答案。
```

基本思路

做这道题肯定需要计算每个字符出现的频率，然后你可以用很多种其他方法把字符按照频率排序。我这里用优先级队列来实现排序的效果，详细看代码。

解法代码

```
class Solution {
    public String frequencySort(String s) {
        char[] chars = s.toCharArray();
        // s 中的字符 -> 该字符出现的频率
        HashMap<Character, Integer> charToFreq = new HashMap<>();
        for (char ch : chars) {
            charToFreq.put(ch, charToFreq.getOrDefault(ch, 0) + 1);
        }

        PriorityQueue<Map.Entry<Character, Integer>>
                pq = new PriorityQueue<>((entry1, entry2) -> {
            // 队列按照键值对中的值（字符出现频率）从大到小排序
            return entry2.getValue().compareTo(entry1.getValue());
        });

        // 按照字符频率排序
        for (Map.Entry<Character, Integer> entry : charToFreq.entrySet())
{
            pq.offer(entry);
        }
```

```
        StringBuilder sb = new StringBuilder();
        while (!pq.isEmpty()) {
            // 把频率最高的字符排在前面
            Map.Entry<Character, Integer> entry = pq.poll();
            String part =
String.valueOf(entry.getKey()).repeat(entry.getValue());
            sb.append(part);
        }

        return sb.toString();
    }
}
```

1834. 单线程 CPU

LeetCode	力扣	难度
1834. Single-Threaded CPU	1834. 单线程 CPU	🟠

Stars 123k 精品课程 查看 公众号 @labuladong B站 @labuladong

- **标签：二叉堆，排序**

给你一个二维数组 `tasks`，用于表示 `n` 项从 `0` 到 `n - 1` 编号的任务。其中 `tasks[i] = [enqueueTime_i, processingTime_i]` 意味着第 `i` 项任务将会于 `enqueueTime_i` 时进入任务队列，需要 `processingTime_i` 的时长完成执行。

现有一个单线程 CPU，同一时间只能执行 **最多一项** 任务，该 CPU 将会按照下述方式运行：

- 如果 CPU 空闲，且任务队列中没有需要执行的任务，则 CPU 保持空闲状态。
- 如果 CPU 空闲，但任务队列中有需要执行的任务，则 CPU 将会选择**执行时间最短**的任务开始执行。如果多个任务具有同样的最短执行时间，则选择下标最小的任务开始执行。
- 一旦某项任务开始执行，CPU 在 **执行完整个任务** 前都不会停止。
- CPU 可以在完成一项任务后，立即开始执行一项新任务。

返回 CPU 处理任务的顺序。

示例 1：

```
输入：tasks = [[1,2],[2,4],[3,2],[4,1]]
输出：[0,2,3,1]
解释：事件按下述流程运行：
- time = 1，任务 0 进入任务队列，可执行任务项 = {0}
- 同样在 time = 1，空闲状态的 CPU 开始执行任务 0，可执行任务项 = {}
- time = 2，任务 1 进入任务队列，可执行任务项 = {1}
- time = 3，任务 2 进入任务队列，可执行任务项 = {1, 2}
- 同样在 time = 3，CPU 完成任务 0 并开始执行队列中用时最短的任务 2，可执行任务项 =
{1}
- time = 4，任务 3 进入任务队列，可执行任务项 = {1, 3}
- time = 5，CPU 完成任务 2 并开始执行队列中用时最短的任务 3，可执行任务项 = {1}
- time = 6，CPU 完成任务 3 并开始执行任务 1，可执行任务项 = {}
- time = 10，CPU 完成任务 1 并进入空闲状态
```

基本思路

这题的难度不算大，就是有些复杂，**难点在于你要同时控制三个变量（开始时间、处理时间、索引）的有序性，而且这三个变量还有优先级**：

首先应该考虑开始时间，因为只要到了开始时间，任务才进入可执行状态；

其次应该考虑任务的处理时间，在所有可以执行的任务中优先选择处理时间最短的；

如果存在处理时间相同的任务，那么优先选择索引最小的。

所以这道题的思路是：

先根据任务「开始时间」排序，维护一个时间线变量 now 来判断哪些任务到了可执行状态，然后借助一个优先级队列 pq 对「处理时间」和「索引」进行动态排序。

利用优先级队列动态排序是有必要的，因为每完成一个任务，时间线 now 就要更新，进而产生新的可执行任务。

解法代码

```
class Solution {
    public int[] getOrder(int[][] tasks) {
        int n = tasks.length;
        // 把原始索引也添加上，方便后面排序用
        ArrayList<int[]> triples = new ArrayList<>();
        for (int i = 0; i < tasks.length; i++) {
            triples.add(new int[]{tasks[i][0], tasks[i][1], i});
        }
        // 数组先按照任务的开始时间排序
        triples.sort((a, b) -> {
            return a[0] - b[0];
        });

        // 按照任务的处理时间排序，如果处理时间相同，按照原始索引排序
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
            if (a[1] != b[1]) {
                // 比较处理时间
                return a[1] - b[1];
            }
            // 比较原始索引
            return a[2] - b[2];
        });

        ArrayList<Integer> res = new ArrayList<>();
        // 记录完成任务的时间线
        int now = 0;
        int i = 0;
        while (res.size() < n) {
            if (!pq.isEmpty()) {
                // 完成队列中的一个任务
                int[] triple = pq.poll();
                res.add(triple[2]);
                // 每完成一个任务，就要推进时间线
                now += triple[1];
            } else if (i < n && triples.get(i)[0] > now) {
                // 队列为空可能因为还没到开始时间，
                // 直接把时间线推进到最近任务的开始时间
                now = triples.get(i)[0];
            }
```

```
                // 由于时间线的推进，会产生可以开始执行的任务
                for (; i < n && triples.get(i)[0] <= now; i++) {
                    pq.offer(triples.get(i));
                }
            }

            // Java 语言特性，将 List 转化成 int[] 格式
            int[] arr = new int[n];
            for (int j = 0; j < n; j++) {
                arr[j] = res.get(j);
            }
            return arr;
        }
    }
```