

# 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
  - 35. 搜索插入位置
  - 354. 俄罗斯套娃信封问题
  - 392. 判断子序列
  - 793. 阶乘函数后 K 个零
  - 875. 爱吃香蕉的珂珂
  - 1011. 在 D 天内送达包裹的能力
- 滑动窗口
  - 3. 无重复字符的最长子串
  - 76. 最小覆盖子串
  - 438. 找到字符串中所有字母异位词
  - 567. 字符串的排列
  - 239. 滑动窗口最大值
- 其他题目
  - 26. 删除有序数组中的重复项
  - 27. 移除元素
  - 283. 移动零
  - 11. 盛最多水的容器
  - 15. 三数之和
  - 18. 四数之和
  - 870. 优势洗牌
  - 42. 接雨水
  - 986. 区间列表的交集

### 链表双指针

- 2. 两数相加
- 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
- 141. 环形链表
- 142. 环形链表 II
- 160. 相交链表
- 876. 链表的中间结点
- 25. K 个一组翻转链表
- 83. 删除排序链表中的重复元素

- 92. 反转链表 II
- 234. 回文链表

## 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
- 560. 和为 K 的子数组

## 差分数组

- 370. 区间加法
- 1094. 拼车
- 1109. 航班预订统计

## 队列/栈算法

- 20. 有效的括号
- 921. 使括号有效的最少添加
- 1541. 平衡括号字符串的最少插入次数
- 32. 最长有效括号
- 225. 用队列实现栈
- 232. 用栈实现队列
- 239. 滑动窗口最大值

## 二叉堆

- 23. 合并 K 个升序链表
- 215. 数组中的第 K 个最大元素
- 295. 数据流的中位数
- 703. 数据流中的第 K 大元素

## 数据结构设计

- 146. LRU 缓存机制
- 341. 扁平化嵌套列表迭代器
- 380.  $O(1)$  时间插入、删除和获取随机元素
- 460. LFU 缓存
- 895. 最大频率栈

# 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
- 102. 二叉树的层序遍历
- 103. 二叉树的锯齿形层序遍历
- 104. 二叉树的最大深度
- 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
- 107. 二叉树的层序遍历 II
- 111. 二叉树的最小深度
- 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
- 144. 二叉树的前序遍历
- 145. 二叉树的后序遍历
- 222. 完全二叉树的节点个数
- 236. 二叉树的最近公共祖先
- 297. 二叉树的序列化与反序列化
- 341. 扁平化嵌套列表迭代器
- 501. 二叉搜索树中的众数
- 543. 二叉树的直径
- 559. N 叉树的最大深度
- 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
- 965. 单值二叉树

## 二叉搜索树

- 95. 不同的二叉搜索树 II
- 96. 不同的二叉搜索树
- 98. 验证二叉搜索树
- 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
- 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
- 1038. 把二叉搜索树转换为累加树
- 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
- 1373. 二叉搜索子树的最大键值和

## 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

- 并查集算法
  - 130. 被围绕的区域
  - 990. 等式方程的可满足性
- 最小生成树
  - 261. 以图判树
  - 1135. 最低成本联通所有城市
  - 1584. 连接所有点的最小费用
- 最短路径
  - 743. 网络延迟时间
  - 1514. 概率最大的路径
  - 1631. 最小体力消耗路径

## 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
- 37. 解数独
- 39. 组合总和
- 46. 全排列
- 77. 组合
- 78. 子集
- 51. N 皇后
- 104. 二叉树的最大深度
- 494. 目标和
- 698. 划分为 k 个相等的子集

### DFS 算法

- 130. 被围绕的区域
- 200. 岛屿数量
- 694. 不同的岛屿数量
- 695. 岛屿的最大面积
- 1020. 飞地的数量
- 1254. 统计封闭岛屿的数目
- 1905. 统计子岛屿

### 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
- 111. 二叉树的最小深度
- 752. 打开转盘锁
- 773. 滑动谜题

## 第四章、动态规划算法

## 一维 DP

- 45. 跳跃游戏 II
- 55. 跳跃游戏
- 53. 最大子序和
- 70. 爬楼梯
- 198. 打家劫舍
- 213. 打家劫舍 II
- 337. 打家劫舍 III
- 300. 最长递增子序列
- 322. 零钱兑换
- 354. 俄罗斯套娃信封问题

## 二维 DP

- 10. 正则表达式匹配
- 62. 不同路径
- 64. 最小路径和
- 72. 编辑距离
- 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
- 174. 地下城游戏
- 312. 戳气球
- 416. 分割等和子集
- 494. 目标和
- 514. 自由之路
- 518. 零钱兑换 II
- 583. 两个字符串的删除操作
- 712. 两个字符串的最小 ASCII 删除和
- 1143. 最长公共子序列
- 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

## 背包问题

- 416. 分割等和子集
- 494. 目标和
- 518. 零钱兑换 II

## 第五章、其他经典算法

### 数学算法

- 17. 电话号码的字母组合
- 77. 组合
- 78. 子集
- 134. 加油站
- 136. 只出现一次的数字
- 191. 位 1 的个数
- 231. 2 的幂
- 172. 阶乘后的零
- 793. 阶乘函数后 K 个零
- 204. 计数质数
- 268. 丢失的数字
- 292. Nim 游戏
- 319. 灯泡开关
- 877. 石子游戏
- 295. 数据流的中位数
- 372. 超级次方
- 382. 链表随机节点
- 398. 随机数索引
- 391. 完美矩形
- 509. 斐波那契数
- 645. 错误的集合
- 710. 黑名单中的随机数

## 区间问题

- 56. 合并区间
- 986. 区间列表的交集
- 1288. 删除被覆盖区间
- 435. 无重叠区间
- 452. 用最少数量的箭引爆气球
- 1024. 视频拼接

# 开篇、刷题笔记阅读指南

---

这本 PDF 是 [labuladong 的刷题三件套](#) 中的第二件：《[labuladong 的刷题笔记](#)》。

我的 [GitHub 算法仓库](#) 目前已经快 100k star 了（疯狂暗示点 star），为了感谢大家一直以来的支持，我制作了刷题三件套。

刷题三件套共包含《[labuladong 的算法秘籍](#)》和《[labuladong 的刷题笔记](#)》这两本 PDF 以及 labuladong 的辅助刷题插件。

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。

你可以把《算法秘籍》理解成教材，《刷题笔记》理解成一本练习册，当然应该先看教材，再通过练习册巩固复习。

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。

制作这本《刷题笔记》的灵感来源于「单词速记卡」的形式，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。

这本 PDF 的内容也可以在我的公众号查看，目录入口如下：



**labuladong 的刷题三件套** 中除了上述两本 PDF，还包含一款 Chrome 刷题辅助插件，完美融合了上述两本 PDF 的内容，能够在力扣题目页面显示《算法秘籍》中对应的详细题解和《刷题笔记》中的简明思路（也支持英文版 LeetCode），是建议每个读者都安装的：

题目描述 评论 (1.4k) 题解 (2.6k) 提交记录 Java

42. 接雨水 labuladong 题解 思路

难度 困难 2860

详细题解 简要思路和参考解法

给定  $n$  个非负整数表示每

对于任意一个位置  $i$ ，能够装的水为：

示例 1：

```
water[i] = min(
    max(height[0..i]),
    max(height[i..end])
) - height[i]
```

Copy

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] 表示的雨水 (蓝色部分表

示例 2：

输入: height = [4, 2, 0, 3, 2, 5, 2, 1, 6, 2, 3, 2, 1]
输出: 9

基本思路

对于任意一个位置  $i$ ，能够装的水为：

水的高度由左右最高的柱子决定，即  $\min(l_{\max}, r_{\max}) - height[i]$

而且插件目前提供了手把手带你刷通所有二叉树题目的功能，未来还会添加手把手刷动态规划等功能，详情见[这里](#)。

扫码关注公众号，后台回复关键词「三件套」即可免费下载两本 PDF 和刷题插件。另外，后续我会持续输出高质量算法文章，公众号菜单有我亲自制作训练营和课程以及刷题打卡活动，大家持续关注公众号的通知即可：



微信搜一搜

Q labuladong 公众号

另外，也建议关注我的微信视频号，每周我都会抽空直播，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：



扫一扫二维码，关注我的视频号

# 34. 在排序数组中查找元素的第一个和最后一个位置



- 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

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：我作了首诗，保你闭着眼睛也能写对二分查找](#)

## 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) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            left = mid + 1; // 注意
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    if (left == 0) return -1;
    return nums[left - 1] == target ? (left - 1) : -1;
}
}
```

- 类似题目：
  - [704. 二分查找（简单）](#)

# 704. 二分查找



- 标签: [二分搜索](#)

给定有  $n$  个元素的升序整型数组  $\text{nums}$  和一个目标值  $\text{target}$ , 写一个函数搜索  $\text{nums}$  中的  $\text{target}$ , 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

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  $\text{target}$  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  $\text{target}$  重复出现多次的情景, 计算  $\text{target}$  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: [我作了首诗, 保你闭着眼睛也能写对二分查找](#)

## 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;
}
}
```

- 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（中等）

# 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;
}
```

# 354. 俄罗斯套娃信封问题



- 标签: 动态规划, 一维动态规划, 二分搜索

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:

宽度 w    高度 h

升序

[ 1 , 8 ]

[ 2 , 3 ]

[ 5 , 4 ] ]

[ 5 , 2 ] ]

降序

[ 6 , 7 ] ]

[ 6 , 4 ] ]

降序



然后在 h 上寻找最长递增子序列：

宽度 w    高度 h

[ 1 , 8 ]

[ 2 , 3 ]

[ 5 , 4 ]

[ 5 , 2 ]

[ 6 , 7 ]

[ 6 , 4 ]



- 详细题解：最长递增子序列之信封嵌套问题

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;
    }
}
```

## 392. 判断子序列



- 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 **S<sub>1</sub>**, **S<sub>2</sub>**, ..., **S<sub>k</sub>** 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。

s      a    b    c  
t      c    a    c    b    h    b    c

公众号: labuladong

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: [二分查找的妙用: 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();
    }
}
```

# 793. 阶乘函数后 K 个零



- 标签: 数学, 二分搜索

$f(x)$  是  $x!$  末尾是 0 的数量,  $x! = 1 * 2 * 3 * \dots * x$ , 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 阶乘后的零 这道题的解法函数 `trailingZeroes`。

搜索有多少个  $n$  满足  $\text{trailingZeroes}(n) == K$ , 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  $[0, \text{LONG\_MAX}]$  中取值, 寻找满足  $\text{trailingZeroes}(n) == K$  的左侧边界和右侧边界, 相减即是答案。

- 详细题解: 阶乘相关的算法题, 东哥又整活儿了

## 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（简单）](#)

# 875. 爱吃香蕉的珂珂



- 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
```

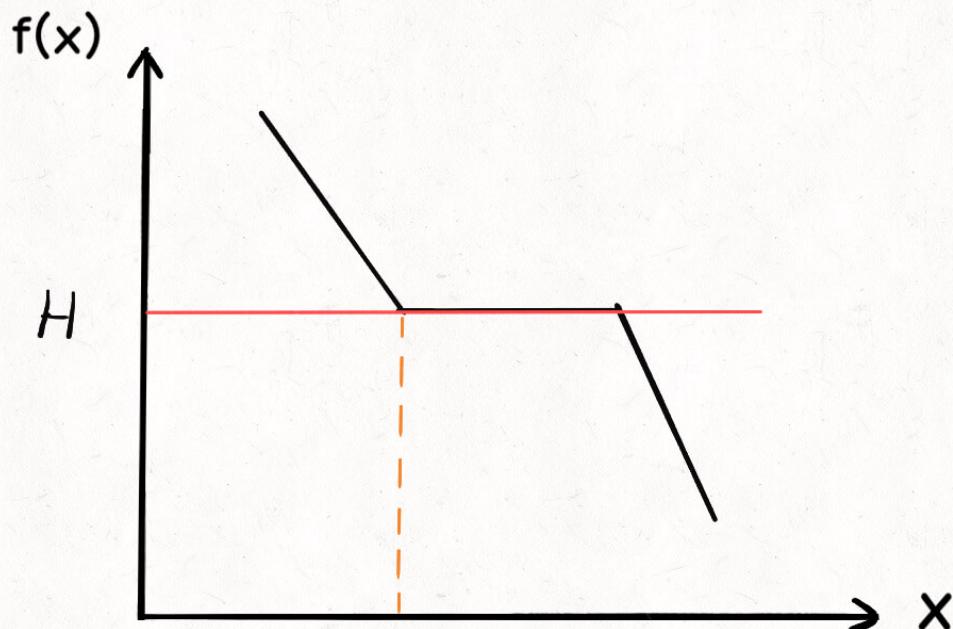
## 基本思路

PS: 这道题在《算法小抄》的第 360 页。

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。

这题珂珂吃香蕉的速度就是自变量  $x$ ，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  $H$ 。

它们的关系如下图：



公众号: labuladong

关于本题二分搜索的具体思路见详细题解。

- 详细题解: 我写了一个套路, 助你随心所欲运用二分搜索

## 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;
}
}
```

- 类似题目：

- [1011. 在 D 天内送达包裹的能力（中等）](#)

# 1011. 在 D 天内送达包裹的能力



- 标签: [二分搜索](#)

传送带上的第  $i$  个包裹的重量为  $\text{weights}[i]$ ，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:  $\text{weights} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ ,  $D = 5$

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

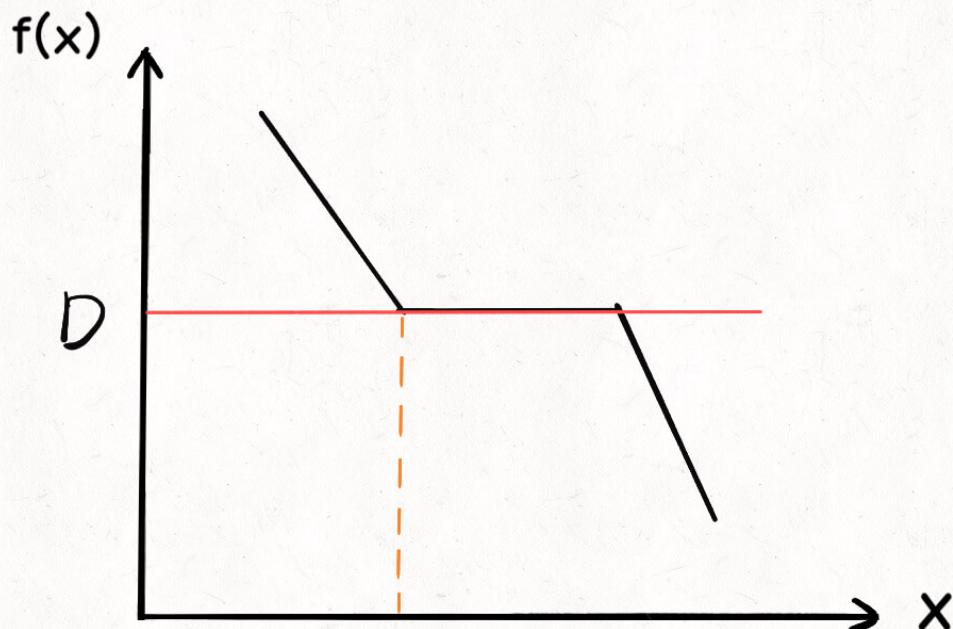
请注意，货物必须按照给定的顺序装运，因此使用载重能力为 14 的船舶并将包装分成 (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) 是不允许的。

## 基本思路

PS：这道题在《算法小抄》的第 360 页。

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  $x$ ,  $f(x)$ ,  $\text{target}$  分别是什么，并写出单调函数  $f$  的代码\*\*。

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， $\text{target}$  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。



公众号: labuladong

关于本题二分搜索的具体思路见详细题解。

- 详细题解: 我写了一个套路, 助你随心所欲运用二分搜索

## 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 w : weights) {
            if (w > x) {
                days++;
            } else {
                w %= x;
                days += w / x;
                if (w % x != 0) days++;
            }
        }
        return days;
    }
}
```

```
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;  
}  
}
```

- 类似题目：
  - [875. 爱吃香蕉的珂珂](#) (中等)

### 3. 无重复字符的最长子串



- 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

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

```
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]--;
            }
            res = max(res, right - left);
        }
        return res;
    }
};
```

```
    }
    // 在这里更新答案
    res = max(res, right - left);
}
return res;
};

};
```

- 类似题目：

- [76.最小覆盖子串](#) (困难)
- [567.字符串的排列](#) (中等)
- [438.找到字符串中所有字母异位词](#) (中等)

# 76. 最小覆盖子串



- 标签: 滑动窗口, 数组双指针

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

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：我写了套框架，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);
    }
};
```

- 类似题目：
  - 567. 字符串的排列（中等）

- 438. 找到字符串中所有字母异位词（中等）
- 3. 无重复字符的最长子串（中等）

# 438. 找到字符串中所有字母异位词



- 标签: 滑动窗口, 数组双指针

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

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: 我写了套框架, 把滑动窗口算法变成了默写题

## 解法代码

```
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;
};

};
```

- 类似题目：

- 76. 最小覆盖子串（困难）
- 567. 字符串的排列（中等）
- 3. 无重复字符的最长子串（中等）

# 567. 字符串的排列



- 标签: 滑动窗口, 数组双指针

给你两个字符串  $s_1$  和  $s_2$ , 写一个函数来判断  $s_2$  是否包含  $s_1$  的排列 ( $s_1$  的排列之一是  $s_2$  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

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: 我写了套框架, 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;
}
};
```

- 类似题目：

- [76. 最小覆盖子串](#) (困难)
- [438. 找到字符串中所有字母异位词](#) (中等)
- [3. 无重复字符的最长子串](#) (中等)

## 239. 滑动窗口最大值



- 标签: 数据结构, 队列, 滑动窗口

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]`

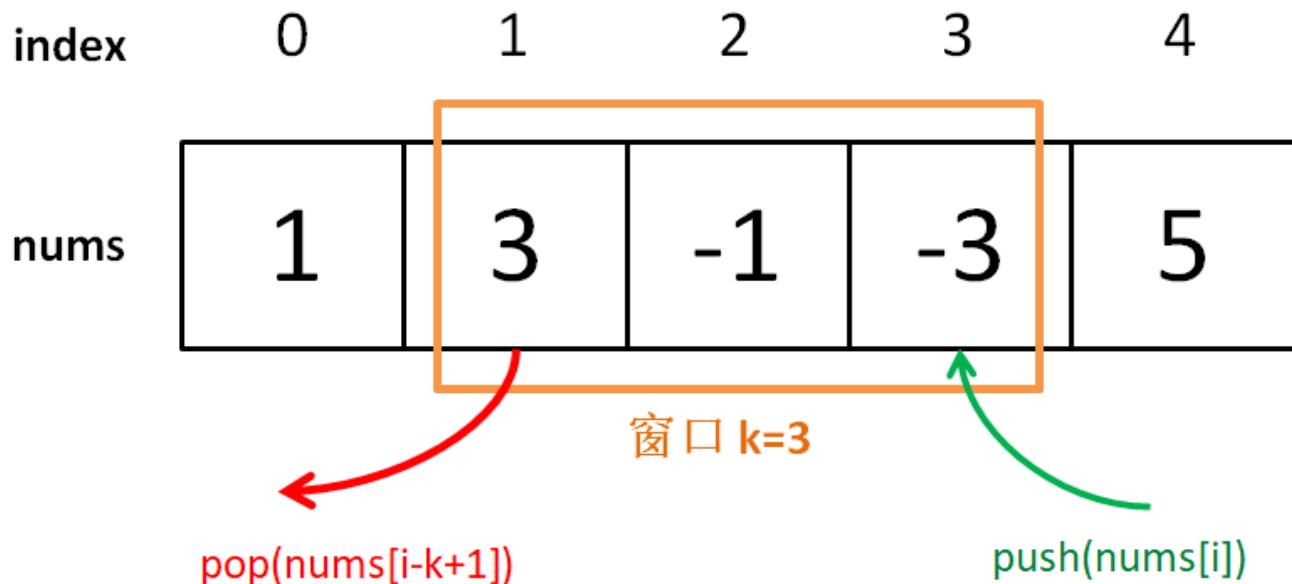
解释:

滑动窗口的位置	最大值
<code>[1 3 -1] -3 5 3 6 7</code>	<code>3</code>
<code>1 [3 -1 -3] 5 3 6 7</code>	<code>3</code>
<code>1 3 [-1 -3 5] 3 6 7</code>	<code>5</code>
<code>1 3 -1 [-3 5 3] 6 7</code>	<code>5</code>
<code>1 3 -1 -3 [5 3 6] 7</code>	<code>6</code>
<code>1 3 -1 -3 5 [3 6 7]</code>	<code>7</code>

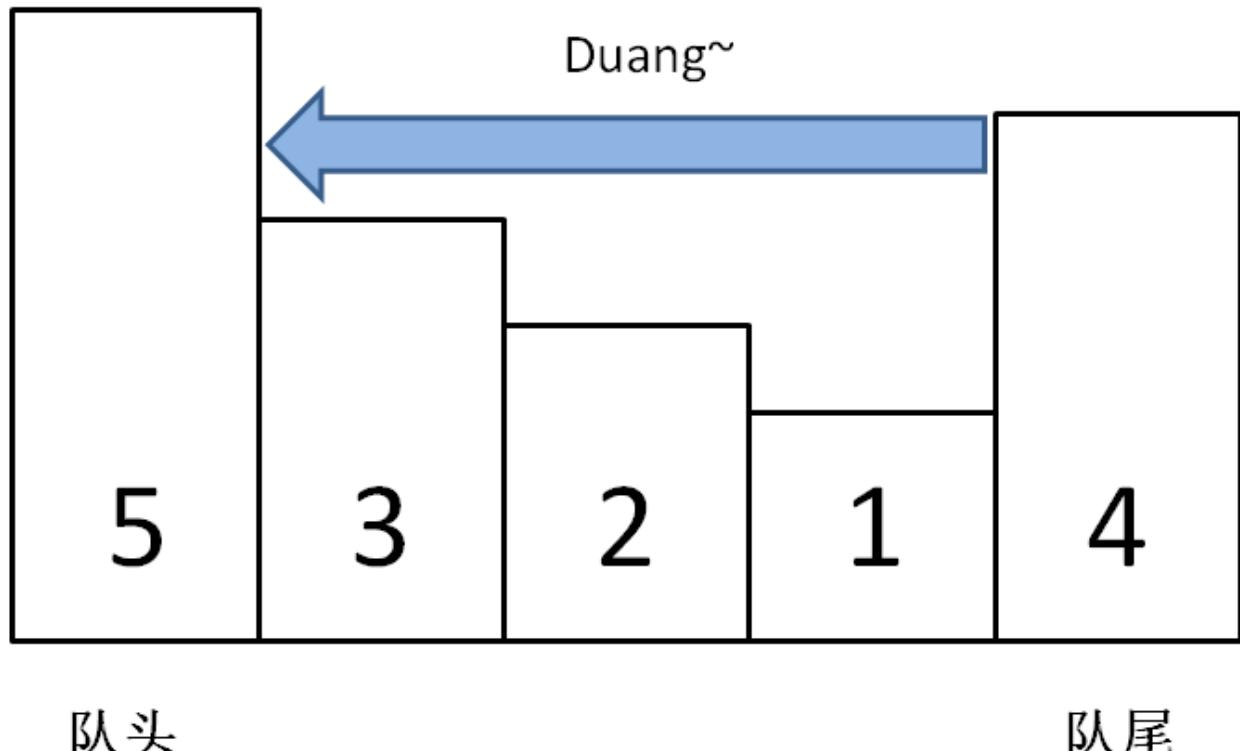
### 基本思路

PS: 这道题在《算法小抄》的第 271 页。

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」数据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;
    }
}
```

## 26. 删除有序数组中的重复项



- 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

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。

nums	0	0	1	2	2	3	3
------	---	---	---	---	---	---	---

公众号: labuladong

- 详细题解: [双指针技巧秒杀四道数组/链表题目](#)

### 解法代码

```
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;
}
```

- 类似题目：

- 83. 删除排序链表中的重复元素（简单）
- 27. 移除元素（简单）
- 283. 移动零（简单）

# 27. 移除元素



- 标签: 数组双指针, 数组

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

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: 双指针技巧秒杀四道数组/链表题目

## 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;
    }
}
```

- 类似题目:
  - 26. 删除有序数组中的重复项 (简单)
  - 83. 删除排序链表中的重复元素 (简单)
  - 283. 移动零 (简单)

# 283. 移动零



- 标签: 数组双指针, 数组

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

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: 双指针技巧秒杀四道数组/链表题目

## 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;
    }
}
```

- 类似题目：

- 26. 删除有序数组中的重复项（简单）
- 83. 删除排序链表中的重复元素（简单）
- 27. 移除元素（简单）

# 11. 盛最多水的容器

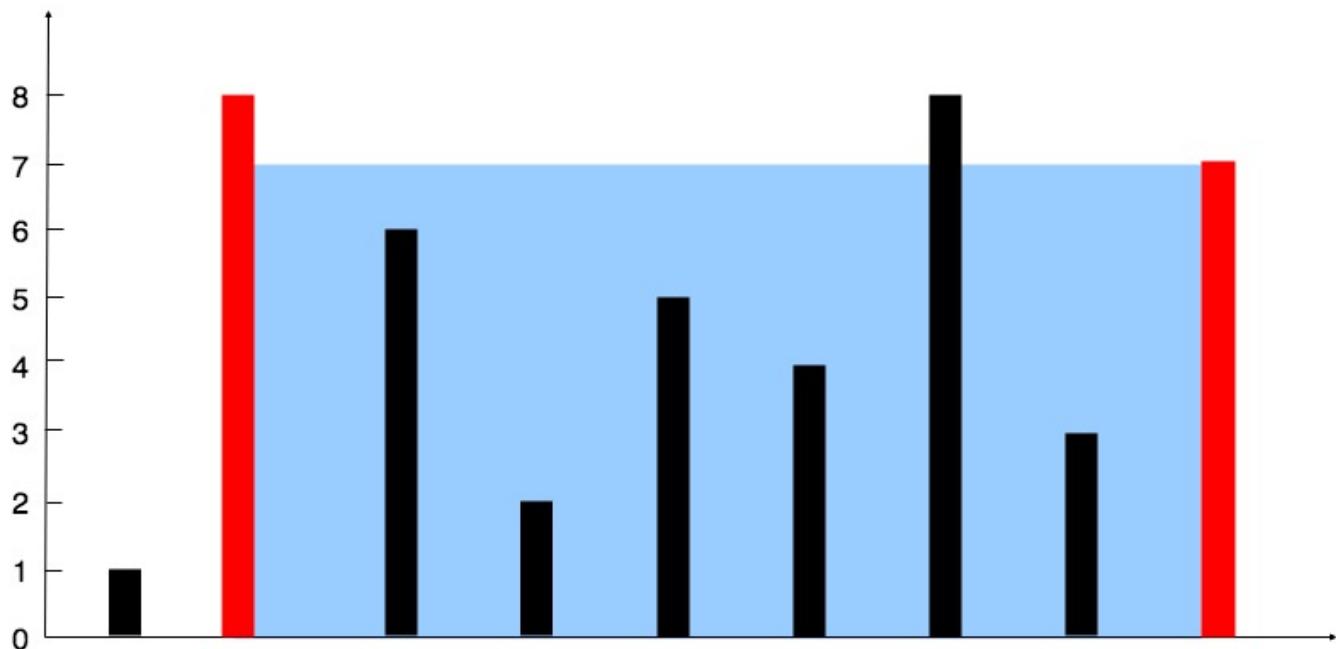


- 标签: 数组双指针

给你  $n$  个非负整数  $a_1, a_2, \dots, a_n$ , 其中每个数代表坐标中的一个点  $(i, a_i)$ 。在坐标内画  $n$  条垂直线, 垂直线  $i$  的两个端点分别为  $(i, a_i)$  和  $(i, 0)$ 。

找出其中的两条线, 使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

示例 1:



输入: [1,8,6,2,5,4,8,3,7]

输出: 49

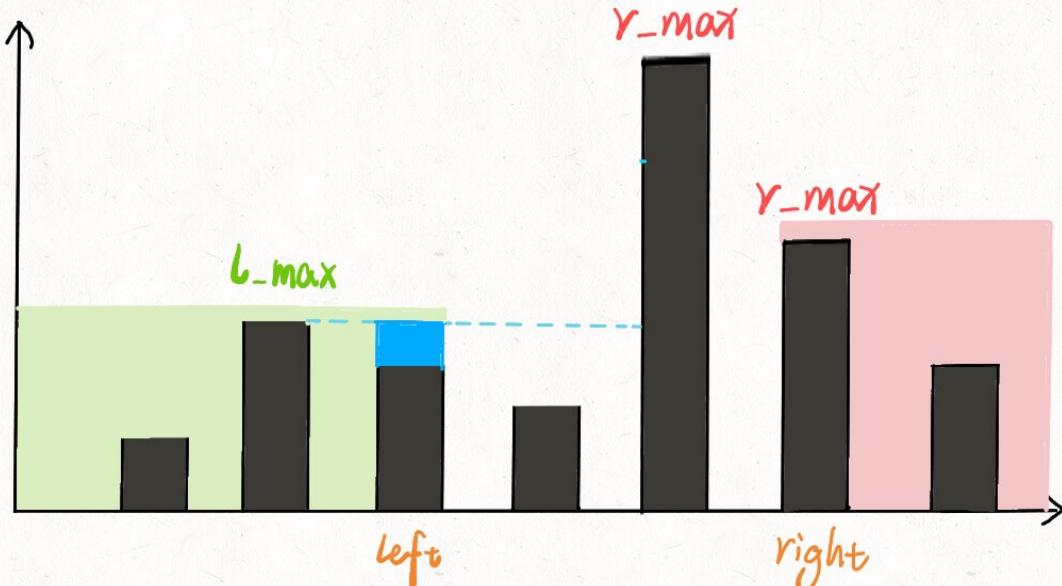
解释: 图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下, 容器能够容纳水 (表示为蓝色部分) 的最大值为 49。

## 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:



公众号: labuladong

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。

## 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);
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }
        return res;
    }
}
```

```
- 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;
}
}
```

# 15. 三数之和



- 标签: 数组双指针, 递归

给你一个包含  $n$  个整数的数组  $\text{nums}$ , 判断  $\text{nums}$  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]]
```

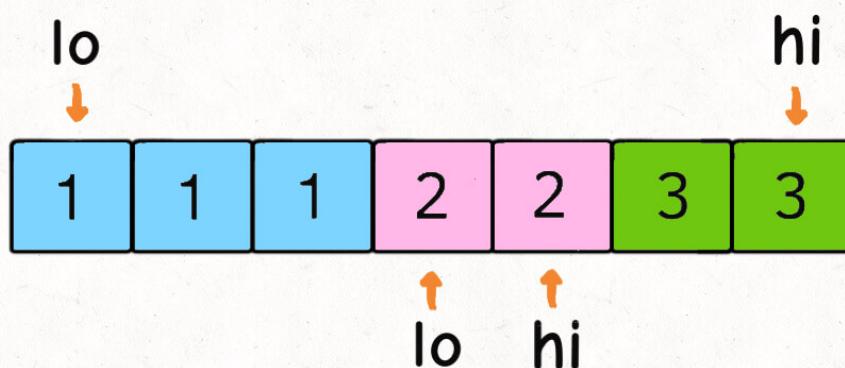
## 基本思路

PS: 这道题在《算法小抄》的第 319 页。

$n$ Sum 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  $\text{lo}$  和  $\text{hi}$  分别在数组开头和结尾, 这样就可以控制  $\text{nums}[\text{lo}]$  和  $\text{nums}[\text{hi}]$  这两数之和的大小:

如果你想让它俩的和大一些, 就让  $\text{lo}++$ , 如果你想让它俩的和小一些, 就让  $\text{hi}--$ 。



公众号: labuladong

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

- 详细题解：一个函数秒杀 2Sum 3Sum 4Sum 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;
}
};
```

- 类似题目：
  - [18. 四数之和（中等）](#)

# 18. 四数之和



- 标签: 数组双指针, 递归

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

- 1、 $0 \leq a, b, c, d < n$ 。
- 2、 $a, b, c$  和  $d$  互不相同。
- 3、 $\text{nums}[a] + \text{nums}[b] + \text{nums}[c] + \text{nums}[d] == \text{target}$ 。

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。

$n$ Sum 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  $\text{lo}$  和  $\text{hi}$  分别在数组开头和结尾, 这样就可以控制  $\text{nums}[\text{lo}]$  和  $\text{nums}[\text{hi}]$  这两数之和的大小:

如果你想让它俩的和大一些, 就让  $\text{lo}++$ , 如果你想让它俩的和小一些, 就让  $\text{hi}--$ 。

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

- 详细题解: 一个函数秒杀 2Sum 3Sum 4Sum 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;
}
};
```

- 类似题目:

- 15. 三数之和 (中等)

# 870. 优势洗牌



- 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,  $\text{nums1}$  就是田忌的马,  $\text{nums2}$  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: 算法大师——孙膑

## 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;
}
```

## 42. 接雨水



- 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

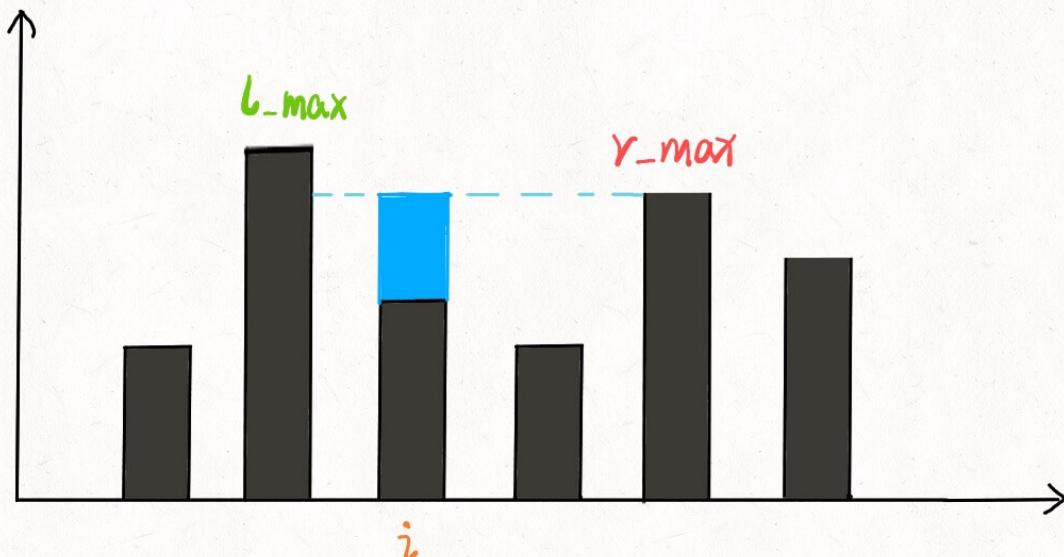
解释: 上面是由数组 `[0,1,0,2,1,0,1,3,2,1,2,1]` 表示的高度图, 在这种情况下, 可以接 6 个单位的雨水 (蓝色部分表示雨水)。

### 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]
```



公众号: labuladong

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：手把手搞懂接雨水问题的多种解法](#)

## 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;
    }
}
```

# 986. 区间列表的交集



- 标签: 区间问题, 数组双指针

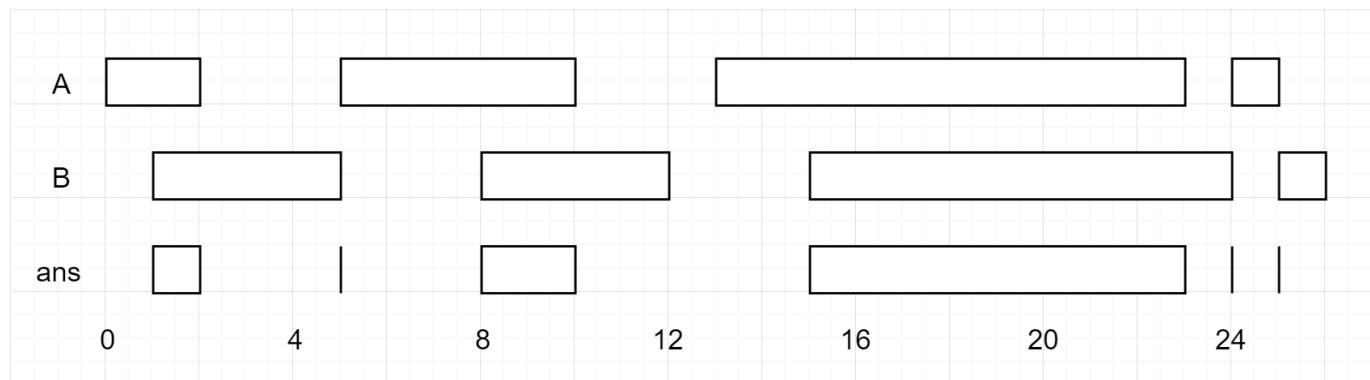
给定两个由一些闭区间组成的列表, `firstList` 和 `secondList`, 其中 `firstList[i] = [starti, endi]` 而 `secondList[j] = [startj, endj]`。每个区间列表都是成对不相交的, 并且已经排序。

返回这两个区间列表的交集。

形式上, 闭区间  $[a, b]$  (其中  $a \leq b$ ) 表示实数  $x$  的集合, 而  $a \leq x \leq b$ 。

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  $[2, 3]$ 。

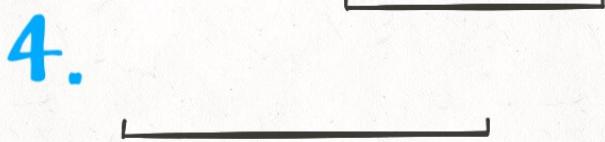
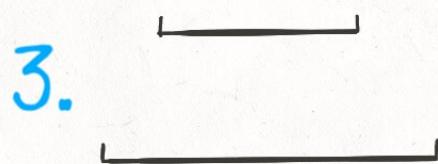
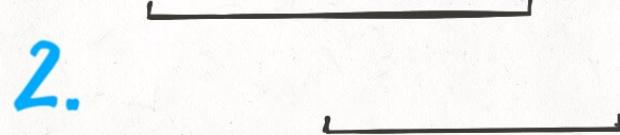
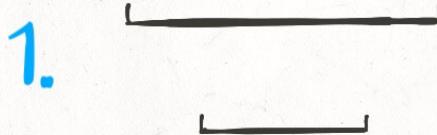
示例 1:



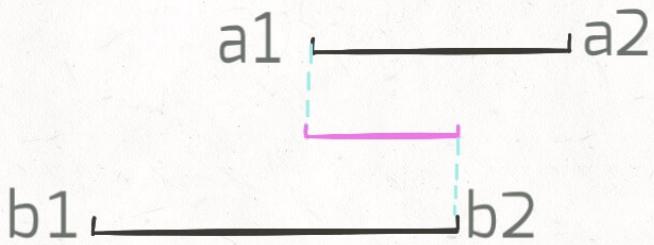
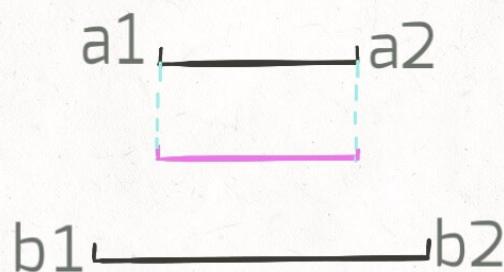
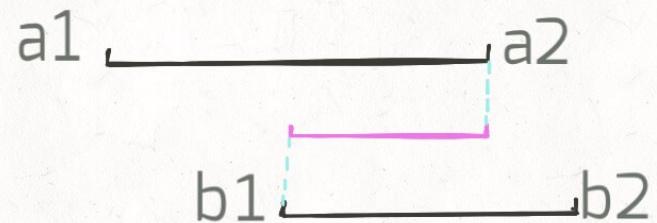
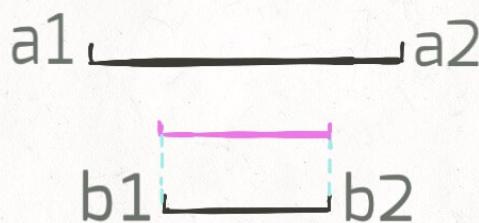
```
输入: firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]  
输出: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

## 基本思路

我们用  $[a_1, a_2]$  和  $[b_1, b_2]$  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  $b_2 \geq a_1$  &  $a_2 \geq b_1$ , 分下面四种情况:



根据上图可以发现规律，假设交集区间是  $[c_1, c_2]$ ，那么  $c_1 = \max(a_1, b_1)$ ,  $c_2 = \min(a_2, b_2)$ :



这一点就是寻找交集的核心。

- 详细题解：一文秒杀所有区间相关问题

## 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]);
}
```

- 类似题目：

- [1288. 删除被覆盖区间](#) (中等)
- [56. 区间合并](#) (中等)

## 2. 两数相加

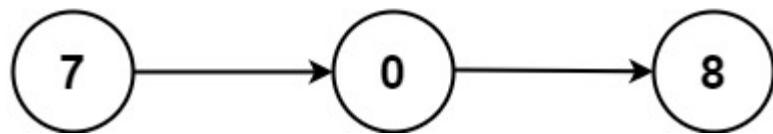
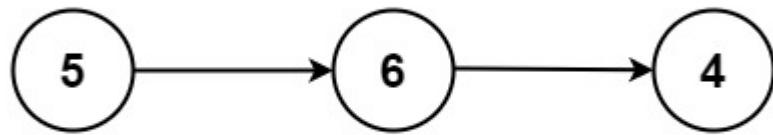
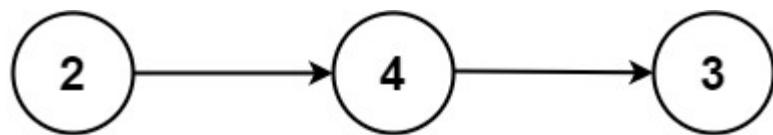


- 标签: 链表双指针, 数据结构

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。

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。

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;
    }
}
```

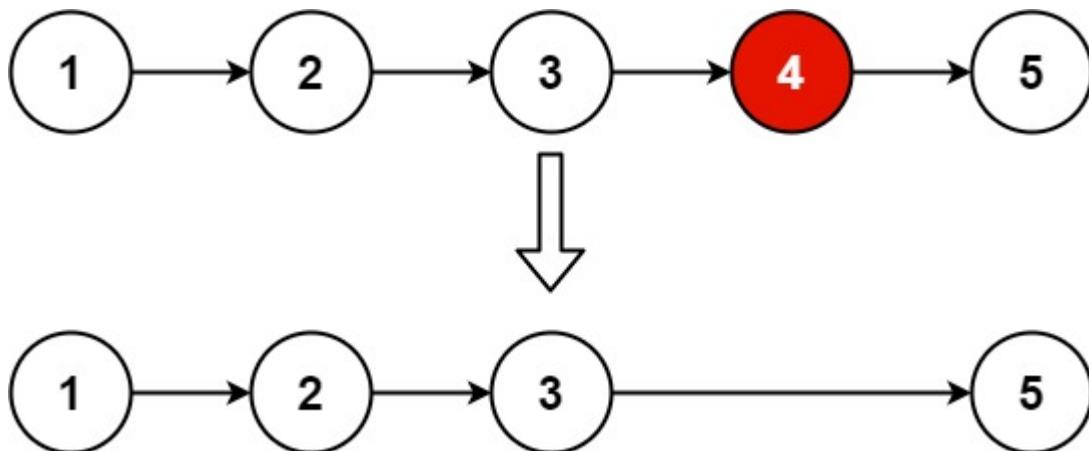
# 19. 删链表的倒数第 N 个结点



- 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]
```

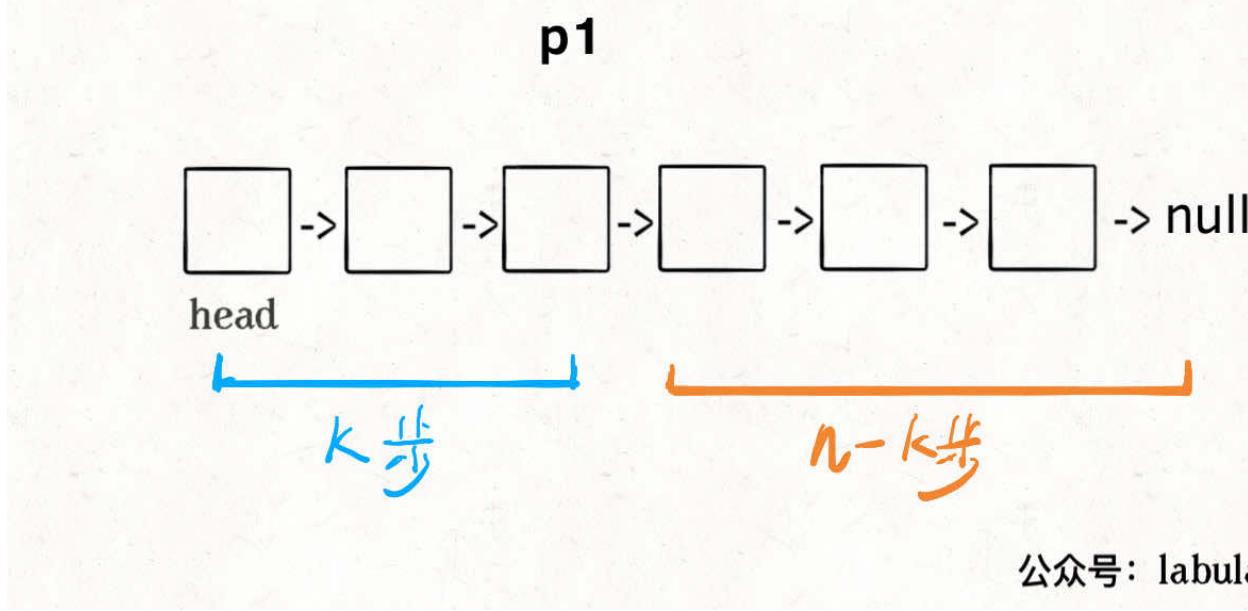
## 基本思路

PS: 这道题在《算法小抄》的第 64 页。

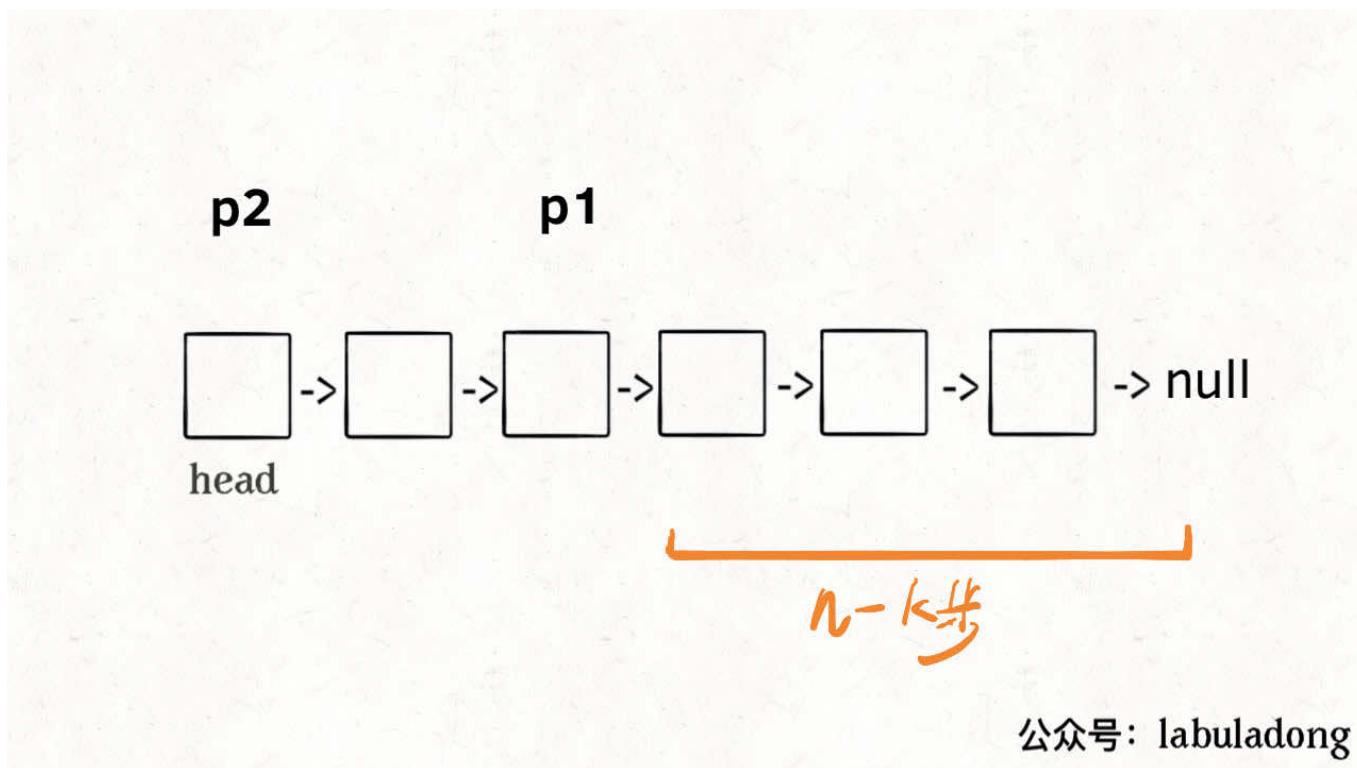
要删除倒数第  $n$  个节点, 就得获得倒数第  $n + 1$  个节点的引用。

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。

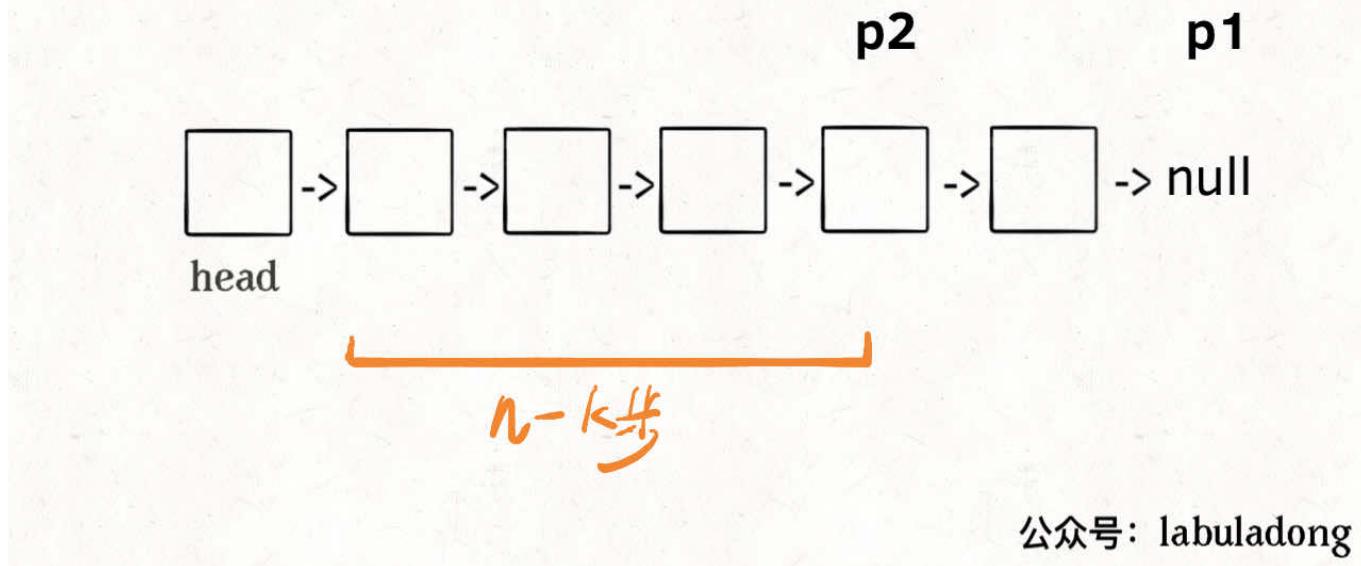
第一步, 我们先让一个指针  $p1$  指向链表的头节点  $head$ , 然后走  $k$  步:



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：单链表的六大解题套路，你都见过么？

## 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;  
}  
}
```

- 类似题目：

- 21. 合并两个有序链表（简单）
- 23. 合并 K 个升序链表（困难）
- 141. 环形链表（简单）
- 142. 环形链表 II（中等）
- 876. 链表的中间结点（简单）
- 160. 相交链表（简单）

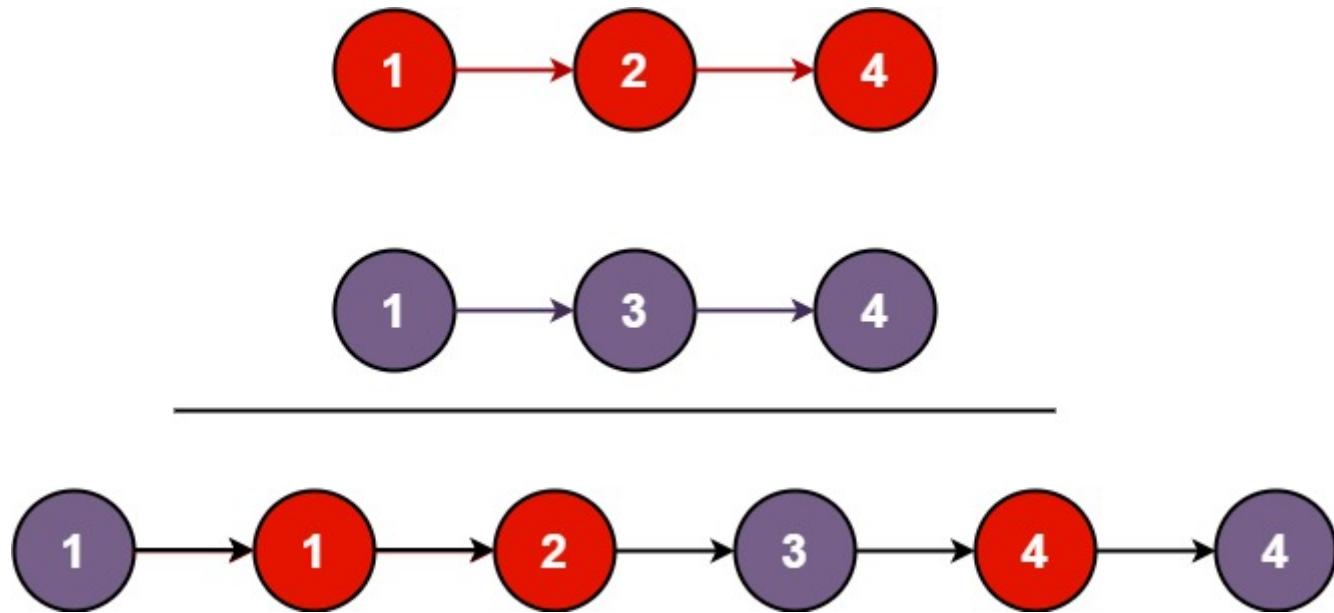
# 21. 合并两个有序链表



- 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:

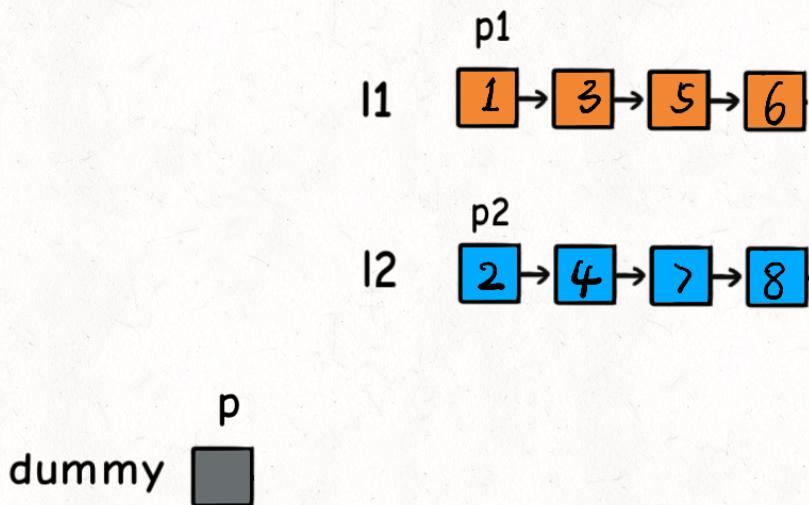


输入: `l1 = [1,2,4], l2 = [1,3,4]`

输出: `[1,1,2,3,4,4]`

## 基本思路

经典算法题了，[双指针技巧](#) 用起来。



公众号： labuladong

这个算法的逻辑类似于「拉拉链」， $l_1$ ,  $l_2$  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  $\text{dummy}$  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：[单链表的六大解题套路，你都见过么？](#)

## 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;
}
}
```

- 类似题目：

- 23. 合并 K 个升序链表（困难）
- 141. 环形链表（简单）
- 142. 环形链表 II（中等）
- 876. 链表的中间结点（简单）
- 160. 相交链表（简单）
- 19. 删除链表的倒数第 N 个结点（中等）

# 23. 合并 K 个升序链表



- 标签: 数据结构, 链表, 链表双指针, 二叉堆

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

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: 单链表的六大解题套路, 你都见过么?

## 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  
            p.next = pq.poll();  
            p = p.next;  
        }  
        return dummy.next;  
    }  
}
```

```
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：

- 21. 合并两个有序链表（简单）
- 141. 环形链表（简单）
- 142. 环形链表 II（中等）
- 876. 链表的中间结点（简单）
- 160. 相交链表（简单）
- 19. 删除链表的倒数第 N 个结点（中等）

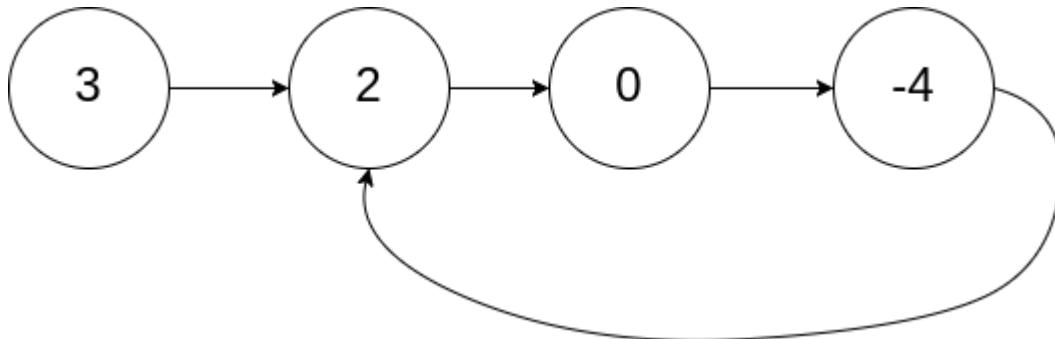
# 141. 环形链表



- 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

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: [单链表的六大解题套路，你都见过么？](#)

## 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;  
            }  
        }  
        return false;  
    }  
}
```

```
        return true;
    }
}
// 不包含环
return false;
}
}
```

- 类似题目：

- 21. 合并两个有序链表（简单）
- 23. 合并 K 个升序链表（困难）
- 142. 环形链表 II（中等）
- 876. 链表的中间结点（简单）
- 160. 相交链表（简单）
- 19. 删除链表的倒数第 N 个结点（中等）

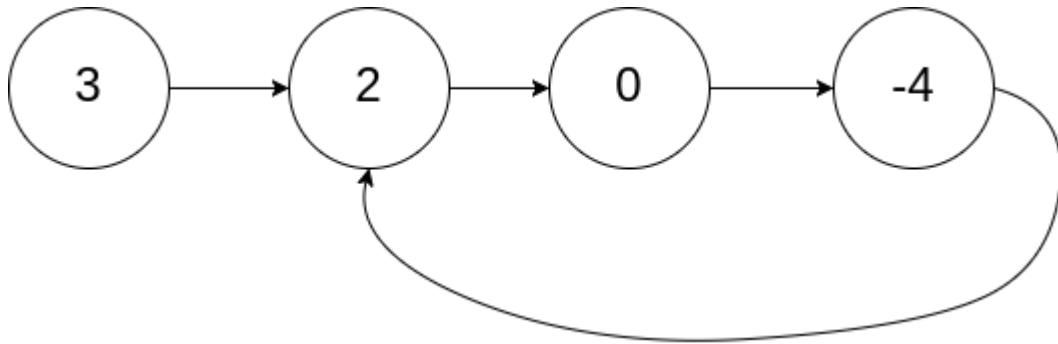
## 142. 环形链表 II



- 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:

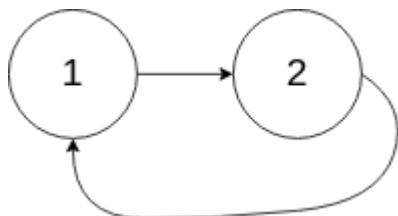


输入: `head = [3,2,0,-4], pos = 1`

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`

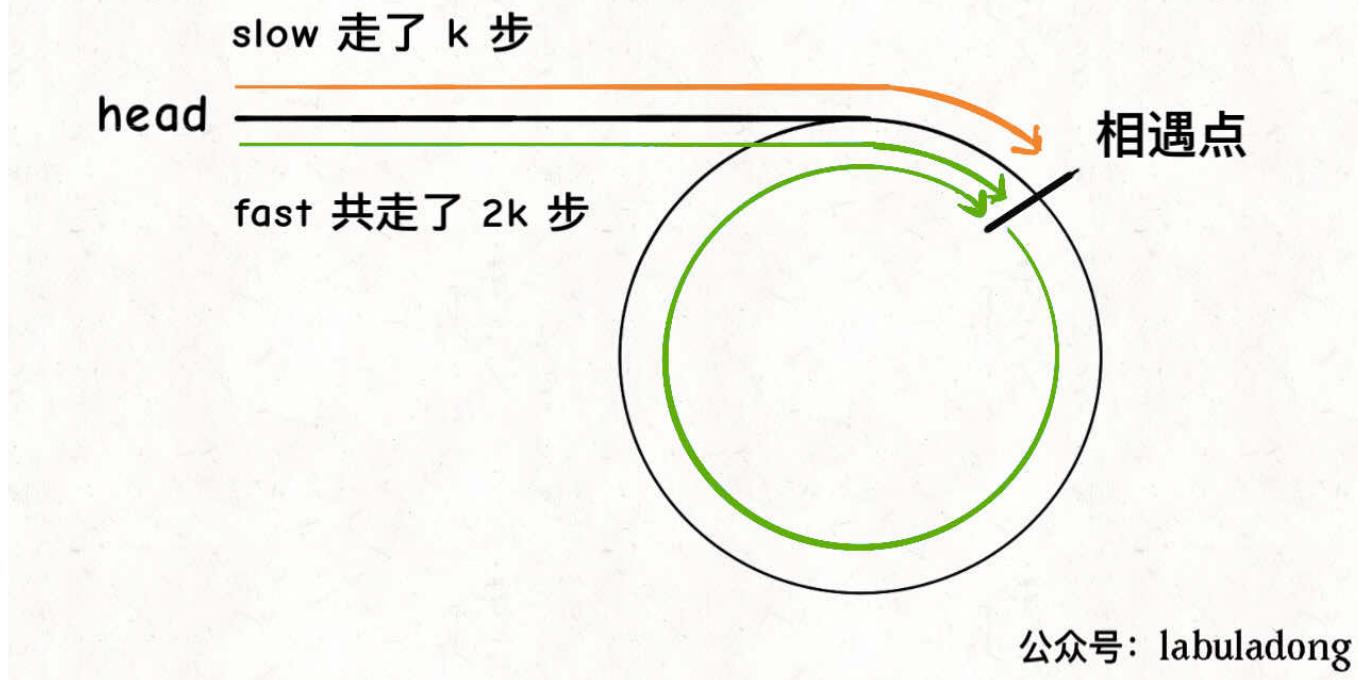
输出: 返回索引为 0 的链表节点

解释: 链表中有一个环, 其尾部连接到第一个节点。

### 基本思路

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。

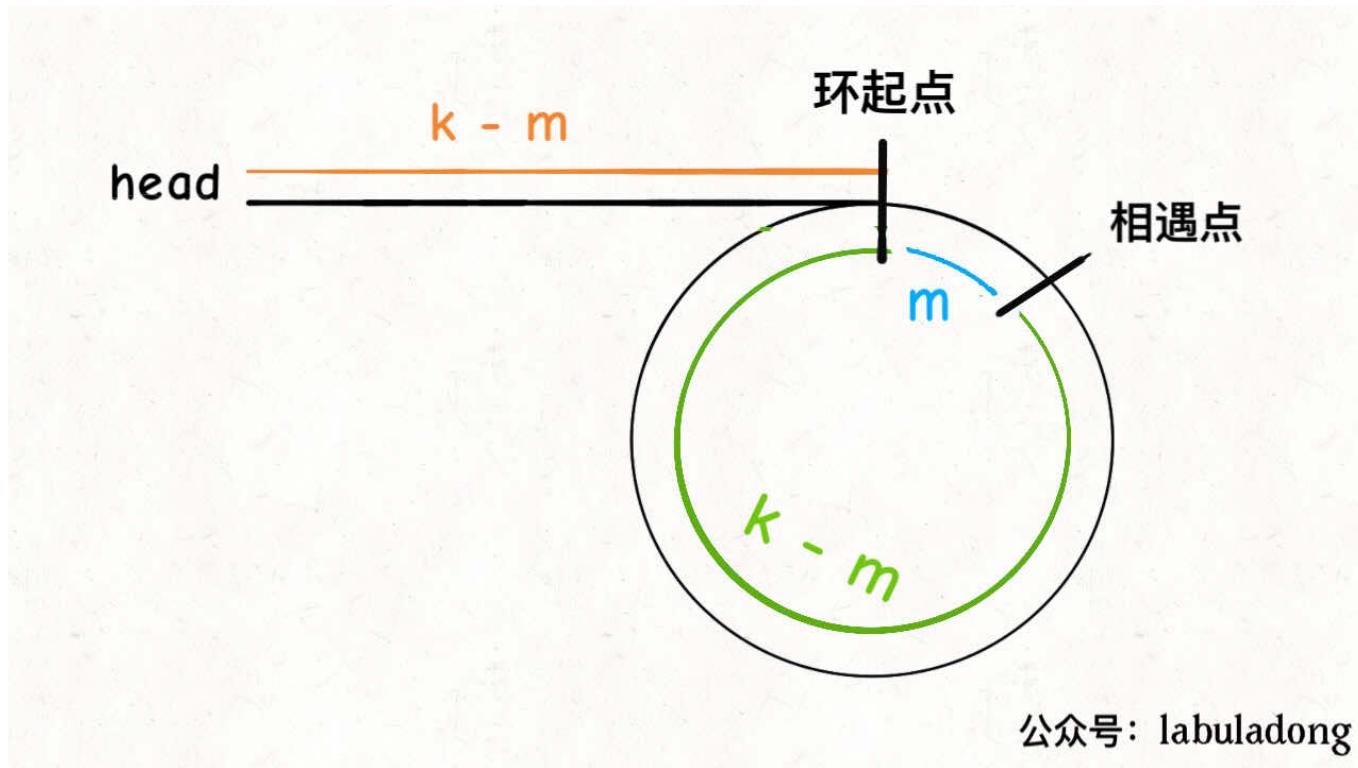
原理也简单说下吧，我们假设快慢指针相遇时，慢指针 `slow` 走了 `k` 步，那么快指针 `fast` 一定走了 `2k` 步：



**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：单链表的六大解题套路，你都见过么？

## 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;  
    }  
}
```

- 类似题目：

- 21. 合并两个有序链表（简单）
- 23. 合并 K 个升序链表（困难）
- 141. 环形链表（简单）
- 876. 链表的中间结点（简单）
- 160. 相交链表（简单）
- 19. 删除链表的倒数第 N 个结点（中等）

# 160. 相交链表

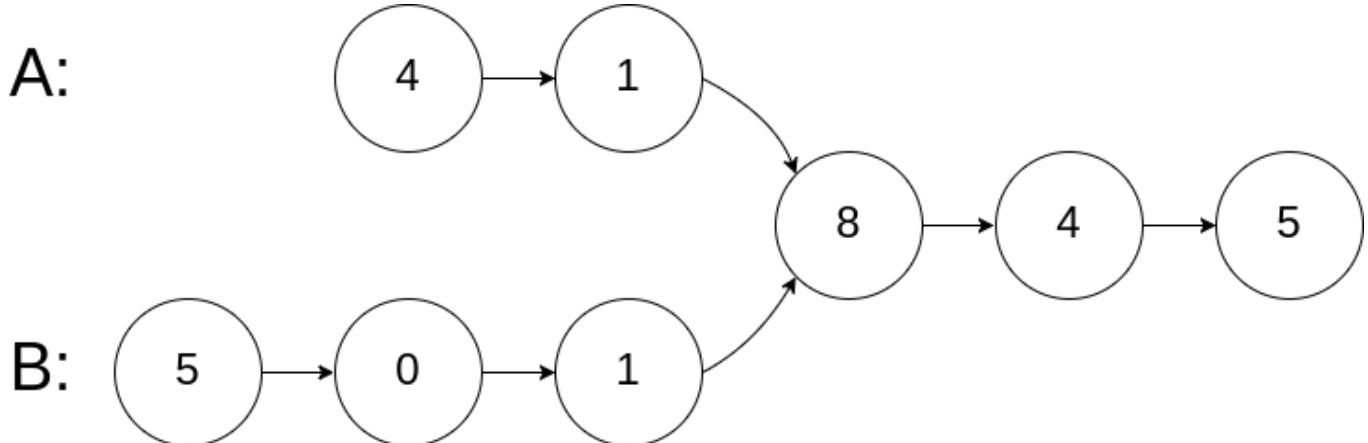


- 标签: 数据结构, 链表, 链表双指针

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
```

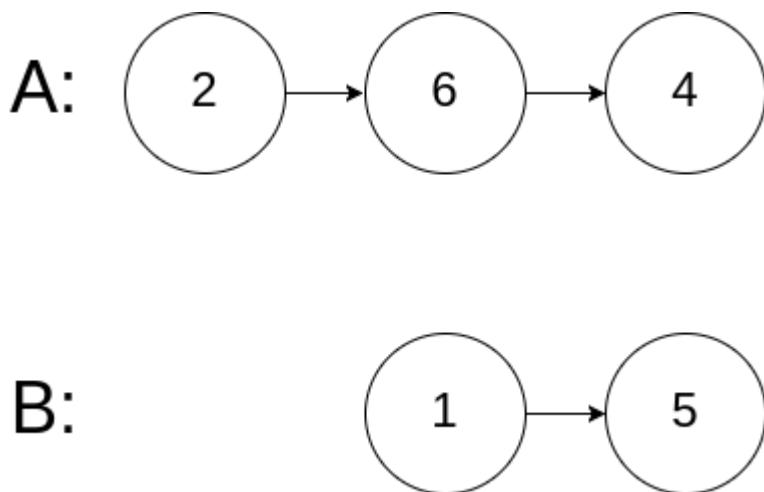
输出: Intersected at '8'

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:

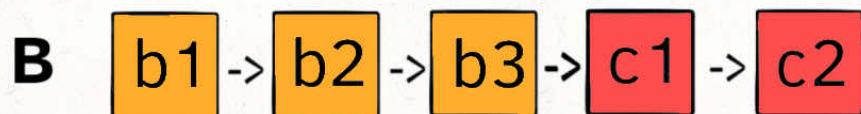


输入: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2  
输出: null  
解释: 从各自的表头开始算起, 链表 A 为 [2,6,4], 链表 B 为 [1,5]。  
由于这两个链表不相交, 所以 intersectVal 必须为 0, 而 skipA 和 skipB 可以是任意值。  
这两个链表不相交, 因此返回 null。

## 基本思路

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:



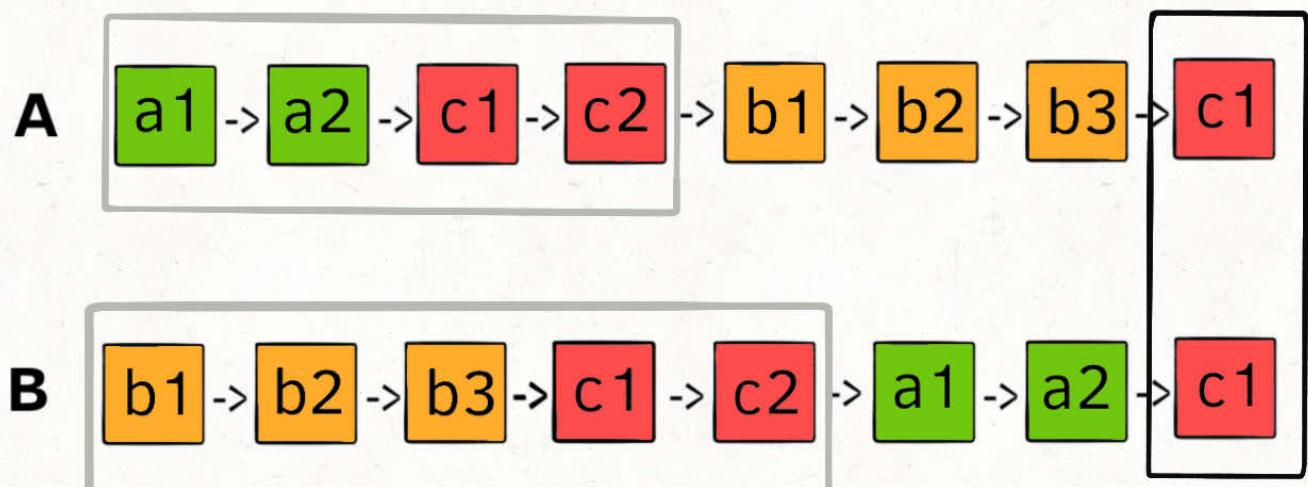
公众号: labuladong

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」两条链表接在一起了。

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  $c1$ :



公众号: labuladong

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：单链表的六大解题套路，你都见过么？

## 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;
    }
}
```

- 类似题目：
  - 21. 合并两个有序链表（简单）
  - 23. 合并 K 个升序链表（困难）
  - 141. 环形链表（简单）
  - 142. 环形链表 II（中等）
  - 876. 链表的中间结点（简单）
  - 19. 删除链表的倒数第 N 个结点（中等）

# 876. 链表的中间结点



- 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

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: 单链表的六大解题套路, 你都见过么?

## 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;
}
```

- 类似题目：

- 21. 合并两个有序链表（简单）
- 23. 合并 K 个升序链表（困难）
- 141. 环形链表（简单）
- 142. 环形链表 II（中等）
- 160. 相交链表（简单）
- 19. 删除链表的倒数第 N 个结点（中等）

# 25. K 个一组翻转链表



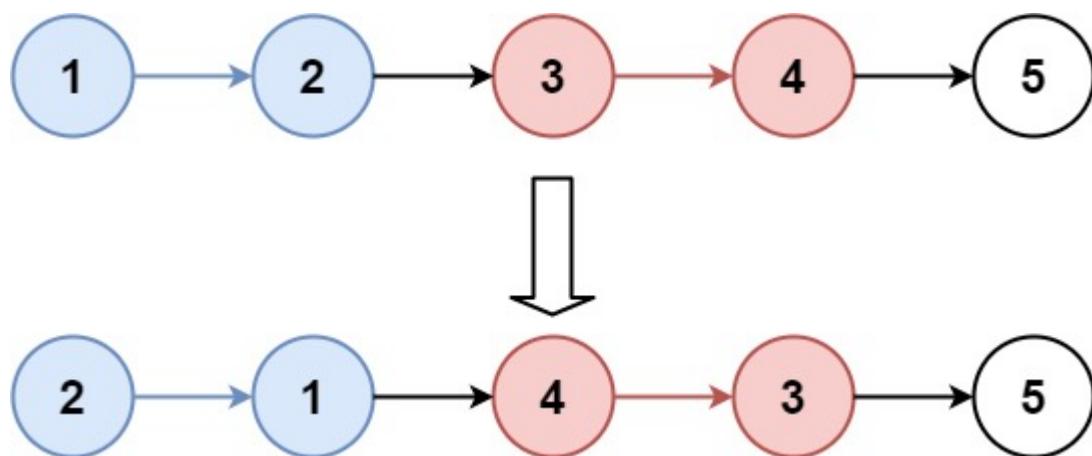
- 标签: 数据结构, 链表, 链表双指针

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]
```

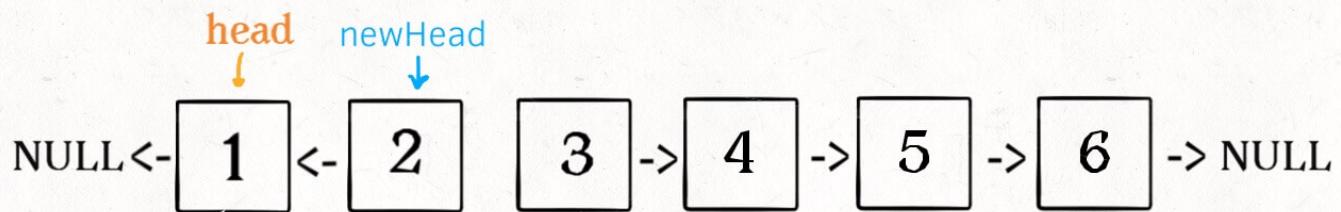
## 基本思路

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。

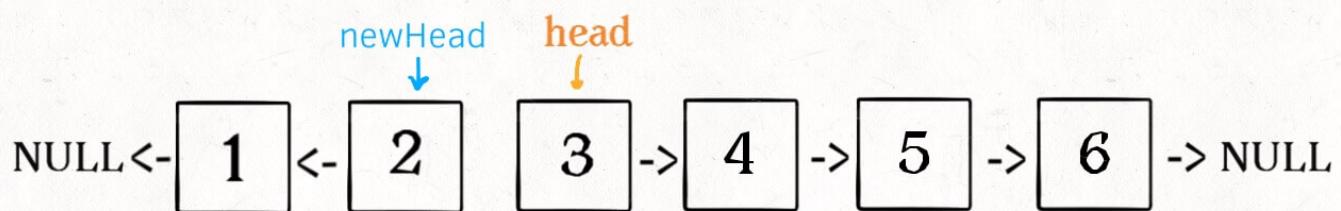
我们要充分利用这个递归函数的定义, 把原问题分解成规模更小的子问题进行求解。

1、先反转以 `head` 开头的  $k$  个元素。



公众号: labuladong

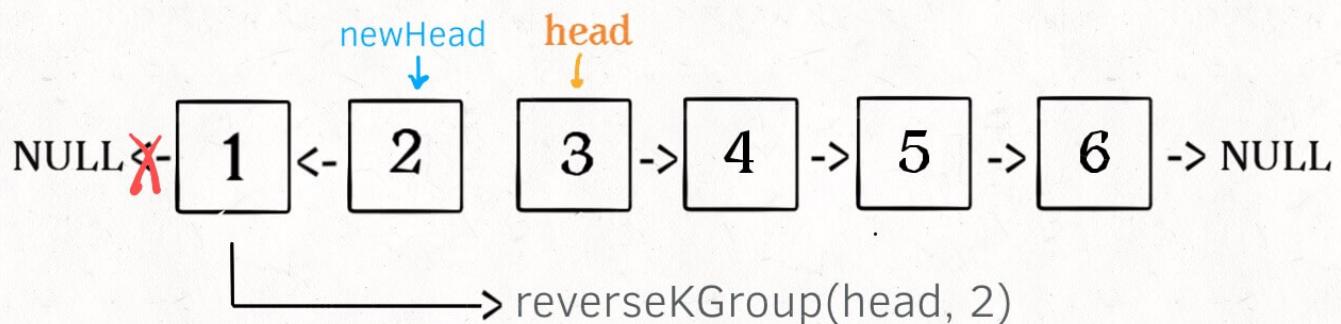
2、将第  $k + 1$  个元素作为 `head` 递归调用 `reverseKGroup` 函数。



`reverseKGroup(head, 2)`

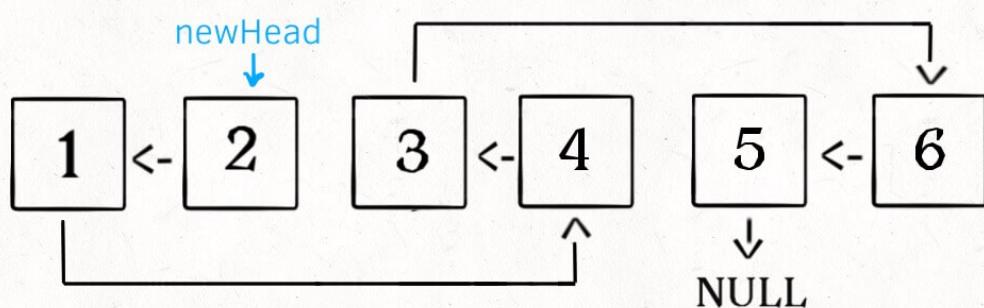
公众号: labuladong

3、将上述两个过程的结果连接起来。



公众号: labuladong

最后函数递归完成之后就是这个结果，完全符合题意：



公众号: labuladong

- 详细题解：递归思维：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;
}
}
```

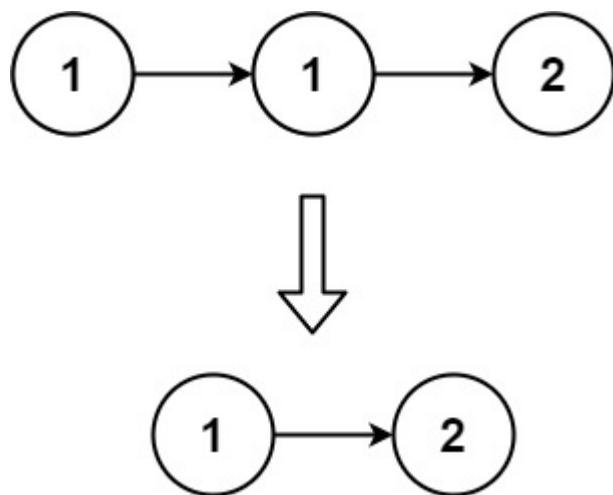
## 83. 删除排序链表中的重复元素



- 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:



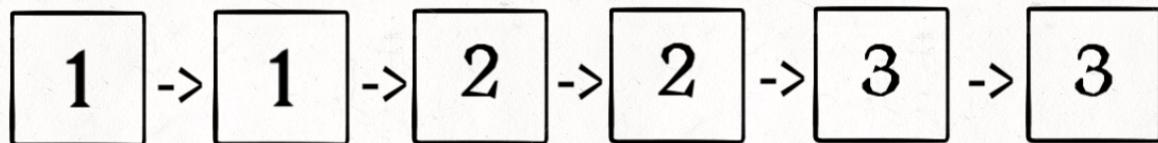
```
输入: head = [1,1,2]
输出: [1,2]
```

### 基本思路

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。

head



公众号: labuladong

- 详细题解: 双指针技巧秒杀四道数组/链表题目

## 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;
    }
}
```

- 类似题目:
  - 26. 删除有序数组中的重复项 (简单)
  - 27. 移除元素 (简单)
  - 283. 移动零 (简单)

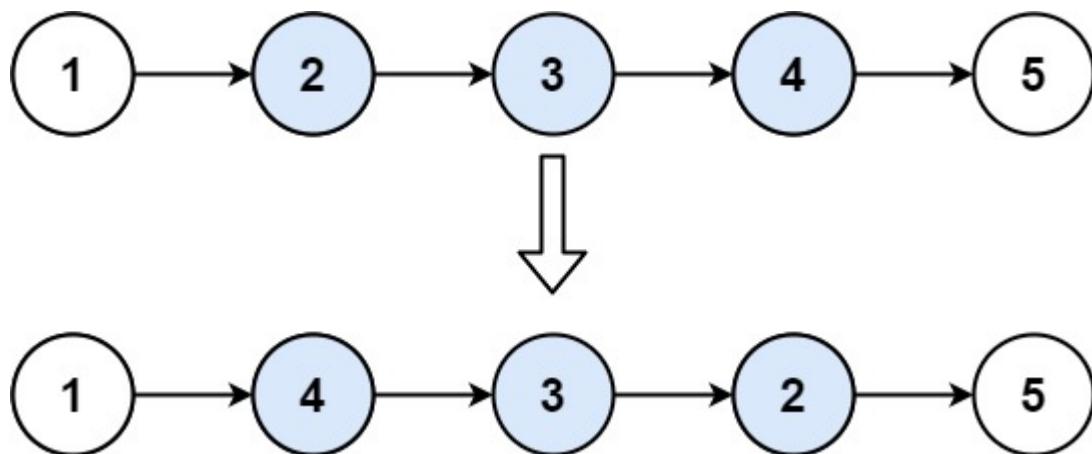
## 92. 反转链表 II



- 标签: 数据结构, 链表, 递归, 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: 递归反转链表: 如何拆解复杂问题

### 解法代码

```
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;
}
}
```

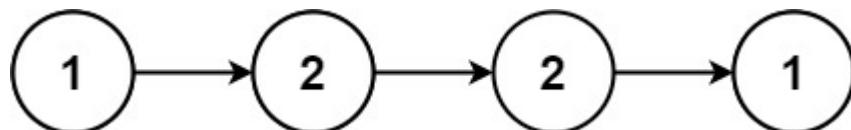
## 234. 回文链表



- 标签: 数据结构, 链表, 回文问题, 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

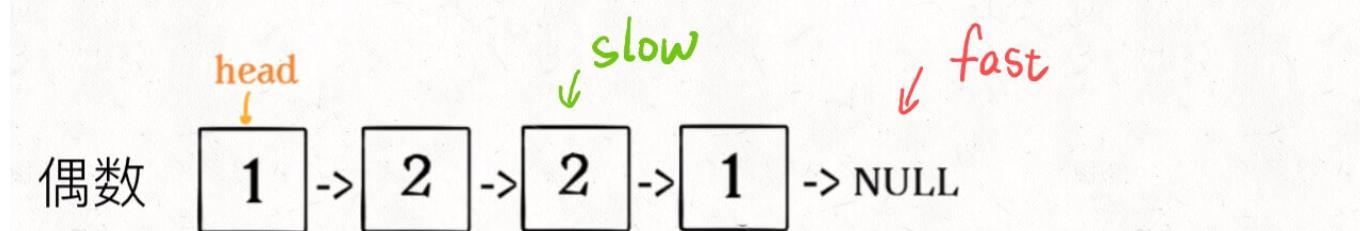
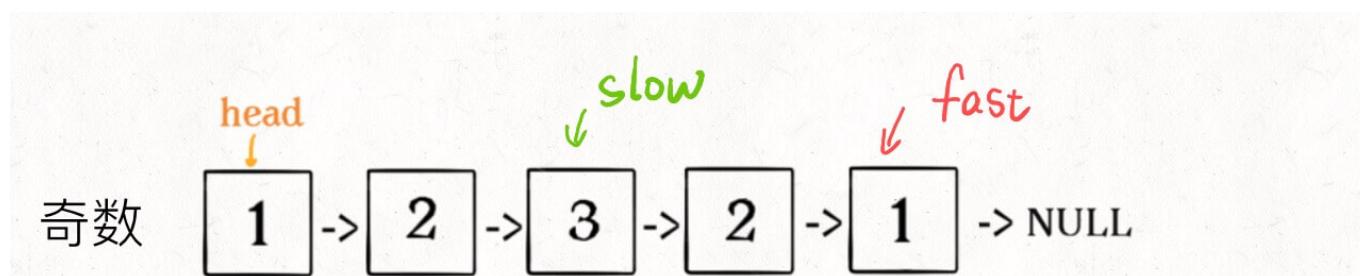
PS: 这道题在《算法小抄》的第 277 页。

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。

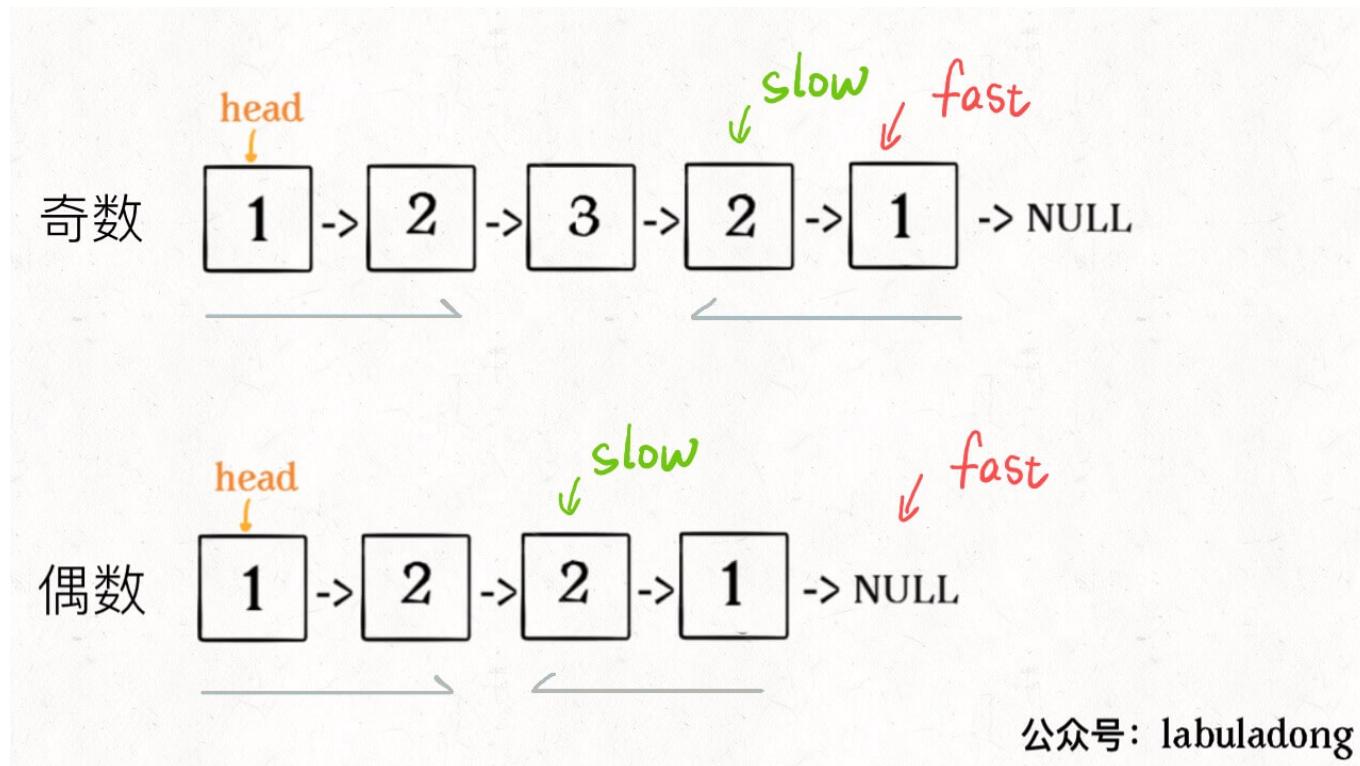
更聪明一些的办法是借助双指针算法:

1、先通过 双指针技巧 中的快慢指针来找到链表的中点:

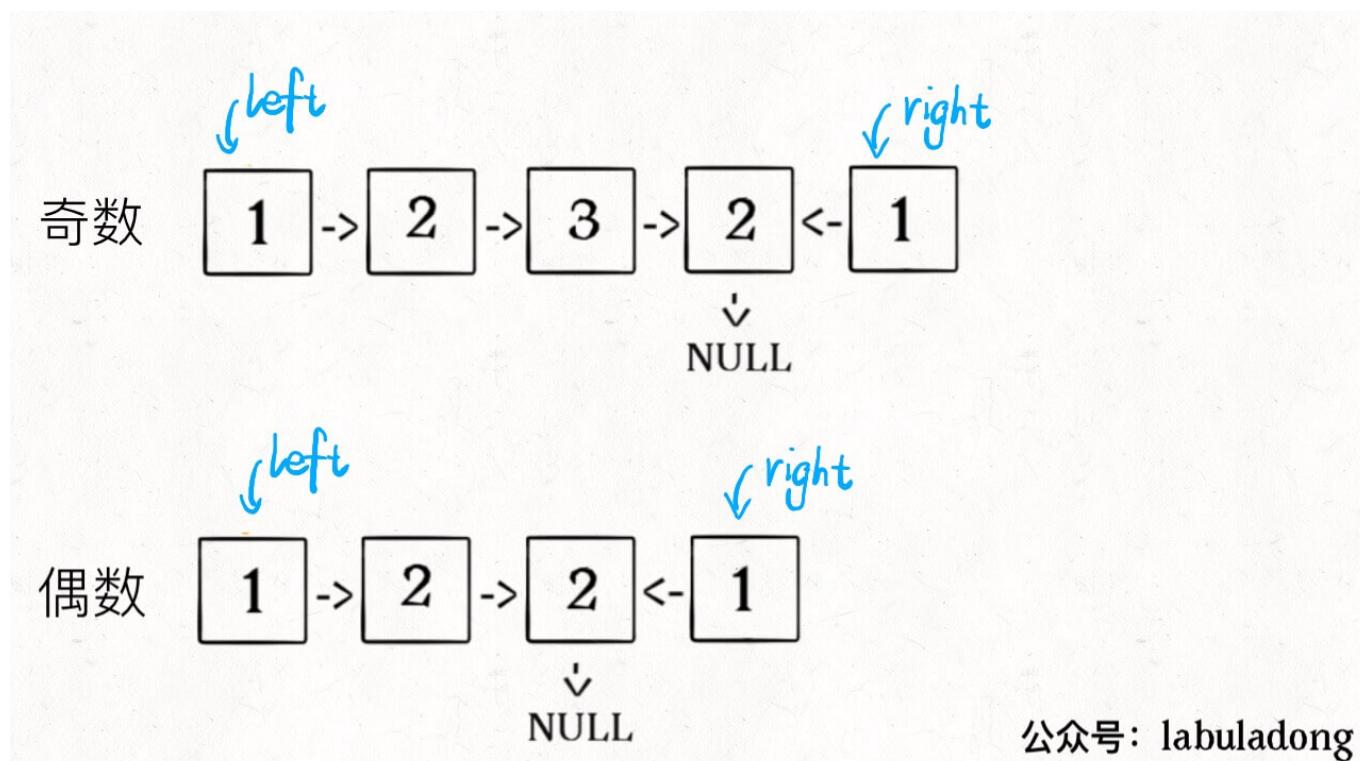


公众号: labuladong

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: 如何高效判断回文单链表?

## 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;
}
}
```

# 303. 区域和检索 - 数组不可变



- 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

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  $10 = 3 + 5 + 2$ :

	0	1	2	3	4	5
nums	3	5	2	-2	4	1

	0	1	2	3	4	5	6
preSum	0	3	8	10	8	12	13

公众号: labuladong

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 = 0; i < preSum.length - 1; i++) {
            preSum[i + 1] = preSum[i] + nums[i];
        }
    }

    /* 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（中等）](#)
- [560. 和为K的子数组（中等）](#)

## 304. 二维区域和检索 - 矩阵不可变



- 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:

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

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]
```

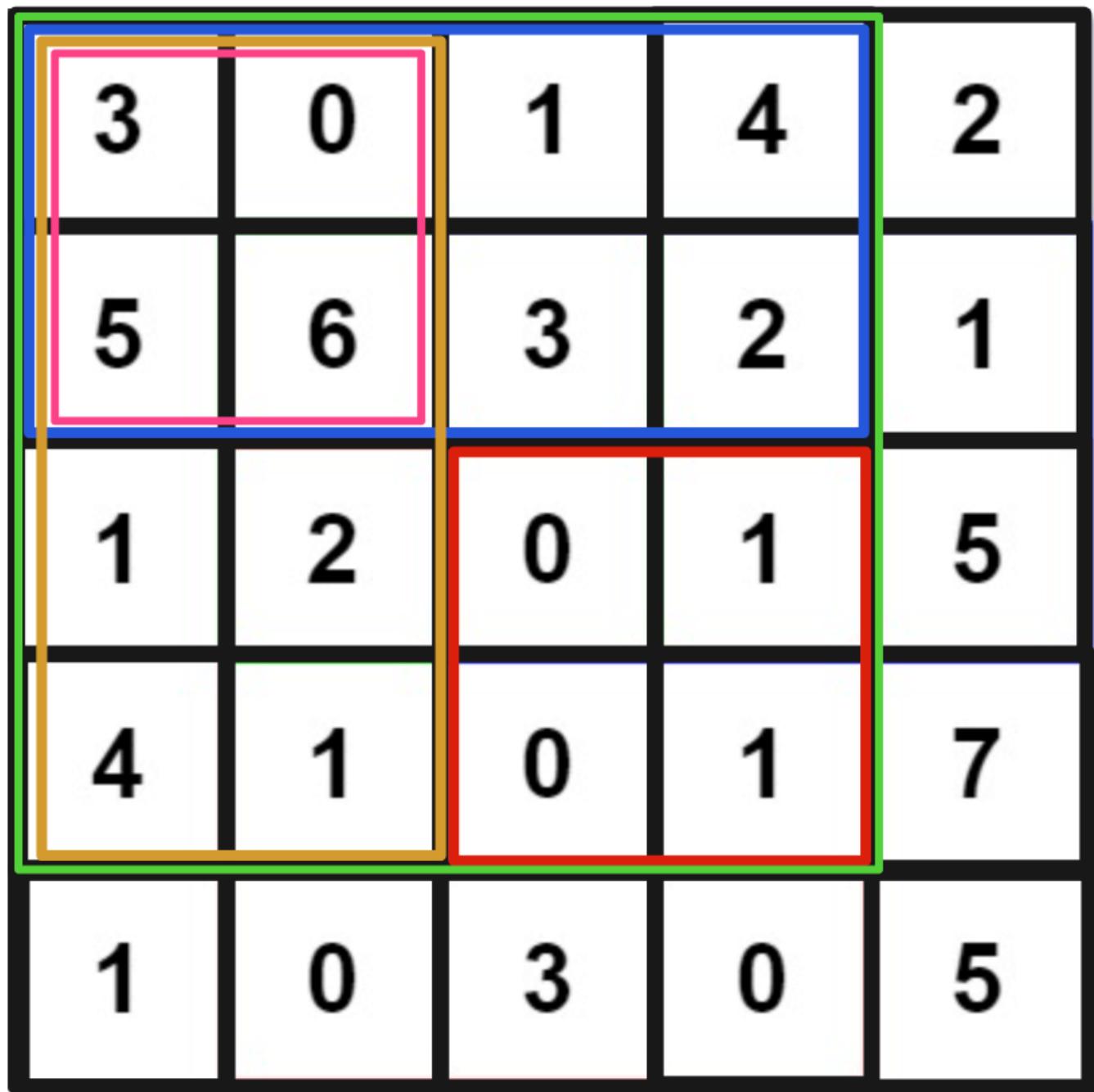
解释:

```
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 (绿色矩阵框的元素总和)  
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩阵框的元素总和)
```

## 基本思路

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, j] 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：
  - 303. 区域和检索 - 数组不可变（中等）
  - 560. 和为K的子数组（中等）

# 560. 和为 K 的子数组



- 标签: 前缀和, 哈希表, 数组

给你一个整数数组 `nums` 和一个整数 `k`, 请你统计并返回该数组中和为 `k` 的连续子数组的个数。

示例 1:

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

## 基本思路

PS: 这道题在《算法小抄》的第 341 页。

子数组求和的经典技巧就是前缀和, 原理就是对数组进行预处理, 计算前缀和数组, 从而在  $O(1)$  时间计算子数组和。

```
int n = nums.length;
// 前缀和数组
int[] preSum = new int[n + 1];
preSum[0] = 0;
for (int i = 0; i < n; i++)
    preSum[i + 1] = preSum[i] + nums[i];
```

`preSum[i]` 就是 `nums[0..i-1]` 的和, 如果想求 `nums[i..j]` 的和, 只需要 `preSum[j+1] - preSum[i]` 即可:

	0	1	2	3	4	5
nums	3	5	2	-2	4	1

	0	1	2	3	4	5	6
preSum	0	3	8	10	8	12	13

公众号: labuladong

对于这道题，把前缀和和哈希表结合起来，时间复杂度为  $O(N)$ 。

- 详细题解: 前缀和技巧: 解决子数组问题

## 解法代码

```

class Solution {
    public int subarraySum(int[] nums, int k) {
        int n = nums.length;
        // map: 前缀和 -> 该前缀和出现的次数
        HashMap<Integer, Integer>
            preSum = new HashMap<>();
        // base case
        preSum.put(0, 1);

        int res = 0, sum0_i = 0;
        for (int i = 0; i < n; i++) {
            sum0_i += nums[i];
            // 这是我们想找的前缀和 nums[0..j]
            int sum0_j = sum0_i - k;
            // 如果前面有这个前缀和，则直接更新答案
            if (preSum.containsKey(sum0_j))
                res += preSum.get(sum0_j);
            // 把前缀和 nums[0..i] 加入并记录出现次数
            preSum.put(sum0_i,
                preSum.getOrDefault(sum0_i, 0) + 1);
        }
        return res;
    }
}

```

# 370. 区间加法



- 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:  $[\text{startIndex}, \text{endIndex}, \text{inc}]$ ，你需要将子数组  $A[\text{startIndex} \dots \text{endIndex}]$  (包括  $\text{startIndex}$  和  $\text{endIndex}$ ) 增加  $\text{inc}$ 。

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：

- [1109. 航班预订统计（中等）](#)
- [1094. 拼车](#)

# 1094. 拼车



- 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();
        for (int num : res) {
            if (num > capacity) {
                return false;
            }
        }
        return true;
    }
}
```

```
// 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：

- [370. 区间加法](#) (中等)
- [1109. 航班预订统计](#) (中等)

# 1109. 航班预订统计



- 标签: 数组, 差分数组

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  $\text{bookings}$ ，表中第  $i$  条预订记录  $\text{bookings}[i] = [\text{first}_i, \text{last}_i, \text{seats}_i]$  意味着在从  $\text{first}_i$  到  $\text{last}_i$  (包含  $\text{first}_i$  和  $\text{last}_i$ ) 的每个航班上预订了  $\text{seats}_i$  个座位。

请你返回一个长度为  $n$  的数组  $\text{answer}$ ，里面的元素是每个航班预定的座位总数。

示例 1:

输入:  $\text{bookings} = [[1, 2, 10], [2, 3, 20], [2, 5, 25]]$ ,  $n = 5$

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,  $\text{answer} = [10, 55, 45, 25, 25]$

## 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];
}
```

nums	8	2	6	3	1
------	---	---	---	---	---

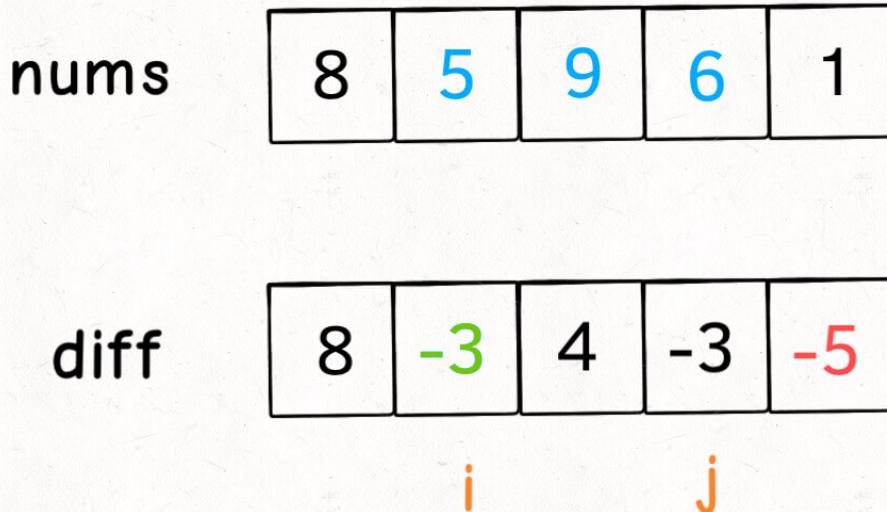
diff	8	-6	4	-3	-2
------	---	----	---	----	----

公众号: labuladong

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：



公众号: labuladong

本题就相当于给你输入一个长度为  $n$  的数组  $\text{nums}$ , 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: 论那些小而美的算法技巧: 差分数组/前缀和

## 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;
}
}

}
```

- 类似题目：

- [370. 区间加法 \(中等\)](#)
- [1094. 拼车](#)

# 20. 有效的括号



- 标签: 栈, 括号问题

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: true

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: 手把手解决三道括号相关的算法题

## 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;
        }
        return left.isEmpty();
    }

    private char leftOf(char c) {
        if (c == ')') return '(';
        if (c == '}') return '{';
        if (c == ']') return '[';
        return '\0';
    }
}
```

```
    }
    // 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 '[';
}
}
```

- 类似题目：

- [921. 使括号有效的最小添加](#) (中等)
- [1541. 平衡括号串的最少插入](#) (中等)

# 921. 使括号有效的最少添加



- 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: 手把手解决三道括号相关的算法题

## 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;
    }
}
```

- 类似题目：

- 20. 有效的括号（简单）
- 1541. 平衡括号串的最少插入（中等）

# 1541. 平衡括号字符串的最少插入次数



- 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：手把手解决三道括号相关的算法题

## 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;
}
}
```

- 类似题目：

- [20. 有效的括号（简单）](#)
- [921. 使括号有效的最小添加（中等）](#)

## 32. 最长有效括号



- 标签: 括号问题, 栈

给你一个只包含 '(', ')' 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 = 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

```
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;
}
```

# 225. 用队列实现栈



- 标签: 数据结构, 队列, 栈

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`)。

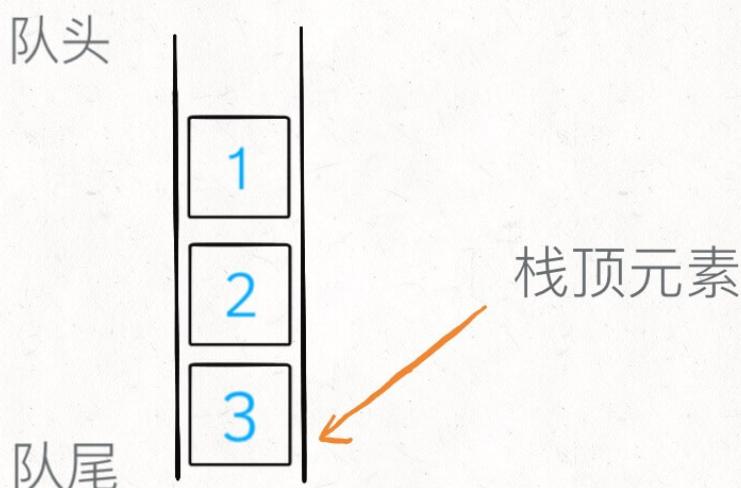
实现 `MyStack` 类：

- 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：



公众号: labuladong

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：

队头



队尾

公众号: labuladong

- 详细题解: 数据结构设计: 用栈实现队列/用队列实现栈

## 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（简单）](#)

## 232. 用栈实现队列



- 标签: 数据结构, 队列, 栈

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`) :

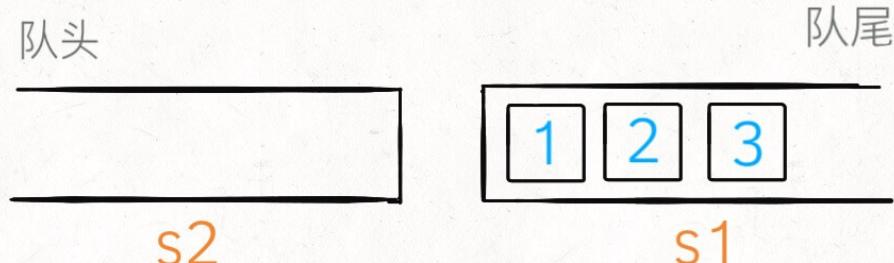
实现 `MyQueue` 类:

- 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。

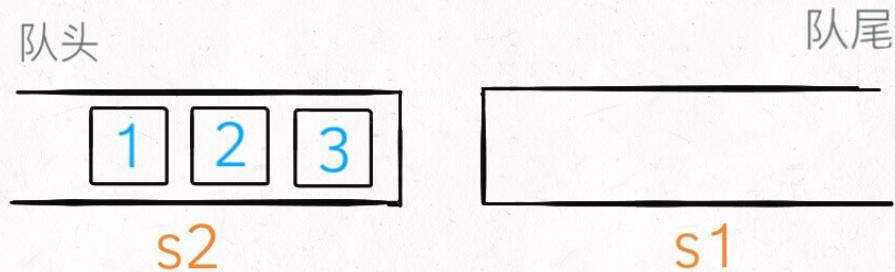
当调用 `push` 让元素入队时, 只要把元素压入 `s1` 即可:



双栈实现的队列

公众号: labuladong

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:



双栈实现的队列

公众号: labuladong

- 详细题解: 数据结构设计: 用栈实现队列/用队列实现栈

## 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（简单）](#)

## 239. 滑动窗口最大值



- 标签: 数据结构, 队列, 滑动窗口

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]`

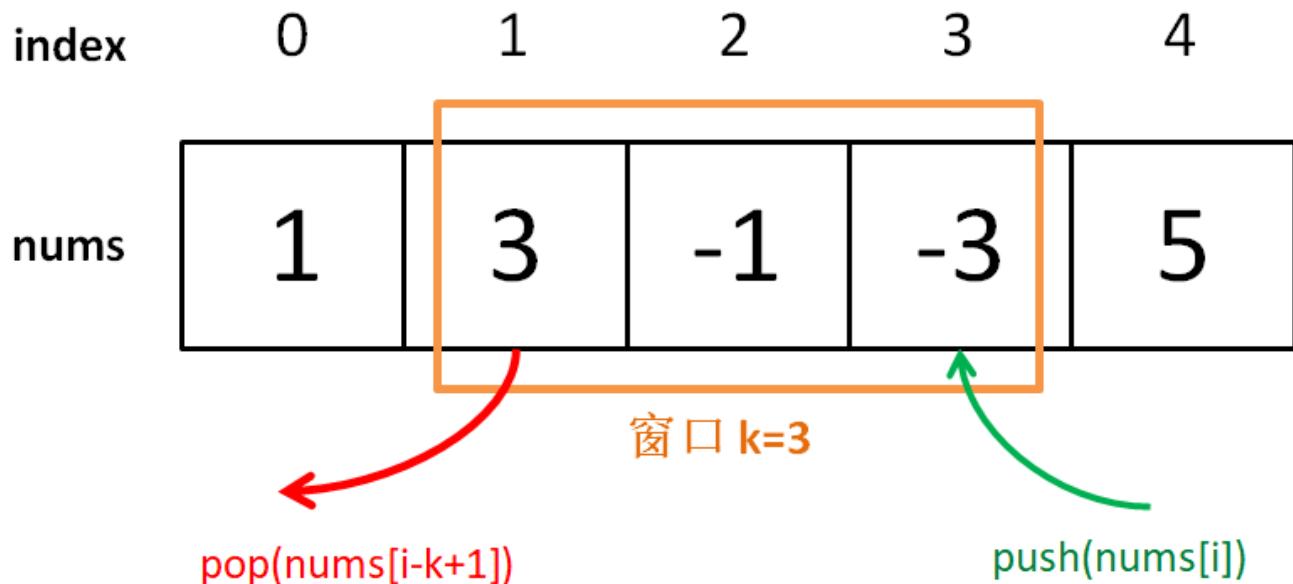
解释:

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

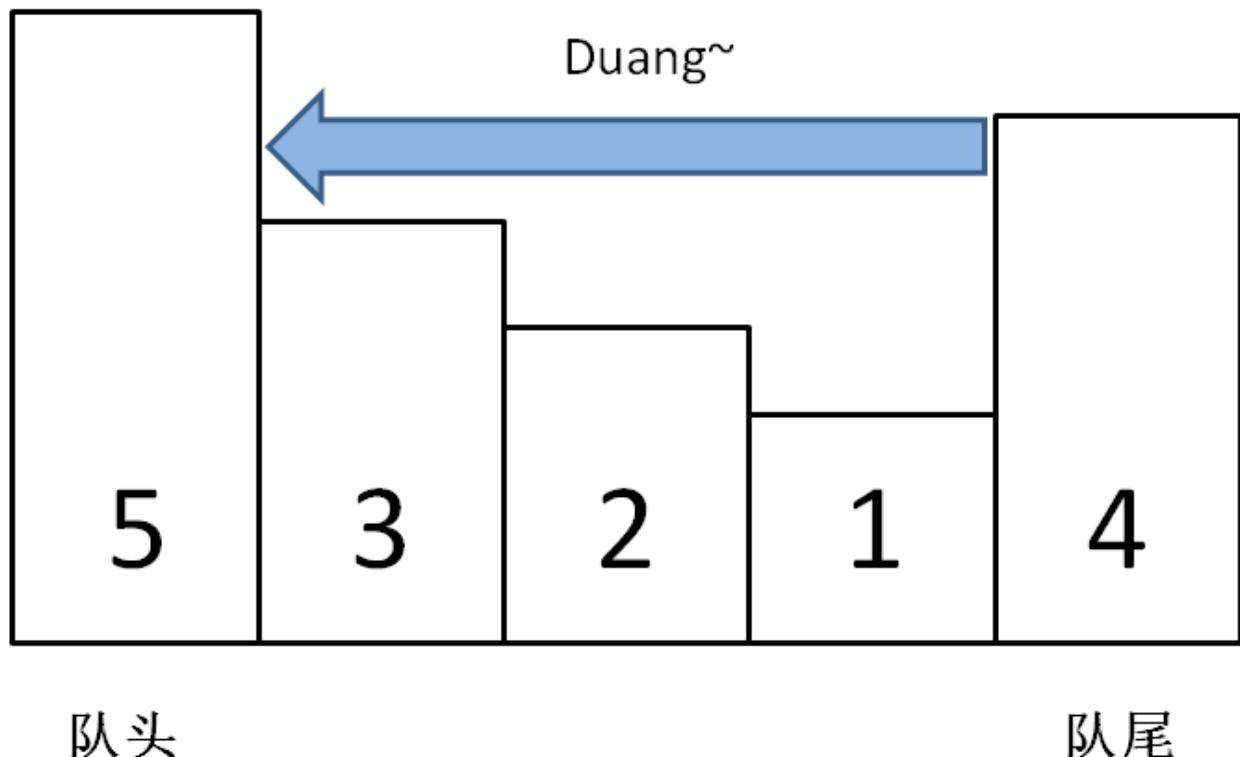
## 基本思路

PS: 这道题在《算法小抄》的第 271 页。

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」数据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;
    }
}
```

# 23. 合并 K 个升序链表



- 标签: 数据结构, 链表, 链表双指针, 二叉堆

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

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: 单链表的六大解题套路, 你都见过么?

## 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  
            p.next = pq.poll();  
            p = p.next;  
        }  
        return dummy.next;  
    }  
}
```

```
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：

- [21. 合并两个有序链表](#) (简单)
- [141. 环形链表](#) (简单)
- [142. 环形链表 II](#) (中等)
- [876. 链表的中间结点](#) (简单)
- [160. 相交链表](#) (简单)
- [19. 删除链表的倒数第 N 个结点](#) (中等)

# 215. 数组中的第 K 个最大元素



- 标签: 二叉堆, 数组, 快速选择算法

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  $O(\log k)$ , 再套一层 for 循环, 总的时间复杂度就是  $O(N \log k)$ 。

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: 快排亲兄弟: 快速选择算法详解

## 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();
}
```

# 295. 数据流的中位数



- 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。

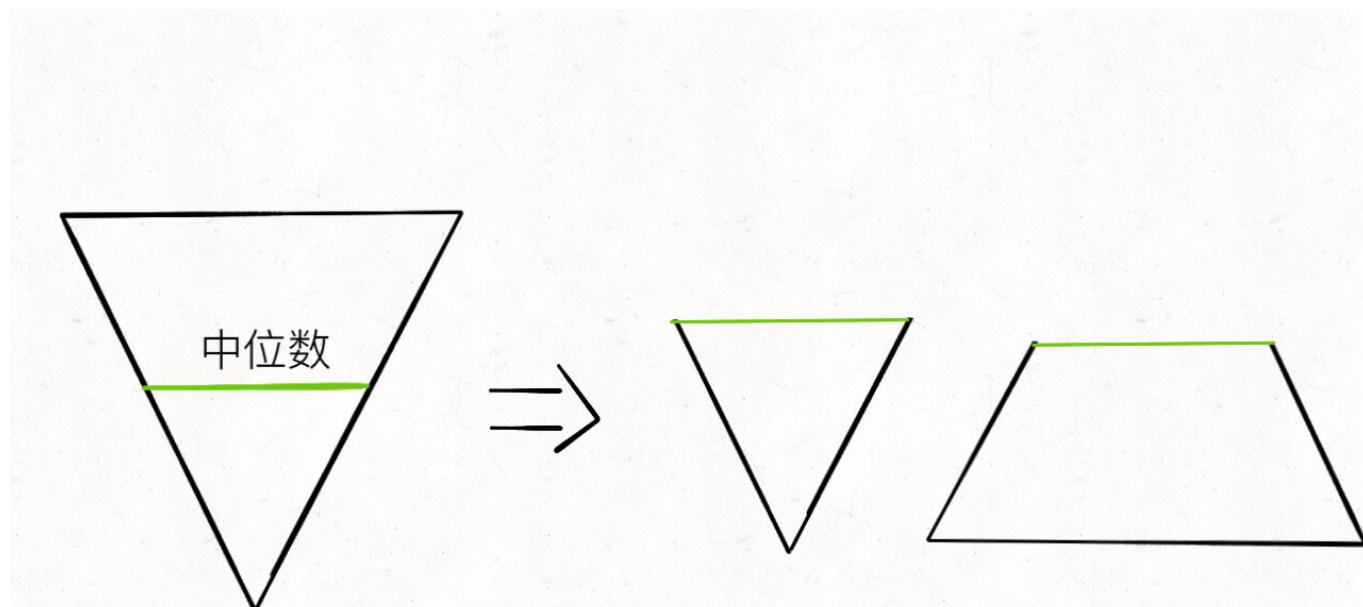
2、`double findMedian()` 返回目前所有元素的中位数。

示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

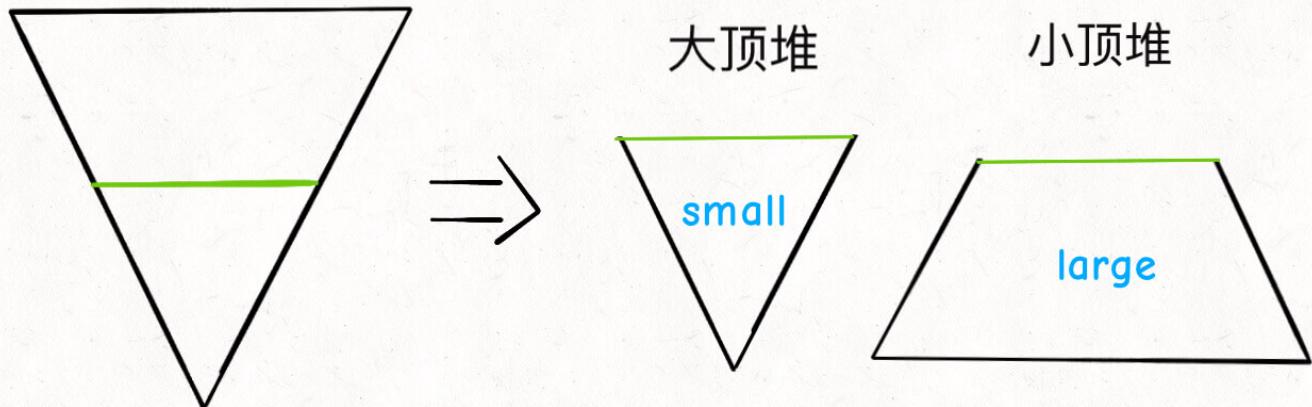
## 基本思路

本题的核心思路是使用两个优先级队列。



公众号: labuladong

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：



公众号: labuladong

- 详细题解: 啊这, 一道找中位数的算法题把东哥整不会了...

## 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);
        } else {
            large.offer(num);
        }
    }
}
```

```
        large.offer(small.poll());
    } else {
        large.offer(num);
        small.offer(large.poll());
    }
}
```

# 703. 数据流中的第 K 大元素



- 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", "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();
}
}
```

# 146. LRU 缓存机制



- 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

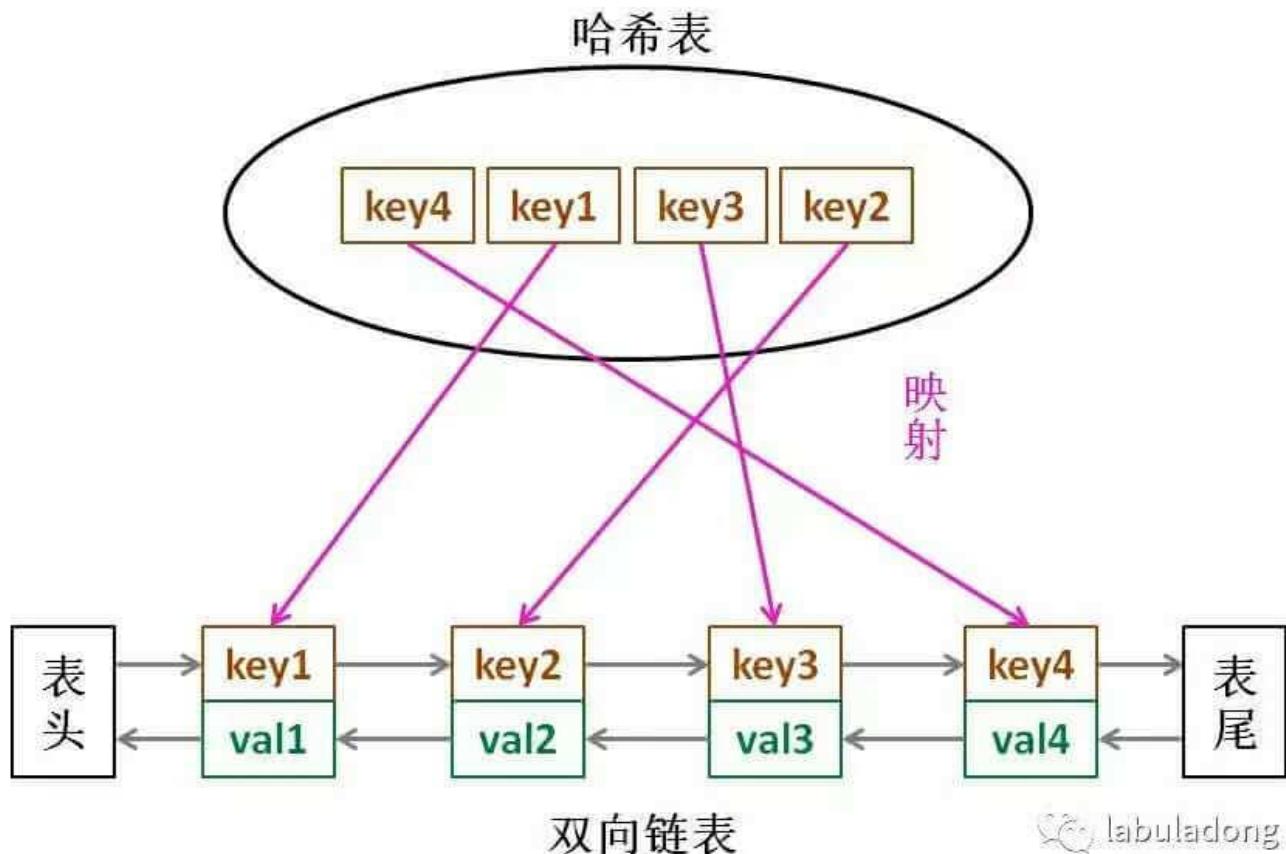
PS: 这道题在《算法小抄》的第 216 页。

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**：



 labuladong

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：[算法题就像搭乐高：手把手带你拆解 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 打到队尾
        makeRecently(key);
        return cache.get(key);
    }

    public void put(int key, int value) {
        if (cache.containsKey(key)) {
            // 修改已有 key 的值
            cache.put(key, value);
            // 将 key 打到队尾
            makeRecently(key);
        } else {
            // 插入新 key
            cache.put(key, value);
            if (cache.size() > cap) {
                // 移除最久未使用的 key
                Integer oldestKey = cache.keySet().iterator().next();
                cache.remove(oldestKey);
            }
        }
    }

    private void makeRecently(int key) {
        // 将指定 key 提到队尾
        cache.remove(key);
        cache.put(key, cache.get(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);
}
}
```

# 341. 扁平化嵌套列表迭代器



- 标签: 数据结构, 二叉树, 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;
    }

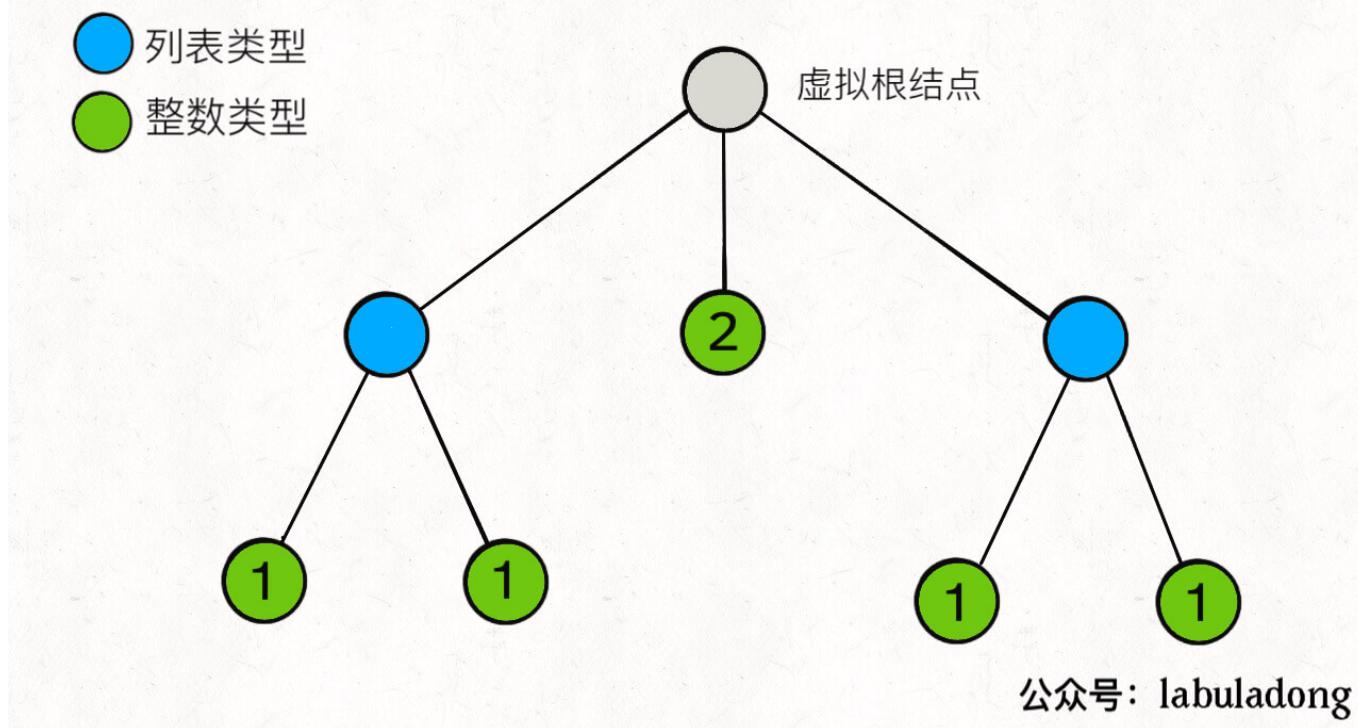
    // 如果其中存的是一个整数，则返回这个整数，否则返回 null
    public Integer getInteger() {
        return this.val;
    }

    // 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;
}
```



```
        }
    }
    return !list.isEmpty();
}
}
```

# 380. 0(1) 时间插入、删除和获取随机元素



- 标签: 数据结构, 设计, 数组, 哈希表, 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  $val$ ，可以把这个元素交换到数组的尾部，然后再  $pop$  掉。

- 详细题解：给我  $O(1)$  时间，我能查找/删除数组中的任意元素

## 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：
  - [710. 黑名单中的随机数（困难）](#)

# 460. LFU 缓存



- 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个。

具体思路略微复杂，请查看详细题解。

- [详细题解：算法题就像搭乐高：手把手带你拆解 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



- 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,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。

- 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;  
    }  
}
```

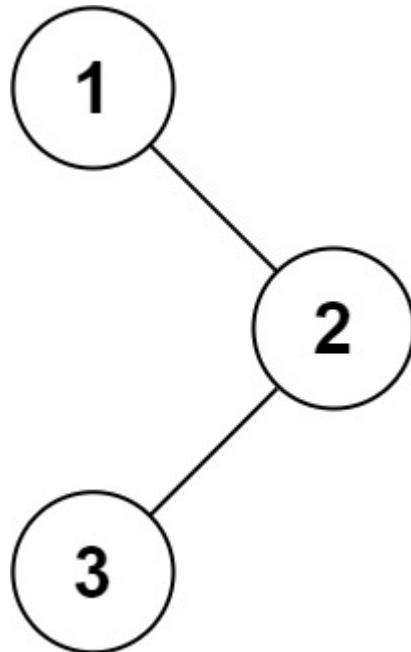
## 94. 二叉树的中序遍历



- 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:

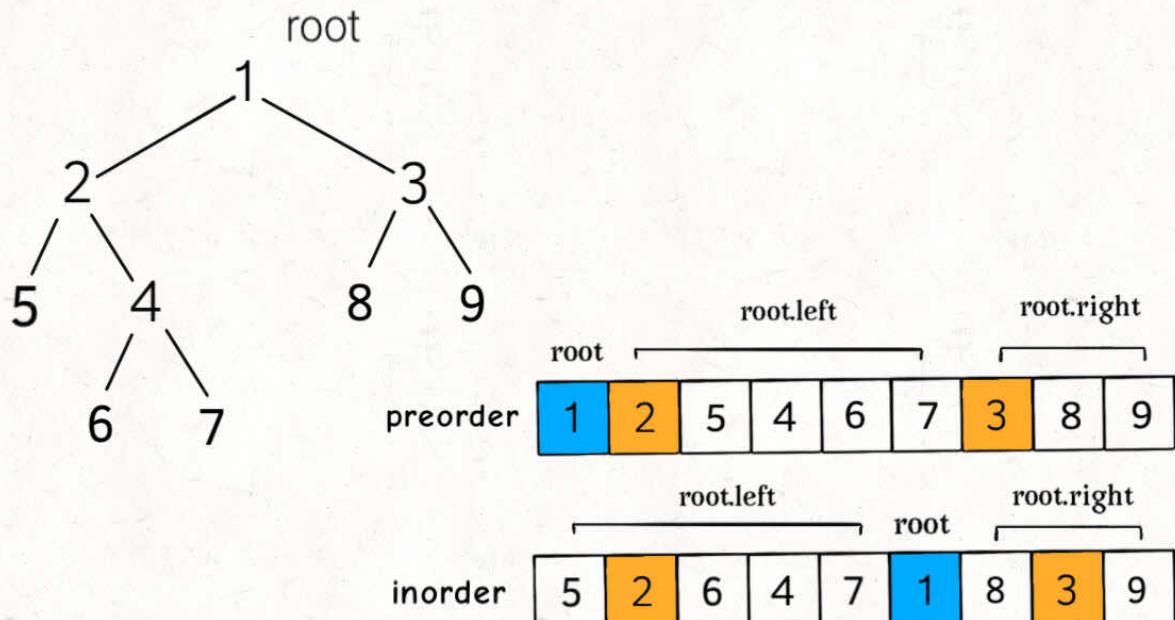


```
输入: root = [1,null,2,3]
输出: [1,3,2]
```

### 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:



公众号: labuladong

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);  
}  
}
```

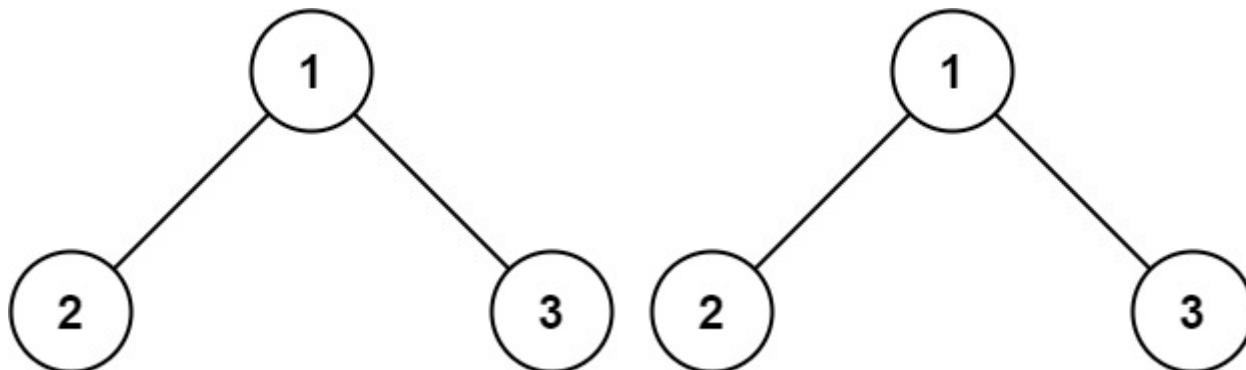
# 100. 相同的树



- 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

这题很简单，就是使用 [学习算法和刷题的框架思维](#) 中说到的二叉树遍历框架遍历一遍二叉树，然后对比它们的节点是否相同就行了。

## 解法代码

```
class Solution {
    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);
    }
}
```



## 102. 二叉树的层序遍历

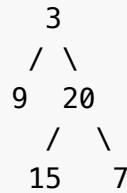


- 标签: [二叉树](#), [BFS 算法](#)

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],`



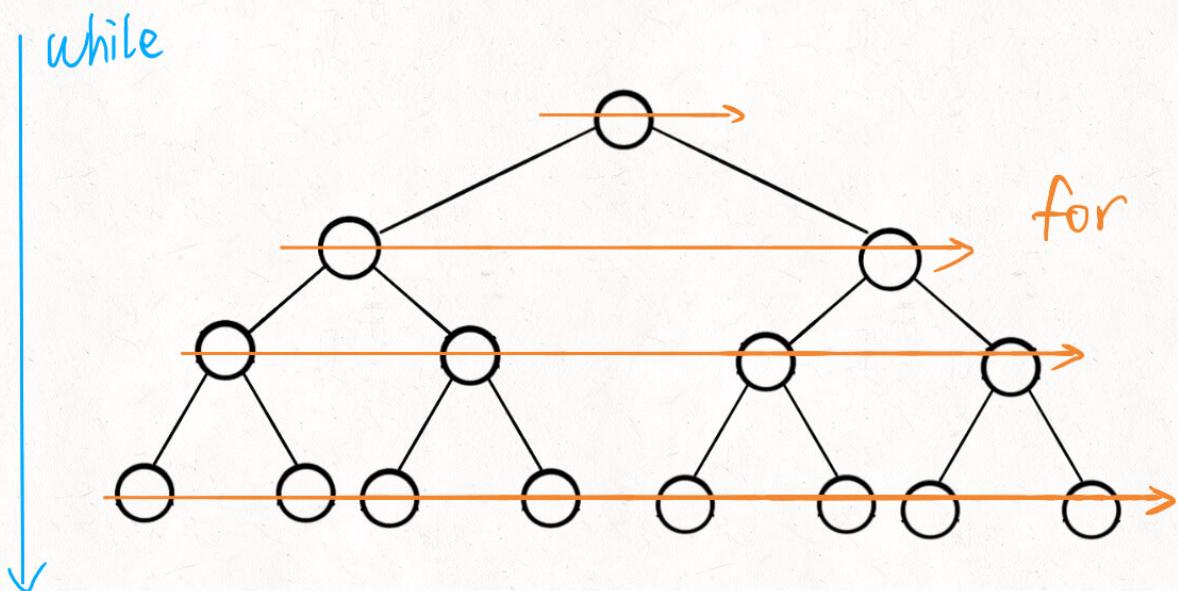
返回其层序遍历结果：

```
[  
  [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：



公众号: labuladong

## 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;
    }
}
```

# 103. 二叉树的锯齿形层序遍历

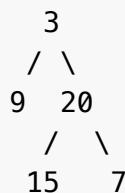


- 标签: [二叉树](#), [BFS 算法](#)

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]`,



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;  
}  
}
```

# 104. 二叉树的最大深度



- 标签: 二叉树, 回溯算法, 动态规划

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

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。

## 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++;
        if (root.left == null && root.right == null) {
            res = Math.max(res, depth);
        }
        traverse(root.left);
        traverse(root.right);
        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);
    }
}
```

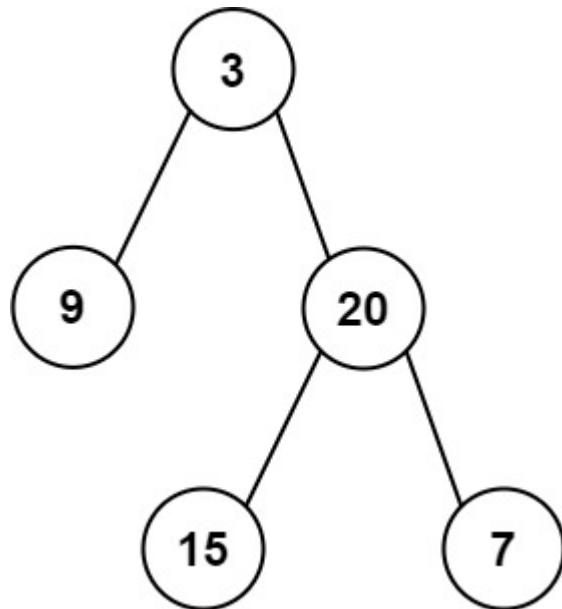
# 105. 从前序与中序遍历序列构造二叉树



- 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:

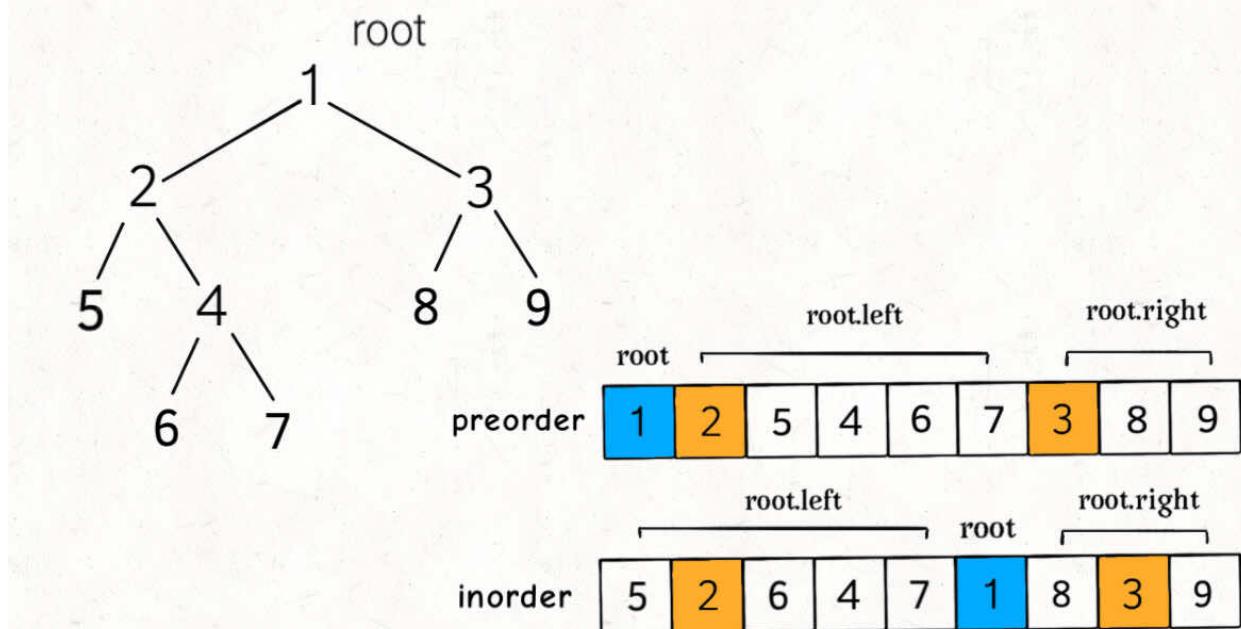


```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

## 基本思路

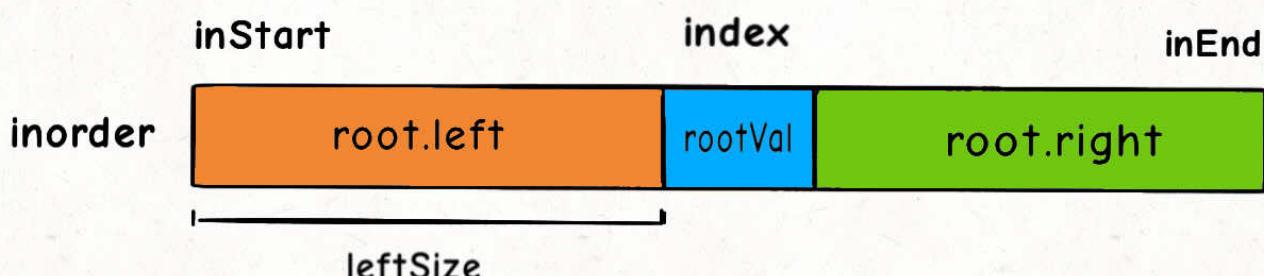
构造二叉树, 第一件事一定是找根节点, 然后想办法构造左右子树。

二叉树的前序和中序遍历结果的特点如下:



公众号: labuladong

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。



公众号: labuladong

结合这个图看代码辅助理解。

- 详细题解：东哥手把手帮你刷通二叉树|第二期

## 解法代码

```
class Solution {
```

```
/* 主函数 */
TreeNode buildTree(int[] preorder, int[] inorder) {
    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 = 0;
    for (int i = inStart; i <= inEnd; i++) {
        if (inorder[i] == rootVal) {
            index = i;
            break;
        }
    }

    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：
  - 654. 最大二叉树（中等）
  - 106. 从中序与后序遍历序列构造二叉树（中等）

# 106. 从中序与后序遍历序列构造二叉树



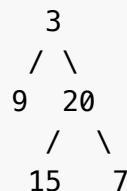
- 标签: 二叉树, 数据结构

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]
```

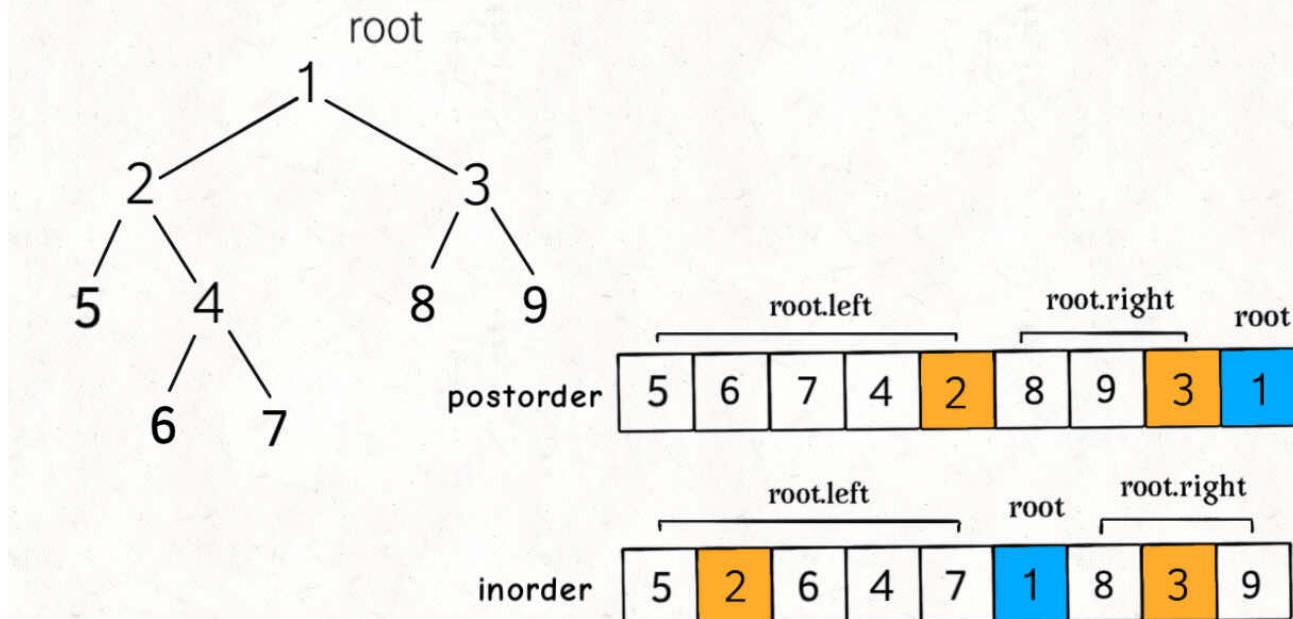
返回如下的二叉树：



## 基本思路

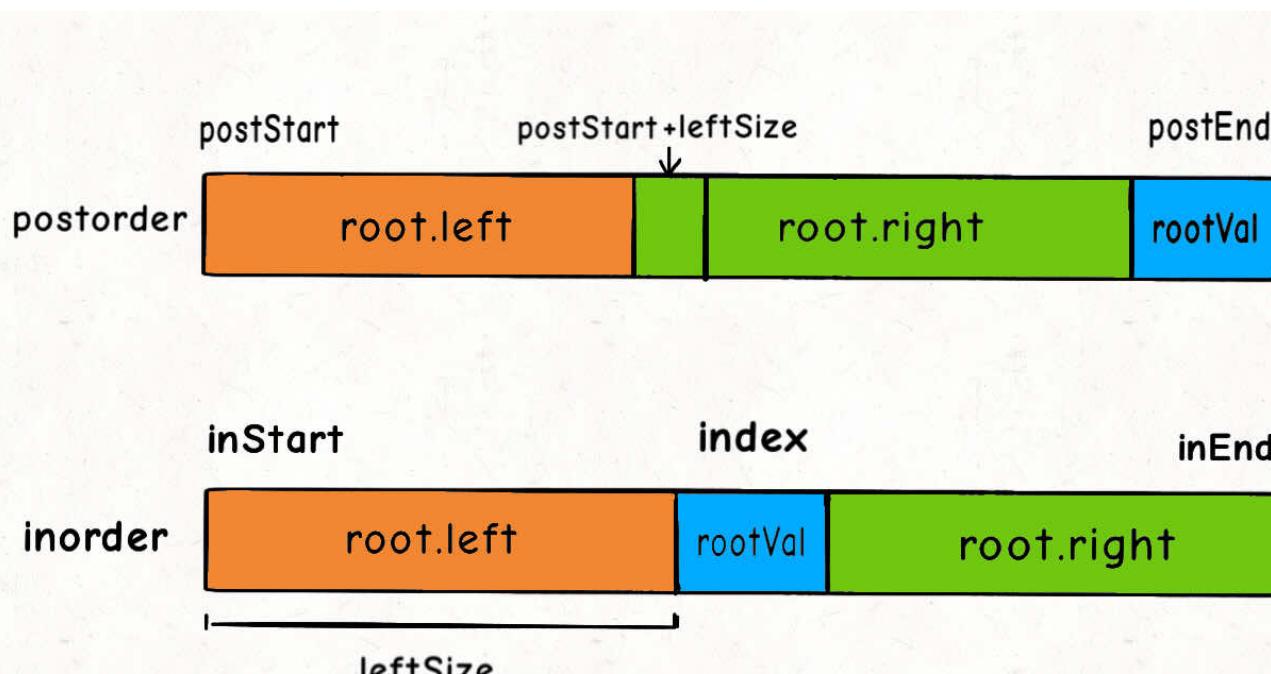
构造二叉树，第一件事一定是找根节点，然后想办法构造左右子树。

二叉树的后序和中序遍历结果的特点如下：



公众号: labuladong

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。



公众号: labuladong

结合这个图看代码辅助理解。

- 详细题解：东哥手把手帮你刷通二叉树|第二期

## 解法代码

```
class Solution {
    public TreeNode buildTree(int[] inorder, int[] postorder) {
```

```
        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 = 0;
        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == rootVal) {
                index = i;
                break;
            }
        }
        // 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;
    }
}
```

- 类似题目：
  - 654. 最大二叉树（中等）
  - 105. 从前序与中序遍历序列构造二叉树（中等）

# 654. 最大二叉树



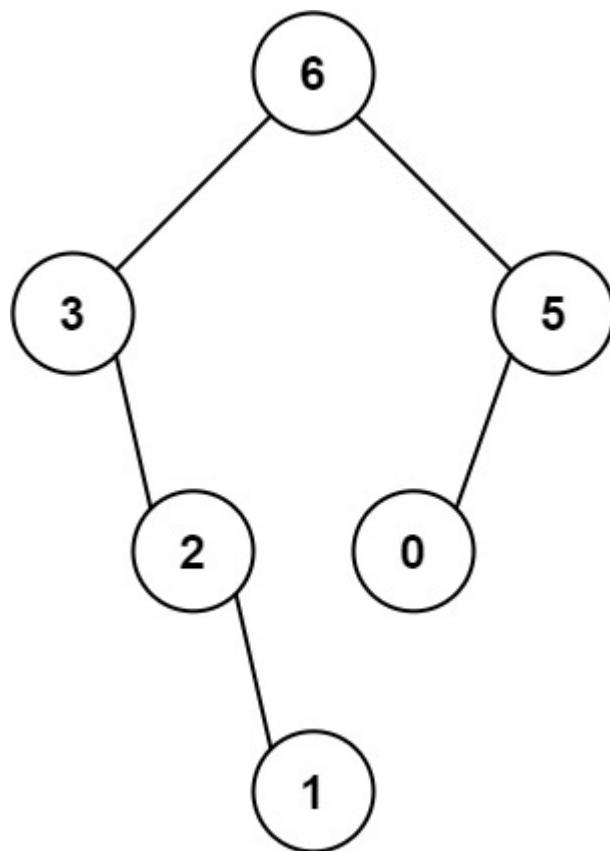
- 标签: 数据结构, 二叉树

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：东哥手把手帮你刷通二叉树|第二期](#)

## 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;  
    }  
}
```

- **类似题目：**

- [105. 从前序与中序遍历序列构造二叉树（中等）](#)
- [106. 从中序与后序遍历序列构造二叉树（中等）](#)

# 107. 二叉树的层序遍历 II

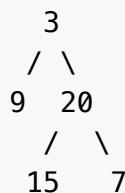


- 标签: 二叉树, BFS 算法

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]，



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;
}
```

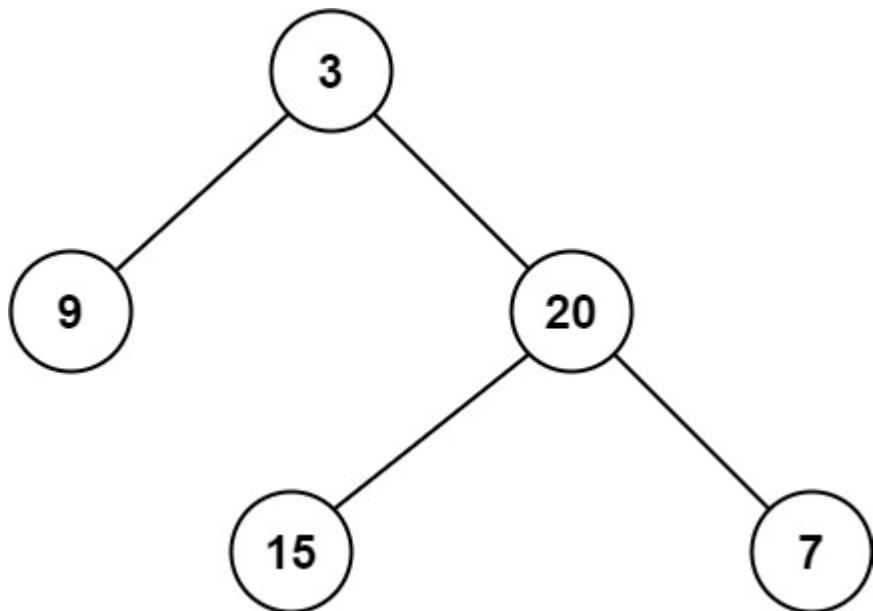
# 111. 二叉树的最小深度



- 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

PS：这道题在《算法小抄》的第 53 页。

基本的二叉树层序遍历方法，值得一提的是，BFS 算法框架就是二叉树层序遍历代码的衍生。

BFS 算法和 DFS（回溯）算法的一大区别就是，BFS 第一次搜索到的结果是最优的，这个得益于 BFS 算法的搜索逻辑，可见详细题解。

- 详细题解: **BFS 算法框架套路详解**

## 解法代码

```
class Solution {
    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;
}
```

- 类似题目：
  - [752. 打开转盘锁（中等）](#)

# 114. 二叉树展开为链表



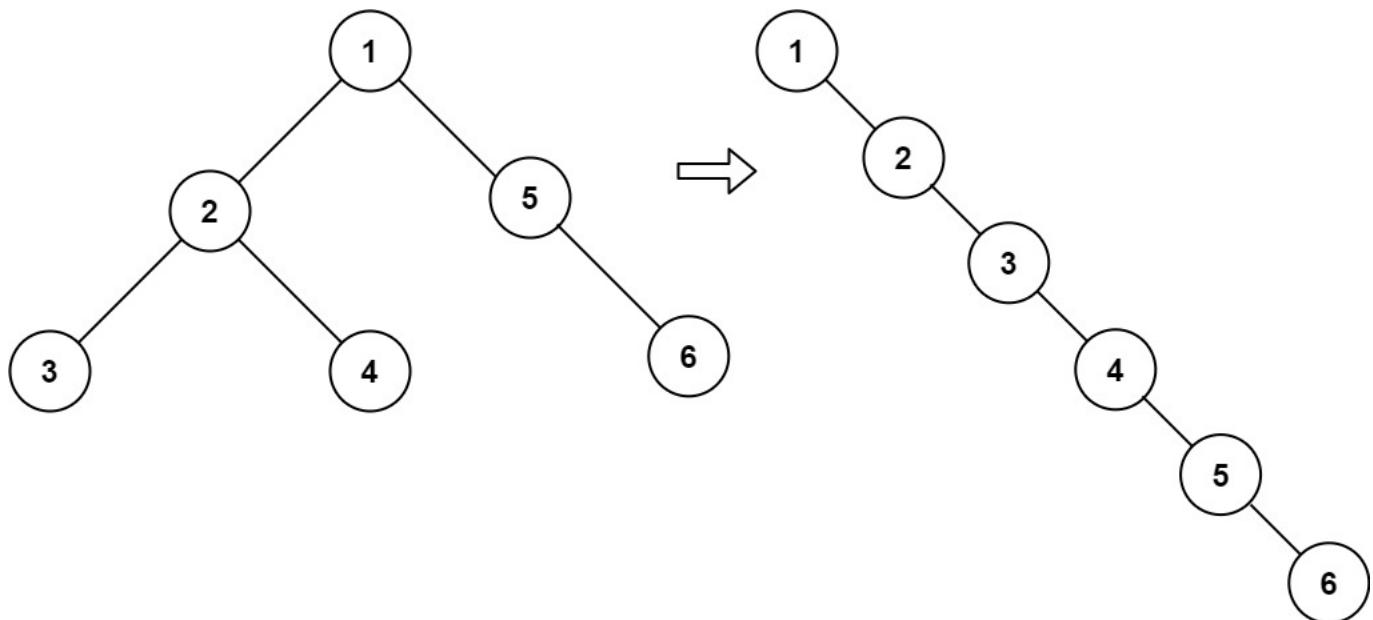
- 标签: 数据结构, 二叉树

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

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。

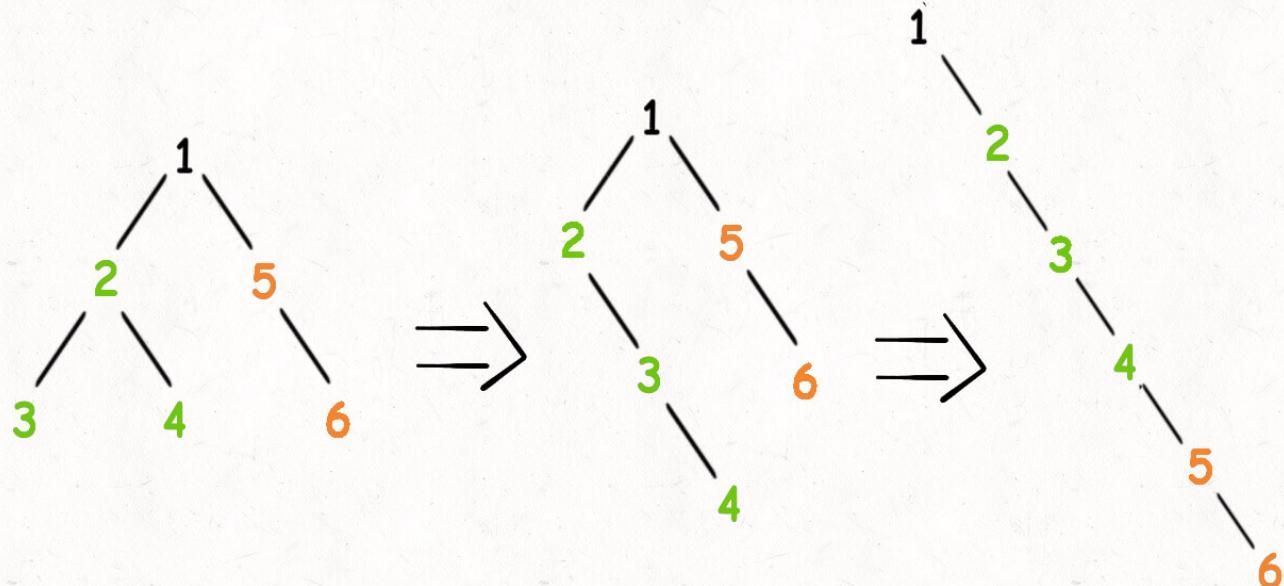
这题属于后者, `flatten` 函数的定义:

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。



公众号： labuladong

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：[东哥手把手带你套框架刷通二叉树|第一期](#)

## 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;
}
```

- 类似题目：

- [226. 翻转二叉树](#) (简单)
- [116. 填充每个节点的下一个右侧节点指针](#) (中等)

# 116. 填充每个节点的下一个右侧节点指针



- 标签: 数据结构, 二叉树

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：

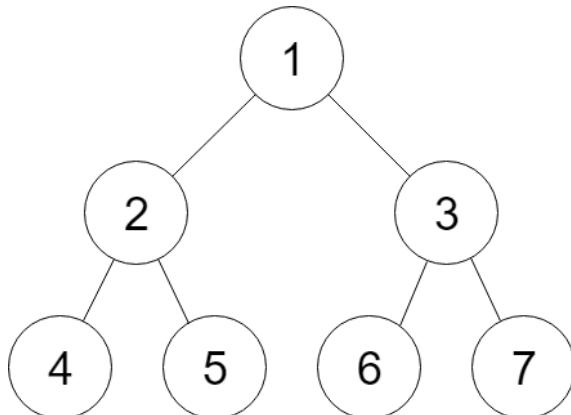


Figure A

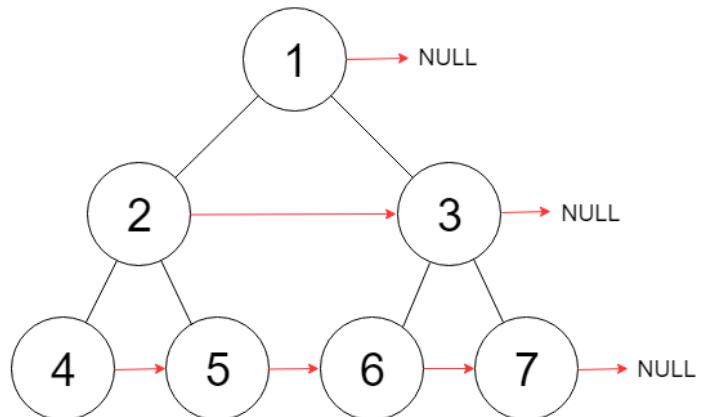


Figure B

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。

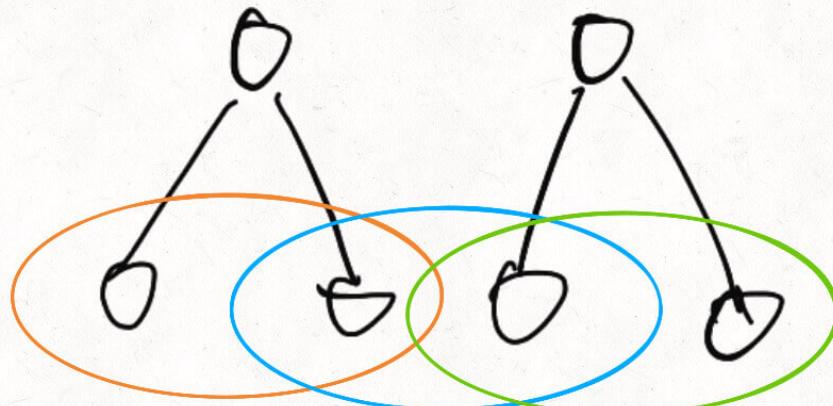
## 基本思路

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。

这题属于前者，可以理解为遍历二叉树的所有相邻节点，然后顺手把这些相邻节点连接起来。

如果是遍历相邻节点，那么就不止是递归左右子树了，因为每一对儿相邻节点可以衍生出三对儿相邻节点：



公众号： labuladong

- 详细题解：[东哥手把手带你套框架刷通二叉树|第一期](#)

## 解法代码

```
class Solution {
    public Node connect(Node root) {
        if (root == null) return null;
        connectTwoNode(root.left, root.right);
        return root;
    }

    // 辅助函数
    void connectTwoNode(Node node1, Node node2) {
        if (node1 == null || node2 == null) {
            return;
        }
        /****前序遍历位置****/
        // 将传入的两个节点连接
        node1.next = node2;

        // 连接相同父节点的两个子节点
        connectTwoNode(node1.left, node1.right);
        connectTwoNode(node2.left, node2.right);
        // 连接跨越父节点的两个子节点
        connectTwoNode(node1.right, node2.left);
    }
}
```

- 类似题目：

- [226. 翻转二叉树（简单）](#)
- [114. 二叉树展开为链表（中等）](#)

## 226. 翻转二叉树

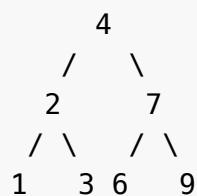


- 标签: 数据结构, 二叉树

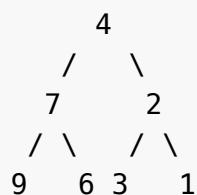
翻转一棵二叉树。

示例：

输入：



输出：



### 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。

本题属于前者，二叉树的递归框架就是这几行代码，可以遍历二叉树的所有节点：

```
// 二叉树遍历框架
void traverse(TreeNode root) {
    // 前序遍历
    traverse(root.left)
    // 中序遍历
    traverse(root.right)
    // 后序遍历
}
```

那么如何翻转二叉树？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嘛。

那怎么实现？那当然是遍历二叉树的所有节点，并且对每个节点实施「交换子节点」的操作喽。

- 详细题解：[东哥手把手带你套框架刷通二叉树|第一期](#)

## 解法代码

```
class Solution {
    // 将整棵树的节点翻转
    public TreeNode invertTree(TreeNode root) {
        // base case
        if (root == null) {
            return null;
        }

        /***前序遍历位置***/
        // 对于 root 节点，需要交换它的左右子节点
        TreeNode tmp = root.left;
        root.left = root.right;
        root.right = tmp;

        // 递归框架，遍历并交换子树所有节点
        invertTree(root.left);
        invertTree(root.right);

        return root;
    }
}
```

- 类似题目：
  - [114. 二叉树展开为链表（中等）](#)
  - [116. 填充每个节点的下一个右侧节点指针（中等）](#)

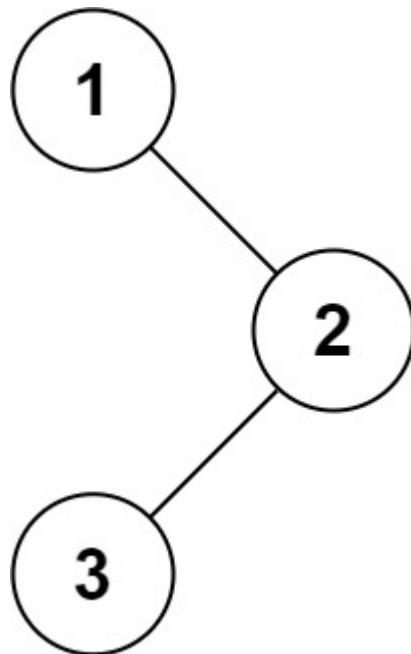
## 144. 二叉树的前序遍历



- 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

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);
}
}
```

# 145. 二叉树的后序遍历



- 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> postorderTraversal(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);
}
}
```

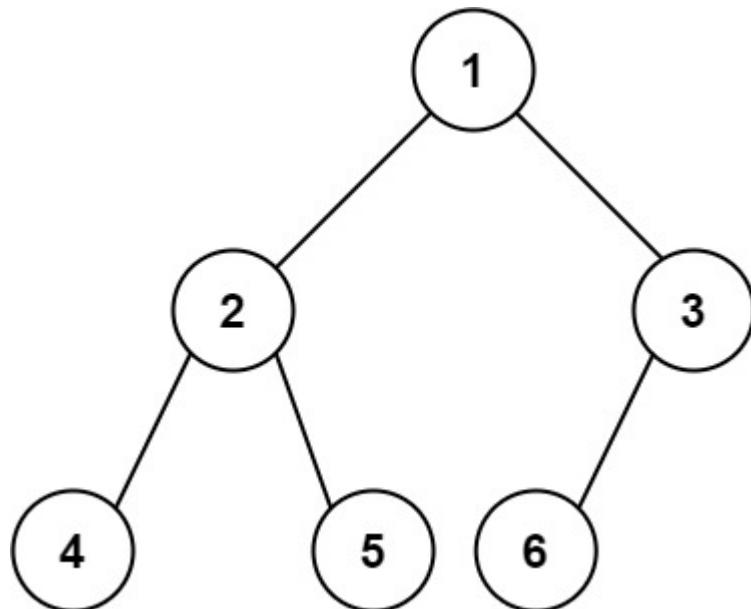
## 222. 完全二叉树的节点个数



- 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:



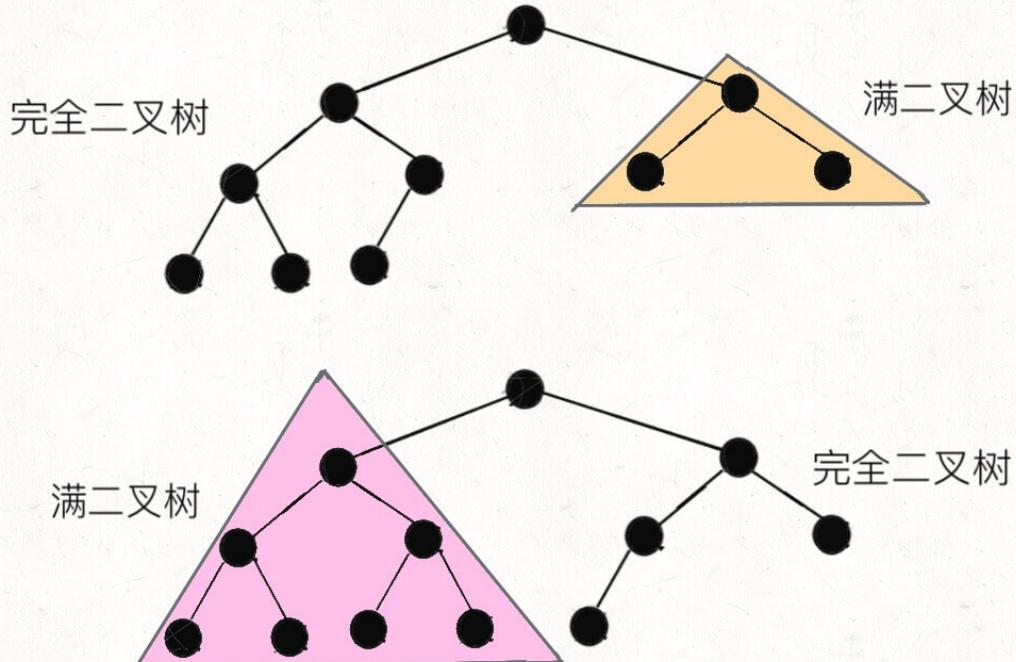
```
输入: root = [1,2,3,4,5,6]
```

```
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:



公众号: labuladong

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：完全二叉树的节点数，你真的会算吗？

## 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);
    }
}
```

# 236. 二叉树的最近公共祖先

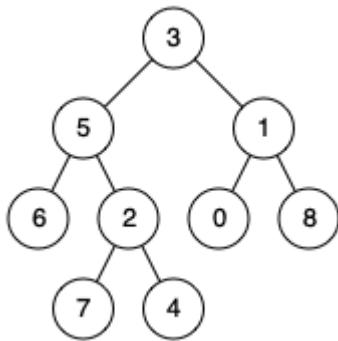


- 标签: 二叉树

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` 呀。

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;
    }
}
```

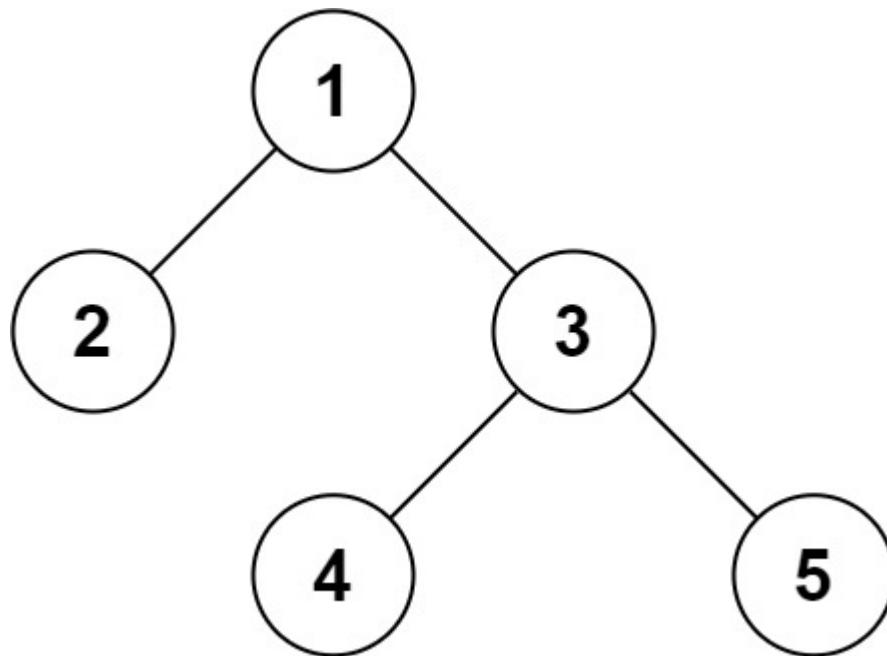
## 297. 二叉树的序列化与反序列化



- 标签: [数据结构](#), [二叉树](#), [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:



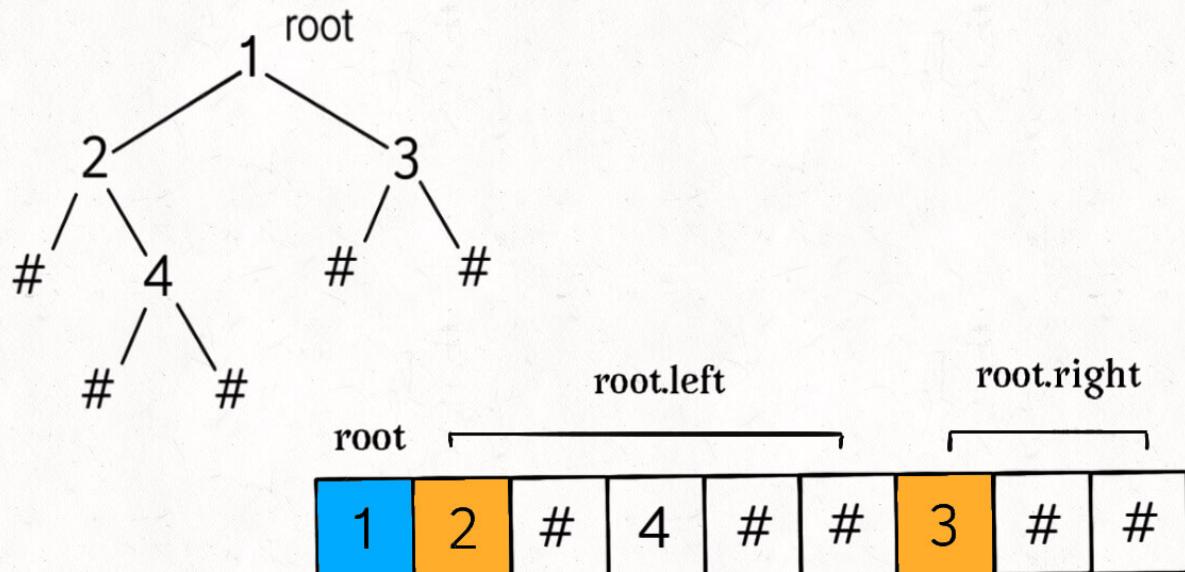
```
输入: root = [1,2,3,null,null,4,5]
输出: [1,2,3,null,null,4,5]
```

### 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：



公众号: labuladong

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：[二叉树的题，就那几个框架，枯燥至极](#)

## 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);
        /*****后序遍历位置*****/
        serialize(root.left, sb);
        serialize(root.right, sb);
    }
}

```

```
    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;
}
}
```

# 341. 扁平化嵌套列表迭代器



- 标签: 数据结构, 二叉树, 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;
    }

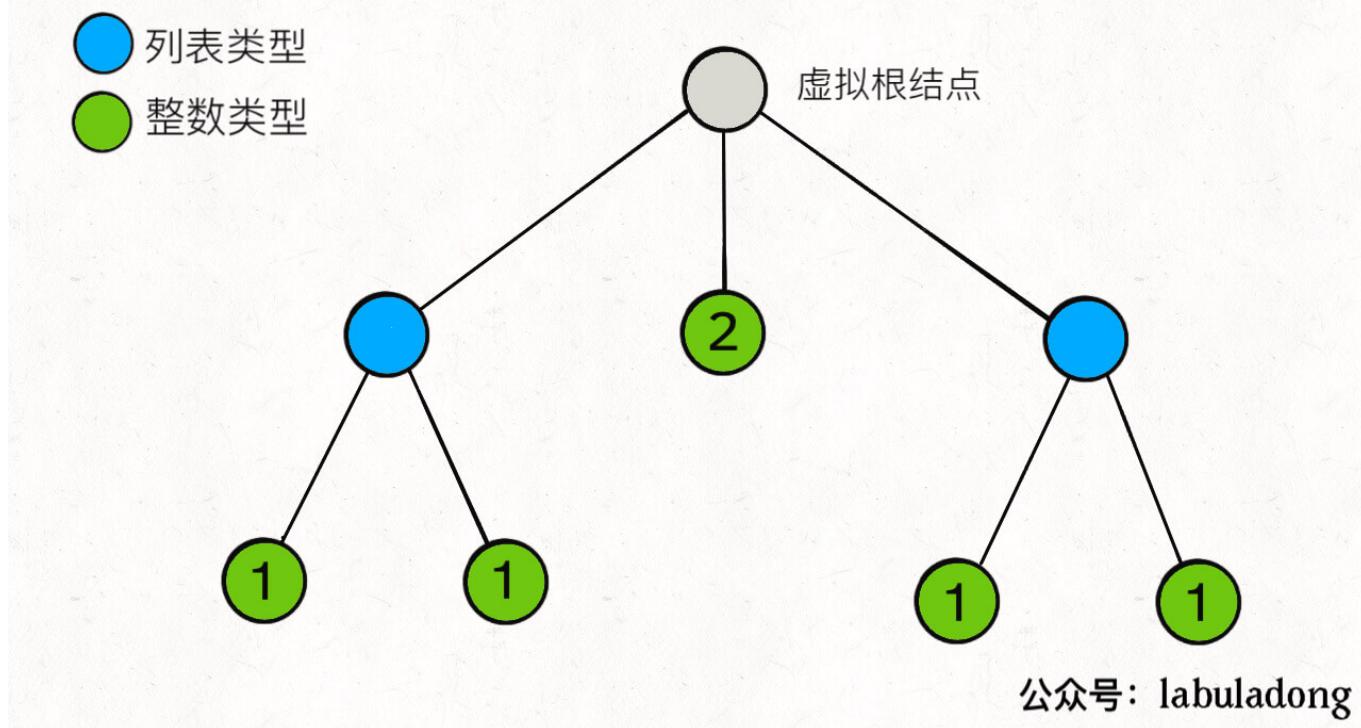
    // 如果其中存的是一个整数，则返回这个整数，否则返回 null
    public Integer getInteger() {
        return this.val;
    }

    // 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;
}
```



```
        }
    }
    return !list.isEmpty();
}
}
```

# 501. 二叉搜索树中的众数



- 标签: 二叉树, 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);
        if (prev != null && prev.val == root.val) {
            curCount++;
        } else {
            curCount = 1;
        }
        if (curCount > maxCount) {
            maxCount = curCount;
            mode.clear();
            mode.add(root.val);
        } else if (curCount == maxCount) {
            mode.add(root.val);
        }
        prev = root;
        traverse(root.right);
    }
}
```

```
        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);
}
}
```

# 543. 二叉树的直径



- 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



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

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。

## 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);
        maxDiameter = Math.max(maxDiameter, leftMax + rightMax);
        return Math.max(leftMax, rightMax) + 1;
    }
}
```

```
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);
    }
}
```

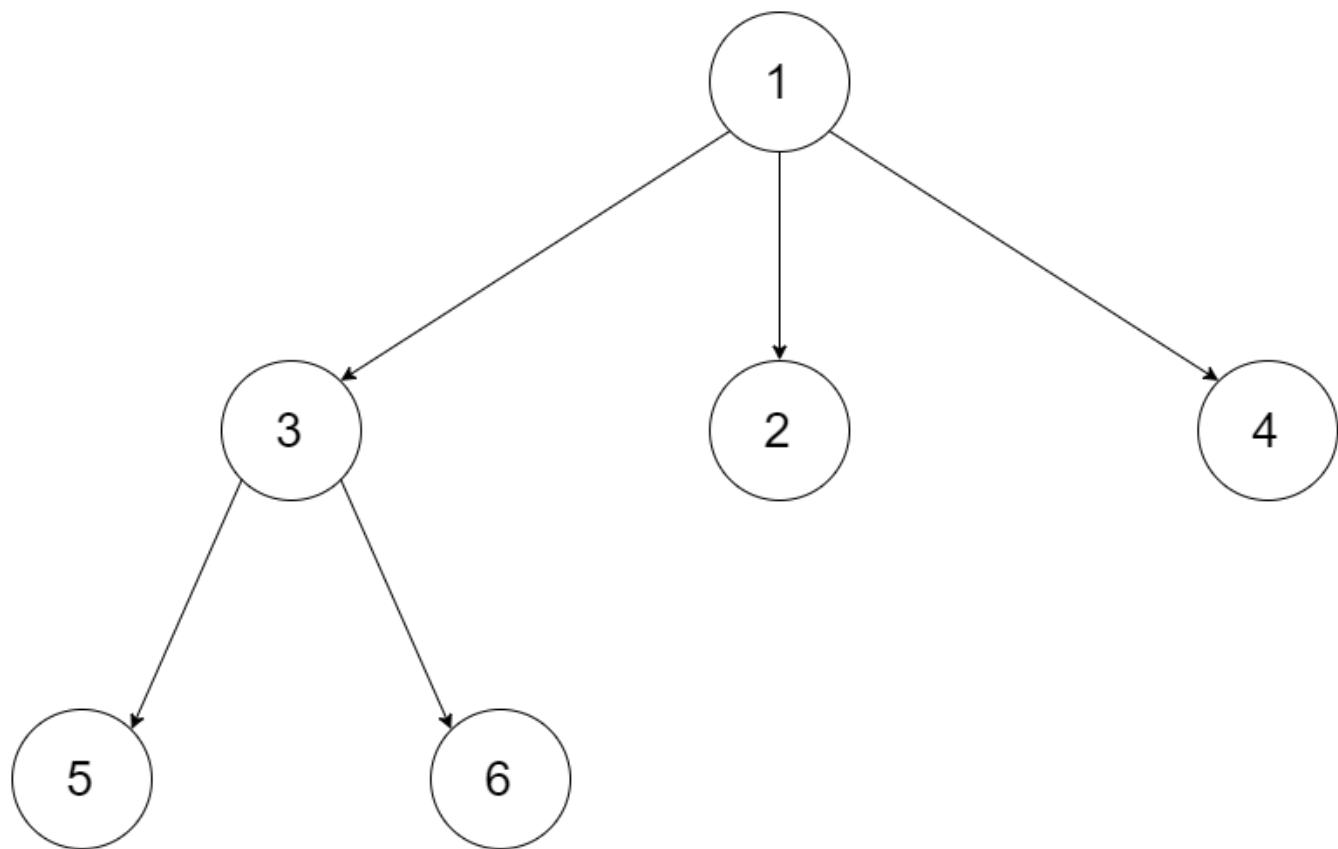
# 559. N 叉树的最大深度



- 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

```
// 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--;  
    }  
}
```

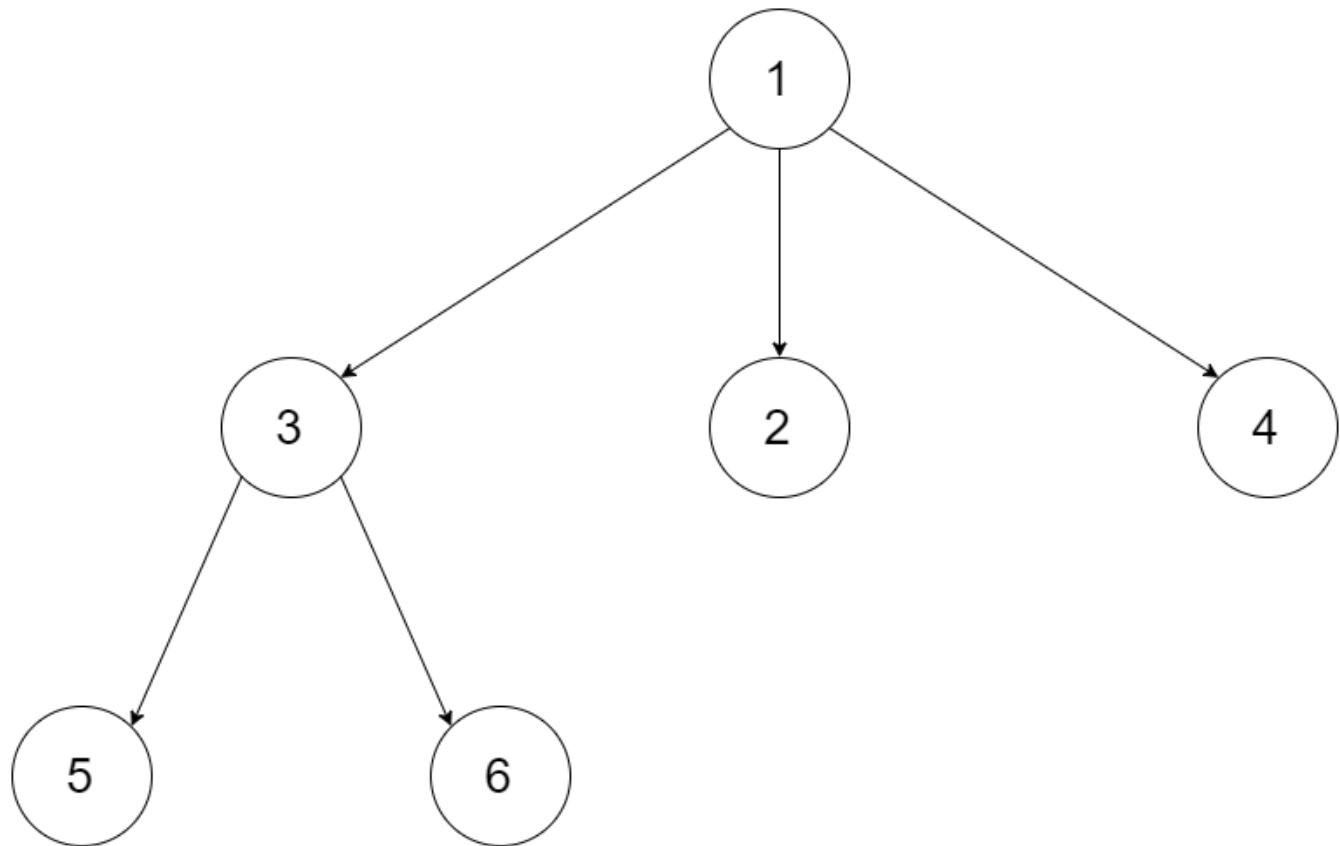
# 589. N 叉树的前序遍历



- 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  
}  
}
```

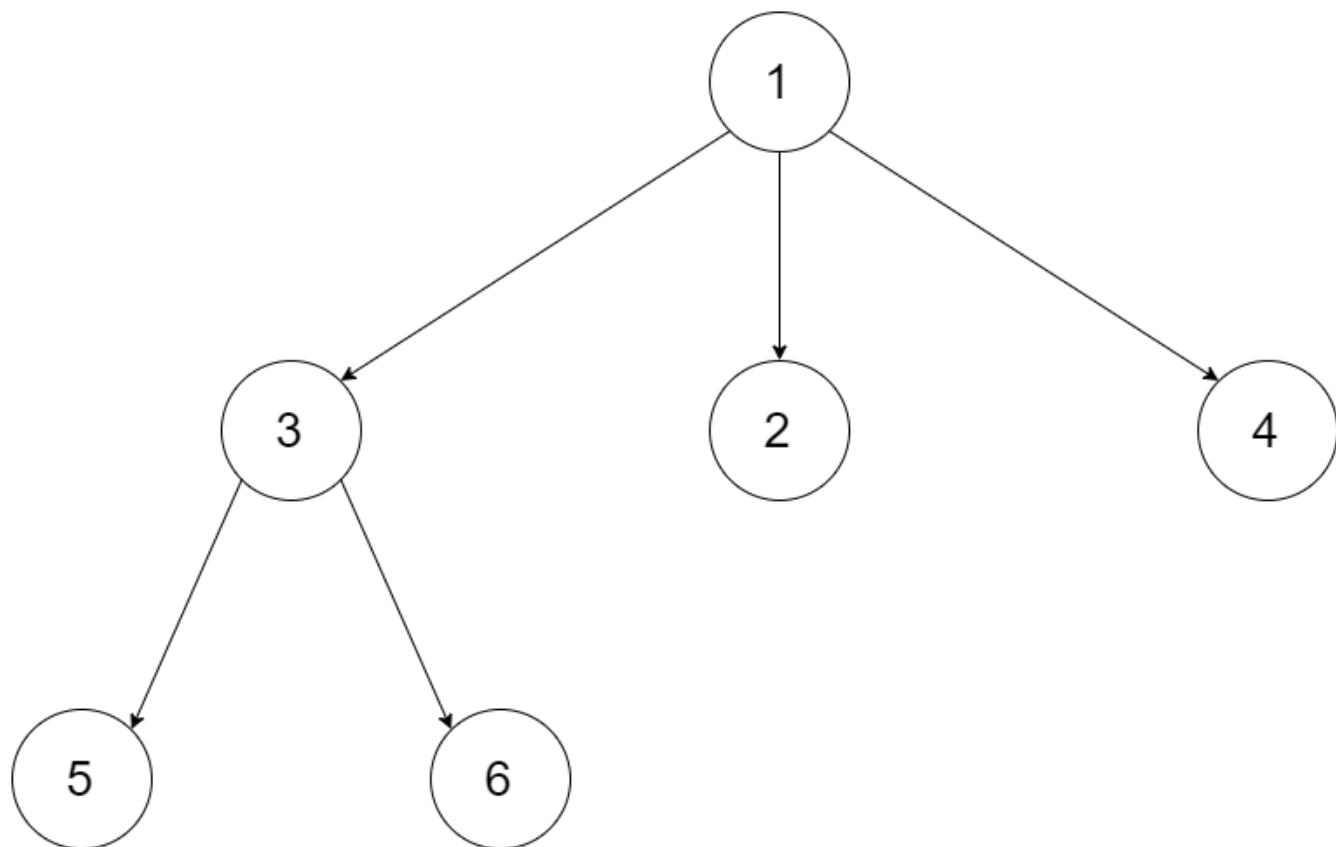
# 590. N 叉树的后序遍历



- 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);
}
```

# 652. 寻找重复的子树

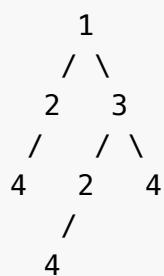


- 标签: [二叉树](#)

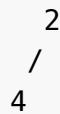
给定一棵二叉树，返回所有重复的子树。对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。

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:



下面是两个重复的子树：



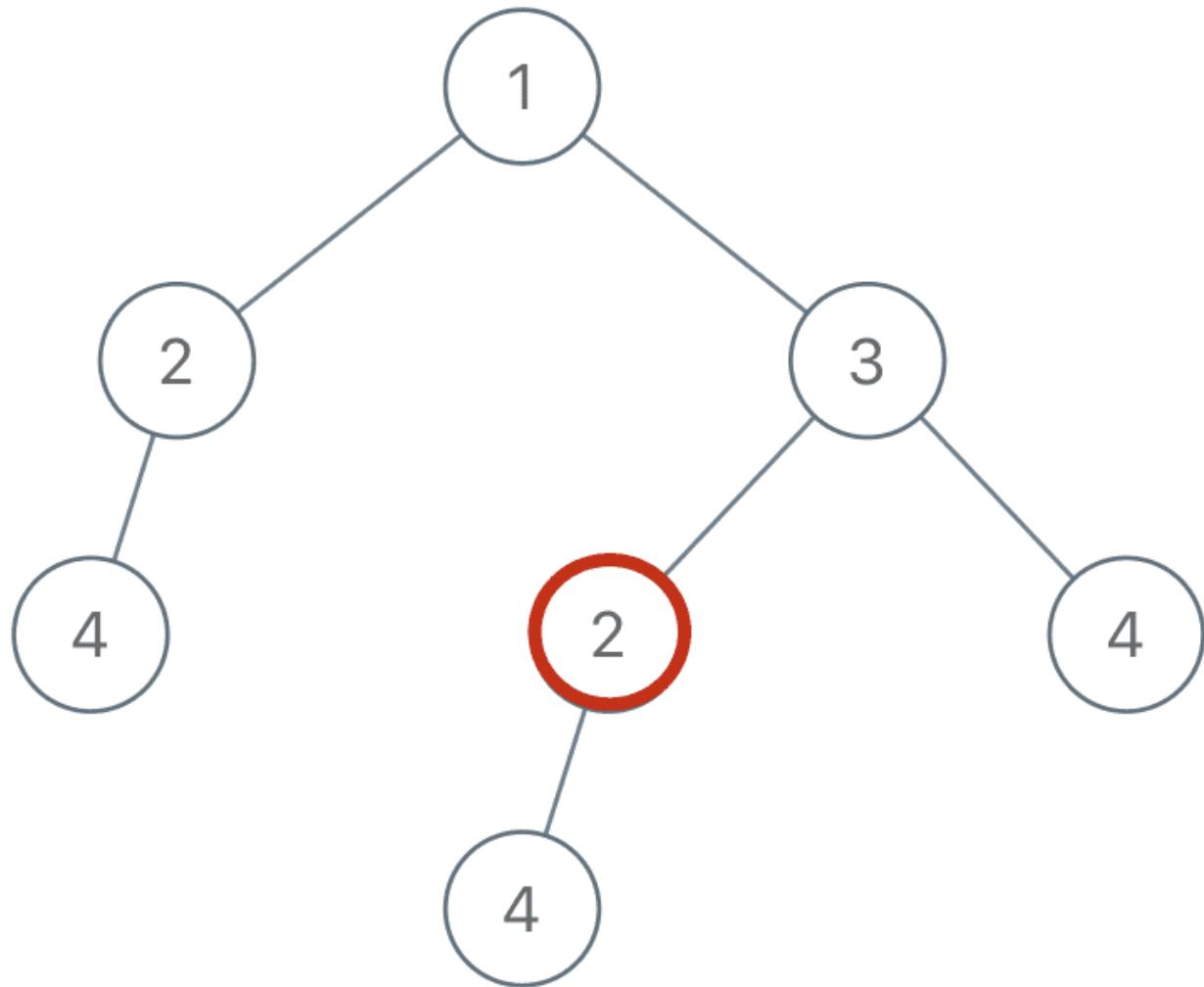
和

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。

- 详细题解：[东哥手把手带你刷二叉树|第三期](#)

## 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<>();

    /* 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;
    }
}
```

# 965. 单值二叉树

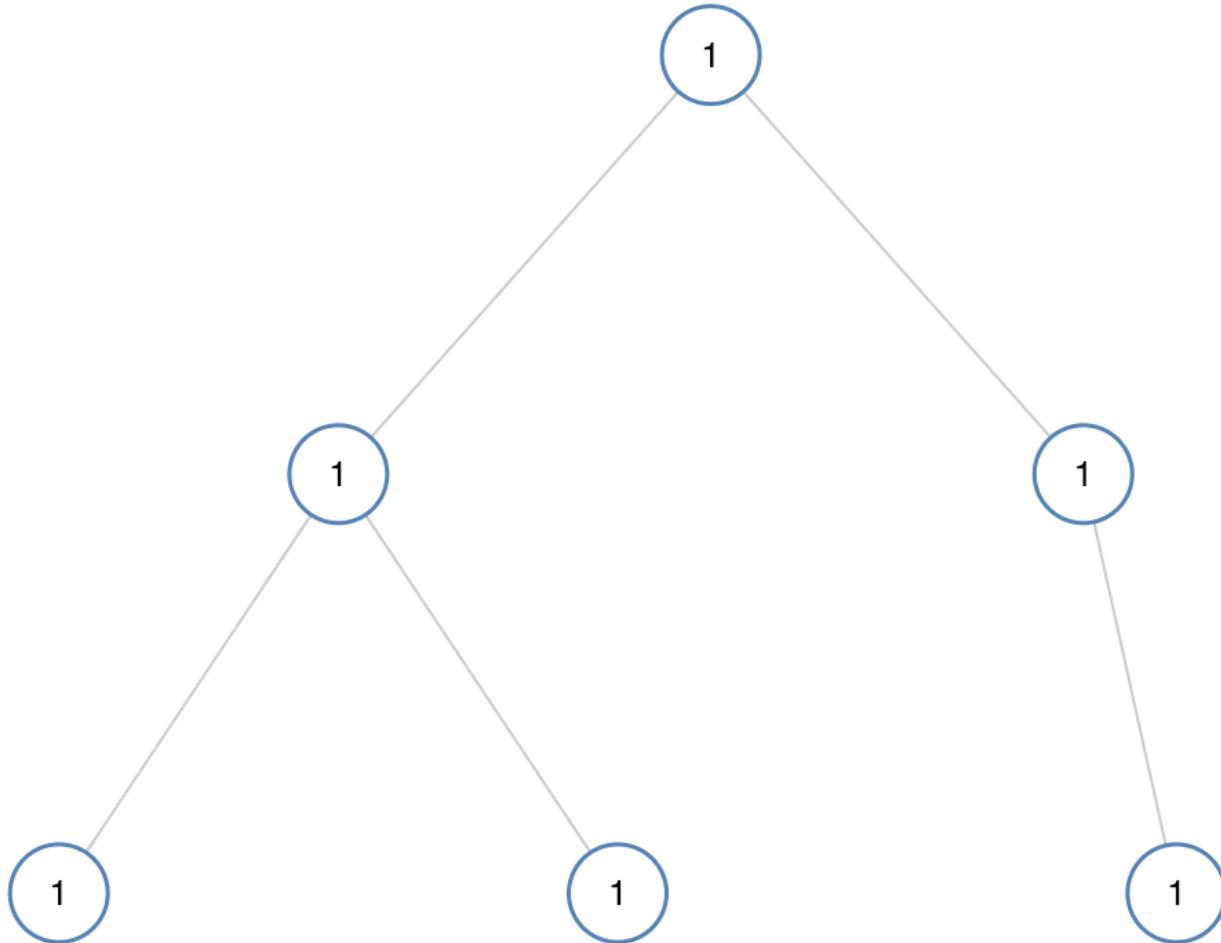


- 标签: [二叉树](#)

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);
    }
}
```

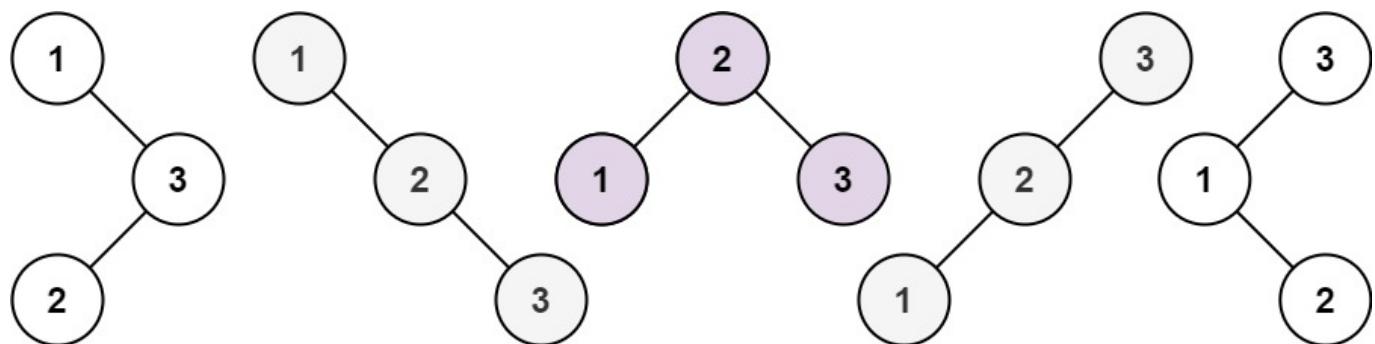
# 95. 不同的二叉搜索树 II



- 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:  $[[1, \text{null}, 2, \text{null}, 3], [1, \text{null}, 3, 2], [2, 1, 3], [3, 1, \text{null}, \text{null}, 2], [3, 2, \text{null}, 1]]$

## 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：[手把手带你刷通二叉搜索树（第三期）](#)

## 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;
}
```

- 类似题目：
  - [96. 不同的二叉搜索树（简单）](#)

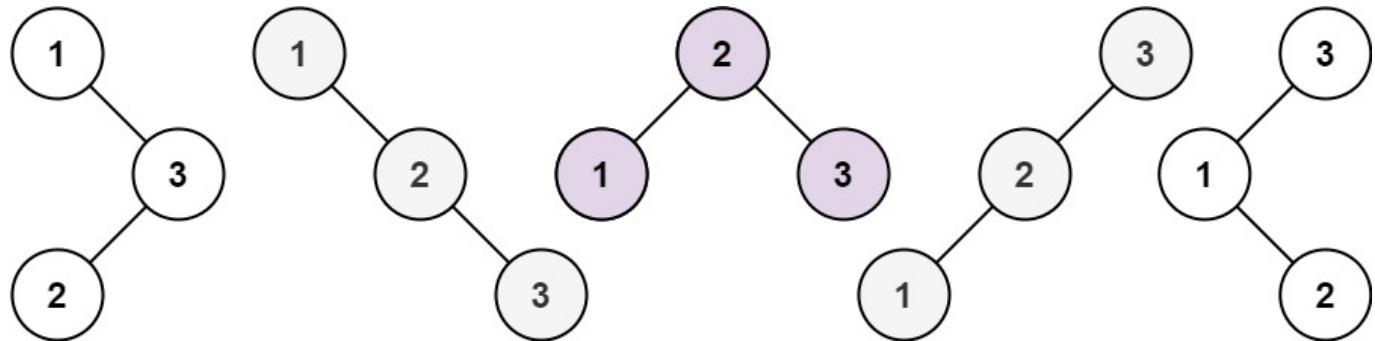
# 96. 不同的二叉搜索树



- 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:



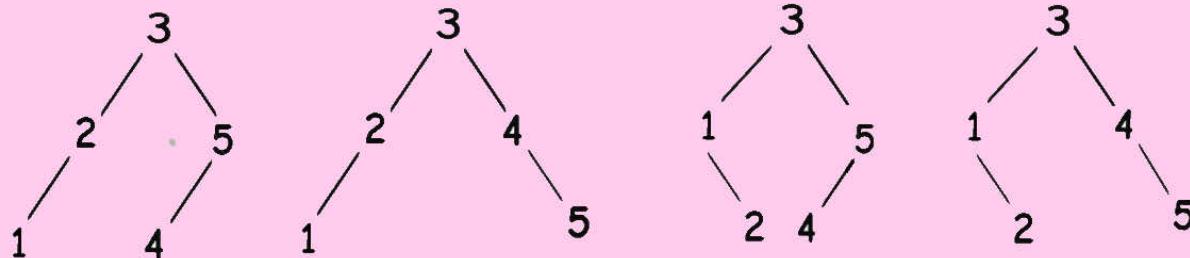
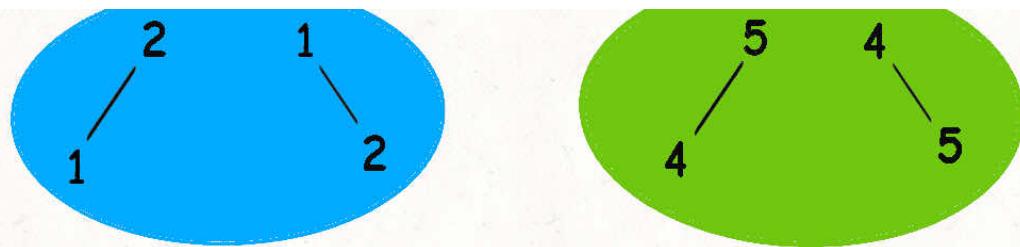
输入:  $n = 3$

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:



$2 \times 2 = 4$  种

公众号: labuladong

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：[手把手带你刷通二叉搜索树（第三期）](#)

## 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;
}

}
```

- 类似题目：
  - [95. 不同的二叉搜索树 II \(中等\)](#)

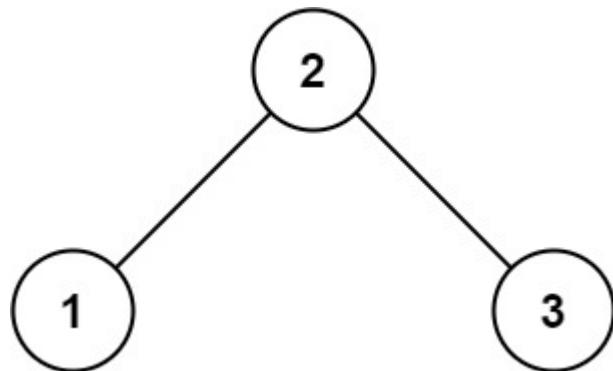
## 98. 验证二叉搜索树



- 标签: 数据结构, 二叉搜索树

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: 手把手刷二叉搜索树 (第二期)

### 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);
}
}
```

- 类似题目:

- 450. 删除二叉搜索树中的节点 (中等)
- 701. 二叉搜索树中的插入操作 (中等)
- 700. 二叉搜索树中的搜索 (简单)

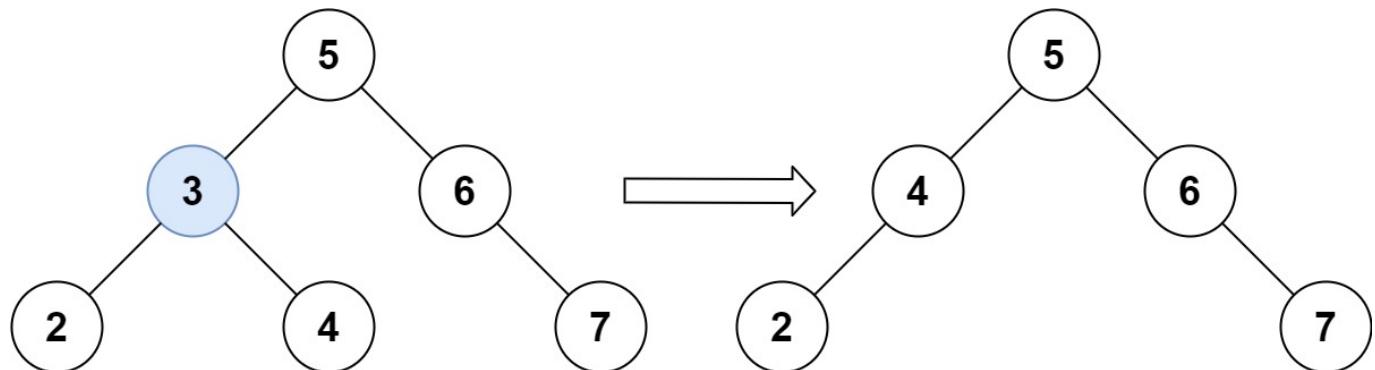
# 450. 删除二叉搜索树中的节点



- 标签: 数据结构, 二叉搜索树

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]`。

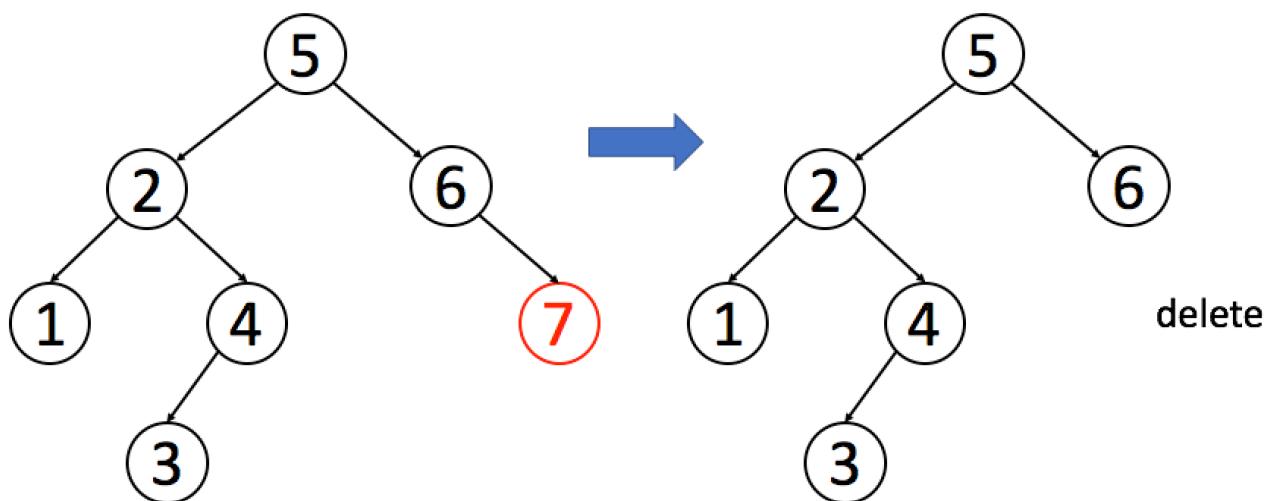
## 基本思路

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:

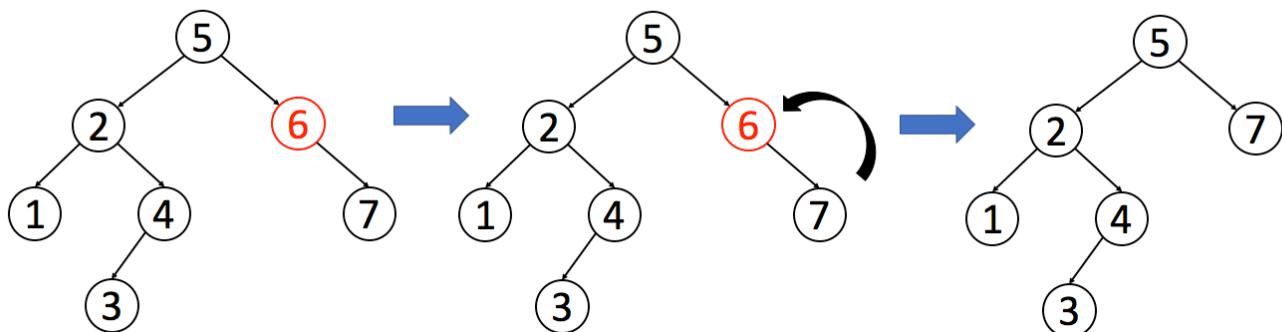
情况 1: A 恰好是末端节点, 两个子节点都为空, 那么它可以当场去世了:

## Case 1: No Child



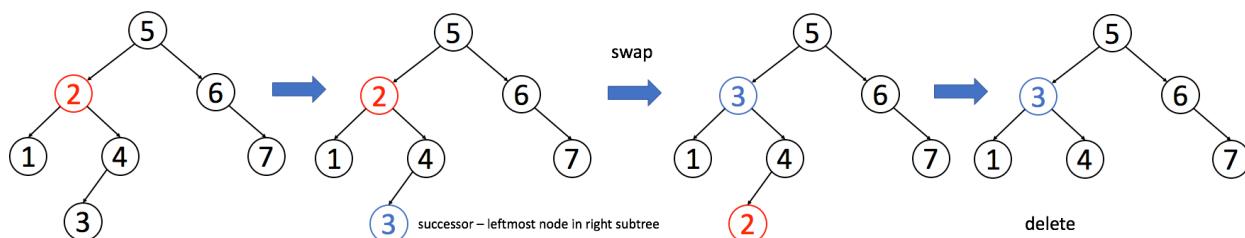
情况 2: A 只有一个非空子节点, 那么它要让这个孩子接替自己的位置:

## Case 2: One Child



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:

## Case 3: Two Children



- 详细题解: 手把手刷二叉搜索树 (第二期)

## 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;
    }
}
```

- 类似题目：

- [701. 二叉搜索树中的插入操作](#) (中等)
- [700. 二叉搜索树中的搜索](#) (简单)
- [98. 验证二叉搜索树](#) (中等)

# 700. 二叉搜索树中的搜索



- 标签: 数据结构, 二叉搜索树

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：



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：[手把手刷二叉搜索树（第二期）](#)

## 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;
    }
}
```

```
    }  
}
```

- 类似题目：

- [450. 删除二叉搜索树中的节点 \(中等\)](#)
- [701. 二叉搜索树中的插入操作 \(中等\)](#)
- [98. 验证二叉搜索树 \(中等\)](#)

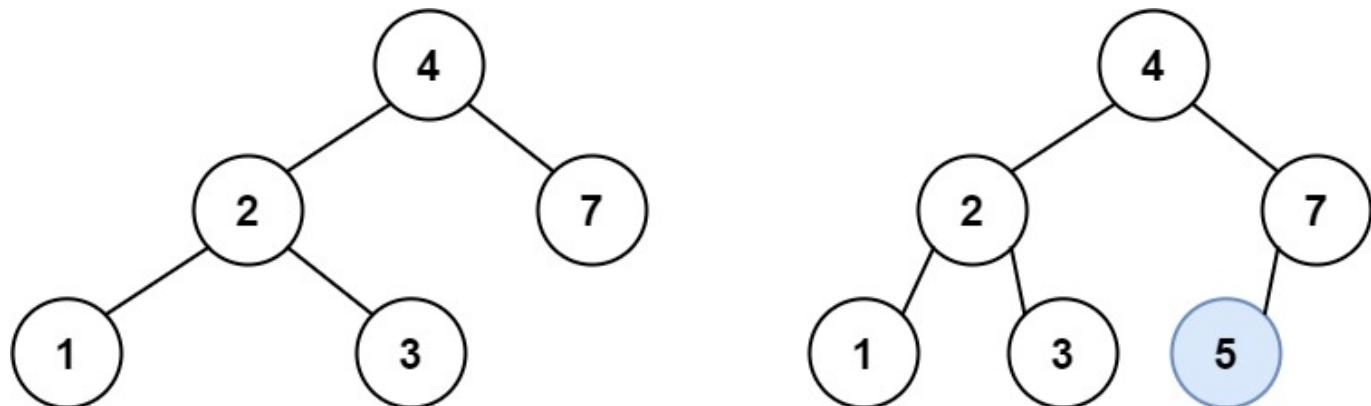
# 701. 二叉搜索树中的插入操作



- 标签: [数据结构](#), [二叉搜索树](#)

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。

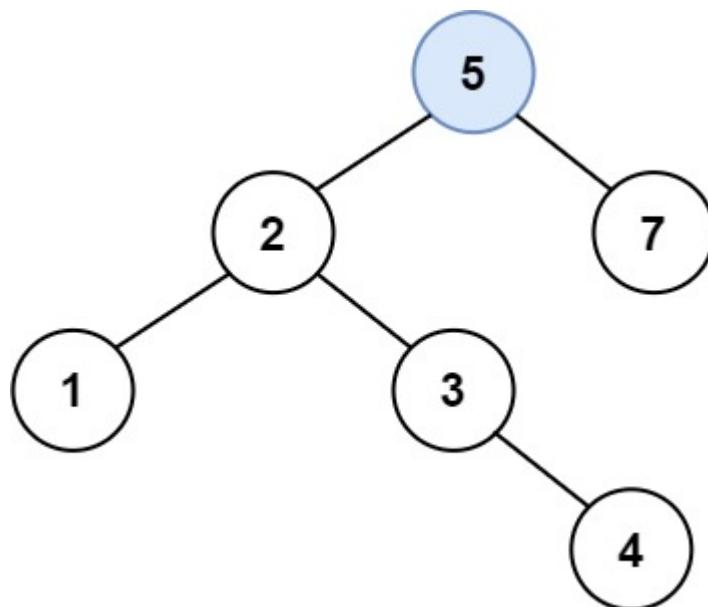
示例 1:



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：[手把手刷二叉搜索树（第二期）](#)

## 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;  
    }  
}
```

- 类似题目：
  - [450. 删除二叉搜索树中的节点（中等）](#)
  - [700. 二叉搜索树中的搜索（简单）](#)
  - [98. 验证二叉搜索树（中等）](#)

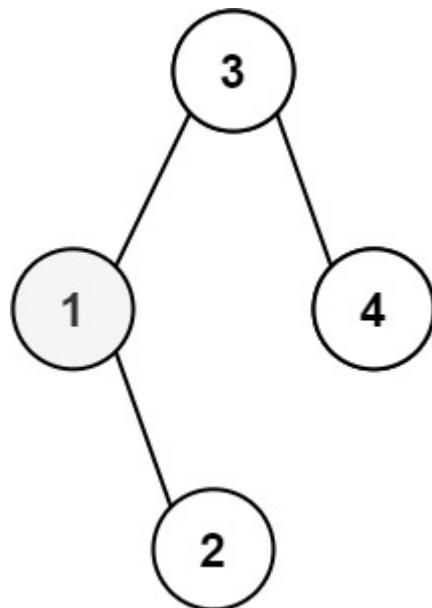
# 230. 二叉搜索树中第 K 小的元素



- 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: [手把手刷二叉搜索树 \(第一期\)](#)

## 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);
}
}
```

- 类似题目：
  - [538. 二叉搜索树转化累加树](#) (中等)
  - [1038.BST 转累加树](#) (中等)

# 538. 把二叉搜索树转换为累加树

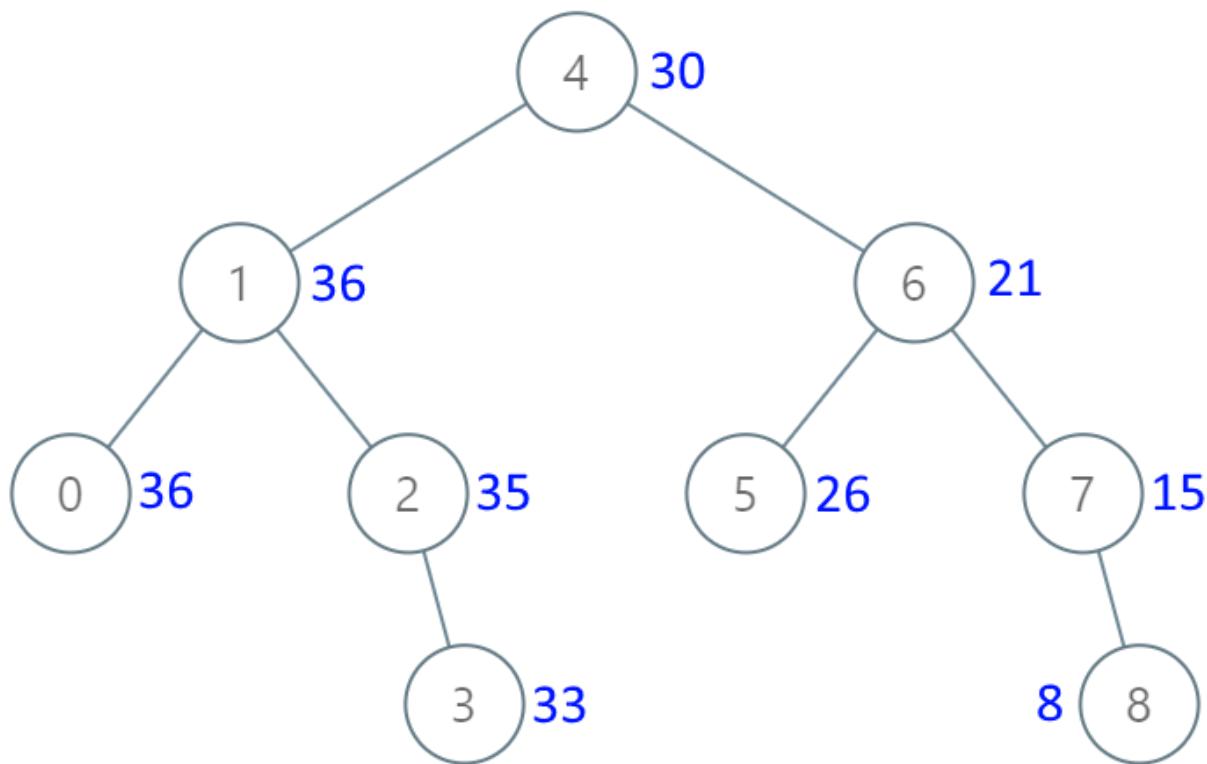


- 标签: 数据结构, 二叉搜索树

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：[手把手刷二叉搜索树（第一期）](#)

## 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);
    }
}
```

- 类似题目：
  - [230.BST 第 K 小的元素](#) (中等)
  - [1038.BST 转累加树](#) (中等)

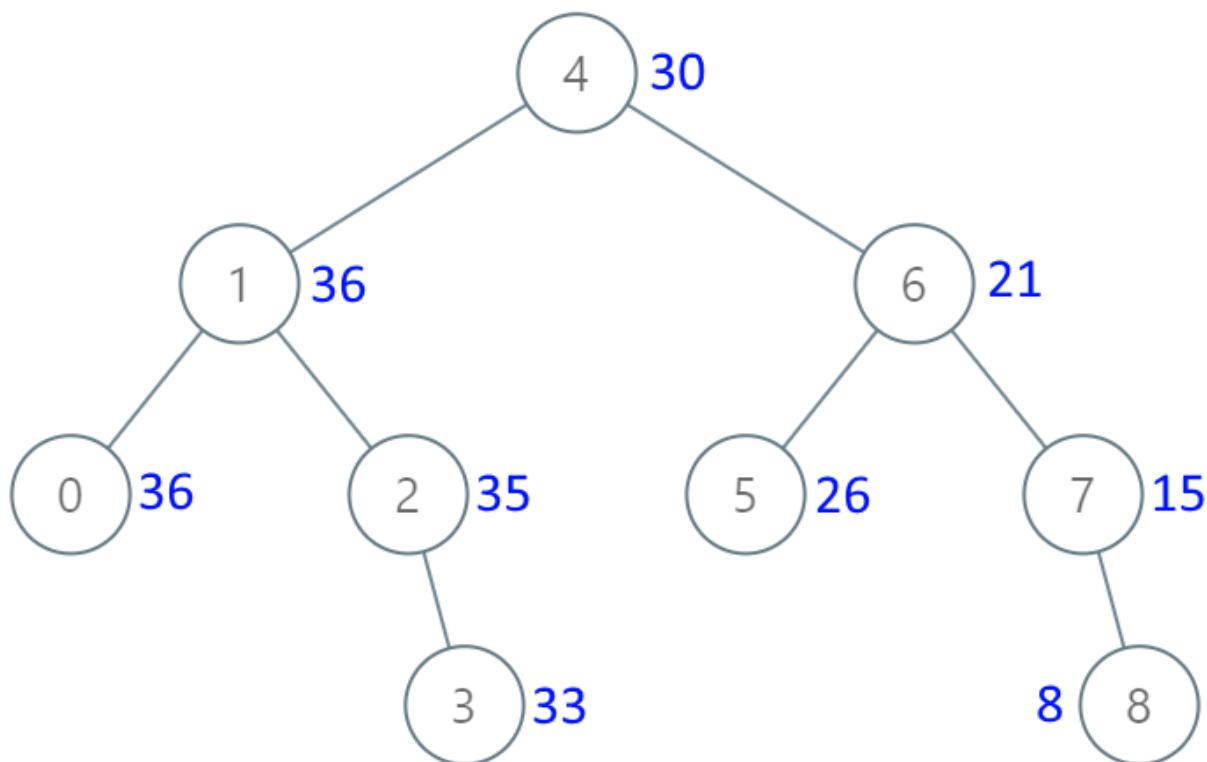
# 1038. 把二叉搜索树转换为累加树



- 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: [手把手带你刷二叉搜索树（第一期）](#)

## 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);
    }
}
```

- 类似题目：
  - [230.BST第K小的元素](#) (中等)
  - [538.二叉搜索树转化累加树](#) (中等)

# 501. 二叉搜索树中的众数



- 标签: 二叉树, 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);
        if (prev != null && prev.val == root.val) {
            curCount++;
        } else {
            curCount = 1;
        }
        if (curCount > maxCount) {
            maxCount = curCount;
            mode.clear();
            mode.add(root.val);
        } else if (curCount == maxCount) {
            mode.add(root.val);
        }
        prev = root;
        traverse(root.right);
    }
}
```

```
        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);
}
}
```

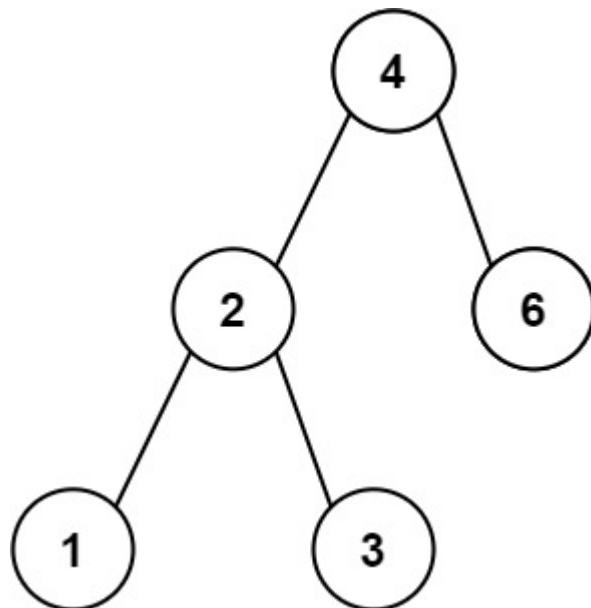
# 530. 二叉搜索树的最小绝对差



- 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。

中序遍历会有序遍历 BST 的节点, 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);
}
```

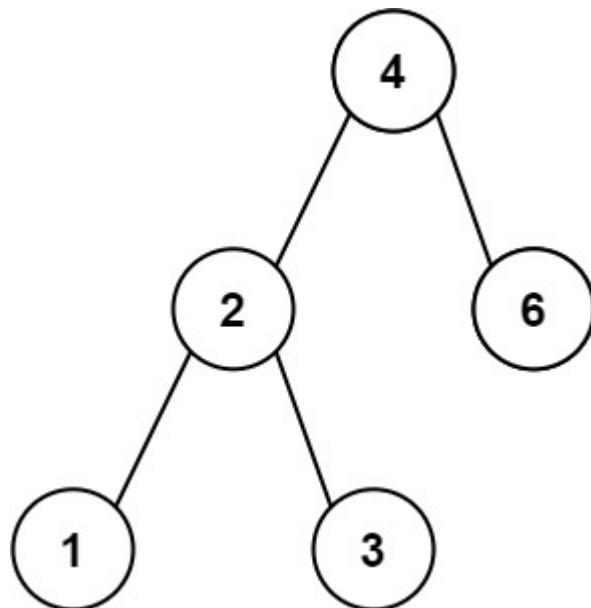
# 783. 二叉搜索树节点最小距离



- 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);
}
}
```

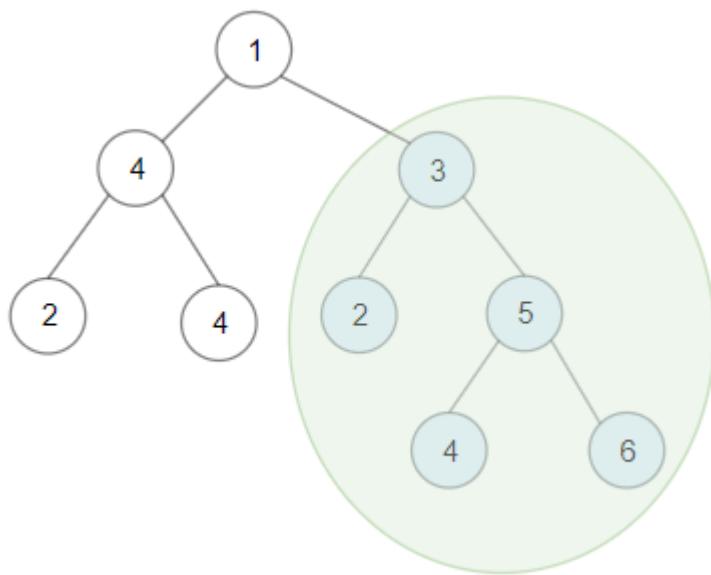
# 1373. 二叉搜索子树的最大键值和



- 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，如果这两儿子有一个不是 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 `traverse` 函数，`traverse(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：美团面试官：你对二叉树后序遍历一无所知](#)

## 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[] traverse(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 = traverse(root.left);
        int[] right = traverse(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] = root.val;
            // 计算以 root 为根的这棵 BST 的最大值
            res[2] = root.val;
            // 计算以 root 为根的这棵 BST 的节点值之和
            res[3] = left[3] + right[3] + root.val;
        }
        else {
            res[0] = 0;
            res[1] = Integer.MAX_VALUE;
            res[2] = Integer.MIN_VALUE;
            res[3] = 0;
        }
        return res;
    }
}
```

```
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;
}
}
```

# 797. 所有可能的路径

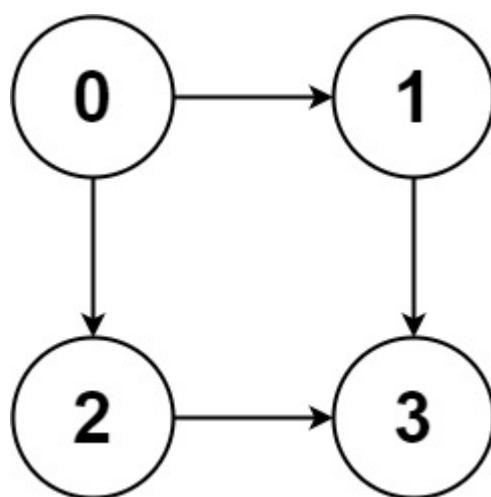


- 标签: 数据结构, 图论算法

给你一个有  $n$  个节点的有向无环图 (DAG) ,  $\text{graph}[i]$  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

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: [为什么我没写过「图」相关的算法?](#)

## 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;
    }

    void traverse(int[][] graph, int s, LinkedList<Integer> path) {
        path.add(s);
        if (s == graph.length - 1) {
            res.add(new ArrayList(path));
        } else {
            for (int t : graph[s]) {
                traverse(graph, t, path);
            }
        }
        path.removeLast();
    }
}
```

```
}

/* 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();
}
}
```

# 785. 判断二分图



- 标签: 图论算法, 二分图

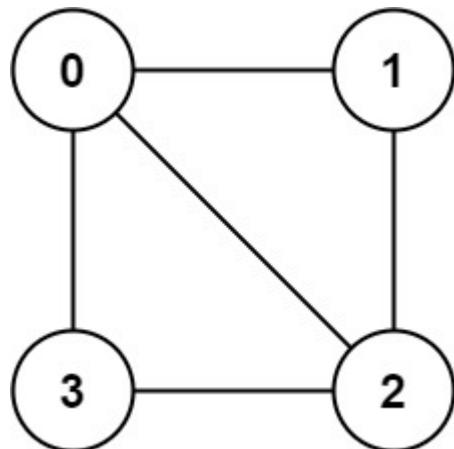
存在一个无向图，图中有  $n$  个节点。其中每个节点都有一个介于  $0$  到  $n - 1$  之间的唯一编号。给你一个二维数组  $\text{graph}$ ，其中  $\text{graph}[u]$  是一个节点数组，由节点  $u$  的邻接节点组成。形式上，对于  $\text{graph}[u]$  中的每个  $v$ ，都存在一条位于节点  $u$  和节点  $v$  之间的无向边。该无向图同时具有以下属性：

- 1、不存在自环 ( $\text{graph}[u]$  不包含  $u$ )。
- 2、不存在平行边 ( $\text{graph}[u]$  不包含重复值)。
- 3、如果  $v$  在  $\text{graph}[u]$  内，那么  $u$  也应该在  $\text{graph}[v]$  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  $v$  之间可能存在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:  $\text{graph} = [[1,2,3], [0,2], [0,1,3], [0,2]]$

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

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

## 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;
                }
            }
        }
    }
}
```

```
        ok = false;
    }
}
}

}
```

- 类似题目：
  - [886. 可能的二分法（中等）](#)

# 886. 可能的二分法



- 标签: 图论算法, 二分图

给定一组  $N$  人 (编号为  $1, 2, \dots, N$ ) , 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  $\text{dislikes}[i] = [a, b]$ , 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

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

## 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;
            }
        }
    }
}
}
```

- 类似题目：
  - 785. 判断二分图（中等）

# 207. 课程表



- 标签: 数据结构, 图论算法, 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

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。

- 详细题解: 拓扑排序, YYDS

## 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;  
}
```

- 类似题目：

- 210. 课程表 II

# 210. 课程表 II



- 标签: 数据结构, 图论算法, 拓扑排序

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:

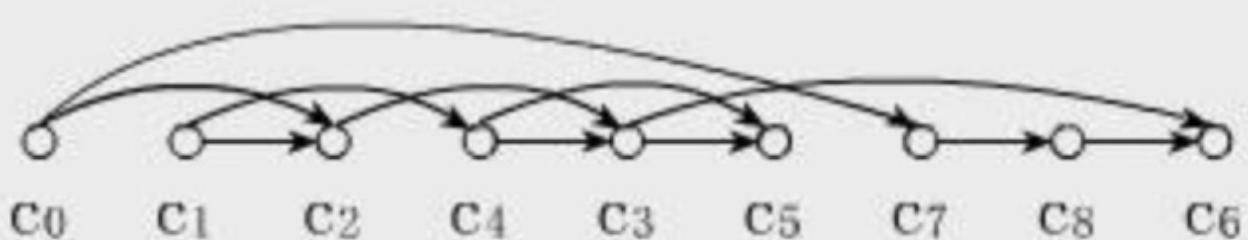
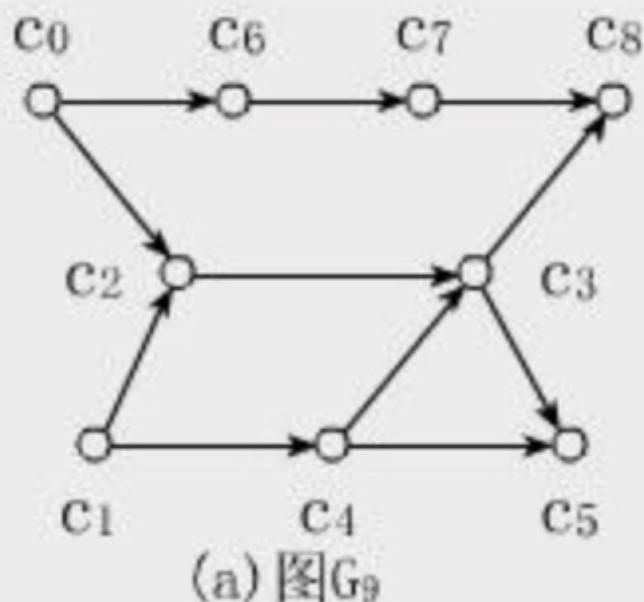
输入: `numCourses = 2, prerequisites = [[1,0]]`

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

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:

(b) 图 $G_9$ 的拓补次序排列

表示课程之间依赖关系的有向图

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。

拓扑排序本身特别简单，记结论：

将后序遍历的结果进行反转（逆后序遍历顺序），就是拓扑排序的结果。

跟多讨论请看详细题解。

- 详细题解：拓扑排序，YYDS

## 解法代码

```
class Solution {
    // 记录后序遍历结果
    List<Integer> postorder = new ArrayList<>();
    // 记录是否存在环
    boolean hasCycle = false;
    boolean[] visited, onPath;

    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = buildGraph(numCourses, prerequisites);
        visited = new boolean[numCourses];
        onPath = new boolean[numCourses];
        // 遍历图
        for (int i = 0; i < numCourses; i++) {
            traverse(graph, i);
        }
        // 有环图无法进行拓扑排序
        if (hasCycle) {
            return new int[]{};
        }
        // 逆后序遍历结果即为拓扑排序结果
        Collections.reverse(postorder);
        int[] res = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            res[i] = postorder.get(i);
        }
        return res;
    }

    void traverse(List<Integer>[] graph, int s) {
        if (onPath[s]) {
            hasCycle = true;
        }
        if (visited[s] || hasCycle) {
            return;
        }
        // 前序遍历位置
        onPath[s] = true;
        visited[s] = true;
        for (int t : graph[s]) {
            traverse(graph, t);
        }
        // 后序遍历位置
        postorder.add(s);
        onPath[s] = false;
    }

    // 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];  
        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;  
}
```

- 类似题目：
  - [207. 课程表](#)

# 130. 被围绕的区域



- 标签: **DFS 算法, 并查集算法**

给你一个  $m \times n$  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:

X	X	X	X
X	O	O	X
X	X	O	X
X	O	X	X

X	X	X	X
X	X	X	X
X	X	X	X
X	O	X	X

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" ]]`

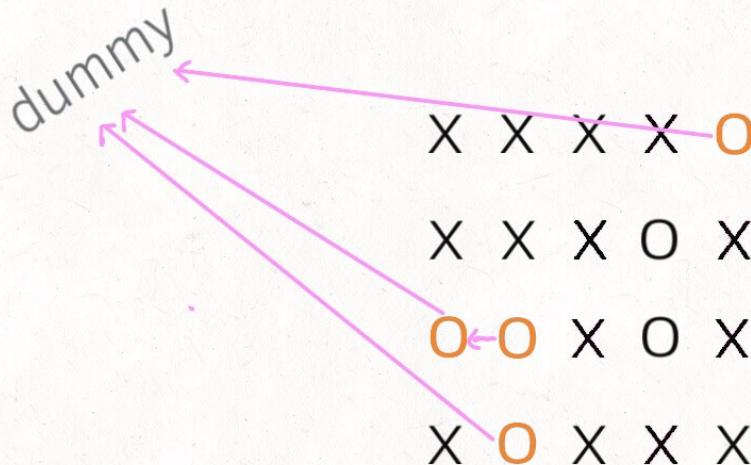
解释: 被围绕的区间不会存在于边界上, 换句话说, 任何边界上的 '`O`' 都不会被填充为 '`X`'。任何不在边界上, 或不与边界上的 '`O`' 相连的 '`O`' 最终都会被填充为 '`X`'。如果两个元素在水平或垂直方向相邻, 则称它们是“相连”的。

## 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:



公众号: labuladong

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: [Union-Find 算法怎么应用?](#)

## 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') {
                    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, dummy);
                    }
                }
            }
        }
    }
}
```

```
for (int i = 1; i < m - 1; i++)
    for (int j = 1; j < n - 1; j++) {
        if (board[i][j] == '0') {
            // 将此 0 与上下左右的 0 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] == '0')
                    uf.union(x * n + y, i * n + j);
            }
        }
    }
}

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：

- [990. 等式方程的可满足性](#) (中等)

# 990. 等式方程的可满足性



- 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: [Union-Find 算法怎么应用?](#)

## 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（中等）](#)

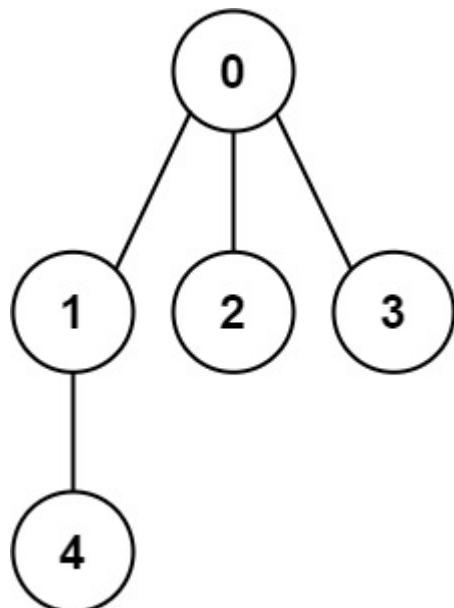
# 261. 以图判树



- 标签: 并查集算法, 最小生成树, 图论算法

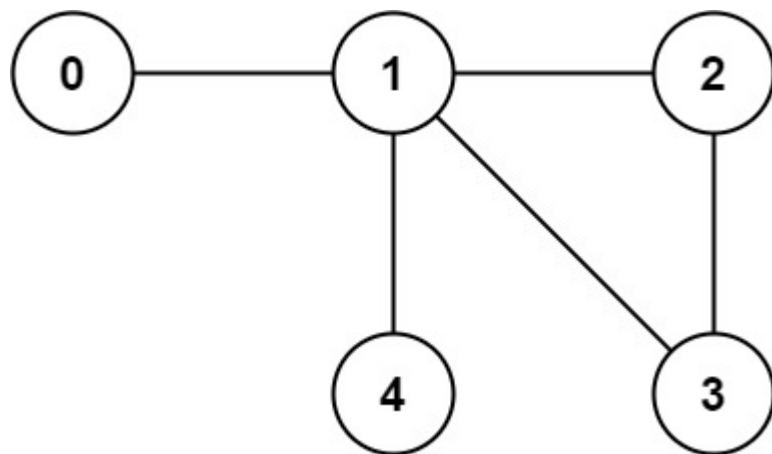
给定从  $0$  到  $n-1$  标号的  $n$  个结点, 和一个无向边列表 (每条边以结点对来表示), 请编写一个函数用来判断这些边是否能够形成一个合法有效的树结构。

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:

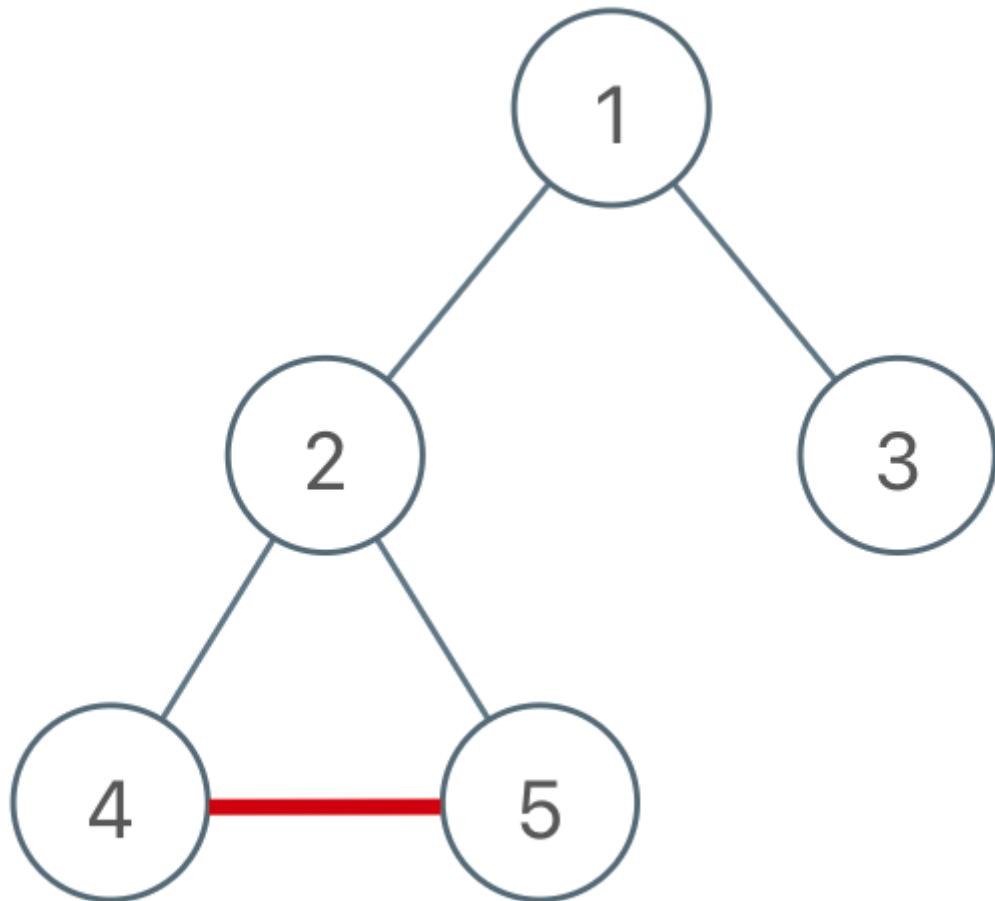


```
输入: n = 5, 边列表 edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]  
输出: false
```

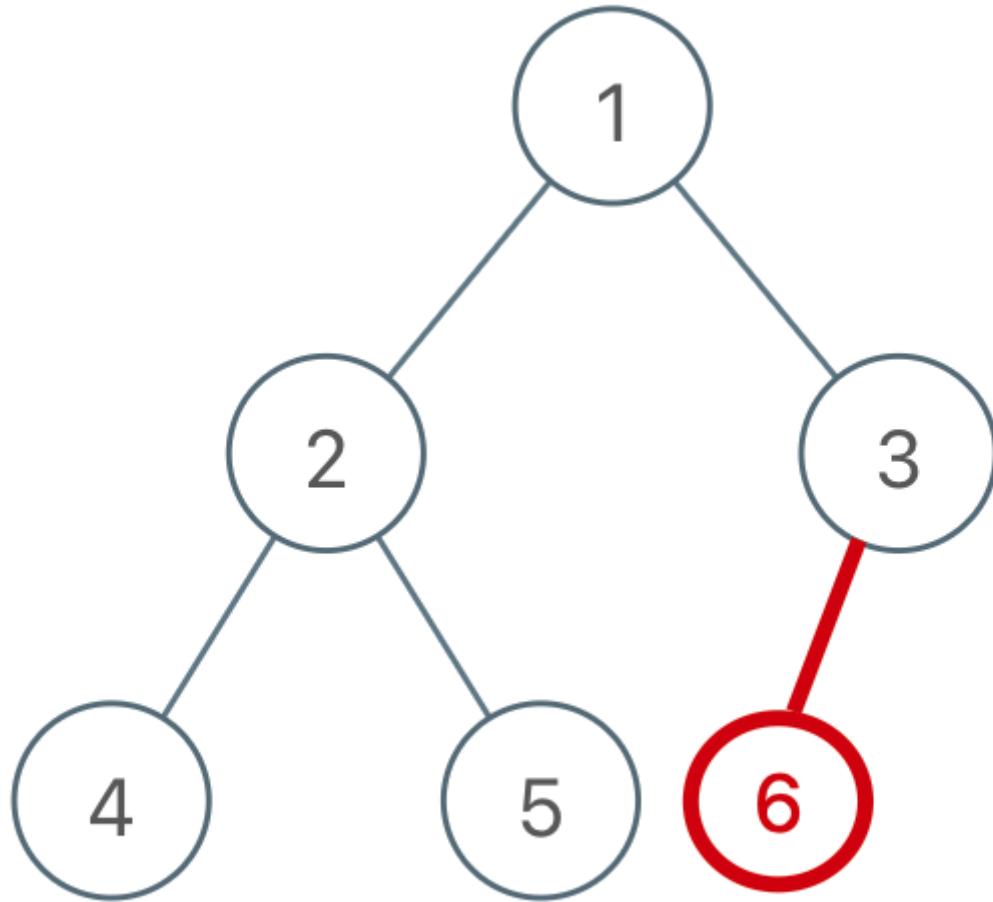
## 基本思路

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：[东哥带你刷图论第五期：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.isConnected(u, v)) return false;
            uf.union(u, v);
        }
        return uf.getCount() == 1;
    }
}
```

```
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（中等）
- 1584. 连接所有点的最小费用（中等）

# 1135. 最低成本联通所有城市



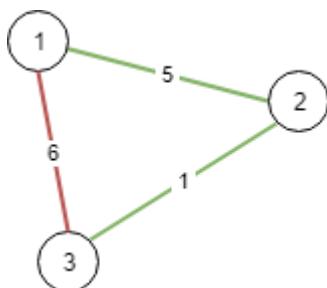
- 标签: 并查集算法, 最小生成树, 图论算法

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: 东哥带你刷图论第五期: Kruskal 最小生成树算法

## 解法代码

```
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：
  - [261. 以图判树（中等）](#)
  - [1584. 连接所有点的最小费用（中等）](#)

# 1584. 连接所有点的最小费用



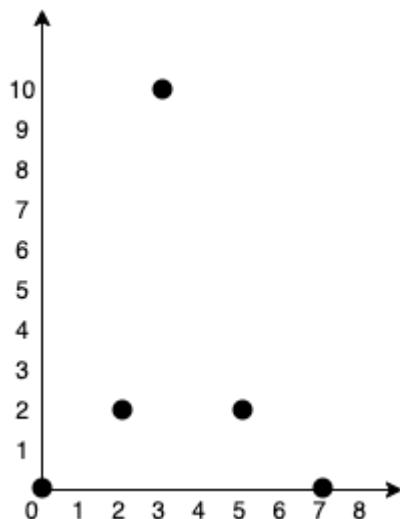
- 标签: 并查集算法, 最小生成树, 图论算法

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：东哥带你刷图论第五期：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：

- 261. 以图判树（中等）
- 1135. 最低成本联通所有城市（中等）

# 743. 网络延迟时间

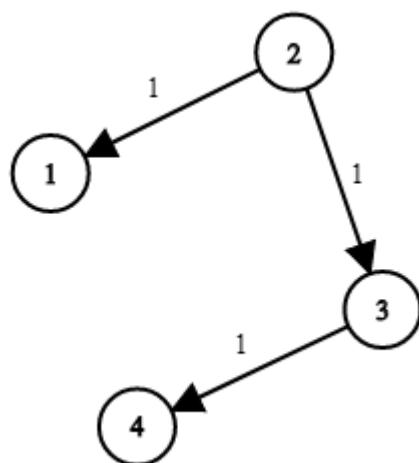


- 标签: 图论算法, Dijkstra 算法, 最短路径算法

有  $n$  个网络节点, 标记为 1 到  $n$ , 给你一个列表  $\text{times}$ , 表示信号经过有向边的传递时间。 $\text{times}[i] = (\text{ui}, \text{vi}, \text{wi})$ , 其中  $\text{ui}$  是源节点,  $\text{vi}$  是目标节点,  $\text{wi}$  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:  $\text{times} = [[2,1,1], [2,3,1], [3,4,1]]$ ,  $n = 4$ ,  $k = 2$   
 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: 我写了一个模板, 把 Dijkstra 算法变成了默写题

## 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;
}
```

- 类似题目：

- 1514. 概率最大的路径（中等）
- 1631. 最小体力消耗路径（中等）

# 1514. 概率最大的路径



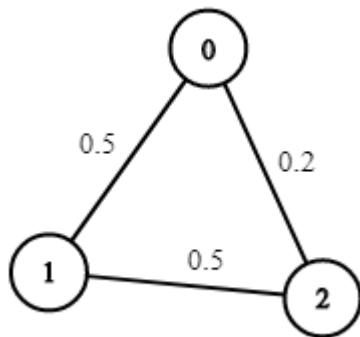
- 标签: 图论算法, Dijkstra 算法, 最短路径算法

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  $\text{edges}[i] = [a, b]$  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  $\text{succProb}[i]$ 。

指定两个节点分别作为起点  $\text{start}$  和终点  $\text{end}$ , 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  $\text{start}$  到  $\text{end}$  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:  $n = 3$ ,  $\text{edges} = [[0,1],[1,2],[0,2]]$ ,  $\text{succProb} = [0.5,0.5,0.2]$ ,  $\text{start} = 0$ ,  $\text{end} = 2$

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: 我写了一个模板, 把 Dijkstra 算法变成了默写题

## 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
    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;
}

}
```

- 类似题目：

- [743. 网络延迟时间（中等）](#)
- [1631. 最小体力消耗路径（中等）](#)

# 1631. 最小体力消耗路径

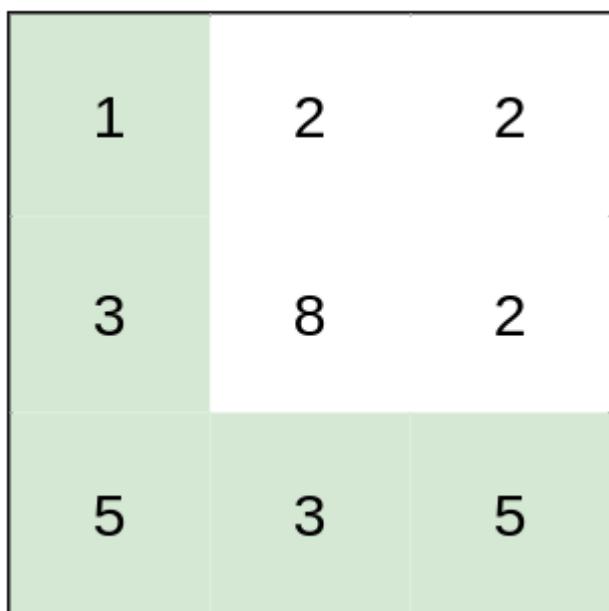


- 标签: 图论算法, Dijkstra 算法, 最短路径算法

你准备参加一场远足活动。给你一个二维  $rows \times columns$  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：我写了一个模板，把 Dijkstra 算法变成了默写题

## 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：
  - 743. 网络延迟时间（中等）
  - 1514. 概率最大的路径（中等）

# 17. 电话号码的字母组合



- 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 [回溯算法之子集、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);
        }
    }
}
}
```

## 22. 括号生成



- 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  $0 \leq i < \text{len}(p)$  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  $\text{left}$  记录还可以使用多少个左括号, 用  $\text{right}$  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: 合法括号生成](#)

### 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 right 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
}
}
```

## 37. 解数独



- 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：

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],  
[4,2,6,8,3,7,9,1,5],  
[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],  
[3,5,4,8,6,9,7,2,1]]
```

[ "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：

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

## 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  $3 \times 3$  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：搞懂回溯算法，我终于能做数独

## 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)) {
                board[i][j] = ch;
                if (backtrack(board, i, j + 1)) {
                    return true;
                }
                board[i][j] = '.';
            }
        }
        return false;
    }

    boolean isValid(char[][] board, int i, int j, char num) {
        // 检查行
        for (int col = 0; col < 9; col++) {
            if (board[i][col] == num) {
                return false;
            }
        }
        // 检查列
        for (int row = 0; row < 9; row++) {
            if (board[row][j] == num) {
                return false;
            }
        }
        // 检查 3x3 区域
        int startRow = (i / 3) * 3;
        int startCol = (j / 3) * 3;
        for (int row = startRow; row < startRow + 3; row++) {
            for (int col = startCol; col < startCol + 3; col++) {
                if (board[row][col] == num) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

```
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 × 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



- 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

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？

## 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();
        }
    }
}
```



# 46. 全排列



- 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。

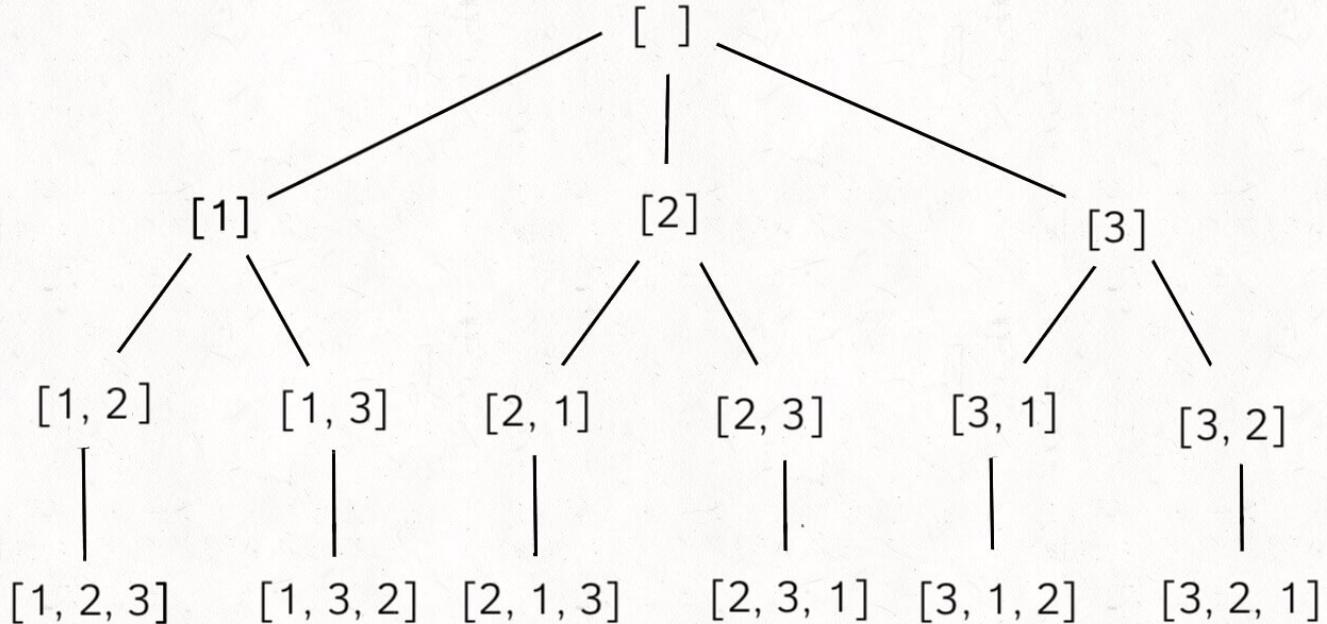
示例 1:

```
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

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:



公众号: labuladong

写代码遍历这棵回溯树即可。

- 详细题解: 回溯算法团灭排列/组合/子集问题

## 解法代码

```
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 */
public List<List<Integer>> permute(int[] nums) {
    // 记录「路径」
    LinkedList<Integer> track = new LinkedList<>();
    backtrack(nums, track);
    return res;
}

void backtrack(int[] nums, LinkedList<Integer> track) {
    // 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 (track.contains(nums[i]))
            continue;
        // 做选择
        track.add(nums[i]);
        // 进入下一层决策树
        backtrack(nums, track);
        // 取消选择
        track.removeLast();
    }
}
}
```

- 类似题目：

- 78. 子集（中等）
- 77. 组合（中等）

# 77. 组合



- 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
```

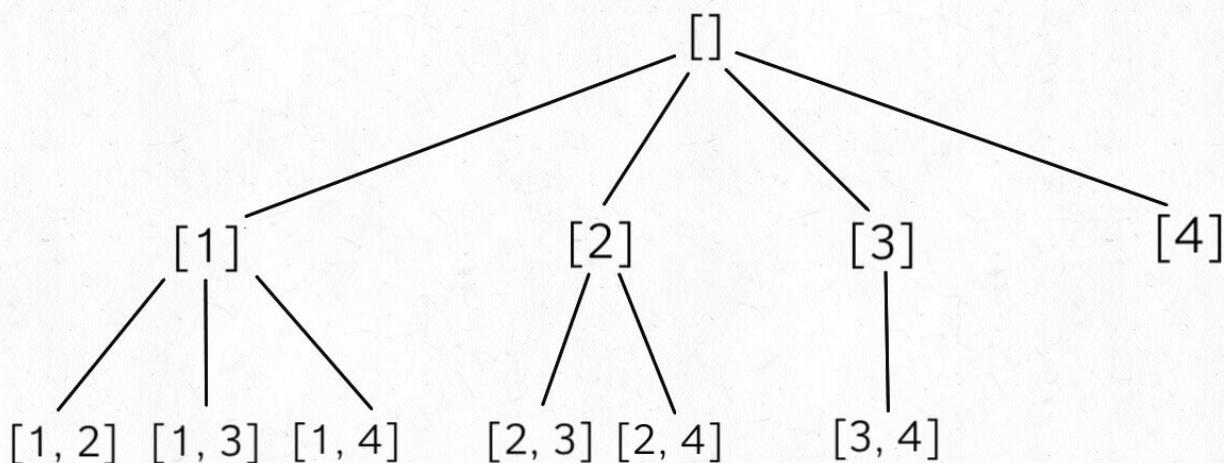
```
输出:
```

```
[  
    [2,4],  
    [3,4],  
    [2,3],  
    [1,2],  
    [1,3],  
    [1,4],  
]
```

## 基本思路

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：回溯算法团灭排列/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();
        }
    }
};
```

- 类似题目：

- 78. 子集（中等）
- 46. 全排列（中等）

## 78. 子集



- 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:

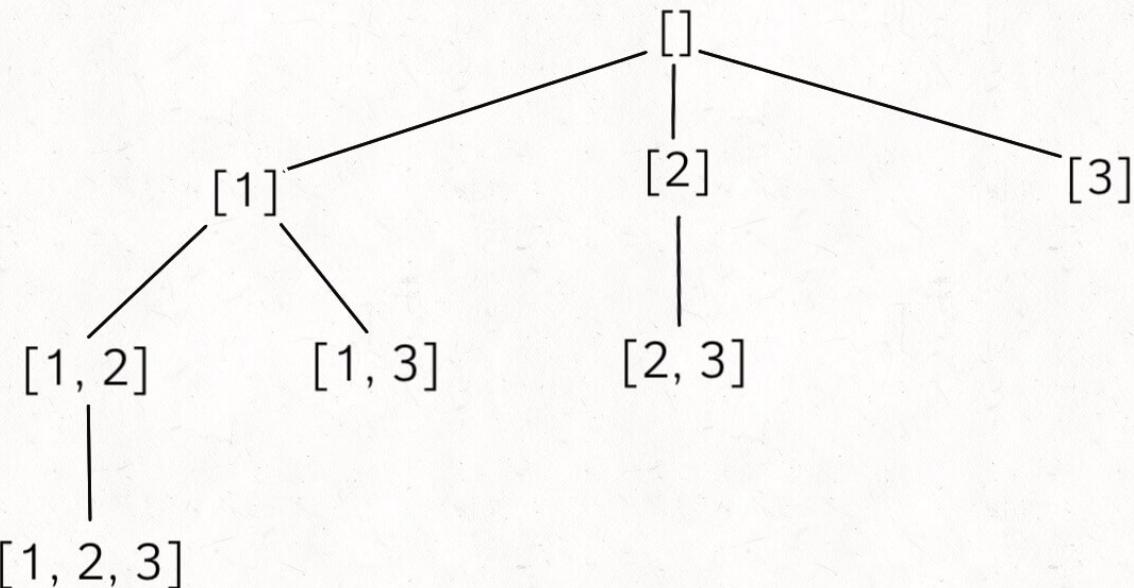
```
输入: nums = [1,2,3]
输出: [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
```

### 基本思路

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, 可以套旧文 [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:



公众号: labuladong

- 详细题解: [回溯算法团灭排列/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();
        }
    }
};
```

- 类似题目：

- [46. 全排列（中等）](#)
- [77. 组合（中等）](#)

# 51. N 皇后

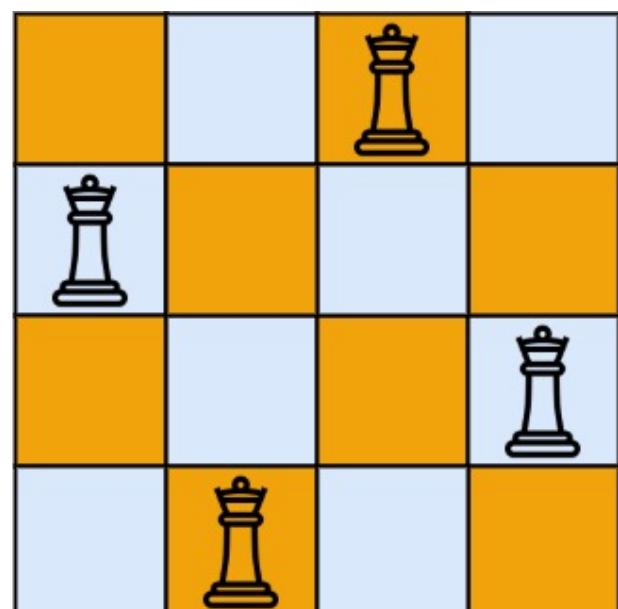
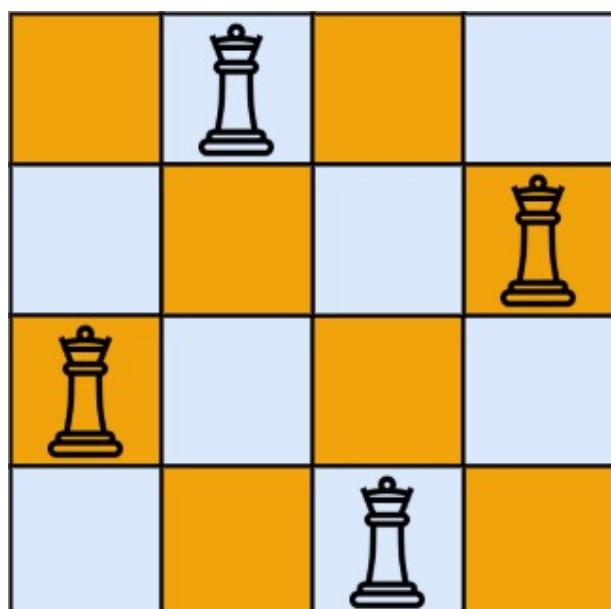


- 标签: 回溯算法

**n 皇后问题** 研究的是如何将  $n$  个皇后放置在  $n \times n$  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)
```

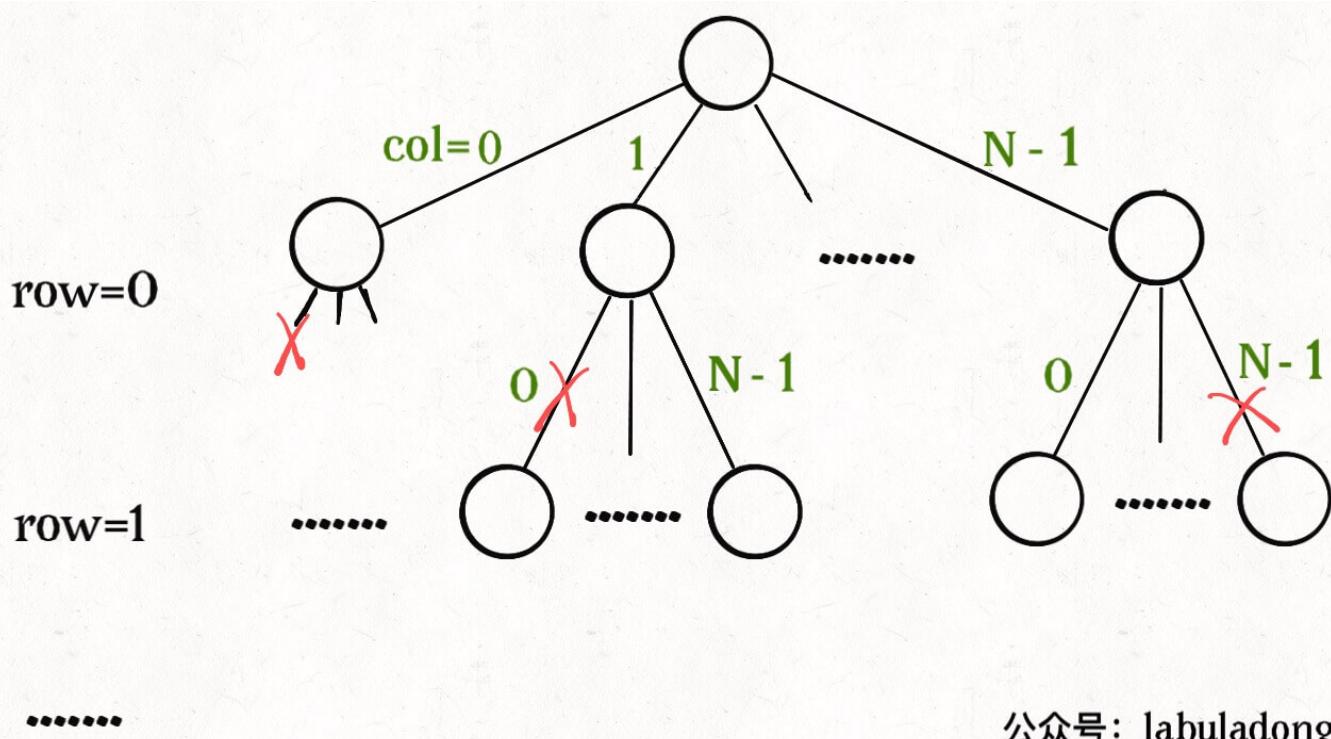
```

    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：



公众号： labuladong

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：[回溯算法详解（修订版）](#)

## 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 < n; 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;  
}  
};
```

- 类似题目：
  - 46. 全排列（中等）

# 104. 二叉树的最大深度



- 标签: 二叉树, 回溯算法, 动态规划

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

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。

## 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++;
        if (root.left == null && root.right == null) {
            res = Math.max(res, depth);
        }
        traverse(root.left);
        traverse(root.right);
        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);
    }
}
```

# 494. 目标和



- 标签: 背包问题, 回溯算法, 动态规划, 二维动态规划

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: 回溯算法和动态规划，到底谁是谁爹？

## 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 rest) {
    // base case
    if (i == nums.length) {
        if (rest == 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 + "," + rest;
    // 避免重复计算
    if (memo.containsKey(key)) {
        return memo.get(key);
    }
    // 还是穷举
    int result = dp(nums, i + 1, rest - nums[i]) + dp(nums, i + 1,
    rest + nums[i]);
    // 记入备忘录
    memo.put(key, result);
    return result;
}
```

# 698. 划分为 k 个相等的子集



- 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。

- 详细题解: [回溯算法牛逼!](#)

## 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;

        boolean[] used = new boolean[nums.length];
        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);
    }

    boolean backtrack(int k, int bucket,
                     int[] nums, int start, boolean[] used, int target) {
        // base case
        if (k == 0) return true;
        if (bucket == target) return backtrack(k - 1, 0, nums, 0, used, target);
        if (bucket > target) return false;

        for (int i = start; i < nums.length; i++) {
            if (used[i]) continue;
            if (bucket + nums[i] > target) break;
            used[i] = true;
            if (backtrack(k, bucket + nums[i], nums, i + 1, used, target)) return true;
            used[i] = false;
        }
        return false;
    }
}
```

```
if (k == 0) {
    // 所有桶都被装满了，而且 nums 一定全部用完了
    // 因为 target == sum / k
    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
    return backtrack(k - 1, 0, nums, 0, used, target);
}

// 从 start 开始向后探查有效的 nums[i] 装入当前桶
for (int i = start; i < nums.length; i++) {
    // 剪枝
    if (used[i]) {
        // nums[i] 已经被装入别的桶中
        continue;
    }
    if (nums[i] + bucket > target) {
        // 当前桶装不下 nums[i]
        continue;
    }
    // 做选择，将 nums[i] 装入当前桶中
    used[i] = true;
    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[i] = false;
    bucket -= nums[i];
}
// 穷举了所有数字，都无法装满当前桶
return false;
}
```

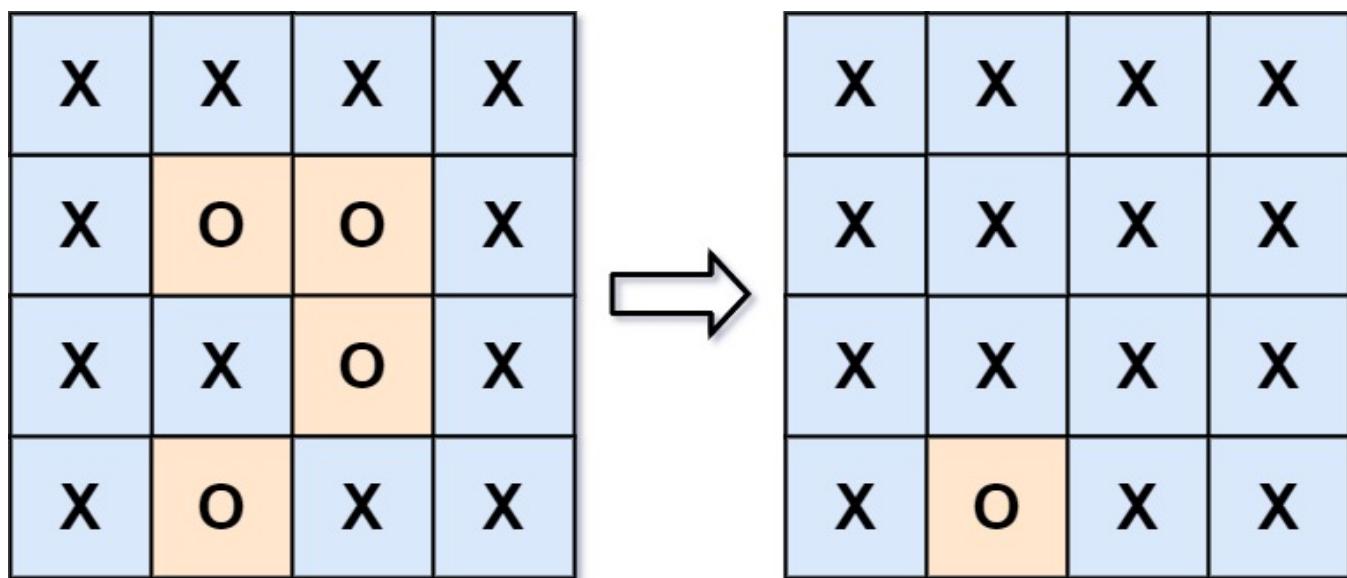
# 130. 被围绕的区域



- 标签: **DFS 算法, 并查集算法**

给你一个  $m \times n$  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" ]]`

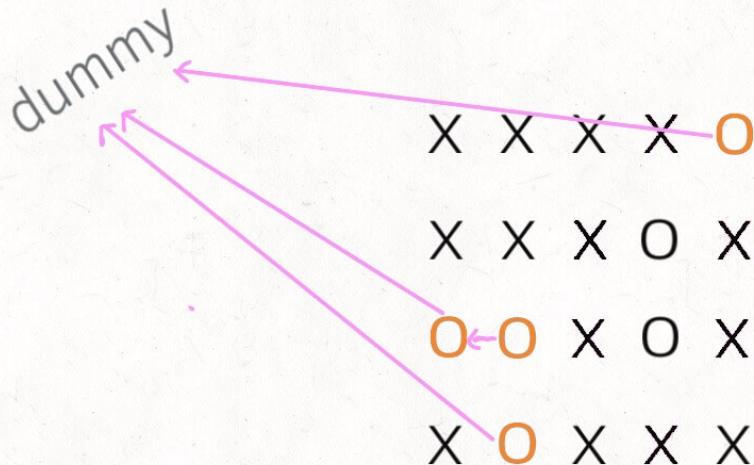
解释: 被围绕的区间不会存在于边界上, 换句话说, 任何边界上的 '`O`' 都不会被填充为 '`X`'。任何不在边界上, 或不与边界上的 '`O`' 相连的 '`O`' 最终都会被填充为 '`X`'。如果两个元素在水平或垂直方向相邻, 则称它们是“相连”的。

## 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:



公众号: labuladong

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: [Union-Find 算法怎么应用?](#)

## 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 0 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 0 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' && !uf.connected(i * n + j, dummy))
                    uf.union(i * n + j, dummy);
    }
}
```

```
for (int i = 1; i < m - 1; i++)
    for (int j = 1; j < n - 1; j++) {
        if (board[i][j] == '0') {
            // 将此 0 与上下左右的 0 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] == '0')
                    uf.union(x * n + y, i * n + j);
            }
        }
    }
}

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：

- [990. 等式方程的可满足性](#) (中等)

# 200. 岛屿数量



- 标签: **DFS 算法, 二维矩阵**

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格, 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: [DFS 算法秒杀五道岛屿题目](#)

## 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);
}
```

- 类似题目：

- [1254. 统计封闭岛屿的数目](#) (中等)
- [1020. 飞地的数量](#)
- [695. 岛屿的最大面积](#)
- [1905. 统计子岛屿](#)
- [694. 不同的岛屿数量](#)

# 694. 不同的岛屿数量



- 标签: **DFS 算法, 二维矩阵**

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，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:

1	1	0	0	0
1	1	0	0	0
0	0	0	1	1
0	0	0	1	1

给定上图，返回结果 1。

示例 2:

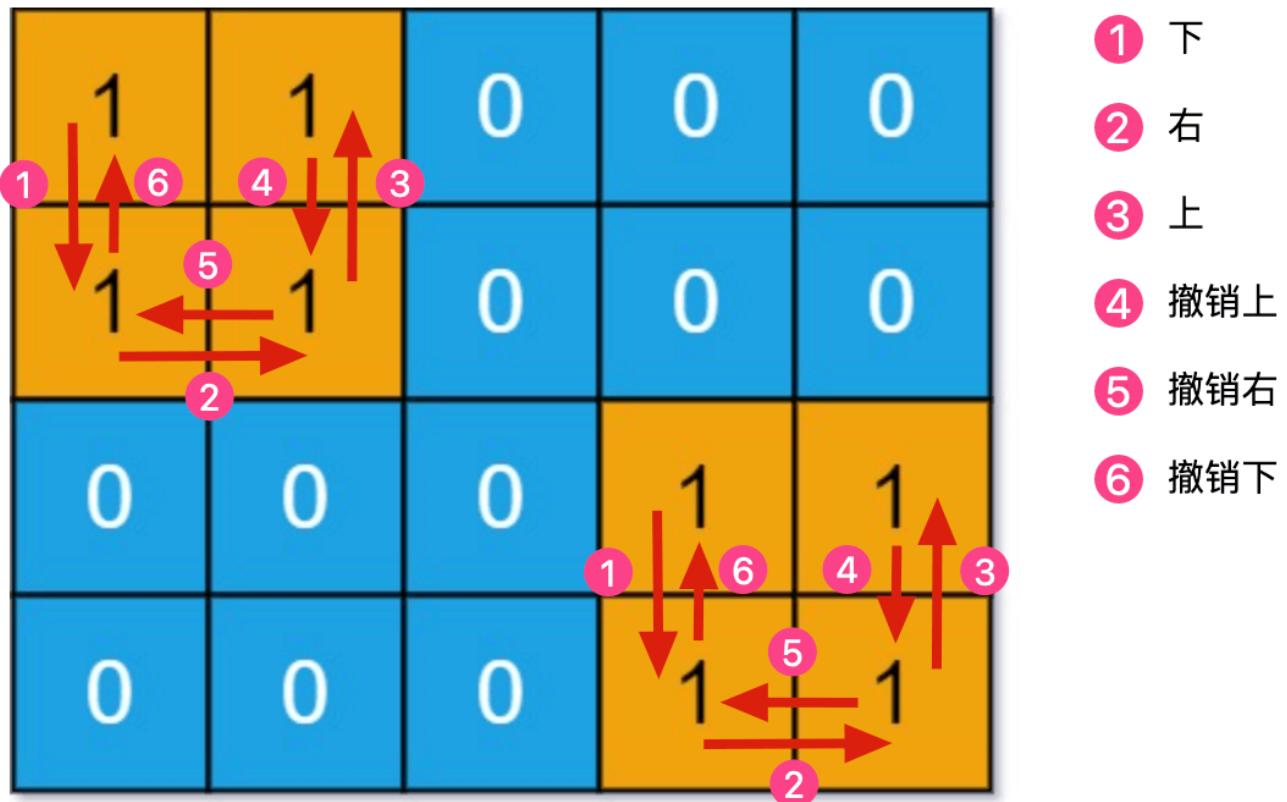
1	1	0	1	1
1	0	0	0	0
0	0	0	0	1
1	1	0	1	1

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: [DFS 算法秒杀五道岛屿题目](#)

## 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(',', ',');
    }
}
```

{  
}

- 类似题目：

- 200. 岛屿数量
- 1254. 统计封闭岛屿的数目（中等）
- 1020. 飞地的数量
- 695. 岛屿的最大面积
- 1905. 统计子岛屿

# 695. 岛屿的最大面积



- 标签: 二维矩阵, DFS 算法

给你一个大小为  $m \times n$  的二维矩阵  $\text{grid}$ , 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  $\text{grid}$  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  $\text{grid}$  中最大的岛屿面积。如果没有岛屿, 则返回面积为 0。

示例 1:

0	0	1	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	0	1	0	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	0	1	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0

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

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：DFS 算法秒杀五道岛屿题目

## 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;
    }
}
```

- 类似题目：

- 200. 岛屿数量（中等）
- 1254. 统计封闭岛屿的数目（中等）
- 1020. 飞地的数量
- 1905. 统计子岛屿

- 694. 不同的岛屿数量

# 1020. 飞地的数量



- 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:

0	0	0	0
1	0	1	0
0	1	1	0
0	0	0	0

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: [DFS 算法秒杀五道岛屿题目](#)

## 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);
    }
}
```

- 类似题目：

- [200. 岛屿数量](#) (中等)
- [1254. 统计封闭岛屿的数目](#) (中等)
- [695. 岛屿的最大面积](#)
- [1905. 统计子岛屿](#)
- [694. 不同的岛屿数量](#)

# 1254. 统计封闭岛屿的数目



- 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:

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],  
[1,0,0,0,1,0,1],[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: **DFS 算法秒杀五道岛屿题目**

## 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);
    }
}
```

- 类似题目：

- 200. 岛屿数量（中等）

- 1020. 飞地的数量
- 695. 岛屿的最大面积
- 1905. 统计子岛屿
- 694. 不同的岛屿数量

# 1905. 统计子岛屿



- 标签: **DFS 算法, 二维矩阵**

给你两个  $m \times n$  的二进制矩阵  $\text{grid1}$  和  $\text{grid2}$ , 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  $\text{grid2}$  的一个岛屿, 被  $\text{grid1}$  的一个岛屿完全包含, 也就是说  $\text{grid2}$  中该岛屿的每一个格子都被  $\text{grid1}$  中同一个岛屿完全包含, 那么我们称  $\text{grid2}$  中的这个岛屿为子岛屿。

请你返回  $\text{grid2}$  中子岛屿的数目。

**示例 1:**

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

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  $\text{grid1}$ , 右边为  $\text{grid2}$ 。  
 $\text{grid2}$  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  $\text{grid2}$  中的一个岛屿 **B** 是  $\text{grid1}$  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  $\text{grid2}$  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: DFS 算法秒杀五道岛屿题目**

## 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);
    }
}
```

- 类似题目：

- [200. 岛屿数量](#) (中等)
- [1254. 统计封闭岛屿的数目](#) (中等)
- [1020. 飞地的数量](#)
- [695. 岛屿的最大面积](#)
- [694. 不同的岛屿数量](#)

## 102. 二叉树的层序遍历

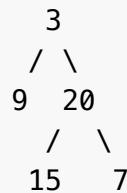


- 标签: [二叉树](#), [BFS 算法](#)

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],`



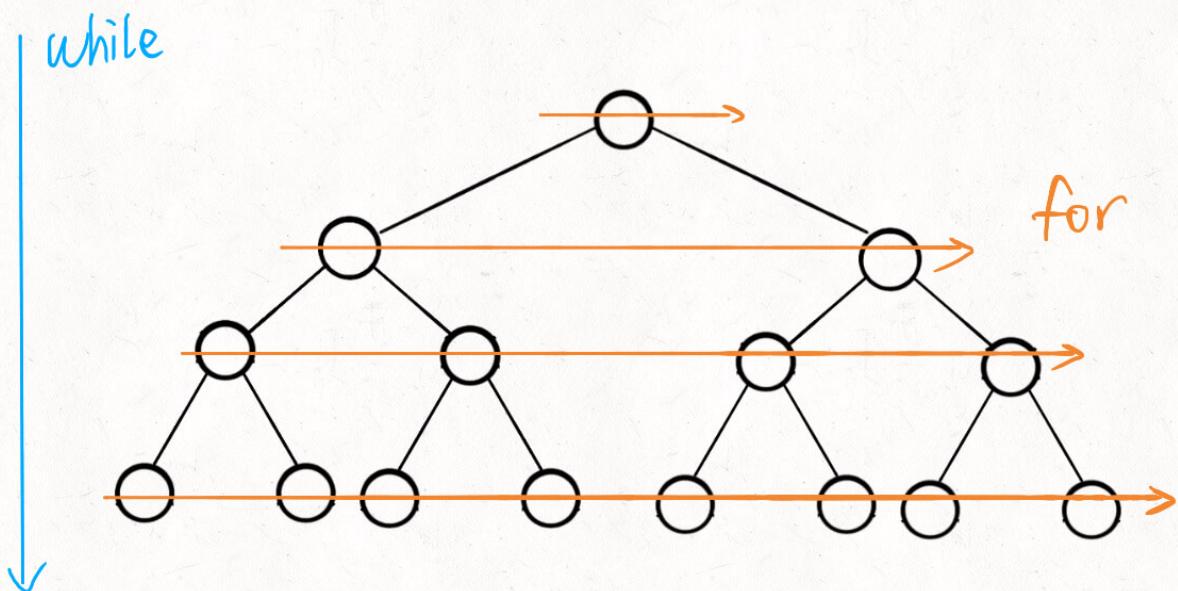
返回其层序遍历结果：

```
[  
  [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：



公众号: labuladong

## 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;
    }
}
```

# 103. 二叉树的锯齿形层序遍历

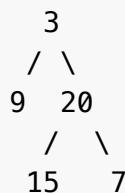


- 标签: 二叉树, BFS 算法

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]，



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;  
}  
}
```

# 107. 二叉树的层序遍历 II

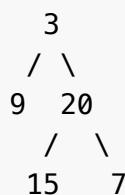


- 标签: 二叉树, BFS 算法

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]，



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;
}
```

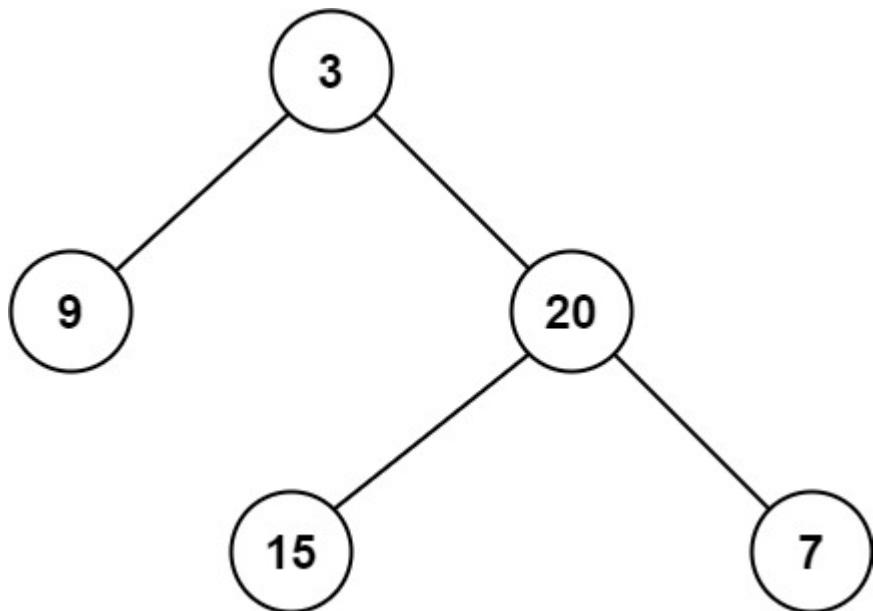
# 111. 二叉树的最小深度



- 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

PS: 这道题在《算法小抄》的第 53 页。

基本的二叉树层序遍历方法，值得一提的是，BFS 算法框架就是二叉树层序遍历代码的衍生。

BFS 算法和 DFS（回溯）算法的一大区别就是，BFS 第一次搜索到的结果是最优的，这个得益于 BFS 算法的搜索逻辑，可见详细题解。

- 详细题解: **BFS 算法框架套路详解**

## 解法代码

```
class Solution {
    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;
}
```

- 类似题目：
  - [752. 打开转盘锁（中等）](#)

# 752. 打开转盘锁



- 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

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（简单）](#)

# 773. 滑动谜题



- 标签：字符串，[BFS 算法](#)

在一个  $2 \times 3$  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：[益智游戏克星：BFS 暴力搜索算法](#)

## 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);
                    }
                }
            }
        }
        return -1;
    }

    private char[] swap(char[] arr, int i, int j) {
        char temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        return arr;
    }
}
```

```
        }
    }
    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);
}

}
```

# 45. 跳跃游戏 II



- 标签: 动态规划, 贪心算法, 一维动态规划

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。

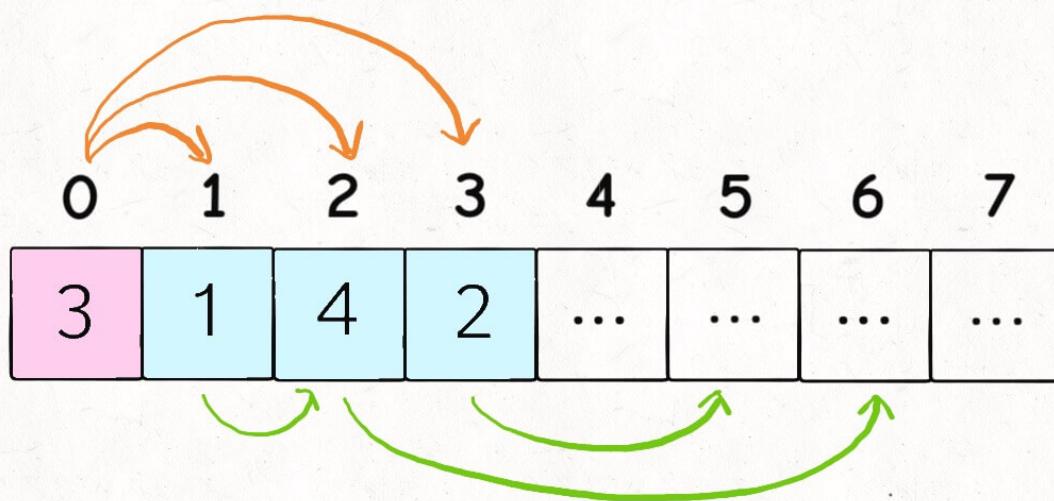
从下标为 0 跳到下标为 1 的位置, 跳 1 步, 然后跳 3 步到达数组的最后一个位置。

## 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。



公众号: labuladong

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 步跳到索引 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：[经典贪心算法：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;
    }
}
```

- 类似题目：
  - [55. 跳跃游戏（中等）](#)

# 55. 跳跃游戏



- 标签: 动态规划, 贪心算法, 一维动态规划

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: 经典贪心算法: 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;
}
}
```

- 类似题目：
  - 45. 跳跃游戏 II (中等)

# 53. 最大子序和



- 标签: 动态规划, 数组, 一维动态规划

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 **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: 动态规划套路: 最大子数组和

## 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;
    }
}
```

# 70. 爬楼梯



- 标签: 动态规划, 一维动态规划

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](#)。

这题很像斐波那契数列：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：
        memo[n] = dp(n - 1) + dp(n - 2);
        return memo[n];
    }
}
```

```
// 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];  
}  
}
```

# 198. 打家劫舍



- 标签: 动态规划, 一维动态规划

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。

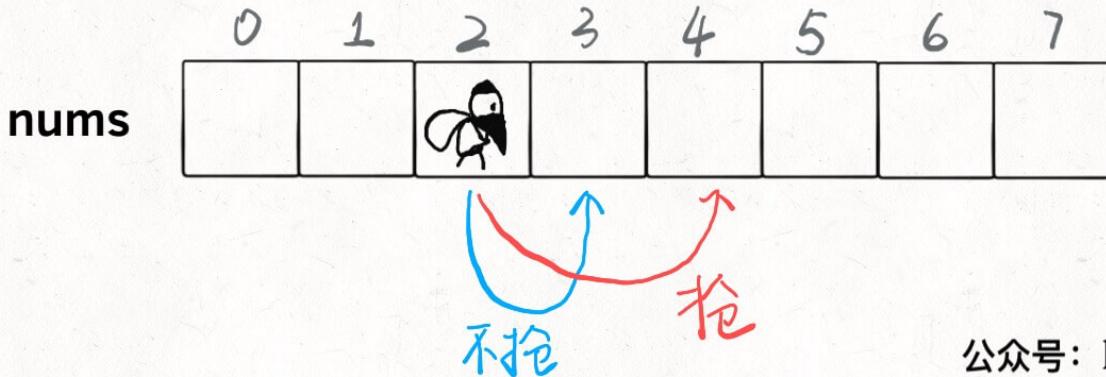
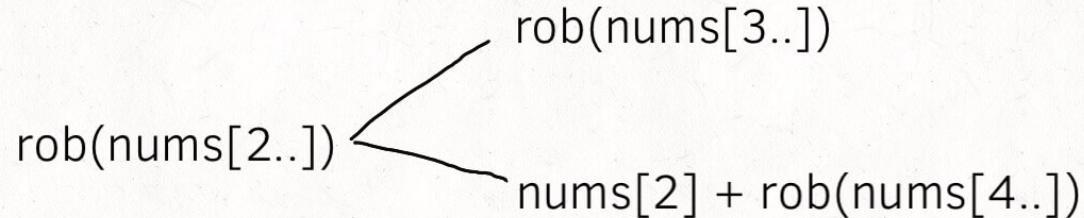
## 基本思路

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。



公众号: labuladong

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：[经典动态规划：打家劫舍系列问题](#)

## 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;
    }
}
```

- 类似题目：
  - [213. 打家劫舍 II \(中等\)](#)
  - [337. 打家劫舍 III \(中等\)](#)

## 213. 打家劫舍 II



- 标签: 动态规划, 一维动态规划

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。

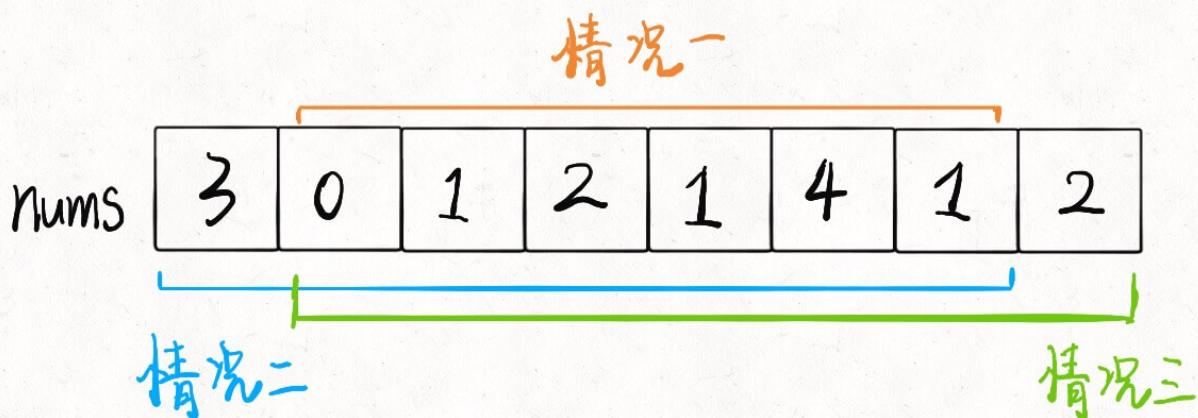
示例 1:

```
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。



公众号: labuladong

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：[经典动态规划：打家劫舍系列问题](#)

## 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;
    }
}
```

- 类似题目：
  - [198. 打家劫舍（简单）](#)
  - [337. 打家劫舍 III（中等）](#)

# 337. 打家劫舍 III

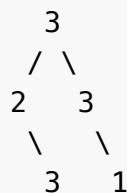


- 标签: 动态规划, 一维动态规划

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]



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: 经典动态规划: 打家劫舍系列问题

## 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_it = rob(root.left) + rob(root.right);
        memo.put(root, Math.max(do_it, not_it));
        return Math.max(do_it, not_it);
    }
}
```

```
int not_do = rob(root.left) + rob(root.right);

int res = Math.max(do_it, not_do);
memo.put(root, res);
return res;
}
}
```

- 类似题目：

- [198. 打家劫舍](#) (简单)
- [213. 打家劫舍 II](#) (中等)

# 300. 最长递增子序列



- 标签: 动态规划, 子序列, 一维动态规划

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: [从最长递增子序列学会如何推状态转移方程](#)

## 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;
    }
}
```

```
    }  
}
```

## 322. 零钱兑换



- 标签: 动态规划, 一维动态规划, 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

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: 动态规划详解 (修订版)

### 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);
    }

    private int dp(int[] coins, int amount) {
        if (amount == 0) return 0;
        if (memo[amount] != -666) return memo[amount];
        int res = Integer.MAX_VALUE;
        for (int coin : coins) {
            if (coin > amount) continue;
            int subProblemRes = dp(coins, amount - coin);
            if (subProblemRes != -1) res = Math.min(res, subProblemRes + 1);
        }
        memo[amount] = res == Integer.MAX_VALUE ? -1 : res;
        return memo[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];  
}  
}
```

- 类似题目：
  - [509. 斐波那契数（简单）](#)

# 354. 俄罗斯套娃信封问题



- 标签: 动态规划, 一维动态规划, 二分搜索

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:

宽度 w    高度 h

升序

[ 1 , 8 ]

[ 2 , 3 ]

[ 5 , 4 ] ]

[ 5 , 2 ] ]

降序

[ 6 , 7 ] ]

[ 6 , 4 ] ]

降序



然后在 h 上寻找最长递增子序列：

宽度 w    高度 h

[ 1 , 8 ]

[ 2 , 3 ]

[ 5 , 4 ]

[ 5 , 2 ]

[ 6 , 7 ]

[ 6 , 4 ]



- 详细题解：最长递增子序列之信封嵌套问题

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;
    }
}
```

# 10. 正则表达式匹配



- 标签: 动态规划, 字符串, 二维动态规划

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  $dp(s, i, p, j) = \text{true}$ , 则表示  $s[i..]$  可以匹配  $p[j..]$ ; 若  $dp(s, i, p, j) = \text{false}$ , 则表示  $s[i..]$  无法匹配  $p[j..]$ 。

- 详细题解: 东哥手写正则通配符算法, 结构清晰, 包教包会!

## 解法代码

```
class Solution {  
public:  
    bool isMatch(string s, string p) {  
        // 指针 i, j 从索引 0 开始移动  
        return dp(s, 0, p, 0);  
    }  
  
    // 备忘录  
    unordered_map<string, bool> memo;
```

```
/* 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;
    }
    // 记录状态 (i, j), 消除重叠子问题
    string key = to_string(i) + "," + to_string(j);
    if (memo.count(key)) return memo[key];

    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[key] = res;

    return res;
}
};
```

## 62. 不同路径



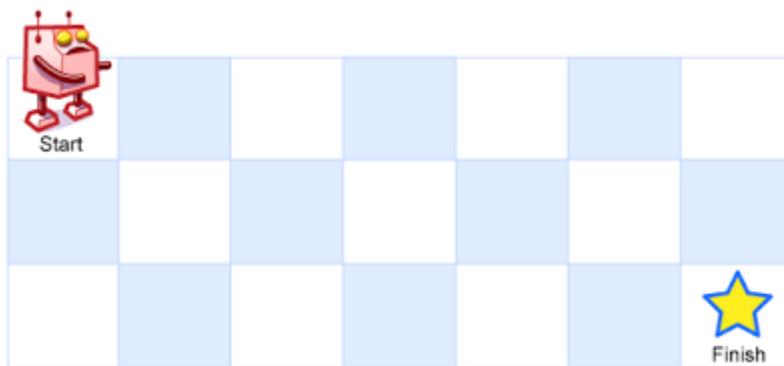
- 标签: [二维动态规划](#), [动态规划](#), [二维矩阵](#)

一个机器人位于一个  $m \times n$  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];
    }
}
```

## 64. 最小路径和



- 标签: 动态规划, 二维矩阵, 二维动态规划

给定一个包含非负整数的  $m \times n$  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:

1	3	1
1	5	1
4	2	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: 经典动态规划: 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];
    }
}
```

## 72. 编辑距离



- 标签: 动态规划, 二维动态规划

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

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]:  
    哪都别做 (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));  
    }  
}
```

# 121. 买卖股票的最佳时机



- 标签: 动态规划, 二维动态规划

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];
    }
}
```

- 类似题目：
  - [122.买卖股票的最佳时机 II](#) (简单)
  - [123.买卖股票的最佳时机 III](#) (困难)
  - [188.买卖股票的最佳时机 IV](#) (困难)
  - [309.最佳买卖股票时机含冷冻期](#) (中等)
  - [714.买卖股票的最佳时机含手续费](#) (中等)

# 122. 买卖股票的最佳时机 II



- 标签: 动态规划, 二维动态规划

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];
    }
}
```

- 类似题目：

- 121.买卖股票的最佳时机（简单）
- 123.买卖股票的最佳时机 III（困难）
- 188.买卖股票的最佳时机 IV（困难）
- 309.最佳买卖股票时机含冷冻期（中等）
- 714.买卖股票的最佳时机含手续费（中等）

# 123. 买卖股票的最佳时机 III



- 标签: 动态规划, 三维动态规划

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]`  
 $0 \leq i \leq n - 1, 1 \leq k \leq K$   
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：

$dp[-1][...][0] = dp[...][0][0] = 0$   
 $dp[-1][...][1] = dp[...][0][1] = -\infty$

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  $n \times \text{max\_k} \times 2$  个状态，正确。
        return dp[n - 1][max_k][0];
    }
}
```

- 类似题目：

- [121.买卖股票的最佳时机](#) (简单)
- [122.买卖股票的最佳时机 II](#) (简单)
- [188.买卖股票的最佳时机 IV](#) (困难)
- [309.最佳买卖股票时机含冷冻期](#) (中等)
- [714.买卖股票的最佳时机含手续费](#) (中等)

# 188. 买卖股票的最佳时机 IV



- 标签: 动态规划, 三维动态规划

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];
}
```

- 类似题目：

- [121.买卖股票的最佳时机](#) (简单)
- [122.买卖股票的最佳时机 II](#) (简单)
- [123.买卖股票的最佳时机 III](#) (困难)
- [309.最佳买卖股票时机含冷冻期](#) (中等)
- [714.买卖股票的最佳时机含手续费](#) (中等)

# 309. 最佳买卖股票时机含冷冻期



- 标签: 动态规划, 二维动态规划

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]);
                // = 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];
    }
}
```

- 类似题目：

- [121.买卖股票的最佳时机（简单）](#)
- [122.买卖股票的最佳时机 II（简单）](#)
- [123.买卖股票的最佳时机 III（困难）](#)

- 188.买卖股票的最佳时机 IV (困难)
- 714.买卖股票的最佳时机含手续费 (中等)

# 714. 买卖股票的最佳时机含手续费



- 标签: 动态规划, 三维动态规划

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。

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][...] = dp[...][0] = 0
dp[-1][...] = dp[...][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];
    }
}
```

- 类似题目：

- 121. 买卖股票的最佳时机（简单）
- 122. 买卖股票的最佳时机 II（简单）
- 123. 买卖股票的最佳时机 III（困难）

- 188.买卖股票的最佳时机 IV (困难)
- 309.最佳买卖股票时机含冷冻期 (中等)

# 174. 地下城游戏



- 标签: 动态规划, 二维矩阵, 二维动态规划

地下城是由  $M \times N$  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 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  $\rightarrow$  右  $\rightarrow$  下  $\rightarrow$  下，则骑士的初始生命值至少为 7。

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

## 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：动态规划算法帮我通关了魔塔！

## 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];
    }
}
```

# 312. 戳气球



- 标签: 动态规划, 二维动态规划

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  $\text{nums}$  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  $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$  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

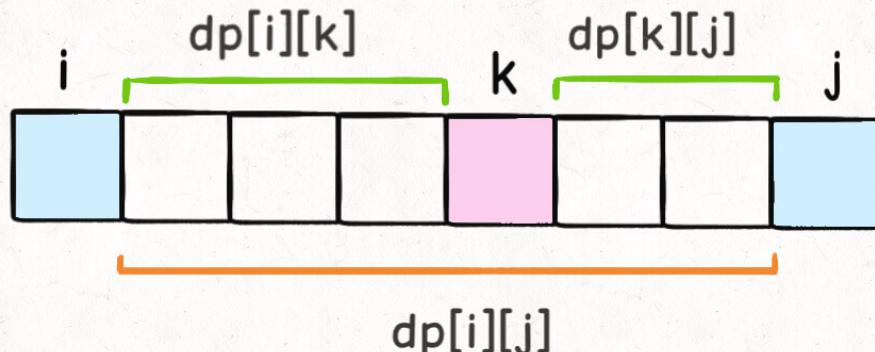
PS: 这道题在《算法小抄》的第 181 页。

这题比较难, 建议看详细题解。

$\text{dp}$  数组的含义:  $\text{dp}[i][j] = x$  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:

$$\text{dp}[i][j] = \text{dp}[i][k] + \text{dp}[k][j] + \text{points}[i]*\text{points}[k]*\text{points}[j]$$



公众号: labuladong

- 详细题解: 经典动态规划: 戳气球问题

## 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];
    }
}
```

# 416. 分割等和子集



- 标签: 动态规划, 背包问题, 二维动态规划

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: 0-1 背包问题的变体

## 解法代码

```
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];
}
```

# 494. 目标和



- 标签: 背包问题, 回溯算法, 动态规划, 二维动态规划

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: 回溯算法和动态规划, 到底谁是谁爹?

## 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 rest) {
    // base case
    if (i == nums.length) {
        if (rest == 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 + "," + rest;
    // 避免重复计算
    if (memo.containsKey(key)) {
        return memo.get(key);
    }
    // 还是穷举
    int result = dp(nums, i + 1, rest - nums[i]) + dp(nums, i + 1,
    rest + nums[i]);
    // 记入备忘录
    memo.put(key, result);
    return result;
}
```

# 514. 自由之路



- 标签: 动态规划, 二维动态规划

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。

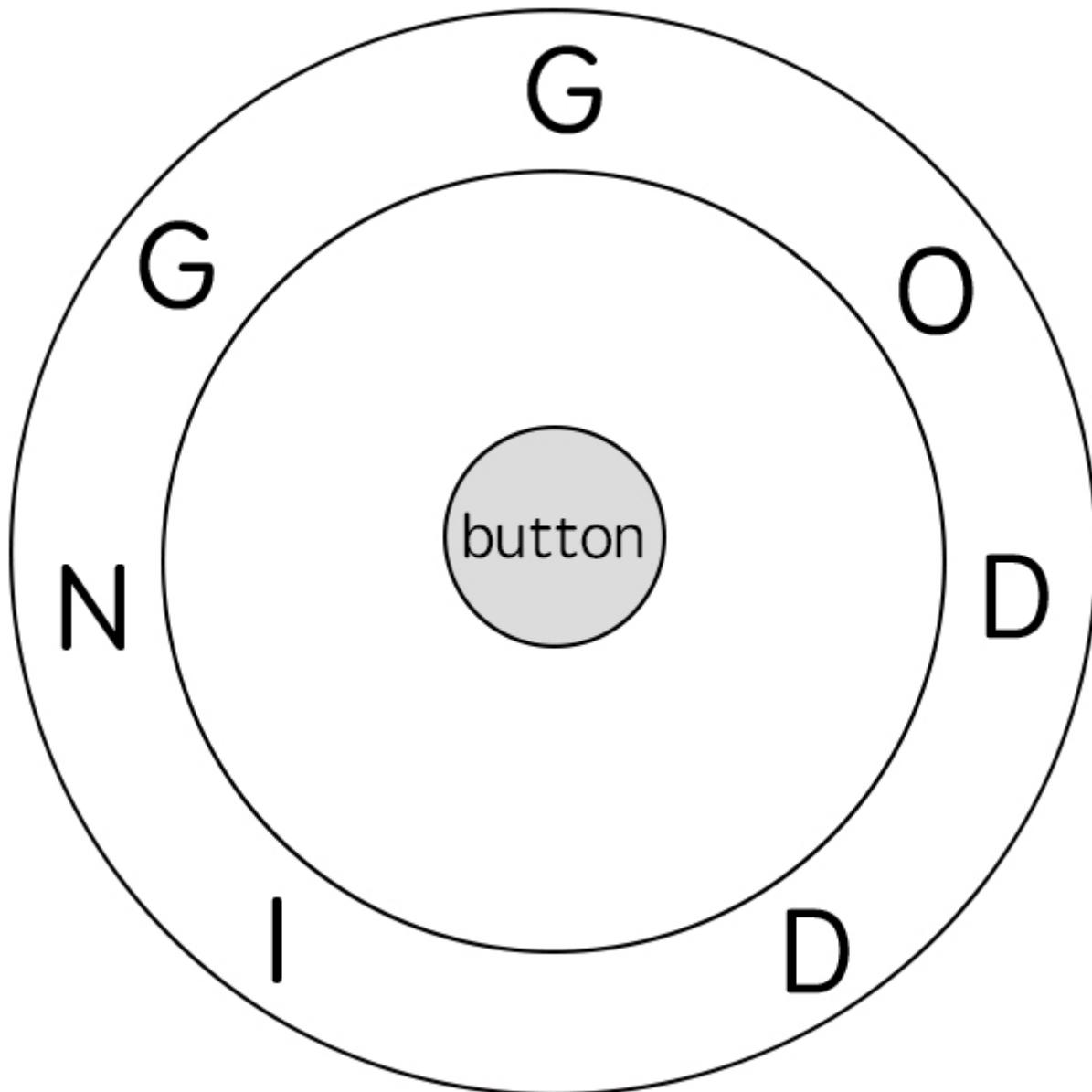
最初, **ring**的第一个字符与 12:00 方向对齐。你需要顺时针或逆时针旋转**ring**以使 **key**的一个字符在 12:00 方向对齐, 然后按下中心按钮, 以此逐个拼写完 **key**中的所有字符。

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：练琴时悟出的动态规划算法，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;
    }
};
```

# 518. 零钱兑换 II



- 标签: 动态规划, 背包问题, 二维动态规划

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];
    }
}
```

# 583. 两个字符串的删除操作



- 标签：子序列，动态规划，二维动态规划

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 [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：[详解最长公共子序列问题，秒杀三道动态规划题目](#)

## 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 {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[m][n];
    }
}
```

```
    } 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];
}
}
```

- 类似题目：

- [1143. 最长公共子序列](#) (中等)
- [712. 两个字符串的最小 ASCII 删除和](#) (中等)

# 712. 两个字符串的最小 ASCII 删除和



- 标签: 动态规划, 二维动态规划

给定两个字符串  $s_1$ ,  $s_2$ , 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 [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: [详解最长公共子序列问题, 秒杀三道动态规划题目](#)

## 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, s2, 0, 0);
    }

    private int dp(String s1, String s2, int i, int j) {
        if (i == m && j == n)
            return 0;
        if (i == m)
            return s2.substring(j).chars().sum();
        if (j == n)
            return s1.substring(i).chars().sum();
        if (memo[i][j] != -1)
            return memo[i][j];
        if (s1.charAt(i) == s2.charAt(j))
            memo[i][j] = dp(s1, s2, i + 1, j + 1);
        else
            memo[i][j] = Math.min(
                dp(s1, s2, i + 1, j) + s1.charAt(i),
                dp(s1, s2, i, j + 1) + s2.charAt(j));
        return memo[i][j];
    }
}
```

```
        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];
    }
}
```

- 类似题目：
  - [1143. 最长公共子序列（中等）](#)
  - [583. 两个字符串的删除操作（中等）](#)

# 1143. 最长公共子序列



- 标签：子序列，动态规划，二维动态规划

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。

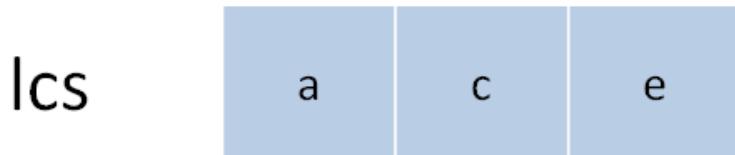
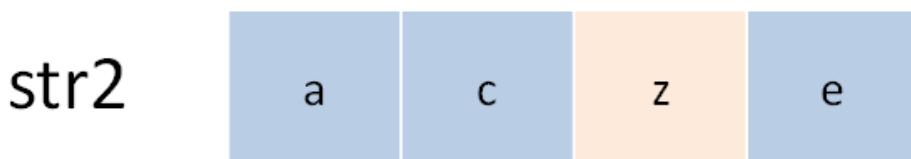
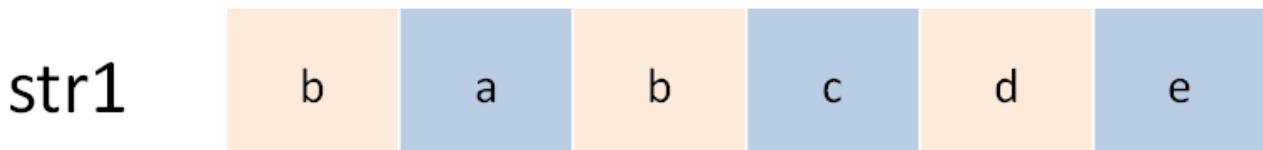
示例 1：

```
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。
```

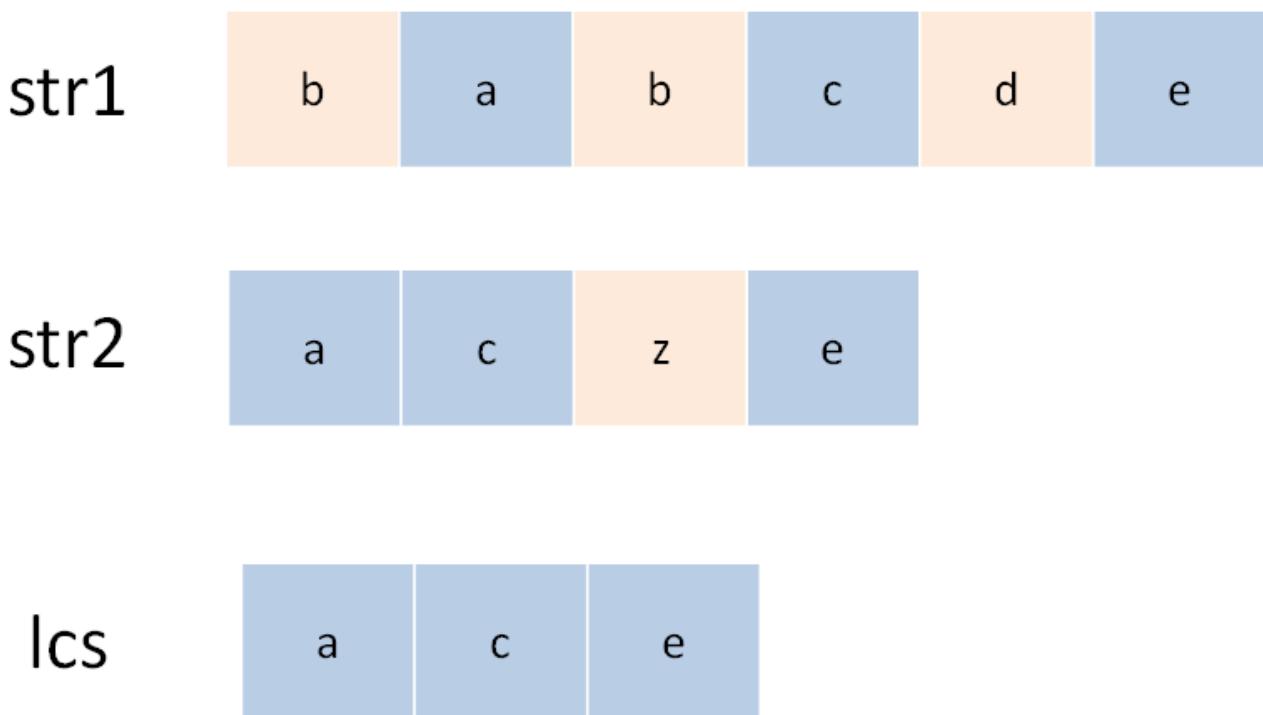
## 基本思路

PS：这道题在《算法小抄》的第 117 页。

和 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：[详解最长公共子序列问题，秒杀三道动态规划题目](#)

## 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];
    }
}

```

- 类似题目：

- [583. 两个字符串的删除操作（中等）](#)
- [712. 两个字符串的最小 ASCII 删除和（中等）](#)

# 787. K 站中转内最便宜的航班



- 标签: 动态规划, 图论算法, 最短路径算法, 二维动态规划

有  $n$  个城市通过一些航班连接。给你一个数组  $\text{flights}$ , 其中  $\text{flights}[i] = [\text{from}_i, \text{to}_i, \text{price}_i]$ , 表示该航班都从城市  $\text{from}_i$  开始, 以价格  $\text{price}_i$  抵达  $\text{to}_i$ 。

现在给定所有的城市和航班, 以及出发城市  $\text{src}$  和目的地  $\text{dst}$ , 你的任务是找到出一条最多经过  $k$  站中转的路线, 使得从  $\text{src}$  到  $\text{dst}$  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。

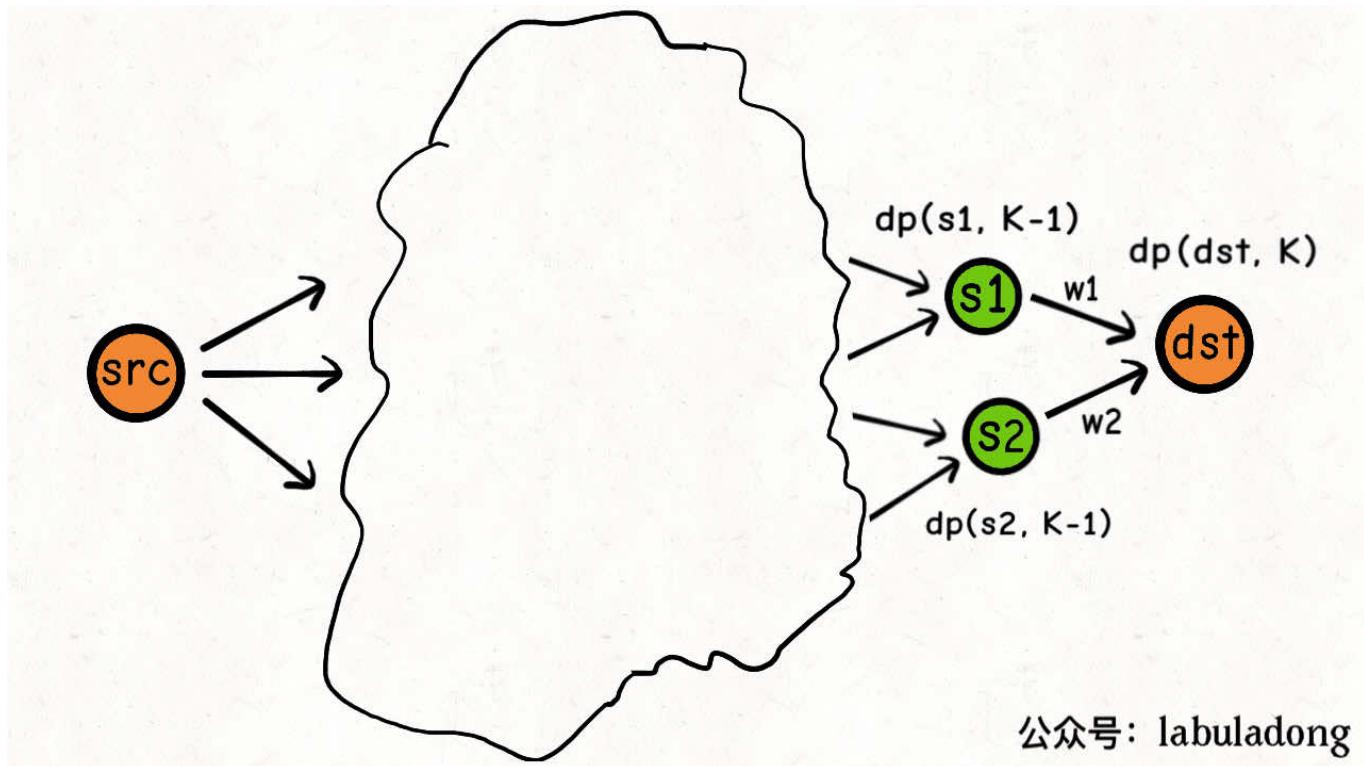
## 基本思路

$\text{dp}$  函数的定义: 从起点  $\text{src}$  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  $\text{dp}(s, k)$ 。

这样, 题目想求的最小机票开销就可以用  $\text{dp}(\text{dst}, K+1)$  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: 毕业旅行

## 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];
}
}
```

# 887. 鸡蛋掉落



- 标签: 动态规划, 二维动态规划

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  $0 \leq f \leq n$ , 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  $1 \leq x \leq n$ )。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, 经典动态规划: 高楼扔鸡蛋 (进阶篇)

## 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;
}
```

# 931. 下降路径最小和



- 标签: 动态规划, 二维动态规划

给你一个  $n \times n$  的整数数组  $\text{matrix}$ , 请你找出并返回  $\text{matrix}$  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  $(\text{row}, \text{col})$  的下一个元素应当是  $(\text{row} + 1, \text{col} - 1)$ 、 $(\text{row} + 1, \text{col})$  或者  $(\text{row} + 1, \text{col} + 1)$ 。

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  $\text{matrix}[i][j]$ , 只有可能从  $\text{matrix}[i-1][j]$ ,  $\text{matrix}[i-1][j-1]$ ,  $\text{matrix}[i-1][j+1]$  这三个位置转移过来。

$\text{dp}$  函数的定义: 从第一行 ( $\text{matrix}[0][..]$ ) 向下落, 落到位置  $\text{matrix}[i][j]$  的最小路径和为  $\text{dp}(\text{matrix}, i, j)$ , 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: 春节期间, 读者留言最多的问题

## 解法代码

```
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));
}
}
```

# 416. 分割等和子集



- 标签: 动态规划, 背包问题, 二维动态规划

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: 0-1 背包问题的变体

## 解法代码

```
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];
}
```

# 494. 目标和



- 标签: 背包问题, 回溯算法, 动态规划, 二维动态规划

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: [回溯算法和动态规划，到底谁是谁爹？](#)

## 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 rest) {
    // base case
    if (i == nums.length) {
        if (rest == 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 + "," + rest;
    // 避免重复计算
    if (memo.containsKey(key)) {
        return memo.get(key);
    }
    // 还是穷举
    int result = dp(nums, i + 1, rest - nums[i]) + dp(nums, i + 1,
    rest + nums[i]);
    // 记入备忘录
    memo.put(key, result);
    return result;
}
}
```

# 518. 零钱兑换 II



- 标签: 动态规划, 背包问题, 二维动态规划

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];
    }
}
```

# 17. 电话号码的字母组合



- 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 [回溯算法之子集、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);
        }
    }
}
}
```

# 77. 组合



- 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
```

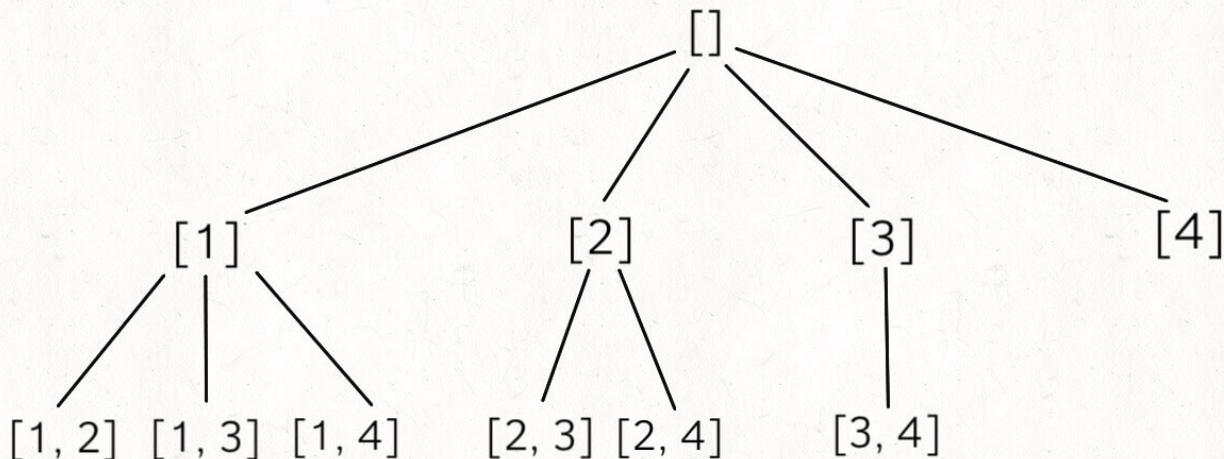
```
输出:
```

```
[  
    [2,4],  
    [3,4],  
    [2,3],  
    [1,2],  
    [1,3],  
    [1,4],  
]
```

## 基本思路

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：回溯算法团灭排列/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();
        }
    }
};
```

- 类似题目：

- 78. 子集（中等）
- 46. 全排列（中等）

## 78. 子集



- 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:

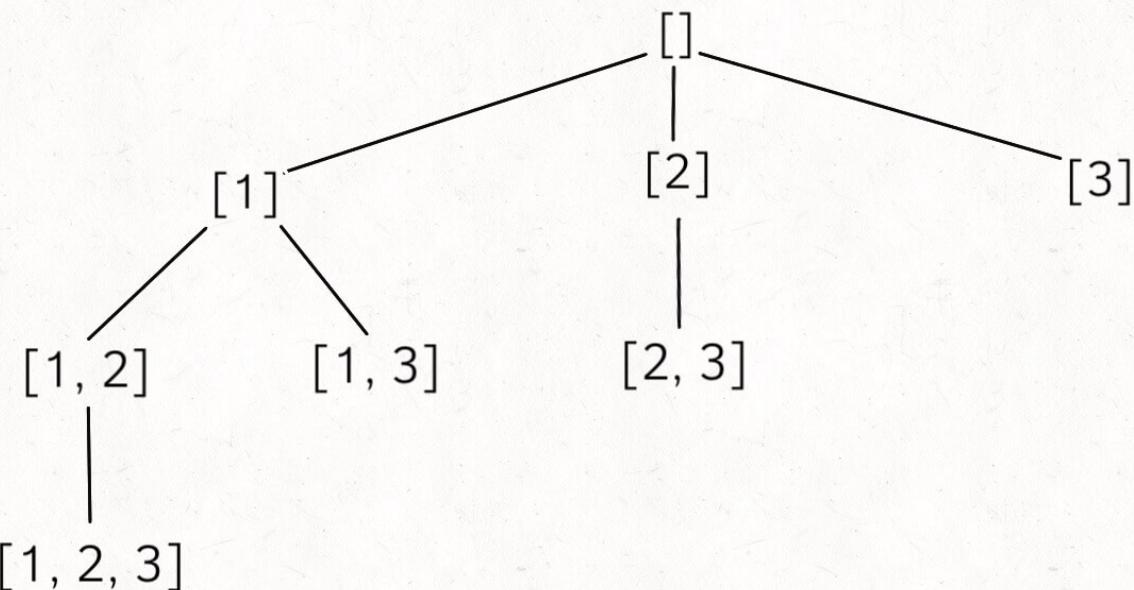
```
输入: nums = [1,2,3]
输出: [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
```

### 基本思路

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, 可以套旧文 [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:



公众号: labuladong

- 详细题解: [回溯算法团灭排列/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();
        }
    }
};
```

- 类似题目：

- 46. 全排列（中等）
- 77. 组合（中等）

# 134. 加油站



- 标签: 贪心算法, 数学

在一条环路上有  $N$  个加油站，其中第  $i$  个加油站有汽油  $\text{gas}[i]$  升。你有一辆油箱容量无限的汽车，从第  $i$  个加油站开往第  $i+1$  个加油站需要消耗汽油  $\text{cost}[i]$  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]
```

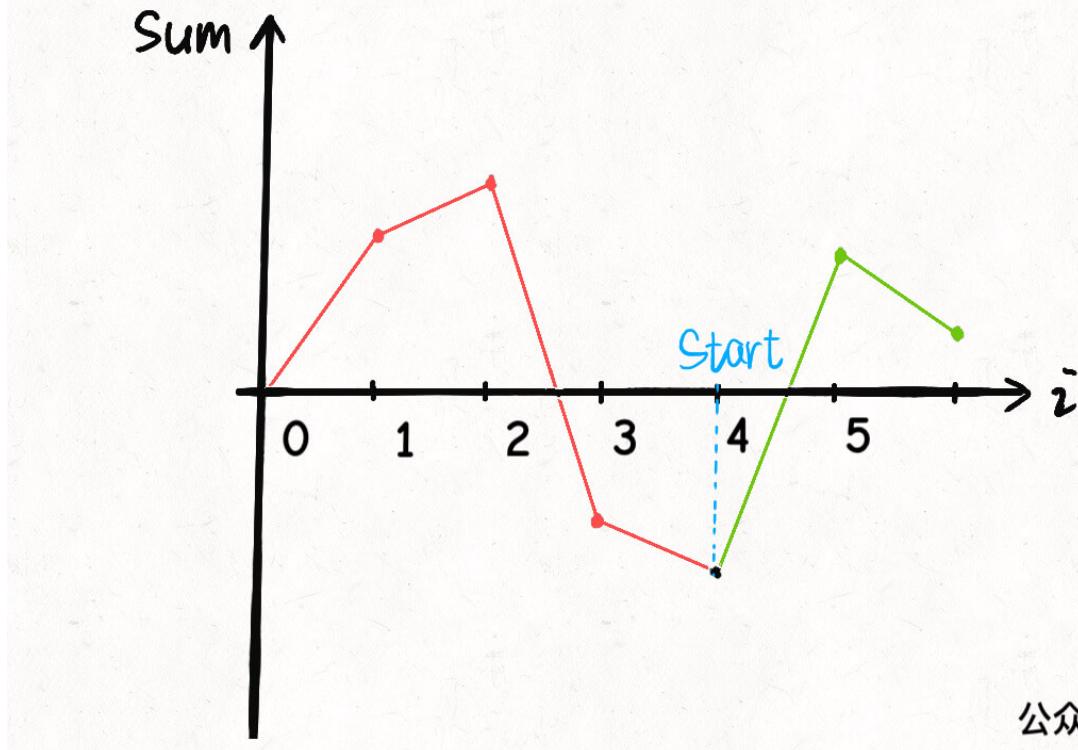
输出: 3

解释:

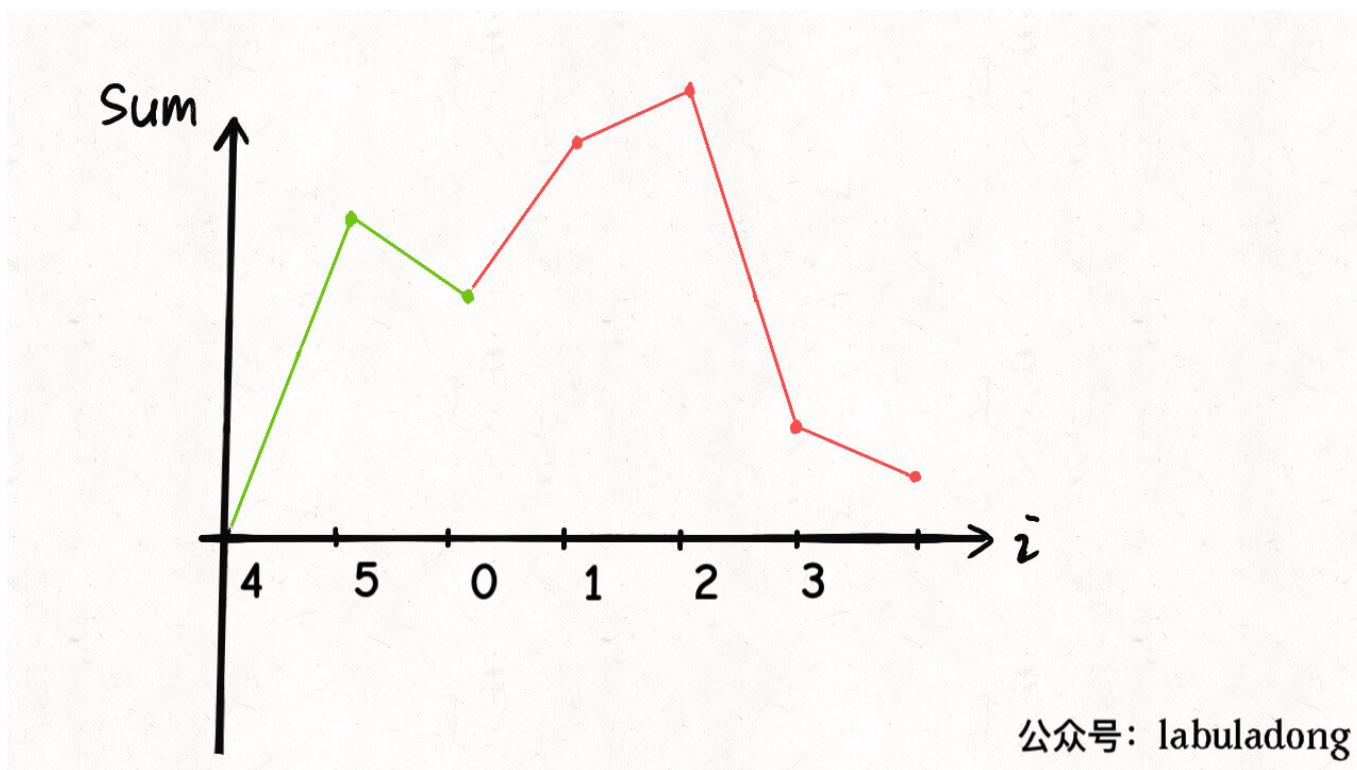
从 3 号加油站（索引为 3 处）出发，可获得 4 升汽油。此时油箱有  $= 0 + 4 = 4$  升汽油  
开往 4 号加油站，此时油箱有  $4 - 1 + 5 = 8$  升汽油  
开往 0 号加油站，此时油箱有  $8 - 2 + 1 = 7$  升汽油  
开往 1 号加油站，此时油箱有  $7 - 3 + 2 = 6$  升汽油  
开往 2 号加油站，此时油箱有  $6 - 4 + 3 = 5$  升汽油  
开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。  
因此，3 可为起始索引。

## 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，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：当老司机学会了贪心算法

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;
    }
}
```

# 136. 只出现一次的数字



- 标签: 数学, 位运算

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  $a \wedge a = 0$ ；一个数和 0 做异或运算的结果为它本身，即  $a \wedge 0 = a$ 。

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：东哥教你几招常用的位运算技巧

## 解法代码

```
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for (int n : nums) {
            res ^= n;
        }
        return res;
    }
}
```

- 类似题目：
  - 191. 位 1 的个数（简单）
  - 2312 的幂（简单）

# 191. 位 1 的个数

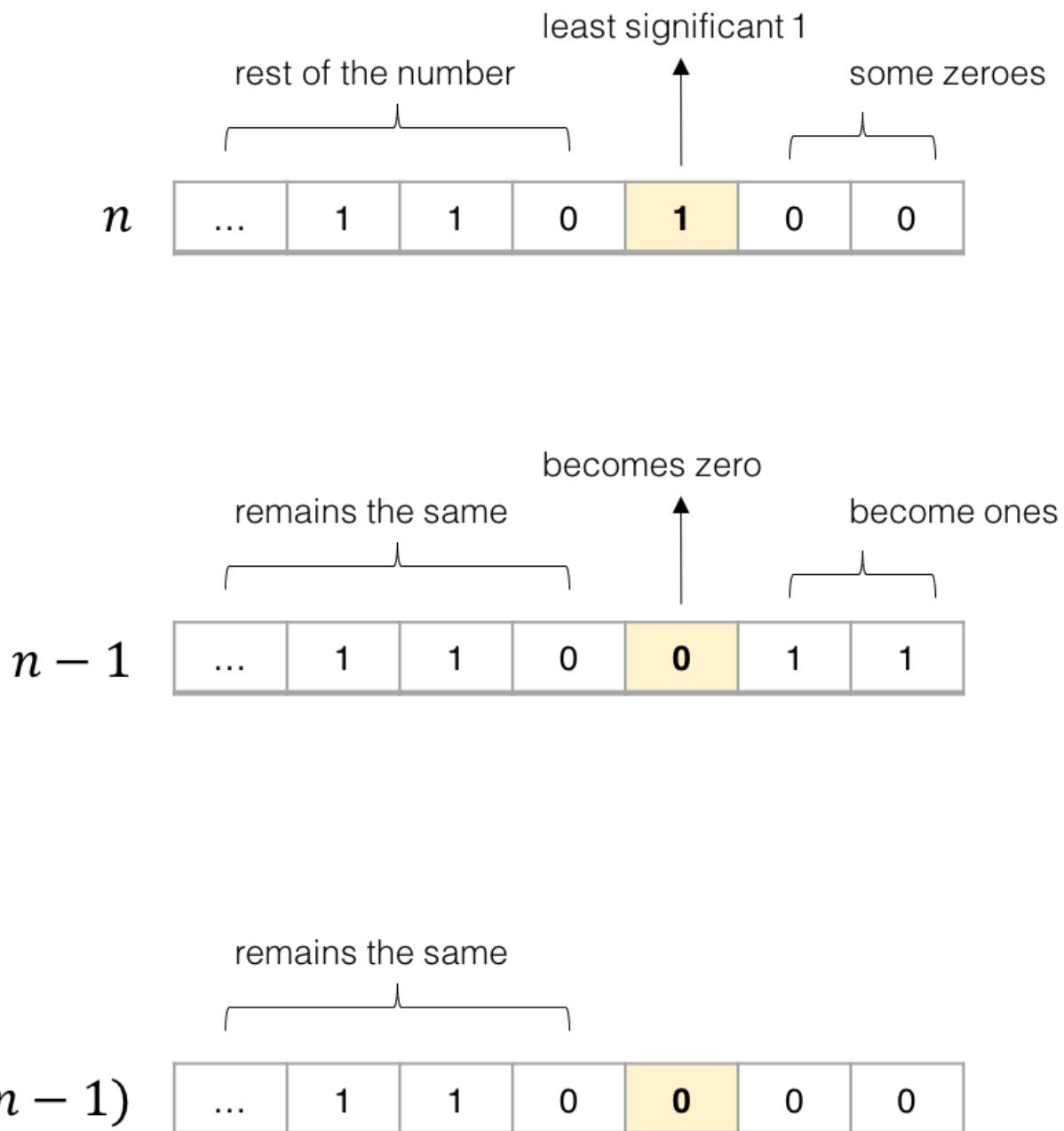


- 标签: 数学, 位运算

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：东哥教你几招常用的位运算技巧

## 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;  
    }  
}
```

- 类似题目：
  - 231.2 的幂（简单）

# 231. 2 的幂



- 标签: 数学, 位运算

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: 东哥教你几招常用的位运算技巧

## 解法代码

```
class Solution {
    public boolean isPowerOfTwo(int n) {
        if (n <= 0) return false;
        return (n & (n - 1)) == 0;
    }
}
```

- 类似题目:
  - 191. 位 1 的个数 (简单)

# 172. 阶乘后的零



- 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:  $n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$

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  $125 = 5 \times 5 \times 5$ , 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: [阶乘相关的算法题, 东哥又整活儿了](#)

## 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;  
}  
}
```

- 类似题目：
  - [793. 阶乘后 K 个零](#) (困难)

# 793. 阶乘函数后 K 个零



- 标签: 数学, 二分搜索

$f(x)$  是  $x!$  末尾是 0 的数量,  $x! = 1 * 2 * 3 * \dots * x$ , 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 阶乘后的零 这道题的解法函数 `trailingZeroes`。

搜索有多少个  $n$  满足  $\text{trailingZeroes}(n) == K$ , 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  $[0, \text{LONG\_MAX}]$  中取值, 寻找满足  $\text{trailingZeroes}(n) == K$  的左侧边界和右侧边界, 相减即是答案。

- 详细题解: 阶乘相关的算法题, 东哥又整活儿了

## 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](#) (简单)

## 204. 计数质数



- 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: 如何用算法高效寻找素数?

### 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;
    }
}
```

# 268. 丢失的数字



- 标签: 数学, 位运算

给定一个包含  $[0, n]$  中  $n$  个数的数组  $\text{nums}$ , 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:  $\text{nums} = [3, 0, 1]$

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  $\text{nums}$  中。

## 基本思路

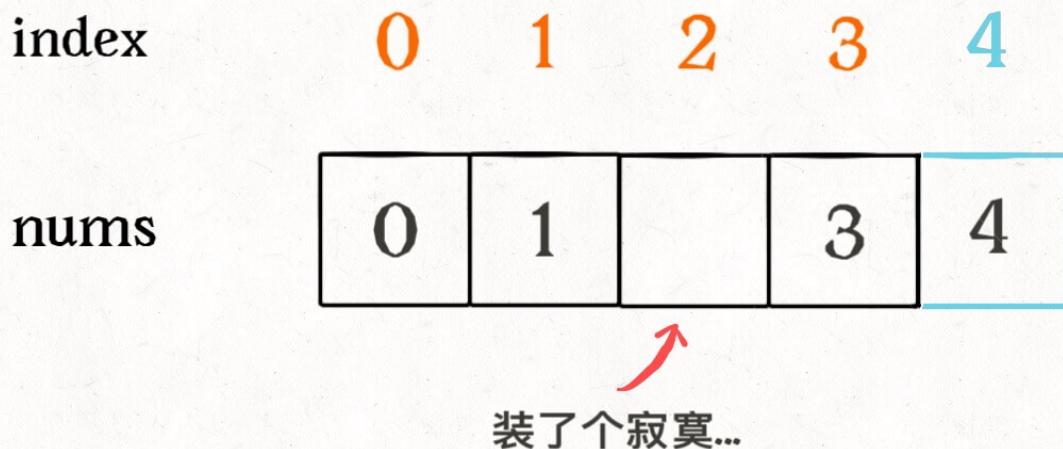
假设  $\text{nums} = [0, 3, 1, 4]$ :

index            0        1        2        3

nums            0        3        1        4

公众号: labuladong

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:



公众号: labuladong

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：这个问题不简单：寻找缺失元素

## 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;
    }
}
```

# 292. Nim 游戏



- 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智力题

## 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;
```

```
    }  
}
```

- 类似题目：

- [877. 石子游戏 \(中等\)](#)
- [319. 灯泡开关 \(中等\)](#)

# 319. 灯泡开关



- 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。

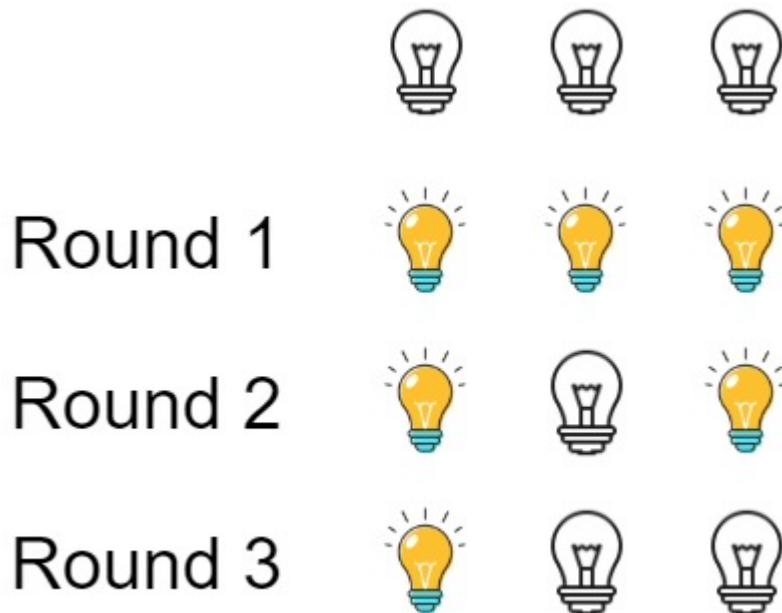
第 2 轮, 每两个灯泡切换一次开关。即每两个灯泡关闭一个。

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  $16 = 1 \times 16 = 2 \times 8 = 4 \times 4$ ，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智力题](#)

## 解法代码

```
class Solution {
    public int bulbSwitch(int n) {
        return (int)Math.sqrt(n);
    }
}
```

- 类似题目：

- [292. Nim 游戏（简单）](#)
- [877. 石子游戏（中等）](#)

# 877. 石子游戏



- 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智力题](#)

## 解法代码

```
class Solution {
    public boolean stoneGame(int[] piles) {
        return true;
}
```

- 类似题目：

- 292.Nim 游戏（简单）
- 319. 灯泡开关（中等）

# 295. 数据流的中位数



- 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。

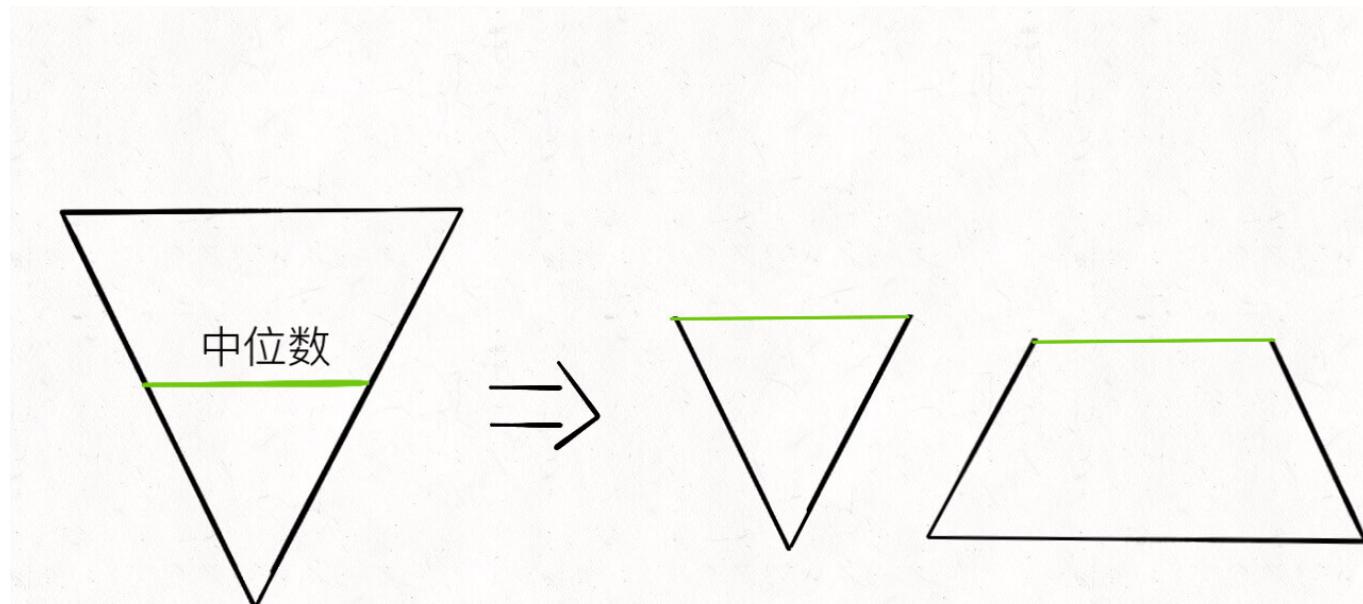
2、`double findMedian()` 返回目前所有元素的中位数。

示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

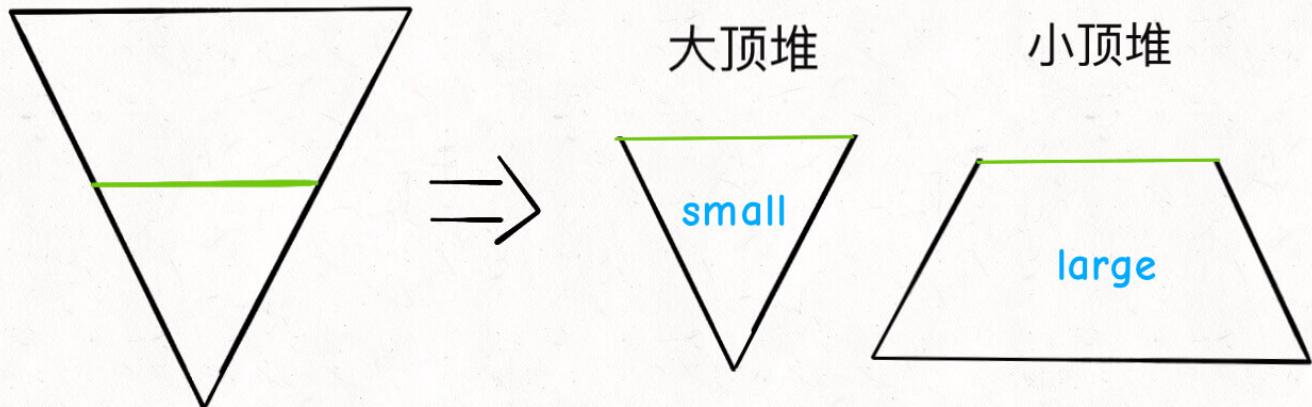
## 基本思路

本题的核心思路是使用两个优先级队列。



公众号: labuladong

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：



公众号: labuladong

- 详细题解: 啊这, 一道找中位数的算法题把东哥整不会了...

## 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);
        } else {
            large.offer(num);
        }
    }
}
```

```
        large.offer(small.poll());
    } else {
        large.offer(num);
        small.offer(large.poll());
    }
}
```

## 372. 超级次方



- 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:

$$\begin{aligned} & a^{[1, 5, 6, 4]} \\ &= a^4 \times a^{[1, 5, 6, 0]} \\ &= a^4 \times (a^{[1, 5, 6]})^{10} \end{aligned}$$

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: [Super Pow: 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;
    }
};
```

## 382. 链表随机节点



- 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：随机算法之水塘抽样算法

### 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：

- [398. 随机数索引](#) (中等)

# 398. 随机数索引



- 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。

所以这题想考察水塘抽样算法，每次 `pick` 都遍历一遍 `nums` 数组，然后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：随机算法之水塘抽样算法**

## 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](#) (中等)

# 391. 完美矩形



- 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」。

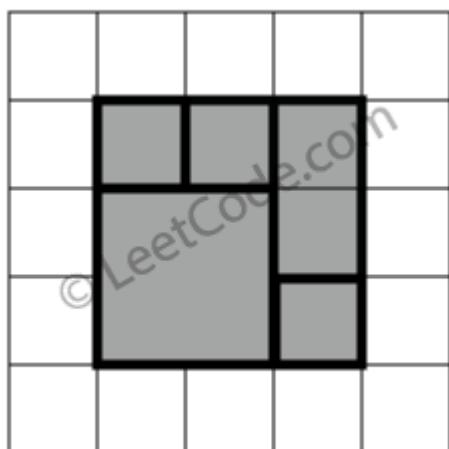
每个矩形用左下角的点和右上角的点的坐标来表示。例如，一个单位正方形可以表示为  $[1, 1, 2, 2]$ ，左下角的点的坐标为  $(1, 1)$  以及右上角的点的坐标为  $(2, 2)$ 。

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]
]
```

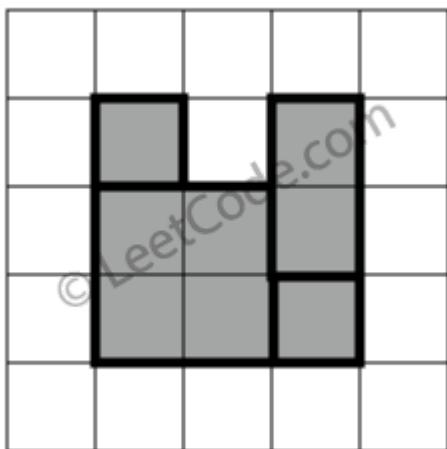
返回 `true`。5 个矩形一起可以精确地覆盖一个矩形区域。



示例 2:

```
rectangles = [
    [1,1,3,3],
    [3,1,4,2],
    [1,3,2,4],
    [3,2,4,4]
]
```

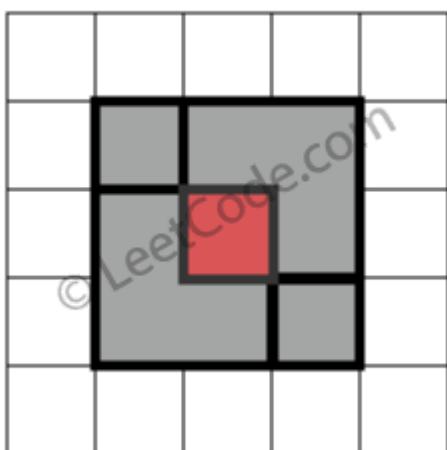
返回 `false`。图形顶端留有间隔，无法覆盖成一个矩形。



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。

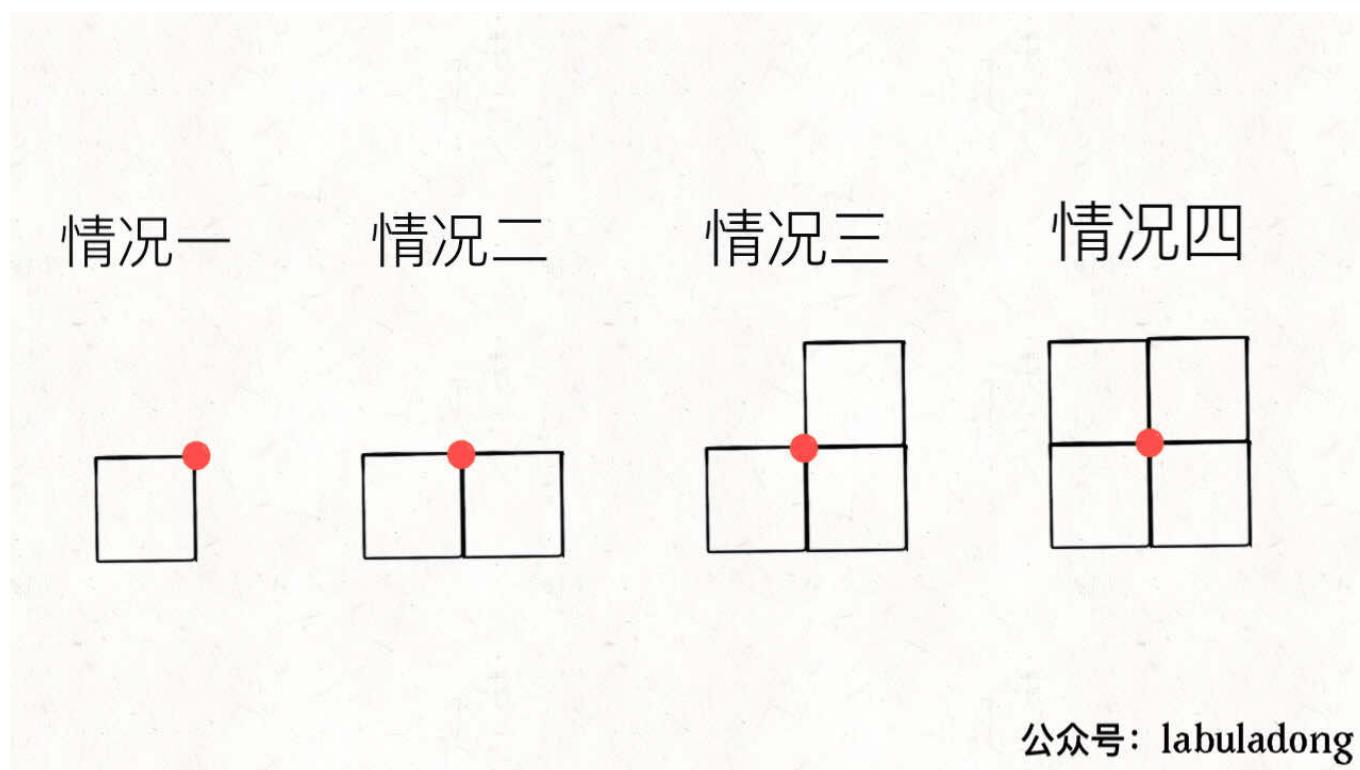
如果所有小矩形的面积之和不等于这个完美矩形的理论面积，那么说明最终形成的图形肯定存在空缺或者重叠，肯定不是完美矩形。

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：



公众号：labuladong

按照「面积」和「顶点」两个角度来写代码即可。

- 详细题解：这道「完美矩形」给我整不会了...

## 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')  
  
        actual_area = 0  
        # 哈希集合，记录最终图形的顶点  
        points = set()  
        for x1, y1, x2, y2 in rectangles:  
            X1, Y1 = min(X1, x1), min(Y1, y1)
```

```
X2, Y2 = max(X2, x2), max(Y2, y2)

actual_area += (x2 - x1) * (y2 - y1)
# 先算出小矩形每个点的坐标
p1, p2 = (x1, y1), (x1, y2)
p3, p4 = (x2, y1), (x2, y2)
# 对于每个点，如果存在集合中，删除它；
# 如果不存在集合中，添加它；
# 在集合中剩下的点都是出现奇数次的点
for p in [p1, p2, p3, p4]:
    if p in points: points.remove(p)
    else: points.add(p)

expected_area = (X2 - X1) * (Y2 - Y1)
if actual_area != expected_area:
    return False

return True
```

# 509. 斐波那契数



- 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：

$$\begin{aligned}F(0) &= 0, F(1) = 1 \\F(n) &= F(n - 1) + F(n - 2), \text{ 其中 } n > 1\end{aligned}$$

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

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 < 1) return 0;
        if (n == 2 || n == 1)
            return 1;
        int prev = 1, curr = 1;
        for (int i = 3; i <= n; i++) {
            int sum = prev + curr;
            prev = curr;
            curr = sum;
        }
        return curr;
    }
}
```

- 类似题目：
  - [322.零钱兑换（中等）](#)

# 645. 错误的集合



- 标签: 数组, 数学

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  $\text{nums}$  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:

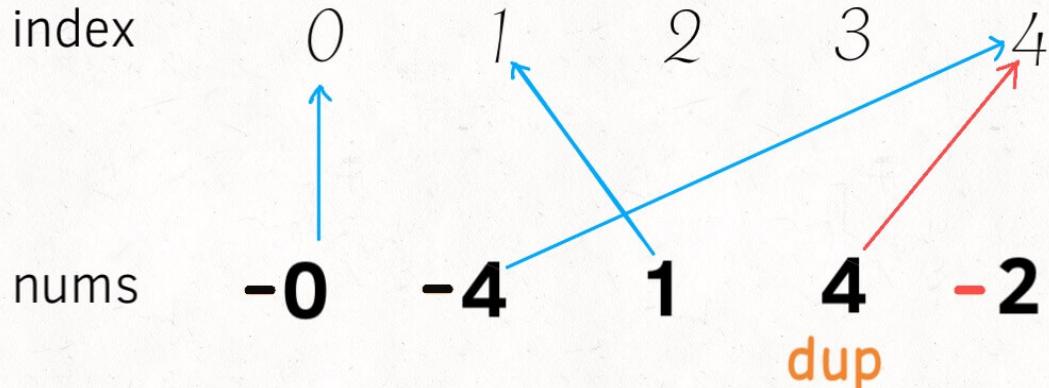
输入:  $\text{nums} = [1, 2, 2, 4]$

输出:  $[2, 3]$

## 基本思路

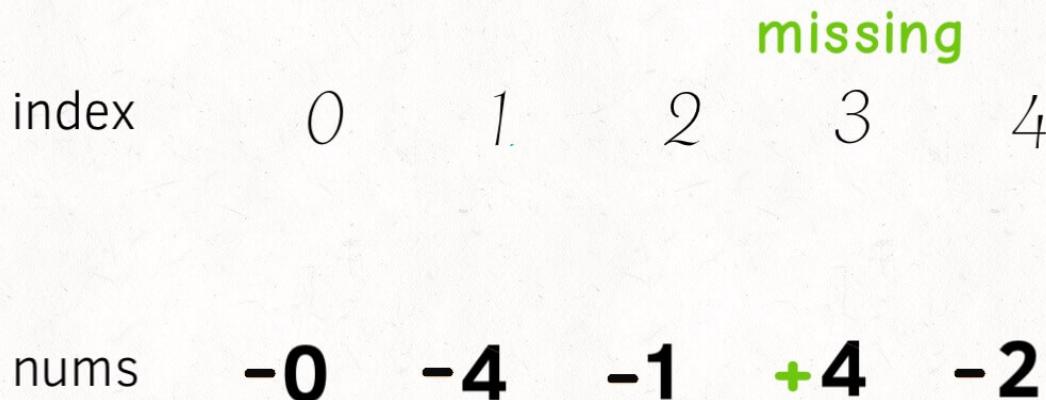
这题的核心思路是将每个索引对应的元素变成负数, 以表示这个索引被对应过一次。

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:



公众号: labuladong

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:



公众号: labuladong

依据这个特点，就能找到缺失和重复的元素了。

- 详细题解: [高效寻找缺失和重复的数字](#)

## 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};
    }
}
```

# 710. 黑名单中的随机数



- 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: 给我常数时间, 我可以删除/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.常数时间插入、删除和获取随机元素（中等）

## 56. 合并区间



- 标签: 区间问题

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:

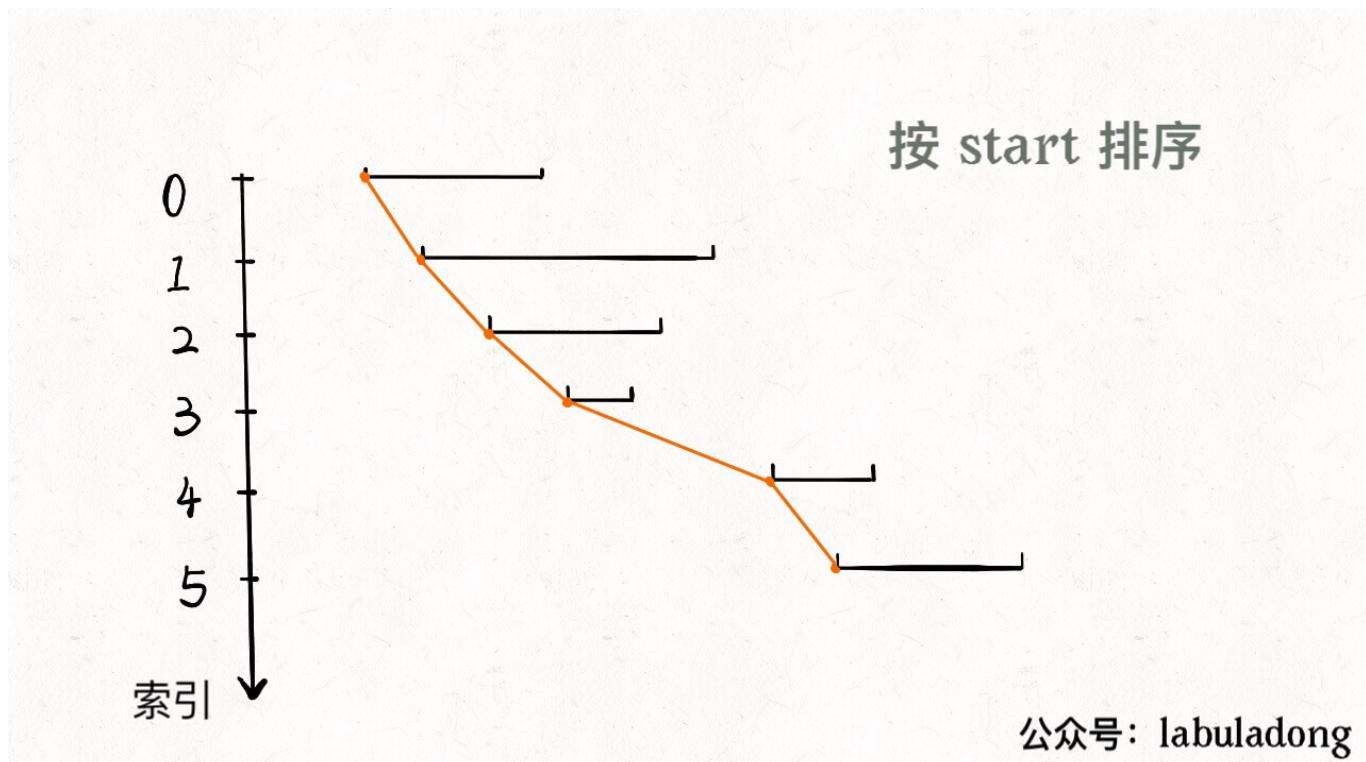
```
输入: intervals = [[1,3],[2,6],[8,10],[15,18]]  
输出: [[1,6],[8,10],[15,18]]  
解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

示例 2:

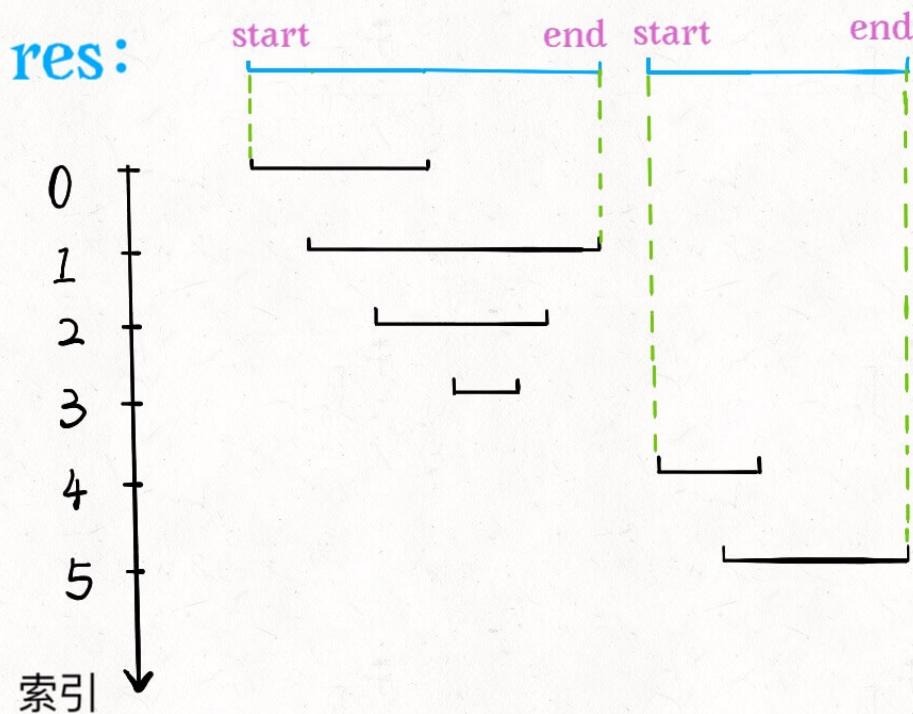
```
输入: intervals = [[1,4],[4,5]]  
输出: [[1,5]]  
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

### 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：



公众号： labuladong

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：一文秒杀所有区间相关问题

## 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]);
    }
}
```

```
    }  
}
```

- 类似题目：
  - [1288. 删除被覆盖区间（中等）](#)
  - [986. 区间列表的交集（中等）](#)

# 986. 区间列表的交集



- 标签: 区间问题, 数组双指针

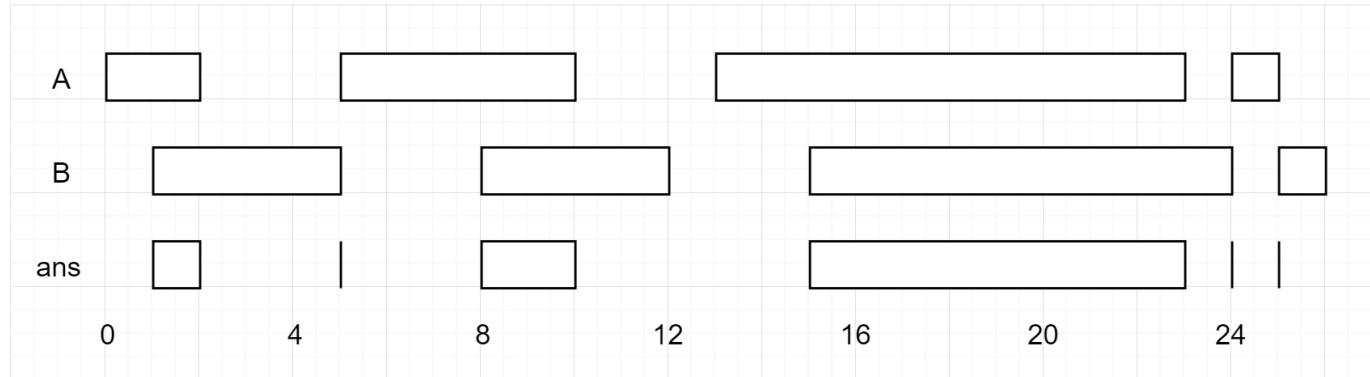
给定两个由一些闭区间组成的列表, `firstList` 和 `secondList`, 其中 `firstList[i] = [starti, endi]` 而 `secondList[j] = [startj, endj]`。每个区间列表都是成对不相交的, 并且已经排序。

返回这两个区间列表的交集。

形式上, 闭区间  $[a, b]$  (其中  $a \leq b$ ) 表示实数  $x$  的集合, 而  $a \leq x \leq b$ 。

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  $[2, 3]$ 。

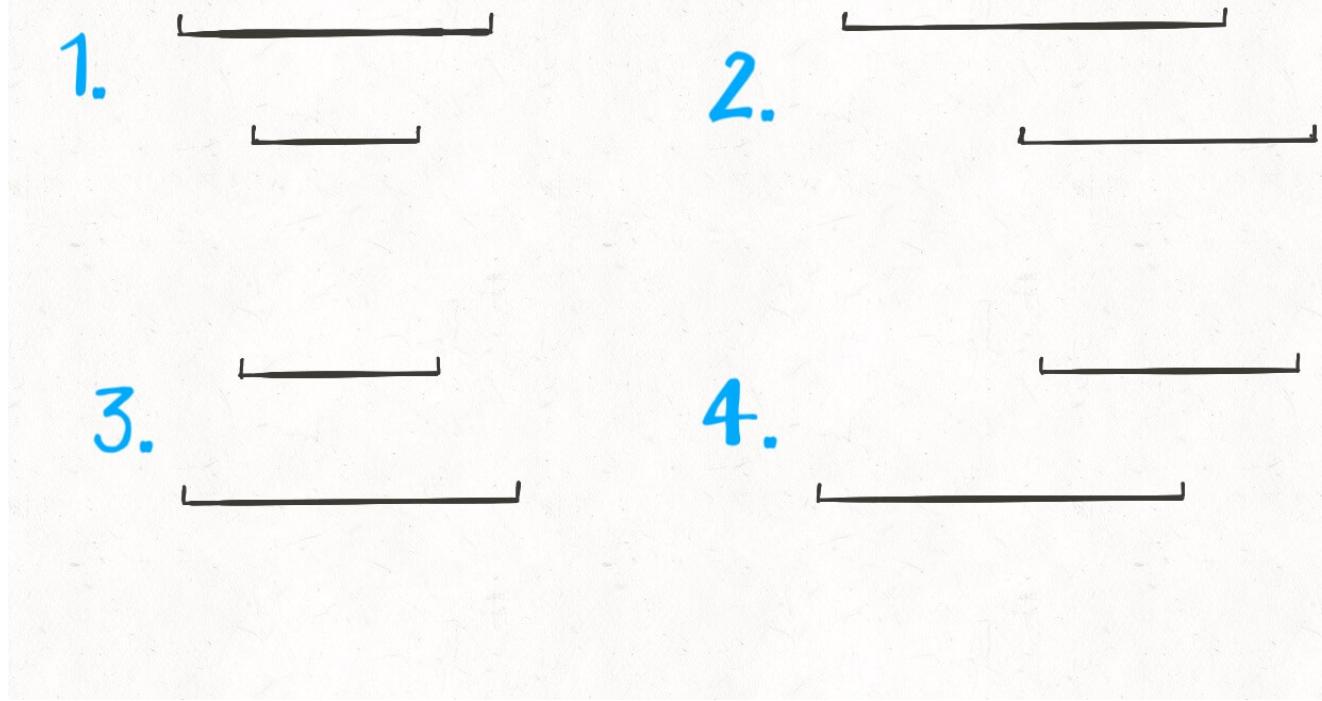
示例 1:



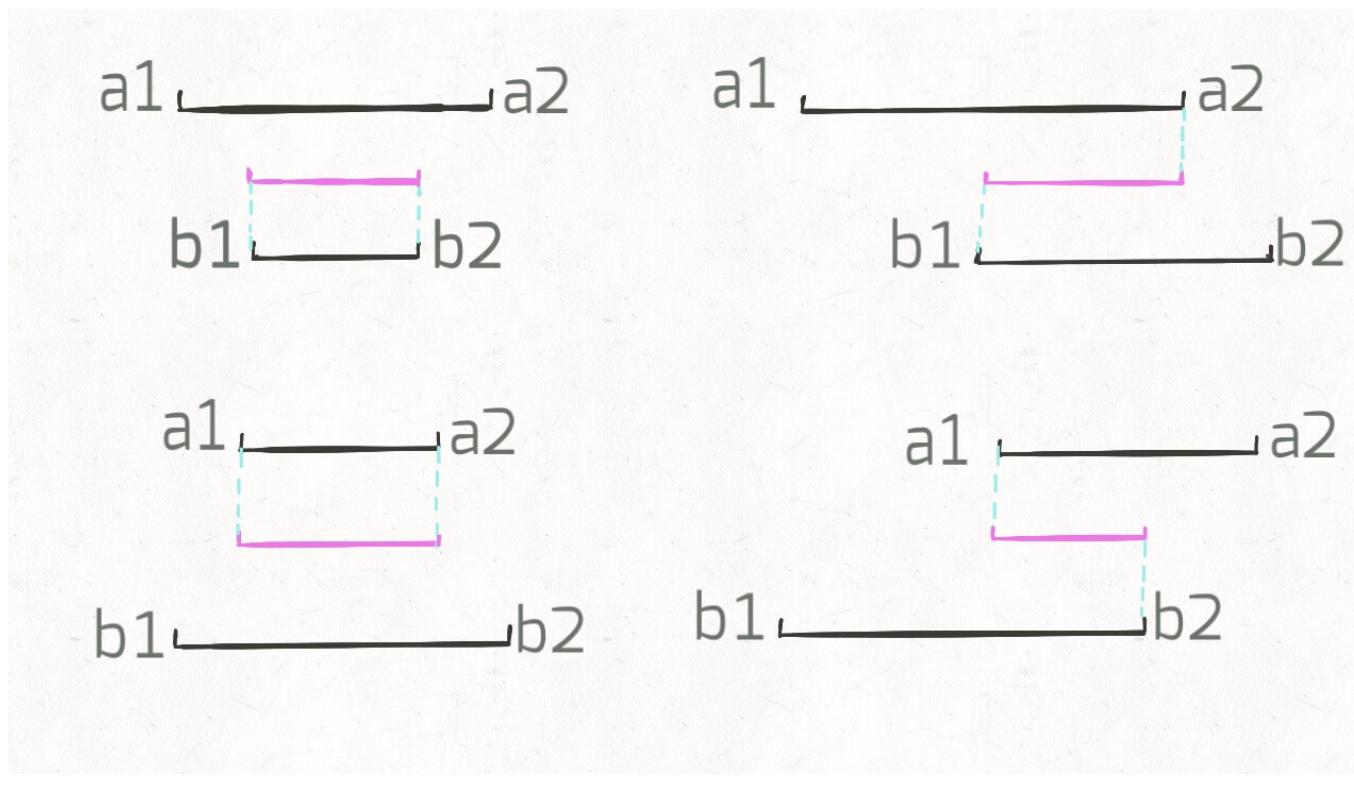
```
输入: firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]
输出: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

## 基本思路

我们用  $[a_1, a_2]$  和  $[b_1, b_2]$  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  $b_2 \geq a_1$   $\&$   $a_2 \geq b_1$ , 分下面四种情况:



根据上图可以发现规律，假设交集区间是  $[c_1, c_2]$ ，那么  $c_1 = \max(a_1, b_1)$ ,  $c_2 = \min(a_2, b_2)$ :



这一点就是寻找交集的核心。

- 详细题解：一文秒杀所有区间相关问题

## 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]);
}
```

- 类似题目：

- 1288. 删除被覆盖区间（中等）
- 56. 区间合并（中等）

# 1288. 删除被覆盖区间



- 标签: 区间问题

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  $c \leq a$  且  $b \leq d$  时，我们认为区间  $[a, b]$  被区间  $[c, d]$  覆盖。

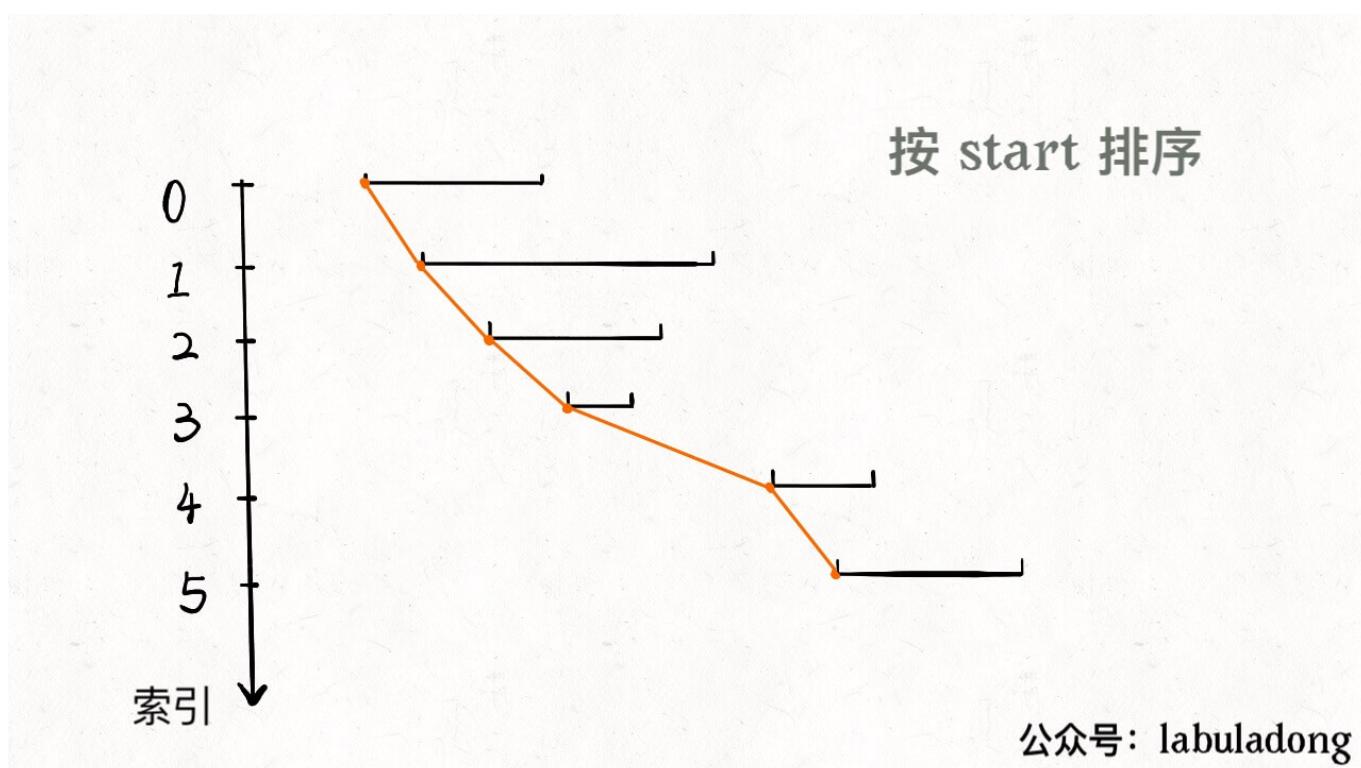
在完成所有删除操作后，请你返回列表中剩余区间的数目。

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。
```

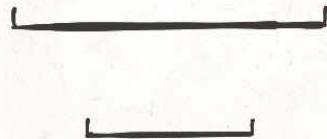
## 基本思路

按照区间的起点进行升序排序：

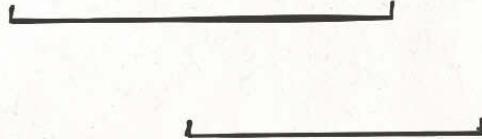


排序之后，两个相邻区间可能有如下三种情况：

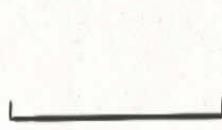
1.



2.



3.



公众号: labuladong

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：一文秒杀所有区间相关问题

## 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]) {  
                continue;  
            } else if (right < intv[0]) {  
                right = intv[1];  
            } else if (left > intv[1]) {  
                left = intv[0];  
            } else {  
                right = Math.max(right, intv[1]);  
            }  
            res++;  
        }  
        return res;  
    }  
}
```

```
        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;
}
}
```

- 类似题目：

- [56. 区间合并（中等）](#)
- [986. 区间列表的交集（中等）](#)

# 435. 无重叠区间



- 标签: 区间问题

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。

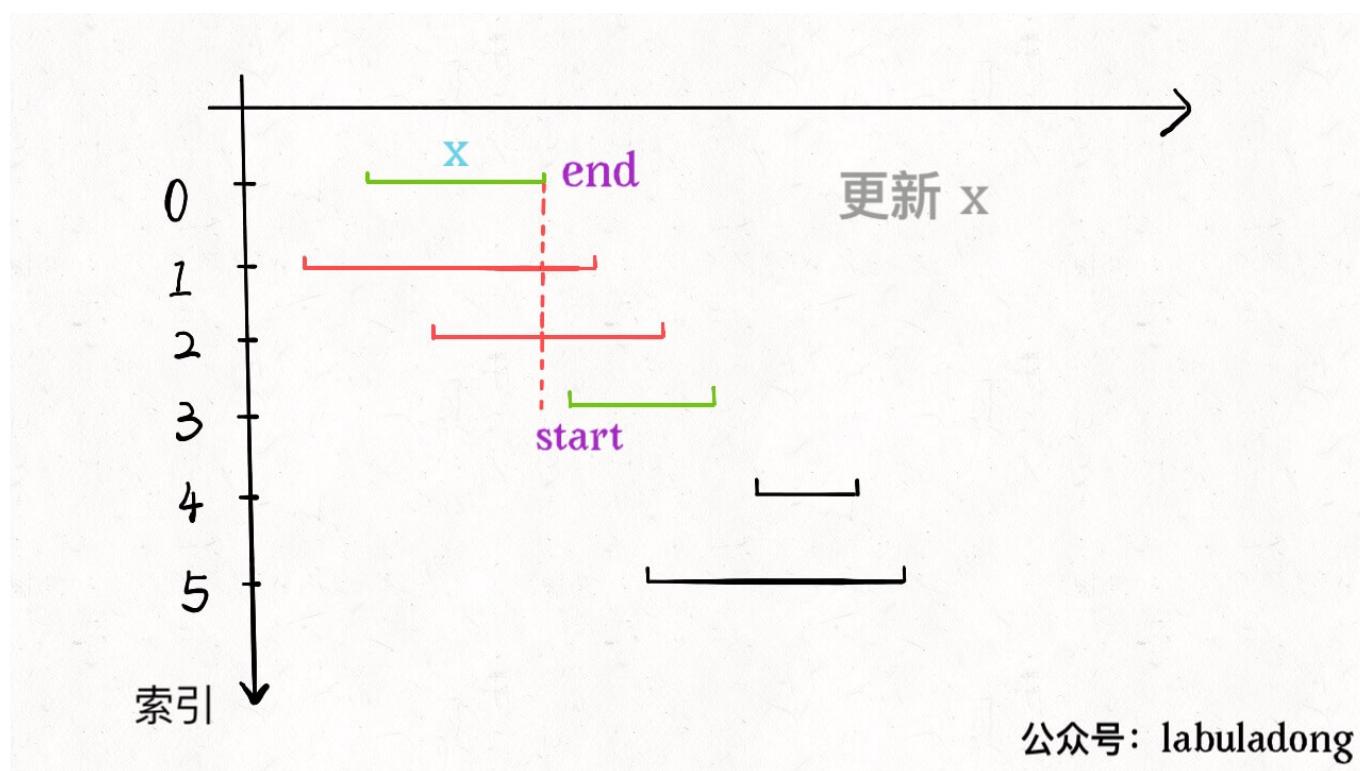
## 基本思路

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：运用贪心算法来做时间管理

## 解法代码

```
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;
}
}
```

- 类似题目：
  - [452. 用最少数量的箭引爆气球](#) (中等)

# 452. 用最少数量的箭引爆气球



- 标签: 区间问题

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  $start \leq x \leq end$  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:  $points = [[10,16], [2,8], [1,6], [7,12]]$

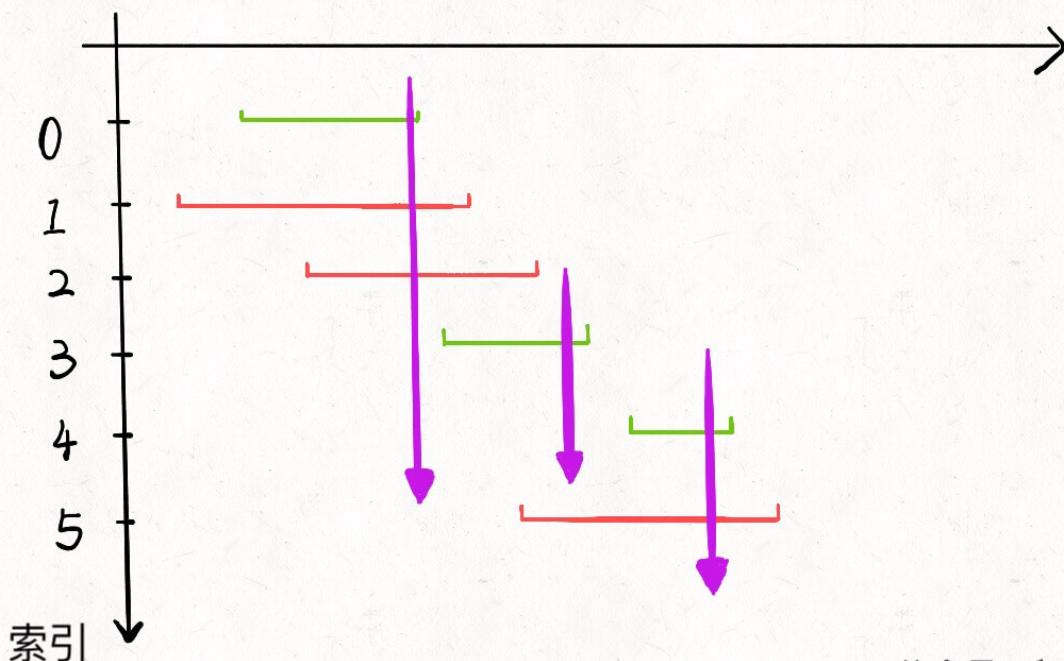
输出: 2

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: [运用贪心算法来做时间管理](#)

## 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;  
    }  
}
```

- 类似题目:
  - [435. 无重叠区间 \(中等\)](#)

# 1024. 视频拼接



- 标签：贪心算法，区间问题

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  $\text{clips}[i]$  都用区间进行表示：开始于  $\text{clips}[i][0]$  并于  $\text{clips}[i][1]$  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:  $\text{clips} = [[0, 2], [4, 6], [8, 10], [1, 9], [1, 5], [5, 9]]$ ,  $T = 10$

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：

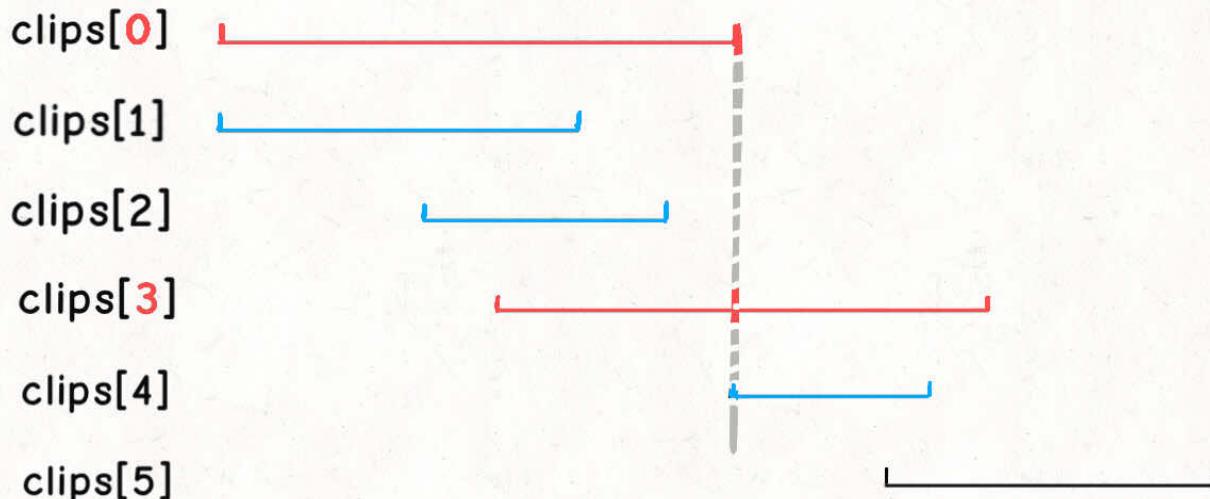
将  $[1, 9]$  再剪辑为  $[1, 2] + [2, 8] + [8, 9]$ 。

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。



公众号: labuladong

排序之后，从第一个区间开始选，每当选中一个区间  $\times$ （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;
}
```