

## 二分查找

- 153. 寻找旋转排序数组中的最小值
- 34. 在排序数组中查找元素的第一个和最后一个位置
- 4. 寻找两个正序数组的中位数
- 33. 搜索旋转排序数组
- 35. 搜索插入位置
- 74. 搜索二维矩阵

## 二叉树

- 101. 对称二叉树
- 104. 二叉树的最大深度
- 108. 将有序数组转换为二叉搜索树
- 124. 二叉树中的最大路径和
- 226. 翻转二叉树
- 236. 二叉树的最近公共祖先
- 437. 路径总和III
- 543. 二叉树的直径
- 94. 二叉树的中序遍历
- 102. 二叉树的层序遍历
- 105. 从前序与中序遍历序列构造二叉树
- 114. 二叉树展开为链表
- 199. 二叉树的右视图
- 230. 二叉搜索树中第K小的元素
- 297. 二叉树的序列化与反序列化
- 538. 把二叉搜索树转换为累加树
- 617. 合并二叉树
- 98. 验证二叉搜索树

## 动态规划

- 118. 杨辉三角
- 152. 乘积最大子数组
- 300. 最长递增子序列
- 322. 零钱兑换
- 53. 最大子数组和
- 139. 单词拆分
- 198. 打家劫舍
- 32. 最长有效括号
- 416. 分割等和子集
- 70. 爬楼梯

## 双指针

- 11. 盛最多水的容器
- 283. 移动零
- 15. 三数之和
- 42. 接雨水

## 哈希

- 1. 两数之和
- 49. 字母异位词分组
- 128. 最长连续序列

## 回溯

- 131. 分割回文串
- 22. 括号生成
- 46. 全排列
- 78. 子集
- 17. 电话号码的字母组合
- 39. 组合总和
- 51. N 皇后
- 79. 单词搜索

## 图论

- 200. 岛屿数量
- 208. 实现 Trie (前缀树)
- 207. 课程表
- 994. 腐烂的橘子

## 堆

- 215. 数组中的第K个最大元素
- 347. 前 K 个高频元素
- 295. 数据流的中位数

## 多维动态规划

- 1143. 最长公共子序列
- 62. 不同路径
- 72. 编辑距离
- 5. 最长回文子串
- 64. 最小路径和

## 子串

- 239. 滑动窗口最大值
- 76. 最小覆盖子串
- 560. 和为 K 的子数组

## 技巧

- 136. 只出现一次的数字
- 287. 寻找重复数
- 169. 多数元素
- 31. 下一个排列
- 75. 颜色分类

## 普通数组

- 189. 轮转数组
- 41. 缺失的第一个正数
- 238. 除自身以外数组的乘积
- 53. 最大子数组和
- 56. 合并区间

## 栈

- 155. 最小栈
- 394. 字符串解码
- 84. 柱状图中最大的矩形
- 20. 有效的括号
- 739. 每日温度

## 滑动窗口

- 3. 无重复字符的最长子串
- 438. 找到字符串中所有字母异位词

## 矩阵

- 240. 搜索二维矩阵 II
- 54. 螺旋矩阵
- 48. 旋转图像
- 73. 矩阵置零

## 贪心算法

- 121. 买卖股票的最佳时机
- 55. 跳跃游戏
- 45. 跳跃游戏 II
- 763. 划分字母区间

## 链表

- 138. 随机链表的复制
- 142. 环形链表 II
- 148. 排序链表
- 19. 删除链表的倒数第 N 个结点
- 206. 反转链表
- 23. 合并 K 个升序链表
- 24. 两两交换链表中的节点
- 141. 环形链表
- 146. LRU 缓存
- 160. 相交链表
- 2. 两数相加
- 21. 合并两个有序链表
- 234. 回文链表
- 25. K 个一组翻转链表

## 二分查找 / 153. 寻找旋转排序数组中的最小值

### 153. 寻找旋转排序数组中的最小值

#### 题目描述

已知一个长度为 n 的数组，预先按照升序排列，经由 1 到 n 次 旋转 后，得到输入数组。例如，原数组 nums = [0,1,2,4,5,6,7] 在变化后可能得到：

- 若旋转 4 次，则可以得到 [4,5,6,7,0,1,2]
- 若旋转 7 次，则可以得到 [0,1,2,4,5,6,7]

注意，数组 [a[0], a[1], a[2], ..., a[n-1]] 旋转一次 的结果为数组 [a[n-1], a[0], a[1], a[2], ..., a[n-2]]。

给你一个元素值 互不相同 的数组 nums，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 最小元素。

你必须设计一个时间复杂度为 O(log n) 的算法解决此问题。

#### 示例 1：

输入: nums = [3,4,5,1,2]  
输出: 1  
解释: 原数组为 [1,2,3,4,5] , 旋转 3 次得到输入数组。

#### 示例 2:

输入: nums = [4,5,6,7,0,1,2]  
输出: 0  
解释: 原数组为 [0,1,2,4,5,6,7] , 旋转 4 次得到输入数组。

#### 示例 3:

输入: nums = [11,13,15,17]  
输出: 11  
解释: 原数组为 [11,13,15,17] , 旋转 4 次得到输入数组。

#### 提示:

- $n == \text{nums.length}$
- $1 \leq n \leq 5000$
- $-5000 \leq \text{nums}[i] \leq 5000$
- $\text{nums}$  中的所有整数互不相同
- $\text{nums}$  原来是一个升序排序的数组，并进行了  $1$  至  $n$  次旋转

#### 思考过程:

这道题要求在一个旋转排序数组中找到最小值。旋转排序数组是指一个有序数组在某个未知点进行了旋转，例如 [0,1,2,4,5,6,7] 可能会变成 [4,5,6,7,0,1,2]。数组中没有重复元素。

#### 核心思路:

我们可以利用二分查找的思想来找到最小值。旋转排序数组的特点是，最小值是唯一一个比它右边的元素小的元素（如果数组没有旋转，则第一个元素就是最小值）。

1. **初始化:** `left = 0, right = len(nums) - 1`。
2. **循环条件:** `left < right` (当 `left == right` 时, `left` 或 `right` 指向的就是最小值)。
3. **计算 mid:** `mid = left + (right - left) // 2`。
4. **比较 `nums[mid]` 与 `nums[right]`:**
  - 如果 `nums[mid] < nums[right]`：
    - 这说明 `mid` 到 `right` 这一段是升序的，最小值可能在 `mid` 或 `mid` 的左边。所以，将 `right = mid`。
  - 如果 `nums[mid] > nums[right]`：
    - 这说明 `mid` 到 `right` 这一段不是升序的，最小值一定在 `mid` 的右边 (因为 `nums[mid]` 比 `nums[right]` 大，说明 `mid` 肯定不是最小值，且旋转点在 `mid` 的右边)。所以，将 `left = mid + 1`。
5. **循环结束时, `left` (或 `right`) 指向的就是最小值。**

#### 特殊情况:

- 如果数组没有旋转 (即 `nums[0] < nums[len(nums) - 1]`)，那么第一个元素就是最小值。

#### Python 代码:

```
class Solution:  
    def findMin(self, nums: list[int]) -> int:  
        left, right = 0, len(nums) - 1  
  
        # 如果数组没有旋转，或者只有一个元素，则第一个元素就是最小值  
        if nums[left] <= nums[right]:  
            return nums[left]  
  
        while left < right:  
            mid = left + (right - left) // 2  
  
            # 如果 mid 处的元素小于最右边的元素，说明 mid 到 right 是有序的
```

```

# 最小值可能在 mid 或 mid 的左边
if nums[mid] < nums[right]:
    right = mid
# 如果 mid 处的元素大于最右边的元素，说明 mid 到 right 是无序的
# 最小值一定在 mid 的右边
else:
    left = mid + 1

return nums[left]

```

## 二分查找 / 33. 搜索旋转排序数组

### 33. 搜索旋转排序数组

#### 题目描述

整数数组 nums 按升序排列，数组中的值互不相同。

在传递给函数之前，nums 在预先未知的某个下标  $k$  ( $0 \leq k < \text{nums.length}$ ) 上进行了旋转，使数组变为  $[\text{nums}[k], \text{nums}[k+1], \dots, \text{nums}[\text{n}-1], \text{nums}[0], \text{nums}[1], \dots, \text{nums}[\text{k}-1]]$  (下标从 0 开始计数)。例如， $[0,1,2,4,5,6,7]$  在下标 3 处经旋转后可能变为  $[4,5,6,7,0,1,2]$ 。

给你旋转后的数组 nums 和一个整数 target，如果 nums 中存在这个目标值 target，则返回它的下标，否则返回 -1。

你必须设计一个时间复杂度为  $O(\log n)$  的算法解决此问题。

#### 示例 1：

```

输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4

```

#### 示例 2：

```

输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1

```

#### 示例 3：

```

输入: nums = [1], target = 0
输出: -1

```

#### 提示：

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- nums 中的每个值都独一无二
- 题目数据保证 nums 在预先未知的某个下标上进行了旋转
- $-10^4 \leq \text{target} \leq 10^4$

#### 思考过程：

这道题要求在一个旋转排序数组中搜索目标值。旋转排序数组是指一个有序数组在某个未知点进行了旋转，例如  $[0,1,2,4,5,6,7]$  可能会变成  $[4,5,6,7,0,1,2]$ 。

#### 核心思路：

虽然数组被旋转了，但它仍然保持着局部有序的特性。我们可以利用二分查找的思想，每次将搜索范围缩小一半。关键在于判断 mid 所在的区间是左半部分有序还是右半部分有序。

1. **初始化：**  $\text{left} = 0, \text{right} = \text{len}(\text{nums}) - 1$ 。
2. **循环条件：**  $\text{left} \leq \text{right}$ 。
3. **计算 mid：**  $\text{mid} = \text{left} + (\text{right} - \text{left}) // 2$ 。
4. **判断  $\text{nums}[\text{mid}]$  是否等于 target：** 如果是，直接返回 mid。
5. **判断 mid 所在的区间是左半部分有序还是右半部分有序：**
  - **情况一：左半部分有序 ( $\text{nums}[\text{left}] \leq \text{nums}[\text{mid}]$ )**
    - 如果  $\text{target}$  在左半部分范围内 ( $\text{nums}[\text{left}] \leq \text{target} < \text{nums}[\text{mid}]$ )，则  $\text{right} = \text{mid} - 1$ 。

- 否则，`target` 在右半部分，`left = mid + 1`。
- 情况二：右半部分有序 (`nums[left] > nums[mid]`)
  - 如果 `target` 在右半部分范围内 (`nums[mid] < target <= nums[right]`)，则 `left = mid + 1`。
  - 否则，`target` 在左半部分，`right = mid - 1`。
- 6. 如果循环结束仍未找到，返回 `-1`。

Python 代码：

```
class Solution:
    def search(self, nums: list[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            if nums[mid] == target:
                return mid

            # 判断左半部分是否有序
            if nums[left] <= nums[mid]:
                # target 在左半部分有序区间内
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                # target 在右半部分无序区间内
                else:
                    left = mid + 1
            # 右半部分有序
            else:
                # target 在右半部分有序区间内
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                # target 在左半部分无序区间内
                else:
                    right = mid - 1

        return -1
```

## 二分查找 / 34. 在排序数组中查找元素的第一个和最后一个位置

### 34. 在排序数组中查找元素的第一个和最后一个位置

#### 题目描述

给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你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]`。

你必须设计并实现时间复杂度为  $O(\log n)$  的算法解决此问题。

#### 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]
```

#### 示例 3：

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

#### 提示:

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $\text{nums}$  是一个非递减数组
- $-10^9 \leq \text{target} \leq 10^9$

#### 思考过程:

这道题要求在一个有序数组中查找给定目标值的第一个和最后一个位置。如果目标值不存在，则返回 `[-1, -1]`。

#### 核心思路:

我们可以使用两次二分查找来解决这个问题：一次查找目标值的第一个出现位置，另一次查找目标值的最后一个出现位置。

#### 查找第一个出现位置 (`find_first_occurrence`):

1. **初始化:** `left = 0, right = len(nums) - 1, first_pos = -1`。
2. **循环条件:** `left <= right`。
3. **计算 mid:** `mid = left + (right - left) // 2`。
4. **比较 `nums[mid]` 与 `target`:**
  - 如果 `nums[mid] == target`:
    - `first_pos = mid` (记录当前位置)
    - `right = mid - 1` (继续向左查找，看是否有更小的索引)
  - 如果 `nums[mid] < target`:
    - `left = mid + 1` (目标值在右侧)
  - 如果 `nums[mid] > target`:
    - `right = mid - 1` (目标值在左侧)
5. **返回 `first_pos`。**

#### 查找最后一个出现位置 (`find_last_occurrence`):

1. **初始化:** `left = 0, right = len(nums) - 1, last_pos = -1`。
2. **循环条件:** `left <= right`。
3. **计算 mid:** `mid = left + (right - left) // 2`。
4. **比较 `nums[mid]` 与 `target`:**
  - 如果 `nums[mid] == target`:
    - `last_pos = mid` (记录当前位置)
    - `left = mid + 1` (继续向右查找，看是否有更大的索引)
  - 如果 `nums[mid] < target`:
    - `left = mid + 1` (目标值在右侧)
  - 如果 `nums[mid] > target`:
    - `right = mid - 1` (目标值在左侧)
5. **返回 `last_pos`。**

#### 主函数逻辑:

1. 调用 `find_first_occurrence` 得到第一个位置。
2. 调用 `find_last_occurrence` 得到最后一个位置。
3. 返回 `[first_pos, last_pos]`。

#### Python 代码:

```
class Solution:
    def searchRange(self, nums: list[int], target: int) -> list[int]:
        def find_first_occurrence(nums, target):
            left, right = 0, len(nums) - 1
            first_pos = -1
            while left <= right:
                mid = left + (right - left) // 2
                if nums[mid] == target:
                    first_pos = mid
                if nums[mid] < target:
                    left = mid + 1
                else:
                    right = mid - 1
            return first_pos

        def find_last_occurrence(nums, target):
            left, right = 0, len(nums) - 1
            last_pos = -1
            while left <= right:
                mid = left + (right - left) // 2
                if nums[mid] == target:
                    last_pos = mid
                if nums[mid] > target:
                    right = mid - 1
                else:
                    left = mid + 1
            return last_pos

        first_pos = find_first_occurrence(nums, target)
        last_pos = find_last_occurrence(nums, target)
        return [first_pos, last_pos]
```

```

        right = mid - 1 # 继续向左查找
    elif nums[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
    return first_pos

def find_last_occurrence(nums, target):
    left, right = 0, len(nums) - 1
    last_pos = -1
    while left <= right:
        mid = left + (right - left) // 2
        if nums[mid] == target:
            last_pos = mid
            left = mid + 1 # 继续向右查找
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return last_pos

first = find_first_occurrence(nums, target)
last = find_last_occurrence(nums, target)

return [first, last]

```

## 二分查找 / 35. 搜索插入位置

### 35. 搜索插入位置

#### 题目描述

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

#### 示例 3：

输入: nums = [1,3,5,6], target = 7  
输出: 4

#### 提示：

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- nums 为 无重复元素 的 升序 排列数组
- $-10^4 \leq \text{target} \leq 10^4$

#### 思考过程:

这道题要求在一个有序数组中找到目标值，如果目标值存在，则返回其索引；如果不存在，则返回它应该被插入的位置，以保持数组的有序性。这是一个典型的二分查找问题。

#### 核心思路：

二分查找的核心思想是每次将搜索范围缩小一半。

1. **初始化**: 设置 `left = 0` 和 `right = len(nums) - 1`，表示搜索范围的左右边界。
2. **循环条件**: 当 `left <= right` 时，继续循环。
3. **计算中间索引**: `mid = left + (right - left) // 2`。
4. **比较 `nums[mid]` 与 `target`**:
  - 如果 `nums[mid] == target`，说明找到了目标值，返回 `mid`。
  - 如果 `nums[mid] < target`，说明目标值在 `mid` 的右侧，更新 `left = mid + 1`。
  - 如果 `nums[mid] > target`，说明目标值在 `mid` 的左侧，更新 `right = mid - 1`。

#### 插入位置的确定：

当循环结束时（即 `left > right`），说明目标值不在数组中。此时 `left` 指向的就是目标值应该被插入的位置。

- 如果 `target` 比数组中所有元素都大，`left` 会一直向右移动，最终 `left` 会等于 `len(nums)`。
- 如果 `target` 比数组中所有元素都小，`left` 会保持为 0。
- 如果 `target` 在数组中间的某个位置，`left` 会指向第一个大于 `target` 的元素的索引。

#### Python 代码：

```
class Solution:  
    def searchInsert(self, nums: list[int], target: int) -> int:  
        left, right = 0, len(nums) - 1  
  
        while left <= right:  
            mid = left + (right - left) // 2  
            if nums[mid] == target:  
                return mid  
            elif nums[mid] < target:  
                left = mid + 1  
            else:  
                right = mid - 1  
  
        # 循环结束时, left 指向的就是目标值应该插入的位置  
        return left
```

## 二分查找 / 4. 寻找两个正序数组的中位数

### 4. 寻找两个正序数组的中位数

#### 题目描述

给定两个大小分别为 m 和 n 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 中位数。

算法的时间复杂度应该为  $O(\log(m+n))$ 。

#### 示例 1：

```
输入: nums1 = [1,3], nums2 = [2]  
输出: 2.00000  
解释: 合并数组 = [1,2,3] , 中位数 2
```

#### 示例 2：

```
输入: nums1 = [1,2], nums2 = [3,4]  
输出: 2.50000  
解释: 合并数组 = [1,2,3,4] , 中位数 (2 + 3) / 2 = 2.5
```

#### 提示：

- $\text{nums1.length} == m$
- $\text{nums2.length} == n$
- $0 \leq m \leq 1000$
- $0 \leq n \leq 1000$
- $1 \leq m + n \leq 2000$
- $-10^6 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^6$

#### 思考过程:

这道题要求在两个已排序的数组 `nums1` 和 `nums2` 中找到它们合并后的中位数。时间复杂度要求为  $O(\log(m+n))$ ，这暗示我们需要使用二分查找。

#### 核心思路:

中位数的定义是将一个集合分成两个长度相等的子集，其中一个子集中的所有元素都大于另一个子集中的所有元素。对于两个已排序的数组，我们可以通过在其中一个数组上进行二分查找，来找到一个“分割点”，使得分割后的两部分满足中位数的定义。

假设我们要在 `nums1` 中找到一个分割点 `i`，使得 `nums1` 被分成 `nums1[0...i-1]` 和 `nums1[i...m-1]`。那么 `nums2` 也需要有一个对应的分割点 `j`，使得 `nums2` 被分成 `nums2[0...j-1]` 和 `nums2[j...n-1]`。

为了满足中位数的定义，我们需要满足以下条件：

1. 左右两部分长度相等或左边多一个:  $i + j = (m + n + 1) // 2$  (对于奇数长度，左边多一个；对于偶数长度，两边相等)。
2. 左边最大值  $\leq$  右边最小值:

- `max(nums1[i-1], nums2[j-1]) <= min(nums1[i], nums2[j])`

#### 二分查找的范围:

我们可以在较短的数组上进行二分查找，例如在 `nums1` 上查找 `i`。`i` 的范围是 `[0, m]`。

#### 具体步骤:

1. 确保 `nums1` 是较短的数组。如果不是，交换 `nums1` 和 `nums2`。
2. 初始化 `left = 0, right = m` (`m` 是 `nums1` 的长度)。
3. 在 `[left, right]` 范围内进行二分查找 `i`：
  - `i = left + (right - left) // 2`
  - `j = (m + n + 1) // 2 - i`
  - 获取 `nums1_left_max, nums1_right_min, nums2_left_max, nums2_right_min`。
    - 如果 `i == 0`, `nums1_left_max = -float('inf')`
    - 如果 `i == m`, `nums1_right_min = float('inf')`
    - 如果 `j == 0`, `nums2_left_max = -float('inf')`
    - 如果 `j == n`, `nums2_right_min = float('inf')`
4. 判断条件:
  - 如果 `nums1_left_max <= nums2_right_min` 且 `nums2_left_max <= nums1_right_min`:
    - 找到了正确的分割点。
    - 如果  $(m + n)$  是奇数，中位数是 `max(nums1_left_max, nums2_left_max)`。
    - 如果  $(m + n)$  是偶数，中位数是 `(max(nums1_left_max, nums2_left_max) + min(nums1_right_min, nums2_right_min)) / 2.0`。
  - 如果 `nums1_left_max > nums2_right_min`:
    - 说明 `nums1` 的左半部分太大了，需要将 `i` 向左移动，即 `right = i - 1`。
  - 否则 (`nums2_left_max > nums1_right_min`):
    - 说明 `nums2` 的左半部分太大了，需要将 `i` 向右移动，即 `left = i + 1`。

#### Python 代码:

```
class Solution:
    def findMedianSortedArrays(self, nums1: list[int], nums2: list[int]) -> float:
        m, n = len(nums1), len(nums2)

        # 确保 nums1 是较短的数组，这样可以减少二分查找的范围
        if m > n:
            nums1, nums2 = nums2, nums1
            m, n = n, m

        left, right = 0, m
        half_len = (m + n + 1) // 2

        while left <= right:
```

```

i = left + (right - left) // 2 # nums1 的分割点
j = half_len - i # nums2 的分割点

# 获取分割点左右的四个值
nums1_left_max = nums1[i-1] if i != 0 else -float('inf')
nums1_right_min = nums1[i] if i != m else float('inf')
nums2_left_max = nums2[j-1] if j != 0 else -float('inf')
nums2_right_min = nums2[j] if j != n else float('inf')

# 检查是否找到了正确的分割点
if nums1_left_max <= nums2_right_min and nums2_left_max <= nums1_right_min:
    # 找到了正确的分割点
    # 计算中位数
    if (m + n) % 2 == 1: # 总长度为奇数
        return float(max(nums1_left_max, nums2_left_max))
    else: # 总长度为偶数
        return (max(nums1_left_max, nums2_left_max) + min(nums1_right_min, nums2_right_min))/2.0
elif nums1_left_max > nums2_right_min:
    # nums1 的左半部分太大，需要将 i 向左移动
    right = i - 1
else:
    # nums2 的左半部分太大，需要将 i 向右移动
    left = i + 1

return 0.0 # 理论上不会执行到这里

```

## 二分查找 / 74. 搜索二维矩阵

### 74. 搜索二维矩阵

#### 题目描述

给你一个满足下述两条属性的  $m \times n$  整数矩阵 matrix：

- 每行中的整数从左到右按非递减顺序排列。
- 每行的第一个整数大于前一行的最后一个整数。

给你一个整数 target，如果 target 在矩阵中，返回 true；否则，返回 false。

示例 1：

1	3	5	7
10	11	16	20
23	30	34	60

输入：matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 5

输出：true

示例 2：

1	3	5	7
10	11	16	20
23	30	34	60

```
输入: matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 13
输出: true
```

示例 3:

1	3	5	7
10	11	16	20
23	30	34	60

```
输入: matrix = [[1,4,7,11],[2,5,8,12],[3,6,9,16],[10,13,14,17]], target = 20
输出: false
```

提示:

- $m == \text{matrix.length}$
- $n == \text{matrix[i].length}$
- $1 \leq m, n \leq 100$
- $-10^4 \leq \text{matrix}[i][j], \text{target} \leq 10^4$

思考过程:

这道题要求在一个  $m \times n$  的二维矩阵中搜索一个目标值。这个矩阵有以下特点:

1. 每行中的整数从左到右按升序排列。
2. 每行的第一个整数大于前一行的最后一个整数。

这些特点使得我们可以将整个矩阵看作一个一维的有序数组，然后进行二分查找。

核心思路:

将二维矩阵的索引  $(\text{row}, \text{col})$  映射到一维数组的索引  $\text{index}$ ，反之亦然。

- 从  $(\text{row}, \text{col})$  到  $\text{index}$ :  $\text{index} = \text{row} * n + \text{col}$  (其中  $n$  是矩阵的列数)。
- 从  $\text{index}$  到  $(\text{row}, \text{col})$ :  $\text{row} = \text{index} // n, \text{col} = \text{index} \% n$ 。

有了这个映射关系，我们就可以对这个“虚拟”的一维数组进行二分查找。

1. 初始化:

- $m = \text{len}(\text{matrix})$  (行数)
- $n = \text{len}(\text{matrix}[0])$  (列数)
- $\text{left} = 0$  (虚拟一维数组的起始索引)
- $\text{right} = m * n - 1$  (虚拟一维数组的结束索引)

2. 循环条件: 当  $\text{left} \leq \text{right}$  时，继续循环。

3. 计算中间索引:  $\text{mid} = \text{left} + (\text{right} - \text{left}) // 2$ 。

4. 将  $\text{mid}$  映射回二维矩阵的  $(\text{row}, \text{col})$ :

- $\text{mid\_row} = \text{mid} // n$
- $\text{mid\_col} = \text{mid} \% n$

5. 比较  $\text{matrix}[\text{mid\_row}][\text{mid\_col}]$  与  $\text{target}$ :

- 如果  $\text{matrix}[\text{mid\_row}][\text{mid\_col}] == \text{target}$ ，说明找到了目标值，返回 `True`。
- 如果  $\text{matrix}[\text{mid\_row}][\text{mid\_col}] < \text{target}$ ，说明目标值在  $\text{mid}$  的右侧，更新  $\text{left} = \text{mid} + 1$ 。
- 如果  $\text{matrix}[\text{mid\_row}][\text{mid\_col}] > \text{target}$ ，说明目标值在  $\text{mid}$  的左侧，更新  $\text{right} = \text{mid} - 1$ 。

6. 如果循环结束仍未找到，返回 `False`。

Python 代码:

```
class Solution:
    def searchMatrix(self, matrix: list[list[int]], target: int) -> bool:
        m = len(matrix)
        if m == 0:
            return False
```

```

n = len(matrix[0])
if n == 0:
    return False

left, right = 0, m * n - 1

while left <= right:
    mid = left + (right - left) // 2
    mid_row = mid // n
    mid_col = mid % n

    if matrix[mid_row][mid_col] == target:
        return True
    elif matrix[mid_row][mid_col] < target:
        left = mid + 1
    else:
        right = mid - 1

return False

```

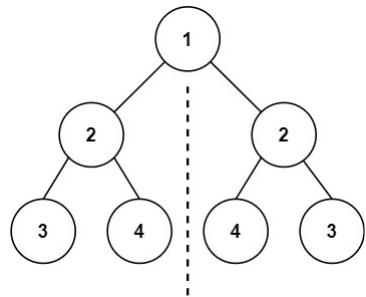
## 二叉树 / 101. 对称二叉树

### 101. 对称二叉树

#### 题目描述

给你一个二叉树的根节点 `root`，检查它是否轴对称。

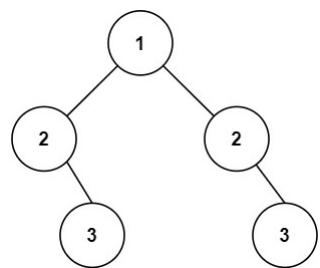
示例 1:



输入: `root = [1,2,2,3,4,4,3]`

输出: `true`

示例 2:



输入: `root = [1,2,2,null,3,null,3]`

输出: `false`

提示:

- 树中节点数目在范围 `[1, 1000]` 内
- `-100 <= Node.val <= 100`

解题思路

## 方法一：递归

### 思考过程

判断一棵树是否对称，等价于判断其根节点的左子树和右子树是否“镜像对称”。

如何定义两棵树（或子树）是镜像对称的？

1. 它们的根节点具有相同的值。
2. 树 A 的左子树与树 B 的右子树是镜像对称的。
3. 树 A 的右子树与树 B 的左子树是镜像对称的。

基于这个定义，我们可以设计一个递归函数来比较两个子树是否镜像对称。

### 算法流程

1. **主函数 `isSymmetric(root)` :**
  - 如果根节点 `root` 为空，那么它本身是对称的，返回 `true`。
  - 调用一个辅助函数 `isMirror(p, q)`，并传入根节点的左子树 `root.left` 和右子树 `root.right`。
2. **辅助函数 `isMirror(p, q)` :**
  - 该函数用于判断两个节点 `p` 和 `q` 所代表的子树是否镜像对称。
  - **递归终止条件:**
    - 如果 `p` 和 `q` 都为 `null`，说明到达了对称的叶子节点，返回 `true`。
    - 如果 `p` 或 `q` 中只有一个为 `null`，或者 `p.val != q.val`，说明结构不对称或值不对称，返回 `false`。
  - **递归过程:**
    - 递归地比较 `p` 的左子树和 `q` 的右子树: `isMirror(p.left, q.right)`。
    - 递归地比较 `p` 的右子树和 `q` 的左子树: `isMirror(p.right, q.left)`。
  - **返回值:** 只有当以上两个递归调用都返回 `true` 时，才说明 `p` 和 `q` 的子树是镜像对称的，返回 `true`。

### 复杂度分析

- **时间复杂度:**  $O(N)$ ，其中  $N$  是二叉树的节点数。我们对每个节点只访问一次。
- **空间复杂度:**  $O(H)$ ，其中  $H$  是二叉树的高度。递归调用栈的深度取决于树的高度。在最坏情况下（树退化为链表），空间复杂度为  $O(N)$ 。

## 方法二：迭代（使用队列）

### 思考过程

除了递归，我们也可以使用迭代的方式来实现。我们可以用一个队列来模拟递归的过程。每次从队列中取出两个节点进行比较，然后将它们的子节点按镜像对称的顺序（左对右，右对左）放入队列中。

### 算法流程

1. **初始化:**
  - 创建一个队列 `queue`。
  - 将根节点的左子树 `root.left` 和右子树 `root.right` 入队。
2. **迭代比较:**
  - 当队列不为空时，循环执行以下操作：
    - 从队列中同时取出两个节点 `p` 和 `q`。
    - 如果 `p` 和 `q` 都为 `null`，继续下一次循环。
    - 如果 `p` 或 `q` 中只有一个为 `null`，或者 `p.val != q.val`，返回 `false`。
  - **按镜像顺序入队:**
    - 将 `p.left` 和 `q.right` 入队。
    - 将 `p.right` 和 `q.left` 入队。
3. **返回值:** 如果循环正常结束，说明整棵树都是对称的，返回 `true`。

### 复杂度分析

- **时间复杂度:**  $O(N)$ ，其中  $N$  是二叉树的节点数。每个节点都会被访问一次。
- **空间复杂度:**  $O(W)$ ，其中  $W$  是二叉树的最大宽度。在最坏情况下，队列中最多会存储  $N/2$  个节点，空间复杂度为  $O(N)$ 。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
```

```

# 方法一：递归
class Solution_Recursive:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True

        def isMirror(p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
            if not p and not q:
                return True
            if not p or not q or p.val != q.val:
                return False

            return isMirror(p.left, q.right) and isMirror(p.right, q.left)

        return isMirror(root.left, root.right)

# 方法二：迭代
import collections

class Solution_Iterative:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True

        queue = collections.deque([root.left, root.right])

        while queue:
            p = queue.popleft()
            q = queue.popleft()

            if not p and not q:
                continue

            if not p or not q or p.val != q.val:
                return False

            queue.append(p.left)
            queue.append(q.right)
            queue.append(p.right)
            queue.append(q.left)

        return True

```

## 二叉树 / 102. 二叉树的层序遍历

### 102. 二叉树的层序遍历

#### 题目描述

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 (即逐层地，从左到右访问所有节点)。

#### 示例

##### 示例：

二叉树： [3,9,20,null,null,15,7] ，



```
/ \  
15   7
```

返回其层序遍历结果：

```
[  
 [3],  
 [9,20],  
 [15,7]  
]
```

## 提示

- 树中节点数目在范围  $[0, 2000]$  内
- $-1000 \leq \text{Node.val} \leq 1000$

## 解题思路

### 方法：广度优先搜索 (BFS)

层序遍历的本质是广度优先搜索 (BFS)。我们可以使用一个队列来实现。

算法流程：

- 如果根节点 `root` 为空，直接返回空列表 `[]`。
- 初始化一个队列 `queue`，并将根节点 `root` 入队。
- 初始化一个结果列表 `result`，用于存储每一层的节点值。
- 当队列 `queue` 不为空时，进行循环：
  - 获取当前层的节点数量 `level_size`。
  - 创建一个临时列表 `current_level`，用于存储当前层的所有节点值。
  - 循环 `level_size` 次，处理当前层的所有节点：
    - 从队列中取出一个节点 `node`。
    - 将 `node.val` 添加到 `current_level` 中。
    - 如果 `node` 的左子节点不为空，将其入队。
    - 如果 `node` 的右子节点不为空，将其入队。
  - 将 `current_level` 添加到 `result` 中。
- 循环结束后，返回 `result`。

复杂度分析：

- 时间复杂度： $O(n)$ ，其中  $n$  是二叉树的节点数。每个节点都会被访问一次。
- 空间复杂度： $O(w)$ ，其中  $w$  是二叉树的最大宽度。在最坏情况下（完全二叉树的最后一层），空间复杂度为  $O(n/2) = O(n)$ 。

## Python 代码实现

```
# Definition for a binary tree node.  
# class TreeNode:  
#     def __init__(self, val=0, left=None, right=None):  
#         self.val = val  
#         self.left = left  
#         self.right = right  
  
from collections import deque  
  
class Solution:  
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:  
        if not root:  
            return []  
  
        result = []  
        queue = deque([root])  
  
        while queue:
```

```

level_size = len(queue)
current_level = []

for _ in range(level_size):
    node = queue.popleft()
    current_level.append(node.val)

    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)

result.append(current_level)

return result

```

## 二叉树 / 104. 二叉树的最大深度

### 104. 二叉树的最大深度

#### 题目描述

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

**说明:** 叶子节点是指没有子节点的节点。

**示例:**

给定二叉树 [3,9,20,null,null,15,7]，



返回它的最大深度 3。

#### 解题思路

##### 方法一：深度优先搜索（DFS）

##### 思考过程

要求二叉树的最大深度，很自然地会想到使用递归。一棵二叉树的最大深度，等于其左子树的最大深度和右子树的最大深度的较大者，再加上根节点自身的深度 1。

- 如果根节点为空，则深度为 0。
- 如果根节点不为空，则其最大深度为  $\max(\text{左子树深度}, \text{右子树深度}) + 1$ 。

这正是一个典型的深度优先搜索的应用场景。

#### 算法流程

- 递归终止条件:** 如果当前节点 `root` 为 `null`，说明到达了叶子节点的下一层，深度为 0，直接返回 0。
- 递归过程:**
  - 递归计算左子树的最大深度 `left_depth = maxDepth(root.left)`。
  - 递归计算右子树的最大深度 `right_depth = maxDepth(root.right)`。
- 返回值:** 返回  $\max(\text{left\_depth}, \text{right\_depth}) + 1$ ，即左右子树深度的较大值加上当前节点的深度 1。

#### 复杂度分析

- 时间复杂度:**  $O(N)$ ，其中  $N$  是二叉树的节点数。我们需要遍历每个节点一次。
- 空间复杂度:**  $O(H)$ ，其中  $H$  是二叉树的高度。递归调用栈的深度取决于树的高度。在最坏情况下（树退化为链表），空间复杂度为  $O(N)$ ；在最好情况下（完全二叉树），空间复杂度为  $O(\log N)$ 。

##### 方法二：广度优先搜索（BFS）

##### 思考过程

我们也可以使用广度优先搜索（层序遍历）来计算二叉树的最大深度。其思想是一层一层地遍历二叉树，遍历了多少层，最大深度就是多少。

## 算法流程

### 1. 初始化：

- 如果根节点 `root` 为空，返回深度 0。
- 创建一个队列 `queue`，并将根节点 `root` 入队。
- 初始化深度 `depth` 为 0。

### 2. 层序遍历：

- 当队列不为空时，循环执行以下操作：
  - 记录当前层的节点数量 `level_size`。
  - 将深度 `depth` 加 1。
  - 循环 `level_size` 次，处理当前层的所有节点：
    - 从队列中取出一个节点 `node`。
    - 如果 `node` 的左子节点不为空，将其左子节点入队。
    - 如果 `node` 的右子节点不为空，将其右子节点入队。

### 3. 返回值：循环结束后，`depth` 即为二叉树的最大深度。

## 复杂度分析

- 时间复杂度：O(N)，其中 N 是二叉树的节点数。每个节点都会被访问一次。
- 空间复杂度：O(W)，其中 W 是二叉树的最大宽度。在最坏情况下（完全二叉树），队列中最多会存储  $N/2$  个节点，空间复杂度为 O(N)。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

# 方法一：深度优先搜索（DFS）
class Solution_DFS:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        return max(left_depth, right_depth) + 1

# 方法二：广度优先搜索（BFS）
import collections

class Solution_BFS:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        queue = collections.deque([root])
        depth = 0

        while queue:
            depth += 1
            level_size = len(queue)
            for _ in range(level_size):
                node = queue.popleft()
                if node.left:
                    queue.append(node.left)
                if node.right:
```

```
queue.append(node.right)

return depth
```

## 二叉树 / 105. 从前序与中序遍历序列构造二叉树

### 105. 从前序与中序遍历序列构造二叉树

#### 题目描述

给定两个整数数组 preorder 和 inorder，其中 preorder 是二叉树的 先序遍历， inorder 是同一棵树的 中序遍历，请构造二叉树并返回其根节点。

#### 示例 1:

```
输入: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
输出: [3,9,20,null,null,15,7]
```

#### 示例 2:

```
输入: preorder = [-1], inorder = [-1]
输出: [-1]
```

#### 提示:

- $1 \leq \text{preorder.length} \leq 3000$
- $\text{inorder.length} == \text{preorder.length}$
- $-3000 \leq \text{preorder}[i], \text{inorder}[i] \leq 3000$
- preorder 和 inorder 均 无重复 元素
- inorder 均出现在 preorder
- preorder 保证为二叉树的前序遍历序列
- inorder 保证为二叉树的中序遍历序列

## 解题思路

### 方法一：递归

要通过前序遍历和中序遍历序列构造一棵二叉树，关键在于理解这两种遍历方式的特性：

- **前序遍历** 的顺序是 [根节点, [左子树的前序遍历], [右子树的前序遍历]]。
- **中序遍历** 的顺序是 [[左子树的中序遍历], 根节点, [右子树的中序遍历]]。

根据这个特性，我们可以确定：

1. 前序遍历序列的第一个元素一定是当前树（或子树）的根节点。3
2. 在中序遍历序列中，根节点左边的所有元素都属于左子树，右边的所有元素都属于右子树。4

基于以上两点，我们可以使用递归的方法来构建整棵树。

#### 算法流程：

1. 创建一个辅助哈希表 inorder\_map，用于快速查找中序遍历中每个元素对应的索引。这可以避免每次递归都线性扫描 inorder 数组，从而优化时间复杂度。
2. 定义一个递归函数 build(pre\_start, pre\_end, in\_start, in\_end)，该函数根据指定范围的前序和中序序列来构建子树。
  - a. 确定根节点：当前子树的根节点值是 preorder[pre\_start]。
  - b. 创建根节点：root = TreeNode(root\_val)。
  - c. 找到根节点在中序遍历中的位置：in\_root\_idx = inorder\_map[root\_val]。
  - d. 计算左子树的大小：left\_subtree\_size = in\_root\_idx - in\_start。
- e. 递归构建左子树：左子树的前序遍历范围是 [pre\_start + 1, pre\_start + left\_subtree\_size]，中序遍历范围是 [in\_start, in\_root\_idx - 1]。调用 build 函数构建左子树，并将其赋值给 root.left。
- f. 递归构建右子树：右子树的前序遍历范围是 [pre\_start + left\_subtree\_size + 1, pre\_end]，中序遍历范围是 [in\_root\_idx + 1, in\_end]。调用 build 函数构建右子树，并将其赋值给 root.right。
3. 返回根节点 root。

#### 复杂度分析：

- 时间复杂度：O(N)，其中 N 是树中节点的数量。我们需要遍历每个节点一次来构建树。哈希表的查询操作是 O(1) 的。5
- 空间复杂度：O(N)。我们需要一个哈希表来存储中序遍历的映射，空间为 O(N)。此外，递归的深度在最坏情况下（链状树）也是 O(N)，因此递归栈的空间也是 O(N)。6

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if not preorder or not inorder:
            return None

        # 使用哈希表来快速查找中序遍历中元素的位置
        inorder_map = {val: i for i, val in enumerate(inorder)}

        def build(pre_start, pre_end, in_start, in_end):
            # 递归终止条件
            if pre_start > pre_end or in_start > in_end:
                return None

            # 前序遍历的第一个节点是根节点
            root_val = preorder[pre_start]
            root = TreeNode(root_val)

            # 在中序遍历中找到根节点的位置
            in_root_idx = inorder_map[root_val]

            # 计算左子树的节点数量
            left_subtree_size = in_root_idx - in_start

            # 递归构建左子树
            root.left = build(
                pre_start + 1,
                pre_start + left_subtree_size,
                in_start,
                in_root_idx - 1
            )

            # 递归构建右子树
            root.right = build(
                pre_start + left_subtree_size + 1,
                pre_end,
                in_root_idx + 1,
                in_end
            )

            return root

        return build(0, len(preorder) - 1, 0, len(inorder) - 1)
```

---

## 二叉树 / 108. 将有序数组转换为二叉搜索树

### 题目描述

给你一个整数数组 `nums`，其中元素已经按 升序 排列，请你将其转换为一棵 高度平衡 二叉搜索树。

高度平衡 二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1」的二叉树。

## 解题思路

这道题的核心要求有两个：

1. 构建一棵 **二叉搜索树 (BST)**。
2. 这棵树必须是 **高度平衡的**。

利用数组已经 **升序排列** 这一关键特性，我们可以很自然地构建出满足要求的树。

**核心思想：**

为了保证树是高度平衡的，我们应该尽量让每个节点的左右子树的节点数量大致相等。对于一个升序数组，最好的分割点就是数组的 **中点**。

因此，我们可以采用以下递归的策略：

1. 选择数组的中间元素作为当前子树的根节点。
2. 数组中点左边的部分，是根节点的左子树的元素。我们对这部分数组递归地构建左子树。
3. 数组中点右边的部分，是根节点的右子树的元素。我们对这部分数组递归地构建右子树。
4. 递归的终止条件是，当处理的子数组为空时，返回 `null`。

通过这种方式，每次都选择中间元素作为根，可以确保左右子树的高度差不会超过 1，从而构建出一棵高度平衡的二叉搜索树。

## 算法步骤

1. 定义一个辅助函数 `build(nums, left, right)`，它负责将 `nums` 数组在 `[left, right]` 闭区间内的元素转换成一棵平衡二叉搜索树，并返回根节点。
2. **递归终止条件**：如果 `left > right`，说明当前区间没有元素，无法构成节点，返回 `None`（或 `null`）。
3. **寻找根节点**：计算区间的中间位置 `mid = left + (right - left) // 2`。
4. **创建根节点**：使用 `nums[mid]` 的值创建一个新的树节点 `TreeNode`。
5. **构建左子树**：递归调用 `build(nums, left, mid - 1)` 来构建根节点的左子树，并将返回的子树根节点连接到当前根的 `left` 指针上。
6. **构建右子树**：递归调用 `build(nums, mid + 1, right)` 来构建根节点的右子树，并将返回的子树根节点连接到当前根的 `right` 指针上。
7. 返回当前创建的根节点。

主函数 `sortedArrayToBST` 只需调用 `build(nums, 0, len(nums) - 1)` 即可启动整个过程。

## Python 代码实现

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def sortedArrayToBST(self, nums: list[int]) -> TreeNode | None:
        """
        将一个升序数组转换为一棵高度平衡的二叉搜索树。
        """

        def build(left: int, right: int) -> TreeNode | None:
            # 如果左边界大于右边界，说明子数组为空，返回 None
            if left > right:
                return None

            # 选择中间位置的元素作为根节点
            # 这样可以保证左右子树的节点数量差不超过1
            mid = left + (right - left) // 2

            # 创建根节点
            root = TreeNode(nums[mid])

            # 递归地构建左子树
            root.left = build(left, mid - 1)

            # 递归地构建右子树
            root.right = build(mid + 1, right)

        return build(0, len(nums) - 1)
```

```
    return root

    # 从整个数组范围开始构建
    return build(0, len(nums) - 1)
```

## 复杂度分析

- 时间复杂度:  $O(N)$

其中  $N$  是数组中元素的数量。我们需要遍历数组中的每个元素一次，将其转换为一个树节点。

- 空间复杂度:  $O(\log N)$

空间复杂度主要取决于递归调用的栈深度。由于我们每次都选择中间元素作为根，构建出的树是高度平衡的，其高度大约为  $\log N$ 。因此，递归栈的最大深度是  $O(\log N)$ 。这不包括存储输出树本身所需的  $O(N)$  空间。

## 二叉树 / 114. 二叉树展开为链表

### 114. 二叉树展开为链表

#### 题目描述

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 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`。
- 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]
```

#### 示例 2：

```
输入: root = []
输出: []
```

#### 示例 3：

```
输入: root = [0]
输出: [0]
```

#### 提示：

- 树中结点数在范围  $[0, 2000]$  内
- $-100 \leq \text{Node.val} \leq 100$

进阶：你可以使用原地算法（ $O(1)$  额外空间）展开这棵树吗？

#### 解题思路

##### 方法一：递归（后序遍历思想）

题目的要求是原地将二叉树展开为链表，并且顺序是先序遍历的顺序。我们可以通过递归的方式来解决这个问题。对于一个节点 `root`，我们可以将其左子树和右子树分别展开为链表，然后再进行拼接。

具体的步骤如下：

1. 定义递归函数 `flatten(root)`：该函数的功能是将以 `root` 为根的二叉树原地展开为链表。
2. 递归处理左右子树：首先，递归调用 `flatten(root.left)` 和 `flatten(root.right)`，将左右子树分别展开。
3. 拼接操作（后序遍历位置）：当左右子树都已经被拉平后，我们需要将它们拼接到根节点上。
  - a. 记录下原来的右子树 `right_tree = root.right`。
  - b. 将拉平后的左子树作为新的右子树：`root.right = root.left`，并将左子树置空 `root.left = null`。
  - c. 找到新右子树（即原左子树）的末端节点。
  - d. 将原来的右子树 `right_tree` 接到新右子树的末端。

这个过程利用了后序遍历的思想，因为我们需要先处理完左右子树，才能处理根节点。

## 算法流程：

1. 如果 `root` 为空，直接返回。
2. 递归调用 `flatten(root.left)`。
3. 递归调用 `flatten(root.right)`。
4. 保存 `root` 的右子树：`temp = root.right`。
5. 将 `root` 的左子树移动到右子树的位置：`root.right = root.left`, `root.left = null`。
6. 找到当前右子树的最后一个节点 `p`。
7. 将保存的右子树 `temp` 接到 `p.right`。

## 复杂度分析：

- 时间复杂度：O(N)，其中 N 是二叉树的节点数。我们需要遍历每个节点一次。
- 空间复杂度：O(H)，其中 H 是二叉树的高度。递归栈的空间取决于树的高度，最坏情况下为 O(N)。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def flatten(self, root: Optional[TreeNode]) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        if not root:
            return

        # 递归地将左右子树拉平
        self.flatten(root.left)
        self.flatten(root.right)

        # 后序遍历位置
        # 1. 保存右子树
        temp_right = root.right

        # 2. 将左子树作为右子树
        root.right = root.left
        root.left = None

        # 3. 将原右子树接到新右子树的末端
        p = root
        while p.right:
            p = p.right
        p.right = temp_right
```

## 方法二：寻找前驱节点（原地算法 O(1) 空间）

为了实现 O(1) 的空间复杂度，我们可以避免使用递归带来的栈空间开销。我们可以迭代地处理每个节点，思路如下：

1. 找到 `curr` 左子树的最右侧节点，这个节点是 `curr` 在先序遍历中的前一个节点（前驱节点）。我们称之为 `predecessor`。
2. 将 `curr` 的右子树接到 `predecessor` 的右子树上。
3. 将 `curr` 的左子树移动到 `curr` 的右子树位置。
4. 将 `curr` 的左子树置为 `null`。
5. 然后继续处理下一个节点，即 `curr = curr.right`。

如果当前节点 `curr` 没有左子树，说明它已经是链表的一部分或者它左边没有需要处理的节点了，直接处理下一个节点 `curr = curr.right`。

## 算法流程：

1. 初始化当前节点 `curr = root`。
2. 当 `curr` 不为空时，循环：
  - a. 如果 `curr.left` 不为空：
    - i. 找到 `curr` 左子树的最右节点 `predecessor`。
    - ii. 将 `curr.right` 挂到 `predecessor.right`。
    - iii. 将 `curr.left` 挂到 `curr.right`。
    - iv. 将 `curr.left` 置为 `null`。
  - b. 移动到下一个节点： `curr = curr.right`。

### 复杂度分析：

- 时间复杂度：O(N)。每个节点最多被访问两次（一次是作为 `curr`，一次是作为 `predecessor` 被寻找）。
- 空间复杂度：O(1)。没有使用额外的存储空间。

### Python 代码实现

```

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def flatten(self, root: Optional[TreeNode]) -> None:
        curr = root
        while curr:
            if curr.left:
                # 找到左子树的最右节点
                predecessor = curr.left
                while predecessor.right:
                    predecessor = predecessor.right

                # 将 curr 的右子树接到 predecessor 的右边
                predecessor.right = curr.right

                # 将 curr 的左子树变成右子树
                curr.right = curr.left
                curr.left = None

            # 继续处理下一个节点
            curr = curr.right

```

## 二叉树 / 124. 二叉树中的最大路径和

### 题目描述

**路径** 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 **至多出现一次**。该路径 **至少包含一个** 节点，且不一定经过根节点。

**路径和** 是路径中各节点值的总和。

给你一个二叉树的根节点 `root`，返回其 **最大路径和**。

### 解题思路

这道题的关键在于理解“路径”的定义以及如何通过递归来解决。

1. **路径的特点**：路径可以从任何节点开始，到任何节点结束。这意味着它不一定穿过根节点。路径的形状看起来像一个倒置的 "V"，但也可以是一条直线。
2. **递归函数的角色**：我们需要一个递归函数，对于树中的每一个节点，我们想知道：
  - 以该节点为“拐点”的最大路径和是多少。
  - 该节点能为它的父节点贡献的最大路径和是多少。
3. **核心思想**：我们定义一个递归函数 `dfs(node)`，它的功能是计算从 `node` 出发，向下延伸的**单边最大路径和**。也就是说，它只能走向左子树或者右子树，不能同时走。

对于任意一个节点 `node`，我们可以进行如下分析：

- **计算左、右子树的贡献：**我们递归地调用 `dfs(node.left)` 和 `dfs(node.right)` 来获取左右子树能提供的最大单边路径和。如果子树返回的路径和是负数，我们就不再选择它，因为走这条路只会让总和变小，所以我们取 `max(0, dfs(child))`。
- **更新全局最大值：**以 `node` 为“拐点”的路径和是 `node.val + left_gain + right_gain`（其中 `left_gain` 和 `right_gain` 是左右子树处理过的贡献值）。这个值有可能成为全局的最大路径和，所以我们用一个全局变量 `max_sum` 来实时更新我们找到的最大值。
- **确定返回值：**`dfs(node)` 函数需要返回的是 `node` 节点能为**其父节点**贡献的最大路径和。这个路径必须是单边的，所以它等于 `node.val + max(left_gain, right_gain)`。父节点只能选择走左边或者右边下来，不能两边都要。

通过对整棵树进行一次深度优先搜索（DFS），我们就可以保证在遍历每个节点时，都计算了以它为拐点的最大路径和，并更新了全局结果。

## 算法步骤

1. 初始化一个全局变量 `max_sum` 为负无穷大，用来存储最终结果。
2. 实现一个递归函数 `dfs(node)`：
  - **递归基线条件：**如果 `node` 为空，返回 0，因为它对路径和没有任何贡献。
  - **递归调用：**
    - 递归计算左子树的单边最大路径和 `left_gain = max(0, dfs(node.left))`。
    - 递归计算右子树的单边最大路径和 `right_gain = max(0, dfs(node.right))`。
  - **更新结果：**以当前节点为拐点的路径和为 `node.val + left_gain + right_gain`。用它来更新 `max_sum`: `max_sum = max(max_sum, node.val + left_gain + right_gain)`。
  - **返回值：**返回当前节点能向上贡献的单边最大路径和: `node.val + max(left_gain, right_gain)`。
3. 从根节点 `root` 开始调用 `dfs(root)`。
4. 最终返回 `max_sum`。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

import math

class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        # 初始化一个全局变量来存储最大路径和
        # 使用负无穷大是为了处理所有节点值都是负数的情况
        self.max_sum = -math.inf

        def dfs(node: Optional[TreeNode]) -> int:
            """
            递归函数，返回以该节点为端点的单边最大路径和。
            """

            if not node:
                return 0

            # 递归计算左子树提供的最大路径贡献。
            # 如果贡献值为负，则不选择该路径，记为0。
            left_gain = max(0, dfs(node.left))

            # 递归计算右子树提供的最大路径贡献。
            right_gain = max(0, dfs(node.right))

            # 以当前节点为“拐点”的路径和，尝试更新全局最大值。
            # 这个路径是 left -> node -> right
            current_path_sum = node.val + left_gain + right_gain
            self.max_sum = max(self.max_sum, current_path_sum)

            # 返回以当前节点为起点的“单边”最大路径和，用于向父节点贡献。
            # 父节点只能选择走左边或者右边。
            return node.val + max(left_gain, right_gain)

    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        # ...
        pass
```

```

        return node.val + max(left_gain, right_gain)

    # 从根节点开始深度优先搜索
    dfs(root)

    return self.max_sum

```

## 复杂度分析

- 时间复杂度:  $O(N)$ , 其中  $N$  是二叉树中的节点数。我们对每个节点只访问一次。
- 空间复杂度:  $O(H)$ , 其中  $H$  是二叉树的高度。这是递归调用栈所占用的空间。在最坏的情况下 (树退化成链表),  $H$  的值可能为  $N$ , 即  $O(N)$ ; 在最好的情况下 (完全平衡二叉树),  $H$  的值为  $\log N$ , 即  $O(\log N)$ 。

## 二叉树 / 199. 二叉树的右视图

### 题目描述

给定一个二叉树的根节点 `root`, 想象自己站在它的右侧, 按照从顶部到底部的顺序, 返回从右侧所能看到的节点值。

示例:

- 输入:** [1,2,3,null,5,null,4]
- 输出:** [1, 3, 4]
- 输入:** [1,null,3]
- 输出:** [1, 3]
- 输入:** []
- 输出:** []

### 解题思路

这道题的核心是找到每一层的最右边的那个节点。通常有两种经典的方法可以解决: 广度优先搜索 (BFS) 和深度优先搜索 (DFS)。

#### 方法一：广度优先搜索 (BFS)

广度优先搜索 (层序遍历) 非常直观地符合题意。我们可以对二叉树进行层序遍历, 并只保留每一层的最后一个节点的值。

算法步骤:

- 如果根节点为空, 直接返回空列表。
- 创建一个队列 `queue`, 并将根节点 `root` 入队。
- 创建一个列表 `result` 用于存储最终结果。
- 当队列不为空时, 循环执行以下操作:
  - 记录当前层的节点数量 `level_size`。
  - 遍历当前层的所有节点 (循环 `level_size` 次)。
  - 从队列中取出一个节点 `node`。
  - 关键点:** 如果这个节点是当前层的最后一个节点 (即循环的最后一次), 就将它的值加入 `result` 列表。
  - 如果 `node` 的左子节点不为空, 将其入队。
  - 如果 `node` 的右子节点不为空, 将其入队。
- 循环结束后, 返回 `result` 列表。

```

import collections
from typing import List, Optional

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:

```

```

        return []

result = []
queue = collections.deque([root])

while queue:
    level_size = len(queue)
    for i in range(level_size):
        node = queue.popleft()

        # 如果是当前层的最后一个节点，则加入结果列表
        if i == level_size - 1:
            result.append(node.val)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

return result

```

#### 复杂度分析:

- 时间复杂度:  $O(N)$ , 其中  $N$  是二叉树中节点的数量。每个节点都只会被访问一次。
- 空间复杂度:  $O(W)$ , 其中  $W$  是二叉树的最大宽度。在最坏的情况下 (完全二叉树), 队列中最多可以存储大约  $N/2$  个节点, 因此空间复杂度为  $O(N)$ 。

## 方法二：深度优先搜索 (DFS)

我们也可以使用深度优先搜索来解决这个问题。DFS 的关键在于, 我们通过记录当前的深度, 来判断是否到达了一个新的层级。

#### 算法思想:

我们进行一次 DFS 遍历 (推荐 根-右-左 的顺序)。在遍历时, 我们传递当前的深度 `depth`。我们用结果列表 `result` 的长度来表示我们已经记录过的最大深度。

当 `depth == len(result)` 时, 说明我们是 第一次 到达这个深度。由于我们是先遍历右子树, 所以第一次到达该深度的节点必然是这一层最右边的节点。此时, 我们将该节点的值加入 `result`。

#### 算法步骤:

1. 如果根节点为空, 返回空列表。
2. 创建一个列表 `result` 用于存储结果。
3. 定义一个递归函数 `dfs(node, depth):`
  - a. 如果 `node` 为空, 直接返回。
  - b. 关键点: 如果 `depth == len(result)`, 说明这是第一次访问该深度, 将 `node.val` 添加到 `result` 中。
  - c. 优先递归右子树: `dfs(node.right, depth + 1)`。
  - d. 然后递归左子树: `dfs(node.left, depth + 1)`。
4. 从 `dfs(root, 0)` 开始调用。
5. 返回 `result`。

```

from typing import List, Optional

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        result = []

```

```

        self._dfs(root, 0, result)
        return result

    def _dfs(self, node: Optional[TreeNode], depth: int, result: List[int]):
        if not node:
            return

        # 如果当前深度等于结果列表的长度，说明这是该层第一个被访问的节点
        # 由于我们先访问右子树，所以它就是右视图的一部分
        if depth == len(result):
            result.append(node.val)

        # 先递归右子树，确保右边的节点先被访问
        self._dfs(node.right, depth + 1, result)
        self._dfs(node.left, depth + 1, result)

```

#### 复杂度分析:

- 时间复杂度:  $O(N)$ , 其中  $N$  是二叉树中节点的数量。每个节点都只会被访问一次。
- 空间复杂度:  $O(H)$ , 其中  $H$  是二叉树的高度。这是递归调用栈所占用的空间。在最坏的情况下(树退化成链表)，高度为  $N$ ，空间复杂度为  $O(N)$ 。

#### 总结

- BFS** 方法更符合直觉，通过层序遍历直接找到每层的最右侧节点。
- DFS** 方法则更为巧妙，通过特定顺序(根-右-左)的遍历和深度的判断，来确保每层第一个遇到的节点就是最右侧的节点。

## 二叉树 / 226. 翻转二叉树

### 226. 翻转二叉树

#### 题目描述

给你一棵二叉树的根节点 `root`，翻转这棵二叉树，并返回其根节点。

#### 示例

##### 示例 1:

```

输入: root = [4,2,7,1,3,6,9]
输出: [4,7,2,9,6,3,1]

```

##### 示例 2:

```

输入: root = [2,1,3]
输出: [2,3,1]

```

##### 示例 3:

```

输入: root = []
输出: []

```

#### 提示

- 树中节点数目范围在  $[0, 100]$  内
- $-100 \leq \text{Node.val} \leq 100$

#### 解题思路

##### 方法一：递归

翻转二叉树的过程可以看作是：对于每个节点，交换其左右子树。这个问题可以通过递归来解决。

##### 算法流程：

1. 如果当前节点 `root` 为空，直接返回 `None`。
2. 递归翻转左子树，得到新的右子树。
3. 递归翻转右子树，得到新的左子树。
4. 将当前节点的左右子树指针进行交换。
5. 返回当前节点。

**复杂度分析：**

- **时间复杂度：**  $O(n)$ ，其中  $n$  是二叉树的节点数。每个节点都需要被访问一次。
- **空间复杂度：**  $O(h)$ ，其中  $h$  是二叉树的高度。在最坏情况下（树退化为链表），空间复杂度为  $O(n)$ ；在最好情况下（完全二叉树），空间复杂度为  $O(\log n)$ 。

## 方法二：迭代（使用队列）

我们也可以使用迭代的方式，通过层序遍历（BFS）来翻转二叉树。

**算法流程：**

1. 如果根节点为空，直接返回 `None`。
2. 创建一个队列，将根节点入队。
3. 当队列不为空时：
  - 从队列中取出一个节点。
  - 交换该节点的左右子树。
  - 如果左子节点不为空，将其加入队列。
  - 如果右子节点不为空，将其加入队列。
4. 返回根节点。

**复杂度分析：**

- **时间复杂度：**  $O(n)$ ，其中  $n$  是二叉树的节点数。每个节点都需要被访问一次。
- **空间复杂度：**  $O(w)$ ，其中  $w$  是二叉树的最大宽度。在最坏情况下（完全二叉树的最后一层），空间复杂度为  $O(n/2) = O(n)$ 。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        """
        方法一：递归
        """
        if not root:
            return None

        # 递归翻转左右子树
        right = self.invertTree(root.left)
        left = self.invertTree(root.right)

        # 交换左右子树
        root.left = left
        root.right = right

        return root

    def invertTree_iterative(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        """
        方法二：迭代（使用队列）
        """
        if not root:
            return None
```

```

from collections import deque
queue = deque([root])

while queue:
    node = queue.popleft()
    # 交换左右子树
    node.left, node.right = node.right, node.left

    # 将子节点加入队列
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)

return root

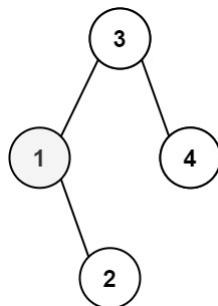
```

## 二叉树 / 230. 二叉搜索树中第K小的元素

### 题目描述

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）。

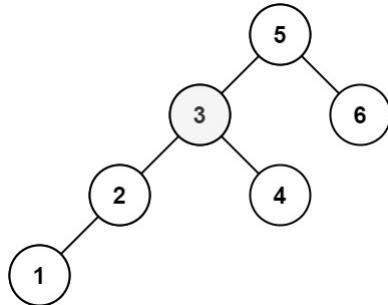
示例 1：



输入：root = [3,1,4,null,2], k = 1

输出：1

示例 2：



输入：root = [5,3,6,2,4,null,null,1], k = 3

输出：3

**提示：**

- 树中的节点数为 n。
- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq 10^4$

**进阶：**如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化算法？

**方法一：中序遍历（递归）**

最直观的思路是，对二叉搜索树进行一次完整的中序遍历，将所有节点值存入一个列表。由于中序遍历的性质，这个列表将是严格递增的。之后，我们直接返回列表的第  $k-1$  个元素（因为索引从0开始）即可。

**Python 代码实现:**

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        # 用于存储中序遍历结果的列表
        res = []

        def inorder(node):
            if not node:
                return
            # 1. 遍历左子树
            inorder(node.left)
            # 2. 访问根节点
            res.append(node.val)
            # 3. 遍历右子树
            inorder(node.right)

        # 执行中序遍历
        inorder(root)
        # 返回第 k 小的元素
        return res[k-1]
```

**复杂度分析:**

- 时间复杂度:  $O(N)$ ，其中  $N$  是树中节点的总数。因为需要遍历树中的所有节点。
- 空间复杂度:  $O(N)$ ，需要一个列表来存储所有节点的值。递归调用也会占用  $O(H)$  的栈空间， $H$  是树的高度。

## 方法二：中序遍历（迭代优化）

方法一虽然简单，但无论  $k$  多小，都需要遍历整棵树，并且需要  $O(N)$  的额外空间。我们可以对此进行优化。

思路是使用迭代的方式进行中序遍历，并用一个计数器来记录访问过的节点数。当我们访问到第  $k$  个节点时，就可以提前终止遍历并返回结果。这种方法不需要存储所有节点，从而节省了空间，并且在  $k$  较小时能节省时间。

**算法步骤:**

1. 初始化一个空栈 `stack`。
2. 当 `root` 不为空或 `stack` 不为空时，循环执行：
  - a. 不断将当前节点 `root` 及其所有左子节点压入栈中，直到 `root` 为空。
  - b. 从栈中弹出一个节点，这个节点就是当前中序遍历访问到的节点。
  - c. 将  $k$  减 1。
  - d. 如果  $k$  变为 0，说明当前弹出的节点就是第  $k$  小的元素，立即返回它的值。
  - e. 将 `root` 指向当前弹出节点的右子节点，继续下一次循环。

**Python 代码实现:**

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
```

```

class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        stack = []
        while root or stack:
            # 将所有左子节点入栈
            while root:
                stack.append(root)
                root = root.left

            # 弹出节点并访问
            root = stack.pop()
            k -= 1
            if k == 0:
                return root.val

            # 转向右子树
            root = root.right

```

#### 复杂度分析:

- 时间复杂度:  $O(H + k)$ , 其中  $H$  是树的高度。在最坏情况下 (树退化为链表且  $k=N$ ), 复杂度为  $O(N)$ 。但在平均情况下, 尤其是当  $k$  较小时, 此方法效率更高。
- 空间复杂度:  $O(H)$ , 主要由栈的深度决定, 在最坏情况下为  $O(N)$ , 平均情况下为  $O(\log N)$ 。

#### 进阶问题

如果二叉搜索树经常被修改 (插入/删除), 并且你需要频繁地查找第  $k$  小的值, 你将如何优化?

对于这种场景, 每次都重新遍历是不高效的。更优化的方法是改造树的节点结构, 使其包含额外信息。一种常见的做法是在每个节点中维护一个 `left_count` 字段, 记录其左子树的节点总数。

##### • 查找时:

- 从根节点开始, 比较  $k$  和左子树的节点数 `left_count`。
- 如果  $k == left\_count + 1$ , 则当前节点就是第  $k$  小的元素。
- 如果  $k <= left\_count$ , 则第  $k$  小的元素在左子树中, 继续在左子树中查找第  $k$  小的元素。
- 如果  $k > left\_count + 1$ , 则第  $k$  小的元素在右子树中。我们需要在右子树中查找第  $k - (left\_count + 1)$  小的元素。

##### • 插入/删除时:

在执行插入或删除操作时, 需要沿着路径更新受影响节点的 `left_count` 值。

这种方法将查找操作的时间复杂度优化到了  $O(H)$ , 即树的高度。

## 二叉树 / 236. 二叉树的最近公共祖先

### 236. 二叉树的最近公共祖先

#### 题目描述

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

[百度百科](#)中最近公共祖先的定义为: “对于有根树  $T$  的两个节点  $p, q$ , 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。

```

#### 示例 2:

```

输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

```

### 示例 3：

```
输入: root = [1,2], p = 1, q = 2
输出: 1
```

### 提示：

- 树中节点数目在范围  $[2, 10^5]$  内。
- $-10^9 \leq \text{Node.val} \leq 10^9$
- 所有 `Node.val` 互不相同。
- $p \neq q$
- $p$  和  $q$  均存在于给定的二叉树中。

## 解题思路

### 方法一：递归

这道题的经典解法是使用递归（深度优先搜索）。我们的目标是找到一个节点，它同时是  $p$  和  $q$  的祖先，并且深度最大。我们可以定义一个递归函数 `lowestCommonAncestor(root, p, q)`，它的功能是在以 `root` 为根的树中查找  $p$  和  $q$  的最近公共祖先。

递归的逻辑可以分为以下几种情况：

#### 1. 基本情况（递归终止条件）：

- 如果当前 `root` 为 `null`，说明在这条路径上没有找到  $p$  或  $q$ ，返回 `null`。
- 如果当前 `root` 等于  $p$  或者  $q$ ，说明我们找到了其中一个节点。根据定义，一个节点也可以是它自己的祖先，所以直接返回 `root`。

#### 2. 递归探索左右子树：

- 在左子树中递归查找：`left = lowestCommonAncestor(root.left, p, q)`。
- 在右子树中递归查找：`right = lowestCommonAncestor(root.right, p, q)`。

#### 3. 根据左右子树的返回结果判断：

- 如果 `left` 和 `right` 都非空，这意味着  $p$  和  $q$  分别位于 `root` 的左右两侧。因此，`root` 就是它们的最近公共祖先，返回 `root`。
- 如果 `left` 为空，`right` 非空，说明  $p$  和  $q$  都不在左子树中，那么它们的最近公共祖先就在右子树中，返回 `right`。
- 如果 `right` 为空，`left` 非空，说明  $p$  和  $q$  都不在右子树中，那么它们的最近公共祖先就在左子树中，返回 `left`。
- 如果 `left` 和 `right` 都为空，说明  $p$  和  $q$  在以 `root` 为根的子树中均未找到，返回 `null`。

这个递归过程自底向上地返回信息，最终在最近公共祖先节点处，其左右子树的递归调用会分别返回  $p$  和  $q$ （或包含它们的子树的根），从而确定该节点为LCA。

### 算法流程：

1. 定义递归函数 `lowestCommonAncestor(root, p, q)`。
2. 如果 `root` 是 `null`、`p` 或 `q`，则返回 `root`。
3. 递归调用 `left = lowestCommonAncestor(root.left, p, q)`。
4. 递归调用 `right = lowestCommonAncestor(root.right, p, q)`。
5. 如果 `left` 和 `right` 都存在，返回 `root`。
6. 否则，返回 `left` 或 `right` 中不为空的那个（如果都为空则返回 `null`）。

### 复杂度分析：

- 时间复杂度：O(N)，其中 N 是二叉树的节点数。我们需要遍历所有节点一次。
- 空间复杂度：O(H)，其中 H 是二叉树的高度。递归调用栈的深度取决于树的高度，最坏情况下（链状树）为 O(N)。

### Python 代码实现

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

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        # 递归终止条件
        if not root or root == p or root == q:
            return root

        # 在左子树中查找 p 和 q
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        # 如果左右子树都不为空，说明当前节点是最近公共祖先
        if left and right:
            return root
        else:
            # 返回不为空的子树，或者如果两者都为空，则返回 None
            return left if left else right
```

```

left = self.lowestCommonAncestor(root.left, p, q)

# 在右子树中查找 p 和 q
right = self.lowestCommonAncestor(root.right, p, q)

# 如果 p 和 q 分别在左右子树中，则 root 是 LCA
if left and right:
    return root

# 如果 p 和 q 都在左子树中，则 left 是 LCA
# 如果 p 和 q 都在右子树中，则 right 是 LCA
return left if left else right

```

## 二叉树 / 297. 二叉树的序列化与反序列化

### 297. 二叉树的序列化与反序列化

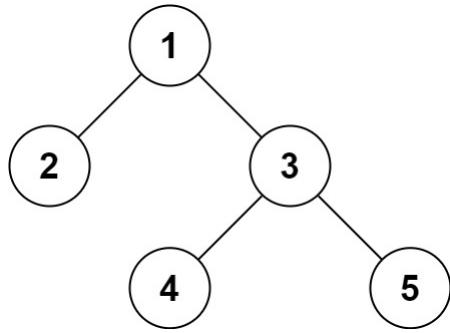
#### 题目描述

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

**提示:** 输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 [LeetCode 序列化二叉树的格式](#)。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

#### 示例 1:



输入: root = [1,2,3,null,null,4,5]

输出: [1,2,3,null,null,4,5]

#### 示例 2:

输入: root = []

输出: []

#### 示例 3:

输入: root = [1]

输出: [1]

#### 示例 4:

输入: root = [1,2]

输出: [1,2]

#### 提示:

- 树中结点数在范围  $[0, 10^4]$  内

```
• -1000 <= Node.val <= 1000
```

## 解题思路

### 方法一：前序遍历 + 递归

#### 思考过程

序列化和反序列化是一对相反的操作：

- **序列化：**将二叉树转换为字符串
- **反序列化：**将字符串转换回二叉树

我们可以使用前序遍历来实现：

1. 序列化时，按照“根-左-右”的顺序遍历，空节点用特殊符号表示
2. 反序列化时，按照相同的顺序重建树

#### 算法流程

##### 序列化 `serialize(root)` :

1. 如果节点为空，返回特殊标记（如 "null"）
2. 否则，返回当前节点值 + 分隔符 + 左子树序列化结果 + 右子树序列化结果

##### 反序列化 `deserialize(data)` :

1. 将字符串按分隔符分割成列表
2. 使用递归函数重建树：
  - 如果当前值是空标记，返回 None
  - 否则，创建节点，递归构建左右子树

#### 复杂度分析

- **时间复杂度：**  $O(N)$ ，其中  $N$  是二叉树的节点数
- **空间复杂度：**  $O(N)$ ，递归调用栈和存储序列化结果的空间

### 方法二：层序遍历 + 队列

#### 思考过程

我们也可以使用层序遍历（BFS）来实现序列化和反序列化：

1. 序列化时，按层遍历，记录每个位置的节点值（包括空节点）
2. 反序列化时，按层重建树

#### 算法流程

##### 序列化：

1. 使用队列进行层序遍历
2. 对每个节点，记录其值，并将其子节点（包括空节点）加入队列

##### 反序列化：

1. 创建根节点
2. 使用队列，按层重建树的结构

#### 复杂度分析

- **时间复杂度：**  $O(N)$
- **空间复杂度：**  $O(N)$

### 方法三：前序遍历 + 迭代

#### 思考过程

我们也可以使用迭代的方式来实现前序遍历的序列化和反序列化。

#### Python 代码实现

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

```

# 方法一：前序遍历 + 递归

class Codec_PreOrder:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        def preorder(node):
            if not node:
                return "null"
            return str(node.val) + "," + preorder(node.left) + "," + preorder(node.right)

        return preorder(root)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        def build_tree():
            val = next(values)
            if val == "null":
                return None

            node = TreeNode(int(val))
            node.left = build_tree()
            node.right = build_tree()
            return node

        values = iter(data.split(","))
        return build_tree()

# 方法二：层序遍历 + 队列
from collections import deque

class Codec_LevelOrder:
    def serialize(self, root):
        if not root:
            return ""

        result = []
        queue = deque([root])

        while queue:
            node = queue.popleft()
            if node:
                result.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)
            else:
                result.append("null")

        return ",".join(result)

    def deserialize(self, data):

```

```

if not data:
    return None

values = data.split(",")
root = TreeNode(int(values[0]))
queue = deque([root])
i = 1

while queue and i < len(values):
    node = queue.popleft()

    # 处理左子节点
    if values[i] != "null":
        node.left = TreeNode(int(values[i]))
        queue.append(node.left)
    i += 1

    # 处理右子节点
    if i < len(values) and values[i] != "null":
        node.right = TreeNode(int(values[i]))
        queue.append(node.right)
    i += 1

return root

# 方法三: 前序遍历 + 迭代
class Codec_Iterative:
    def serialize(self, root):
        if not root:
            return ""

        result = []
        stack = [root]

        while stack:
            node = stack.pop()
            if node:
                result.append(str(node.val))
                stack.append(node.right)  # 先压入右子节点
                stack.append(node.left)   # 再压入左子节点
            else:
                result.append("null")

        return ",".join(result)

    def deserialize(self, data):
        if not data:
            return None

        values = data.split(",")
        root = TreeNode(int(values[0]))
        stack = [root]
        i = 1

        while stack and i < len(values):
            node = stack.pop()

            # 处理左子节点

```

```

        if values[i] != "null":
            node.left = TreeNode(int(values[i]))
            stack.append(node.left)
            i += 1

        # 处理右子节点
        if i < len(values) and values[i] != "null":
            node.right = TreeNode(int(values[i]))
            stack.append(node.right)
            i += 1

    return root

# 推荐解法: 前序遍历 + 递归 (最简洁)
class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string."""
        def dfs(node):
            if not node:
                vals.append("null")
            else:
                vals.append(str(node.val))
                dfs(node.left)
                dfs(node.right)

        vals = []
        dfs(root)
        return ",".join(vals)

    def deserialize(self, data):
        """Decodes your encoded data to tree."""
        def dfs():
            val = next(vals)
            if val == "null":
                return None
            node = TreeNode(int(val))
            node.left = dfs()
            node.right = dfs()
            return node

        vals = iter(data.split(","))
        return dfs()

# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))

```

## 二叉树 / 437. 路径总和III

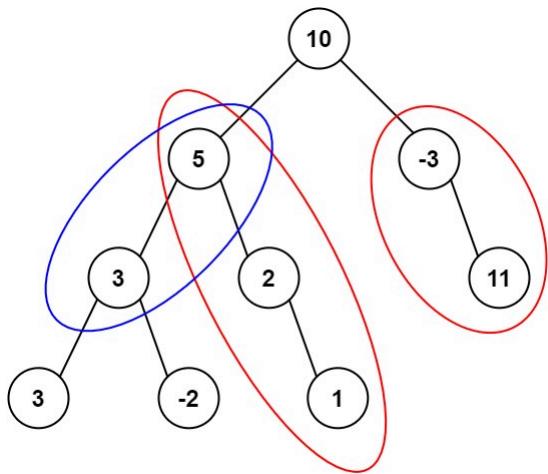
### 437. 路径总和 III

#### 题目描述

给定一个二叉树的根节点 `root`，和一个整数 `targetSum`，求该二叉树里节点值之和等于 `targetSum` 的路径的数目。

路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

示例 1：



输入: `root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8`

输出: 3

解释: 和等于 8 的路径有 3 条, 如图所示。

#### 示例 2:

输入: `root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22`

输出: 3

#### 提示:

- 二叉树的节点个数的范围是 `[0,1000]`
- `-10^9 <= Node.val <= 10^9`
- `-1000 <= targetSum <= 1000`

#### 解题思路

##### 方法一：双重递归（暴力解法）

##### 思考过程

最直观的思路是对每个节点都尝试以它为起点的所有路径，看有多少条路径的和等于目标值。

具体来说：

1. 对于每个节点，计算以该节点为起点的路径数
2. 递归处理左子树和右子树

##### 算法流程

1. 主函数 `pathSum(root, targetSum)`：
  - 如果根节点为空，返回 0
  - 计算以当前节点为起点的路径数: `dfs(root, targetSum)`
  - 递归计算左子树的路径数: `pathSum(root.left, targetSum)`
  - 递归计算右子树的路径数: `pathSum(root.right, targetSum)`
  - 返回三者之和
2. 辅助函数 `dfs(node, targetSum)`：
  - 如果节点为空，返回 0
  - 如果当前节点值等于目标值，计数加1
  - 递归计算以左子节点为下一个节点的路径数: `dfs(node.left, targetSum - node.val)`
  - 递归计算以右子节点为下一个节点的路径数: `dfs(node.right, targetSum - node.val)`
  - 返回总计数

##### 复杂度分析

- 时间复杂度:  $O(N^2)$ , 其中  $N$  是二叉树的节点数。对于每个节点，都要进行一次 DFS，最坏情况下每次 DFS 需要遍历  $N$  个节点。
- 空间复杂度:  $O(H)$ , 其中  $H$  是二叉树的高度。递归调用栈的深度取决于树的高度。

##### 方法二：前缀和 + 哈希表（优化解法）

##### 思考过程

方法一的时间复杂度较高，我们可以使用前缀和的思想来优化。

关键观察：如果我们知道从根节点到当前节点的路径和为 `currSum`，那么以当前节点结尾且和为 `targetSum` 的路径数，就等于路径上前缀和为 `currSum - targetSum` 的节点个数。

我们可以用哈希表记录从根节点到当前路径上每个前缀和出现的次数。

## 算法流程

### 1. 初始化：

- 创建哈希表 `prefixSum`，记录前缀和的出现次数
- 初始化 `prefixSum[0] = 1`（表示空路径的前缀和为0）

### 2. DFS遍历：

- 更新当前路径和：`currSum += node.val`
- 查找 `currSum - targetSum` 在哈希表中的出现次数，累加到结果中
- 将当前前缀和加入哈希表：`prefixSum[currSum] += 1`
- 递归处理左右子树
- 回溯：将当前前缀和从哈希表中移除：`prefixSum[currSum] -= 1`

## 复杂度分析

- 时间复杂度：O(N)，其中 N 是二叉树的节点数。每个节点只被访问一次。
- 空间复杂度：O(N)，哈希表最多存储 N 个前缀和，递归调用栈深度为 O(H)。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

# 方法一：双重递归
class Solution_BruteForce:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        if not root:
            return 0

        def dfs(node: Optional[TreeNode], targetSum: int) -> int:
            if not node:
                return 0

            count = 0
            if node.val == targetSum:
                count += 1

            count += dfs(node.left, targetSum - node.val)
            count += dfs(node.right, targetSum - node.val)

            return count

        # 以当前节点为起点的路径数 + 左子树的路径数 + 右子树的路径数
        return dfs(root, targetSum)+self.pathSum(root.left, targetSum)+self.pathSum(root.right, targetSum)

# 方法二：前缀和 + 哈希表
class Solution_PrefixSum:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        def dfs(node: Optional[TreeNode], currSum: int, prefixSum: dict) -> int:
            if not node:
                return 0

            # 更新当前路径和
            currSum += node.val
```

```

currSum += node.val

# 查找满足条件的路径数
count = prefixSum.get(currSum - targetSum, 0)

# 将当前前缀和加入哈希表
prefixSum[currSum] = prefixSum.get(currSum, 0) + 1

# 递归处理左右子树
count += dfs(node.left, currSum, prefixSum)
count += dfs(node.right, currSum, prefixSum)

# 回溯: 移除当前前缀和
prefixSum[currSum] -= 1

return count

# 初始化前缀和哈希表, 空路径的前缀和为0
prefixSum = {0: 1}
return dfs(root, 0, prefixSum)

# 更简洁的写法
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        def dfs(node, curr_sum):
            if not node:
                return 0

            curr_sum += node.val
            count = prefixSum.get(curr_sum - targetSum, 0)

            prefixSum[curr_sum] = prefixSum.get(curr_sum, 0) + 1

            count += dfs(node.left, curr_sum) + dfs(node.right, curr_sum)

            prefixSum[curr_sum] -= 1

            return count

        prefixSum = {0: 1}
        return dfs(root, 0)

```

## 二叉树 / 538. 把二叉搜索树转换为累加树

### 538. 把二叉搜索树转换为累加树

#### 题目描述

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 Sum Tree），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。

提醒一下，二叉搜索树满足下列约束条件：

- 节点的左子树仅包含键 小于 节点键的节点。
- 节点的右子树仅包含键 大于 节点键的节点。
- 左右子树也必须是二叉搜索树。

#### 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]
```

#### 示例 2:

```
输入: root = [0,null,1]
输出: [1,null,1]
```

#### 示例 3:

```
输入: root = [1,0,2]
输出: [3,1,2]
```

#### 示例 4:

```
输入: root = [3,2,4,1]
输出: [7,9,4,10]
```

#### 提示:

- 树中的节点数介于 0 和  $10^4$  之间。
- 每个节点的值介于  $-10^4$  和  $10^4$  之间。
- 树中的所有值互不相同。
- 给定的树为二叉搜索树。

### 解题思路

#### 方法一：反向中序遍历

##### 思考过程

要将BST转换为累加树，每个节点的新值应该是原树中所有大于或等于该节点值的节点值之和。

关键观察：

- 在BST中，中序遍历得到的是递增序列
- 如果我们进行反向中序遍历（右-根-左），得到的就是递减序列
- 在反向中序遍历过程中，我们可以维护一个累加和，每访问一个节点就更新其值

##### 算法流程

- 初始化：**设置一个全局变量 `sum` 来记录累加和
- 反向中序遍历：**
  - 先递归遍历右子树
  - 处理当前节点：
    - 将当前节点值加到累加和中： `sum += node.val`
    - 更新当前节点值为累加和： `node.val = sum`
  - 再递归遍历左子树
- 返回值：**返回修改后的根节点

##### 复杂度分析

- 时间复杂度：  $O(N)$ ，其中  $N$  是二叉树的节点数。每个节点被访问一次。
- 空间复杂度：  $O(H)$ ，其中  $H$  是二叉树的高度。递归调用栈的深度。

### 方法二：迭代实现

##### 思考过程

我们也可以使用迭代的方式来实现反向中序遍历，使用栈来模拟递归过程。

##### 算法流程

- 初始化：**
  - 创建栈 `stack`
  - 设置累加和 `sum = 0`
  - 设置当前节点指针 `curr = root`

## 2. 迭代过程：

- 当 `curr` 不为空或栈不为空时：
  - 先将所有右子节点入栈
  - 处理当前节点（更新值）
  - 移动到左子节点

3. 返回值：返回修改后的根节点

## 复杂度分析

- 时间复杂度： $O(N)$
- 空间复杂度： $O(H)$ ，栈的最大深度

## 方法三：Morris遍历（空间优化）

### 思考过程

Morris遍历可以在 $O(1)$ 空间复杂度下完成树的遍历，通过临时修改树的结构来实现。

### 算法流程

1. 初始化：设置累加和 `sum = 0`，当前节点 `curr = root`

2. Morris遍历：

- 当 `curr` 不为空时：
  - 如果右子树为空，处理当前节点，移动到左子节点
  - 如果右子树不为空，找到右子树的最左节点作为前驱
    - 如果前驱的左指针为空，建立连接，移动到右子节点
    - 如果前驱的左指针指向当前节点，断开连接，处理当前节点，移动到左子节点

## 复杂度分析

- 时间复杂度： $O(N)$
- 空间复杂度： $O(1)$

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

# 方法一：反向中序遍历（递归）
class Solution_Recursive:
    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        self.sum = 0

        def reverse_inorder(node):
            if not node:
                return

            # 先遍历右子树
            reverse_inorder(node.right)

            # 处理当前节点
            self.sum += node.val
            node.val = self.sum

            # 再遍历左子树
            reverse_inorder(node.left)

        reverse_inorder(root)
        return root

# 方法二：迭代实现
```

```

class Solution_Iterative:
    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return root

        stack = []
        curr = root
        sum_val = 0

        while curr or stack:
            # 先将所有右子节点入栈
            while curr:
                stack.append(curr)
                curr = curr.right

            # 处理当前节点
            curr = stack.pop()
            sum_val += curr.val
            curr.val = sum_val

            # 移动到左子节点
            curr = curr.left

        return root

# 方法三: Morris遍历 (空间优化)
class Solution_Morris:
    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        sum_val = 0
        curr = root

        while curr:
            if not curr.right:
                # 没有右子树, 处理当前节点
                sum_val += curr.val
                curr.val = sum_val
                curr = curr.left
            else:
                # 有右子树, 找到右子树的最左节点
                predecessor = curr.right
                while predecessor.left and predecessor.left != curr:
                    predecessor = predecessor.left

                if not predecessor.left:
                    # 建立连接
                    predecessor.left = curr
                    curr = curr.right
                else:
                    # 断开连接, 处理当前节点
                    predecessor.left = None
                    sum_val += curr.val
                    curr.val = sum_val
                    curr = curr.left

        return root

# 推荐解法: 递归实现 (最简洁)
class Solution:

```

```

def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
    def dfs(node, sum_val):
        if not node:
            return sum_val

        # 先处理右子树，返回累加和
        sum_val = dfs(node.right, sum_val)

        # 处理当前节点
        sum_val += node.val
        node.val = sum_val

        # 处理左子树
        sum_val = dfs(node.left, sum_val)

    return sum_val

dfs(root, 0)
return root

```

## 二叉树 / 543. 二叉树的直径

### 543. 二叉树的直径

#### 题目描述

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，它的长度是路径 [4, 2, 1, 3] 或者 [5, 2, 1, 3]。

注意：两结点之间的路径长度是以它们之间边的数目表示。

#### 解题思路

##### 深度优先搜索（DFS）

##### 思考过程

二叉树的直径，可以理解为树中“最长的一条路径”。这条最长路径不一定经过根节点，它可能完全位于左子树，也可能完全位于右子树，或者是穿过根节点连接左右子树中的两个节点。

对于任意一个节点 `node` 来说，穿过该节点的最长路径长度，等于其左子树的最大深度加上右子树的最大深度。

因此，我们可以遍历树中的每一个节点，以每个节点为“根”（路径的最高点），计算穿过该节点的最长路径，并记录下全局的最大值。这个过程可以通过一次深度优先搜索来完成。

在进行深度优先搜索（计算节点深度的同时），我们可以顺便计算并更新全局的直径最大值。

#### 算法流程

##### 1. 设计一个辅助函数 `depth(node)`：

- 该函数用于计算以 `node` 为根的子树的最大深度。
- 递归终止条件：如果 `node` 为 `null`，返回深度 0。
- 递归过程：
  - 递归计算左子树的最大深度 `L = depth(node.left)`。
  - 递归计算右子树的最大深度 `R = depth(node.right)`。

- **更新直径：**在计算深度的过程中，我们发现穿过 `node` 节点的最长路径长度为 `L + R`。我们用一个全局变量 `self.max_diameter` 来实时更新找到的最大直径，即 `self.max_diameter = max(self.max_diameter, L + R)`。
- **返回值：**函数返回当前节点 `node` 的最大深度，即 `max(L, R) + 1`。这个返回值是给其父节点用来计算深度的。

## 2. 主函数 `diameterOfBinaryTree(root)`：

- 初始化一个全局变量 `self.max_diameter = 0` 用于存储最大直径。
- 调用 `depth(root)` 开始递归计算。
- 最终返回 `self.max_diameter`。

## 复杂度分析

- **时间复杂度：** $O(N)$ ，其中  $N$  是二叉树的节点数。我们对每个节点只访问一次。
- **空间复杂度：** $O(H)$ ，其中  $H$  是二叉树的高度。递归调用栈的深度取决于树的高度。在最坏情况下（树退化为链表），空间复杂度为  $O(N)$ ；在最好情况下（完全二叉树），空间复杂度为  $O(\log N)$ 。

## Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        self.max_diameter = 0

        def depth(node):
            if not node:
                return 0

            # 递归计算左右子树的深度
            left_depth = depth(node.left)
            right_depth = depth(node.right)

            # 更新全局最大直径
            # 直径 = 左子树深度 + 右子树深度
            self.max_diameter = max(self.max_diameter, left_depth + right_depth)

            # 返回当前节点的深度
            return max(left_depth, right_depth) + 1

        depth(root)
        return self.max_diameter
```

## 二叉树 / 617. 合并二叉树

### 617. 合并二叉树

#### 题目描述

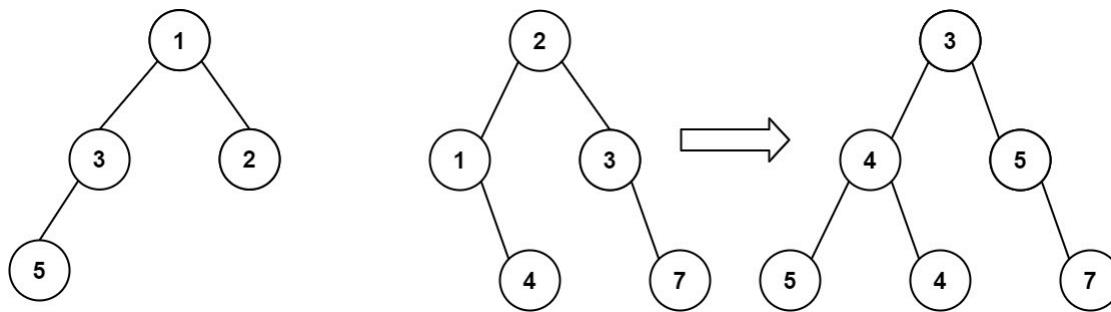
给你两棵二叉树：`root1` 和 `root2`。

想象一下，当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠（而另一些不会）。你需要将这两棵树合并成一棵新树。合并的规则是：如果两个节点重叠，那么将这两个节点的值相加作为合并后节点的新值；否则，不为 `null` 的节点将直接作为新树的节点。

返回合并后的树。

**注意：**合并过程必须从两个树的根节点开始。

**示例 1：**



输入: `root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]`  
 输出: `[3,4,5,5,4,null,7]`

#### 示例 2:

输入: `root1 = [1], root2 = [1,2]`  
 输出: `[2,2]`

#### 提示:

- 两棵树中的节点数目在范围 `[0, 2000]` 内
- `-10^4 <= Node.val <= 10^4`

### 解题思路

#### 方法一：递归（创建新树）

##### 思考过程

最直观的思路是递归地合并两棵树。对于每个位置，我们需要考虑以下几种情况：

- 如果两个节点都存在，创建新节点，值为两个节点值的和
- 如果只有一个节点存在，直接使用该节点
- 如果两个节点都不存在，返回 `null`

##### 算法流程

- 递归终止条件：**
  - 如果 `root1` 为空，返回 `root2`
  - 如果 `root2` 为空，返回 `root1`
- 递归过程：**
  - 创建新节点，值为 `root1.val + root2.val`
  - 递归合并左子树: `mergeTrees(root1.left, root2.left)`
  - 递归合并右子树: `mergeTrees(root1.right, root2.right)`
- 返回值：** 返回新创建的节点

##### 复杂度分析

- 时间复杂度:  $O(\min(m, n))$ ，其中  $m$  和  $n$  分别是两棵树的节点数。我们最多访问较小树的所有节点。
- 空间复杂度:  $O(\min(m, n))$ ，递归调用栈的深度取决于较小树的高度。

#### 方法二：递归（原地修改）

##### 思考过程

我们也可以直接在 `root1` 上进行修改，而不创建新的树，这样可以节省空间。

##### 算法流程

- 递归终止条件：**
  - 如果 `root1` 为空，返回 `root2`
  - 如果 `root2` 为空，返回 `root1`
- 递归过程：**
  - 将 `root2` 的值加到 `root1` 上: `root1.val += root2.val`
  - 递归处理左子树: `root1.left = mergeTrees(root1.left, root2.left)`
  - 递归处理右子树: `root1.right = mergeTrees(root1.right, root2.right)`
- 返回值：** 返回修改后的 `root1`

##### 复杂度分析

- 时间复杂度:  $O(\min(m, n))$ , 其中  $m$  和  $n$  分别是两棵树的节点数。
- 空间复杂度:  $O(\min(m, n))$ , 递归调用栈的深度。

### 方法三：迭代（使用栈）

#### 思考过程

我们也可以使用迭代的方式来实现合并，使用栈来模拟递归过程。

#### 算法流程

- 初始化:**
  - 如果其中一棵树为空，直接返回另一棵树
  - 创建栈，将两个根节点入栈
- 迭代过程:**
  - 当栈不为空时，循环执行：
    - 从栈中取出两个节点  $\text{node}_1$  和  $\text{node}_2$
    - 将  $\text{node}_2$  的值加到  $\text{node}_1$  上
    - 处理左子树和右子树，将需要合并的节点对入栈
- 返回值:** 返回修改后的  $\text{root}_1$

#### 复杂度分析

- 时间复杂度:  $O(\min(m, n))$
- 空间复杂度:  $O(\min(m, n))$ , 栈的最大深度

### Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

# 方法一: 递归 (创建新树)
class Solution_NewTree:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root1:
            return root2
        if not root2:
            return root1

        # 创建新节点
        merged = TreeNode(root1.val + root2.val)
        merged.left = self.mergeTrees(root1.left, root2.left)
        merged.right = self.mergeTrees(root1.right, root2.right)

        return merged

# 方法二: 递归 (原地修改)
class Solution_InPlace:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root1:
            return root2
        if not root2:
            return root1

        # 在 root1 上进行修改
        root1.val += root2.val
        root1.left = self.mergeTrees(root1.left, root2.left)
        root1.right = self.mergeTrees(root1.right, root2.right)

        return root1
```

```

# 方法三: 迭代 (使用栈)
class Solution_Iterative:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root1:
            return root2
        if not root2:
            return root1

        stack = [(root1, root2)]

        while stack:
            node1, node2 = stack.pop()

            if not node1 or not node2:
                continue

            node1.val += node2.val

            # 处理左子树
            if not node1.left:
                node1.left = node2.left
            else:
                stack.append((node1.left, node2.left))

            # 处理右子树
            if not node1.right:
                node1.right = node2.right
            else:
                stack.append((node1.right, node2.right))

        return root1

# 推荐解法: 递归 (原地修改)
class Solution:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root1:
            return root2
        if not root2:
            return root1

        root1.val += root2.val
        root1.left = self.mergeTrees(root1.left, root2.left)
        root1.right = self.mergeTrees(root1.right, root2.right)

        return root1

```

## 二叉树 / 94. 二叉树的中序遍历

### 94. 二叉树的中序遍历

#### 题目描述

给定一个二叉树的根节点 `root`，返回它的 **中序** 遍历。

#### 示例

##### 示例 1:

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

#### 示例 2:

```
输入: root = []
输出: []
```

#### 示例 3:

```
输入: root = [1]
输出: [1]
```

### 提示

- 树中节点数目在范围  $[0, 100]$  内
- $-100 \leq \text{Node.val} \leq 100$

### 解题思路

#### 方法一：递归

二叉树的中序遍历遵循“左-根-右”的顺序。使用递归是实现中序遍历最直观的方法。

##### 算法流程：

1. 定义一个辅助函数 `inorder(node)`，用于递归遍历。
2. 如果当前节点 `node` 为空，则直接返回。
3. 递归调用 `inorder(node.left)` 遍历左子树。
4. 将当前节点的值 `node.val` 添加到结果列表中。
5. 递归调用 `inorder(node.right)` 遍历右子树。

##### 复杂度分析：

- 时间复杂度： $O(n)$ ，其中  $n$  是二叉树的节点数。每个节点都会被访问一次。
- 空间复杂度： $O(n)$ ，在最坏的情况下（例如，树退化成链表），递归调用的深度可以达到  $n$ ，因此需要  $O(n)$  的栈空间。

#### 方法二：迭代 (使用栈)

除了递归，我们还可以使用迭代的方式，借助一个栈来模拟递归的过程。

##### 算法流程：

1. 初始化一个空栈 `stack` 和一个空的结果列表 `result`。
2. 初始化一个指针 `curr` 指向根节点 `root`。
3. 当 `curr` 不为空或 `stack` 不为空时，进行循环：
  - 当 `curr` 不为空时，将 `curr` 压入栈中，然后将 `curr` 更新为其左子节点 (`curr = curr.left`)。
  - 当 `curr` 为空时，说明左子树已经遍历完毕。从栈中弹出一个节点，将其值添加到 `result` 中，然后将 `curr` 更新为该节点的右子节点 (`curr = node.right`)。
4. 循环结束后，`result` 中就保存了中序遍历的结果。

##### 复杂度分析：

- 时间复杂度： $O(n)$ ，其中  $n$  是二叉树的节点数。每个节点都会被压入和弹出栈一次。
- 空间复杂度： $O(n)$ ，在最坏的情况下，栈中最多会存储  $n$  个节点。

### Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
```

```

"""
方法一：递归
"""

result = []
def inorder(node):
    if not node:
        return
    inorder(node.left)
    result.append(node.val)
    inorder(node.right)

inorder(root)
return result

def inorderTraversal_iterative(self, root: Optional[TreeNode]) -> List[int]:
"""
方法二：迭代
"""

result = []
stack = []
curr = root

while curr or stack:
    while curr:
        stack.append(curr)
        curr = curr.left

    curr = stack.pop()
    result.append(curr.val)
    curr = curr.right

return result

```

## 二叉树 / 98. 验证二叉搜索树

### 98. 验证二叉搜索树

#### 题目描述

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。

**有效** 二叉搜索树定义如下：

- 节点的左子树只包含 **小于** 当前节点的数。
- 节点的右子树只包含 **大于** 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

#### 示例 1：

```

输入: root = [2,1,3]
输出: true

```

#### 示例 2：

```

输入: root = [5,1,4,null,null,3,6]
输出: false
解释: 根节点的值是 5，但是右子节点的值是 4。

```

#### 提示：

- 树中节点数目范围在 `[1, 10^4]` 内

- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

## 解题思路

### 方法一：递归

验证一个二叉树是否为二叉搜索树 (BST)，最直观的方法是利用其定义进行递归检查。对于每个节点，我们需要确保其值满足以下条件：

1. 该节点的值大于其左子树中所有节点的值。
2. 该节点的值小于其右子树中所有节点的值。
3. 其左右子树本身也必须是二叉搜索树。

仅仅比较一个节点和它的直接左右孩子是不够的。例如，在 `[5, 1, 4, null, null, 3, 6]` 这个例子中，`5` 的右孩子是 `4`，不满足 BST 定义。但在 `[10, 5, 15, null, null, 6, 20]` 中，虽然 `15 > 10` 且 `6 < 15`，但 `6` 位于 `10` 的右子树中，却小于 `10`，这同样违反了 BST 的性质。`4`

因此，我们需要在递归时传递一个范围 `(min_val, max_val)`，表示当前节点的值必须落在这个开区间内。`3`

#### 算法流程：

1. 定义一个递归函数 `helper(node, lower, upper)`，`lower` 和 `upper` 分别是节点值的下界和上界。
2. 基本情况：如果 `node` 为 `null`，返回 `true`，因为空树是有效的 BST。
3. 合法性检查：检查 `node.val` 是否在 `(lower, upper)` 范围内。如果 `node.val <= lower` 或 `node.val >= upper`，则不是有效的 BST，返回 `false`。
4. 递归调用：
  - 对于左子树，其所有节点的值都必须小于 `node.val`，所以递归调用 `helper(node.left, lower, node.val)`。
  - 对于右子树，其所有节点的值都必须大于 `node.val`，所以递归调用 `helper(node.right, node.val, upper)`。
5. 只有当左右子树的递归调用都返回 `true` 时，才返回 `true`。

初始调用时，范围是 `(-inf, +inf)`。

#### 复杂度分析：

- 时间复杂度： $O(N)$ ，其中  $N$  是树的节点数，因为我们需要访问每个节点一次。
- 空间复杂度： $O(H)$ ，其中  $H$  是树的高度。递归栈的深度取决于树的高度，最坏情况下为  $O(N)$ 。

#### Python 代码实现

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def helper(node, lower=float('-inf'), upper=float('inf')):
            if not node:
                return True

            if not (lower < node.val < upper):
                return False

            # 对于左子树，上界更新为当前节点的值
            # 对于右子树，下界更新为当前节点的值
            return helper(node.left, lower, node.val) and \
                   helper(node.right, node.val, upper)

        return helper(root)
```

### 方法二：中序遍历

二叉搜索树的一个重要性质是，它的中序遍历结果是一个严格递增的序列。`13` 我们可以利用这个性质来验证 BST。

我们可以在中序遍历的过程中，用一个变量 `prev` 记录前一个节点的值。在访问当前节点时，比较当前节点的值是否大于 `prev`。如果不是，则说明序列不是递增的，该树不是 BST。

#### 算法流程（迭代实现）：

1. 初始化一个栈 `stack` 和一个变量 `prev`（用于存储前一个节点的值），`prev` 初始为负无穷大。
2. 使用一个 `while` 循环，条件是 `root` 不为 `null` 或 `stack` 不为空。
3. 将所有左子节点入栈，直到 `root` 为 `null`。
4. 从栈中弹出一个节点，这个节点是当前中序遍历访问到的节点。
5. 比较该节点的值与 `prev`：
  - 如果 `node.val <= prev`，则不是 BST，返回 `false`。
  - 否则，更新 `prev = node.val`。
6. 将 `root` 指向当前节点的右孩子，继续循环。
7. 如果循环结束都没有返回 `false`，则说明是有效的 BST，返回 `true`。

### 复杂度分析：

- 时间复杂度：O(N)，因为每个节点都被访问一次。
- 空间复杂度：O(H)，栈的空间取决于树的高度，最坏情况下为 O(N)。

### Python 代码实现

```

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        stack = []
        prev = float('-inf')
        curr = root

        while curr or stack:
            # 不断将左子节点入栈
            while curr:
                stack.append(curr)
                curr = curr.left

            # 弹出节点，进行比较
            curr = stack.pop()
            if curr.val <= prev:
                return False
            prev = curr.val

            # 转向右子树
            curr = curr.right

        return True

```

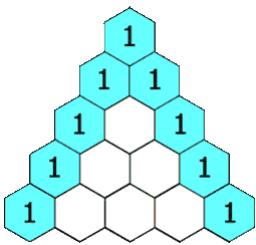
## 动态规划 / 118. 杨辉三角

### 118. 杨辉三角

#### 题目描述

给定一个非负整数 `numRows`，生成「杨辉三角」的前 `numRows` 行。

在「杨辉三角」中，每个数是它左上方和右上方的数的和。



示例 1:

```
输入: numRows = 5
输出: [[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1]]
```

示例 2:

```
输入: numRows = 1
输出: [[1]]
```

提示:

- $1 \leq \text{numRows} \leq 30$

思考过程:

这道题要求生成杨辉三角的前 `numRows` 行。杨辉三角的特点是：

1. 每行的第一个和最后一个数字都是 1。
2. 从第三行开始，每个数字是它正上方和左上方两个数字的和。

核心思路 (动态规划):

我们可以使用动态规划的思想来生成杨辉三角。每一行都依赖于上一行的数据。

设 `triangle[i]` 表示杨辉三角的第 `i` 行。

- `triangle[0] = [1]`
- `triangle[1] = [1, 1]`
- 对于  $i > 1$  的行 `triangle[i]`：
  - `triangle[i]` 的第一个元素是 1。
  - `triangle[i]` 的最后一个元素是 1。
  - 对于中间的元素 `triangle[i][j]`，它等于 `triangle[i-1][j-1] + triangle[i-1][j]`。

具体步骤:

1. 初始化一个空列表 `triangle` 来存储杨辉三角。
2. 如果 `numRows` 为 0，返回空列表。
3. 添加第一行 `[1]` 到 `triangle`。
4. 遍历 `i` 从 1 到 `numRows - 1` (生成第 `i+1` 行)。
  - 创建当前行 `current_row`，并初始化为 `[1]`。
  - 获取上一行 `prev_row = triangle[i-1]`。
  - 遍历 `j` 从 1 到 `i - 1`：
    - `current_row.append(prev_row[j-1] + prev_row[j])`。
    - 将 1 添加到 `current_row` 的末尾。
    - 将 `current_row` 添加到 `triangle`。
5. 返回 `triangle`。

Python 代码:

```
class Solution:
    def generate(self, numRows: int) -> list[list[int]]:
        triangle = []

        if numRows == 0:
            return triangle
```

```

triangle.append([1]) # 第一行

for i in range(1, numRows):
    prev_row = triangle[i-1]
    current_row = [1] # 当前行的第一个元素是1

    for j in range(1, i):
        current_row.append(prev_row[j-1] + prev_row[j])

    current_row.append(1) # 当前行的最后一个元素是1
    triangle.append(current_row)

return triangle

```

## 动态规划 / 139. 单词拆分

### 139. 单词拆分

#### 题目描述

给你一个字符串 s 和一个字符串列表 wordDict 作为字典。请你判断是否可以利用字典中出现的单词拼接出 s。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

#### 示例 1：

```

输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以由 "leet" 和 "code" 拼接成。

```

#### 示例 2：

```

输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以由 "apple" "pen" "apple" 拼接成。
注意，你可以重复使用字典中的单词。

```

#### 示例 3：

```

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

```

#### 提示：

- $1 \leq s.length \leq 300$
- $1 \leq wordDict.length \leq 1000$
- $1 \leq wordDict[i].length \leq 20$
- s 和 wordDict[i] 仅有小写英文字母组成
- wordDict 中的所有字符串 互不相同

#### 思考过程：

这道题要求判断一个字符串 s 是否可以被拆分成一个或多个字典 wordDict 中的单词。

#### 核心思路 (动态规划)：

设  $dp[i]$  表示字符串 s 的前  $i$  个字符  $s[0...i-1]$  是否可以被拆分成字典中的单词。

- **目标:**  $dp[len(s)]$ 。
- **状态转移方程:**
  - $dp[i]$  为 True 的条件是：存在一个  $j < i$ ，使得  $dp[j]$  为 True，并且  $s[j...i-1]$  是字典中的一个单词。
  - $dp[i] = dp[j] \text{ and } s[j...i-1] \text{ in wordDict}.$

### 基本情况:

- `dp[0] = True` (空字符串可以被拆分)。

### 具体步骤:

1. 创建一个布尔数组 `dp`，大小为 `len(s) + 1`，并用 `False` 初始化所有元素，除了 `dp[0] = True`。
2. 将 `wordDict` 转换为 `set`，以便快速查找。
3. 遍历 `i` 从 1 到 `len(s)` (表示字符串的长度)。
4. 遍历 `j` 从 0 到 `i - 1` (表示分割点)。
5. 如果 `dp[j]` 为 `True` 并且 `s[j:i]` 在 `word_set` 中，则设置 `dp[i] = True`，并跳出内层循环 (因为只要找到一种拆分方式即可)。
6. 最后，返回 `dp[len(s)]`。

### Python 代码:

```
class Solution:  
    def wordBreak(self, s: str, wordDict: list[str]) -> bool:  
        n = len(s)  
        dp = [False] * (n + 1)  
        dp[0] = True  
  
        word_set = set(wordDict)  
  
        for i in range(1, n + 1):  
            for j in range(i):  
                if dp[j] and s[j:i] in word_set:  
                    dp[i] = True  
                    break  
  
        return dp[n]
```

## 动态规划 / 152. 乘积最大子数组

### 152. 乘积最大子数组

#### 题目描述

给你一个整数数组 `nums`，请你找出数组中乘积最大的非空连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

测试用例的答案是一个 32-位 整数。

子数组 是数组的连续子序列。

#### 示例 1:

```
输入: nums = [2,3,-2,4]  
输出: 6  
解释: 子数组 [2,3] 有最大乘积 6。
```

#### 示例 2:

```
输入: nums = [-2,0,-1]  
输出: 0  
解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。
```

#### 提示:

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-10 \leq \text{nums}[i] \leq 10$
- `nums` 的任何前缀或后缀的乘积都 保证 是一个 32-位 整数

#### 思考过程:

这道题要求找到数组中乘积最大的连续子数组。

## 核心思路 (动态规划):

与“最大子数组和”不同，乘积涉及到负数。两个负数相乘会得到正数，这使得问题变得复杂。因此，我们需要同时维护以当前位置结尾的最大乘积和最小乘积。

设  $\max_{dp}[i]$  表示以  $nums[i]$  结尾的最大乘积。

设  $\min_{dp}[i]$  表示以  $nums[i]$  结尾的最小乘积。

## 状态转移方程:

- $\max_{dp}[i] = \max(nums[i], nums[i] * \max_{dp}[i-1], nums[i] * \min_{dp}[i-1])$
- $\min_{dp}[i] = \min(nums[i], nums[i] * \max_{dp}[i-1], nums[i] * \min_{dp}[i-1])$

## 基本情况:

- $\max_{dp}[0] = nums[0]$
- $\min_{dp}[0] = nums[0]$

## 具体步骤:

1. 初始化  $\max_{so\_far} = nums[0]$  (全局最大乘积)。
2. 初始化  $\text{current\_max} = nums[0]$  (以当前位置结尾的最大乘积)。
3. 初始化  $\text{current\_min} = nums[0]$  (以当前位置结尾的最小乘积)。
4. 遍历  $i$  从 1 到  $\text{len}(nums) - 1$ 。
5. 对于每个  $nums[i]$ ：
  - 由于  $nums[i]$  可能是负数， $\text{current\_max}$  和  $\text{current\_min}$  可能会互换。
  - $\text{temp\_max} = \max(nums[i], nums[i] * \text{current\_max}, nums[i] * \text{current\_min})$
  - $\text{current\_min} = \min(nums[i], nums[i] * \text{current\_max}, nums[i] * \text{current\_min})$
  - $\text{current\_max} = \text{temp\_max}$
  - 更新  $\max_{so\_far} = \max(\max_{so\_far}, \text{current\_max})$ 。
6. 返回  $\max_{so\_far}$ 。

## Python 代码:

```
class Solution:  
    def maxProduct(self, nums: list[int]) -> int:  
        if not nums:  
            return 0  
  
        max_so_far = nums[0]  
        current_max = nums[0]  
        current_min = nums[0]  
  
        for i in range(1, len(nums)):  
            # 如果当前数字是负数，交换 current_max 和 current_min  
            if nums[i] < 0:  
                current_max, current_min = current_min, current_max  
  
            current_max = max(nums[i], nums[i] * current_max)  
            current_min = min(nums[i], nums[i] * current_min)  
  
        max_so_far = max(max_so_far, current_max)  
  
        return max_so_far
```

## 动态规划 / 198. 打家劫舍

### 198. 打家劫舍

#### 题目描述

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

#### 示例 1:

输入: [1,2,3,1]  
输出: 4  
解释: 偷窃 1 号房屋 (金额 = 1) , 然后偷窃 3 号房屋 (金额 = 3)。  
偷窃到的最高金额 = 1 + 3 = 4 。

#### 示例 2:

输入: [2,7,9,3,1]  
输出: 12  
解释: 偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9), 接着偷窃 5 号房屋 (金额 = 1)。  
偷窃到的最高金额 = 2 + 9 + 1 = 12 。

#### 提示:

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 400$

#### 思考过程:

这道题要求计算在一条街上，相邻的房屋不能同时被抢劫的情况下，能够抢劫到的最高金额。

#### 核心思路 (动态规划):

设  $\text{dp}[i]$  表示抢劫到第  $i$  间房屋时能够获得的最大金额。

- 如果抢劫第  $i$  间房屋: 那么就不能抢劫第  $i-1$  间房屋，所以最大金额是  $\text{dp}[i-2] + \text{nums}[i]$ 。
- 如果不抢劫第  $i$  间房屋: 那么最大金额就是  $\text{dp}[i-1]$ 。
- 因此， $\text{dp}[i] = \max(\text{dp}[i-1], \text{dp}[i-2] + \text{nums}[i])$ 。

#### 基本情况:

- $\text{dp}[0] = \text{nums}[0]$  (只有一间房屋，抢劫它)
- $\text{dp}[1] = \max(\text{nums}[0], \text{nums}[1])$  (两间房屋，抢劫金额大的那间)

#### Python 代码:

```
class Solution:  
    def rob(self, nums: list[int]) -> int:  
        n = len(nums)  
        if n == 0:  
            return 0  
        if n == 1:  
            return nums[0]  
  
        dp = [0] * n  
        dp[0] = nums[0]  
        dp[1] = max(nums[0], nums[1])  
  
        for i in range(2, n):  
            dp[i] = max(dp[i-1], dp[i-2] + nums[i])  
  
        return dp[n-1]
```

## 动态规划 / 300. 最长递增子序列

### 300. 最长递增子序列

#### 题目描述

给你一个整数数组  $\text{nums}$ ，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，删除（或不删除）数组中的元素而不改变其余元素的顺序。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, 18]，因此长度为 4。

#### 示例 2:

输入: nums = [0, 1, 0, 3, 2, 3]  
输出: 4

#### 示例 3:

输入: nums = [7, 7, 7, 7, 7, 7, 7]  
输出: 1

#### 提示:

- $1 \leq \text{nums.length} \leq 2500$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

#### 进阶:

- 你能将算法的时间复杂度降低到  $O(n \log(n))$  吗？

#### 思考过程:

这道题要求找到给定无序数组中，最长递增子序列的长度。

#### 核心思路 (动态规划):

设 `dp[i]` 表示以 `nums[i]` 结尾的最长递增子序列的长度。

- **目标:** `max(dp)`。
- **状态转移方程:**
  - 对于每个 `nums[i]`，我们需要向前遍历所有 `j < i`。
  - 如果 `nums[i] > nums[j]`，说明 `nums[i]` 可以接在以 `nums[j]` 结尾的递增子序列后面。
  - `dp[i] = max(dp[i], dp[j] + 1)`。
- **初始化:** `dp` 数组的所有元素都初始化为 1 (因为每个元素本身都可以构成一个长度为 1 的递增子序列)。

#### 具体步骤:

1. 创建一个 `dp` 数组，大小为 `len(nums)`，并用 1 初始化所有元素。
2. 遍历 `i` 从 1 到 `len(nums) - 1`。
3. 对于每个 `i`，遍历 `j` 从 0 到 `i - 1`。
4. 如果 `nums[i] > nums[j]`，则更新 `dp[i] = max(dp[i], dp[j] + 1)`。
5. 最后，返回 `max(dp)`。

#### Python 代码:

```
class Solution:  
    def lengthOfLIS(self, nums: list[int]) -> int:  
        if not nums:  
            return 0  
  
        n = len(nums)  
        dp = [1] * n  
  
        for i in range(n):  
            for j in range(i):  
                if nums[i] > nums[j]:  
                    dp[i] = max(dp[i], dp[j] + 1)  
  
        return max(dp)
```

## 动态规划 / 32. 最长有效括号

### 32. 最长有效括号

#### 题目描述

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例 1：

```
输入: s = "( () "
输出: 2
解释: 最长有效括号子串是 "( () "
```

示例 2：

```
输入: s = ")()())"
输出: 4
解释: 最长有效括号子串是 ")()("
```

示例 3：

```
输入: s = ""
输出: 0
```

提示：

- $0 \leq s.length \leq 3 * 10^4$
- $s[i]$  为 '(' 或 ')'.

思考过程：

这道题要求找到给定只包含 '(' 和 ')' 的字符串中，最长有效（格式正确且连续）括号子串的长度。

核心思路 (动态规划)：

设  $dp[i]$  表示以  $s[i]$  结尾的最长有效括号子串的长度。

- 目标:  $\max(dp)$ 。
- 状态转移方程：
  - 如果  $s[i] == '('$ ，那么  $dp[i] = 0$  (因为以左括号结尾的有效括号子串长度为 0)。
  - 如果  $s[i] == ')'$ ：
    - 情况一:  $s[i-1] == '('$  (即 ...())
      - $dp[i] = dp[i-2] + 2$  (如果  $i-2 \geq 0$ )。如果  $i-2 < 0$ ，则  $dp[i] = 2$ 。
    - 情况二:  $s[i-1] == ')' (即 ...))$ 
      - 如果  $s[i - dp[i-1] - 1] == '('$  (即 ...((...)))
        - $dp[i] = dp[i-1] + 2 + dp[i - dp[i-1] - 2]$  (如果  $i - dp[i-1] - 2 \geq 0$ )。如果  $i - dp[i-1] - 2 < 0$ ，则  $dp[i] = dp[i-1] + 2$ 。

具体步骤：

1. 创建一个  $dp$  数组，大小为  $\text{len}(s)$ ，并用 0 初始化所有元素。
2. 初始化  $\text{max\_len} = 0$ 。
3. 遍历  $i$  从 1 到  $\text{len}(s) - 1$ 。
4. 根据  $s[i]$  和  $s[i-1]$  的情况更新  $dp[i]$ 。
5. 更新  $\text{max\_len} = \max(\text{max\_len}, dp[i])$ 。
6. 返回  $\text{max\_len}$ 。

Python 代码：

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        n = len(s)
        if n < 2:
            return 0
```

```

dp = [0] * n
max_len = 0

for i in range(1, n):
    if s[i] == ')':
        if s[i-1] == '(': # ...
            dp[i] = (dp[i-2] if i-2 >= 0 else 0) + 2
        elif i - dp[i-1] > 0 and s[i - dp[i-1] - 1] == '(': # ...
            dp[i] = dp[i-1] + 2 + (dp[i - dp[i-1] - 2] if i - dp[i-1] - 2 >= 0 else 0)

    max_len = max(max_len, dp[i])

return max_len

```

## 动态规划 / 322. 零钱兑换

### 322. 零钱兑换

#### 题目描述

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

```

#### 示例 2:

```

输入: coins = [2], amount = 3
输出: -1

```

#### 示例 3:

```

输入: coins = [1], amount = 0
输出: 0

```

#### 提示:

- $1 \leq \text{coins.length} \leq 12$
- $1 \leq \text{coins}[i] \leq 2^{31} - 1$
- $0 \leq \text{amount} \leq 10^4$

#### 思考过程:

这道题要求计算凑成总金额 amount 所需的最少硬币个数。给定不同面额的硬币 coins。如果无法凑成，则返回 -1。

#### 核心思路 (动态规划):

设 `dp[i]` 表示凑成金额 `i` 所需的最少硬币个数。

- **目标:** `dp[amount]`。
- **状态转移方程:** `dp[i] = min(dp[i], dp[i - coin] + 1)`，其中 `coin` 是 `coins` 数组中的每个硬币面额。
- **初始化:**
  - `dp[0] = 0` (凑成金额 0 需要 0 个硬币)。
  - `dp[i] = float('inf')` (对于其他金额，初始化为无穷大，表示暂时无法凑成)。

#### 具体步骤:

1. 创建一个 `dp` 数组，大小为 `amount + 1`，并用 `float('inf')` 初始化所有元素，除了 `dp[0]`。

2. 遍历金额  $i$  从 1 到  $amount$ 。
3. 对于每个金额  $i$ ，遍历所有硬币面额  $coin$ ：
  - 如果  $i - coin \geq 0$  (当前金额可以减去硬币面额)，则更新  $dp[i] = \min(dp[i], dp[i - coin] + 1)$ 。
4. 最后，如果  $dp[amount]$  仍然是 `float('inf')`，说明无法凑成，返回 -1；否则返回  $dp[amount]$ 。

**Python 代码:**

```
class Solution:
    def coinChange(self, coins: list[int], amount: int) -> int:
        dp = [float('inf')] * (amount + 1)
        dp[0] = 0

        for i in range(1, amount + 1):
            for coin in coins:
                if i - coin >= 0:
                    dp[i] = min(dp[i], dp[i - coin] + 1)

        return dp[amount] if dp[amount] != float('inf') else -1
```

## 动态规划 / 416. 分割等和子集

### 416. 分割等和子集

#### 题目描述

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] 。
```

#### 示例 2:

```
输入: nums = [1,2,3,5]
输出: false
解释: 数组不能分割成两个元素和相等的子集。
```

#### 提示:

- $1 \leq \text{nums.length} \leq 200$
- $1 \leq \text{nums}[i] \leq 100$
- $\text{nums}$  中的所有元素的和不超过 10000

#### 思考过程:

这道题要求判断一个非空数组是否可以被分割成两个子集，使得这两个子集的元素和相等。

#### 核心思路 (动态规划 - 0/1 背包问题):

如果数组可以被分割成两个和相等的子集，那么这两个子集的和都应该是数组总和的一半。因此，问题可以转化为：是否存在一个子集，其元素和等于  $\text{sum}(\text{nums}) / 2$ 。这是一个经典的 0/1 背包问题。

设  $dp[i]$  表示是否存在一个子集，其元素和为  $i$ 。

- **目标:**  $dp[\text{target\_sum}]$  (其中  $\text{target\_sum} = \text{sum}(\text{nums}) / 2$ )。
- **状态转移方程:**
  - 对于每个数字  $\text{num} \in \text{nums}$ ：
    - 从  $\text{target\_sum}$  倒序遍历到  $\text{num}$ 。
      - $dp[j] = dp[j] \text{ or } dp[j - \text{num}]$  (如果  $dp[j]$  已经是 True，或者  $dp[j - \text{num}]$  是 True，那么  $dp[j]$  就可以是 True)。
  - **初始化:**  $dp[0] = \text{True}$  (和为 0 的子集总是存在的，即空集)。

#### 具体步骤:

1. 计算数组 `nums` 的总和 `total_sum`。
2. 如果 `total_sum` 是奇数，则无法分割成两个等和子集，返回 `False`。
3. 计算目标和 `target_sum = total_sum // 2`。
4. 创建一个布尔数组 `dp`，大小为 `target_sum + 1`，并用 `False` 初始化所有元素，除了 `dp[0] = True`。
5. 遍历 `nums` 中的每个数字 `num`。
6. 对于每个 `num`，从 `target_sum` 倒序遍历到 `num` (倒序是为了避免重复使用同一个数字)。
7. `dp[j] = dp[j] or dp[j - num]`。
8. 最后，返回 `dp[target_sum]`。

**Python 代码:**

```
class Solution:
    def canPartition(self, nums: list[int]) -> bool:
        total_sum = sum(nums)

        if total_sum % 2 != 0:
            return False

        target_sum = total_sum // 2

        dp = [False] * (target_sum + 1)
        dp[0] = True

        for num in nums:
            # 从 target_sum 倒序遍历到 num
            for j in range(target_sum, num - 1, -1):
                dp[j] = dp[j] or dp[j - num]

        return dp[target_sum]
```

## 动态规划 / 53. 最大子数组和

### 53. 最大子数组和

#### 题目描述

给你一个整数数组 `nums`，请你找出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。

**子数组** 是数组中的一个连续部分。

**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。
```

**示例 2:**

```
输入: nums = [1]
输出: 1
```

**示例 3:**

```
输入: nums = [5,4,-1,7,8]
输出: 23
```

**提示:**

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

## 解题引导

### 思路：动态规划

思考过程：

这个问题是经典的动态规划问题。我们希望找到一个连续子数组，使其和最大。

状态定义：

`dp[i]`：表示以 `nums[i]` 结尾的连续子数组的最大和。

状态转移方程：

对于 `dp[i]`，我们有两种选择：

1. 将 `nums[i]` 独立成一个子数组，那么最大和就是 `nums[i]`。
2. 将 `nums[i]` 加入到以 `nums[i-1]` 结尾的子数组中，形成一个更长的子数组。这个子数组的和是 `dp[i-1] + nums[i]`。

我们需要在这两种选择中取一个较大值。所以状态转移方程是：

```
dp[i] = max(nums[i], dp[i-1] + nums[i])
```

这个方程的含义是：以 `nums[i]` 结尾的最大子数组和，要么是 `nums[i]` 本身，要么是 `nums[i]` 加上之前以 `nums[i-1]` 结尾的最大子数组和。如果 `dp[i-1]` 是负数，那么 `dp[i-1] + nums[i]` 就会比 `nums[i]` 小，我们宁愿从 `nums[i]` 开始一个新的子数组。

最终结果：

`dp` 数组中的每一个值都代表一个以某个元素结尾的局部最优解。而全局的最大子数组可能在任何位置结束，所以我们需要返回整个 `dp` 数组中的最大值。

空间优化：

观察状态转移方程 `dp[i] = max(nums[i], dp[i-1] + nums[i])`，我们发现 `dp[i]` 的计算只依赖于 `dp[i-1]`。因此，我们不需要一个完整的 `dp` 数组，只需要一个变量来记录前一个状态即可。

算法流程（空间优化后）：

1. 初始化两个变量：
  - `current_max`：相当于 `dp[i]`，记录以当前元素结尾的最大子数组和。初始值为 `nums[0]`。
  - `global_max`：记录全局的最大子数组和。初始值为 `nums[0]`。
2. 从数组的第二个元素开始遍历（`i` 从 1 到 `n-1`）：
  - 更新 `current_max`：`current_max = max(nums[i], current_max + nums[i])`。
  - 更新 `global_max`：`global_max = max(global_max, current_max)`。
3. 遍历结束后，返回 `global_max`。

复杂度分析：

- 时间复杂度： $O(n)$ 。我们只遍历数组一次。
- 空间复杂度： $O(1)$ 。经过优化后，我们只需要常数级别的额外空间。

## Python 代码实现

```
from typing import List

class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0:
            return 0

        # current_max 相当于 dp[i]，表示以当前元素结尾的最大子数组和
        # global_max 是全局的最大子数组和
        current_max = nums[0]
        global_max = nums[0]

        for i in range(1, n):
            # 状态转移：选择是连接前面的子数组，还是从当前元素开始新子数组
            current_max = max(nums[i], current_max + nums[i])
            # 更新全局最大值
            if current_max > global_max:
```

```
    global_max = current_max

    return global_max
```

## 动态规划 / 70. 爬楼梯

### 70. 爬楼梯

#### 题目描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

#### 示例 1：

```
输入: n = 2
输出: 2
解释: 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

#### 示例 2：

```
输入: n = 3
输出: 3
解释: 有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

#### 提示：

- $1 \leq n \leq 45$

#### 思考过程：

这道题要求计算爬到楼顶有多少种不同的方法。每次你可以爬 1 或 2 个台阶。

#### 核心思路 (动态规划)：

这是一个经典的动态规划问题。

- 设  $dp[i]$  表示爬到第  $i$  个台阶的不同方法数。
- 要爬到第  $i$  个台阶，可以从第  $i-1$  个台阶爬 1 步，也可以从第  $i-2$  个台阶爬 2 步。
- 因此， $dp[i] = dp[i-1] + dp[i-2]$ 。

#### 基本情况：

- $dp[0] = 1$  (爬到第 0 个台阶，可以看作只有一种方法，即不爬)
- $dp[1] = 1$  (爬到第 1 个台阶，只有一种方法：1 步)
- $dp[2] = 2$  (爬到第 2 个台阶，有两种方法：1+1 步，或 2 步)

#### Python 代码：

```
class Solution:
    def climbStairs(self, n: int) -> int:
        if n == 1:
            return 1

        dp = [0] * (n + 1)
        dp[1] = 1
        dp[2] = 2
```

```

for i in range(3, n + 1):
    dp[i] = dp[i-1] + dp[i-2]

return dp[n]

```

## 双指针 / 11. 盛最多水的容器

### 11. 盛最多水的容器

#### 题目描述

给定一个长度为  $n$  的整数数组  $\text{height}$ 。有  $n$  条垂线，第  $i$  条线的两个端点是  $(i, 0)$  和  $(i, \text{height}[i])$ 。

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。

返回容器可以储存的最大水量。

**说明：**你不能倾斜容器。

#### 示例

示例 1：

输入:  $\text{height} = [1, 8, 6, 2, 5, 4, 8, 3, 7]$

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。

示例 2：

输入:  $\text{height} = [1, 1]$

输出: 1

#### 约束条件

- $n == \text{height.length}$
- $2 \leq n \leq 10^5$
- $0 \leq \text{height}[i] \leq 10^4$

#### 思考过程

##### 思考第一步：理解问题

- 我们要找两条垂线，围成一个容器，使其容纳的水最多。
- 容器的容量由什么决定？**宽度** 和 **高度**。
- **宽度** 是两条线在  $x$  轴上的距离。
- **高度** 是两条线中较短那条的高度，因为水会从短板处溢出。
- 面积 = 宽度  $\times$  高度

##### 思考第二步：暴力解法

最直接的想法是，尝试所有可能的线对组合，计算它们的面积，然后找到最大的那个。

##### 思考题：

- 如果用两层循环来实现，时间复杂度是多少？
- 对于  $n \leq 10^5$  的约束，这个时间复杂度能接受吗？

▶ 点击查看分析

##### 思考第三步：寻找优化思路

暴力解法的问题在于，它做了很多不必要的计算。我们能不能用一种更聪明的方式来移动指针，而不是盲目地遍历所有组合？

想象一下，我们把两个指针分别放在数组的**最左端**和**最右端**。这样，我们就得到了一个初始容器。

- 这个容器的**宽度是最大的**。
- 接下来，我们应该移动哪个指针呢？左指针还是右指针？

##### 思考题：

- 假设  $\text{height}[left] < \text{height}[right]$ ，即左边的板子更短。
- 如果我们移动右指针 ( $right--$ )，会发生什么？
- 如果我们移动左指针 ( $left++$ )，又会发生什么？

▶ 点击查看分析

## 💡 第四步：双指针法

基于上面的分析，我们可以得出双指针算法的流程：

1. 初始化 `left = 0, right = n - 1, max_area = 0`。
2. 当 `left < right` 时，循环执行：
  - a. 计算当前容器的宽度 `width = right - left`。
  - b. 计算当前容器的高度 `h = min(height[left], height[right])`。
  - c. 更新最大面积 `max_area = max(max_area, width * h)`。
  - d. 比较 `height[left]` 和 `height[right]`，移动指向较短板的指针。
3. 循环结束后，`max_area` 就是最终答案。

时间复杂度：O(n) - 因为 `left` 和 `right` 指针总共只会移动 n 次。

空间复杂度：O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
def maxArea(height: list[int]) -> int:  
    """  
    使用双指针计算最大容器面积  
    """  
  
    left, right = 0, len(height) - 1  
    max_area = 0  
  
    while left < right:  
        # 计算当前容器的宽度  
        width = right - left  
  
        # 计算当前容器的高度（由较短的板决定）  
        current_height = min(height[left], height[right])  
  
        # 计算当前面积并更新最大面积  
        current_area = width * current_height  
        max_area = max(max_area, current_area)  
  
        # 移动指向较短板的指针  
        if height[left] < height[right]:  
            left += 1  
        else:  
            right -= 1  
  
    return max_area
```

## 关键点总结

1. 双指针：从两端开始，向中间收敛，这是一种常见的优化技巧。
2. 贪心策略：每次都移动较短的板，因为移动较长的板不可能使面积增大。这是解题的关键。
3. 面积计算：容器的面积由 宽度 和 短板高度 共同决定。

## 双指针 / 15. 三数之和

### 15. 三数之和

#### 题目描述

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k`，同时满足 `nums[i] + nums[j] + nums[k] == 0`。

请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

## 示例

示例 1：

输入: `nums = [-1, 0, 1, 2, -1, -4]`

输出: `[[-1, -1, 2], [-1, 0, 1]]`

示例 2：

输入: `nums = [0, 1, 1]`

输出: `[]`

示例 3：

输入: `nums = [0, 0, 0]`

输出: `[[0, 0, 0]]`

## 约束条件

- $3 \leq \text{nums.length} \leq 3000$
- $-10^5 \leq \text{nums}[i] \leq 10^5$

## 思考过程

### 思考 1：理解问题与暴力解法

- 我们要找到所有独特的三个数，它们的和为 0。
- 最直接的想法是使用三层循环，遍历所有可能的三数组合。

思考题：

- 三层循环的时间复杂度是多少？
- 如何处理找到的三元组的重复问题？例如 `[-1, 0, 1]` 和 `[0, -1, 1]` 是同一个组合。

▶ 点击查看分析

### 思考 2：降维思考

$O(n^3)$  太慢，我们需要优化。能不能把问题简化一下？

如果我们能把问题从“找三个数”降维到“找两个数”，情况就会好很多。

核心思路：先固定一个数，然后问题就变成了在剩下的数中寻找“两数之和”等于某个目标值。

$$a + b + c = 0 \Rightarrow b + c = -a$$

这样，我们可以用一层循环来固定 `a`，然后用一个更高效的方法在剩余部分找到 `b` 和 `c`。

### 思考 3：排序 + 双指针

“两数之和”问题有一个经典的解法：如果数组是有序的，就可以使用双指针。

这给了我们一个清晰的优化路径：

1. **排序**：首先对整个数组 `nums` 进行排序。这不仅能让双指针生效，还极大地简化了后续的去重操作。
2. **主循环**：遍历排序后的数组，用 `i` 来固定第一个数 `nums[i]`。
3. **双指针查找**：在 `i` 后面的部分 `[i+1, n-1]`，设置 `left = i + 1` 和 `right = n - 1`。我们的目标是在这个范围内找到 `nums[left] + nums[right] == -nums[i]`。

思考题：

- 在双指针移动过程中，如果 `nums[left] + nums[right]` 太小了怎么办？如果太大了又怎么办？
- 排序后，如何高效地跳过重复的三元组？

▶ 点击查看分析

### 思考 4：算法步骤总结

1. 对数组 `nums` 进行升序排序。
2. 遍历数组，对于每个元素 `nums[i]`：
  - a. **剪枝优化**：如果 `nums[i] > 0`，因为数组已经排序，后面的数都比它大，三数之和不可能为 0，可以直接结束循环。
  - b. **去重**：如果 `i > 0` 且 `nums[i] == nums[i-1]`，则跳过，避免重复计算。
  - c. 初始化双指针 `left = i + 1, right = len(nums) - 1`。
  - d. 在 `left < right` 的条件下循环：
    - i. 计算三数之和 `s = nums[i] + nums[left] + nums[right]`。
    - ii. 如果 `s < 0`，则 `left++`。

- iii. 如果  $s > 0$ ，则  $right--$ 。
- iv. 如果  $s == 0$ ，则找到了一个解：
- 将  $[nums[i], nums[left], nums[right]]$  添加到结果列表。
- 对  $left$  和  $right$  去重：移动  $left$  和  $right$  指针，跳过所有重复的元素。
- 最后，将  $left$  和  $right$  再各移动一位，开始寻找下一个可能的解。

3. 返回结果列表。

时间复杂度：O( $n^2$ ) - 排序占 O( $n \log n$ )，两层循环占 O( $n^2$ )。

空间复杂度：O(1) 或 O( $\log n$ ) - 取决于排序算法使用的栈空间。

## 代码实现

### Python

```
def threeSum(nums: list[int]) -> list[list[int]]:
    """
    使用“排序 + 双指针”解决三数之和问题
    """

    nums.sort() # 排序是关键
    result = []
    n = len(nums)

    # 外层循环，固定第一个数
    for i in range(n - 2):
        # 剪枝：如果最小的数都大于0，直接结束
        if nums[i] > 0:
            break

        # 去重：跳过重复的第一个数
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        # 双指针在 i 后面的部分寻找另外两个数
        left, right = i + 1, n - 1
        target = -nums[i] # 目标和

        while left < right:
            current_sum = nums[left] + nums[right]

            if current_sum == target:
                # 找到一组解
                result.append([nums[i], nums[left], nums[right]])

            # 去重：跳过所有重复的第二个数
            while left < right and nums[left] == nums[left + 1]:
                left += 1

            # 去重：跳过所有重复的第三个数
            while left < right and nums[right] == nums[right - 1]:
                right -= 1

            # 移动指针，寻找新的可能性
            left += 1
            right -= 1

            elif current_sum < target:
                left += 1 # 和太小，左指针右移
            else:
                right -= 1 # 和太大，右指针左移

    return result
```

## 关键点总结

1. **排序**: 是使用双指针和高效去重的前提。
2. **降维**: 将三数之和问题转化为“固定一数 + 两数之和”问题。
3. **双指针**: 在有序数组中高效查找两数之和。
4. **去重**: 是本题的难点和重点，需要在三个位置（固定的数、左指针、右指针）都进行考虑。

## 双指针 / 283. 移动零

### 283. 移动零

#### 题目描述

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意，必须在不复制数组的情况下原地对数组进行操作。

#### 示例

示例 1：

输入: `nums = [0,1,0,3,12]`

输出: `[1,3,12,0,0]`

示例 2：

输入: `nums = [0]`

输出: `[0]`

#### 约束条件

- $1 \leq \text{nums.length} \leq 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

#### 思考过程

##### 💡 第一步：理解问题

让我们先分析一下这个问题：

- 需要将所有的 `0` 移动到数组末尾
- 保持非零元素的相对顺序不变
- 必须原地操作，不能使用额外的数组

思考题：如果允许使用额外空间，你会怎么做？

▶ 点击查看提示

##### 💡 第二步：寻找模式

让我们手动模拟一下 `[0,1,0,3,12]` 的处理过程：

初始: `[0, 1, 0, 3, 12]`

目标: `[1, 3, 12, 0, 0]`

思考题：你能发现什么规律吗？非零元素是如何移动的？

▶ 点击查看分析

##### 💡 第三步：双指针策略

现在让我们思考双指针的具体实现。一个指针 `right` 用来遍历整个数组，另一个指针 `left` 用来指向“下一个非零元素应该被放置的位置”。

#### 算法思路：

1. 初始化 `left = 0`。
2. 用 `right` 指针从头到尾遍历数组。
3. 如果 `nums[right]` 是一个非零元素，我们就把它放到 `nums[left]` 的位置，然后将 `left` 指针向后移动一位 (`left++`)。
4. 遍历结束后，所有非零元素都被按顺序移动到了数组的前 `left` 个位置。
5. 最后，将数组从 `left` 位置到末尾的所有元素都设置为 `0`。

##### 💡 第四步：手动模拟

让我们用这个思路手动模拟 `[0,1,0,3,12]`：

```

初始状态: [0, 1, 0, 3, 12], left = 0

right = 0, nums[0] is 0. 什么都不做。
[0, 1, 0, 3, 12], left = 0

right = 1, nums[1] is 1 (非零). 将它放到 nums[left] (即 nums[0]) 的位置. left++.
[1, 1, 0, 3, 12], left = 1

right = 2, nums[2] is 0. 什么都不做。
[1, 1, 0, 3, 12], left = 1

right = 3, nums[3] is 3 (非零). 将它放到 nums[left] (即 nums[1]) 的位置. left++.
[1, 3, 0, 3, 12], left = 2

right = 4, nums[4] is 12 (非零). 将它放到 nums[left] (即 nums[2]) 的位置. left++.
[1, 3, 12, 3, 12], left = 3

遍历结束。现在 left = 3。将从索引 3 开始的位置填充为 0。
nums[3] = 0
nums[4] = 0

最终结果: [1, 3, 12, 0, 0]

```

这个方法被称为**快慢指针**, `right` 是快指针, `left` 是慢指针。

## 代码实现

### Python

```

def moveZeroes(nums: list[int]) -> None:
    """
    使用快慢指针原地移动零，同时保持非零元素相对顺序。
    Do not return anything, modify nums in-place instead.
    """
    left = 0 # 慢指针，指向下一个非零元素应该放置的位置

    # 第一次遍历：将所有非零元素移到前面
    for right in range(len(nums)): # 快指针，遍历整个数组
        if nums[right] != 0:
            # 如果快慢指针不指向同一个位置，才进行赋值，避免不必要的操作
            if left != right:
                nums[left] = nums[right]
            left += 1

    # 第二次遍历：将慢指针之后的所有位置填充为0
    while left < len(nums):
        nums[left] = 0
        left += 1

```

## 复杂度分析

- 时间复杂度:  $O(n)$  - 因为我们只对数组进行了常数次遍历。
- 空间复杂度:  $O(1)$  - 我们是在原地操作，只使用了常数个额外变量。

## 关键点总结

- 快慢指针**: 这是一种特殊的双指针，一个指针用于遍历，另一个指针用于记录有效位置，常用于解决原地修改数组的问题。
- 原地操作**: 通过覆盖的方式实现，避免了创建新数组带来的额外空间开销。
- 保持相对顺序**: 由于慢指针 `left` 和快指针 `right` 都是从左到右单向移动，非零元素被处理的顺序与它们在原数组中的顺序一致，因此相对顺序得以保持。

## 双指针 / 42. 接雨水

### 42. 接雨水

#### 题目描述

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

#### 示例

示例 1：

输入: `height = [0,1,0,2,1,0,1,3,2,1,2,1]`

输出: 6

示例 2：

输入: `height = [4,2,0,3,2,5]`

输出: 9

#### 约束条件

- $n == \text{height.length}$
- $1 \leq n \leq 2 * 10^4$
- $0 \leq \text{height}[i] \leq 10^5$

#### 思考过程

##### 💡 第一步：理解问题

- 我们要计算的是凹槽能装多少水。
- 关键在于，对于任何一个位置  $i$ ，它上面能装多少水？
- 这取决于它左边的最高柱子和右边的最高柱子中，较矮的那个。
- $i$  位置的储水量 =  $\min(\text{左边最高点}, \text{右边最高点}) - \text{height}[i]$
- 当然，如果  $\text{height}[i]$  本身就很高，导致这个计算结果是负数，那说明它这里存不了水，储水量为 0。

##### 💡 第二步：暴力解法（按列计算）

基于上面的分析，最直接的方法是：

1. 遍历每一个位置  $i$ （从 1 到  $n-2$ ，因为两端存不了水）。
2. 对于每个  $i$ ，再向左遍历一遍找到  $[0...i]$  的最高点  $\text{left\_max}$ 。
3. 再向右遍历一遍找到  $[i...n-1]$  的最高点  $\text{right\_max}$ 。
4. 计算  $\text{water}_i = \min(\text{left\_max}, \text{right\_max}) - \text{height}[i]$ 。
5. 把所有  $\text{water}_i$  加起来。

思考题：这个方法的时间复杂度是多少？对于  $n \leq 2 * 10^4$  是否可行？

▶ 点击查看分析

##### 💡 第三步：优化（动态规划）

暴力解法的瓶颈在于，对于每个  $i$ ，我们都重复计算了  $\text{left\_max}$  和  $\text{right\_max}$ 。

我们可以用空间换时间，提前把这些信息存起来。

1. 创建一个数组  $\text{left\_max\_arr}$ ， $\text{left\_max\_arr}[i]$  表示  $[0...i]$  区间的最大高度。
2. 创建一个数组  $\text{right\_max\_arr}$ ， $\text{right\_max\_arr}[i]$  表示  $[i...n-1]$  区间的最大高度。
3. 这两个数组都可以通过一次  $O(n)$  的遍历来填充。
4. 最后，再遍历一次，使用预算好的  $\text{left\_max\_arr}[i]$  和  $\text{right\_max\_arr}[i]$  来计算每个位置的储水量并累加。

思考题：这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

##### 💡 第四步：终极优化（双指针）

动态规划方法使用了  $O(n)$  的额外空间。我们能不能把空间复杂度降到  $O(1)$ ？

让我们回到问题的核心： $i$  位置的储水量由  $\min(\text{left\_max}, \text{right\_max})$  决定。

想象一下，我们用两个指针  $\text{left}$  和  $\text{right}$  分别指向数组的两端。

同时，我们维护两个变量  $\text{left\_max}$  和  $\text{right\_max}$ ，分别记录  $[0...left]$  和  $[right...n-1]$  的历史最高高度。

关键问题：在  $\text{left}$  和  $\text{right}$  指针处，我们能确定其中一个位置的储水量吗？

核心洞察：

- 如果 `left_max < right_max`, 那么对于 `left` 指针所指的位置, 它的右边最高点至少是 `right_max`, 而它的左边最高点就是我们已经维护的 `left_max`。
- 因此, `min(左边最高点, 右边最高点)` 就等于 `left_max`。
- 这时, 我们就可以完全确定 `left` 位置的储水量: `left_max - height[left]`。
- 反之, 如果 `right_max <= left_max`, 我们就可以完全确定 `right` 位置的储水量。

算法流程:

- 初始化 `left = 0, right = n - 1, left_max = 0, right_max = 0, total_water = 0`。
- 当 `left < right` 时循环:
  - 比较 `height[left]` 和 `height[right]`。
  - 如果 `height[left] < height[right]`:
    - 更新 `left_max = max(left_max, height[left])`。
    - 计算储水量 `total_water += left_max - height[left]`。
    - `left++`。
  - 否则 (`height[right] <= height[left]`):
    - 更新 `right_max = max(right_max, height[right])`。
    - 计算储水量 `total_water += right_max - height[right]`。
    - `right--`。
- 循环结束, 返回 `total_water`。

时间复杂度:  $O(n)$  - `left` 和 `right` 指针只会相遇一次。

空间复杂度:  $O(1)$  - 只使用了常数个额外变量。

## 代码实现

### Python (双指针最优解)

```
def trap(height: list[int]) -> int:
    """
    使用双指针计算接雨水问题, 空间复杂度为 O(1)
    """

    if not height or len(height) < 3:
        return 0

    left, right = 0, len(height) - 1
    left_max, right_max = 0, 0
    total_water = 0

    while left < right:
        # 核心: 对于 left 和 right 两个指针, 总是处理高度较小的那一侧
        if height[left] < height[right]:
            # 此处, 我们能确定 left 位置的储水量
            # 因为右侧的屏障至少是 height[right], 而左侧的屏障是 left_max
            # 储水量取决于较矮的屏障, 即 left_max
            if height[left] >= left_max:
                # 当前柱子是新的左侧最高点, 无法储水
                left_max = height[left]
            else:
                # 可以储水, 水量为 left_max - 当前高度
                total_water += left_max - height[left]
            left += 1
        else:
            # 此处, 我们能确定 right 位置的储水量
            # 因为左侧的屏障至少是 height[left], 而右侧的屏障是 right_max
            # 储水量取决于较矮的屏障, 即 right_max
            if height[right] >= right_max:
                # 当前柱子是新的右侧最高点, 无法储水
                right_max = height[right]
            else:
                # 可以储水, 水量为 right_max - 当前高度
                total_water += right_max - height[right]
            right -= 1
```

```
return total_water
```

## 关键点总结

- 按列计算：理解每个位置的储水量由其左右两边的最高柱子中的较矮者决定，是解决问题的基础。
- 双指针优化：通过从两端向中间收敛，并维护左右两个 `max` 值，巧妙地将空间复杂度从  $O(n)$  降为  $O(1)$ 。
- 决策依据：双指针算法的核心在于，我们总是可以对 `left_max` 和 `right_max` 中较小的一方做出关于储水量的确定性计算。

## 哈希 / 1. 两数之和

### 1. 两数之和

#### 题目描述

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出和为目标值 `target` 的那两个整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

#### 示例 1：

```
输入: nums = [2,7,11,15], target = 9
输出: [0,1]
解释: 因为 nums[0] + nums[1] == 9，返回 [0, 1] 。
```

#### 示例 2：

```
输入: nums = [3,2,4], target = 6
输出: [1,2]
```

#### 示例 3：

```
输入: nums = [3,3], target = 6
输出: [0,1]
```

#### 提示：

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- 只会存在一个有效答案

进阶：你可以想出一个时间复杂度小于  $O(n^2)$  的算法吗？

## 解题引导

### 思路一：暴力解法

#### 思考过程：

最容易想到的方法是什么？题目要求我们找到两个数，它们的和是 `target`。那我们就干脆遍历数组，对于每一个数，再遍历它后面的所有数，看看两者的和是不是 `target`。

- 用第一层循环，从第一个元素开始，依次选中数组中的每个元素 `nums[i]`。
- 用第二层循环，从 `i + 1` 开始，依次选中 `nums[i]` 后面的每个元素 `nums[j]`。
- 在第二层循环中，判断 `nums[i] + nums[j]` 是否等于 `target`。
- 如果等于，那么 `i` 和 `j` 就是我们要找的下标，直接返回 `[i, j]`。

#### 复杂度分析：

- 时间复杂度： $O(n^2)$ 。因为我们用了两层嵌套循环，外层循环  $n$  次，内层循环平均  $n/2$  次。
- 空间复杂度： $O(1)$ 。我们只使用了常数级别的额外空间。

这种方法虽然简单，但是效率不高。题目的“进阶”提示我们，有更好的方法。

## 思路二：哈希表优化

### 思考过程：

暴力解法的瓶颈在哪里？在于“寻找 `target - nums[i]`”这一步，我们是通过又一次遍历来实现的。有没有更快的方法来寻找一个数是否存在于数组中呢？

答案是 **哈希表**（在 Python 中是字典 `dict`）。哈希表提供了近乎  $O(1)$  时间复杂度的查找、插入和删除操作。

我们可以这样做：

1. 创建一个空的哈希表 `hash_map`，用来存放我们已经遍历过的数字和它们的下标。格式为 `(数字: 下标)`。
2. 遍历数组 `nums` 中的每一个数 `num`（同时获取它的下标 `i`）。
3. 对于每个 `num`，我们计算需要配对的另一个数 `complement = target - num`。
4. 然后，我们在哈希表 `hash_map` 中查找 `complement` 是否存在。
  - 如果存在，说明我们找到了配对的两个数！`complement` 的下标已经存在哈希表里了，当前数 `num` 的下标是 `i`。我们直接返回 `[hash_map[complement], i]`。
  - 如果不存在，说明到目前为止，还没有数字能和当前的 `num` 配对。我们把当前的 `num` 和它的下标 `i` 存入哈希表 `hash_map` 中，即 `hash_map[num] = i`，以便后续的数字来和它配对。

### 为什么这样可行？

当我们遍历到 `nums[i]` 时，我们去哈希表里寻找 `target - nums[i]`。如果找到了，说明 `target - nums[i]` 这个数在 `nums[i]` 之前出现过。这样我们就凑成了一对，并且由于我们是从前往后遍历的，所以两个数的下标肯定不同。

### 复杂度分析：

- **时间复杂度：**  $O(n)$ 。我们只需要遍历一次数组。哈希表的查找和插入操作平均时间复杂度都是  $O(1)$ 。
- **空间复杂度：**  $O(n)$ 。在最坏的情况下，我们需要把数组中所有的数都存入哈希表。

## Python 代码实现

```
from typing import List

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        """
        使用哈希表来优化查找过程
        """
        hash_map = {} # 创建一个哈希表（字典）来存储数字及其索引

        # 遍历数组，enumerate可以同时获得索引和值
        for i, num in enumerate(nums):
            # 计算我们需要寻找的另一个数
            complement = target - num

            # 检查 complement 是否已经在哈希表中
            if complement in hash_map:
                # 如果在，说明找到了答案，返回两个数的索引
                return [hash_map[complement], i]

            # 如果 complement 不在哈希表中，将当前的数和它的索引存入哈希表
            # 注意：我们是在检查之后再存入，这样可以避免同一个元素被使用两次
            # 例如 nums = [3, 2, 4], target = 6
            # 遍历到 3 时，寻找 3，没找到，存入 {3: 0}
            # 遍历到 2 时，寻找 4，没找到，存入 {3: 0, 2: 1}
            # 遍历到 4 时，寻找 2，找到了！hash_map[2] 是 1，当前索引是 2，返回 [1, 2]
            hash_map[num] = i

        # 题目保证有解，所以这里实际上不会执行到
        return []
```

# 哈希 / 128. 最长连续序列

## 128. 最长连续序列

### 题目描述

给定一个未排序的整数数组 `nums`，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为  $O(n)$  的算法解决此问题。

示例 1：

```
输入: nums = [100, 4, 200, 1, 3, 2]
输出: 4
解释: 最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

示例 2：

```
输入: nums = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1]
输出: 9
```

提示：

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

### 解题引导

#### 思路：哈希集合（Set）优化

思考过程：

首先，一个直观的想法是先对数组进行排序，然后遍历排序后的数组来寻找最长连续序列。排序的时间复杂度是  $O(n \log n)$ ，不符合题目  $O(n)$  的要求。

我们需要一种更快的方法。 $O(n)$  的时间复杂度通常意味着我们只能遍历数组常数次。这让我们想到了哈希表/哈希集合，因为它们提供了  $O(1)$  的平均查找时间。

我们可以使用一个哈希集合（`set`）来优化查找过程：

1. **去重与存储：**首先，将所有数组元素存入一个哈希集合中。这一步有两个目的：一是方便我们以  $O(1)$  的时间复杂度检查某个数是否存在；二是可以自动处理数组中的重复元素。
2. **遍历与计算：**再次遍历原始数组 `nums` 中的每个数字 `num`。
3. **关键优化点：**对于每个 `num`，我们如何避免重复计算？想象一下序列 `[1, 2, 3, 4]`。当我们遍历到 `2` 时，我们没有必要向上和向下查找，因为当我们遍历到 `1` 时，肯定已经把 `[1, 2, 3, 4]` 这个序列完整地找过了。所以，我们只对一个连续序列的起始点进行查找和扩展。
4. **如何判断一个数是起始点？**如果一个数 `num` 的前一个数 `num - 1` 不存在于哈希集合中，那么 `num` 就可能是一个连续序列的起始点。

5. 算法流程：

- 将 `nums` 存入哈希集合 `num_set`。
- 初始化一个变量 `max_length` 来记录最长序列的长度，初始值为 0。
- 遍历 `nums` 数组中的每个 `num`：
  - 检查 `num - 1` 是否在 `num_set` 中。
  - 如果 `num - 1` 不在 `num_set` 中，说明 `num` 是一个潜在的序列起点。
  - 从 `num` 开始，不断检查 `num + 1, num + 2, ...` 是否在 `num_set` 中，直到找不到为止。记录下这个序列的长度 `current_length`。
  - 更新 `max_length = max(max_length, current_length)`。
- 遍历结束后，`max_length` 就是最终答案。

为什么时间复杂度是  $O(n)$ ？

看起来我们有两层循环，但实际上，内层的 `while` 循环对于每个数字只会被执行一次。整个数组中的每个数字最多只会被访问两次（一次是外层循环，一次是作为某个序列的一部分在内层 `while` 循环中被访问）。因此，总的时间复杂度是线性的，即  $O(n)$ 。

复杂度分析：

- **时间复杂度：**  $O(n)$ 。将 `nums` 存入哈希集合是  $O(n)$ 。外层循环是  $O(n)$ 。内层 `while` 循环虽然看起来是嵌套的，但每个元素最多只进入一次，所以总共也是  $O(n)$ 。
- **空间复杂度：**  $O(n)$ 。我们需要一个哈希集合来存储所有不重复的元素。

### Python 代码实现

```

from typing import List

class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        """
        使用哈希集合优化查找，只对序列的起点进行扩展
        """
        if not nums:
            return 0

        # 1. 将所有数字存入哈希集合，O(n) 时间，O(n) 空间
        num_set = set(nums)
        max_length = 0

        # 2. 遍历哈希集合中的每个数字
        for num in num_set:
            # 3. 关键点：只对序列的第一个数（即 num-1 不在集合中）进行处理
            if num - 1 not in num_set:
                current_num = num
                current_length = 1

            # 4. 从当前数开始，向后查找连续的数
            while current_num + 1 in num_set:
                current_num += 1
                current_length += 1

            # 5. 更新最大长度
            max_length = max(max_length, current_length)

        return max_length

```

## 哈希 / 49. 字母异位词分组

### 49. 字母异位词分组

#### 题目描述

给你一个字符串数组，请你将 **字母异位词** 组合在一起。可以按任意顺序返回结果列表。

**字母异位词** 是由相同字母按不同顺序排列形成的字符串。

#### 示例 1:

```

输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
输出: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]

```

#### 示例 2:

```

输入: strs = []
输出: [[]]

```

#### 示例 3:

```

输入: strs = ["a"]
输出: [["a"]]

```

#### 提示:

- $1 \leq \text{strs.length} \leq 10^4$
- $0 \leq \text{strs[i].length} \leq 100$
- $\text{strs[i]}$  仅包含小写字母

## 解题引导

### 思路：哈希表 + 排序

思考过程：

题目的核心是要识别哪些字符串是“字母异位词”。字母异位词的定义是：由相同字母按不同顺序排列形成的字符串。这意味着，如果我们将一个字母异位词的所有字母重新排序，它们都会得到同一个字符串。

例如，“eat”，“tea”，“ate”这三个词，排序后都变成了“aet”。

这就给了我们一个绝佳的“钥匙”(Key) 来识别这些词是否属于同一组。我们可以：

1. 创建一个哈希表（字典），用于存储分组结果。这个哈希表的 `key` 将是排序后的字符串，`value` 将是一个列表，存放所有排序后等于这个 `key` 的原始字符串。
2. 遍历输入的字符串数组 `strs`。
3. 对于每个字符串 `s`，我们都将它转换成字符数组，然后进行排序，再合并成一个新的字符串，我们称之为 `sorted_s`。
4. 将 `sorted_s` 作为哈希表的 `key`。检查这个 `key` 是否已经存在于哈希表中。
  - 如果不存在，说明这是一个新的分组。我们在哈希表中创建一个新的条目，`key` 是 `sorted_s`，`value` 是一个只包含当前字符串 `s` 的列表。
  - 如果存在，说明当前的字符串 `s` 属于这个已有的分组。我们只需要把 `s` 添加到 `key` 对应的 `value` 列表中即可。
5. 遍历完所有字符串后，哈希表中所有的 `value` 就是我们想要的结果。我们只需要将它们提取出来即可。

复杂度分析：

- **时间复杂度：**  $O(n * k \log k)$ 。其中 `n` 是 `strs` 的长度，`k` 是 `strs` 中字符串的最大长度。我们需要遍历 `n` 个字符串，对于每个字符串，我们需要花费  $O(k \log k)$  的时间来进行排序。
- **空间复杂度：**  $O(n * k)$ 。我们需要一个哈希表来存储所有的字符串。

## Python 代码实现

```
from typing import List
import collections

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        """
        使用哈希表，key 为排序后的字符串，value 为字母异位词列表
        """

        # 创建一个默认值为列表的字典
        hash_map = collections.defaultdict(list)

        # 遍历输入列表中的每个字符串
        for s in strs:
            # 对字符串进行排序，得到一个唯一的 key
            # 例如 "eat", "tea", "ate" 排序后都是 "aet"
            key = ''.join(sorted(s))

            # 将原始字符串 s 添加到对应 key 的列表中
            hash_map[key].append(s)

        # 字典中的 values 就是我们想要的结果
        return list(hash_map.values())
```

## 回溯 / 131. 分割回文串

### 131. 分割回文串

#### 题目描述

给你一个字符串 `s`, 请你将 `s` 分割成一些子串, 使每个子串都是回文串。返回 `s` 所有可能的分割方案。

回文串是正着读和反着读都一样的字符串。

#### 示例 1:

输入: `s = "aab"`

输出: `[["a", "a", "b"], ["aa", "b"]]`

#### 示例 2:

输入: `s = "raceacar"`

输出: `[["r", "a", "c", "e", "a", "c", "a", "r"], ["r", "a", "ce", "c", "a", "r"], ["r", "ace", "ca", "r"], ["raceacar"]]`

#### 示例 3:

输入: `s = "a"`

输出: `[["a"]]`

#### 提示:

- `1 <= s.length <= 16`
- `s` 仅由小写英文字母组成

#### 思考过程:

这道题要求我们将一个给定的字符串 `s` 分割成若干个子串, 使得每个子串都是回文串, 并返回所有可能的分割方案。这是一个典型的回溯问题。

#### 核心思路:

我们可以通过递归和回溯的方式来解决这个问题。在每一步, 我们尝试从当前位置开始, 截取不同长度的子串。对于每个截取的子串, 我们检查它是否是回文串。

- 如果是回文串, 我们就将这个子串添加到当前的分割方案中, 然后对字符串的剩余部分进行递归调用。
- 如果不是回文串, 我们就跳过这个子串, 尝试下一个长度的子串。

#### 回溯函数 `backtrack(start_index, current_partition)` 的设计:

##### 参数:

- `start_index`: 当前考虑的子串在原字符串 `s` 中的起始索引。
- `current_partition`: 一个列表, 存储了当前已经找到的回文子串。

##### 终止条件:

- 如果 `start_index` 等于字符串 `s` 的长度, 这意味着我们已经成功地将整个字符串分割成了回文子串。此时, 将 `current_partition` 的一个副本添加到最终结果列表 `results` 中。

##### 递归过程:

- 遍历 `end_index` 从 `start_index` 到 `len(s) - 1`。
- 提取子串 `substring = s[start_index : end_index + 1]`。
- 使用一个辅助函数 `is_palindrome(sub)` 来检查 `substring` 是否是回文串。
- 如果 `substring` 是回文串:
  - 将 `substring` 添加到 `current_partition`。
  - 递归调用 `backtrack(end_index + 1, current_partition)`, 继续处理字符串的剩余部分。
- 回溯: 在递归调用返回后, 从 `current_partition` 中移除 `substring`。这是为了撤销当前的选择, 以便探索其他可能的分割方案。

#### 辅助函数 `is_palindrome(sub)`:

- 这个函数用于判断一个字符串 `sub` 是否是回文串。最简单的方法是比较字符串和它的反转字符串是否相等。

#### 整体结构:

1. 初始化一个空列表 `results` 来存储所有有效的分割方案。
2. 定义 `is_palindrome` 辅助函数。
3. 定义 `backtrack` 回溯函数。
4. 调用 `backtrack(0, [])` 开始回溯过程。
5. 返回 `results`。

#### Python 代码:

```

class Solution:
    def partition(self, s: str) -> list[list[str]]:
        results = []
        n = len(s)

        def is_palindrome(sub: str) -> bool:
            return sub == sub[::-1]

        def backtrack(start_index, current_partition):
            if start_index == n:
                results.append(list(current_partition)) # 添加副本
                return

            for end_index in range(start_index, n):
                substring = s[start_index : end_index + 1]
                if is_palindrome(substring):
                    current_partition.append(substring)
                    backtrack(end_index + 1, current_partition)
                    current_partition.pop() # 回溯

        backtrack(0, [])
        return results

```

## 回溯 / 17. 电话号码的字母组合

### 17. 电话号码的字母组合

#### 题目描述

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。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"]

```

#### 示例 2:

```

输入: digits = ""
输出: []

```

#### 示例 3:

```

输入: digits = "2"
输出: ["a", "b", "c"]

```

#### 提示:

- $0 \leq \text{digits.length} \leq 4$
- $\text{digits}[i]$  是范围 [2, 9] 的一个数字。

#### 思考过程:

这道题是典型的回溯问题，需要生成所有可能的字母组合。我们可以将数字映射到相应的字母，然后通过递归的方式构建组合。

## 核心思路:

1. **数字到字母的映射:** 首先, 我们需要一个字典或哈希表来存储数字到字母的映射关系, 例如 `{'2': 'abc', '3': 'def', ...}`。

2. **回溯函数 `backtrack(index, current_combination)` 的设计:**

- **参数:**

- `index`: 当前正在处理的数字在 `digits` 字符串中的索引。
- `current_combination`: 当前已经构建的字母组合字符串。

- **终止条件:**

- 如果 `index` 等于 `digits` 的长度, 说明已经处理完所有数字, 将 `current_combination` 添加到结果列表 `res` 中。

- **递归过程:**

- 获得当前数字 `digit = digits[index]`。
- 根据映射关系, 获得 `digit` 对应的所有字母 `letters = mapping[digit]`。
- 遍历 `letters` 中的每一个字母 `letter`:
- 递归调用 `backtrack(index + 1, current_combination + letter)`。

## 主函数逻辑:

1. 初始化一个空列表 `res` 来存储所有结果。

2. 定义数字到字母的映射 `mapping`。

3. 如果输入的 `digits` 为空, 直接返回空列表。

4. 调用 `backtrack(0, "")` 开始回溯过程。

5. 返回 `res`。

## Python 代码:

```
class Solution:
    def letterCombinations(self, digits: str) -> list[str]:
        if not digits:
            return []

        mapping = {
            '2': 'abc',
            '3': 'def',
            '4': 'ghi',
            '5': 'jkl',
            '6': 'mno',
            '7': 'pqrs',
            '8': 'tuv',
            '9': 'wxyz'
        }

        res = []

        def backtrack(index, current_combination):
            if index == len(digits):
                res.append(current_combination)
                return

            digit = digits[index]
            letters = mapping[digit]

            for letter in letters:
                backtrack(index + 1, current_combination + letter)

        backtrack(0, "")
        return res
```

---

## 回溯 / 22. 括号生成

### 22. 括号生成

## 题目描述

数字  $n$  代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的 括号组合。

### 示例 1：

```
输入: n = 3
输出: ["((()))", "(()())", "(())()", "()(())", "()()()"]
```

### 示例 2：

```
输入: n = 1
输出: ["()"]
```

### 提示：

- $1 \leq n \leq 8$

### 思考过程：

这道题是典型的回溯算法问题。我们需要生成所有有效的括号组合。一个有效的括号组合需要满足两个条件：

1. 任意前缀中，左括号的数量大于或等于右括号的数量。
2. 最终，左括号的数量等于右括号的数量。

我们可以使用递归来构建括号字符串。在每一步，我们有两个选择：添加一个左括号或添加一个右括号。但是，我们需要确保这两个选择是有效的。

- **添加左括号的条件：**只要当前左括号的数量小于  $n$ ，我们就可以添加一个左括号。
- **添加右括号的条件：**只要当前右括号的数量小于左括号的数量，我们就可以添加一个右括号。这是为了保证任意前缀中左括号的数量大于或等于右括号的数量。

当左括号和右括号的数量都等于  $n$  时，我们就找到了一个有效的括号组合，将其添加到结果列表中。

### Python 代码：

```
class Solution:
    def generateParenthesis(self, n: int) -> list[str]:
        res = []

        def backtrack(left_count, right_count, current_string):
            # 终止条件：当左右括号数量都达到n时，说明找到了一个有效组合
            if left_count == n and right_count == n:
                res.append(current_string)
                return

            # 剪枝：如果右括号数量大于左括号数量，则当前路径无效
            if right_count > left_count:
                return

            # 尝试添加左括号
            if left_count < n:
                backtrack(left_count + 1, right_count, current_string + "(")

            # 尝试添加右括号
            if right_count < n:
                backtrack(left_count, right_count + 1, current_string + ")")

        backtrack(0, 0, "")
        return res
```

---

## 回溯 / 39. 组合总和

### 39. 组合总和

## 题目描述

给你一个无重复元素的整数数组 candidates 和一个目标整数 target，找出 candidates 中可以使数字和为目标数 target 的所有不同组合，并以列表形式返回。你可以按任意顺序返回这些组合。

candidates 中的同一个数字可以无限重复被选取。如果至少一个数字的被选数量不同，则两种组合是不同的。

对于给定的输入，保证和为 target 的不同组合数少于 150 个。

#### 示例 1：

```
输入: candidates = [2,3,6,7], target = 7
输出: [[2,2,3],[7]]
解释:
2 和 3 可以形成一组候选, 2 + 2 + 3 = 7 。注意 2 可以使用多次。
7 也是一个候选, 7 = 7 。
仅有这两种组合。
```

#### 示例 2：

```
输入: candidates = [2,3,5], target = 8
输出: [[2,2,2,2],[2,3,3],[3,5]]
```

#### 示例 3：

```
输入: candidates = [2], target = 1
输出: []
```

#### 提示：

- $1 \leq \text{candidates.length} \leq 30$
- $2 \leq \text{candidates}[i] \leq 40$
- candidates 中的所有元素互不相同
- $1 \leq \text{target} \leq 40$

#### 思考过程：

这道题是典型的回溯问题，要求从给定的数组 candidates 中找出所有和为 target 的组合。candidates 中的数字可以重复使用。

#### 核心思路：

我们可以使用回溯算法来解决这个问题。在每一步，我们尝试从 candidates 中选择一个数字，并将其添加到当前组合中。然后，我们递归地处理剩余的 target。

#### 回溯函数 `backtrack(remain, current_combination, start_index)` 的设计：

- **参数：**
  - `remain`: 还需要凑齐的目标和。
  - `current_combination`: 当前已经构建的组合。
  - `start_index`: 从 candidates 数组的哪个索引开始选择数字。这是为了避免生成重复的组合（例如 [2, 3] 和 [3, 2] 视为同一种组合）。
- **终止条件：**
  - 如果 `remain == 0`，说明当前组合的和等于 target，将 `current_combination` 的一个副本添加到结果列表 `res` 中。
  - 如果 `remain < 0`，说明当前组合的和已经超过 target，这条路径无效，直接返回。
- **递归过程：**
  - 遍历 candidates 数组，从 `start_index` 开始。
  - 对于每个数字 `candidate = candidates[i]`：
    - 将 `candidate` 添加到 `current_combination`。
    - 递归调用 `backtrack(remain - candidate, current_combination, i)`。注意这里传入的 `start_index` 仍然是 `i`，因为同一个数字可以重复使用。
    - 回溯：在递归调用返回后，从 `current_combination` 中移除 `candidate`。这是为了撤销当前的选择，以便探索其他可能的组合。

#### 主函数逻辑：

1. 初始化一个空列表 `res` 来存储所有结果。
2. 对 candidates 数组进行排序。虽然这道题不强制要求排序，但排序可以帮助我们更好地理解和优化剪枝。
3. 调用 `backtrack(target, [], 0)` 开始回溯过程。
4. 返回 `res`。

#### Python 代码：

```

class Solution:
    def combinationSum(self, candidates: list[int], target: int) -> list[list[int]]:
        res = []
        candidates.sort() # 排序有助于剪枝，虽然本题不强制要求

        def backtrack(remain, current_combination, start_index):
            if remain == 0:
                res.append(list(current_combination)) # 添加副本
                return
            if remain < 0:
                return

            for i in range(start_index, len(candidates)):
                # 剪枝：如果当前数字已经大于剩余目标和，则后续数字也无需考虑
                if candidates[i] > remain:
                    break

                current_combination.append(candidates[i])
                backtrack(remain - candidates[i], current_combination, i) # 注意这里是i，表示可以重复使用当前数
                current_combination.pop() # 回溯

        backtrack(target, [], 0)
        return res

```

## 回溯 / 46. 全排列

### 46. 全排列

#### 题目描述

给定一个不含重复数字的数组 `nums`，返回其所有可能的全排列。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]]

```

#### 示例 2：

```

输入: nums = [0,1]
输出: [[0,1],[1,0]]

```

#### 示例 3：

```

输入: nums = [1]
输出: [[1]]

```

#### 提示：

- $1 \leq \text{nums.length} \leq 6$
- $-10 \leq \text{nums}[i] \leq 10$
- $\text{nums}$  中的所有整数互不相同

#### 思考过程：

这道题是典型的回溯问题，要求生成给定数组 `nums` 的所有可能的全排列。

#### 核心思路：

我们可以使用回溯算法来解决这个问题。在每一步，我们从 `nums` 中选择一个尚未使用的数字，并将其添加到当前排列中。然后，我们递归地处理剩余的数字。

#### 回溯函数 `backtrack(current_permutation)` 的设计：

- **参数:**
  - `current_permutation`: 当前已经构建的排列。
- **全局变量/外部状态:**
  - `nums`: 原始数组。
  - `res`: 存储所有找到的全排列。
  - `used`: 一个布尔数组或集合，用于标记 `nums` 中哪些数字已经被使用过。
- **终止条件:**
  - 如果 `len(current_permutation) == len(nums)`，说明当前排列的长度等于 `nums` 的长度，已经构建了一个完整的排列。将 `current_permutation` 的一个副本添加到结果列表 `res` 中。
- **递归过程:**
  - 遍历 `nums` 数组。
  - 对于每个数字 `num = nums[i]`:
    - **剪枝:** 如果 `nums[i]` 已经被使用过，则跳过。
    - **选择:**
      - 将 `nums[i]` 添加到 `current_permutation`。
      - 将 `used[i]` 标记为 `True`。
    - **递归:** 调用 `backtrack(current_permutation)`。
    - **回溯:** 在递归调用返回后，撤销当前的选择，以便探索其他可能性：
      - 从 `current_permutation` 中移除 `nums[i]`。
      - 将 `used[i]` 标记为 `False`。

#### 主函数逻辑:

1. 初始化一个空列表 `res` 来存储所有结果。
2. 初始化 `used` 数组为 `[False] * len(nums)`。
3. 调用 `backtrack([])` 开始回溯过程。
4. 返回 `res`。

#### Python 代码:

```
class Solution:
    def permute(self, nums: list[int]) -> list[list[int]]:
        res = []
        used = [False] * len(nums)

        def backtrack(current_permutation):
            if len(current_permutation) == len(nums):
                res.append(list(current_permutation)) # 添加副本
                return

            for i in range(len(nums)):
                if not used[i]:
                    current_permutation.append(nums[i])
                    used[i] = True
                    backtrack(current_permutation)
                    used[i] = False # 回溯
                    current_permutation.pop() # 回溯

        backtrack([])
        return res
```

## 回溯 / 51. N 皇后

### 51. N 皇后

#### 题目描述

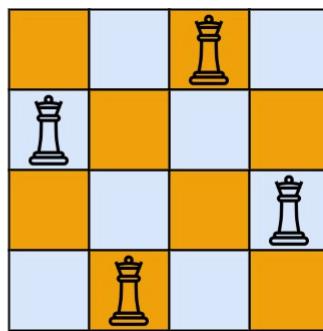
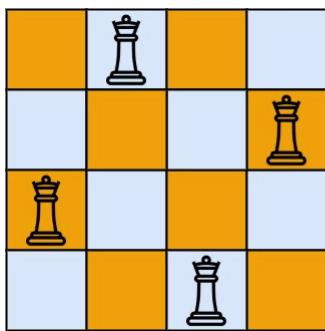
按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

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

给你一个整数 n，返回所有不同的 n 皇后问题 的解决方案。

每一种解法包含一个不同的 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。

#### 示例 2：

输入: `n = 1`

输出: `[["Q"]]`

#### 提示:

- `1 <= n <= 9`

#### 思考过程:

N 皇后问题是一个经典的回溯算法问题。目标是在一个  $N \times N$  的棋盘上放置 N 个皇后，使得任意两个皇后都不能互相攻击。这意味着任何两个皇后都不能在同一行、同一列或同一对角线上。

#### 核心思路:

我们可以逐行放置皇后。对于每一行，我们尝试在不同的列上放置皇后。在放置之前，我们需要检查这个位置是否安全（即不会被之前放置的皇后攻击）。如果安全，我们就放置皇后并递归地进入下一行；如果不安全，我们就尝试当前行的下一个列。如果当前行所有列都尝试完毕，或者无法放置皇后，我们就回溯到上一行，撤销之前的放置，并尝试其他列。

#### 如何判断位置是否安全？

一个位置 `(row, col)` 是否安全，需要检查：

1. **同一列:** 检查当前列 `col` 是否已经有皇后。
2. **主对角线:** 检查 `row - col` 是否已经有皇后。在同一主对角线上的所有单元格，它们的行索引减去列索引的值是相同的。
3. **副对角线:** 检查 `row + col` 是否已经有皇后。在同一副对角线上的所有单元格，它们的行索引加上列索引的值是相同的。

为了高效地进行这些检查，我们可以使用三个集合来分别存储已占用列的索引、已占用主对角线的值 (`row - col`) 和已占用副对角线的值 (`row + col`)。

#### 回溯函数 `backtrack(row)` 的设计：

##### 参数:

- `row`: 当前正在尝试放置皇后的行索引。

##### 全局变量/外部状态:

- `n`: 棋盘的大小。
- `board`: 当前棋盘的状态，可以用一个列表的字符串来表示，例如 `[..., ".Q..", "...Q", "..Q."]`。
- `cols`: 存储已占用列的集合。
- `diag1`: 存储已占用主对角线 (`row - col`) 的集合。
- `diag2`: 存储已占用副对角线 (`row + col`) 的集合。
- `solutions`: 存储所有找到的有效棋盘布局。

##### 终止条件:

- 如果 `row == n`，说明所有 N 个皇后都已成功放置，找到了一个有效的解决方案。将当前 `board` 的副本添加到 `solutions` 中。

##### 递归过程:

- 遍历当前行 `row` 的所有列 `col` (从 0 到 `n-1`)。
- 对于每个 `col`：
  - **检查安全性:** 如果 `col` 不在 `cols` 中，`row - col` 不在 `diag1` 中，且 `row + col` 不在 `diag2` 中，则当前位置 `(row, col)` 是安全的。
  - **放置皇后:**

- 更新 `board[row]`，将 `.` 替换为 `Q`。
- 将 `col` 添加到 `cols` 集合。
- 将 `row - col` 添加到 `diag1` 集合。
- 将 `row + col` 添加到 `diag2` 集合。
- **递归:** 调用 `backtrack(row + 1)`，尝试在下一行放置皇后。
- **回溯:** 在递归调用返回后，撤销当前放置的皇后，以便探索其他可能性：
  - 更新 `board[row]`，将 `Q` 替换回 `.`。
  - 从 `cols` 集合中移除 `col`。
  - 从 `diag1` 集合中移除 `row - col`。
  - 从 `diag2` 集合中移除 `row + col`。

#### 主函数逻辑:

1. 初始化 `n`。
2. 初始化 `board` 为一个  $N \times N$  的空棋盘（全部是 `.`）。
3. 初始化 `cols`, `diag1`, `diag2` 为空集合。
4. 初始化 `solutions` 为空列表。
5. 调用 `backtrack(0)` 开始回溯过程。
6. 返回 `solutions`。

#### Python 代码:

```
class Solution:
    def solveNQueens(self, n: int) -> list[list[str]]:
        solutions = []
        # 初始化棋盘，全部是 '.'
        board = ["."] * n for _ in range(n)

        # 记录已占用的列、主对角线和副对角线
        cols = set()
        diag1 = set() # row - col
        diag2 = set() # row + col

        def backtrack(row):
            # 终止条件：所有皇后都已放置
            if row == n:
                solutions.append(list(board)) # 添加当前棋盘的副本
                return

            # 遍历当前行的所有列
            for col in range(n):
                # 检查当前位置是否安全
                if col not in cols and \
                   (row - col) not in diag1 and \
                   (row + col) not in diag2:

                    # 放置皇后
                    board[row] = board[row][:col] + "Q" + board[row][col+1:]
                    cols.add(col)
                    diag1.add(row - col)
                    diag2.add(row + col)

                    # 递归到下一行
                    backtrack(row + 1)

                    # 回溯：撤销放置
                    board[row] = board[row][:col] + "." + board[row][col+1:]
                    cols.remove(col)
                    diag1.remove(row - col)
                    diag2.remove(row + col)

        backtrack(0)
        return solutions
```

```
backtrack(0)
    return solutions
```

## 回溯 / 78. 子集

### 78. 子集

#### 题目描述

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]]
```

#### 示例 2：

```
输入: nums = [0]
输出: [[], [0]]
```

#### 提示：

- $1 \leq \text{nums.length} \leq 10$
- $-10 \leq \text{nums}[i] \leq 10$
- `nums` 中的所有元素互不相同

#### 思考过程：

这道题是典型的回溯问题，要求生成给定数组 `nums` 的所有可能的子集（包括空集）。

#### 核心思路：

我们可以使用回溯算法来解决这个问题。对于 `nums` 中的每一个数字，我们都有两种选择：

1. 选择当前数字，将其添加到当前子集中，然后递归处理下一个数字。
2. 不选择当前数字，直接递归处理下一个数字。

#### 回溯函数 `backtrack(index, current_subset)` 的设计：

- **参数：**
  - `index`：当前正在处理的数字在 `nums` 数组中的索引。
  - `current_subset`：当前已经构建的子集。
- **全局变量/外部状态：**
  - `nums`：原始数组。
  - `res`：存储所有找到的子集。
- **终止条件：**
  - 当 `index == len(nums)` 时，说明已经遍历完 `nums` 中的所有数字。此时，将 `current_subset` 的一个副本添加到结果列表 `res` 中。
- **递归过程：**
  1. **选择当前数字：**
    - 将 `nums[index]` 添加到 `current_subset`。
    - 递归调用 `backtrack(index + 1, current_subset)`。
    - 回溯：从 `current_subset` 中移除 `nums[index]`。
  2. **不选择当前数字：**
    - 直接递归调用 `backtrack(index + 1, current_subset)`。

#### 主函数逻辑：

1. 初始化一个空列表 `res` 来存储所有结果。
2. 调用 `backtrack(0, [])` 开始回溯过程。
3. 返回 `res`。

#### Python 代码：

```

class Solution:
    def subsets(self, nums: list[int]) -> list[list[int]]:
        res = []
        n = len(nums)

        def backtrack(index, current_subset):
            # 每次进入递归，都将当前子集添加到结果中
            # 这是因为每个节点都代表一个有效的子集
            res.append(list(current_subset))

            for i in range(index, n):
                current_subset.append(nums[i])
                backtrack(i + 1, current_subset)
                current_subset.pop() # 回溯

        backtrack(0, [])
        return res

```

## 回溯 / 79. 单词搜索

### 79. 单词搜索

#### 题目描述

给定一个  $m \times n$  二维字符网格 board 和一个字符串单词 word。如果 word 存在于网格中，返回 true；否则，返回 false。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

#### 示例 1：

A	B	C	E
S	F	C	S
A	D	E	E

输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCED"

输出: true

#### 示例 2：

A	B	C	E
S	F	C	S
A	D	E	E

输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "SEE"

输出: true

#### 示例 3：

A	B	C	E
S	F	C	S
A	D	E	E

输入: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCBS"  
输出: false

#### 提示:

- $m == \text{board.length}$
- $n = \text{board[i].length}$
- $1 \leq m, n \leq 6$
- $1 \leq \text{word.length} \leq 15$
- $\text{board}$  和  $\text{word}$  仅由大小写英文字母组成

**进阶:** 你可以使用搜索剪枝的技术来优化解决方案，使其在  $\text{board}$  更大的情况下可以更快解决问题？

#### 思考过程:

这道题是典型的回溯算法在二维网格中的应用。我们需要在给定的  $\text{board}$  中查找是否存在  $\text{word}$ 。我们可以从  $\text{board}$  中的任意一个单元格开始，然后向上下左右四个方向移动，尝试匹配  $\text{word}$  中的下一个字符。

为了避免重复访问同一个单元格，我们需要一个  $\text{visited}$  数组来记录已经访问过的单元格。

#### 回溯函数 `backtrack(row, col, index)` 的设计:

- **参数:**
  - $\text{row}$ : 当前单元格的行索引。
  - $\text{col}$ : 当前单元格的列索引。
  - $\text{index}$ : 当前要匹配的  $\text{word}$  中的字符索引。
- **终止条件:**
  - 如果  $\text{index}$  等于  $\text{word}$  的长度，说明  $\text{word}$  中的所有字符都已匹配成功，返回 `True`。
  - 如果当前单元格超出边界，或者当前单元格的字符与  $\text{word}[\text{index}]$  不匹配，或者当前单元格已经被访问过，返回 `False`。
- **递归过程:**
  1. 标记当前单元格为已访问。
  2. 尝试向上下左右四个方向移动，递归调用 `backtrack` 函数。
  3. 如果任何一个方向的递归调用返回 `True`，则说明找到了  $\text{word}$ ，返回 `True`。
  4. **回溯:** 在当前路径探索结束后，将当前单元格标记为未访问，以便其他路径可以访问它。这是回溯算法的关键一步。

#### 主函数逻辑:

遍历  $\text{board}$  中的每一个单元格，以每个单元格作为起点调用 `backtrack` 函数。如果任何一次调用返回 `True`，则说明找到了  $\text{word}$ ，返回 `True`。如果遍历完所有单元格都没有找到，则返回 `False`。

#### Python 代码:

```
class Solution:
    def exist(self, board: list[list[str]], word: str) -> bool:
        rows = len(board)
        cols = len(board[0])
        visited = [[False] * cols for _ in range(rows)]

        def backtrack(row, col, index):
            # 终止条件: 如果word中的所有字符都已匹配成功
            if index == len(word):
                return True

            # 剪枝: 如果超出边界，或者字符不匹配，或者已经访问过
            if not (0 <= row < rows and 0 <= col < cols) or \
                board[row][col] != word[index] or \
                visited[row][col]:
                return False

            # 标记当前单元格为已访问
            visited[row][col] = True

            # 尝试向上下左右四个方向移动
            for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                if backtrack(row + dr, col + dc, index + 1):
                    return True

            # 回溯: 将当前单元格标记为未访问
            visited[row][col] = False

        for r in range(rows):
            for c in range(cols):
                if backtrack(r, c, 0):
                    return True

        return False
```

```

        return False

    # 标记当前单元格为已访问
    visited[row][col] = True

    # 尝试向四个方向移动
    found = (backtrack(row + 1, col, index + 1) or
              backtrack(row - 1, col, index + 1) or
              backtrack(row, col + 1, index + 1) or
              backtrack(row, col - 1, index + 1))

    # 回溯：取消标记，以便其他路径可以访问
    visited[row][col] = False

    return found

# 遍历所有可能的起点
for r in range(rows):
    for c in range(cols):
        if backtrack(r, c, 0):
            return True

return False

```

## 图论 / 200. 岛屿数量

### 200. 岛屿数量

#### 题目描述

给你一个由 '1' (陆地) 和 '0' (水) 组成的数据 `grid`，请你计算 `grid` 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 `grid` 的四条边均被水包围。

#### 示例

示例 1：

输入：`grid` = [  
 ["1", "1", "1", "1", "0"],  
 ["1", "1", "0", "1", "0"],  
 ["1", "1", "0", "0", "0"],  
 ["0", "0", "0", "0", "0"]  
]

输出：1

示例 2：

输入：`grid` = [  
 ["1", "1", "0", "0", "0"],  
 ["1", "1", "0", "0", "0"],  
 ["0", "0", "1", "0", "0"],  
 ["0", "0", "0", "1", "1"]  
]

输出：3

#### 约束条件

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`

- `grid[i][j]` 为 '0' 或 '1'

## 思考过程

### 思考 第一步：理解问题

- 我们有一个二维网格，由陆地 ('1') 和水 ('0') 组成。
- 岛屿是由水平或垂直方向相连的陆地组成的。
- 我们需要计算网格中有多少个独立的岛屿。

### 思考 第二步：如何识别和计数岛屿？

想象一下，当你看到一个岛屿时，你会怎么做？你会从一个陆地开始，然后沿着所有相连的陆地走，直到把整个岛屿都“走遍”。走遍之后，这个岛屿就被“标记”了，下次再遇到它的一部分时，就知道它已经属于一个已知的岛屿了。

这正是图的遍历算法（DFS 或 BFS）的思路。

**核心思路：**

1. 遍历整个网格。
2. 每当遇到一个未访问过的陆地 ('1')，就说明我们发现了一个新的岛屿。
3. 岛屿数量加一。
4. 然后，从这个陆地开始，使用 DFS 或 BFS 遍历所有与它相连的陆地，并将它们标记为已访问（例如，将其值改为 '0' 或 '2'），以避免重复计数。

### 思考 第三步：深度优先搜索 (DFS)

DFS 是一种递归的遍历方式。

**DFS 辅助函数 `dfs(row, col)` :**

- **基本情况：**如果 `(row, col)` 超出网格边界，或者 `grid[row][col]` 是水 ('0')，或者已经访问过（例如 '2'），则直接返回。
- **标记当前陆地：**将 `grid[row][col]` 标记为已访问（例如改为 '0'）。
- **递归探索相邻陆地：**对当前陆地的上、下、左、右四个方向的相邻陆地递归调用 `dfs`。

### 思考 第四步：算法步骤总结

1. 初始化 `num_islands = 0`。
2. 获取网格的行数 `m` 和列数 `n`。
3. 遍历网格中的每一个单元格 `(i, j)`：
  - a. 如果 `grid[i][j]` 是陆地 ('1')：
  - i. `num_islands` 加一。
  - ii. 调用 `dfs(i, j)` 来淹没（标记）整个岛屿。
4. 返回 `num_islands`。

**时间复杂度：**  $O(mn)$  - 每个单元格最多被访问一次。

**空间复杂度：**  $O(mn)$  - 递归栈的深度，最坏情况下整个网格都是陆地。

## 代码实现

### Python

```
def numIslands(grid: list[list[str]]) -> int:
    """
    使用深度优先搜索 (DFS) 计算岛屿数量。
    """

    if not grid or not grid[0]:
        return 0

    m, n = len(grid), len(grid[0])
    num_islands = 0

    # DFS 辅助函数，用于淹没（标记）整个岛屿
    def dfs(row, col):
        # 边界条件：超出网格范围，或者遇到水，或者已经访问过
        if not (0 <= row < m and 0 <= col < n) or grid[row][col] == '0':
            return

        # 标记当前陆地为已访问（将其变为 '0'，表示已淹没）
        grid[row][col] = '0'

        # 递归探索相邻陆地
        dfs(row + 1, col)
        dfs(row - 1, col)
        dfs(row, col + 1)
        dfs(row, col - 1)

    for i in range(m):
        for j in range(n):
            if grid[i][j] == '1':
                num_islands += 1
                dfs(i, j)

    return num_islands
```

```

# 递归探索相邻的陆地
dfs(row + 1, col) # 下
dfs(row - 1, col) # 上
dfs(row, col + 1) # 右
dfs(row, col - 1) # 左

# 遍历整个网格
for i in range(m):
    for j in range(n):
        if grid[i][j] == '1':
            num_islands += 1 # 发现一个新岛屿
            dfs(i, j) # 淹没整个岛屿

return num_islands

```

## 关键点总结

1. **图的遍历**: 将二维网格问题抽象为图的遍历问题，每个陆地单元格是节点，相邻陆地之间有边。
2. **DFS/BFS**: 两种常用的图遍历算法都可以解决此问题。DFS 递归实现简洁，BFS 迭代实现更适合处理层级关系。
3. **原地标记**: 通过修改 `grid` 中的值（例如将 '1' 改为 '0' 或 '2'），来标记已访问的陆地，避免重复计数和重复遍历。
4. **边界条件**: 在递归或迭代中，需要仔细处理边界条件，确保不会越界。

## 图论 / 207. 课程表

### 207. 课程表

#### 题目描述

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。例如，想要学习课程 0，你需要先学课程 1，表示为 `[0,1]`。

给定 `numCourses` 以及一个表示学习顺序的整数数组 `prerequisites`，其中 `prerequisites[i] = [ai, bi]` 表示学习课程 `ai` 之前，你需要先学习课程 `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

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，你需要先学习课程 0。所以是可能的。

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 之前需要学习课程 0，学习课程 0 之前需要学习课程 1。这是一个死锁，无法完成所有课程的学习。

示例 3：

输入: `numCourses = 5, prerequisites = [[1,4],[2,4],[3,1],[3,2]]`

输出: `true`

#### 约束条件

- `1 <= numCourses <= 2000`
- `0 <= prerequisites.length <= 5000`
- `prerequisites[i].length == 2`
- `0 <= ai, bi < numCourses`
- `ai != bi`
- `prerequisites` 中不存在重复的边

#### 思考过程

##### 第一步：理解问题

- 我们有 `numCourses` 门课程，以及一些先修课程的要求。
- `[ai, bi]` 表示学习 `ai` 之前必须先学习 `bi`。
- 我们需要判断是否可以完成所有课程的学习。

## 💡 第二步：抽象为图论问题

这个问题可以抽象为一个**有向图**。

- **节点**: 每门课程是一个节点。
- **边**: 如果学习 `ai` 需要先学习 `bi`，那么就有一条从 `bi` 指向 `ai` 的有向边。

**思考题**: 在这样的图中，什么情况会导致无法完成所有课程的学习？

► 点击查看分析

## 💡 第三步：检测有向图中的环

检测有向图中的环有两种常用方法：

### 1. 深度优先搜索 (DFS):

- 在 DFS 遍历过程中，如果遇到一个已经访问过，并且当前正在递归栈中的节点，就说明存在环。

### 2. 拓扑排序 (Topological Sort):

- 如果一个有向无环图 (DAG) 可以进行拓扑排序，那么它就没有环。
- 拓扑排序的原理是：每次选择一个入度为 0 的节点（没有先修课程的课程），将其从图中移除，并更新其邻居的入度。如果最终所有节点都能被移除，则没有环。

考虑到拓扑排序更直观，我们选择拓扑排序。

## 💡 第四步：拓扑排序算法细节

### 1. 构建图：

- 使用邻接表 `adj` 来表示图：`adj[u]` 存储 `u` 的所有邻居（即 `u` 是 `v` 的先修课程，`u -> v`）。
- 使用一个数组 `in_degree` 来存储每个节点的入度：`in_degree[i]` 表示课程 `i` 的先修课程数量。

### 2. 初始化队列：

- 将所有入度为 0 的节点（没有先修课程的课程）加入队列 `queue`。

### 3. BFS 过程：

- 维护一个 `visited_courses` 计数器，记录已完成的课程数量。
- 当队列不为空时，循环执行：
  - 从队列中取出一个节点 `u`。
  - `visited_courses` 加一。
  - 对于 `u` 的每一个邻居 `v`：
    - 将 `v` 的入度减一：`in_degree[v]--`。
    - 如果 `in_degree[v]` 变为 0，说明 `v` 的所有先修课程都已完成，将 `v` 加入队列。

### 4. 判断结果：

- 如果 `visited_courses` 等于 `numCourses`，说明所有课程都能完成，返回 `True`。
- 否则，说明存在环，无法完成所有课程，返回 `False`。

## 💡 第五步：算法步骤总结

1. 初始化 `adj = [[] for _ in range(numCourses)]` (邻接表)。
2. 初始化 `in_degree = [0] * numCourses` (入度数组)。
3. 构建图和计算入度：
  - 遍历 `prerequisites` 数组中的每一对 `[ai, bi]`：
    - `adj[bi].append(ai)` (从 `bi` 到 `ai` 有一条边)。
    - `in_degree[ai] += 1` (课程 `ai` 的入度加一)。
4. 初始化 `queue = collections.deque()`。
5. 将所有入度为 0 的节点加入队列：
  - 遍历 `in_degree` 数组，如果 `in_degree[i] == 0`，则 `queue.append(i)`。
6. 初始化 `visited_courses = 0`。
7. BFS 拓扑排序：
  - 当 `queue` 不为空时：
    - `u = queue.popleft()`。
    - `visited_courses += 1`。
    - 对于 `adj[u]` 中的每一个邻居 `v`：
      - `in_degree[v] -= 1`。
      - 如果 `in_degree[v] == 0`，则 `queue.append(v)`。
8. 返回 `visited_courses == numCourses`。

时间复杂度:  $O(V + E)$  -  $V$  是节点数 (numCourses),  $E$  是边数 (prerequisites.length)。构建图  $O(E)$ , BFS 遍历  $O(V+E)$ 。

空间复杂度:  $O(V + E)$  - 邻接表和入度数组。

## 代码实现

### Python

```
import collections

def canFinish(numCourses: int, prerequisites: list[list[int]]) -> bool:
    """
    使用拓扑排序 (Kahn 算法) 判断课程是否可以完成。
    """

    # 1. 构建图和计算入度
    adj = [[] for _ in range(numCourses)] # 邻接表
    in_degree = [0] * numCourses # 入度数组

    for course, pre_course in prerequisites:
        adj[pre_course].append(course) # pre_course -> course
        in_degree[course] += 1

    # 2. 初始化队列, 加入所有入度为 0 的课程
    queue = collections.deque()
    for i in range(numCourses):
        if in_degree[i] == 0:
            queue.append(i)

    # 3. BFS 拓扑排序
    visited_courses = 0 # 记录已完成的课程数量
    while queue:
        u = queue.popleft()
        visited_courses += 1

        # 遍历 u 的所有邻居 v
        for v in adj[u]:
            in_degree[v] -= 1 # 移除 u 后, v 的入度减一
            if in_degree[v] == 0:
                queue.append(v) # 如果 v 的入度变为 0, 则加入队列

    # 4. 判断是否所有课程都已完成
    return visited_courses == numCourses
```

## 关键点总结

1. 图的抽象: 将课程和先修课程关系抽象为有向图, 是解决问题的关键一步。
2. 环的检测: 判断是否能完成所有课程, 等价于判断图中是否存在环。
3. 拓扑排序: 通过 Kahn 算法 (基于 BFS 的拓扑排序) 来检测环。如果所有节点都能被拓扑排序, 则无环。
4. 入度: 入度是拓扑排序的核心概念, 它表示一个节点有多少个前置依赖。

## 图论 / 208. 实现 Trie (前缀树)

### 208. 实现 Trie (前缀树)

#### 题目描述

Trie (发音类似 "try") 或者说 前缀树 是一种树形数据结构, 用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用, 例如自动补全和拼写检查。

请你实现 Trie 类:

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word`。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中, 返回 `true`。否则, 返回 `false`。

- `boolean startsWith(String prefix)` 如果之前插入的字符串中有一个前缀与 `prefix` 匹配，返回 `true`。否则，返回 `false`。

## 示例

示例 1：

输入

```
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[], ["apple"], ["apple"], ["app"], ["app"], ["app"]]
```

输出

```
[null, null, true, false, true, null, true]
```

解释

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");    // 返回 True
trie.search("app");     // 返回 False
trie.startsWith("app"); // 返回 True
trie.insert("app");
trie.search("app");     // 返回 True
```

## 约束条件

- `1 <= word.length, prefix.length <= 2000`
- `word` 和 `prefix` 仅由小写英文字母组成
- `insert`、`search` 和 `startsWith` 调用次数总计不超过 `3 * 10^4` 次

## 思考过程

### 💡 第一步：理解问题

- 我们需要实现一个前缀树（Trie）。
- 它支持三种操作：插入单词、查找单词、查找前缀。

### 💡 第二步：Trie 的结构

Trie 是一种树形结构，它的每个节点代表一个字符。从根节点到某个节点的路径，就代表一个字符串。

思考题：

- Trie 的每个节点需要存储什么信息？

▶ 点击查看分析

### 💡 第三步：实现 `TrieNode` 类

为了更好地组织代码，我们可以先定义一个 `TrieNode` 类。

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
```

### 💡 第四步：实现 `Trie` 类的方法

`insert(word)`

- 从根节点开始。
- 遍历 `word` 中的每个字符。
- 如果当前字符在当前节点的 `children` 中不存在，就创建一个新的 `TrieNode` 并添加到 `children` 中。
- 移动到下一个节点。
- 遍历完 `word` 后，将最后一个节点的 `is_end_of_word` 设为 `True`。

`search(word)`

- 从根节点开始。
- 遍历 `word` 中的每个字符。
- 如果当前字符在当前节点的 `children` 中不存在，说明 `word` 不存在，返回 `False`。
- 移动到下一个节点。
- 遍历完 `word` 后，检查最后一个节点的 `is_end_of_word` 是否为 `True`。如果是，返回 `True`，否则返回 `False`。

```

startsWith(prefix)

• 从根节点开始。
• 遍历 prefix 中的每个字符。
• 如果当前字符在当前节点的 children 中不存在，说明 prefix 不存在，返回 False。
• 移动到下一个节点。
• 遍历完 prefix 后，返回 True（因为只要能走到这里，就说明存在这个前缀）。

```

## 💡 第五步：算法步骤总结

1. **TrieNode 类：**
  - `children`: 字典，键为字符，值为 `TrieNode` 对象。
  - `is_end_of_word`: 布尔值，表示是否是单词结尾。
2. **Trie 类：**
  - `__init__(self)` : 初始化 `self.root = TrieNode()`。
  - `insert(self, word)` :
    - `curr = self.root`
    - 遍历 word 中的 char:
      - 如果 char 不在 curr.children 中，`curr.children[char] = TrieNode()`。
      - `curr = curr.children[char]`。
    - `curr.is_end_of_word = True`。
  - `search(self, word)` :
    - `curr = self.root`
    - 遍历 word 中的 char:
      - 如果 char 不在 curr.children 中，返回 False。
      - `curr = curr.children[char]`。
    - 返回 `curr.is_end_of_word`。
  - `startsWith(self, prefix)` :
    - `curr = self.root`
    - 遍历 prefix 中的 char:
      - 如果 char 不在 curr.children 中，返回 False。
      - `curr = curr.children[char]`。
    - 返回 True。

### 时间复杂度：

- `insert`:  $O(L)$  - L 是单词长度。
- `search`:  $O(L)$  - L 是单词长度。
- `startsWith`:  $O(L)$  - L 是前缀长度。

空间复杂度：  $O(\text{总字符数})$  - 最坏情况下，所有单词都没有共同前缀，空间复杂度为所有单词长度之和。

## 代码实现

### Python

```

class TrieNode:
    """
    Trie 树的节点定义。
    """

    def __init__(self):
        self.children = {} # 存储子节点，键为字符，值为 TrieNode 对象
        self.is_end_of_word = False # 标记是否是某个单词的结尾

class Trie:
    """
    Trie (前缀树) 的实现。
    """

    def __init__(self):
        self.root = TrieNode() # 根节点

    def insert(self, word: str) -> None:
        """
        向 Trie 中插入一个单词。
        """

```

```

"""
curr = self.root
for char in word:
    if char not in curr.children:
        curr.children[char] = TrieNode()
    curr = curr.children[char]
curr.is_end_of_word = True

def search(self, word: str) -> bool:
"""
查找 Trie 中是否存在一个单词。
"""

curr = self.root
for char in word:
    if char not in curr.children:
        return False
    curr = curr.children[char]
return curr.is_end_of_word

def startsWith(self, prefix: str) -> bool:
"""
查找 Trie 中是否存在以给定前缀开头的单词。
"""

curr = self.root
for char in prefix:
    if char not in curr.children:
        return False
    curr = curr.children[char]
return True

```

## 关键点总结

1. **树形结构:** Trie 是一种特殊的树，每个节点代表一个字符，路径代表字符串。
2. **节点设计:** 每个节点包含指向子节点的指针（通常用哈希表实现）和一个布尔标记 (`is_end_of_word`) 来表示是否是单词的结尾。
3. **高效查找:** 通过沿着字符路径遍历，可以快速查找单词或前缀，时间复杂度与字符串长度成正比，与字典大小无关。
4. **空间换时间:** Trie 的空间开销可能较大，但它提供了非常高效的字符串查找和前缀匹配功能。

## 图论 / 994. 腐烂的橘子

### 994. 腐烂的橘子

#### 题目描述

在给定的 `m x n` 网格 `grid` 中，每个单元格可以有以下三种值之一：

- 0 代表空单元格
- 1 代表新鲜橘子
- 2 代表腐烂的橘子

每分钟，任何与腐烂的橘子（在 4 个正方向上）相邻的新鲜橘子都会腐烂。

返回直到所有新鲜橘子都腐烂的最短时间。如果不可能，返回 `-1`。

#### 示例

示例 1：

输入: `grid = [[2,1,1],[1,1,0],[0,1,1]]`

输出: 4

解释:

第 0 分钟: `[[2,1,1],[1,1,0],[0,1,1]]`

第 1 分钟: `[[2,2,1],[2,1,0],[0,1,1]]`

第 2 分钟: [[2,2,2],[2,2,0],[0,1,1]]

第 3 分钟: [[2,2,2],[2,2,0],[0,2,1]]

第 4 分钟: [[2,2,2],[2,2,0],[0,2,2]]

所有新鲜橘子都腐烂了。

示例 2:

输入: grid = [[0,2]]

输出: 0

解释: 初始状态下没有新鲜橘子, 所以时间为 0。

示例 3:

输入: grid = [[2,1,1],[0,1,1],[1,0,1]]

输出: -1

解释: 左下角的橘子 (row 2, col 0) 永远不会腐烂, 因为它被一个空单元格阻挡。

## 约束条件

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 10
- 0 <= grid[i][j] <= 2

## 思考过程

### 💡 第一步：理解问题

- 我们有一个网格, 里面有新鲜橘子、腐烂橘子和空单元格。
- 腐烂会“传染”: 每分钟, 腐烂橘子会使相邻的新鲜橘子腐烂。
- 我们需要找到所有新鲜橘子都腐烂的最短时间。
- 如果有新鲜橘子永远不会腐烂, 返回 -1。

### 💡 第二步：最短时间 -> BFS

“最短时间”或“最短路径”问题通常会联想到广度优先搜索(BFS)。

核心思路:

- 这是一个多源 BFS 问题。所有初始腐烂的橘子都是 BFS 的起点。
- BFS 能够一层一层地向外扩散, 天然地就能找到最短路径。
- 每一层扩散代表一分钟。

### 💡 第三步：BFS 算法细节

#### 1. 初始化:

- 统计新鲜橘子的数量 fresh\_oranges。
- 创建一个队列 queue, 将所有初始腐烂橘子的位置 (row, col) 加入队列。
- 初始化时间 minutes = 0。

#### 2. BFS 过程:

- 当队列不为空, 且还有新鲜橘子时, 循环执行:

- minutes 加一。
- 获取当前层腐烂橘子的数量 (即队列的当前大小 level\_size)。
- 遍历当前层的所有腐烂橘子:
  - 从队列中取出一个腐烂橘子 (r, c)。
  - 检查其四个相邻方向 (nr, nc):
    - 如果 (nr, nc) 在网格内, 且 grid[nr][nc] 是新鲜橘子 ('1'):
      - 将 grid[nr][nc] 标记为腐烂 ('2')。
      - fresh\_oranges 减一。
      - 将 (nr, nc) 加入队列。

#### 3. 判断结果:

- 如果 fresh\_oranges == 0, 说明所有新鲜橘子都腐烂了, 返回 minutes。
- 否则, 说明有新鲜橘子无法腐烂, 返回 -1。

### 💡 第四步：算法步骤总结

1. 初始化 fresh\_oranges = 0, queue = collections.deque()。
2. 第一次遍历网格:

- 统计 `fresh_oranges` 的数量。
- 将所有初始腐烂橘子 `(r, c)` 加入 `queue`。

3. 初始化 `minutes = 0`。

4. 当 `queue` 不为空且 `fresh_oranges > 0` 时:

- `minutes += 1`。
- `level_size = len(queue)`。
- 遍历 `level_size` 次:
  - `r, c = queue.popleft()`。
  - 对于 `(r, c)` 的四个相邻方向 `(nr, nc)`:
    - 如果 `(nr, nc)` 有效且 `grid[nr][nc] == 1`:
    - `grid[nr][nc] = 2`。
    - `fresh_oranges -= 1`。
    - `queue.append((nr, nc))`。

5. 返回 `minutes` 如果 `fresh_oranges == 0`, 否则返回 `-1`。

时间复杂度:  $O(mn)$  - 每个单元格最多被访问一次。

空间复杂度:  $O(mn)$  - 队列的最大大小。

## 代码实现

### Python

```
import collections

def orangesRotting(grid: list[list[int]]) -> int:
    """
    使用广度优先搜索 (BFS) 计算所有新鲜橘子腐烂的最短时间。
    """

    m, n = len(grid), len(grid[0])
    queue = collections.deque()
    fresh_oranges = 0

    # 1. 统计新鲜橘子数量，并将所有初始腐烂橘子加入队列
    for r in range(m):
        for c in range(n):
            if grid[r][c] == 1:
                fresh_oranges += 1
            elif grid[r][c] == 2:
                queue.append((r, c)) # 初始腐烂橘子作为 BFS 的起点

    # 如果没有新鲜橘子，直接返回 0 分钟
    if fresh_oranges == 0:
        return 0

    minutes = 0
    # 定义四个方向
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # 2. BFS 过程
    while queue and fresh_oranges > 0:
        minutes += 1 # 每分钟腐烂扩散一层
        level_size = len(queue) # 当前层腐烂橘子的数量

        for _ in range(level_size):
            r, c = queue.popleft()

            # 检查四个相邻方向
            for dr, dc in directions:
                nr, nc = r + dr, c + dc

                # 检查是否在网格内，并且是新鲜橘子
                if 0 <= nr < m and 0 <= nc < n and grid[nr][nc] == 1:
                    grid[nr][nc] = 2
                    fresh_oranges -= 1
                    queue.append((nr, nc))

    return minutes if fresh_oranges == 0 else -1
```

```

        if 0 <= nr < m and 0 <= nc < n and grid[nr][nc] == 1:
            grid[nr][nc] = 2 # 标记为腐烂
            fresh_oranges -= 1
            queue.append((nr, nc)) # 加入队列，等待下一分钟扩散

    # 3. 判断结果
    if fresh_oranges == 0:
        return minutes
    else:
        return -1 # 还有新鲜橘子未腐烂

```

## 关键点总结

- 多源 BFS**: 将所有初始腐烂的橘子作为 BFS 的起始点，同时进行扩散。
- 层序遍历**: BFS 的特性决定了它能够找到最短时间，因为它是按层（分钟）进行扩散的。
- fresh\_oranges 计数器**: 用于判断是否所有新鲜橘子都已腐烂，以及是否存在无法腐烂的橘子。
- 时间步进**: 通过在 BFS 的每一层循环开始时 `minutes += 1` 来记录时间。

## 堆 / 215. 数组中的第K个最大元素

### 215. 数组中的第K个最大元素

#### 题目描述

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。

你必须设计并实现时间复杂度为  $O(n)$  的算法解决此问题。

#### 示例 1：

```

输入: [3,2,1,5,6,4], k = 2
输出: 5

```

#### 示例 2：

```

输入: [3,2,3,1,2,4,5,5,6], k = 4
输出: 4

```

#### 提示：

- `1 <= k <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`

## 解题引导

### 思路：最小堆

#### 思考过程：

要找到第 `k` 个最大的元素，我们可以使用最小堆来解决这个问题。具体思路如下：

- 维护一个大小为 `k` 的最小堆，堆中存储最大的 `k` 个元素。
- 遍历数组，对于每个元素：
  - 如果堆的大小小于 `k`，直接将元素加入堆。
  - 如果堆的大小等于 `k`，将当前元素与堆顶元素比较：
    - 如果当前元素大于堆顶元素，则弹出堆顶元素，将当前元素加入堆。
    - 否则，跳过当前元素。
- 遍历结束后，堆顶元素就是第 `k` 个最大的元素。

这个方法的优点是：

- 我们只需要维护大小为 `k` 的堆，而不需要对整个数组排序。

2. 每次操作的时间复杂度是  $O(\log k)$ , 总的时间复杂度是  $O(n \log k)$ 。

3. 空间复杂度是  $O(k)$ 。

#### 算法流程：

1. 创建一个最小堆。
2. 遍历数组 `nums`:
  - 如果堆的大小小于  $k$ , 将当前元素加入堆。
  - 如果堆的大小等于  $k$ :
    - 如果当前元素大于堆顶元素, 弹出堆顶元素, 将当前元素加入堆。
    - 否则, 继续遍历下一个元素。
3. 返回堆顶元素。

#### 复杂度分析：

- 时间复杂度:  $O(n \log k)$ 。遍历数组需要  $O(n)$ , 每次堆操作需要  $O(\log k)$ 。
- 空间复杂度:  $O(k)$ 。需要额外的堆空间来存储  $k$  个元素。

## Python 代码实现

```
from typing import List
import heapq

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # 创建一个最小堆
        heap = []

        # 遍历数组
        for num in nums:
            # 如果堆的大小小于k, 直接加入
            if len(heap) < k:
                heapq.heappush(heap, num)
            # 如果当前元素大于堆顶元素, 弹出堆顶, 将当前元素加入
            elif num > heap[0]:
                heapq.heapreplace(heap, num)

        # 返回堆顶元素, 即第k个最大的元素
        return heap[0]
```

## 堆 / 295. 数据流的中位数

### 295. 数据流的中位数

#### 题目描述

中位数是有序列表中间的数。如果列表长度是偶数, 中位数则是中间两个数的平均值。

例如,

`[2, 3, 4]` 的中位数是 `3`

`[2, 3]` 的中位数是  $(2 + 3) / 2 = 2.5$

设计一个支持以下两种操作的数据结构:

- `void addNum(int num)` - 从数据流中添加一个整数到数据结构中。
- `double findMedian()` - 返回目前所有元素的中位数。

#### 示例:

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

```
addNum(3)
findMedian() -> 2.0
```

#### 进阶：

1. 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？
2. 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

## 解题引导

### 思路：双堆（最大堆 + 最小堆）

#### 思考过程：

我们需要一个能够动态维护中位数的数据结构。中位数将所有元素分成两部分：较小的一半和较大的一半。

我们可以用两个堆来维护这两部分数据：

1. **最大堆 (max\_heap)**：存储较小的一半元素。堆顶是这部分元素中的最大值。
2. **最小堆 (min\_heap)**：存储较大的一半元素。堆顶是这部分元素中的最小值。

为了能够快速计算中位数，我们需要保持这两个堆的平衡，满足以下两个条件：

1. `max_heap` 中的所有元素都小于或等于 `min_heap` 中的所有元素。
2. 两个堆的大小要么相等，要么 `max_heap` 的大小比 `min_heap` 多一个。

#### 如何计算中位数？

- 如果两个堆大小相等（总元素为偶数），中位数是 `(max_heap 堆顶 + min_heap 堆顶) / 2`。
- 如果 `max_heap` 比 `min_heap` 多一个元素（总元素为奇数），中位数就是 `max_heap` 的堆顶。

#### 如何添加元素 `addNum`？

当新元素 `num` 到来时，为了维持上述两个条件，我们执行以下步骤：

1. **决定将 `num` 添加到哪个堆：**
  - 如果 `max_heap` 为空，或者 `num` 小于等于 `max_heap` 的堆顶，则将 `num` 添加到 `max_heap`。
  - 否则，将 `num` 添加到 `min_heap`。
2. **平衡两个堆的大小：**
  - 如果 `max_heap` 的大小比 `min_heap` 多出超过一个（即 `len(max_heap) > len(min_heap) + 1`），则将 `max_heap` 的堆顶元素移动到 `min_heap`。
  - 如果 `min_heap` 的大小大于 `max_heap`（即 `len(min_heap) > len(max_heap)`），则将 `min_heap` 的堆顶元素移动到 `max_heap`。

注意：Python 的 `heapq` 模块实现的是最小堆。要模拟最大堆，我们可以在存入元素时取其相反数。

#### 复杂度分析：

- `addNum` 时间复杂度： $O(\log n)$ 。每次添加元素，我们最多进行两次堆的插入和一次堆的弹出操作。
- `findMedian` 时间复杂度： $O(1)$ 。只需要访问两个堆的堆顶即可。
- 空间复杂度： $O(n)$ 。需要存储所有的数据流中的元素。

## Python 代码实现

```
import heapq

class MedianFinder:

    def __init__(self):
        # max_heap 存储较小的一半，用相反数实现最大堆
        self.max_heap = []
        # min_heap 存储较大的一半
        self.min_heap = []

    def addNum(self, num: int) -> None:
        # 决定将 num 添加到哪个堆
        if not self.max_heap or num <= -self.max_heap[0]:
            heapq.heappush(self.max_heap, -num)
        else:
```

```

    heapq.heappush(self.min_heap, num)

    # 平衡两个堆的大小
    # 1. max_heap 过大
    if len(self.max_heap) > len(self.min_heap) + 1:
        heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))
    # 2. min_heap 过大
    elif len(self.min_heap) > len(self.max_heap):
        heapq.heappush(self.max_heap, -heapq.heappop(self.min_heap))

def findMedian(self) -> float:
    # 如果总元素个数是偶数
    if len(self.max_heap) == len(self.min_heap):
        return (-self.max_heap[0] + self.min_heap[0]) / 2.0
    # 如果总元素个数是奇数
    else:
        return -self.max_heap[0]

# Your MedianFinder object will be instantiated and called as such:
# obj = MedianFinder()
# obj.addNum(num)
# param_2 = obj.findMedian()

```

## 堆 / 347. 前 K 个高频元素

### 347. 前 K 个高频元素

#### 题目描述

给你一个整数数组 `nums` 和一个整数 `k`，请你返回其中出现频率前 `k` 高的元素。你可以按 **任意顺序** 返回答案。

#### 示例 1:

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

#### 示例 2:

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

#### 提示:

- `1 <= nums.length <= 10^5`
- `k` 在 `[1, 数组中不同元素的个数]` 范围内
- 题目数据保证答案唯一，换句话说，数组中前 `k` 个高频元素的集合是唯一的

**进阶：**你所设计算法的时间复杂度 **必须** 优于 `O(n log n)`，其中 `n` 是数组的大小。

## 解题引导

### 思路：哈希表 + 最小堆

#### 思考过程:

这个问题要求我们找到出现频率最高的 `k` 个元素。我们可以分两步解决：

- 1 **统计频率：**首先，我们需要统计数组中每个元素出现的频率。哈希表（在 Python 中是字典）是完成这个任务的理想数据结构。
- 2 **找出前 k 高频：**在得到频率后，问题就变成了“如何从一组（元素，频率）对中找出频率最高的前 `k` 个”。这和“数组中的第K个最大元素”非常相似。我们可以使用一个大小为 `k` 的最小堆来解决。

#### 算法流程:

1. 使用哈希表 `counts` 统计 `nums` 中每个元素的出现频率。
2. 创建一个最小堆 `min_heap`。堆中存储的是一个元组 (频率, 元素)。
3. 遍历 `counts` 哈希表中的每个 (元素, 频率) 对：
  - 如果堆的大小小于 `k`, 直接将 (频率, 元素) 推入堆中。
  - 如果堆的大小等于 `k`, 将当前元素的频率与堆顶元素的频率 (也就是当前堆中 `k` 个元素里频率最小的) 进行比较。
    - 如果当前频率大于堆顶频率, 则弹出堆顶元素, 并将新的 (频率, 元素) 推入堆中。
    - 否则, 不进行任何操作。
4. 遍历结束后, 堆中剩下的 `k` 个元素就是频率最高的前 `k` 个元素。将它们从堆中取出并返回。

复杂度分析:

- **时间复杂度:**  $O(n \log k)$ 。
  - 统计频率需要  $O(n)$ 。
  - 遍历哈希表中的  $m$  个不同元素 ( $m \leq n$ ), 并将它们与堆进行比较。每次堆操作的时间复杂度是  $O(\log k)$ , 总共是  $O(m \log k)$ 。
  - 因此, 总的时间复杂度是  $O(n + m \log k)$ 。在最坏的情况下  $m$  接近  $n$ , 所以是  $O(n \log k)$ 。
- **空间复杂度:**  $O(m + k)$ 。
  - 哈希表需要  $O(m)$  的空间来存储  $m$  个不同元素。
  - 堆需要  $O(k)$  的空间。

## Python 代码实现

```
from typing import List
import collections
import heapq

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # 1. 统计元素出现频率
        counts = collections.Counter(nums)

        # 2. 使用最小堆找出频率前 k 高的元素
        min_heap = []
        for num, freq in counts.items():
            if len(min_heap) < k:
                heapq.heappush(min_heap, (freq, num))
            elif freq > min_heap[0][0]:
                heapq.heapreplace(min_heap, (freq, num))

        # 3. 返回结果
        return [item[1] for item in min_heap]
```

## 多维动态规划 / 1143. 最长公共子序列

### 1143. 最长公共子序列

#### 题目描述

给定两个字符串 `text1` 和 `text2`, 返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列**, 返回 0。

一个字符串的 **子序列** 是指这样一个新的字符串: 它是由原字符串在不改变字符的相对顺序的情况下删除某些字符 (也可以不删除任何字符) 后组成的新字符串。

- 例如, "ace" 是 "abcde" 的子序列, 但 "aec" 不是 "abcde" 的子序列。

两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列。

#### 示例 1:

```
输入: text1 = "abcde", text2 = "ace"
输出: 3
解释: 最长公共子序列是 "ace" , 它的长度为 3 。
```

## 示例 2:

```
输入: text1 = "abc", text2 = "abc"
输出: 3
解释: 最长公共子序列是 "abc" , 它的长度为 3 。
```

## 示例 3:

```
输入: text1 = "abc", text2 = "def"
输出: 0
解释: 两个字符串没有公共子序列, 返回 0 。
```

## 提示:

- $1 \leq \text{text1.length}, \text{text2.length} \leq 1000$
- `text1` 和 `text2` 仅由小写英文字母组成。

## 解题引导

### 思路：动态规划

#### 思考过程：

这是一个经典的二维动态规划问题。

#### 状态定义：

`dp[i][j]`：表示 `text1` 的前  $i$  个字符 (`text1[0...i-1]`) 和 `text2` 的前  $j$  个字符 (`text2[0...j-1]`) 的最长公共子序列的长度。

#### 状态转移方程：

我们考虑 `text1[i-1]` 和 `text2[j-1]` 这两个字符：

1. 如果 `text1[i-1] == text2[j-1]`：

这两个字符是相同的，所以它们可以构成公共子序列的一部分。这个公共子序列的长度等于 `text1` 的前  $i-1$  个字符和 `text2` 的前  $j-1$  个字符的最长公共子序列长度加一。

```
dp[i][j] = dp[i-1][j-1] + 1
```

2. 如果 `text1[i-1] != text2[j-1]`：

这两个字符不相同，那么它们不能同时作为公共子序列的末尾。我们需要考虑两种情况，并取其中的最大值：

- `text1` 的前  $i$  个字符和 `text2` 的前  $j-1$  个字符的最长公共子序列（相当于忽略 `text2[j-1]`）。

- `text1` 的前  $i-1$  个字符和 `text2` 的前  $j$  个字符的最长公共子序列（相当于忽略 `text1[i-1]`）。

```
dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

#### 边界条件：

`dp` 表的大小是  $(m+1) \times (n+1)$ ，其中  $m$  和  $n$  分别是 `text1` 和 `text2` 的长度。`dp[0][j]` 和 `dp[i][0]` 都应该初始化为 0，因为空字符串和任何字符串的最长公共子序列长度都是 0。

#### 最终结果：

`dp[m][n]` 就是我们要求的答案。

#### 算法流程：

1. 获取 `text1` 和 `text2` 的长度  $m$  和  $n$ 。
2. 创建一个  $(m+1) \times (n+1)$  的 `dp` 表，并初始化为 0。
3. 使用两层循环遍历 `dp` 表， $i$  从 1 到  $m$ ， $j$  从 1 到  $n$ 。
4. 在循环中，根据 `text1[i-1]` 和 `text2[j-1]` 是否相等，应用上述状态转移方程来填充 `dp[i][j]`。
5. 返回 `dp[m][n]`。

#### 复杂度分析：

- 时间复杂度： $O(m * n)$ 。我们需要填充整个 `dp` 表。
- 空间复杂度： $O(m * n)$ 。需要一个二维数组来存储 `dp` 表。

## Python 代码实现

```

class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)

        # dp[i][j] 表示 text1[0...i-1] 和 text2[0...j-1] 的最长公共子序列长度
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                # 如果当前字符相同
                if text1[i-1] == text2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                # 如果当前字符不同
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])

        return dp[m][n]

```

## 多维动态规划 / 5. 最长回文子串

### 5. 最长回文子串

#### 题目描述

给你一个字符串  $s$ ，找到  $s$  中最长的回文子串。

#### 示例 1：

```

输入: s = "babad"
输出: "bab"
解释: "aba" 同样是符合题意的答案。

```

#### 示例 2：

```

输入: s = "cbbd"
输出: "bb"

```

#### 提示：

- $1 \leq s.length \leq 1000$
- $s$  仅由数字和英文字母组成

## 解题引导

### 思路：动态规划

#### 思考过程：

回文串有一个重要的特性：如果一个字符串是回文串，那么去掉它的首尾两个字符后，剩下的部分仍然是回文串。例如，“abccba”是回文串，“bccb”也是回文串。

这个特性启发我们使用动态规划来解决问题。我们可以定义一个二维布尔数组  $dp$ ，其中  $dp[i][j]$  表示字符串  $s$  从索引  $i$  到  $j$  的子串  $s[i..j]$  是否是回文串。

#### 状态定义：

$dp[i][j]$ ：表示子串  $s[i..j]$  是否是回文串。

#### 状态转移方程：

要判断  $dp[i][j]$  是否为  $true$ ，需要满足两个条件：

- $s[i]$  必须等于  $s[j]$ 。
- $s[i+1..j-1]$  也必须是回文串，即  $dp[i+1][j-1]$  为  $true$ 。

所以，状态转移方程是： $dp[i][j] = (s[i] == s[j]) \text{ and } dp[i+1][j-1]$

## 边界条件：

- 当子串长度为 1 时（即  $j - i + 1 = 1$  或  $i == j$ ），它一定是回文串， $dp[i][i] = true$ 。
- 当子串长度为 2 时（即  $j - i + 1 = 2$  或  $j = i + 1$ ），只要  $s[i] == s[j]$ ，它就是回文串， $dp[i][i+1] = (s[i] == s[i+1])$ 。

注意到，计算  $dp[i][j]$  需要依赖  $dp[i+1][j-1]$ ，这意味着我们需要先计算出短的子串的结果，再用它来计算长的子串。在填充  $dp$  表时，我们需要注意遍历顺序。

## 算法流程：

- 初始化一个  $n \times n$  的二维布尔数组  $dp$ ，其中  $n$  是字符串  $s$  的长度。
- 初始化变量  $start$  和  $max\_len$  来记录最长回文子串的起始位置和长度。
- 遍历所有可能的子串长度  $L$ （从 1 到  $n$ ）。
- 对于每个长度  $L$ ，遍历所有可能的起始位置  $i$ 。
- 计算结束位置  $j = i + L - 1$ 。
- 根据状态转移方程和边界条件计算  $dp[i][j]$  的值。
- 如果  $dp[i][j]$  为  $true$ ，并且当前的长度  $L$  大于  $max\_len$ ，就更新  $max\_len$  和  $start$ 。
- 最后，根据  $start$  和  $max\_len$  返回最长回文子串。

## 复杂度分析：

- 时间复杂度： $O(n^2)$ 。我们需要填充一个  $n \times n$  的  $dp$  表。
- 空间复杂度： $O(n^2)$ 。我们需要一个  $n \times n$  的  $dp$  表来存储状态。

## Python 代码实现

```
class Solution:  
    def longestPalindrome(self, s: str) -> str:  
        n = len(s)  
        if n < 2:  
            return s  
  
        dp = [[False] * n for _ in range(n)]  
  
        start = 0  
        max_len = 1  
  
        # 所有长度为 1 的子串都是回文串  
        for i in range(n):  
            dp[i][i] = True  
  
        # 遍历所有可能的子串长度  
        for L in range(2, n + 1):  
            # 遍历所有可能的起始位置  
            for i in range(n):  
                # 计算结束位置  
                j = i + L - 1  
                if j >= n:  
                    break  
  
                if s[i] != s[j]:  
                    dp[i][j] = False  
                else:  
                    # 如果子串长度小于等于 3，只要首尾相等就是回文  
                    if j - i < 3:  
                        dp[i][j] = True  
                    else:  
                        # 否则状态取决于内部子串  
                        dp[i][j] = dp[i+1][j-1]  
  
                # 如果找到了更长的回文子串，更新结果  
                if dp[i][j] and L > max_len:
```

```
max_len = L  
start = i  
  
return s[start:start + max_len]
```

## 多维动态规划 / 62. 不同路径

### 62. 不同路径

#### 题目描述

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
```

#### 示例 3：

```
输入: m = 7, n = 3  
输出: 28
```

#### 示例 4：

```
输入: m = 3, n = 3  
输出: 6
```

#### 提示：

- $1 \leq m, n \leq 100$
- 题目数据保证答案小于等于  $2 * 10^9$

## 解题引导

### 思路：动态规划

#### 思考过程：

这是一个经典的路径问题，非常适合用动态规划来解决。

机器人要到达网格中的任意一个点  $(i, j)$ ，它只能从两个方向过来：

1. 从它的上方，即  $(i-1, j)$ ，向下移动一步。
2. 从它的左方，即  $(i, j-1)$ ，向右移动一步。

因此，到达点  $(i, j)$  的总路径数，就等于到达点  $(i-1, j)$  的路径数与到达点  $(i, j-1)$  的路径数之和。

### 状态定义：

`dp[i][j]`：表示从左上角  $(0, 0)$  到达网格中点  $(i, j)$  的不同路径的总数。

### 状态转移方程：

`dp[i][j] = dp[i-1][j] + dp[i][j-1]`

### 边界条件（初始化）：

- 对于网格的第一行 ( $i = 0$ )，机器人只能一直向右走才能到达。所以，`dp[0][j] = 1` 对于所有的  $j$ 。
- 对于网格的第一列 ( $j = 0$ )，机器人只能一直向下走才能到达。所以，`dp[i][0] = 1` 对于所有的  $i$ 。
- `dp[0][0]` 既是第一行也是第一列，值为 1。

### 最终结果：

我们要求的是到达右下角  $(m-1, n-1)$  的路径数，所以结果是 `dp[m-1][n-1]`。

### 算法流程：

- 创建一个  $m \times n$  的二维数组 `dp`。
- 初始化第一行和第一列的所有值为 1。
- 使用两层循环，从  $(1, 1)$  开始遍历到  $(m-1, n-1)$ 。
- 在循环中，应用状态转移方程 `dp[i][j] = dp[i-1][j] + dp[i][j-1]`。
- 遍历结束后，返回 `dp[m-1][n-1]`。

### 空间优化：

注意到 `dp[i][j]` 的计算只依赖于当前行的左边元素和上一行的同一列元素。我们可以将二维 `dp` 数组压缩为一维数组。用一个大小为  $n$  的数组 `dp`，`dp[j]` 存储的是当前  $(i, j)$  位置的路径数。更新 `dp[j]` 时，`dp[j]` 的旧值代表 `dp[i-1][j]`，而 `dp[j-1]` 代表 `dp[i][j-1]`。所以转移方程变为 `dp[j] = dp[j] + dp[j-1]`。

### 复杂度分析（优化后）：

- 时间复杂度： $O(m * n)$ 。需要遍历整个网格。
- 空间复杂度： $O(n)$ 。使用了一维数组进行优化。

## Python 代码实现（空间优化）

```
class Solution:  
    def uniquePaths(self, m: int, n: int) -> int:  
        # 使用一维数组进行空间优化  
        # dp[j] 表示到达第 i 行第 j 列的路径数  
        dp = [1] * n  
  
        for i in range(1, m):  
            for j in range(1, n):  
                # dp[j] 的旧值是上一行的 dp[i-1][j]  
                # dp[j-1] 是当前行的 dp[i][j-1]  
                dp[j] = dp[j] + dp[j-1]  
  
        return dp[n-1]
```

## 多维动态规划 / 64. 最小路径和

### 64. 最小路径和

#### 题目描述

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。

说明：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。

#### 示例 2:

输入: `grid = [[1,2,3],[4,5,6]]`

输出: 12

#### 提示:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 200`
- `0 <= grid[i][j] <= 100`

## 解题引导

### 思路：动态规划

#### 思考过程：

这是一个典型的动态规划问题。我们要求从左上角到右下角的最小路径和。路径的每一步都只能向右或向下。

#### 状态定义：

`dp[i][j]`：表示从左上角 `(0, 0)` 到达网格位置 `(i, j)` 的最小路径和。

#### 状态转移方程：

要到达位置 `(i, j)`，我们只能从它的上方 `(i-1, j)` 或者左方 `(i, j-1)` 过来。

因此，到达 `(i, j)` 的最小路径和，就是从 `(i-1, j)` 过来的最小路径和与从 `(i, j-1)` 过来的最小路径和中的较小者，再加上当前位置 `grid[i][j]` 的值。

`dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]`

#### 边界条件：

• 第一行：对于第一行的任意位置 `(0, j)`，我们只能从左边过来。所以 `dp[0][j] = dp[0][j-1] + grid[0][j]`。

• 第一列：对于第一列的任意位置 `(i, 0)`，我们只能从上边过来。所以 `dp[i][0] = dp[i-1][0] + grid[i][0]`。

• 起点：`dp[0][0] = grid[0][0]`。

#### 空间优化：

我们可以发现，计算 `dp[i][j]` 只需要 `dp[i-1][j]` 和 `dp[i][j-1]` 的信息。这意味着我们实际上可以在原始的 `grid` 数组上直接进行修改，从而将空间复杂度优化到  $O(1)$ （不考虑输入数据本身占用的空间）。

#### 算法流程（空间优化后）：

1. **初始化第一行：** 从第二个元素开始，`grid[0][j] = grid[0][j] + grid[0][j-1]`。
2. **初始化第一列：** 从第二个元素开始，`grid[i][0] = grid[i][0] + grid[i-1][0]`。
3. **遍历剩余部分：** 从 `(1, 1)` 开始遍历网格的其余部分。
  - `grid[i][j] = min(grid[i-1][j], grid[i][j-1]) + grid[i][j]`。
4. **返回结果：** 最终，右下角 `grid[m-1][n-1]` 的值就是从左上角到右下角的最小路径和。

#### 复杂度分析：

- **时间复杂度：**  $O(m * n)$ 。我们需要遍历整个网格一次。
- **空间复杂度：**  $O(1)$ 。我们直接在输入数组上进行修改，没有使用额外的空间。

```

from typing import List

class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])

        # 初始化第一行
        for j in range(1, n):
            grid[0][j] += grid[0][j-1]

        # 初始化第一列
        for i in range(1, m):
            grid[i][0] += grid[i-1][0]

        # 遍历剩余的网格
        for i in range(1, m):
            for j in range(1, n):
                grid[i][j] += min(grid[i-1][j], grid[i][j-1])

        return grid[m-1][n-1]

```

## 多维动态规划 / 72. 编辑距离

### 72. 编辑距离

#### 题目描述

给你两个单词 word1 和 word2，请返回将 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 -> intention (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')

```

#### 提示：

- $0 \leq \text{word1.length}, \text{word2.length} \leq 500$

- `word1` 和 `word2` 由小写英文字母组成

## 解题引导

### 思路：动态规划

#### 思考过程：

这个问题是另一个经典的二维动态规划问题，通常被称为“编辑距离”问题。

#### 状态定义：

`dp[i][j]`：表示将 `word1` 的前 `i` 个字符 (`word1[0...i-1]`) 转换成 `word2` 的前 `j` 个字符 (`word2[0...j-1]`) 所需的最少操作数。

#### 状态转移方程：

我们考虑如何从 `word1` 的子串得到 `word2` 的子串。对于 `dp[i][j]`，我们关注 `word1[i-1]` 和 `word2[j-1]` 这两个字符：

1. 如果 `word1[i-1] == word2[j-1]`：

这两个字符是相同的，我们不需要对它们进行任何操作。所以，将 `word1[0...i-1]` 转换成 `word2[0...j-1]` 的操作数，就等于将 `word1[0...i-2]` 转换成 `word2[0...j-2]` 的操作数。

`dp[i][j] = dp[i-1][j-1]`

2. 如果 `word1[i-1] != word2[j-1]`：

这两个字符不相同，我们需要进行操作。有三种可能的操作可以达到 `dp[i][j]` 的状态：

- 替换：将 `word1[i-1]` 替换成 `word2[j-1]`。这个操作需要 1 步。之后，问题就变成了将 `word1[0...i-2]` 转换成 `word2[0...j-2]`。总操作数是 `dp[i-1][j-1] + 1`。

- 删除：将 `word1[i-1]` 删除。这个操作需要 1 步。之后，问题变成了将 `word1[0...i-2]` 转换成 `word2[0...j-1]`。总操作数是 `dp[i-1][j] + 1`。

- 插入：在 `word1[0...i-1]` 的末尾插入一个字符 `word2[j-1]`。这个操作需要 1 步。之后，问题变成了将 `word1[0...i-1]` 转换成 `word2[0...j-2]`。总操作数是 `dp[i][j-1] + 1`。

我们需要在这三种操作中选择代价最小的一个。

`dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1`

#### 边界条件：

- `dp[i][0]`：将 `word1` 的前 `i` 个字符转换成空字符串，需要 `i` 次删除操作。所以 `dp[i][0] = i`。

- `dp[0][j]`：将空字符串转换成 `word2` 的前 `j` 个字符，需要 `j` 次插入操作。所以 `dp[0][j] = j`。

#### 最终结果：

`dp[m][n]` 就是我们要求的答案，其中 `m` 和 `n` 分别是 `word1` 和 `word2` 的长度。

#### 算法流程：

1. 获取 `word1` 和 `word2` 的长度 `m` 和 `n`。
2. 创建一个 `(m+1) x (n+1)` 的 `dp` 表。
3. 初始化边界条件：`dp[i][0] = i` 和 `dp[0][j] = j`。
4. 使用两层循环遍历 `dp` 表，`i` 从 1 到 `m`，`j` 从 1 到 `n`。
5. 在循环中，根据 `word1[i-1]` 和 `word2[j-1]` 是否相等，应用上述状态转移方程来填充 `dp[i][j]`。
6. 返回 `dp[m][n]`。

#### 复杂度分析：

- 时间复杂度： $O(m * n)$ 。我们需要填充整个 `dp` 表。
- 空间复杂度： $O(m * n)$ 。需要一个二维数组来存储 `dp` 表。

## Python 代码实现

```
class Solution:  
    def minDistance(self, word1: str, word2: str) -> int:  
        m, n = len(word1), len(word2)  
  
        # dp[i][j] 表示 word1 的前 i 个字符转换成 word2 的前 j 个字符所需的最少操作数  
        dp = [[0] * (n + 1) for _ in range(m + 1)]  
  
        # 初始化边界  
        for i in range(m + 1):  
            dp[i][0] = i  
        for j in range(n + 1):
```

```

dp[0][j] = j

# 填充 dp 表
for i in range(1, m + 1):
    for j in range(1, n + 1):
        if word1[i-1] == word2[j-1]:
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = min(dp[i-1][j-1],      # 替换
                           dp[i-1][j],       # 删除
                           dp[i][j-1]) + 1 # 插入

return dp[m][n]

```

## 子串 / 239. 滑动窗口最大值

### 239. 滑动窗口最大值

#### 题目描述

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。你只可以看到在滑动窗口内的 `k` 个数字。每次滑动窗口向右移动一位。

返回 滑动窗口中的最大值 组成的数组。

#### 示例

##### 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>

##### 示例 2:

输入: `nums = [1]`, `k = 1`

输出: `[1]`

#### 约束条件

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- $1 \leq k \leq \text{nums.length}$

#### 思考过程

##### 💡 第一步：理解问题

- 我们有一个固定大小为 `k` 的窗口在数组 `nums` 上滑动。
- 每次窗口向右移动一格。
- 我们需要记录每个窗口状态下的最大值。
- 最终返回一个包含所有这些最大值的列表。

##### 💡 第二步：暴力解法

最直接的想法是，模拟窗口的滑动过程。

- 1 遍历 `s` 中所有可能的窗口起点（从 `0` 到 `n-k`）。
- 2 对于每个窗口，遍历其中的 `k` 个元素，找到最大值。

3. 将找到的最大值存入结果列表。

**思考题：**这个方法的时间复杂度是多少？对于  $n \leq 10^5$  是否可行？

► 点击查看分析

### 💡 第三步：寻找优化思路

暴力解法的瓶颈在于，每次窗口滑动时，我们都重新计算了最大值。而相邻的窗口大部分元素是重合的，这里面有大量的信息被浪费了。

我们需要一个数据结构，能够：

1. 随着窗口滑动，高效地加入新元素（窗口右侧）。
2. 随着窗口滑动，高效地删除旧元素（窗口左侧）。
3. 在  $O(1)$  时间内返回当前窗口的最大值。

**思考题：**

- 普通的队列能做到吗？（入队出队  $O(1)$ ，但找最大值  $O(k)$ ）
- 大顶堆（Priority Queue）能做到吗？（找最大值  $O(1)$ ，入队  $O(\log k)$ ，但删除指定元素  $O(k)$ ）

看来我们需要一个更特殊的数据结构。

### 💡 第四步：单调队列（Deque）

让我们设计一个特殊的队列，这个队列具有一个性质：队列内的元素从队头到队尾是单调递减的。

这个队列我们用双端队列（Deque）来实现，里面存储的是元素的索引，而不是元素本身。存储索引的好处是，我们既可以通过索引知道元素的值，也可以通过索引判断元素是否已经滑出窗口。

**这个单调队列如何工作？**

#### 1. 入队规则（从队尾入）：

当一个新元素  $\text{nums}[i]$  要进入队列时，我们从队尾开始，将所有比  $\text{nums}[i]$  小的元素的索引都弹出。然后再将  $i$  的索引加入队尾。

**为什么这么做？** 因为如果  $\text{nums}[j] < \text{nums}[i]$  且  $j < i$ ，那么只要  $\text{nums}[i]$  还在窗口内， $\text{nums}[j]$  就永远不可能成为最大值了，所以可以放心地把它从队列中扔掉。

#### 2. 出队规则（从队头出）：

队列的队头，始终保存着当前窗口内最大值的索引。

每次窗口滑动时，我们需要检查队头的索引是否已经过期（即  $\text{dequeue}[0] \leq i - k$ ）。如果过期了，就从队头弹出。

**算法流程：**

1. 初始化一个双端队列 `dequeue` 和一个结果列表 `result`。
2. 遍历 `nums` 数组，索引从  $i = 0$  到  $n-1$ ：
  - a. 维护单调性：当队列不为空，且队尾索引对应的元素小于等于  $\text{nums}[i]$  时，从队尾弹出。
  - b. 入队：将当前索引  $i$  加入队尾。
  - c. 检查队头是否过期：如果队头索引小于  $i - k + 1$ ，说明队头元素已经滑出窗口，从队头弹出。
  - d. 记录结果：当窗口形成后（即  $i \geq k - 1$ ），队头元素就是当前窗口的最大值。将  $\text{nums}[\text{dequeue}[0]]$  加入 `result` 列表。

**时间复杂度：**  $O(n)$  - 每个元素的索引最多入队一次，出队一次。

**空间复杂度：**  $O(k)$  - 双端队列中最多存储  $k$  个元素的索引。

## 代码实现

### Python

```
import collections

def maxSlidingWindow(nums: list[int], k: int) -> list[int]:
    """
    使用单调队列（双端队列）解决滑动窗口最大值问题。
    """

    if not nums or k == 0:
        return []

    # deque 存储的是元素的索引，并且索引对应的元素值是单调递减的
    deque = collections.deque()
    result = []

    for i, num in enumerate(nums):
        # 1. 移除过期的队头元素
        # i - k 是窗口前一个位置的索引，如果队头索引小于等于它，说明已滑出窗口
        if deque[0] <= i - k:
            deque.popleft()

        # 2. 将当前索引加入队尾
        deque.append(i)

        # 3. 当窗口形成后（i >= k - 1），队头元素就是当前窗口的最大值
        if i >= k - 1:
            result.append(nums[deque[0]])

    return result
```

```

if deque and deque[0] <= i - k:
    deque.popleft()

# 2. 维护队列的单调递减性
# 从队尾移除所有小于当前元素的索引
while deque and nums[deque[-1]] < num:
    deque.pop()

# 3. 将当前元素索引加入队列
deque.append(i)

# 4. 当窗口完全形成后，记录最大值
# 窗口形成是从索引 k-1 开始的
if i >= k - 1:
    # 此时的队头，就是当前窗口最大值的索引
    result.append(nums[deque[0]])

return result

```

## 关键点总结

- 单调队列：**是解决滑动窗口最值问题的“神器”。其核心思想是，在队列中只保留那些“有潜力”成为未来最大值的元素，从而避免了重复的比较。
- 存储索引：**在队列中存储索引而不是元素本身，是一个非常关键的技巧。索引既能告诉我们元素的值，也能告诉我们元素的位置，方便判断其是否过期。
- 均摊复杂度：**虽然 `while` 循环看起来可能会增加复杂度，但每个元素最多只入队和出队一次，所以总的时间复杂度是均摊  $O(n)$  的。

## 子串 / 560. 和为 K 的子数组

### 560. 和为 K 的子数组

#### 题目描述

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 子数组 的个数。

**子数组** 是数组中元素的连续非空序列。

#### 示例

示例 1：

输入： `nums = [1,1,1]`, `k = 2`  
 输出： 2  
 解释： 和为 2 的子数组是 `[1,1]` 和 `[1,1]`。

示例 2：

输入： `nums = [1,2,3]`, `k = 3`  
 输出： 2  
 解释： 和为 3 的子数组是 `[1,2]` 和 `[3]`。

#### 约束条件

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^7 \leq k \leq 10^7$

#### 思考过程

##### 💡 第一步：理解问题

- 我们要找的是 **连续** 的子数组。
- 这些子数组的元素之和需要恰好等于 `k`。
- 我们需要返回的是满足条件的子数组的 **个数**。
- 重要提示：** `nums` 数组中包含负数！这一点非常关键。

##### 💡 第二步：暴力解法

最直接的想法是，找出所有可能的子数组，计算它们的和，然后看是否等于  $k$ 。

思考题：

- 如何用循环实现找到所有子数组？
- 这个方法的时间复杂度是多少？

► 点击查看分析

### 💡 第三步：为什么滑动窗口不行？

一看到“连续子数组”，我们可能会想到滑动窗口。但是，标准的滑动窗口方法在这里行不通。

思考题：为什么滑动窗口在这里会失效？

► 点击查看分析

### 💡 第四步：前缀和 + 哈希表

既然滑动窗口不行，我们需要换个思路。 $O(n^2)$  的瓶颈在于，对于每个起点  $i$ ，我们都重新计算了到终点  $j$  的和。

核心思路：

- 子数组  $\text{nums}[i:j+1]$  的和，可以用前缀和来快速计算。
- 设  $\text{prefix\_sum}[x]$  为  $\text{nums}[0 \dots x]$  的和。
- 那么，子数组  $\text{nums}[i \dots j]$  的和就等于  $\text{prefix\_sum}[j] - \text{prefix\_sum}[i-1]$ 。

我们的目标是找到满足下面条件的  $(i, j)$  对的数量：

```
prefix_sum[j] - prefix_sum[i-1] == k
```

把这个公式变形一下：

```
prefix_sum[i-1] == prefix_sum[j] - k
```

这个公式给了我们一个绝妙的思路：

- 当我们遍历到  $j$  时，我们计算出了当前的  $\text{prefix\_sum}[j]$ 。
- 此时，我们不再需要回头去找  $i$ 。
- 我们只需要问一个问题：“在  $j$  之前，有多少个  $i-1$ ，使得  $\text{prefix\_sum}[i-1]$  恰好等于  $\text{prefix\_sum}[j] - k$ ？”

这个问题可以用一个哈希表来完美解决！

### 💡 第五步：算法步骤总结

1. 初始化一个哈希表  $\text{prefix\_sum\_map}$ ，用来存储某个前缀和出现的次数。 $\text{key}$  是前缀和的值， $\text{value}$  是该值出现的次数。
2. 为了处理从索引 0 开始的子数组（即  $i=0$  的情况），我们需要一个  $\text{prefix\_sum}[-1]$ ，它的值为 0。所以，我们在哈希表中预先放入  $\{0: 1\}$ 。
3. 初始化  $\text{current\_sum} = 0$ ,  $\text{count} = 0$ 。
4. 遍历数组  $\text{nums}$  中的每个数字  $\text{num}$ ：
  - a. 更新当前的前缀和： $\text{current\_sum} += \text{num}$ 。
  - b. 查找目标：在哈希表中查找  $\text{target} = \text{current\_sum} - k$ 。如果找到了，说明存在若干个以当前位置为终点的子数组，其和为  $k$ 。我们将这些子数组的数量加入  $\text{count}$ ： $\text{count} += \text{prefix\_sum\_map.get(target, 0)}$ 。
  - c. 更新哈希表：将当前的前缀和  $\text{current\_sum}$  存入哈希表（或更新其出现次数）： $\text{prefix\_sum\_map}[\text{current\_sum}] = \text{prefix\_sum\_map.get(current\_sum, 0)} + 1$ 。
5. 遍历结束后，返回  $\text{count}$ 。

时间复杂度： $O(n)$  - 我们只遍历一次数组。

空间复杂度： $O(n)$  - 在最坏情况下，所有前缀和都不同，哈希表需要存储  $n$  个键值对。

## 代码实现

### Python

```
def subarraySum(nums: list[int], k: int) -> int:  
    """  
    使用前缀和 + 哈希表统计和为 k 的子数组个数。  
    """  
  
    # prefix_sum_map: {前缀和: 出现次数}  
    prefix_sum_map = {0: 1} # 初始化，处理从索引0开始的子数组  
    count = 0  
    current_sum = 0  
  
    for num in nums:  
        # 1. 计算当前的前缀和  
        current_sum += num  
  
        # 2. 查找 target = current_sum - k  
        if current_sum - k in prefix_sum_map:  
            count += prefix_sum_map[current_sum - k]  
  
        # 3. 将当前的前缀和存入哈希表  
        prefix_sum_map[current_sum] = prefix_sum_map.get(current_sum, 0) + 1
```

```

# 2. 寻找目标前缀和
# 我们需要找的前缀和是 current_sum - k
target = current_sum - k
if target in prefix_sum_map:
    count += prefix_sum_map[target]

# 3. 将当前前缀和存入哈希表
prefix_sum_map[current_sum] = prefix_sum_map.get(current_sum, 0) + 1

return count

```

## 关键点总结

- 前缀和：**是解决子数组和相关问题的利器，能将求任意子数组和的时间复杂度从  $O(n)$  降到  $O(1)$ 。
- 哈希表优化：**通过 `prefix_sum[i-1] == prefix_sum[j] - k` 的转换，将两层循环的查找问题变成了  $O(1)$  的哈希表查找，是本题从  $O(n^2)$  优化到  $O(n)$  的关键。
- 负数的重要性：**题目中的负数使得问题不能用常规的滑动窗口解决，从而引导我们走向前缀和的思路。
- 初始化 {0: 1}：**这个细节非常重要，它相当于一个虚拟的 `prefix_sum[-1] = 0`，用于正确统计那些从数组开头 `nums[0]` 就开始且和为 `k` 的子数组。

## 子串 / 76. 最小覆盖子串

### 76. 最小覆盖子串

#### 题目描述

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 `""`。

注意：

- 对于 `t` 中重复字符，我们寻找的子串中该字符数量必须不少于 `t` 中该字符数量。
- 如果 `s` 中存在这样的子串，我们保证它是唯一的。

#### 示例

示例 1：

输入： `s = "ADOBECODEBANC"`, `t = "ABC"`

输出： `"BANC"`

解释： 最小覆盖子串 `"BANC"` 包含来自字符串 `t` 的 `'A'`、`'B'` 和 `'C'`。

示例 2：

输入： `s = "a"`, `t = "a"`

输出： `"a"`

示例 3：

输入： `s = "a"`, `t = "aa"`

输出： `""`

解释： `t` 中两个 `'a'` 无法匹配。`s` 中不存在这样的子串。

#### 约束条件

- $1 \leq s.length, t.length \leq 10^5$
- `s` 和 `t` 由英文字母组成

#### 思考过程

##### 💡 第一步：理解问题

- 我们要在字符串 `s` 中找一个最短的连续子串。
- 这个子串必须包含字符串 `t` 中的所有字符。
- “包含”的意思是，对于 `t` 中的每个字符，它在子串中出现的次数必须大于或等于它在 `t` 中出现的次数。
- 如果找不到，返回空字符串。

##### 💡 第二步：朴素解法

遍历 `s` 中所有的子串，检查每个子串是否“覆盖”了 `t`，然后在所有覆盖的子串中找到最短的那个。

## 思考题：

- 如何检查一个子串是否“覆盖”了 `t`？
- 这个方法的时间复杂度是多少？

► 点击查看分析

## 💡 第三步：滑动窗口

这个问题是滑动窗口思想的完美应用场景。

### 核心思路：

- 我们用一个窗口在 `s` 上滑动。
- 先不断**扩大**窗口（移动右指针 `right`），直到窗口内的子串**刚好**覆盖了 `t` 中所有的字符。
- 一旦覆盖，我们就找到了一个**可行解**。但它不一定是最短的。所以，我们现在尝试**收缩**窗口（移动左指针 `left`），看看在**保持覆盖**的前提下，能把这个子串缩到多短。
- 收缩直到窗口内的子串**不再**覆盖 `t` 为止。此时，我们就得到了以这个 `left` 为起点的最短可行解。
- 重复步骤 2-4，直到 `right` 指针遍历完整个 `s`。

## 💡 第四步：滑动窗口算法细节

我们需要一些辅助工具来实现这个逻辑：

- `t_freq` 哈希表：用来存储 `t` 中所需字符的频率。
- `window_freq` 哈希表：用来存储当前窗口中，我们**关心**的字符（即在 `t` 中出现的字符）的频率。
- `valid_chars` 计数器：记录 `window_freq` 中有多少个字符的频率已经满足了 `t_freq` 的要求。当 `valid_chars` 的值等于 `t_freq` 中不同字符的总数时，就说明窗口已经完全覆盖了 `t`。

### 算法流程：

- 初始化 `t_freq`, `window_freq` 两个哈希表。
- 统计 `t` 的字符频率到 `t_freq`。
- 初始化 `left = 0`, `valid_chars = 0`。
- 初始化 `min_len = infinity`, `start_index = -1` 用于记录最短子串的长度和起点。
- `right` 指针遍历 `s` (从 0 到 `n-1`):
  - 扩大窗口：**将 `char_in = s[right]` 移入窗口。
  - 如果 `char_in` 是 `t` 中需要的字符，则更新 `window_freq`。如果更新后 `window_freq[char_in]` 恰好等于 `t_freq[char_in]`，说明一个种类的字符已经满足要求，`valid_chars++`。
  - 检查是否需要收缩：**当 `valid_chars` 等于 `t_freq` 中不同字符的总数时，进入一个 `while` 循环开始收缩左边界。
    - 更新结果：**当前窗口 `s[left:right+1]` 是一个可行解。计算其长度，如果比 `min_len` 更短，则更新 `min_len` 和 `start_index`。
    - 收缩窗口：**将 `char_out = s[left]` 移出窗口。
    - 如果 `char_out` 是 `t` 中需要的字符，则更新 `window_freq`。如果更新前 `window_freq[char_out]` 恰好等于 `t_freq[char_out]`，说明一个满足要求的字符即将离开窗口，覆盖状态被破坏，所以 `valid_chars--`。
    - `left++`，收缩窗口。
- 遍历结束后，如果 `start_index` 仍为 -1，说明没找到，返回 `""`。否则，返回 `s[start_index : start_index + min_len]`。

时间复杂度：O(N+M) - N是s的长度，M是t的长度。 `left` 和 `right` 指针都只遍历一次 `s`。

空间复杂度：O(K) - K是字符集的大小。最坏情况下是 O(N+M)。

## 代码实现

### Python

```
import collections

def minWindow(s: str, t: str) -> str:
    """
    使用滑动窗口寻找最小覆盖子串。
    """

    if not t or not s or len(s) < len(t):
        return ""

    # 1. 初始化 t 的频率表和窗口的频率表
    t_freq = collections.Counter(t)
    window_freq = collections.Counter()

    # `required_chars` 是 t 中不同字符的数量
    required_chars = len(t_freq)
```

```

# `valid_chars` 是窗口中已满足频率要求的字符数量
valid_chars = 0

# 用于记录最短子串的长度和起始索引
min_len = float('inf')
start_index = -1

left = 0
for right, char in enumerate(s):
    # 2. 扩大窗口
    if char in t_freq:
        window_freq[char] += 1
        if window_freq[char] == t_freq[char]:
            valid_chars += 1

    # 3. 当窗口满足条件时，开始收缩
    while valid_chars == required_chars:
        # a. 更新最小子串记录
        current_len = right - left + 1
        if current_len < min_len:
            min_len = current_len
            start_index = left

        # b. 移出左边界字符
        left_char = s[left]
        if left_char in t_freq:
            # 如果移出的字符是关键字符，并且它的数量正好是达标数量
            # 那么移出后，达标字符数就要减一
            if window_freq[left_char] == t_freq[left_char]:
                valid_chars -= 1
                window_freq[left_char] -= 1

        # c. 收缩窗口
        left += 1

    return "" if start_index == -1 else s[start_index : start_index + min_len]

```

## 关键点总结

- 滑动窗口框架：**本题是滑动窗口思想的一个集大成者，完整地体现了“扩大窗口以满足条件 -> 收缩窗口以寻找最优解”的核心循环。
- 需求与计数：**使用哈希表 (`t_freq`) 来定义“需求”，用另一个哈希表 (`window_freq`) 和计数器 (`valid_chars`) 来跟踪当前窗口的“满足”情况，是实现算法的关键。
- 收缩条件：**`while (valid_chars == required_chars)` 是整个算法的精髓。它确保了我们总是在一个“可行解”的窗口中去尝试寻找“最优解”，直到这个解不再可行 (`valid_chars` 减少)，然后再次进入扩大窗口的阶段。

## 技巧 / 136. 只出现一次的数字

### 136. 只出现一次的数字

#### 题目描述

给你一个非空整数数组 `nums`，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。

你必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

#### 示例 1：

```

输入: nums = [2,2,1]
输出: 1

```

#### 示例 2：

```
输入: nums = [4,1,2,1,2]
输出: 4
```

#### 示例 3:

```
输入: nums = [1]
输出: 1
```

#### 提示:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-3 \cdot 10^4 \leq \text{nums}[i] \leq 3 \cdot 10^4$
- 除了某个元素只出现一次以外，其余每个元素均出现两次。

#### 思考过程:

这道题要求在一个非空整数数组中，除了某个元素只出现一次以外，其余每个元素均出现两次。我们需要找出那个只出现了一次的元素。

#### 核心思路 (位运算 - 异或):

异或运算  $\wedge$  有以下特性：

1.  $a \wedge 0 = a$
2.  $a \wedge a = 0$
3. 异或运算满足交换律和结合律： $a \wedge b \wedge a = (a \wedge a) \wedge b = 0 \wedge b = b$

利用这些特性，我们可以将数组中所有元素进行异或运算。所有出现两次的元素最终会相互抵消为 0，而只出现一次的元素会保留下来。

#### 具体步骤:

1. 初始化一个变量 `result = 0`。
2. 遍历数组 `nums` 中的每一个数字 `num`。
3. 将 `result` 与 `num` 进行异或运算：`result = result  $\wedge$  num`。
4. 遍历结束后，`result` 就是那个只出现一次的数字。

#### Python 代码:

```
class Solution:
    def singleNumber(self, nums: list[int]) -> int:
        result = 0
        for num in nums:
            result ^= num
        return result
```

---

## 技巧 / 169. 多数元素

### 169. 多数元素

#### 题目描述

给定一个大小为  $n$  的数组 `nums`，返回其中的多数元素。多数元素是指在数组中出现次数大于  $\lfloor n/2 \rfloor$  的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

#### 示例 1:

```
输入: nums = [3,2,3]
输出: 3
```

#### 示例 2:

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

### 提示:

- $n == \text{nums.length}$
- $1 \leq n \leq 5 * 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

进阶: 尝试设计时间复杂度为  $O(n)$ 、空间复杂度为  $O(1)$  的算法解决此问题。

### 思考过程:

这道题要求找到数组中出现次数超过  $\lfloor n/2 \rfloor$  的元素。

### 核心思路 (摩尔投票法):

摩尔投票法是一种非常巧妙的算法, 用于在  $O(n)$  时间复杂度和  $O(1)$  空间复杂度下找到数组中的多数元素。

### 算法思想:

#### 1. 候选人 (candidate) 和计数器 (count):

- 初始化 `candidate` 为数组的第一个元素, `count` 为 1。

#### 2. 遍历数组:

- 如果当前元素与 `candidate` 相同, `count` 加 1。
- 如果当前元素与 `candidate` 不同:
  - `count` 减 1。
  - 如果 `count` 变为 0, 说明当前的 `candidate` 已经被抵消完了, 需要更换新的 `candidate` 为当前元素, 并将 `count` 重置为 1。

#### 3. 返回 `candidate`。

### 为什么摩尔投票法有效?

因为多数元素的出现次数超过  $\lfloor n/2 \rfloor$ , 这意味着它的出现次数比其他所有元素的出现次数之和还要多。在抵消过程中, 多数元素最终会“胜出”。

### Python 代码:

```
class Solution:
    def majorityElement(self, nums: list[int]) -> int:
        candidate = nums[0]
        count = 1

        for i in range(1, len(nums)):
            if nums[i] == candidate:
                count += 1
            else:
                count -= 1
                if count == 0:
                    candidate = nums[i]
                    count = 1

        return candidate
```

---

## 技巧 / 287. 寻找重复数

### 287. 寻找重复数

#### 题目描述

给定一个包含  $n + 1$  个整数的数组 `nums`, 其数字都在  $[1, n]$  范围内 (包括 1 和  $n$ ), 可知至少存在一个重复的整数。

假设 `nums` 只有一个重复的整数, 返回这个重复的数。

你设计的解决方案必须 不修改 数组 `nums` 且只用常量级  $O(1)$  的额外空间。

#### 示例 1:

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

#### 示例 2:

```
输入: nums = [3,1,3,4,2]
输出: 3
```

#### 提示:

- $1 \leq n \leq 10^5$
- $\text{nums.length} == n + 1$
- $1 \leq \text{nums}[i] \leq n$
- $\text{nums}$  中只有一个整数出现两次或多次，其余整数均只出现一次

#### 进阶:

- 如何证明  $\text{nums}$  中至少存在一个重复的数字？
- 你可以设计一个线性级时间复杂度  $O(n)$  的解决方案吗？

#### 思考过程:

这道题要求在一个包含  $n + 1$  个整数的数组  $\text{nums}$  中找到唯一的重复数。数组中的整数都在  $1$  到  $n$  之间（包括  $1$  和  $n$ ）。

#### 核心思路 (快慢指针 - 类似链表找环):

这道题可以被看作是一个链表找环的问题。

- 将数组的索引看作链表的节点， $\text{nums}[i]$  的值看作  $i$  指向的下一个节点。
- 由于数组中有重复的数字，这意味着至少有两个索引指向同一个值，从而形成一个环。
- 例如，如果  $\text{nums} = [1, 3, 4, 2, 2]$ ：
  - $0 \rightarrow \text{nums}[0] = 1$
  - $1 \rightarrow \text{nums}[1] = 3$
  - $2 \rightarrow \text{nums}[2] = 4$
  - $3 \rightarrow \text{nums}[3] = 2$
  - $4 \rightarrow \text{nums}[4] = 2$
  - 这里  $3 \rightarrow 2$  和  $4 \rightarrow 2$  形成了环。

#### 具体步骤 (弗洛伊德的循环查找算法):

##### 1. 找到环中的相遇点:

- 使用快慢指针。 $\text{slow}$  每次走一步， $\text{fast}$  每次走两步。
- $\text{slow} = \text{nums}[0]$
- $\text{fast} = \text{nums}[\text{nums}[0]]$
- 循环直到  $\text{slow} == \text{fast}$ 。

##### 2. 找到环的入口点 (即重复的数字):

- 将  $\text{slow}$  重新指向数组的开头 ( $\text{slow} = 0$ )。
- $\text{slow}$  和  $\text{fast}$  同时每次走一步。
- 它们再次相遇的点就是环的入口点，也就是重复的数字。

#### Python 代码:

```
class Solution:
    def findDuplicate(self, nums: list[int]) -> int:
        # 找到环中的相遇点
        slow = nums[0]
        fast = nums[nums[0]]

        while slow != fast:
            slow = nums[slow]
            fast = nums[nums[fast]]


        # 找到环的入口点
        slow = 0
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]

        return slow
```

## 技巧 / 31. 下一个排列

### 31. 下一个排列

#### 题目描述

整数数组的一个 排列 就是将其所有成员以序列或线性顺序排列。

例如，`arr = [1,2,3]`，以下这些都可以视作 `arr` 的排列：[1,2,3]、[1,3,2]、[3,1,2]、[2,3,1]。

整数数组的 下一个排列 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 下一个排列 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

例如，`arr = [1,2,3]` 的下一个排列是 [1,3,2]。

类似地，`arr = [2,3,1]` 的下一个排列是 [3,1,2]。

而 `arr = [3,2,1]` 的下一个排列是 [1,2,3]，因为 [3,2,1] 不存在一个字典序更大的排列。

给你一个整数数组 `nums`，找出 `nums` 的下一个排列。

必须 原地 修改，只允许使用额外常数空间。

#### 示例 1：

输入: `nums = [1, 2, 3]`

输出: [1, 3, 2]

#### 示例 2：

输入: `nums = [3, 2, 1]`

输出: [1, 2, 3]

#### 示例 3：

输入: `nums = [1, 1, 5]`

输出: [1, 5, 1]

#### 提示：

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 100$

#### 思考过程：

这道题要求找到给定整数数组的下一个字典序更大的排列。如果不存在下一个更大的排列，则将其重新排列成最小的排列（即升序排列）。

#### 核心思路：

1. 从右向左找到第一个下降的元素 `i`：
  - 从数组的倒数第二个元素开始向前遍历，找到第一个满足 `nums[i] < nums[i+1]` 的元素 `nums[i]`。
  - 如果找不到这样的 `i`（即数组是完全降序的），说明当前已经是最大的排列，直接将整个数组反转即可得到最小排列。
2. 从右向左找到第一个比 `nums[i]` 大的元素 `j`：
  - 从数组的最后一个元素开始向前遍历，找到第一个满足 `nums[j] > nums[i]` 的元素 `nums[j]`。
3. 交换 `nums[i]` 和 `nums[j]`。
4. 反转 `nums[i+1]` 到数组末尾的子数组：
  - 这样可以确保 `nums[i+1]` 到末尾的子数组是升序的，从而使得整个排列是下一个字典序更大的排列。

#### Python 代码：

```
class Solution:  
    def nextPermutation(self, nums: list[int]) -> None:  
        """  
        Do not return anything, modify nums in-place instead.  
        """  
        n = len(nums)  
  
        # 1. 从右向左找到第一个下降的元素 i  
        i = n - 2
```

```

while i >= 0 and nums[i] >= nums[i+1]:
    i -= 1

# 如果找到了下降的元素 i
if i >= 0:
    # 2. 从右向左找到第一个比 nums[i] 大的元素 j
    j = n - 1
    while j >= 0 and nums[j] <= nums[i]:
        j -= 1
    # 3. 交换 nums[i] 和 nums[j]
    nums[i], nums[j] = nums[j], nums[i]

# 4. 反转 nums[i+1] 到数组末尾的子数组
# 如果没有找到下降的元素 i, 说明整个数组是降序的, 直接反转整个数组
left, right = i + 1, n - 1
while left < right:
    nums[left], nums[right] = nums[right], nums[left]
    left += 1
    right -= 1

```

## 技巧 / 75. 颜色分类

### 75. 颜色分类

#### 题目描述

给定一个包含红色、白色和蓝色、共  $n$  个元素的数组  $\text{nums}$ ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

必须在不使用库的 `sort` 函数的情况下解决这个问题。

#### 示例 1：

```

输入: nums = [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]

```

#### 示例 2：

```

输入: nums = [2,0,1]
输出: [0,1,2]

```

#### 提示：

- $n == \text{nums.length}$
- $1 \leq n \leq 300$
- $\text{nums}[i]$  为 0、1 或 2

#### 进阶：

- 你可以不使用代码库中的排序函数来解决这道题吗？
- 你能想出一个仅使用常数空间的一趟扫描算法吗？

#### 思考过程：

这道题要求将一个包含红色、白色、蓝色三种元素的数组进行排序，使得相同颜色的元素排在一起，并且颜色顺序为红、白、蓝。可以使用整数 0、1 和 2 分别表示红色、白色和蓝色。

#### 核心思路 (双指针)：

这道题可以使用双指针（或三指针）的方法在一次遍历中完成排序。

##### 1. 定义指针：

- $p_0$ ：指向下一个 0 应该放置的位置（红色区域的右边界）。
- $p_2$ ：指向下一个 2 应该放置的位置（蓝色区域的左边界）。

- curr: 当前遍历的元素。

## 2. 初始化:

- p0 = 0
- p2 = len(nums) - 1
- curr = 0

3. 遍历数组: 当 curr <= p2 时, 进行循环。

- 如果 nums[curr] == 0 (红色):
  - 将 nums[curr] 与 nums[p0] 交换。
  - p0 右移一位。
  - curr 右移一位。
- 如果 nums[curr] == 2 (蓝色):
  - 将 nums[curr] 与 nums[p2] 交换。
  - p2 左移一位。
  - 注意: 此时 curr 不动, 因为交换过来的 nums[curr] 可能是 0 或 1, 需要再次判断。
- 如果 nums[curr] == 1 (白色):
  - curr 右移一位。

Python 代码:

```
class Solution:  
    def sortColors(self, nums: list[int]) -> None:  
        """  
        Do not return anything, modify nums in-place instead.  
        """  
        p0 = 0 # 指向下一个0应该放置的位置  
        p2 = len(nums) - 1 # 指向下一个2应该放置的位置  
        curr = 0 # 当前遍历的元素  
  
        while curr <= p2:  
            if nums[curr] == 0:  
                nums[curr], nums[p0] = nums[p0], nums[curr]  
                p0 += 1  
                curr += 1  
            elif nums[curr] == 2:  
                nums[curr], nums[p2] = nums[p2], nums[curr]  
                p2 -= 1  
                # 注意: 此时 curr 不动, 因为交换过来的 nums[curr] 可能是 0 或 1, 需要再次判断  
            else: # nums[curr] == 1  
                curr += 1
```

---

## 普通数组 / 189. 轮转数组

### 189. 轮转数组

#### 题目描述

给定一个整数数组 nums, 将数组中的元素向右轮转 k 个位置, 其中 k 是非负数。

#### 示例

示例 1:

输入: nums = [1,2,3,4,5,6,7], k = 3

输出: [5,6,7,1,2,3,4]

解释:

向右轮转 1 步: [7,1,2,3,4,5,6]

向右轮转 2 步: [6,7,1,2,3,4,5]

向右轮转 3 步: [5,6,7,1,2,3,4]

示例 2:

输入: nums = [-1,-100,3,99], k = 2

```
输出: [3,99,-1,-100]
```

解释:

向右轮转 1 步: [99,-1,-100,3]

向右轮转 2 步: [3,99,-1,-100]

## 约束条件

- $1 \leq \text{nums.length} \leq 10^5$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $0 \leq k \leq 10^5$

## 思考过程

### 💡 第一步：理解问题

- 我们需要将数组 `nums` 中的元素向右移动 `k` 个位置。
- 移动是“轮转”的，这意味着末尾的元素会移动到开头。
- `k` 可以是非负数，并且可能大于数组的长度。

### 💡 第二步：处理 `k` 的有效值

如果 `k` 大于 `nums` 的长度，那么实际上只需要移动 `k % len(nums)` 个位置。例如，长度为 7 的数组，向右轮转 7 步，相当于没有轮转；向右轮转 10 步，相当于向右轮转 3 步。

所以，首先将 `k = k % len(nums)`。

### 💡 第三步：暴力解法（额外数组）

最直观的方法是创建一个新的数组，然后将原数组的元素按照轮转后的顺序放入新数组。

思考题：

- 如何确定每个元素在新数组中的位置？
- 这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

### 💡 第四步：原地解法（多次反转）

题目通常会要求原地操作，即空间复杂度为  $O(1)$ 。

考虑一个数组 `[1,2,3,4,5,6,7]`，`k = 3`。

目标是 `[5,6,7,1,2,3,4]`。

我们可以观察到，轮转后，数组被分成了两部分：

- 后 `k` 个元素 `[5,6,7]` 移动到了前面。
- 前 `n-k` 个元素 `[1,2,3,4]` 移动到了后面。

这启发我们使用三次反转的方法：

1. 反转整个数组: `[7,6,5,4,3,2,1]`
2. 反转前 `k` 个元素: `[5,6,7,4,3,2,1]` (即 `[7,6,5]` 反转为 `[5,6,7]`)。
3. 反转后 `n-k` 个元素: `[5,6,7,1,2,3,4]` (即 `[4,3,2,1]` 反转为 `[1,2,3,4]`)。

思考题：为什么这个方法有效？

▶ 点击查看分析

### 💡 第五步：算法步骤总结（三次反转）

1. 计算有效的轮转步数 `k = k % len(nums)`。
2. 定义一个辅助函数 `reverse(arr, start, end)`，用于反转数组 `arr` 中从 `start` 到 `end` (包含) 的元素。
3. 第一次反转：反转整个数组 `nums`，即 `reverse(nums, 0, n-1)`。
4. 第二次反转：反转前 `k` 个元素，即 `reverse(nums, 0, k-1)`。
5. 第三次反转：反转后 `n-k` 个元素，即 `reverse(nums, k, n-1)`。

时间复杂度:  $O(n)$  - 每次反转都是  $O(n)$ ，总共三次。

空间复杂度:  $O(1)$  - 原地操作。

## 代码实现

### Python

```

def rotate(nums: list[int], k: int) -> None:
    """
    使用三次反转法原地轮转数组。
    Do not return anything, modify nums in-place instead.
    """
    n = len(nums)
    k = k % n # 处理 k 大于 n 的情况

    # 辅助函数：反转数组的一部分
    def reverse(arr, start, end):
        while start < end:
            arr[start], arr[end] = arr[end], arr[start]
            start += 1
            end -= 1

    # 1. 反转整个数组
    reverse(nums, 0, n - 1)

    # 2. 反转前 k 个元素
    reverse(nums, 0, k - 1)

    # 3. 反转后 n-k 个元素
    reverse(nums, k, n - 1)

```

## 关键点总结

1. **k 的有效值**: 首先对 `k` 取模，确保 `k` 在有效范围内。
2. **三次反转**: 这是一个非常巧妙且高效的方法，将数组的两个部分分别反转，再整体反转，最终达到轮转的效果。
3. **原地操作**: 通过直接修改原数组，避免了额外的空间开销，满足了题目对空间复杂度的要求。

## 普通数组 / 238. 除自身以外数组的乘积

### 238. 除自身以外数组的乘积

#### 题目描述

给你一个整数数组 `nums`，返回一个数组 `answer`，其中 `answer[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

题目数据保证数组 `nums` 之中任意元素的全部前缀元素和后缀元素的乘积都在 32 位整数范围内。

请不要使用除法，且时间复杂度为  $O(n)$ 。

#### 示例

示例 1:

输入: `nums = [1,2,3,4]`  
输出: `[24,12,8,6]`

示例 2:

输入: `nums = [-1,1,0,-3,3]`  
输出: `[0,0,9,0,0]`

#### 约束条件

- $2 \leq \text{nums.length} \leq 10^5$
- $-30 \leq \text{nums}[i] \leq 30$
- 题目数据保证数组 `nums` 之中任意元素的全部前缀元素和后缀元素的乘积都在 32 位整数范围内。

#### 思考过程

##### 💡 第一步：理解问题

- 对于数组 `nums` 中的每个元素 `nums[i]`，我们需要计算一个乘积，这个乘积是 `nums` 中除了 `nums[i]` 之外所有元素的乘积。

- 限制条件：不能使用除法，时间复杂度必须是  $O(n)$ 。

## 💡 第二步：暴力解法

最直接的想法是，对于每个 `nums[i]`，遍历整个数组，跳过 `nums[i]`，然后将其他元素相乘。

思考题：这个方法的时间复杂度是多少？

▶ 点击查看分析

## 💡 第三步： $O(n)$ 思路 - 前缀积和后缀积

$O(n^2)$  太慢，我们需要  $O(n)$  的解法。不能用除法，意味着我们不能先算出总乘积，再除以 `nums[i]`。

让我们考虑 `answer[i]` 的构成：

```
answer[i] = (nums[0] * ... * nums[i-1]) * (nums[i+1] * ... * nums[n-1])
```

这可以分解为两部分：

1. `nums[i]` 左边所有元素的乘积（前缀积）。
2. `nums[i]` 右边所有元素的乘积（后缀积）。

思考题：

- 如何高效地计算所有元素的前缀积？
- 如何高效地计算所有元素的后缀积？

▶ 点击查看分析

## 💡 第四步：算法步骤总结 ( $O(n)$ 时间, $O(n)$ 空间)

1. 初始化一个结果数组 `answer`，大小与 `nums` 相同，并全部填充为 1。

### 2. 计算前缀积：

- 初始化 `left_product = 1`。
- 从左到右遍历 `nums`：
  - `answer[i] = left_product` (此时 `answer[i]` 存储的是 `nums[i]` 左边的乘积)。
  - `left_product *= nums[i]`。

### 3. 计算后缀积并合并：

- 初始化 `right_product = 1`。
- 从右到左遍历 `nums`：
  - `answer[i] *= right_product` (将 `nums[i]` 左边的乘积与右边的乘积相乘)。
  - `right_product *= nums[i]`。

4. 返回 `answer`。

时间复杂度： $O(n)$  - 两次遍历。

空间复杂度： $O(1)$  - 如果不考虑 `answer` 数组的额外空间（因为它是结果数组）。如果考虑，则是  $O(n)$ 。

## 代码实现

### Python

```
def productExceptSelf(nums: list[int]) -> list[int]:
    """
    使用前缀积和后缀积的方法，在  $O(n)$  时间和  $O(1)$  额外空间（不计输出数组）内解决。
    """

    n = len(nums)
    answer = [1] * n # 初始化结果数组，用于存储最终结果

    # 1. 计算前缀积
    # answer[i] 此时存储的是 nums[i] 左边所有元素的乘积
    left_product = 1
    for i in range(n):
        answer[i] = left_product
        left_product *= nums[i]

    # 2. 计算后缀积并与前缀积相乘
    # right_product 存储的是 nums[i] 右边所有元素的乘积
    right_product = 1
    for i in range(n - 1, -1, -1):
        answer[i] *= right_product
        right_product *= nums[i]
```

```
    right_product *= nums[i]

    return answer
```

## 关键点总结

1. 分解问题：将“除自身以外的乘积”分解为“左边元素的乘积”和“右边元素的乘积”的乘积。
2. 两次遍历：一次从左到右计算前缀积，一次从右到左计算后缀积并合并结果。
3.  $O(1)$  空间：巧妙地利用输出数组 `answer` 来存储中间结果，从而避免了额外的  $O(n)$  空间。
4. 避免除法：严格遵守了题目不能使用除法的要求。

## 普通数组 / 41. 缺失的第一个正数

### 41. 缺失的第一个正数

#### 题目描述

给你一个未排序的整数数组 `nums`，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为  $O(n)$  并且只使用常数级别额外空间的解决方案。

#### 示例

示例 1：

输入：`nums = [1, 2, 0]`  
输出：3

示例 2：

输入：`nums = [3, 4, -1, 1]`  
输出：2

示例 3：

输入：`nums = [7, 8, 9, 11, 12]`  
输出：1

#### 约束条件

- $1 \leq \text{nums.length} \leq 5 * 10^5$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

#### 思考过程

##### ➊ 第一步：理解问题

- 我们要找的是数组中没有出现的最小的正整数。
- 数组是未排序的。
- 关键限制：时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ 。

##### ➋ 第二步：初步分析与排除法

- 正整数：这意味着我们只关心大于 0 的数。负数和 0 都可以忽略。
- 最小的正整数：如果数组中有 1，那么答案可能是 2 或更大；如果没有 1，那么答案就是 1。
- 数组长度 `n`：如果数组长度为 `n`，那么我们关心的正整数范围是 `[1, n+1]`。为什么是 `n+1`？因为如果 `1` 到 `n` 都出现了，那么最小的缺失正整数就是 `n+1`。

##### ➌ 第三步：暴力解法（排序或哈希表）

- 排序：先对数组排序，然后从 1 开始遍历，看哪个正整数没有出现。时间复杂度  $O(n \log n)$ ，不满足  $O(n)$ 。
- 哈希表/集合：将所有正整数存入哈希集合，然后从 1 开始遍历，看哪个正整数不在集合中。时间复杂度  $O(n)$ ，空间复杂度  $O(n)$ ，不满足  $O(1)$ 。

##### ➍ 第四步：原地哈希（ $O(1)$ 空间的关键）

既然不能用额外空间，我们只能利用原数组的空间。如何利用？

核心思想：

- 我们可以尝试将数字 `x` 放到它“应该在”的位置 `x-1` 上。
- 例如，如果 `nums[i] = 1`，我们就把它放到 `nums[0]` 的位置。
- 如果 `nums[i] = 2`，我们就把它放到 `nums[1]` 的位置。

### 具体操作：

遍历数组 `nums`。对于每个 `nums[i]`：

- 如果 `nums[i]` 是一个正数，并且在 `[1, n]` 的范围内。
- 并且 `nums[i]` 不在它“应该在”的位置上（即 `nums[i] != nums[nums[i]-1]`）。
- 那么，我们就交换 `nums[i]` 和 `nums[nums[i]-1]`。
- 这个过程需要持续进行，直到 `nums[i]` 归位，或者 `nums[i]` 不符合条件（负数、0、超出范围）。

### 思考题：

- 为什么只关心 `[1, n]` 范围内的正整数？
- 交换操作如何保证时间复杂度是  $O(n)$ ？

► 点击查看分析

## 💡 第五步：算法步骤总结

- 获取数组长度 `n = len(nums)`。
- 原地调整：**遍历数组 `nums`，索引 `i` 从 `0` 到 `n-1`：
  - 使用一个 `while` 循环：
    - 条件 1： `nums[i] > 0` (必须是正数)
    - 条件 2： `nums[i] <= n` (必须在 `[1, n]` 范围内)
    - 条件 3： `nums[i] != nums[nums[i]-1]` (当前数字不在它应该的位置上)
  - 如果以上三个条件都满足，则交换 `nums[i]` 和 `nums[nums[i]-1]`。
  - 如果不满足，或者交换后 `nums[i]` 已经归位，则跳出 `while` 循环，处理下一个 `i`。
- 查找缺失：**再次遍历调整后的数组 `nums`，索引 `i` 从 `0` 到 `n-1`：
  - 如果 `nums[i] != i + 1`，说明 `i + 1` 这个正整数缺失了，返回 `i + 1`。
- 如果遍历完整个数组，所有 `nums[i]` 都等于 `i + 1`，说明 `1` 到 `n` 都存在，那么缺失的最小正整数就是 `n + 1`，返回 `n + 1`。

时间复杂度：  $O(n)$  - 两次遍历，加上内部的交换操作均摊  $O(n)$ 。

空间复杂度：  $O(1)$  - 原地修改数组。

## 代码实现

### Python

```
def firstMissingPositive(nums: list[int]) -> int:  
    """  
    使用原地哈希（交换法）寻找缺失的第一个正数。  
    """  
  
    n = len(nums)  
  
    # 1. 将数字放到其“应该在”的位置上  
    # 例如，1 放到索引 0, 2 放到索引 1, 以此类推  
    for i in range(n):  
        # 循环条件：  
        # 1. nums[i] 必须是正数  
        # 2. nums[i] 必须在 [1, n] 的范围内  
        # 3. nums[i] 不在它应该在的位置上 (即 nums[i] != nums[nums[i]-1])  
        while 1 <= nums[i] <= n and nums[i] != nums[nums[i]-1]:  
            # 交换 nums[i] 和 nums[nums[i]-1]  
            # temp_idx 是 nums[i] 应该去的位置  
            temp_idx = nums[i] - 1  
            nums[i], nums[temp_idx] = nums[temp_idx], nums[i]  
  
    # 2. 查找第一个不符合“nums[i] == i + 1”的位置  
    for i in range(n):  
        if nums[i] != i + 1:  
            return i + 1  
  
    # 3. 如果 1 到 n 都存在，那么缺失的第一个正数就是 n + 1  
    return n + 1
```

## 关键点总结

1. **原地哈希**: 利用数组本身的索引作为哈希表的键, 将数字放到其对应的位置上, 从而实现  $O(1)$  的空间复杂度。
2. **有效范围**: 只关注  $[1, n]$  范围内的正整数, 因为其他数字对结果没有影响。
3. **循环交换**: 内部的 `while` 循环确保每个数字都被正确地放置或被忽略。虽然是循环, 但每个数字最多被交换两次, 所以总时间复杂度是  $O(n)$ 。
4. **最终查找**: 在调整后的数组中, 第一个  $\text{nums}[i] \neq i + 1$  的位置  $i$ , 就意味着  $i + 1$  是缺失的第一个正数。

## 普通数组 / 53. 最大子数组和

### 53. 最大子数组和

#### 题目描述

给你一个整数数组 `nums`, 请你找出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。

子数组 是数组中的一个连续部分。

#### 示例

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。

示例 2:

输入: `nums = [1]`

输出: 1

示例 3:

输入: `nums = [5, 4, -1, 7, 8]`

输出: 23

#### 约束条件

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

#### 思考过程

##### 💡 第一步：理解问题

- 我们要找的是一个连续的子数组。
- 这个子数组的和必须是所有连续子数组中最大的。
- 子数组至少包含一个元素。
- 数组中可能包含负数。

##### 💡 第二步：暴力解法

最直接的想法是, 找出所有可能的连续子数组, 计算它们的和, 然后找到最大的那个。

思考题:

- 如何用循环实现找到所有连续子数组?
- 这个方法的时间复杂度是多少?

► 点击查看分析

##### 💡 第三步：动态规划

$O(n^2)$  太慢, 我们需要更高效的方法。这个问题具有“最优子结构”和“重叠子问题”的特点, 这提示我们可以使用动态规划。

定义 `dp[i]`:

`dp[i]` 表示以 `nums[i]` 结尾的连续子数组的最大和。

思考题:

- `dp[i]` 和 `dp[i-1]` 之间有什么关系?
- 如何推导出状态转移方程?

► 点击查看分析

##### 💡 第四步：动态规划算法步骤

1. 创建一个 `dp` 数组, 大小与 `nums` 相同。

2. 初始化 `dp[0] = nums[0]`。
3. 初始化 `max_sum = nums[0]`，用于记录全局最大和。
4. 从 `i = 1` 到 `n-1` 遍历 `nums`：
  - a. `dp[i] = max(nums[i], dp[i-1] + nums[i])`。
  - b. `max_sum = max(max_sum, dp[i])`。
5. 返回 `max_sum`。

**时间复杂度：**  $O(n)$  - 只遍历一次数组。

**空间复杂度：**  $O(n)$  - 使用了一个 `dp` 数组。

### 💡 第五步：空间优化（Kadane's Algorithm）

观察状态转移方程 `dp[i] = max(nums[i], dp[i-1] + nums[i])`，我们发现 `dp[i]` 的计算只依赖于 `dp[i-1]`。这意味着我们不需要整个 `dp` 数组，只需要一个变量来存储前一个 `dp` 值即可。

**算法流程：**

1. 初始化 `current_max = nums[0]`（表示以当前元素结尾的最大和）。
2. 初始化 `global_max = nums[0]`（表示全局最大和）。
3. 从 `i = 1` 到 `n-1` 遍历 `nums`：
  - a. `current_max = max(nums[i], current_max + nums[i])`。
  - b. `global_max = max(global_max, current_max)`。
4. 返回 `global_max`。

**时间复杂度：**  $O(n)$  - 依然只遍历一次数组。

**空间复杂度：**  $O(1)$  - 只使用了常数个额外变量。

这就是著名的 **Kadane's Algorithm**。

## 代码实现

### Python

```
def maxSubArray(nums: list[int]) -> int:
    """
    使用 Kadane's Algorithm 寻找最大子数组和。
    """

    if not nums:
        return 0

    current_max = nums[0] # 以当前元素结尾的最大和
    global_max = nums[0] # 全局最大和

    for i in range(1, len(nums)):
        # 决定以 nums[i] 结尾的最大和:
        # 1. 只包含 nums[i] 本身
        # 2. 包含 nums[i] 和前面连续子数组的最大和
        current_max = max(nums[i], current_max + nums[i])

        # 更新全局最大和
        global_max = max(global_max, current_max)

    return global_max
```

## 关键点总结

1. **动态规划思想：** 将问题分解为子问题，并利用子问题的解来构建原问题的解。
2. **Kadane's Algorithm：** 一个非常经典的动态规划优化，将空间复杂度从  $O(n)$  降到  $O(1)$ 。
3. **current\_max 的含义：** 它始终代表着“以当前遍历到的元素为结尾的连续子数组的最大和”。如果 `current_max` 加上当前元素后变得更小，甚至小于当前元素本身，那就说明之前的子数组对当前元素是“拖累”，不如从当前元素重新开始一个子数组。

## 普通数组 / 56. 合并区间

### 56. 合并区间

## 题目描述

以数组 `intervals` 表示若干个区间的集合，其中每个区间表示为 `intervals[i] = [starti, endi]`。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组中的区间通常按 `starti` 进行升序排列。

## 示例

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]` 可以被视为重叠区间。

## 约束条件

- $1 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].length == 2$
- $0 \leq \text{starti} \leq \text{endi} \leq 10^4$

## 思考过程

### 💡 第一步：理解问题

- 我们有一系列区间，每个区间由起始点和结束点组成。
- 如果两个区间有重叠，我们需要将它们合并成一个更大的区间。
- 最终返回一个不重叠的区间数组。
- 结果数组中的区间通常按起始点升序排列。

### 💡 第二步：如何判断重叠？

两个区间 `[a, b]` 和 `[c, d]` 如何判断它们是否重叠？

思考题：

- 什么样的条件表示两个区间重叠？
- 什么样的条件表示两个区间不重叠？

▶ 点击查看分析

### 💡 第三步：朴素解法（尝试合并）

我们可以遍历所有区间对，如果发现重叠就合并，然后重复这个过程直到没有区间可以合并。

思考题：这个方法有什么问题？时间复杂度如何？

▶ 点击查看分析

### 💡 第四步：排序 + 遍历

朴素解法的问题在于，它没有利用区间之间的顺序关系。如果区间是无序的，我们确实需要两两比较。

但是，如果我们将区间按照它们的起始点进行排序，会发生什么？

思考题：排序后，相邻区间之间有什么特性？这能帮助我们简化问题吗？

▶ 点击查看分析

### 💡 第五步：算法步骤总结

1. **排序**: 首先，将所有区间按照它们的起始点 `starti` 进行升序排序。如果起始点相同，则按结束点 `endi` 升序排序（虽然本题不强制，但通常是好习惯）。
2. **遍历合并**:
  - a. 初始化一个结果列表 `merged_intervals`，并将排序后的第一个区间加入。
  - b. 从第二个区间开始遍历 `intervals`。
    - i. 获取 `merged_intervals` 中最后一个区间 `last_merged = [last_start, last_end]`。
    - ii. 判断是否重叠: 如果 `current_start <= last_end`，说明当前区间与 `last_merged` 重叠。
      - 合并它们: 更新 `last_merged` 的结束点为 `max(last_end, current_end)`。
      - iii. 不重叠: 如果 `current_start > last_end`，说明当前区间与 `last_merged` 不重叠。
        - 将 `current_interval` 直接加入 `merged_intervals`。
  3. 返回 `merged_intervals`。

时间复杂度:  $O(N \log N)$  - 主要消耗在排序上。遍历是  $O(N)$ 。

空间复杂度:  $O(N)$  - 存储结果列表。

## 代码实现

### Python

```
def merge(intervals: list[list[int]]) -> list[list[int]]:
    """
    先排序，然后遍历合并重叠区间。
    """

    if not intervals:
        return []

    # 1. 按照区间的起始点进行排序
    intervals.sort(key=lambda x: x[0])

    merged_intervals = []

    # 2. 遍历排序后的区间
    for interval in intervals:
        # 如果 merged_intervals 是空的，或者当前区间与 merged_intervals 中最后一个区间不重叠
        # (即当前区间的起始点大于已合并区间的结束点)
        if not merged_intervals or interval[0] > merged_intervals[-1][1]:
            merged_intervals.append(interval)
        else:
            # 如果重叠，则更新 merged_intervals 中最后一个区间的结束点
            # 取当前区间的结束点和已合并区间的结束点中的较大值
            merged_intervals[-1][1] = max(merged_intervals[-1][1], interval[1])

    return merged_intervals
```

## 关键点总结

- 排序：是解决区间合并问题的核心步骤。通过排序，我们将无序的区间问题转化为有序的线性扫描问题。
- 贪心策略：每次只考虑当前区间与前一个已合并区间是否重叠。如果重叠，就合并；不重叠，就开启一个新的合并区间。
- 边界条件：注意处理空输入和只有一个区间的情况。

## 栈 / 155. 最小栈

### 155. 最小栈

#### 题目描述

设计一个支持 `push`, `pop`, `top` 操作，并能在常数时间内检索到最小元素的栈。

实现 `MinStack` 类:

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

示例 1:

输入:

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

输出:

```
[null,null,null,null,-3,null,0,-2]
```

解释:

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

提示:

- $-2^{31} \leq val \leq 2^{31} - 1$
- pop、top 和 getMin 操作总是在 非空栈 上调用
- push、pop、top 和 getMin 最多被调用  $3 * 10^4$  次

思考过程:

这道题要求设计一个支持 push、pop、top 和 getMin 操作的栈，并且 getMin 操作的时间复杂度为 O(1)。

核心思路:

如果只用一个栈来存储元素，getMin 操作需要遍历整个栈，时间复杂度为 O(n)。为了实现 O(1) 的 getMin，我们需要额外的空间来记录最小值。

方法一：使用两个栈

1. **数据栈 (data\_stack):** 用于存储所有入栈的元素，与普通栈操作相同。
2. **最小栈 (min\_stack):** 用于存储当前栈中的最小值。

- push(x)：
  - 将 x 压入 data\_stack。
  - 如果 min\_stack 为空，或者 x 小于等于 min\_stack 的栈顶元素，则将 x 压入 min\_stack。
- pop()：
  - 从 data\_stack 弹出元素 val。
  - 如果 val 等于 min\_stack 的栈顶元素，则从 min\_stack 弹出元素。
- top()：返回 data\_stack 的栈顶元素。
- getMin()：返回 min\_stack 的栈顶元素。

方法二：一个栈，每个元素存储 (value, min\_value)

1. **一个栈 (stack):** 栈中的每个元素都是一个元组 `(value, min_value)`，其中 value 是当前入栈的元素，min\_value 是当前栈中（包括 value 在内）的最小值。

- push(x)：
  - 如果栈为空，`min_val = x`。
  - 如果栈不为空，`min_val = min(x, stack[-1][1])`。
  - 将 `(x, min_val)` 压入栈。
- pop()：从栈中弹出元素。
- top()：返回栈顶元素的 value 部分。
- getMin()：返回栈顶元素的 min\_value 部分。

Python 代码 (方法一，更常用):

```
class MinStack:

    def __init__(self):
        self.data_stack = []
        self.min_stack = []

    def push(self, val: int) -> None:
        self.data_stack.append(val)
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self) -> None:
        if self.data_stack:
```

```

    val = self.data_stack.pop()
    if val == self.min_stack[-1]:
        self.min_stack.pop()

def top(self) -> int:
    if self.data_stack:
        return self.data_stack[-1]
    # 根据题目要求, 如果栈为空, 可能需要抛出异常或返回特定值
    # 这里假设题目保证调用 top() 时栈不为空

def getMin(self) -> int:
    if self.min_stack:
        return self.min_stack[-1]
    # 根据题目要求, 如果栈为空, 可能需要抛出异常或返回特定值
    # 这里假设题目保证调用 getMin() 时栈不为空

# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# param_3 = obj.top()
# param_4 = obj.getMin()

```

## 栈 / 20. 有效的括号

### 20. 有效的括号

#### 题目描述

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。
3. 每个右括号都有一个对应的相同类型的左括号。

#### 示例 1：

```

输入: s = "()"
输出: true

```

#### 示例 2：

```

输入: s = "()[]{}"
输出: true

```

#### 示例 3：

```

输入: s = "(]"
输出: false

```

#### 提示：

- $1 \leq s.length \leq 10^4$
- s 仅由括号 '()'[]{}' 组成

#### 思考过程：

这道题要求判断一个字符串是否是有效的括号字符串。有效的括号字符串需要满足以下条件：

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。
3. 每个右括号都有一个对应的相同类型的左括号。

这是一个典型的栈应用问题。

#### 核心思路:

我们可以使用一个栈来解决这个问题。

1. 遍历字符串: 逐个字符遍历输入的字符串 `s`。
2. 遇到左括号: 如果遇到左括号 `(, {, [`, 将其压入栈中。
3. 遇到右括号: 如果遇到右括号 `), }, ]`:
  - 检查栈是否为空: 如果栈为空, 说明没有对应的左括号, 直接返回 `False`。
  - 检查栈顶元素: 弹出栈顶元素, 并检查它是否与当前右括号类型匹配。
    - 如果匹配, 继续遍历。
    - 如果不匹配, 说明括号不匹配, 返回 `False`。
4. 遍历结束: 字符串遍历结束后, 检查栈是否为空。
  - 如果栈为空, 说明所有括号都已正确匹配, 返回 `True`。
  - 如果栈不为空, 说明还有未匹配的左括号, 返回 `False`。

#### Python 代码:

```
class Solution:  
    def isValid(self, s: str) -> bool:  
        stack = []  
        mapping = {")": "(", "}": "{", "]": "["}  
  
        for char in s:  
            if char in mapping.values(): # 左括号  
                stack.append(char)  
            elif char in mapping.keys(): # 右括号  
                if not stack or mapping[char] != stack.pop():  
                    return False  
            else:  
                # 忽略其他字符, 或者根据题目要求处理  
                pass  
  
        return not stack
```

---

## 栈 / 394. 字符串解码

### 394. 字符串解码

#### 题目描述

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: `k[encoded_string]`, 表示其中方括号内部的 `encoded_string` 正好重复 `k` 次。注意 `k` 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 `k`，例如不会出现像 `3a` 或 `2[4]` 的输入。

#### 示例 1:

```
输入: s = "3[a]2[bc]"  
输出: "aaabcbc"
```

#### 示例 2:

```
输入: s = "2[abc]3[cd]ef"  
输出: "abcabcccdcdcdef"
```

### 示例 3：

```
输入: s = "abc3[cd]xyz"
输出: "abcccdcdxyz"
```

#### 提示：

- $1 \leq s.length \leq 30$
- $s$  由小写英文字母、数字和方括号 '[]' 组成
- $s$  保证是一个有效的输入。
- $s$  中所有整数的取值范围为 [1, 300]

#### 思考过程：

这道题要求解码一个字符串，其中包含用数字和方括号表示的重复子字符串。例如，“3[a]2[bc]”解码为“aaabcbc”。

#### 核心思路：

这道题可以使用栈来解决。当遇到数字时，我们将其压入一个栈（用于存储重复次数）；当遇到左括号时，我们将其压入另一个栈（用于存储当前字符串）。当遇到右括号时，我们弹出栈顶的重复次数和字符串，进行解码，并将解码后的字符串重新压入字符串栈。

#### 具体步骤：

我们可以使用两个栈：

1. `num_stack`: 存储数字（重复次数）。
2. `str_stack`: 存储字符串（在遇到 '[' 之前已经构建好的字符串）。

同时，我们需要一个 `current_num` 来构建当前数字，一个 `current_str` 来构建当前字符串。

#### 遍历字符串 s：

- 如果 `char` 是数字：将 `char` 转换为数字并更新 `current_num = current_num * 10 + int(char)`。
- 如果 `char` 是 '['：
  - 将 `current_str` 压入 `str_stack`。
  - 将 `current_num` 压入 `num_stack`。
  - 重置 `current_str = ""`。
  - 重置 `current_num = 0`。
- 如果 `char` 是 ']'：
  - 从 `num_stack` 弹出 `repeat_times`。
  - 从 `str_stack` 弹出 `prev_str`。
  - `current_str = prev_str + current_str * repeat_times`。
- 如果 `char` 是字母：将 `char` 添加到 `current_str`。

最后，返回 `current_str`。

#### Python 代码：

```
class Solution:
    def decodeString(self, s: str) -> str:
        num_stack = []
        str_stack = []
        current_num = 0
        current_str = ""

        for char in s:
            if '0' <= char <= '9':
                current_num = current_num * 10 + int(char)
            elif char == '[':
                num_stack.append(current_num)
                str_stack.append(current_str)
                current_num = 0
                current_str = ""
            elif char == ']':
                repeat_times = num_stack.pop()
                prev_str = str_stack.pop()
                current_str = prev_str + current_str * repeat_times
        return current_str
```

```

        current_str = prev_str + current_str * repeat_times
    else: # 字母
        current_str += char

    return current_str

```

## 栈 / 739. 每日温度

### 739. 每日温度

#### 题目描述

给定一个整数数组 `temperatures`，表示每天的温度，返回一个数组 `answer`，其中 `answer[i]` 是指对于第 `i` 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 0 来代替。

#### 示例 1:

```

输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]

```

#### 示例 2:

```

输入: temperatures = [30,40,50,60]
输出: [1,1,1,0]

```

#### 示例 3:

```

输入: temperatures = [30,60,90]
输出: [1,1,0]

```

#### 提示:

- $1 \leq \text{temperatures.length} \leq 10^5$
- $30 \leq \text{temperatures}[i] \leq 100$

#### 思考过程:

这道题要求对于给定的每日温度列表 `temperatures`，返回一个列表 `answer`，其中 `answer[i]` 表示第 `i` 天之后，至少需要等待多少天才能等到一个更高的温度。如果等不到，则为 0。

#### 核心思路:

这是一个典型的单调栈问题。我们需要找到每个元素右侧第一个比它大的元素。

#### 单调栈的应用:

我们可以使用一个单调递减栈来存储温度的索引。当遍历温度列表时：

1. 如果栈为空，或者当前温度小于等于栈顶索引对应的温度，将当前温度的索引压入栈中。
2. 如果当前温度大于栈顶索引对应的温度，说明找到了栈顶索引对应温度的第一个更高温度。
  - 弹出栈顶索引 `prev_index`。
  - 计算等待天数： `answer[prev_index] = current_index - prev_index`。
  - 重复此过程，直到栈为空，或者当前温度小于等于新的栈顶索引对应的温度。
  - 最后，将当前温度的索引压入栈中。

#### 初始化:

- `answer` 列表，长度与 `temperatures` 相同，初始值都为 0。
- `stack` (空列表，用于存储索引)。

#### 遍历 `temperatures` 列表:

- `for i, temp in enumerate(temperatures):`

#### Python 代码:

```

class Solution:
    def dailyTemperatures(self, temperatures: list[int]) -> list[int]:
        n = len(temperatures)
        answer = [0] * n
        stack = [] # 存储索引，单调递减栈

        for i, temp in enumerate(temperatures):
            # 当栈不为空，且当前温度大于栈顶索引对应的温度时
            while stack and temp > temperatures[stack[-1]]:
                prev_index = stack.pop()
                answer[prev_index] = i - prev_index
            stack.append(i)

        return answer

```

## 栈 / 84. 柱状图中最大的矩形

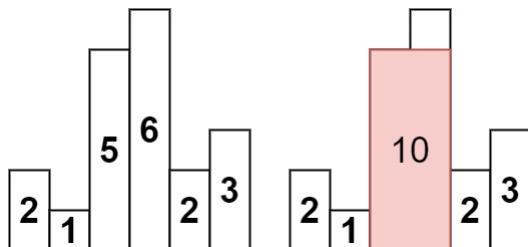
### 84. 柱状图中最大的矩形

#### 题目描述

给定  $n$  个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。

求在该柱状图中，能够勾勒出来的矩形的最大面积。

#### 示例 1:

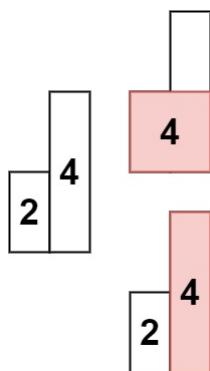


输入: heights = [2,1,5,6,2,3]

输出: 10

解释: 最大的矩形为图中红色区域，面积为 10

#### 示例 2:



输入: heights = [2,4]

输出: 4

#### 提示:

- $1 \leq \text{heights.length} \leq 10^5$
- $0 \leq \text{heights}[i] \leq 10^4$

#### 思考过程:

这道题要求在给定的柱状图中找到能够形成的最大的矩形面积。每个柱子的高度由 `heights` 数组给出，宽度都为 1。

#### 核心思路：

对于每个柱子，如果它作为矩形的高度，那么这个矩形的宽度可以向左和向右延伸，直到遇到比它矮的柱子。因此，我们需要找到每个柱子左侧和右侧第一个比它矮的柱子。这可以使用单调栈来高效地解决。

#### 单调栈的应用：

我们可以使用两个单调栈来分别找到每个柱子左侧和右侧第一个比它矮的柱子。

##### 1 找到每个柱子左侧第一个比它矮的柱子 (`left_less`)：

- 使用一个单调递增栈。
- 遍历 `heights` 数组。对于每个柱子 `heights[i]`：
  - 当栈不为空且 `heights[stack.top()] >= heights[i]` 时，弹出栈顶元素。
  - 如果栈为空，说明左侧没有比它矮的，`left_less[i] = -1`。
  - 否则，`left_less[i] = stack.top()`。
  - 将 `i` 压入栈。

##### 2 找到每个柱子右侧第一个比它矮的柱子 (`right_less`)：

- 使用一个单调递增栈。
- 从右向左遍历 `heights` 数组。对于每个柱子 `heights[i]`：
  - 当栈不为空且 `heights[stack.top()] >= heights[i]` 时，弹出栈顶元素。
  - 如果栈为空，说明右侧没有比它矮的，`right_less[i] = n` (`n` 是数组长度)。
  - 否则，`right_less[i] = stack.top()`。
  - 将 `i` 压入栈。

##### 3. 计算最大面积：

- 遍历 `heights` 数组。对于每个柱子 `heights[i]`：
  - 宽度 `width = right_less[i] - left_less[i] - 1`。
  - 面积 `area = heights[i] * width`。
  - 更新最大面积。

#### 优化：一次遍历

实际上，我们可以通过一次遍历来完成。当一个元素从单调栈中弹出时，它就是栈顶元素右侧第一个比它小的元素。而它左侧第一个比它小的元素就是它弹出后新的栈顶元素。

#### 一次遍历的单调栈：

- 在 `heights` 数组的两端添加一个高度为 0 的柱子，这样可以确保所有柱子都能被处理到。
- 使用一个单调递增栈，存储柱子的索引。
- 遍历修改后的 `heights` 数组。对于每个柱子 `heights[i]`：
  - 当栈不为空且 `heights[i] < heights[stack.top()]` 时：
    - 弹出栈顶索引 `h_index = stack.pop()`。
    - 当前柱子 `heights[h_index]` 的高度为 `h = heights[h_index]`。
    - 它的右边界是 `i`。
    - 它的左边界是新的栈顶元素 `stack.top()`。
    - 宽度 `width = i - stack.top() - 1`。
    - 面积 `area = h * width`。
    - 更新最大面积。
  - 将 `i` 压入栈。

#### Python 代码 (一次遍历的单调栈)：

```
class Solution:  
    def largestRectangleArea(self, heights: list[int]) -> int:  
        n = len(heights)  
        # 在 heights 数组两端添加高度为 0 的柱子，方便处理边界情况  
        new_heights = [0] + heights + [0]  
  
        stack = [] # 存储索引，单调递增栈  
        max_area = 0  
  
        for i in range(len(new_heights)):  
            # 当当前柱子高度小于栈顶柱子高度时，说明栈顶柱子可以出栈计算面积了  
            while stack and new_heights[i] < new_heights[stack[-1]]:  
                h_index = stack.pop()  
                h = new_heights[h_index]  
                left_index = stack[-1]  
                width = i - left_index - 1  
                area = h * width  
                max_area = max(max_area, area)
```

```

    h_index = stack.pop() # 弹出栈顶柱子的索引
    h = new_heights[h_index] # 栈顶柱子的高度

    # 计算宽度: 当前柱子索引 - 栈顶元素(左边界) - 1
    w = i - stack[-1] - 1
    max_area = max(max_area, h * w)

stack.append(i)

return max_area

```

## 滑动窗口 / 3. 无重复字符的最长子串

### 3. 无重复字符的最长子串

#### 题目描述

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

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。

示例 2：

输入：`s = "bbbbbb"`

输出：1

解释：因为无重复字符的最长子串是 `"b"`，所以其长度为 1。

示例 3：

输入：`s = "pwwkew"`

输出：3

解释：因为无重复字符的最长子串是 `"wke"`，所以其长度为 3。

请注意，你的答案必须是 子串 的长度，`"pwke"` 是一个子序列，不是子串。

#### 约束条件

- $0 \leq s.length \leq 5 * 10^4$
- `s` 由英文字母、数字、符号和空格组成。

#### 思考过程

##### 思考第一步：理解问题

- 我们要找的是 **子串**，意味着字符必须是连续的。
- 这个子串必须 **没有重复字符**。
- 我们需要返回的是满足条件的最长子串的 **长度**。

##### 思考第二步：暴力解法

最直接的想法是，找出所有的子串，然后逐一检查它们是否含有重复字符，并记录最长的长度。

#### 思考题：

- 如何用循环实现找到所有子串？
- 如何检查一个子串是否有重复字符？
- 这个方法的时间复杂度是多少？

► 点击查看分析

##### 思考第三步：寻找优化思路 -> 滑动窗口

暴力解法做了太多重复的检查。例如，当我们检查完 `"abc"` 后，再检查 `"abca"` 时，我们其实不需要从头开始。我们知道 `"abc"` 是无重复的，问题出在最后一个 `'a'` 上。

这启发我们使用一种更聪明的移动方式来扫描字符串，这就是 **滑动窗口**。

想象一个窗口在字符串上滑动。这个窗口代表我们当前正在考察的子串。

- 我们需要一个 **右指针** `right`，负责扩大窗口，将新字符纳入考察范围。
- 我们需要一个 **左指针** `left`，在窗口内出现重复字符时，负责收缩窗口，将重复的源头排除出去。
- 我们还需要一个数据结构（比如哈希集合或哈希表）来快速判断窗口内是否存在某个字符。

## 第四步：滑动窗口算法

- 初始化 `left = 0, right = 0, max_len = 0`。
- 用一个哈希集合 `window_chars` 来存储当前窗口内的所有字符。
- 当 `right` 指针没有越界时，循环执行：
  - 将 `s[right]` 字符移入窗口。
  - 检查 `s[right]` 是否已经在 `window_chars` 中？
    - 如果不在：说明当前窗口仍然没有重复字符。我们将 `s[right]` 加入 `window_chars`，然后更新 `max_len = max(max_len, right - left + 1)`。之后，移动 `right` 指针，扩大窗口 (`right++`)。
    - 如果在：说明 `s[right]` 与窗口中的某个字符重复了。我们需要收缩窗口。我们将 `s[left]` 从 `window_chars` 中移除，然后移动 `left` 指针 (`left++`)。我们持续这个收缩过程，直到 `s[right]` 这个重复的源头被移除出窗口左侧为止。

优化一下逻辑：

上面的收缩逻辑可以更清晰地表述为：

当 `s[right]` 已经在 `window_chars` 中时，我们就进入一个 `while` 循环，不断地从窗口左侧移除字符 `s[left]` 并 `left++`，直到 `s[right]` 不再是重复项为止。然后才将新的 `s[right]` 加入窗口。

再优化一下：

我们可以用哈希表（字典）来代替哈希集合，存储每个字符及其最新的索引。`window[char] = index`。

- 当我们遇到一个新字符 `s[right]` 时，如果它已经在 `window` 中，并且它的旧索引 `window[s[right]]` 大于等于我们当前的左边界 `left`，这意味着在当前窗口内发生了重复。
- 此时，我们不需要一步步地移动 `left`，而是可以直接将 `left` 跳到重复字符的旧位置的下一个位置：`left = window[s[right]] + 1`。
- 无论是否重复，我们都更新 `s[right]` 的最新索引到哈希表中，并计算当前窗口长度 `right - left + 1`，更新最大长度。

## 第五步：算法步骤总结（优化后）

- 初始化一个哈希表 `window` 用于存储字符的最新索引，`left = 0, max_len = 0`。
- 用 `right` 指针遍历字符串 `s`（从 0 到 `n-1`）：
  - 获取当前字符 `char = s[right]`。
  - 检查重复：如果 `char` 在 `window` 中，并且 `window[char] >= left`，说明在当前窗口 `[left, right]` 内发现重复。更新左边界 `left = window[char] + 1`。
  - 更新索引：将 `char` 的最新索引存入 `window`：`window[char] = right`。
  - 更新结果：计算当前无重复子串的长度 `right - left + 1`，并更新 `max_len`。
- 遍历结束后，返回 `max_len`。

时间复杂度：O(n) - 因为 `left` 和 `right` 指针都只从左到右遍历一次字符串。

空间复杂度：O(k) - 其中 k 是字符串中不同字符的数量。最坏情况下，所有字符都不同，空间复杂度为 O(n)。

## 代码实现

### Python

```
def lengthOfLongestSubstring(s: str) -> int:  
    """  
    使用滑动窗口和哈希表寻找最长无重复子串的长度。  
    """  
  
    if not s:  
        return 0  
  
    # window 用来存储字符及其最新出现的索引  
    window = {}  
    max_len = 0  
    left = 0 # 窗口的左边界  
  
    # right 是窗口的右边界  
    for right, char in enumerate(s):  
        # 检查字符是否在当前窗口内重复  
        # 如果 char 在 window 中，并且它的索引在 left 右侧（或重合），说明重复  
        if char in window and window[char] >= left:  
            # 发现重复，将左边界直接移动到重复字符的下一个位置  
            left = window[char] + 1  
            window[char] = right  
        else:  
            window[char] = right  
        max_len = max(max_len, right - left + 1)
```

```

    left = window[char] + 1

    # 无论如何，都更新字符的最新位置
    window[char] = right

    # 计算当前窗口的长度，并更新最大长度
    current_len = right - left + 1
    max_len = max(max_len, current_len)

return max_len

```

## 关键点总结

1. **滑动窗口**: 是解决一系列子串问题的通用高效思想，避免了暴力解法的冗余计算。
2. **哈希表/集合**: 是滑动窗口的得力助手，用于快速判断窗口内的元素情况（是否存在、出现次数、最新位置等）。
3. **指针移动策略**:
  - 右指针 `right` 总是向右移动，用于扩展窗口。
  - 左指针 `left` 只在需要收缩窗口时移动，并且优化后的算法可以让它“跳跃”而不是“爬行”，从而提高效率。

## 滑动窗口 / 438. 找到字符串中所有字母异位词

### 438. 找到字符串中所有字母异位词

#### 题目描述

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词** 的子串，返回这些子串的起始索引。

**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

示例 1：

输入: `s = "cbaebabacd"`, `p = "abc"`

输出: `[0, 6]`

解释:

起始索引为 0 的子串是 `"cba"`，它是 `"abc"` 的异位词。

起始索引为 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:

起始索引为 0 的子串是 `"ab"`，它是 `"ab"` 的异位词。

起始索引为 1 的子串是 `"ba"`，它是 `"ab"` 的异位词。

起始索引为 2 的子串是 `"ab"`，它是 `"ab"` 的异位词。

#### 约束条件

- $1 \leq s.length, p.length \leq 3 * 10^4$
- `s` 和 `p` 仅包含小写英文字母

#### 思考过程

##### 💡 第一步：理解问题

- 我们要在长字符串 `s` 中，找到所有与短字符串 `p` 构成字母异位词的子串。
- “字母异位词”意味着两个字符串包含的字符种类和数量完全相同。
- 我们需要返回的是这些子串的**起始索引**。
- 所有子串的长度都必须和 `p` 的长度相等。

##### 💡 第二步：朴素解法

最直接的想法是，遍历 `s` 中所有与 `p` 等长的子串，然后判断每个子串是否是 `p` 的异位词。

思考题：

- 如何判断两个字符串是否是异位词？
- 这个朴素解法的时间复杂度是多少？

► 点击查看分析

### ⌚ 第三步：滑动窗口

朴素解法的瓶颈在于，每次都对一个全新的子串进行完整的字符统计，而相邻的两个子串其实大部分是重合的。例如，当我们检查完 `s[0:m]` 后，再去检查 `s[1:m+1]`，我们实际上只是移除了 `s[0]` 并加入了 `s[m+1]`。

这正是滑动窗口大显身手的场景。

**核心思路：**

1. 维护一个固定大小 (`len(p)`) 的窗口在 `s` 上滑动。
2. 我们需要一种方法来快速判断窗口内的子串是否和 `p` 是异位词。
3. 使用哈希表（或数组）来存储 `p` 的字符频率 (`p_freq`) 和当前窗口内子串的字符频率 (`window_freq`)。
4. 窗口每滑动一次，我们只需要更新 `window_freq` 中两个字符的计数（一个移出，一个移入），然后比较 `window_freq` 和 `p_freq` 是否相等。

### ⌚ 第四步：滑动窗口算法细节

#### 1. 初始化：

- 创建 `p_freq` 哈希表并统计好 `p` 的字符频率。
- 创建 `window_freq` 哈希表。
- 初始化 `left = 0, right = 0`。
- 创建一个结果列表 `result`。

#### 2. 窗口滑动：

- `right` 指针不断向右移动，将 `s[right]` 加入窗口，并更新 `window_freq`。
- 判断窗口大小：`right - left + 1` 达到 `len(p)` 时，我们进行一次判断。
  - 比较频率：检查 `window_freq` 是否和 `p_freq` 完全相等。如果相等，说明找到了一个异位词子串，将当前的 `left` 存入 `result`。
  - 收缩窗口：为了保持窗口大小固定，我们将 `s[left]` 移出窗口（更新 `window_freq`），然后 `left++`。

**优化：**

每次都完整比较两个哈希表还是有点慢。我们可以用一个变量 `match_count` 来记录 `window_freq` 中有多少个字符的频率已经满足了 `p_freq` 的要求。

- 当 `right` 扩大窗口，`s[right]` 的频率在 `window_freq` 中从 `p_freq[char]-1` 变为 `p_freq[char]` 时，`match_count++`。
- 当 `left` 收缩窗口，`s[left]` 的频率在 `window_freq` 中从 `p_freq[char]` 变为 `p_freq[char]-1` 时，`match_count--`。
- 只有当 `match_count` 等于 `p_freq` 中不同字符的总数时，才说明窗口内的子串是一个异位词。

### ⌚ 第五步：算法步骤总结（优化后）

1. 初始化 `p_freq` 和 `window_freq` 两个哈希表（或长度26的数组）。
2. 统计 `p` 的字符频率到 `p_freq`。
3. 初始化 `left = 0, valid_chars = 0`（相当于上面说的 `match_count`），`result = []`。
4. `right` 指针遍历 `s`：
  - a. 获取 `char_in = s[right]`。
  - b. 如果 `char_in` 是 `p` 中需要的字符（即 `p_freq[char_in] > 0`），则更新 `window_freq`，并检查 `window_freq[char_in]` 是否已经等于 `p_freq[char_in]`。如果是，`valid_chars++`。
  - c. 检查窗口是否需要收缩：当窗口大小 `right - left + 1 >= len(p)` 时：
    - i. 如果 `valid_chars` 等于 `p_freq` 中不同字符的数量，说明找到了一个解，将 `left` 加入 `result`。
    - ii. 获取 `char_out = s[left]`。
    - iii. 如果 `char_out` 是 `p` 中需要的字符，更新 `window_freq`。如果更新前 `window_freq[char_out]` 正好等于 `p_freq[char_out]`，说明一个匹配的字符即将离开窗口，所以 `valid_chars--`。
    - iv. `left++`，收缩窗口。
5. 遍历结束，返回 `result`。

**时间复杂度：**  $O(N)$  -  $N$  是 `s` 的长度。`left` 和 `right` 指针都只遍历一次 `s`。

**空间复杂度：**  $O(K)$  -  $K$  是字符集的大小。因为题目说是小写字母，所以是  $O(26)$ ，即  $O(1)$ 。

## 代码实现

### Python

```
from collections import Counter

def findAnagrams(s: str, p: str) -> list[int]:
    """
    使用滑动窗口和哈希表寻找所有字母异位词的起始索引。
    """
    result = []
    # ... (implementation details)
    return result
```

```

if len(s) < len(p):
    return []

# 1. 初始化
p_freq = Counter(p)
window_freq = Counter()

result = []
left = 0

# 2. 滑动窗口
for right, char in enumerate(s):
    # 扩大窗口
    window_freq[char] += 1

    # 3. 当窗口大小达到 p 的长度时，开始判断和收缩
    if right - left + 1 == len(p):
        # 判断当前窗口是否为异位词
        if window_freq == p_freq:
            result.append(left)

        # 收缩窗口
        left_char = s[left]
        window_freq[left_char] -= 1
        if window_freq[left_char] == 0:
            del window_freq[left_char] # 保持 Counter 干净

        left += 1

return result

```

## 关键点总结

1. 异位词判断：核心是比较字符的种类和数量，使用哈希表（Counter）是最高效的方式。
2. 固定大小的滑动窗口：本题是滑动窗口的一个典型应用。窗口大小固定为 `len(p)`，随着 `right` 指针移动，`left` 指针也同步移动，保持窗口大小不变。
3. 频率更新：相比于每次都重新计算整个窗口的频率，滑动窗口的优势在于每次只更新移入和移出两个字符的频率，大大降低了计算量。

## 矩阵 / 240. 搜索二维矩阵 II

### 240. 搜索二维矩阵 II

#### 题目描述

编写一个高效的算法来搜索 `m x n` 矩阵 `matrix` 中的一个目标值 `target`。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

#### 示例

示例 1：

输入：`matrix = [[1, 4, 7, 11, 15], [2, 5, 8, 12, 19], [3, 6, 9, 16, 22], [10, 13, 14, 17, 24], [18, 21, 23, 26, 30]]`, `target = 5`  
输出：`true`

示例 2：

输入：`matrix = [[1, 4, 7, 11, 15], [2, 5, 8, 12, 19], [3, 6, 9, 16, 22], [10, 13, 14, 17, 24], [18, 21, 23, 26, 30]]`, `target = 20`  
输出：`false`

#### 约束条件

- `m == matrix.length`

- `n == matrix[0].length`
- `1 <= n, m <= 300`
- `-10^9 <= matrix[i][j] <= 10^9`
- `-10^9 <= target <= 10^9`

## 思考过程

### 💡 第一步：理解问题

- 我们有一个 `m x n` 的矩阵。
- 矩阵的行和列都是升序排列的。
- 我们需要判断一个目标值 `target` 是否存在于矩阵中。

### 💡 第二步：朴素解法

最直接的想法是，遍历整个矩阵，逐个比较元素是否等于 `target`。

**思考题：**这个方法的时间复杂度是多少？

▶ 点击查看分析

### 💡 第三步：利用矩阵特性

矩阵的特性是：每行升序，每列升序。这非常重要！

**思考题：**

- 如果我们从左上角开始搜索，会遇到什么问题？
- 如果我们从左下角或右上角开始搜索，会有什么优势？

▶ 点击查看分析

### 💡 第四步：算法步骤总结（从右上角开始）

1. 初始化 `row = 0, col = n - 1`。
2. 当 `row < m` 且 `col >= 0` 时，循环执行：
  - a. 获取当前元素 `current_val = matrix[row][col]`。
  - b. 如果 `current_val == target`，返回 `True`。
  - c. 如果 `current_val < target`，说明当前行所有元素都比 `target` 小，向下移动：`row++`。
  - d. 如果 `current_val > target`，说明当前列所有元素都比 `target` 大，向左移动：`col--`。
3. 如果循环结束仍未找到 `target`，返回 `False`。

**时间复杂度：**  $O(m + n)$  - 在最坏情况下，指针会从右上角移动到左下角，最多移动  $m + n$  步。

**空间复杂度：**  $O(1)$  - 只使用了常数个额外变量。

## 代码实现

### Python

```
def searchMatrix(matrix: list[list[int]], target: int) -> bool:
    """
    从右上角（或左下角）开始搜索，利用矩阵的有序性高效查找目标值。
    """

    if not matrix or not matrix[0]:
        return False

    m, n = len(matrix), len(matrix[0])

    # 从右上角开始搜索
    row = 0
    col = n - 1

    while row < m and col >= 0:
        current_val = matrix[row][col]

        if current_val == target:
            return True
        elif current_val < target:
            # 当前值小于目标值，说明当前行不可能有目标值（因为行是升序的）
            # 只能向下移动一行
            row += 1
        else:
            col -= 1
```

```

        row += 1
    else: # current_val > target
        # 当前值大于目标值，说明当前列不可能有目标值（因为列是升序的）
        # 只能向左移动一列
        col -= 1

    return False

```

## 关键点总结

- 选择合适的起点：**从右上角（或左下角）开始搜索，可以保证每次决策都能排除一行或一列，从而实现高效查找。
- 单向移动：**每次移动都只有一个方向，避免了二叉搜索中可能出现的两个分支。
- 时间复杂度优化：**将  $O(m \cdot n)$  的暴力解法优化到  $O(m+n)$ ，在大型矩阵中效率显著提升。

## 矩阵 / 48. 旋转图像

### 48. 旋转图像

#### 题目描述

给定一个  $n \times n$  的二维矩阵表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 **原地** 旋转图像，这意味着你不能使用另一个二维矩阵来旋转图像。请你直接修改输入的二维矩阵。

#### 示例

示例 1：

输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]  
输出：[[7,4,1],[8,5,2],[9,6,3]]

示例 2：

输入：matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,18,16]]  
输出：[[15,13,2,5],[14,3,4,1],[18,6,8,9],[16,7,10,11]]

#### 约束条件

- $n == \text{matrix.length} == \text{matrix[i].length}$
- $1 \leq n \leq 20$
- $-1000 \leq \text{matrix[i][j]} \leq 1000$

#### 思考过程

##### 💡 第一步：理解问题

- 我们有一个  $n \times n$  的正方形矩阵。
- 需要将其顺时针旋转 90 度。
- 必须原地旋转，不能使用额外的矩阵。

##### 💡 第二步：观察旋转规律

让我们以  $[[1,2,3],[4,5,6],[7,8,9]]$  为例，旋转后变为  $[[7,4,1],[8,5,2],[9,6,3]]$ 。

观察元素位置的变化：

- $\text{matrix}[0][0] \rightarrow \text{matrix}[0][2] \rightarrow \text{matrix}[2][2] \rightarrow \text{matrix}[2][0] \rightarrow \text{matrix}[0][0]$
- $\text{matrix}[i][j]$  旋转后会到哪里？

思考题：`matrix[row][col]` 旋转 90 度后，它的新位置 `matrix[new_row][new_col]` 是什么？

▶ 点击查看分析

##### 💡 第三步：原地旋转的挑战

如果我们直接按照 `matrix[row][col] -> matrix[col][n - 1 - row]` 的规则进行赋值，会覆盖掉原始数据，导致后续的旋转无法进行。

思考题：如何在原地完成旋转？

▶ 点击查看分析

##### 💡 第四步：方法二：先转置再反转（推荐）

这个方法更直观，也更容易实现。

### 1. 转置 (Transpose):

- 将矩阵沿着主对角线（从左上到右下）进行翻转。
- 也就是将 `matrix[i][j]` 和 `matrix[j][i]` 进行交换。
- 只需要遍历矩阵的上半部分（或下半部分），避免重复交换。
- 遍历 `i` 从 0 到 `n-1`，`j` 从 `i+1` 到 `n-1`。

例如 `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]` 转置后变为 `[[1, 4, 7], [2, 5, 8], [3, 6, 9]]`。

### 2. 反转每一行 (Reverse each row):

- 将转置后的矩阵的每一行进行左右反转。
- 也就是将 `matrix[i][j]` 和 `matrix[i][n-1-j]` 进行交换。

例如 `[[1, 4, 7], [2, 5, 8], [3, 6, 9]]` 反转每一行后变为 `[[7, 4, 1], [8, 5, 2], [9, 6, 3]]`。

**思考题：**为什么“先转置再反转每一行”能实现顺时针 90 度旋转？

► 点击查看分析

## 💡 第五步：算法步骤总结

### 1. 获取矩阵的维度 `n = len(matrix)`。

### 2. 第一步：转置矩阵

- 遍历 `i` 从 0 到 `n-1`。
- 遍历 `j` 从 `i+1` 到 `n-1`。
- 交换 `matrix[i][j]` 和 `matrix[j][i]`。

### 3. 第二步：反转每一行

- 遍历 `i` 从 0 到 `n-1`（每一行）。
- 对 `matrix[i]` 这一行进行左右反转（可以使用双指针 `left` 和 `right`）。

**时间复杂度：**  $O(n^2)$  - 两次遍历矩阵。

**空间复杂度：**  $O(1)$  - 原地操作。

## 代码实现

### Python

```
def rotate(matrix: list[list[int]]) -> None:
    """
    先转置再反转每一行，实现矩阵顺时针 90 度原地旋转。
    Do not return anything, modify matrix in-place instead.
    """
    n = len(matrix)

    # 1. 转置矩阵（主对角线翻转）
    for i in range(n):
        for j in range(i + 1, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]

    # 2. 反转每一行
    for i in range(n):
        # 对当前行进行左右反转
        left, right = 0, n - 1
        while left < right:
            matrix[i][left], matrix[i][right] = matrix[i][right], matrix[i][left]
            left += 1
            right -= 1
```

## 关键点总结

- 分解问题：**将复杂的 90 度旋转分解为两个简单的操作：转置和反转每一行。
- 原地操作：**通过直接修改矩阵元素，避免了额外的空间开销。
- 转置的遍历范围：**转置时只需要遍历矩阵的对角线以上（或以下）的部分，避免重复交换。

## 矩阵 / 54. 螺旋矩阵

### 54. 融合矩阵

#### 题目描述

给你一个  $m \times n$  矩阵 `matrix`，请按照顺时针螺旋顺序，返回矩阵中的所有元素。

#### 示例

示例 1：

输入： `matrix = [[1,2,3],[4,5,6],[7,8,9]]`

输出： `[1,2,3,6,9,8,7,4,5]`

示例 2：

输入： `matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]`

输出： `[1,2,3,4,8,12,11,10,9,5,6,7]`

#### 约束条件

- $m == matrix.length$
- $n == matrix[i].length$
- $1 \leq m, n \leq 10$
- $-100 \leq matrix[i][j] \leq 100$

#### 思考过程

##### 思考 1：理解问题

- 我们需要按照顺时针螺旋的顺序遍历矩阵中的所有元素。
- 最终返回一个包含所有元素的列表。

##### 思考 2：模拟路径

螺旋遍历的路径是：从左到右 -> 从上到下 -> 从右到左 -> 从下到上，然后缩小一圈，重复这个过程。

##### 思考题：

- 如何控制遍历的边界？
- 什么时候停止遍历？

▶ 点击查看分析

##### 思考 3：算法步骤总结

1. 初始化结果列表 `result`。
2. 初始化四个边界变量：`top = 0, bottom = m - 1, left = 0, right = n - 1`。
3. 当 `top <= bottom` 且 `left <= right` 时，循环执行：
  - a. 从左到右：遍历 `matrix[top][j]`，`j` 从 `left` 到 `right`。遍历完后，`top++`。
  - b. 从上到下：如果 `top <= bottom`，遍历 `matrix[i][right]`，`i` 从 `top` 到 `bottom`。遍历完后，`right--`。
  - c. 从右到左：如果 `left <= right` 且 `top <= bottom`，遍历 `matrix[bottom][j]`，`j` 从 `right` 到 `left` (倒序)。遍历完后，`bottom--`。
  - d. 从下到上：如果 `left <= right` 且 `top <= bottom`，遍历 `matrix[i][left]`，`i` 从 `bottom` 到 `top` (倒序)。遍历完后，`left++`。
4. 返回 `result`。

时间复杂度：O( $m \times n$ ) - 每个元素只遍历一次。

空间复杂度：O(1) - 不考虑结果列表的额外空间。

#### 代码实现

##### Python

```
def spiralOrder(matrix: list[list[int]]) -> list[int]:  
    """  
    按照顺时针螺旋顺序遍历矩阵。  
    """  
  
    if not matrix or not matrix[0]:  
        return []  
  
    m, n = len(matrix), len(matrix[0])  
    result = []
```

```

# 定义四个边界
top, bottom = 0, m - 1
left, right = 0, n - 1

while top <= bottom and left <= right:
    # 1. 从左到右遍历上边界
    for j in range(left, right + 1):
        result.append(matrix[top][j])
    top += 1

    # 2. 从上到下遍历右边界
    if top <= bottom: # 检查是否还有行可遍历
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1

    # 3. 从右到左遍历下边界
    if top <= bottom and left <= right: # 检查是否还有行和列可遍历
        for j in range(right, left - 1, -1):
            result.append(matrix[bottom][j])
        bottom -= 1

    # 4. 从下到上遍历左边界
    if top <= bottom and left <= right: # 检查是否还有行和列可遍历
        for i in range(bottom, top - 1, -1):
            result.append(matrix[i][left])
        left += 1

return result

```

## 关键点总结

1. **边界控制**: 使用 `top`, `bottom`, `left`, `right` 四个变量来精确控制每次遍历的范围，并及时收缩边界。
2. **循环条件**: `while top <= bottom and left <= right` 确保在所有元素遍历完毕后停止。
3. **特殊情况处理**: 在每次遍历一个方向后，需要再次检查 `top <= bottom` 和 `left <= right`，以防止在奇数行/列或单行/单列矩阵中重复遍历或越界。

## 矩阵 / 73. 矩阵置零

### 73. 矩阵置零

#### 题目描述

给定一个 `m x n` 的矩阵，如果一个元素为 `0`，则将其所在行和列的所有元素都设为 `0`。请使用 **原地** 算法。

#### 示例

示例 1:

输入: `matrix = [[1,1,1],[1,0,1],[1,1,1]]`  
输出: `[[1,0,1],[0,0,0],[1,0,1]]`

示例 2:

输入: `matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]`  
输出: `[[0,0,0,0],[0,4,5,0],[0,3,1,0]]`

#### 约束条件

- `m == matrix.length`
- `n == matrix[0].length`
- `1 <= m, n <= 200`

```
• -2^31 <= matrix[i][j] <= 2^31 - 1
```

## 思考过程

### 思考题：理解问题

- 我们有一个  $m \times n$  的矩阵。
- 如果矩阵中的某个元素是 0，那么它所在的整行和整列都要被置为 0。
- 必须使用原地算法，这意味着不能使用额外的矩阵来存储结果。

### 思考题：朴素解法（额外空间）

最直观的想法是，先遍历一遍矩阵，记录所有 0 的位置。然后，再遍历一遍矩阵，根据记录的位置将对应的行和列置为 0。

思考题：

- 如何记录 0 的位置？
- 这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

### 思考题：原地算法的挑战

题目要求原地算法，即空间复杂度为  $O(1)$ 。这意味着我们不能使用额外的  $O(m+n)$  空间来记录需要置零的行和列。

思考题：

- 如果我们直接在原矩阵上置零，会发生什么问题？

▶ 点击查看分析

### 思考题：利用第一行/列作为标记

既然不能用额外的  $O(m+n)$  空间，那我们能不能利用矩阵本身的空间呢？

核心思路：

- 我们可以将第一行和第一列作为“标记”数组。
- 如果 `matrix[i][j]` 为 0，我们就将 `matrix[i][0]` 和 `matrix[0][j]` 置为 0。

思考题：

- 这种方法有什么潜在的问题？
- 如何解决第一行和第一列本身可能包含 0 的情况？

▶ 点击查看分析

### 思考题：算法步骤总结 ( $O(1)$ 空间)

#### 1. 检查第一列是否需要置零：

- 用一个布尔变量 `col0_has_zero` 记录第一列是否有 0。
- 遍历第一列，如果遇到 `matrix[i][0] == 0`，则将 `col0_has_zero` 设为 `True`。

#### 2. 使用第一行/列作为标记：

- 从 `matrix[0][1]` 开始遍历矩阵（跳过第一列）。
- 如果 `matrix[i][j] == 0`，则将 `matrix[i][0]` 和 `matrix[0][j]` 置为 0。

#### 3. 根据标记置零（从后往前）：

- 从 `matrix[m-1][n-1]` 开始，从下往上，从右往左遍历矩阵（跳过第一行和第一列）。
- 如果 `matrix[i][0] == 0` 或者 `matrix[0][j] == 0`，则将 `matrix[i][j]` 置为 0。
- 为什么从后往前？这样可以避免在置零过程中，将原本不是 0 的标记位也置为 0，从而影响后续的判断。

#### 4. 处理第一行：

- 如果 `matrix[0][0] == 0`，则将第一行所有元素置为 0。

#### 5. 处理第一列：

- 如果 `col0_has_zero` 为 `True`，则将第一列所有元素置为 0。

时间复杂度： $O(m \cdot n)$  - 几次遍历。

空间复杂度： $O(1)$  - 只使用了一个布尔变量。

## 代码实现

### Python

```
def setZeroes(matrix: list[list[int]]) -> None:  
    """  
    使用第一行/列作为标记，实现  $O(1)$  空间复杂度的矩阵置零。  
    Do not return anything, modify matrix in-place instead.  
    """
```

```

"""
m, n = len(matrix), len(matrix[0])
col0_has_zero = False

# 1. 检查第一列是否有 0，并用第一行/列作为标记
for i in range(m):
    if matrix[i][0] == 0:
        col0_has_zero = True

# 从第二列开始遍历，如果遇到 0，则标记对应的行和列
for j in range(1, n):
    if matrix[i][j] == 0:
        matrix[i][0] = 0 # 标记行
        matrix[0][j] = 0 # 标记列

# 2. 根据标记，从后往前置零（避免影响标记位）
# 注意：这里从 m-1 开始，到 0 结束，步长为 -1
for i in range(m - 1, -1, -1):
    # 注意：这里从 n-1 开始，到 1 结束，步长为 -1
    for j in range(n - 1, 0, -1):
        if matrix[i][0] == 0 or matrix[0][j] == 0:
            matrix[i][j] = 0

# 3. 最后处理第一列（根据 col0_has_zero 标记）
if col0_has_zero:
    matrix[i][0] = 0

```

## 关键点总结

- 原地标记：**利用矩阵的第一行和第一列作为标记空间，是实现  $O(1)$  空间复杂度的核心。
- 特殊处理第一列：**由于 `matrix[0][0]` 同时作为第一行和第一列的标记，需要一个额外的布尔变量来独立记录第一列是否需要置零。
- 倒序遍历：**在根据标记置零时，从矩阵的右下角开始，向左上角遍历，可以避免在置零过程中，将原本不是 0 的标记位也置为 0，从而影响后续的判断。

## 贪心算法 / 121. 买卖股票的最佳时机

### 121. 买卖股票的最佳时机

#### 题目描述

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天卖出该股票。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。

```

#### 示例 2：

```

输入: prices = [7,6,4,3,1]
输出: 0
解释: 在这种情况下，没有交易完成，所以最大利润为 0。

```

#### 提示：

- $1 \leq \text{prices.length} \leq 10^5$

- $0 \leq \text{prices}[i] \leq 10^4$

#### 思考过程:

这道题要求我们找到在给定股票价格数组中，通过一次买入和一次卖出操作，能够获得的最大利润。

#### 核心思路:

为了获得最大利润，我们应该在价格最低的时候买入，在价格最高的时候卖出。由于我们只能先买后卖，所以我们只需要记录到目前为止的最低价格，然后用当前价格减去这个最低价格，就能得到当前的利润。不断更新这个最大利润即可。

#### 具体步骤:

##### 1. 初始化:

- `min_price`: 记录到目前为止的最低价格，初始化为一个非常大的值（例如 `float('inf')` 或 `prices[0]`）。
- `max_profit`: 记录到目前为止的最大利润，初始化为 `0`。

##### 2. 遍历价格数组:

- 对于 `prices` 数组中的每一个价格 `price`:
  - 更新 `min_price = min(min_price, price)`。
  - 计算当前利润 `current_profit = price - min_price`。
  - 更新 `max_profit = max(max_profit, current_profit)`。

##### 3. 返回 `max_profit`。

#### Python 代码:

```
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        min_price = float('inf')
        max_profit = 0

        for price in prices:
            min_price = min(min_price, price)
            max_profit = max(max_profit, price - min_price)

        return max_profit
```

## 贪心算法 / 45. 跳跃游戏 II

### 45. 跳跃游戏 II

#### 题目描述

给定一个长度为  $n$  的 0 索引整数数组 `nums`。初始位置为 `nums[0]`。

每个元素 `nums[i]` 表示从索引  $i$  向前跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处：

- $0 \leq j \leq \text{nums}[i]$
- $i + j < n$

返回到达 `nums[n - 1]` 的最小跳跃次数。生成的测试用例可以到达 `nums[n - 1]`。

#### 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。
```

#### 示例 2:

```
输入: nums = [2,3,0,1,4]
输出: 2
```

#### 提示:

- $1 \leq \text{nums.length} \leq 10^4$

- $0 \leq \text{nums}[i] \leq 1000$
- 题目保证可以到达  $\text{nums}[n-1]$

#### 思考过程:

这道题要求找到从数组的第一个位置跳到最后一个位置所需的最小跳跃次数。数组中的每个元素  $\text{nums}[i]$  代表从该位置可以跳跃的最大长度。

#### 核心思路 (贪心算法):

这道题可以使用贪心算法来解决。我们希望在每一步都跳到最远的位置，以减少总的跳跃次数。

我们可以维护三个变量：

1. `jumps`：记录跳跃次数。
2. `current_reach`：当前跳跃能够到达的最远位置。
3. `next_reach`：下一次跳跃能够到达的最远位置。

#### 具体步骤:

##### 1. 初始化:

- `jumps = 0`
- `current_reach = 0`
- `next_reach = 0`

##### 2. 遍历数组 (除了最后一个元素):

- `for i in range(len(nums) - 1):`
- 更新 `next_reach = max(next_reach, i + nums[i])`。
- 如果当前位置 `i` 达到了 `current_reach`，说明需要进行一次跳跃。
  - `jumps += 1`
  - `current_reach = next_reach`
- 剪枝：如果 `current_reach` 已经能够到达或超过数组的最后一个位置 (`len(nums) - 1`)，说明已经到达终点，直接返回 `jumps`。

##### 3. 返回 `jumps`。

#### Python 代码:

```
class Solution:
    def jump(self, nums: list[int]) -> int:
        n = len(nums)
        if n <= 1:
            return 0

        jumps = 0
        current_reach = 0 # 当前跳跃能到达的最远位置
        next_reach = 0 # 下一次跳跃能到达的最远位置

        for i in range(n - 1):
            # 更新下一次跳跃能到达的最远位置
            next_reach = max(next_reach, i + nums[i])

            # 如果当前位置达到了当前跳跃能到达的最远位置，说明需要进行一次跳跃
            if i == current_reach:
                jumps += 1
                current_reach = next_reach

            # 如果已经到达或超过终点，直接返回跳跃次数
            if current_reach >= n - 1:
                return jumps

        return jumps
```

## 贪心算法 / 55. 跳跃游戏

### 55. 跳跃游戏

#### 题目描述

给你一个非负整数数组 `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，如果可以，返回 `true`；否则，返回 `false`。

#### 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。

#### 提示：

- $1 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 10^5$

#### 思考过程：

这道题要求判断是否能够从数组的第一个位置跳到最后一个位置。数组中的每个元素 `nums[i]` 代表从该位置可以跳跃的最大长度。

#### 核心思路 (贪心算法)：

我们可以维护一个变量 `max_reach`，表示当前能够到达的最远位置。

1. 从数组的第一个位置开始遍历。
2. 在遍历过程中，更新 `max_reach = max(max_reach, i + nums[i])`。
3. 如果当前位置 `i` 已经超过了 `max_reach`，说明无法到达当前位置，也就无法到达终点，直接返回 `False`。
4. 如果 `max_reach` 已经能够到达或超过数组的最后一个位置 (`len(nums) - 1`)，说明可以到达终点，返回 `True`。

#### Python 代码：

```
class Solution:  
    def canJump(self, nums: list[int]) -> bool:  
        max_reach = 0  
        n = len(nums)  
  
        for i in range(n):  
            # 如果当前位置已经超过了最大可达范围，说明无法到达当前位置，也就无法到达终点  
            if i > max_reach:  
                return False  
  
            # 更新最大可达范围  
            max_reach = max(max_reach, i + nums[i])  
  
            # 如果最大可达范围已经到达或超过终点，说明可以到达  
            if max_reach >= n - 1:  
                return True  
  
        return False # 遍历完所有位置，如果还没返回True，说明无法到达
```

## 贪心算法 / 763. 划分子母区间

### 763. 划分子母区间

#### 题目描述

给你一个字符串 `s`。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。

注意，划分结果需要满足：将所有划分结果按顺序连接，得到的字符串仍然是 `s`。

返回一个表示每个字符串片段的长度的列表。

#### 示例 1：

输入: `s = "ababcbacadefegdehijhklij"`

输出: `[9, 7, 8]`

解释:

划分结果为 `"ababcaca"`、`"defegde"`、`"hijhklij"`。

每个字母最多出现在一个片段中。

像 `"ababcbacadefegde"`，`"hijhklij"` 这样的划分是错误的，因为划分的片段数较少。

#### 示例 2：

输入: `s = "eccbbbdec"`

输出: `[10]`

#### 提示:

- `1 <= s.length <= 500`
- `s` 仅由小写英文字母组成

#### 思考过程:

这道题要求将一个字符串 `s` 划分为尽可能多的片段，使得每个片段中的每个字母只在该片段中出现一次。

#### 核心思路 (贪心算法):

为了将字符串划分为尽可能多的片段，我们应该让每个片段尽可能短。

对于每个字符，我们需要知道它最后一次出现的位置。

然后，我们从字符串的开头开始遍历，维护当前片段的结束位置 `end`。

当遍历到某个字符时，更新 `end` 为当前字符最后一次出现的位置和当前 `end` 的最大值。

如果当前遍历到的索引 `i` 等于 `end`，说明当前片段已经完整，可以将其作为一个独立的片段。

#### 具体步骤:

##### 1. 记录每个字符最后一次出现的位置:

- 使用一个字典或数组 `last_occurrence` 来存储每个字符最后一次出现的索引。
- 遍历字符串 `s`，`last_occurrence[char] = index`。

##### 2. 遍历字符串进行划分:

- `start = 0` (当前片段的起始索引)
- `end = 0` (当前片段的结束索引)
- `result = []` (存储每个片段的长度)
- `for i, char in enumerate(s):`
- 更新 `end = max(end, last_occurrence[char])`。
- 如果 `i == end`:
  - 说明当前片段已经完整，将其长度添加到 `result`。
  - `result.append(end - start + 1)`。
  - 更新 `start = i + 1` (开始下一个片段)。

##### 3. 返回 `result`。

#### Python 代码:

```
class Solution:  
    def partitionLabels(self, s: str) -> list[int]:  
        last_occurrence = {}  
        for i, char in enumerate(s):  
            last_occurrence[char] = i  
  
        result = []  
        start = 0  
        end = 0  
  
        for i, char in enumerate(s):  
            if i > end:  
                result.append(end - start + 1)  
                start = i  
            end = max(end, last_occurrence[char])
```

```

    # 更新当前片段的结束位置
    end = max(end, last_occurrence[char])

    # 如果当前遍历到的索引等于当前片段的结束位置，说明可以划分一个片段了
    if i == end:
        result.append(end - start + 1)
        start = i + 1 # 更新下一个片段的起始位置

    return result

```

## 链表 / 138. 随机链表的复制

### 138. 随机链表的复制

#### 题目描述

给你一个长度为  $n$  的链表，每个节点包含一个额外增加的随机指针 `random`，该指针可以指向链表中的任何节点或 `null`。

构造这个链表的 **深拷贝**。深拷贝应该正好由  $n$  个新节点组成，其中每个新节点的值与其对应的原节点的值相同。新节点的 `next` 指针和 `random` 指针也都应指向复制链表中的新节点，并与原链表中的对应指针保持一致。也就是说，如果一个节点  $N$  的 `next` 指针指向  $M$ ，那么在复制链表上，对应的  $N'$  的 `next` 指针也应指向  $M'$ 。如果一个节点  $N$  的 `random` 指针指向  $X$ ，那么在复制链表上，对应的  $N'$  的 `random` 指针也应指向  $X'$ 。

返回复制链表的头节点。

#### 提示：

- 链表中的节点数目在范围  $[0, 1000]$  内。
- $-10^4 \leq \text{Node.val} \leq 10^4$
- `random` 指针可能指向 `null`，或者指向链表中的任何节点。

#### 思考过程

##### 💡 第一步：理解问题

- 我们需要对一个带有 `random` 指针的链表进行**深拷贝**。
- 深拷贝意味着所有节点都是新的，并且 `next` 和 `random` 指针都指向新链表中的对应节点。

##### 💡 第二步：朴素解法（哈希表）

最直观的想法是，分两步进行复制：

- 复制节点：**遍历原链表，为每个节点创建一个新节点，并建立原节点到新节点的映射关系（使用哈希表）。同时复制 `val` 和 `next` 指针（指向新节点）。
- 复制 `random` 指针：**再次遍历原链表，对于每个原节点 `old_node`，找到其对应的 `new_node`。然后，如果 `old_node.random` 不为 `None`，就将 `new_node.random` 指向 `old_node.random` 对应的新节点（通过哈希表查找）。

#### 思考题：

- 这个方法的时间和空间复杂度是多少？

► 点击查看分析

##### 💡 第三步：O(1) 空间的关键 → 穿插节点

为了实现  $O(1)$  空间，我们不能使用额外的哈希表。我们需要一种方法，在不使用额外空间的情况下，快速找到原节点对应的复制节点。

#### 核心思路：

- 将复制节点穿插到原节点之后。
- 例如： $A \rightarrow B \rightarrow C$  变为  $A \rightarrow A' \rightarrow B \rightarrow B' \rightarrow C \rightarrow C'$ 。
- 这样，对于任何一个原节点 `old_node`，它的复制节点 `new_node` 总是 `old_node.next`。

#### 算法流程：

##### 1. 第一次遍历：复制节点并穿插

- 遍历原链表，对于每个原节点 `curr`：
  - 创建一个新节点 `new_node`，值为 `curr.val`。
  - 将 `new_node` 插入到 `curr` 和 `curr.next` 之间：`new_node.next = curr.next, curr.next = new_node`。
  - `curr` 移动到 `new_node.next`（即原链表的下一个节点）。

##### 2. 第二次遍历：复制 `random` 指针

- 遍历原链表（现在是  $A \rightarrow A' \rightarrow B \rightarrow B' \dots$  这种结构）。
- 对于每个原节点 `curr`（即  $A, B, C\dots$ ）：

- 它的复制节点是 `curr.next` (即 A', B', C' ... )。
- 如果 `curr.random` 不为 `None`，那么 `curr.next.random` 应该指向 `curr.random` 对应的复制节点。由于复制节点就在原节点之后，所以 `curr.next.random = curr.random.next`。

### 3. 第三次遍历：拆分链表

- 将原链表和复制链表分离。
- 初始化 `new_head = head.next` (复制链表的头节点)。
- 遍历原链表，对于每个原节点 `curr`：
  - `new_node = curr.next`。
  - `curr.next = new_node.next` (恢复原链表)。
  - `new_node.next = curr.next.next` (连接复制链表)。
- 返回 `new_head`。

时间复杂度：O(n) - 三次遍历链表。

空间复杂度：O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for a Node.
class Node:
    def __init__(self, x: int, next: 'Node' = None, random: 'Node' = None):
        self.val = int(x)
        self.next = next
        self.random = random

def copyRandomList(head: 'Node') -> 'Node':
    """
    使用穿插节点法复制带有随机指针的链表。
    """

    if not head:
        return None

    # 1. 复制节点并穿插到原节点之后
    curr = head
    while curr:
        new_node = Node(curr.val, curr.next, None)
        curr.next = new_node
        curr = new_node.next

    # 2. 复制 random 指针
    curr = head
    while curr:
        if curr.random:
            curr.next.random = curr.random.next
        curr = curr.next.next # 跳过复制节点，移动到下一个原节点

    # 3. 拆分链表
    new_head = head.next
    curr = head
    while curr:
        new_node = curr.next
        curr.next = new_node.next # 恢复原链表
        if new_node.next: # 避免空指针异常
            new_node.next = new_node.next.next # 连接复制链表
        else:
            new_node.next = None # 最后一个复制节点的 next 为 None
        curr = curr.next # 移动到下一个原节点
```

```
return new_head
```

## 关键点总结

1. 穿插节点：将复制节点插入到原节点之后，巧妙地建立了原节点与其复制节点之间的  $O(1)$  映射关系，从而避免了哈希表。
2. 三步走策略：
  - 第一步：复制节点并穿插。
  - 第二步：复制 `random` 指针（利用穿插后的映射关系）。
  - 第三步：拆分链表，恢复原链表并形成新的复制链表。
3.  $O(1)$  空间：通过修改链表结构本身来存储映射关系，实现了常数级别的空间复杂度。

## 链表 / 141. 环形链表

### 141. 环形链表

#### 题目描述

给你一个链表的头节点 `head`，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表中节点的索引（从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。**注意：**`pos` 不作为参数进行传递，仅仅是为了标识环的实际位置。

如果链表中存在环，则返回 `true`。否则，返回 `false`。

#### 示例

示例 1：

输入: `head = [3, 2, 0, -4]`, `pos = 1`

输出: `true`

解释: 链表中有一个环，其尾部连接到第二个节点。

示例 2：

输入: `head = [1, 2]`, `pos = 0`

输出: `true`

解释: 链表中有一个环，其尾部连接到第一个节点。

示例 3：

输入: `head = [1]`, `pos = -1`

输出: `false`

解释: 链表中没有环。

#### 约束条件

- 链表中节点的数目范围是  $[0, 10^4]$
- $-10^5 \leq \text{Node.val} \leq 10^5$
- `pos` 为 -1 或者链表中的一个 **有效索引**。

#### 思考过程

##### 💡 第一步：理解问题

- 我们需要判断一个单链表是否包含环。
- 环意味着链表的某个节点的 `next` 指针指向了链表前面的某个节点，形成一个闭环。
- `pos` 只是为了说明问题，我们不能直接使用它。

##### 💡 第二步：朴素解法（哈希表）

最直观的想法是，遍历链表，并将访问过的节点存储到一个哈希集合中。如果再次访问到哈希集合中已有的节点，就说明存在环。

#### 思考题：

- 这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

##### 💡 第三步： $O(1)$ 空间的关键 → 快慢指针（Floyd 判圈算法）

为了实现  $O(1)$  空间，我们不能使用额外的存储。我们需要一种巧妙的方法来检测环。

## 核心思路：

- 使用两个指针，一个快指针 `fast`，一个慢指针 `slow`。
- `fast` 每次移动两步，`slow` 每次移动一步。
- 如果链表中存在环，那么 `fast` 指针最终一定会追上 `slow` 指针。
- 如果链表中没有环，`fast` 指针会先到达链表末尾（`None`）。

## 思考题：

- 为什么快指针一定会追上慢指针？

► 点击查看分析

## 💡 第四步：算法步骤总结

- 处理特殊情况：如果链表为空或只有一个节点，不可能有环，返回 `False`。
- 初始化快慢指针：`slow = head, fast = head.next`。
- 当 `fast` 和 `fast.next` 都不为 `None` 时，循环执行：
  - 如果 `slow == fast`，说明相遇，存在环，返回 `True`。
  - `slow` 移动一步：`slow = slow.next`。
  - `fast` 移动两步：`fast = fast.next.next`。
- 如果循环结束，`fast` 或 `fast.next` 为 `None`，说明 `fast` 已经到达链表末尾，没有环，返回 `False`。

时间复杂度：O(n) - 快慢指针最多遍历链表两次。

空间复杂度：O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def hasCycle(head: ListNode) -> bool:
    """
    使用快慢指针（Floyd 判圈算法）判断链表是否有环。
    """
    if not head or not head.next:
        return False

    slow = head
    fast = head.next # fast 比 slow 先走一步，避免一开始就相等

    while fast and fast.next:
        if slow == fast:
            return True # 快慢指针相遇，存在环

        slow = slow.next
        fast = fast.next.next

    return False # fast 到达链表末尾，没有环
```

## 关键点总结

- 快慢指针：`fast` 每次走两步，`slow` 每次走一步。这是检测环的核心思想。
- 相遇即有环：如果 `fast` 和 `slow` 相遇，则链表中存在环。
- `fast` 到达末尾即无环：如果 `fast` 或 `fast.next` 变为 `None`，说明链表没有环。
- O(1) 空间：通过巧妙的指针移动，实现了常数级别的空间复杂度。

## 链表 / 142. 环形链表 II

## 142. 环形链表 II

### 题目描述

给定一个链表的头节点 `head`，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表中节点的索引（从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识环的实际位置。

不允许修改给定的链表。

### 示例

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。

示例 3：

输入: `head = [1]`, `pos = -1`

输出: 返回 `null`

解释: 链表中没有环。

### 约束条件

- 链表中节点的数目范围在  $[0, 10^4]$  内
- $-10^5 \leq \text{Node.val} \leq 10^5$
- `pos` 的值为 -1 或者链表中的一个有效索引。

### 思考过程

#### 💡 第一步：理解问题

- 我们需要找到链表开始入环的第一个节点。
- 如果没有环，返回 `None`。
- 不允许修改链表。

#### 💡 第二步：如何检测环？

这和 141 题一样，可以使用快慢指针（Floyd 判圈算法）。

- `slow` 每次走一步，`fast` 每次走两步。
- 如果 `slow` 和 `fast` 相遇，说明有环。
- 如果 `fast` 或 `fast.next` 走到 `None`，说明无环。

#### 💡 第三步：如何找到入环点？

这是本题的关键。当快慢指针相遇时，它们都在环内。如何从相遇点找到环的入口？

#### 数学推导：

- 设链表头到环入口的距离为 `a`。
- 设环的长度为 `b`。
- 设相遇点距离环入口的距离为 `k`。
- `slow` 走过的距离:  $a + k$
- `fast` 走过的距离:  $a + k + n * b$  (其中 `n` 是 `fast` 在环中多走的圈数)

因为 `fast` 的速度是 `slow` 的两倍，所以 `fast` 走过的距离是 `slow` 的两倍：

$$\begin{aligned} 2 * (a + k) &= a + k + n * b \\ 2a + 2k &= a + k + n * b \\ a + k &= n * b \\ a &= n * b - k \\ a &= (n - 1) * b + (b - k) \end{aligned}$$

这个公式告诉我们：

- 从链表头到环入口的距离 `a`。

- 从相遇点沿着环的方向走到环入口的距离  $b - k$ 。

#### 关键洞察：

- 如果我们让一个指针从链表头开始，另一个指针从相遇点开始，它们都以每次一步的速度前进。
- 当它们相遇时，相遇点就是环的入口！

#### 为什么？

- 从链表头到环入口的距离是  $a$ 。
- 从相遇点到环入口的距离是  $b - k$ 。
- 我们的推导  $a = (n - 1) * b + (b - k)$  表明，从链表头走  $a$  步，和从相遇点走  $b - k$  步（或者说  $(n-1)$  圈加上  $b-k$  步），它们最终都会到达环的入口。

### 💡 第四步：算法步骤总结

#### 1. 检测环：

- 使用快慢指针 `slow` 和 `fast`，都从 `head` 开始。
- `slow` 每次走一步，`fast` 每次走两步。
- 如果 `fast` 或 `fast.next` 变为 `None`，说明无环，返回 `None`。
- 如果 `slow == fast`，说明有环，跳出循环。

#### 2. 寻找入环点：

- 将 `slow` 指针重新指向 `head`。
- 此时，`slow` 和 `fast` 都以每次一步的速度前进。
- 当 `slow == fast` 时，它们相遇的节点就是环的入口。

3. 返回相遇的节点。

时间复杂度：O(n) - 快慢指针最多遍历链表两次。

空间复杂度：O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def detectCycle(head: ListNode) -> ListNode:
    """
    使用快慢指针(Floyd 判圈算法) 检测环并寻找入环点。
    """

    if not head or not head.next:
        return None

    slow = head
    fast = head

    # 1. 检测环: 快慢指针相遇
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            break # 相遇, 存在环
    else:
        return None # fast 到达链表末尾, 没有环

    # 2. 寻找入环点
    # 将 slow 重新指向 head
    # slow 和 fast 同时以一步的速度前进, 相遇点即为环的入口
    slow = head
    while slow != fast:
```

```

        slow = slow.next
        fast = fast.next

    return slow # 返回环的入口节点

```

## 关键点总结

1. 环的检测：与 141 题相同，使用快慢指针判断链表是否存在环。
2. 入环点的数学推导：通过  $2 * (a + k) = a + k + n * b$  推导出  $a = (n - 1) * b + (b - k)$ ，这是找到入环点的理论基础。
3. 双指针策略：当快慢指针相遇后，将其中一个指针（通常是 `slow`）重新指向链表头，然后两个指针都以一步的速度前进，它们再次相遇的节点就是环的入口。

## 链表 / 146. LRU 缓存

### 146. LRU 缓存

#### 题目描述

请你设计并实现一个满足 **LRU (最近最少使用) 缓存** 约束的数据结构。

实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以正整数作为容量 `capacity` 初始化 LRU 缓存。
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1`。每次访问（`get` 或 `put` 操作）都会使该关键字成为最近最少使用的。
- `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值；如果关键字不存在，则插入该组 `(key, value)`。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的关键字。

函数 `get` 和 `put` 必须以  $O(1)$  的平均时间复杂度运行。

#### 示例

示例 1：

输入

```
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "put", "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 (最近使用过 key 1) 缓存是 {2=2, 1=1}
lRUCache.put(3, 3); // 该操作会使得 key 2 作废，因为最近最少使用，并且容量已满，所以缓存是 {1=1, 3=3}
lRUCache.get(2); // 返回 -1 (未找到 key 2)
lRUCache.put(4, 4); // 该操作会使得 key 1 作废，因为最近最少使用，并且容量已满，所以缓存是 {3=3, 4=4}
lRUCache.get(1); // 返回 -1 (未找到 key 1)
lRUCache.get(3); // 返回 3
lRUCache.get(4); // 返回 4
```

#### 约束条件

- $1 \leq capacity \leq 3000$
- $0 \leq key \leq 10^4$
- $0 \leq value \leq 10^5$
- 最多调用  $2 * 10^5$  次 `get` 和 `put` 方法

#### 思考过程

##### 💡 第一步：理解问题

- 我们需要实现一个 LRU 缓存。
- **LRU (Least Recently Used)**：最近最少使用的元素会被淘汰。
- `get(key)`：如果存在，返回 `value`，并更新为最近使用。否则返回 `-1`。
- `put(key, value)`：如果 `key` 存在，更新 `value`。如果 `key` 不存在：

- 如果容量未满，直接添加。
- 如果容量已满，淘汰最近最少使用的元素，再添加。
- `get` 和 `put` 必须是  $O(1)$  平均时间复杂度。

## 💡 第二步：数据结构的选择

要实现  $O(1)$  的 `get` 和 `put`，我们需要：

1. **快速查找**：根据 `key` 快速找到 `value`。这需要一个哈希表（字典）。
2. **快速删除最近最少使用的元素**：这需要一个能够维护元素访问顺序，并且能够快速删除头部（最近最少使用）和尾部（最近使用）的数据结构。双向链表是理想的选择。

核心思路：

- **哈希表 (Dictionary)**：存储 `key` 到双向链表节点的映射。`{key: Node}`。
- **双向链表 (Doubly Linked List)**：维护元素的访问顺序。链表头部是最近最少使用的元素，链表尾部是最近使用的元素。

## 💡 第三步：双向链表的设计

为了方便操作，我们通常会使用一个伪头节点 (`dummy head`) 和一个伪尾节点 (`dummy tail`)。

- `head` 节点：`head.next` 永远指向最近最少使用的元素。
- `tail` 节点：`tail.prev` 永远指向最近使用的元素。

操作：

- **添加节点到链表尾部 (最近使用)**: `add_node(node)`
  - `node.prev = tail.prev`
  - `node.next = tail`
  - `tail.prev.next = node`
  - `tail.prev = node`
- **删除节点 (最近最少使用)**: `remove_node(node)`
  - `node.prev.next = node.next`
  - `node.next.prev = node.prev`
- **将节点移动到链表尾部 (更新为最近使用)**: `move_to_tail(node)`
  - 先 `remove_node(node)`
  - 再 `add_node(node)`

## 💡 第四步：`get` 和 `put` 操作的实现

`get(key)`

1. 在哈希表中查找 `key`。
2. 如果 `key` 不存在，返回 `-1`。
3. 如果 `key` 存在：
  - 获取对应的节点 `node`。
  - 将 `node` 从当前位置移除，并移动到链表尾部（表示最近使用）。
  - 返回 `node.val`。

`put(key, value)`

1. 在哈希表中查找 `key`。
2. 如果 `key` 存在：
  - 更新 `node.val = value`。
  - 将 `node` 移动到链表尾部（表示最近使用）。
3. 如果 `key` 不存在：
  - 创建新节点 `new_node(key, value)`。
  - 如果缓存已满 (`len(cache) == capacity`)：
    - 获取最近最少使用的节点 (`head.next`)。
    - 从哈希表中删除该节点的 `key`。
    - 从链表中删除该节点。
    - 将 `new_node` 添加到链表尾部。
    - 将 `new_node` 的 `key` 和 `new_node` 存入哈希表。

## 💡 第五步：算法步骤总结

1. **Node 类**：定义双向链表节点，包含 `key`, `val`, `prev`, `next`。

2. **LRUCache 类**:

- `__init__(self, capacity):`
  - `self.capacity`
  - `self.cache = {} (哈希表)`
  - `self.head = Node(0, 0) (伪头节点)`

```

    • self.tail = Node(0, 0) (伪尾节点)
    • self.head.next = self.tail
    • self.tail.prev = self.head

• _add_node(self, node) : 将节点添加到链表尾部。
• _remove_node(self, node) : 从链表中移除节点。
• _move_to_tail(self, node) : 将节点移动到链表尾部。
• get(self, key) :
    • 如果 key 在 self.cache 中:
        • node = self.cache[key]
        • self._move_to_tail(node)
        • 返回 node.val
    • 否则返回 -1。
• put(self, key, value) :
    • 如果 key 在 self.cache 中:
        • node = self.cache[key]
        • node.val = value
        • self._move_to_tail(node)
    • 否则:
        • new_node = Node(key, value)
        • 如果 len(self.cache) == self.capacity:
            • lru_node = self.head.next
            • del self.cache[lru_node.key]
            • self._remove_node(lru_node)
        • self._add_node(new_node)
        • self.cache[key] = new_node

```

**时间复杂度:** O(1) - 所有操作都是常数时间。

**空间复杂度:** O(capacity) - 哈希表和双向链表存储的节点数量。

## 代码实现

### Python

```

class Node:
    """
    双向链表节点
    """

    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    """
    LRU 缓存实现，使用哈希表和双向链表。
    """

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {} # 哈希表: key -> Node

        # 双向链表: head.next 是最近最少使用的, tail.prev 是最近使用的
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_node(self, node: Node):
        """
        将节点添加到链表尾部（最近使用）。
        """

```

```

"""
node.prev = self.tail.prev
node.next = self.tail
self.tail.prev.next = node
self.tail.prev = node

def _remove_node(self, node: Node):
"""
从链表中移除节点。
"""
node.prev.next = node.next
node.next.prev = node.prev

def _move_to_tail(self, node: Node):
"""
将节点移动到链表尾部（更新为最近使用）。
"""
self._remove_node(node)
self._add_node(node)

def get(self, key: int) -> int:
if key in self.cache:
    node = self.cache[key]
    self._move_to_tail(node) # 访问后，更新为最近使用
    return node.value
else:
    return -1

def put(self, key: int, value: int) -> None:
if key in self.cache:
    node = self.cache[key]
    node.value = value # 更新值
    self._move_to_tail(node) # 更新为最近使用
else:
    new_node = Node(key, value)
    self.cache[key] = new_node
    self._add_node(new_node)

# 如果容量已满，淘汰最近最少使用的
if len(self.cache) > self.capacity:
    lru_node = self.head.next # 最近最少使用的节点
    self._remove_node(lru_node)
    del self.cache[lru_node.key]

```

## 关键点总结

1. 哈希表 + 双向链表：这是实现 LRU 缓存的经典组合。哈希表提供 O(1) 的查找，双向链表提供 O(1) 的插入和删除，以及维护访问顺序。
2. 伪头尾节点：简化了链表操作的边界条件处理，避免了对空链表或只有一个节点时的特殊判断。
3. `_add_node`, `_remove_node`, `_move_to_tail` 辅助函数：将链表操作封装起来，使 `get` 和 `put` 方法的逻辑更清晰。
4. O(1) 平均时间复杂度：通过哈希表和双向链表的特性，所有核心操作都能在常数时间内完成。

---

## 链表 / 148. 排序链表

### 148. 排序链表

#### 题目描述

给你链表的头结点 `head`，请将其按 升序 排列并返回排序后的链表。

#### 示例

示例 1:

输入: head = [4,2,1,3]

输出: [1,2,3,4]

示例 2:

输入: head = [-1,5,3,4,0]

输出: [-1,0,3,4,5]

示例 3:

输入: head = []

输出: []

## 约束条件

- 链表中节点的数目在范围  $[0, 5 * 10^4]$  内
- $-10^5 \leq \text{Node.val} \leq 10^5$

## 思考过程

### 💡 第一步：理解问题

- 我们需要对一个单链表进行排序。
- 排序后返回新的头节点。
- 链表是单向的，不能随机访问。

### 💡 第二步：朴素解法（转换为数组）

最简单的方法是，将链表中的所有节点值取出，存入一个数组。对数组进行排序，然后根据排序后的数组重新构建一个新的链表。

思考题：

- 这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

### 💡 第三步：链表排序算法的选择

对于链表这种数据结构，由于其无法随机访问的特性，一些基于数组的排序算法（如快速排序、堆排序）效率会很低。

比较适合链表的排序算法是：

#### 1. 归并排序 (Merge Sort):

- 递归地将链表分成两半，分别排序。
- 然后将两个排序好的子链表合并。
- 归并排序的时间复杂度是  $O(n \log n)$ ，空间复杂度是  $O(\log n)$  (递归栈空间) 或  $O(1)$  (迭代实现)。

#### 2. 插入排序 (Insertion Sort):

- 逐个将节点插入到已排序的链表中。
- 时间复杂度最坏  $O(n^2)$ ，最好  $O(n)$ 。不适合大规模数据。

考虑到时间复杂度要求，**归并排序**是最佳选择。

### 💡 第四步：归并排序（自顶向下）

核心思路：

- 分割**：使用快慢指针将链表从中间分成两半。
- 递归**：对两半链表分别进行排序。
- 合并**：将两个排序好的子链表合并成一个。

辅助函数 `mergeTwoLists(l1, l2)`：合并两个有序链表（与 21 题相同）。

### 💡 第五步：算法步骤总结

#### 1. `sortList(head)` 函数：

- 基本情况**：如果 `head` 为 `None` 或 `head.next` 为 `None`，说明链表为空或只有一个节点，已经有序，直接返回 `head`。

- 分割链表**：

- 使用快慢指针 `slow`, `fast`。`slow` 每次走一步，`fast` 每次走两步。
- 同时，需要一个 `prev` 指针来记录 `slow` 的前一个节点，以便在分割时断开链表。
- 当 `fast` 到达末尾时，`slow` 就在中间。将 `prev.next` 置为 `None`，断开链表。
- 得到两个子链表：`head` 到 `prev`，以及 `slow` 到末尾。

- 递归排序**：

- `left_sorted = sortList(head)`
- `right_sorted = sortList(slow)`
- d. 合并:
  - `merged_head = mergeTwoLists(left_sorted, right_sorted)`
  - 返回 `merged_head`。

时间复杂度:  $O(n \log n)$  - 每次分割  $O(n)$ ,  $\log n$  层递归, 每层合并  $O(n)$ 。

空间复杂度:  $O(\log n)$  - 递归栈的深度。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def sortList(head: ListNode) -> ListNode:
    """
    使用自顶向下归并排序法排序链表。
    """

    if not head or not head.next:
        return head

    # 1. 分割链表
    slow = head
    fast = head
    prev = None # 用于断开链表

    while fast and fast.next:
        prev = slow
        slow = slow.next
        fast = fast.next.next

    # 断开链表, 得到两部分
    if prev: # 确保不是只有一个节点的情况
        prev.next = None

    # 2. 递归排序两部分
    left_sorted = sortList(head)
    right_sorted = sortList(slow)

    # 3. 合并两部分
    return mergeTwoLists(left_sorted, right_sorted)

# 辅助函数: 合并两个有序链表 (与 21 题相同)
def mergeTwoLists(l1: ListNode, l2: ListNode) -> ListNode:
    dummy_head = ListNode(0)
    current = dummy_head

    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next
```

```

if 11:
    current.next = 11
elif 12:
    current.next = 12

return dummy_head.next

```

## 关键点总结

1. **归并排序**: 是链表排序的经典算法，因为它不依赖于随机访问，且时间复杂度为  $O(n \log n)$ 。
2. **快慢指针分割**: 高效地找到链表的中间点，实现链表的二分。
3. **断开链表**: 在递归调用之前，必须将链表从中间断开，否则会形成环或导致无限递归。
4. **合并辅助函数**: 利用合并两个有序链表的子问题来解决归并排序的最后一步。

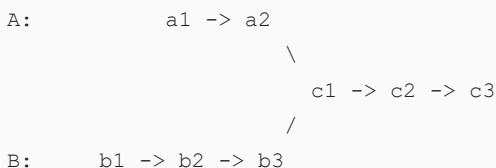
## 链表 / 160. 相交链表

### 160. 相交链表

#### 题目描述

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。如果两个链表不存在任何交点，返回 `null`。

图示两个链表在节点 `c1` 开始相交：



题目数据 保证 整个链式结构中不存在环。

#### 注意：

- 如果两个链表没有交点，返回 `null`。
- 函数返回结果后，链表必须保持原样。
- 可假定整个链表结构中没有循环。
- 程序尽量满足  $O(m + n)$  时间复杂度，且仅用  $O(1)$  内存。

#### 示例

##### 示例 1：

输入: `intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], 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它们背后必然是同一条链表）。

从各自的表头开始算起，链表 A 为 `[4,1,8,4,5]`，链表 B 为 `[5,6,1,8,4,5]`。

在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。

##### 示例 2：

输入: `intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1`  
输出: `Intersected at '2'`

解释：相交节点的值为 2（注意，如果两个链表相交则它们背后必然是同一条链表）。

从各自的表头开始算起，链表 A 为 `[1,9,1,2,4]`，链表 B 为 `[3,2,4]`。

在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。

##### 示例 3：

输入: `intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2`  
输出: `No intersection`

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，所以返回 `null`。

## 约束条件

- `listA` 中节点数目为 `m`
- `listB` 中节点数目为 `n`
- $1 \leq m, n \leq 3 * 10^4$
- $0 \leq \text{Node.val} \leq 10^5$
- `skipA` 和 `skipB` 保证是有效的。

## 思考过程

### 💡 第一步：理解问题

- 我们要找到两个单链表的第一个公共节点。
- 如果没有公共节点，返回 `None`。
- 链表没有环。
- 要求时间  $O(m+n)$ ，空间  $O(1)$ 。

### 💡 第二步：朴素解法（哈希表）

最直观的想法是，遍历其中一个链表，将其所有节点存入一个哈希集合。然后遍历另一个链表，检查每个节点是否在哈希集合中。

思考题：

- 这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

### 💡 第三步：双指针法（ $O(1)$ 空间的关键）

既然不能用额外空间，我们只能利用链表本身的结构。两个链表可能长度不同，但如果它们相交，那么从相交点到链表末尾的这部分是公共的。

核心思路：

- 设链表 A 的长度为 `L_A`，链表 B 的长度为 `L_B`。
- 设相交点前的非公共部分长度分别为 `a` 和 `b`。
- 设公共部分的长度为 `c`。
- 那么  $L_A = a + c$ ,  $L_B = b + c$ 。

我们让两个指针 `ptrA` 和 `ptrB` 分别从 `headA` 和 `headB` 开始遍历。

思考题：

- 如果 `ptrA` 走到了链表 A 的末尾，它应该去哪里？
- 如果 `ptrB` 走到了链表 B 的末尾，它应该去哪里？

▶ 点击查看分析

### 💡 第四步：算法步骤总结

1. 初始化两个指针 `ptrA = headA, ptrB = headB`。
2. 当 `ptrA != ptrB` 时，循环执行：
  - a. 如果 `ptrA` 为 `None`，则 `ptrA = headB`；否则 `ptrA = ptrA.next`。
  - b. 如果 `ptrB` 为 `None`，则 `ptrB = headA`；否则 `ptrB = ptrB.next`。
3. 循环结束后，`ptrA` (或 `ptrB`) 就是相交节点，或者 `None` (如果不相交)。

时间复杂度： $O(m+n)$  - 两个指针最多走  $m + n$  步。

空间复杂度： $O(1)$  - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def getIntersectionNode(headA: ListNode, headB: ListNode) -> ListNode:
    """
    使用双指针法寻找两个链表的相交节点。
    """
    if not headA or not headB:
```

```

    return None

ptrA = headA
ptrB = headB

# 当 ptrA 和 ptrB 不相等时循环
# 如果相交，它们会在交点相遇
# 如果不相交，它们会同时走到 None
while ptrA != ptrB:
    # 如果 ptrA 走到链表 A 的末尾，则从链表 B 的头部开始
    ptrA = ptrA.next if ptrA else headB

    # 如果 ptrB 走到链表 B 的末尾，则从链表 A 的头部开始
    ptrB = ptrB.next if ptrB else headA

return ptrA # 此时 ptrA (或 ptrB) 就是相交节点或 None

```

## 关键点总结

1. 双指针交替遍历：让两个指针分别从两个链表的头开始，当一个指针到达链表末尾时，切换到另一个链表的头部继续遍历。这种方式巧妙地抵消了两个链表长度的差异。
2. 等长路径：无论链表是否相交，两个指针最终都会走过相同长度的路径。如果相交，它们会在交点相遇；如果不相交，它们会同时到达 `None`。
3.  $O(1)$  空间：通过这种双指针策略，我们避免了使用额外的存储空间。

## 链表 / 19. 删除链表的倒数第 N 个结点

### 19. 删除链表的倒数第 N 个结点

#### 题目描述

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

示例 1：

输入：`head = [1,2,3,4,5], n = 2`  
输出：`[1,2,3,5]`

示例 2：

输入：`head = [1], n = 1`  
输出：`[]`

示例 3：

输入：`head = [1,2], n = 1`  
输出：`[1]`

#### 约束条件

- 链表中结点的数目范围是 `[1, 30]`
- $1 \leq n \leq$  链表中结点总数

#### 思考过程

##### 💡 第一步：理解问题

- 我们需要删除链表的倒数第  $n$  个节点。
- 返回删除后的链表头节点。

##### 💡 第二步：朴素解法（两次遍历）

最直观的想法是，先遍历一遍链表，计算出链表的总长度  $L$ 。然后，倒数第  $n$  个节点就是正数第  $L - n + 1$  个节点。再遍历一遍链表，找到这个节点的前一个节点，然后进行删除操作。

#### 思考题：

- 如何处理删除头节点的情况？

- 这个方法的时间和空间复杂度是多少？

► 点击查看分析

### 💡 第三步：一次遍历（快慢指针）

为了只遍历一次链表，我们需要一种方法，在遍历到链表末尾时，同时知道倒数第  $n$  个节点的位置。

**核心思路：**

- 使用两个指针 `fast` 和 `slow`。
- 先让 `fast` 指针向前走  $n$  步。
- 然后 `fast` 和 `slow` 同时向前走，直到 `fast` 到达链表末尾。
- 此时，`slow` 指针正好在倒数第  $n$  个节点的前一个节点。

**思考题：**

- 为什么 `slow` 会在倒数第  $n$  个节点的前一个节点？
- 如何处理删除头节点的情况？

► 点击查看分析

### 💡 第四步：算法步骤总结

1. 创建一个哑节点 `dummy_head = ListNode(0)`，并让 `dummy_head.next = head`。
2. 初始化 `fast = dummy_head`, `slow = dummy_head`。
3. 让 `fast` 指针先向前走  $n$  步。
  - 循环  $n$  次，每次 `fast = fast.next`。
4. 然后 `fast` 和 `slow` 同时向前走，直到 `fast` 到达链表末尾（即 `fast.next` 为 `None`）。
  - 循环条件：`while fast.next:`
  - 每次 `fast = fast.next`, `slow = slow.next`。
5. 此时，`slow` 指针指向的节点就是待删除节点的前一个节点。
6. 执行删除操作：`slow.next = slow.next.next`。
7. 返回 `dummy_head.next`。

**时间复杂度：**  $O(L)$  - 一次遍历链表。

**空间复杂度：**  $O(1)$  - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def removeNthFromEnd(head: ListNode, n: int) -> ListNode:
    """
    使用快慢指针一次遍历删除链表的倒数第 N 个结点。
    """

    # 创建一个哑节点，简化对头节点的处理
    dummy_head = ListNode(0)
    dummy_head.next = head

    fast = dummy_head
    slow = dummy_head

    # 1. fast 指针先向前走 n 步
    for _ in range(n):
        fast = fast.next

    # 2. fast 和 slow 同时向前走，直到 fast 到达链表末尾
    # 此时 slow 位于待删除节点的前一个节点
    while fast.next:
        fast = fast.next
        slow = slow.next

    # 删除操作
    slow.next = slow.next.next

    return dummy_head.next
```

```

        slow = slow.next

    # 3. 删除节点
    slow.next = slow.next.next

    return dummy_head.next # 返回新的头节点

```

## 关键点总结

- 快慢指针：通过保持 `fast` 和 `slow` 之间  $n$  个节点的距离，当 `fast` 到达末尾时，`slow` 自然就定位到了目标节点的前一个位置。
- 哑节点：为了统一处理删除头节点的情况，使用哑节点是一个非常方便的技巧。
- 一次遍历：相比于两次遍历，快慢指针法只需要一次遍历即可完成任务，提高了效率。

## 链表 / 2. 两数相加

### 2. 两数相加

#### 题目描述

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 **逆序** 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

示例 1：

输入: `l1 = [2,4,3], l2 = [5,6,4]`  
 输出: `[7,0,8]`  
 解释:  $342 + 465 = 807.$

示例 2：

输入: `l1 = [0], l2 = [0]`  
 输出: `[0]`

示例 3：

输入: `l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]`  
 输出: `[8,9,9,9,0,0,0,1]`

#### 约束条件

- 每个链表中的节点数在范围 `[1, 100]` 内
- `0 <= Node.val <= 9`
- 题目数据保证列表表示的数字不含前导零，除非该数字为 0 本身。

#### 思考过程

##### 💡 第一步：理解问题

- 我们有两个链表，每个链表代表一个非负整数。
- 数字是**逆序**存储的，即个位在链表头，十位在第二个节点，以此类推。
- 我们需要将这两个数相加，并以同样逆序的链表形式返回结果。
- 每个节点只存储一位数字。

##### 💡 第二步：模拟加法过程

这就像我们小学学习的竖式加法。

#### 思考题：

- 竖式加法需要考虑哪些因素？
- 链表如何模拟这个过程？

▶ 点击查看分析

##### 💡 第三步：算法步骤总结

1. 创建一个哑节点 `dummy_head = ListNode(0)`，用于构建结果链表。
2. 创建一个 `current` 指针，指向 `dummy_head`。
3. 初始化 `carry = 0`，用于存储进位。
4. 使用 `p1 = l1, p2 = l2` 分别遍历两个输入链表。
5. 当 `p1` 或 `p2` 不为 `None`，或者 `carry` 不为 `0` 时，循环执行：
  - a. 获取当前位的数字：`vall = p1.val` (如果 `p1` 为 `None` 则为 0)，`val2 = p2.val` (如果 `p2` 为 `None` 则为 0)。
  - b. 计算当前位的和：`sum_val = vall + val2 + carry`。
  - c. 计算当前位的新节点值：`digit = sum_val % 10`。
  - d. 计算新的进位：`carry = sum_val // 10`。
  - e. 创建新节点并连接：`current.next = ListNode(digit)`。
  - f. 移动 `current` 指针：`current = current.next`。
  - g. 移动 `p1` 和 `p2` 指针 (如果它们不为 `None`)。
6. 循环结束后，返回 `dummy_head.next`。

**时间复杂度：**  $O(\max(m, n))$  - 遍历两个链表中最长的一个。

**空间复杂度：**  $O(\max(m, n))$  - 新链表的长度。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    """
    模拟竖式加法，将两个逆序存储的链表表示的数字相加。
    """

    dummy_head = ListNode(0) # 哑节点，用于构建结果链表
    current = dummy_head      # current 指针用于移动和连接新节点
    carry = 0                  # 进位

    # 循环条件：只要有一个链表还有节点，或者还有进位，就继续循环
    while l1 or l2 or carry:
        # 获取当前位的数字，如果链表已遍历完，则为 0
        vall = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # 计算当前位的和，包括进位
        sum_val = vall + val2 + carry

        # 计算当前位的新节点值和新的进位
        digit = sum_val % 10
        carry = sum_val // 10

        # 创建新节点并连接到结果链表
        current.next = ListNode(digit)
        current = current.next

        # 移动 l1 和 l2 指针
        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    return dummy_head.next # 返回结果链表的头节点
```

## 关键点总结

1. 模拟竖式加法：从低位（链表头）开始相加，并处理进位。
2. 哑节点：简化了结果链表头部的处理，避免了对第一个节点的特殊判断。
3. 循环条件：`while l1 or l2 or carry` 确保所有位都被处理，包括最后可能存在的进位。
4. 处理不同长度链表：通过将已遍历完的链表的值视为 0，优雅地处理了两个链表长度不同的情况。

## 链表 / 206. 反转链表

### 206. 反转链表

#### 题目描述

给你单链表的头节点 `head`，请你反转链表，并返回反转后的链表头节点。

#### 示例

示例 1：

输入：`head = [1,2,3,4,5]`

输出：`[5,4,3,2,1]`

示例 2：

输入：`head = [1,2]`

输出：`[2,1]`

示例 3：

输入：`head = []`

输出：`[]`

#### 约束条件

- 链表中节点的数目范围是 `[0, 5000]`
- $-5000 \leq \text{Node.val} \leq 5000$

#### 思考过程

##### 💡 第一步：理解问题

- 我们需要将一个单链表进行反转。
- 反转意味着每个节点的 `next` 指针将指向它的前一个节点。
- 最终返回反转后的链表的头节点。

##### 💡 第二步：如何反转一个节点？

考虑一个节点 `curr`。它的 `next` 指针原本指向 `curr.next`。反转后，它应该指向它的前一个节点 `prev`。

```
curr.next = prev
```

但是，当我们修改了 `curr.next` 后，我们就失去了对 `curr` 原本的下一个节点的引用。这会是一个问题。

##### 💡 第三步：迭代法

为了解决失去下一个节点引用的问题，我们需要在修改 `curr.next` 之前，先保存 `curr` 的下一个节点。

#### 核心思路：

- 使用三个指针：`prev`, `curr`, `next_temp`。
- `prev`：指向当前节点 `curr` 的前一个节点（初始为 `None`）。
- `curr`：指向当前正在处理的节点（初始为 `head`）。
- `next_temp`：临时保存 `curr` 的下一个节点，以便在修改 `curr.next` 后还能访问到它。

#### 算法流程：

1. 初始化 `prev = None`, `curr = head`。
2. 当 `curr` 不为 `None` 时，循环执行：
  - 保存 `curr` 的下一个节点：`next_temp = curr.next`。
  - 反转 `curr` 的 `next` 指针：`curr.next = prev`。
  - 移动 `prev` 指针：`prev = curr`。
  - 移动 `curr` 指针：`curr = next_temp`。
3. 循环结束后，`prev` 将指向原链表的最后一个节点，也就是反转后链表的头节点。返回 `prev`。

时间复杂度: O(n) - 遍历一次链表。

空间复杂度: O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseList(head: ListNode) -> ListNode:
    """
    使用迭代法反转链表。
    """

    prev = None # 指向当前节点的前一个节点
    curr = head # 指向当前正在处理的节点

    while curr:
        # 1. 临时保存当前节点的下一个节点，防止断链
        next_temp = curr.next

        # 2. 反转当前节点的 next 指针，使其指向前一个节点
        curr.next = prev

        # 3. 移动 prev 指针到当前节点，为下一个节点的反转做准备
        prev = curr

        # 4. 移动 curr 指针到下一个节点
        curr = next_temp

    return prev # 循环结束后，prev 就是反转后链表的头节点
```

## 关键点总结

- 三指针: `prev, curr, next_temp` 是迭代法反转链表的关键。`next_temp` 用于保存 `curr` 的下一个节点，防止在修改 `curr.next` 后丢失对后续节点的引用。
- 指针更新顺序: 严格按照“保存下一个 -> 反转当前 -> 移动 `prev` -> 移动 `curr`”的顺序进行指针更新，是确保算法正确性的关键。
- O(1) 空间: 迭代法通过修改现有节点的指针，实现了原地反转，无需额外空间。

## 链表 / 21. 合并两个有序链表

### 21. 合并两个有序链表

#### 题目描述

将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

#### 示例

示例 1:

输入: `l1 = [1,2,4], l2 = [1,3,4]`  
输出: `[1,1,2,3,4,4]`

示例 2:

输入: `l1 = [], l2 = []`  
输出: `[]`

示例 3:

```
输入: l1 = [], l2 = [0]
输出: [0]
```

## 约束条件

- 两个链表的节点数目范围是 `[0, 50]`
- `-100 <= Node.val <= 100`
- `l1` 和 `l2` 均按 非递减顺序 排列

## 思考过程

### 💡 第一步：理解问题

- 我们有两个已经排好序的链表。
- 需要将它们合并成一个新的、同样排好序的链表。
- 新链表包含所有节点。

### 💡 第二步：如何合并两个有序数组？

这个问题和合并两个有序数组非常相似。我们可以使用双指针的方法。

思考题：

- 如果是数组，你会怎么做？

▶ 点击查看分析

### 💡 第三步：链表的合并

链表的合并思路与数组类似，但操作的是节点和指针。

核心思路：

- 创建一个哑节点（dummy node）作为新链表的头，这样可以简化代码，避免对头节点的特殊处理。
- 使用一个 `current` 指针来构建新链表，它总是指向新链表的最后一个节点。
- 使用两个指针 `p1` 和 `p2` 分别遍历 `l1` 和 `l2`。

### 💡 第四步：算法步骤总结

- 创建一个哑节点 `dummy_head = ListNode(0)`。
- 创建一个 `current` 指针，指向 `dummy_head`。
- 使用 `p1 = l1, p2 = l2` 分别遍历两个链表。
- 当 `p1` 和 `p2` 都不为 `None` 时，循环执行：
  - 比较 `p1.val` 和 `p2.val`。
  - 如果 `p1.val <= p2.val`，将 `p1` 连接到 `current.next`，然后 `p1 = p1.next`。
  - 否则，将 `p2` 连接到 `current.next`，然后 `p2 = p2.next`。
  - 移动 `current` 指针：`current = current.next`。
- 循环结束后，如果 `p1` 还有剩余节点，将 `p1` 的剩余部分连接到 `current.next`。
- 如果 `p2` 还有剩余节点，将 `p2` 的剩余部分连接到 `current.next`。
- 返回 `dummy_head.next`。

时间复杂度：O(m + n) - 遍历两个链表一次。

空间复杂度：O(1) - 只使用了常数个额外变量（不考虑新链表本身的存储空间）。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(l1: ListNode, l2: ListNode) -> ListNode:
    """
    使用迭代法合并两个有序链表。
    """
    # 创建一个哑节点，简化头节点的处理
    dummy_head = ListNode(0)
```

```

current = dummy_head # current 指针用于构建新链表

# 遍历两个链表，直到其中一个遍历完
while l1 and l2:
    if l1.val <= l2.val:
        current.next = l1
        l1 = l1.next
    else:
        current.next = l2
        l2 = l2.next
    current = current.next # 移动 current 指针

# 将剩余的链表部分直接连接到新链表末尾
if l1:
    current.next = l1
elif l2:
    current.next = l2

return dummy_head.next # 返回新链表的头节点

```

## 关键点总结

1. **哑节点**: 使用哑节点 `dummy_head` 是处理链表头节点操作的常用技巧，可以避免对空链表或第一个节点的特殊判断。
2. **双指针**: `l1` 和 `l2` 用于遍历两个输入链表，`current` 用于构建结果链表。
3. **迭代合并**: 通过比较两个链表当前节点的值，将较小的节点连接到结果链表，并移动相应指针，直到一个链表为空。
4. **处理剩余部分**: 当一个链表遍历完后，另一个链表可能还有剩余节点，直接将剩余部分连接到结果链表末尾即可。

## 链表 / 23. 合并 K 个升序链表

### 23. 合并 K 个升序链表

#### 题目描述

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，并返回合并后的链表。

#### 示例

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`

示例 2:

输入: `lists = []`

输出: `[]`

示例 3:

输入: `lists = [[]]`

输出: `[]`

#### 约束条件

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `0 <= sum(lists[i].length) <= 10^4`
- `-10^4 <= Node.val <= 10^4`
- `lists[i]` 按升序排列

## 思考过程

### 💡 第一步：理解问题

- 我们有 `k` 个已经排好序的链表。
- 需要将它们全部合并成一个大的、同样排好序的链表。
- 返回合并后的链表的头节点。

### 💡 第二步：朴素解法（逐个合并）

最直观的想法是，我们可以重复使用“合并两个有序链表”的函数（21题）。

思考题：

- 如果我们每次合并两个链表，总共需要合并多少次？
- 这个方法的时间复杂度是多少？

▶ 点击查看分析

### 💡 第三步：优化思路 - 分治法

逐个合并效率太低，因为每次合并的链表长度都在增加。我们可以借鉴归并排序的思想，使用分治法。

核心思路：

- 将 `k` 个链表分成两半，递归地合并左半部分和右半部分。
- 然后将合并后的两个大链表再合并起来。

思考题：

- 这种分治合并的时间复杂度是多少？

▶ 点击查看分析

### 💡 第四步：最优解 - 最小堆（优先队列）

分治法已经很高效了，但还有一种更通用的方法，尤其适用于 `k` 很大的情况：使用最小堆（优先队列）。

核心思路：

- 最小堆可以帮助我们高效地找到所有链表当前头节点中的最小值。
- 我们将所有链表的头节点放入最小堆中。
- 每次从堆中取出最小的节点，将其添加到结果链表。
- 如果取出的节点还有下一个节点，就将下一个节点放入堆中。

思考题：

- 堆中存储什么？
- 堆操作的时间复杂度是多少？

▶ 点击查看分析

### 💡 第五步：算法步骤总结（最小堆）

1. 导入 `heapq` 模块。
2. 创建一个哑节点 `dummy_head = ListNode(0)`，用于构建结果链表。
3. 创建一个最小堆 `min_heap`。
4. 将所有非空的链表的头节点放入堆中。堆中存储 `(node.val, index, node)`。`index` 是为了处理节点值相同的情况，确保元组可比较。
5. 当堆不为空时，循环执行：
  - a. 从堆中取出最小的节点 `(val, index, node)`。
  - b. 将 `node` 连接到结果链表：`current.next = node`，然后 `current = current.next`。
  - c. 如果 `node.next` 不为 `None`，则将 `(node.next.val, new_index, node.next)` 放入堆中 (`new_index` 递增)。
6. 返回 `dummy_head.next`。

时间复杂度：O(TotalN \* log k) - TotalN 是所有链表中的总节点数。

空间复杂度：O(k) - 堆中最多存储 `k` 个节点。

## 代码实现

### Python

```

import heapq

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeKLists(lists: list[ListNode]) -> ListNode:
    """
    使用最小堆（优先队列）合并 k 个升序链表。
    """
    dummy_head = ListNode(0) # 哑节点
    current = dummy_head # 用于构建结果链表
    min_heap = [] # 最小堆

    # 为了处理节点值相同的情况，需要一个计数器来保证元组的唯一性
    # Python 的 heapq 在比较元组时，如果第一个元素相同，会比较第二个，以此类推
    # 如果节点本身不可比较，就会报错。所以加入一个递增的 index
    entry_index = 0

    # 1. 将所有链表的头节点放入最小堆
    for head in lists:
        if head:
            heapq.heappush(min_heap, (head.val, entry_index, head))
            entry_index += 1

    # 2. 从堆中取出最小节点，并将其下一个节点放入堆中
    while min_heap:
        val, _, node = heapq.heappop(min_heap) # 取出最小节点

        current.next = node # 连接到结果链表
        current = current.next

        if node.next: # 如果当前节点还有下一个节点，则将其放入堆中
            heapq.heappush(min_heap, (node.next.val, entry_index, node.next))
            entry_index += 1

    return dummy_head.next

```

## 关键点总结

1. **最小堆**：是解决“从多个有序源中选择最小元素”问题的最佳数据结构。它能高效地在  $k$  个链表的当前头节点中找到最小值。
2. **哑节点**：简化了结果链表头部的处理。
3. **堆中存储内容**：存储 (节点值, 唯一标识, 节点) 的元组，确保堆能够正确比较和处理节点。
4. **时间复杂度**： $O(\text{TotalN} * \log k)$ ，其中  $\text{TotalN}$  是所有链表的总节点数， $k$  是链表的数量。这是非常高效的解决方案。

## 链表 / 234. 回文链表

### 234. 回文链表

#### 题目描述

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

示例 1：

输入：`head = [1,2,2,1]`

输出: true

示例 2:

输入: head = [1, 2]

输出: false

## 约束条件

- 链表中节点数目在范围  $[1, 10^5]$  内
- $0 \leq \text{Node.val} \leq 9$

## 思考过程

### 💡 第一步：理解问题

- 我们需要判断一个单链表是否是回文的。
- 回文意味着从前往后读和从后往前读是相同的。
- 链表是单向的，这意味着我们不能直接从后往前遍历。

### 💡 第二步：朴素解法（转换为数组）

最直观的想法是，将链表的所有节点值存储到一个列表中（或数组中），然后判断这个列表是否是回文的。

思考题：

- 这个方法的时间和空间复杂度是多少？

▶ 点击查看分析

### 💡 第三步：O(1) 空间的关键 > 快慢指针 + 反转链表

为了实现 O(1) 空间，我们不能将整个链表存储起来。我们需要一种方法，在不改变链表结构太多的情况下，同时访问链表的前半部分和后半部分。

核心思路：

- 找到链表的中间节点：使用快慢指针。快指针每次走两步，慢指针每次走一步。当快指针到达链表末尾时，慢指针正好在中间。
- 反转链表的后半部分：从中间节点开始，将链表的后半部分进行反转。
- 比较前半部分和反转后的后半部分：从链表头部开始遍历前半部分，同时从反转后的后半部分头部开始遍历，逐一比较节点值。
- 恢复链表（可选）：为了不改变原链表结构，可以在比较完成后，将反转的后半部分再反转回来，并重新连接。

### 💡 第四步：算法步骤总结

- 处理特殊情况：如果链表为空或只有一个节点，它一定是回文的，返回 True。
- 找到中间节点：
  - 使用快慢指针 slow 和 fast，都从 head 开始。
  - fast 每次走两步，slow 每次走一步。
  - 当 fast 到达链表末尾（fast is None 或 fast.next is None）时，slow 就在中间节点。
- 反转后半部分：
  - 从 slow 指针指向的节点开始，反转链表的后半部分。这会得到一个新的头节点 second\_half\_head。
- 比较：
  - 使用两个指针 p1 = head 和 p2 = second\_half\_head。
  - 逐一比较 p1.val 和 p2.val。如果发现不相等，则不是回文，返回 False。
  - 遍历直到 p2 到达末尾。
- 恢复链表（可选，但通常是好习惯）：
  - 将反转的后半部分再反转回来，并重新连接到前半部分。
- 如果所有比较都通过，返回 True。

时间复杂度：O(n) - 找到中间 O(n)，反转 O(n)，比较 O(n)。

空间复杂度：O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

```

def isPalindrome(head: ListNode) -> bool:
    """
    使用快慢指针找到中点，反转后半部分，然后比较。
    """

    if not head or not head.next:
        return True

    # 1. 找到链表的中间节点
    slow = head
    fast = head
    while fast.next and fast.next.next:
        slow = slow.next
        fast = fast.next.next

    # 此时 slow 指向前半部分的最后一个节点
    # 如果链表长度为偶数，slow 指向前半部分的最后一个节点
    # 如果链表长度为奇数，slow 指向中间节点

    # 2. 反转链表的后半部分
    # 后半部分的起始节点是 slow.next
    second_half_head = reverseList(slow.next)

    # 断开前半部分和后半部分，或者说，将前半部分的末尾指向 None
    # 这一步不是必须的，但可以帮助理解
    slow.next = None

    # 3. 比较前半部分和反转后的后半部分
    p1 = head
    p2 = second_half_head
    is_palindrome = True
    while p1 and p2:
        if p1.val != p2.val:
            is_palindrome = False
            break
        p1 = p1.next
        p2 = p2.next

    # 4. 恢复链表（可选，但通常是好习惯）
    # 再次反转 second_half_head，并重新连接
    # slow.next = reverseList(second_half_head) # 这一步需要根据实际情况调整

    return is_palindrome

# 辅助函数：反转链表（与 206 题相同）
def reverseList(head: ListNode) -> ListNode:
    prev = None
    curr = head
    while curr:
        next_temp = curr.next
        curr.next = prev
        prev = curr
        curr = next_temp
    return prev

```

## 关键点总结

1. 快慢指针：高效地找到链表的中间节点，将链表一分为二。
2. 原地反转：利用反转链表的技巧，将后半部分链表原地反转，避免了 O(n) 的额外空间。

3. 分段比较：将回文判断转化为前半部分和反转后半部分的逐一比较。
4. 恢复链表：虽然不是判断回文的必要步骤，但在实际应用中，为了不改变原链表结构，通常会进行恢复操作。

## 链表 / 24. 两两交换链表中的节点

### 24. 两两交换链表中的节点

#### 题目描述

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成。

#### 示例

示例 1：

输入：head = [1,2,3,4]  
输出：[2,1,4,3]

示例 2：

输入：head = []  
输出：[]

示例 3：

输入：head = [1]  
输出：[1]

#### 约束条件

- 链表中节点的数目在范围 [0, 100] 内
- 0 <= Node.val <= 100

#### 思考过程

##### 💡 第一步：理解问题

- 我们需要将链表中相邻的两个节点进行交换。
- 交换的是节点本身，而不是节点的值。
- 返回交换后的链表头节点。

##### 💡 第二步：如何交换两个节点？

考虑两个相邻节点 node1 和 node2，其中 node1.next = node2。

交换后，我们希望 node2 在 node1 前面，并且 node2.next = node1。

思考题：

- 交换 node1 和 node2 后，它们的前一个节点和后一个节点如何连接？

▶ 点击查看分析

##### 💡 第三步：迭代法

我们可以使用迭代的方式，每次处理一对节点。

核心思路：

- 使用一个哑节点 dummy\_head 来简化对头节点的处理。
- 使用一个 prev 指针，它总是指向当前要交换的两个节点的前一个节点。

##### 💡 第四步：算法步骤总结

- 创建一个哑节点 dummy\_head = ListNode(0)，并让 dummy\_head.next = head。
- 初始化 prev = dummy\_head。
- 当 prev.next 和 prev.next.next 都不为 None 时，循环执行（确保至少有两节点可以交换）：
  - 定义 node1 = prev.next (第一个节点)
  - 定义 node2 = prev.next.next (第二个节点)
  - 执行交换：
    - prev.next = node2 (将 prev 连接到 node2)
    - node1.next = node2.next (将 node1 连接到下一对的开头)
    - node2.next = node1 (将 node2 连接到 node1)
  - 移动 prev 指针到下一对的前一个位置：prev = node1。

4. 返回 `dummy_head.next`。

时间复杂度:  $O(n)$  - 遍历链表一次。

空间复杂度:  $O(1)$  - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def swapPairs(head: ListNode) -> ListNode:
    """
    使用迭代法两两交换链表中的相邻节点。
    """
    dummy_head = ListNode(0) # 哑节点, 简化头节点处理
    dummy_head.next = head

    prev = dummy_head # prev 指向当前要交换的两个节点的前一个节点

    # 确保至少有两个节点可以交换
    while prev.next and prev.next.next:
        node1 = prev.next # 第一个节点
        node2 = prev.next.next # 第二个节点

        # 执行交换操作
        prev.next = node2 # prev 连接到 node2
        node1.next = node2.next # node1 连接到下一对的开头
        node2.next = node1 # node2 连接到 node1

        # 移动 prev 指针到下一对的前一个位置
        prev = node1

    return dummy_head.next
```

## 关键点总结

1. **哑节点**: 再次体现了哑节点在链表操作中的重要性, 它使得对头节点的处理与其他节点一致。
2. **三指针**: `prev`, `node1`, `node2` 协同工作, 精确地完成节点的重新连接。
3. **指针更新顺序**: 在交换过程中, 指针的更新顺序至关重要, 需要仔细推敲, 确保不会丢失对后续节点的引用。
4.  **$O(1)$  空间**: 通过修改指针实现了原地交换。

## 链表 / 25. K 个一组翻转链表

### 25. K 个一组翻转链表

#### 题目描述

给你链表的头节点 `head`, 每 `k` 个节点一组进行翻转, 并返回修改后的链表。

`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

示例 1:

输入: head = [1,2,3,4,5], k = 2

输出: [2,1,4,3,5]

示例 2:

输入: head = [1,2,3,4,5], k = 3

输出: [3,2,1,4,5]

示例 3:

输入: head = [1,2,3,4,5], k = 1

输出: [1,2,3,4,5]

示例 4:

输入: head = [1], k = 1

输出: [1]

## 约束条件

- 链表中的节点数目为 n
- $1 \leq k \leq n \leq 5000$
- $0 \leq \text{Node.val} \leq 1000$

## 思考过程

### 第一步：理解问题

- 我们需要将链表每 k 个节点一组进行翻转。
- 如果最后不足 k 个，则保持原样。
- 必须是节点交换，不能只修改值。

### 第二步：如何翻转一组节点？

这和反转链表（206 题）类似，但我们需要处理的是一个子链表，并且要将翻转后的子链表重新连接到原链表中。

思考题：

- 翻转 k 个节点需要哪些信息？
- 翻转后如何连接？

▶ 点击查看分析

### 第三步：迭代法 + 辅助函数

我们可以使用迭代的方式，每次处理 k 个节点。为了方便操作，可以引入一个辅助函数来翻转指定范围的链表。

核心思路：

- 创建一个哑节点 dummy\_head，简化头节点处理。
- 使用 prev\_group\_tail 指针，它总是指向前一组翻转后的尾部（也是当前组的前一个节点）。
- 每次处理一组 k 个节点：
  - 检查是否足够 k 个节点：从 prev\_group\_tail 开始，向后走 k 步，找到当前组的尾部 curr\_group\_tail。如果不 k 个，则停止。
  - 保存下一组的头部：next\_group\_head = curr\_group\_tail.next。
  - 断开连接：将 curr\_group\_tail.next 置为 None，将当前组从链表中暂时断开。
  - 翻转当前组：调用反转链表的辅助函数，翻转从 prev\_group\_tail.next 到 curr\_group\_tail 的这 k 个节点。翻转后，原先的 prev\_group\_tail.next 变成了新组的尾部，curr\_group\_tail 变成了新组的头部。
  - 重新连接：
    - 将 prev\_group\_tail.next 连接到新组的头部（即翻转前的 curr\_group\_tail）。
    - 将新组的尾部（即翻转前的 prev\_group\_tail.next）连接到 next\_group\_head。
  - 更新 prev\_group\_tail 为当前组翻转后的尾部（即翻转前的 prev\_group\_tail.next）。

### 第四步：算法步骤总结

- 创建一个哑节点 dummy\_head = ListNode(0)，并让 dummy\_head.next = head。
- 初始化 prev\_group\_tail = dummy\_head。
- 循环直到链表遍历完毕：
  - 找到当前组的尾部：从 prev\_group\_tail 开始，向后走 k 步，用 curr\_group\_tail 指向它。
  - 检查是否足够 k 个节点：如果 curr\_group\_tail 为 None，说明剩余节点不足 k 个，跳出循环。
  - 保存下一组的头部：next\_group\_head = curr\_group\_tail.next。
  - 断开当前组：curr\_group\_tail.next = None。

e. 翻转当前组：调用 `reverseList(prev_group_tail.next)`，得到翻转后的新头部 `new_group_head`。

f. 重新连接：

i. `prev_group_tail.next` 连接到 `new_group_head`。

ii. 原先的 `prev_group_tail.next` (现在是翻转后的尾部) 连接到 `next_group_head`。

g. 更新 `prev_group_tail` 为当前组翻转后的尾部 (即原先的 `prev_group_tail.next`)。

4. 返回 `dummy_head.next`。

辅助函数 `reverseList(head)`：反转一个链表并返回新的头节点和尾节点 (或者只返回头节点，尾节点就是原头节点)。

时间复杂度：O(n) - 每个节点被访问和修改常数次。

空间复杂度：O(1) - 只使用了常数个额外变量。

## 代码实现

### Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseKGroup(head: ListNode, k: int) -> ListNode:
    """
    每 k 个节点一组翻转链表。
    """
    dummy_head = ListNode(0) # 哑节点
    dummy_head.next = head

    prev_group_tail = dummy_head # 指向前一组翻转后的尾部

    while True:
        # 1. 检查剩余节点是否足够 k 个
        curr_group_tail = prev_group_tail
        for _ in range(k):
            if not curr_group_tail.next:
                return dummy_head.next # 不足 k 个，直接返回
            curr_group_tail = curr_group_tail.next

        # 此时 curr_group_tail 指向当前组的最后一个节点
        curr_group_head = prev_group_tail.next # 当前组的第一个节点
        next_group_head = curr_group_tail.next # 下一组的第一个节点

        # 2. 断开当前组与下一组的连接
        curr_group_tail.next = None

        # 3. 翻转当前组
        # 翻转后，curr_group_head 变为新组的尾部，curr_group_tail 变为新组的头部
        new_group_head = reverseList(curr_group_head)

        # 4. 重新连接
        # 前一组的尾部连接到新组的头部
        prev_group_tail.next = new_group_head
        # 新组的尾部连接到下一组的头部
        curr_group_head.next = next_group_head

        # 5. 更新 prev_group_tail 为当前组翻转后的尾部
        prev_group_tail = curr_group_head

# 辅助函数：反转链表 (与 206 题类似，但这里只返回新头)
def reverseList(head: ListNode) -> ListNode:
    prev = None
```

```
curr = head
while curr:
    next_temp = curr.next
    curr.next = prev
    prev = curr
    curr = next_temp
return prev
```

## 关键点总结

1. **分治思想**: 将大问题分解为“翻转 k 个节点”的子问题，然后迭代解决。
  2. **哑节点**: 简化了对链表头部的处理。
  3. **prev\_group\_tail 的作用**: 它连接了前一组翻转后的尾部和当前组翻转后的头部，是连接各组的关键。
  4. **指针的精细操作**: 在断开、翻转、重新连接过程中，需要非常小心地处理各个指针的指向，确保链表的完整性和正确性。
-