

0300. 最长递增子序列

- 标签：数组、二分查找、动态规划
- 难度：中等

题目链接

- [0300. 最长递增子序列 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ 。

要求：找到其中最长严格递增子序列的长度。

说明：

- **子序列：**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。
- $1 \leq nums.length \leq 2500$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。

示例：

- **示例 1：**

```
输入: nums = [10, 9, 2, 5, 3, 7, 101, 18]
输出: 4
解释: 最长递增子序列是 [2, 3, 7, 101]，因此长度为 4。
```

- **示例 2：**

```
输入: nums = [0, 1, 0, 3, 2, 3]
输出: 4
```

解题思路

思路 1：动态规划

1. 划分阶段

按照子序列的结尾位置进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：以 $nums[i]$ 结尾的最长递增子序列长度。

3. 状态转移方程

一个较小的数后边如果出现一个较大的数，则会形成一个更长的递增子序列。

对于满足 $0 \leq j < i$ 的数组元素 $nums[j]$ 和 $nums[i]$ 来说：

- 如果 $nums[j] < nums[i]$ ，则 $nums[i]$ 可以接在 $nums[j]$ 后面，此时以 $nums[i]$ 结尾的最长递增子序列长度会在「以 $nums[j]$ 结尾的最长递增子序列长度」的基础上加 1，即 $dp[i] = dp[j] + 1$ 。
- 如果 $nums[j] \leq nums[i]$ ，则 $nums[i]$ 不可以接在 $nums[j]$ 后面，可以直接跳过。

综上，我们的状态转移方程为： $dp[i] = \max(dp[i], dp[j] + 1), 0 \leq j < i, nums[j] < nums[i]$ 。

4. 初始条件

默认状态下，把数组中的每个元素都作为长度为 1 的递增子序列。即 $dp[i] = 1$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：以 $nums[i]$ 结尾的最长递增子序列长度。那为了计算出最大的最长递增子序列长度，则需要再遍历一遍 dp 数组，求出最大值即为最终结果。

思路 1：动态规划代码

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        size = len(nums)
        dp = [1 for _ in range(size)]

        for i in range(size):
            for j in range(i):
                if nums[i] > nums[j]:
                    dp[i] = max(dp[i], dp[j] + 1)

        return max(dp)
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。两重循环遍历的时间复杂度是 $O(n^2)$ ，最后求最大值的时间复杂度是 $O(n)$ ，所以总体时间复杂度为 $O(n^2)$ 。
- 空间复杂度： $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。

0303. 区域和检索 - 数组不可变

- 标签：设计、数组、前缀和
- 难度：简单

题目链接

- [0303. 区域和检索 - 数组不可变 - 力扣](#)

题目大意

描述：给定一个整数数组 `nums`。

要求：实现 `NumArray` 类，该类能处理区间为 `[left, right]` 之间的区间求和的多次查询。

`NumArray` 类：

- `NumArray(int[] nums)` 使用数组 `nums` 初始化对象。
- `int sumRange(int i, int j)` 返回数组 `nums` 中索引 `left` 和 `right` 之间的元素的总和，包含 `left` 和 `right` 两点（也就是 `nums[left] + nums[left + 1] + ... + nums[right]`）。

说明：

- $1 \leq nums.length \leq 10^4$ 。
- $-10^5 \leq nums[i] \leq 10^5$ 。
- $0 \leq left \leq right < nums.length$ 。
- `sumRange` 方法调用次数不超过 10^4 次。

示例：

- 示例 1：

给定 `nums = [-2, 0, 3, -5, 2, -1]`

求和 `sumRange(0, 2) -> 1`

求和 `sumRange(2, 5) -> -1`

求和 `sumRange(0, 5) -> -3`

解题思路

思路 1：线段树

- 根据 `nums` 数组，构建一棵线段树。每个线段树的节点类存储当前区间的左右边界和该区间的和。

这样构建线段树的时间复杂度为 $O(\log n)$ ，单次区间查询的时间复杂度为 $O(\log n)$ 。总体时间复杂度为 $O(\log n)$ 。

思路 1 线段树代码：

```

# 线段树的节点类
class SegTreeNode:
    def __init__(self, val=0):
        self.left = -1 # 区间左边界
        self.right = -1 # 区间右边界
        self.val = val # 节点值 (区间值)

# 线段树类
class SegmentTree:
    # 初始化线段树接口
    def __init__(self, nums, function):
        self.size = len(nums)
        self.tree = [SegTreeNode() for _ in range(4 * self.size)] # 维护 SegTreeNode 数组
        self.nums = nums # 原始数据
        self.function = function # function 是一个函数, 左右区间的聚合方法
        if self.size > 0:
            self.__build(0, 0, self.size - 1)

    # 单点更新接口: 将 nums[i] 更改为 val
    def update_point(self, i, val):
        self.nums[i] = val
        self.__update_point(i, val, 0)

    # 区间查询接口: 查询区间为 [q_left, q_right] 的区间值
    def query_interval(self, q_left, q_right):
        return self.__query_interval(q_left, q_right, 0)

    # 获取 nums 数组接口: 返回 nums 数组
    def get_nums(self):
        for i in range(self.size):
            self.nums[i] = self.query_interval(i, i)
        return self.nums

    # 以下为内部实现方法

    # 构建线段树实现方法: 节点的存储下标为 index, 节点的区间为 [left, right]
    def __build(self, index, left, right):
        self.tree[index].left = left
        self.tree[index].right = right
        if left == right: # 叶子节点, 节点值为对应位置的元素值
            self.tree[index].val = self.nums[left]
            return

        mid = left + (right - left) // 2 # 左右节点划分点
        left_index = index * 2 + 1 # 左子节点的存储下标
        right_index = index * 2 + 2 # 右子节点的存储下标
        self.__build(left_index, left, mid) # 递归创建左子树
        self.__build(right_index, mid + 1, right) # 递归创建右子树
        self.__pushup(index) # 向上更新节点的区间值

    # 单点更新实现方法: 将 nums[i] 更改为 val。节点的存储下标为 index, 节点的区间为 [left, right]
    def __update_point(self, i, val, index):
        left = self.tree[index].left
        right = self.tree[index].right

        if left == right:
            self.tree[index].val = val # 叶子节点, 节点值修改为 val
            return

```

```

        mid = left + (right - left) // 2           # 左右节点划分点
        left_index = index * 2 + 1                  # 左子节点的存储下标
        right_index = index * 2 + 2                  # 右子节点的存储下标
        if i <= mid:                                # 在左子树中更新节点值
            self.__update_point(i, val, left_index)
        else:                                       # 在右子树中更新节点值
            self.__update_point(i, val, right_index)

        self.__pushup(index)                         # 向上更新节点的区间值

# 区间查询实现方法：在线段树的 [left, right] 区间范围内搜索区间为 [q_left, q_right] 的区间值
def __query_interval(self, q_left, q_right, index):
    left = self.tree[index].left
    right = self.tree[index].right

    if left >= q_left and right <= q_right:      # 节点所在区间被 [q_left, q_right] 所覆盖
        return self.tree[index].val                # 直接返回节点值
    if right < q_left or left > q_right:          # 节点所在区间与 [q_left, q_right] 无关
        return 0

    mid = left + (right - left) // 2           # 左右节点划分点
    left_index = index * 2 + 1                  # 左子节点的存储下标
    right_index = index * 2 + 2                  # 右子节点的存储下标
    res_left = 0                                # 左子树查询结果
    res_right = 0                               # 右子树查询结果
    if q_left <= mid:                           # 在左子树中查询
        res_left = self.__query_interval(q_left, q_right, left_index)
    if q_right > mid:                           # 在右子树中查询
        res_right = self.__query_interval(q_left, q_right, right_index)

    return self.function(res_left, res_right)    # 返回左右子树元素值的聚合计算结果

# 向上更新实现方法：下标为 index 的节点区间值 等于 该节点左右子节点元素值的聚合计算结果
def __pushup(self, index):
    left_index = index * 2 + 1                  # 左子节点的存储下标
    right_index = index * 2 + 2                  # 右子节点的存储下标
    self.tree[index].val = self.function(self.tree[left_index].val, self.tree[right_index].val)

class NumArray:

    def __init__(self, nums: List[int]):
        self.STree = SegmentTree(nums, lambda x, y: x + y)

    def sumRange(self, left: int, right: int) -> int:
        return self.STree.query_interval(left, right)

```

0304. 二维区域和检索 - 矩阵不可变

- 标签：设计、数组、矩阵、前缀和
- 难度：中等

题目链接

- [0304. 二维区域和检索 - 矩阵不可变 - 力扣](#)

题目大意

给定一个二维矩阵 `matrix`。

要求：满足以下多个请求：

- `def sumRegion(self, row1: int, col1: int, row2: int, col2: int) -> int:` 计算以 `(row1, col1)` 为左上角、`(row2, col2)` 为右下角的子矩阵中各个元素的和。
- `def __init__(self, matrix: List[List[int]]):` 对二维矩阵 `matrix` 进行初始化操作。

解题思路

在进行初始化的时候做预处理，这样在多次查询时可以减少重复计算，也可以减少时间复杂度。

在进行初始化的时候，使用一个二维数组 `pre_sum` 记录下以 `(0, 0)` 为左上角，以当前 `(row, col)` 为右下角的子数组各个元素和，即 `pre_sum[row + 1][col + 1]`。

则在查询时，以 `(row1, col1)` 为左上角、`(row2, col2)` 为右下角的子矩阵中各个元素的和就等于以 `(0, 0)` 到 `(row2, col2)` 的大子矩阵减去左边 `(0, 0)` 到 `(row2, col1 - 1)` 的子矩阵，再减去上边 `(0, 0)` 到 `(row1 - 1, col2)` 的子矩阵，再加上左上角 `(0, 0)` 到 `(row1 - 1, col1 - 1)` 的子矩阵（因为之前重复减了）。即

```
pre_sum[row2 + 1][col2 + 1] - self.pre_sum[row2 + 1][col1] - self.pre_sum[row1][col2 + 1] + self.pre_sum[row1][col1].
```

代码

```
class NumMatrix:

    def __init__(self, matrix: List[List[int]]):
        rows = len(matrix)
        cols = len(matrix[0])
        self.pre_sum = [[0 for _ in range(cols + 1)] for _ in range(rows + 1)]
        for row in range(rows):
            for col in range(cols):
                self.pre_sum[row + 1][col + 1] = self.pre_sum[row + 1][col] + self.pre_sum[row][col + 1] - self.pre_sum[row][col] + ma

    def sumRegion(self, row1: int, col1: int, row2: int, col2: int) -> int:
        return self.pre_sum[row2 + 1][col2 + 1] - self.pre_sum[row2 + 1][col1] - self.pre_sum[row1][col2 + 1] + self.pre_sum[row1][col1]
```

0307. 区域和检索 - 数组可修改

- 标签：设计、树状数组、线段树、数组
- 难度：中等

题目链接

- [0307. 区域和检索 - 数组可修改 - 力扣](#)

题目大意

描述：给定一个数组 `nums`。

要求：

1. 完成两类查询：
 - 要求将数组元素 `nums[index]` 的值更新为 `val`。
 - 要求返回数组 `nums` 中区间 `[left, right]` 之间（包含 `left`、`right`）的 `nums` 元素的和。其中 $left \leq right$ 。
2. 实现 `NumArray` 类：
 - `NumArray(int[] nums)` 用整数数组 `nums` 初始化对象。
 - `void update(int index, int val)` 将 `nums[index]` 的值更新为 `val`。
 - `int sumRange(int left, int right)` 返回数组 `nums` 中索引 `left` 和索引 `right` 之间（包含）的 `nums` 元素的和（即 `nums[left] + nums[left + 1], ..., nums[right]`）。

说明:

- $1 \leq \text{nums.length} \leq 3 * 10^4$ 。
- $-100 \leq \text{nums}[i] \leq 100$ 。
- $0 \leq \text{index} < \text{num.length}$ 。
- $0 \leq \text{left} \leq \text{right} < \text{nums.length}$ 。
- 调用 `update` 和 `sumRange` 的方法次数不大于 $3 * 10^4$ 次。

示例:

- 示例 1:

```
给定 nums = [1, 3, 5]
求和 sumRange(0, 2) -> 9
更新 update(1, 2)
求和 sumRange(0, 2) -> 8
```

解题思路

思路 1：线段树

根据 `nums` 数组，构建一棵线段树。每个线段树的节点类存储当前区间的左右边界和该区间的和。

这样构建线段树的时间复杂度为 $O(\log n)$ ，每次单点更新的时间复杂度为 $O(\log n)$ ，每次区间查询的时间复杂度为 $O(\log n)$ 。总体时间复杂度为 $O(\log n)$ 。

思路 1 线段树代码：

```

# 线段树的节点类
class SegTreeNode:
    def __init__(self, val=0):
        self.left = -1 # 区间左边界
        self.right = -1 # 区间右边界
        self.val = val # 节点值 (区间值)

# 线段树类
class SegmentTree:
    # 初始化线段树接口
    def __init__(self, nums, function):
        self.size = len(nums)
        self.tree = [SegTreeNode() for _ in range(4 * self.size)] # 维护 SegTreeNode 数组
        self.nums = nums # 原始数据
        self.function = function # function 是一个函数, 左右区间的聚合方法
        if self.size > 0:
            self.__build(0, 0, self.size - 1)

    # 单点更新接口: 将 nums[i] 更改为 val
    def update_point(self, i, val):
        self.nums[i] = val
        self.__update_point(i, val, 0)

    # 区间查询接口: 查询区间为 [q_left, q_right] 的区间值
    def query_interval(self, q_left, q_right):
        return self.__query_interval(q_left, q_right, 0)

    # 获取 nums 数组接口: 返回 nums 数组
    def get_nums(self):
        for i in range(self.size):
            self.nums[i] = self.query_interval(i, i)
        return self.nums

    # 以下为内部实现方法

    # 构建线段树实现方法: 节点的存储下标为 index, 节点的区间为 [left, right]
    def __build(self, index, left, right):
        self.tree[index].left = left
        self.tree[index].right = right
        if left == right: # 叶子节点, 节点值为对应位置的元素值
            self.tree[index].val = self.nums[left]
            return

        mid = left + (right - left) // 2 # 左右节点划分点
        left_index = index * 2 + 1 # 左子节点的存储下标
        right_index = index * 2 + 2 # 右子节点的存储下标
        self.__build(left_index, left, mid) # 递归创建左子树
        self.__build(right_index, mid + 1, right) # 递归创建右子树
        self.__pushup(index) # 向上更新节点的区间值

    # 单点更新实现方法: 将 nums[i] 更改为 val。节点的存储下标为 index, 节点的区间为 [left, right]
    def __update_point(self, i, val, index):
        left = self.tree[index].left
        right = self.tree[index].right

        if left == right:
            self.tree[index].val = val # 叶子节点, 节点值修改为 val
            return

```

```

        mid = left + (right - left) // 2           # 左右节点划分点
        left_index = index * 2 + 1                  # 左子节点的存储下标
        right_index = index * 2 + 2                 # 右子节点的存储下标
        if i <= mid:                                # 在左子树中更新节点值
            self.__update_point(i, val, left_index)
        else:                                       # 在右子树中更新节点值
            self.__update_point(i, val, right_index)

        self.__pushup(index)                         # 向上更新节点的区间值

# 区间查询实现方法：在线段树的 [left, right] 区间范围内搜索区间为 [q_left, q_right] 的区间值
def __query_interval(self, q_left, q_right, index):
    left = self.tree[index].left
    right = self.tree[index].right

    if left >= q_left and right <= q_right:      # 节点所在区间被 [q_left, q_right] 所覆盖
        return self.tree[index].val                # 直接返回节点值
    if right < q_left or left > q_right:          # 节点所在区间与 [q_left, q_right] 无关
        return 0

    mid = left + (right - left) // 2           # 左右节点划分点
    left_index = index * 2 + 1                  # 左子节点的存储下标
    right_index = index * 2 + 2                 # 右子节点的存储下标
    res_left = 0                                # 左子树查询结果
    res_right = 0                               # 右子树查询结果
    if q_left <= mid:                           # 在左子树中查询
        res_left = self.__query_interval(q_left, q_right, left_index)
    if q_right > mid:                           # 在右子树中查询
        res_right = self.__query_interval(q_left, q_right, right_index)

    return self.function(res_left, res_right)    # 返回左右子树元素值的聚合计算结果

# 向上更新实现方法：下标为 index 的节点区间值 等于 该节点左右子节点元素值的聚合计算结果
def __pushup(self, index):
    left_index = index * 2 + 1                  # 左子节点的存储下标
    right_index = index * 2 + 2                 # 右子节点的存储下标
    self.tree[index].val = self.function(self.tree[left_index].val, self.tree[right_index].val)

class NumArray:

    def __init__(self, nums: List[int]):
        self.STree = SegmentTree(nums, lambda x, y: x + y)

    def update(self, index: int, val: int) -> None:
        self.STree.update_point(index, val)

    def sumRange(self, left: int, right: int) -> int:
        return self.STree.query_interval(left, right)

```

0309. 最佳买卖股票时机含冷冻期

- 标签：数组、动态规划
- 难度：中等

题目链接

- [0309. 最佳买卖股票时机含冷冻期 - 力扣](#)

题目大意

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- 卖出股票后，你无法在第二天买入股票（即冷冻期为 1 天）。

解题思路

这道题是「0122. 买卖股票的最佳时机 II」的升级版。

冷冻期的意思是：如果昨天卖出了，那么今天不能买。在考虑的时候只要判断一下前一天是不是刚卖出。

对于每一天结束时的状态总共有以下几种：

- 买入状态：
 - 今日买入
 - 之前买入，之后一直持有无操作
- 卖出状态：
 - 今日卖出，正处于冷冻期
 - 昨天卖出，今天结束后度过了冷冻期
 - 之前卖出，度过了冷冻期后无操作

在买入状态中，今日买入和之前买入的状态其实可以看做是股票的持有状态，可以将其合并为一种状态。

在卖出状态中，昨天卖出和之前卖出的状态其实可以看做是无股票并度过了冷冻期状态，可以将其合并为一种状态。

这样总结下来可以划分为三个状态：

- 股票的持有状态。
- 无股票，并且处于冷冻期状态。
- 无股票，并且不处于冷冻期状态。

所以我们可以定义状态 $dp[i][j]$ ，表示为：第 i 天第 j 种情况 ($0 \leq j \leq 2$) 下，所获取的最大利润。

注意：这里第 j 种情况，

接下来确定状态转移公式：

- 第 0 种状态（股票的持有状态）下可以有两种状态推出，取最大的那一种赋值：
 - 昨天就已经持有的： $dp[i][0] = dp[i - 1][0]$:
 - 今天刚买入的（则昨天不能持有股票也不能处于冷冻期，应来自于前天卖出状态）： $dp[i][0] = dp[i - 1][2] - prices[i]$
- 第 1 种状态（无股票，并且处于冷冻期状态）下可以有一种状态推出：
 - 今天卖出： $dp[i] = dp[i - 1][0] + prices[i]$
- 第 2 种状态（无股票，并且不处于冷冻期状态）下可以有两种状态推出，取最大的那一种赋值：
 - 昨天卖出： $dp[i] = dp[i - 1][1]$
 - 之前卖出： $dp[i] = dp[i - 1][2]$

下面确定初始化的边界值：

可以很明显看出第一天不做任何操作就是 $dp[0][0] = 0$ ，第一次买入就是 $dp[0][1] = -prices[0]$ 。

第一次卖出的话，可以视作为没有盈利（当天买卖，价格没有变化），即 $dp[0][2] = 0$ 。第二次买入的话，就是 $dp[0][3] = -prices[1]$ 。同理第二次卖出就是 $dp[0][4] = 0$ 。

在递推结束后，最大利润肯定是无操作、第一次卖出、第二次卖出这三种情况里边，且为最大值。我们在维护的时候维护的是最大值，则第一次卖出、第二次卖出所获得的利润肯定大于等于 0。而且，如果最优情况为一笔交易，那么在转移状态时，我们允许在一天内进行两次交易，则一笔交易的状态可以转移至两笔交易。所以最终答案为 $dp[size - 1][4]$ 。 $size$ 为股票天数。

代码

```

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        size = len(prices)
        if size == 0:
            return 0
        dp = [[0 for _ in range(4)] for _ in range(size)]

        dp[0][0] = -prices[0]
        for i in range(1, size):
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][2] - prices[i])
            dp[i][1] = dp[i - 1][0] + prices[i]
            dp[i][2] = max(dp[i - 1][1], dp[i - 1][2])
        return max(dp[size - 1][0], dp[size - 1][1], dp[size - 1][2])

```

0310. 最小高度树

- 标签：深度优先搜索、广度优先搜索、图、拓扑排序
- 难度：中等

题目链接

- [0310. 最小高度树 - 力扣](#)

题目大意

描述：有一棵包含 n 个节点的树，节点编号为 $0 \sim n - 1$ 。给定一个数字 n 和一个有 $n - 1$ 条无向边的 $edges$ 列表来表示这棵树。其中 $edges[i] = [ai, bi]$ 表示树中节点 ai 和 bi 之间存在一条无向边。

可以选择树中的任何一个节点作为根，当选择节点 x 作为根节点时，设结果树的高度为 h 。在所有可能的树种，具有最小高度的树（即 $\min(h)$ ）被成为最小高度树。

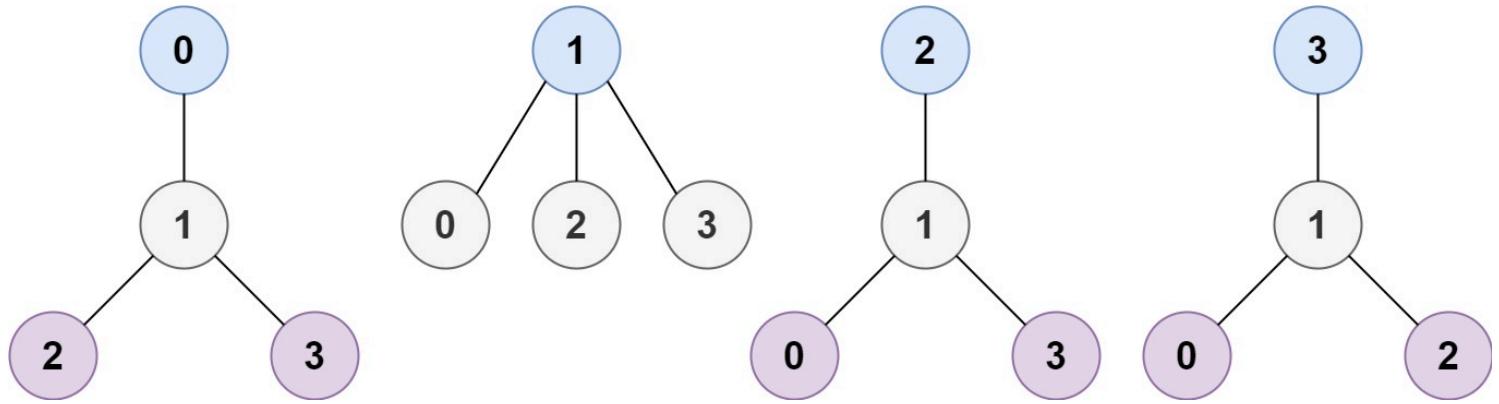
要求：找到所有的最小高度树并按照任意顺序返回他们的根节点编号列表。

说明：

- **树的高度：**指根节点和叶子节点之间最长向下路径上边的数量。
- $1 \leq n \leq 2 * 10^4$ 。
- $edges.length == n - 1$ 。
- $0 \leq ai, bi < n$ 。
- $ai \neq bi$ 。
- 所有 (ai, bi) 互不相同。
- 给定的输入保证是一棵树，并且不会有重复的边。

示例：

- **示例 1：**

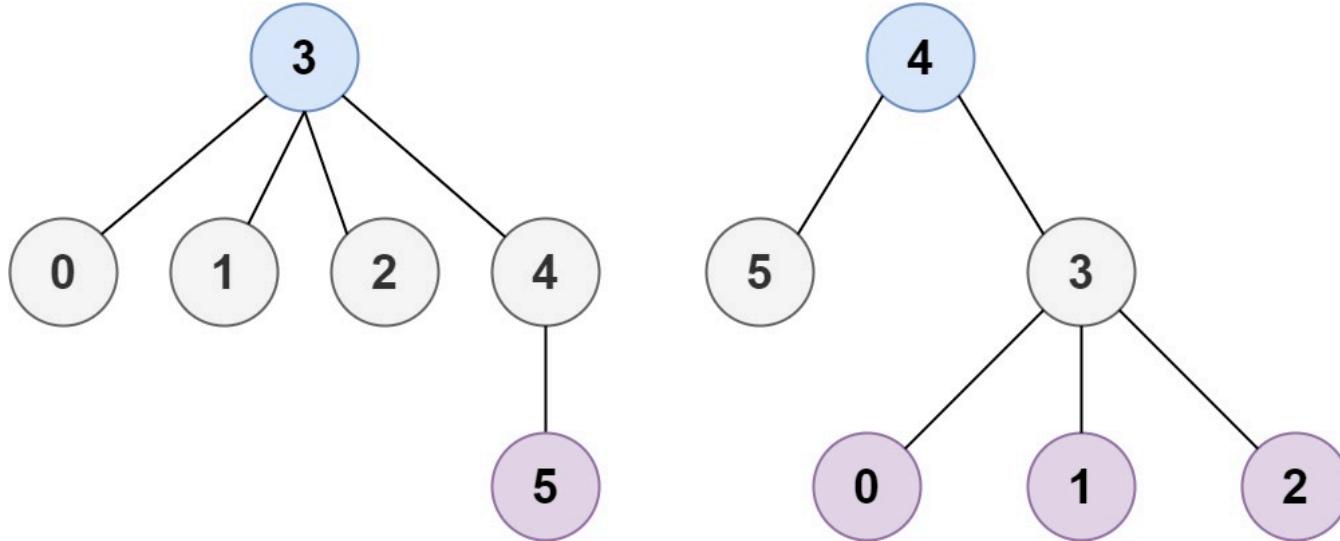


输入: `n = 4, edges = [[1,0],[1,2],[1,3]]`

输出: [1]

解释: 如图所示, 当根是标签为 1 的节点时, 树的高度是 1, 这是最小高度树。

- 示例 2:



输入: `n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]`

输出: [3, 4]

解题思路

思路 1：树形 DP + 二次遍历换根法

最容易想到的做法是: 枚举 n 个节点, 以每个节点为根节点, 然后进行深度优先搜索, 求出每棵树的高度。最后求出所有树中的最小高度即为答案。但这种做法的时间复杂度为 $O(n^2)$, 而 n 的范围为 $[1, 2 * 10^4]$, 这样做会导致超时, 因此需要进行优化。

在上面的算法中, 在一轮深度优先搜索中, 除了可以得到整棵树的高度之外, 在搜索过程中, 其实还能得到以每个子节点为根节点的树的高度。如果我们能够利用这些子树的高度信息, 快速得到以其他节点为根节点的树的高度, 那么我们就能改进算法, 以更小的时间复杂度解决这道题。这就是二次遍历与换根法的思想。

- 第一次遍历: 自底向上的计算出每个节点 u 向下走 (即由父节点 u 向子节点 v 走) 的最长路径 $down1[u]$ 、次长路径 $down2[i]$, 并记录向下走最长路径所经过的子节点 $p[u]$, 方便第二次遍历时计算。
- 第二次遍历: 自顶向下的计算出每个节点 v 向上走 (即由子节点 v 向父节点 u 走) 的最长路径 $up[v]$ 。需要注意判断 u 向下走的最长路径是否经过了节点 v 。
 - 如果经过了节点 v , 则向上走的最长路径, 取决于「父节点 u 向上走的最长路径」与「父节点 u 向下走的次长路径」的较大值, 再加上 1。
 - 如果没有经过节点 v , 则向上走的最长路径, 取决于「父节点 u 向上走的最长路径」与「父节点 u 向下走的最长路径」的较大值, 再加上 1。
- 接下来, 我们通过枚举 n 个节点向上走的最长路径与向下走的最长路径, 从而找出所有树中的最小高度, 并将所有最小高度树的根节点放入答案数组中并返回。

整个算法具体步骤如下：

1. 使用邻接表的形式存储树。
2. 定义第一个递归函数 `dfs(u, fa)` 用于计算每个节点向下走的最长路径 $down1[u]$ 、次长路径 $down2[u]$ ，并记录向下走的最长路径所经过的子节点 $p[u]$ 。
 - i. 对当前节点的相邻节点进行遍历。
 - ii. 如果相邻节点是父节点，则跳过。
 - iii. 递归调用 `dfs(v, u)` 函数计算邻居节点的信息。
 - iv. 根据邻居节点的信息计算当前节点的高度，并更新当前节点向下走的最长路径 $down1[u]$ 、当前节点向下走的次长路径 $down2$ 、取得最长路径的子节点 $p[u]$ 。
3. 定义第二个递归函数 `reroot(u, fa)` 用于计算每个节点作为新的根节点时向上走的最长路径 $up[v]$ 。
 - i. 对当前节点的相邻节点进行遍历。
 - ii. 如果相邻节点是父节点，则跳过。
 - iii. 根据当前节点 u 的高度和相邻节点 v 的信息更新 $up[v]$ 。同时需要判断节点 u 向下走的最长路径是否经过了节点 v 。
 - a. 如果经过了节点 v ，则向上走的最长路径，取决于「父节点 u 向上走的最长路径」与「父节点 u 向下走的次长路径」的较大值，再加上 1，即： $up[v] = \max(up[u], down2[u]) + 1$ 。
 - b. 如果没有经过节点 v ，则向上走的最长路径，取决于「父节点 u 向上走的最长路径」与「父节点 u 向下走的最长路径」的较大值，再加上 1，即： $up[v] = \max(up[u], down1[u]) + 1$ 。
 - iv. 递归调用 `reroot(v, u)` 函数计算邻居节点的信息。
4. 调用 `dfs(0, -1)` 函数计算每个节点的最长路径。
5. 调用 `reroot(0, -1)` 函数计算每个节点作为新的根节点时的最长路径。
6. 找到所有树中的最小高度。
7. 将所有最小高度的节点放入答案数组中并返回。

思路 1：代码

```

class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        graph = [[] for _ in range(n)]
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        # down1 用于记录向下走的最长路径
        down1 = [0 for _ in range(n)]
        # down2 用于记录向下走的最长路径
        down2 = [0 for _ in range(n)]
        p = [0 for _ in range(n)]
        # 自底向上记录最长路径、次长路径
        def dfs(u, fa):
            for v in graph[u]:
                if v == fa:
                    continue
                # 自底向上统计信息
                dfs(v, u)
                height = down1[v] + 1
                if height >= down1[u]:
                    down2[u] = down1[u]
                    down1[u] = height
                    p[u] = v
                elif height > down2[u]:
                    down2[u] = height

        # 进行换根动态规划，自顶向下统计向上走的最长路径
        up = [0 for _ in range(n)]
        def reroot(u, fa):
            for v in graph[u]:
                if v == fa:
                    continue
                if p[u] == v:
                    up[v] = max(up[u], down2[u]) + 1
                else:
                    up[v] = max(up[u], down1[u]) + 1
            # 自顶向下统计信息
            reroot(v, u)

        dfs(0, -1)
        reroot(0, -1)

        # 找到所有树中的最小高度
        min_h = 1e9
        for i in range(n):
            min_h = min(min_h, max(down1[i], up[i]))

        # 将所有最小高度的节点放入答案数组中并返回
        res = []
        for i in range(n):
            if max(down1[i], up[i]) == min_h:
                res.append(i)

        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

参考资料

- 【题解】C++ 容易理解的换根动态规划解法 - 最小高度树
- 【题解】310. 最小高度树 - 力扣
- 【题解】310. 最小高度树 - 力扣

0312. 戳气球

- 标签：数组、动态规划
- 难度：困难

题目链接

- [0312. 戳气球 - 力扣](#)

题目大意

描述：有 n 个气球，编号为 $0 \sim n - 1$ ，每个气球上都有一个数字，这些数字存在数组 nums 中。现在开始戳破气球。其中戳破第 i 个气球，可以获得 $\text{nums}[i - 1] \times \text{nums}[i] \times \text{nums}[i + 1]$ 枚硬币，这里的 $i - 1$ 和 $i + 1$ 代表和 i 相邻的两个气球的编号。如果 $i - 1$ 或 $i + 1$ 超出了数组的边界，那么就当它是一个数字为 1 的气球。

要求：求出能获得硬币的最大数量。

说明：

- $n == \text{nums.length}$ 。
- $1 \leq n \leq 300$ 。
- $0 \leq \text{nums}[i] \leq 100$ 。

示例：

- 示例 1：

```
输入: nums = [3, 1, 5, 8]
输出: 167
解释:
nums = [3, 1, 5, 8] --> [3, 5, 8] --> [3, 8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167
```

- 示例 2：

```
输入: nums = [1, 5]
输出: 10
解释:
nums = [1, 5] --> [5] --> []
coins = 1*1*5 + 1*5*1 = 10
```

解题思路

思路 1：动态规划

根据题意，如果 $i - 1$ 或 $i + 1$ 超出了数组的边界，那么就当它是一个数字为 1 的气球。我们可以预先在 nums 的首尾位置，添加两个数字为 1 的虚拟气球，这样变成了 $n + 2$ 个气球，气球对应编号也变为了 $0 \sim n + 1$ 。

对应问题也变成了：给定 $n + 2$ 个气球，每个气球上有 1 个数字，代表气球上的硬币数量，当我们戳破气球 $\text{nums}[i]$ 时，就能得到对应 $\text{nums}[i - 1] \times \text{nums}[i] \times \text{nums}[i + 1]$ 枚硬币。现在要戳破 $0 \sim n + 1$ 之间的所有气球（不包括编号 0 和编号 $n + 1$ 的气球），请问最多能获得多少枚硬币？

1. 划分阶段

按照区间长度进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：戳破所有气球 i 与气球 j 之间的气球（不包含气球 i 和气球 j ），所能获取的最多硬币数。

3. 状态转移方程

假设气球 i 与气球 j 之间最后一个被戳破的气球编号为 k 。则 $dp[i][j]$ 取决于由 k 作为分割点分割出的两个区间 (i, k) 与 (k, j) 上所能获取的最多硬币数 + 戳破气球 k 所能获得的硬币数，即状态转移方程为：

$$dp[i][j] = \max\{dp[i][k] + dp[k][j] + nums[i] \times nums[k] \times nums[j]\}, \quad i < k < j$$

4. 初始条件

- $dp[i][j]$ 表示的是开区间，则 $i < j - 1$ 。而当 $i \geq j - 1$ 时，所能获得的硬币数为 0，即 $dp[i][j] = 0$ ， $i \geq j - 1$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：戳破所有气球 i 与气球 j 之间的气球（不包含气球 i 和气球 j ），所能获取的最多硬币数。。所以最终结果为 $dp[0][n + 1]$ 。

思路 1：代码

```
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        size = len(nums)
        arr = [0 for _ in range(size + 2)]
        arr[0] = arr[size + 1] = 1
        for i in range(1, size + 1):
            arr[i] = nums[i - 1]

        dp = [[0 for _ in range(size + 2)] for _ in range(size + 2)]

        for l in range(3, size + 3):
            for i in range(0, size + 2):
                j = i + l - 1
                if j >= size + 2:
                    break
                for k in range(i + 1, j):
                    dp[i][j] = max(dp[i][j], dp[i][k] + dp[k][j] + arr[i] * arr[j] * arr[k])

        return dp[0][size + 1]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^3)$ ，其中 n 为气球数量。
- 空间复杂度： $O(n^2)$ 。

0315. 计算右侧小于当前元素的个数

- 标签：树状数组、线段树、数组、二分查找、分治、有序集合、归并排序
- 难度：困难

题目链接

- [0315. 计算右侧小于当前元素的个数 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ 。

要求：返回一个新数组 $counts$ 。其中 $counts[i]$ 的值是 $nums[i]$ 右侧小于 $nums[i]$ 的元素的数量。

说明：

- $1 \leq nums.length \leq 10^5$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。

示例：

- **示例 1：**

```
输入: nums = [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1)
2 的右侧仅有 1 个更小的元素 (1)
6 的右侧有 1 个更小的元素 (1)
1 的右侧有 0 个更小的元素
```

- **示例 2：**

```
输入: nums = [-1]
输出: [0]
```

解题思路

思路 1：归并排序

在使用归并排序对数组进行排序时，每当遇到 $left_nums[left_i] \leq right_nums[right_i]$ 时，意味着：在合并前，左子数组当前元素 $left_nums[left_i]$ 右侧一定有 $left_i$ 个元素比 $left_nums[left_i]$ 小。则我们可以在归并排序的同时，记录 $nums[i]$ 右侧小于 $nums[i]$ 的元素的数量。

1. 将元素值、对应下标、右侧小于 $nums[i]$ 的元素的数量存入数组中。
2. 对其进行归并排序。
3. 当遇到 $left_nums[left_i] \leq right_nums[right_i]$ 时，记录 $left_nums[left_i]$ 右侧比 $left_nums[left_i]$ 小的元素数量，即：`left_nums[left_i][2] += right_i`。
4. 当合并时 $left_nums[left_i]$ 仍有剩余时，说明 $left_nums[left_i]$ 右侧有 $right_i$ 个小于 $left_nums[left_i]$ 的元素，记录下来，即：`left_nums[left_i][2] += right_i`。
5. 根据下标及右侧小于 $nums[i]$ 的元素的数量，组合出答案数组，并返回答案数组。

思路 1：代码

```

class Solution:
    # 合并过程
    def merge(self, left_nums, right_nums):
        nums = []
        left_i, right_i = 0, 0
        while left_i < len(left_nums) and right_i < len(right_nums):
            # 将两个有序子数组中较小元素依次插入到结果数组中
            if left_nums[left_i] <= right_nums[right_i]:
                nums.append(left_nums[left_i])
                # left_nums[left_i] 右侧有 right_i 个比 left_nums[left_i] 小的
                left_nums[left_i][2] += right_i
                left_i += 1
            else:
                nums.append(right_nums[right_i])
                right_i += 1

        # 如果左子数组有剩余元素，则将其插入到结果数组中
        while left_i < len(left_nums):
            nums.append(left_nums[left_i])
            # left_nums[left_i] 右侧有 right_i 个比 left_nums[left_i] 小的
            left_nums[left_i][2] += right_i
            left_i += 1

        # 如果右子数组有剩余元素，则将其插入到结果数组中
        while right_i < len(right_nums):
            nums.append(right_nums[right_i])
            right_i += 1

        # 返回合并后的结果数组
        return nums

    # 分解过程
    def mergeSort(self, nums):
        # 数组元素个数小于等于 1 时，直接返回原数组
        if len(nums) <= 1:
            return nums

        mid = len(nums) // 2
        left_nums = self.mergeSort(nums[0: mid])           # 将数组从中间位置分为左右两个数组
        right_nums = self.mergeSort(nums[mid:])           # 递归将左子数组进行分解和排序
        return self.merge(left_nums, right_nums)           # 递归将右子数组进行分解和排序
                                                # 把当前数组组中有序子数组逐层向上，进行两两合并

def countSmaller(self, nums: List[int]) -> List[int]:
    size = len(nums)

    # 将元素值、对应下标、右侧小于 nums[i] 的元素的数量存入数组中
    nums = [[num, i, 0] for i, num in enumerate(nums)]
    nums = self.mergeSort(nums)
    ans = [0 for _ in range(size)]

    for num in nums:
        ans[num[1]] = num[2]

    return ans

```

思路 1：复杂度分析

- 时间复杂度： $O(n \times \log n)$ 。
- 空间复杂度： $O(n)$ 。

思路 2：树状数组

1. 首先对数组进行离散化处理。把原始数组中的数据映射到 $[0, \text{len}(\text{nums}) - 1]$ 这个区间。
2. 然后逆序顺序从数组 nums 中遍历元素 $\text{nums}[i]$ 。
 - i. 计算其离散化后的排名 index , 查询比 index 小的数有多少个。将其记录到答案数组的对应位置 $\text{ans}[i]$ 上。
 - ii. 然后在树状数组下标为 index 的位置上, 更新值为 1。
3. 遍历完所有元素, 最后输出答案数组 ans 即可。

思路 2：代码

```
import bisect

class BinaryIndexTree:

    def __init__(self, n):
        self.size = n
        self.tree = [0 for _ in range(n + 1)]

    def lowbit(self, index):
        return index & (-index)

    def update(self, index, delta):
        while index <= self.size:
            self.tree[index] += delta
            index += self.lowbit(index)

    def query(self, index):
        res = 0
        while index > 0:
            res += self.tree[index]
            index -= self.lowbit(index)
        return res

class Solution:
    def countSmaller(self, nums: List[int]) -> List[int]:
        size = len(nums)
        if size == 0:
            return []
        if size == 1:
            return [0]

        # 离散化
        sort_nums = list(set(nums))
        sort_nums.sort()
        size_s = len(sort_nums)
        bit = BinaryIndexTree(size_s)

        ans = [0 for _ in range(size)]
        for i in range(size - 1, -1, -1):
            index = bisect.bisect_left(sort_nums, nums[i]) + 1
            ans[i] = bit.query(index - 1)
            bit.update(index, 1)

        return ans
```

思路 2：复杂度分析

- 时间复杂度: $O(n \times \log n)$ 。
- 空间复杂度: $O(n)$ 。

0316. 去除重复字母

- 标签：栈、贪心、字符串、单调栈
- 难度：中等

题目链接

- [0316. 去除重复字母 - 力扣](#)

题目大意

描述：给定一个字符串 `s`。

要求：去除字符串中重复的字母，使得每个字母只出现一次。需要保证「返回结果的字典序最小（要求不能打乱其他字符的相对位置）」。

说明：

- $1 \leq s.length \leq 10^4$ 。
- `s` 由小写英文字母组成。

示例：

- **示例 1：**

```
输入: s = "bcabc"
输出: "abc"
```

- **示例 2：**

```
输入: s = "cbacdcbc"
输出: "acdb"
```

解题思路

思路 1：哈希表 + 单调栈

针对题目的三个要求：去重、不能打乱其他字符顺序、字典序最小。我们来一一分析。

1. **去重：**可以通过「使用哈希表存储字母出现次数」的方式，将每个字母出现的次数统计起来，再遍历一遍，去除重复的字母。
2. **不能打乱其他字符顺序：**按顺序遍历，将非重复的字母存储到答案数组或者栈中，最后再拼接起来，就能保证不打乱其他字符顺序。
3. **字典序最小：**意味着字典序小的字母应该尽可能放在前面。
 - i. 对于第 `i` 个字符 `s[i]` 而言，如果第 `0 ~ i - 1` 之间的某个字符 `s[j]` 在 `s[i]` 之后不再出现了，那么 `s[j]` 必须放到 `s[i]` 之前。
 - ii. 而如果 `s[j]` 在之后还会出现，并且 `s[j]` 的字典序大于 `s[i]`，我们则可以先舍弃 `s[j]`，把 `s[i]` 尽可能的放到前面。后边再考虑使用 `s[j]` 所对应的字符。

要满足第 3 条需求，我们可以使用「单调栈」来解决。我们使用单调栈存储 `s[i]` 之前出现的非重复、并且字典序最小的字符序列。整个算法步骤如下：

1. 先遍历一遍字符串，用哈希表 `letter_counts` 统计出每个字母出现的次数。
2. 然后使用单调递减栈保存当前字符之前出现的非重复、并且字典序最小的字符序列。
3. 当遍历到 `s[i]` 时，如果 `s[i]` 没有在栈中出现过：
 - i. 比较 `s[i]` 和栈顶元素 `stack[-1]` 的字典序。如果 `s[i]` 的字典序小于栈顶元素 `stack[-1]`，并且栈顶元素之后的出现次数大于 `0`，则将栈顶元素弹出。
 - ii. 然后继续判断 `s[i]` 和栈顶元素 `stack[-1]`，并且知道栈顶元素出现次数为 `0` 时停止弹出。此时将 `s[i]` 添加到单调栈中。
4. 从哈希表 `letter_counts` 中减去 `s[i]` 出现的次数，继续遍历。
5. 最后将单调栈中的字符依次拼接为答案字符串，并返回。

思路 1：代码

```

class Solution:
    def removeDuplicateLetters(self, s: str) -> str:
        stack = []
        letter_counts = dict()
        for ch in s:
            if ch in letter_counts:
                letter_counts[ch] += 1
            else:
                letter_counts[ch] = 1

        for ch in s:
            if ch not in stack:
                while stack and ch < stack[-1] and stack[-1] in letter_counts and letter_counts[stack[-1]] > 0:
                    stack.pop()
                stack.append(ch)
                letter_counts[ch] -= 1

        return ''.join(stack)

```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(|\Sigma|)$ ，其中 Σ 为字符集合， $|\Sigma|$ 为字符种类个数。由于栈中字符不能重复，因此栈中最多有 $|\Sigma|$ 个字符。

参考资料

- 【题解】去除重复数组 - 去除重复字母 - 力扣（LeetCode）

0318. 最大单词长度乘积

- 标签：位运算、数组、字符串
- 难度：中等

题目链接

- [0318. 最大单词长度乘积 - 力扣](#)

题目大意

给定一个字符串数组 `words`。字符串中只包含英语的小写字母。

要求：计算当两个字符串 `words[i]` 和 `words[j]` 不包含相同字符时，它们长度的乘积的最大值。如果没有不包含相同字符的一对字符串，返回 0。

解题思路

这道题的核心难点是判断任意两个字符串之间是否包含相同字符。最直接的做法是先遍历第一个字符串的每个字符，再遍历第二个字符串查看是否有相同字符。但是这样做的话，时间复杂度过高。考虑怎么样可以优化一下。

题目中说字符串中只包含英语的小写字母，也就是 26 种字符。一个 32 位的 `int` 整数每一个二进制位都可以表示一种字符的有无，那么我们就可以通过一个整数来表示一个字符串中所拥有的字符种类。延伸一下，我们可以用一个整数数组来表示一个字符串数组中，每个字符串所拥有的字符种类。

接下来事情就简单了，两重循环遍历整数数组，遇到两个字符串不包含相同字符的情况，就计算一下他们长度的乘积，并维护一个乘积最大值。最后输出最大值即可。

代码

```

class Solution:
    def maxProduct(self, words: List[str]) -> int:
        size = len(words)
        arr = [0 for _ in range(size)]
        for i in range(size):
            word = words[i]
            len_word = len(word)
            for j in range(len_word):
                arr[i] |= 1 << (ord(word[j]) - ord('a'))
        ans = 0
        for i in range(size):
            for j in range(i + 1, size):
                if arr[i] & arr[j] == 0:
                    k = len(words[i]) * len(words[j])
                    ans = k if ans < k else ans
        return ans

```

0322. 零钱兑换

- 标签：广度优先搜索、数组、动态规划
- 难度：中等

题目链接

- 0322. 零钱兑换 - 力扣

题目大意

描述：给定代表不同面额的硬币数组 coins 和一个总金额 amount 。

要求：求出凑成总金额所需的最少的硬币个数。如果无法凑出，则返回 -1 。

说明：

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

示例：

- 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

```

解题思路

思路 1：广度优先搜索

我们可以从 $amount$ 开始，每次从 $coins$ 的硬币中选中 1 枚硬币，并记录当前挑选硬币的次数。则最快减到 0 的次数就是凑成总金额所需的最少的硬币个数。这道题就变成了从 $amount$ 减到 0 的最短路径问题。我们可以用广度优先搜索的方法来做。

1. 定义 $visited$ 为标记已访问值的集合变量， $queue$ 为存放值的队列。
2. 将 $amount$ 状态标记为访问，并将其加入队列 $queue$ 。
3. 令当前步数加 1，然后将当前队列中的所有值依次出队，并遍历硬币数组：
 - i. 如果当前值等于当前硬币值，则说明当前硬币刚好能凑成当前值，则直接返回当前次数。
 - ii. 如果当前值大于当前硬币值，并且当前值减去当前硬币值的差值没有出现在已访问集合 $visited$ 中，则将差值添加到队列和访问集合中。
4. 重复执行第 3 步，直到队列为空。
5. 如果队列为空，也未能减到 0，则返回 -1 。

思路 1：代码

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        if amount == 0:
            return 0

        visited = set([amount])
        queue = collections.deque([amount])

        step = 0
        while queue:
            step += 1
            size = len(queue)
            for _ in range(size):
                cur = queue.popleft()
                for coin in coins:
                    if cur == coin:
                        step += 1
                        return step
                    elif cur > coin and cur - coin not in visited:
                        queue.append(cur - coin)
                        visited.add(cur - coin)

        return -1
```

思路 1：复杂度分析

- 时间复杂度： $O(amount \times size)$ 。其中 $amount$ 表示总金额， $size$ 表示硬币的种类数。
- 空间复杂度： $O(amount)$ 。

思路 2：完全背包问题

这道题可以转换为：有 n 种不同的硬币， $coins[i]$ 表示第 i 种硬币的面额，每种硬币可以无限次使用。请问恰好凑成总金额为 $amount$ 的背包，最少需要多少硬币？

与普通完全背包问题不同的是，这里求解的是最少硬币数量。我们可以改变一下「状态定义」和「状态转移方程」。

1. 划分阶段

按照当前背包的载重上限进行阶段划分。

2. 定义状态

定义状态 $dp[c]$ 表示为：凑成总金额为 c 的最少硬币数量。

3. 状态转移方程

$$dp[c] = \begin{cases} dp[c] & c < coins[i - 1] \\ \min\{dp[c], dp[c - coins[i - 1]] + 1\} & c \geq coins[i - 1] \end{cases}$$

1. 当 $c < coins[i - 1]$ 时:

i. 不使用第 $i - 1$ 枚硬币, 只使用前 $i - 1$ 枚硬币凑成金额 w 的最少硬币数量, 即 $dp[c]$ 。

2. 当 $c \geq coins[i - 1]$ 时, 取下面两种情况中的较小值:

i. 不使用第 $i - 1$ 枚硬币, 只使用前 $i - 1$ 枚硬币凑成金额 w 的最少硬币数量, 即 $dp[c]$ 。

ii. 凑成金额 $c - coins[i - 1]$ 的最少硬币数量, 再加上当前硬币的数量 1, 即 $dp[c - coins[i - 1]] + 1$ 。

4. 初始条件

- 凑成总金额为 0 的最少硬币数量为 0, 即 $dp[0] = 0$ 。

- 默认情况下, 在不使用硬币时, 都不能恰好凑成总金额为 w , 此时将状态值设置为一个极大值 (比如 $n + 1$), 表示无法凑成。

5. 最终结果

根据我们之前定义的状态, $dp[c]$ 表示为: 凑成总金额为 c 的最少硬币数量。则最终结果为 $dp[amount]$ 。

1. 如果 $dp[amount] \neq amount + 1$, 则说明: $dp[amount]$ 为凑成金额 $amount$ 的最少硬币数量, 则返回 $dp[amount]$ 。

2. 如果 $dp[amount] = amount + 1$, 则说明: 无法凑成金额 $amount$, 则返回 -1 。

思路 2: 代码

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        size = len(coins)
        dp = [(amount + 1) for _ in range(amount + 1)]
        dp[0] = 0

        # 枚举前 i 种物品
        for i in range(1, size + 1):
            # 正序枚举背包装载重量
            for c in range(coins[i - 1], amount + 1):
                dp[c] = min(dp[c], dp[c - coins[i - 1]] + 1)

        if dp[amount] != amount + 1:
            return dp[amount]
        return -1
```

思路 2: 复杂度分析

- 时间复杂度: $O(amount \times size)$ 。其中 $amount$ 表示总金额, $size$ 表示硬币的种类数。
- 空间复杂度: $O(amount)$ 。# 0323. 无向图中连通分量的数目
- 标签: 深度优先搜索、广度优先搜索、并查集、图
- 难度: 中等

题目链接

- 0323. 无向图中连通分量的数目 - 力扣

题目大意

描述: 给定 n 个节点 (编号从 0 到 $n - 1$) 的图的无向边列表 $edges$, 其中 $edges[i] = [u, v]$ 表示节点 u 和节点 v 之间有一条无向边。

要求: 计算该无向图中连通分量的数量。

说明:

- $1 \leq n \leq 2000$ 。
- $1 \leq edges.length \leq 5000$ 。

- $\text{edges}[i].length == 2$ 。
- $0 \leq ai \leq bi < n$ 。
- $ai! = bi$ 。
- `edges` 中不会出现重复的边。

示例：

- 示例 1：

```
输入: n = 5 和 edges = [[0, 1], [1, 2], [3, 4]]
0      3
|      |
1 --- 2      4
输出: 2
```

- 示例 2：

```
输入: n = 5 和 edges = [[0, 1], [1, 2], [2, 3], [3, 4]]
0      4
|      |
1 --- 2 --- 3
输出: 1
```

解题思路

先来看一下图论中相关的名次解释。

- **连通图**：在无向图中，如果可以从顶点 v_i 到达 v_j ，则称 v_i 和 v_j 连通。如果图中任意两个顶点之间都连通，则称该图为连通图。
- **无向图的连通分量**：如果该图为连通图，则连通分量为本身；否则将无向图中的极大连通子图称为连通分量，每个连通分量都是一个连通图。
- **无向图的连通分量个数**：无向图的极大连通子图的个数。

接下来我们来解决这道题。

思路 1：深度优先搜索

1. 使用 `visited` 数组标记遍历过的节点，使用 `count` 记录连通分量数量。
2. 从未遍历过的节点 u 出发，连通分量数量加 1。然后遍历与 u 节点构成无向边，且为遍历过的的节点 v 。
3. 再从 v 出发继续深度遍历。
4. 直到遍历完与 u 直接相关、间接相关的节点之后，再遍历另一个未遍历过的节点，继续上述操作。
5. 最后输出连通分量数目。

思路 1：代码

```

class Solution:
    def dfs(self, visited, i, graph):
        visited[i] = True
        for j in graph[i]:
            if not visited[j]:
                self.dfs(visited, j, graph)

    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        count = 0
        visited = [False for _ in range(n)]
        graph = [[] for _ in range(n)]

        for x, y in edges:
            graph[x].append(y)
            graph[y].append(x)

        for i in range(n):
            if not visited[i]:
                count += 1
                self.dfs(visited, i, graph)
        return count

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 是顶点个数。
- 空间复杂度: $O(n)$ 。

思路 2：广度优先搜索

- 使用变量 `count` 记录连通分量个数。使用集合变量 `visited` 记录访问过的节点，使用邻接表 `graph` 记录图结构。
- 从 `0` 开始，依次遍历 `n` 个节点。
- 如果第 `i` 个节点未访问过：
 - 将其添加到 `visited` 中。
 - 并且连通分量个数累加，即 `count += 1`。
 - 定义一个队列 `queue`，将第 `i` 个节点加入到队列中。
 - 从队列中取出第一个节点，遍历与其链接的节点，并将未遍历过的节点加入到队列 `queue` 和 `visited` 中。
 - 直到队列为空，则继续向后遍历。
- 最后输出连通分量数目 `count`。

思路 2：代码

```

import collections

class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        count = 0
        visited = set()
        graph = [[] for _ in range(n)]

        for x, y in edges:
            graph[x].append(y)
            graph[y].append(x)

        for i in range(n):
            if i not in visited:
                visited.add(i)
                count += 1
                queue = collections.deque([i])
                while queue:
                    node_u = queue.popleft()
                    for node_v in graph[node_u]:
                        if node_v not in visited:
                            visited.add(node_v)
                            queue.append(node_v)
        return count

```

思路 2：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 是顶点个数。
- 空间复杂度: $O(n)$ 。

0324. 摆动排序 II

- 标签: 数组、分治、快速选择、排序
- 难度: 中等

题目链接

- [0324. 摆动排序 II - 力扣](#)

题目大意

给你一个整数数组 `nums`。

要求: 将它重新排列成 `nums[0] < nums[1] > nums[2] < nums[3] ...` 的顺序。可以假设所有输入数组都可以得到满足题目要求的结果。

注意:

- $1 \leq \text{nums.length} \leq 5 * 10^4$ 。
- $0 \leq \text{nums}[i] \leq 5000$ 。

解题思路

`num[i]` 的取值在 `[0, 5000]`。所以我们可以用桶排序算法将排序算法的时间复杂度降到 $O(n)$ 。然后按照下标的奇偶性遍历两次数组，第一次遍历将桶中的元素从末尾到头部依次放到对应奇数位置上。第二次遍历将桶中剩余元素从末尾到头部依次放到对应偶数位置上。

代码

```

class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """

        buckets = [0 for _ in range(5010)]
        for num in nums:
            buckets[num] += 1

        size = len(nums)
        big = size - 2 if (size & 1) == 1 else size - 1
        small = size - 1 if (size & 1) == 1 else size - 2

        index = 5000
        for i in range(1, big + 1, 2):
            while buckets[index] == 0:
                index -= 1
            nums[i] = index
            buckets[index] -= 1
        for i in range(0, small + 1, 2):
            while buckets[index] == 0:
                index -= 1
            nums[i] = index
            buckets[index] -= 1

```

0326. 3 的幂

- 标签: 递归、数学
- 难度: 简单

题目链接

- [0326. 3 的幂 - 力扣](#)

题目大意

给定一个整数 n ，判断 n 是否是 3 的幂次方。 $-2^{31} \leq n \leq 2^{31} - 1$

解题思路

首先排除负数，因为 3 的幂次方不可能为负数。

因为 n 的最大值为 $2^{31} - 1$ 。计算出在 n 的范围内，3 的幂次方最大为 $3^{19} = 1162261467$ 。

3 为质数，则 3^{19} 的除数只有 $3^0, 3^1, \dots, 3^{19}$ 。所以若 n 为 3 的幂次方，则 n 肯定能被 3^{19} 整除，直接判断即可。

代码

```

class Solution:
    def isPowerOfThree(self, n: int) -> bool:
        if n <= 0:
            return False
        if (3 ** 19) % n == 0:
            return True
        return False

```

0328. 奇偶链表

- 标签：链表
- 难度：中等

题目链接

- 0328. 奇偶链表 - 力扣

题目大意

描述：给定一个单链表的头节点 `head`。

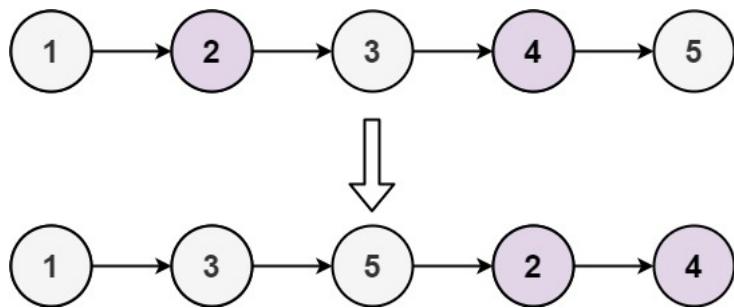
要求：将链表中的奇数位置上的节点排在前面，偶数位置上的节点排在后面，返回新的链表节点。

说明：

- 要求空间复杂度为 $O(1)$ 。
- n 等于链表中的节点数。
- $0 \leq n \leq 10^4$ 。
- $-10^6 \leq \text{Node.val} \leq 10^6$ 。

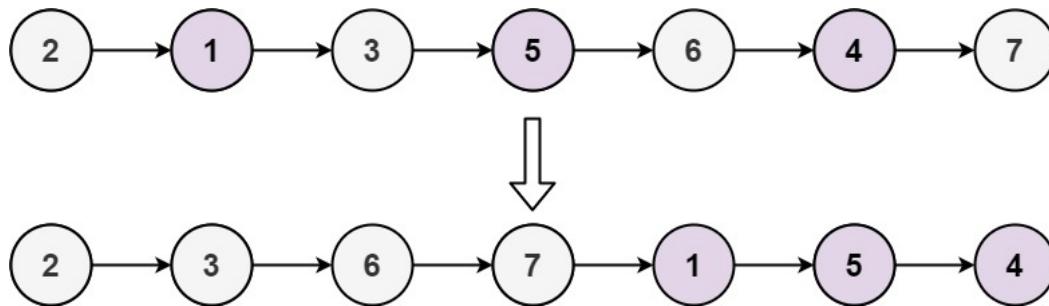
示例：

- **示例 1：**



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

- **示例 2：**



输入：`head = [2, 1, 3, 5, 6, 4, 7]`
输出：`[2, 3, 6, 7, 1, 5, 4]`

解题思路

思路 1：拆分后合并

1. 使用两个指针 `odd`、`even` 分别表示奇数节点链表和偶数节点链表。
2. 先将奇数位置上的节点和偶数位置上的节点分成两个链表，再将偶数节点的链表接到奇数链表末尾。
3. 过程中需要使用几个必要指针用于保留必要位置（比如原链表初始位置、偶数链表初始位置、当前遍历节点位置）。

思路 1：代码

```
class Solution:
    def oddEvenList(self, head: ListNode) -> ListNode:
        if not head or not head.next or not head.next.next:
            return head

        evenHead = head.next
        odd, even = head, evenHead
        isOdd = True

        curr = head.next.next

        while curr:
            if isOdd:
                odd.next = curr
                odd = curr
            else:
                even.next = curr
                even = curr
            isOdd = not isOdd
            curr = curr.next
        odd.next = evenHead
        even.next = None
        return head
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0329. 矩阵中的最长递增路径

- 标签：深度优先搜索、广度优先搜索、图、拓扑排序、记忆化搜索、数组、动态规划、矩阵
- 难度：困难

题目链接

- [0329. 矩阵中的最长递增路径 - 力扣](#)

题目大意

给定一个 $m * n$ 大小的整数矩阵 `matrix`。要求：找出其中最长递增路径的长度。

对于每个单元格，可以往上、下、左、右四个方向移动，不能向对角线方向移动或移动到边界外。

解题思路

深度优先搜索。使用二维数组 `record` 存储遍历过的单元格最大路径长度，已经遍历过的单元格就不需要再次遍历了。

代码

```

class Solution:
    max_len = 0
    directions = {(1, 0), (-1, 0), (0, 1), (0, -1)}

    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        if not matrix:
            return 0
        rows, cols = len(matrix), len(matrix[0])
        record = [[0 for _ in range(cols)] for _ in range(rows)]

        def dfs(i, j):
            record[i][j] = 1
            for direction in self.directions:
                new_i, new_j = i + direction[0], j + direction[1]
                if 0 <= new_i < rows and 0 <= new_j < cols and matrix[new_i][new_j] > matrix[i][j]:
                    if record[new_i][new_j] == 0:
                        dfs(new_i, new_j)
                    record[i][j] = max(record[i][j], record[new_i][new_j] + 1)
            self.max_len = max(self.max_len, record[i][j])

        for i in range(rows):
            for j in range(cols):
                if record[i][j] == 0:
                    dfs(i, j)
        return self.max_len

```

0334. 递增的三元子序列

- 标签：贪心、数组
- 难度：中等

题目链接

- 0334. 递增的三元子序列 - 力扣

题目大意

描述：给定一个整数数组 $nums$ 。

要求：判断数组中是否存在长度为 3 的递增子序列。

说明：

- 要求算法时间复杂度为 $O(n)$ 、空间复杂度为 $O(1)$ 。
- **长度为 3 的递增子序列：**存在这样的三元组下标 (i, j, k) 且满足 $i < j < k$ ，使得 $nums[i] < nums[j] < nums[k]$ 。
- $1 \leq nums.length \leq 5 \times 10^5$ 。
- $-2^{31} \leq nums[i] \leq 2^{31} - 1$ 。

示例：

- 示例 1：

```

输入: nums = [1,2,3,4,5]
输出: true
解释: 任何 i < j < k 的三元组都满足题意

```

- 示例 2：

输入: `nums = [5, 4, 3, 2, 1]`
 输出: `false`
 解释: 不存在满足题意的三元组

解题思路

思路 1：快慢指针

常规方法是三重 `for` 循环遍历三个数，但是时间复杂度为 $O(n^3)$ ，肯定会超时的。

那么如何才能只进行一次遍历，就找到长度为 3 的递增子序列呢？

假设长度为 3 的递增子序列元素为 a 、 b 、 c , $a < b < c$ 。

先来考虑 a 和 b 。如果我们要使得一个数组 $i < j$, 并且 $nums[i] < nums[j]$ 。那么应该使得 a 尽可能的小，这样子我们下一个数字 b 才可以尽可能地满足条件。

同样对于 b 和 c ，也应该使得 b 尽可能的小，下一个数字 c 才可以尽可能的满足条件。

所以，我们的目的是：在 $a < b$ 的前提下，保证 a 尽可能小。在 $b < c$ 的条件下，保证 b 尽可能小。

我们可以使用两个数 a 、 b 指向无穷大。遍历数组：

- 如果当前数字小于等于 a ，则更新 `a = num`；
- 如果当前数字大于等于 a ，则说明当前数满足 $num > a$ ，则判断：
 - 如果 $num \leq b$ ，则更新 `b = num`；
 - 如果 $num > b$ ，则说明找到了长度为 3 的递增子序列，直接输出 `True`。
- 如果遍历完仍未找到，则输出 `False`。

思路 1：代码

```
class Solution:
    def increasingTriplet(self, nums: List[int]) -> bool:
        a = float('inf')
        b = float('inf')
        for num in nums:
            if num <= a:
                a = num
            elif num <= b:
                b = num
            else:
                return True
        return False
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。

0336. 回文对

- 标签：字典树、数组、哈希表、字符串
- 难度：困难

题目链接

- 0336. 回文对 - 力扣

题目大意

给定一组互不相同的单词列表 `words`。

要求：找出所有不同的索引对 `(i, j)`，使得列表中的两个单词 `words[i] + words[j]`，可拼接成回文串。

解题思路

如果字符串 `words[i] + words[j]` 能构成一个回文串，把 `words[i]` 分成 `words_left[i]` 和 `words_right[i]` 两部分。即 `words[i] + words[j] = words_left[i] + words_right[i] + words[j]`。则：

- `words_right[i]` 本身是回文串，`words_left[i]` 和 `words[j]` 互为逆序。

同理，如果 `words[j] + word[i]` 能构成一个回文串，把 `word[i]` 分成 `words_left[i]` 和 `words_right[i]` 两部分。即 `words[j] + word[i] = words[j] + words_left[i] + words_right[i]`。则：

- `words_left[i]` 本身是回文串，`words[j]` 和 `words_right[i]` 互为逆序。

从上面的表述可以得知，`words[j]` 可以通过拆分 `words[i]` 之后逆序得出。

我们使用两重循环遍历。一重循环遍历单词列表 `words` 中的每一个单词 `words[i]`，二重循环遍历每个单词的拆分位置 `j`。然后将每一个单词 `words[i]` 拆分成 `words[i][0:j+1]` 和 `words[i][j+1:]`。然后分别判断 `words[i][0:j+1]` 的逆序和 `words[i][j+1:]` 的逆序是否在单词列表中，如果在单词列表中，则将「`words[i]` 和 `words[i][0:j+1]` 对应的索引」或者「`words[i]` 和 `words[i][j+1:]` 对应的索引」插入到答案数组中。

至于判断 `words[i][0:j+1]` 的逆序和 `words[i][j+1:]` 的逆序是否在单词列表中，以及获取 `words[i][0:j+1]` 的逆序和 `words[i][j+1:]` 的逆序所对应单词的索引下标可以通过构建字典树的方式获取。

代码

```

class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.children = dict()
        self.isEnd = False
        self.index = -1

    def insert(self, word: str, index: int) -> None:
        """
        Inserts a word into the trie.
        """
        cur = self
        for ch in word:
            if ch not in cur.children:
                cur.children[ch] = Trie()
            cur = cur.children[ch]
        cur.isEnd = True
        cur.index = index

    def search(self, word: str) -> int:
        """
        Returns if the word is in the trie.
        """
        cur = self
        for ch in word:
            if ch not in cur.children:
                return -1
            cur = cur.children[ch]

        if cur is not None and cur.isEnd:
            return cur.index
        return -1

class Solution:

    def isPalindrome(self, word: str) -> bool:
        left, right = 0, len(word) - 1
        while left < right:
            if word[left] != word[right]:
                return False
            left += 1
            right -= 1
        return True

    def palindromePairs(self, words: List[str]) -> List[List[int]]:
        trie_tree = Trie()
        size = len(words)
        for i in range(size):
            word = words[i]
            trie_tree.insert(word, i)

        res = []
        for i in range(size):
            word = words[i]
            for j in range(len(word)):
                if self.isPalindrome(word[:j+1]):
                    temp = word[j+1:][::-1]
                    index = trie_tree.search(temp)
                    if index != i and index != -1:

```

```

        res.append([index, i])
        if temp == "":
            res.append([i, index])
        if self.isPalindrome(word[j+1:]):
            temp = word[:j+1][::-1]
            index = trie_tree.search(temp)
            if index != i and index != -1:
                res.append([i, index])
    return res

```

0337. 打家劫舍 III

- 标签：树、深度优先搜索、动态规划、二叉树
- 难度：中等

题目链接

- [0337. 打家劫舍 III - 力扣](#)

题目大意

小偷发现了一个新的可行窃的地区，这个地区的形状是一棵二叉树。这个地区只有一个入口，称为「根」。除了「根」之外，每栋房子只有一个「父」房子与之相连。如果两个直接相连的房子在同一天被打劫，房屋将自动报警。

现在给定这个代表地区房间的二叉树，每个节点值代表该房间所拥有的金额。要求计算在不触动警报的情况下，小偷一晚上能盗取的最高金额。

解题思路

树形动态规划问题。

对于当前节点 `cur`，不能选择子节点，也不能选择父节点。所以对于一棵子树来说，有两种情况：

- 选择了根节点
- 没有选择根节点

1. 选择根节点

如果选择了根节点，则不能再选择左右儿子节点，这种情况下最大值为：当前节点 + 左子树不选择根节点 + 右子树不选择根节点。

2. 不选择根节点

如果不选择根节点，则可以选择左右儿子节点，共四种可能：

- 左子树选择根节点 + 右子树选择根节点
- 左子树选择根节点 + 右子树不选根节点
- 左子树不选根节点 + 右子树选择根节点
- 左子树不选根节点 + 右子树不选根节点

选择其中最大值。

上述描述中，当前节点的选择来自于子节点信息的选择，然后逐层向上，直到根节点。所以我们使用「后序遍历」的方式进行递归遍历。

代码

```

class Solution:
    def dfs(self, root: TreeNode):
        if not root:
            return [0, 0]
        left = self.dfs(root.left)
        right = self.dfs(root.right)

        val_stole = root.val + left[1] + right[1]
        val_no_stole = max(left[0], left[1]) + max(right[0], right[1])
        return [val_stole, val_no_stole]
    def rob(self, root: TreeNode) -> int:
        res = self.dfs(root)
        return max(res[0], res[1])

```

0338. 比特位计数

- 标签：位运算、动态规划
- 难度：简单

题目链接

- [0338. 比特位计数 - 力扣](#)

题目大意

描述：给定一个整数 n 。

要求：对于 $0 \leq i \leq n$ 的每一个 i ，计算其二进制表示中 1 的个数，返回一个长度为 $n + 1$ 的数组 ans 作为答案。

说明：

- $0 \leq n \leq 10^5$ 。
- 使用线性时间复杂度 $O(n)$ 解决此问题。
- 不使用任何内置函数解决此问题。

示例：

- **示例 1：**

```

输入: n = 5
输出: [0,1,1,2,1,2]
解释:
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101

```

解题思路

思路 1：动态规划

根据整数的二进制特点可以将整数分为两类：

- 奇数：其二进制表示中 1 的个数一定比前面相邻的偶数多一个 1 。

- 偶数：其二进制表示中 1 的个数一定与该数除以 2 之后的数一样多。

另外，边界 0 的二进制表示中 1 的个数为 0。

于是可以根据规律，从 0 开始到 n 进行递推求解。

1. 划分阶段

按照整数 n 进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：整数 i 对应二进制表示中 1 的个数。

3. 状态转移方程

- 如果 i 为奇数，则整数 i 对应二进制表示中 1 的个数等于整数 $i - 1$ 对应二进制表示中 1 的个数加 1，即 $dp[i] = dp[i - 1] + 1$ 。
- 如果 i 为偶数，则整数 i 对应二进制表示中 1 的个数等于整数 $i // 2$ 对应二进制表示中 1 的个数，即 $dp[i] = dp[i // 2]$ 。

4. 初始条件

整数 0 对应二进制表示中 1 的个数为 0。

5. 最终结果

整个 dp 数组即为最终结果，将其返回即可。

思路 1：动态规划代码

```
class Solution:
    def countBits(self, n: int) -> List[int]:
        dp = [0 for _ in range(n + 1)]
        for i in range(1, n + 1):
            if i % 2 == 1:
                dp[i] = dp[i - 1] + 1
            else:
                dp[i] = dp[i // 2]
        return dp
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。一重循环的时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(n)$ 。用到了一位数组保存状态，所以总的时间复杂度为 $O(n)$ 。

0340. 至多包含 K 个不同字符的最长子串

- 标签：哈希表、字符串、滑动窗口
- 难度：中等

题目链接

- [0340. 至多包含 K 个不同字符的最长子串 - 力扣](#)

题目大意

给定一个字符串 s ，

要求：返回至多包含 k 个不同字符的最长子串 t 的长度。

解题思路

用滑动窗口 `window_counts` 来记录各个字符个数，`window_counts` 为哈希表类型。用 `ans` 来维护至多包含 `k` 个不同字符的最长子串 `t` 的长度。

设定两个指针：`left`、`right`，分别指向滑动窗口的左右边界，保证窗口中不超过 `k` 种字符。

- 一开始，`left`、`right` 都指向 `0`。
- 将最右侧字符 `s[right]` 加入当前窗口 `window_counts` 中，记录该字符个数，向右移动 `right`。
- 如果该窗口中字符的种数多于 `k` 个，即 `len(window_counts) > k`，则不断右移 `left`，缩小滑动窗口长度，并更新窗口中对应字符的个数，直到 `len(window_counts) <= k`。
- 维护更新至多包含 `k` 个不同字符的最长子串 `t` 的长度。然后继续右移 `right`，直到 `right >= len(s)` 结束。
- 输出答案 `ans`。

代码

```
class Solution:
    def lengthOfLongestSubstringKDistinct(self, s: str, k: int) -> int:
        ans = 0
        window_counts = dict()
        left, right = 0, 0

        while right < len(s):
            if s[right] in window_counts:
                window_counts[s[right]] += 1
            else:
                window_counts[s[right]] = 1

            while len(window_counts) > k:
                window_counts[s[left]] -= 1
                if window_counts[s[left]] == 0:
                    del window_counts[s[left]]
                left += 1
            ans = max(ans, right - left + 1)
            right += 1

        return ans
```

0341. 扁平化嵌套列表迭代器

- 标签：栈、树、深度优先搜索、设计、队列、迭代器
- 难度：中等

题目链接

- [0341. 扁平化嵌套列表迭代器 - 力扣](#)

题目大意

给定一个嵌套的整数列表 `nestedList`。列表中元素类型为 `NestedInteger` 类。每个元素（`NestedInteger` 对象）要么是一个整数，要么是一个列表；该列表的元素也可能是整数或者是其他列表。

`NestedInteger` 类提供了三个方法：

- `isInteger()`，判断当前存储的对象是否为 `int`；
- `getInteger()`，如果当前存储的元素是 `int` 型的，那么返回当前的结果 `int`，否则调用会失败；
- `getList()`，如果当前存储的元素是 `List<NestedInteger>` 型的，那么返回该 `List`，否则调用会失败。

要求：实现一个迭代器将其扁平化，使之能够遍历这个列表中的所有整数。

实现扁平迭代器类 NestedIterator:

- `NestedIterator(List<NestedInteger> nestedList)` 用嵌套列表 `nestedList` 初始化迭代器。
- `int next()` 返回嵌套列表的下一个整数。
- `boolean hasNext()` 如果仍然存在待迭代的整数, 返回 `True`; 否则, 返回 `False`。

解题思路

初始化时不对元素进行预处理。而是将所有的 `NestedInteger` 逆序放到栈中, 当需要展开的时候才进行展开。

代码

```
class NestedIterator:
    def __init__(self, nestedList: [NestedInteger]):
        self.stack = []
        size = len(nestedList)
        for i in range(size - 1, -1, -1):
            self.stack.append(nestedList[i])

    def next(self) -> int:
        cur = self.stack.pop()
        return cur.getInteger()

    def hasNext(self) -> bool:
        while self.stack:
            cur = self.stack[-1]
            if cur.isInteger():
                return True
            self.stack.pop()
            for i in range(len(cur.getList()) - 1, -1, -1):
                self.stack.append(cur.getList()[i])
        return False
```

0342. 4的幂

- 标签: 位运算、递归、数学
- 难度: 简单

题目链接

- [0342. 4的幂 - 力扣](#)

题目大意

给定一个整数 n , 判断 n 是否是 4 的幂次方, 如果是的话, 返回 `True`。不是的话, 返回 `False`。

解题思路

通过循环可以直接做。但有更好的方法。

n 如果是 4 的幂次方, 那么 n 肯定是 2 的幂次方, 2 的幂次方二进制表示只含有一个 1, 可以通过 $n \& (n - 1)$ 将 n 的最后位置上的 1 置为 0, 通过判断 n 是否满足 $n \& (n - 1) == 0$ 来判断 n 是否是 2 的幂次方。

若根据上述判断, 得出 n 是 2 的幂次方, 则可以写为: $n = x^{2k}$ 或者 $n = x^{2k+1}$ 。如果 n 是 4 的幂次方, 则 $n = 2^k$ 。

下面来看一下 2^{2x} 、 $2^{2x} + 1$ 的情况：

- $(2^{2x} \bmod 3) = (4^x \bmod 3) = ((3+1)^x \bmod 3) == 1$
- $(2^{2x+1} \bmod 3) = ((2 \times 4^x) \bmod 3) = ((2 \times (3+1)^x) \bmod 3) == 2$

则如果 $n \bmod 3 == 1$, 则 n 为 4 的幂次方。

代码

```
class Solution:
    def isPowerOfFour(self, n: int) -> bool:
        return n > 0 and (n & (n-1)) == 0 and (n-1) % 3 == 0
```

0343. 整数拆分

- 标签：数学、动态规划
- 难度：中等

题目链接

- [0343. 整数拆分 - 力扣](#)

题目大意

描述：给定一个正整数 n , 将其拆分为 $k(k \geq 2)$ 个正整数的和, 并使这些整数的乘积最大化。

要求：返回可以获得的最大乘积。

说明：

- $2 \leq n \leq 58$ 。

示例：

- **示例 1：**

```
输入: n = 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。
```

- **示例 2：**

```
输入: n = 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。
```

解题思路

思路 1：动态规划

1. 划分阶段

按照正整数进行划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：将正整数 i 拆分为至少 2 个正整数的和之后，这些正整数的最大乘积。

3. 状态转移方程

当 $i \geq 2$ 时, 假设正整数 i 拆分出的第 1 个正整数是 j ($1 \leq j < i$), 则有两种方法:

1. 将 i 拆分为 j 和 $i - j$ 的和, 且 $i - j$ 不再拆分为多个正整数, 此时乘积为: $j \times (i - j)$ 。
2. 将 i 拆分为 j 和 $i - j$ 的和, 且 $i - j$ 继续拆分为多个正整数, 此时乘积为: $j \times dp[i - j]$ 。

则 $dp[i]$ 取两者中的最大值。即: $dp[i] = \max(j \times (i - j), j \times dp[i - j])$ 。

由于 $1 \leq j < i$, 需要遍历 j 得到 $dp[i]$ 的最大值, 则状态转移方程如下:

$$dp[i] = \max_{1 \leq j < i} \{ \max(j \times (i - j), j \times dp[i - j]) \}.$$

4. 初始条件

- 0 和 1 都不能被拆分, 所以 $dp[0] = 0, dp[1] = 0$ 。

5. 最终结果

根据我们之前定义的状态, $dp[i]$ 表示为: 将正整数 i 拆分为至少 2 个正整数的和之后, 这些正整数的最大乘积。则最终结果为 $dp[n]$ 。

思路 1: 代码

```
class Solution:
    def integerBreak(self, n: int) -> int:
        dp = [0 for _ in range(n + 1)]
        for i in range(2, n + 1):
            for j in range(i):
                dp[i] = max(dp[i], (i - j) * j, dp[i - j] * j)
        return dp[n]
```

思路 1: 复杂度分析

- 时间复杂度: $O(n^2)$ 。
- 空间复杂度: $O(n)$ 。

0344. 反转字符串

- 标签: 双指针、字符串
- 难度: 简单

题目链接

- [0344. 反转字符串 - 力扣](#)

题目大意

描述: 给定一个字符数组 s 。

要求: 将其反转。

说明:

- 不能使用额外的数组空间, 必须原地修改输入数组、使用 $O(1)$ 的额外空间解决问题。
- $1 \leq s.length \leq 10^5$ 。
- $s[i]$ 都是 ASCII 码表中的可打印字符。

示例:

- 示例 1:

输入: `s = ["h", "e", "l", "l", "o"]`
 输出: `["o", "l", "l", "e", "h"]`

- 示例 2:

输入: `s = ["H", "a", "n", "n", "a", "h"]`
 输出: `["h", "a", "n", "n", "a", "H"]`

解题思路

思路 1：对撞指针

- 使用两个指针 `left`, `right`。`left` 指向字符数组开始位置, `right` 指向字符数组结束位置。
- 交换 `s[left]` 和 `s[right]`, 将 `left` 右移、`right` 左移。
- 如果遇到 `left == right`, 跳出循环。

思路 1：代码

```
class Solution:
    def reverseString(self, s: List[str]) -> None:
        left, right = 0, len(s) - 1
        while left < right:
            s[left], s[right] = s[right], s[left]
            left += 1
            right -= 1
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。

0345. 反转字符串中的元音字母

- 标签: 双指针、字符串
- 难度: 简单

题目链接

- [0345. 反转字符串中的元音字母 - 力扣](#)

题目大意

描述: 给定一个字符串 `s`。

要求: 将字符串中的元音字母进行反转。

说明:

- 元音字母包括 `'a'`、`'e'`、`'i'`、`'o'`、`'u'`，且可能以大小写两种形式出现不止一次。
- $1 \leq s.length \leq 3 \times 10^5$ 。
- `s` 由可打印的 ASCII 字符组成。

示例:

- 示例 1:

输入: `s = "hello"`
输出: `"holle"`

- 示例 2:

输入: `s = "leetcode"`
输出: `"leotcede"`

解题思路

思路 1：对撞指针

- 因为 Python 的字符串是不可变的，所以我们先将字符串转为数组。
- 使用两个指针 `left`, `right`。`left` 指向字符串开始位置, `right` 指向字符串结束位置。
- 然后 `left` 依次从左到右移动查找元音字母, `right` 依次从右到左查找元音字母。
- 如果都找到了元音字母，则交换字符，然后继续进行查找。
- 如果遇到 `left == right` 时停止。
- 最后返回对应的字符串即可。

思路 1：代码

```
class Solution:
    def reverseVowels(self, s: str) -> str:
        vowels = ['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']
        left = 0
        right = len(s)-1
        s_list = list(s)
        while left < right:
            if s_list[left] not in vowels:
                left += 1
                continue
            if s_list[right] not in vowels:
                right -= 1
                continue
            s_list[left], s_list[right] = s_list[right], s_list[left]
            left += 1
            right -= 1
        return ''.join(s_list)
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 为字符串 s 的长度。
- 空间复杂度: $O(1)$ 。# 0346. 数据流中的移动平均值
- 标签: 设计、队列、数组、数据流
- 难度: 简单

题目链接

- 0346. 数据流中的移动平均值 - 力扣

题目大意

给定一个整数 `val` 和一个窗口大小 `size`。

要求: 根据滑动窗口的大小, 计算滑动窗口里所有数字的平均值。要实现 `MovingAverage` 类:

- `MovingAverage(int size)` 用窗口大小 `size` 初始化对象。

- `double next(int val)` 成员函数 `next` 每次调用的时候都会往滑动窗口增加一个整数，请计算并返回数据流中最后 `size` 个值的移动平均值，即滑动窗口里所有数字的平均值。

解题思路

使用队列保存滑动窗口的元素，并记录对应窗口大小和元素和。

在小于窗口大小的时候，直接向队列中添加元素，并记录元素和。

在等于窗口大小的时候，先将队列头部元素弹出，再添加元素，并记录元素和。

然后根据元素和和队列中元素个数计算出平均值。

代码

```
class MovingAverage:

    def __init__(self, size: int):
        .....
        Initialize your data structure here.
        .....
        self.queue = []
        self.size = size
        self.sum = 0

    def next(self, val: int) -> float:
        if len(self.queue) < self.size:
            self.queue.append(val)
        else:
            if self.queue:
                self.sum -= self.queue[0]
                self.queue.pop(0)
            self.queue.append(val)
        self.sum += val
        return self.sum / len(self.queue)
```

0347. 前 K 个高频元素

- 标签：数组、哈希表、分治、桶排序、计数、快速选择、排序、堆（优先队列）
- 难度：中等

题目链接

- [0347. 前 K 个高频元素 - 力扣](#)

题目大意

描述：给定一个整数数组 `nums` 和一个整数 `k`。

要求：返回出现频率前 `k` 高的元素。可以按任意顺序返回答案。

说明：

- $1 \leq \text{nums.length} \leq 10^5$ 。
- `k` 的取值范围是 $[1, \text{数组中不相同的元素的个数}]$ 。
- 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。

示例：

- 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]
```

解题思路

思路 1：哈希表 + 优先队列

1. 使用哈希表记录下数组中各个元素的频数。
2. 然后将哈希表中的元素去重，转换为新数组。时间复杂度 $O(n)$ ，空间复杂度 $O(n)$ 。
3. 使用二叉堆构建优先队列，优先级为元素频数。此时堆顶元素即为频数最高的元素。时间复杂度 $O(n)$ ，空间复杂度 $O(n)$ 。
4. 将堆顶元素加入到答案数组中，进行出队操作。时间复杂度 $O(\log n)$ 。
 - 出队操作：交换堆顶元素与末尾元素，将末尾元素移出堆。继续调整大顶堆。
5. 不断重复第 4 步，直到 k 次结束。调整 k 次的时间复杂度 $O(n \times \log n)$ 。

思路 1：代码

```

class Heap:
    # 堆调整方法：调整为大顶堆
    def heapAdjust(self, nums: [int], nums_dict, index: int, end: int):
        left = index * 2 + 1
        right = left + 1
        while left <= end:
            # 当前节点为非叶子结点
            max_index = index
            if nums_dict[nums[left]] > nums_dict[nums[max_index]]:
                max_index = left
            if right <= end and nums_dict[nums[right]] > nums_dict[nums[max_index]]:
                max_index = right
            if index == max_index:
                # 如果不用交换，则说明已经交换结束
                break
            nums[index], nums[max_index] = nums[max_index], nums[index]
            # 继续调整子树
            index = max_index
            left = index * 2 + 1
            right = left + 1

    # 将数组构建为二叉堆
    def heapify(self, nums: [int], nums_dict):
        size = len(nums)
        # (size - 2) // 2 是最后一个非叶节点，叶节点不用调整
        for i in range((size - 2) // 2, -1, -1):
            # 调用调整堆函数
            self.heapAdjust(nums, nums_dict, i, size - 1)

    # 入队操作
    def heappush(self, nums: list, nums_dict, value):
        nums.append(value)
        size = len(nums)
        i = size - 1
        # 寻找插入位置
        while (i - 1) // 2 >= 0:
            cur_root = (i - 1) // 2
            # value 小于当前根节点，则插入到当前位置
            if nums_dict[nums[cur_root]] > nums_dict[value]:
                break
            # 继续向上查找
            nums[i] = nums[cur_root]
            i = cur_root
        # 找到插入位置或者到达根位置，将其插入
        nums[i] = value

    # 出队操作
    def heappop(self, nums: list, nums_dict) -> int:
        size = len(nums)
        nums[0], nums[-1] = nums[-1], nums[0]
        # 得到最大值（堆顶元素）然后调整堆
        top = nums.pop()
        if size > 0:
            self.heapAdjust(nums, nums_dict, 0, size - 2)

        return top

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # 统计元素频数
        nums_dict = dict()
        for num in nums:

```

```

if num in nums_dict:
    nums_dict[num] += 1
else:
    nums_dict[num] = 1

# 使用 set 方法去重，得到新数组
new_nums = list(set(nums))
size = len(new_nums)

heap = Heapp()
queue = []
for num in new_nums:
    heap.heappush(queue, nums_dict, num)

res = []
for i in range(k):
    res.append(heap.heappop(queue, nums_dict))
return res

```

思路 1：复杂度分析

- 时间复杂度: $O(n \times \log n)$ 。
- 空间复杂度: $O(n)$ 。# 0349. 两个数组的交集
- 标签: 数组、哈希表、双指针、二分查找、排序
- 难度: 简单

题目链接

- [0349. 两个数组的交集 - 力扣](#)

题目大意

描述: 给定两个数组 $nums1$ 和 $nums2$ 。

要求: 返回两个数组的交集。重复元素只计算一次。

说明:

- $1 \leq nums1.length, nums2.length \leq 1000$ 。
- $0 \leq nums1[i], nums2[i] \leq 1000$ 。

示例:

- 示例 1:

```

输入: nums1 = [1,2,2,1], nums2 = [2,2]
输出: [2]
示例 2:

```

- 示例 2:

```

输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
输出: [9,4]
解释: [4,9] 也是可通过的

```

解题思路

思路 1：哈希表

1. 先遍历第一个数组，利用哈希表来存放第一个数组的元素，对应字典值设为 1。
2. 然后遍历第二个数组，如果哈希表中存在该元素，则将该元素加入到答案数组中，并且将该键值清空。

思路 1：代码

```
class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        numDict = dict()
        nums = []
        for num in nums1:
            if num not in numDict:
                numDict[num] = 1
        for num in nums2:
            if num in numDict and numDict[num] != 0:
                numDict[num] -= 1
                nums.append(num)
        return nums
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

思路 2：分离双指针

1. 对数组 $nums1$ 、 $nums2$ 先排序。
2. 使用两个指针 $left_1$ 、 $left_2$ 。 $left_1$ 指向第一个数组的第一个元素，即: $left_1 = 0$, $left_2$ 指向第二个数组的第一个元素，即: $left_2 = 0$ 。
3. 如果 $nums1[left_1]$ 等于 $nums2[left_2]$ ，则将其加入答案数组（注意去重），并将 $left_1$ 和 $left_2$ 右移。
4. 如果 $nums1[left_1]$ 小于 $nums2[left_2]$ ，则将 $left_1$ 右移。
5. 如果 $nums1[left_1]$ 大于 $nums2[left_2]$ ，则将 $left_2$ 右移。
6. 最后返回答案数组。

思路 2：代码

```
class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        nums1.sort()
        nums2.sort()

        left_1 = 0
        left_2 = 0
        res = []
        while left_1 < len(nums1) and left_2 < len(nums2):
            if nums1[left_1] == nums2[left_2]:
                if nums1[left_1] not in res:
                    res.append(nums1[left_1])
                left_1 += 1
                left_2 += 1
            elif nums1[left_1] < nums2[left_2]:
                left_1 += 1
            elif nums1[left_1] > nums2[left_2]:
                left_2 += 1
        return res
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0350. 两个数组的交集 II

- 标签：数组、哈希表
- 难度：简单

题目链接

- [0350. 两个数组的交集 II - 力扣](#)

题目大意

描述：给定两个数组 $nums1$ 和 $nums2$ 。

要求：返回两个数组的交集。可以不考虑输出结果的顺序。

说明：

- 输出结果中，每个元素出现的次数，应该与元素在两个数组中都出现的次数一致（如果出现次数不一致，则考虑取较小值）。
- $1 \leq nums1.length, nums2.length \leq 1000$ 。
- $0 \leq nums1[i], nums2[i] \leq 1000$ 。

示例：

输入：`nums1 = [1,2,2,1]`, `nums2 = [2,2]`

输出：`[2,2]`

输入：`nums1 = [4,9,5]`, `nums2 = [9,4,9,8,4]`

输出：`[4,9]`

解题思路

思路 1：哈希表

- 先遍历第一个数组，利用字典来存放第一个数组的元素出现次数。
- 然后遍历第二个数组，如果字典中存在该元素，则将该元素加入到答案数组中，并减少字典中该元素出现的次数。
- 遍历完之后，返回答案数组。

思路 1：代码

```
class Solution:
    def intersect(self, nums1: List[int], nums2: List[int]) -> List[int]:
        numDict = dict()
        nums = []
        for num in nums1:
            if num in numDict:
                numDict[num] += 1
            else:
                numDict[num] = 1
        for num in nums2:
            if num in numDict and numDict[num] != 0:
                numDict[num] -= 1
                nums.append(num)
        return nums
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。# 0351. 安卓系统手势解锁
- 标签: 动态规划、回溯
- 难度: 中等

题目链接

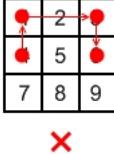
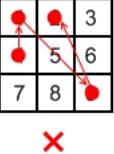
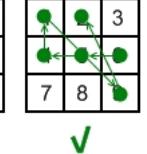
- [0351. 安卓系统手势解锁 - 力扣](#)

题目大意

描述: 安卓系统手势解锁的界面是一个编号为 1 ~ 9、大小为 3×3 的网格。用户可以设定一个「解锁模式」，按照一定顺序经过 k 个点，构成一个「解锁手势」。现在给定两个整数，分别为 m 和 n 。

要求: 计算出有多少种不同且有效的解锁模式数量，其中每种解锁模式至少需要经过 m 个点，但是不超过 n 个点。

说明:

- 有效的解锁模式:
 - 解锁模式中所有点不能重复。
 - 如果解锁模式中两个点是按顺序经过的，那么这两个点之间的手势轨迹不能跨过其他任何未被经过的点。
- 一些有效和无效解锁模式示例:
 - 
X
 - 
X
 - 
√
 - 
√
- 无效手势: [4, 1, 3, 6]，连接点 1 和点 3 时经过了未被连接过的 2 号点。
- 无效手势: [4, 1, 9, 2]，连接点 1 和点 9 时经过了未被连接过的 5 号点。
- 有效手势: [2, 4, 1, 3, 6]，连接点 1 和点 3 是有效的，因为虽然它经过了点 2，但是点 2 在该手势中之前已经被连过了。
- 有效手势: [6, 5, 4, 1, 9, 2]，连接点 1 和点 9 是有效的，因为虽然它经过了按键 5，但是点 5 在该手势中之前已经被连过了。
- $1 \leq m, n \leq 9$ 。
- 如果经过的点不同或者经过点的顺序不同，表示为不同的解锁模式。

示例:

- 示例 1:

```
输入: m = 1, n = 1
输出: 9
```

- 示例 2:

```
输入: m = 1, n = 2
输出: 65
```

解题思路

思路 1：状态压缩 + 记忆化搜索

因为手势解锁的界面是一个编号为 $1 \sim 9$ 、大小为 3×3 的网格，所以我们可以用一个 9 位长度的二进制数 $state$ 来表示当前解锁模式中按键的选取情况。

因为解锁模式中两个点之间的手势轨迹不能跨过其他任何未被经过的点，所以我们可以预先使用一个哈希表 $graph$ 将手势轨迹跨过其他点的情况存储下来，便于判断当前手势轨迹是否有效。

接下来我们使用深度优先搜索方法，将所有有效的解锁模式统计出来，具体做法如下：

1. 定义一个全局变量 ans 用于统计所有有效的解锁模式的方案数。
2. 定义一个深度优先搜索方法为 `def dfs(state, cur, step):`，表示当前键位选择情况为 $state$ ，从当前键位 cur 出发，已经走了 $step$ 的有效解锁模式。
 - i. 当 $step$ 在区间 $[m, n]$ 中时，统计有效解锁模式方案数，即：令 ans 加 1。
 - ii. 当 $step$ 到达步数上限 n 时，直接返回。
 - iii. 遍历下一步（第 $step + 1$ 步）可选择的键位 k ，判断键位 k 是否有效。
 - iv. 如果到达 k 没有跨过其他键 (k 不在 $graph[cur]$ 中)，或者到达 k 跨过的键位是已经经过的键 ($state >> graph[cur][k] \& 1 == 1$)，则继续调用 `dfs(state | (1 << k), k, step + 1)`，其中 `stete | (1 << k)` 表示下一步选择 k 的状态。
3. 遍历开始位置 $1 \sim 9$ ，从 $1 \sim 9$ 每个数字开始出发，调用 `dfs(1 << i, i, 1)`，进行所有有效的解锁模式的统计。
4. 最后输出 ans 。

思路 1：代码

```

class Solution:
    def numberOfPatterns(self, m: int, n: int) -> int:
        # 将手势轨迹跨过点的情况存入哈希表中
        graph = {
            1: {3: 2, 7: 4, 9: 5},
            2: {8: 5},
            3: {1: 2, 7: 5, 9: 6},
            4: {6: 5},
            5: {},
            6: {4: 5},
            7: {1: 4, 3: 5, 9: 8},
            8: {2: 5},
            9: {1: 5, 3: 6, 7: 8},
        }

        ans = 0

        def dfs(state, cur, step):
            nonlocal ans
            if m <= step <= n:
                ans += 1

            if step == n:
                return

            for k in range(1, 10):
                if state >> k & 1 != 0:
                    continue
                if k not in graph[cur] or state >> graph[cur][k] & 1:
                    dfs(state | (1 << k), k, step + 1)

        for i in range(1, 10):
            dfs(1 << i, i, 1) # 从 1 ~ 9 每个数字开始出发

        return ans

```

思路 1：复杂度分析

- 时间复杂度: $O(n!)$ 。
- 空间复杂度: $O(1)$ 。

参考资料

- 【题解】[LeetCode-351. 安卓系统手势解锁 - mkdocs_blog](#)

0354. 俄罗斯套娃信封问题

- 标签：数组、二分查找、动态规划、排序
- 难度：困难

题目链接

- [0354. 俄罗斯套娃信封问题 - 力扣](#)

题目大意

给定一个二维整数数组 envelopes 表示信封，其中 $envelopes[i] = [w_i, h_i]$ ，表示第 i 个信封的宽度 w_i 和高度 h_i 。

当一个信封的宽度和高度比另一个信封大时，则小的信封可以放进大信封里，就像俄罗斯套娃一样。

现在要求：计算最多能有多少个信封组成一组「俄罗斯套娃」信封。

注意：不允许旋转信封（也就是说宽高不能互换）。

解题思路

如果最多有 k 个信封可以组成「俄罗斯套娃」信封。那么这 k 个信封按照宽高关系排序一定满足：

- $w_0 < w_1 < \dots < w_{k-1}$
- $h_0 < h_1 < \dots < h_{k-1}$

因为原二维数组是无序的，直接暴力搜索宽高升序序列不容易。所以我们可以先固定一个维度，将其变为升序状态。再在另一个维度上进行选择。比如固定宽度为升序，则我们的问题就变为了：在高度这一维度下，求解数组的最长递增序列的长度。就变为了经典的「最长递增序列的长度问题」。即 [0300. 最长递增子序列](#)。

「最长递增序列的长度问题」的思路如下：

动态规划的状态 `dp[i]` 表示为：以第 i 个数字结尾的前 i 个元素中最长严格递增子序列的长度。

遍历前 i 个数字， $0 \leq j \leq i$ ：

- 当 $\text{nums}[j] < \text{nums}[i]$ 时， $\text{nums}[i]$ 可以接在 $\text{nums}[j]$ 后面，此时以第 i 个数字结尾的最长严格递增子序列长度 + 1，即 $\text{dp}[i] = \text{dp}[j] + 1$ 。
- 当 $\text{nums}[j] \geq \text{nums}[i]$ 时，可以直接跳过。

则状态转移方程为： $\text{dp}[i] = \max(\text{dp}[i], \text{dp}[j] + 1)$ ， $0 \leq j \leq i$ ， $\text{nums}[j] < \text{nums}[i]$ 。

最后再遍历一遍 `dp` 数组，求出最大值即可。

代码

```
class Solution:
    def maxEnvelopes(self, envelopes: List[List[int]]) -> int:
        if not envelopes:
            return 0
        size = len(envelopes)
        envelopes.sort(key=lambda x: (x[0], -x[1]))

        dp = [1 for _ in range(size)]

        for i in range(size):
            for j in range(i):
                if envelopes[j][1] < envelopes[i][1]:
                    dp[i] = max(dp[i], dp[j] + 1)

        return max(dp)
```

0357. 统计各位数字都不同的数字个数

- 标签：数学、动态规划、回溯
- 难度：中等

题目链接

- [0357. 统计各位数字都不同的数字个数 - 力扣](#)

题目大意

描述：给定一个整数 n 。

要求：统计并返回区间 $[0, 10^n)$ 上各位数字都不相同的数字 x 的个数。

说明：

- $0 \leq n \leq 8$ 。
- $0 \leq x < 10^n$ 。

示例：

- **示例 1：**

输入: `n = 2`

输出: `91`

解释: 答案应为除去 `11、22、33、44、55、66、77、88、99` 外, 在 `0 ≤ x < 100` 范围内的所有数字。

- **示例 2：**

输入: `n = 0`

输出: `1`

解题思路

思路 1：动态规划 + 数位 DP

题目求解区间 $[0, 10^n)$ 范围内各位数字都不相同的数字个数。则我们先将 $10^n - 1$ 转换为字符串 s , 定义递归函数

`def dfs(pos, state, isLimit, isNum):` 表示构造第 pos 位及之后所有数位的合法方案数。接下来按照如下步骤进行递归。

1. 从 `dfs(0, 0, True, False)` 开始递归。`dfs(0, 0, True, False)` 表示:

- 从位置 0 开始构造。
- 初始没有使用数字 (即前一位所选数字集合为 0)。
- 开始时受到数字 n 对应最高位数位的约束。
- 开始时没有填写数字。

2. 如果遇到 $pos == len(s)$, 表示到达数位末尾, 此时:

- 如果 `isNum == True`, 说明当前方案符合要求, 则返回方案数 1。
- 如果 `isNum == False`, 说明当前方案不符合要求, 则返回方案数 0。

3. 如果 $pos \neq len(s)$, 则定义方案数 `ans`, 令其等于 0, 即: `ans = 0`。

4. 如果遇到 `isNum == False`, 说明之前位数没有填写数字, 当前位可以跳过, 这种情况下方案数等于 $pos + 1$ 位置上没有受到 pos 位的约束, 并且之前没有填写数字时的方案数, 即: `ans = dfs(i + 1, state, False, False)`。

5. 如果 `isNum == True`, 则当前位必须填写一个数字。此时:

- 根据 `isNum` 和 `isLimit` 来决定填当前位数位所能选择的最小数字 (`minX`) 和所能选择的最大数字 (`maxX`),

- 然后根据 $[minX, maxX]$ 来枚举能够填入的数字 d 。

iii. 如果之前没有选择 d , 即 d 不在之前选择的数字集合 `state` 中, 则方案数累加上当前位选择 d 之后的方案数,

即: `ans += dfs(pos + 1, state | (1 << d), isLimit and d == maxX, True)`。

a. `state | (1 << d)` 表示之前选择的数字集合 `state` 加上 d 。

b. `isLimit and d == maxX` 表示 $pos + 1$ 位受到之前位限制和 pos 位限制。

c. `isNum == True` 表示 pos 位选择了数字。

6. 最后的方案数为 `dfs(0, 0, True, False) + 1`, 因为之前计算时没有考虑 0, 所以最后统计方案数时要加 1。

思路 1：代码

```

class Solution:
    def countNumbersWithUniqueDigits(self, n: int) -> int:
        s = str(10 ** n - 1)

        @cache
        # pos: 第 pos 个数位
        # state: 之前选过的数字集合。
        # isLimit: 表示是否受到选择限制。如果为真，则第 pos 位填入数字最多为 s[pos]；如果为假，则最大可为 9。
        # isNum: 表示 pos 前面的数位是否填了数字。如果为真，则当前位不可跳过；如果为假，则当前位可跳过。
        def dfs(pos, state, isLimit, isNum):
            if pos == len(s):
                # isNum 为 True，则表示当前方案符合要求
                return int(isNum)

            ans = 0
            if not isNum:
                # 如果 isNum 为 False，则可以跳过当前数位
                ans = dfs(pos + 1, state, False, False)

            # 如果前一位没有填写数字，则最小可选择数字为 0，否则最少为 1（不能含有前导 0）。
            minX = 0 if isNum else 1
            # 如果受到选择限制，则最大可选择数字为 s[pos]，否则最大可选择数字为 9。
            maxX = int(s[pos]) if isLimit else 9

            # 枚举可选择的数字
            for d in range(minX, maxX + 1):
                # d 不在选择的数字集合中，即之前没有选择过 d
                if (state >> d) & 1 == 0:
                    ans += dfs(pos + 1, state | (1 << d), isLimit and d == maxX, True)
            return ans

        return dfs(0, 0, True, False) + 1

```

思路 1：复杂度分析

- 时间复杂度: $O(n \times 10 \times 2^{10})$ 。
- 空间复杂度: $O(n \times 2^{10})$ 。

0359. 日志速率限制器

- 标签: 设计、哈希表
- 难度: 简单

题目链接

- [0359. 日志速率限制器 - 力扣](#)

题目大意

设计一个日志系统，可以流式接受消息和消息的时间戳。每条不重复的消息最多每 10 秒打印一次。即如果在时间 t 打印了 A 信息，则直到 $t+10$ 的时间，才能再次打印这条信息。

要求实现 Logger 类：

- `def __init__(self):`: 初始化 logger 对象
- `def shouldPrintMessage(self, timestamp: int, message: str) -> bool:`
 - 如果该条消息 message 在给定时间戳 timestamp 能够打印出来，则返回 True，否则返回 False。

解题思路

初始化一个哈希表，用来存储消息 message 最后一次打印的时间戳。

当新的消息到达时，先判断之前是否出现过相同的消息，如果未出现则可打印，存储时间戳，并返回 True。

如果出现过，且上一次相同的消息在 10 秒之前打印的，则该消息也可打印，更新时间戳，并返回 True。

如果上一次相同的消息是在 10 秒内打印的，则该信息不可打印，直接返回 False。

代码

```
class Logger:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.msg_dict = dict()

    def shouldPrintMessage(self, timestamp: int, message: str) -> bool:
        """
        Returns true if the message should be printed in the given timestamp, otherwise returns false.
        If this method returns false, the message will not be printed.
        The timestamp is in seconds granularity.
        """
        if message not in self.msg_dict:
            self.msg_dict[message] = timestamp
            return True
        if timestamp - self.msg_dict[message] >= 10:
            self.msg_dict[message] = timestamp
            return True
        else:
            return False
```

0360. 有序转化数组

- 标签：数组、数学、双指针、排序
- 难度：中等

题目链接

- [0360. 有序转化数组 - 力扣](#)

题目大意

描述：给定一个已经排好的整数数组 $nums$ 和整数 a 、 b 、 c 。

要求：对于数组中的每一个数 x ，计算函数值 $f(x) = ax^2 + bx + c$ ，请将函数值产生的数组返回。

说明：

- 返回的这个数组必须按照升序排列，并且我们所期望的解法时间复杂度为 $O(n)$ 。
- $1 \leq nums.length \leq 200$ 。
- $-100 \leq nums[i], a, b, c \leq 100$ 。
- $nums$ 按照升序排列。

示例：

- 示例 1:

```
输入: nums = [-4,-2,2,4], a = 1, b = 3, c = 5
输出: [3,9,15,33]
```

- 示例 2:

```
输入: nums = [-4,-2,2,4], a = -1, b = 3, c = 5
输出: [-23,-5,1,7]
```

解题思路

思路 1：数学 + 对撞指针

这是一道数学题。需要根据一元二次函数的性质来解决问题。因为返回的数组必须按照升序排列，并且期望的解法时间复杂度为 $O(n)$ 。这就不能先计算再排序了，而是要在线性时间复杂度内考虑问题。

我们先定义一个函数用来计算 $f(x)$ 。然后进行分情况讨论。

- 如果 $a == 0$, 说明函数是一条直线。则根据 b 值的正负来确定数组遍历顺序。
 - 如果 $b \geq 0$, 说明这条直线是一条递增直线。则按照从头到尾的顺序依次计算函数值，并依次存入答案数组。
 - 如果 $b < 0$, 说明这条直线是一条递减直线。则按照从尾到头的顺序依次计算函数值，并依次存入答案数组。
- 如果 $a > 0$, 说明函数是一条开口向上的抛物线，最小值横坐标为 $diad = \frac{-b}{2.0*a}$, 离 $diad$ 越远，函数值越大。则可以使用双指针从远到近，由大到小依次填入数组。具体步骤如下：
 - 使用双指针 $left$ 、 $right$, 令 $left$ 指向数组第一个元素位置, $right$ 指向数组最后一个元素位置。再定义 $index = len(nums) - 1$ 作为答案数组填入顺序的索引值。
 - 比较 $left - diad$ 与 $right - diad$ 的绝对值大小。大的就是目前距离 $diad$ 最远的那个。
 - 如果 $abs(nums[left] - diad)$ 更大，则将其填入答案数组对应位置，并令 $left += 1$ 。
 - 如果 $abs(nums[right] - diad)$ 更大，则将其填入答案数组对应位置，并令 $right -= 1$ 。
 - 令 $index -= 1$ 。
 - 直到 $left == right$, 最后将 $nums[left]$ 填入答案数组对应位置。
- 如果 $a < 0$, 说明函数是一条开口向下的抛物线，最大值横坐标为 $diad = \frac{-b}{2.0*a}$, 离 $diad$ 越远，函数值越小。则可以使用双指针从远到近，由小到大依次填入数组。具体步骤如下：
 - 使用双指针 $left$ 、 $right$, 令 $left$ 指向数组第一个元素位置, $right$ 指向数组最后一个元素位置。再定义 $index = 0$ 作为答案数组填入顺序的索引值。
 - 比较 $left - diad$ 与 $right - diad$ 的绝对值大小。大的就是目前距离 $diad$ 最远的那个。
 - 如果 $abs(nums[left] - diad)$ 更大，则将其填入答案数组对应位置，并令 $left += 1$ 。
 - 如果 $abs(nums[right] - diad)$ 更大，则将其填入答案数组对应位置，并令 $right -= 1$ 。
 - 令 $index += 1$ 。
 - 直到 $left == right$, 最后将 $nums[left]$ 填入答案数组对应位置。

思路 1：代码

```

class Solution:
    def calFormula(self, x, a, b, c):
        return a * x * x + b * x + c

    def sortTransformedArray(self, nums: List[int], a: int, b: int, c: int) -> List[int]:
        size = len(nums)
        res = [0 for _ in range(size)]

        # 直线
        if a == 0:
            if b >= 0:
                index = 0
                for i in range(size):
                    res[index] = self.calFormula(nums[i], a, b, c)
                    index += 1
            else:
                index = 0
                for i in range(size - 1, -1, -1):
                    res[index] = self.calFormula(nums[i], a, b, c)
                    index += 1
        else:
            diad = -(b / (2.0 * a))
            left, right = 0, size - 1

            if a > 0:
                index = size - 1
                while left < right:
                    if abs(diad - nums[left]) > abs(diad - nums[right]):
                        res[index] = self.calFormula(nums[left], a, b, c)
                        left += 1
                    else:
                        res[index] = self.calFormula(nums[right], a, b, c)
                        right -= 1
                    index -= 1
                res[index] = self.calFormula(nums[left], a, b, c)
            else:
                diad = -(b / (2.0 * a))
                left, right = 0, size - 1
                index = 0
                while left < right:
                    if abs(diad - nums[left]) > abs(diad - nums[right]):
                        res[index] = self.calFormula(nums[left], a, b, c)
                        left += 1
                    else:
                        res[index] = self.calFormula(nums[right], a, b, c)
                        right -= 1
                    index += 1
                res[index] = self.calFormula(nums[left], a, b, c)
        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$, 不考虑最终返回值的空间占用。

0367. 有效的完全平方数

- 标签: 数学、二分查找
- 难度: 简单

题目链接

- [0367. 有效的完全平方数 - 力扣](#)

题目大意

描述：给定一个正整数 num 。

要求：判断 num 是不是完全平方数。

说明：

- 要求不能使用内置的库函数，如 `sqrt`。
- $1 \leq num \leq 2^{31} - 1$ 。

示例：

- 示例 1：

```
输入: num = 16
输出: True
解释: 返回 true, 因为 4 * 4 = 16 且 4 是一个整数。
```

- 示例 2：

```
输入: num = 14
输出: False
解释: 返回 false, 因为 3.742 * 3.742 = 14 但 3.742 不是一个整数。
```

解题思路

思路 1：二分查找

如果 num 是完全平方数，则 $num = x \times x$, x 为整数。问题就变为了对于正整数 num , 是否能找到一个整数 x , 使得 $x \times x = num$ 。

而对于 x , 我们可以通过二分查找算法快速找到。

思路 1：代码

```
class Solution:
    def isPerfectSquare(self, num: int) -> bool:
        left = 0
        right = num
        while left < right:
            mid = left + (right - left) // 2
            if mid * mid > num:
                right = mid - 1
            elif mid * mid < num:
                left = mid + 1
            else:
                left = mid
                break
        return left * left == num
```

思路 1：复杂度分析

- 时间复杂度： $O(\log n)$, 其中 n 为正整数 num 的最大值。
- 空间复杂度： $O(1)$ 。

0370. 区间加法

- 标签：数组、前缀和
- 难度：中等

题目链接

- [0370. 区间加法 - 力扣](#)

题目大意

描述：给定一个数组的长度 `length`，初始情况下数组中所有数字均为 `0`。再给定 `k` 个更新操作。其中每个操作是一个三元组 `[startIndex, endIndex, inc]`，表示将子数组 `nums[startIndex ... endIndex]`（包括 `startIndex`、`endIndex`）上所有元素增加 `inc`。

要求：返回 `k` 次操作后的数组。

示例：

- **示例 1：**

```
给定 length = 5, 即 nums = [0, 0, 0, 0, 0]
```

```
操作 [1, 3, 2] -> [0, 2, 2, 2, 0]
```

```
操作 [2, 4, 3] -> [0, 2, 5, 5, 3]
```

```
操作 [0, 2, -2] -> [-2, 0, 3, 5, 3]
```

解题思路

思路 1：线段树

- 初始化一个长度为 `length`，值全为 `0` 的 `nums` 数组。
- 然后根据 `nums` 数组构建一棵线段树。每个线段树的节点类存储当前区间的左右边界和该区间的和。并且线段树使用延迟标记。
- 然后遍历三元组操作，进行区间累加运算。
- 最后从线段树中查询数组所有元素，返回该数组即可。

这样构建线段树的时间复杂度为 $O(\log n)$ ，单次区间更新的时间复杂度为 $O(\log n)$ ，单次区间查询的时间复杂度为 $O(\log n)$ 。总体时间复杂度为 $O(\log n)$ 。

思路 1：线段树代码

```

# 线段树的节点类
class SegTreeNode:
    def __init__(self, val=0):
        self.left = -1 # 区间左边界
        self.right = -1 # 区间右边界
        self.val = val # 节点值 (区间值)
        self.lazy_tag = None # 区间和问题的延迟更新标记

# 线段树类
class SegmentTree:
    # 初始化线段树接口
    def __init__(self, nums, function):
        self.size = len(nums)
        self.tree = [SegTreeNode() for _ in range(4 * self.size)] # 维护 SegTreeNode 数组
        self.nums = nums # 原始数据
        self.function = function # function 是一个函数, 左右区间的聚合方法
        if self.size > 0:
            self.__build(0, 0, self.size - 1)

    # 单点更新接口: 将 nums[i] 更改为 val
    def update_point(self, i, val):
        self.nums[i] = val
        self.__update_point(i, val, 0)

    # 区间更新接口: 将区间为 [q_left, q_right] 上的所有元素值加上 val
    def update_interval(self, q_left, q_right, val):
        self.__update_interval(q_left, q_right, val, 0)

    # 区间查询接口: 查询区间为 [q_left, q_right] 的区间值
    def query_interval(self, q_left, q_right):
        return self.__query_interval(q_left, q_right, 0)

    # 获取 nums 数组接口: 返回 nums 数组
    def get_nums(self):
        for i in range(self.size):
            self.nums[i] = self.query_interval(i, i)
        return self.nums

    # 以下为内部实现方法

    # 构建线段树实现方法: 节点的存储下标为 index, 节点的区间为 [left, right]
    def __build(self, index, left, right):
        self.tree[index].left = left
        self.tree[index].right = right
        if left == right: # 叶子节点, 节点值为对应位置的元素值
            self.tree[index].val = self.nums[left]
            return

        mid = left + (right - left) // 2 # 左右节点划分点
        left_index = index * 2 + 1 # 左子节点的存储下标
        right_index = index * 2 + 2 # 右子节点的存储下标
        self.__build(left_index, left, mid) # 递归创建左子树
        self.__build(right_index, mid + 1, right) # 递归创建右子树
        self.__pushup(index) # 向上更新节点的区间值

    # 单点更新实现方法: 将 nums[i] 更改为 val, 节点的存储下标为 index
    def __update_point(self, i, val, index):
        left = self.tree[index].left
        right = self.tree[index].right

```

```

if left == right:
    self.tree[index].val = val          # 叶子节点，节点值修改为 val
    return

mid = left + (right - left) // 2        # 左右节点划分点
left_index = index * 2 + 1              # 左子节点的存储下标
right_index = index * 2 + 2             # 右子节点的存储下标
if i <= mid:                          # 在左子树中更新节点值
    self.__update_point(i, val, left_index)
else:                                  # 在右子树中更新节点值
    self.__update_point(i, val, right_index)

self.__pushup(index)                  # 向上更新节点的区间值

# 区间更新实现方法
def __update_interval(self, q_left, q_right, val, index):
    left = self.tree[index].left
    right = self.tree[index].right

    if left >= q_left and right <= q_right:      # 节点所在区间被 [q_left, q_right] 所覆盖
        if self.tree[index].lazy_tag is not None:
            self.tree[index].lazy_tag += val        # 将当前节点的延迟标记增加 val
        else:
            self.tree[index].lazy_tag = val        # 将当前节点的延迟标记增加 val
        interval_size = (right - left + 1)         # 当前节点所在区间大小
        self.tree[index].val += val * interval_size # 当前节点所在区间每个元素值增加 val
        return

    if right < q_left or left > q_right:        # 节点所在区间与 [q_left, q_right] 无关
        return

    self.__pushdown(index)                      # 向下更新节点的区间值

    mid = left + (right - left) // 2        # 左右节点划分点
    left_index = index * 2 + 1              # 左子节点的存储下标
    right_index = index * 2 + 2             # 右子节点的存储下标
    if q_left <= mid:                      # 在左子树中更新区间值
        self.__update_interval(q_left, q_right, val, left_index)
    if q_right > mid:                      # 在右子树中更新区间值
        self.__update_interval(q_left, q_right, val, right_index)

    self.__pushup(index)                  # 向上更新节点的区间值

# 区间查询实现方法：在线段树中搜索区间为 [q_left, q_right] 的区间值
def __query_interval(self, q_left, q_right, index):
    left = self.tree[index].left
    right = self.tree[index].right

    if left >= q_left and right <= q_right:      # 节点所在区间被 [q_left, q_right] 所覆盖
        return self.tree[index].val                # 直接返回节点值
    if right < q_left or left > q_right:        # 节点所在区间与 [q_left, q_right] 无关
        return 0

    self.__pushdown(index)

    mid = left + (right - left) // 2        # 左右节点划分点
    left_index = index * 2 + 1              # 左子节点的存储下标
    right_index = index * 2 + 2             # 右子节点的存储下标
    res_left = 0                           # 左子树查询结果
    res_right = 0                          # 右子树查询结果
    if q_left <= mid:                      # 在左子树中查询
        res_left = self.__query_interval(q_left, q_right, left_index)
    if q_right > mid:                      # 在右子树中查询
        res_right = self.__query_interval(q_left, q_right, right_index)

```

```

        return self.function(res_left, res_right) # 返回左右子树元素值的聚合计算结果

# 向上更新实现方法：更新下标为 index 的节点区间值 等于 该节点左右子节点元素值的聚合计算结果
def __pushup(self, index):
    left_index = index * 2 + 1                      # 左子节点的存储下标
    right_index = index * 2 + 2                     # 右子节点的存储下标
    self.tree[index].val = self.function(self.tree[left_index].val, self.tree[right_index].val)

# 向下更新实现方法：更新下标为 index 的节点所在区间的左右子节点的值和懒惰标记
def __pushdown(self, index):
    lazy_tag = self.tree[index].lazy_tag
    if lazy_tag is None:
        return

    left_index = index * 2 + 1                      # 左子节点的存储下标
    right_index = index * 2 + 2                     # 右子节点的存储下标

    if self.tree[left_index].lazy_tag is not None:
        self.tree[left_index].lazy_tag += lazy_tag # 更新左子节点懒惰标记
    else:
        self.tree[left_index].lazy_tag = lazy_tag
    left_size = (self.tree[left_index].right - self.tree[left_index].left + 1)
    self.tree[left_index].val += lazy_tag * left_size # 左子节点每个元素值增加 lazy_tag

    if self.tree[right_index].lazy_tag is not None:
        self.tree[right_index].lazy_tag += lazy_tag # 更新右子节点懒惰标记
    else:
        self.tree[right_index].lazy_tag = lazy_tag
    right_size = (self.tree[right_index].right - self.tree[right_index].left + 1)
    self.tree[right_index].val += lazy_tag * right_size # 右子节点每个元素值增加 lazy_tag

    self.tree[index].lazy_tag = None                 # 更新当前节点的懒惰标记

class Solution:
    def getModifiedArray(self, length: int, updates: List[List[int]]) -> List[int]:
        nums = [0 for _ in range(length)]
        self.ST = SegmentTree(nums, lambda x, y: x + y)
        for update in updates:
            self.ST.update_interval(update[0], update[1], update[2])

        return self.ST.get_nums()

```

0371. 两整数之和

- 标签：位运算、数学
- 难度：中等

题目链接

- [0371. 两整数之和 - 力扣](#)

题目大意

描述：给定两个整数 a 和 b 。

要求：不使用运算符 `+` 和 `-`，计算两整数 a 和 b 的和。

说明：

- $-1000 \leq a, b \leq 1000$ 。

示例：

- 示例 1：

```
输入: a = 1, b = 2
输出: 3
```

- 示例 2：

```
输入: a = 2, b = 3
输出: 5
```

解题思路

思路 1：位运算

需要用到位运算的一些知识。

- 异或运算 `a ^ b`：可以获得 $a + b$ 无进位的加法结果。
- 与运算 `a & b`：对应位置为 1，说明 a 、 b 该位置上原来都为 1，则需要进位。
- 左移运算 `a << 1`：将 a 对应二进制数左移 1 位。

这样，通过 `a ^ b` 运算，我们可以得到相加后无进位结果，再根据 `(a & b) << 1`，计算进位后结果。

进行 `a ^ b` 和 `(a & b) << 1` 操作之后判断进位是否为 0，若不为 0，则继续上一步操作，直到进位为 0。

注意：

Python 的整数类型是无限长整数类型，负数不确定符号位是第几位。所以我们可以将输入的数字手动转为 32 位无符号整数。

通过 `a &= 0xFFFFFFFF` 即可将 a 转为 32 位无符号整数。最后通过对 a 的范围判断，将其结果映射为有符号整数。

思路 1：代码

```
class Solution:
    def getSum(self, a: int, b: int) -> int:
        MAX_INT = 0x7FFFFFFF
        MASK = 0xFFFFFFFF
        a &= MASK
        b &= MASK
        while b:
            carry = ((a & b) << 1) & MASK
            a ^= b
            b = carry
        if a <= MAX_INT:
            return a
        else:
            return ~(a ^ MASK)
```

思路 1：复杂度分析

- 时间复杂度： $O(\log k)$ ，其中 k 为 `int` 所能表达的最大整数。
- 空间复杂度： $O(1)$ 。

0374. 猜数字大小

- 标签：二分查找、交互
- 难度：简单

题目链接

- 0374. 猜数字大小 - 力扣

题目大意

描述：猜数字游戏。给定一个整数 n 和一个接口 `def guess(num: int) -> int:`，题目会从 $1 \sim n$ 中随机选取一个数 x 。我们只能通过调用接口来判断自己猜测的数是否正确。

要求：要求返回题目选取的数字 x 。

说明：

- `def guess(num: int) -> int:` 返回值：
 - -1 : 我选出的数字比你猜的数字小，即 $pick < num$;
 - 1 : 我选出的数字比你猜的数字大 $pick > num$;
 - 0 : 我选出的数字和你猜的数字一样。恭喜！你猜对了！ $pick == num$ 。

示例：

- 示例 1：

```
输入: n = 10, pick = 6
输出: 6
```

- 示例 2：

```
输入: n = 1, pick = 1
输出: 1
```

解题思路

思路 1：二分查找

利用两个指针 $left$ 、 $right$ 。 $left$ 指向数字 1， $right$ 指向数字 n 。每次从中间开始调用接口猜测是否正确。

- 如果猜测的数比选中的数大，则将 $right$ 向左移，令 `right = mid - 1`，继续从中间调用接口猜测；
- 如果猜测的数比选中的数小，则将 $left$ 向右移，令 `left = mid + 1`，继续从中间调用的接口猜测；
- 如果猜测正确，则直接返回该数。

思路 1：二分查找代码

```
class Solution:
    def guessNumber(self, n: int) -> int:
        left = 1
        right = n
        while left <= right:
            mid = left + (right - left) // 2
            ans = guess(mid)
            if ans == 1:
                left = mid + 1
            elif ans == -1:
                right = mid - 1
            else:
                return mid
        return 0
```

思路 1：复杂度分析

- 时间复杂度： $O(\log n)$ 。二分查找算法的时间复杂度为 $O(\log n)$ 。
- 空间复杂度： $O(1)$ 。只用到了常数空间存放若干变量。

0375. 猜数字大小 II

- 标签：数学、动态规划、博弈
- 难度：中等

题目链接

- [0375. 猜数字大小 II - 力扣](#)

题目大意

描述：现在两个人来玩一个猜数游戏，游戏规则如下：

- 对方从 $1 \sim n$ 中选择一个数字。
- 我们来猜对方选了哪个数字。
- 如果我们猜到了正确数字，就会赢得游戏。
- 如果我们猜错了，那么对方就会告诉我们，所选的数字比我们猜的数字更大或者更小，并且需要我们继续猜数。
- 每当我们猜了数字 x 并且猜错了的时候，我们需要支付金额为 x 的现金。如果我们花光了钱，就会输掉游戏。

现在给定一个特定数字 n 。

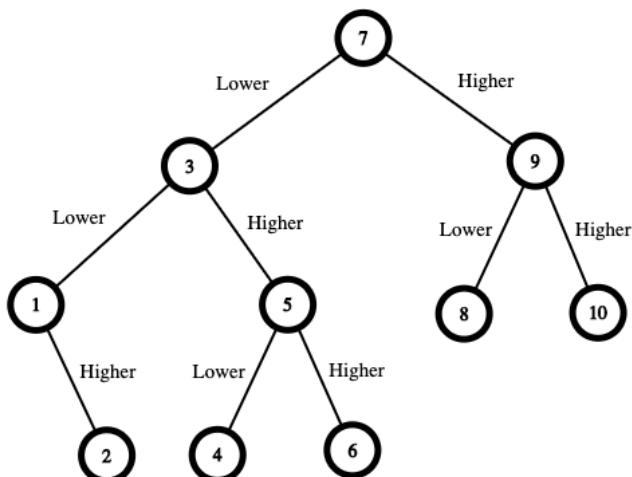
要求：返回能够确保我们获胜的最小现金数（不管对方选择哪个数字）。

说明：

- $1 \leq n \leq 200$ 。

示例：

- 示例 1：



输入: n = 10

输出: 16

解释: 制胜策略如下:

- 数字范围是 [1, 10]。你先猜测数字为 7。
- 如果这是我选中的数字, 你的总费用为 \$0。否则, 你需要支付 \$7。
- 如果我的数字更大, 则下一步需要猜测的数字范围是 [8, 10]。你可以猜测数字为 9。
 - 如果这是我选中的数字, 你的总费用为 \$7。否则, 你需要支付 \$9。
 - 如果我的数字更大, 那么这个数字一定是 10。你猜测数字为 10 并赢得游戏, 总费用为 \$7 + \$9 = \$16。
 - 如果我的数字更小, 那么这个数字一定是 8。你猜测数字为 8 并赢得游戏, 总费用为 \$7 + \$9 = \$16。
- 如果我的数字更小, 则下一步需要猜测的数字范围是 [1, 6]。你可以猜测数字为 3。
 - 如果这是我选中的数字, 你的总费用为 \$7。否则, 你需要支付 \$3。
 - 如果我的数字更大, 则下一步需要猜测的数字范围是 [4, 6]。你可以猜测数字为 5。
 - 如果这是我选中的数字, 你的总费用为 \$7 + \$3 = \$10。否则, 你需要支付 \$5。
 - 如果我的数字更大, 那么这个数字一定是 6。你猜测数字为 6 并赢得游戏, 总费用为 \$7 + \$3 + \$5 = \$15。
 - 如果我的数字更小, 那么这个数字一定是 4。你猜测数字为 4 并赢得游戏, 总费用为 \$7 + \$3 + \$5 = \$15。
 - 如果我的数字更小, 则下一步需要猜测的数字范围是 [1, 2]。你可以猜测数字为 1。
 - 如果这是我选中的数字, 你的总费用为 \$7 + \$3 = \$10。否则, 你需要支付 \$1。
 - 如果我的数字更大, 那么这个数字一定是 2。你猜测数字为 2 并赢得游戏, 总费用为 \$7 + \$3 + \$1 = \$11。

在最糟糕的情况下, 你需要支付 \$16。因此, 你只需要 \$16 就可以确保自己赢得游戏。

• 示例 2:

输入: n = 2

输出: 1

解释: 有两个可能的数字 1 和 2。

- 你可以先猜 1。
 - 如果这是我选中的数字, 你的总费用为 \$0。否则, 你需要支付 \$1。
 - 如果我的数字更大, 那么这个数字一定是 2。你猜测数字为 2 并赢得游戏, 总费用为 \$1。

最糟糕的情况下, 你需要支付 \$1。

解题思路

思路 1：动态规划

直觉上这道题应该通过二分查找来求解, 但实际上并不能通过二分查找来求解。

因为我们可以通过二分查找方法, 能够找到猜中的最小次数, 但这个猜中的最小次数所对应的支付金额, 并不是最小现金数。

也就是说, 通过二分查找的策略, 并不能找到确保我们获胜的最小现金数。所以我们需要转换思路。

我们可以用递归的方式来思考。

对于 $1 \sim n$ 中每一个数 x :

1. 如果 x 恰好是正确数字, 则获胜, 付出的现金数为 0。
2. 如果 x 不是正确数字, 则付出现金数为 x , 同时我们得知, 正确数字比 x 更大还是更小。
 - i. 如果正确数字比 x 更小, 我们只需要求出 $1 \sim x - 1$ 中能够获胜的最小现金数, 再加上 x 就是确保我们获胜的最小现金数。
 - ii. 如果正确数字比 x 更大, 我们只需要求出 $x + 1 \sim n$ 中能够获胜的最小现金数, 再加上 x 就是确保我们获胜的最小现金数。
 - iii. 因为正确数字可能比 x 更小, 也可能比 x 更大。在考虑最坏情况下也能获胜, 我们需要准备的最小现金应该为两种情况下的最小代价的最大值, 再加上 x 本身。

我们可以通过枚举 x , 并求出所有情况下的最小值, 即为确保我们获胜的最小现金数。

我们可以定义一个方法 $f(1)(n)$ 来表示 $1 \sim n$ 中能够获胜的最小现金数, 则可以得到递推公式: $f(1)(n) = \min_{x=1}^{x=n} \{ \max\{f(1)(x-1), f(x+1)(n)\} + x \}$ 。

将递推公式应用到 $i \sim j$ 中, 可得: $f(i)(j) = \min_{x=i}^{x=j} \{ \max\{f(i)(x-1), f(x+1)(j)\} + x \}$

接下来我们就可以通过动态规划的方式解决这道题了。

1. 划分阶段

按照区间长度进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：数字 $i \sim j$ 中能够确保我们获胜的最小现金数。

3. 状态转移方程

$$dp[i][j] = \min_{x=i}^{x=j} \{ \max\{dp[i][x-1], dp[x+1][j]\} + x \}$$

4. 初始条件

- 默认数字 $i \sim j$ 中能够确保我们获胜的最小现金数为无穷大。
- 当区间长度为 1 时，区间中只有 1 个数，肯定为正确数字，则付出最小现金数为 0，即 $dp[i][i] = 0$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：数字 $i \sim j$ 中能够确保我们获胜的最小现金数。所以最终结果为 $dp[1][n]$ 。

思路 1：代码

```
class Solution:
    def getMoneyAmount(self, n: int) -> int:
        dp = [[0 for _ in range(n + 2)] for _ in range(n + 2)]
        for l in range(2, n + 1):
            for i in range(1, n + 1):
                j = i + l - 1
                if j > n:
                    break
                dp[i][j] = float('inf')
                for k in range(i, j):
                    dp[i][j] = min(dp[i][j], max(dp[i][k - 1] + k, dp[k + 1][j] + k))

        return dp[1][n]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^3)$ ，其中 n 为给定整数。
- 空间复杂度： $O(n^2)$ 。

0376. 摆动序列

- 标签：贪心、数组、动态规划
- 难度：中等

题目链接

- [0376. 摆动序列 - 力扣](#)

题目大意

如果一个数组序列中，连续项之间的差值是严格的在正数、负数之间交替，则称该数组序列为「摆动序列」。第一个差值可能为正数，也可能为负数。只有一个元素或者还有两个不等元素的数组序列也可以看做是摆动序列。

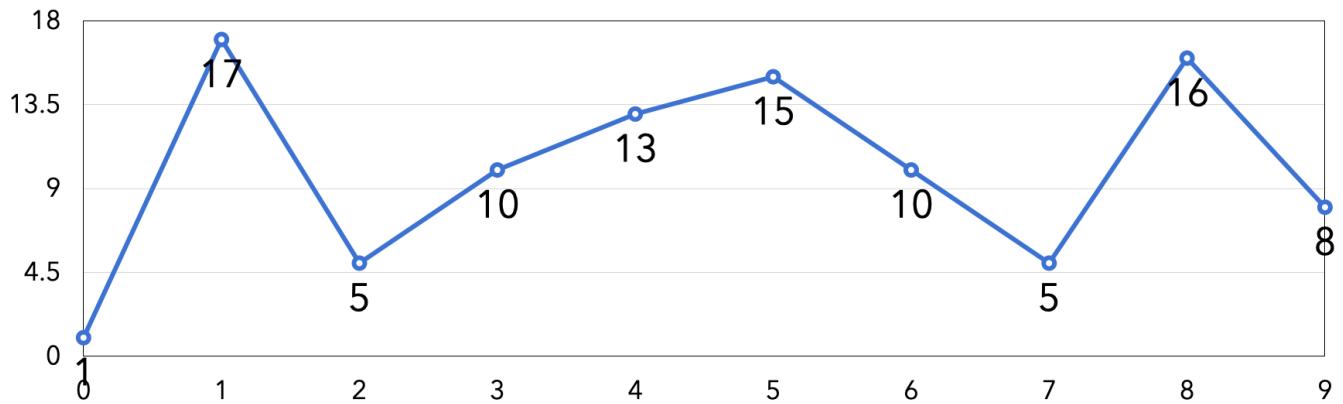
- 例如：[1, 7, 4, 9, 2, 5] 是摆动序列，因为差值 (6, -3, 5, -7, 3) 是正负交替出现的。
- 相反，[1, 4, 7, 2, 5] 和 [1, 7, 4, 5, 5] 不是摆动序列。第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

现在给定一个整数数组 $nums$ ，返回 $nums$ 中作为「摆动序列」的「最长子序列长度」。

解题思路

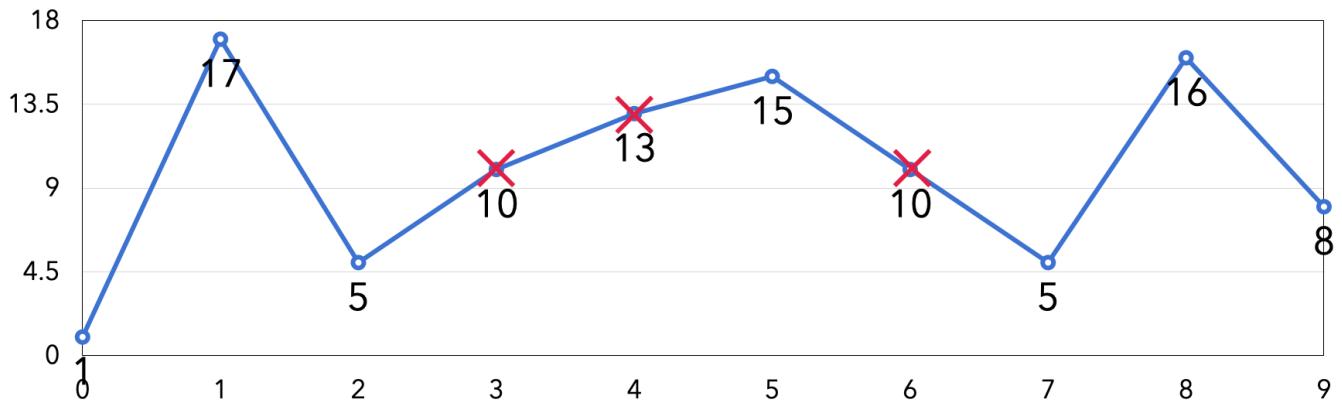
我们先通过一个例子来说明如何求摆动数组最长子序列的长度。

下图是 `nums = [1, 17, 5, 10, 13, 15, 10, 5, 16, 8]` 的图示。



根据题意可知，摆动数组中连续项的差值是正负交替的，直观表现就像是一条高低起伏的山脉，或者像一把锯齿。

观察图像可知，貌似当我们不断交错的选择山脉的「波峰」和「波谷」作为子序列的元素，就会使摆动数组的子序列尽可能的长。例如下图选择 `[1, 17, 5, 15, 5, 16, 8]`。



可是为什么选择「峰」「谷」就能使摆动数组的子序列尽可能的长？为什么我们不选择「中间元素」呢？

其实也可以选择「中间元素」，因为一路从波谷爬坡到波峰再到波谷，和从波谷爬坡到半山腰再回到波谷所形成的摆动数组最长子序列的长度是一样的。

只不过如果选择中间元素，这个中间元素两侧必有波峰和波谷，我们假设选择的序列出现顺序为：「谷 -> 中间元素 -> 谷」，则「谷」和「谷」中间的「峰」必定没有出现在选择的序列中，我们必然可以将选择的「中间元素」替换为「峰」。

同理，「峰 -> 中间元素 -> 峰」中选择的「中间元素」必然也可以替换为「谷」。

所以既然中可以替换，所以我们干脆直接选择「峰」「谷」就可以满足最长子序列的长度。

所以题目就变为了：统计序列中「峰」「谷」的数量。

记录下前一对连续项的差值、当前对连续项的差值，并判断是否是互为正负的。

- 如果互为正负，则为「峰」或「谷」，记录下个数，并更新前一对连续项的差值。
- 如果符号相同，则继续向后判断。

代码

```
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        size = len(nums)
        cur_diff = 0
        pre_diff = 0
        res = 1
        for i in range(size - 1):
            cur_diff = nums[i + 1] - nums[i]
            if (cur_diff > 0 and pre_diff <= 0) or (pre_diff >= 0 and cur_diff < 0):
                res += 1
                pre_diff = cur_diff
        return res
```

0377. 组合总和 IV

- 标签：数组、动态规划
- 难度：中等

题目链接

- [0377. 组合总和 IV - 力扣](#)

题目大意

描述：给定一个由不同整数组成的数组 $nums$ 和一个目标整数 $target$ 。

要求：从 $nums$ 中找出并返回总和为 $target$ 的元素组合个数。

说明：

- 题目数据保证答案符合 32 位整数范围。
- $1 \leq nums.length \leq 200$ 。
- $1 \leq nums[i] \leq 1000$ 。
- $nums$ 中的所有元素互不相同。
- $1 \leq target \leq 1000$ 。

示例：

- **示例 1：**

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

```
输出: 7
```

解释：

所有可能的组合为：

```
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
```

请注意，顺序不同的序列被视作不同的组合。

- **示例 2：**

```
输入: nums = [9], target = 3
```

```
输出: 0
```

解题思路

思路 1：动态规划

「完全背包问题求方案数」的变形。本题与「完全背包问题求方案数」不同点在于：方案中不同的物品顺序代表不同方案。

比如「完全背包问题求方案数」中，凑成总和为 4 的方案 $[1, 3]$ 算 1 种方案，但是在本题中 $[1, 3]$ 、 $[3, 1]$ 算 2 种方案数。

我们需要在考虑某一总和 w 时，需要将 $nums$ 中所有元素都考虑到。对应到循环关系时，即将总和 w 的遍历放到外侧循环，将 $nums$ 数组元素的遍历放到内侧循环，即：

```

for w in range(target + 1):
    for i in range(1, len(nums) + 1):
        xxxx

```

1. 划分阶段

按照总和进行阶段划分。

2. 定义状态

定义状态 $dp[w]$ 表示为：凑成总和 w 的组合数。

3. 状态转移方程

凑成总和为 w 的组合数 = 「不使用当前 $nums[i - 1]$, 只使用之前整数凑成和为 w 的组合数」 + 「使用当前 $nums[i - 1]$ 凑成和为 $w - nums[i - 1]$ 的方案数」。即状态转移方程为： $dp[w] = dp[w] + dp[w - nums[i - 1]]$ 。

4. 初始条件

- 凑成总和 0 的组合数为 1, 即 $dp[0] = 1$ 。

5. 最终结果

根据我们之前定义的状态, $dp[w]$ 表示为：凑成总和 w 的组合数。所以最终结果为 $dp[target]$ 。

思路 1：代码

```

class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        size = len(nums)
        dp = [0 for _ in range(target + 1)]
        dp[0] = 1

        for w in range(target + 1):
            for i in range(1, size + 1):
                if w >= nums[i - 1]:
                    dp[w] = dp[w] + dp[w - nums[i - 1]]

        return dp[target]

```

思路 1：复杂度分析

- 时间复杂度: $O(n \times target)$, 其中 n 为数组 $nums$ 的元素个数, $target$ 为目标整数。
- 空间复杂度: $O(target)$ 。

0378. 有序矩阵中第 K 小的元素

- 标签：数组、二分查找、矩阵、排序、堆（优先队列）
- 难度：中等

题目链接

- [0378. 有序矩阵中第 K 小的元素 - 力扣](#)

题目大意

给定一个 $n * n$ 矩阵 $matrix$, 其中每行和每列元素均按升序排序。

要求：找到矩阵中第 k 小的元素。

注意：它是排序后的第 k 小元素，而不是第 k 个不同的元素。

解题思路

已知二维矩阵 `matrix` 每行每列是按照升序排序的。那么二维矩阵的下界就是左上角元素 `matrix[0][0]`，上界就是右下角元素 `matrix[rows - 1][cols - 1]`。那么我们可以使用二分查找的方法在上界、下界之间搜索所有值，找到第 `k` 小的元素。

我们可以通过判断矩阵中比 `mid` 小的元素个数是否等于 `k` 来确定是否找到第 `k` 小的元素。

- 如果比 `mid` 小的元素个数大于等于 `k`，说明最终答案 `ans` 小于等于 `mid`。
- 如果比 `mid` 小的元素个数小于 `k`，说明最终答案 `ans` 大于 `mid`。

代码

```
class Solution:
    def kthSmallest(self, matrix: List[List[int]], k: int) -> int:
        rows, cols = len(matrix), len(matrix[0])
        left, right = matrix[0][0], matrix[rows - 1][cols - 1] + 1
        while left < right:
            mid = left + (right - left) // 2
            if self.counterKthSmallest(mid, matrix) >= k:
                right = mid
            else:
                left = mid + 1
        return left

    def counterKthSmallest(self, mid, matrix):
        rows, cols = len(matrix), len(matrix[0])
        count = 0
        j = cols - 1
        for i in range(rows):
            while j >= 0 and mid < matrix[i][j]:
                j -= 1
            count += j + 1
        return count
```

0380. 常数时间插入、删除和获取随机元素

- 标签：设计、数组、哈希表、数学、随机化
- 难度：中等

题目链接

- [0380. 常数时间插入、删除和获取随机元素 - 力扣](#)

题目大意

设计一个数据结构，支持时间复杂度为 O(1) 的以下操作：

- `insert(val)`: 当元素 `val` 不存在时，向集合中插入该项。
- `remove(val)`: 元素 `val` 存在时，从集合中移除该项。
- `getRandom`: 随机返回现有集合中的一项。每个元素应该有相同的概率被返回。

解题思路

普通动态数组进行访问操作，需要线性时间查找解决。我们可以利用哈希表记录下每个元素的下标，这样在访问时可以做到常数时间内访问元素了。对应的插入、删除、后去随机元素需要做相应的变化。

- 插入操作：将元素直接插入到数组尾部，并用哈希表记录插入元素的下标位置。

- 删除操作：使用哈希表找到待删除元素所在位置，将其与数组末尾位置元素相互交换，更新哈希表中交换后元素的下标值，并将末尾元素删除。
- 获取随机元素：使用 `random.choice` 获取。

代码

```
import random

class RandomizedSet:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.dict = dict()
        self.list = list()

    def insert(self, val: int) -> bool:
        """
        Inserts a value to the set. Returns true if the set did not already contain the specified element.
        """
        if val in self.dict:
            return False
        self.dict[val] = len(self.list)
        self.list.append(val)
        return True

    def remove(self, val: int) -> bool:
        """
        Removes a value from the set. Returns true if the set contained the specified element.
        """
        if val in self.dict:
            idx = self.dict[val]
            last = self.list[-1]
            self.list[idx] = last
            self.dict[last] = idx
            self.list.pop()
            self.dict.pop(val)
            return True
        return False

    def getRandom(self) -> int:
        """
        Get a random element from the set.
        """
        return random.choice(self.list)
```

0383. 赎金信

- 标签：哈希表、字符串、计数
- 难度：简单

题目链接

- [0383. 赎金信 - 力扣](#)

题目大意

描述：为了不在赎金信中暴露字迹，从杂志上搜索各个需要的字母，组成单词来表达意思。

给定一个赎金信字符串 *ransomNote* 和一个杂志字符串 *magazine*。

要求：判断 *ransomNote* 能不能由 *magazines* 里面的字符构成。如果可以构成，返回 `True`；否则返回 `False`。

说明：

- *magazine* 中的每个字符只能在 *ransomNote* 中使用一次。
- $1 \leq \text{ransomNote.length}, \text{magazine.length} \leq 10^5$ 。
- *ransomNote* 和 *magazine* 由小写英文字母组成。

示例：

- **示例 1：**

```
输入: ransomNote = "a", magazine = "b"
输出: False
```

- **示例 2：**

```
输入: ransomNote = "aa", magazine = "ab"
输出: False
```

解题思路

思路 1：哈希表

暴力做法是双重循环遍历字符串 *ransomNote* 和 *magazines*。我们可以用哈希表来减少算法的时间复杂度。具体做法如下：

- 先用哈希表存储 *magazines* 中各个字符的个数（哈希表可用字典或数组实现）。
- 再遍历字符串 *ransomNote* 中每个字符，对于每个字符：
 - 如果在哈希表中个数为 0，直接返回 `False`。
 - 如果在哈希表中个数不为 0，将其个数减 1。
- 遍历到最后，则说明 *ransomNote* 能由 *magazines* 里面的字符构成。返回 `True`。

思路 1：代码

```
class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        magazine_counts = [0 for _ in range(26)]

        for ch in magazine:
            num = ord(ch) - ord('a')
            magazine_counts[num] += 1

        for ch in ransomNote:
            num = ord(ch) - ord('a')
            if magazine_counts[num] == 0:
                return False
            else:
                magazine_counts[num] -= 1

        return True
```

思路 1：复杂度分析

- **时间复杂度：** $O(m + n)$ ，其中 m 是字符串 *ransomNote* 的长度， n 是字符串 *magazines* 的长度。

- 空间复杂度: $O(|S|)$, 其中 S 是字符集, 本题中 $|S| = 26$ 。

0384. 打乱数组

- 标签: 数组、数学、随机化
- 难度: 中等

题目链接

- [0384. 打乱数组 - 力扣](#)

题目大意

描述: 给定一个整数数组 $nums$ 。

要求: 设计算法来打乱一个没有重复元素的数组。打乱后, 数组的所有排列应该是等可能的。

实现 `Solution class`:

- `Solution(int[] nums)` 使用整数数组 $nums$ 初始化对象。
- `int[] reset()` 重设数组到它的初始状态并返回。
- `int[] shuffle()` 返回数组随机打乱后的结果。

说明:

- $1 \leq nums.length \leq 50$ 。
- $-10^6 \leq nums[i] \leq 10^6$ 。
- $nums$ 中的所有元素都是唯一的。
- 最多可以调用 10^4 次 `reset` 和 `shuffle`。

示例:

- 示例 1:

```
输入:
["Solution", "shuffle", "reset", "shuffle"]
[[[1, 2, 3]], [], [], []]
输出:
[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]
```

解释:

```
Solution solution = new Solution([1, 2, 3]);
solution.shuffle();    // 打乱数组 [1,2,3] 并返回结果。任何 [1,2,3] 的排列返回的概率应该相同。例如, 返回 [3, 1, 2]
solution.reset();     // 重设数组到它的初始状态 [1, 2, 3]。返回 [1, 2, 3]
solution.shuffle();   // 随机返回数组 [1, 2, 3] 打乱后的结果。例如, 返回 [1, 3, 2]
```

解题思路

思路 1：洗牌算法

题目要求在打乱顺序后, 数组的所有排列应该是等可能的。对于长度为 n 的数组, 我们可以把问题转换为: 分别在 n 个位置上, 选择填入某个数的概率是相同的。具体选择方法如下:

- 对于第 0 个位置, 我们从 $0 \sim n - 1$ 总共 n 个数中随机选择一个数, 将该数与第 0 个位置上的数进行交换。则每个数被选到的概率为 $\frac{1}{n}$ 。
- 对于第 1 个位置, 我们从剩下 $n - 1$ 个数中随机选择一个数, 将该数与第 1 个位置上的数进行交换。则每个数被选到的概率为 $\frac{n-1}{n} \times \frac{1}{n-1} = \frac{1}{n}$ (第一次没选到并且第二次被选中)。
- 对于第 2 个位置, 我们从剩下 $n - 2$ 个数中随机选择一个数, 将该数与第 2 个位置上的数进行交换。则每个数被选到的概率为 $\frac{n-1}{n} \times \frac{n-2}{n-1} \times \frac{1}{n-2} = \frac{1}{n}$ (第一次没选到、第二次没选到, 并且第三次被选中)。

- 依次类推，对于每个位置上，每个数被选中的概率都是 $\frac{1}{n}$ 。

思路 1：洗牌算法代码

```
class Solution:

    def __init__(self, nums: List[int]):
        self.nums = nums

    def reset(self) -> List[int]:
        return self.nums

    def shuffle(self) -> List[int]:
        self.shuffle_nums = self.nums.copy()
        for i in range(len(self.shuffle_nums)):
            swap_index = random.randrange(i, len(self.shuffle_nums))
            self.shuffle_nums[i], self.shuffle_nums[swap_index] = self.shuffle_nums[swap_index], self.shuffle_nums[i]
        return self.shuffle_nums
```

参考资料

- 【题解】「Python/Java/JavaScript/Go」 洗牌算法 - 打乱数组 - 力扣# 0386. 字典序排数
- 标签：深度优先搜索、字典树
- 难度：中等

题目链接

- [0386. 字典序排数 - 力扣](#)

题目大意

给定一个整数 n 。

要求：按字典序返回范围 $[1, n]$ 的所有整数。并且要求时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

解题思路

按照字典序进行深度优先搜索。实质上算是构造一棵字典树，然后将 $[1, n]$ 中的数插入到字典树中，并将遍历结果存储到列表中。

代码

```

class Solution:
    def dfs(self, cur, n, res):
        if cur > n:
            return
        res.append(cur)
        for i in range(10):
            num = 10 * cur + i
            if num > n:
                return
            self.dfs(num, n, res)

    def lexicalOrder(self, n: int) -> List[int]:
        res = []
        for i in range(1, 10):
            self.dfs(i, n, res)
        return res

```

0387. 字符串中的第一个唯一字符

- 标签: 队列、哈希表、字符串、计数
- 难度: 简单

题目链接

- 0387. 字符串中的第一个唯一字符 - 力扣

题目大意

给定一个只包含小写字母的字符串 `s`。

要求: 找到第一个不重复的字符, 并返回它的索引。

解题思路

遍历字符串, 使用哈希表存储字符串中每个字符的出现次数。然后第二次遍历时, 找出只出现一次的字符。

代码

```

class Solution:
    def firstUniqChar(self, s: str) -> int:
        strDict = dict()
        for i in range(len(s)):
            if s[i] in strDict:
                strDict[s[i]] += 1
            else:
                strDict[s[i]] = 1

        for i in range(len(s)):
            if s[i] in strDict and strDict[s[i]] == 1:
                return i
        return -1

```

- 思路 2 代码:

0389. 找不同

- 标签：位运算、哈希表、字符串、排序
- 难度：简单

题目链接

- [0389. 找不同 - 力扣](#)

题目大意

给定两个只包含小写字母的字符串 s 、 t 。字符串 t 是由 s 进行随机重拍之后，再在随机位置添加一个字母得到的。要求：找出字符串 t 中被添加的字母。

解题思路

字符串 t 比字符串 s 多了一个随机字母。可以使用哈希表存储一下字符串 s 中各个字符的数量，再遍历一遍字符串 t 中的字符，从哈希表中减去对应数量的字符，最后剩的那个字符就是多余的字符。

代码

```
class Solution:
    def findTheDifference(self, s: str, t: str) -> str:
        s_dict = dict()
        for ch in s:
            if ch in s_dict:
                s_dict[ch] += 1
            else:
                s_dict[ch] = 1

        for ch in t:
            if ch in s_dict and s_dict[ch] != 0:
                s_dict[ch] -= 1
            else:
                return ch
```

0391. 完美矩形

- 标签：数组、扫描线
- 难度：困难

题目链接

- [0391. 完美矩形 - 力扣](#)

题目大意

描述：给定一个数组 rectangles ，其中 $\text{rectangles}[i] = [x_i, y_i, a_i, b_i]$ 表示一个坐标轴平行的矩形。这个矩形的左下顶点是 (x_i, y_i) ，右上顶点是 (a_i, b_i) 。

要求：如果所有矩形一起精确覆盖了某个矩形区域，则返回 `True`；否则，返回 `False`。

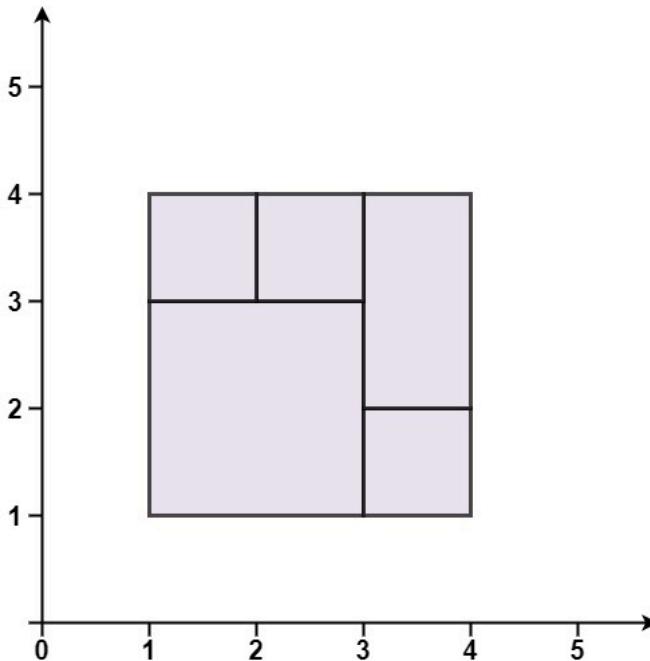
说明：

- $1 \leq \text{rectangles.length} \leq 2 * 10^4$ 。
- $\text{rectangles}[i].length == 4$ 。

- $-10^5 \leq xi, yi, ai, bi \leq 10^5$ 。

示例:

- 示例 1:

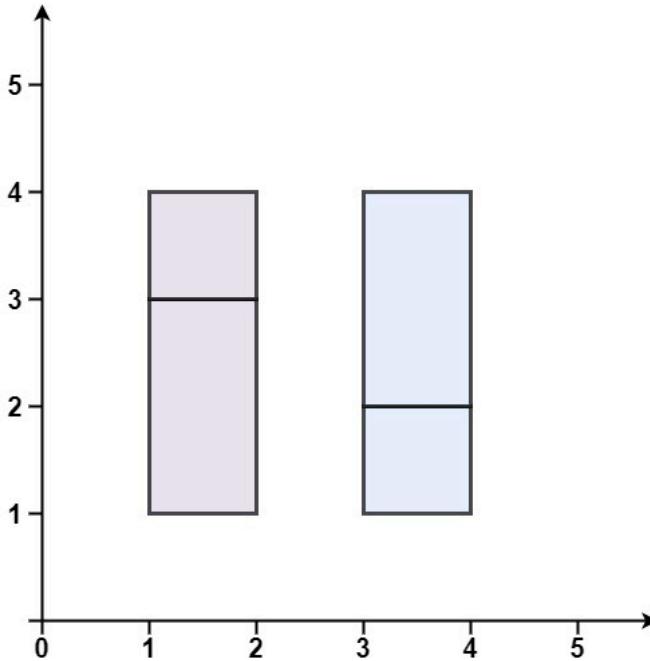


输入: `rectangles = [[1, 1, 3, 3], [3, 1, 4, 2], [3, 2, 4, 4], [1, 3, 2, 4], [2, 3, 3, 4]]`

输出: `True`

解释: 5 个矩形一起可以精确地覆盖一个矩形区域。

- 示例 2:

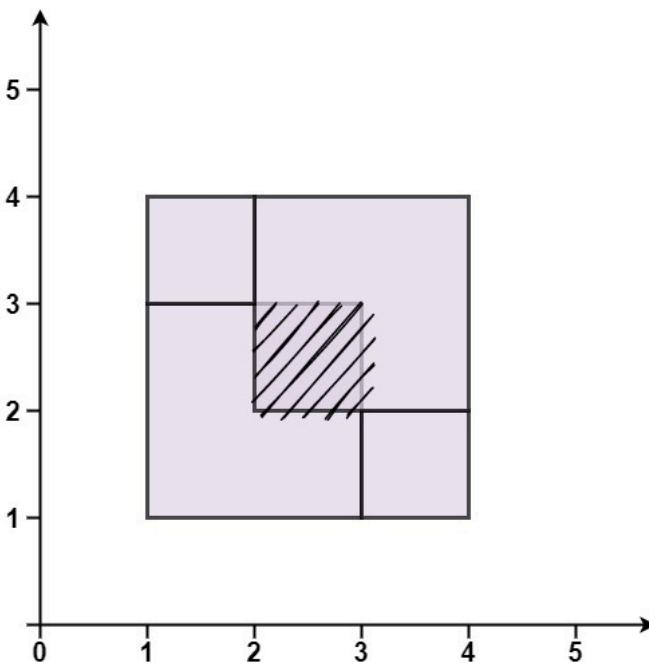


输入: `rectangles = [[1, 1, 2, 3], [1, 3, 2, 4], [3, 1, 4, 2], [3, 2, 4, 4]]`

输出: `false`

解释: 两个矩形之间有间隔, 无法覆盖成一个矩形。

- 示例 3:



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

输出: `False`

解释: 因为中间有相交区域, 虽然形成了矩形, 但不是精确覆盖。

解题思路

思路 1: 线段树

首先我们要先判断所有小矩形的面积和是否等于外接矩形区域的面积。如果不相等, 则说明出现了重叠或者空缺, 明显不符合题意。

在两者面积相等的情况下, 还可能会发生重叠的情况。接下来我们要思考如何判断重叠。

- 第一种思路: 暴力枚举所有矩形对, 两两进行比较, 判断是否出现了重叠。这样的时间复杂度是 $O(n^2)$, 容易超时。
- 第二种思路:
 - 如果所有小矩形可以精确覆盖某个矩形区域, 那这些小矩形一定是相互挨着的, 也就是说相邻两个矩形的边会重合在一起。比如说 矩形 A 下边刚好是 B 的上边, 或者是 B 的上边的一部分。
 - 我们可以固定一个坐标轴, 比如说固定 y 轴, 然后只看水平方向上所有矩形的边。然后我们就会发现, 满足题意要求的矩形区域中, 纵坐标为 y 的平行线上, 「所有上边纵坐标为 y 的矩形上边区间」与「所有下边纵坐标为 y 的矩形下边区间」是完全一样, 或者说重合在一起的(除了矩形矩形最上边和最下边只有一条, 不会重合之外)。
 - 这样我们就可以用扫描线的思路, 建立一个线段树。然后先固定纵坐标 y, 将「所有上边纵坐标为 y 的矩形上边区间」对应的区间值减 1, 再将「所有下边纵坐标为 y 的矩形下边区间」对应的区间值加 1。然后查询整个线段树区间值, 如果区间值超过 1, 则说明发生了重叠, 不符合题目要求。如果扫描完所有的纵坐标, 没有发生重叠, 则说明符合题意要求。
 - 因为横坐标的范围为 $[-10^5, 10^5]$, 但是最多只有 $2 * 10^4$ 个横坐标, 所以我们可以先对所有坐标做一下离散化处理, 再根据离散化之后的横坐标建立线段树。

具体步骤如下:

- 通过遍历所有小矩形, 计算出所有小矩形的面积和为 `area`。同时计算出矩形区域四个顶点位置, 并根据四个顶点计算出矩形区域的面积为 `total_area`。如果所有小矩形面积不等于矩形区域的面积, 则直接返回 `False`。
- 再次遍历所有小矩形, 将所有坐标点进行离散化处理, 将其编号存入两个哈希表 `x_dict`、`y_dict`。
- 使用哈希表 `top_dict`、`bottom_dict` 分别存储每个矩阵的上下两条边。将上下两条边的横坐标 `x1`、`x2`。分别存入到 `top_dict[y_dict[y2]]`、`top_dict[y_dict[y2]]` 中。
- 建立区间长度为横坐标个数的线段树 `STree`。
- 遍历所有的纵坐标, 对于纵坐标 `i`:
 - 先遍历当前纵坐标下矩阵的上边数组, 即 `top_dict[i]`, 取出边的横坐标 `x1`、`x2`。令区间 `[x1, x2 - 1]` 上的值减 1。
 - 再遍历当前纵坐标下矩阵的下边数组, 即 `bottom_dict[i]`, 取出边的横坐标 `x1`、`x2`。令区间 `[x1, x2 - 1]` 上的值加 1。
 - 如果上下边覆盖完之后, 被覆盖次数超过了 1, 则说明出现了重叠, 直接返回 `False`。

6. 如果遍历完所有的纵坐标，没有发现重叠，则返回 `True`。

思路 1：线段树代码

```

# 线段树的节点类
class SegTreeNode:
    def __init__(self, val=0):
        self.left = -1 # 区间左边界
        self.right = -1 # 区间右边界
        self.val = val # 节点值 (区间值)
        self.lazy_tag = None # 区间和问题的延迟更新标记

# 线段树类
class SegmentTree:
    # 初始化线段树接口
    def __init__(self, nums, function):
        self.size = len(nums)
        self.tree = [SegTreeNode() for _ in range(4 * self.size)] # 维护 SegTreeNode 数组
        self.nums = nums # 原始数据
        self.function = function # function 是一个函数, 左右区间的聚合方法
        if self.size > 0:
            self.__build(0, 0, self.size - 1)

    # 单点更新接口: 将 nums[i] 更改为 val
    def update_point(self, i, val):
        self.nums[i] = val
        self.__update_point(i, val, 0)

    # 区间更新接口: 将区间为 [q_left, q_right] 上的所有元素值加上 val
    def update_interval(self, q_left, q_right, val):
        self.__update_interval(q_left, q_right, val, 0)

    # 区间查询接口: 查询区间为 [q_left, q_right] 的区间值
    def query_interval(self, q_left, q_right):
        return self.__query_interval(q_left, q_right, 0)

    # 获取 nums 数组接口: 返回 nums 数组
    def get_nums(self):
        for i in range(self.size):
            self.nums[i] = self.query_interval(i, i)
        return self.nums

    # 以下为内部实现方法

    # 构建线段树实现方法: 节点的存储下标为 index, 节点的区间为 [left, right]
    def __build(self, index, left, right):
        self.tree[index].left = left
        self.tree[index].right = right
        if left == right: # 叶子节点, 节点值为对应位置的元素值
            self.tree[index].val = self.nums[left]
            return

        mid = left + (right - left) // 2 # 左右节点划分点
        left_index = index * 2 + 1 # 左子节点的存储下标
        right_index = index * 2 + 2 # 右子节点的存储下标
        self.__build(left_index, left, mid) # 递归创建左子树
        self.__build(right_index, mid + 1, right) # 递归创建右子树
        self.__pushup(index) # 向上更新节点的区间值

    # 区间更新实现方法
    def __update_interval(self, q_left, q_right, val, index):
        left = self.tree[index].left
        right = self.tree[index].right

```

```

if left >= q_left and right <= q_right:      # 节点所在区间被 [q_left, q_right] 所覆盖
    if self.tree[index].lazy_tag is not None:
        self.tree[index].lazy_tag += val      # 将当前节点的延迟标记增加 val
    else:
        self.tree[index].lazy_tag = val      # 将当前节点的延迟标记增加 val
        self.tree[index].val += val          # 当前节点所在区间每个元素值增加 val
    return

if right < q_left or left > q_right:          # 节点所在区间与 [q_left, q_right] 无关
    return

self.__pushdown(index)                         # 向下更新节点的区间值

mid = left + (right - left) // 2               # 左右节点划分点
left_index = index * 2 + 1                      # 左子节点的存储下标
right_index = index * 2 + 2                      # 右子节点的存储下标
if q_left <= mid:                             # 在左子树中更新区间值
    self.__update_interval(q_left, q_right, val, left_index)
if q_right > mid:                            # 在右子树中更新区间值
    self.__update_interval(q_left, q_right, val, right_index)

self.__pushup(index)                          # 向上更新节点的区间值

# 区间查询实现方法: 在线段树中搜索区间为 [q_left, q_right] 的区间值
def __query_interval(self, q_left, q_right, index):
    left = self.tree[index].left
    right = self.tree[index].right

    if left >= q_left and right <= q_right:      # 节点所在区间被 [q_left, q_right] 所覆盖
        return self.tree[index].val                # 直接返回节点值
    if right < q_left or left > q_right:          # 节点所在区间与 [q_left, q_right] 无关
        return 0

    self.__pushdown(index)

    mid = left + (right - left) // 2               # 左右节点划分点
    left_index = index * 2 + 1                      # 左子节点的存储下标
    right_index = index * 2 + 2                      # 右子节点的存储下标
    res_left = 0                                     # 左子树查询结果
    res_right = 0                                    # 右子树查询结果
    if q_left <= mid:                            # 在左子树中查询
        res_left = self.__query_interval(q_left, q_right, left_index)
    if q_right > mid:                           # 在右子树中查询
        res_right = self.__query_interval(q_left, q_right, right_index)

    return self.function(res_left, res_right)      # 返回左右子树元素值的聚合计算结果

# 向上更新实现方法: 更新下标为 index 的节点区间值 等于 该节点左右子节点元素值的聚合计算结果
def __pushup(self, index):
    left_index = index * 2 + 1                      # 左子节点的存储下标
    right_index = index * 2 + 2                      # 右子节点的存储下标
    self.tree[index].val = self.function(self.tree[left_index].val, self.tree[right_index].val)

# 向下更新实现方法: 更新下标为 index 的节点所在区间的左右子节点的值和懒惰标记
def __pushdown(self, index):
    lazy_tag = self.tree[index].lazy_tag
    if lazy_tag is None:
        return

    left_index = index * 2 + 1                      # 左子节点的存储下标
    right_index = index * 2 + 2                      # 右子节点的存储下标

    if self.tree[left_index].lazy_tag is not None:
        self.tree[left_index].lazy_tag += lazy_tag  # 更新左子节点懒惰标记
    else:

```

```

        self.tree[left_index].lazy_tag = lazy_tag
        self.tree[left_index].val += lazy_tag

    if self.tree[right_index].lazy_tag is not None:
        self.tree[right_index].lazy_tag += lazy_tag # 更新右子节点懒惰标记
    else:
        self.tree[right_index].lazy_tag = lazy_tag
    self.tree[right_index].val += lazy_tag
    self.tree[index].lazy_tag = None           # 更新当前节点的懒惰标记


class Solution:
    def isRectangleCover(self, rectangles) -> bool:
        left, right, bottom, top = math.inf, -math.inf, math.inf, -math.inf
        area = 0
        x_set, y_set = set(), set()

        for rectangle in rectangles:
            x1, y1, x2, y2 = rectangle
            left, right = min(left, x1), max(right, x2)
            bottom, top = min(bottom, y1), max(top, y2)
            area += (y2 - y1) * (x2 - x1)
            x_set.add(x1)
            x_set.add(x2)
            y_set.add(y1)
            y_set.add(y2)

        total_area = (top - bottom) * (right - left)

        # 判断所有小矩形面积是否等于所有矩形顶点构成最大矩形面积，不等于则直接返回 False
        if area != total_area:
            return False

        # 离散化处理所有点的横坐标、纵坐标
        x_dict, y_dict = dict(), dict()

        idx = 0
        for x in sorted(list(x_set)):
            x_dict[x] = idx
            idx += 1

        idy = 0
        for y in sorted(list(y_set)):
            y_dict[y] = idy
            idy += 1

        # 使用哈希表 top_dict, bottom_dict 分别存储每个矩阵的上下两条边。
        bottom_dict, top_dict = collections.defaultdict(list), collections.defaultdict(list)
        for i in range(len(rectangles)):
            x1, y1, x2, y2 = rectangles[i]
            bottom_dict[y_dict[y1]].append([x_dict[x1], x_dict[x2]])
            top_dict[y_dict[y2]].append([x_dict[x1], x_dict[x2]])

        # 建立线段树
        self.STree = SegmentTree([0 for _ in range(len(x_set))], lambda x, y: max(x, y))

        for i in range(idy):
            for x1, x2 in top_dict[i]:
                self.STree.update_interval(x1, x2 - 1, -1)
            for x1, x2 in bottom_dict[i]:
                self.STree.update_interval(x1, x2 - 1, 1)
        cnt = self.STree.query_interval(0, len(x_set) - 1)
        if cnt > 1:

```

```
    return False
return True
```

参考资料

- 【题解】线段树+扫描线 - 完美矩形 - 力扣# 0392. 判断子序列
- 标签：双指针、字符串、动态规划
- 难度：简单

题目链接

- 0392. 判断子序列 - 力扣

题目大意

描述：给定字符串 s 和 t 。

要求：判断 s 是否为 t 的子序列。

说明：

- $0 \leq s.length \leq 100$ 。
- $0 \leq t.length \leq 10^4$ 。
- 两个字符串都只由小写字符组成。

示例：

- 示例 1：

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

- 示例 2：

```
输入: s = "axc", t = "ahbgdc"
输出: False
```

解题思路

思路 1：双指针

使用两个指针 i 、 j 分别指向字符串 s 和 t ，然后对两个字符串进行遍历。

- 遇到 $s[i] == t[j]$ 的情况，则 i 向右移。
- 不断右移 j 。
- 如果超过 s 或 t 的长度则跳出。
- 最后判断指针 i 是否指向了 s 的末尾，即：判断 i 是否等于 s 的长度。如果等于，则说明 s 是 t 的子序列，如果不等于，则不是。

思路 1：代码

```
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        size_s = len(s)
        size_t = len(t)
        i, j = 0, 0
        while i < size_s and j < size_t:
            if s[i] == t[j]:
                i += 1
            j += 1
        return i == size_s
```

思路 1：复杂度分析

- 时间复杂度: $O(n + m)$, 其中 n 、 m 分别为字符串 s 、 t 的长度。
- 空间复杂度: $O(1)$ 。

0394. 字符串解码

- 标签: 栈、递归、字符串
- 难度: 中等

题目链接

- [0394. 字符串解码 - 力扣](#)

题目大意

描述: 给定一个经过编码的字符串 s 。

要求: 返回 s 经过解码之后的字符串。

说明:

- 编码规则: $k[encoded_string]$ 。 $encoded_string$ 为字符串, k 为整数。表示字符串 $encoded_string$ 重复 k 次。
- $1 \leq s.length \leq 30$ 。
- 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 $[1, 300]$ 。

示例:

- 示例 1:

```
输入: s = "3[a]2[bc]"
输出: "aaabcbc"
```

- 示例 2:

```
输入: s = "3[a2[c]]"
输出: "accaccacc"
```

解题思路

思路 1：栈

1. 使用两个栈 `stack1`、`stack2`。`stack1` 用来保存左括号前已经解码的字符串，`stack2` 用来存储左括号前的数字。
2. 用 `res` 存储待解码的字符串、`num` 存储当前数字。
3. 遍历字符串。
 - i. 如果遇到数字，则累加数字到 `num`。
 - ii. 如果遇到左括号，将当前待解码字符串入栈 `stack1`，当前数字入栈 `stack2`，然后将 `res`、`nums` 清空。
 - iii. 如果遇到右括号，则从 `stack1` 的取出待解码字符串 `res`，从 `stack2` 中取出当前数字 `num`，将其解码拼合成字符串赋值给 `res`。
 - iv. 如果遇到其他情况（遇到字母），则将当前字母加入 `res` 中。
4. 遍历完输出解码之后的字符串 `res`。

思路 1：代码

```
class Solution:
    def decodeString(self, s: str) -> str:
        stack1 = []
        stack2 = []
        num = 0
        res = ""
        for ch in s:
            if ch.isdigit():
                num = num * 10 + int(ch)
            elif ch == '[':
                stack1.append(res)
                stack2.append(num)
                res = ""
                num = 0
            elif ch == ']':
                cur_res = stack1.pop()
                cur_num = stack2.pop()
                res = cur_res + res * cur_num
            else:
                res += ch
        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

0395. 至少有 K 个重复字符的最长子串

- 标签：哈希表、字符串、分治、滑动窗口
- 难度：中等

题目链接

- [0395. 至少有 K 个重复字符的最长子串 - 力扣](#)

题目大意

给定一个字符串 `s` 和一个整数 `k`。

要求：找出 `s` 中的最长子串，要求该子串中的每一字符出现次数都不少于 `k`。返回这一子串的长度。

注意：`s` 仅由小写英文字母构成。

解题思路

这道题看起来很像是常规滑动窗口套路的问题，但是用普通滑动窗口思路无法解决问题。

如果窗口需要保证「各种字符出现次数都大于等于 k 」这一性质。那么当向右移动 $right$ ，扩大窗口时，如果 $s[right]$ 是第一次出现的元素，窗口内的字符种类数量必然会增加，此时缩小 $s[left]$ 也不一定满足窗口内「各种字符出现次数都大于等于 k 」这一性质。那么我们就无法通过这种方式来进行滑动窗口。

但是我们可以通过固定字符种类数的方式进行滑动窗口。因为给定字符串 s 仅有小写字母构成，则最长子串中的字符种类数目，最少为 1 种，最多为 26 种。我们通过枚举最长子串中可能出现的字符种类数目，从而固定窗口中出现的字符种类数目 i ($1 \leq i \leq 26$)，再进行滑动数组。窗口内需要保证出现的字符种类数目等于 i 。向右移动 $right$ ，扩大窗口时，记录窗口内各种类字符数量。当窗口内出现字符数量大于 i 时，则不断右移 $right$ ，保证窗口内出现字符种类等于 i 。同时，记录窗口内出现次数小于 k 的字符数量，当窗口中出现次数小于 k 的字符数量为 0 时，就可以记录答案，并维护答案最大值了。

整个算法的具体步骤如下：

- 使用 ans 记录满足要求的最长子串长度。
- 枚举最长子串中的字符种类数目 i ，最小为 1 种，最大为 26 种。对于给定字符种类数目 i ：
 - 使用两个指针 $left$ 、 $right$ 指向滑动窗口的左右边界。
 - 使用 $window_count$ 变量来统计窗口内字符种类数目，保证窗口中的字符种类数目 $window_count$ 不多于 i 。
 - 使用 $letter_map$ 哈希表记录窗口中各个字符出现的数目。使用 $less_k_count$ 记录窗口内出现次数小于 k 次的字符数量。
 - 向右移动 $right$ ，将最右侧字符 $s[right]$ 加入当前窗口，用 $letter_map$ 记录该字符个数。
 - 如果该字符第一次出现，即 $letter_map[s[right]] == 1$ ，则窗口内字符种类数目 +1，即 $window_count += 1$ 。同时窗口内小于 k 次的字符数量 +1（等到 $letter_map[s[right]] >= k$ 时再减去），即 $less_k_count += 1$ 。
 - 如果该字符已经出现过 k 次，即 $letter_map[s[right]] == k$ ，则窗口内小于 k 次的字符数量 -1，即 $less_k_count -= 1$ 。
 - 当窗口内字符种类数目 $window_count$ 大于给定字符种类数目 i 时，即 $window_count > i$ ，则不断右移 $left$ ，缩小滑动窗口长度，直到 $window_count == i$ 。
 - 如果此时窗口内字符种类数目 $window_count$ 等于给定字符种类 i 并且小于 k 次的字符数量为 0，即 $window_count == i$ and $less_k_count == 0$ 时，维护更新答案为 $ans = max(right - left + 1, ans)$ 。
- 最后输出答案 ans 。

代码

```

class Solution:
    def longestSubstring(self, s: str, k: int) -> int:
        ans = 0
        for i in range(1, 27):
            left, right = 0, 0
            window_count = 0
            less_k_count = 0
            letter_map = dict()
            while right < len(s):
                if s[right] in letter_map:
                    letter_map[s[right]] += 1
                else:
                    letter_map[s[right]] = 1

                if letter_map[s[right]] == 1:
                    window_count += 1
                    less_k_count += 1
                if letter_map[s[right]] == k:
                    less_k_count -= 1

                while window_count > i:
                    letter_map[s[left]] -= 1
                    if letter_map[s[left]] == 0:
                        window_count -= 1
                        less_k_count -= 1
                    if letter_map[s[left]] == k - 1:
                        less_k_count += 1
                    left += 1

                if window_count == i and less_k_count == 0:
                    ans = max(right - left + 1, ans)
                right += 1
        return ans

```

0399. 除法求值

- 标签：深度优先搜索、广度优先搜索、并查集、图、数组、最短路
- 难度：中等

题目链接

- [0399. 除法求值 - 力扣](#)

题目大意

描述：给定一个变量对数组 $equations$ 和一个实数数组 $values$ 作为已知条件，其中 $equations[i] = [Ai, Bi]$ 和 $values[i]$ 共同表示 $Ai / Bi = values[i]$ 。每个 Ai 或 Bi 是一个表示单个变量的字符串。

再给定一个表示多个问题的数组 $queries$ ，其中 $queries[j] = [Cj, Dj]$ 表示第 j 个问题，要求：根据已知条件找出 $Cj / Dj = ?$ 的结果作为答案。

要求：返回所有问题的答案。如果某个答案无法确定，则用 -1.0 代替，如果问题中出现了给定的已知条件中没有出现的表示变量的字符串，则也用 -1.0 代替这个答案。

说明：

- 未在等式列表中出现的变量是未定义的，因此无法确定它们的答案。
- $1 \leq equations.length \leq 20$ 。

- $\text{equations}[i].length == 2$ 。
- $1 \leq A_i.length, B_i.length \leq 5$ 。
- $\text{values}.length == \text{equations}.length$ 。
- $0.0 < \text{values}[i] \leq 20.0$ 。
- $1 \leq \text{queries}.length \leq 20$ 。
- $\text{queries}[i].length == 2$ 。
- $1 \leq C_j.length, D_j.length \leq 5$ 。
- A_i, B_i, C_j, D_j 由小写英文字母与数字组成。

示例：

- 示例 1：

```
输入: equations = [["a","b"], ["b","c"]], values = [2.0,3.0], queries = [["a","c"], ["b","a"], ["a","e"], ["a","a"], ["x","x"]]
输出: [6.00000,0.50000,-1.00000,1.00000,-1.00000]
解释:
条件: a / b = 2.0, b / c = 3.0
问题: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?
结果: [6.0, 0.5, -1.0, 1.0, -1.0 ]
注意: x 是未定义的 => -1.0
```

- 示例 2：

```
输入: equations = [["a","b"], ["b","c"], ["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"], ["c","b"], ["bc","cd"], ["cd","bc"]]
输出: [3.75000,0.40000,5.00000,0.20000]
```

解题思路

思路 1：并查集

在「等式方程的可满足性」的基础上增加了倍数关系。在「等式方程的可满足性」中我们处理传递关系使用了并查集，这道题也是一样，不过在使用并查集的同时还要维护倍数关系。

举例说明：

- $a / b = 2.0$ ：说明 $a == 2b$, a 和 b 在同一个集合。
- $b / c = 3.0$ ：说明 $b == 3c$, b 和 c 在同一个集合。

根据上述两式可得： a, b, c 都在一个集合中，且 $a == 2b == 6c$ 。

我们可以将同一集合中的变量倍数关系都转换为与根节点变量的倍数关系，比如上述例子中都转变为与 a 的倍数关系。

具体操作如下：

- 定义并查集结构，并在并查集中定义一个表示倍数关系的 *multiples* 数组。
- 遍历 *equations* 数组、*values* 数组，将每个变量按顺序编号，并使用 *union* 将其并入相同集合。
- 遍历 *queries* 数组，判断两个变量是否在并查集中，并且是否在同一集合。如果找到对应关系，则将计算后的倍数关系存入答案数组，否则则将 -1 存入答案数组。
- 最终输出答案数组。

并查集中维护倍数相关方法说明：

- *find* 方法：
 - 递推寻找根节点，并将倍数累乘，然后进行路径压缩，并且更新当前节点的倍数关系。
- *union* 方法：
 - 如果两个节点属于同一集合，则直接返回。
 - 如果两个节点不属于同一个集合，合并之前当前节点的倍数关系更新，然后再进行更新。
- *is_connected* 方法：
 - 如果两个节点不属于同一集合，返回 -1 。
 - 如果两个节点属于同一集合，则返回倍数关系。

思路 1：代码

```

class UnionFind:

    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.multiples = [1 for _ in range(n)]

    def find(self, x):
        multiple = 1.0
        origin = x
        while x != self.parent[x]:
            multiple *= self.multiples[x]
            x = self.parent[x]
        self.parent[origin] = x
        self.multiples[origin] = multiple
        return x

    def union(self, x, y, multiple):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x == root_y:
            return
        self.parent[root_x] = root_y
        self.multiples[root_x] = multiple * self.multiples[y] / self.multiples[x]
        return

    def is_connected(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            return -1.0

        return self.multiples[x] / self.multiples[y]

class Solution:

    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        equations_size = len(equations)
        hash_map = dict()
        union_find = UnionFind(2 * equations_size)

        id = 0
        for i in range(equations_size):
            equation = equations[i]
            var1, var2 = equation[0], equation[1]
            if var1 not in hash_map:
                hash_map[var1] = id
                id += 1
            if var2 not in hash_map:
                hash_map[var2] = id
                id += 1
            union_find.union(hash_map[var1], hash_map[var2], values[i])

        queries_size = len(queries)
        res = []
        for i in range(queries_size):
            query = queries[i]
            var1, var2 = query[0], query[1]
            if var1 not in hash_map or var2 not in hash_map:
                res.append(-1.0)
            else:
                id1 = hash_map[var1]
                id2 = hash_map[var2]
                res.append(union_find.is_connected(id1, id2))

```

```
return res
```

思路 1：复杂度分析

- 时间复杂度: $O((m + n) \times \alpha(m + n))$, α 是反 Ackerman 函数。
- 空间复杂度: $O(m + n)$ 。

0400. 第 N 位数字

- 标签: 数学、二分查找
- 难度: 中等

题目链接

- [0400. 第 N 位数字 - 力扣](#)

题目大意

描述: 给你一个整数 n 。

要求: 在无限的整数序列 $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, \dots]$ 中找出并返回第 n 位上的数字。

说明:

- $1 \leq n \leq 2^{31} - 1$ 。

示例:

- 示例 1:

```
输入: n = 3
输出: 3
```

- 示例 2:

```
输入: n = 11
输出: 0
解释: 第 11 位数字在序列 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... 里是 0, 它是 10 的一部分。
```

解题思路

思路 1：找规律

数字以 0123456789101112131415... 的格式序列化到一个字符序列中。在这个序列中，第 5 位（从下标 0 开始计数）是 5，第 13 位是 1，第 19 位是 4，等等。

根据题意中的字符串，找数学规律：

- 1 位数字有 9 个，共 9 位: 123456789。
- 2 位数字有 90 个，共 2×90 位: 10111213...9899。
- 3 位数字有 900 个，共 3×900 位: 100...999。
- 4 位数字有 9000 个，共 4×9000 位: 1000...9999。
-

则我们可以按照以下步骤解决这道题：

- 我们可以先找到第 n 位所在整数 $number$ 所对应的位数 $digit$ 。

2. 同时找到该位数 $digit$ 的起始整数 $start$ 。
3. 再计算出 n 所在整数 $number$ 。 $number$ 等于从起始数字 $start$ 开始的第 $\lfloor \frac{n-1}{digit} \rfloor$ 个数字。即 $number = start + (n - 1) // digit$ 。
4. 然后确定 n 对应的是数字 $number$ 中的哪一位。即 $digit_idx = (n - 1) \bmod digit$ 。
5. 最后返回结果。

思路 1：代码

```
class Solution:
    def findNthDigit(self, n: int) -> int:
        digit = 1
        start = 1
        base = 9
        while n > base:
            n -= base
            digit += 1
            start *= 10
            base = start * digit * 9

        number = start + (n - 1) // digit
        digit_idx = (n - 1) % digit
        return int(str(number)[digit_idx])
```

思路 1：复杂度分析

- 时间复杂度: $O(\log n)$ 。
- 空间复杂度: $O(1)$ 。

思路 2：二分查找

假设第 n 位数字所在的整数是 $digit$ 位数，我们可以定义一个方法 $totalDigits(x)$ 用于计算所有位数不超过 x 的整数的所有位数和。

根据题意我们可知，所有位数不超过 $digit - 1$ 的整数的所有位数和一定小于 n ，并且所有不超过 $digit$ 的整数的所有位数和一定大于等于 n 。

因为所有位数不超过 x 的整数的所有位数和 $totalDigits(x)$ 是关于 x 单调递增的，所以我们可以使用二分查找的方式，确定第 n 位数字所在的整数的位数 $digit$ 。

n 的最大值为 $2^{31} - 1$ ，约为 2×10^9 。而 9 位数字有 9×10^8 个，共 $9 \times 9 \times 10^8 = 8.1 \times 10^9 > 2 \times 10^9$ ，所以第 n 位所在整数的位数 $digit$ 最多为 9 位，最小为 1 位。即 $digit$ 的取值范围为 $[1, 9]$ 。

我们使用二分查找算法得到 $digit$ 之后，还可以计算出不超过 $digit - 1$ 的整数的所有位数和 $pre_digits = totalDigits(digit - 1)$ ，则第 n 位数字所在整数在所有 $digit$ 位数中的下标是 $idx = n - pre_digits - 1$ 。

得到下标 idx 后，可以计算出 n 所在整数 $number$ 。 $number$ 等于从起始数字 $10^{digit-1}$ 开始的第 $\lfloor \frac{idx}{digit} \rfloor$ 个数字。即 $number = 10 ** (digit - 1) + idx // digit$ 。

该整数 $number$ 中第 $idx \bmod digit$ 即为第 n 位上的数字，将其作为答案返回即可。

思路 2：代码

```

class Solution:
    def totalDigits(self, x):
        digits = 0
        digit, cnt = 1, 9
        while digit <= x:
            digits += digit * cnt
            digit *= 10
            cnt *= 10
        return digits

    def findNthDigit(self, n: int) -> int:
        left, right = 1, 9
        while left < right:
            mid = left + (right - left) // 2
            if self.totalDigits(mid) < n:
                left = mid + 1
            else:
                right = mid

        digit = left
        pre_digits = self.totalDigits(digit - 1)
        idx = n - pre_digits - 1
        number = 10 ** (digit - 1) + idx // digit
        digit_idx = idx % digit

        return int(str(number)[digit_idx])

```

思路 2：复杂度分析

- 时间复杂度： $\log n \times \log \log n$ ，位数上限 D 为 $\log n$ ，二分查找的时间复杂度为 $\log D$ ，每次执行的时间复杂度为 D ，总的时间复杂度为 $D \times \log D = O(\log n \times \log \log n)$ 。
- 空间复杂度： $O(1)$ 。

参考资料

- 【题解】[400. 第 N 位数字 - 清晰易懂的找规律解法\(击败100%, 几乎双百\)](#)
- 【题解】[400. 第 N 位数字 - 方法一：二分查找](#)

0403. 青蛙过河

- 标签：数组、动态规划
- 难度：困难

题目链接

- [0403. 青蛙过河 - 力扣](#)

题目大意

描述：一只青蛙要过河，这条河被等分为若干个单元格，每一个单元格内可能放油一块石子（也可能没有）。青蛙只能跳到有石子的单元格内，不能跳到没有石子的单元格内。

现在给定一个严格按照升序排序的数组 $stones$ ，其中 $stones[i]$ 代表第 i 块石子所在的单元格序号。默认第 0 块石子序号为 0（即 $stones[0] == 0$ ）。

开始时，青蛙默认站在序号为 0 石子上（即 $stones[0]$ ），并且假定它第 1 步只能跳跃 1 个单位（即只能从序号为 0 的单元格跳到序号为 1 的单元格）。

如果青蛙在上一步向前跳跃了 k 个单位，则下一步只能向前跳跃 $k - 1$ 、 k 或者 $k + 1$ 个单位。

要求：判断青蛙能否成功过河（即能否在最后一步跳到最后一块石子上）。如果能，则返回 `True`；否则，则返回 `False`。

说明：

- $2 \leq stones.length \leq 2000$ 。
- $0 \leq stones[i] \leq 2^{31} - 1$ 。
- $stones[0] == 0$ 。
- $stones$ 按严格升序排列。

示例：

- **示例 1：**

```
输入: stones = [0, 1, 3, 5, 6, 8, 12, 17]
```

```
输出: true
```

解释：青蛙可以成功过河，按照如下方案跳跃：跳 1 个单位到第 2 块石子，然后跳 2 个单位到第 3 块石子，接着跳 2 个单位到第 4 块石子，然后跳 3 个单位到第 6 块石子。

解题思路

思路 1：动态规划

题目中说：如果青蛙在上一步向前跳跃了 k 个单位，则下一步只能向前跳跃 $k - 1$ 、 k 或者 $k + 1$ 个单位。则下一步的状态可以由 3 种状态转移而来。

- 上一步所在石子到下一步所在石头的距离为 $k - 1$ 。
- 上一步所在石子到下一步所在石头的距离为 k 。
- 上一步所在石子到下一步所在石头的距离为 $k + 1$ 。

则我们可以通过石子块数，跳跃距离来进行阶段划分和定义状态，以及推导状态转移方程。

1. 划分阶段

按照石子块数进行阶段划分。

2. 定义状态

定义状态 $dp[i][k]$ 表示为：青蛙能否以长度为 k 的距离，到达第 i 块石子。

3. 状态转移方程

1. 外层循环遍历每一块石子 i ，对于每一块石子 i ，使用内层循环遍历石子 i 之前所有的石子 j 。
2. 并计算出上一步所在石子 j 到当前所在石子 i 之间的距离为 k 。
3. 如果上一步所在石子 j 通过上一步以长度为 $k - 1$ 、 k 或者 $k + 1$ 的距离到达石子 j ，那么当前步所在石子也可以通过 k 的距离到达石子 i 。即通过检查 $dp[j][k - 1]$ 、 $dp[j][k]$ 、 $dp[j][k + 1]$ 中是否至少有一个为真，即可判断 $dp[i][k]$ 是否为真。
 - 即： $dp[i][k] = dp[j][k - 1] \text{ or } dp[j][k] \text{ or } dp[j][k + 1]$ 。

4. 初始条件

刚开始青蛙站在序号为 0 石子上（即 $stones[0]$ ），肯定能以长度为 0 的距离，到达第 0 块石子，即 $dp[0][0] = True$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][k]$ 表示为：青蛙能否以长度为 k 的距离，到达第 i 块石子。则如果 $dp[size - 1][k]$ 为真，则说明青蛙能成功过河（即能在最后一步跳到最后一块石子上）；否则则说明青蛙不能成功过河。

思路 1：动态规划代码

```

class Solution:
    def canCross(self, stones: List[int]) -> bool:
        size = len(stones)

        stone_dict = dict()
        for i in range(size):
            stone_dict[stones[i]] = i

        dp = [[False for _ in range(size + 1)] for _ in range(size)]
        dp[0][0] = True

        for i in range(1, size):
            for j in range(i):
                k = stones[i] - stones[j]
                if k <= 0 or k > j + 1:
                    continue

                dp[i][k] = dp[j][k - 1] or dp[j][k] or dp[j][k + 1]

            if dp[size - 1][k]:
                return True

        return False

```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。两重循环遍历的时间复杂度是 $O(n^2)$ ，所以总的时间复杂度为 $O(n^2)$ 。
- 空间复杂度： $O(n^2)$ 。用到了二维数组保存状态，所以总体空间复杂度为 $O(n^2)$ 。

参考资料

- [【题解】【403. 青蛙过河】理解理解动态规划与dfs - 青蛙过河 - 力扣# 0404. 左叶子之和](#)
- 标签：树、深度优先搜索、广度优先搜索、二叉树
- 难度：简单

题目链接

- [0404. 左叶子之和 - 力扣](#)

题目大意

给定一个二叉树，计算所有左叶子之和。

解题思路

深度优先搜索递归遍历二叉树，若当前节点不为空，且左孩子节点不为空，且左孩子节点的左右孩子节点都为空，则该节点的左孩子节点为左叶子节点。将其值累加起来，即为答案。

代码

```

class Solution:
    def sumOfLeftLeaves(self, root: TreeNode) -> int:
        self.ans = 0
        def dfs(node):
            if not node:
                return None
            if node.left and not node.left.left and not node.left.right:
                self.ans += node.left.val
            dfs(node.left)
            dfs(node.right)
        dfs(root)
        return self.ans

```

0405. 数字转换为十六进制数

- 标签：位运算、数学
- 难度：简单

题目链接

- [0405. 数字转换为十六进制数 - 力扣](#)

题目大意

描述：给定一个整数 num 。

要求：编写一个算法将这个数转换为十六进制数。对于负整数，我们通常使用「补码运算」方法。

说明：

- 十六进制中所有字母 ($a \sim f$) 都必须是小写。
- 十六进制字符串中不能包含多余的前导零。如果要转化的数为 0，那么以单个字符 0 来表示。
- 对于其他情况，十六进制字符串中的第一个字符将不会是 0 字符。
- 给定的数确保在 32 位有符号整数范围内。
- 不能使用任何由库提供的将数字直接转换或格式化为十六进制的方法。

示例：

- **示例 1：**

输入：
26

输出：
"1a"

- **示例 2：**

输入：
-1

输出：
"ffffffff"

解题思路

思路 1：模拟

主要是对不同情况的处理。

- 当 num 为 0 时，直接返回 0。
- 当 num 为负数时，对负数进行「补码运算」，转换为对应的十进制正数（将其绝对值与 $2^{32} - 1$ 异或再加 1），然后执行和 $nums$ 为正数一样的操作。
- 当 num 为正数时，将其对 16 取余，并转为对应的十六进制字符，并按位拼接到字符串中，再将 num 除以 16，继续对 16 取余，直到 num 变为 0。
- 最后将拼接好的字符串逆序返回就是答案。

思路 1：代码

```
class Solution:
    def toHex(self, num: int) -> str:
        res = ''
        if num == 0:
            return '0'

        if num < 0:
            num = (abs(num) ^ (2 ** 32 - 1)) + 1

        while num:
            digit = num % 16
            if digit >= 10:
                digit = chr(ord('a') + digit - 10)
            else:
                digit = str(digit)
            res += digit
            num >>= 4
        return res[::-1]
```

思路 1：复杂度分析

- 时间复杂度： $O(C)$ ，其中 C 为构造的十六进制数的长度。
- 空间复杂度： $O(C)$ 。

0406. 根据身高重建队列

- 标签：贪心、树状数组、线段树、数组、排序
- 难度：中等

题目链接

- [0406. 根据身高重建队列 - 力扣](#)

题目大意

n 个人打乱顺序排成一排，给定一个数组 $people$ 表示队列中人的属性（顺序是打乱的）。其中 $people[i] = [h_i, k_i]$ 表示第 i 个人的身高为 h_i ，前面正好有 k_i 个身高大于或等于 h_i 的人。

现在重新构造并返回输入数组 $people$ 所表示的队列 $queue$ 。其中 $queue[j] = [h_j, k_j]$ 是队列中第 j 个人的信息，表示为身高为 h_j ，前面正好有 k_j 个身高大于或等于 h_j 的人。

解题思路

这道题目有两个维度，身高 h_j 和满足条件的数量 k_j 。进行排序的时候如果同时考虑两个维度条件，就有点复杂了。我们可以考虑固定一个维度，先排好序，再考虑另一个维度的要求。

我们可以先确定身高维度。将数组按身高从高到低进行排序，身高相同的则按照 k 值升序排列。这样排序之后可以确定目前对于第 j 个人来说，前面的 $j - 1$ 个人肯定比他都高。

然后建立一个包含 n 个位置的空队列 $queue$ ，按照上边排好的顺序遍历，依次将其插入到第 k_j 位置上。最后返回新的队列。

代码

```
class Solution:
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        queue = []
        people.sort(key = lambda x: (-x[0], x[1]))
        for p in people:
            queue.insert(p[1], p)
        return queue
```

0409. 最长回文串

- 标签：贪心、哈希表、字符串
- 难度：简单

题目链接

- [0409. 最长回文串 - 力扣](#)

题目大意

给定一个包含大写字母和小写字母的字符串 s 。

要求：找到通过这些字母构造成的最长的回文串。

注意：

- 在构造过程中，请注意区分大小写。比如 Aa 不能当做一个回文字符串。
- 假设字符串的长度不会超过 1010 。

解题思路

这道题目是通过给定字母构造回文串，并找到最长的回文串长度。那就要先看看回文串的特点。在回文串中，最多只有一个字母出现过奇数次，其余字符都出现过偶数次。且相同字母是中心对称的。

则我们可以用哈希表统计字符出现次数。对于每个字符，使用尽可能多的偶数次字符作为回文串的两侧，并记录下使用的字符个数，记录到答案中。再使用一个 $flag$ 标记下是否有奇数次的字符，如果有的话，最终答案再加 1。最后输出答案。

代码

```

class Solution:
    def longestPalindrome(self, s: str) -> int:
        word_dict = dict()
        for ch in s:
            if ch in word_dict:
                word_dict[ch] += 1
            else:
                word_dict[ch] = 1

        ans = 0
        flag = False
        for value in word_dict.values():
            ans += value // 2 * 2
            if value % 2 == 1:
                flag = True

        if flag:
            ans += 1

        return ans

```

0410. 分割数组的最大值

- 标签：贪心、数组、二分查找、动态规划、前缀和
- 难度：困难

题目链接

- [0410. 分割数组的最大值 - 力扣](#)

题目大意

描述：给定一个非负整数数组 $nums$ 和一个整数 k ，将数组分成 m 个非空的连续子数组。

要求：使 m 个子数组各自和的最大值最小，并求出子数组各自和的最大值。

说明：

- $1 \leq nums.length \leq 1000$ 。
- $0 \leq nums[i] \leq 10^6$ 。
- $1 \leq k \leq \min(50, nums.length)$ 。

示例：

- **示例 1：**

输入: $nums = [7, 2, 5, 10, 8]$, $k = 2$

输出: 18

解释:

一共有四种方法将 $nums$ 分割为 2 个子数组。

其中最好的方式是将其分为 $[7, 2, 5]$ 和 $[10, 8]$ 。

因为此时这两个子数组各自的和的最大值为 18，在所有情况中最小。

- **示例 2：**

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

解题思路

思路 1：二分查找算法

先来理解清楚题意。题目的目的是使得 m 个连续子数组各自和的最大值最小。意思是将数组按顺序分成 m 个子数组，然后计算每个子数组的和，然后找出 m 个和中的最大值，要求使这个最大值尽可能小。最后输出这个尽可能小的和最大值。

可以用二分查找来找这个子数组和的最大值，我们用 ans 来表示这个值。 ans 最小为数组 $nums$ 所有元素的最大值，最大为数组 $nums$ 所有元素的和。即 ans 范围是 $[max(nums), sum(nums)]$ 。

所以就确定了二分查找的两个指针位置。 $left$ 指向 $max(nums)$, $right$ 指向 $sum(nums)$ 。然后取中间值 mid , 计算当子数组和的最大值为 mid 时，所需要分割的子数组最少个数。

- 如果需要分割的子数组最少个数大于 m 个，则说明子数组和的最大值取小了，不满足条件，应该继续调大，将 $left$ 右移，从右区间继续查找。
- 如果需要分割的子数组最少个数小于或等于 m 个，则说明子数组和的最大值满足条件，并且还可以继续调小，将 $right$ 左移，从左区间继续查找，看是否有更小的数组和满足条件。
- 最终，返回符合条件的最小值即可。

思路 1：代码

```
class Solution:
    def splitArray(self, nums: List[int], m: int) -> int:
        def get_count(x):
            total = 0
            count = 1
            for num in nums:
                if total + num > x:
                    count += 1
                    total = num
                else:
                    total += num
            return count

        left = max(nums)
        right = sum(nums)
        while left < right:
            mid = left + (right - left) // 2
            if get_count(mid) > m:
                left = mid + 1
            else:
                right = mid
        return left
```

思路 1：复杂度分析

- 时间复杂度: $O(n \times \log(\sum nums))$, 其中 n 为数组中的元素个数。
- 空间复杂度: $O(1)$ 。

0412. Fizz Buzz

- 标签: 数学、字符串、模拟
- 难度: 简单

题目链接

- 0412. Fizz Buzz - 力扣

题目大意

给定一个整数 n，按照规则，输出 1~n 的字符串表示。

规则：

- 如果 i 是 3 的倍数，输出 "Fizz"；
- 如果 i 是 5 的倍数，输出 "Buzz"；
- 如果 i 是 3 和 5 的倍数，则输出 "FizzBuzz"。

解题思路

简单题，按照题目规则输出即可。

代码

```
class Solution:
    def fizzBuzz(self, n: int) -> List[str]:
        ans = []
        for i in range(1, n+1):
            if i % 15 == 0:
                ans.append("FizzBuzz")
            elif i % 3 == 0:
                ans.append("Fizz")
            elif i % 5 == 0:
                ans.append("Buzz")
            else:
                ans.append(str(i))
        return ans
```

0415. 字符串相加

- 标签：数学、字符串、模拟
- 难度：简单

题目链接

- 0415. 字符串相加 - 力扣

题目大意

描述：给定两个字符串形式的非负整数 num1 和 num2。

要求：计算它们的和，并同样以字符串形式返回。

说明：

- $1 \leq \text{num1.length}, \text{num2.length} \leq 10^4$ 。
- num1 和 num2 都只包含数字 0 ~ 9。
- num1 和 num2 都不包含任何前导零。
- 你不能使用任何内建 BigInteger 库，也不能直接将输入的字符串转换为整数形式。

示例：

- 示例 1：

```
输入: num1 = "11", num2 = "123"
输出: "134"
```

- 示例 2：

```
输入: num1 = "456", num2 = "77"
输出: "533"
```

解题思路

思路 1：双指针

需要用字符串的形式来模拟大数加法。

加法的计算方式是：从个位数开始，由低位到高位，按位相加，如果相加之后超过 10，就需要向前进位。

模拟加法的做法是：

- 用一个数组存储按位相加后的结果，每一位对应一位数。
- 然后分别使用一个指针变量，对两个数 `num1`、`num2` 字符串进行反向遍历，将相加后的各个位置上的结果保存在数组中，这样计算完成之后就得到了一个按位反向的结果。
- 最后返回结果的时候将数组反向转为字符串即可。

注意需要考虑 `num1`、`num2` 不等长的情况，让短的那个字符串对应位置按 0 计算即可。

思路 1：代码

```
class Solution:
    def addStrings(self, num1: str, num2: str) -> str:
        # num1 位数
        digit1 = len(num1) - 1
        # num2 位数
        digit2 = len(num2) - 1

        # 进位
        carry = 0
        # sum 存储反向结果
        sum = []
        # 逆序相加
        while carry > 0 or digit1 >= 0 or digit2 >= 0:
            # 获取对应位数上的数字
            num1_d = int(num1[digit1]) if digit1 >= 0 else 0
            num2_d = int(num2[digit2]) if digit2 >= 0 else 0
            digit1 -= 1
            digit2 -= 1
            # 计算结果，存储，进位
            num = num1_d + num2_d + carry
            sum.append('%d' % (num % 10))
            carry = num // 10
        # 返回计算结果
        return ''.join(sum[::-1])
```

思路 1：复杂度分析

- 时间复杂度： $O(\max(m + n))$ 。其中 m 是字符串 `num1` 的长度， n 是字符串 `num2` 的长度。
- 空间复杂度： $O(\max(m + n))$ 。# 0416. 分割等和子集

- 标签: 数组、动态规划
- 难度: 中等

题目链接

- 0416. 分割等和子集 - 力扣

题目大意

描述: 给定一个只包含正整数的非空数组 $nums$ 。

要求: 判断是否可以将这个数组分成两个子集，使得两个子集的元素和相等。

说明:

- $1 \leq nums.length \leq 200$ 。
- $1 \leq nums[i] \leq 100$ 。

示例:

- **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。
```

解题思路

思路 1：动态规划

这道题换一种说法就是：从数组中选择一些元素组成一个子集，使子集的元素和恰好等于整个数组元素和的一半。

这样的话，这道题就可以转变为「0-1 背包问题」。

1. 把整个数组中的元素和记为 sum ，把元素和的一半 $target = \frac{sum}{2}$ 看做是「0-1 背包问题」中的背包容量。
2. 把数组中的元素 $nums[i]$ 看做是「0-1 背包问题」中的物品。
3. 第 i 件物品的重量为 $nums[i]$ ，价值也为 $nums[i]$ 。
4. 因为物品的重量和价值相等，如果能装满载重上限为 $target$ 的背包，那么得到的最大价值也应该是 $target$ 。

这样问题就转变为：给定一个数组 $nums$ 代表物品，数组元素和的一半 $target = \frac{sum}{2}$ 代表背包的载重上限。其中第 i 件物品的重量为 $nums[i]$ ，价值为 $nums[i]$ ，每件物品有且只有 1 件。请问在总重量不超过背包装载重量上限的情况下，能否将背包装满从而得到最大价值？

1. 划分阶段

当前背包的载重上限进行阶段划分。

2. 定义状态

定义状态 $dp[w]$ 表示为：从数组 $nums$ 中选择一些元素，放入最多能装元素和为 w 的背包中，得到的元素和最大为多少。

3. 状态转移方程

$$dp[w] = \begin{cases} dp[w] & w < nums[i-1] \\ \max\{dp[w], dp[w - nums[i-1]] + nums[i-1]\} & w \geq nums[i-1] \end{cases}$$

4. 初始条件

- 无论背包载重上限为多少，只要不选择物品，可以获得的最大价值一定是 0，即 $dp[w] = 0, 0 \leq w \leq W$ 。

5. 最终结果

根据我们之前定义的状态， $dp[target]$ 表示为：从数组 $nums$ 中选择一些元素，放入最多能装元素和为 $target = \frac{sum}{2}$ 的背包中，得到的元素和最大值。

所以最后判断一下 $dp[target]$ 是否等于 $target$ 。如果 $dp[target] == target$ ，则说明集合中的子集刚好能够凑成总和 $target$ ，此时返回 `True`；否则返回 `False`。

思路 1：代码

```
class Solution:
    # 思路 2: 动态规划 + 滚动数组优化
    def zeroOnePackMethod2(self, weight: [int], value: [int], W: int):
        size = len(weight)
        dp = [0 for _ in range(W + 1)]

        # 枚举前 i 种物品
        for i in range(1, size + 1):
            # 逆序枚举背包载重量 (避免状态值错误)
            for w in range(W, weight[i - 1] - 1, -1):
                # dp[w] 取「前 i - 1 件物品装入载重为 w 的背包中的最大价值」与「前 i - 1 件物品装入载重为 w - weight[i - 1] 的背包中，再装入第 i - 1 件物品」中的较大值
                dp[w] = max(dp[w], dp[w - weight[i - 1]] + value[i - 1])

        return dp[W]

    def canPartition(self, nums: List[int]) -> bool:
        sum_nums = sum(nums)
        if sum_nums & 1:
            return False

        target = sum_nums // 2
        return self.zeroOnePackMethod2(nums, nums, target) == target
```

思路 1：复杂度分析

- 时间复杂度： $O(n \times target)$ ，其中 n 为数组 $nums$ 的元素个数， $target$ 是整个数组元素和的一半。
- 空间复杂度： $O(target)$ 。

0417. 太平洋大西洋水流问题

- 标签：深度优先搜索、广度优先搜索、数组、矩阵
- 难度：中等

题目链接

- [0417. 太平洋大西洋水流问题 - 力扣](#)

题目大意

描述：给定一个 $m * n$ 大小的二维非负整数矩阵 $heights$ 来表示一片大陆上各个单元格的高度。 $heights[i][j]$ 表示第 i 行第 j 列所代表的陆地高度。这个二维矩阵所代表的陆地被太平洋和大西洋所包围着。左上角是「太平洋」，右下角是「大西洋」。规定水流只能按照上、下、左、右四个方向流动，且只能从高处流到低处，或者在同等高度上流动。

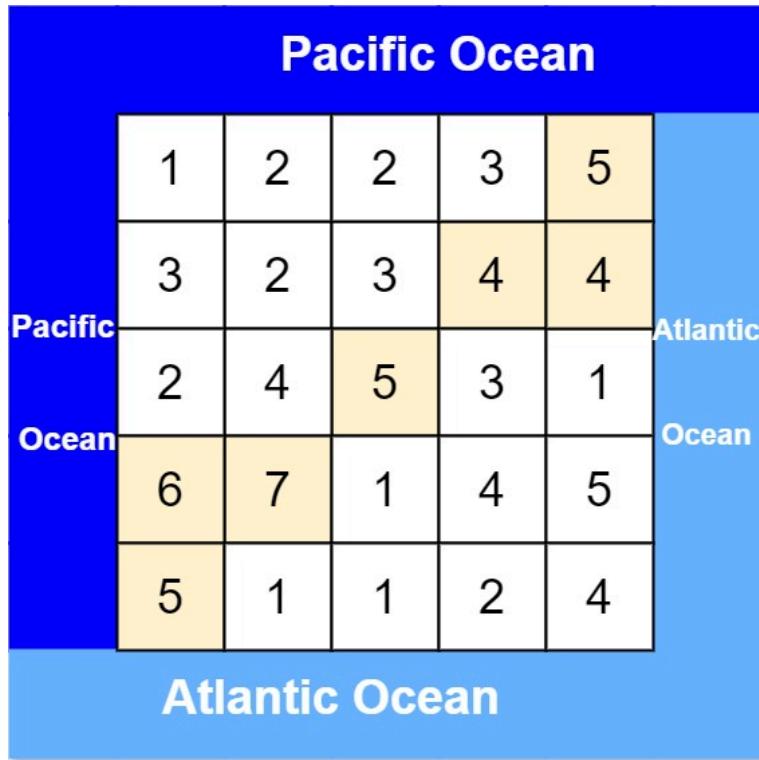
要求：找出代表陆地的二维矩阵中，水流既可以从该处流动到太平洋，又可以流动到大西洋的所有坐标。以二维数组 res 的形式返回，其中 $res[i] = [ri, ci]$ 表示雨水从单元格 (ri, ci) 既可流向太平洋也可流向大西洋。

说明：

- $m == heights.length$ 。
- $n == heights[r].length$ 。
- $1 \leq m, n \leq 200$ 。
- $0 \leq heights[r][c] \leq 10^5$ 。

示例：

- 示例 1：



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

- 示例 2：

```
输入: heights = [[2,1],[1,2]]
输出: [[0,0],[0,1],[1,0],[1,1]]
```

解题思路

思路 1：深度优先搜索

雨水由高处流向低处，如果我们根据雨水的流向搜索，来判断是否能从某一位置流向太平洋和大西洋不太容易。我们可以换个思路。

1. 分别从太平洋和大西洋（就是矩形边缘）出发，逆流而上，找出水流逆流能达到的地方，可以用两个二维数组 `pacific`、`atlantic` 分别记录太平洋和大西洋能到达的位置。
2. 然后再对二维数组进行一次遍历，找出两者交集的位置，就是雨水既可流向太平洋也可流向大西洋的位置，将其加入答案数组 `res` 中。
3. 最后返回答案数组 `res`。

思路 1：代码

```

class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        rows, cols = len(heights), len(heights[0])
        pacific = [[False for _ in range(cols)] for _ in range(rows)]
        atlantic = [[False for _ in range(cols)] for _ in range(rows)]

        directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]

        def dfs(i, j, visited):
            visited[i][j] = True
            for direct in directs:
                new_i = i + direct[0]
                new_j = j + direct[1]
                if new_i < 0 or new_i >= rows or new_j < 0 or new_j >= cols:
                    continue
                if heights[new_i][new_j] >= heights[i][j] and not visited[new_i][new_j]:
                    dfs(new_i, new_j, visited)

        for j in range(cols):
            dfs(0, j, pacific)
            dfs(rows - 1, j, atlantic)

        for i in range(rows):
            dfs(i, 0, pacific)
            dfs(i, cols - 1, atlantic)

        res = []
        for i in range(rows):
            for j in range(cols):
                if pacific[i][j] and atlantic[i][j]:
                    res.append([i, j])
        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(m \times n)$ 。其中 m 和 n 分别为行数和列数。
- 空间复杂度: $O(m \times n)$ 。[# 0421. 数组中两个数的最大异或值](#)
- 标签: 位运算、字典树、数组、哈希表
- 难度: 中等

题目链接

- [# 0421. 数组中两个数的最大异或值 - 力扣](#)

题目大意

给定一个整数数组 `nums`。

要求: 返回 `num[i] XOR num[j]` 的最大运算结果。其中 $0 \leq i \leq j < n$ 。

解题思路

最直接的想法暴力求解。两层循环计算两两之间的异或结果，记录并更新最大异或结果。

更好的做法可以减少一重循环。首先，要取得异或结果的最大值，那么从二进制的高位到低位，尽可能的让每一位异或结果都为 1。

将数组中所有数字的二进制形式从高位到低位依次存入字典树中。然后是利用异或运算交换律：如果 `a ^ b = max` 成立，那么 `a ^ max = b` 与 `b ^ max = a` 均成立。这样当我们知道 `a` 和 `max` 时，可以通过交换律求出 `b`。`a` 是我们遍历的每一个数，`max` 是我们想要尝试的最大值，从

111111... 开始，从高位到低位依次填 1。

对于 `a` 和 `max`，如果我们所求的 `b` 也在字典树中，则表示 `max` 是可以通过 `a` 和 `b` 得到的，那么 `max` 就是所求最大的异或。如果 `b` 不在字典树中，则减小 `max` 值继续判断，或者继续查询下一个 `a`。

代码

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """

        self.children = dict()
        self.isEnd = False

    def insert(self, num: int, max_bit: int) -> None:
        """
        Inserts a word into the trie.
        """

        cur = self
        for i in range(max_bit, -1, -1):
            bit = num >> i & 1
            if bit not in cur.children:
                cur.children[bit] = Trie()
            cur = cur.children[bit]
        cur.isEnd = True

    def search(self, num: int, max_bit: int) -> int:
        """
        Returns if the word is in the trie.
        """

        cur = self
        res = 0
        for i in range(max_bit, -1, -1):
            bit = num >> i & 1
            if 1 - bit not in cur.children:
                res = res * 2
                cur = cur.children[bit]
            else:
                res = res * 2 + 1
                cur = cur.children[1 - bit]
        return res

class Solution:

    def findMaximumXOR(self, nums: List[int]) -> int:
        trie_tree = Trie()
        max_bit = len(format(max(nums), 'b')) - 1
        ans = 0
        for num in nums:
            trie_tree.insert(num, max_bit)
            ans = max(ans, trie_tree.search(num, max_bit))

        return ans
```

0424. 替换后的最长重复字符

- 标签：哈希表、字符串、滑动窗口
- 难度：中等

题目链接

- 0424. 替换后的最长重复字符 - 力扣

题目大意

描述：给定一个仅由大写英文字母组成的字符串 s ，以及一个整数 k 。可以将任意位置上的字符替换成另外的大写字母，最多可替换 k 次。

要求：在进行上述操作后，找到包含重复字母的最长子串长度。

说明：

- $1 \leq s.length \leq 10^5$ 。
- s 仅由大写英文字母组成。
- $0 \leq k \leq s.length$ 。

示例：

- 示例 1：

```
输入: s = "ABAB", k = 2
输出: 4
解释: 用两个 'A' 替换为两个 'B' , 反之亦然。
```

- 示例 2：

```
输入: s = "AABABBA", k = 1
输出: 4
解释:
将中间的一个 'A' 替换为 'B' , 字符串变为 "AABBBA"。
子串 "BBBB" 有最长重复字母, 答案为 4。
可能存在其他的方法来得到同样的结果。
```

解题思路

先来考虑暴力求法。枚举字符串 s 的所有子串，对于每一个子串：

- 统计子串中出现次数最多的字符，替换除它以外的字符 k 次。
- 维护最长子串的长度。

但是这种暴力求法中，枚举子串的时间复杂度为 $O(n^2)$ ，统计出现次数最多的字符和替换字符时间复杂度为 $O(n)$ ，且两者属于平行处理，总体下来的时间复杂度为 $O(n^3)$ 。这样做会超时。

思路 1：滑动窗口

1. 使用 `counts` 数组来统计字母频数。使用 `left`、`right` 双指针分别指向滑动窗口的首尾位置，使用 `max_count` 来维护最长子串的长度。
2. 不断右移 `right` 指针，增加滑动窗口的长度。
3. 对于当前滑动窗口的子串，如果当前窗口的间距 > 当前出现最大次数的字符的次数 + k 时，意味着替换 k 次仍不能使当前窗口中的字符全变为相同字符，则此时应该将左边界右移，同时将原先左界的字符频次减少。

思路 1：代码

```

class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        max_count = 0
        left, right = 0, 0
        counts = [0 for _ in range(26)]
        while right < len(s):
            num_right = ord(s[right]) - ord('A')
            counts[num_right] += 1
            max_count = max(max_count, counts[num_right])
            right += 1
            if right - left > max_count + k:
                num_left = ord(s[left]) - ord('A')
                counts[num_left] -= 1
                left += 1

        return right - left

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为字符串的长度。
- 空间复杂度: $O(|\Sigma|)$, 其中 Σ 是字符集, 本题中 $|\Sigma| = 26$ 。

0425. 单词方块

- 标签: 字典树、数组、字符串、回溯
- 难度: 困难

题目链接

- [0425. 单词方块 - 力扣](#)

题目大意

给定一个单词集合 `words` (没有重复)。

要求: 找出其中所有的单词方块。

- 单词方块: 指从第 `k` 行和第 `k` 列 ($0 \leq k < \max(\text{行数}, \text{列数})$) 来看都是相同的字符串。

例如, 单词序列 `["ball", "area", "lead", "lady"]` 形成了一个单词方块, 因为每个单词从水平方向看和从竖直方向看都是相同的。

```

b a l l
a r e a
l e a d
l a d y

```

解题思路

根据单词方块的第一个单词, 可以推出下一个单词的前缀。

比如第一个单词是 `ball`, 那么单词方块的长度是 `4 * 4`, 则下一个单词 (第二个单词) 的前缀为 `a`。这样我们就又找到了一个以 `a` 为前缀且长度为 `4` 的单词, 即 `area`, 此时就变成了 `[ball, area]`。

那么下一个单词 (第三个单词) 的前缀为 `le`。这样我们就又找到了一个以 `le` 为前缀且长度为 `4` 的单词, 即 `lead`。此时就变成了 `[ball, area, lead]`。

以此类推，就可以得到整个单词方块。

并且我们可以使用字典树（前缀树）来存储单词，并且通过回溯得到所有的解。

代码

```

class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.children = dict()
        self.isEnd = False

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
        cur = self
        for ch in word:
            if ch not in cur.children:
                cur.children[ch] = Trie()
            cur = cur.children[ch]
        cur.isEnd = True

    def search(self, word: str):
        """
        Returns if the word is in the trie.
        """
        cur = self
        res = []
        for ch in word:
            if ch not in cur.children:
                return res
            cur = cur.children[ch]
        cur.dfs(word, res)
        return res

    def dfs(self, word, res):
        cur = self
        if cur and cur.isEnd:
            res.append(word)
            return
        for ch in cur.children:
            node = cur.children[ch]
            node.dfs(word + ch, res)

class Solution:

    def backtrace(self, index, size, path, res, trie_tree):
        if index == size:
            res.append(path[:])
            return
        next_prefix = "" # 下一行的前缀
        for i in range(index):
            next_prefix += path[i][index]

        next_words_with_prefix = trie_tree.search(next_prefix)
        for word in next_words_with_prefix:
            path.append(word)
            self.backtrace(index + 1, size, path, res, trie_tree)
            path.pop(-1)

```

```

def wordSquares(self, words: List[str]) -> List[List[str]]:
    trie_tree = Trie()
    for word in words:
        trie_tree.insert(word)
    size = len(words[0])
    res = []
    path = []
    for word in words:
        path.append(word)
        self.backtrace(1, size, path, res, trie_tree)
        path.pop(-1)
    return res

```

0426. 将二叉搜索树转化为排序的双向链表

- 标签：栈、树、深度优先搜索、二叉搜索树、链表、二叉树、双向链表
- 难度：中等

题目链接

- [0426. 将二叉搜索树转化为排序的双向链表 - 力扣](#)

题目大意

给定一棵二叉树的根节点 `root`。

要求：将这棵二叉树转换为一个已排序的双向循环链表。要求不能创建新的节点，只能调整树中节点指针的指向。

解题思路

通过中序递归遍历可以将二叉树升序排列输出。这道题需要在中序遍历的同时，将节点的左右指向进行改变。使用 `head`、`tail` 存放双向链表的头尾节点，然后从根节点开始，进行中序递归遍历。

具体做法如下：

- 如果当前节点为空，直接返回。
- 如果当前节点不为空：
 - 递归遍历左子树。
 - 如果尾节点不为空，则将尾节点与当前节点进行连接。
 - 如果尾节点为空，则初始化头节点。
 - 将当前节点标记为尾节点。
 - 递归遍历右子树。
- 最后将头节点和尾节点进行连接。

代码

```

class Solution:
    def treeToDoublyList(self, root: 'Node') -> 'Node':
        def dfs(node: 'Node'):
            if not node:
                return

            dfs(node.left)
            if self.tail:
                self.tail.right = node
                node.left = self.tail
            else:
                self.head = node
            self.tail = node
            dfs(node.right)

        if not root:
            return None

        self.head, self.tail = None, None
        dfs(root)
        self.head.left = self.tail
        self.tail.right = self.head
        return self.head

```

0428. 序列化和反序列化 N 叉树

- 标签: 树、深度优先搜索、广度优先搜索、字符串
- 难度: 困难

题目链接

- [0428. 序列化和反序列化 N 叉树 - 力扣](#)

题目大意

要求: 设计一个序列化和反序列化 N 叉树的算法。序列化 / 反序列化算法的算法实现没有限制。你只需要保证 N 叉树可以被序列化为一个字符串并且该字符串可以被反序列化成原树结构即可。

- 序列化是指将一个数据结构转化为位序列的过程，因此可以将其存储在文件中或内存缓冲区中，以便稍后在相同或不同的计算机环境中恢复结构。
- N 叉树是指每个节点都有不超过 N 个孩子节点的有根树。

解题思路

- 序列化: 通过深度优先搜索的方式，递归遍历节点，以 `root.val`、`len(root.children)`、`root.children` 的顺序生成序列化结果，并用 `-` 链接，返回结果字符串。
- 反序列化: 先将字符串按 `-` 分割成数组。然后按照 `root.val`、`len(root.children)`、`root.children` 的顺序解码，并建立对应节点。最后返回根节点。

代码

```

class Codec:
    def serialize(self, root: 'Node') -> str:
        """Encodes a tree to a single string.

        :type root: Node
        :rtype: str
        """
        if not root:
            return 'None'

        data = str(root.val) + '-' + str(len(root.children))
        for child in root.children:
            data += '-' + self.serialize(child)
        return data

    def deserialize(self, data: str) -> 'Node':
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: Node
        """
        dataList = data.split('-')
        return self.dfs(dataList)

    def dfs(self, dataList):
        val = dataList.pop(0)
        if val == 'None':
            return None
        root = Node(int(val))
        root.children = []

        size = int(dataList.pop(0))
        for _ in range(size):
            root.children.append(self.dfs(dataList))
        return root

```

0429. N 叉树的层序遍历

- 标签: 树、广度优先搜索
- 难度: 中等

题目链接

- [0429. N 叉树的层序遍历 - 力扣](#)

题目大意

给定一个 N 叉树的根节点 `root`。

要求: 返回其节点值的层序遍历 (即从左到右, 逐层遍历)。

树的序列化输入是用层序遍历, 每组子节点都由 `null` 值分隔。

解题思路

和二叉树的层序遍历类似。广度优先搜索每次取出第 `i` 层上所有元素。具体步骤如下:

- 根节点入队。
- 当队列不为空时，求出当前队列长度 `size`。
 - 依次从队列中取出这 `size` 个元素，并将元素值存入当前层级列表 `level` 中。
 - 将该层所有节点的所有孩子节点入队，遍历完之后将这层节点数组加入答案数组中，然后继续迭代。
- 当队列为空时，结束。

代码

```
class Solution:
    def levelOrder(self, root: 'Node') -> List[List[int]]:
        ans = []
        if not root:
            return ans

        queue = [root]

        while queue:
            level = []
            size = len(queue)
            for _ in range(size):
                cur = queue.pop(0)
                level.append(cur.val)
                for child in cur.children:
                    queue.append(child)
            ans.append(level)

        return ans
```

0430. 扁平化多级双向链表

- 标签：深度优先搜索、链表、双向链表
- 难度：中等

题目链接

- [0430. 扁平化多级双向链表 - 力扣](#)

题目大意

给定一个带子链表指针 `child` 的双向链表，将 `child` 的子链表进行扁平化处理，使所有节点出现在单级双向链表中。

扁平化处理如下：

```
原链表：
1---2---3---4---5---6---NULL
 |
7---8---9---10---NULL
 |
11---12---NULL

扁平化之后：
1---2---3---7---8---11---12---9---10---4---5---6---NULL
```

解题思路

递归处理多层链表的扁平化。遍历链表，找到 `child` 非空的节点，将其子链表链接到当前节点的 `next` 位置（自身扁平化处理）。然后继续向后遍历，不断找到 `child` 节点，并进行链接。直到处理到尾部位置。

代码

```

class Solution:
    def dfs(self, node: 'Node'):
        # 找到链表的尾节点或 child 链表不为空的节点
        while node.next and not node.child:
            node = node.next
        tail = None
        if node.child:
            # 如果 child 链表不为空, 将 child 链表扁平化
            tail = self.dfs(node.child)

            # 将扁平化的 child 链表链接在该节点之后
            temp = node.next
            node.next = node.child
            node.next.prev = node
            node.child = None
            tail.next = temp
            if temp:
                temp.prev = tail
        # 链接之后, 从 child 链表的尾节点继续向后处理链表
        return self.dfs(tail)
    # child 链表为空, 则该节点是尾节点, 直接返回
    return node
def flatten(self, head: 'Node') -> 'Node':
    if not head:
        return head
    self.dfs(head)
    return head

```

0435. 无重叠区间

- 标签: 贪心、数组、动态规划、排序
- 难度: 中等

题目链接

- [0435. 无重叠区间 - 力扣](#)

题目大意

描述: 给定一个区间的集合 `intervals`，其中 `intervals[i] = [starti, endi]`。从集合中移除部分区间，使得剩下的区间互不重叠。

要求: 返回需要移除区间的最小数量。

说明:

- $1 \leq \text{intervals.length} \leq 10^5$ 。
- $\text{intervals}[i].length == 2$ 。
- $-5 * 10^4 \leq \text{starti} < \text{endi} \leq 5 * 10^4$ 。

示例:

- 示例 1:

```

输入: intervals = [[1,2],[2,3],[3,4],[1,3]]
输出: 1
解释: 移除 [1,3] 后, 剩下的区间没有重叠。

```

- 示例 2:

```
输入: intervals = [ [1,2], [1,2], [1,2] ]
输出: 2
解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。
```

解题思路

思路 1：贪心算法

这道题我们可以转换一下思路。原题要求保证移除区间最少，使得剩下的区间互不重叠。换个角度就是：「如何使得剩下互不重叠区间的数目最多」。那么答案就变为了：「总区间个数 - 不重叠区间的最多个数」。我们的问题也变成了求所有区间中不重叠区间的最多个数。

从贪心算法的角度来考虑，我们应该将区间按照结束时间排序。每次选择结束时间最早的区间，然后再在剩下的时间内选出最多的区间。

我们用贪心三部曲来解决这道题。

- 转换问题：**将原问题转变为，当选择结束时间最早的区间之后，再在剩下的时间内选出最多的区间（子问题）。
- 贪心选择性质：**每次选择时，选择结束时间最早的区间。这样选出来的区间一定是原问题最优解的区间之一。
- 最优子结构性质：**在上面的贪心策略下，贪心选择当前时间最早的区间 + 剩下的时间内选出最多区间的子问题最优解，就是全局最优解。也就是说在贪心选择的方案下，能够使所有区间中不重叠区间的个数最多。

使用贪心算法的代码解决步骤描述如下：

- 将区间集合按照结束坐标升序排列，然后维护两个变量，一个是当前不重叠区间的结束时间 `end_pos`，另一个是不重叠区间的个数 `count`。初始情况下，结束坐标 `end_pos` 为第一个区间的结束坐标，`count` 为 1。
- 依次遍历每段区间。对于每段区间：`intervals[i]`：
 - 如果 `end_pos <= intervals[i][0]`，即 `end_pos` 小于等于区间起始位置，则说明出现了不重叠区间，令不重叠区间数 `count` 加 1，`end_pos` 更新为新区间的结束位置。
- 最终返回「总区间个数 - 不重叠区间的最多个数」即 `len(intervals) - count` 作为答案。

思路 1：代码

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        if not intervals:
            return 0
        intervals.sort(key=lambda x: x[1])
        end_pos = intervals[0][1]
        count = 1
        for i in range(1, len(intervals)):
            if end_pos <= intervals[i][0]:
                count += 1
                end_pos = intervals[i][1]

        return len(intervals) - count
```

思路 1：复杂度分析

- 时间复杂度： $O(n \times \log n)$ ，其中 n 是区间的数量。
- 空间复杂度： $O(\log n)$ 。

0437. 路径总和 III

- 标签：树、深度优先搜索、二叉树
- 难度：中等

题目链接

- 0437. 路径总和 III - 力扣

题目大意

给定一个二叉树的根节点 `root`，和一个整数 `sum`。

要求：求出该二叉树里节点值之和等于 `sum` 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）。

解题思路

直观想法是：

以每一个节点 `node` 为起始节点，向下检测延伸的路径。递归遍历每一个节点所有可能的路径，然后将这些路径数目加起来即为答案。

但是这样会存在许多重复计算。我们可以定义节点的前缀和来减少重复计算。

- 节点的前缀和：从根节点到当前节点路径上所有节点的和。

有了节点的前缀和，我们就可以通过前缀和来计算两节点之间的路劲和。即： 则两节点之间的路径和 = 两节点之间的前缀和之差。

为了计算符合要求的路径数量，我们用哈希表存储「前缀和的节点数量」。哈希表以「当前节点的前缀和」为键，以「该前缀和的节点数量」为值。这样就能通过哈希表直接计算出符合要求的路径数量，从而累加到答案上。

整个算法的具体步骤如下：

- 通过先序遍历方式递归遍历二叉树，计算每一个节点的前缀和 `cur_sum`。
- 从哈希表中取出 `cur_sum - sum` 的路径数量（也就是表示存在从前缀和为 `cur_sum - sum` 所对应的节点到前缀和为 `cur_sum` 所对应的节点的路径个数）累加到答案 `res` 中。
- 然后以「当前节点的前缀和」为键，以「该前缀和的节点数量」为值，存入哈希表中。
- 递归遍历二叉树，并累加答案值。
- 恢复哈希表「当前前缀和的节点数量」，返回答案。

代码

```
class Solution:
    prefixsum_count = dict()

    def dfs(self, root, prefixsum_count, target_sum, cur_sum):
        if not root:
            return 0
        res = 0
        cur_sum += root.val
        res += prefixsum_count.get(cur_sum - target_sum, 0)
        prefixsum_count[cur_sum] = prefixsum_count.get(cur_sum, 0) + 1

        res += self.dfs(root.left, prefixsum_count, target_sum, cur_sum)
        res += self.dfs(root.right, prefixsum_count, target_sum, cur_sum)

        prefixsum_count[cur_sum] -= 1
        return res

    def pathSum(self, root: TreeNode, sum: int) -> int:
        if not root:
            return 0
        prefixsum_count = dict()
        prefixsum_count[0] = 1
        return self.dfs(root, prefixsum_count, sum, 0)
```

0438. 找到字符串中所有字母异位词

- 标签：哈希表、字符串、滑动窗口
- 难度：中等

题目链接

- [0438. 找到字符串中所有字母异位词 - 力扣](#)

题目大意

描述：给定两个字符串 s 和 p 。

要求：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。不考虑答案输出的顺序。

说明：

- **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）。
- $1 \leq s.length, p.length \leq 3 * 10^4$ 。
- s 和 p 仅包含小写字母。

示例：

- **示例 1：**

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

- **示例 2：**

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

解题思路

思路 1：滑动窗口

维护一个固定长度为 $len(p)$ 的滑动窗口。于是问题的难点变为了如何判断 s 的子串和 p 是异位词。可以使用两个字典来分别存储 s 的子串中各个字符个数和 p 中各个字符个数。如果两个字典对应的键值全相等，则说明 s 的子串和 p 是异位词。但是这样每一次比较的操作时间复杂度是 $O(n)$ ，我们可以通过在滑动数组中逐字符比较的方式来减少两个字典之间相互比较的复杂度，并用 $valid$ 记录经过验证的字符个数。整个算法步骤如下：

- 使用哈希表 $need$ 记录 p 中各个字符出现次数。使用字典 $window$ 记录 s 的子串中各个字符出现的次数。使用数组 res 记录答案。使用 $valid$ 记录 s 的子串中经过验证的字符个数。使用 $window_size$ 表示窗口大小，值为 $len(p)$ 。使用两个指针 $left$ 、 $right$ 。分别指向滑动窗口的左右边界。
- 一开始， $left$ 、 $right$ 都指向 0。
- 如果 $s[right]$ 出现在 $need$ 中，将最右侧字符 $s[right]$ 加入当前窗口 $window$ 中，记录该字符个数。并验证该字符是否和 $need$ 中对应字符个数相等。如果相等则验证的字符个数加 1，即 $valid += 1$ 。
- 如果该窗口字符长度大于等于 $window_size$ 个，即 $right - left + 1 \geq window_size$ 。则不断右移 $left$ ，缩小滑动窗口长度。
 - 如果验证字符个数 $valid$ 等于窗口长度 $window_size$ ，则 $s[left, right + 1]$ 为 p 的异位词，所以将 $left$ 加入到答案数组中。
 - 如果 $s[left]$ 在 $need$ 中，则更新窗口中对应字符的个数，同时维护 $valid$ 值。
- 右移 $right$ ，直到 $right \geq len(nums)$ 结束。
- 输出答案数组 res 。

思路 1：代码

```

class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        need = collections.defaultdict(int)
        for ch in p:
            need[ch] += 1

        window = collections.defaultdict(int)
        window_size = len(p)
        res = []
        left, right = 0, 0
        valid = 0
        while right < len(s):
            if s[right] in need:
                window[s[right]] += 1
                if window[s[right]] == need[s[right]]:
                    valid += 1

            if right - left + 1 >= window_size:
                if valid == len(need):
                    res.append(left)
                if s[left] in need:
                    if window[s[left]] == need[s[left]]:
                        valid -= 1
                    window[s[left]] -= 1
                left += 1
            right += 1
        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(n + m + |\Sigma|)$, 其中 n 、 m 分别为字符串 s 、 p 的长度, Σ 为字符集, 本题中 $|\Sigma| = 26$ 。
- 空间复杂度: $|\Sigma|$ 。

0443. 压缩字符串

- 标签: 双指针、字符串
- 难度: 中等

题目链接

- [0443. 压缩字符串 - 力扣](#)

题目大意

描述: 给定一个字符数组 $chars$ 。请使用下述算法压缩：

从一个空字符串 s 开始。对于 $chars$ 中的每组连续重复字符：

- 如果这一组长度为 1，则将字符追加到 s 中。
- 如果这一组长度超过 1，则需要向 s 追加字符，后跟这一组的长度。

压缩后得到的字符串 s 不应该直接返回，需要转储到字符数组 $chars$ 中。需要注意的是，如果组长度为 10 或 10 以上，则在 $chars$ 数组中会被拆分为多个字符。

要求: 在修改完输入数组后，返回该数组的新长度。

说明:

- $1 \leq \text{chars.length} \leq 2000$ 。
- $\text{chars}[i]$ 可以是小写英文字母、大写英文字母、数字或符号。
- 必须设计并实现一个只使用常量额外空间的算法来解决此问题。

示例：

- 示例 1：

```
输入: chars = ["a","a","b","b","c","c"]
输出: 返回 6 , 输入数组的前 6 个字符应该是: ["a","2","b","2","c","3"]
解释: "aa" 被 "a2" 替代。"bb" 被 "b2" 替代。"ccc" 被 "c3" 替代。
```

- 示例 2：

```
输入: chars = ["a"]
输出: 返回 1 , 输入数组的前 1 个字符应该是: ["a"]
解释: 唯一的组是 "a"，它保持未压缩，因为它是一个字符。
```

解题思路

思路 1：快慢指针

题目要求原地修改字符串数组。我们可以使用快慢指针来解决原地修改问题，具体解决方法如下：

- 定义两个快慢指针 $slow$, $fast$ 。其中 $slow$ 指向压缩后的当前字符位置, $fast$ 指向压缩前的当前字符位置。
- 记录下当前待压缩字符的起始位置 $fast_start = start$, 然后过滤掉连续相同的字符。
- 将待压缩字符的起始位置的字符存入压缩后的当前字符位置, 即 $\text{chars}[slow] = \text{chars}[fast_start]$, 并向右移动压缩后的当前字符位置, 即 $slow += 1$ 。
- 判断一下待压缩字符的数目是否大于 1:
 - 如果数量为 1, 则不用记录该数量。
 - 如果数量大于 1 (即 $fast - fast_start > 0$), 则我们需要将对应数量存入压缩后的当前字符位置。这时候还需要判断一下数量是否大于等于 10。
 - 如果数量大于等于 10, 则需要先将数字从个位到高位转为字符, 存入压缩后的当前字符位置 (此时数字为反, 比如原数字是 321, 则此时存入后为 123)。因为数字为反, 所以我们需要将对应位置上的子字符串进行反转。
 - 如果数量小于 10, 则直接将数字存入压缩后的当前字符位置, 无需取反。
- 判断完之后向右移动压缩前的当前字符位置 $fast$, 然后继续压缩字符串, 直到全部压缩完, 则返回压缩后的当前字符位置 $slow$ 即为答案。

思路 1：代码

```

class Solution:

    def compress(self, chars: List[str]) -> int:
        def reverse(left, right):
            while left < right:
                chars[left], chars[right] = chars[right], chars[left]
                left += 1
                right -= 1

        slow, fast = 0, 0
        while fast < len(chars):
            fast_start = fast
            while fast + 1 < len(chars) and chars[fast + 1] == chars[fast]:
                fast += 1

            chars[slow] = chars[fast_start]
            slow += 1

            if fast - fast_start > 0:
                cnt = fast - fast_start + 1
                slow_start = slow
                while cnt != 0:
                    chars[slow] = str(cnt % 10)
                    slow += 1
                    cnt = cnt // 10
                reverse(slow_start, slow - 1)

            fast += 1
        return slow

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为字符串 s 的长度。
- 空间复杂度: $O(1)$ 。

0445. 两数相加 II

- 标签: 栈、链表、数学
- 难度: 中等

题目链接

- [0445. 两数相加 II - 力扣](#)

题目大意

给定两个非空链表的头节点 `l1` 和 `l2` 来代表两个非负整数。数字最高位位于链表开始位置。每个节点只储存一位数字。除了数字 `0` 之外，这两个链表代表的数字都不会以 `0` 开头。

要求：将这两个数相加会返回一个新的链表。

解题思路

链表中最高位位于链表开始位置，最低位位于链表结束位置。这与我们做加法的数位顺序是相反的。为了将链表逆序，从而从低位开始处理数位，我们可以借用两个栈：将链表中所有数字分别压入两个栈中，再依次取出相加。

同时，在相加的时候，还要考虑进位问题。具体步骤如下：

- 将链表 `l1` 中所有节点值压入 `stack1` 栈中，再将链表 `l2` 中所有节点值压入 `stack2` 栈中。
- 使用 `res` 存储新的结果链表，一开始指向 `None`，`carry` 记录进位。
- 如果 `stack1` 或 `stack2` 不为空，或者进位 `carry` 不为 `0`，则：
 - 从 `stack1` 中取出栈顶元素 `num1`，如果 `stack1` 为空，则 `num1 = 0`。
 - 从 `stack2` 中取出栈顶元素 `num2`，如果 `stack2` 为空，则 `num2 = 0`。
 - 计算相加结果，并计算进位。
 - 建立新节点，存储进位后余下的值，并令其指向 `res`。
 - `res` 指向新节点，继续判断。
- 如果 `stack1`、`stack2` 都为空，并且进位 `carry` 为 `0`，则输出 `res`。

代码

```
class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        stack1, stack2 = [], []
        while l1:
            stack1.append(l1.val)
            l1 = l1.next
        while l2:
            stack2.append(l2.val)
            l2 = l2.next

        res = None
        carry = 0
        while stack1 or stack2 or carry != 0:
            num1 = stack1.pop() if stack1 else 0
            num2 = stack2.pop() if stack2 else 0
            cur_sum = num1 + num2 + carry
            carry = cur_sum // 10
            cur_sum %= 10
            cur_node = ListNode(cur_sum)
            cur_node.next = res
            res = cur_node
        return res
```

0447. 回旋镖的数量

- 标签：数组、哈希表、数学
- 难度：中等

题目链接

- [0447. 回旋镖的数量 - 力扣](#)

题目大意

给定平面上点坐标的数组 `points`，其中 $points[i] = [x_i, y_i]$ 。判断 `points` 中是否存在三个点 i, j, k ，满足 i 和 j 之间的距离等于 i 和 k 之间的距离，即 $dist[i, j] = dist[i, k]$ 。找出满足上述关系的答案数量。

解题思路

使用哈希表记录每两个点之间的距离。然后使用两重循环遍历坐标数组，对于每两个点 i 、点 j ，计算两个点之间的距离，并将距离存进哈希表中。再从哈希表中选取距离相同的关系中依次选出两个，作为三个点之间的距离关系 $dist[i, j] = dist[i, k]$ ，因为还需考虑顺序，所以共有 $value * (value - 1)$ 种情况。累加到答案中。

代码

```

class Solution:
    def numberOfBoomerangs(self, points: List[List[int]]) -> int:
        ans = 0
        for point_i in points:
            dis_dict = dict()
            for point_j in points:
                if point_i != point_j:
                    dx = point_i[0] - point_j[0]
                    dy = point_i[1] - point_j[1]
                    dis = dx * dx + dy * dy
                    if dis in dis_dict:
                        dis_dict[dis] += 1
                    else:
                        dis_dict[dis] = 1
            for value in dis_dict.values():
                ans += value*(value-1)
        return ans

```

0450. 删除二叉搜索树中的节点

- 标签: 树、二叉搜索树、二叉树
- 难度: 中等

题目链接

- [0450. 删除二叉搜索树中的节点 - 力扣](#)

题目大意

描述: 给定一个二叉搜索树的根节点 `root`，以及一个值 `key`。

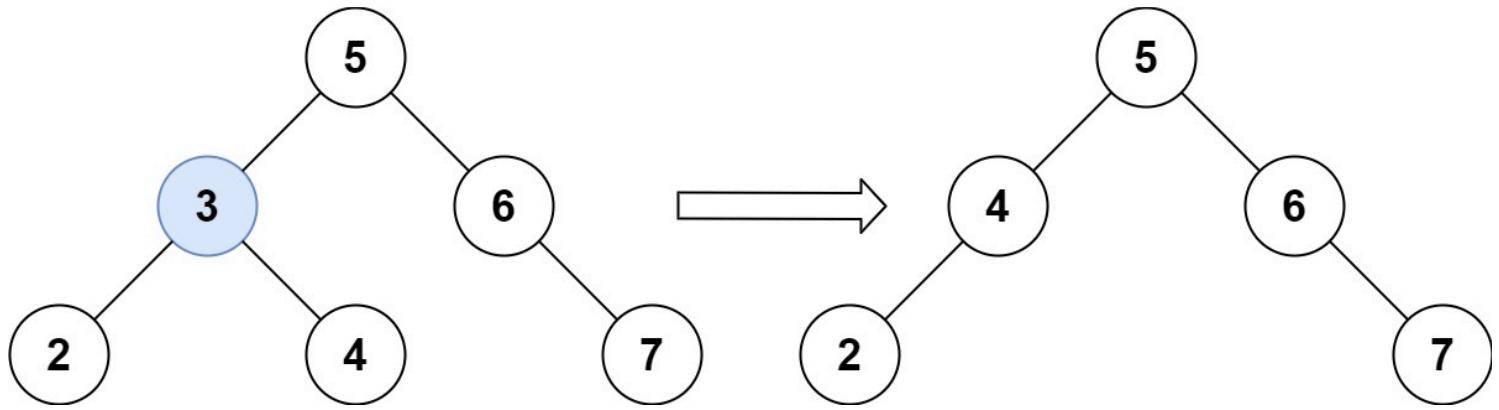
要求: 从二叉搜索树中删除 `key` 对应的节点。并保证删除后的树仍是二叉搜索树。要求算法时间复杂度为 $O(h)$ ， h 为树的高度。最后返回二叉搜索树的根节点。

说明:

- 节点数的范围 $[0, 10^4]$ 。
- $-10^5 \leq Node.val \leq 10^5$ 。
- 节点值唯一。
- `root` 是合法的二叉搜索树。
- $-10^5 \leq key \leq 10^5$ 。

示例:

- **示例 1:**



输入: `root = [5,3,6,2,4,null,7], key = 3`

输出: `[5,4,6,2,null,null,7]`

解释: 给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 `[5,4,6,2,null,null,7]`，如上图所示。

另一个正确答案是 `[5,2,6,null,4,null,7]`。

- 示例 2:

输入: `root = [5,3,6,2,4,null,7], key = 0`

输出: `[5,3,6,2,4,null,7]`

解释: 二叉树不包含值为 0 的节点

解题思路

思路 1：递归

删除分两个步骤：查找和删除。查找通过递归查找，删除的话需要考虑情况。

1. 从根节点 `root` 开始，递归遍历搜索二叉树。
2. 如果当前节点节点为空，返回当前节点。
3. 如果当前节点值大于 `key`，则去左子树中搜索并删除，此时 `root.left` 也要跟着递归更新，递归完成后返回当前节点。
4. 如果当前节点值小于 `key`，则去右子树中搜索并删除，此时 `root.right` 也要跟着递归更新，递归完成后返回当前节点。
5. 如果当前节点值等于 `key`，则该节点就是待删除节点。
 - i. 如果当前节点的左子树为空，则删除该节点之后，则右子树代替当前节点位置，返回右子树。
 - ii. 如果当前节点的右子树为空，则删除该节点之后，则左子树代替当前节点位置，返回左子树。
 - iii. 如果当前节点的左右子树都有，则将左子树转移到右子树最左侧的叶子节点位置上，然后右子树代替当前节点位置。返回右子树。

思路 1：代码

```

class Solution:
    def deleteNode(self, root: TreeNode, key: int) -> TreeNode:
        if not root:
            return root

        if root.val > key:
            root.left = self.deleteNode(root.left, key)
            return root
        elif root.val < key:
            root.right = self.deleteNode(root.right, key)
            return root
        else:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            else:
                curr = root.right
                while curr.left:
                    curr = curr.left
                curr.left = root.left
                return root.right

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 是二叉搜索树的节点数。
- 空间复杂度: $O(n)$ 。

0451. 根据字符出现频率排序

- 标签: 哈希表、字符串、桶排序、计数、排序、堆（优先队列）
- 难度: 中等

题目链接

- [0451. 根据字符出现频率排序 - 力扣](#)

题目大意

描述: 给定一个字符串 s 。

要求: 将字符串 s 里的字符按照出现的频率降序排列。如果有多个答案, 返回其中任何一个。

说明:

- $1 \leq s.length \leq 5 * 10^5$ 。
- s 由大小写英文字母和数字组成。

示例:

- 示例 1:

```

输入: s = "tree"
输出: "eert"
解释: 'e' 出现两次, 'r' 和 't' 都只出现一次。
因此 'e' 必须出现在 'r' 和 't' 之前。此外, "eetr" 也是一个有效的答案。

```

- 示例 2:

```
输入: s = "cccaaa"
输出: "cccaaa"
解释: 'c'和'a'都出现三次。此外, "aaaccc"也是有效的答案。
注意"cacaca"是不正确的, 因为相同的字母必须放在一起。
```

解题思路

思路 1：优先队列

- 使用哈希表 `s_dict` 统计字符频率。
- 然后遍历哈希表 `s_dict`，将字符以及字符频数存入优先队列中。
- 将优先队列中频数最高的元素依次加入答案数组中。
- 最后拼接答案数组为字符串，将其返回。

思路 1：代码

```
import heapq

class Solution:
    def frequencySort(self, s: str) -> str:
        # 统计元素频数
        s_dict = dict()
        for ch in s:
            if ch in s_dict:
                s_dict[ch] += 1
            else:
                s_dict[ch] = 1

        priority_queue = []
        for ch in s_dict:
            heapq.heappush(priority_queue, (-s_dict[ch], ch))

        res = []
        while priority_queue:
            ch = heapq.heappop(priority_queue)[-1]
            times = s_dict[ch]
            while times:
                res.append(ch)
                times -= 1
        return ''.join(res)
```

思路 1：复杂度分析

- 时间复杂度: $O(n + k \times \log_2 k)$ 。其中 n 为字符串 s 的长度, k 是字符串中不同字符的个数。
- 空间复杂度: $O(n + k)$ 。

0452. 用最少数量的箭引爆气球

- 标签: 贪心、数组、排序
- 难度: 中等

题目链接

- [0452. 用最少数量的箭引爆气球 - 力扣](#)

题目大意

描述：在一个坐标系中有许多球形的气球。对于每个气球，给定气球在 x 轴上的开始坐标和结束坐标 (x_{start}, x_{end}) 。

同时，在 x 轴的任意位置都能垂直发出弓箭，假设弓箭发出的坐标就是 x。那么如果有气球满足 $x_{start} \leq x \leq x_{end}$ ，则该气球就会被引爆，且弓箭可以无限前进，可以将满足上述要求的气球全部引爆。

现在给定一个数组 `points`，其中 $points[i] = [x_{start}, x_{end}]$ 代表每个气球的开始坐标和结束坐标。

要求：返回能引爆所有气球的最小弓箭数。

说明：

- $1 \leq points.length \leq 10^5$ 。
- $points[i].length == 2$ 。
- $-2^{31} \leq x_{start} < x_{end} \leq 2^{31} - 1$ 。

示例：

- **示例 1：**

```
输入: points = [[10, 16], [2, 8], [1, 6], [7, 12]]
```

```
输出: 2
```

解释：气球可以用 2 支箭来爆破：

- 在 $x = 6$ 处射出箭，击破气球 $[2, 8]$ 和 $[1, 6]$ 。
- 在 $x = 11$ 处发射箭，击破气球 $[10, 16]$ 和 $[7, 12]$ 。

- **示例 2：**

```
输入: points = [[1, 2], [3, 4], [5, 6], [7, 8]]
```

```
输出: 4
```

解释：每个气球需要射出一支箭，总共需要 4 支箭。

解题思路

思路 1：贪心算法

弓箭的起始位置和结束位置可以看做是一段区间，直观上来看，为了使用最少的弓箭数，可以尽量射中区间重叠最多的地方。

所以问题变为了：**如何寻找区间重叠最多的地方，也就是区间交集最多的地方。**

我们将 `points` 按结束坐标升序排序（为什么按照结束坐标排序后边说）。

然后维护两个变量：一个是当前弓箭的坐标 `arrow_pos`、另一个是弓箭的数目 `count`。

为了尽可能的穿过更多的区间，所以每一支弓箭都应该尽可能的从区间的结束位置穿过，这样才能覆盖更多的区间。

初始情况下，第一支弓箭的坐标为第一个区间的结束位置，然后弓箭数为 1。然后依次遍历每段区间。

如果遇到弓箭坐标小于区间起始位置的情况，说明该弓箭不能引爆该区间对应的气球，需要用新的弓箭来射，所以弓箭数加 1，弓箭坐标也需要更新为新区间

的结束位置。

再来看为什么将 `points` 按结束坐标升序排序而不是按照开始坐标升序排序？

其实也可以，但是按开始坐标排序不如按结束坐标排序简单。

按开始坐标升序排序需要考虑一种情况：有交集关系的区间中，有的区间结束位置比较早。比如 $[0, 6]、[1, 2] [4, 5]$ ，按照开始坐标升序排序的话，就像下图一样：

```
[0.....6]
[1..2]
[4..5]
```

第一箭的位置需要进行迭代判断，取区间 `[0, 6]`、`[1, 2]` 中结束位置最小的位置，即 `arrow_pos = min(points[i][1], arrow_pos)`，然后再判断接下来的区间是否能够引爆。

而按照结束坐标排序的话，箭的位置一开始就确定了，不需要再改变和判断箭的位置，直接判断区间即可。

思路 1：代码

1. 按照结束位置升序排序

```
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if not points:
            return 0
        points.sort(key=lambda x: x[1])
        arrow_pos = points[0][1]
        count = 1
        for i in range(1, len(points)):
            if arrow_pos < points[i][0]:
                count += 1
                arrow_pos = points[i][1]
        return count
```

2. 按照开始位置升序排序

```
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if not points:
            return 0
        points.sort(key=lambda x: x[0])
        arrow_pos = points[0][1]
        count = 1
        for i in range(1, len(points)):
            if arrow_pos < points[i][0]:
                count += 1
                arrow_pos = points[i][1]
            else:
                arrow_pos = min(points[i][1], arrow_pos)
        return count
```

思路 1：复杂度分析

- 时间复杂度: $O(n \times \log n)$ ，其中 n 是数组 `points` 的长度。
- 空间复杂度: $O(\log n)$ 。

0454. 四数相加 II

- 标签: 数组、哈希表
- 难度: 中等

题目链接

- [0454. 四数相加 II - 力扣](#)

题目大意

描述：给定四个整数数组 nums1 、 nums2 、 nums3 、 nums4 。

要求：计算有多少不同的 (i, j, k, l) 满足以下条件。

1. $0 \leq i, j, k, l < n$ 。
2. $\text{nums1}[i] + \text{nums2}[j] + \text{nums3}[k] + \text{nums4}[l] == 0$ 。

说明：

- $n == \text{nums1.length}$ 。
- $n == \text{nums2.length}$ 。
- $n == \text{nums3.length}$ 。
- $n == \text{nums4.length}$ 。
- $1 \leq n \leq 200$ 。
- $-2^{28} \leq \text{nums1}[i], \text{nums2}[i], \text{nums3}[i], \text{nums4}[i] \leq 2^{28}$ 。

示例：

- **示例 1：**

```
输入: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
```

```
输出: 2
```

解释：

两个元组如下：

1. $(0, 0, 0, 1) \rightarrow \text{nums1}[0] + \text{nums2}[0] + \text{nums3}[0] + \text{nums4}[1] = 1 + (-2) + (-1) + 2 = 0$
2. $(1, 1, 0, 0) \rightarrow \text{nums1}[1] + \text{nums2}[1] + \text{nums3}[0] + \text{nums4}[0] = 2 + (-1) + (-1) + 0 = 0$

- **示例 2：**

```
输入: nums1 = [0], nums2 = [0], nums3 = [0], nums4 = [0]
```

```
输出: 1
```

解题思路

思路 1：哈希表

直接暴力搜索的时间复杂度是 $O(n^4)$ 。我们可以降低一下复杂度。

将四个数组分为两组。 nums1 和 nums2 分为一组， nums3 和 nums4 分为一组。

已知 $\text{nums1}[i] + \text{nums2}[j] + \text{nums3}[k] + \text{nums4}[l] == 0$ ，可以得到 $\text{nums1}[i] + \text{nums2}[j] = -(\text{nums3}[k] + \text{nums4}[l])$

建立一个哈希表。两重循环遍历数组 nums1 、 nums2 ，先将 $\text{nums}[i] + \text{nums}[j]$ 的和个数记录到哈希表中，然后再用两重循环遍历数组 nums3 、 nums4 。如果 $-(\text{nums3}[k] + \text{nums4}[l])$ 的结果出现在哈希表中，则将结果数累加到答案中。最终输出累加之后的答案。

思路 1：代码

```

class Solution:
    def fourSumCount(self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) -> int:
        nums_dict = dict()
        for num1 in nums1:
            for num2 in nums2:
                sum = num1 + num2
                if sum in nums_dict:
                    nums_dict[sum] += 1
                else:
                    nums_dict[sum] = 1
        count = 0
        for num3 in nums3:
            for num4 in nums4:
                sum = num3 + num4
                if -sum in nums_dict:
                    count += nums_dict[-sum]

        return count

```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ ，其中 n 为数组的元素个数。
- 空间复杂度： $O(n^2)$ 。

0455. 分发饼干

- 标签：贪心、数组、双指针、排序
- 难度：简单

题目链接

- [0455. 分发饼干 - 力扣](#)

题目大意

描述：一位很棒的家长为孩子们分发饼干。对于每个孩子 i ，都有一个胃口值 $g[i]$ ，即每个小孩希望得到饼干的最小尺寸值。对于每块饼干 j ，都有一个尺寸值 $s[j]$ 。只有当 $s[j] > g[i]$ 时，我们才能将饼干 j 分配给孩子 i 。每个孩子最多只能给一块饼干。

现在给定代表所有孩子胃口值的数组 g 和代表所有饼干尺寸的数组 j 。

要求：尽可能满足越多数量的孩子，并求出这个最大数值。

说明：

- $1 \leq g.length \leq 3 * 10^4$ 。
- $0 \leq s.length \leq 3 * 10^4$ 。
- $1 \leq g[i], s[j] \leq 2^{31} - 1$ 。

示例：

- 示例 1：

输入： $g = [1, 2, 3]$, $s = [1, 1]$

输出：1

解释：你有三个孩子和两块小饼干，3 个孩子的胃口值分别是：1, 2, 3。虽然你有两块小饼干，由于他们的尺寸都是 1，你只能让胃口值是 1 的孩子满足。所以应该输出 1。

- 示例 2：

输入: `g = [1, 2], s = [1, 2, 3]`

输出: `2`

解释: 你有两个孩子和三块饼干, `2`个孩子的胃口值分别是`1`, `2`。你拥有的饼干数量和尺寸都足以让所有孩子满足。所以你应该输出 `2`。

解题思路

思路 1：贪心算法

为了尽可能的满足更多的小孩, 而且一块饼干不能掰成两半, 所以我们应该尽量让胃口小的孩子吃小块饼干, 这样胃口大的孩子才有大块饼干吃。

所以, 从贪心算法的角度来考虑, 我们应该按照孩子的胃口从小到大对数组 `g` 进行排序, 然后按照饼干的尺寸大小从小到大对数组 `s` 进行排序, 并且对于每个孩子, 应该选择满足这个孩子的胃口且尺寸最小的饼干。

下面我们使用贪心算法三步走的方法解决这道题。

1. **转换问题:** 将原问题转变为, 当胃口最小的孩子选择完满足这个孩子的胃口且尺寸最小的饼干之后, 再解决剩下孩子的选择问题 (子问题)。
2. **贪心选择性质:** 对于当前孩子, 用尺寸尽可能小的饼干满足这个孩子的胃口。
3. **最优子结构性质:** 在上面的贪心策略下, 当前孩子的贪心选择 + 剩下孩子的子问题最优解, 就是全局最优解。也就是说在贪心选择的方案下, 能够使得满足胃口的孩子数量达到最大。

使用贪心算法的代码解决步骤描述如下:

1. 对数组 `g`、`s` 进行从小到大排序, 使用变量 `index_g` 和 `index_s` 分别指向 `g`、`s` 初始位置, 使用变量 `res` 保存结果, 初始化为 `0`。
2. 对比每个元素 `g[index_g]` 和 `s[index_s]`:
 - i. 如果 `g[index_g] <= s[index_s]`, 说明当前饼干满足当前孩子胃口, 则答案数量加 `1`, 并且向右移动 `index_g` 和 `index_s`。
 - ii. 如果 `g[index_g] > s[index_s]`, 说明当前饼干无法满足当前孩子胃口, 则向右移动 `index_s`, 判断下一块饼干是否可以满足当前孩子胃口。
3. 遍历完输出答案 `res`。

思路 1：代码

```
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        g.sort()
        s.sort()
        index_g, index_s = 0, 0
        res = 0
        while index_g < len(g) and index_s < len(s):
            if g[index_g] <= s[index_s]:
                res += 1
                index_g += 1
                index_s += 1
            else:
                index_s += 1
        return res
```

思路 1：复杂度分析

- **时间复杂度:** $O(m \times \log m + n \times \log n)$, 其中 m 和 n 分别是数组 g 和 s 的长度。
- **空间复杂度:** $O(\log m + \log n)$ 。

0459. 重复的子字符串

- 标签: 字符串、字符串匹配
- 难度: 简单

题目链接

- 0459. 重复的子字符串 - 力扣

题目大意

描述：给定一个非空的字符串 `s`。

要求：检查该字符串 `s` 是否可以通过由它的一个子串重复多次构成。

说明：

- $1 \leq s.length \leq 10^4$ 。
- `s` 由小写英文字母组成

示例：

- 示例 1：

```
输入: s = "abab"
输出: true
解释: 可由子串 "ab" 重复两次构成。
```

- 示例 2：

```
输入: s = "aba"
输出: false
```

解题思路

思路 1：KMP 算法

这道题我们可以使用 KMP 算法的 `next` 数组来解决。我们知道 `next[j]` 表示的含义是：记录下标 `j` 之前（包括 `j`）的模式串 `p` 中，最长相等前后缀的长度。

而如果整个模式串 `p` 的最长相等前后缀长度不为 `0`，即 `next[len(p) - 1] != 0`，则说明整个模式串 `p` 中有最长相同的前后缀，假设 `next[len(p) - 1] == k`，则说明 `p[0: k] == p[m - k: m]`。比如字符串 "abcabcabc"，最长相同前后缀为 "abcabc" = "abcabc"。

- 如果最长相等的前后缀是重叠的，比如之前的例子 "abcabcabc"。
 - 如果我们去除字符串中相同的前后缀的重叠部分，剩下两头前后缀部分（这两部分是相同的）。然后再去除剩余的后缀部分，只保留剩余的前缀部分。比如字符串 "abcabcabc" 去除重叠部分和剩余的后缀部分之后就是 "abc"。实际上这个部分就是字符串去除整个后缀部分的剩余部分。
 - 如果整个字符串可以通过子串重复构成的话，那么这部分就是最小周期的子串。
 - 我们只需要判断整个子串的长度是否是剩余部分长度的整数倍即可。也就是判断 `len(p) % (len(p) - next[size - 1]) == 0` 是否成立，如果成立，则字符串 `s` 可由 `s[0: len(p) - next[size - 1]]` 构成的子串重复构成，返回 `True`。否则返回 `False`。
- 如果最长相等的前后缀是不重叠的，那我们可将重叠部分视为长度为 `0` 的空串，则剩余的部分其实就是去除后缀部分的剩余部分，上述结论依旧成立。

思路 1：代码

```

class Solution:
    def generateNext(self, p: str):
        m = len(p)
        next = [0 for _ in range(m)]

        left = 0
        for right in range(1, m):
            while left > 0 and p[left] != p[right]:
                left = next[left - 1]
            if p[left] == p[right]:
                left += 1
            next[right] = left

        return next

    def repeatedSubstringPattern(self, s: str) -> bool:
        size = len(s)
        if size == 0:
            return False
        next = self.generateNext(s)
        if next[size - 1] != 0 and size % (size - next[size - 1]) == 0:
            return True
        return False

```

思路 1：复杂度分析

- 时间复杂度: $O(m)$, 其中模式串 p 的长度为 m 。
- 空间复杂度: $O(m)$ 。

0461. 汉明距离

- 标签: 位运算
- 难度: 简单

题目链接

- [0461. 汉明距离 - 力扣](#)

题目大意

给定两个整数 x 和 y , 计算他们之间的汉明距离。

- 汉明距离: 两个数字对应二进制位上不同的位置的数目

解题思路

先对两个数进行异或运算 (相同位置上, 值相同, 结果为 0, 值不同, 结果为 1), 用于记录 x 和 y 不同位置上的异同情况。

然后再按位统计异或结果中 1 的位数。

这里统计 1 的位数可以逐位移动, 检查每一位是否为 1。

也可以借助 $n \& (n - 1)$ 运算。这个运算刚好可以将 n 的二进制中最低位的 1 变为 0。

代码

1. 逐位移动

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        xor = x ^ y
        distance = 0
        while xor:
            if xor & 1:
                distance += 1
            xor >>= 1
        return distance
```

2. $n \& (n - 1)$ 运算

```
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        xor = x ^ y
        distance = 0
        while xor:
            distance += 1
            xor = xor & (xor - 1)
        return distance
```

0463. 岛屿的周长

- 标签: 深度优先搜索、广度优先搜索、数组、矩阵
- 难度: 简单

题目链接

- [0463. 岛屿的周长 - 力扣](#)

题目大意

描述: 给定一个 $\text{row} * \text{col}$ 大小的二维网格地图 grid ，其中： $\text{grid}[i][j] = 1$ 表示陆地， $\text{grid}[i][j] = 0$ 表示水域。

网格中的格子水平和垂直方向相连（对角线方向不相连）。整个网格被水完全包围，但其中恰好有一个岛屿（多个表示陆地的格子相连组成）。

岛屿内部中没有「湖」（指水域在岛屿内部且不和岛屿周围的水相连）。格子是边长为 1 的正方形。网格为长方形，且宽度和高度均不超过 100。

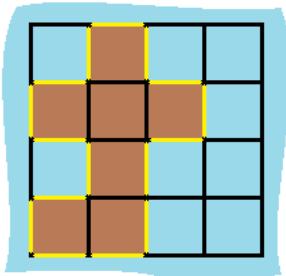
要求: 计算这个岛屿的周长。

说明:

- $\text{row} == \text{grid.length}$ 。
- $\text{col} == \text{grid[i].length}$ 。
- $1 \leq \text{row}, \text{col} \leq 100$ 。
- $\text{grid}[i][j]$ 为 0 或 1。

示例:

- 示例 1:



输入: `grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]`

输出: `16`

解释: 它的周长是上面图片中的 `16` 个黄色的边

- 示例 2:

输入: `grid = [[1]]`

输出: `4`

解题思路

思路 1：广度优先搜索

- 使用整形变量 `count` 存储周长，使用队列 `queue` 用于进行广度优先搜索。
- 遍历一遍二维数组 `grid`，对 `grid[row][col] == 1` 的区域进行广度优先搜索。
- 先将起始点 `(row, col)` 加入队列。
- 如果队列不为空，则取出队头坐标 `(row, col)`。先将 `(row, col)` 标记为 `2`，避免重复统计。
- 然后遍历上、下、左、右四个方向的相邻区域，如果遇到边界或者水域，则周长加 1。
- 如果相邻区域 `grid[new_row][new_col] == 1`，则将其赋值为 `2`，并将坐标加入队列。
- 继续执行 4 ~ 6 步，直到队列为空时返回 `count`。

思路 1：代码

```

class Solution:
    def bfs(self, grid, rows, cols, row, col):
        directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        queue = collections.deque([(row, col)])

        count = 0
        while queue:
            row, col = queue.popleft()
            # 避免重复统计
            grid[row][col] = 2
            for direct in directs:
                new_row = row + direct[0]
                new_col = col + direct[1]
                # 遇到边界或者水域，则周长加 1
                if new_row < 0 or new_row >= rows or new_col < 0 or new_col >= cols or grid[new_row][new_col] == 0:
                    count += 1
                # 相邻区域为陆地，则将其标记为 2，加入队列
                elif grid[new_row][new_col] == 1:
                    grid[new_row][new_col] = 2
                    queue.append((new_row, new_col))
            # 相邻区域为 2 的情况不做处理
        return count

    def islandPerimeter(self, grid: List[List[int]]) -> int:
        rows, cols = len(grid), len(grid[0])
        for row in range(rows):
            for col in range(cols):
                if grid[row][col] == 1:
                    return self.bfs(grid, rows, cols, row, col)

```

思路 1：复杂度分析

- 时间复杂度： $O(n \times m)$ ，其中 m 和 n 分别为行数和列数。
- 空间复杂度： $O(n \times m)$ 。

参考资料

- 【题解】Golang BFS 实现，性能比dfs要高 - 岛屿的周长 - 力扣

0464. 我能赢吗

- 标签：位运算、记忆化搜索、数学、动态规划、状态压缩、博弈
- 难度：中等

题目链接

- [0464. 我能赢吗 - 力扣](#)

题目大意

描述：给定两个整数， $maxChoosableInteger$ 表示可以选择的最大整数， $desiredTotal$ 表示累计和。现在开始玩一个游戏，两个玩家轮流从 $1 \sim maxChoosableInteger$ 中不重复的抽取一个整数，直到累积整数和大于等于 $desiredTotal$ 时，这个人就赢得比赛。假设两位玩家玩游戏时都表现最佳。

要求：判断先出手的玩家是否能够稳赢，如果能稳赢，则返回 `True`，否则返回 `False`。

说明：

- $1 \leq \text{maxChoosableInteger} \leq 20$ 。
- $0 \leq \text{desiredTotal} \leq 300$ 。

示例：

- 示例 1：

```
输入: maxChoosableInteger = 10, desiredTotal = 11
```

```
输出: False
```

解释：

无论第一个玩家选择哪个整数，他都会失败。

第一个玩家可以选择从 1 到 10 的整数。

如果第一个玩家选择 1，那么第二个玩家只能选择从 2 到 10 的整数。

第二个玩家可以通过选择整数 10（那么累积和为 $11 \geq \text{desiredTotal}$ ），从而取得胜利。

同样地，第一个玩家选择任意其他整数，第二个玩家都会赢。

- 示例 2：

```
输入: maxChoosableInteger = 10, desiredTotal = 0
```

```
输出: True
```

解题思路

思路 1：状态压缩 + 记忆化搜索

$\text{maxChoosableInteger}$ 的区间范围是 $[1, 20]$ ，数据量不是很大，我们可以使用状态压缩来判断当前轮次中数字的选取情况。

题目假设两位玩家玩游戏时都表现最佳，则每个人都会尽力去赢，在每轮次中，每个人都会分析此次选择后，对后续轮次的影响，判断自己是必赢还是必输。

1. 如果当前轮次选择某个数之后，自己一定会赢时，才会选择这个数。
2. 如果当前轮次无论选择哪个数，自己一定会输时，那无论选择哪个数其实都已经无所谓了。

这样我们可以定义一个递归函数 `dfs(state, curTotal)`，用于判断处于状态 $state$ ，并且当前累计和为 $curTotal$ 时，自己是否一定会赢。如果自己一定会赢，返回 `True`，否则返回 `False`。递归函数内容如下：

1. 从 $1 \sim \text{maxChoosableInteger}$ 中选择一个之前没有选过的数 k 。
2. 如果选择的数 k 加上当前的整数和 $curTotal$ 之后大于等于 desiredTotal ，则自己一定会赢。
3. 如果选择的数 k 之后，对方必输（即递归调用 `dfs(state | (1 << (k - 1)), curTotal + k)` 为 `False` 时），则自己一定会赢。
4. 如果无论选择哪个数，自己都赢不了，则自己必输，返回 `False`。

这样，我们从 $state = 0, curTotal = 0$ 开始调用递归方法 `dfs(state, curTotal)`，即可判断先出手的玩家是否能够稳赢。

接下来，我们还需要考虑一些边界条件。

1. 当 $\text{maxChoosableInteger}$ 直接大于等于 desiredTotal ，则先手玩家无论选什么，直接就赢了，这种情况下，我们直接返回 `True`。
2. 当 $1 \sim \text{maxChoosableInteger}$ 中所有数加起来都小于 desiredTotal ，则先手玩家无论怎么选，都无法稳赢，题目要求我们判断先出手的玩家是否能够稳赢，既然先手无法稳赢，我们直接返回 `False`。

思路 1：代码

```

class Solution:
    def canIWin(self, maxChoosableInteger: int, desiredTotal: int) -> bool:
        @cache
        def dfs(state, curTotal):
            for k in range(1, maxChoosableInteger + 1):
                if state >> (k - 1) & 1 != 0:
                    continue
                if curTotal + k >= desiredTotal:
                    return True
                if not dfs(state | (1 << (k - 1)), curTotal + k):
                    return True
            return False

        # maxChoosableInteger 直接大于等于 desiredTotal，则先手玩家一定赢
        if maxChoosableInteger >= desiredTotal:
            return True

        # 1 ~ maxChoosableInteger 所有数加起来都不够 desiredTotal，则先手玩家一定输
        if (1 + maxChoosableInteger) * maxChoosableInteger // 2 < desiredTotal:
            return False
        return dfs(0, 0)

```

思路 1：复杂度分析

- 时间复杂度: $O(n \times 2^n)$, 其中 n 为 $maxChoosableInteger$ 。
- 空间复杂度: $O(2^n)$ 。

0467. 环绕字符串中唯一的子字符串

- 标签：字符串、动态规划
- 难度：中等

题目链接

- [0467. 环绕字符串中唯一的子字符串 - 力扣](#)

题目大意

把字符串 s 看作是 $abcdefghijklmnopqrstuvwxyz$ 的无限环绕字符串，所以 s 看起来是这样的： $\dots zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz\dots$ 。

给定一个字符串 p 。

要求：你需要的是找出 s 中有多少个唯一的 p 的非空子串，尤其是当你的输入是字符串 p ，你需要输出字符串 s 中 p 的不同的非空子串的数目。

注意： p 仅由小写的英文字母组成， p 的大小可能超过 10000 。

解题思路

字符串 s 是个 $a \sim z$ 无限循环的字符串，题目要求计算字符串 s 和字符串 p 中有多少个相等的非空子串。发现以该字符结尾的连续子串的长度，就等于以该字符结尾的相等子串的个数。所以我们可以按以下步骤求解：

- 记录以每个字符结尾的字符串最长长度。
- 将其累加起来就是最终答案。

代码

```

class Solution:
    def findSubstringInWraproundString(self, p: str) -> int:
        dp = collections.defaultdict(int)
        dp[p[0]] = 1
        max_len = 1
        for i in range(1, len(p)):
            if (ord(p[i]) - ord(p[i - 1])) % 26 == 1:
                max_len += 1
            else:
                max_len = 1
            dp[p[i]] = max(dp[p[i]], max_len)

        ans = 0
        for key, value in dp.items():
            ans += value
        return ans

```

0468. 验证IP地址

- 标签: 字符串
- 难度: 中等

题目链接

- [0468. 验证IP地址 - 力扣](#)

题目大意

描述: 给定一个字符串 `queryIP`。

要求: 如果是有效的 IPv4 地址，返回 `"IPv4"`；如果是有效的 IPv6 地址，返回 `"IPv6"`；如果不是上述类型的 IP 地址，返回 `"Neither"`。

说明:

- **有效的 IPv4 地址:** 格式为 `"x1.x2.x3.x4"` 形式的 IP 地址。其中：
 - $0 \leq xi \leq 255$ 。
 - xi 不能包含前导零。
- 例如: `"192.168.1.1"`、`"192.168.1.0"` 为有效 IPv4 地址，`"192.168.01.1"` 为无效 IPv4 地址，`"192.168.1.00"`、`"192.168@1.1"` 为无效 IPv4 地址。
- **有效的 IPv6 地址:** 格式为 `"x1:x2:x3:x4:x5:x6:x7:x8"` 的 IP 地址，其中：
 - $1 \leq xi.length \leq 4$ 。
 - xi 是一个十六进制字符串，可以包含数字、小写英文字母（'a' 到 'f'）和大写英文字母（'A' 到 'F'）。
 - 在 xi 中允许前导零。
- 例如: `"2001:0db8:85a3:0000:0000:8a2e:0370:7334"` 和 `"2001:db8:85a3:0:0:8A2E:0370:7334"` 是有效的 IPv6 地址，而 `"2001:0db8:85a3::8A2E:037j:7334"` 和 `"02001:0db8:85a3:0000:0000:8a2e:0370:7334"` 是无效的 IPv6 地址。
- `queryIP` 仅由英文字母，数字，字符 `'.'` 和 `::` 组成。

示例:

- **示例 1:**

```

输入: queryIP = "172.16.254.1"
输出: "IPv4"
解释: 有效的 IPv4 地址, 返回 "IPv4"

```

- 示例 2:

```
输入: queryIP = "2001:0db8:85a3:0:0:8A2E:0370:7334"
输出: "IPv6"
解释: 有效的 IPv6 地址, 返回 "IPv6"
```

解题思路

思路 1：模拟

根据题意以及有效的 IPv4 地址规则、有效的 IPv6 地址规则，我们可以分两步来做：第一步，验证是否为有效的 IPv4 地址。第二步，验证是否为有效的 IPv6 地址。

1. 验证是否为有效的 IPv4 地址

- 将字符串按照 '.' 进行分割，将不同分段存入数组 `path` 中。
- 如果分段数组 `path` 长度等于 4，则说明该字符串为 IPv4 地址，接下来验证是否为有效的 IPv4 地址。
- 遍历分段数组 `path`，去验证每个分段 `sub`。
 - 如果当前分段 `sub` 为空，或者不是纯数字，则返回 "Neither"。
 - 如果当前分段 `sub` 有前导 0，并且长度不为 1，则返回 "Neither"。
 - 如果当前分段 `sub` 对应的值不在 0 ~ 255 范围内，则返回 "Neither"。
- 遍历完分段数组 `path`，仍未发现问题，则该字符串为有效的 IPv4 地址，返回 `IPv4`。

2. 验证是否为有效的 IPv6 地址

- 将字符串按照 ':' 进行分割，将不同分段存入数组 `path` 中。
- 如果分段数组 `path` 长度等于 8，则说明该字符串为 IPv6 地址，接下来验证是否为有效的 IPv6 地址。
- 定义一个代表十六进制不同字符的字符串 `valid = "0123456789abcdefABCDEF"`，用于验证分段的每一位是否为 16 进制数。
- 遍历分段数组 `path`，去验证每个分段 `sub`。
 - 如果当前分段 `sub` 为空，则返回 "Neither"。
 - 如果当前分段 `sub` 长度超过 4，则返回 "Neither"。
 - 如果当前分段 `sub` 对应的每一位的值不在 `valid` 内，则返回 "Neither"。
- 遍历完分段数组 `path`，仍未发现问题，则该字符串为有效的 IPv6 地址，返回 `IPv6`。

如果通过上面两步验证，该字符串既不是有效的 IPv4 地址，也不是有效的 IPv6 地址，则返回 "Neither"。

思路 1：代码

```

class Solution:
    def validIPAddress(self, queryIP: str) -> str:
        path = queryIP.split('.')
        if len(path) == 4:
            for sub in path:
                if not sub or not sub.isdecimal():
                    return "Neither"
                if sub[0] == '0' and len(sub) != 1:
                    return "Neither"
                if int(sub) > 255:
                    return "Neither"
            return "IPv4"

        path = queryIP.split(':')
        if len(path) == 8:
            valid = "0123456789abcdefABCDEF"
            for sub in path:
                if not sub:
                    return "Neither"
                if len(sub) > 4:
                    return "Neither"
                for digit in sub:
                    if digit not in valid:
                        return "Neither"
            return "IPv6"

        return "Neither"

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为字符串 `queryIP` 的长度。
- 空间复杂度: $O(n)$ 。

0473. 火柴拼正方形

- 标签: 位运算、数组、动态规划、回溯、状态压缩
- 难度: 中等

题目链接

- [0473. 火柴拼正方形 - 力扣](#)

题目大意

描述: 给定一个表示火柴长度的数组 `matchsticks`, 其中 `matchsticks[i]` 表示第 i 根火柴的长度。

要求: 找出一种能使用所有火柴拼成一个正方形的方法。不能折断火柴, 可以将火柴连接起来, 并且每根火柴都要用到。如果能拼成正方形, 则返回 `True`, 否则返回 `False`。

说明:

- $1 \leq \text{matchsticks.length} \leq 15$ 。
- $1 \leq \text{matchsticks}[i] \leq 10^8$ 。

示例:

- 示例 1:

输入: `matchsticks = [1,1,2,2,2]`

输出: `True`

解释: 能拼成一个边长为 `2` 的正方形, 每边两根火柴。

- 示例 2:

输入: `matchsticks = [3,3,3,3,4]`

输出: `False`

解释: 不能用所有火柴拼成一个正方形。

解题思路

思路 1：回溯算法

- 先排除数组为空和火柴总长度不是 4 的倍数的情况, 直接返回 `False`。
- 然后将火柴按照从大到小排序。用数组 `sums` 记录四个边长分组情况。
- 将火柴分为 4 组, 把每一根火柴依次向 4 条边上放。
- 直到放置最后一根, 判断能否构成正方形, 若能构成正方形, 则返回 `True`, 否则返回 `False`。

思路 1：代码

```
class Solution:
    def dfs(self, index, sums, matchsticks, size, side_len):
        if index == size:
            return True

        for i in range(4):
            # 如果两条边的情况相等, 只需要计算一次, 没必要多次重复计算
            if i > 0 and sums[i] == sums[i - 1]:
                continue
            sums[i] += matchsticks[index]
            if sums[i] <= side_len and self.dfs(index + 1, sums, matchsticks, size, side_len):
                return True
            sums[i] -= matchsticks[index]

        return False

    def makesquare(self, matchsticks: List[int]) -> bool:
        if not matchsticks:
            return False
        size = len(matchsticks)
        sum_len = sum(matchsticks)
        if sum_len % 4 != 0:
            return False

        side_len = sum_len // 4
        matchsticks.sort(reverse=True)

        sums = [0 for _ in range(4)]
        return self.dfs(0, sums, matchsticks, size, side_len)
```

思路 1：复杂度分析

- 时间复杂度: $O(4^n)$ 。 n 是火柴的数目。
- 空间复杂度: $O(n)$ 。递归栈的空间复杂度为 $O(n)$ 。

0474. 一和零

- 标签：数组、字符串、动态规划
- 难度：中等

题目链接

- [0474. 一和零 - 力扣](#)

题目大意

描述：给定一个二进制字符串数组 $strs$ ，以及两个整数 m 和 n 。

要求：找出并返回 $strs$ 的最大子集的大小，该子集中最多有 m 个 0 和 n 个 1。

说明：

- 如果 x 的所有元素也是 y 的元素，集合 x 是集合 y 的子集。
- $1 \leq strs.length \leq 600$ 。
- $1 \leq strs[i].length \leq 100$ 。
- $strs[i]$ 仅由 '0' 和 '1' 组成。
- $1 \leq m, n \leq 100$ 。

示例：

- **示例 1：**

```
输入: strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
```

输出: 4

解释: 最多有 5 个 0 和 3 个 1 的最大子集是 {"10", "0001", "1", "0"}，因此答案是 4。

其他满足题意但较小的子集包括 {"0001", "1"} 和 {"10", "1", "0"}。{"111001"} 不满足题意，因为它含 4 个 1，大于 n 的值 3。

- **示例 2：**

```
输入: strs = ["10", "0", "1"], m = 1, n = 1
```

输出: 2

解释: 最大的子集是 {"0", "1"}，所以答案是 2。

解题思路

思路 1：动态规划

这道题可以转换为「二维 0-1 背包问题」来做。

把 0 的个数和 1 的个数视作一个二维背包的容量。每一个字符串都当做是一件物品，其成本为字符串中 1 的数量和 0 的数量，每个字符串的价值为 1。

1. 划分阶段

按照物品的序号、当前背包的载重上限进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：最多有 i 个 0 和 j 个 1 的字符串 $strs$ 的最大子集的大小。

3. 状态转移方程

填满最多由 i 个 0 和 j 个 1 构成的二维背包的最多物品数为下面两种情况中的最大值：

- 使用之前字符串填满容量为 $i - zero_num$ 、 $j - one_num$ 的背包的物品数 + 当前字符串价值
- 选择之前字符串填满容量为 i, j 的物品数。

则状态转移方程为: $dp[i][j] = \max(dp[i][j], dp[i - zero_num][j - one_num] + 1)$ 。

4. 初始条件

- 无论有多少个 0, 多少个 1, 只要不选 0, 也不选 1, 则最大子集的大小为 0。

5. 最终结果

根据我们之前定义的状态, $dp[i][j]$ 表示为: 最多有 i 个 0 和 j 个 1 的字符串 $strs$ 的最大子集的大小。所以最终结果为 $dp[m][n]$ 。

思路 1: 代码

```
class Solution:
    def findMaxForm(self, strs: List[str], m: int, n: int) -> int:
        dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

        for str in strs:
            one_num = 0
            zero_num = 0
            for ch in str:
                if ch == '0':
                    zero_num += 1
                else:
                    one_num += 1
            for i in range(m, zero_num - 1, -1):
                for j in range(n, one_num - 1, -1):
                    dp[i][j] = max(dp[i][j], dp[i - zero_num][j - one_num] + 1)

        return dp[m][n]
```

思路 1: 复杂度分析

- 时间复杂度: $O(l \times m \times n)$, 其中 l 为字符串 $strs$ 的长度。
- 空间复杂度: $O(m \times n)$ 。

0480. 滑动窗口中位数

- 标签: 数组、哈希表、滑动窗口、堆 (优先队列)
- 难度: 困难

题目链接

- [0480. 滑动窗口中位数 - 力扣](#)

题目大意

描述: 给定一个数组 $nums$, 有一个长度为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数, 每次窗口向右移动 1 位。

要求: 找出每次窗口移动后得到的新窗口中元素的中位数, 并输出由它们组成的数组。

说明:

- 中位数:** 有序序列最中间的那个数。如果序列的长度是偶数, 则没有最中间的数; 此时中位数是最中间的两个数的平均数。
- 例如:
 - $[2, 3, 4]$, 中位数是 3
 - $[2, 3]$, 中位数是 $(2 + 3)/2 = 2.5$ 。
- 你可以假设 k 始终有效, 即: k 始终小于等于输入的非空数组的元素个数。
- 与真实值误差在 10^{-5} 以内的答案将被视作正确答案。

示例:

- 示例 1:

给出 `nums = [1, 3, -1, -3, 5, 3, 6, 7]`, 以及 `k = 3`。

窗口位置 中位数

[1 3 -1] -3 5 3 6 7	1
1 [3 -1 -3] 5 3 6 7	-1
1 3 [-1 -3 5] 3 6 7	-1
1 3 -1 [-3 5 3] 6 7	3
1 3 -1 -3 [5 3 6] 7	5
1 3 -1 -3 5 [3 6 7]	6

因此, 返回该滑动窗口的中位数数组 `[1, -1, -1, 3, 5, 6]`。

解题思路

思路 1: 小顶堆 + 大顶堆

题目要求动态维护长度为 k 的窗口中元素的中位数。如果对窗口元素进行排序, 时间复杂度一般是 $O(k \times \log k)$ 。如果对每个区间都进行排序, 那时间复杂度就更大了, 肯定会超时。

我们需要借助一个内部有序的数据结构, 来降低取窗口中位数的时间复杂度。Python 可以借助 `heapq` 构建大顶堆和小顶堆。通过 k 的奇偶性和堆顶元素来获取中位数。

接下来还要考虑几个问题: 初始化问题、取中位数问题、窗口滑动中元素的添加删除操作。接下来一一解决。

初始化问题:

我们将所有大于中位数的元素放到 `heap_max` (小顶堆) 中, 并且元素个数向上取整。然后再将所有小于等于中位数的元素放到 `heap_min` (大顶堆) 中, 并且元素个数向下取整。这样当 k 为奇数时, `heap_max` 比 `heap_min` 多一个元素, 中位数就是 `heap_max` 堆顶元素。当 k 为偶数时, `heap_max` 和 `heap_min` 中的元素个数相同, 中位数就是 `heap_min` 堆顶元素和 `heap_max` 堆顶元素的平均数。这个过程操作如下:

- 先将数组中前 k 个元素放到 `heap_max` 中。
- 再从 `heap_max` 中取出 $k/2$ 个堆顶元素放到 `heap_min` 中。

取中位数问题 (上边提到过):

- 当 k 为奇数时, 中位数就是 `heap_max` 堆顶元素。当 k 为偶数时, 中位数就是 `heap_max` 堆顶元素和 `heap_min` 堆顶元素的平均数。

窗口滑动过程中元素的添加和删除问题:

- 删除: 每次滑动将窗口左侧元素删除。由于 `heapq` 没有提供删除中间特定元素相对应的方法。所以我们使用「延迟删除」的方式先把待删除的元素标记上, 等到待删除的元素出现在堆顶时, 再将其移除。我们使用 `removes` (哈希表) 来记录待删除元素个数。
 - 将窗口左侧元素删除的操作为: `removes[nums[left]] += 1`。
- 添加: 每次滑动在窗口右侧添加元素。需要根据上一步删除的结果来判断需要添加到哪一个堆上。我们用 `banlance` 记录 `heap_max` 和 `heap_min` 元素个数的差值。
 - 如果窗口左边界 `nums[left]` 小于等于 `heap_max` 堆顶元素, 则说明上一步删除的元素在 `heap_min` 上, 则让 `banlance -= 1`。
 - 如果窗口左边界 `nums[left]` 大于 `heap_max` 堆顶元素, 则说明上一步删除的元素在 `heap_max` 上, 则让 `banlance += 1`。
 - 如果窗口右边界 `nums[right]` 小于等于 `heap_max` 堆顶元素, 则说明待添加元素需要添加到 `heap_min` 上, 则让 `banlance += 1`。
 - 如果窗口右边界 `nums[right]` 大于 `heap_max` 堆顶元素, 则说明待添加元素需要添加到 `heap_max` 上, 则让 `banlance -= 1`。
- 经过上述操作, `banlance` 的取值为 0、-2、2 中的一种。需要经过调整使得 `banlance == 0`。
 - 如果 `banlance == 0`, 已经平衡, 不需要再做操作。
 - 如果 `banlance == -2`, 则说明 `heap_min` 比 `heap_max` 的元素多了两个。则从 `heap_min` 中取出堆顶元素添加到 `heap_max` 中。
 - 如果 `banlance == 2`, 则说明 `heap_max` 比 `heap_min` 的元素多了两个。则从 `heap_max` 中取出堆顶元素添加到 `heap_min` 中。
- 调整完之后, 分别检查 `heap_max` 和 `heap_min` 的堆顶元素。
 - 如果 `heap_max` 堆顶元素恰好为待删除元素, 即 `removes[-heap_max[0]] > 0`, 则弹出 `heap_max` 堆顶元素。
 - 如果 `heap_min` 堆顶元素恰好为待删除元素, 即 `removes[heap_min[0]] > 0`, 则弹出 `heap_min` 堆顶元素。
- 最后取中位数放入答案数组中, 然后继续滑动窗口。

思路 1：代码

```

import collections
import heapq

class Solution:
    def median(self, heap_max, heap_min, k):
        if k % 2 == 1:
            return -heap_max[0]
        else:
            return (-heap_max[0] + heap_min[0]) / 2

    def medianSlidingWindow(self, nums: List[int], k: int) -> List[float]:
        heap_max, heap_min = [], []
        removes = collections.Counter()

        for i in range(k):
            heapq.heappush(heap_max, -nums[i])
        for i in range(k // 2):
            heapq.heappush(heap_min, -heapq.heappop(heap_max))

        res = [self.median(heap_max, heap_min, k)]

        for i in range(k, len(nums)):
            banlance = 0
            left, right = i - k, i
            removes[nums[left]] += 1
            if heap_max and nums[left] <= -heap_max[0]:
                banlance -= 1
            else:
                banlance += 1

            if heap_max and nums[right] <= -heap_max[0]:
                heapq.heappush(heap_max, -nums[i])
                banlance += 1
            else:
                banlance -= 1
                heapq.heappush(heap_min, nums[i])

            if banlance == -2:
                heapq.heappush(heap_max, -heapq.heappop(heap_min))
            if banlance == 2:
                heapq.heappush(heap_min, -heapq.heappop(heap_max))

            while heap_max and removes[-heap_max[0]] > 0:
                removes[-heapq.heappop(heap_max)] -= 1
            while heap_min and removes[heap_min[0]] > 0:
                removes[heapq.heappop(heap_min)] -= 1
            res.append(self.median(heap_max, heap_min, k))

        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(n \times \log n)$ 。
- 空间复杂度: $O(n)$ 。

参考资料

- 【题解】《风险对冲》: 双堆对顶, 大堆小堆同时维护, 44ms - 滑动窗口中位数 - 力扣

0485. 最大连续 1 的个数

- 标签: 数组
- 难度: 简单

题目链接

- [0485. 最大连续 1 的个数 - 力扣](#)

题目大意

描述: 给定一个二进制数组 $nums$, 数组中只包含 0 和 1。

要求: 计算其中最大连续 1 的个数。

说明:

- $1 \leq nums.length \leq 10^5$ 。
- $nums[i]$ 不是 0 就是 1。

示例:

- **示例 1:**

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

解释: 开头的两位和最后的三位都是连续 1, 所以最大连续 1 的个数是 3.

- **示例 2:**

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

解题思路

思路 1: 一次遍历

1. 使用两个变量 cnt 和 ans 。 cnt 用于存储当前连续 1 的个数, ans 用于存储最大连续 1 的个数。
2. 然后进行一次遍历, 统计当前连续 1 的个数, 并更新最大的连续 1 个数。
3. 最后返回 ans 作为答案。

思路 1: 代码

```
class Solution:
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:
        ans = 0
        cnt = 0
        for num in nums:
            if num == 1:
                cnt += 1
                ans = max(ans, cnt)
            else:
                cnt = 0
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0486. 预测赢家

- 标签：递归、数组、数学、动态规划、博弈
- 难度：中等

题目链接

- [0486. 预测赢家 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ 。玩家 1 和玩家 2 基于这个数组设计了一个游戏。

玩家 1 和玩家 2 轮流进行自己的回合，玩家 1 先手。

开始时，两个玩家的初始分值都是 0。每一回合，玩家从数组的任意一端取一个数字（即 $nums[0]$ 或 $nums[nums.length - 1]$ ），取到的数字将会从数组中移除（数组长度减 1）。玩家选中的数字将会加到他的得分上。当数组中没有剩余数字可取时，游戏结束。

要求：如果玩家 1 能成为赢家，则返回 `True`。否则返回 `False`。如果两个玩家得分相等，同样认为玩家 1 是游戏的赢家，也返回 `True`。假设每个玩家的玩法都会使他的分数最大化。

说明：

- $1 \leq nums.length \leq 20$ 。
- $0 \leq nums[i] \leq 10^7$ 。

示例：

- 示例 1：

输入：`nums = [1, 5, 2]`

输出：`False`

解释：一开始，玩家 1 可以从 1 和 2 中进行选择。

如果他选择 2 (或者 1)，那么玩家 2 可以从 1 (或者 2) 和 5 中进行选择。如果玩家 2 选择了 5，那么玩家 1 则只剩下 1 (或者 2) 可选。

所以，玩家 1 的最终分为 $1 + 2 = 3$ ，而玩家 2 为 5。

因此，玩家 1 永远不会成为赢家，返回 `False`。

- 示例 2：

输入：`nums = [1, 5, 233, 7]`

输出：`True`

解释：玩家 1 一开始选择 1。然后玩家 2 必须从 5 和 7 中进行选择。无论玩家 2 选择了哪个，玩家 1 都可以选择 233。

最终，玩家 1 (234 分) 比玩家 2 (12 分) 获得更多的分数，所以返回 `True`，表示玩家 1 可以成为赢家。

解题思路

思路 1：动态规划

1. 划分阶段

按照区间长度进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：玩家 1 与玩家 2 在 $nums[i] \dots nums[j]$ 之间互相选取，玩家 1 比玩家 2 多的最大分数。

3. 状态转移方程

根据状态的定义，只有在 $i \leq j$ 时才有意义，所以当 $i > j$ 时， $dp[i][j] = 0$ 。

1. 当 $i == j$ 时，当前玩家只能拿取 $nums[i]$ ，因此对于所有 $0 \leq i < nums.length$ ，都有： $dp[i][i] = nums[i]$ 。
2. 当 $i < j$ 时，当前玩家可以选择 $nums[i]$ 或 $nums[j]$ ，并是自己的分数最大化，然后换另一位玩家从剩下部分选取数字。则转移方程为： $dp[i][j] = \max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1])$ 。

4. 初始条件

- 当 $i > j$ 时， $dp[i][j] = 0$ 。
- 当 $i == j$ 时， $dp[i][j] = nums[i]$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：玩家 1 与玩家 2 在 $nums[i] \dots nums[j]$ 之间互相选取，玩家 1 比玩家 2 多的最大分数。则如果玩家 1 想要赢，则 $dp[0][size - 1]$ 必须大于等于 0。所以最终结果为 $dp[0][size - 1] \geq 0$ 。

思路 1：代码

```
class Solution:
    def PredictTheWinner(self, nums: List[int]) -> bool:
        size = len(nums)
        dp = [[0 for _ in range(size)] for _ in range(size)]

        for l in range(1, size + 1):
            for i in range(size):
                j = i + l - 1
                if j >= size:
                    break
                if l == 1:
                    dp[i][j] = nums[i]
                else:
                    dp[i][j] = max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1])
        return dp[0][size - 1] >= 0
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(n^2)$ 。

0487. 最大连续1的个数 II

- 标签：数组、动态规划、滑动窗口
- 难度：中等

题目链接

- [0487. 最大连续1的个数 II - 力扣](#)

题目大意

描述：给定一个二进制数组 $nums$ ，可以最多将 1 个 0 翻转为 1。

要求：如果最多可以翻转一个 0，则返回数组中连续 1 的最大个数。

说明：

- $1 \leq \text{nums.length} \leq 105$
- $\text{nums}[i]$ 不是 0 就是 1。

示例：

- 示例 1：

```
输入: nums = [1, 0, 1, 1, 0]
输出: 4
解释: 翻转第一个 0 可以得到最长的连续 1。当翻转以后, 最大连续 1 的个数为 4。
```

- 示例 2：

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

解题思路

思路 1：滑动窗口

暴力做法是尝试将每个位置的 0 分别变为 1，然后统计最大连续 1 的个数。但这样复杂度就太高了。

我们可以使用滑动窗口来解决问题。保证滑动窗口内最多有 1 个 0。具体做法如下：

设定两个指针： $left$ 、 $right$ ，分别指向滑动窗口的左右边界，保证滑动窗口内最多有 1 个 0。使用 $zero_count$ 统计窗口内 1 的个数。使用 ans 记录答案。

- 一开始， $left$ 、 $right$ 都指向 0。
- 如果 $\text{nums}[right] == 0$ ，则窗口内 1 的个数加 1。
- 如果该窗口中 1 的个数多于 1 个，即 $zero_count > 1$ ，则不断右移 $left$ ，缩小滑动窗口长度，并更新窗口中 1 的个数，直到 $zero_count \leq 1$ 。
- 维护更新最大连续 1 的个数。然后右移 $right$ ，直到 $right \geq \text{len}(\text{nums})$ 结束。
- 输出最大连续 1 的个数。

思路 1：代码

```
class Solution:
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:
        left, right = 0, 0
        ans = 0
        zero_count = 0

        while right < len(nums):
            if nums[right] == 0:
                zero_count += 1
            while zero_count > 1:
                if nums[left] == 0:
                    zero_count -= 1
                left += 1
            ans = max(ans, right - left + 1)
            right += 1

        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为数组 nums 的长度。
- 空间复杂度： $O(1)$ 。

0491. 递增子序列

- 标签：位运算、数组、哈希表、回溯
- 难度：中等

题目链接

- [0491. 递增子序列 - 力扣](#)

题目大意

给定一个整数数组 `nums`，找出并返回该数组的所有递增子序列，递增子序列的长度至少为 2。

解题思路

可以利用回溯算法求解。

建立两个数组 `res`、`path`。`res` 用于存放所有递增子序列，`path` 用于存放当前的递增子序列。

定义回溯方法，从 `start_index = 0` 的位置开始遍历。

- 如果当前子序列的长度大于等于 2，则将当前递增子序列添加到 `res` 数组中（注意：不用返回，因为还要继续向下查找）
- 对数组 `[start_index, len(nums) - 1]` 范围内的元素进行取值，判断当前元素是否在本层出现过。如果出现过则跳出循环。
 - 将 `nums[i]` 标记为使用过。
 - 将 `nums[i]` 加入到当前 `path` 中。
 - 继续从 `i + 1` 开始遍历下一节点。
 - 进行回退操作。
- 最终返回 `res` 数组。

代码

```
class Solution:
    res = []
    path = []
    def backtrack(self, nums: List[int], start_index):
        if len(self.path) > 1:
            self.res.append(self.path[:])

        num_set = set()
        for i in range(start_index, len(nums)):
            if self.path and nums[i] < self.path[-1] or nums[i] in num_set:
                continue

            num_set.add(nums[i])
            self.path.append(nums[i])
            self.backtrack(nums, i + 1)
            self.path.pop()

    def findSubsequences(self, nums: List[int]) -> List[List[int]]:
        self.res.clear()
        self.path.clear()
        self.backtrack(nums, 0)
        return self.res
```

0494. 目标和

- 标签：数组、动态规划、回溯
- 难度：中等

题目链接

- [0494. 目标和 - 力扣](#)

题目大意

描述：给定一个整数数组 $nums$ 和一个整数 $target$ 。数组长度不超过 20。向数组中每个整数前加 `+` 或 `-`。然后串联起来构成一个表达式。

要求：返回通过上述方法构造的、运算结果等于 $target$ 的不同表达式数目。

说明：

- $1 \leq nums.length \leq 20$ 。
- $0 \leq nums[i] \leq 1000$ 。
- $0 \leq sum(nums[i]) \leq 1000$ 。
- $-1000 \leq target \leq 1000$ 。

示例：

- **示例 1：**

```
输入: nums = [1,1,1,1,1], target = 3
```

```
输出: 5
```

解释：一共有 5 种方法让最终目标和为 3。

```
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

- **示例 2：**

```
输入: nums = [1], target = 1
```

```
输出: 1
```

解题思路

思路 1：深度优先搜索（超时）

使用深度优先搜索对每位数字进行 `+` 或者 `-`，具体步骤如下：

1. 定义从位置 0、和为 0 开始，到达数组尾部位置为止，和为 $target$ 的方案数为 `dfs(0, 0)`。
2. 下面从位置 0、和为 0 开始，以深度优先搜索遍历每个位置。
3. 如果当前位置 i 到达最后一个位置 $size$ ：
 - i. 如果和 cur_sum 等于目标和 $target$ ，则返回方案数 1。
 - ii. 如果和 cur_sum 不等于目标和 $target$ ，则返回方案数 0。
4. 递归搜索 $i + 1$ 位置，和为 $cur_sum - nums[i]$ 的方案数。
5. 递归搜索 $i + 1$ 位置，和为 $cur_sum + nums[i]$ 的方案数。
6. 将 4 ~ 5 两个方案数加起来就是当前位置 i 、和为 cur_sum 的方案数，返回该方案数。
7. 最终方案数为 `dfs(0, 0)`，将其作为答案返回即可。

思路 1：代码

```

class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        size = len(nums)

        def dfs(i, cur_sum):
            if i == size:
                if cur_sum == target:
                    return 1
                else:
                    return 0
            ans = dfs(i + 1, cur_sum - nums[i]) + dfs(i + 1, cur_sum + nums[i])
            return ans

        return dfs(0, 0)

```

思路 1：复杂度分析

- 时间复杂度： $O(2^n)$ 。其中 n 为数组 $nums$ 的长度。
- 空间复杂度： $O(n)$ 。递归调用的栈空间深度不超过 n 。

思路 2：记忆化搜索

在思路 1 中我们单独使用深度优先搜索对每位数字进行 `+` 或者 `-` 的方法超时了。所以我们考虑使用记忆化搜索的方式，避免进行重复搜索。

这里我们使用哈希表

table

记录遍历过的位置 i 及所得到的当前和 cur_sum 下的方案数，来避免重复搜索。具体步骤如下：

1. 定义从位置 0、和为 0 开始，到达数组尾部位置为止，和为 $target$ 的方案数为 `dfs(0, 0)`。
2. 下面从位置 0、和为 0 开始，以深度优先搜索遍历每个位置。
3. 如果当前位置 i 遍历完所有位置：
 - i. 如果和 cur_sum 等于目标和 $target$ ，则返回方案数 1。
 - ii. 如果和 cur_sum 不等于目标和 $target$ ，则返回方案数 0。
4. 如果当前位置 i 、和为 cur_sum 之前记录过（即使用 *table* 记录过对应方案数），则返回该方案数。
5. 如果当前位置 i 、和为 cur_sum 之前没有记录过，则：
 - i. 递归搜索 $i + 1$ 位置，和为 $cur_sum - nums[i]$ 的方案数。
 - ii. 递归搜索 $i + 1$ 位置，和为 $cur_sum + nums[i]$ 的方案数。
 - iii. 将上述两个方案数加起来就是当前位置 i 、和为 cur_sum 的方案数，将其记录到哈希表 *table* 中，并返回该方案数。
6. 最终方案数为 `dfs(0, 0)`，将其作为答案返回即可。

思路 2：代码

```

class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        size = len(nums)
        table = dict()

        def dfs(i, cur_sum):
            if i == size:
                if cur_sum == target:
                    return 1
                else:
                    return 0

            if (i, cur_sum) in table:
                return table[(i, cur_sum)]

            cnt = dfs(i + 1, cur_sum - nums[i]) + dfs(i + 1, cur_sum + nums[i])
            table[(i, cur_sum)] = cnt
            return cnt

        return dfs(0, 0)

```

思路 2：复杂度分析

- 时间复杂度: $O(2^n)$ 。其中 n 为数组 $nums$ 的长度。
- 空间复杂度: $O(n)$ 。递归调用的栈空间深度不超过 n 。

思路 3：动态规划

假设数组中所有元素和为 sum , 数组中所有符号为 $+$ 的元素为 sum_x , 符号为 $-$ 的元素和为 sum_y 。则 $target = sum_x - sum_y$ 。

而 $sum_x + sum_y = sum$ 。根据两个式子可以求出 $2 \times sum_x = target + sum$, 即 $sum_x = (target + sum)/2$ 。

那么这道题就变成了, 如何在数组中找到一个集合, 使集合中元素和为 $(target + sum)/2$ 。这就变为了「0-1 背包问题」中求装满背包的方案数问题。

1. 定义状态

定义状态 $dp[i]$ 表示为: 填满容量为 i 的背包, 有 $dp[i]$ 种方法。

2. 状态转移方程

填满容量为 i 的背包的方法数来源于:

- 不使用当前 num : 只使用之前元素填满容量为 i 的背包的方法数。
- 使用当前 num : 填满容量 $i - num$ 的包的方法数, 再填入 num 的方法数。

则动态规划的状态转移方程为: $dp[i] = dp[i] + dp[i - num]$ 。

3. 初始化

初始状态下, 默认填满容量为 0 的背包有 1 种办法 (什么也不装)。即 $dp[0] = 1$ 。

4. 最终结果

根据状态定义, 最后输出 $dp[size]$ (即填满容量为 $size$ 的背包, 有 $dp[size]$ 种方法) 即可, 其中 $size$ 为数组 $nums$ 的长度。

思路 3：代码

```

class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        sum_nums = sum(nums)
        if abs(target) > abs(sum_nums) or (target + sum_nums) % 2 == 1:
            return 0
        size = (target + sum_nums) // 2
        dp = [0 for _ in range(size + 1)]
        dp[0] = 1
        for num in nums:
            for i in range(size, num - 1, -1):
                dp[i] = dp[i] + dp[i - num]
        return dp[size]

```

思路 3：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为数组 $nums$ 的长度。
- 空间复杂度: $O(n)$ 。

0496. 下一个更大元素 I

- 标签: 栈、数组、哈希表、单调栈
- 难度: 简单

题目链接

- [0496. 下一个更大元素 I - 力扣](#)

题目大意

描述: 给定两个没有重复元素的数组 `nums1` 和 `nums2`，其中 `nums1` 是 `nums2` 的子集。

要求: 找出 `nums1` 中每个元素在 `nums2` 中的下一个比其大的值。

说明:

- `nums1` 中数字 x 的下一个更大元素是指: x 在 `nums2` 中对应位置的右边的第一个比 x 大的元素。如果不存在, 对应位置输出 `-1`。
- $1 \leq \text{nums1.length} \leq \text{nums2.length} \leq 1000$ 。
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^4$ 。
- `nums1` 和 `nums2` 中所有整数互不相同。
- `nums1` 中的所有整数同样出现在 `nums2` 中。

示例:

- 示例 1:

```

输入: nums1 = [4,1,2], nums2 = [1,3,4,2].
输出: [-1,3,-1]
解释: nums1 中每个值的下一个更大元素如下所述:
- 4 , 用加粗斜体标识, nums2 = [1,3,4,2]。不存在下一个更大元素, 所以答案是 -1 。
- 1 , 用加粗斜体标识, nums2 = [1,3,4,2]。下一个更大元素是 3 。
- 2 , 用加粗斜体标识, nums2 = [1,3,4,2]。不存在下一个更大元素, 所以答案是 -1 。

```

- 示例 2:

```
输入: nums1 = [2,4], nums2 = [1,2,3,4].
输出: [3,-1]
解释: nums1 中每个值的下一个更大元素如下所述:
- 2 , 用加粗斜体标识, nums2 = [1,2,3,4]。下一个更大元素是 3 。
- 4 , 用加粗斜体标识, nums2 = [1,2,3,4]。不存在下一个更大元素, 所以答案是 -1 。
```

解题思路

最直接的思路是根据题意直接暴力求解。遍历 `nums1` 中的每一个元素。对于 `nums1` 的每一个元素 `nums1[i]`，再遍历一遍 `nums2`，查找 `nums2` 中对应位置右边第一个比 `nums1[i]` 大的元素。这种解法的时间复杂度是 $O(n^2)$ 。

另一种思路是单调栈。

思路 1：单调栈

因为 `nums1` 是 `nums2` 的子集，所以我们可以先遍历一遍 `nums2`，并构造单调递增栈，求出 `nums2` 中每个元素右侧下一个更大的元素。然后将其存储到哈希表中。然后再遍历一遍 `nums1`，从哈希表中取出对应结果，存放到答案数组中。这种解法的时间复杂度是 $O(n)$ 。具体做法如下：

1. 使用数组 `res` 存放答案。使用 `stack` 表示单调递增栈。使用哈希表 `num_map` 用于存储 `nums2` 中下一个比当前元素大的数值，映射关系为
当前元素值: 下一个比当前元素大的数值。
2. 遍历数组 `nums2`，对于当前元素：
 - i. 如果当前元素值较小，则直接让当前元素值入栈。
 - ii. 如果当前元素值较大，则一直出栈，直到当前元素值小于栈顶元素。
a. 出栈时，第一个大于栈顶元素值的元素，就是当前元素。则将其映射到 `num_map` 中。
3. 遍历完数组 `nums2`，建立好所有元素下一个更大元素的映射关系之后，再遍历数组 `nums1`。
4. 从 `num_map` 中取出对应的值，将其加入到答案数组中。
5. 最终输出答案数组 `res`。

思路 1：代码

```
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        res = []
        stack = []
        num_map = dict()
        for num in nums2:
            while stack and num > stack[-1]:
                num_map[stack[-1]] = num
                stack.pop()
            stack.append(num)

        for num in nums1:
            res.append(num_map.get(num, -1))
        return res
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

0498. 对角线遍历

- 标签: 数组、矩阵、模拟
- 难度: 中等

题目链接

- 0498. 对角线遍历 - 力扣

题目大意

描述：给定一个大小为 $m \times n$ 的矩阵 mat 。

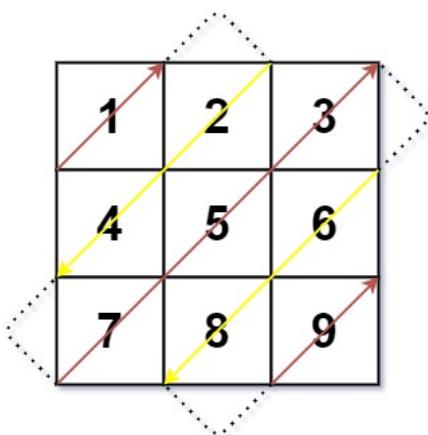
要求：以对角线遍历的顺序，用一个数组返回这个矩阵中的所有元素。

说明：

- $m == mat.length$ 。
- $n == mat[i].length$ 。
- $1 \leq m, n \leq 10^4$ 。
- $1 \leq m \times n \leq 10^4$ 。
- $-10^5 \leq mat[i][j] \leq 10^5$ 。

示例：

- 示例 1：



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

- 示例 2：

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

解题思路

思路 1：找规律 + 考虑边界问题

这道题的关键是「找规律」和「考虑边界问题」。

找规律：

1. 当「行号 + 列号」为偶数时，遍历方向为从左下到右上。可以记为右上方向 $(-1, +1)$ ，即行号减 1，列号加 1。
2. 当「行号 + 列号」为奇数时，遍历方向为从右上到左下。可以记为左下方向 $(+1, -1)$ ，即行号加 1，列号减 1。

边界情况：

1. 向右上方向移动时：
 - i. 如果在最后一列，则向下方移动，即 $x += 1$ 。

- ii. 如果在第一行，则向右方移动，即 `y += 1`。
 - iii. 其余情况想右上方向移动，即 `x -= 1`、`y += 1`。
2. 向左下方向移动时：
- i. 如果在最后一行，则向右方移动，即 `y += 1`。
 - ii. 如果在第一列，则向下方移动，即 `x += 1`。
 - iii. 其余情况向左下方向移动，即 `x += 1`、`y -= 1`。

思路 1：代码

```
class Solution:
    def findDiagonalOrder(self, mat: List[List[int]]) -> List[int]:
        rows = len(mat)
        cols = len(mat[0])
        count = rows * cols
        x, y = 0, 0
        ans = []

        for i in range(count):
            ans.append(mat[x][y])

            if (x + y) % 2 == 0:
                # 最后一列
                if y == cols - 1:
                    x += 1
                # 第一行
                elif x == 0:
                    y += 1
                # 右上方方向
                else:
                    x -= 1
                    y += 1
            else:
                # 最后一行
                if x == rows - 1:
                    y += 1
                # 第一列
                elif y == 0:
                    x += 1
                # 左下方向
                else:
                    x += 1
                    y -= 1
        return ans
```

思路 1：复杂度分析

- 时间复杂度： $O(m \times n)$ 。其中 m 、 n 分别为二维矩阵的行数、列数。
- 空间复杂度： $O(m \times n)$ 。如果算上答案数组的空间占用，则空间复杂度为 $O(m \times n)$ 。不算上则空间复杂度为 $O(1)$ 。

参考资料

- 【题解】「498. 对角线遍历」最简单易懂! - 对角线遍历 - 力扣 (LeetCode)

0501. 二叉搜索树中的众数

- 标签：树、深度优先搜索、二叉搜索树、二叉树
- 难度：简单

题目链接

- [0501. 二叉搜索树中的众数 - 力扣](#)

题目大意

给定一个有相同值的二叉搜索树 (BST)，要求找出 BST 中所有众数（出现频率最高的元素）。

二叉搜索树定义：

- 若左子树不为空，则左子树上所有节点值均小于它的根节点值；
- 若右子树不为空，则右子树上所有节点值均大于它的根节点值；
- 任意节点的左、右子树也分别为二叉搜索树。

解题思路

中序递归遍历二叉搜索树所得到的结果是一个有序数组，所以问题就变为了如何统计有序数组的众数。

定义几个变量。`count` 用来统计当前元素值对应的节点个数，`max_count` 用来元素出现次数最多的次数。数组 `res` 用来存储所有众数结果（因为众数可能不止一个）。

因为中序递归遍历二叉树，比较的元素肯定是相邻节点，所以需要再使用一个变量 `pre` 来指向前一节点。下面就开始愉快的递归了。

- 如果当前节点为空，直接返回。
- 递归遍历左子树。
- 比较当前节点和前一节点：
 - 如果前一节点为空，则当前元素频率赋值为 1。
 - 如果前一节点值与当前节点值相同，则当前元素频率 + 1。
 - 如果前一节点值与当前节点值不同，则重新计算当前元素频率，将当前元素频率赋值为 1。
- 判断当前元素频率和最高频率关系：
 - 如果当前元素频率和最高频率值相等，则将对应元素值加入 `res` 数组。
 - 如果当前元素频率大于最高频率值，则更新最高频率值，并清空原 `res` 数组，将当前元素加入 `res` 数组。
- 递归遍历右子树。

最终得到的 `res` 数组即为所求的众数。

代码

```

class Solution:
    res = []
    count = 0
    max_count = 0
    pre = None

    def search(self, cur: TreeNode):
        if not cur:
            return
        self.search(cur.left)
        if not self.pre:
            self.count = 1
        elif self.pre.val == cur.val:
            self.count += 1
        else:
            self.count = 1

        self.pre = cur

        if self.count == self.max_count:
            self.res.append(cur.val)
        elif self.count > self.max_count:
            self.max_count = self.count
            self.res.clear()
            self.res.append(cur.val)

        self.search(cur.right)
        return

    def findMode(self, root: TreeNode) -> List[int]:
        self.count = 0
        self.max_count = 0
        self.res.clear()
        self.pre = None
        self.search(root)
        return self.res

```

0503. 下一个更大元素 II

- 标签: 栈、数组、单调栈
- 难度: 中等

题目链接

- [0503. 下一个更大元素 II - 力扣](#)

题目大意

给定一个循环数组 `nums`（最后一个元素的下一个元素是数组的第一个元素）。

要求：输出每个元素的下一个更大元素。如果不存在，则输出 `-1`。

- 数字 `x` 的下一个更大的元素：按数组遍历顺序，这个数字之后的第一个比它更大的数。这意味着你应该循环地搜索它的下一个更大的数。

解题思路

第一种思路是根据题意直接暴力求解。遍历 `nums` 中的每一个元素。对于 `nums` 的每一个元素 `nums[i]`，查找 `nums[i]` 右边第一个比 `nums[i]` 大的元素。这种解法的时间复杂度是 $O(n^2)$ 。

第二种思路是使用单调递增栈。遍历数组 `nums`，构造单调递增栈，求出 `nums` 中每个元素右侧下一个更大的元素。然后将其存储到答案数组中。这种解法的时间复杂度是 $O(n)$ 。

而循环数组的求解方法可以将 `nums` 复制一份到末尾，生成长度为 `len(nums) * 2` 的数组，或者通过取模运算将下标映射到 `0 ~ len(nums) * 2 - 1` 之间。

具体做法如下：

- 使用数组 `res` 存放答案，初始值都赋值为 `-1`。使用变量 `stack` 表示单调递增栈。
- 遍历数组 `nums`，对于当前元素：
 - 如果当前元素值小于栈顶元素，则说明当前元素「下一个更大元素」与栈顶元素的「下一个更大元素」相同。应该直接让当前元素的下标入栈。
 - 如果当前元素值大于栈顶元素，则说明当前元素是之前元素的「下一个更大元素」，则不断将栈顶元素出栈。直到当前元素值小于栈顶元素值。
 - 出栈时，出栈元素的「下一个更大元素」是当前元素。则将当前元素值存入到答案数组 `res` 中出栈元素所对应的位置中。
- 最终输出答案数组 `res`。

代码

```
size = len(nums)
res = [-1 for _ in range(size)]
stack = []
for i in range(size * 2):
    while stack and nums[i % size] > nums[stack[-1]]:
        index = stack.pop()
        res[index] = nums[i % size]
    stack.append(i % size)

return res
```

0504. 七进制数

- 标签：数学
- 难度：简单

题目链接

- [0504. 七进制数 - 力扣](#)

题目大意

描述：给定一个整数 `num`。

要求：将其转换为 7 进制数，并以字符串形式输出。

说明：

- $-10^7 \leq num \leq 10^7$ 。

示例：

- 示例 1：

```
输入: num = 100
输出: "202"
```

- 示例 2：

输入: num = -7
输出: "-10"

解题思路

思路 1：模拟

1. num 不断对 7 取余整除。
2. 然后将取到的余数进行拼接成字符串即可。

思路 1：代码

```
class Solution:
    def convertToBase7(self, num: int) -> str:
        if num == 0:
            return "0"
        if num < 0:
            return "-" + self.convertToBase7(-num)
        ans = ""
        while num:
            ans = str(num % 7) + ans
            num //= 7
        return ans
```

思路 1：复杂度分析

- 时间复杂度: $O(\log |n|)$ 。
- 空间复杂度: $O(\log |n|)$ 。

0506. 相对名次

- 标签: 数组、排序、堆（优先队列）
- 难度: 简单

题目链接

- [0506. 相对名次 - 力扣](#)

题目大意

描述: 给定一个长度为 n 的数组 $score$ 。其中 $score[i]$ 表示第 i 名运动员在比赛中的成绩。所有成绩互不相同。

要求: 找出他们的相对名次，并授予前三名对应的奖牌。前三名运动员将会被分别授予「金牌（"Gold Medal"）」，「银牌（"Silver Medal"）」和「铜牌（"Bronze Medal"）」。

说明:

- $n == score.length$ 。
- $1 \leq n \leq 10^4$ 。
- $0 \leq score[i] \leq 10^6$ 。
- $score$ 中的所有值互不相同。

示例:

- 示例 1:

```
输入: score = [5,4,3,2,1]
输出: ["Gold Medal","Silver Medal","Bronze Medal","4","5"]
解释: 名次为 [1st, 2nd, 3rd, 4th, 5th] 。
```

- 示例 2:

```
输入: score = [10,3,8,9,4]
输出: ["Gold Medal","5","Bronze Medal","Silver Medal","4"]
解释: 名次为 [1st, 5th, 3rd, 2nd, 4th] 。
```

解题思路

思路 1：排序

- 先对数组 `score` 进行排序。
- 再将对应前三个位置上的元素替换成对应的字符串: `"Gold Medal"`, `"Silver Medal"`, `"Bronze Medal"`。

思路 1：代码

```
class Solution:
    def shellSort(self, arr):
        size = len(arr)
        gap = size // 2

        while gap > 0:
            for i in range(gap, size):
                temp = arr[i]
                j = i
                while j >= gap and arr[j - gap] < temp:
                    arr[j] = arr[j - gap]
                    j -= gap
                arr[j] = temp
            gap = gap // 2
        return arr

    def findRelativeRanks(self, score: List[int]) -> List[str]:
        nums = score.copy()
        nums = self.shellSort(nums)
        score_map = dict()
        for i in range(len(score)):
            score_map[nums[i]] = i + 1

        res = []
        for i in range(len(score)):
            if score[i] == nums[0]:
                res.append("Gold Medal")
            elif score[i] == nums[1]:
                res.append("Silver Medal")
            elif score[i] == nums[2]:
                res.append("Bronze Medal")
            else:
                res.append(str(score_map[score[i]]))
        return res
```

思路 1：复杂度分析

- 时间复杂度: $O(n \times \log n)$ 。因为采用了时间复杂度为 $O(n \times \log n)$ 的希尔排序。
- 空间复杂度: $O(n)$ 。

0509. 斐波那契数

- 标签: 递归、记忆化搜索、数学、动态规划
- 难度: 简单

题目链接

- [0509. 斐波那契数 - 力扣](#)

题目大意

描述: 给定一个整数 n 。

要求: 计算第 n 个斐波那契数。

说明:

- 斐波那契数列的定义如下:
 - $f(0) = 0, f(1) = 1$ 。
 - $f(n) = f(n - 1) + f(n - 2)$, 其中 $n > 1$ 。
- $0 \leq n \leq 30$ 。

示例:

- **示例 1:**

```
输入: n = 2
输出: 1
解释: F(2) = F(1) + F(0) = 1 + 0 = 1
```

- **示例 2:**

```
输入: n = 3
输出: 2
解释: F(3) = F(2) + F(1) = 1 + 1 = 2
```

解题思路

思路 1：递归算法

根据我们的递推三步走策略，写出对应的递归代码。

1. 写出递推公式: $f(n) = f(n - 1) + f(n - 2)$ 。
2. 明确终止条件: $f(0) = 0, f(1) = 1$ 。
3. 翻译为递归代码:
 - i. 定义递归函数: `fib(self, n)` 表示输入参数为问题的规模 n ，返回结果为第 n 个斐波那契数。
 - ii. 书写递归主体: `return self.fib(n - 1) + self.fib(n - 2)`。
 - iii. 明确递归终止条件:
 - a. `if n == 0: return 0`
 - b. `if n == 1: return 1`

思路 1：代码

```
class Solution:
    def fib(self, n: int) -> int:
        if n == 0:
            return 0
        if n == 1:
            return 1
        return self.fib(n - 1) + self.fib(n - 2)
```

思路 1：复杂度分析

- 时间复杂度： $O((\frac{1+\sqrt{5}}{2})^n)$ 。具体证明方法参考 [递归求斐波那契数列的时间复杂度，不要被网上的答案误导了 - 知乎](#)。
- 空间复杂度： $O(n)$ 。每次递归的空间复杂度是 $O(1)$ ，调用栈的深度为 n ，所以总的空间复杂度就是 $O(n)$ 。

思路 2：动态规划算法

1. 划分阶段

我们可以按照整数顺序进行阶段划分，将其划分为整数 $0 \sim n$ 。

2. 定义状态

定义状态 $dp[i]$ 为：第 i 个斐波那契数。

3. 状态转移方程

根据题目中所给的斐波那契数列的定义 $f(n) = f(n - 1) + f(n - 2)$ ，则直接得出状态转移方程为 $dp[i] = dp[i - 1] + dp[i - 2]$ 。

4. 初始条件

根据题目中所给的初始条件 $f(0) = 0, f(1) = 1$ 确定动态规划的初始条件，即 $dp[0] = 0, dp[1] = 1$ 。

5. 最终结果

根据状态定义，最终结果为 $dp[n]$ ，即第 n 个斐波那契数为 $dp[n]$ 。

思路 2：代码

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

        dp = [0 for _ in range(n + 1)]
        dp[0] = 0
        dp[1] = 1
        for i in range(2, n + 1):
            dp[i] = dp[i - 2] + dp[i - 1]

        return dp[n]
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。一重循环遍历的时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。因为 $dp[i]$ 的状态只依赖于 $dp[i - 1]$ 和 $dp[i - 2]$ ，所以可以使用 3 个变量来分别表示 $dp[i]$ 、 $dp[i - 1]$ 、 $dp[i - 2]$ ，从而将空间复杂度优化到 $O(1)$ 。

0513. 找树左下角的值

- 标签：树、深度优先搜索、广度优先搜索、二叉树
- 难度：中等

题目链接

- 0513. 找树左下角的值 - 力扣

题目大意

描述：给定一个二叉树的根节点 `root`。

要求：找出该二叉树「最底层」的「最左边」节点的值。

说明：

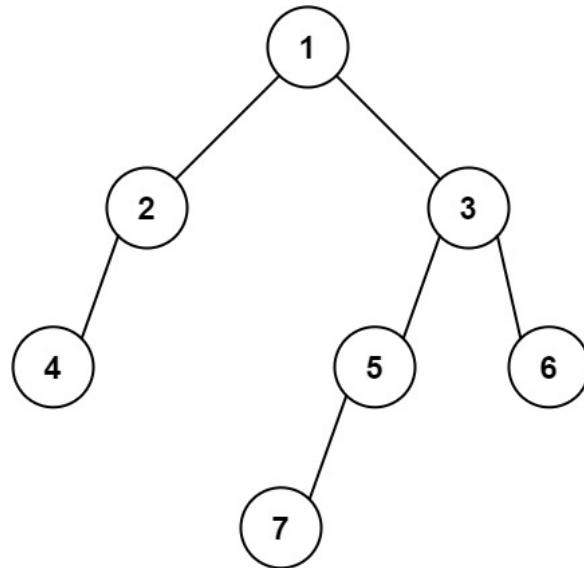
- 假设二叉树中至少有一个节点。
- 二叉树的节点个数的范围是 $[1, 10^4]$ 。
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$ 。

示例：

- 示例 1：

输入： [1, 2, 3, 4, null, 5, 6, null, null, 7]

输出： 7



解题思路

思路 1：层序遍历

这个问题可以拆分为两个问题：

1. 如何找到「最底层」。
2. 在「最底层」如何找到最左边的节点。

第一个问题，我们可以通过层序遍历直接确定最底层节点。而第二个问题可以通过改变层序遍历的左右节点访问顺序从而找到「最底层」的「最左边节点」。具体方法如下：

1. 对二叉树进行层序遍历。每层元素先访问右节点，再访问左节点。
2. 当遍历到最后一个元素时，此时最后一个元素就是「最底层」的「最左边」节点，即左下角的节点，将该节点的值返回即可。

思路 1：层序遍历代码

```
import collections
class Solution:
    def findBottomLeftValue(self, root: TreeNode) -> int:
        if not root:
            return -1
        queue = collections.deque()
        queue.append(root)
        while queue:
            cur = queue.popleft()
            if cur.right:
                queue.append(cur.right)
            if cur.left:
                queue.append(cur.left)
        return cur.val
```

0515. 在每个树行中找最大值

- 标签：树、深度优先搜索、广度优先搜索、二叉树
- 难度：中等

题目链接

- [0515. 在每个树行中找最大值 - 力扣](#)

题目大意

给定一棵二叉树的根节点 `root`。

要求：找出二叉树中每一层的最大值。

解题思路

利用队列进行层序遍历，并记录下每一层的最大值，将其存入答案数组中。

代码

0516. 最长回文子序列

- 标签：字符串、动态规划
- 难度：中等

题目链接

- [0516. 最长回文子序列 - 力扣](#)

题目大意

描述：给定一个字符串 `s`。

要求：找出其中最长的回文子序列，并返回该序列的长度。

说明：

- **子序列**: 不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。
- $1 \leq s.length \leq 1000$ 。
- s 仅由小写英文字母组成。

示例：

- 示例 1:

```
输入: s = "bbbabb"
输出: 4
解释: 一个可能的最长回文子序列为 "bbbb"。
```

- 示例 2:

```
输入: s = "cbbd"
输出: 2
解释: 一个可能的最长回文子序列为 "bb"。
```

解题思路

思路 1：动态规划

1. 划分阶段

按照区间长度进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：字符串 s 在区间 $[i, j]$ 范围内的最长回文子序列长度。

3. 状态转移方程

我们对区间 $[i, j]$ 边界位置上的字符 $s[i]$ 与 $s[j]$ 进行分类讨论：

1. 如果 $s[i] = s[j]$ ，则 $dp[i][j]$ 为区间 $[i + 1, j - 1]$ 范围内最长回文子序列长度 + 2，即 $dp[i][j] = dp[i + 1][j - 1] + 2$ 。
2. 如果 $s[i] \neq s[j]$ ，则 $dp[i][j]$ 取决于以下两种情况，取其最大的一种：
 - i. 加入 $s[i]$ 所能组成的最长回文子序列长度，即： $dp[i][j] = dp[i][j - 1]$ 。
 - ii. 加入 $s[j]$ 所能组成的最长回文子序列长度，即： $dp[i][j] = dp[i - 1][j]$ 。

则状态转移方程为：

$$dp[i][j] = \begin{cases} \max\{dp[i + 1][j - 1] + 2\} & s[i] = s[j] \\ \max\{dp[i][j - 1], dp[i - 1][j]\} & s[i] \neq s[j] \end{cases}$$

4. 初始条件

- 单个字符的最长回文序列是 1，即 $dp[i][i] = 1$ 。

5. 最终结果

由于 $dp[i][j]$ 依赖于 $dp[i + 1][j - 1]$ 、 $dp[i + 1][j]$ 、 $dp[i][j - 1]$ ，所以我们应该按照从下到上、从左到右的顺序进行遍历。

根据我们之前定义的状态， $dp[i][j]$ 表示为：字符串 s 在区间 $[i, j]$ 范围内的最长回文子序列长度。所以最终结果为 $dp[0][size - 1]$ 。

思路 1：代码

```
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        size = len(s)
        dp = [[0 for _ in range(size)] for _ in range(size)]
        for i in range(size):
            dp[i][i] = 1

        for i in range(size - 1, -1, -1):
            for j in range(i + 1, size):
                if s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1] + 2
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

        return dp[0][size - 1]
```

思路 1：复杂度分析

- 时间复杂度: $O(n^2)$, 其中 n 为字符串 s 的长度。
- 空间复杂度: $O(n^2)$ 。