

0123. 买卖股票的最佳时机 III

- 标签：数组、动态规划
- 难度：困难

题目链接

- [0123. 买卖股票的最佳时机 III - 力扣](#)

题目大意

给定一个数组 `prices` 代表一只股票，其中 `prices[i]` 代表这只股票第 `i` 天的价格。最多可完成两笔交易，且不同同时参与躲避交易（必须在再次购买前出售掉之前的股票）。

现在要求：计算所能获取的最大利润。

解题思路

动态规划求解。

最多可完成两笔交易意味着总共有三种情况：买卖一次，买卖两次，不买卖。

具体到每一天结束总共有 5 种状态：

0. 未进行买卖状态；
1. 第一次买入状态；
2. 第一次卖出状态；
3. 第二次买入状态；
4. 第二次卖出状态。

所以我们可以定义状态 `dp[i][j]`，表示为：第 `i` 天第 `j` 种情况 ($0 \leq j \leq 4$) 下，所获取的最大利润。

注意：这里第 `j` 种情况，并不一定是这一天一定要买入或卖出，而是这一天所处于的买入卖出状态。比如说前一天是第一次买入，第二天没有操作，则第二天就沿用前一天的第一次买入状态。

接下来确定状态转移公式：

- 第 0 种状态下显然利润为 0，可以直接赋值为昨天获取的最大利润，即 `dp[i][0] = dp[i - 1][0]`。
- 第 1 种状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天买入状态所得的最大利润：`dp[i][1] = dp[i - 1][1]`。
 - 第一次买入：`dp[i][1] = dp[i - 1][0] - prices[i]`。
- 第 2 种状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天卖出状态所得的最大利润：`dp[i][2] = dp[i - 1][2]`。
 - 第一次卖出：`dp[i][2] = dp[i - 1][1] + prices[i]`。
- 第 3 种状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天买入状态所得的最大利润：`dp[i][3] = dp[i - 1][3]`。
 - 第二次买入：`dp[i][3] = dp[i - 1][2] - prices[i]`。
- 第 4 种状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天卖出状态所得的最大利润：`dp[i][4] = dp[i - 1][4]`。
 - 第二次卖出：`dp[i][4] = dp[i - 1][3] + prices[i]`。

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[0]`。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(5)] for _ in range(size)]

        dp[0][1] = -prices[0]
        dp[0][3] = -prices[0]

        for i in range(1, size):
            dp[i][0] = dp[i - 1][0]
            dp[i][1] = max(dp[i - 1][1], dp[i - 1][0] - prices[i])
            dp[i][2] = max(dp[i - 1][2], dp[i - 1][1] + prices[i])
            dp[i][3] = max(dp[i - 1][3], dp[i - 1][2] - prices[i])
            dp[i][4] = max(dp[i - 1][4], dp[i - 1][3] + prices[i])
        return dp[size - 1][4]
```

0124. 二叉树中的最大路径和

- 标签：树、深度优先搜索、动态规划、二叉树
- 难度：困难

题目链接

- [0124. 二叉树中的最大路径和 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

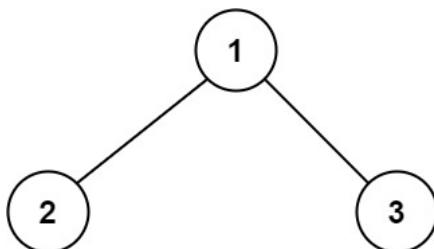
要求：返回其最大路径和。

说明：

- **路径：**被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
- **路径和：**路径中各节点值的总和。
- 树中节点数目范围是 $[1, 3 * 10^4]$ 。
- $-1000 \leq Node.val \leq 1000$ 。

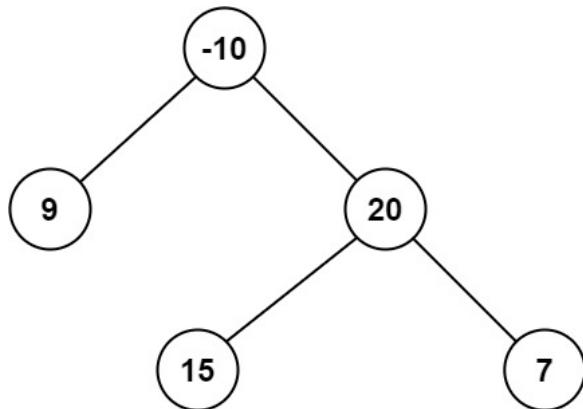
示例：

- **示例 1：**



输入: `root = [1,2,3]`
 输出: `6`
 解释: 最优路径是 `2 -> 1 -> 3`, 路径和为 `2 + 1 + 3 = 6`

- 示例 2:



输入: `root = [-10,9,20,null,null,15,7]`
 输出: `42`
 解释: 最优路径是 `15 -> 20 -> 7`, 路径和为 `15 + 20 + 7 = 42`

解题思路

思路 1：树形 DP + 深度优先搜索

根据最大路径和中对应路径是否穿过根节点，我们可以将二叉树分为两种：

1. 最大路径和中对应路径穿过根节点。
2. 最大路径和中对应路径不穿过根节点。

如果最大路径和中对应路径穿过根节点，则：该二叉树的最大路径和 = 左子树中最大贡献值 + 右子树中最大贡献值 + 当前节点值。

而如果最大路径和中对应路径不穿过根节点，则：该二叉树的最大路径和 = 所有子树中最大路径和。

即：该二叉树的最大路径和 = $\max(\text{左子树中最大贡献值} + \text{右子树中最大贡献值} + \text{当前节点值}, \text{所有子树中最大路径和})$ 。

对此我们可以使用深度优先搜索递归遍历二叉树，并在递归遍历的同时，维护一个最大路径和变量 `ans`。

然后定义函数 `def dfs(self, node):`：计算二叉树中以该节点为根节点，并且经过该节点的最大贡献值。

计算的结果可能的情况有 2 种：

1. 经过空节点的最大贡献值等于 0。
2. 经过非空节点的最大贡献值等于 **当前节点值 + 左右子节点提供的最大贡献值中较大的一个**。如果该贡献值为负数，可以考虑舍弃，即最大贡献值为 0。

在递归时，我们先计算左右子节点的最大贡献值，再更新维护当前最大路径和变量。最终 `ans` 即为答案。具体步骤如下：

1. 如果根节点 `root` 为空，则返回 0。
2. 递归计算左子树的最大贡献值为 `left_max`。
3. 递归计算右子树的最大贡献值为 `right_max`。
4. 更新维护最大路径和变量，即 `self.ans = max{self.ans, left_max + right_max + node.val}`。
5. 返回以当前节点为根节点，并且经过该节点的最大贡献值。即返回 **当前节点值 + 左右子节点提供的最大贡献值中较大的一个**。
6. 最终 `self.ans` 即为答案。

思路 1：代码

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

    def dfs(self, node):
        if not node:
            return 0
        left_max = max(self.dfs(node.left), 0)      # 左子树提供的最大贡献值
        right_max = max(self.dfs(node.right), 0)      # 右子树提供的最大贡献值

        cur_max = left_max + right_max + node.val    # 包含当前节点和左右子树的最大路径和
        self.ans = max(self.ans, cur_max)             # 更新所有路径中的最大路径和

        return max(left_max, right_max) + node.val   # 返回包含当前节点的子树的最大贡献值

    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        self.dfs(root)
        return self.ans
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是二叉树的节点数目。
- 空间复杂度: $O(n)$ 。递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 n , 所以空间复杂度为 $O(n)$ 。

0125. 验证回文串

- 标签: 双指针、字符串
- 难度: 简单

题目链接

- [0125. 验证回文串 - 力扣](#)

题目大意

描述: 给定一个字符串 s 。

要求: 判断是否为回文串 (只考虑字符串中的字母和数字字符, 并且忽略字母的大小写)。

说明:

- 回文串: 正着读和反着读都一样的字符串。
- $1 \leq s.length \leq 2 * 10^5$ 。
- s 仅由可打印的 ASCII 字符组成。

示例:

- 示例 1:

输入: "A man, a plan, a canal: Panama"

输出: true

解释: "amanaplanacanalpanama" 是回文串。

- 示例 2:

输入: "race a car"

输出: false

解释: "raceacar" 不是回文串。

解题思路

思路 1：对撞指针

- 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。
- 判断两个指针对应字符是否是字母或数字。通过 `left` 右移、`right` 左移的方式过滤掉字母和数字以外的字符。
- 然后判断 `s[left]` 是否和 `s[right]` 相等 (注意大小写)。
 - 如果相等, 则将 `left` 右移、`right` 左移, 继续进行下一次过滤和判断。
 - 如果不相等, 则说明不是回文串, 直接返回 `False`。
- 如果遇到 `left == right`, 跳出循环, 则说明该字符串是回文串, 返回 `True`。

思路 1：代码

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        left = 0
        right = len(s) - 1

        while left < right:
            if not s[left].isalnum():
                left += 1
                continue
            if not s[right].isalnum():
                right -= 1
                continue

            if s[left].lower() == s[right].lower():
                left += 1
                right -= 1
            else:
                return False
        return True
```

思路 1：复杂度分析

- 时间复杂度: $O(len(s))$ 。
- 空间复杂度: $O(len(s))$ 。

0127. 单词接龙

- 标签: 广度优先搜索、哈希表、字符串
- 难度: 困难

题目链接

- [0127. 单词接龙 - 力扣](#)

题目大意

给定两个单词 `beginWord` 和 `endWord`，以及一个字典 `wordList`。找到从 `beginWord` 到 `endWord` 的最短转换序列中的单词数目。如果不存在这样的转换序列，则返回 0。

转换需要遵守的规则如下：

- 每次转换只能改变一个字母。
- 转换过程中的中间单词必须为字典中的单词。

解题思路

广度优先搜索。使用队列存储将要遍历的单词和单词数目。

从 `beginWord` 开始变换，把单词的每个字母都用 `a ~ z` 变换一次，变换后的单词是否是 `endWord`，如果是则直接返回。

否则查找变换后的词是否在 `wordList` 中。如果在 `wordList` 中找到就加入队列，找不到就输出 `0`。然后按照广度优先搜索的算法急需要遍历队列中的节点，直到所有单词都出队时结束。

代码

```
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        if not wordList or endWord not in wordList:
            return 0
        word_set = set(wordList)
        if beginWord in word_set:
            word_set.remove(beginWord)

        queue = collections.deque()
        queue.append((beginWord, 1))
        while queue:
            word, level = queue.popleft()
            if word == endWord:
                return level

            for i in range(len(word)):
                for j in range(26):
                    new_word = word[:i] + chr(ord('a') + j) + word[i + 1:]
                    if new_word in word_set:
                        word_set.remove(new_word)
                        queue.append((new_word, level + 1))

        return 0
```

0128. 最长连续序列

- 标签：并查集、数组、哈希表
- 难度：中等

题目链接

- [0128. 最长连续序列 - 力扣](#)

题目大意

描述：给定一个未排序的整数数组 `nums`。

要求：找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。并且要用时间复杂度为 $O(n)$ 的算法解决此问题。

说明：

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

示例：

- 示例 1：

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

- 示例 2：

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

解题思路

暴力做法有两种思路。

- 第 1 种思路是先排序再依次判断，这种做法时间复杂度最少是 $O(n \log_2 n)$ 。
- 第 2 种思路是枚举数组中的每个数 `num`，考虑以其为起点，不断尝试匹配 `num + 1`、`num + 2`、`...` 是否存在，最长匹配次数为 `len(nums)`。这样下来时间复杂度为 $O(n^2)$ 。

我们可以使用哈希表优化这个过程。

思路 1：哈希表

1. 先将数组存储到集合中进行去重，然后使用 `curr_streak` 维护当前连续序列长度，使用 `ans` 维护最长连续序列长度。
2. 遍历集合中的元素，对每个元素进行判断，如果该元素不是序列的开始（即 `num - 1` 在集合中），则跳过。
3. 如果 `num - 1` 不在集合中，说明 `num` 是序列的开始，判断 `num + 1`、`num + 2`、`...` 是否在哈希表中，并不断更新当前连续序列长度 `curr_streak`。并在遍历结束之后更新最长序列的长度。
4. 最后输出最长序列长度。

思路 1：代码

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        ans = 0
        nums_set = set(nums)
        for num in nums_set:
            if num - 1 not in nums_set:
                curr_num = num
                curr_streak = 1

                while curr_num + 1 in nums_set:
                    curr_num += 1
                    curr_streak += 1
                ans = max(ans, curr_streak)

        return ans
```

思路 1：复杂度分析

- **时间复杂度：** $O(n)$ 。将数组存储到集合中进行去重的操作的时间复杂度是 $O(n)$ 。查询每个数是否在集合中的时间复杂度是 $O(1)$ ，并且跳过了所有不是起点的元素。更新当前连续序列长度 `curr_streak` 的时间复杂度是 $O(n)$ ，所以最终的时间复杂度是 $O(n)$ 。

- 空间复杂度: $O(n)$ 。

参考资料

- 【题解】128. 最长连续序列 - 力扣（Leetcode）

0129. 求根节点到叶节点数字之和

- 标签：树、深度优先搜索、二叉树
- 难度：中等

题目链接

- 0129. 求根节点到叶节点数字之和 - 力扣

题目大意

描述：给定一个二叉树的根节点 `root`，树中每个节点都存放有一个 `0` 到 `9` 之间的数字。每条从根节点到叶节点的路径都代表一个数字。例如，从根节点到叶节点的路径是 `1 -> 2 -> 3`，表示数字 `123`。

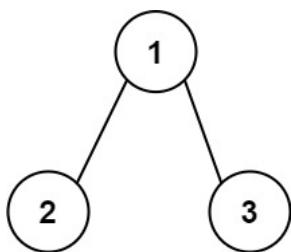
要求：计算从根节点到叶节点生成的所有数字的和。

说明：

- **叶节点：**指没有子节点的节点。
- 树中节点的数目在范围 $[1, 1000]$ 内。
- $0 \leq Node.val \leq 9$ 。
- 树的深度不超过 10。

示例：

- **示例 1：**



输入: `root = [1,2,3]`

输出: `25`

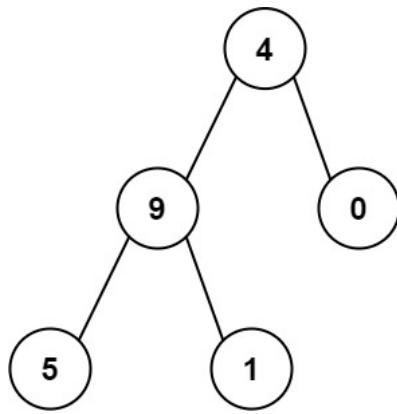
解释:

从根到叶子节点路径 `1->2` 代表数字 `12`

从根到叶子节点路径 `1->3` 代表数字 `13`

因此, 数字总和 = `12 + 13 = 25`

- **示例 2：**



输入: `root = [4, 9, 0, 5, 1]`

输出: `1026`

解释:

从根到叶子节点路径 `4->9->5` 代表数字 `495`

从根到叶子节点路径 `4->9->1` 代表数字 `491`

从根到叶子节点路径 `4->0` 代表数字 `40`

因此, 数字总和 = `495 + 491 + 40 = 1026`

解题思路

思路 1：深度优先搜索

1. 记录下路径上所有节点构成的数字, 使用变量 `pre_total` 保存下当前路径上构成的数字。
2. 如果遇到叶节点, 则直接返回当前数字。
3. 如果没有遇到叶节点, 则递归遍历左右子树, 并累加对应结果。

思路 1：代码

```

class Solution:
    def dfs(self, root, pre_total):
        if not root:
            return 0
        total = pre_total * 10 + root.val
        if not root.left and not root.right:
            return total
        return self.dfs(root.left, total) + self.dfs(root.right, total)

    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        return self.dfs(root, 0)
  
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是二叉树的节点数目。
- 空间复杂度: $O(n)$ 。递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 n , 所以空间复杂度为 $O(n)$ 。

0130. 被围绕的区域

- 标签: 深度优先搜索、广度优先搜索、并查集、数组、矩阵
- 难度: 中等

题目链接

- [0130. 被围绕的区域 - 力扣](#)

题目大意

描述：给定一个 $m * n$ 的矩阵 `board`，由若干字符 `X` 和 `O` 构成。

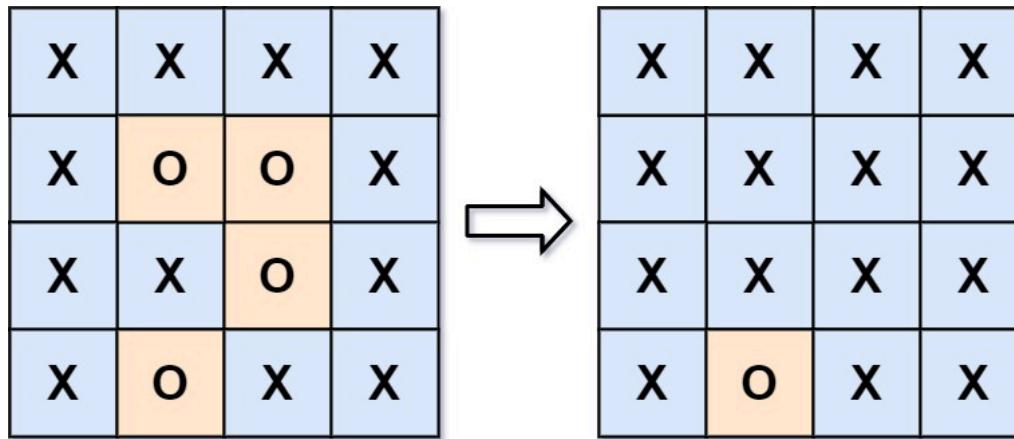
要求：找到所有被 `X` 围绕的区域，并将这些区域里所有的 `O` 用 `X` 填充。

说明：

- $m == board.length$ 。
- $n == board[i].length$ 。
- $1 <= m, n <= 200$ 。
- $board[i][j]$ 为 `'X'` 或 `'O'`。

示例：

- 示例 1：



输入：`board = [["X", "X", "X", "X"], ["X", "O", "O", "X"], ["X", "X", "O", "X"], ["X", "O", "X", "X"]]`

输出：`[["X", "X", "X", "X"], ["X", "X", "X", "X"], ["X", "X", "X", "X"], ["X", "O", "X", "X"]]`

解释：被围绕的区间不会存在于边界上，换句话说，任何边界上的 '`O`' 都不会被填充为 '`X`'。任何不在边界上，或不与边界相连的 '`O`' 最终都会被填充为 '`X`'。如果

- 示例 2：

```
输入: board = [[ "X" ]]
输出: [[ "X" ]]
```

解题思路

思路 1：深度优先搜索

根据题意，任何边界上的 `O` 都不会被填充为 `X`。而被填充 `X` 的 `O` 一定在内部不在边界上。

所以我们可以用深度优先搜索先搜索边界上的 `O` 以及与边界相连的 `O`，将其先标记为 `#`。

最后遍历一遍 `board`，将所有 `#` 变换为 `O`，将所有 `O` 变换为 `X`。

思路 1：代码

```

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

        if not board:
            return
        rows, cols = len(board), len(board[0])

        def dfs(x, y):
            if not 0 <= x < rows or not 0 <= y < cols or board[x][y] != '0':
                return
            board[x][y] = '#'
            dfs(x + 1, y)
            dfs(x - 1, y)
            dfs(x, y + 1)
            dfs(x, y - 1)

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

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

        for i in range(rows):
            for j in range(cols):
                if board[i][j] == '#':
                    board[i][j] = '0'
                elif board[i][j] == '0':
                    board[i][j] = 'X'

```

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: $O(n \times m)$. # 0131. 分割回文串
- 标签: 字符串、动态规划、回溯
- 难度: 中等

题目链接

- [0131. 分割回文串 - 力扣](#)

题目大意

给定一个字符串 s ，将 s 分割成一些子串，保证每个子串都是「回文串」。返回 s 所有可能的分割方案。

解题思路

回溯算法，建立两个数组 res 、 $path$ 。 res 用于存放所有满足题意的组合， $path$ 用于存放当前满足题意的一个组合。

在回溯的时候判断当前子串是否为回文串，如果不是则跳过，如果是则继续向下一层遍历。

定义判断是否为回文串的方法和回溯方法，从 $start_index = 0$ 的位置开始回溯。

- 如果 $start_index \geq len(s)$ ，则将 $path$ 中的元素加入到 res 数组中。
- 然后对 $[start_index, len(s) - 1]$ 范围内的子串进行遍历取值。

- 如果字符串 `s` 在范围 `[start_index, i]` 所代表的子串是回文串，则将其加入 `path` 数组。
- 递归遍历 `[i + 1, len(s) - 1]` 范围上的子串。
- 然后将遍历的范围 `[start_index, i]` 所代表的子串进行回退。
- 最终返回 `res` 数组。

代码

```

class Solution:
    res = []
    path = []
    def backtrack(self, s: str, start_index: int):
        if start_index >= len(s):
            self.res.append(self.path[:])
            return
        for i in range(start_index, len(s)):
            if self.ispalindrome(s, start_index, i):
                self.path.append(s[start_index: i+1])
                self.backtrack(s, i + 1)
                self.path.pop()

    def ispalindrome(self, s: str, start: int, end: int):
        i, j = start, end
        while i < j:
            if s[i] != s[j]:
                return False
            i += 1
            j -= 1
        return True

    def partition(self, s: str) -> List[List[str]]:
        self.res.clear()
        self.path.clear()
        self.backtrack(s, 0)
        return self.res

```

0133. 克隆图

- 标签：深度优先搜索、广度优先搜索、图、哈希表
- 难度：中等

题目链接

- [0133. 克隆图 - 力扣](#)

题目大意

描述：以每个节点的邻接列表形式（二维列表）给定一个无向连通图，其中 `adjList[i]` 表示值为 $i + 1$ 的节点的邻接列表，`adjList[i][j]` 表示值为 $i + 1$ 的节点与值为 `adjList[i][j]` 的节点有一条边。

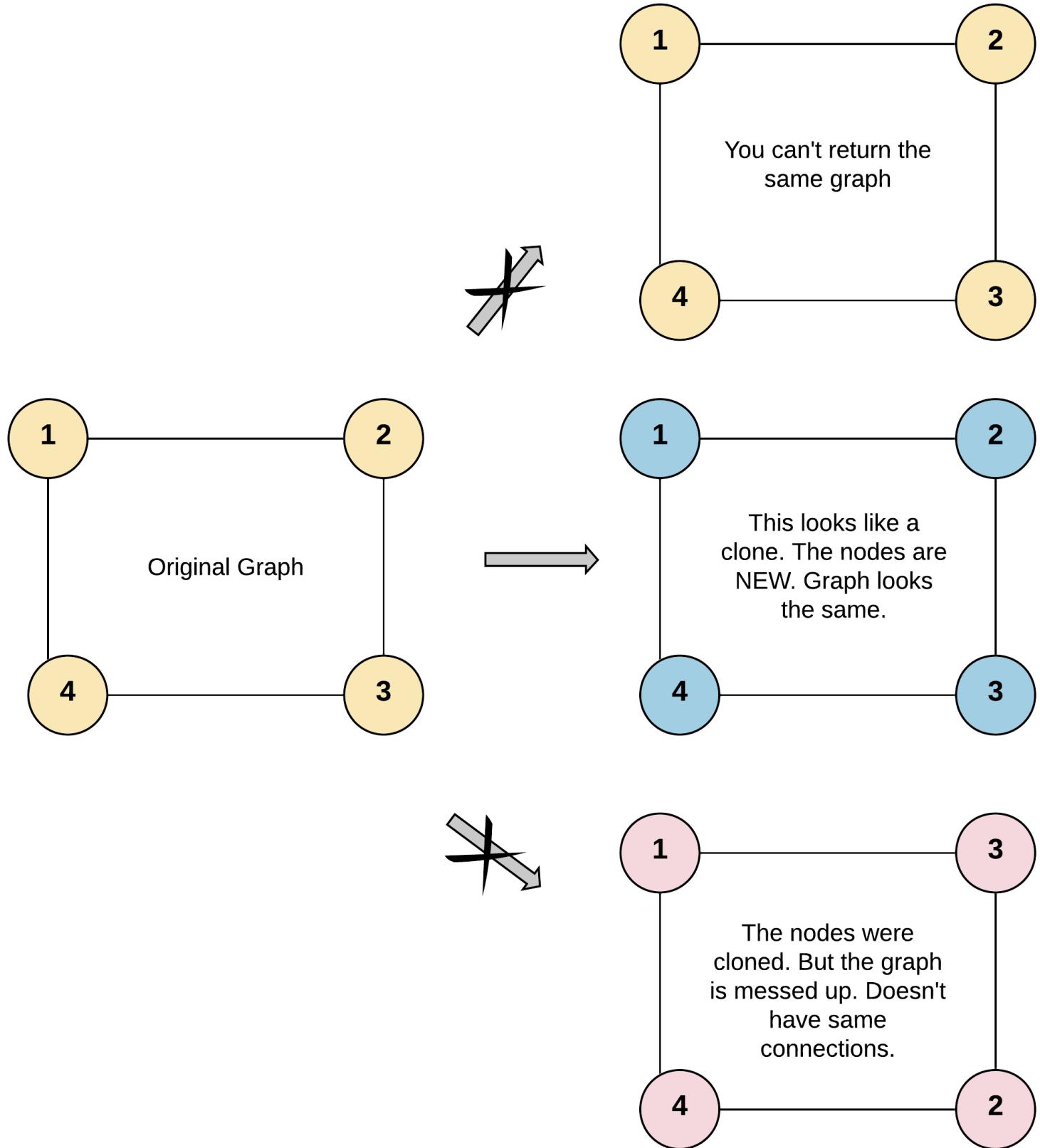
要求：返回该图的深拷贝。

说明：

- 节点数不超过 100。
- 每个节点值 `Node.val` 都是唯一的， $1 \leq Node.val \leq 100$ 。
- 无向图是一个简单图，这意味着图中没有重复的边，也没有自环。
- 由于图是无向的，如果节点 p 是节点 q 的邻居，那么节点 q 也必须是节点 p 的邻居。
- 图是连通图，你可以从给定节点访问到所有节点。

示例:

- 示例 1:



输入: adjList = [[2,4],[1,3],[2,4],[1,3]]

输出: [[2,4],[1,3],[2,4],[1,3]]

解释:

图中有 4 个节点。

节点 1 的值是 1，它有两个邻居: 节点 2 和 4。

节点 2 的值是 2，它有两个邻居: 节点 1 和 3。

节点 3 的值是 3，它有两个邻居: 节点 2 和 4。

节点 4 的值是 4，它有两个邻居: 节点 1 和 3。

- 示例 2:



输入: adjList = [[2],[1]]

输出: [[2],[1]]

解题思路

所谓深拷贝，就是构建一张与原图结构、值均一样的图，但是所用的节点不再是原图节点的引用，即每个节点都要新建。

可以用深度优先搜索或者广度优先搜索来做。

思路 1：深度优先搜索

- 使用哈希表 *visitedDict* 来存储原图中被访问过的节点和克隆图中对应节点，键值对为「原图被访问过的节点: 克隆图中对应节点」。
- 从给定节点开始，以深度优先搜索的方式遍历原图。
 - 如果当前节点被访问过，则返回克隆图中对应节点。
 - 如果当前节点没有被访问过，则创建一个新的节点，并保存在哈希表中。
 - 遍历当前节点的邻接节点列表，递归调用当前节点的邻接节点，并将其放入克隆图中对应节点。
- 递归结束，返回克隆节点。

思路 1：代码

```

class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        if not node:
            return node
        visited = dict()

        def dfs(node: 'Node') -> 'Node':
            if node in visited:
                return visited[node]

            clone_node = Node(node.val, [])
            visited[node] = clone_node
            for neighbor in node.neighbors:
                clone_node.neighbors.append(dfs(neighbor))
            return clone_node

        return dfs(node)
  
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 为图中节点数量。
- 空间复杂度: $O(n)$ 。

思路 2：广度优先搜索

1. 使用哈希表 `visited` 来存储原图中被访问过的节点和克隆图中对应节点，键值对为「原图被访问过的节点：克隆图中对应节点」。使用队列 `queue` 存放节点。
2. 根据起始节点 `node`，创建一个新的节点，并将其添加到哈希表 `visited` 中，即 `visited[node] = Node(node.val, [])`。然后将起始节点放入队列中，即 `queue.append(node)`。
3. 从队列中取出第一个节点 `node_u`。访问节点 `node_u`。
4. 遍历节点 `node_u` 的所有未访问邻接节点 `node_v`（节点 `node_v` 不在 `visited` 中）。
5. 根据节点 `node_v` 创建一个新的节点，并将其添加到哈希表 `visited` 中，即 `visited[node_v] = Node(node_v.val, [])`。
6. 然后将节点 `node_v` 放入队列 `queue` 中，即 `queue.append(node_v)`。
7. 重复步骤 3 ~ 6，直到队列 `queue` 为空。
8. 广度优先搜索结束，返回起始节点的克隆节点（即 `visited[node]`）。

思路 2：代码

```
class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        if not node:
            return node

        visited = dict()
        queue = collections.deque()

        visited[node] = Node(node.val, [])
        queue.append(node)

        while queue:
            node_u = queue.popleft()
            for node_v in node_u.neighbors:
                if node_v not in visited:
                    visited[node_v] = Node(node_v.val, [])
                    queue.append(node_v)
                    visited[node_u].neighbors.append(visited[node_v])

        return visited[node]
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 为图中节点数量。
- 空间复杂度： $O(n)$ 。[# 0134. 加油站](#)
- 标签：贪心、数组
- 难度：中等

题目链接

- [0134. 加油站 - 力扣](#)

题目大意

一条环路上有 N 个加油站，第 i 个加油站有 $gas[i]$ 升汽油。

现在有一辆油箱无限容量的汽车，从第 i 个加油站开往第 $i + 1$ 个加油站需要消耗汽油 $cost[i]$ 升。如果汽车上携带的有两不够 $cost[i]$ ，则无法从第 i 个加油站开往第 $i + 1$ 个加油站。

现在从其中一个加油站开始出发，且出发时油箱为空。如果能绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

解题思路

1. 暴力求解

分别考虑从第 0 个点、第 1 个点、...、第 i 个点出发，能否回到第 0 个点、第 1 个点、...、第 i 个点。

2. 贪心算法

- 如果加油站提供的油总和大于等于消耗的汽油量，则必定可以绕环路行驶一周
- 假设先不考虑油量为负的情况，我们从「第 0 个加油站」出发，环行一周。记录下汽油量 $gas[i]$ 和 $cost[i]$ 差值总和 sum_diff ，同时记录下油箱剩余油量的最小值 min_sum 。
- 如果差值总和 $sum_diff < 0$ ，则无论如何都不能环行一周。油不够啊，亲！！
- 如果 $min_sum \geq 0$ ，则行驶过程中油箱始终有油，则可以从 0 个加油站出发环行一周。
- 如果 $min_sum < 0$ ，则说明行驶过程中油箱油不够了，那么考虑更换开始的起点。
 - 从右至左遍历，计算汽油量 $gas[i]$ 和 $cost[i]$ 差值，看哪个加油站能将 min_sum 填平。如果最终达到 $min_sum \geq 0$ ，则说明从该点开始出发，油箱中的油始终不为空，则返回该点下标。
 - 如果找不到最返回 -1。

代码

```
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        sum_diff, min_sum = 0, float('inf')
        for i in range(len(gas)):
            sum_diff += gas[i] - cost[i]
            min_sum = min(min_sum, sum_diff)

        if sum_diff < 0:
            return -1

        if min_sum >= 0:
            return 0

        for i in range(len(gas)-1, -1, -1):
            min_sum += gas[i] - cost[i]
            if min_sum >= 0:
                return i
        return -1
```

参考链接

- [贪心算法/前缀和 - 加油站 - 力扣 \(LeetCode\)](#)

0135. 分发糖果

- 标签：贪心、数组
- 难度：困难

题目链接

- [0135. 分发糖果 - 力扣](#)

题目大意

描述： n 个孩子站成一排。老师会根据每个孩子的表现，给每个孩子进行评分。然后根据下面的规则给孩子们分发糖果：

- 每个孩子至少得 1 个糖果。
- 评分更高的孩子必须比他两侧相邻位置上的孩子分得更多的糖果。

现在给定 n 个孩子的表现分数数组 $ratings$ ，其中 $ratings[i]$ 表示第 i 个孩子的评分。

要求：返回最少需要准备的糖果数目。

说明：

- $n == ratings.length$ 。
- $1 \leq n \leq 2 \times 10^4$ 。
- $0 \leq ratings[i] \leq 2 * 10^4$ 。

示例：

- 示例 1：

```
输入: ratings = [1,0,2]
输出: 5
解释: 你可以分别给第一个、第二个、第三个孩子分发 2、1、2 颗糖果。
```

- 示例 2：

```
输入: ratings = [1,2,2]
输出: 4
解释: 你可以分别给第一个、第二个、第三个孩子分发 1、2、1 颗糖果。
      第三个孩子只得到 1 颗糖果，这满足题面中的两个条件。
```

解题思路

思路 1：贪心算法

先来看分发糖果的规则。

「每个孩子至少得 1 个糖果」：说明糖果数目至少为 N 个。

「评分更高的孩子必须比他两侧相邻位置上的孩子分得更多的糖果」：可以看做为以下两种条件：

- 当 $ratings[i - 1] < ratings[i]$ 时，第 i 个孩子的糖果数量比第 $i - 1$ 个孩子的糖果数量多；
- 当 $ratings[i] > ratings[i + 1]$ 时，第 i 个孩子的糖果数量比第 $i + 1$ 个孩子的糖果数量多。

根据以上信息，我们可以设定一个长度为 N 的数组 sweets 来表示每个孩子分得的最少糖果数，初始每个孩子分得糖果数都为 1。

然后遍历两遍数组，第一遍遍历满足当 $ratings[i - 1] < ratings[i]$ 时，第 i 个孩子的糖果数量比第 $i - 1$ 个孩子的糖果数量多 1 个。第二遍遍历满足当 $ratings[i] > ratings[i + 1]$ 时，第 i 个孩子的糖果数量取「第 $i + 1$ 个孩子的糖果数量多 1 个」和「第 $i + 1$ 个孩子目前拥有的糖果数量」中的最大值。

然后再遍历求所有孩子的糖果数量和即为答案。

思路 1：代码

```
class Solution:
    def candy(self, ratings: List[int]) -> int:
        size = len(ratings)
        sweets = [1 for _ in range(size)]

        for i in range(1, size):
            if ratings[i] > ratings[i - 1]:
                sweets[i] = sweets[i - 1] + 1

        for i in range(size - 2, -1, -1):
            if ratings[i] > ratings[i + 1]:
                sweets[i] = max(sweets[i], sweets[i + 1] + 1)

        res = sum(sweets)
        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是数组 `ratings` 的长度。
- 空间复杂度： $O(n)$ 。

0136. 只出现一次的数字

- 标签：位运算、数组
- 难度：简单

题目链接

- [0136. 只出现一次的数字 - 力扣](#)

题目大意

描述：给定一个非空整数数组 `nums`，`nums` 中除了某个元素只出现一次以外，其余每个元素均出现两次。

要求：找出那个只出现了一次的元素。

说明：

- 要求不能使用额外的存储空间。

示例：

- 示例 1：

```
输入: [2,2,1]
输出: 1
```

- 示例 2：

```
输入: [4,1,2,1,2]
输出: 4
```

解题思路

思路 1：位运算

如果没有时间复杂度和空间复杂度的限制，可以使用哈希表 / 集合来存储每个元素出现的次数，如果哈希表中没有该数字，则将该数字加入集合，如果集合中有了该数字，则从集合中删除该数字，最终成对的数字都被删除了，只剩下单次出现的元素。

但是题目要求不使用额外的存储空间，就需要用到位运算中的异或运算。

异或运算 \oplus 的三个性质：

- 任何数和 0 做异或运算，结果仍然是原来的数，即 $a \oplus 0 = a$ 。
- 数和其自身做异或运算，结果是 0，即 $a \oplus a = 0$ 。
- 异或运算满足交换率和结合律： $a \oplus b \oplus a = b \oplus a \oplus a = b \oplus (a \oplus a) = b \oplus 0 = b$ 。

根据异或运算的性质，对 n 个数不断进行异或操作，最终可得到单次出现的元素。

思路 1：代码

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return nums[0]
        ans = 0
        for i in range(len(nums)):
            ans ^= nums[i]
        return ans
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。# 0137. 只出现一次的数字 II
- 标签: 位运算、数组
- 难度: 中等

题目链接

- [0137. 只出现一次的数字 II - 力扣](#)

题目大意

描述: 给定一个整数数组 $nums$, 除了某个元素仅出现一次外, 其余每个元素恰好出现三次。

要求: 找到并返回那个只出现了一次的元素。

说明:

- $1 \leq nums.length \leq 3 * 10^4$ 。
- $-2^{31} \leq nums[i] \leq 2^{31} - 1$ 。
- $nums$ 中, 除某个元素仅出现一次外, 其余每个元素都恰出现三次。

示例:

- 示例 1:

```
输入: nums = [2,2,3,2]
输出: 3
```

- 示例 2:

```
输入: nums = [0,1,0,1,0,1,99]
输出: 99
```

解题思路

思路 1：哈希表

- 利用哈希表统计出每个元素的出现次数。
- 再遍历一次哈希表, 找到仅出现一次的元素。

思路 1：代码

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        nums_dict = dict()
        for num in nums:
            if num in nums_dict:
                nums_dict[num] += 1
            else:
                nums_dict[num] = 1
        for key in nums_dict:
            value = nums_dict[key]
            if value == 1:
                return key
        return 0
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是数组 $nums$ 的元素个数。
- 空间复杂度: $O(n)$ 。

思路 2：位运算

将出现三次的元素换成二进制形式放在一起，其二进制对应位置上，出现 1 的个数一定是 3 的倍数（包括 0）。此时，如果在放进来只出现一次的元素，则某些二进制位置上出现 1 的个数就不是 3 的倍数了。

将这些二进制位置上出现 1 的个数不是 3 的倍数位置值置为 1，是 3 的倍数则置为 0。这样对应下来的二进制就是答案所求。

注意：因为 Python 的整数没有位数限制，所以不能通过最高位确定正负。所以 Python 中负整数的补码会被当做正整数。所以在遍历到最后 31 位时进行 $ans = (1 << 31)$ 操作，目的是将负数的补码转换为「负号 + 原码」的形式。这样就可以正常识别二进制下的负数。参考：[Two's Complement Binary in Python? - Stack Overflow](#)

思路 2：代码

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        ans = 0
        for i in range(32):
            count = 0
            for j in range(len(nums)):
                count += (nums[j] >> i) & 1
            if count % 3 != 0:
                if i == 31:
                    ans -= (1 << 31)
                else:
                    ans = ans | 1 << i
        return ans
```

思路 2：复杂度分析

- 时间复杂度: $O(n \log m)$, 其中 n 是数组 $nums$ 的长度, m 是数据范围, 本题中 $m = 32$ 。
- 空间复杂度: $O(1)$ 。

0138. 复制带随机指针的链表

- 标签：哈希表、链表
- 难度：中等

题目链接

- 0138. 复制带随机指针的链表 - 力扣

题目大意

描述：给定一个链表的头节点 `head`，链表中每个节点除了 `next` 指针之外，还包含一个随机指针 `random`，该指针可以指向链表中的任何节点或者空节点。

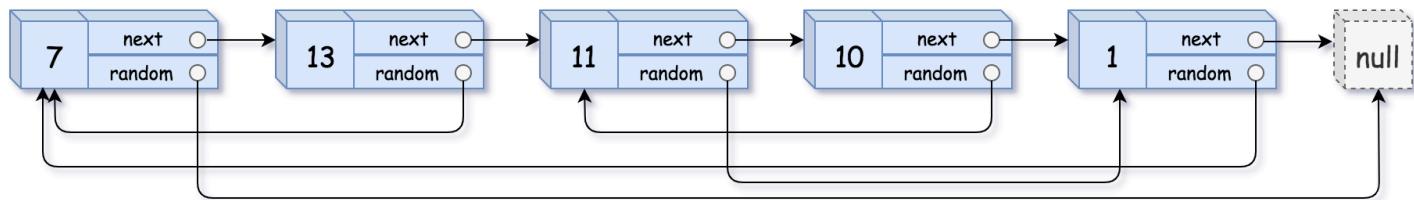
要求：将该链表进行深拷贝。返回复制链表的头节点。

说明：

- $0 \leq n \leq 1000$ 。
- $-10^4 \leq \text{Node.val} \leq 10^4$ 。
- `Node.random` 为 `null` 或指向链表中的节点。

示例：

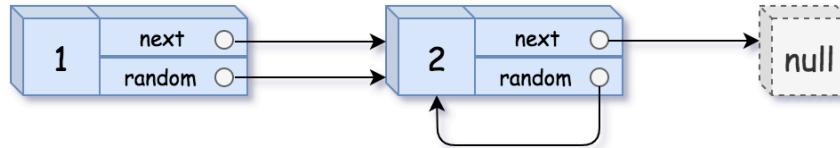
- 示例 1：



输入: `head = [[7,null],[13,0],[11,4],[10,2],[1,0]]`

输出: `[[7,null],[13,0],[11,4],[10,2],[1,0]]`

- 示例 2：



输入: `head = [[1,1],[2,1]]`

输出: `[[1,1],[2,1]]`

解题思路

思路 1：迭代

1. 遍历链表，利用哈希表，以 `旧节点: 新节点` 为映射关系，将节点关系存储下来。
2. 再次遍历链表，将新链表的 `next` 和 `random` 指针设置好。

思路 1：代码

```

class Solution:
    def copyRandomList(self, head: 'Node') -> 'Node':
        if not head:
            return None
        node_dict = dict()
        curr = head
        while curr:
            new_node = Node(curr.val, None, None)
            node_dict[curr] = new_node
            curr = curr.next
        curr = head
        while curr:
            if curr.next:
                node_dict[curr].next = node_dict[curr.next]
            if curr.random:
                node_dict[curr].random = node_dict[curr.random]
            curr = curr.next
        return node_dict[head]

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

0139. 单词拆分

- 标签：字典树、记忆化搜索、数组、哈希表、字符串、动态规划
- 难度：中等

题目链接

- [0139. 单词拆分 - 力扣](#)

题目大意

描述：给定一个非空字符串 s 和一个包含非空单词的列表 $wordDict$ 作为字典。

要求：判断是否可以利用字典中出现的单词拼接出 s 。

说明：

- 不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。
- $1 \leq s.length \leq 300$ 。
- $1 \leq wordDict.length \leq 1000$ 。
- $1 \leq wordDict[i].length \leq 20$ 。
- s 和 $wordDict[i]$ 仅有小写英文字母组成。
- $wordDict$ 中的所有字符串互不相同。

示例：

- 示例 1：

```

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

```

- 示例 2：

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

解题思路

思路 1：动态规划

1. 划分阶段

按照单词结尾位置进行阶段划分。

2. 定义状态

s 能否拆分为单词表的单词，可以分解为：

- 前 i 个字符构成的字符串，能否分解为单词。
- 剩余字符串，能否分解为单词。

定义状态 $dp[i]$ 表示：长度为 i 的字符串 $s[0 : i]$ 能否拆分成单词，如果为 $True$ 则表示可以拆分，如果为 $False$ 则表示不能拆分。

3. 状态转移方程

- 如果 $s[0 : j]$ 可以拆分为单词（即 $dp[j] == True$ ），并且字符串 $s[j : i]$ 出现在字典中，则 $dp[i] = True$ 。
- 如果 $s[0 : j]$ 不可以拆分为单词（即 $dp[j] == False$ ），或者字符串 $s[j : i]$ 没有出现在字典中，则 $dp[i] = False$ 。

4. 初始条件

- 长度为 0 的字符串 $s[0 : i]$ 可以拆分为单词，即 $dp[0] = True$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示：长度为 i 的字符串 $s[0 : i]$ 能否拆分成单词。则最终结果为 $dp[size]$ ， $size$ 为字符串长度。

思路 1：代码

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        size = len(s)
        dp = [False for _ in range(size + 1)]
        dp[0] = True
        for i in range(size + 1):
            for j in range(i):
                if dp[j] and s[j: i] in wordDict:
                    dp[i] = True
        return dp[size]
```

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： $O(n)$ 。

0140. 单词拆分 II

- 标签：字典树、记忆化搜索、数组、哈希表、字符串、动态规划、回溯
- 难度：困难

题目链接

- 0140. 单词拆分 II - 力扣

题目大意

给定一个非空字符串 `s` 和一个包含非空单词列表的字典 `wordDict`。

要求：在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。

说明：

- 分隔时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

解题思路

回溯 + 记忆化搜索。

对于字符串 `s`，如果某个位置左侧部分是单词列表中的单词，则拆分出该单词，然后对 `s` 右侧剩余部分进行递归拆分。如果可以将整个字符串 `s` 拆分成单词列表中的单词，则得到一个句子。

使用 `memo` 数组进行记忆化存储，这样可以减少重复计算。

代码

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> List[str]:
        size = len(s)
        memo = [None for _ in range(size + 1)]

        def dfs(start):
            if start > size - 1:
                return []
            if memo[start]:
                return memo[start]
            res = []
            for i in range(start, size):
                word = s[start: i + 1]
                if word in wordDict:
                    rest_res = dfs(i + 1)
                    for item in rest_res:
                        res.append([word] + item)
            memo[start] = res
            return res
        res = dfs(0)
        ans = []
        for item in res:
            ans.append(" ".join(item))
        return ans
```

0141. 环形链表

- 标签：哈希表、链表、双指针
- 难度：简单

题目链接

- [0141. 环形链表 - 力扣](#)

题目大意

描述：给定一个链表的头节点 `head`。

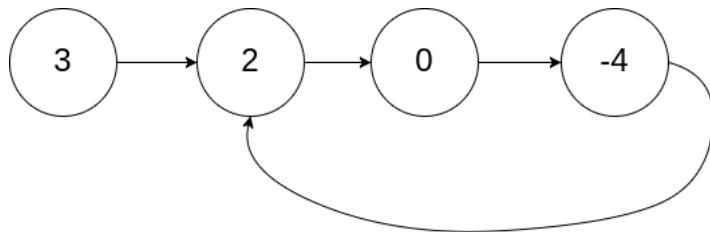
要求：判断链表中是否有环。如果有环则返回 `True`，否则返回 `False`。

说明：

- 链表中节点的数目范围是 $[0, 10^4]$ 。
- $-10^5 \leq Node.val \leq 10^5$ 。
- `pos` 为 `-1` 或者链表中的一个有效索引。

示例：

- **示例 1：**

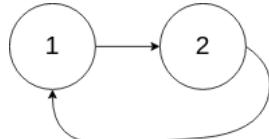


输入：`head = [3, 2, 0, -4], pos = 1`

输出：`True`

解释：链表中有一个环，其尾部连接到第二个节点。

- **示例 2：**



输入：`head = [1, 2], pos = 0`

输出：`True`

解释：链表中有一个环，其尾部连接到第一个节点。

解题思路

思路 1：哈希表

最简单的思路是遍历所有节点，每次遍历节点之前，使用哈希表判断该节点是否被访问过。如果访问过就说明存在环，如果没有访问过则将该节点添加到哈希表中，继续遍历判断。

思路 1：代码

```

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        nodeset = set()

        while head:
            if head in nodeset:
                return True
            nodeset.add(head)
            head = head.next
        return False
  
```

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：快慢指针（Floyd 判圈算法）

这种方法类似于在操场跑道跑步。两个人从同一位置同时出发，如果跑道有环（环形跑道），那么快的一方总能追上慢的一方。

基于上边的想法，Floyd 用两个指针，一个慢指针（龟）每次前进一步，快指针（兔）指针每次前进两步（两步或多步效果是等价的）。如果两个指针在链表头节点以外的某一节点相遇（即相等）了，那么说明链表有环，否则，如果（快指针）到达了某个没有后继指针的节点时，那么说明没环。

思路 2：代码

```
class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        if head == None or head.next == None:
            return False

        slow = head
        fast = head.next

        while slow != fast:
            if fast == None or fast.next == None:
                return False
            slow = slow.next
            fast = fast.next.next

        return True
```

思路 2：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。# 0142. 环形链表 II
- 标签: 哈希表、链表、双指针
- 难度: 中等

题目链接

- [0142. 环形链表 II - 力扣](#)

题目大意

描述：给定一个链表的头节点 `head`。

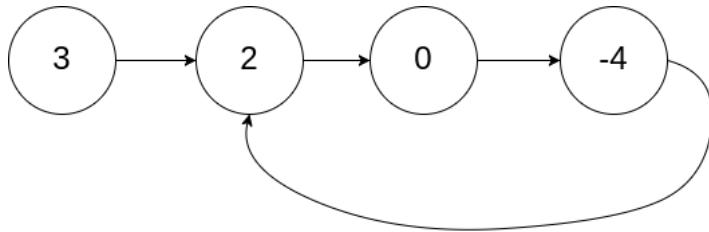
要求：判断链表中是否有环，如果有环则返回入环的第一个节点，无环则返回 `None`。

说明：

- 链表中节点的数目范围在范围 $[0, 10^4]$ 内。
- $-10^5 \leq Node.val \leq 10^5$ 。
- `pos` 的值为 `-1` 或者链表中的一个有效索引。

示例：

- 示例 1：

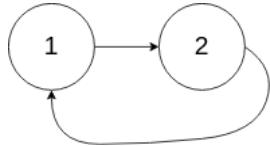


输入: head = [3, 2, 0, -4], pos = 1

输出: 返回索引为 1 的链表节点

解释: 链表中有一个环, 其尾部连接到第二个节点。

- 示例 2:



输入: head = [1, 2], pos = 0

输出: 返回索引为 0 的链表节点

解释: 链表中有一个环, 其尾部连接到第一个节点。

解题思路

思路 1：快慢指针（Floyd 判圈算法）

- 利用两个指针, 一个慢指针 `slow` 每次前进一步, 快指针 `fast` 每次前进两步 (两步或多步效果是等价的)。
- 如果两个指针在链表头节点以外的某一节点相遇 (即相等) 了, 那么说明链表有环。
- 否则, 如果 (快指针) 到达了某个没有后继指针的节点时, 那么说明没环。
- 如果有环, 则再定义一个指针 `ans`, 和慢指针一起每次移动一步, 两个指针相遇的位置即为入口节点。

这是因为: 假设入环位置为 A, 快慢指针在 B 点相遇, 则相遇时慢指针走了 $a + b$ 步, 快指针走了 $a + n(b + c) + b$ 步。

因为快指针总共走的步数是慢指针走的步数的两倍, 即 $2(a + b) = a + n(b + c) + b$, 所以可以推出: $a = c + (n - 1)(b + c)$ 。

我们可以发现: 从相遇点到入环点的距离 c 加上 $n - 1$ 圈的环长 $b + c$ 刚好等于从链表头部到入环点的距离。

思路 1：代码

```

class Solution:
    def detectCycle(self, head: ListNode) -> ListNode:
        fast, slow = head, head
        while True:
            if not fast or not fast.next:
                return None
            fast = fast.next.next
            slow = slow.next
            if fast == slow:
                break

        ans = head
        while ans != slow:
            ans, slow = ans.next, slow.next
        return ans
  
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。

- 空间复杂度: $O(1)$ 。

0143. 重排链表

- 标签: 栈、递归、链表、双指针
- 难度: 中等

题目链接

- 0143. 重排链表 - 力扣

题目大意

描述: 给定一个单链表 L 的头节点 $head$, 单链表 L 表示为: $L_0 \rightarrow L_1 \rightarrow L_2 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$ 。

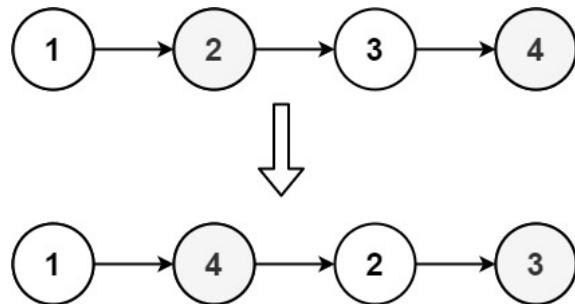
要求: 将单链表 L 重新排列为: $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow L_3 \rightarrow L_{n-3} \rightarrow \dots$ 。

说明:

- 需要将实际节点进行交换。

示例:

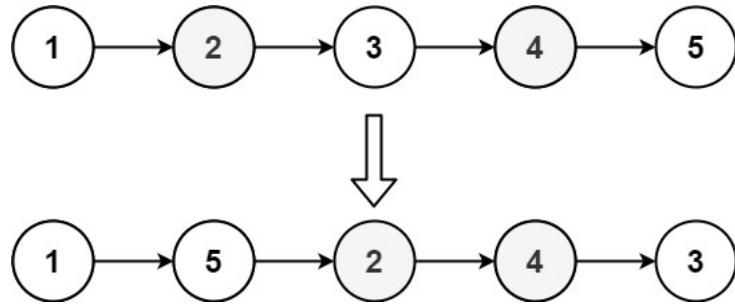
- 示例 1:



输入: head = [1, 2, 3, 4]

输出: [1, 4, 2, 3]

- 示例 2:



输入: head = [1, 2, 3, 4, 5]

输出: [1, 5, 2, 4, 3]

解题思路

思路 1：线性表

因为链表无法像数组那样直接进行随机访问。所以我们可以先将链表转为线性表，然后直接按照题要求的排列顺序访问对应数据元素，重新建立链表。

思路 1：代码

```
class Solution:
    def reorderList(self, head: ListNode) -> None:
        """
        Do not return anything, modify head in-place instead.
        """

        if not head:
            return

        vec = []
        node = head
        while node:
            vec.append(node)
            node = node.next

        left, right = 0, len(vec) - 1
        while left < right:
            vec[left].next = vec[right]
            left += 1
            if left == right:
                break
            vec[right].next = vec[left]
            right -= 1
        vec[left].next = None
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

0144. 二叉树的前序遍历

- 标签：栈、树、深度优先搜索、二叉树
- 难度：简单

题目链接

- [0144. 二叉树的前序遍历 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

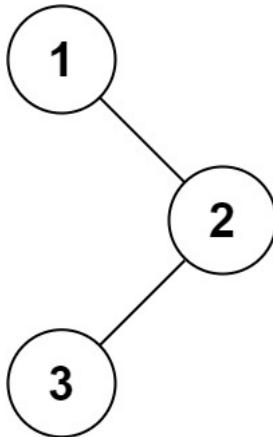
要求：返回该二叉树的前序遍历结果。

说明：

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

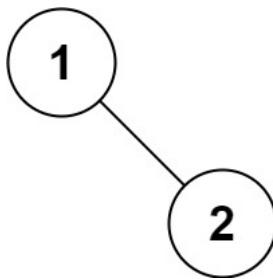
示例：

- 示例 1:



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

- 示例 2:



输入: `root = [1,null,2]`
输出: `[1,2]`

解题思路

思路 1：递归遍历

二叉树的前序遍历递归实现步骤为：

1. 判断二叉树是否为空，为空则直接返回。
2. 先访问根节点。
3. 然后递归遍历左子树。
4. 最后递归遍历右子树。

思路 1：代码

```

class Solution:
    def preorderTraversal(self, root: TreeNode) -> List[int]:
        res = []

        def preorder(root):
            if not root:
                return
            res.append(root.val)
            preorder(root.left)
            preorder(root.right)

        preorder(root)
        return res
  
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。

思路 2：模拟栈迭代遍历

二叉树的前序遍历递归实现的过程，实际上就是调用系统栈的过程。我们也可以使用一个显式栈 `stack` 来模拟递归的过程。

前序遍历的顺序为：根节点 - 左子树 - 右子树，而根据栈的「先入后出」特点，所以入栈的顺序应该为：先放入右子树，再放入左子树。这样可以保证最终为前序遍历顺序。

二叉树的前序遍历显式栈实现步骤如下：

- 判断二叉树是否为空，为空则直接返回。
- 初始化维护一个栈，将根节点入栈。
- 当栈不为空时：
 - 弹出栈顶元素 `node`，并访问该元素。
 - 如果 `node` 的右子树不为空，则将 `node` 的右子树入栈。
 - 如果 `node` 的左子树不为空，则将 `node` 的左子树入栈。

思路 2：代码

```
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root: # 二叉树为空直接返回
            return []

        res = []
        stack = [root]

        while stack: # 栈不为空
            node = stack.pop() # 弹出根节点
            res.append(node.val) # 访问根节点
            if node.right:
                stack.append(node.right) # 右子树入栈
            if node.left:
                stack.append(node.left) # 左子树入栈

        return res
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。[# 0145. 二叉树的后序遍历](#)
- 标签：栈、树、深度优先搜索、二叉树
- 难度：简单

题目链接

- [0145. 二叉树的后序遍历 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

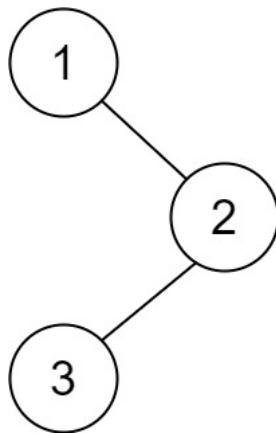
要求：返回该二叉树的后序遍历结果。

说明：

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

示例：

- 示例 1：



输入：root = [1,null,2,3]

输出：[3,2,1]

- 示例 2：

输入：root = []

输出：[]

解题思路

思路 1：递归遍历

二叉树的后序遍历递归实现步骤为：

1. 判断二叉树是否为空，为空则直接返回。
2. 先递归遍历左子树。
3. 然后递归遍历右子树。
4. 最后访问根节点。

思路 1：代码

```

class Solution:
    def postorderTraversal(self, root: TreeNode) -> List[int]:
        res = []
        def postorder(root):
            if not root:
                return
            postorder(root.left)
            postorder(root.right)
            res.append(root.val)

        postorder(root)
        return res
  
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。

思路 2：模拟栈迭代遍历

我们可以使用一个显式栈 `stack` 来模拟二叉树的后序遍历递归的过程。

与前序、中序遍历不同，在后序遍历中，根节点的访问要放在左右子树访问之后。因此，我们要保证：**在左右孩子节点访问结束之前，当前节点不能提前出栈。**

我们应该从根节点开始，先将根节点放入栈中，然后依次遍历左子树，不断将当前子树的根节点放入栈中，直到遍历到左子树最左侧的那个节点，从栈中弹出该元素，并判断该元素的右子树是否已经访问完毕，如果访问完毕，则访问该元素。如果未访问完毕，则访问该元素的右子树。

二叉树的后序遍历显式栈实现步骤如下：

1. 判断二叉树是否为空，为空则直接返回。
2. 初始化维护一个空栈，使用 `prev` 保存前一个访问的节点，用于确定当前节点的右子树是否访问完毕。
3. 当根节点或者栈不为空时，从当前节点开始：
 - i. 如果当前节点有左子树，则不断遍历左子树，并将当前根节点压入栈中。
 - ii. 如果当前节点无左子树，则弹出栈顶元素 `node`。
 - iii. 如果栈顶元素 `node` 无右子树（即 `not node.right`）或者右子树已经访问完毕（即 `node.right == prev`），则访问该元素，然后记录前一节点，并将当前节点标记为空节点。
 - iv. 如果栈顶元素有右子树，则将栈顶元素重新压入栈中，继续访问栈顶元素的右子树。

思路 2：代码

```
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        stack = []
        prev = None # 保存前一个访问的节点，用于确定当前节点的右子树是否访问完毕

        while root or stack:
            while root:
                stack.append(root) # 将当前树的根节点入栈
                root = root.left # 继续访问左子树，找到最左侧节点

            node = stack.pop() # 遍历到最左侧，当前节点无左子树时，将最左侧节点弹出

            # 如果当前节点无右子树或者右子树访问完毕
            if not node.right or node.right == prev:
                res.append(node.val) # 访问该节点
                prev = node # 记录前一节点
                root = None # 将当前根节点标记为空
            else:
                stack.append(node) # 右子树尚未访问完毕，将当前节点重新压回栈中
                root = node.right # 继续访问右子树

        return res
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度： $O(n)$ 。[# 0147. 对链表进行插入排序](#)
- 标签：链表、排序
- 难度：中等

题目链接

- [0147. 对链表进行插入排序 - 力扣](#)

题目大意

描述：给定链表的头节点 `head`。

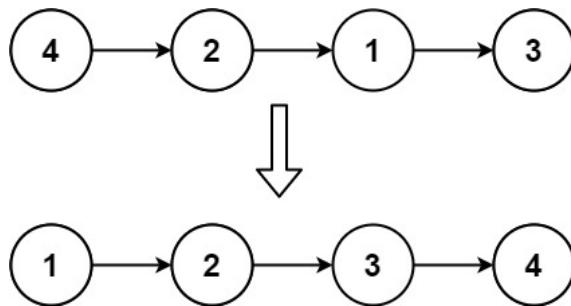
要求：对链表进行插入排序。

说明：

- 插入排序算法：
 - 插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。
 - 每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。
 - 重复直到所有输入数据插入完为止。
- 列表中的节点数在 $[1, 5000]$ 范围内。
- $-5000 \leq \text{Node.val} \leq 5000$ 。

示例：

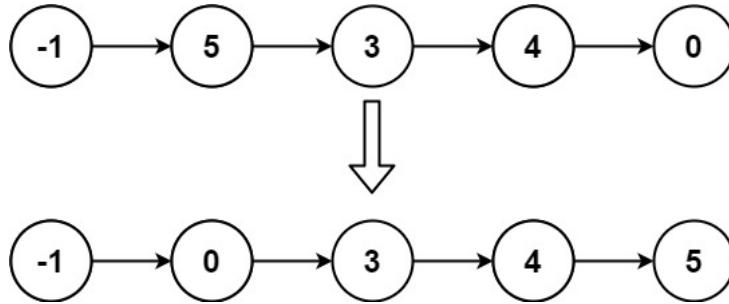
- 示例 1：



输入： `head = [4, 2, 1, 3]`

输出： `[1, 2, 3, 4]`

- 示例 2：



输入： `head = [-1, 5, 3, 4, 0]`

输出： `[-1, 0, 3, 4, 5]`

解题思路

思路 1：链表插入排序

- 先使用哑节点 `dummy_head` 构造一个指向 `head` 的指针，使得可以从 `head` 开始遍历。
- 维护 `sorted_list` 为链表的已排序部分的最后一个节点，初始时，`sorted_list = head`。
- 维护 `prev` 为插入元素位置的前一个节点，维护 `cur` 为待插入元素。初始时，`prev = head`，`cur = head.next`。
- 比较 `sorted_list` 和 `cur` 的节点值。
 - 如果 `sorted_list.val <= cur.val`，说明 `cur` 应该插入到 `sorted_list` 之后，则将 `sorted_list` 后移一位。
 - 如果 `sorted_list.val > cur.val`，说明 `cur` 应该插入到 `head` 与 `sorted_list` 之间。则使用 `prev` 从 `head` 开始遍历，直到找到插入 `cur` 的位置的前一个节点位置。然后将 `cur` 插入。
- 令 `cur = sorted_list.next`，此时 `cur` 为下一个待插入元素。

6. 重复 4、5 步骤，直到 `cur` 遍历结束为空。返回 `dummy_head` 的下一个节点。

思路 1：代码

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

    dummy_head = ListNode(-1)
    dummy_head.next = head
    sorted_list = head
    cur = head.next

    while cur:
        if sorted_list.val <= cur.val:
            # 将 cur 插入到 sorted_list 之后
            sorted_list = sorted_list.next
        else:
            prev = dummy_head
            while prev.next.val <= cur.val:
                prev = prev.next
            # 将 cur 到链表中间
            sorted_list.next = cur.next
            cur.next = prev.next
            prev.next = cur
            cur = sorted_list.next

    return dummy_head.next
```

思路 1：复杂度分析

- 时间复杂度: $O(n^2)$ 。
- 空间复杂度: $O(1)$ 。

0148. 排序链表

- 标签：链表、双指针、分治、排序、归并排序
- 难度：中等

题目链接

- [0148. 排序链表 - 力扣](#)

题目大意

描述：给定链表的头节点 `head`。

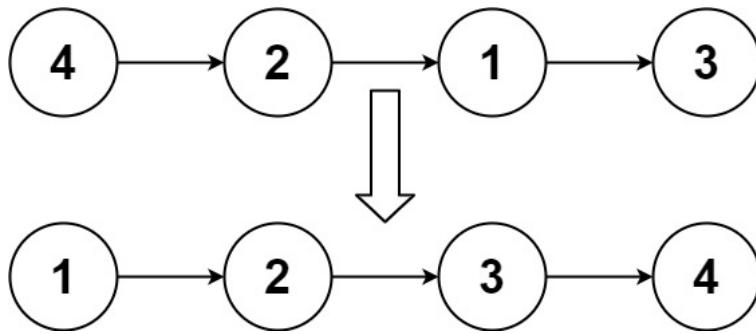
要求：按照升序排列并返回排序后的链表。

说明：

- 链表中节点的数目在范围 $[0, 5 * 10^4]$ 内。
- $-10^5 \leq Node.val \leq 10^5$ 。

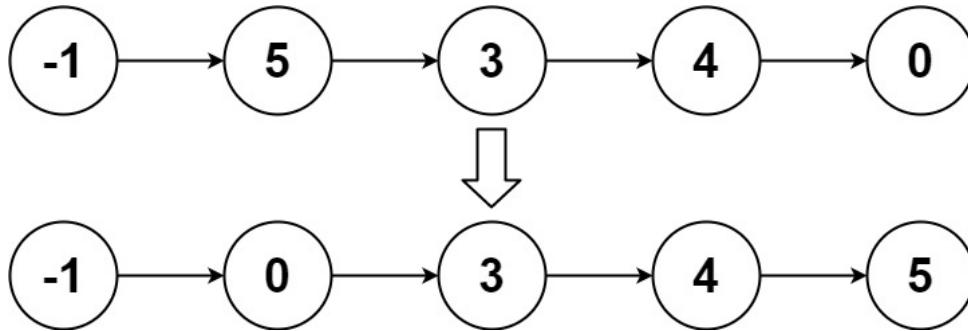
示例：

- 示例 1：



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

- 示例 2:



输入: head = [-1, 5, 3, 4, 0]
输出: [-1, 0, 3, 4, 5]

解题思路

思路 1：链表冒泡排序（超时）

- 使用三个指针 `node_i`、`node_j` 和 `tail`。其中 `node_i` 用于控制外循环次数，循环次数为链节点个数（链表长度）。`node_j` 和 `tail` 用于控制内循环次数和循环结束位置。
- 排序开始前，将 `node_i`、`node_j` 置于头节点位置。`tail` 指向链表末尾，即 `None`。
- 比较链表中相邻两个元素 `node_j.val` 与 `node_j.next.val` 的值大小，如果 `node_j.val > node_j.next.val`，则值相互交换。然后向右移动 `node_j` 指针，直到 `node_j.next == tail` 时停止。
- 一次循环之后，将 `tail` 移动到 `node_j` 所在位置。相当于 `tail` 向左移动了一位。此时 `tail` 节点右侧为链表中最大的链节点。
- 然后移动 `node_i` 节点，并将 `node_j` 置于头节点位置。然后重复第 3、4 步操作。
- 直到 `node_i` 节点移动到链表末尾停止，排序结束。
- 返回链表的头节点 `head`。

思路 1：代码

```

class Solution:
    def bubbleSort(self, head: ListNode):
        node_i = head
        tail = None
        # 外层循环次数为 链表节点个数
        while node_i:
            node_j = head
            while node_j and node_j.next != tail:
                if node_j.val > node_j.next.val:
                    # 交换两个节点的值
                    node_j.val, node_j.next.val = node_j.next.val, node_j.val
                    node_j = node_j.next
            # 尾指针向前移动 1 位，此时尾指针右侧为排好序的链表
            tail = node_j
            node_i = node_i.next

        return head

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.bubbleSort(head)

```

思路 1：复杂度分析

- 时间复杂度: $O(n^2)$ 。
- 空间复杂度: $O(1)$ 。

思路 2：链表选择排序（超时）

- 使用两个指针 `node_i`、`node_j`。`node_i` 既可以用于控制外循环次数，又可以作为当前未排序链表的第一个链节点位置。
- 使用 `min_node` 记录当前未排序链表中值最小的链节点。
- 每一趟排序开始时，先令 `min_node = node_i`（即暂时假设链表中 `node_i` 节点为值最小的节点，经过比较后再确定最小值节点位置）。
- 然后依次比较未排序链表中 `node_j.val` 与 `min_node.val` 的值大小。如果 `node_j.val < min_node.val`，则更新 `min_node` 为 `node_j`。
- 这一趟排序结束时，未排序链表中最小值节点为 `min_node`，如果 `node_i != min_node`，则将 `node_i` 与 `min_node` 值进行交换。如果 `node_i == min_node`，则不用交换。
- 排序结束后，继续向右移动 `node_i`，重复上述步骤，在剩余未排序链表中寻找最小的链节点，并与 `node_i` 进行比较和交换，直到 `node_i == None` 或者 `node_i.next == None` 时，停止排序。
- 返回链表的头节点 `head`。

思路 2：代码

```

class Solution:
    def sectionSort(self, head: ListNode):
        node_i = head
        # node_i 为当前未排序链表的第一个链节点
        while node_i and node_i.next:
            # min_node 为未排序链表中的值最小节点
            min_node = node_i
            node_j = node_i.next
            while node_j:
                if node_j.val < min_node.val:
                    min_node = node_j
                node_j = node_j.next
            # 交换值最小节点与未排序链表中第一个节点的值
            if node_i != min_node:
                node_i.val, min_node.val = min_node.val, node_i.val
            node_i = node_i.next

        return head

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.sectionSort(head)

```

思路 2：复杂度分析

- 时间复杂度: $O(n^2)$ 。
- 空间复杂度: $O(1)$ 。

思路 3：链表插入排序（超时）

- 先使用哑节点 `dummy_head` 构造一个指向 `head` 的指针，使得可以从 `head` 开始遍历。
- 维护 `sorted_list` 为链表的已排序部分的最后一个节点，初始时，`sorted_list = head`。
- 维护 `prev` 为插入元素位置的前一个节点，维护 `cur` 为待插入元素。初始时，`prev = head`，`cur = head.next`。
- 比较 `sorted_list` 和 `cur` 的节点值。
 - 如果 `sorted_list.val <= cur.val`，说明 `cur` 应该插入到 `sorted_list` 之后，则将 `sorted_list` 后移一位。
 - 如果 `sorted_list.val > cur.val`，说明 `cur` 应该插入到 `head` 与 `sorted_list` 之间。则使用 `prev` 从 `head` 开始遍历，直到找到插入 `cur` 的位置的前一个节点位置。然后将 `cur` 插入。
- 令 `cur = sorted_list.next`，此时 `cur` 为下一个待插入元素。
- 重复 4、5 步骤，直到 `cur` 遍历结束为空。返回 `dummy_head` 的下一个节点。

思路 3：代码

```

class Solution:
    def insertionSort(self, head: ListNode):
        if not head or not head.next:
            return head

        dummy_head = ListNode(-1)
        dummy_head.next = head
        sorted_list = head
        cur = head.next

        while cur:
            if sorted_list.val <= cur.val:
                # 将 cur 插入到 sorted_list 之后
                sorted_list = sorted_list.next
            else:
                prev = dummy_head
                while prev.next.val <= cur.val:
                    prev = prev.next
                # 将 cur 到链表中间
                sorted_list.next = cur.next
                cur.next = prev.next
                prev.next = cur
            cur = sorted_list.next

        return dummy_head.next

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.insertionSort(head)

```

思路 3：复杂度分析

- 时间复杂度: $O(n^2)$ 。
- 空间复杂度: $O(1)$ 。

思路 4：链表归并排序（通过）

1. 分割环节：找到链表中心链节点，从中心节点将链表断开，并递归进行分割。

- 使用快慢指针 `fast = head.next`、`slow = head`，让 `fast` 每次移动 2 步，`slow` 移动 1 步，移动到链表末尾，从而找到链表中心链节点，即 `slow`。
- 从中心位置将链表从中心位置分为左右两个链表 `left_head` 和 `right_head`，并从中心位置将其断开，即 `slow.next = None`。
- 对左右两个链表分别进行递归分割，直到每个链表中只包含一个链节点。

2. 归并环节：将递归后的链表进行两两归并，完成一遍后每个子链表长度加倍。重复进行归并操作，直到得到完整的链表。

- 使用哑节点 `dummy_head` 构造一个头节点，并使用 `cur` 指向 `dummy_head` 用于遍历。
- 比较两个链表头节点 `left` 和 `right` 的值大小。将较小的头节点加入到合并后的链表中。并向后移动该链表的头节点指针。
- 然后重复上一步操作，直到两个链表中出现链表为空的情况。
- 将剩余链表插入到合并中的链表中。
- 将哑节点 `dummy_head` 的下一个链节点 `dummy_head.next` 作为合并后的头节点返回。

思路 4：代码

```

class Solution:
    def merge(self, left, right):
        # 归并环节
        dummy_head = ListNode(-1)
        cur = dummy_head
        while left and right:
            if left.val <= right.val:
                cur.next = left
                left = left.next
            else:
                cur.next = right
                right = right.next
            cur = cur.next

        if left:
            cur.next = left
        elif right:
            cur.next = right

        return dummy_head.next

    def mergeSort(self, head: ListNode):
        # 分割环节
        if not head or not head.next:
            return head

        # 快慢指针找到中心链节点
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # 断开左右链节点
        left_head, right_head = head, slow.next
        slow.next = None

        # 归并操作
        return self.merge(self.mergeSort(left_head), self.mergeSort(right_head))

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.mergeSort(head)

```

思路 4：复杂度分析

- 时间复杂度: $O(n \times \log_2 n)$ 。
- 空间复杂度: $O(1)$ 。

思路 5：链表快速排序（超时）

- 从链表中找到一个基准值 `pivot`，这里以头节点为基准值。
- 然后通过快慢指针 `node_i`、`node_j` 在链表中移动，使得 `node_i` 之前的节点值都小于基准值，`node_i` 之后的节点值都大于基准值。从而把数组拆分为左右两个部分。
- 再对左右两个部分分别重复第二步，直到各个部分只有一个节点，则排序结束。

注意：

虽然链表快速排序算法的平均时间复杂度为 $O(n \times \log_2 n)$ 。但链表快速排序算法中基准值 `pivot` 的取值做不到数组快速排序算法中的随机选择。一旦给定序列是有序链表，时间复杂度就会退化到 $O(n^2)$ 。这也是这道题目使用链表快速排序容易超时的原因。

思路 5：代码

```

class Solution:
    def partition(self, left: ListNode, right: ListNode):
        # 左闭右开，区间没有元素或者只有一个元素，直接返回第一个节点
        if left == right or left.next == right:
            return left
        # 选择头节点为基准节点
        pivot = left.val
        # 使用 node_i, node_j 双指针，保证 node_i 之前的节点值都小于基准节点值，node_i 与 node_j 之间的节点值都大于等于基准节点值
        node_i, node_j = left, left.next

        while node_j != right:
            # 发现一个小于基准值的元素
            if node_j.val < pivot:
                # 因为 node_i 之前节点都小于基准值，所以先将 node_i 向右移动一位（此时 node_i 节点值大于等于基准节点值）
                node_i = node_i.next
                # 将小于基准值的元素 node_j 与当前 node_i 换位，换位后可以保证 node_i 之前的节点都小于基准节点值
                node_i.val, node_j.val = node_j.val, node_i.val
            node_j = node_j.next
        # 将基准节点放到正确位置上
        node_i.val, left.val = left.val, node_i.val
        return node_i

    def quickSort(self, left: ListNode, right: ListNode):
        if left == right or left.next == right:
            return left
        pi = self.partition(left, right)
        self.quickSort(left, pi)
        self.quickSort(pi.next, right)
        return left

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next:
            return head
        return self.quickSort(head, None)

```

思路 5：复杂度分析

- 时间复杂度： $O(n \times \log_2 n)$ 。
- 空间复杂度： $O(1)$ 。

思路 6：链表计数排序（通过）

- 使用 `cur` 指针遍历一遍链表。找出链表中最大值 `list_max` 和最小值 `list_min`。
- 使用数组 `counts` 存储节点出现次数。
- 再次使用 `cur` 指针遍历一遍链表。将链表中每个值为 `cur.val` 的节点出现次数，存入数组对应第 `cur.val - list_min` 项中。
- 反向填充目标链表：
 - 建立一个哑节点 `dummy_head`，作为链表的头节点。使用 `cur` 指针指向 `dummy_head`。
 - 从小到大遍历数组 `counts`。对于每个 `counts[i] != 0` 的元素建立一个链节点，值为 `i + list_min`，将其插入到 `cur.next` 上。并向右移动 `cur`。同时 `counts[i] -= 1`。直到 `counts[i] == 0` 后继续向后遍历数组 `counts`。
- 将哑节点 `dummy_head` 的下一个链节点 `dummy_head.next` 作为新链表的头节点返回。

思路 6：代码

```

class Solution:
    def countingSort(self, head: ListNode):
        if not head:
            return head

        # 找出链表中最大值 list_max 和最小值 list_min
        list_min, list_max = float('inf'), float('-inf')
        cur = head
        while cur:
            if cur.val < list_min:
                list_min = cur.val
            if cur.val > list_max:
                list_max = cur.val
            cur = cur.next

        size = list_max - list_min + 1
        counts = [0 for _ in range(size)]

        cur = head
        while cur:
            counts[cur.val - list_min] += 1
            cur = cur.next

        dummy_head = ListNode(-1)
        cur = dummy_head
        for i in range(size):
            while counts[i]:
                cur.next = ListNode(i + list_min)
                counts[i] -= 1
                cur = cur.next
        return dummy_head.next

    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        return self.countingSort(head)

```

思路 6：复杂度分析

- 时间复杂度: $O(n + k)$, 其中 k 代表待排序链表中所有元素的值域。
- 空间复杂度: $O(k)$ 。

思路 7：链表桶排序（通过）

- 使用 `cur` 指针遍历一遍链表。找出链表中最大值 `list_max` 和最小值 `list_min`。
- 通过 $(\text{最大值} - \text{最小值}) / \text{每个桶的大小}$ 计算出桶的个数，即 `bucket_count = (list_max - list_min) // bucket_size + 1` 个桶。
- 定义数组 `buckets` 为桶，桶的个数为 `bucket_count` 个。
- 使用 `cur` 指针再次遍历一遍链表，将每个元素装入对应的桶中。
- 对每个桶内的元素单独排序，可以使用链表插入排序（超时）、链表归并排序（通过）、链表快速排序（超时）等算法。
- 最后按照顺序将桶内的元素拼成新的链表，并返回。

思路 7：代码

```

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    # 将链表节点值 val 添加到对应桶 buckets[index] 中
    def insertion(self, buckets, index, val):
        if not buckets[index]:
            buckets[index] = ListNode(val)
            return

        node = ListNode(val)
        node.next = buckets[index]
        buckets[index] = node

    # 归并环节
    def merge(self, left, right):
        dummy_head = ListNode(-1)
        cur = dummy_head
        while left and right:
            if left.val <= right.val:
                cur.next = left
                left = left.next
            else:
                cur.next = right
                right = right.next
            cur = cur.next

        if left:
            cur.next = left
        elif right:
            cur.next = right

        return dummy_head.next

    def mergeSort(self, head: ListNode):
        # 分割环节
        if not head or not head.next:
            return head

        # 快慢指针找到中心链节点
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # 断开左右链节点
        left_head, right_head = head, slow.next
        slow.next = None

        # 归并操作
        return self.merge(self.mergeSort(left_head), self.mergeSort(right_head))

    def bucketSort(self, head: ListNode, bucket_size=5):
        if not head:
            return head

        # 找出链表中最大值 list_max 和最小值 list_min
        list_min, list_max = float('inf'), float('-inf')
        cur = head
        while cur:

```

```

    if cur.val < list_min:
        list_min = cur.val
    if cur.val > list_max:
        list_max = cur.val
    cur = cur.next

# 计算桶的个数，并定义桶
bucket_count = (list_max - list_min) // bucket_size + 1
buckets = [[] for _ in range(bucket_count)]

# 将链表节点值依次添加到对应桶中
cur = head
while cur:
    index = (cur.val - list_min) // bucket_size
    self.insertion(buckets, index, cur.val)
    cur = cur.next

dummy_head = ListNode(-1)
cur = dummy_head
# 将元素依次出桶，并拼接成有序链表
for bucket_head in buckets:
    bucket_cur = self.mergeSort(bucket_head)
    while bucket_cur:
        cur.next = bucket_cur
        cur = cur.next
        bucket_cur = bucket_cur.next

return dummy_head.next

def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
    return self.bucketSort(head)

```

思路 7：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n + m)$ 。 m 为桶的个数。

思路 8：链表基数排序（解答错误，普通链表基数排序只适合非负数）

- 使用 `cur` 指针遍历链表，获取节点值位数最长的位数 `size`。
- 从个位到高位遍历位数。因为 `0 ~ 9` 共有 `10` 位数字，所以建立 `10` 个桶。
- 以每个节点对应位数上的数字为索引，将节点值放入到对应桶中。
- 建立一个哑节点 `dummy_head`，作为链表的头节点。使用 `cur` 指针指向 `dummy_head`。
- 将桶中元素依次取出，并根据元素值建立链表节点，并插入到新的链表后面。从而生成新的链表。
- 之后依次以十位，百位，...，直到最大值元素的最高位处值为索引，放入到对应桶中，并生成新的链表，最终完成排序。
- 将哑节点 `dummy_head` 的下一个链节点 `dummy_head.next` 作为新链表的头节点返回。

思路 8：代码

```

class Solution:
    def radixSort(self, head: ListNode):
        # 计算位数最长的位数
        size = 0
        cur = head
        while cur:
            val_len = len(str(cur.val))
            if val_len > size:
                size = val_len
            cur = cur.next

        # 从个位到高位遍历位数
        for i in range(size):
            buckets = [[] for _ in range(10)]
            cur = head
            while cur:
                # 以每个节点对应位数上的数字为索引，将节点值放入到对应桶中
                buckets[cur.val // (10 ** i) % 10].append(cur.val)
                cur = cur.next

            # 生成新的链表
            dummy_head = ListNode(-1)
            cur = dummy_head
            for bucket in buckets:
                for num in bucket:
                    cur.next = ListNode(num)
                    cur = cur.next
            head = dummy_head.next

    return head

def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
    return self.radixSort(head)

```

思路 8：复杂度分析

- 时间复杂度： $O(n \times k)$ 。其中 n 是待排序元素的个数， k 是数字位数。 k 的大小取决于数字位的选择（十进制位、二进制位）和待排序元素所属数据类型全集的大小。
- 空间复杂度： $O(n + k)$ 。

参考资料

- 【文章】单链表的冒泡排序_zhao_miao的博客 - CSDN博客
- 【文章】链表排序总结（全）(C++) - 阿祭儿 - CSDN博客
- 【题解】快排、冒泡、选择排序实现列表排序 - 排序链表 - 力扣
- 【题解】归并排序+快速排序 - 排序链表 - 力扣
- 【题解】排序链表（递归+迭代）详解 - 排序链表 - 力扣
- 【题解】Sort List （归并排序链表） - 排序链表 - 力扣

0149. 直线上最多的点数

- 标签：几何、数组、哈希表、数学
- 难度：困难

题目链接

- 0149. 直线上最多的点数 - 力扣

题目大意

给定一个平面上的 n 个点的坐标数组 points，求解最多有多少个点在同一条直线上。

解题思路

两个点可以确定一条直线，固定其中一个点，求其他点与该点的斜率，斜率相同的点则在同一条直线上。可以考虑把斜率当做哈希表的键值，存储经过该点，不同斜率的直线上经过的点数目。

对于点 i，查找经过该点的直线只需要考虑 (i+1, n-1) 位置上的点即可，因为 i-1 之前的数据已经在遍历点 i-2 的时候考虑过了。

斜率的计算公式为 $\frac{dy}{dx} = \frac{y_j - y_i}{x_j - x_i}$ 。

因为斜率是小数会有精度误差，所以我们考虑使用 (dx, dy) 的元组作为哈希表的 key。

注意：

需要处理倍数关系，dy、dx 异号情况，以及处理垂直直线（两点横坐标差为 0）的水平直线（两点横坐标差为 0）的情况。

代码

```
class Solution:
    def maxPoints(self, points: List[List[int]]) -> int:
        n = len(points)
        if n < 3:
            return n
        ans = 0
        for i in range(n):
            line_dict = dict()
            line_dict[0] = 0
            same = 1
            for j in range(i+1, n):
                dx = points[j][0] - points[i][0]
                dy = points[j][1] - points[i][1]
                if dx == 0 and dy == 0:
                    same += 1
                    continue
                gcd_dx_dy = math.gcd(abs(dx), abs(dy))
                if (dx > 0 and dy > 0) or (dx < 0 and dy < 0):
                    dx = abs(dx) // gcd_dx_dy
                    dy = abs(dy) // gcd_dx_dy
                elif dx < 0 and dy > 0:
                    dx = -dx // gcd_dx_dy
                    dy = -dy // gcd_dx_dy
                elif dx > 0 and dy < 0:
                    dx = dx // gcd_dx_dy
                    dy = dy // gcd_dx_dy
                elif dx == 0 and dy != 0:
                    dy = 1
                elif dx != 0 and dy == 0:
                    dx = 1
                key = (dx, dy)
                if key in line_dict:
                    line_dict[key] += 1
                else:
                    line_dict[key] = 1
            ans = max(ans, same + max(line_dict.values()))
        return ans
```

0150. 逆波兰表达式求值

- 标签：栈、数组、数学
- 难度：中等

题目链接

- [0150. 逆波兰表达式求值 - 力扣](#)

题目大意

描述：给定一个字符串数组 `tokens`，表示「逆波兰表达式」。

要求：求解表达式的值。

说明：

- **逆波兰表达式：**也称为后缀表达式。
 - 中缀表达式 `(1 + 2) * (3 + 4)`，对应的逆波兰表达式为 `((1 2 +) (3 4 +) *)`。
- $1 \leq tokens.length \leq 10^4$ 。
- `tokens[i]` 是一个算符 (+、-、* 或 /)，或是在范围 $[-200, 200]$ 内的一个整数。

示例：

- **示例 1：**

```
输入: tokens = ["4", "13", "5", "/", "+"]
输出: 6
解释: 该算式转化为常见的中缀算术表达式为: (4 + (13 / 5)) = 6
```

- **示例 2：**

```
输入: tokens = ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
输出: 22
解释: 该算式转化为常见的中缀算术表达式为:
((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

解题思路

思路 1：栈

这道题是栈的典型应用。我们先来简单介绍一下逆波兰表达式。

逆波兰表达式，也叫做后缀表达式，特点是：没有括号，运算符总是放在和它相关的操作数之后。

我们平常见到的表达式是中缀表达式，可写为：`A 运算符 B`。其中 `A`、`B` 都是操作数。

而后缀表达式可写为：`A B 运算符`。

逆波兰表达式的计算遵循从左到右的规律。我们在计算逆波兰表达式的值时，可以使用一个栈来存放当前的操作数，从左到右依次遍历逆波兰表达式，计算出对应的值。具体操作步骤如下：

1. 使用列表 `stack` 作为栈存放操作数，然后遍历表达式的字符串数组。
2. 如果当前字符为运算符，则取出栈顶两个元素，在进行对应的运算之后，再将运算结果入栈。
3. 如果当前字符为数字，则直接将数字入栈。

- 4. 遍历结束后弹出栈中最后剩余的元素，这就是最终结果。

思路 1：代码

```
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for token in tokens:
            if token == '+':
                stack.append(stack.pop() + stack.pop())
            elif token == '-':
                stack.append(-stack.pop() + stack.pop())
            elif token == '*':
                stack.append(stack.pop() * stack.pop())
            elif token == '/':
                stack.append(int(1 / stack.pop() * stack.pop()))
            else:
                stack.append(int(token))
        return stack.pop()
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

0151. 反转字符串中的单词

- 标签：双指针、字符串
- 难度：中等

题目链接

- [0151. 反转字符串中的单词 - 力扣](#)

题目大意

描述：给定一个字符串 `s`。

要求：反转字符串中所有单词的顺序。

说明：

- 单词**：由非空格字符组成的字符串。`s` 中使用至少一个空格将字符串中的单词分隔开。
- 输入字符串 `s` 中可能会存在前导空格、尾随空格或者单词间的多个空格。
- 返回的结果字符串中，单词间应当仅用单个空格分隔，且不包含任何额外的空格。
- $1 \leq s.length \leq 10^4$ 。
- `s` 包含英文大小写字母、数字和空格 ‘ ’
- `s` 中至少存在一个单词。

示例：

- 示例 1：

```
输入: s = " hello world "
输出: "world hello"
解释: 反转后的字符串中不能存在前导空格和尾随空格。
```

- 示例 2：

```
输入: s = "a good example"
输出: "example good a"
解释: 如果两个单词间有多余的空格, 反转后的字符串需要将单词间的空格减少到仅有一个。
```

解题思路

思路 1：调用库函数

直接调用 Python 的库函数，对字符串进行切片，翻转，然后拼合成字符串。

思路 1：代码

```
class Solution:
    def reverseWords(self, s: str) -> str:
        return " ".join(reversed(s.split()))
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是字符串 s 的长度。
- 空间复杂度: $O(1)$ 。

思路 2：模拟

第二种思路根据 API 的思路写出模拟代码，具体步骤如下：

- 使用数组 words 存放单词，使用字符串变量 cur 存放当前单词。
- 遍历字符串，对于当前字符 ch 。
 - 如果遇到空格，则：
 - 如果当前单词不为空，则将当前单词存入数组 words 中，并将当前单词置为空串
 - 如果遇到字符，则：
 - 将其存入当前单词中，即 $\text{cur} += \text{ch}$ 。
- 如果遍历完，当前单词不为空，则将当前单词存入数组 words 中。
- 然后对数组 words 进行翻转操作，令 $\text{words}[i], \text{words}[\text{len}(\text{words}) - 1 - i]$ 交换元素。
- 最后将 words 中的单词连接起来，中间拼接上空格，将其作为答案返回。

思路 2：代码

```
class Solution:
    def reverseWords(self, s: str) -> str:
        words = []
        cur = ""
        for ch in s:
            if ch == ' ':
                if cur:
                    words.append(cur)
                    cur = ""
            else:
                cur += ch
        if cur:
            words.append(cur)

        for i in range(len(words) // 2):
            words[i], words[len(words) - 1 - i] = words[len(words) - 1 - i], words[i]

        return " ".join(words)
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 是字符串 s 的长度。
- 空间复杂度： $O(1)$ 。# 0152. 乘积最大子数组
- 标签：数组、动态规划
- 难度：中等

题目链接

- [0152. 乘积最大子数组 - 力扣](#)

题目大意

描述：给定一个整数数组 nums 。

要求：找出数组中乘积最大的连续子数组（最少包含一个数字），并返回该子数组对应的乘积。

说明：

- 测试用例的答案是一个 32-位整数。
- 子数组**：数组的连续子序列。
- $1 \leq \text{nums.length} \leq 2 * 10^4$ 。
- $-10 \leq \text{nums}[i] \leq 10$ 。
- nums 的任何前缀或后缀的乘积都保证是一个 32-位整数。

示例：

- 示例 1：

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

- 示例 2：

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

解题思路

思路 1：动态规划

这道题跟「0053. 最大子序和」有点相似，不过一个求的是和的最大值，这道题求解的是乘积的最大值。

乘积有个特殊情况，两个正数、两个负数相乘都会得到正数。所以求解的时候需要考虑负数的情况。

若想要最终的乘积最大，则应该使子数组中的正数元素尽可能的大，负数元素尽可能的小。所以我们可以维护一个最大值变量和最小值变量。

1. 划分阶段

按照子数组的结尾位置进行阶段划分。

2. 定义状态

定义状态 $\text{dp_max}[i]$ 为：以第 i 个元素结尾的乘积最大子数组的乘积。

定义状态 $\text{dp_min}[i]$ 为：以第 i 个元素结尾的乘积最小子数组的乘积。

3. 状态转移方程

$$\text{dp_max}[i] = \max(\text{dp_max}[i - 1] * \text{nums}[i], \text{nums}[i], \text{dp_min}[i - 1] * \text{nums}[i])$$

- `dp_min[i] = min(dp_min[i - 1] * nums[i], nums[i], dp_max[i - 1] * nums[i])`

4. 初始条件

- 以第 0 个元素结尾的乘积最大子数组的乘积为 `nums[0]`，即 `dp_max[0] = nums[0]`。
- 以第 0 个元素结尾的乘积最小子数组的乘积为 `nums[0]`，即 `dp_min[0] = nums[0]`。

5. 最终结果

根据状态定义，最终结果为 dp_{max} 中最大值，即乘积最大子数组的乘积。

思路 1：代码

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        size = len(nums)
        dp_max = [0 for _ in range(size)]
        dp_min = [0 for _ in range(size)]
        dp_max[0] = nums[0]
        dp_min[0] = nums[0]
        ans = nums[0]
        for i in range(1, size):
            dp_max[i] = max(dp_max[i - 1] * nums[i], nums[i], dp_min[i - 1] * nums[i])
            dp_min[i] = min(dp_min[i - 1] * nums[i], nums[i], dp_max[i - 1] * nums[i])
        return max(dp_max)
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为整数数组 `nums` 的元素个数。
- 空间复杂度： $O(n)$ 。

思路 2：动态规划 + 滚动优化

因为状态转移方程中只涉及到当前元素和前一个元素，所以我们也可以不使用数组，只使用两个变量来维护 $dp_{max}[i]$ 和 $dp_{min}[i]$ 。

思路 2：代码

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        size = len(nums)
        max_num, min_num = nums[0], nums[0]
        ans = nums[0]
        for i in range(1, size):
            temp_max = max_num
            temp_min = min_num
            max_num = max(temp_max * nums[i], nums[i], temp_min * nums[i])
            min_num = min(temp_min * nums[i], nums[i], temp_max * nums[i])
            ans = max(max_num, ans)
        return ans
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为整数数组 `nums` 的元素个数。
- 空间复杂度： $O(1)$ 。

0153. 寻找旋转排序数组中的最小值

- 标签：数组、二分查找
- 难度：中等

题目链接

- 0153. 寻找旋转排序数组中的最小值 - 力扣

题目大意

描述：给定一个数组 $nums$, $nums$ 是有升序数组经过「旋转」得到的。但是旋转次数未知。数组中不存在重复元素。

要求：找出数组中的最小元素。

说明：

- 旋转操作：将数组整体右移若干位置。
- $n == nums.length$ 。
- $1 \leq n \leq 5000$ 。
- $-5000 \leq nums[i] \leq 5000$ 。
- $nums$ 中的所有整数互不相同。
- $nums$ 原来是一个升序排序的数组，并进行了 1 至 n 次旋转。

示例：

- 示例 1：

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

- 示例 2：

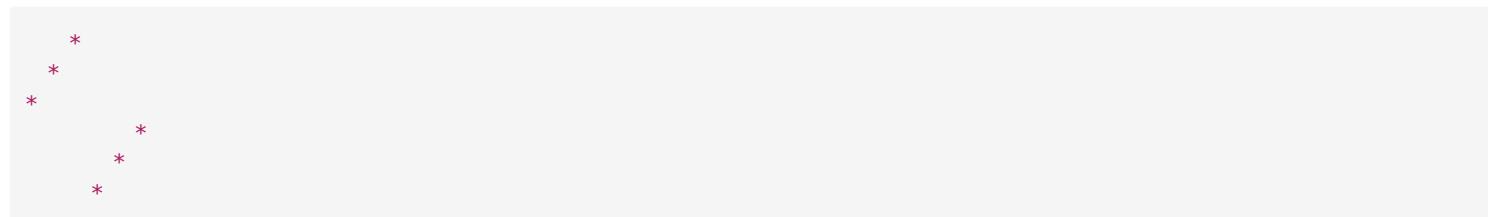
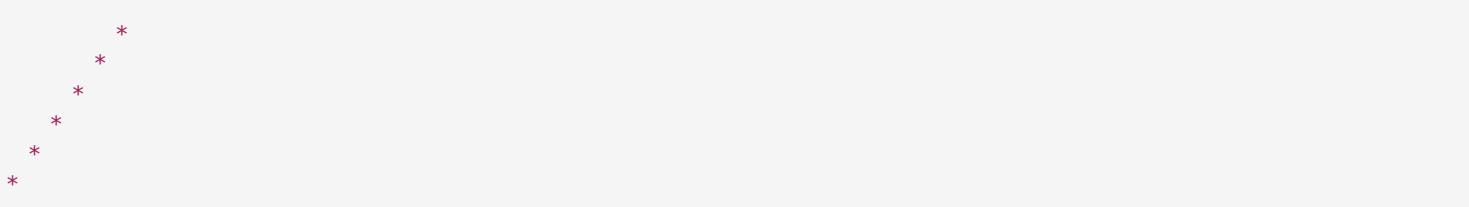
```
输入: nums = [4,5,6,7,0,1,2]
输出: 0
解释: 原数组为 [0,1,2,4,5,6,7]，旋转 4 次得到输入数组。
```

解题思路

思路 1：二分查找

数组经过「旋转」之后，会有两种情况，第一种就是原先的升序序列，另一种是两段升序的序列。

第一种的最小值在最左边。第二种最小值在第二段升序序列的第一个元素。



最直接的办法就是遍历一遍，找到最小值。但是还可以有更好的方法。考虑用二分查找来降低算法的时间复杂度。

创建两个指针 $left$ 、 $right$ ，分别指向数组首尾。然后计算出两个指针中间值 mid 。将 mid 与两个指针做比较。

- 如果 $\text{nums}[\text{mid}] > \text{nums}[\text{right}]$, 则最小值不可能在 mid 左侧, 一定在 mid 右侧, 则将 left 移动到 $\text{mid} + 1$ 位置, 继续查找右侧区间。
- 如果 $\text{nums}[\text{mid}] \leq \text{nums}[\text{right}]$, 则最小值一定在 mid 左侧, 或者 mid 位置, 将 right 移动到 mid 位置上, 继续查找左侧区间。

思路 1：代码

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

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。

154. 寻找旋转排序数组中的最小值 II

- 标签: 数组、二分查找
- 难度: 困难

题目链接

- [154. 寻找旋转排序数组中的最小值 II - 力扣](#)

题目大意

描述: 给定一个数组 nums , nums 是有升序数组经过 $1 \sim n$ 次「旋转」得到的。但是旋转次数未知。数组中可能存在重复元素。

要求: 找出数组中的最小元素。

说明:

- 旋转: 将数组整体右移 1 位。数组 $[a[0], a[1], a[2], \dots, a[n - 1]]$ 旋转一次的结果为数组 $[a[n - 1], a[0], a[1], a[2], \dots, a[n - 2]]$ 。
- $n == \text{nums.length}$ 。
- $1 \leq n \leq 5000$ 。
- $-5000 \leq \text{nums}[i] \leq 5000$
- nums 原来是一个升序排序的数组, 并进行了 $1 \sim n$ 次旋转。

示例:

- 示例 1:

```
输入: nums = [1,3,5]
输出: 1
```

- 示例 2:

```
输入: nums = [2,2,2,0,1]
输出: 0
```

解题思路

思路 1：二分查找

数组经过「旋转」之后，会有两种情况，第一种就是原先的升序序列，另一种是两段升序的序列。

第一种的最小值在最左边。

```
*  
*  
*  
*  
*  
*
```

第二种最小值在第二段升序序列的第一个元素。

```
*  
*  
*  
*  
*  
*
```

最直接的办法就是遍历一遍，找到最小值。但是还可以有更好的方法。考虑用二分查找来降低算法的时间复杂度。

创建两个指针 *left*、*right*，分别指向数组首尾。然后计算出两个指针中间值 *mid*。将 *mid* 与右边界进行比较。

- 如果 $\text{nums}[\text{mid}] > \text{nums}[\text{right}]$ ，则最小值不可能在 *mid* 左侧，一定在 *mid* 右侧，则将 *left* 移动到 *mid* + 1 位置，继续查找右侧区间。
- 如果 $\text{nums}[\text{mid}] < \text{nums}[\text{right}]$ ，则最小值一定在 *mid* 左侧，令右边界 *right* 为 *mid*，继续查找左侧区间。
- 如果 $\text{nums}[\text{mid}] == \text{nums}[\text{right}]$ ，无法判断在 *mid* 的哪一侧，可以采用 $\text{right} = \text{right} - 1$ 逐步缩小区域。

思路 1：代码

```
class Solution:  
    def findMin(self, nums: List[int]) -> int:  
        left = 0  
        right = len(nums) - 1  
        while left < right:  
            mid = left + (right - left) // 2  
            if nums[mid] > nums[right]:  
                left = mid + 1  
            elif nums[mid] < nums[right]:  
                right = mid  
            else:  
                right = right - 1  
        return nums[left]
```

思路 1：复杂度分析

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

0155. 最小栈

- 标签：栈、设计
- 难度：中等

题目链接

- 0155. 最小栈 - 力扣

题目大意

要求：设计一个「栈」。实现 `push`，`pop`，`top`，`getMin` 操作，其中 `getMin` 要求能在常数时间内实现。

说明：

- $-2^{31} \leq val \leq 2^{31} - 1$ 。
- `pop`、`top` 和 `getMin` 操作总是在非空栈上调用
- `push`，`pop`，`top` 和 `getMin` 最多被调用 $3 * 10^4$ 次。

示例：

- 示例 1：

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

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

解释：

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

解题思路

题目要求在常数时间内获取最小值，所以我们不能在 `getMin` 操作时，再去计算栈中的最小值。而是应该在 `push`、`pop` 操作时就已经计算好了最小值。我们有两种思路来解决这道题。

思路 1：辅助栈

使用辅助栈保存当前栈中的最小值。在元素入栈出栈时，两个栈同步保持插入和删除。具体做法如下：

- `push` 操作：当一个元素入栈时，取辅助栈的栈顶存储的最小值，与当前元素进行比较得出最小值，将最小值插入到辅助栈中；该元素也插入到正常栈中。
- `pop` 操作：当一个元素要出栈时，将辅助栈的栈顶元素一起弹出。
- `top` 操作：返回正常栈的栈顶元素值。
- `getMin` 操作：返回辅助栈的栈顶元素值。

思路 1：代码

```
class MinStack:

    def __init__(self):
        self.stack = []
        self.minstack = []

    def push(self, val: int) -> None:
        if not self.stack:
            self.stack.append(val)
            self.minstack.append(val)
        else:
            self.stack.append(val)
            self.minstack.append(min(val, self.minstack[-1])))

    def pop(self) -> None:
        self.stack.pop()
        self.minstack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.minstack[-1]
```

思路 1：复杂度分析

- 时间复杂度： $O(1)$ 。栈的插入、删除、读取操作都是 $O(1)$ 。
- 空间复杂度： $O(n)$ 。其中 n 为总操作数。

思路 2：单个栈

使用单个栈，保存元组：（当前元素值，当前栈内最小值）。具体操作如下：

- `push` 操作：如果栈不为空，则判断当前元素值与栈顶元素所保存的最小值，并更新当前最小值，然后将新元素和当前最小值组成的元组保存到栈中。
- `pop` 操作：正常出栈，即将栈顶元素弹出。
- `top` 操作：返回栈顶元素保存的值。
- `getMin` 操作：返回栈顶元素保存的最小值。

思路 2：代码

```

class MinStack:
    def __init__(self):
        .....
        initialize your data structure here.
        .....
        self.stack = []

    class Node:
        def __init__(self, x):
            self.val = x
            self.min = x

    def push(self, val: int) -> None:
        node = self.Node(val)
        if len(self.stack) == 0:
            self.stack.append(node)
        else:
            topNode = self.stack[-1]
            if node.min > topNode.min:
                node.min = topNode.min

        self.stack.append(node)

    def pop(self) -> None:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1].val

    def getMin(self) -> int:
        return self.stack[-1].min

```

思路 2：复杂度分析

- 时间复杂度： $O(1)$ 。栈的插入、删除、读取操作都是 $O(1)$ 。
- 空间复杂度： $O(n)$ 。其中 n 为总操作数。[# 0159. 至多包含两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

- [# 0159. 至多包含两个不同字符的最长子串 - 力扣](#)

题目大意

给定一个字符串 s ，找出之多包含两个不同字符的最长子串 t ，并返回该子串的长度。

解题思路

使用滑动窗口来求解。

$left$, $right$ 指向字符串开始位置。

不断向右移动 $right$ 指针，使用 $count$ 变量来统计滑动窗口中共有多少个字符，以及使用哈希表来统计当前字符的频数。

当滑动窗口的字符多于 2 个时，向右 移动 $left$ 指针，并减少哈希表中对应原 $left$ 指向字符的频数。

最后使用 max_count 来维护最长子串 t 的长度。

代码

```

import collections
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        max_count = 0
        k = 2
        counts = collections.defaultdict(int)
        count = 0
        left, right = 0, 0
        while right < len(s):
            if counts[s[right]] == 0:
                count += 1
            counts[s[right]] += 1
            right += 1
            if count > k:
                if counts[s[left]] == 1:
                    count -= 1
                counts[s[left]] -= 1
                left += 1
            max_count = max(max_count, right - left)
        return max_count

```

0160. 相交链表

- 标签: 哈希表、链表、双指针
- 难度: 简单

题目链接

- [0160. 相交链表 - 力扣](#)

题目大意

描述: 给定 `listA`、`listB` 两个链表。

要求: 判断两个链表是否相交，返回相交的起始点。如果不相交，则返回 `None`。

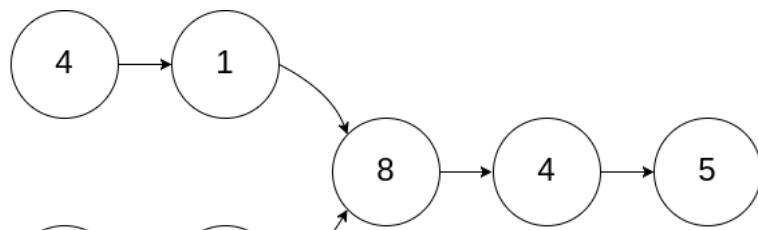
说明:

- `listA` 中节点数目为 m 。
- `listB` 中节点数目为 n 。
- $1 \leq m, n \leq 3 * 10^4$ 。
- $1 \leq Node.val \leq 10^5$ 。
- $0 \leq skipA \leq m$ 。
- $0 \leq skipB \leq n$ 。
- 如果 `listA` 和 `listB` 没有交点，`intersectVal` 为 0。
- 如果 `listA` 和 `listB` 有交点，`intersectVal == listA[skipA] == listB[skipB]`。

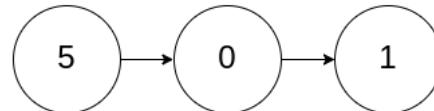
示例:

- 示例 1:

A:



B:



输入: `intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3`

输出: Intersected at '8'

解释: 相交节点的值为 8 (注意, 如果两个链表相交则不能为 0)。

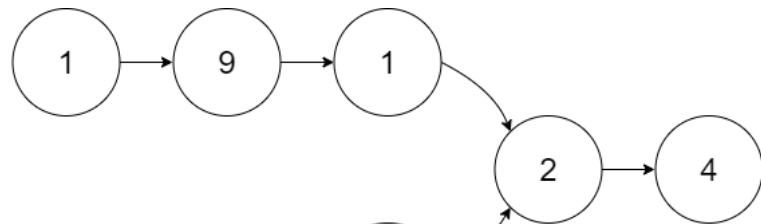
从各自的表头开始算起, 链表 A 为 [4,1,8,4,5], 链表 B 为 [5,6,1,8,4,5]。

在 A 中, 相交节点前有 2 个节点; 在 B 中, 相交节点前有 3 个节点。

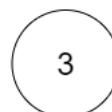
– 请注意相交节点的值不为 1, 因为在链表 A 和链表 B 之中值为 1 的节点 (A 中第二个节点和 B 中第三个节点) 是不同的节点。换句话说, 它们在内存中指向两个不同的位置,

- 示例 2:

A:



B:



输入: `intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1`

输出: Intersected at '2'

解释: 相交节点的值为 2 (注意, 如果两个链表相交则不能为 0)。

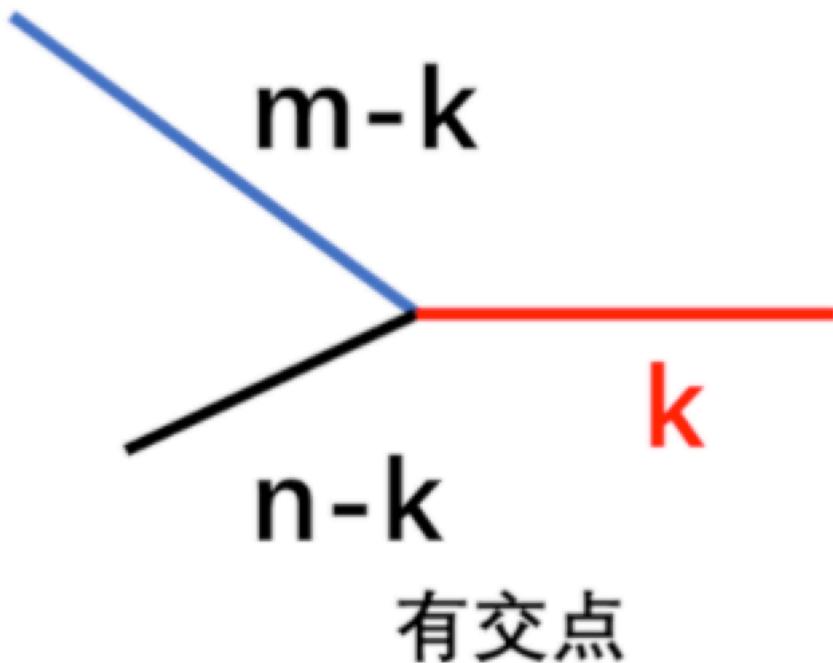
从各自的表头开始算起, 链表 A 为 [1,9,1,2,4], 链表 B 为 [3,2,4]。

在 A 中, 相交节点前有 3 个节点; 在 B 中, 相交节点前有 1 个节点。

解题思路

思路 1：双指针

如果两个链表相交, 那么从相交位置开始, 到结束, 必有一段等长且相同的节点。假设链表 `listA` 的长度为 m 、链表 `listB` 的长度为 n , 他们的相交序列有 k 个, 则相交情况可以如下所示:

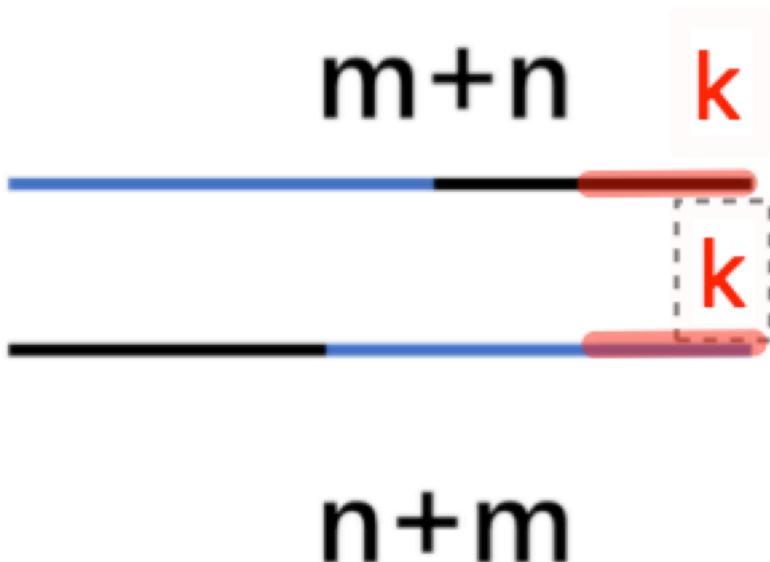


现在问题是如何找到 $m - k$ 或者 $n - k$ 的位置。

考虑将链表 `listA` 的末尾拼接上链表 `listB`，链表 `listB` 的末尾拼接上链表 `listA`。

然后使用两个指针 `pA`、`pB`，分别从链表 `listA`、链表 `listB` 的头节点开始遍历，如果走到共同的节点，则返回该节点。

否则走到两个链表末尾，返回 `None`。



思路 1：代码

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        if headA == None or headB == None:
            return None
        pA = headA
        pB = headB
        while pA != pB:
            pA = pA.next if pA != None else headB
            pB = pB.next if pB != None else headA
        return pA
```

思路 1：复杂度分析

- 时间复杂度: $O(m + n)$ 。
- 空间复杂度: $O(1)$ 。

0162. 寻找峰值

- 标签: 数组、二分查找
- 难度: 中等

题目链接

- [0162. 寻找峰值 - 力扣](#)

题目大意

描述: 给定一个整数数组 `nums`。

要求: 找到峰值元素并返回其索引。必须实现时间复杂度为 $O(\log n)$ 的算法来解决此问题。

说明:

- 峰值元素**: 指其值严格大于左右相邻值的元素。
- 数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。
- 可以假设 $nums[-1] = nums[n] = -\infty$ 。
- $1 \leq nums.length \leq 1000$ 。
- $-2^{31} \leq nums[i] \leq 2^{31} - 1$ 。
- 对于所有有效的 i 都有 $nums[i]! = nums[i + 1]$ 。

示例:

- 示例 1:

```
输入: nums = [1, 2, 3, 1]
输出: 2
解释: 3 是峰值元素，你的函数应该返回其索引 2。
```

- 示例 2:

```
输入: nums = [1, 2, 1, 3, 5, 6, 4]
输出: 1 或 5
解释: 你的函数可以返回索引 1，其峰值元素为 2；或者返回索引 5，其峰值元素为 6。
```

解题思路

思路 1：二分查找

1. 使用两个指针 `left`、`right`。`left` 指向数组第一个元素，`right` 指向数组最后一个元素。
2. 取区间中间节点 `mid`，并比较 `nums[mid]` 和 `nums[mid + 1]` 的值大小。
 - i. 如果 `nums[mid]` 小于 `nums[mid + 1]`，则右侧存在峰值，令 `left = mid + 1`。
 - ii. 如果 `nums[mid]` 大于等于 `nums[mid + 1]`，则左侧存在峰值，令 `right = mid`。
3. 最后，当 `left == right` 时，跳出循环，返回 `left`。

思路 1：代码

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

思路 1：复杂度分析

- 时间复杂度： $O(\log_2 n)$ 。
- 空间复杂度： $O(1)$ 。[# 0164. 最大间距](#)
- 标签：数组、桶排序、基数排序、排序
- 难度：困难

题目链接

- [0164. 最大间距 - 力扣](#)

题目大意

描述：给定一个无序数组 `nums`。

要求：找出数组在排序之后，相邻元素之间最大的差值。如果数组元素个数小于 2，则返回 0。

说明：

- 所有元素都是非负整数，且数值在 32 位有符号整数范围内。
- 请尝试在线性时间复杂度和空间复杂度的条件下解决此问题。

示例：

- **示例 1：**

```
输入: nums = [3,6,9,1]
输出: 3
解释: 排序后的数组是 [1,3,6,9]，其中相邻元素 (3,6) 和 (6,9) 之间都存在最大差值 3。
```

- **示例 2：**

```
输入: nums = [10]
输出: 0
解释: 数组元素个数小于 2, 因此返回 0。
```

解题思路

思路 1：基数排序

这道题的难点在于要求时间复杂度和空间复杂度为 $O(n)$ 。

这道题分为两步：

- 1. 数组排序。
- 2. 计算相邻元素之间的差值。

第 2 步直接遍历数组求解即可，时间复杂度为 $O(n)$ 。所以关键点在于找到一个时间复杂度和空间复杂度为 $O(n)$ 的排序算法。根据题意可知所有元素都是非负整数，且数值在 32 位有符号整数范围内。所以我们可以选择基数排序。基数排序的步骤如下：

- 遍历数组元素，获取数组最大值元素，并取得位数。
- 以个位元素为索引，对数组元素排序。
- 合并数组。
- 之后依次以十位，百位，...，直到最大值元素的最高位处值为索引，进行排序，并合并数组，最终完成排序。

最后，还要注意数组元素个数小于 2 的情况需要特别判断一下。

思路 1：代码

```
class Solution:
    def radixSort(self, arr):
        size = len(str(max(arr)))

        for i in range(size):
            buckets = [[] for _ in range(10)]
            for num in arr:
                buckets[num // (10 ** i) % 10].append(num)
            arr.clear()
            for bucket in buckets:
                for num in bucket:
                    arr.append(num)

        return arr

    def maximumGap(self, nums: List[int]) -> int:
        if len(nums) < 2:
            return 0
        arr = self.radixSort(nums)
        return max(arr[i] - arr[i - 1] for i in range(1, len(arr)))
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。# 0166. 分数到小数
- 标签：哈希表、数学、字符串
- 难度：中等

题目链接

- 0166. 分数到小数 - 力扣

题目大意

给定两个整数，分别表示分数的分子 numerator 和分母 denominator，要求以字符串的形式返回该分数对应小数结果。

- 如果小数部分为循环小数，则将循环的小数部分括在括号内。

解题思路

先处理特殊数据，例如 0、负数等。

然后利用整除运算，计算出分数的整数部分。在根据取余运算结果，判断是否含有小数部分。

因为小数部分可能会有循环部分，所以使用哈希表来判断是否出现了循环小数。哈希表所存键值为 数字：数字开始位置。

然后计算小数部分，每次将被除数 * 10 然后对除数进行整除，再对被除数进行取余操作，直到被除数变为 0，或者在字典中出现了循环小数为止。

代码

```
class Solution:
    def fractionToDecimal(self, numerator: int, denominator: int) -> str:
        if numerator == 0:
            return '0'
        res = []
        if numerator ^ denominator < 0:
            res.append('-')
        numerator, denominator = abs(numerator), abs(denominator)
        res.append(str(numerator // denominator))
        numerator %= denominator
        if numerator == 0:
            return ''.join(res)
        res.append('.')
        record = dict()
        while numerator:
            if numerator not in record:
                record[numerator] = len(res)
                numerator *= 10
                res.append(str(numerator // denominator))
                numerator %= denominator
            else:
                res.insert(record[numerator], '(')
                res.append(')')
                break
        return ''.join(res)
```

0167. 两数之和 II - 输入有序数组

- 标签：数组、双指针、二分查找
- 难度：中等

题目链接

- [0167. 两数之和 II - 输入有序数组 - 力扣](#)

题目大意

描述：给定一个下标从 1 开始计数、升序排列的整数数组：*numbers* 和一个目标值 *target*。

要求：从数组中找出满足相加之和等于 $target$ 的两个数，并返回两个数在数组中下的标值。

说明：

- $2 \leq numbers.length \leq 3 \times 10^4$ 。
- $-1000 \leq numbers[i] \leq 1000$ 。
- $numbers$ 按非递减顺序排列。
- $-1000 \leq target \leq 1000$ 。
- 仅存在一个有效答案。

示例：

- 示例 1：

```
输入: numbers = [2,7,11,15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9。因此 index1 = 1, index2 = 2。返回 [1, 2]。
```

- 示例 2：

```
输入: numbers = [2,3,4], target = 6
输出: [1,3]
解释: 2 与 4 之和等于目标数 6。因此 index1 = 1, index2 = 3。返回 [1, 3]。
```

解题思路

这道题如果暴力遍历数组，从中找到相加之和等于 $target$ 的两个数，时间复杂度为 $O(n^2)$ ，可以尝试一下。

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        size = len(numbers)
        for i in range(size):
            for j in range(i + 1, size):
                if numbers[i] + numbers[j] == target:
                    return [i + 1, j + 1]
        return [-1, -1]
```

结果不出意外的超时了。所以我们要想办法降低时间复杂度。

思路 1：二分查找

因为数组是有序的，可以考虑使用二分查找来减少时间复杂度。具体做法如下：

1. 使用一重循环遍历数组，先固定第一个数，即 $numbers[i]$ 。
2. 然后使用二分查找的方法寻找符合要求的第二个数。
3. 使用两个指针 $left$, $right$ 。 $left$ 指向数组第一个数的下一个数， $right$ 指向数组值最大元素位置。
4. 判断第一个数 $numbers[i]$ 和两个指针中间元素 $numbers[mid]$ 的和与目标值的关系。
 - i. 如果 $numbers[mid] + numbers[i] < target$, 排除掉不可能区间 $[left, mid]$, 在 $[mid + 1, right]$ 中继续搜索。
 - ii. 如果 $numbers[mid] + numbers[i] \geq target$, 则第二个数可能在 $[left, mid]$ 中，则在 $[left, mid]$ 中继续搜索。
5. 直到 $left$ 和 $right$ 移动到相同位置停止检测。如果 $numbers[left] + numbers[i] == target$, 则返回两个元素位置 $[left + 1, i + 1]$ (下标从 1 开始计数)。
6. 如果最终仍没找到，则返回 $[-1, -1]$ 。

思路 1：代码

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        for i in range(len(numbers)):
            left, right = i + 1, len(numbers) - 1
            while left < right:
                mid = left + (right - left) // 2
                if numbers[mid] + numbers[i] < target:
                    left = mid + 1
                else:
                    right = mid
                if numbers[left] + numbers[i] == target:
                    return [i + 1, left + 1]

        return [-1, -1]
```

思路 1：复杂度分析

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

思路 2：对撞指针

可以考虑使用对撞指针来减少时间复杂度。具体做法如下：

1. 使用两个指针 $left$, $right$ 。 $left$ 指向数组第一个值最小的元素位置, $right$ 指向数组值最大元素位置。
2. 判断两个位置上的元素的和与目标值的关系。
 - i. 如果元素和等于目标值, 则返回两个元素位置。
 - ii. 如果元素和大于目标值, 则让 $right$ 左移, 继续检测。
 - iii. 如果元素和小于目标值, 则让 $left$ 右移, 继续检测。
3. 直到 $left$ 和 $right$ 移动到相同位置停止检测。
4. 如果最终仍没找到, 则返回 $[-1, -1]$ 。

思路 2：代码

```
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        left = 0
        right = len(numbers) - 1
        while left < right:
            total = numbers[left] + numbers[right]
            if total == target:
                return [left + 1, right + 1]
            elif total < target:
                left += 1
            else:
                right -= 1
        return [-1, -1]
```

思路 2：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。只用到了常数空间存放若干变量。

0168. Excel表列名称

- 标签: 数学、字符串
- 难度: 简单

题目链接

- [0168. Excel表列名称 - 力扣](#)

题目大意

描述：给定一个正整数 `columnNumber`。

要求：返回它在 Excel 表中相对应的列名称。

1 -> A, 2 -> B, 3 -> C, ..., 26 -> Z, ..., 28 -> AB

解题思路

实质上就是 10 进制转 26 进制。不过映射范围是 1~26，而不是 0~25，如果将 `columnNumber` 直接对 26 取余，则结果为 0~25，而本题余数为 1~26。可以直接将 `columnNumber = columnNumber - 1`，这样就可以将范围变为 0~25 就更加容易判断了。

代码

```
class Solution:
    def convertToTitle(self, columnNumber: int) -> str:
        s = ""
        while columnNumber:
            columnNumber -= 1
            s = chr(65 + columnNumber % 26) + s
            columnNumber //= 26
        return s
```

0169. 多数元素

- 标签：数组、哈希表、分治、计数、排序
- 难度：简单

题目链接

- [0169. 多数元素 - 力扣](#)

题目大意

描述：给定一个大小为 n 的数组 `nums`。

要求：返回其中相同元素个数最多的元素。

说明：

- $n == \text{nums.length}$ 。
- $1 \leq n \leq 5 * 10^4$ 。
- $-10^9 \leq \text{nums}[i] \leq 10^9$ 。

示例：

- 示例 1：

```
输入: nums = [3, 2, 3]
输出: 3
```

- 示例 2:

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

解题思路

思路 1：哈希表

- 遍历数组 `nums`。
- 对于当前元素 `num`，用哈希表统计每个元素 `num` 出现的次数。
- 再遍历一遍哈希表，找出元素个数最多的元素即可。

思路 1：代码

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        numDict = dict()
        for num in nums:
            if num in numDict:
                numDict[num] += 1
            else:
                numDict[num] = 1
        max = float('-inf')
        max_index = -1
        for num in numDict:
            if numDict[num] > max:
                max = numDict[num]
                max_index = num
        return max_index
```

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：分治算法

如果 `num` 是数组 `nums` 的众数，那么我们将 `nums` 分为两部分，则 `num` 至少是其中一部分的众数。

则我们可以用分治法来解决这个问题。具体步骤如下：

- 将数组 `nums` 递归地将当前序列平均分成左右两个数组，直到所有子数组长度为 1。
- 长度为 1 的子数组众数肯定是数组中唯一的数，将其返回即可。
- 将两个子数组依次向上两两合并。
 - 如果两个子数组的众数相同，则说明合并后的数组众数为：两个子数组的众数。
 - 如果两个子数组的众数不同，则需要比较两个众数在整个区间的众数。
- 最后返回整个数组的众数。

思路 2：代码

```

class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        def get_mode(low, high):
            if low == high:
                return nums[low]

            mid = low + (high - low) // 2
            left_mod = get_mode(low, mid)
            right_mod = get_mode(mid + 1, high)

            if left_mod == right_mod:
                return left_mod

            left_mod_cnt, right_mod_cnt = 0, 0
            for i in range(low, high + 1):
                if nums[i] == left_mod:
                    left_mod_cnt += 1
                if nums[i] == right_mod:
                    right_mod_cnt += 1

                if left_mod_cnt > right_mod_cnt:
                    return left_mod
            return right_mod

        return get_mode(0, len(nums) - 1)

```

思路 2：复杂度分析

- 时间复杂度: $O(n \times \log n)$ 。
- 空间复杂度: $O(\log n)$ 。# 0170. 两数之和 III - 数据结构设计
- 标签: 设计、数组、哈希表、双指针、数据流
- 难度: 简单

题目链接

- [0170. 两数之和 III - 数据结构设计 - 力扣](#)

题目大意

设计一个接受整数流的数据结构，使该数据结构支持检查是否存在两数之和等于特定值。

实现 TwoSum 类：

- `TwoSum()`：使用空数组初始化 TwoSum 对象
- `def add(self, number: int) -> None`：向数据结构添加一个数 `number`
- `def find(self, value: int) -> bool`：寻找数据结构中是否存在一对整数，使得两数之和与给定的值 `value` 相等。如果存在，返回 `True`；否则，返回 `False`。

解题思路

使用哈希表存储数组元素值与元素频数的关系。哈希表中键值对信息为 `number: count`。`count` 为 `number` 在数组中的频数。

- `add(number)` 函数中：在哈希表添加 `number` 与其频数之间的关系。
- `find(number)` 函数中：遍历哈希表，对于每个 `number`，检测哈希表中是否存在 `value - number`，如果存在则终止循环并返回结果。
 - 如果 `number == value - number`，则判断哈希表中 `number` 的数目是否大于等于 2。

代码

```

class TwoSum:

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

    def add(self, number: int) -> None:
        """
        Add the number to an internal data structure..
        """
        if number in self.num_counts:
            self.num_counts[number] += 1
        else:
            self.num_counts[number] = 1

    def find(self, value: int) -> bool:
        """
        Find if there exists any pair of numbers which sum is equal to the value.
        """
        for number in self.num_counts.keys():
            number2 = value - number
            if number == number2:
                if self.num_counts[number] > 1:
                    return True
            else:
                if number2 in self.num_counts:
                    return True
        return False

```

0171. Excel 表列序号

- 标签: 数学、字符串
- 难度: 简单

题目链接

- [0171. Excel 表列序号 - 力扣](#)

题目大意

给你一个字符串 `columnTitle`，表示 Excel 表格中的列名称。

要求：返回该列名称对应的列序号。

解题思路

Excel 表的列名称由大写字母组成，共有 26 个，因此列名称的表示实质是 26 进制，需要将 26 进制转换成十进制。转换过程如下：

- 将每一位对应列名称转换成整数（注意列序号从 1 开始）。
- 将当前结果乘上进制数（26），然后累加上当前位上的整数。

最后输出答案。

代码

```
class Solution:
    def titleToNumber(self, columnTitle: str) -> int:
        ans = 0
        for ch in columnTitle:
            num = ord(ch) - ord('A') + 1
            ans = ans * 26 + num
        return ans
```

0172. 阶乘后的零

- 标签: 数学
- 难度: 中等

题目链接

- [0172. 阶乘后的零 - 力扣](#)

题目大意

给定一个整数 n 。

要求: 返回 $n!$ 结果中尾随零的数量。

注意: $0 \leq n \leq 10^4$

解题思路

阶乘中, 末尾 0 的来源只有 $2 * 5$ 。所以尾随 0 的个数为 2 的倍数个数和 5 的倍数个数的最小值。又因为 $2 < 5$, 2 的倍数个数肯定小于等于 5 的倍数, 所以直接统计 5 的倍数个数即可。

代码

```
class Solution:
    def trailingZeroes(self, n: int) -> int:
        count = 0
        while n > 0:
            count += n // 5
            n = n // 5
        return count
```

0173. 二叉搜索树迭代器

- 标签: 栈、树、设计、二叉搜索树、二叉树、迭代器
- 难度: 中等

题目链接

- [0173. 二叉搜索树迭代器 - 力扣](#)

题目大意

要求：实现一个二叉搜索树的迭代器 BSTIterator。表示一个按中序遍历二叉搜索树（BST）的迭代器：

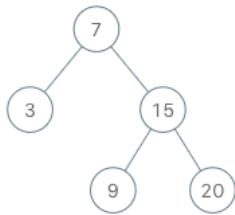
- `def __init__(self, root: TreeNode):` : 初始化 BSTIterator 类的一个对象，会给出二叉搜索树的根节点。
- `def hasNext(self) -> bool:` : 如果向右指针遍历存在数字，则返回 True，否则返回 False。
- `def next(self) -> int:` : 将指针向右移动，返回指针处的数字。

说明：

- 指针初始化为一个不存在于 BST 中的数字，所以对 `next()` 的首次调用将返回 BST 中的最小元素。
- 可以假设 `next()` 调用总是有效的，也就是说，当调用 `next()` 时，BST 的中序遍历中至少存在一个下一个数字。
- 树中节点的数目在范围 $[1, 10^5]$ 内。
- $0 \leq Node.val \leq 10^6$ 。
- 最多调用 10^5 次 `hasNext` 和 `next` 操作。
- 进阶：设计一个满足下述条件的解决方案，`next()` 和 `hasNext()` 操作均摊时间复杂度为 $O(1)$ ，并使用 $O(h)$ 内存。其中 h 是树的高度。

示例：

- 示例 1：



```

输入
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], []]
输出
[null, 3, 7, true, 9, true, 15, true, 20, false]
  
```

解题思路

思路 1：中序遍历二叉搜索树

中序遍历的顺序是：左、根、右。我们使用一个栈来保存节点，以便于迭代的时候取出对应节点。

- 初始的遍历当前节点的左子树，将其路径上的节点存储到栈中。
- 调用 `next` 方法的时候，从栈顶取出节点，因为之前已经将路径上的左子树全部存入了栈中，所以此时该节点的左子树为空，这时候取出节点右子树，再将右子树的左子树进行递归遍历，并将其路径上的节点存储到栈中。
- 调用 `hasNext` 的方法的时候，直接判断栈中是否有值即可。

思路 1：代码

```
class BSTIterator:

    def __init__(self, root: TreeNode):
        self.stack = []
        self.in_order(root)

    def in_order(self, node):
        while node:
            self.stack.append(node)
            node = node.left

    def next(self) -> int:
        node = self.stack.pop()
        if node.right:
            self.in_order(node.right)
        return node.val

    def hasNext(self) -> bool:
        return len(self.stack) != 0
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为树中节点数量。
- 空间复杂度: $O(n)$ 。

0179. 最大数

- 标签: 贪心、数组、字符串、排序
- 难度: 中等

题目链接

- [0179. 最大数 - 力扣](#)

题目大意

描述: 给定一个非负整数数组 `nums`。

要求: 重新排列数组中每个数的顺序, 使之将数组中所有数字按顺序拼接起来所组成的整数最大。

说明:

- $1 \leq \text{nums.length} \leq 100$ 。
- $0 \leq \text{nums}[i] \leq 10^9$ 。

示例:

- 示例 1:

```
输入: nums = [10,2]
输出: "210"
```

- 示例 2:

```
输入: nums = [3,30,34,5,9]
输出: "9534330"
```

解题思路

思路 1：排序

本质上是给数组进行排序。假设 `x`、`y` 是数组 `nums` 中的两个元素。如果拼接字符串 `x + y < y + x`，则 `y > x`。`y` 应该排在 `x` 前面。反之，则 `y < x`。

按照上述规则，对原数组进行排序即可。这里我们使用了 `functools.cmp_to_key` 自定义排序函数。

思路 1：代码

```
import functools

class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        def cmp(a, b):
            if a + b == b + a:
                return 0
            elif a + b > b + a:
                return 1
            else:
                return -1
        nums_s = list(map(str, nums))
        nums_s.sort(key=functools.cmp_to_key(cmp), reverse=True)
        return str(int(''.join(nums_s)))
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。其中 n 是给定数组 `nums` 的大小。
- 空间复杂度： $O(n)$ 。[# 0188. 买卖股票的最佳时机 IV](#)
- 标签：数组、动态规划
- 难度：困难

题目链接

- [# 0188. 买卖股票的最佳时机 IV - 力扣](#)

题目大意

给定一个数组 `prices` 代表一只股票，其中 `prices[i]` 代表这只股票第 `i` 天的价格。再给定一个整数 `k`，表示最多可完成 `k` 笔交易，且不能同时参与多笔交易（必须在再次购买前出售掉之前的股票）。

现在要求：计算所能获取的最大利润。

解题思路

动态规划求解。这道题是「[0123. 买卖股票的最佳时机 III](#)」的升级版，不过思路一样

最多可完成两笔交易意味着总共有三种情况：买卖一次，买卖两次，不买卖。

具体到每一天结束总共有 `2 * k + 1` 种状态：

0. 未进行买卖状态；
1. 第 1 次买入状态；
2. 第 1 次卖出状态；
3. 第 2 次买入状态；
4. 第 2 次卖出状态。
5. ...
6. 第 `m` 次买入状态。

7. 第 m 次卖出状态。

因为买入、卖出为两种状态，干脆我们直接让偶数序号表示买入状态，奇数序号表示卖出状态。

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

注意：这里第 j 种情况，并不一定是这一天一定要买入或卖出，而是这一天所处的买入卖出状态。比如说前一天是第一次买入，第二天没有操作，则第二天就沿用前一天的第一次买入状态。

接下来确定状态转移公式：

- 第 0 种状态下显然利润为 0 ，可以直接赋值为昨天获取的最大利润，即 $dp[i][0] = dp[i - 1][0]$ 。
- 第 1 次买入状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天买入状态所得的最大利润： $dp[i][1] = dp[i - 1][1]$ 。
 - 第 1 次买入： $dp[i][1] = dp[i - 1][0] - prices[i]$ 。
- 第 1 次卖出状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天卖出状态所得的最大利润： $dp[i][2] = dp[i - 1][2]$ 。
 - 第 1 次卖出： $dp[i][2] = dp[i - 1][1] + prices[i]$ 。
- 第 2 次买入状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天买入状态所得的最大利润： $dp[i][3] = dp[i - 1][3]$ 。
 - 第 2 次买入： $dp[i][3] = dp[i - 1][2] - prices[i]$ 。
- 第 2 次卖出状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天卖出状态所得的最大利润： $dp[i][4] = dp[i - 1][4]$ 。
 - 第 2 次卖出： $dp[i][4] = dp[i - 1][3] + prices[i]$ 。
- ...
 - 第 m 次 ($j = 2 * m$) 买入状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天卖出状态所得的最大利润： $dp[i][j] = dp[i - 1][j]$ 。
 - 第 m 次买入： $dp[i][j] = dp[i - 1][j - 1] - prices[i]$ 。
 - 第 m 次 ($j = 2 * m + 1$) 卖出状态下可以有两种状态推出，取最大的那一种赋值：
 - 不做任何操作，直接沿用前一天卖出状态所得的最大利润： $dp[i][j] = dp[i - 1][j]$ 。
 - 第 m 次卖出： $dp[i][j] = dp[i - 1][j - 1] + prices[i]$ 。

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 m 次买入 ($j = 2 * m$) 就是 $dp[0][j] = -prices[i]$ 。

第 m 次 ($j = 2 * m + 1$) 卖出的话，可以视作为没有盈利（当天买卖，价格没有变化），即 $dp[0][j] = 0$ 。

在递推结束后，最大利润肯定是无操作、第 m 次卖出这几种种情况里边，且为最大值。我们在维护的时候维护的是最大值，则第 m 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 $m - 1$ 笔交易，那么在转移状态时，我们允许在一天内进行多次交易，则 $m - 1$ 笔交易的状态可以转移至 m 笔交易，最终都可以转移至 k 比交易。

所以最终答案为 $dp[size - 1][2 * k]$ 。 $size$ 为股票天数。

代码

```

class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        size = len(prices)
        if size == 0:
            return 0

        dp = [[0 for _ in range(2 * k + 1)] for _ in range(size)]

        for j in range(1, 2 * k, 2):
            dp[0][j] = -prices[0]

        for i in range(1, size):
            for j in range(1, 2 * k + 1):
                if j % 2 == 1:
                    dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - 1] - prices[i])
                else:
                    dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - 1] + prices[i])
        return dp[size - 1][2 * k]

```

0189. 轮转数组

- 标签：数组、数学、双指针
- 难度：中等

题目链接

- [0189. 轮转数组 - 力扣](#)

题目大意

描述：给定一个数组 $nums$, 再给定一个数字 k 。

要求：将数组中的元素向右移动 k 个位置。

说明：

- $1 \leq nums.length \leq 10^5$ 。
- $-2^{31} \leq nums[i] \leq 2^{31} - 1$ 。
- $0 \leq k \leq 10^5$ 。
- 使用空间复杂度为 $O(1)$ 的原地算法解决这个问题。

示例：

- **示例 1：**

```

输入: nums = [1,2,3,4,5,6,7], k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右轮转 1 步: [7,1,2,3,4,5,6]
向右轮转 2 步: [6,7,1,2,3,4,5]
向右轮转 3 步: [5,6,7,1,2,3,4]

```

- **示例 2：**

```

输入: nums = [-1,-100,3,99], k = 2
输出: [3,99,-1,-100]
解释:
向右轮转 1 步: [99,-1,-100,3]
向右轮转 2 步: [3,99,-1,-100]

```

解题思路

思路 1：数组翻转

可以用一个新数组，先保存原数组的后 k 个元素，再保存原数组的前 $n - k$ 个元素。但题目要求不使用额外的数组空间，那么就需要在原数组上做操作。

我们可以先把整个数组翻转一下，这样后半段元素就到了前边，前半段元素就到了后边，只不过元素顺序是反着的。我们再从 k 位置分隔开，将 $[0...k - 1]$ 区间上的元素和 $[k...n - 1]$ 区间上的元素再翻转一下，就得到了最终结果。

具体步骤：

1. 将数组 $[0, n - 1]$ 位置上的元素全部翻转。
2. 将数组 $[0, k - 1]$ 位置上的元素进行翻转。
3. 将数组 $[k, n - 1]$ 位置上的元素进行翻转。

思路 1：代码

```

class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        n = len(nums)
        k = k % n
        self.reverse(nums, 0, n-1)
        self.reverse(nums, 0, k-1)
        self.reverse(nums, k, n-1)
    def reverse(self, nums: List[int], left: int, right: int) -> None:
        while left < right :
            tmp = nums[left]
            nums[left] = nums[right]
            nums[right] = tmp
            left += 1
            right -= 1

```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。翻转的时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(1)$ 。[# 0190. 颠倒二进制位](#)
- 标签：位运算、分治
- 难度：简单

题目链接

- [0190. 颠倒二进制位 - 力扣](#)

题目大意

描述：给定一个 32 位无符号整数 n 。

要求：将 n 所有二进位进行翻转，并返回翻转后的整数。

说明：

- 输入是一个长度为 32 的二进制字符串。

示例：

- 示例 1：

```
输入: n = 0000001010010100000111010011100
输出: 964176192 (0011100101111000001010010100000)
解释: 输入的二进制串 0000001010010100000111010011100 表示无符号整数 43261596,
因此返回 964176192, 其二进制表示形式为 0011100101111000001010010100000。
```

- 示例 2：

```
输入: n = 111111111111111111111111111111101
输出: 3221225471 (10111111111111111111111111111111)
解释: 输入的二进制串 111111111111111111111111111111101 表示无符号整数 4294967293,
因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111。
```

解题思路

思路 1：逐位翻转

- 用一个变量 `res` 存储翻转后的结果。
- 将 `n` 不断进行右移（即 `n >> 1`），从低位到高位进行枚举，此时 `n` 的最低位就是我们枚举的二进位。
- 同时 `res` 不断左移（即 `res << 1`），并将当前枚举的二进位翻转后的结果（即 `n & 1`）拼接到 `res` 的末尾（即 `(res << 1) | (n & 1)`）。

思路 1：代码

```
class Solution:
    def reverseBits(self, n: int) -> int:
        res = 0
        for i in range(32):
            res = (res << 1) | (n & 1)
            n >>= 1
        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(\log n)$ 。
- 空间复杂度： $O(1)$ 。# 0191. 位1的个数
- 标签：位运算、分治
- 难度：简单

题目链接

- 0191. 位1的个数 - 力扣

题目大意

描述：给定一个无符号整数 `n`。

要求：统计其对应二进制表达式中 1 的个数。

说明：

- 输入必须是长度为 32 的二进制串。

示例：

- 示例 1：

```
输入: n = 000000000000000000000000000000001011
输出: 3
解释: 输入的二进制串 000000000000000000000000000000001011 中, 共有三位为 '1'。
```

- 示例 2:

```
输入: n = 0000000000000000000000000000000010000000
输出: 1
解释: 输入的二进制串 0000000000000000000000000000000010000000 中, 共有一位为 '1'。
```

解题思路

思路 1：循环按位计算

- 对整数 n 的每一位进行按位与运算，并统计结果。

思路 1：代码

```
class Solution:
    def hammingWeight(self, n: int) -> int:
        ans = 0
        while n:
            ans += (n & 1)
            n = n >> 1
        return ans
```

思路 1：复杂度分析

- 时间复杂度: $O(k)$, 其中 k 是二进位的位数, $k = 32$ 。
- 空间复杂度: $O(1)$ 。

思路 2：改进位运算

利用 $n \& (n - 1)$ 。这个运算刚好可以将 n 的二进制中最末位的 1 变为 0。

比如 $n = 6$ 时, $6 = 110_{(2)}$, $6 - 1 = 101_{(2)}$, $110 \& 101 = 100$ 。

利用这个位运算，不断的将 n 中最末位的 1 变为 0，直到 n 变为 0 即可，其变换次数就是我们要求的结果。

思路 2：代码

```
class Solution:
    def hammingWeight(self, n: int) -> int:
        ans = 0
        while n:
            n = n & (n - 1)
            ans += 1
        return ans
```

思路 2：复杂度分析

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

0198. 打家劫舍

- 标签: 数组、动态规划

- 难度：中等

题目链接

- [0198. 打家劫舍 - 力扣](#)

题目大意

描述：给定一个数组 $nums$, $nums[i]$ 代表第 i 间房屋存放的金额。相邻的房屋装有防盗系统，假如相邻的两间房屋同时被偷，系统就会报警。

要求：假如你是一名专业的小偷，计算在不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

说明：

- $1 \leq nums.length \leq 100$ 。
- $0 \leq nums[i] \leq 400$ 。

示例：

- 示例 1：

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

- 示例 2：

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

解题思路

思路 1：动态规划

1. 划分阶段

按照房屋序号进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：前 i 间房屋所能偷窃到的最高金额。

3. 状态转移方程

i 间房屋的最后一个房子是 $nums[i - 1]$ 。

如果房屋数大于等于 2 间，则偷窃第 $i - 1$ 间房屋的时候，就有两种状态：

- 偷窃第 $i - 1$ 间房屋，那么第 $i - 2$ 间房屋就不能偷窃了，偷窃的最高金额为：前 $i - 2$ 间房屋的最高总金额 + 第 $i - 1$ 间房屋的金额，即 $dp[i] = dp[i - 2] + nums[i - 1]$ ；
- 不偷窃第 $i - 1$ 间房屋，那么第 $i - 2$ 间房屋可以偷窃，偷窃的最高金额为：前 $i - 1$ 间房屋的最高总金额，即 $dp[i] = dp[i - 1]$ 。

然后这两种状态取最大值即可，即状态转移方程为：

$$dp[i] = \begin{cases} nums[0] & i = 1 \\ \max(dp[i - 2] + nums[i - 1], dp[i - 1]) & i \geq 2 \end{cases}$$

4. 初始条件

- 前 0 间房屋所能偷窃到的最高金额为 0，即 $dp[0] = 0$ 。
- 前 1 间房屋所能偷窃到的最高金额为 $nums[0]$ ，即： $dp[1] = nums[0]$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：前 i 间房屋所能偷窃到的最高金额。则最终结果为 $dp[size]$ ， $size$ 为总的房屋数。

思路 1：代码

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

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

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

        return dp[size]
```

思路 1：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 $O(n)$ 。

0199. 二叉树的右视图

- 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

- [0199. 二叉树的右视图 - 力扣](#)

题目大意

描述：给定一棵二叉树的根节点 `root`。

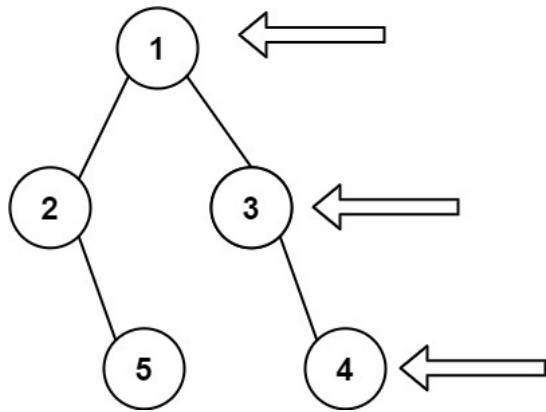
要求：按照从顶部到底部的顺序，返回从右侧能看到的节点值。

说明：

- 二叉树的节点个数的范围是 $[0, 100]$ 。
- $-100 \leq Node.val \leq 100$ 。

示例：

- 示例 1：



输入: [1,2,3,null,5,null,4]
输出: [1,3,4]

- 示例 2:

输入: [1,null,3]
输出: [1,3]

解题思路

思路 1：广度优先搜索

使用广度优先搜索对二叉树进行层次遍历。在遍历每层节点的时候，只需要将最后一个节点加入结果数组即可。

思路 1：代码

```

class Solution:
    def rightSideView(self, root: TreeNode) -> List[int]:
        if not root:
            return []
        queue = [root]
        order = []
        while queue:
            size = len(queue)
            for i in range(size):
                curr = queue.pop(0)
                if curr.left:
                    queue.append(curr.left)
                if curr.right:
                    queue.append(curr.right)
                if i == size - 1:
                    order.append(curr.val)
        return order
  
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ ，其中 n 是二叉树的节点数目。
- 空间复杂度: $O(n)$ 。递归函数需要用到栈空间，栈空间取决于递归深度，最坏情况下递归深度为 n ，所以空间复杂度为 $O(n)$ 。

0200. 岛屿数量

- 标签: 深度优先搜索、广度优先搜索、并查集、数组、矩阵
- 难度: 中等

题目链接

- 0200. 岛屿数量 - 力扣

题目大意

描述：给定一个由字符 '`1`' (陆地) 和字符 '`0`' (水) 组成的二维网格 `grid`。

要求：计算网格中岛屿的数量。

说明：

- 岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。
- 此外，你可以假设该网格的四条边均被水包围。
- $m == \text{grid.length}$ 。
- $n == \text{grid}[i].length$ 。
- $1 \leq m, n \leq 300$ 。
- $\text{grid}[i][j]$ 的值为 '`0`' 或 '`1`'。

示例：

- 示例 1：

```
输入: grid = [
    ["1", "1", "1", "1", "0"],
    ["1", "1", "0", "1", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "0", "0", "0"]
]
输出: 1
```

- 示例 2：

```
输入: grid = [
    ["1", "1", "0", "0", "0"],
    ["1", "1", "0", "0", "0"],
    ["0", "0", "1", "0", "0"],
    ["0", "0", "0", "1", "1"]
]
输出: 3
```

解题思路

如果把上下左右相邻的字符 '`1`' 看做是 1 个连通块，这道题的目的就是求解一共有多少个连通块。

使用深度优先搜索或者广度优先搜索都可以。

思路 1：深度优先搜索

1. 遍历 `grid`。
2. 对于每一个字符为 '`1`' 的元素，遍历其上下左右四个方向，并将该字符置为 '`0`'，保证下次不会被重复遍历。
3. 如果超出边界，则返回 0。
4. 对于 (i, j) 位置的元素来说，递归遍历的位置就是 $(i - 1, j)$ 、 $(i, j - 1)$ 、 $(i + 1, j)$ 、 $(i, j + 1)$ 四个方向。每次遍历到底，统计数记录一次。
5. 最终统计出深度优先搜索的次数就是我们要求的岛屿数量。

思路 1：代码

```

class Solution:
    def dfs(self, grid, i, j):
        n = len(grid)
        m = len(grid[0])
        if i < 0 or i >= n or j < 0 or j >= m or grid[i][j] == '0':
            return 0
        grid[i][j] = '0'
        self.dfs(grid, i + 1, j)
        self.dfs(grid, i, j + 1)
        self.dfs(grid, i - 1, j)
        self.dfs(grid, i, j - 1)

    def numIslands(self, grid: List[List[str]]) -> int:
        count = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    self.dfs(grid, i, j)
                    count += 1
        return count

```

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: $O(m \times n)$ 。

0201. 数字范围按位与

- 标签: 位运算
- 难度: 中等

题目链接

- [0201. 数字范围按位与 - 力扣](#)

题目大意

描述: 给定两个整数 $left$ 和 $right$, 表示区间 $[left, right]$ 。

要求: 返回此区间内所有数字按位与的结果 (包含 $left$ 、 $right$ 端点)。

说明:

- $0 \leq left \leq right \leq 2^{31} - 1$ 。

示例:

- 示例 1:

```

输入: left = 5, right = 7
输出: 4

```

- 示例 2:

```

输入: left = 1, right = 2147483647
输出: 0

```

解题思路

思路 1：位运算

很容易想到枚举算法：对于区间 $[left, right]$ ，如果使用枚举算法，对区间范围内的数依次进行按位与操作，最后输出结果。

但是枚举算法在区间范围很大的时候会超时，所以我们应该换个思路来解决这道题。

我们知道与运算的规则如下：

- $0 \& 0 == 0$
- $0 \& 1 == 0$
- $1 \& 0 == 0$
- $1 \& 1 == 1$ 。

只有对应位置上都为 1 的情况下，按位与才能得到 1。而对应位置上只要出现 0，则该位置上最终的按位与结果一定为 0。

那么我们可以先来求一下区间所有数对应二进制的公共前缀，假设这个前缀的长度为 x 。

公共前缀部分因为每个位置上的二进制值完全一样，所以按位与的结果也相同。

接下来考虑除了公共前缀的剩余的二进制位部分。

这时候剩余部分有两种情况：

- $x = 31$ 。则 $left == right$ ，其按位与结果就是 $left$ 本身。
- $0 \leq x < 31$ 。这种情况下因为 $left < right$ ，所以 $left$ 的第 $x + 1$ 位必然为 0， $right$ 的第 $x + 1$ 位必然为 1。
 - 注意： $left$ 、 $right$ 第 $x + 1$ 位上不可能同为 0 或 1，这样就是公共前缀了。
 - 注意：同样不可能是 $left$ 第 $x + 1$ 位为 1， $right$ 第 $x + 1$ 位为 0，这样就是 $left > right$ 了。

而从第 $x + 1$ 位起，从 $left$ 到 $right$ 。肯定会经过 10000... 的位置，从而使得除了公共前缀的剩余部分（后面的 $31 - x$ 位）的按位与结果一定为 0。

举个例子， $x = 27$ ，则除了公共前缀的剩余部分长度为 4。则剩余部分从 0XXX 到 1XXX 必然会经过 1000，则剩余部分的按位与结果为 0000。

那么这道题就转变为了求 $[left, right]$ 区间范围内所有数的二进制公共前缀，然后在后缀位置上补上 0。

求解公共前缀，我们借助于 Brian Kernighan 算法中的 $n \& (n - 1)$ 公式来计算。

- $n \& (n - 1)$ 公式：对 n 和 $n - 1$ 进行按位与运算后， n 最右边的 1 会变成 0，也就是清除了 n 对应二进制的最右侧的 1。比如 $n = 10110100_{(2)}$ ，进行 $n \& (n - 1)$ 操作之后，就变为了 $n = 10110000_{(2)}$ 。

具体计算步骤如下：

1. 对于给定的区间范围 $[left, right]$ ，对 $right$ 进行 $right \& (right - 1)$ 迭代。
2. 直到 $right$ 小于等于 $left$ ，此时区间内非公共前缀的 1 均变为了 0。
3. 最后输出 $right$ 作为答案。

思路 1：位运算代码

```
class Solution:
    def rangeBitwiseAnd(self, left: int, right: int) -> int:
        while left < right:
            right = right & (right - 1)
        return right
```

思路 1：复杂度分析

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

参考资料

- 【题解】巨好理解的位运算思路 - 数字范围按位与 - 力扣

0202. 快乐数

- 标签: 哈希表、数学、双指针
- 难度: 简单

题目链接

- [0202. 快乐数 - 力扣](#)

题目大意

描述: 给定一个整数 n 。

要求: 判断 n 是否为快乐数。

说明:

- 快乐数定义:
 - 对于一个正整数, 每一次将该数替换为它每个位置上的数字的平方和。
 - 然后重复这个过程直到这个数变为 1, 也可能是 无限循环 但始终变不到 1。
 - 如果 可以变为 1, 那么这个数就是快乐数。
- $1 \leq n \leq 2^{31} - 1$ 。

示例:

- 示例 1:

```
输入: n = 19
输出: True
解释:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1
```

- 示例 2:

```
输入: n = 2
输出: False
```

解题思路

思路 1: 哈希表 / 集合

根据题意, 不断重复操作, 数可能变为 1, 也可能是无限循环。无限循环其实就相当于链表形成了闭环, 可以用哈希表来存储为一位生成的数, 每次判断该数是否存在于哈希表中。如果已经出现在哈希表里, 则说明进入了无限循环, 该数就不是快乐数。如果没有出现则将该数加入到哈希表中, 进行下一次计算。不断重复这个过程, 直到形成闭环或者变为 1。

思路 1：代码

```

class Solution:
    def getNext(self, n: int):
        total_sum = 0
        while n > 0:
            n, digit = divmod(n, 10)
            total_sum += digit ** 2
        return total_sum

    def isHappy(self, n: int) -> bool:
        num_set = set()
        while n != 1 and n not in num_set:
            num_set.add(n)
            n = self.getNext(n)
        return n == 1

```

思路 1：复杂度分析

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

0203. 移除链表元素

- 标签: 递归、链表
- 难度: 简单

题目链接

- [0203. 移除链表元素 - 力扣](#)

题目大意

描述: 给定一个链表的头节点 `head` 和一个值 `val`。

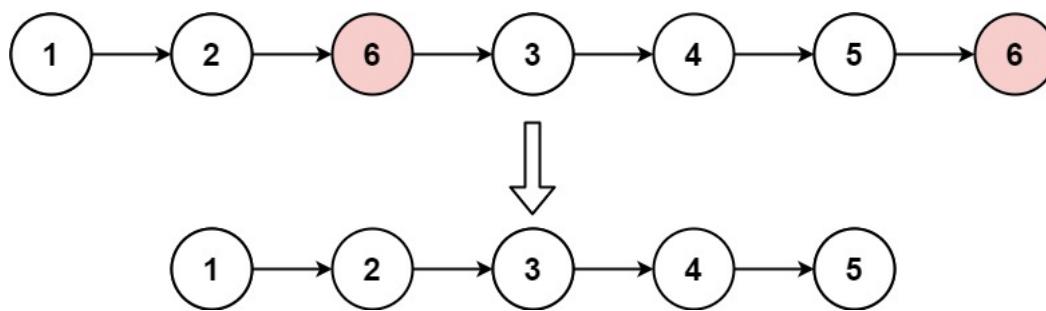
要求: 删去链表中值为 `val` 的节点, 并返回新的链表头节点。

说明:

- 列表中的节点数目在范围 $[0, 10^4]$ 内。
- $1 \leq Node.val \leq 50$ 。
- $0 \leq val \leq 50$ 。

示例:

- 示例 1:



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

- 示例 2:

输入: head = [], val = 1
 输出: []

解题思路

思路 1：迭代

- 使用两个指针 `prev` 和 `curr`。`prev` 指向前一节点和当前节点，`curr` 指向当前节点。
- 从前向后遍历链表，遇到值为 `val` 的节点时，将 `prev` 的 `next` 指针指向当前节点的下一个节点，继续递归遍历。没有遇到则将 `prev` 指针向后移动一步。
- 向右移动 `curr`，继续遍历。

需要注意的是：因为要删除的节点可能包含了头节点，我们可以考虑在遍历之前，新建一个头节点，让其指向原来的头节点。这样，最终如果删除的是头节点，则直接删除原头节点，然后最后返回新建头节点的下一个节点即可。

思路 1：代码

```
class Solution:
    def removeElements(self, head: ListNode, val: int) -> ListNode:
        newHead = ListNode(0, head)
        newHead.next = head

        prev, curr = newHead, head
        while curr:
            if curr.val == val:
                prev.next = curr.next
            else:
                prev = curr
            curr = curr.next
        return newHead.next
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。

0204. 计数质数

- 标签: 数组、数学、枚举、数论
- 难度: 中等

题目链接

- [0204. 计数质数 - 力扣](#)

题目大意

描述: 给定一个非负整数 n 。

要求: 统计小于 n 的质数数量。

说明：

- $0 \leq n \leq 5 * 10^6$ 。

示例：

- 示例 1：

```
输入: n = 10
输出: 4
解释: 小于 10 的质数一共有 4 个，它们是 2, 3, 5, 7。
```

- 示例 2：

```
输入: n = 1
输出: 0
```

解题思路

思路 1：枚举算法（超时）

对于小于 n 的每一个数 x ，我们可以枚举区间 $[2, x - 1]$ 上的数是否是 x 的因数，即是否存在能被 x 整数的数。如果存在，则该数 x 不是质数。如果不存，在，则该数 x 是质数。

这样我们就可以通过枚举 $[2, n - 1]$ 上的所有数 x ，并判断 x 是否为质数。

在遍历枚举的同时，我们维护一个用于统计小于 n 的质数数量的变量 `cnt`。如果符合要求，则将计数 `cnt` 加 1。最终返回该数目作为答案。

考虑到如果 i 是 x 的因数，则 $\frac{x}{i}$ 也必然是 x 的因数，则我们只需要检验这两个因数中的较小数即可。而较小数一定会落在 $[2, \sqrt{x}]$ 上。因此我们在检验 x 是否为质数时，只需要枚举 $[2, \sqrt{x}]$ 中的所有数即可。

利用枚举算法单次检查单个数的时间复杂度为 $O(\sqrt{n})$ ，检查 n 个数的整体时间复杂度为 $O(n\sqrt{n})$ 。

思路 1：代码

```
class Solution:
    def isPrime(self, x):
        for i in range(2, int(pow(x, 0.5)) + 1):
            if x % i == 0:
                return False
        return True

    def countPrimes(self, n: int) -> int:
        cnt = 0
        for x in range(2, n):
            if self.isPrime(x):
                cnt += 1
        return cnt
```

思路 1：复杂度分析

- 时间复杂度： $O(n \times \sqrt{n})$ 。
- 空间复杂度： $O(1)$ 。

思路 2：埃氏筛法

可以用「埃氏筛」进行求解。这种方法是由古希腊数学家埃拉托斯尼提出的，具体步骤如下：

- 使用长度为 n 的数组 `is_prime` 来判断一个数是否是质数。如果 `is_prime[i] == True`，则表示 i 是质数，如果 `is_prime[i] == False`，则表示 i 不是质数。并使用变量 `count` 标记质数个数。
- 然后从 $[2, n - 1]$ 的第一个质数（即数字 2）开始，令 `count` 加 1，并将该质数在 $[2, n - 1]$ 范围内所有倍数（即 4、6、8、...）都标记为非质数。

- 然后根据数组 `is_prime` 中的信息，找到下一个没有标记为非质数的质数（即数字 3），令 `count` 加 1，然后将该质数在 $[2, n - 1]$ 范围内的所有倍数（即 6、9、12、...）都标记为非质数。
- 以此类推，直到所有小于或等于 $n - 1$ 的质数和质数的倍数都标记完毕时，输出 `count`。

优化：对于一个质数 x ，我们可以直接从 $x \times x$ 开始标记，这是因为 $2 \times x$ 、 $3 \times x$ 、... 这些数已经在 x 之前就被其他数的倍数标记过了，例如 2 的所有倍数、3 的所有倍数等等。

思路 2：代码

```
class Solution:
    def countPrimes(self, n: int) -> int:
        is_prime = [True] * n
        count = 0
        for i in range(2, n):
            if is_prime[i]:
                count += 1
                for j in range(i * i, n, i):
                    is_prime[j] = False
        return count
```

思路 2：复杂度分析

- 时间复杂度： $O(n \times \log_2 \log_2 n)$ 。
- 空间复杂度： $O(n)$ 。

0205. 同构字符串

- 标签：哈希表、字符串
- 难度：简单

题目链接

- [0205. 同构字符串 - 力扣](#)

题目大意

描述：给定两个字符串 s 和 t 。

要求：判断字符串 s 和 t 是否是同构字符串。

说明：

- 同构字符串**：如果 s 中的字符可以按某种映射关系替换得到 t 相同位置上的字符，那么两个字符串是同构的。
- 每个字符都应当映射到另一个字符，且不改变字符顺序。不同字符不能映射到统一字符上，相同字符只能映射到同一个字符上，字符可以映射到自己本身。
- $1 \leq s.length \leq 5 \times 10^4$ 。
- $t.length == s.length$ 。
- s 和 t 由任意有效的 ASCII 字符组成。

示例：

- 示例 1：

```
输入: s = "egg", t = "add"
输出: True
```

- 示例 2：

输入: `s = "foo", t = "bar"`
 输出: `False`

解题思路

思路 1：哈希表

根据题目意思，字符串 s 和 t 每个位置上的字符是一一对应的。 s 的每个字符都与 t 对应位置上的字符对应。可以考虑用哈希表来存储 $s[i] : t[i]$ 的对应关系。但是这样不能只能保证对应位置上的字符是对应的，但不能保证是唯一对应的。所以还需要另一个哈希表来存储 $t[i] : s[i]$ 的对应关系来判断是否是唯一对应的。

思路 1：代码

```
class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:
        s_dict = dict()
        t_dict = dict()
        for i in range(len(s)):
            if s[i] in s_dict and s_dict[s[i]] != t[i]:
                return False
            if t[i] in t_dict and t_dict[t[i]] != s[i]:
                return False
            s_dict[s[i]] = t[i]
            t_dict[t[i]] = s[i]
        return True
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为字符串长度。
- 空间复杂度: $O(|S|)$, 其中 S 是字符串字符集。

0206. 反转链表

- 标签: 递归、链表
- 难度: 简单

题目链接

- [0206. 反转链表 - 力扣](#)

题目大意

描述: 给定一个单链表的头节点 `head`。

要求: 将该单链表进行反转。可以迭代或递归地反转链表。

说明:

- 链表中节点的数目范围是 $[0, 5000]$ 。
- $-5000 \leq Node.val \leq 5000$ 。

示例:

- 示例 1:

```

输入: head = [1,2,3,4,5]
输出: [5,4,3,2,1]
解释:
翻转前  1->2->3->4->5->NULL
反转后  5->4->3->2->1->NULL

```

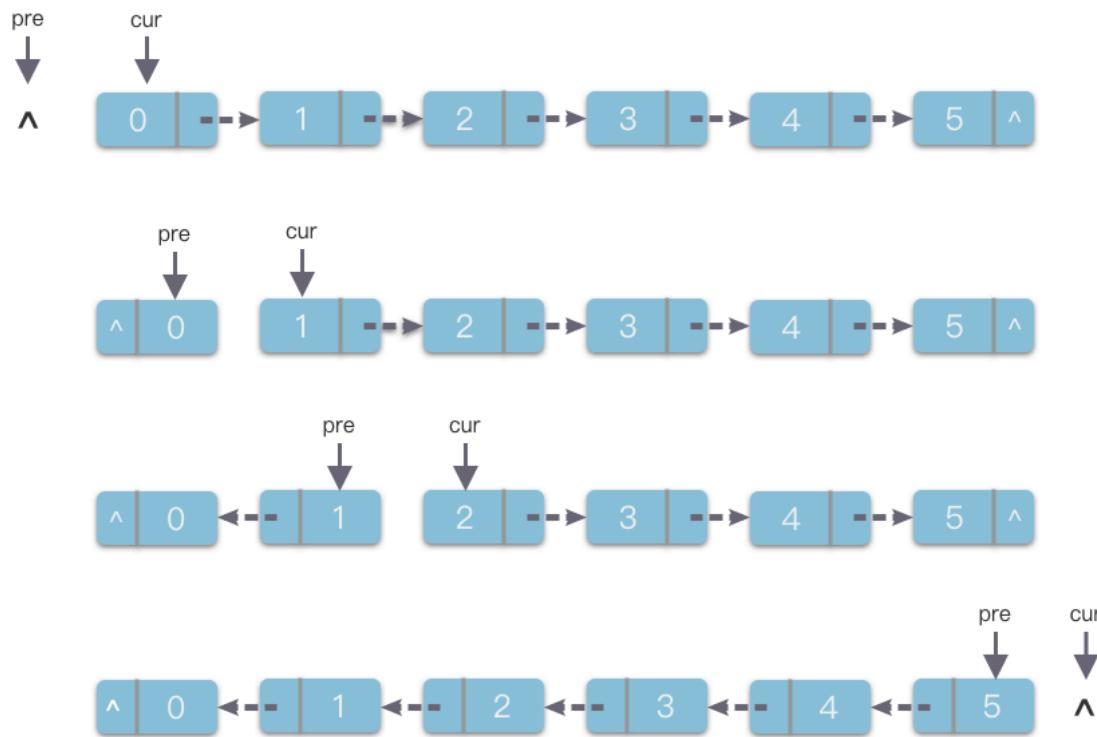
解题思路

思路 1：迭代

1. 使用两个指针 `cur` 和 `pre` 进行迭代。`pre` 指向 `cur` 前一个节点位置。初始时，`pre` 指向 `None`，`cur` 指向 `head`。
2. 将 `pre` 和 `cur` 的前后指针进行交换，指针更替顺序为：
 - i. 使用 `next` 指针保存当前节点 `cur` 的后一个节点，即 `next = cur.next`；
 - ii. 断开当前节点 `cur` 的后一节点链接，将 `cur` 的 `next` 指针指向前一节点 `pre`，即 `cur.next = pre`；
 - iii. `pre` 向前移动一步，移动到 `cur` 位置，即 `pre = cur`；
 - iv. `cur` 向前移动一步，移动到之前 `next` 指针保存的位置，即 `cur = next`。
3. 继续执行第 2 步中的 1、2、3、4。
4. 最后等到 `cur` 遍历到链表末尾，即 `cur == None`，时，`pre` 所在位置就是反转后链表的头节点，返回新的头节点 `pre`。

使用迭代法反转链表的示意图如下所示：

反转链表（迭代）



思路 1：代码

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pre = None
        cur = head
        while cur != None:
            next = cur.next
            cur.next = pre
            pre = cur
            cur = next
        return pre
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。

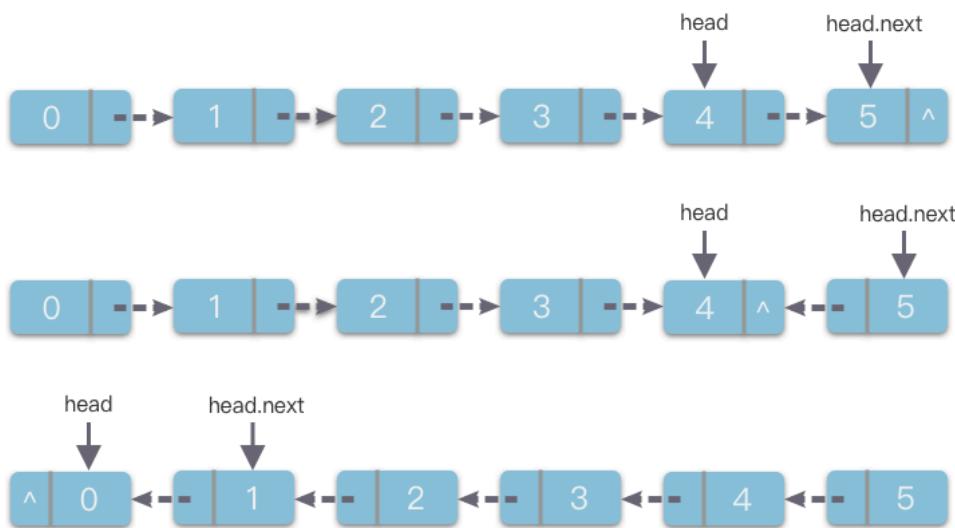
思路 2：递归

具体做法如下：

- 首先定义递归函数含义为：将链表反转，并返回反转后的头节点。
- 然后从 `head.next` 的位置开始调用递归函数，即将 `head.next` 为头节点的链表进行反转，并返回该链表的头节点。
- 递归到链表的最后一个节点，将其作为最终的头节点，即为 `new_head`。
- 在每次递归函数返回的过程中，改变 `head` 和 `head.next` 的指向关系。也就是将 `head.next` 的 `next` 指针先指向当前节点 `head`，即 `head.next.next = head`。
- 然后让当前节点 `head` 的 `next` 指针指向 `None`，从而实现从链表尾部开始的局部反转。
- 当递归从未尾开始顺着递归栈的退出，从而将整个链表进行反转。
- 最后返回反转后的链表头节点 `new_head`。

使用递归法反转链表的示意图如下所示：

反转链表（递归）



思路 2：代码

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        if head == None or head.next == None:
            return head
        new_head = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return new_head
```

思路 2：复杂度分析

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$ 。最多需要 n 层栈空间。

参考资料

- 【题解】[反转链表 - 反转链表 - 力扣](#)
- 【题解】[【反转链表】：双指针，递归，妖魔化的双指针 - 反转链表 - 力扣（LeetCode）](#)

0207. 课程表

- 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

- [0207. 课程表 - 力扣](#)

题目大意

描述：给定一个整数 $numCourses$, 代表这学期必须选修的课程数量, 课程编号为 $0 \sim numCourses - 1$ 。再给定一个数组 $prerequisites$ 表示先修课程关系, 其中 $prerequisites[i] = [ai, bi]$ 表示如果要学习课程 ai 则必须要先完成课程 bi 。

要求：判断是否可能完成所有课程的学习。如果可以, 返回 `True`, 否则, 返回 `False`。

说明：

- $1 \leq numCourses \leq 10^5$ 。
- $0 \leq prerequisites.length \leq 5000$ 。
- $prerequisites[i].length == 2$ 。
- $0 \leq ai, bi < numCourses$ 。
- $prerequisites[i]$ 中所有课程对互不相同。

示例：

- 示例 1:

```
输入: numCourses = 2, prerequisites = [[1,0]]
输出: true
解释: 总共有 2 门课程。学习课程 1 之前, 你需要完成课程 0。这是可能的。
```

- 示例 2:

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

输出: false

解释: 总共有 2 门课程。学习课程 1 之前, 你需要先完成课程 0; 并且学习课程 0 之前, 你还应先完成课程 1。这是不可能的。

解题思路

思路 1: 拓扑排序

1. 使用哈希表 *graph* 存放课程关系图, 并统计每门课程节点的入度, 存入入度列表 *indegrees*。
2. 借助队列 *S*, 将所有入度为 0 的节点入队。
3. 从队列中选择一个节点 *u*, 并令课程数减 1。
4. 从图中删除该顶点 *u*, 并且删除从该顶点出发的有向边 $< u, v >$ (也就是把该顶点可达的顶点入度都减 1)。如果删除该边后顶点 *v* 的入度变为 0, 则将其加入队列 *S* 中。
5. 重复上述步骤 3 ~ 4, 直到队列中没有节点。
6. 最后判断剩余课程数是否为 0, 如果为 0, 则返回 `True`, 否则, 返回 `False`。

思路 1: 代码

```
import collections

class Solution:
    def topologicalSorting(self, numCourses, graph):
        indegrees = {u: 0 for u in graph}
        for u in graph:
            for v in graph[u]:
                indegrees[v] += 1

        S = collections.deque([u for u in indegrees if indegrees[u] == 0])

        while S:
            u = S.pop()
            numCourses -= 1
            for v in graph[u]:
                indegrees[v] -= 1
                if indegrees[v] == 0:
                    S.append(v)

        if numCourses == 0:
            return True
        return False

    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        graph = dict()
        for i in range(numCourses):
            graph[i] = []

        for v, u in prerequisites:
            graph[u].append(v)

        return self.topologicalSorting(numCourses, graph)
```

思路 1: 复杂度分析

- 时间复杂度: $O(n + m)$, 其中 n 为课程数, m 为先修课程的要求数。
- 空间复杂度: $O(n + m)$ 。

0208. 实现 Trie (前缀树)

- 标签: 设计、字典树、哈希表、字符串
- 难度: 中等

题目链接

- [0208. 实现 Trie \(前缀树\) - 力扣](#)

题目大意

要求: 实现前缀树数据结构的相关类 `Trie` 类。

`Trie` 类:

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word`。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中, 返回 `True` (即, 在检索之前已经插入); 否则, 返回 `False`。
- `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix`, 返回 `True`; 否则, 返回 `False`。

说明:

- $1 \leq word.length, prefix.length \leq 2000$ 。
- `word` 和 `prefix` 仅由小写英文字母组成。
- `insert`、`search` 和 `startsWith` 调用次数 **总计** 不超过 $3 * 10^4$ 次。

示例:

- 示例 1:

```
输入:
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[], ["apple"], ["apple"], ["app"], ["app"], ["app"]]
输出:
[null, null, true, false, true, null, true]
```

解释:

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple"); // 返回 True
trie.search("app"); // 返回 False
trie.startsWith("app"); // 返回 True
trie.insert("app");
trie.search("app"); // 返回 True
```

解题思路

思路 1: 前缀树 (字典树)

前缀树 (字典树) 是一棵多叉树, 其中每个节点包含指向子节点的指针数组 `children`, 以及布尔变量 `isEnd`。`children` 用于存储当前字符节点, 一般长度为所含字符种类个数, 也可以使用哈希表代替指针数组。`isEnd` 用于判断该节点是否为字符串的结尾。

下面依次讲解插入、查找前缀的具体步骤:

插入字符串:

- 从根节点开始插入字符串。对于待插入的字符, 有两种情况:
 - 如果该字符对应的节点存在, 则沿着指针移动到子节点, 继续处理下一个字符。
 - 如果该字符对应的节点不存在, 则创建一个新的节点, 保存在 `children` 中对应位置上, 然后沿着指针移动到子节点, 继续处理下一个字符。

- 重复上述步骤，直到最后一个字符，然后将该节点标记为字符串的结尾。

查找前缀：

- 从根节点开始查找前缀，对于待查找的字符，有两种情况：
 - 如果该字符对应的节点存在，则沿着指针移动到子节点，继续查找下一个字符。
 - 如果该字符对应的节点不存在，则说明字典树中不包含该前缀，直接返回空指针。
- 重复上述步骤，直到最后一个字符搜索完毕，则说明字典树中存在该前缀。

思路 1：代码

```
class Node:
    def __init__(self):
        self.children = dict()
        self.isEnd = False

class Trie:

    def __init__(self):
        self.root = Node()

    def insert(self, word: str) -> None:
        cur = self.root
        for ch in word:
            if ch not in cur.children:
                cur.children[ch] = Node()
            cur = cur.children[ch]
        cur.isEnd = True

    def search(self, word: str) -> bool:
        cur = self.root
        for ch in word:
            if ch not in cur.children:
                return False
            cur = cur.children[ch]

        return cur is not None and cur.isEnd

    def startsWith(self, prefix: str) -> bool:
        cur = self.root
        for ch in prefix:
            if ch not in cur.children:
                return False
            cur = cur.children[ch]
        return cur is not None
```

思路 1：复杂度分析

- 时间复杂度：初始化为 $O(1)$ 。插入操作、查找操作的时间复杂度为 $O(|S|)$ 。其中 $|S|$ 是每次插入或查找字符串的长度。
- 空间复杂度： $O(|T| \times \sum)$ 。其中 $|T|$ 是所有插入字符串的长度之和， \sum 是字符集的大小。

0209. 长度最小的子数组

- 标签：数组、二分查找、前缀和、滑动窗口
- 难度：中等

题目链接

- [0209. 长度最小的子数组 - 力扣](#)

题目大意

描述：给定一个只包含正整数的数组 $nums$ 和一个正整数 $target$ 。

要求：找出数组中满足和大于等于 $target$ 的长度最小的「连续子数组」，并返回其长度。如果不存在符合条件的子数组，返回 0。

说明：

- $1 \leq target \leq 10^9$ 。
- $1 \leq nums.length \leq 10^5$ 。
- $1 \leq nums[i] \leq 10^5$ 。

示例：

- **示例 1：**

```
输入: target = 7, nums = [2,3,1,2,4,3]
输出: 2
解释: 子数组 [4,3] 是该条件下的长度最小的子数组。
```

- **示例 2：**

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

解题思路

思路 1：滑动窗口（不定长度）

最直接的做法是暴力枚举，时间复杂度为 $O(n^2)$ 。但是我们可以利用滑动窗口的方法，在时间复杂度为 $O(n)$ 的范围内解决问题。

用滑动窗口来记录连续子数组的和，设定两个指针： $left$ 、 $right$ ，分别指向滑动窗口的左右边界，保证窗口中的和刚好大于等于 $target$ 。

1. 一开始， $left$ 、 $right$ 都指向 0。
2. 向右移动 $right$ ，将最右侧元素加入当前窗口和 $window_sum$ 中。
3. 如果 $window_sum \geq target$ ，则不断右移 $left$ ，缩小滑动窗口长度，并更新窗口和的最小值，直到 $window_sum < target$ 。
4. 然后继续右移 $right$ ，直到 $right \geq len(nums)$ 结束。
5. 输出窗口和的最小值作为答案。

思路 1：代码

```
class Solution:
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        size = len(nums)
        ans = size + 1
        left = 0
        right = 0
        window_sum = 0

        while right < size:
            window_sum += nums[right]

            while window_sum >= target:
                ans = min(ans, right - left + 1)
                window_sum -= nums[left]
                left += 1

            right += 1

        return ans if ans != size + 1 else 0
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0210. 课程表 II

- 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

- [0210. 课程表 II - 力扣](#)

题目大意

描述：给定一个整数 $numCourses$, 代表这学期必须选修的课程数量, 课程编号为 $0 \sim numCourses - 1$ 。再给定一个数组 $prerequisites$ 表示先修课程关系, 其中 $prerequisites[i] = [ai, bi]$ 表示如果要学习课程 ai 则必须要先完成课程 bi 。

要求：返回学完所有课程所安排的学习顺序。如果有多个正确的顺序, 只要返回其中一种即可。如果无法完成所有课程, 则返回空数组。

说明：

- $1 \leq numCourses \leq 2000$ 。
- $0 \leq prerequisites.length \leq numCourses \times (numCourses - 1)$ 。
- $prerequisites[i].length == 2$ 。
- $0 \leq ai, bi < numCourses$ 。
- $ai \neq bi$ 。
- 所有 $[ai, bi]$ 互不相同。

示例：

- 示例 1:

```
输入: numCourses = 2, prerequisites = [[1,0]]
输出: [0,1]
解释: 总共有 2 门课程。要学习课程 1, 你需要先完成课程 0。因此, 正确的课程顺序为 [0,1]。
```

- 示例 2:

```
输入: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
输出: [0,2,1,3]
解释: 总共有 4 门课程。要学习课程 3, 你应该先完成课程 1 和课程 2。并且课程 1 和课程 2 都应该排在课程 0 之后。
因此, 一个正确的课程顺序是 [0,1,2,3]。另一个正确的排序是 [0,2,1,3]。
```

解题思路

思路 1：拓扑排序

这道题是「[0207. 课程表](#)」的升级版, 只需要在上一题的基础上增加一个答案数组 $order$ 即可。

- 使用哈希表 $graph$ 存放课程关系图, 并统计每门课程节点的入度, 存入入度列表 $indegrees$ 。
- 借助队列 S , 将所有入度为 0 的节点入队。
- 从队列中选择一个节点 u , 并将其加入到答案数组 $order$ 中。
- 从图中删除该顶点 u , 并且删除从该顶点出发的有向边 $< u, v >$ (也就是把该顶点可达的顶点入度都减 1)。如果删除该边后顶点 v 的入度变为 0, 则将其加入队列 S 中。
- 重复上述步骤 3 ~ 4, 直到队列中没有节点。

6. 最后判断总的顶点数和拓扑序列中的顶点数是否相等，如果相等，则返回答案数组 `order`，否则，返回空数组。

思路 1：代码

```

import collections

class Solution:
    # 拓扑排序，graph 中包含所有顶点的有向边关系（包括无边顶点）
    def topologicalSortingKahn(self, graph: dict):
        indegrees = {u: 0 for u in graph}  # indegrees 用于记录所有顶点入度
        for u in graph:
            for v in graph[u]:
                indegrees[v] += 1  # 统计所有顶点入度

        # 将入度为 0 的顶点存入集合 S 中
        S = collections.deque([u for u in indegrees if indegrees[u] == 0])
        order = []  # order 用于存储拓扑序列

        while S:
            u = S.pop()  # 从集合中选择一个没有前驱的顶点 u
            order.append(u)  # 将其输出到拓扑序列 order 中
            for v in graph[u]:  # 遍历顶点 u 的邻接顶点 v
                indegrees[v] -= 1  # 删除从顶点 u 出发的有向边
                if indegrees[v] == 0:  # 如果删除该边后顶点 v 的入度变为 0
                    S.append(v)  # 将其放入集合 S 中

        if len(indegrees) != len(order):  # 还有顶点未遍历（存在环），无法构成拓扑序列
            return []
        return order  # 返回拓扑序列

    def findOrder(self, numCourses: int, prerequisites: List[List[int]]):
        graph = defaultdict(list)
        for i in range(numCourses):
            graph[i] = []

        for v, u in prerequisites:
            graph[u].append(v)

        return self.topologicalSortingKahn(graph)

```

思路 1：复杂度分析

- 时间复杂度： $O(n + m)$ ，其中 n 为课程数， m 为先修课程的要求数。
- 空间复杂度： $O(n + m)$ 。

0211. 添加与搜索单词 - 数据结构设计

- 标签：深度优先搜索、设计、字典树、字符串
- 难度：中等

题目链接

- [0211. 添加与搜索单词 - 数据结构设计 - 力扣](#)

题目大意

要求：设计一个数据结构，支持「添加新单词」和「查找字符串是否与任何先前添加的字符串匹配」。

实现词典类 WordDictionary：

- `WordDictionary()` 初始化词典对象。
- `void addWord(word)` 将 `word` 添加到数据结构中，之后可以对它进行匹配。
- `bool search(word)` 如果数据结构中存在字符串与 `word` 匹配，则返回 `True`；否则，返回 `False`。`word` 中可能包含一些 `.`，每个 `.` 都可以表示任何一个字母。

说明：

- $1 \leq word.length \leq 25$ 。
- `addWord` 中的 `word` 由小写英文字母组成。
- `search` 中的 `word` 由 `'.'` 或小写英文字母组成。
- 最多调用 10^4 次 `addWord` 和 `search`。

示例：

- 示例 1：

```
输入:
["WordDictionary","addWord","addWord","addWord","addWord","search","search","search"]
[[],["bad"], ["dad"], ["mad"], ["pad"], ["bad"], [".ad"], ["b.."]]
输出:
[null,null,null,null,false,true,true]
```

解释：

```
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // 返回 False
wordDictionary.search("bad"); // 返回 True
wordDictionary.search(".ad"); // 返回 True
wordDictionary.search("b.."); // 返回 True
```

解题思路

思路 1：字典树

使用前缀树（字典树）。具体做法如下：

- 初始化词典对象时，构造一棵字典树。
- 添加 `word` 时，将 `word` 插入到字典树中。
- 搜索 `word` 时：
 - 如果遇到 `.`，则递归匹配当前节点所有子节点，并依次向下查找。匹配到了，则返回 `True`，否则返回 `False`。
 - 如果遇到其他小写字母，则按 `word` 顺序匹配节点。
 - 如果当前节点为 `word` 的结尾，则返回 `True`。

思路 1：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) -> bool:
        """
        Returns if the word is in the trie.
        """
        def dfs(index, node) -> bool:
            if index == len(word):
                return node.isEnd

            ch = word[index]
            if ch == '.':
                for child in node.children.values():
                    if child is not None and dfs(index + 1, child):
                        return True
            else:
                if ch not in node.children:
                    return False
                child = node.children[ch]
                if child is not None and dfs(index + 1, child):
                    return True
            return False

        return dfs(0, self)

class WordDictionary:

    def __init__(self):
        self.trie_tree = Trie()

    def addWord(self, word: str) -> None:
        self.trie_tree.insert(word)

    def search(self, word: str) -> bool:
        return self.trie_tree.search(word)

```

思路 1：复杂度分析

- 时间复杂度：初始化操作为 $O(1)$ 。添加单词为 $O(|S|)$ ，搜索单词的平均时间复杂度为 $O(|S|)$ ，最坏情况下所有字符都是 `'.'`，所以最坏时间复杂度为 `ParseError: KaTeX parse error: Got function '\sum' with no arguments as superscript at position 7: O(|S|^{\sum})`。其中 $|S|$ 为单词长度， \sum 为字符串集的大小，此处为 26。
- 空间复杂度： $O(|T| * n)$ 。其中 $|T|$ 为所有添加单词的最大长度， n 为添加字符串个数。

0212. 单词搜索 II

- 标签：字典树、数组、字符串、回溯、矩阵
- 难度：困难

题目链接

- [0212. 单词搜索 II - 力扣](#)

题目大意

给定一个 $m * n$ 二维字符网格 `board` 和一个单词（字符串）列表 `words`。

要求：找出所有同时在二维网格和字典中出现的单词。

注意：单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中「相邻」单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

解题思路

- 先将单词列表 `words` 中的所有单词存入字典树中。
- 然后遍历二维字符网格 `board` 的每一个字符 `board[i][j]`。
- 从当前单元格出发，从上下左右四个方向深度优先搜索遍历路径。每经过一个单元格，就将该单元格的字母修改为特殊字符，避免重复遍历，深度优先搜索完毕之后再恢复该单元格。
 - 如果当前路径恰好是 `words` 列表中的单词，则将结果添加到答案数组中。
 - 如果是 `words` 列表中单词的前缀，则继续搜索。
 - 如果不是 `words` 列表中单词的前缀，则停止搜索。
- 最后输出答案数组。

代码

```

class Trie:

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

    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
        cur.word = word

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

        return cur is not None and cur.isEnd

class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        trie_tree = Trie()
        for word in words:
            trie_tree.insert(word)

        directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        rows = len(board)
        cols = len(board[0])

        def dfs(cur, row, col):
            ch = board[row][col]
            if ch not in cur.children:
                return

            cur = cur.children[ch]
            if cur.isEnd:
                ans.add(cur.word)

            board[row][col] = "#"
            for direct in directs:
                new_row = row + direct[0]
                new_col = col + direct[1]
                if 0 <= new_row < rows and 0 <= new_col < cols:
                    dfs(cur, new_row, new_col)
            board[row][col] = ch

        ans = set()
        for row in range(rows):
            for col in range(cols):
                dfs(trie_tree, row, col)

        return list(ans)

```

```

ans = set()
for i in range(rows):
    for j in range(cols):
        dfs(trie_tree, i, j)

return list(ans)

```

0213. 打家劫舍 II

- 标签: 数组、动态规划
- 难度: 中等

题目链接

- [0213. 打家劫舍 II - 力扣](#)

题目大意

描述: 给定一个数组 $nums$, $num[i]$ 代表第 i 间房屋存放的金额, 假设房屋可以围成一圈, 最后一间房屋跟第一间房屋可以相连。相邻的房屋装有防盗系统, 假如相邻的两间房屋同时被偷, 系统就会报警。

要求: 假如你是一名专业的小偷, 计算在不触动警报装置的情况下, 一夜之内能够偷窃到的最高金额。

说明:

- $1 \leq nums.length \leq 100$ 。
- $0 \leq nums[i] \leq 1000$ 。

示例:

- **示例 1:**

```

输入: nums = [2,3,2]
输出: 3
解释: 你不能先偷窃 1 号房屋 (金额 = 2), 然后偷窃 3 号房屋 (金额 = 2) , 因为他们是相邻的。

```

- **示例 2:**

```

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

```

解题思路

思路 1：动态规划

这道题可以看做是「[198. 打家劫舍](#)」的升级版。

如果房屋数大于等于 3 间, 偷窃了第 1 间房屋, 则不能偷窃最后一间房屋。同样偷窃了最后一间房屋则不能偷窃第 1 间房屋。

假设总共房屋数量为 $size$, 这种情况可以转换为分别求解 $[0, size - 2]$ 和 $[1, size - 1]$ 范围下首尾不相连的房屋所能偷窃的最高金额, 然后再取这两种情况下的最大值。而求解 $[0, size - 2]$ 和 $[1, size - 1]$ 范围下首尾不相连的房屋所能偷窃的最高金额问题就跟「[198. 打家劫舍](#)」所求问题一致了。

这里来复习一下「[198. 打家劫舍](#)」的解题思路。

1. 划分阶段

按照房屋序号进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：前 i 间房屋所能偷窃到的最高金额。

3. 状态转移方程

i 间房屋的最后一个房子是 $nums[i - 1]$ 。

如果房屋数大于等于 2 间，则偷窃第 $i - 1$ 间房屋的时候，就有两种状态：

1. 偷窃第 $i - 1$ 间房屋，那么第 $i - 2$ 间房屋就不能偷窃了，偷窃的最高金额为：前 $i - 2$ 间房屋的最高总金额 + 第 $i - 1$ 间房屋的金额，即 $dp[i] = dp[i - 2] + nums[i - 1]$ ；
2. 不偷窃第 $i - 1$ 间房屋，那么第 $i - 2$ 间房屋可以偷窃，偷窃的最高金额为：前 $i - 1$ 间房屋的最高总金额，即 $dp[i] = dp[i - 1]$ 。

然后这两种状态取最大值即可，即状态转移方程为：

$$dp[i] = \begin{cases} nums[0] & i = 1 \\ \max(dp[i - 2] + nums[i - 1], dp[i - 1]) & i \geq 2 \end{cases}$$

4. 初始条件

- 前 0 间房屋所能偷窃到的最高金额为 0，即 $dp[0] = 0$ 。
- 前 1 间房屋所能偷窃到的最高金额为 $nums[0]$ ，即： $dp[1] = nums[0]$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：前 i 间房屋所能偷窃到的最高金额。假设求解 $[0, size - 2]$ 和 $[1, size - 1]$ 范围下（ $size$ 为总的房屋数）首尾不相连的房屋所能偷窃的最高金额问题分别为 $ans1$ 、 $ans2$ ，则最终结果为 $\max(ans1, ans2)$ 。

思路 1：动态规划代码

```
class Solution:
    def helper(self, nums):
        size = len(nums)
        if size == 0:
            return 0

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

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

        return dp[size]

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

        ans1 = self.helper(nums[:size - 1])
        ans2 = self.helper(nums[1:])
        return max(ans1, ans2)
```

思路 1：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 $O(n)$ 。

0215. 数组中的第K个最大元素

- 标签：数组、分治、快速排序、排序、堆（优先队列）

- 难度：中等

题目链接

- [0215. 数组中的第K个最大元素 - 力扣](#)

题目大意

描述：给定一个未排序的整数数组 $nums$ 和一个整数 k 。

要求：返回数组中第 k 个最大的元素。

说明：

- 要求使用时间复杂度为 $O(n)$ 的算法解决此问题。
- $1 \leq k \leq nums.length \leq 10^5$ 。
- $-10^4 \leq nums[i] \leq 10^4$ 。

示例：

- 示例 1：

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

- 示例 2：

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

解题思路

很不错的一道题，面试常考。

直接可以想到的思路是：排序后输出数组上对应第 k 位大的数。所以问题关键在于排序方法的复杂度。

冒泡排序、选择排序、插入排序时间复杂度 $O(n^2)$ 太高了，很容易超时。

可考虑堆排序、归并排序、快速排序。

这道题的要求是找到第 k 大的元素，使用归并排序只有到最后排序完毕才能返回第 k 大的数。而堆排序每次排序之后，就会确定一个元素的准确排名，同理快速排序也是如此。

思路 1：堆排序

升序堆排序的思路如下：

- 将无序序列构造成第 1 个大顶堆（初始堆），使得 n 个元素的最大值处于序列的第 1 个位置。
- 调整堆：**交换序列的第 1 个元素（最大值元素）与第 n 个元素的位置。将序列前 $n - 1$ 个元素组成的子序列调整成一个新的大顶堆，使得 $n - 1$ 个元素的最大值处于序列第 1 个位置，从而得到第 2 个最大值元素。
- 调整堆：**交换子序列的第 1 个元素（最大值元素）与第 $n - 1$ 个元素的位置。将序列前 $n - 2$ 个元素组成的子序列调整成一个新的大顶堆，使得 $n - 2$ 个元素的最大值处于序列第 1 个位置，从而得到第 3 个最大值元素。
- 依次类推，不断交换子序列的第 1 个元素（最大值元素）与当前子序列最后一个元素位置，并将其调整成新的大顶堆。直到获取第 k 个最大值元素为止。

思路 1：代码

```

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # 调整为大顶堆
        def heapify(nums, index, end):
            left = index * 2 + 1
            right = left + 1
            while left <= end:
                # 当前节点为非叶子节点
                max_index = index
                if nums[left] > nums[max_index]:
                    max_index = left
                if right <= end and nums[right] > 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

        # 初始化大顶堆
        def buildMaxHeap(nums):
            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):
                heapify(nums, i, size - 1)
            return nums

        buildMaxHeap(nums)
        size = len(nums)
        for i in range(k-1):
            nums[0], nums[size-i-1] = nums[size-i-1], nums[0]
            heapify(nums, 0, size-i-2)
        return nums[0]

```

思路 1：复杂度分析

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

思路 2：快速排序

使用快速排序在每次调整时，都会确定一个元素的最终位置，且以该元素为界限，将数组分成了左右两个子数组，左子数组中的元素都比该元素小，右子数组中的元素都比该元素大。

这样，只要某次划分的元素恰好是第 k 个下标就找到了答案。并且我们只需关注第 k 个最大元素所在区间的排序情况，与第 k 个最大元素无关的区间排序都可以忽略。这样进一步减少了执行步骤。

思路 2：代码

```

import random

class Solution:
    # 随机哨兵划分：从 nums[low: high + 1] 中随机挑选一个基准数，并进行移位排序
    def randomPartition(self, nums: [int], low: int, high: int) -> int:
        # 随机挑选一个基准数
        i = random.randint(low, high)
        # 将基准数与最低位互换
        nums[i], nums[low] = nums[low], nums[i]
        # 以最低位为基准数，然后将数组中比基准数大的元素移动到基准数右侧，比他小的元素移动到基准数左侧。最后将基准数放到正确位置上
        return self.partition(nums, low, high)

    # 哨兵划分：以第 1 位元素 nums[low] 为基准数，然后将比基准数小的元素移动到基准数左侧，将比基准数大的元素移动到基准数右侧，最后将基准数放到正确位置上
    def partition(self, nums: [int], low: int, high: int) -> int:
        # 以第 1 位元素为基准数
        pivot = nums[low]

        i, j = low, high
        while i < j:
            # 从右向左找到第 1 个小于基准数的元素
            while i < j and nums[j] >= pivot:
                j -= 1
            # 从左向右找到第 1 个大于基准数的元素
            while i < j and nums[i] <= pivot:
                i += 1
            # 交换元素
            nums[i], nums[j] = nums[j], nums[i]

        # 将基准数放到正确位置上
        nums[j], nums[low] = nums[low], nums[j]
        return j

    def quickSort(self, nums: [int], low: int, high: int, k: int, size: int) -> [int]:
        if low < high:
            # 按照基准数的位置，将数组划分为左右两个子数组
            pivot_i = self.randomPartition(nums, low, high)
            if pivot_i == size - k:
                return nums[size - k]
            if pivot_i > size - k:
                self.quickSort(nums, low, pivot_i - 1, k, size)
            if pivot_i < size - k:
                self.quickSort(nums, pivot_i + 1, high, k, size)

        return nums[size - k]

    def findKthLargest(self, nums: List[int], k: int) -> int:
        size = len(nums)
        return self.quickSort(nums, 0, len(nums) - 1, k, size)

```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。证明过程可参考「算法导论 9.2：期望为线性的选择算法」。
- 空间复杂度： $O(\log n)$ 。递归使用栈空间的空间代价期望为 $O(\log n)$ 。

思路 3：借用标准库（不建议）

提交代码中的最快代码是调用了 Python 的 `sort` 方法。这种做法适合在打算法竞赛的时候节省时间，日常练习可以尝试一下自己写。

思路 3：代码

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        nums.sort()
        return nums[len(nums) - k]
```

思路 3：复杂度分析

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

思路 4：优先队列

- 遍历数组元素，对于当前元素 num ：
 - 如果优先队列中的元素个数小于 k 个，则将当前元素 num 放入优先队列中。
 - 如果优先队列中的元素个数大于等于 k 个，并且当前元素 num 大于优先队列的队头元素，则弹出队头元素，并将当前元素 num 插入到优先队列中。
- 遍历完，此时优先队列的队头元素就是第 k 个最大元素，将其弹出并返回即可。

这里我们借助了 Python 中的 `heapq` 模块实现优先队列算法，这一步也可以通过手写堆的方式实现优先队列。

思路 4：代码

```
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        res = []
        for num in nums:
            if len(res) < k:
                heapq.heappush(res, num)
            elif num > res[0]:
                heapq.heappop(res)
                heapq.heappush(res, num)
        return heapq.heappop(res)
```

思路 4：复杂度分析

- 时间复杂度: $O(n \times \log k)$ 。
- 空间复杂度: $O(k)$ 。# 0217. 存在重复元素
- 标签: 数组、哈希表、排序
- 难度: 简单

题目链接

- 0217. 存在重复元素 - 力扣

题目大意

描述: 给定一个整数数组 `nums`。

要求: 判断是否存在重复元素。如果有元素在数组中出现至少两次，返回 `True`；否则返回 `False`。

说明:

- $1 \leq \text{nums.length} \leq 10^5$ 。
- $-10^9 \leq \text{nums}[i] \leq 10^9$ 。

示例:

- 示例 1:

```
输入: nums = [1,2,3,1]
输出: True
```

- 示例 2:

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

解题思路

思路 1：哈希表

- 使用一个哈希表存储元素和对应元素数量。
- 遍历元素，如果哈希表中出现了该元素，则直接输出 `True`。如果没有出现，则向哈希表中插入该元素。
- 如果遍历完也没发现重复元素，则输出 `False`。

思路 1：代码

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        numDict = dict()
        for num in nums:
            if num in numDict:
                return True
            else:
                numDict[num] = num
        return False
```

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：集合

- 使用一个 `set` 集合存储数组中所有元素。
- 如果集合中元素个数与数组元素个数不同，则说明出现了重复元素，返回 `True`。
- 如果集合中元素个数与数组元素个数相同，则说明没有出现了重复元素，返回 `False`。

思路 2：集合代码

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        return len(set(nums)) != len(nums)
```

思路 2：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

思路 3：排序

- 对数组进行排序。
- 排序之后，遍历数组，判断相邻元素之间是否出现重复元素。
- 如果相邻元素相同，则说明出现了重复元素，返回 `True`。

- 如果遍历完也没发现重复元素，则输出 `False`。

思路 3：排序代码

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        nums.sort()
        for i in range(1, len(nums)):
            if nums[i - 1] == nums[i]:
                return True
        return False
```

思路 3：复杂度分析

- 时间复杂度： $O(n \times \log n)$ 。
- 空间复杂度： $O(1)$ 。# 0218. 天际线问题
- 标签：树状数组、线段树、数组、分治、有序集合、扫描线、堆（优先队列）
- 难度：困难

题目链接

- [0218. 天际线问题 - 力扣](#)

题目大意

城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。

给定所有建筑物的位置和高度所组成的数组 `buildings`。其中三元素 `buildings[i] = [left_i, right_i, height_i]` 表示 `left_i` 是第 `i` 座建筑物左边界的一个 `x` 坐标。`right_i` 是第 `i` 座建筑物右边界的一个 `x` 坐标，`height_i` 是第 `i` 座建筑物的高度。

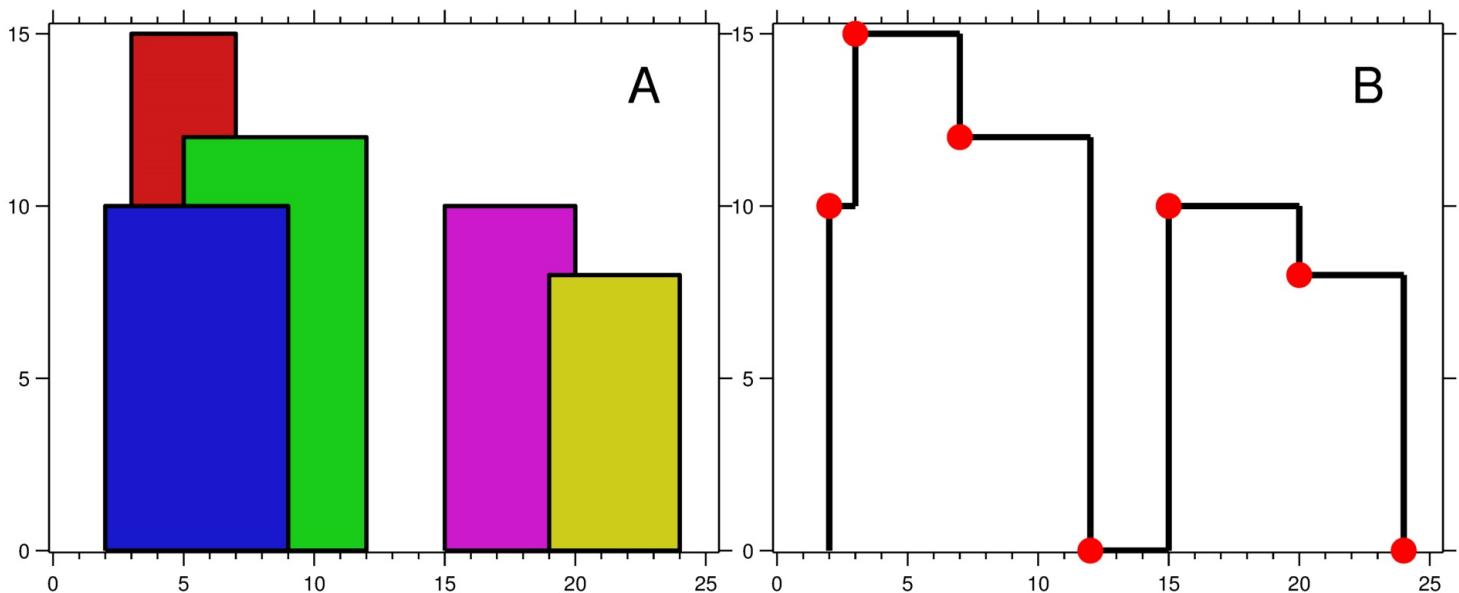
要求：返回由这些建筑物形成的天际线。

- 天际线：由“关键点”组成的列表，格式 `[[x1, y1], [x2, y2], [x3, y3], ...]`，并按 `x` 坐标进行排序。
- 关键点：水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，`y` 坐标始终为 `0`，仅用于标记天际线的终点。此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。

注意：输出天际线中不得有连续的相同高度的水平线。

- 例如 `[..., [2 3], [4 5], [7 5], [11 5], [12 7], ...]` 是不正确的答案；三条高度为 `5` 的线应该在最终输出中合并为一个：`[..., [2 3], [4 5], [12 7], ...]`。

示例：



- 图 A 显示输入的所有建筑物的位置和高度。
- 图 B 显示由这些建筑物形成的天际线。图 B 中的红点表示输出列表中的关键点。

解题思路

可以看出来：关键点的横坐标都在建筑物的左右边界上。

我们可以将左右边界最高处的坐标存入 `points` 数组中，然后按照建筑物左边界、右界的高度进行排序。

然后用一条条「垂直于 x 轴的扫描线」，从所有建筑物的最左侧依次扫描到最右侧。从而将建筑物分割成规则的矩形。

不难看出：相邻的两个坐标的横坐标与矩形所能达到的最大高度构成了一个矩形。相邻两个坐标的横坐标可以从排序过的 `points` 数组中依次获取，矩形所能达到的最大高度可以用一个优先队列（堆）`max_heap` 来维护。使用数组 `ans` 来作为答案。

在依次从左到右扫描坐标时：

- 当扫描到建筑物的左边界时，说明必然存在一条向右延伸的边。此时将高度加入到优先队列中。
- 当扫描到建筑物的右边界时，说明从之前的左边界延伸的边结束了，此时将高度从优先队列中移除。

因为三条高度相同的线应该合并为一个，所以我们用 `prev` 来记录之前上一个矩形高度。

- 如果当前矩形高度 `curr` 与之前矩形高度 `prev` 相同，则跳过。
- 如果当前矩形高度 `curr` 与之前矩形高度 `prev` 不相同，则将其加入到答案数组中，并更新上一矩形高度 `prev` 的值。

最后，输出答案 `ans`。

代码

```

from sortedcontainers import SortedList

class Solution:
    def getSkyline(self, buildings: List[List[int]]) -> List[List[int]]:
        ans = []
        points = []
        for building in buildings:
            left, right, height = building[0], building[1], building[2]
            points.append([left, -height])
            points.append([right, height])
        points.sort(key=lambda x:(x[0], x[1]))

        prev = 0
        max_heap = SortedList([prev])

        for point in points:
            x, height = point[0], point[1]
            if height < 0:
                max_heap.add(-height)
            else:
                max_heap.remove(height)

            curr = max_heap[-1]
            if curr != prev:
                ans.append([x, curr])
            prev = curr
        return ans

```

参考资料

- 【题解】【宫水三叶】扫描线算法基本思路 & 优先队列维护当前最大高度 - 天际线问题 - 力扣

0219. 存在重复元素 II

- 标签：数组、哈希表、滑动窗口
- 难度：简单

题目链接

- [0219. 存在重复元素 II - 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：判断是否存在 $nums[i] == nums[j]$ ($i \neq j$)，并且 i 和 j 的差绝对值至多为 k 。

说明：

- $1 \leq nums.length \leq 10^5$ 。
- $-10^9 \leq nums[i] \leq 10^9$ 。
- $0 \leq k \leq 10^5$ 。

示例：

- 示例 1：

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

解题思路

思路 1：哈希表

维护一个最多有 k 个元素的哈希表。遍历 $nums$ ，对于数组中的每个整数 $nums[i]$ ，判断哈希表中是否存在这个整数。

- 如果存在，则说明出现了两次，且 $i \neq j$ ，直接返回 `True`。
- 如果不存在，则将 $nums[i]$ 加入哈希表。
- 判断哈希表长度是否超过了 k ，如果超过了 k ，则删除哈希表中最旧的元素 $nums[i - k]$ 。
- 如果遍历完仍旧找不到，则返回 `False`。

思路 1：代码

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        nums_dict = dict()
        for i in range(len(nums)):
            if nums[i] in nums_dict:
                return True
            nums_dict[nums[i]] = 1
            if len(nums_dict) > k:
                del nums_dict[nums[i - k]]
        return False
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

0220. 存在重复元素 III

- 标签: 数组、桶排序、有序集合、排序、滑动窗口
- 难度: 中等

题目链接

- [0220. 存在重复元素 III - 力扣](#)

题目大意

描述: 给定一个整数数组 $nums$ ，以及两个整数 k 、 t 。

要求: 判断数组中是否存在两个不同下标的 i 和 j ，其对应元素满足 $abs(nums[i] - nums[j]) \leq t$ ，同时满足 $abs(i - j) \leq k$ 。如果满足条件则返回 `True`，不满足条件返回 `False`。

说明:

- $0 \leq nums.length \leq 2 \times 10^4$ 。
- $-2^{31} \leq nums[i] \leq 2^{31} - 1$ 。
- $0 \leq k \leq 10^4$ 。
- $0 \leq t \leq 2^{31} - 1$ 。

示例:

- 示例 1:

```
输入: nums = [1,2,3,1], k = 3, t = 0
输出: True
```

- 示例 2:

```
输入: nums = [1,0,1,1], k = 1, t = 2
输出: True
```

解题思路

题目中需要满足两个要求，一个是元素值的要求 ($\text{abs}(\text{nums}[i] - \text{nums}[j]) \leq t$)，一个是下标范围的要求 ($\text{abs}(i - j) \leq k$)。

对于任意一个位置 i 来说，合适的 j 应该在区间 $[i - k, i + k]$ 内，同时 $\text{nums}[j]$ 值应该在区间 $[\text{nums}[i] - t, \text{nums}[i] + t]$ 内。

最简单的做法是两重循环遍历数组，第一重循环遍历位置 i ，第二重循环遍历 $[i - k, i + k]$ 的元素，判断是否满足 $\text{abs}(\text{nums}[i] - \text{nums}[j]) \leq t$ 。但是这样做的时间复杂度为 $O(n \times k)$ ，其中 n 是数组 nums 的长度。

我们需要优化一下检测相邻 $2 \times k$ 个元素是否满足 $\text{abs}(\text{nums}[i] - \text{nums}[j]) \leq t$ 的方法。有两种思路：「桶排序」和「滑动窗口（固定长度）」。

思路 1：桶排序

- 利用桶排序的思想，将桶的大小设置为 $t + 1$ 。只需要使用一重循环遍历位置 i ，然后根据 $\lfloor \frac{\text{nums}[i]}{t+1} \rfloor$ ，从而决定将 $\text{nums}[i]$ 放入哪个桶中。
- 这样在同一个桶内各个元素之间的差值绝对值都小于等于 t 。而相邻桶之间的元素，只需要校验一下两个桶之间的差值是否不超过 t 。这样就可以以 $O(1)$ 的时间复杂度检测相邻 $2 \times k$ 个元素是否满足 $\text{abs}(\text{nums}[i] - \text{nums}[j]) \leq t$ 。
- 而 $\text{abs}(i - j) \leq k$ 条件则可以通过在一重循环遍历时，将超出范围的 $\text{nums}[i - k]$ 从对应桶中删除，从而保证桶中元素一定满足 $\text{abs}(i - j) \leq k$ 。

具体步骤如下：

- 将每个桶的大小设置为 $t + 1$ 。我们将元素按照大小依次放入不同的桶中。
- 遍历数组 nums 中的元素，对于元素 $\text{nums}[i]$ ：
 - 如果 $\text{nums}[i]$ 放入桶之前桶里已经有元素了，那么这两个元素必然满足 $\text{abs}(\text{nums}[i] - \text{nums}[j]) \leq t$ ，
 - 如果之前桶里没有元素，那么就将 $\text{nums}[i]$ 放入对应桶中。
 - 再判断左右桶的左右两侧桶中是否有元素满足 $\text{abs}(\text{nums}[i] - \text{nums}[j]) \leq t$ 。
 - 然后将 $\text{nums}[i - k]$ 之前的桶清空，因为这些桶中的元素与 $\text{nums}[i]$ 已经不满足 $\text{abs}(i - j) \leq k$ 了。
- 最后上述满足条件的情况就返回 `True`，最终遍历完仍不满足条件就返回 `False`。

思路 1：代码

```

class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:
        bucket_dict = dict()
        for i in range(len(nums)):
            # 将 nums[i] 划分到大小为 t + 1 的不同桶中
            num = nums[i] // (t + 1)

            # 桶中已经有元素了
            if num in bucket_dict:
                return True

            # 把 nums[i] 放入桶中
            bucket_dict[num] = nums[i]

            # 判断左侧桶是否满足条件
            if (num - 1) in bucket_dict and abs(bucket_dict[num - 1] - nums[i]) <= t:
                return True
            # 判断右侧桶是否满足条件
            if (num + 1) in bucket_dict and abs(bucket_dict[num + 1] - nums[i]) <= t:
                return True
            # 将 i - k 之前的旧桶清除，因为之前的桶已经不满足条件了
            if i >= k:
                bucket_dict.pop(nums[i - k] // (t + 1))

        return False

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。 n 是给定数组长度。
- 空间复杂度: $O(\min(n, k))$ 。桶中最多包含 $\min(n, k + 1)$ 个元素。

思路 2：滑动窗口（固定长度）

- 使用一个长度为 k 的滑动窗口，每次遍历到 $nums[right]$ 时，滑动窗口内最多包含 $nums[right]$ 之前最多 k 个元素。只需要检查前 k 个元素是否在 $[nums[right] - t, nums[right] + t]$ 区间内即可。
- 检查 k 个元素是否在 $[nums[right] - t, nums[right] + t]$ 区间，可以借助保证有序的数据结构（比如 `SortedList`）+ 二分查找来解决，从而减少时间复杂度。

具体步骤如下：

- 使用有序数组类 `window` 维护一个长度为 k 的窗口，满足数组内元素有序，且支持增加和删除操作。
- `left`、`right` 都指向序列的第一个元素。即：`left = 0`，`right = 0`。
- 将当前元素填入窗口中，即 `window.add(nums[right])`。
- 当窗口元素大于 k 个时，即当 $right - left > k$ 时，移除窗口最左侧元素，并向右移动 `left`。
- 当窗口元素小于等于 k 个时：
 - 使用二分查找算法，查找 $nums[right]$ 在 `window` 中的位置 `idx`。
 - 判断 `window[idx]` 与相邻位置上元素差值绝对值，若果满足 $abs(window[idx] - window[idx - 1]) \leq t$ 或者 $abs(window[idx + 1] - window[idx]) \leq t$ 时返回 `True`。
- 向右移动 `right`。
- 重复 3 ~ 6 步，直到 `right` 到达数组末尾，如果还没找到满足条件的情况，则返回 `False`。

思路 2：代码

```

from sortedcontainers import SortedList

class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int], k: int, t: int) -> bool:
        size = len(nums)
        window = SortedList()
        left, right = 0, 0
        while right < size:
            window.add(nums[right])

            if right - left > k:
                window.remove(nums[left])
                left += 1

            idx = bisect.bisect_left(window, nums[right])

            if idx > 0 and abs(window[idx] - window[idx - 1]) <= t:
                return True
            if idx < len(window) - 1 and abs(window[idx + 1] - window[idx]) <= t:
                return True

            right += 1

        return False

```

思路 2：复杂度分析

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

参考资料

- 【题解】利用桶的原理O(n), Python3 - 存在重复元素 III - 力扣

0221. 最大正方形

- 标签: 数组、动态规划、矩阵
- 难度: 中等

题目链接

- [0221. 最大正方形 - 力扣](#)

题目大意

描述: 给定一个由 '0' 和 '1' 组成的二维矩阵 $matrix$ 。

要求: 找到只包含 '1' 的最大正方形, 并返回其面积。

说明:

- $m == matrix.length$ 。
- $n == matrix[i].length$ 。
- $1 \leq m, n \leq 300$ 。
- $matrix[i][j]$ 为 '0' 或 '1'。

示例：

- 示例 1：

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

输入：matrix = [[“1”, “0”, “1”, “0”, “0”], [“1”, “0”, “1”, “1”, “1”], [“1”, “1”, “1”, “1”, “1”], [“1”, “0”, “0”, “1”, “0”]]

输出：4

- 示例 2：

0	1
1	0

输入：matrix = [[“0”, “1”], [“1”, “0”]]

输出：1

解题思路

思路 1：动态规划

1. 划分阶段

按照正方形的右下角坐标进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 表示为：以矩阵位置 (i, j) 为右下角，且值包含 1 的正方形的最大边长。

3. 状态转移方程

只有当矩阵位置 (i, j) 值为 1 时，才有可能存在正方形。

- 如果矩阵位置 (i, j) 上值为 0，则 $dp[i][j] = 0$ 。
- 如果矩阵位置 (i, j) 上值为 1，则 $dp[i][j]$ 的值由该位置上方、左侧、左上方三者共同约束的，为三者中最小值加 1。即： $dp[i][j] = \min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1$ 。

4. 初始条件

- 默认所有以矩阵位置 (i, j) 为右下角，且值包含 1 的正方形的最大边长都为 0，即 $dp[i][j] = 0$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i][j]$ 表示为：以矩阵位置 (i, j) 为右下角，且值包含 1 的正方形的最大边长。则最终结果为所有 $dp[i][j]$ 中的最大值。

思路 1：代码

```
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        rows, cols = len(matrix), len(matrix[0])
        max_size = 0
        dp = [[0 for _ in range(cols + 1)] for _ in range(rows + 1)]
        for i in range(rows):
            for j in range(cols):
                if matrix[i][j] == '1':
                    if i == 0 or j == 0:
                        dp[i][j] = 1
                    else:
                        dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1
                    max_size = max(max_size, dp[i][j])
        return max_size * max_size
```

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 $matrix$ 的行数和列数。
- 空间复杂度: $O(m \times n)$ 。

0222. 完全二叉树的节点个数

- 标签: 树、深度优先搜索、二分查找、二叉树
- 难度: 中等

题目链接

- [0222. 完全二叉树的节点个数 - 力扣](#)

题目大意

给定一棵完全二叉树的根节点 `root`，返回该树的节点个数。

- 完全二叉树: 除了最底层节点可能没有填满外，其余各层节点数都达到了最大值，并且最下面一层的节点都集中在最左边的若干位置。若最底层在第 h 层，则该层包含 $1 \sim 2^h$ 个节点。

解题思路

根据题意可知公式：当前根节点的节点个数 = 左子树节点个数 + 右子树节点个数 + 1。

根据上述公式递归遍历左右子树节点，并返回左右子树节点数 + 1。

代码

```
class Solution:
    def countNodes(self, root: TreeNode) -> int:
        if not root:
            return 0
        return 1 + self.countNodes(root.left) + self.countNodes(root.right)
```

0223. 矩形面积

- 标签: 几何、数学

- 难度：中等

题目链接

- [0223. 矩形面积 - 力扣](#)

题目大意

给定两个矩形的左下角坐标、右上角坐标 $(ax1, ay1, ax2, ay2, bx1, by1, bx2, by2)$ 。其中 $(ax1, ay1)$ 表示第一个矩形左下角坐标， $(ax2, ay2)$ 表示第一个矩形右上角坐标， $(bx1, by1)$ 表示第二个矩形左下角坐标， $(bx2, by2)$ 表示第二个矩形右上角坐标。

要求：计算出两个矩形覆盖的总面积。

解题思路

两个矩形覆盖的总面积 = 第一个矩形面积 + 第二个矩形面积 - 重叠部分面积。

需要分别计算出两个矩形面积，还有求出相交部分的长、宽，并计算出对应重叠部分的面积。

代码

```
class Solution:
    def computeArea(self, ax1: int, ay1: int, ax2: int, ay2: int, bx1: int, by1: int, bx2: int, by2: int) -> int:
        area_a = (ax2 - ax1) * (ay2 - ay1)
        area_b = (bx2 - bx1) * (by2 - by1)
        overlap_width = max(0, min(ax2, bx2) - max(ax1, bx1))
        overlap_height = max(0, min(ay2, by2) - max(ay1, by1))
        area_overlap = overlap_width * overlap_height

        return area_a + area_b - area_overlap
```

0225. 用队列实现栈

- 标签：栈、设计、队列
- 难度：简单

题目链接

- [0225. 用队列实现栈 - 力扣](#)

题目大意

要求：仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的四种操作：`push`、`top`、`pop` 和 `empty`。

要求实现 `MyStack` 类：

- `void push(int x)` 将元素 `x` 压入栈顶。
- `int pop()` 移除并返回栈顶元素。
- `int top()` 返回栈顶元素。
- `boolean empty()` 如果栈是空的，返回 `True`；否则，返回 `False`。

说明：

- 只能使用队列的基本操作 —— 也就是 `push to back`、`peek/pop from front`、`size` 和 `is empty` 这些操作。
- 所使用的语言也许不支持队列。你可以使用 `list`（列表）或者 `deque`（双端队列）来模拟一个队列，只要是标准的队列操作即可。

示例：

- 示例 1：

```
输入:  
["MyStack", "push", "push", "top", "pop", "empty"]  
[[], [1], [2], [], [], []]  
输出:  
[null, null, null, 2, 2, false]
```

解释：

```
MyStack myStack = new MyStack();  
myStack.push(1);  
myStack.push(2);  
myStack.top(); // 返回 2  
myStack.pop(); // 返回 2  
myStack.empty(); // 返回 False
```

解题思路

思路 1：双队列

使用两个队列。`pushQueue` 用作入栈，`popQueue` 用作出栈。

- `push` 操作：将新加入的元素压入 `pushQueue` 队列中，并且将之前保存在 `popQueue` 队列中的元素从队头开始依次压入 `pushQueue` 中，此时 `pushQueue` 队列中头节点存放的是新加入的元素，尾部存放的是之前的元素。而 `popQueue` 则为空。再将 `pushQueue` 和 `popQueue` 相互交换，保持 `pushQueue` 为空，`popQueue` 则用于 `pop`、`top` 等操作。
- `pop` 操作：直接将 `popQueue` 队头元素取出。
- `top` 操作：返回 `popQueue` 队头元素。
- `empty`：判断 `popQueue` 是否为空。

思路 1：代码

```

class MyStack:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.pushQueue = collections.deque()
        self.popQueue = collections.deque()

    def push(self, x: int) -> None:
        """
        Push element x onto stack.
        """
        self.pushQueue.append(x)
        while self.popQueue:
            self.pushQueue.append(self.popQueue.popleft())
        self.pushQueue, self.popQueue = self.popQueue, self.pushQueue

    def pop(self) -> int:
        """
        Removes the element on top of the stack and returns that element.
        """
        return self.popQueue.popleft()

    def top(self) -> int:
        """
        Get the top element.
        """
        return self.popQueue[0]

    def empty(self) -> bool:
        """
        Returns whether the stack is empty.
        """
        return not self.popQueue

# Your MyStack object will be instantiated and called as such:
# obj = MyStack()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.top()
# param_4 = obj.empty()

```

思路 1：复杂度分析

- 时间复杂度：入栈操作的时间复杂度为 $O(n)$ 。出栈、取栈顶元素、判断栈是否为空的时间复杂度为 $O(1)$ 。
- 空间复杂度： $O(n)$ 。[# 0226. 翻转二叉树](#)
- 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

- [0226. 翻转二叉树 - 力扣](#)

题目大意

描述：给定一个二叉树的根节点 `root`。

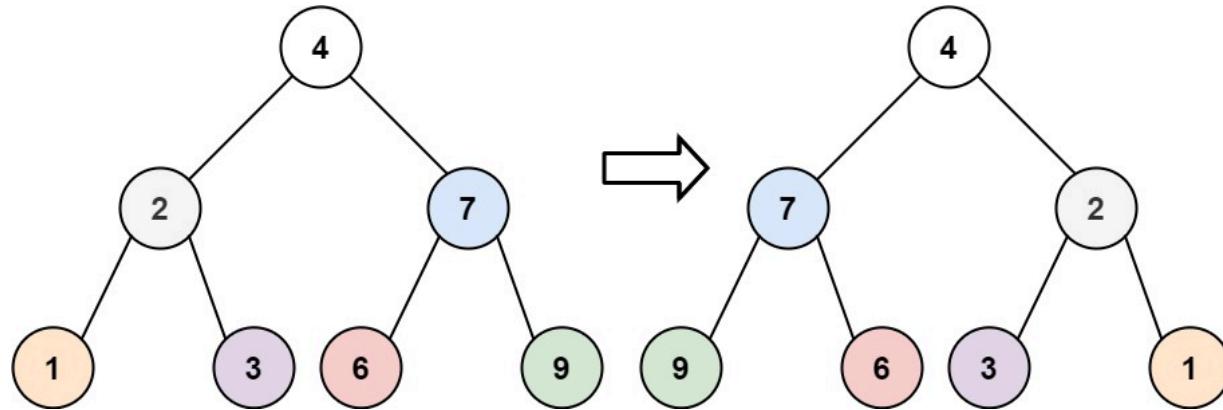
要求：将该二叉树进行左右翻转。

说明：

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

示例：

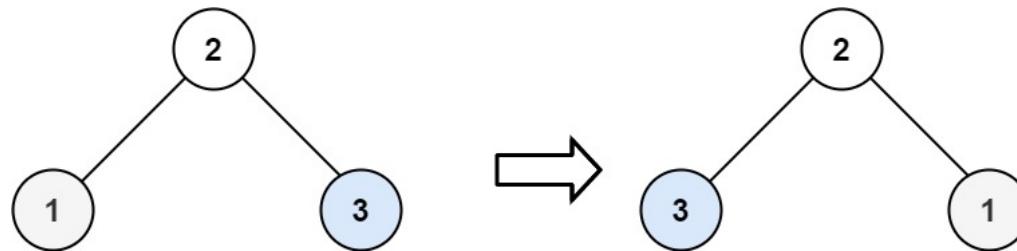
- 示例 1：



输入：`root = [4, 2, 7, 1, 3, 6, 9]`

输出：`[4, 7, 2, 9, 6, 3, 1]`

- 示例 2：



输入：`root = [2, 1, 3]`

输出：`[2, 3, 1]`

解题思路

思路 1：递归遍历

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：

- i. 递归遍历翻转左子树。
- ii. 递归遍历翻转右子树。
- iii. 交换当前根节点 `root` 的左右子树。

2. 明确终止条件：当前节点 `root` 为 `None`。

3. 翻译为递归代码：

- i. 定义递归函数：`invertTree(self, root)` 表示输入参数为二叉树的根节点 `root`，返回结果为翻转后二叉树的根节点。
- ii. 书写递归主体：

```

left = self.invertTree(root.left)
right = self.invertTree(root.right)
root.left = right
root.right = left
return root

```

iii. 明确递归终止条件: `if not root: return None`

4. 返回根节点 `root`。

思路 1: 代码

```

class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return None
        left = self.invertTree(root.left)
        right = self.invertTree(root.right)
        root.left = right
        root.right = left
        return root

```

思路 1: 复杂度分析

- 时间复杂度: $O(n)$, 其中 n 是二叉树的节点数目。
- 空间复杂度: $O(n)$ 。递归函数需要用到栈空间, 栈空间取决于递归深度, 最坏情况下递归深度为 n , 所以空间复杂度为 $O(n)$ 。

0227. 基本计算器 II

- 标签: 栈、数学、字符串
- 难度: 中等

题目链接

- [0227. 基本计算器 II - 力扣](#)

题目大意

描述: 给定一个字符串表达式 `s`, 表达式中所有整数为非负整数, 运算符只有 `+`、`-`、`*`、`/`, 没有括号。

要求: 实现一个基本计算器来计算并返回它的值。

说明:

- $1 \leq s.length \leq 3 * 10^5$ 。
- `s` 由整数和算符 (`+`、`-`、`*`、`/`) 组成, 中间由一些空格隔开。
- `s` 表示一个有效表达式。
- 表达式中的所有整数都是非负整数, 且在范围 $[0, 2^{31} - 1]$ 内。
- 题目数据保证答案是一个 32-bit 整数。

示例:

- 示例 1:

```

输入: s = "3+2*2"
输出: 7

```

- 示例 2:

输入: s = " 3/2 "

输出: 1

解题思路

思路 1：栈

计算表达式中，乘除运算优先于加减运算。我们可以先进行乘除运算，再将进行乘除运算后的整数值放入原表达式中相应位置，再依次计算加减。

可以考虑使用一个栈来保存进行乘除运算后的整数值。正整数直接压入栈中，负整数，则将对应整数取负号，再压入栈中。这样最终计算结果就是栈中所有元素的和。

具体做法：

1. 遍历字符串 s，使用变量 op 来标记数字之前的运算符，默认为 +。
2. 如果遇到数字，继续向后遍历，将数字进行累积，得到完整的整数 num。判断当前 op 的符号。
 - i. 如果 op 为 +，则将 num 压入栈中。
 - ii. 如果 op 为 -，则将 -num 压入栈中。
 - iii. 如果 op 为 *，则将栈顶元素 top 取出，计算 top * num，并将计算结果压入栈中。
 - iv. 如果 op 为 /，则将栈顶元素 top 取出，计算 int(top / num)，并将计算结果压入栈中。
3. 如果遇到 +、-、*、/ 操作符，则更新 op。
4. 最后将栈中整数进行累加，并返回结果。

思路 1：代码

```
class Solution:
    def calculate(self, s: str) -> int:
        size = len(s)
        stack = []
        op = '+'
        index = 0
        while index < size:
            if s[index] == ' ':
                index += 1
                continue
            if s[index].isdigit():
                num = ord(s[index]) - ord('0')
                while index + 1 < size and s[index+1].isdigit():
                    index += 1
                    num = 10 * num + ord(s[index]) - ord('0')
                if op == '+':
                    stack.append(num)
                elif op == '-':
                    stack.append(-num)
                elif op == '*':
                    top = stack.pop()
                    stack.append(top * num)
                elif op == '/':
                    top = stack.pop()
                    stack.append(int(top / num))
            elif s[index] in "+-*/":
                op = s[index]
            index += 1
        return sum(stack)
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

0231. 2 的幂

- 标签：位运算、递归、数学
- 难度：简单

题目链接

- [0231. 2 的幂 - 力扣](#)

题目大意

描述：给定一个整数 n 。

要求：判断该整数 n 是否是 2 的幂次方。如果是，返回 `True`；否则，返回 `False`。

说明：

- $-2^{31} \leq n \leq 2^{31} - 1$

示例：

- **示例 1：**

```
输入: n = 1
输出: True
解释: 2^0 = 1
```

- **示例 2：**

```
输入: n = 16
输出: True
解释: 2^4 = 16
```

解题思路

思路 1：循环判断

1. 不断判断 n 是否能整除 2。
 - i. 如果不能整除，则返回 `False`。
 - ii. 如果能整除，则让 n 整除 2，直到 $n < 2$ 。
2. 如果最后 $n == 1$ ，则返回 `True`，否则则返回 `False`。

思路 1：代码

```
class Solution:
    def isPowerOfTwo(self, n: int) -> bool:
        if n <= 0:
            return False

        while n % 2 == 0:
            n //= 2
        return n == 1
```

思路 1：复杂度分析

- 时间复杂度： $O(\log_2 n)$ 。
- 空间复杂度： $O(1)$ 。

思路 2：数论判断

因为 n 能取的最大值为 $2^{31} - 1$ 。我们可以计算出：在 n 的范围内， 2 的幂次方最大为 $2^{30} = 1073741824$ 。

因为 2 为质数，则 2^{30} 的除数只有 $2^0, 2^1, \dots, 2^{30}$ 。所以如果 n 为 2 的幂次方，则 2^{30} 肯定能被 n 整除，直接判断即可。

思路 2：代码

```
class Solution:
    def isPowerOfTwo(self, n: int) -> bool:
        return n > 0 and 1073741824 % n == 0
```

思路 2：复杂度分析

- 时间复杂度： $O(1)$ 。
- 空间复杂度： $O(1)$ 。

0232. 用栈实现队列

- 标签：栈、设计、队列
- 难度：简单

题目链接

- [0232. 用栈实现队列 - 力扣](#)

题目大意

要求：仅使用两个栈实现先入先出队列。

要求实现 `MyQueue` 类：

- `void push(int x)` 将元素 `x` 推到队列的末尾。
- `int pop()` 从队列的开头移除并返回元素。
- `int peek()` 返回队列开头的元素。
- `boolean empty()` 如果队列为空，返回 `True`；否则，返回 `False`。

说明：

- 只能使用标准的栈操作 —— 也就是只有 `push to top`, `peek / pop from top`, `size`, 和 `is empty` 操作是合法的。
- 可以使用 `list` 或者 `deque` (双端队列) 来模拟一个栈，只要是标准的栈操作即可。
- $1 \leq x \leq 9$ 。
- 最多调用 100 次 `push`、`pop`、`peek` 和 `empty`。
- 假设所有操作都是有效的 (例如，一个空的队列不会调用 `pop` 或者 `peek` 操作)。
- 进阶：实现每个操作均摊时间复杂度为 `O(1)` 的队列。换句话说，执行 `n` 个操作的总时间复杂度为 `O(n)`，即使其中一个操作可能花费较长时间。

示例：

- 示例 1：

```
输入:  
["MyQueue", "push", "push", "peek", "pop", "empty"]  
[], [1], [2], [], [], []  
输出:  
[null, null, null, 1, 1, false]
```

解释:

```
MyQueue myQueue = new MyQueue();  
myQueue.push(1); // queue is: [1]  
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)  
myQueue.peek(); // return 1  
myQueue.pop(); // return 1, queue is [2]  
myQueue.empty(); // return false
```

解题思路

思路 1：双栈

使用两个栈，`inStack` 用于输入，`outStack` 用于输出。

- `push` 操作：将元素压入 `inStack` 中。
- `pop` 操作：如果 `outStack` 输出栈为空，将 `inStack` 输入栈元素依次取出，按顺序压入 `outStack` 栈。这样 `outStack` 栈的元素顺序和之前 `inStack` 元素顺序相反，`outStack` 顶层元素就是要取出的队头元素，将其移出，并返回该元素。如果 `outStack` 输出栈不为空，则直接取出顶层元素。
- `peek` 操作：和 `pop` 操作类似，只不过最后一步不需要取出顶层元素，直接将其返回即可。
- `empty` 操作：如果 `inStack` 和 `outStack` 都为空，则队列为空，否则队列不为空。

思路 1：代码

```

class MyQueue:

    def __init__(self):
        self.inStack = []
        self.outStack = []
        .....
        Initialize your data structure here.
        .....

    def push(self, x: int) -> None:
        self.inStack.append(x)
        .....
        Push element x to the back of queue.
        .....

    def pop(self) -> int:
        if(len(self.outStack) == 0):
            while(len(self.inStack) != 0):
                self.outStack.append(self.inStack[-1])
                self.inStack.pop()
            top = self.outStack[-1]
            self.outStack.pop()
            return top
        .....
        Removes the element from in front of queue and returns that element.
        .....

    def peek(self) -> int:
        if (len(self.outStack) == 0):
            while (len(self.inStack) != 0):
                self.outStack.append(self.inStack[-1])
                self.inStack.pop()
            top = self.outStack[-1]
            return top
        .....
        Get the front element.
        .....

    def empty(self) -> bool:
        return len(self.outStack) == 0 and len(self.inStack) == 0
    .....
    Returns whether the queue is empty.
    .....

```

思路 1：复杂度分析

- 时间复杂度：`push` 和 `empty` 为 $O(1)$, `pop` 和 `peek` 为均摊 $O(1)$ 。
- 空间复杂度： $O(n)$ 。

0233. 数字 1 的个数

- 标签：递归、数学、动态规划
- 难度：困难

题目链接

- 0233. 数字 1 的个数 - 力扣

题目大意

描述：给定一个整数 n 。

要求：计算所有小于等于 n 的非负整数中数字 1 出现的个数。

说明：

- $0 \leq n \leq 10^9$ 。

示例：

- 示例 1：

```
输入: n = 13
输出: 6
```

- 示例 2：

```
输入: n = 0
输出: 0
```

解题思路

思路 1：动态规划 + 数位 DP

将 n 转换为字符串 s , 定义递归函数 `def dfs(pos, cnt, isLimit):` 表示构造第 pos 位及之后所有数位中数字 1 出现的个数。接下来按照如下步骤进行递归。

1. 从 `dfs(0, 0, True)` 开始递归。`dfs(0, 0, True)` 表示：
 - 从位置 0 开始构造。
 - 初始数字 1 出现的个数为 0。
 - 开始时受到数字 n 对应最高位数位的约束。
2. 如果遇到 $pos == len(s)$, 表示到达数位末尾, 此时：返回数字 1 出现的个数 cnt 。
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：
 - 因为不需要考虑前导 0 所以当前位数位所能选择的最小数字 ($minX$) 为 0。
 - 根据 $isLimit$ 来决定填当前位数位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 。
 - 方案数累加上当前位选择 d 之后的方案数, 即：`ans += dfs(pos + 1, cnt + (d == 1), isLimit and d == maxX)`。
 - `cnt + (d == 1)` 表示之前数字 1 出现的个数加上当前位为数字 1 的个数。
 - `isLimit and d == maxX` 表示 $pos + 1$ 位受到之前位 pos 位限制。
6. 最后的方案数为 `dfs(0, 0, True)`, 将其返回即可。

思路 1：代码

```

class Solution:
    def countDigitOne(self, n: int) -> int:
        # 将 n 转换为字符串 s
        s = str(n)

        @cache
        # pos: 第 pos 个数位
        # cnt: 之前数字 1 出现的个数。
        # 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。
        def dfs(pos, cnt, isLimit):
            if pos == len(s):
                return cnt

            ans = 0
            # 不需要考虑前导 0，则最小可选择数字为 0
            minX = 0
            # 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):
                ans += dfs(pos + 1, cnt + (d == 1), isLimit and d == maxX)
            return ans

        return dfs(0, 0, True)

```

思路 1：复杂度分析

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

0234. 回文链表

- 标签: 栈、递归、链表、双指针
- 难度: 简单

题目链接

- [0234. 回文链表 - 力扣](#)

题目大意

描述: 给定一个链表的头节点 `head`。

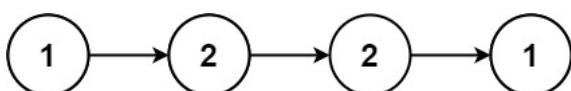
要求: 判断该链表是否为回文链表。

说明:

- 链表中节点数目在范围 $[1, 10^5]$ 内。
- $0 \leq Node.val \leq 9$ 。

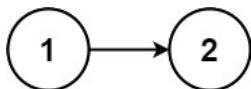
示例:

- 示例 1:



输入: head = [1,2,2,1]
输出: True

- 示例 2:



输入: head = [1,2]
输出: False

解题思路

思路 1：利用数组 + 双指针

- 利用数组，将链表元素依次存入。
- 然后再使用两个指针，一个指向数组开始位置，一个指向数组结束位置。
- 依次判断首尾对应元素是否相等，如果都相等，则为回文链表。如果不相等，则不是回文链表。

思路 1：代码

```

class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        nodes = []
        p1 = head
        while p1 != None:
            nodes.append(p1.val)
            p1 = p1.next
        return nodes == nodes[::-1]
  
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(n)$ 。

0235. 二叉搜索树的最近公共祖先

- 标签: 树、深度优先搜索、二叉搜索树、二叉树
- 难度: 中等

题目链接

- [0235. 二叉搜索树的最近公共祖先 - 力扣](#)

题目大意

描述: 给定一个二叉搜索树的根节点 `root`，以及两个指定节点 `p` 和 `q`。

要求: 找到该树中两个指定节点的最近公共祖先。

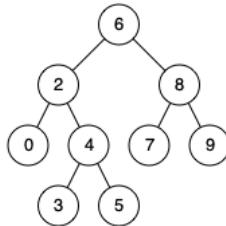
说明:

- 祖先**: 若节点 `p` 在节点 `node` 的左子树或右子树中，或者 `p == node`，则称 `node` 是 `p` 的祖先。
- 最近公共祖先**: 对于树的两个节点 `p`、`q`，最近公共祖先表示为一个节点 `lca_node`，满足 `lca_node` 是 `p`、`q` 的祖先且 `lca_node` 的深度尽可能大（一个节点也可以是自己的祖先）。

- 所有节点的值都是唯一的。
- `p`、`q` 为不同节点且均存在于给定的二叉搜索树中。

示例：

- 示例 1：



输入： `root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8`

输出： 6

解释： 节点 2 和节点 8 的最近公共祖先是 6。

- 示例 2：

输入： `root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4`

输出： 2

解释： 节点 2 和节点 4 的最近公共祖先是 2，因为根据定义最近公共祖先节点可以为节点本身。

解题思路

思路 1：递归遍历

对于节点 `p`、节点 `q`，最近公共祖先就是从根节点分别到它们路径上的分岔点，也是路径中最后一个相同的节点，现在我们的问题就是求这个分岔点。

我们可以使用递归遍历查找二叉搜索树的最近公共祖先，具体方法如下。

- 从根节点 `root` 开始遍历。
- 如果当前节点的值大于 `p`、`q` 的值，说明 `p` 和 `q` 应该在当前节点的左子树，因此将当前节点移动到它的左子节点，继续遍历；
- 如果当前节点的值小于 `p`、`q` 的值，说明 `p` 和 `q` 应该在当前节点的右子树，因此将当前节点移动到它的右子节点，继续遍历；
- 如果当前节点不满足上面两种情况，则说明 `p` 和 `q` 分别在当前节点的左右子树上，则当前节点就是分岔点，直接返回该节点即可。

思路 1：代码

```

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        ancestor = root
        while True:
            if ancestor.val > p.val and ancestor.val > q.val:
                ancestor = ancestor.left
            elif ancestor.val < p.val and ancestor.val < q.val:
                ancestor = ancestor.right
            else:
                break
        return ancestor
  
```

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个数。
- 空间复杂度： $O(1)$ 。[# 0236. 二叉树的最近公共祖先](#)
- 标签：树、深度优先搜索、二叉树
- 难度：中等

题目链接

- 0236. 二叉树的最近公共祖先 - 力扣

题目大意

描述：给定一个二叉树的根节点 `root`，以及二叉树中两个节点 `p` 和 `q`。

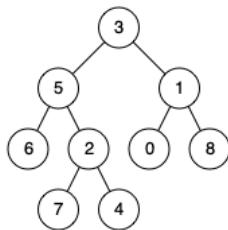
要求：找到该二叉树中指定节点 `p`、`q` 的最近公共祖先。

说明：

- 祖先：如果节点 `p` 在节点 `node` 的左子树或右子树中，或者 `p == node`，则称 `node` 是 `p` 的祖先。
- 最近公共祖先：对于树的两个节点 `p`、`q`，最近公共祖先表示为一个节点 `lca_node`，满足 `lca_node` 是 `p`、`q` 的祖先且 `lca_node` 的深度尽可能大（一个节点也可以是自己的祖先）。
- 树中节点数目在范围 $[2, 10^5]$ 内。
- $-10^9 \leq \text{Node.val} \leq 10^9$ 。
- 所有 `Node.val` 互不相同。
- `p != q`。
- `p` 和 `q` 均存在于给定的二叉树中。

示例：

- 示例 1：

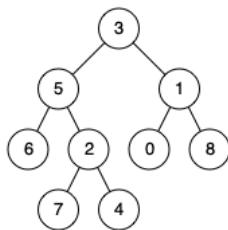


输入：`root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1`

输出：`3`

解释：节点 `5` 和节点 `1` 的最近公共祖先是节点 `3`。

- 示例 2：



输入：`root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4`

输出：`5`

解释：节点 `5` 和节点 `4` 的最近公共祖先是节点 `5`。因为根据定义最近公共祖先节点可以为节点本身。

解题思路

思路 1：递归遍历

设 `lca_node` 为节点 `p`、`q` 的最近公共祖先。则 `lca_node` 只能是下面几种情况：

1. `p`、`q` 在 `lca_node` 的子树中，且分别在 `lca_node` 的两侧子树中。

2. `p == lca_node`, 且 `q` 在 `lca_node` 的左子树或右子树中。
3. `q == lca_node`, 且 `p` 在 `lca_node` 的左子树或右子树中。

下面递归求解 `lca_node`。递归需要满足以下条件：

- 如果 `p`、`q` 都不为空, 则返回 `p`、`q` 的公共祖先。
- 如果 `p`、`q` 只有一个存在, 则返回存在的一个。
- 如果 `p`、`q` 都不存在, 则返回 `None`。

具体思路为：

1. 如果当前节点 `node` 等于 `p` 或者 `q`, 那么 `node` 就是 `p`、`q` 的最近公共祖先, 直接返回 `node`。
2. 如果当前节点 `node` 不为 `None`, 则递归遍历左子树、右子树, 并判断左右子树结果。
 - i. 如果左右子树都不为空, 则说明 `p`、`q` 在当前根节点的两侧, 当前根节点就是他们的最近公共祖先。
 - ii. 如果左子树为空, 则返回右子树。
 - iii. 如果右子树为空, 则返回左子树。
 - iv. 如果左右子树都为空, 则返回 `None`。
3. 如果当前节点 `node` 为 `None`, 则说明 `p`、`q` 不在 `node` 的子树中, 不可能为公共祖先, 直接返回 `None`。

思路 1：代码

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if root == p or root == q:
            return root

        if root:
            node_left = self.lowestCommonAncestor(root.left, p, q)
            node_right = self.lowestCommonAncestor(root.right, p, q)
            if node_left and node_right:
                return root
            elif not node_left:
                return node_right
            else:
                return node_left
        return None
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。其中 n 是二叉树的节点数目。
- 空间复杂度: $O(n)$ 。

0237. 删除链表中的节点

- 标签: 链表
- 难度: 中等

题目链接

- [0237. 删除链表中的节点 - 力扣](#)

题目大意

删除链表的给定节点。

解题思路

直接将该节点的后续节点覆盖该节点即可。即让该节点的值等于下一节点值, 并让其 `next` 指针指向下一节点的下一节点。

代码

```
class Solution:
    def deleteNode(self, node):
        node.val = node.next.val
        node.next = node.next.next
```

0238. 除自身以外数组的乘积

- 标签：数组、前缀和
- 难度：中等

题目链接

- [0238. 除自身以外数组的乘积 - 力扣](#)

题目大意

描述：给定一个数组 nums 。

要求：返回数组 answer , 其中 $\text{answer}[i]$ 等于 nums 中除 $\text{nums}[i]$ 之外其余各元素的乘积。

说明：

- 题目数据保证数组 nums 之中任意元素的全部前缀元素和后缀的乘积都在 32 位整数范围内。
- 请不要使用除法, 且在 $O(n)$ 时间复杂度内解决问题。
- **进阶：**在 $O(1)$ 的额外空间复杂度内完成这个题目。
- $2 \leq \text{nums.length} \leq 10^5$ 。
- $-30 \leq \text{nums}[i] \leq 30$ 。

示例：

- **示例 1：**

```
输入: nums = [1,2,3,4]
输出: [24,12,8,6]
```

- **示例 2：**

```
输入: nums = [-1,1,0,-3,3]
输出: [0,0,9,0,0]
```

解题思路

思路 1：两次遍历

1. 构造一个答案数组 res , 长度和数组 nums 长度一致。
2. 先从左到右遍历一遍 nums 数组, 将 $\text{nums}[i]$ 左侧的元素乘积累积起来, 存储到 res 数组中。
3. 再从右到左遍历一遍, 将 $\text{nums}[i]$ 右侧的元素乘积累积起来, 再乘以原本 $\text{res}[i]$ 的值, 即为 nums 中除了 $\text{nums}[i]$ 之外的其他所有元素乘积。

思路 1：代码

```

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

        left = 1
        for i in range(size):
            res[i] *= left
            left *= nums[i]

        right = 1
        for i in range(size-1, -1, -1):
            res[i] *= right
            right *= nums[i]
        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。

0239. 滑动窗口最大值

- 标签: 队列、数组、滑动窗口、单调队列、堆 (优先队列)
- 难度: 困难

题目链接

- [0239. 滑动窗口最大值 - 力扣](#)

题目大意

描述: 给定一个整数数组 `nums`，再给定一个整数 `k`，表示为大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。我们只能看到滑动窗口内的 `k` 个数字，滑动窗口每次只能向右移动一位。

要求: 返回滑动窗口中的最大值。

说明:

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

示例:

- 示例 1:

输入: `nums = [1,3,-1,-3,5,3,6,7], k = 3`

输出: `[3,3,5,5,6,7]`

解释:

滑动窗口的位置 最大值

<code>[1 3 -1] -3 5 3 6 7</code>	3
<code>1 [3 -1 -3] 5 3 6 7</code>	3
<code>1 3 [-1 -3 5] 3 6 7</code>	5
<code>1 3 -1 [-3 5 3] 6 7</code>	5
<code>1 3 -1 -3 [5 3 6] 7</code>	6
<code>1 3 -1 -3 5 [3 6 7]</code>	7

- 示例 2:

输入: `nums = [1], k = 1`

输出: `[1]`

解题思路

暴力求解的话，需要使用二重循环遍历，其时间复杂度为 $O(n * k)$ 。根据题目给定的数据范围，肯定会超时。

我们可以使用优先队列来做。

思路 1：优先队列

- 初始的时候将前 k 个元素加入优先队列的二叉堆中。存入优先队列的是数组值与索引构成的元组。优先队列将数组值作为优先级。
- 然后滑动窗口从第 k 个元素开始遍历，将当前数组值和索引的元组插入到二叉堆中。
- 当二叉堆堆顶元素的索引已经不在滑动窗口的范围内时，即 $q[0][1] <= i - k$ 时，不断删除堆顶元素，直到最大值元素的索引在滑动窗口的范围内。
- 将最大值加入到答案数组中，继续向右滑动。
- 滑动结束时，输出答案数组。

思路 1：代码

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        size = len(nums)
        q = [(-nums[i], i) for i in range(k)]
        heapq.heapify(q)
        res = [-q[0][0]]

        for i in range(k, size):
            heapq.heappush(q, (-nums[i], i))
            while q[0][1] <= i - k:
                heapq.heappop(q)
            res.append(-q[0][0])
        return res
```

思路 1：复杂度分析

- 时间复杂度: $O(n \times \log_2 n)$ 。
- 空间复杂度: $O(k)$ 。

0240. 搜索二维矩阵 II

- 标签: 二分查找、分治算法
- 难度: 中等

题目链接

- 0240. 搜索二维矩阵 II - 力扣

题目大意

描述：给定一个 $m \times n$ 大小的有序整数矩阵 $matrix$ 。 $matrix$ 中的每行元素从左到右升序排列，每列元素从上到下升序排列。再给定一个目标值 $target$ 。

要求：判断矩阵中是否可以找到 $target$ ，如果可以找到 $target$ ，返回 `True`，否则返回 `False`。

说明：

- $m == matrix.length$ 。
- $n == matrix[i].length$ 。
- $1 \leq n, m \leq 300$ 。
- $-10^9 \leq matrix[i][j] \leq 10^9$ 。
- 每行的所有元素从左到右升序排列。
- 每列的所有元素从上到下升序排列。
- $-10^9 \leq target \leq 10^9$ 。

示例：

- **示例 1：**

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
输入: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 5
输出: True
```

- **示例 2：**

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
输入: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]], target = 20
输出: False
```

解题思路

思路 1：二分查找

矩阵是有序的，可以考虑使用二分查找来做。

1. 迭代对角线元素，假设对角线元素的坐标为 (row, col) 。把数组元素按对角线分为右上角部分和左下角部分。
2. 对于当前对角线元素右侧第 row 行、对角线元素下侧第 col 列分别进行二分查找。
 - i. 如果找到目标，直接返回 `True`。
 - ii. 如果找不到目标，则缩小范围，继续查找。
 - iii. 直到所有对角线元素都遍历完，依旧没找到，则返回 `False`。

思路 1：代码

```

class Solution:
    def diagonalBinarySearch(self, matrix, diagonal, target):
        left = 0
        right = diagonal
        while left < right:
            mid = left + (right - left) // 2
            if matrix[mid][mid] < target:
                left = mid + 1
            else:
                right = mid
        return left

    def rowBinarySearch(self, matrix, begin, cols, target):
        left = begin
        right = cols
        while left < right:
            mid = left + (right - left) // 2
            if matrix[begin][mid] < target:
                left = mid + 1
            elif matrix[begin][mid] > target:
                right = mid - 1
            else:
                left = mid
                break
        return begin <= left <= cols and matrix[begin][left] == target

    def colBinarySearch(self, matrix, begin, rows, target):
        left = begin + 1
        right = rows
        while left < right:
            mid = left + (right - left) // 2
            if matrix[mid][begin] < target:
                left = mid + 1
            elif matrix[mid][begin] > target:
                right = mid - 1
            else:
                left = mid
                break
        return begin <= left <= rows and matrix[left][begin] == target

    def searchMatrix(self, matrix, target: int) -> bool:
        rows = len(matrix)
        if rows == 0:
            return False
        cols = len(matrix[0])
        if cols == 0:
            return False

        min_val = min(rows, cols)
        index = self.diagonalBinarySearch(matrix, min_val - 1, target)
        if matrix[index][index] == target:
            return True
        for i in range(index + 1):
            row_search = self.rowBinarySearch(matrix, i, cols - 1, target)
            col_search = self.colBinarySearch(matrix, i, rows - 1, target)
            if row_search or col_search:
                return True
        return False

```

思路 1：复杂度分析

- 时间复杂度: $O(\min(m, n) \times (\log_2 m + \log_2 n))$, 其中 m 是矩阵的行数, n 是矩阵的列数。

- 空间复杂度: $O(1)$ 。# 0241. 为运算表达式设计优先级
- 标签: 递归、记忆化搜索、数学、字符串、动态规划
- 难度: 中等

题目链接

- 0241. 为运算表达式设计优先级 - 力扣

题目大意

描述: 给定一个由数字和运算符组成的字符串 `expression`。

要求: 按不同优先级组合数字和运算符，计算并返回所有可能组合的结果。你可以按任意顺序返回答案。

说明:

- 生成的测试用例满足其对应输出值符合 32 位整数范围，不同结果的数量不超过 10^4 。
- $1 \leq expression.length \leq 20$ 。
- `expression` 由数字和算符 '+'、'-' 和 '*' 组成。
- 输入表达式中的所有整数值在范围 [0, 99]。

示例:

- **示例 1:**

```
输入: expression = "2-1-1"
输出: [0, 2]
解释:
((2-1)-1) = 0
(2-(1-1)) = 2
```

- **示例 2:**

```
输入: expression = "2*3-4*5"
输出: [-34, -14, -10, -10, 10]
解释:
(2*(3-(4*5))) = -34
((2*3)-(4*5)) = -14
((2*(3-4))*5) = -10
(2*((3-4)*5)) = -10
(((2*3)-4)*5) = 10
```

解题思路

思路 1：分治算法

给定的字符串 `expression` 只包含有数字和字符，可以写成类似 `x op y` 的形式，其中 `x`、`y` 为表达式或数字，`op` 为字符。

则我们可以根据字符的位置，将其递归分解为 `x`、`y` 两个部分，接着分别计算 `x` 部分的结果与 `y` 部分的结果。然后再将其合并。

思路 1：代码

```

class Solution:
    def diffWaysToCompute(self, expression: str) -> List[int]:
        res = []
        if len(expression) <= 2:
            res.append(int(expression))
            return res

        for i in range(len(expression)):
            ch = expression[i]
            if ch == '+' or ch == '-' or ch == '*':
                left_cnts = self.diffWaysToCompute(expression[:i])
                right_cnts = self.diffWaysToCompute(expression[i+1:])

                for left in left_cnts:
                    for right in right_cnts:
                        if ch == '+':
                            res.append(left + right)
                        elif ch == '-':
                            res.append(left - right)
                        else:
                            res.append(left * right)

        return res

```

思路 1：复杂度分析

- 时间复杂度: $O(C_n)$, 其中 n 为结果数组的大小, C_n 是第 n 个卡特兰数。
- 空间复杂度: $O(C_n)$ 。

0242. 有效的字母异位词

- 标签: 哈希表、字符串、排序
- 难度: 简单

题目链接

- [0242. 有效的字母异位词 - 力扣](#)

题目大意

描述: 给定两个字符串 s 和 t 。

要求: 判断 t 和 s 是否使用了相同的字符构成 (字符出现的种类和数目都相同)。

说明:

- 字母异位词: 如果 s 和 t 中每个字符出现的次数都相同, 则称 s 和 t 互为字母异位词。
- $1 \leq s.length, t.length \leq 5 \times 10^4$ 。
- s 和 t 仅包含小写字母。

示例:

- 示例 1:

```

输入: s = "anagram", t = "nagaram"
输出: True

```

- 示例 2:

```
输入: s = "rat", t = "car"
输出: False
```

解题思路

思路 1：哈希表

- 先判断字符串 s 和 t 的长度，不一样直接返回 `False`；
- 分别遍历字符串 s 和 t 。先遍历字符串 s ，用哈希表存储字符串 s 中字符出现的频次；
- 再遍历字符串 t ，哈希表中减去对应字符的频次，出现频次小于 0 则输出 `False`；
- 如果没有出现频次小于 0，则输出 `True`。

思路 1：代码

```
def isAnagram(self, s: str, t: str) -> bool:
    if len(s) != len(t):
        return False
    strDict = dict()
    for ch in s:
        if ch in strDict:
            strDict[ch] += 1
        else:
            strDict[ch] = 1
    for ch in t:
        if ch in strDict:
            strDict[ch] -= 1
            if strDict[ch] < 0:
                return False
        else:
            return False
    return True
```

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: $O(|S|)$, 其中 S 为字符集大小, 此处 $S == 26$ 。

0249. 移位字符串分组

- 标签: 数组、哈希表、字符串
- 难度: 中等

题目链接

- [0249. 移位字符串分组 - 力扣](#)

题目大意

给定一个仅包含小写字母的字符串列表。其中每个字符串都可以进行「移位」操作，也就是将字符串中的每个字母变为其在字母表中后续的字母。比如：`abc` -> `bcd`。

要求：将该列表中满足「移位」操作规律的组合进行分组并返回。

解题思路

我们可以先将满足相同「移位」操作规律的组合翻译为相同的模式，然后利用哈希表进行存储。哈希表对应关系为 翻译后模式：该模式对应的原字符串列表。

代码

```
import collections
class Solution:
    def groupStrings(self, strings: List[str]) -> List[List[str]]:
        str_dict = collections.defaultdict(list)
        for string in strings:
            if string[0] == 'a':
                str_dict[string].append(string)
            else:
                list_string = list(string)
                for i in range(len(list_string)):
                    num = (ord(list_string[i]) - ord(string[0]) + 26) % 26
                    list_string[i] = chr(num + ord('a'))
                temp_string = ''.join(list_string)
                str_dict[temp_string].append(string)
        res = list()
        for string, sublist in str_dict.items():
            res.append(sublist)
        return res
```

0257. 二叉树的所有路径

- 标签：树、深度优先搜索、字符串、回溯、二叉树
- 难度：简单

题目链接

- [0257. 二叉树的所有路径 - 力扣](#)

题目大意

给定一个二叉树，返回所有从根节点到叶子节点的路径。

解题思路

深度优先搜索。在递归遍历时，需考虑当前节点和左右孩子节点。

- 如果当前节点不是叶子节点，则当前拼接路径中加入该点，并继续递归遍历。
- 如果当前节点是叶子节点，则当前拼接路径中加入该点，并将当前路径加入答案数组。

代码

```
class Solution:
    def binaryTreePaths(self, root: TreeNode) -> List[str]:
        res = []
        def dfs(root, path):
            if not root:
                return
            path += str(root.val)
            if not root.left and not root.right:
                res.append(path)
            elif not root.right:
                dfs(root.left, path + "->")
            elif not root.left:
                dfs(root.right, path + "->")
            else:
                dfs(root.left, path + "->")
                dfs(root.right, path + "->")
        dfs(root, "")
        return res
```

0258. 各位相加

- 标签: 数学、数论、模拟
- 难度: 简单

题目链接

- [0258. 各位相加 - 力扣](#)

题目大意

给定一个非负整数 num，反复将各个位上的数字相加，直到结果为一位数。

解题思路

根据题意，循环模拟累加即可。

代码

```
class Solution:
    def addDigits(self, num: int) -> int:
        while num >= 10:
            cur = 0
            while num:
                cur += num % 10
                num //= 10
            num = cur
        return num
```

0259. 较小的三数之和

- 标签: 数组、双指针、二分查找、排序
- 难度: 中等

题目链接

- [0259. 较小的三数之和 - 力扣](#)

题目大意

描述：给定一个长度为 n 的整数数组和一个目标值 $target$ 。

要求：寻找能够使条件 $nums[i] + nums[j] + nums[k] < target$ 成立的三元组 (i, j, k) 的个数 $(0 \leq i < j < k < n)$ 。

说明：

- 最好在 $O(n^2)$ 的时间复杂度内解决问题。
- $n == nums.length$ 。
- $0 \leq n \leq 3500$ 。
- $-100 \leq nums[i] \leq 100$ 。
- $-100 \leq target \leq 100$ 。

示例：

- 示例 1：

```
输入: nums = [-2, 0, 1, 3], target = 2
```

```
输出: 2
```

解释：因为一共有两个三元组满足累加和小于 2：

```
[ -2, 0, 1 ]
```

```
[ -2, 0, 3 ]
```

- 示例 2：

```
输入: nums = [], target = 0
```

```
输出: 0
```

解题思路

思路 1：排序 + 双指针

三元组直接枚举的时间复杂度是 $O(n^3)$ ，明显不符合题目要求。那么可以考虑使用双指针减少循环内的时问复杂度。具体做法如下：

- 先对数组进行从小到大排序。
- 遍历数组，对于数组元素 $nums[i]$ ，使用两个指针 $left$ 、 $right$ 。 $left$ 指向第 $i + 1$ 个元素位置， $right$ 指向数组的最后一个元素位置。
- 在区间 $[left, right]$ 中查找满足 $nums[i] + nums[left] + nums[right] < target$ 的方案数。
- 计算 $nums[i]$ 、 $nums[left]$ 、 $nums[right]$ 的和，将其与 $target$ 比较。
 - 如果 $nums[i] + nums[left] + nums[right] < target$ ，则说明 i 、 $left$ 、 $right$ 作为三元组满足题目要求，同时说明区间 $[left, right]$ 中的元素作为 $right$ 都满足条件，此时将 $left$ 右移，继续判断。
 - 如果 $nums[i] + nums[left] + nums[right] \geq target$ ，则说明 $right$ 太大了，应该缩小 $right$ ，然后继续判断。
- 当 $left == right$ 时，区间搜索完毕，继续遍历 $nums[i + 1]$ 。

这种思路使用了两重循环，其中内层循环当 $left == right$ 时循环结束，时间复杂度为 $O(n)$ ，外层循环时间复杂度也是 $O(n)$ 。所以算法的整体时间复杂度为 $O(n^2)$ ，符合题目要求。

思路 1：代码

```
class Solution:
    def threeSumSmaller(self, nums: List[int], target: int) -> int:
        nums.sort()
        size = len(nums)
        res = 0
        for i in range(size):
            left, right = i + 1, size - 1
            while left < right:
                total = nums[i] + nums[left] + nums[right]
                if total < target:
                    res += (right - left)
                    left += 1
                else:
                    right -= 1
        return res
```

思路 1：复杂度分析

- 时间复杂度: $O(n^2)$ 。
- 空间复杂度: $O(\log n)$ 。

0260. 只出现一次的数字 III

- 标签: 位运算、数组
- 难度: 中等

题目链接

- [0260. 只出现一次的数字 III - 力扣](#)

题目大意

描述: 给定一个整数数组 $nums$ 。 $nums$ 中恰好有两个元素只出现一次，其余所有元素均出现两次。

要求: 找出只出现一次的那两个元素。可以按任意顺序返回答案。要求时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

说明:

- $2 \leq nums.length \leq 3 \times 10^4$ 。
- $-2^{31} \leq nums[i] \leq 2^{31} - 1$ 。
- 除两个只出现一次的整数外， $nums$ 中的其他数字都出现两次。

示例:

- 示例 1:

```
输入: nums = [1,2,1,3,2,5]
输出: [3,5]
解释: [5, 3] 也是有效的答案。
```

- 示例 2:

```
输入: nums = [-1,0]
输出: [-1,0]
```

解题思路

思路 1：位运算

求解这道题之前，我们先来看看如何求解「一个数组中除了某个元素只出现一次以外，其余每个元素均出现两次。」即「[136. 只出现一次的数字](#)」问题。

我们可以对所有数不断进行异或操作，最终可得到单次出现的元素。

下面我们再来看这道题。

如果数组中有两个数字只出现一次，其余每个元素均出现两次。那么经过全部异或运算。我们可以得到只出现一次的两个数字的异或结果。

根据异或结果的性质，异或运算中如果某一位上为 1，则说明异或的两个数在该位上是不同的。根据这个性质，我们将数字分为两组：

1. 一组是和该位为 0 的数字，
2. 一组是该位为 1 的数字。

然后将这两组分别进行异或运算，就可以得到最终要求的两个数字。

思路 1：代码

```
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        all_xor = 0
        for num in nums:
            all_xor ^= num
        # 获取所有异或中最低位的 1
        mask = 1
        while all_xor & mask == 0:
            mask <= 1

        a_xor, b_xor = 0, 0
        for num in nums:
            if num & mask == 0:
                a_xor ^= num
            else:
                b_xor ^= num

        return a_xor, b_xor
```

思路 1：复杂度分析

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

0263. 丑数

- 标签：数学
- 难度：简单

题目链接

- [0263. 丑数 - 力扣](#)

题目大意

给定一个整数 n 。

要求：判断 n 是否为丑数。如果是，则返回 `True`，否则，返回 `False`。

- 丑数：只包含质因数 2、3、5 的正整数。

解题思路

- 如果 $n \leq 0$ ，则 n 必然不是丑数，直接返回 `False`。
- 对 n 分别进行 2、3、5 的整除操作，直到 n 被除完，如果 n 最终为 1，则 n 是丑数，否则不是丑数。

代码

```
class Solution:
    def isUgly(self, n: int) -> bool:
        if n <= 0:
            return False
        factors = [2, 3, 5]
        for factor in factors:
            while n % factor == 0:
                n //= factor

        return n == 1
```

0264. 丑数 II

- 标签：哈希表、数学、动态规划、堆（优先队列）
- 难度：中等

题目链接

- [0264. 丑数 II - 力扣](#)

题目大意

给定一个整数 n 。

要求：找出并返回第 n 个丑数。

- 丑数：只包含质因数 2、3、5 的正整数。

解题思路

动态规划求解。

定义状态 $dp[i]$ 表示第 i 个丑数。

状态转移方程为： $dp[i] = \min(dp[p2] * 2, dp[p3] * 3, dp[p5] * 5)$ ，其中 $p2$ 、 $p3$ 、 $p5$ 分别表示当前 i 中 2、3、5 的质因子数量。

代码

```

class Solution:
    def nthUglyNumber(self, n: int) -> int:
        dp = [1 for _ in range(n)]
        p2, p3, p5 = 0, 0, 0
        for i in range(1, n):
            dp[i] = min(dp[p2] * 2, dp[p3] * 3, dp[p5] * 5)
            if dp[i] == dp[p2] * 2:
                p2 += 1
            if dp[i] == dp[p3] * 3:
                p3 += 1
            if dp[i] == dp[p5] * 5:
                p5 += 1
        return dp[n - 1]

```

0268. 丢失的数字

- 标签：位运算、数组、哈希表、数学、二分查找、排序
- 难度：简单

题目链接

- [0268. 丢失的数字 - 力扣](#)

题目大意

描述：给定一个包含 $[0, n]$ 中 n 个数的数组 nums 。

要求：找出 $[0, n]$ 这个范围内没有出现在数组中的那个数。

说明：

- $n == \text{nums.length}$
- $1 \leq n \leq 10^4$
- $0 \leq \text{nums}[i] \leq n$ 。
- nums 中的所有数字都独一无二。

示例：

- **示例 1：**

```

输入: nums = [3, 0, 1]
输出: 2
解释: n = 3, 因为有 3 个数字, 所以所有的数字都在范围 [0,3] 内。2 是丢失的数字, 因为它没有出现在 nums 中。

```

- **示例 2：**

```

输入: nums = [0, 1]
输出: 2
解释: n = 2, 因为有 2 个数字, 所以所有的数字都在范围 [0,2] 内。2 是丢失的数字, 因为它没有出现在 nums 中。

```

解题思路

$[0, n]$ 的范围有 $n + 1$ 个数（包含 0）。现在给了我们 n 个数，要求找出其中缺失的那个数。

思路 1：哈希表

将 $nums$ 中所有元素插入到哈希表中，然后遍历 $[0, n]$ ，找到缺失的数字。

这里的哈希表也可以用长度为 $n + 1$ 的数组代替。

思路 1：代码

```
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        numSet = set(nums)

        for num in range(len(nums)+1):
            if num not in numSet:
                return num
```

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：数学计算

已知 $[0, n]$ 的求和公式为： $\sum_{i=0}^n i = \frac{n*(n+1)}{2}$ ，则用 $[0, n]$ 的和，减去数组中所有元素的和，就得到了缺失数字。

思路 2：代码

```
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        sum_nums = sum(nums)
        n = len(nums)
        return (n + 1) * n // 2 - sum_nums
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0270. 最接近的二叉搜索树值

- 标签：树、深度优先搜索、二叉搜索树、二分查找、二叉树
- 难度：简单

题目链接

- [0270. 最接近的二叉搜索树值 - 力扣](#)

题目大意

描述：给定一个不为空的二叉搜索树的根节点，以及一个目标值 $target$ 。

要求：在二叉搜索树中找到最接近目标值 $target$ 的数值。

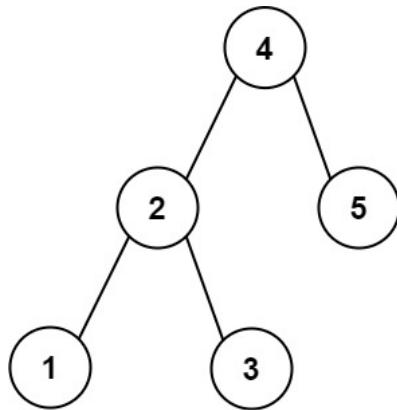
说明：

- 树中节点的数目在范围 $[1, 10^4]$ 内。
- $0 \leq Node.val \leq 10^9$ 。

- $-10^9 \leq target \leq 10^9$ 。

示例：

- 示例 1：



输入: `root = [4, 2, 5, 1, 3], target = 3.714286`
输出: `4`

- 示例 2：

输入: `root = [1], target = 4.428571`
输出: `1`

解题思路

思路 1：二分查找算法

题目中最接近目标值 $target$ 的数值指的就是与 $target$ 相减绝对值最小的数值。

而且根据二叉搜索树的性质，我们可以利用二分搜索的方式，查找与 $target$ 相减绝对值最小的数值。具体做法为：

- 定义一个变量 $closest$ 表示与 $target$ 最接近的数值，初始赋值为根节点的值 $root.val$ 。
- 判断当前节点的值域 $closest$ 值哪个更接近 $target$ ，如果当前值更接近，则更新 $closest$ 。
- 如果 $target <$ 当前节点值，则从当前节点的左子树继续查找。
- 如果 $target \geq$ 当前节点值，则从当前节点的右子树继续查找。

思路 1：代码

```

class Solution:
    def closestValue(self, root: TreeNode, target: float) -> int:
        closest = root.val
        while root:
            if abs(target - root.val) < abs(target - closest):
                closest = root.val
            if target < root.val:
                root = root.left
            else:
                root = root.right
        return closest
  
```

思路 1：复杂度分析

- 时间复杂度： $O(\log n)$ ，其中 n 为二叉搜索树的节点个数。
- 空间复杂度： $O(1)$ 。

0278. 第一个错误的版本

- 标签：数组、二分查找
- 难度：简单

题目链接

- [0278. 第一个错误的版本 - 力扣](#)

题目大意

描述：给你一个整数 n ，代表已经发布的版本号。还有一个用于检测版本是否出错的接口 `isBadVersion(version)`。

要求：找出第一次出错的版本号 bad 。

说明：

- 要求尽可能减少对 `isBadVersion(version)` 接口的调用。
- $1 \leq bad \leq n \leq 2^{31} - 1$ 。

示例：

- 示例 1：

```
输入: n = 5, bad = 4
输出: 4
解释:
调用 isBadVersion(3) -> false
调用 isBadVersion(5) -> true
调用 isBadVersion(4) -> true
所以, 4 是第一个错误的版本。
```

- 示例 2：

```
输入: n = 1, bad = 1
输出: 1
```

解题思路

思路 1：二分查找

题目要求尽可能减少对 `isBadVersion(version)` 接口的调用，所以不能对每个版本都调用接口，而是应该将接口调用的次数降到最低。

可以注意到：如果检测某个版本不是错误版本时，则该版本之前的所有版本都不是错误版本。而当某个版本是错误版本时，则该版本之后的所有版本都是错误版本。我们可以利用这样的性质，在 $[1, n]$ 的区间内使用二分查找方法，从而在 $O(\log n)$ 时间复杂度内找到第一个出错误的版本。

思路 1：代码

```
class Solution:
    def firstBadVersion(self, n):
        left = 1
        right = n
        while left < right:
            mid = (left + right) // 2
            if isBadVersion(mid):
                right = mid
            else:
                left = mid + 1
        return left
```

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。

0279. 完全平方数

- 标签：广度优先搜索、数学、动态规划
- 难度：中等

题目链接

- [0279. 完全平方数 - 力扣](#)

题目大意

描述：给定一个正整数 n 。从中找到若干个完全平方数（比如 $1, 4, 9, 16 \dots$ ），使得它们的和等于 n 。

要求：返回和为 n 的完全平方数的最小数量。

说明：

- $1 \leq n \leq 10^4$ 。

示例：

- 示例 1：

```
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4
```

- 示例 2：

```
输入: n = 13
输出: 2
解释: 13 = 4 + 9
```

解题思路

暴力枚举思路：对于小于 n 的完全平方数，直接暴力枚举所有可能的组合，并且找到平方数个数最小的一个。

并且对于所有小于 n 的完全平方数 ($k = 1, 4, 9, 16, \dots$)，存在公式： $ans(n) = \min(ans(n - k) + 1), k = 1, 4, 9, 16\dots$

即： n 的完全平方数的最小数量 == $n - k$ 的完全平方数的最小数量 + 1。

我们可以使用递归解决这个问题。但是因为重复计算了中间解，会产生堆栈溢出。

那怎么解决重复计算问题和避免堆栈溢出？

我们可以转换一下思维。

1. 将 n 作为根节点，构建一棵多叉树。
2. 从 n 节点出发，如果一个小于 n 的数刚好与 n 相差一个平方数，则以该数为值构造一个节点，与 n 相连。

那么求解和为 n 的完全平方数的最小数量就变成了求解这棵树从根节点 n 到节点 0 的最短路径，或者说树的最小深度。

这个过程可以通过广度优先搜索来做。

思路 1：广度优先搜索

1. 定义 $visited$ 为标记访问节点的 set 集合变量，避免重复计算。定义 $queue$ 为存放节点的队列。使用 $count$ 表示为树的最小深度，也就是和为 n 的完全平方数的最小数量。
2. 首先，我们将 n 标记为已访问，即 `visited.add(n)`。并将其加入队列 $queue$ 中，即 `queue.append(n)`。
3. 令 $count$ 加 1，表示最小深度加 1。然后依次将队列中的节点值取出。
4. 对于取出的节点值 $value$ ，遍历可能出现的平方数（即遍历 $[1, \sqrt{value} + 1]$ 中的数）。
5. 每次从当前节点值减去一个平方数，并将减完的数加入队列。
 - i. 如果此时的数等于 0，则满足题意，返回当前树的最小深度。
 - ii. 如果此时的数不等于 0，则将其加入队列，继续查找。

思路 1：代码

```
class Solution:
    def numSquares(self, n: int) -> int:
        if n == 0:
            return 0

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

        visited.add(n)
        queue.append(n)

        count = 0
        while queue:
            // 最少步数
            count += 1
            size = len(queue)
            for _ in range(size):
                value = queue.pop()
                for i in range(1, int(math.sqrt(value)) + 1):
                    x = value - i * i
                    if x == 0:
                        return count
                    if x not in visited:
                        queue.appendleft(x)
                        visited.add(x)
```

思路 1：复杂度分析

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

思路 2：动态规划

我们可以将这道题转换为「完全背包问题」中恰好装满背包的方案数问题。

1. 将 $k = 1, 4, 9, 16, \dots$ 看做是 k 种物品，每种物品都可以无限次使用。
2. 将 n 看做是背包的装载上限。
3. 这道题就变成了，从 k 种物品中选择一些物品，装入装载上限为 n 的背包中，恰好装满背包最少需要多少件物品。

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 $dp[w]$ 表示为：从完全平方数中挑选一些数，使其和恰好凑成 w ，最少需要多少个完全平方数。

3. 状态转移方程

$$dp[w] = \min\{dp[w], dp[w - num] + 1\}$$

4. 初始条件

- 恰好凑成和为 0，最少需要 0 个完全平方数。
- 默认情况下，在不使用完全平方数时，都不能恰好凑成和为 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， $dp[w]$ 表示为：将物品装入装载上限为 w 的背包中，恰好装满背包，最少需要多少件物品。所以最终结果为 $dp[n]$ 。

1. 如果 $dp[n] \neq n + 1$ ，则说明： $dp[n]$ 为装入装载上限为 n 的背包，恰好装满背包，最少需要的物品数量，则返回 $dp[n]$ 。
2. 如果 $dp[n] = n + 1$ ，则说明：无法恰好装满背包，则返回 -1 。因为 n 肯定能由 n 个 1 组成，所以这种情况并不会出现。

思路 2：代码

```
class Solution:
    def numSquares(self, n: int) -> int:
        dp = [n + 1 for _ in range(n + 1)]
        dp[0] = 0

        for i in range(1, int(sqrt(n)) + 1):
            num = i * i
            for w in range(num, n + 1):
                dp[w] = min(dp[w], dp[w - num] + 1)

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

思路 2：复杂度分析

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

0283. 移动零

- 标签：数组、双指针
- 难度：简单

题目链接

- [0283. 移动零 - 力扣](#)

题目大意

描述：给定一个数组 $nums$ 。

要求：将所有 0 移动到末尾，并保持原有的非 0 数字的相对顺序。

说明：

- 只能在原数组上进行操作。
- $1 \leq \text{nums.length} \leq 10^4$ 。
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$ 。

示例：

- 示例 1：

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

- 示例 2：

```
输入: nums = [0]
输出: [0]
```

解题思路

思路 1：冒泡排序（超时）

冒泡排序的思想，就是通过相邻元素的比较与交换，使得较大元素从前面移到后面。

我们可以借用冒泡排序的思想，将值为 0 的元素移动到数组末尾。

因为数据规模为 10^4 ，而冒泡排序的时间复杂度为 $O(n^2)$ 。所以这种做法会导致超时。

思路 1：代码

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        for i in range(len(nums)):
            for j in range(len(nums) - i - 1):
                if nums[j] == 0 and nums[j + 1] != 0:
                    nums[j], nums[j + 1] = nums[j + 1], nums[j]
```

思路 1：复杂度分析

- 时间复杂度： $O(n^2)$ 。
- 空间复杂度： $O(1)$ 。

思路 2：快慢指针

- 使用两个指针 $slow$, $fast$ 。 $slow$ 指向处理好的非 0 数字数组的尾部， $fast$ 指针指向当前待处理元素。
- 不断向右移动 $fast$ 指针，每次移动到非零数，则将左右指针对应的数交换，交换同时将 $slow$ 右移。
- 此时， $slow$ 指针左侧均为处理好的非零数，而从 $slow$ 指针指向的位置开始， $fast$ 指针左边为止都为 0。

遍历结束之后，则所有 0 都移动到了右侧，且保持了非零数的相对位置。

思路 2：代码

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        slow = 0
        fast = 0
        while fast < len(nums):
            if nums[fast] != 0:
                nums[slow], nums[fast] = nums[fast], nums[slow]
                slow += 1
            fast += 1
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

0285. 二叉搜索树中的中序后继

- 标签：树、深度优先搜索、二叉搜索树、二叉树
- 难度：中等

题目链接

- [0285. 二叉搜索树中的中序后继 - 力扣](#)

题目大意

给定一棵二叉搜索树的根节点 `root`。

要求：按中序遍历顺序将其重新排列为一棵递增顺序搜索树，使树中最左边的节点成为树的根节点，并且每个节点没有左子节点，只有一个右子节点。

解题思路

可以分为两步：

1. 中序遍历二叉搜索树，将节点先存储到列表中。
2. 将列表中的节点构造成一棵递增顺序搜索树。

中序遍历直接按照 `左 -> 根 -> 右` 的顺序递归遍历，然后将遍历的节点存储到 `res` 中。

构造递增顺序搜索树，则用 `head` 保存头节点位置。遍历列表中的每个节点，将其左右指针先置空，再将其连接在上一个节点的右子节点上。

最后返回 `head.right` 即可。

代码

```

class Solution:
    def inOrder(self, root, res):
        if not root:
            return
        self.inOrder(root.left, res)
        res.append(root)
        self.inOrder(root.right, res)

    def increasingBST(self, root: TreeNode) -> TreeNode:
        res = []
        self.inOrder(root, res)

        if not res:
            return
        head = TreeNode(-1)
        cur = head
        for node in res:
            node.left = node.right = None
            cur.right = node
            cur = cur.right
        return head.right

```

0286. 墙与门

- 标签：广度优先搜索、数组、矩阵
- 难度：中等

题目链接

- [0286. 墙与门 - 力扣](#)

题目大意

给定一个 $m * n$ 的二维网络 `rooms`。其中每个元素有三种初始值：

- `-1` 表示墙或者障碍物
- `0` 表示一扇门
- `INF` 表示为一个空的房间。这里用 $2^{31} = 2147483647$ 表示 `INF`。通往门的距离总是小于 2^{31} 。

要求：给每个空房间填上该房间到最近的门的距离，如果无法到达门，则填 `INF`。

解题思路

从每个表示门开始，使用广度优先搜索去照门。因为广度优先搜索保证我们在搜索 `dist + 1` 距离的位置时，距离为 `dist` 的位置都已经搜索过了。所以每到达一个房间的时候一定是最短距离。

代码

```

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

        INF = 2147483647
        rows = len(rooms)
        if rows == 0:
            return
        cols = len(rooms[0])

        directions = {(1, 0), (-1, 0), (0, 1), (0, -1)}
        queue = []
        for i in range(rows):
            for j in range(cols):
                if rooms[i][j] == 0:
                    queue.append((i, j, 0))

        while queue:
            i, j, dist = queue.pop(0)
            for direction in directions:
                new_i = i + direction[0]
                new_j = j + direction[1]
                if 0 <= new_i < rows and 0 <= new_j < cols and rooms[new_i][new_j] == INF:
                    rooms[new_i][new_j] = dist + 1
                    queue.append((new_i, new_j, dist + 1))

```

0287. 寻找重复数

- 标签: 位运算、数组、双指针、二分查找
- 难度: 中等

题目链接

- [0287. 寻找重复数 - 力扣](#)

题目大意

描述: 给定一个包含 $n + 1$ 个整数的数组 $nums$, 里边包含的值都在 $1 \sim n$ 之间。可知至少存在一个重复的整数。

要求: 假设 $nums$ 中只存在一个重复的整数, 要求找出这个重复的数。

说明:

- $1 \leq n \leq 10^5$ 。
- $nums.length == n + 1$ 。
- $1 \leq nums[i] \leq n$ 。
- 要求使用空间复杂度为常数级 $O(1)$, 时间复杂度小于 $O(n^2)$ 的解决方法。

示例:

- **示例 1:**

```

输入: nums = [1,3,4,2,2]
输出: 2

```

- **示例 2:**

输入: `nums = [3, 1, 3, 4, 2]`
 输出: `3`

解题思路

思路 1：二分查找

利用二分查找的思想。

1. 使用两个指针 `left`, `right`。`left` 指向 1, `right` 指向 n 。
2. 将区间 $[1, n]$ 分为 $[left, mid]$ 和 $[mid + 1, right]$ 。
3. 对于中间数 `mid`, 统计 `nums` 中小于等于 `mid` 的数个数 `cnt`。
4. 如果 $cnt \leq mid$, 则重复数一定不会出现在左侧区间, 那么从右侧区间开始搜索。
5. 如果 $cnt > mid$, 则重复数出现在左侧区间, 则从左侧区间开始搜索。

思路 1：代码

```
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        n = len(nums)
        left = 1
        right = n - 1
        while left < right:
            mid = left + (right - left) // 2
            cnt = 0
            for num in nums:
                if num <= mid:
                    cnt += 1

            if cnt <= mid:
                left = mid + 1
            else:
                right = mid

        return left
```

思路 1：复杂度分析

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

0288. 单词的唯一缩写

- 标签: 设计、数组、哈希表、字符串
- 难度: 中等

题目链接

- [0288. 单词的唯一缩写 - 力扣](#)

题目大意

单词缩写规则: <起始字母><中间字母><结尾字母>。如果单词长度不超过 2, 则单词本身就是缩写。

举例:

- `dog --> d1g` : 第一个字母 `d`, 最后一个字母 `g`, 中间隔着 1 个字母。

- `internationalization` --> `i18n`：第一个字母 `i`，最后一个字母 `n`，中间隔着 18 个字母。
- `it` --> `it`：单词只有两个字符，它就是它自身的缩写。

要求实现 `ValidWordAbbr` 类：

- `ValidWordAbbr(dictionary: List[str]):` 使用单词字典初始化对象
- `def isUnique(self, word: str) -> bool:`
 - 如果字典 `dictionary` 中没有其他单词的缩写与该单词 `word` 的缩写相同，返回 `True`。
 - 如果字典 `dictionary` 中所有与该单词 `word` 的缩写相同的单词缩写都与 `word` 相同。

解题思路

将相同缩写的单词进行分类，利用哈希表进行存储。键值对格式为 缩写：该缩写对应的 `word` 列表。

然后初始化的时候，将 `dictionary` 里的单词按照缩写进行哈希表存储。

在判断的时候，先判断单词 `word` 的缩写是否能在哈希表中找到对应的映射关系。

- 如果 `word` 的缩写 `abbr` 没有在哈希表中，则返回 `True`。
- 如果 `word` 的缩写 `abbr` 在哈希表中：
 - 如果缩写 `abbr` 对应的字符串列表只有一个字符串，并且就是 `word`，则返回 `True`。
 - 否则返回 `False`。
- 不满足上述要求也返回 `False`。

代码

```
def isUnique(self, word: str) -> bool:
    if len(word) <= 2:
        abbr = word
    else:
        abbr = word[0] + chr(len(word)-2) + word[-1]
    if abbr not in self.abbr_dict:
        return True
    if len(set(self.abbr_dict[abbr])) == 1 and word in set(self.abbr_dict[abbr]):
        return True
    return False
```

0289. 生命游戏

- 标签：数组、矩阵、模拟
- 难度：中等

题目链接

- [0289. 生命游戏 - 力扣](#)

题目大意

描述：给定一个 $m \times n$ 大小的二维数组 `board`，每一个格子都可以看做是一个细胞。每个细胞都有一个初始状态：1 代表活细胞，0 代表死细胞。每个细胞与其相邻的八个位置（水平、垂直、对角线）细胞遵循以下生存规律：

- 如果活细胞周围八个位置的活细胞数少于 2 个，则该位置活细胞死亡；
- 如果活细胞周围八个位置有 2 个或 3 个活细胞，则该位置活细胞仍然存活；
- 如果活细胞周围八个位置有超过 3 个活细胞，则该位置活细胞死亡；
- 如果死细胞周围正好有 3 个活细胞，则该位置死细胞复活。

二维数组代表的下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的。其中细胞的出生和死亡是同时发生的。

现在给定 $m \times n$ 的二维数组 $board$ 的当前状态。

要求：返回下一个状态。

说明：

- $m == board.length$ 。
- $n == board[i].length$ 。
- $1 \leq m, n \leq 25$ 。
- $board[i][j]$ 为 0 或 1。
- 进阶：
 - 你可以使用原地算法解决本题吗？请注意，面板上所有格子需要同时被更新：你不能先更新某些格子，然后使用它们的更新后的值再更新其他格子。
 - 本题中，我们使用二维数组来表示面板。原则上，面板是无限的，但当活细胞侵占了面板边界时会造成问题。你将如何解决这些问题？

示例：

- 示例 1：

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

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

```
输入: board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]
输出: [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]
```

- 示例 2：

1	1
1	0

1	1
1	1

```
输入: board = [[1,1],[1,0]]
输出: [[1,1],[1,1]]
```

解题思路

思路 1：模拟

因为下一个状态隐含了过去细胞的状态，所以不能直接在原二维数组上直接进行修改。细胞的状态总共有四种情况：

- 死细胞 \rightarrow 死细胞，即 $0 \rightarrow 0$ 。
- 死细胞 \rightarrow 活细胞，即 $0 \rightarrow 1$ 。
- 活细胞 \rightarrow 活细胞，即 $1 \rightarrow 1$ 。
- 活细胞 \rightarrow 死细胞，即 $1 \rightarrow 0$ 。

死细胞 -> 死细胞，活细胞 -> 活细胞，不会对前后状态造成影响，所以主要考虑另外两种情况。我们把活细胞 -> 死细胞暂时标记为 -1 ，并且统计每个细胞周围活细胞数量时，使用绝对值统计，这样 $\text{abs}(-1)$ 也可以暂时标记为活细胞。然后把死细胞 -> 活细胞暂时标记为 2 ，这样判断的时候也不会统计上去。然后开始遍历。

- 遍历二维数组的每一个位置。并对该位置遍历周围八个位置，计算出八个位置上的活细胞数量。
 - 如果此位置是活细胞，并且周围活细胞少于 2 个或超过 3 个，则将其暂时标记为 -1 ，意为此细胞死亡。
 - 如果此位置是死细胞，并且周围有 3 个活细胞，则将暂时标记为 2 ，意为此细胞复活。
- 遍历完之后，再次遍历一遍二维数组，如果该位置为 -1 ，将其赋值为 0 ，如果该位置为 2 ，将其赋值为 1 。

思路 1：代码

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

        directions = {(1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1)}

        rows = len(board)
        cols = len(board[0])

        for row in range(rows):
            for col in range(cols):
                lives = 0
                for direction in directions:
                    new_row = row + direction[0]
                    new_col = col + direction[1]

                    if 0 <= new_row < rows and 0 <= new_col < cols and abs(board[new_row][new_col]) == 1:
                        lives += 1
                if board[row][col] == 1 and (lives < 2 or lives > 3):
                    board[row][col] = -1
                if board[row][col] == 0 and lives == 3:
                    board[row][col] = 2

        for row in range(rows):
            for col in range(cols):
                if board[row][col] == -1:
                    board[row][col] = 0
                elif board[row][col] == 2:
                    board[row][col] = 1
```

思路 1：复杂度分析

- 时间复杂度： $O(m \times n)$ ，其中 m 、 n 分别为 $board$ 的行数和列数。
- 空间复杂度： $O(m \times n)$ 。

0290. 单词规律

- 标签：哈希表、字符串
- 难度：简单

题目链接

- [0290. 单词规律 - 力扣](#)

题目大意

给定一种规律 `pattern` 和一个字符串 `str`，判断 `str` 是否完全匹配相同的规律。

- 完全匹配相同的规律：pattern 的每个字母和字符串 str 中的每个非空单词之间存在这双向连接的对应规律。
- 比如：pattern = "abba", str = "dog cat cat dog", 其对应关系为： a <=> dog, b <=> cat

解题思路

这道题要求判断规律串中的字符与所给字符串中的非空单词，是否是一一对应的。即每个字符都能映射到对应的非空单词，每个非空单词也能映射为字符。

考虑使用两个哈希表，一个用来存储字符到非空单词的映射，另一个用来存储非空单词到字符的映射。

遍历 pattern 中的字符：

- 如果字符出现在第一个字典中，且字典中的值不等于对应的非空单词，则返回 False。
- 如果单词出现在第二个字典中，且字典中的值不等于对应的字符，则返回 False。
- 如果遍历完仍没发现不满足要求的情况，则返回 True。

代码

```
class Solution:
    def wordPattern(self, pattern: str, s: str) -> bool:
        pattern_dict = dict()
        word_dict = dict()
        words = s.split()

        if len(pattern) != len(words):
            return False

        for i in range(len(words)):
            p = pattern[i]
            word = words[i]
            if p in pattern_dict and pattern_dict[p] != word:
                return False
            if word in word_dict and word_dict[word] != p:
                return False
            pattern_dict[p] = word
            word_dict[word] = p
        return True
```

0292. Nim 游戏

- 标签：脑筋急转弯、数学、博弈
- 难度：简单

题目链接

- [0292. Nim 游戏 - 力扣](#)

题目大意

两个人玩 Nim 游戏。游戏规则是这样的：

- 桌上有一堆石子，两个人轮流从石子堆中拿走 1~3 块石头。拿掉最后一块石头的人就是获胜者。
- 假如每个人都尽可能的想赢得比赛，所以每一轮都是最优解。

现在给定一个整数 n 代表石头数目。如果你作为先手，问最终能否赢得比赛。

解题思路

假设石子的数量为 1~3，那么我作为先手，肯定第一次就将所有的石子都拿完了，所以肯定能赢。

假设石子的数量为 4，那么我作为先手，无论第一次拿走 1、2、3 块石头，都不能拿完，而第二个人再拿的时候，会直接将剩下的石头一次性全拿走，所以肯定不会赢。

如果石子数量多于 4，那么我作为先手，为了赢，应该尽可能使得本轮拿走后的石子数为 4，这样对手拿完一次之后，自己肯定会获胜。

所以石子树为 5、6、7 块的时候，我可以分别拿走 1、2、3 块石头，使得剩下的石头数为 4，从而在下一轮获得胜利。

如果石子数为 8 块的时候，我无论怎么拿都不能使剩下石子为 4。而对方又会利用这个机会使得他拿走之后的石子数变为 4，从而使我失败。

所以，很显然：当 n 不是 4 的整数倍时，我一定赢得比赛。当 n 为 4 的整数倍时，我一定赢不了比赛。

代码

```
class Solution:
    def canWinNim(self, n: int) -> bool:
        return n % 4 != 0
```

0295. 数据流的中位数

- 标签：设计、双指针、数据流、排序、堆（优先队列）
- 难度：困难

题目链接

- [0295. 数据流的中位数 - 力扣](#)

题目大意

要求：设计一个支持一下两种操作的数据结构：

- `void addNum(int num)`：从数据流中添加一个整数到数据结构中。
- `double findMedian()`：返回目前所有元素的中位数。

解题思路

使用一个大顶堆 `queMax` 记录大于中位数的数，使用一个小顶堆 `queMin` 小于中位数的数。

- 当添加元素数量为偶数：`queMin` 和 `queMax` 中元素数量相同，则中位数为它们队头的平均值。
- 当添加元素数量为奇数：`queMin` 中的数比 `queMax` 多一个，此时中位数为 `queMin` 的队头。

为了满足上述条件，在进行 `addNum` 操作时，我们应当分情况处理：

- `num > max{queMin}`：此时 `num` 大于中位数，将该数添加到大顶堆 `queMax` 中。新的中位数将大于原来的中位数，所以可能需要将 `queMax` 中的最小数移动到 `queMin` 中。
- `num <= max{queMin}`：此时 `num` 小于中位数，将该数添加到小顶堆 `queMin` 中。新的中位数将小于等于原来的中位数，所以可能需要将 `queMin` 中的最大数移动到 `queMax` 中。

代码

```

import heapq

class MedianFinder:

    def __init__(self):
        .....
        initialize your data structure here.
        .....

        self.queMin = list()
        self.queMax = list()

    def addNum(self, num: int) -> None:
        if not self.queMin or num < -self.queMin[0]:
            heapq.heappush(self.queMin, -num)
            if len(self.queMax) + 1 < len(self.queMin):
                heapq.heappush(self.queMax, -heapq.heappop(self.queMin))
        else:
            heapq.heappush(self.queMax, num)
            if len(self.queMax) > len(self.queMin):
                heapq.heappush(self.queMin, -heapq.heappop(self.queMax))

    def findMedian(self) -> float:
        if len(self.queMin) > len(self.queMax):
            return -self.queMin[0]
        return (-self.queMin[0] + self.queMax[0]) / 2

```

0297. 二叉树的序列化与反序列化

- 标签：树、深度优先搜索、广度优先搜索、设计、字符串、二叉树
- 难度：困难

题目链接

- [0297. 二叉树的序列化与反序列化 - 力扣](#)

题目大意

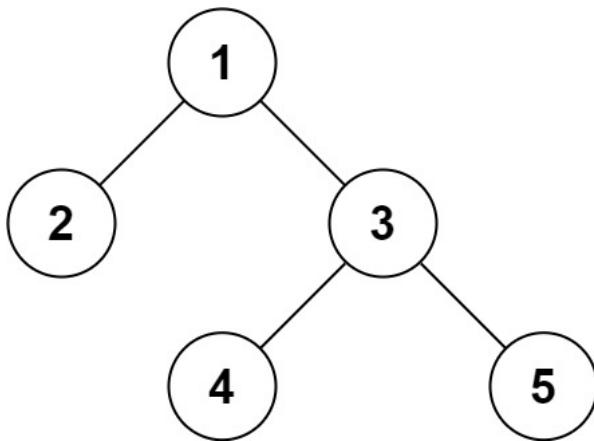
要求：设计一个算法，来实现二叉树的序列化与反序列化。

说明：

- 不限定序列化 / 反序列化算法执行逻辑，只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。
- 树中结点数在范围 $[0, 10^4]$ 内。
- $-1000 \leq Node.val \leq 1000$ 。

示例：

- 示例 1：



输入: `root = [1,2,3,null,null,4,5]`

输出: `[1,2,3,null,null,4,5]`

- 示例 2:

输入: `root = [1,2]`

输出: `[1,2]`

解题思路

思路 1：深度优先搜索

1. 序列化：将二叉树转为字符串数据表示

- 按照前序顺序递归遍历二叉树，并将根节点跟左右子树的值链接起来（中间用 `,` 隔开）。

注意：如果遇到空节点，则将其标记为 `None`，这样在反序列化时才能唯一确定一棵二叉树。

2. 反序列化：将字符串数据转为二叉树结构

- 先将字符串按 `,` 分割成数组。然后递归处理每一个元素。

- 从数组左侧取出一个元素。

i. 如果当前元素为 `None`，则返回 `None`。

ii. 如果当前元素不为空，则新建一个二叉树节点作为根节点，保存值为当前元素值。并递归遍历左右子树，不断重复从数组中取出元素，进行判断。

- 最后返回当前根节点。

思路 1：代码

```
class Codec:

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

        :type root: TreeNode
        :rtype: str
        """
        if not root:
            return 'None'
        return str(root.val) + ',' + self.serialize(root.left)) + ',' + self.serialize(root.right))

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

        :type data: str
        :rtype: TreeNode
        """
        def dfs(datalist):
            val = datalist.pop(0)
            if val == 'None':
                return None
            root = TreeNode(int(val))
            root.left = dfs(datalist)
            root.right = dfs(datalist)
            return root

        datalist = data.split(',')
        return dfs(datalist)
```

思路 1：复杂度分析

- 时间复杂度: $O(n)$, 其中 n 为二叉树的节点数。
- 空间复杂度: $O(n)$ 。