

第一部分 算法

一、贪心算法

1.1 算法简介

本章为附加算法，对应题目的题号是:455.分发饼干，135.分发糖果，435.无重叠区间。

(1) 基本定义

贪心算法 (Greedy Algorithm) 是一种在每一步选择中都采取当前状态下的最优决策 (即局部最优解)，并期望通过一系列的局部最优选择来达到全局最优解的算法策略。

以找零钱问题为例：在一个国家的货币体系中有 1 元、5 角、1 角的硬币，要找零 2 元 3 角，怎样用最少的硬币数量来完成找零。贪心算法会每次都优先选择面额最大的硬币，直到不能再选

(2) 应用场景

活动安排、背包问题、哈弗曼编码

(3) python模板

区间问题：1.首先按照区间的起始端点或末尾端点排序，2.其次遍历区间，对区间进行合并、选择、覆盖等操作。3.最后返回结果。下方以区间融合问题为模板

```
def merge_intervals(intervals):
    if not intervals:
        return []
    # 按照区间的起始端点排序
    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]
    for interval in intervals[1:]:
        # 如果当前区间的起始端点小于等于上一个合并区间的结束端点
        if interval[0] <= merged[-1][1]:
            # 合并区间，更新结束端点
            merged[-1][1] = max(merged[-1][1], interval[1])
        else:
            # 不重叠，添加新的区间
            merged.append(interval)
    return merged
```

1.2 题目

455. 分发饼干

题目描述

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i ，都有一个胃口值 $g[i]$ ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 $s[j]$ 。如果 $s[j] \geq g[i]$ ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是满足尽可能多的孩子，并输出这个最大数值。

示例 1:

输入: `g = [1,2,3]`, `s = [1,1]`

输出: 1

解释:

你有三个孩子和两块小饼干, 3 个孩子的胃口值分别是: 1,2,3。

虽然你有两块小饼干, 由于他们的尺寸都是 1, 你只能让胃口值是 1 的孩子满足。

所以你应该输出 1。

题解

本题的贪心策略是每一次让孩子吃能满足胃口, 尺寸最小的饼干。因此首先对胃口值和饼干尺寸排序; 其次从小到大遍历两个数组, 若满足要求则 i, j 都+1, 否则只有 $j+1$ 。

```
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        g.sort()
        s.sort()
        i = 0
        j = 0
        while i < len(g) and j < len(s):
            if g[i] <= s[j]:
                i += 1
                j += 1
            else:
                j += 1
        return i
```

135. 分发糖果

题目描述

n 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。你需要按照以下要求, 给这些孩子分发糖果:

- 每个孩子至少分配到 1 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。

请你给每个孩子分发糖果, 计算并返回需要准备的 **最少糖果数目**。

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。

题解

本题的贪心策略是: 首先从左到右遍历, 当右侧孩子评分高于左侧时, 则右侧孩子糖果为左侧孩子糖果+1; 其次从右到左遍历, 当左侧孩子评分高于右侧时, 则左侧孩子糖果为其和右侧孩子糖果+1的最大值。

```

class Solution:
    def candy(self, ratings: List[int]) -> int:
        n = len(ratings)
        nums = [1] * n
        for i in range(n-1):
            if ratings[i] < ratings[i+1]:
                nums[i+1] = nums[i] + 1
        for i in range(n-1,0,-1):
            if ratings[i-1] > ratings[i]:
                nums[i-1] = max(nums[i-1],nums[i]+1)
        return sum(nums)

```

435. 无重叠区间

题目描述

给定一个区间的集合 `intervals`，其中 `intervals[i] = [starti, endi]`。返回 *需要移除区间的最小数量，使剩余区间互不重叠。*

注意 只在一点上接触的区间是 **不重叠的**。例如 `[1, 2]` 和 `[2, 3]` 是不重叠的。

示例 1:

输入: `intervals = [[1,2],[2,3],[3,4],[1,3]]`
 输出: 1
 解释: 移除 `[1,3]` 后，剩下的区间没有重叠。

示例 2:

输入: `intervals = [[1,2], [1,2], [1,2]]`
 输出: 2
 解释: 你需要移除两个 `[1,2]` 来使剩下的区间没有重叠。

题解

本题可以把区间理解为活动时间，为了保留更多的活动，需要尽可能把结束时间早的活动留下。因此本题的贪心策略为：首先按区间的末尾端点排序，其次舍去和前一区间有冲突的区间（开始时间小于前一区间的结束时间）。

```

class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x:x[1])
        pre_end = intervals[0][1]
        num = 0
        for interval in intervals[1:]:
            if interval[0] < pre_end:
                num += 1
            else:
                pre_end = interval[1]
        return num

```

二、双指针

本章对应的题目为“双指针”、“滑动窗口”。

2.1 算法简介

(1) 基本定义

双指针是一种在遍历数据结构（如数组、链表等）时，使用两个指针来辅助解决问题的算法。这两个指针可以同向移动，也可以相向移动，通过巧妙地移动指针来降低时间复杂度，更高效地解决问题。

若两个指针是相向移动的，一般用于在排好序的数组中搜索目标，在面试150题中归属于“双指针”；若两个指针是同向移动的，一般用于搜索区间，在面试150题中归属于“滑动窗口”，即两个指针包围的区域被称为窗口。

(2) 应用场景

数组遍历与查找、字符串处理、链表操作、滑动窗口

(3) python模板

双指针问题的难点主要设置移动指针的条件并返回结果。

双指针——两数之和

```
def two_sum(nums, target):
    # 初始化左右指针，分别指向数组的起始位置和末尾位置
    left, right = 0, len(nums) - 1
    # 当左指针小于右指针时，继续循环
    while left < right:
        current_sum = nums[left] + nums[right] # 计算当前两数和
        # 如果两数和与目标值相等，返回左右指针
        if current_sum == target:
            return [left, right]
        # 如果两数和小于目标值，通过左指针加一，增加两数和
        elif current_sum < target:
            left += 1
        # 反之通过右指针减一，减小两数和
        else:
            right -= 1
    return []
```

滑动窗口模板——长度最小的子数组

```
def minSubArrayLen(target, nums):
    # 初始化数组长度
    n = len(nums)
    # 初始化最小长度为无穷大
    min_len = float('inf')
    # 初始化左指针，用于收缩窗口
    left = 0
    # 初始化当前窗口内元素的和
    current_sum = 0

    # 外层循环扩展右边界，内层循环收缩左边界
```

```

# 右指针从 0 开始遍历数组
for right in range(n):
    # 扩大窗口，将当前右指针指向的元素加入窗口和中
    current_sum += nums[right]
    # 当窗口内元素的和大于等于目标值时，开始收缩窗口
    while left <= right and current_sum >= target:
        # 计算当前窗口的长度
        window_len = right - left + 1
        # 更新最小长度
        min_len = min(min_len, window_len)
        # 收缩窗口，将左指针指向的元素从窗口和中移除
        current_sum -= nums[left]
        # 左指针右移一位
        left += 1
# 如果最小长度仍为无穷大，说明没有找到满足条件的子数组，返回 0
if min_len == float('inf'):
    return 0
# 否则返回最小长度
return min_len

```

2.2 题目（双指针）

125. 验证回文串

题目描述

如果在将所有大写字符转换为小写字符、并移除所有非字母数字字符之后，短语正着读和反着读都一样。则可以认为该短语是一个 **回文串**。

字母和数字都属于字母数字字符。

给你一个字符串 `s`，如果它是 **回文串**，返回 `true`；否则，返回 `false`。

示例 1：

输入：s = "A man, a plan, a canal: Panama"

输出：true

解释："amanaplanacanalpanama" 是回文串。

示例 2：

输入：s = "race a car"

输出：false

解释："raceacar" 不是回文串。

示例 3

输入：s = ""

输出：true

解释：在移除非字母数字字符之后，s 是一个空字符串 ""。

由于空字符串正着反着读都一样，所以是回文串。

题解

本题首先要将字符串中字母统一改成小写，并去除空格；其次在左右指针相向移动时，若两指针指向字母不同，则返回错误，当指针相遇时，则返回正确。

```
class Solution:
    def isPalindrome(self, s: str) -> bool:
        s = ''.join(ch.lower() for ch in s if ch.isalnum())
        n = len(s)
        left, right = 0, n-1

        while(left<right):
            if s[left] != s[right]:
                return False
            else:
                left += 1
                right -= 1
        return True
```

392. 判断子序列

题目描述

给定字符串 **s** 和 **t**，判断 **s** 是否为 **t** 的子序列。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace" 是 "abcde" 的一个子序列，而 "aec" 不是）。

示例 1:

输入: s = "abc", t = "ahbgdc"
输出: true

示例 2:

输入: s = "axc", t = "ahbgdc"
输出: false

题解

本题的双指针分别指向字符串s和t的起始位置。两个指针同向移动，当指向字符相同时，各移动一步，否则只有字符串t的指针left2移动。当left1 = n1时，证明s的字母在t中都出现过，返回True。

```
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        left1 = left2 = 0
        n1, n2 = len(s), len(t)
        while left1 < n1 and left2 < n2:
            if s[left1] == t[left2]:
                left1 += 1
                left2 += 1
            else:
                left2 += 1
        return left1 == n1
```

```
if left1 == n1:
    return True
else:
    return False
```

167. 两数之和II - 输入有序数组

题目描述

给你一个下标从 1 开始的整数数组 `numbers`，该数组已按 **非递减顺序排列**，请你从数组中找出满足相加之和等于目标数 `target` 的两个数。如果设这两个数分别是 `numbers[index1]` 和 `numbers[index2]`，则 $1 \leq \text{index1} < \text{index2} \leq \text{numbers.length}$ 。

以长度为 2 的整数数组 `[index1, index2]` 的形式返回这两个整数的下标 `index1` 和 `index2`。

你可以假设每个输入 **只对应唯一的答案**，而且你 **不可以** 重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

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]`。

示例 3:

输入: `numbers = [-1,0]`, `target = -1`

输出: `[1,2]`

解释: -1 与 0 之和等于目标数 -1。因此 `index1 = 1`, `index2 = 2`。返回 `[1, 2]`。

题解

本题是模板题，双指针相向移动，当和大于目标值时，移动右指针能减少和，当和小于目标值时，移动左指针能增加和。由于本题保证一定有解，因此不需要返回无解情况。

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

11. 盛最多水的容器

题目描述

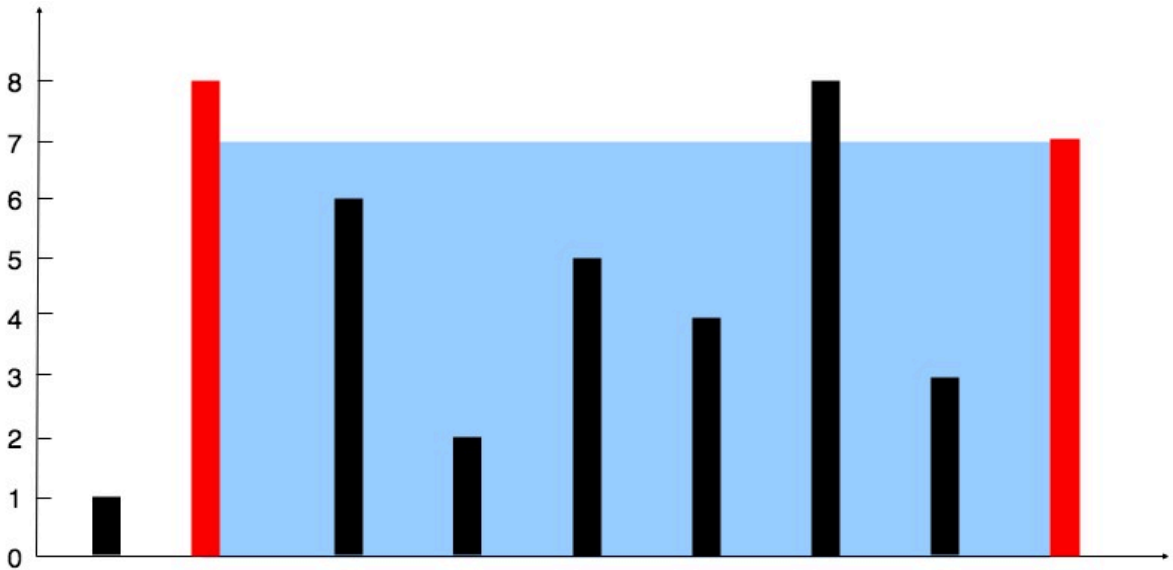
给定一个长度为 n 的整数数组 `height`。有 n 条垂线，第 i 条线的两个端点是 $(i, 0)$ 和 $(i, \text{height}[i])$ 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1:



输入: `[1,8,6,2,5,4,8,3,7]`

输出: 49

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

示例 2:

输入: `height = [1,1]`

输出: 1

题解

本题的双指针相向移动。每一次选择高度较小指针进行移动。因为只有移动高度较小的指针，才有可能获得更大的容积。具体证明见官方题解。

```
class Solution:
    def maxArea(self, height: List[int]) -> int:
        left = 0
        right = len(height) - 1
        max_area = 0
        while(left < right):
            new_area = (right - left) * min(height[left], height[right])
            if new_area > max_area:
```



```

        max_area = new_area
    if height[left] < height[right]:
        left += 1
    else:
        right -= 1
    return max_area

```

15. 三数之和

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k`，同时还满足 `nums[i] + nums[j] + nums[k] == 0`。请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

输入：nums = [-1,0,1,2,-1,-4]
 输出：[[-1,-1,2],[-1,0,1]]
 解释：
 $nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0$ 。
 $nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0$ 。
 $nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0$ 。
 不同的三元组是 [-1,0,1] 和 [-1,-1,2]。
 注意，输出的顺序和三元组的顺序并不重要。

示例 2：

输入：nums = [0,1,1]
 输出：[]
 解释：唯一可能的三元组和不为 0。

示例 3：

输入：nums = [0,0,0]
 输出：[[0,0,0]]
 解释：唯一可能的三元组和为 0。

题解

本题可以遍历数组 `nums[i]`，然后将三数之和转化为目标值为 `0 - nums[i]` 的两数之和问题。为了减少时间复杂度，首先将数组进行排序；其次将左指针设置为 `i+1`（为了避免重复的三元组），右指针设为 `n-1`，按两数之和的方法移动指针。

```

class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        nums.sort()
        ans = []
        for i in range(n):
            if nums[i] > 0:
                return ans

```

```

if (nums[i] == nums[i-1] and i > 0):#避免重复数组
    continue
left = i + 1
right = n-1
while(left<right):
    if nums[left] + nums[right] == -nums[i]:
        ans.append([nums[i],nums[left],nums[right]])
        while(left < right and nums[left] == nums[left+1]):#避免重复数
            left += 1
        while(left < right and nums[right] == nums[right-1]):#避免重复
            right -= 1
        left += 1
        right -= 1
    elif nums[left] + nums[right] < -nums[i]:
        left += 1
    else:
        right -= 1
return ans

```

2.3 题目（滑动窗口）

209. 长度最小的子数组

题目描述

给定一个含有 n 个正整数的数组和一个正整数 `target` 。

找出该数组中满足其总和大于等于 `target` 的长度最小的 **子数组** `[numsl, numsl+1, ..., numsr-1, numsr]`，并返回其长度。如果不存在符合条件的子数组，返回 0 。

示例 1:

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

示例 2:

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

示例 3:

输入: `target = 11, nums = [1,1,1,1,1,1,1,1]`
 输出: 0

题解

本题是模板题，滑动窗口一般是遍历右指针，然后对于每一个右指针，不断移动左指针，减小数组长度，并判断当前子数组是否满足要求，从而获得长度最小的子数组。

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

        for right in range(len(nums)):
            total += nums[right]
            while(left<=right and total>=target):
                min_len = min(min_len, right-left+1)
                total -= nums[left]
                left += 1

        if min_len == len(nums) + 1:
            return 0
        return min_len
```

3. 无重复字符的最长字串

题目描述

给定一个字符串 `s`，请你找出其中不含有重复字符的最长子串的长度。

示例 1:

输入: `s = "abcabcbb"`

输出: `3`

解释: 因为无重复字符的最长子串是 `"abc"`，所以其长度为 `3`。

示例 2:

输入: `s = "bbbbbb"`

输出: `1`

解释: 因为无重复字符的最长子串是 `"b"`，所以其长度为 `1`。

示例 3:

输入: `s = "pwwkew"`

输出: `3`

解释: 因为无重复字符的最长子串是 `"wke"`，所以其长度为 `3`。

请注意，你的答案必须是 子串 的长度，`"pwke"` 是一个子序列，不是子串。

题解

本题首先遍历右指针，如果右指针字母出现在现有的子串中，那么就不断移动左指针，并移除左指针字母，直到现有的子串中不存在重复字母，再移动右指针，将右指针字母添加到子串中。每遍历一次右指针，需要比较当前最长无重复子串与全局最长无重复子串的大小。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        n = len(s)
        left = 0
        max_len = 0
        occ = set()
        for right in range(n):
            while(left <= right and s[right] in occ):
                occ.remove(s[left])
                left += 1
            occ.add(s[right])
            max_len = max(max_len, right - left + 1)

        return max_len
```

三、二分查找

本章对应的题目为“二分查找”。

3.1 算法简介

(1) 基本定义

二分查找是一种在有序数组中查找某一特定元素的搜索算法，每次查找时通过将待查找的区间分成两个部分，然后比较目标值与中间元素的大小。如果目标值等于中间元素，则找到目标；如果目标值小于中间元素，则在左半部分继续查找；如果目标值大于中间元素，则在右半部分继续查找。对于长度为 n 的数组，时间复杂度为 $O(\log n)$ 。

与双指针有些类似。

(2) 应用场景

有序数组查找特定值、边界值。

(3) python模板

有序数组查找特定值

```
def binary_search(arr, target):
    #初始化左边界和右边界
    left, right = 0, len(arr) - 1
    while left <= right:
        #寻找中点
        mid = left + (right - left) // 2
        #与中点进行比较
        if arr[mid] == target:
            return mid
```

```
elif arr[mid] < target:
    left = mid + 1
else:
    right = mid - 1
return -1 # 如果目标值不存在，返回 -1
```

3.2 题目（二分查找）

35.搜索插入位置

题目描述

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

请必须使用时间复杂度为 $O(\log n)$ 的算法。

示例 1:

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

示例 2:

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

示例 3:

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

题解

基本思路和基本定义相同，只是当target不存在于数组中时，需要返回左边界。

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

        while(left <= right):
            mid = left + (right - left) // 2
            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return left
```

74. 搜索二维矩阵

题目描述：

给你一个满足下述两条属性的 $m \times n$ 整数矩阵：

- 每行中的整数从左到右按非严格递增顺序排列。
- 每行的第一个整数大于前一行的最后一个整数。

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

示例 1：

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

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

示例 2：

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

输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]，target = 13
输出：false

题解

本题可以将二维矩阵转化为一维数组。转化公式如下，其他步骤和基本二分查找相同。

$$\text{nums}[\text{mid}] == \text{matrix}[\text{mid} // n][\text{mid} \% n]$$

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
```

```

m = len(matrix)
n = len(matrix[0])
left = 0
right = n*m - 1
while left <= right:
    mid = left + (right-left)//2
    value = matrix[mid//n][mid%n]
    if value == target:
        return True
    elif value < target:
        left += 1
    else:
        right -= 1
return False

```

162. 寻找峰值

题目描述

峰值元素是指其值严格大于左右相邻值的元素。

给你一个整数数组 `nums`，找到峰值元素并返回其索引。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。

你可以假设 `nums[-1] = nums[n] = -∞`。

你必须实现时间复杂度为 $O(\log n)$ 的算法来解决此问题。

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。

题解

在搜索过程，只要一直往更大值的方向走，总能找到峰值。其中存在四种情况：

如果 `nums[i-1] < nums[i] > nums[i+1]`, 直接返回 `i`

如果 `nums[i-1] < nums[i] < nums[i+1]`, `i = i + 1`

如果 `nums[i-1] > nums[i] > nums[i+1]`, `i = i - 1`

如果 `nums[i-1] > nums[i] < nums[i+1]`, 两个方向都可以遍历，本题默认 `i = i + 1`

由此可以简化为：

如果 `nums[i] < nums[i+1]`, `i = i + 1`；否则 `i = i - 1`

本题由于找到一个较大值时，只需要遍历剩余一半的区间，因此可以结合二分法解决。

```

class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        n = len(nums)
        left = 0
        right = n - 1

        def get(i):#处理边界情况
            if i == -1 or i == n:
                return float('-inf')
            else:
                return nums[i]

        while(left<=right):
            mid = left + (right-left)//2
            if get(mid-1) < nums[mid] > get(mid+1):
                return mid
            elif get(mid) < get(mid+1):
                left = mid + 1
            else:
                right = mid - 1

```

33. 搜索旋转排序数组

题目描述

整数数组 `nums` 按升序排列，数组中的值 **互不相同**。

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

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

你必须设计一个时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1:

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

示例 2:

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

示例 3:

输入: `nums = [1]`, `target = 0`
输出: -1

题解

本题仍然基于二分法的模板来写代码，主要难点在于下一步查找空间的左右边界该如何变换。以 [4,5,6,7,0,1,2] 为例分析不同情况：

1. 当 $\text{nums}[\text{mid}] \leq \text{nums}[\text{n}-1]$ 时， $\text{nums}[\text{mid}]$ 一定在 [0,1,2] 子数组内。
 - 1) 当 $\text{nums}[\text{mid}] < \text{target} < \text{nums}[\text{n}-1]$ ， target 会出现在 $\text{nums}[\text{mid}]$ 的右边，调整左边界。
 - 2) 当 $\text{nums}[\text{mid}] > \text{target}$ 以及 $\text{target} > \text{nums}[\text{n}-1]$ 时， target 都会出现在 $\text{nums}[\text{mid}]$ 的左边，调整右边界。
2. 当 $\text{nums}[\text{mid}] > \text{nums}[\text{n}-1]$ 时， $\text{nums}[\text{mid}]$ 一定在 [4,5,6,7] 子数组内。
 - 1) 当 $\text{nums}[0] < \text{target} < \text{nums}[\text{mid}]$ 时， target 会出现在 $\text{nums}[\text{mid}]$ 的左边，调整右边界。
 - 2) 当 $\text{nums}[\text{mid}] < \text{target}$ 以及 $\text{nums}[0] > \text{target}$ 时， target 会出现在 $\text{nums}[\text{mid}]$ 的右边，调整左边界。

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

        while(left <= right):
            mid = left + (right-left)//2
            if nums[mid] == target:
                return mid
            elif nums[mid] < nums[n-1]:
                if nums[mid] < target <= nums[n-1]:
                    left = mid + 1
                else:
                    right = mid - 1
            else:
                if nums[0] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
        return -1
```

34. 在排序数组中查找元素的第一个和最后一个位置

题目描述

给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

你必须设计并实现时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1：

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

示例 2:

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

示例 3:

输入: nums = [], target = 0
输出: [-1,-1]

题解

本题可以编写两个while循环函数，分别用于寻找左边界和右边界，主要修改nums[mid]==target时的代码。在寻找左边界时，需要将右指针继续往左移；而在寻找右边界时，需要将左指针继续往右移。

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

153. 寻找旋转排序数组中的最小值

题目描述

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

- 若旋转 4 次，则可以得到 `[4,5,6,7,0,1,2]`

- 若旋转 7 次，则可以得到 [0,1,2,4,5,6,7]

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

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

你必须设计一个时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1:

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

示例 2:

输入: `nums = [4,5,6,7,0,1,2]`
输出: 0
解释: 原数组为 [0,1,2,4,5,6,7]，旋转 3 次得到输入数组。

示例 3:

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

题解:

本题是33题搜索旋转排序数组的变形题，但是只需要搜索最小值。本题只有两种情况，讨论如下：

当`nums[mid]<nums[n-1]`时，最小值一定在`nums[mid]`左边，值得注意的是右边界`right = mid`，否则会出现`right<left`的情况，导致搜索不到最小值。且`while`循环的判断条件是`left<right`，不能出现等号，否则会无限循环。

其他情况下，最小值一定在`num[mid]`右边。

最后返回`nums[right]`即可

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

四、深度优先搜索

4.1 算法简介

深度优先搜索算法和广度优先搜索算法被广泛的运用在树和图结构中。本章先介绍深度优先搜索算法，下一章介绍广度优先搜索算法。本章对应的题目为“二叉树”、“图”。

(1) 基本定义

深度优先搜索（Depth-First-Search，简称DFS）是一种从深度方向遍历树或图的算法。它从起始节点开始，选择一个相邻节点进行访问，并标记为已访问，然后递归地对该相邻节点的未访问相邻节点进行遍历，直到没有相邻节点或相邻节点都已访问为止，此时回溯到上一个未完全搜索的节点，继续搜索其他分支。

深度优先搜索是先入后出模式，因此常常使用栈结构，也可以使用递归来实现。这里推荐用递归实现。

(2) 应用场景

二叉树：先序、中序、后序遍历，以及计算图的深度等等。

图：遍历无向图、有向图，判断图的连通性，寻找图中的环等等。

(3) python模板

深度优先搜索一般包括一个主函数和一个副函数，主函数用于遍历所有节点，副函数展开搜索。一些简单的题目会将主函数和副函数合并。下面给出一些经典应用场景的python模板。

二叉树——中序遍历（左子树-根节点-右子树）

```
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def inorder_dfs(root):#主函数
        result = []
        def dfs(node):#副函数
            if node:
                # 递归遍历左子树
                dfs(node.left)
                # 访问根节点
                result.append(node.val)
                # 递归遍历右子树
                dfs(node.right)
        dfs(root)
        return result
```

图——二维网格的DFS模板

```
class Solution:
    #grid是一个二维网格（矩阵），visited是一个二维矩阵，用于记录节点是否已访问
    def grid_dfs(grid):#主函数
        rows, cols = len(grid), len(grid[0])
```

```

visited = [[False] * cols for _ in range(rows)]
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)] # 右、左、下、上四个遍历方向

def dfs(x, y):#副函数
    # 检查节点位置是否在矩阵内以及是否已访问
    if 0 <= x < rows and 0 <= y < cols and not visited[x][y]:
        # 标记当前节点为已访问
        visited[x][y] = True
        # 这里根据题目添加操作
        # . . . . .
        # 遍历四个方向
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            dfs(new_x, new_y)

# 可以从网格的任意节点开始, 这里从 (0, 0) 开始
dfs(0, 0)

```

4.2 题目 (二叉树)

104. 二叉树的最大深度

题目描述

给定一个二叉树 `root` , 返回其最大深度。

二叉树的 **最大深度** 是指从根节点到最远叶子节点的最长路径上的节点数。

示例1

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

示例 2:

输入: `root = [1,null,2]`
 输出: 2

题解

二叉树的最大深度 = 左、右子树的最大深度的最大值 + 1。因此递归计算左、右子树的最大深度即可。

```

class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if root == None:
            return 0
        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        return 1 + max(left_depth, right_depth)

```

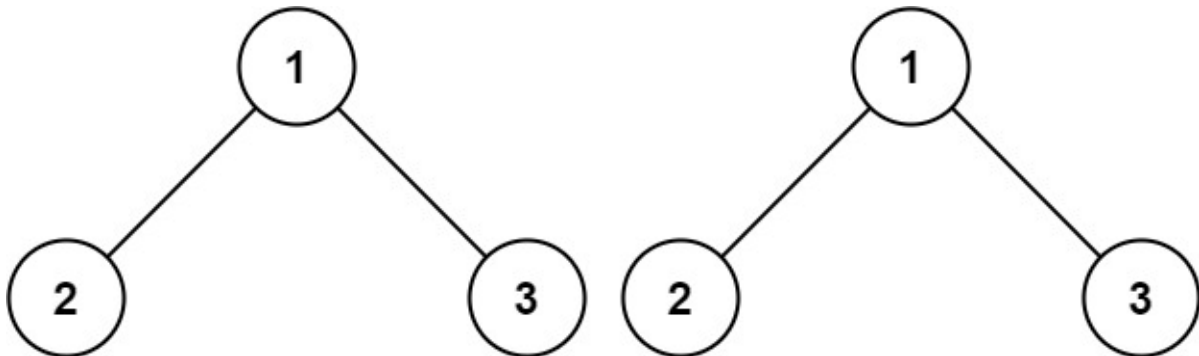
100. 相同的树

题目描述

给你两棵二叉树的根节点 `p` 和 `q`，编写一个函数来检验这两棵树是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

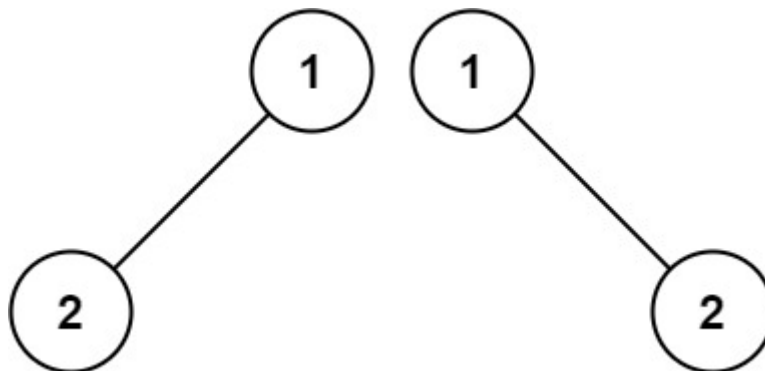
示例 1:



输入: `p = [1,2,3]`, `q = [1,2,3]`

输出: `true`

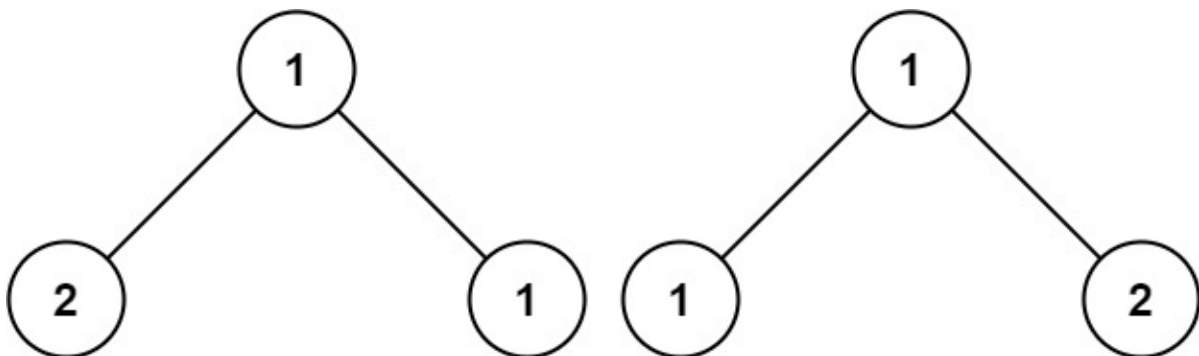
示例 2:



输入: `p = [1,2]`, `q = [1,null,2]`

输出: `false`

示例 3:



输入: `p = [1,2,1]`, `q = [1,1,2]`

输出: `false`

题解

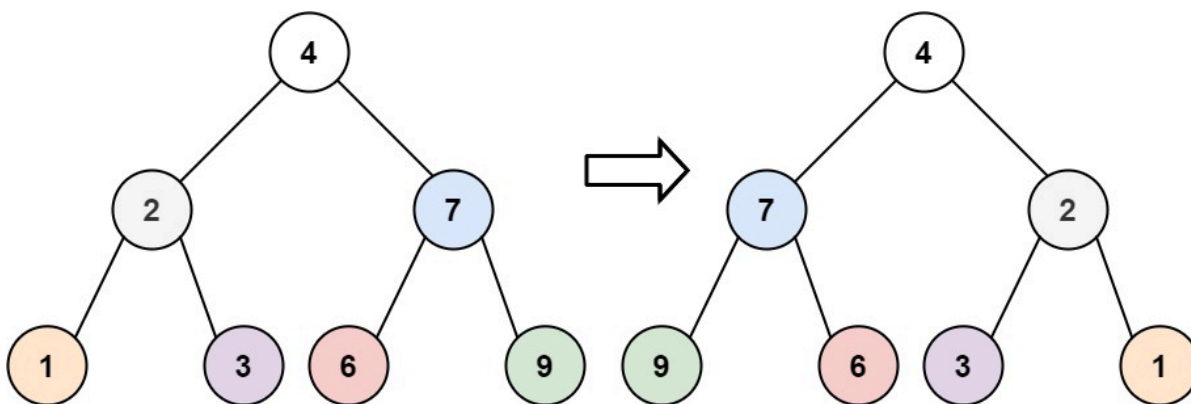
本题需同时dfs两个二叉树，并在搜索过程比较节点的差异，直到两个节点都为空时，返回True，一旦有不同的节点，则返回False。对于每一个节点对，需要其左子节点对和右子节点对都相同，该节点对才能返回True。

```
class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        if p == None and q == None:
            return True
        elif p == None or q == None:
            return False
        elif p.val != q.val:
            return False
        else:
            return self.isSameTree(p.left, q.left) and
                self.isSameTree(p.right, q.right)
```

226. 翻转二叉树

给你一棵二叉树的根节点 `root`，翻转这棵二叉树，并返回其根节点。

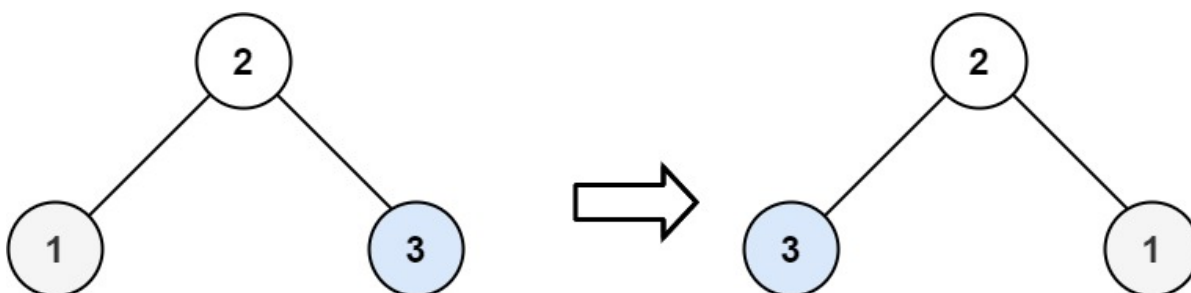
示例 1:



输入: `root = [4,2,7,1,3,6,9]`

输出: `[4,7,2,9,6,3,1]`

示例 2:



输入: `root = [2,1,3]`

输出: `[2,3,1]`

示例 3:

输入: root = []

输出: []

题解

本题在递归遍历的过程中，不断地优先翻转左右子树，再翻转当前的左右子节点即可。

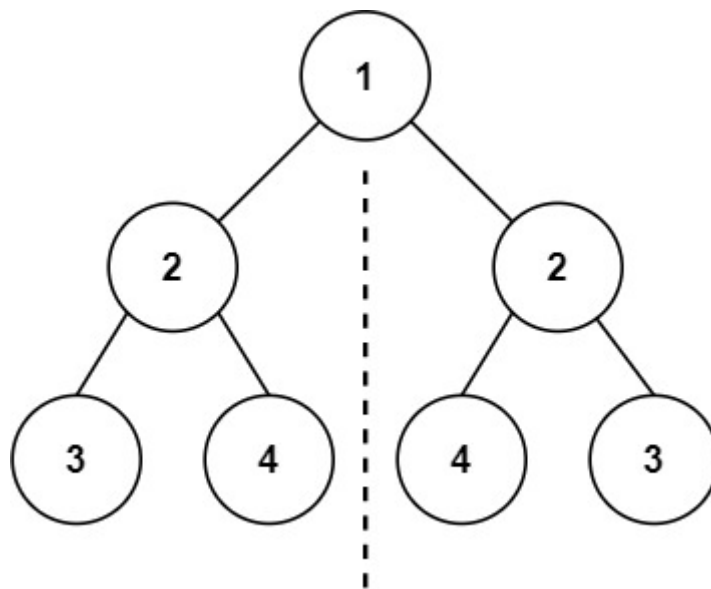
```
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return
        self.invertTree(root.left)
        self.invertTree(root.right)
        temp = root.right
        root.right = root.left
        root.left = temp
        return root
```

101. 对称二叉树

题目描述

给你一个二叉树的根节点 `root`，检查它是否轴对称。

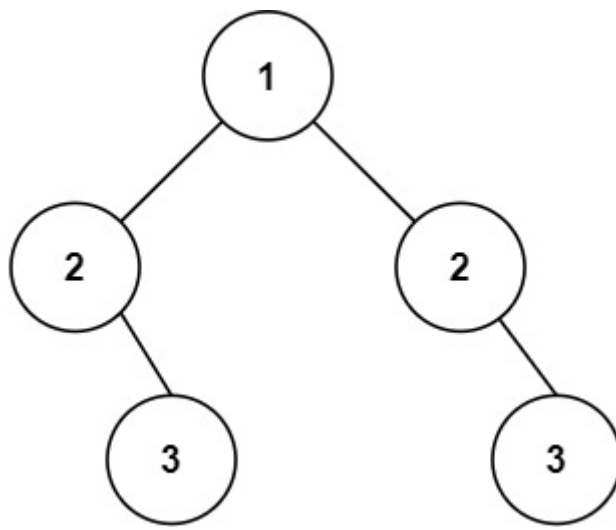
示例 1:



输入: root = [1,2,2,3,4,4,3]

输出: true

示例 2:



输入: root = [1,2,2,null,3,null,3]

输出: false

题解

本题需要编写一个dfs函数，用于判断左右两个子树是否对称。首先要保证左子节点和右子节点相同，其次要保证以下两个条件：

1. 左子节点的左子节点与右子节点的右子节点相同。
2. 左子节点的右子节点与右子节点的左子节点相同。

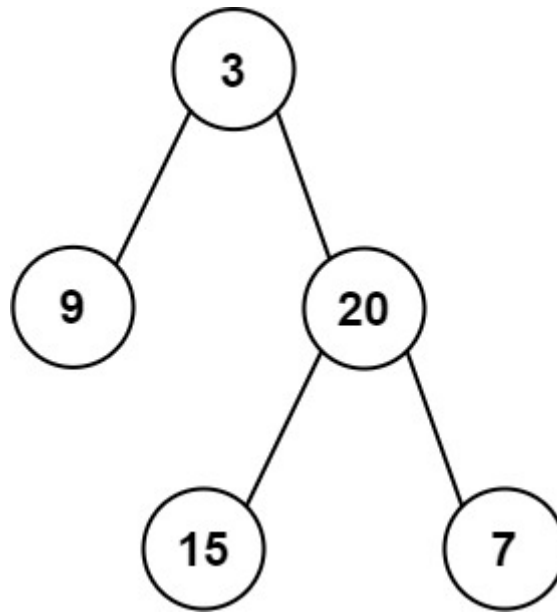
```
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        return self.is_check(root.left, root.right)
    def is_check(self, left: Optional[TreeNode], right: Optional[TreeNode]):
        if left == None and right == None:
            return True
        elif left == None or right == None:
            return False
        elif left.val != right.val:
            return False
        else:
            return self.is_check(left.left, right.right) and
self.is_check(left.right, right.left)
```

105. 从前序与中序遍历序列构造二叉树

题目描述

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

示例 1:



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

示例 2:

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

题解

前序遍历数组的形式为: [根节点, [左子树的前序遍历结果], [右子树的前序遍历结果]]

中序遍历数组的形式为: [[左子树的中序遍历结果], 根节点, [右子树的中序遍历结果]]

假设前序遍历数组的左右指针为preorder_left和preorder_right,中序遍历数组的左右指针为inorder_left和inorder_right。本题根据这四个指针构建二叉树,深度优先搜索方式如下:

1. 通过前序遍历数组的可以找到根节点, `preorder[preorder_left]`, 并根据该元素在中序遍历数组中找到根节点位置, 此处用 `index_root = index[preorder[preorder_left]]` 表示, `index` 是一个指示位置的哈希表。
1. 根据 `index_root` 可以获得左子树的长度, 左子树长度为 `left_subtree_len = index_root - inorder_left`
1. 根据左右子树的长度可以得到前序遍历数组中左右子树的左右指针。以左子树为例, 其左右指针为 `[preorder_left + 1, preorder_left + left_subtree_len]`, 而中序遍历数组中左子树的左右指针为, `[inorder_left, index_root - 1]`
1. 在获得前序遍历数组和中序遍历数组的左右指针后, 可以继续深度搜索, 直到左指针大于右指针。在深度搜索的同时需要构建二叉树。

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        index = {inorder[i]: i for i in range(len(inorder))}

        def dfs(preorder_left, preorder_right, inorder_left, inorder_right):
            if preorder_left > preorder_right:
                return None
            root = TreeNode(preorder[preorder_left])
```

```

        index_root = index[preorder[preorder_left]]
        left_subtree_len = index_root - inorder_left
        root.left = dfs(preorder_left+1,preorder_left+left_subtree_len,
inorder_left,index_root-1)
        root.right = dfs(preorder_left+left_subtree_len+1,preorder_right,
index_root+1,inorder_right)
        return root
    n = len(preorder)
    return dfs(0,n-1,0,n-1)

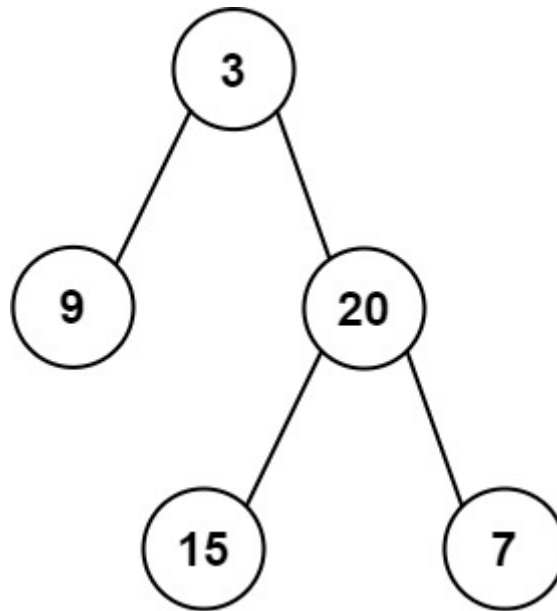
```

106. 从中序与后续遍历序列构造二叉树

题目描述

给定两个整数数组 `inorder` 和 `postorder`，其中 `inorder` 是二叉树的中序遍历，`postorder` 是同一棵树的后序遍历，请你构造并返回这颗 二叉树。

示例 1:



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

示例 2:

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

题解

相比于105题，因为后序遍历数组中，根节点始终是数组的最后一个元素。因此在构建根节点时，只需要执行pop操作即可。本题的dfs函数的输入时inorder_left和inorder_right。

左子树的左右指针为: `inorder_left`和`index(postorder.pop())-1`

右子树的左右指针为: `index(postorder.pop())+1` 和 `inorder_right`

```
class Solution:
```

```
def buildTree(self, inorder: List[int], postorder: List[int]) ->
Optional[TreeNode]:
    index = {inorder[i]: i for i in range(len(inorder))}
    def dfs(inorder_left, inorder_right):
        if inorder_left > inorder_right:
            return None
        root_val = postorder.pop()
        root = TreeNode(root_val)
        index_root = index[root_val]

        root.right = dfs(index_root+1, inorder_right)
        root.left = dfs(inorder_left, index_root-1)

    return root
n = len(inorder)
return dfs(0,n-1)
```

117. 填充每个节点的下一个右侧节点指针

题目描述

给定一个二叉树：

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 `NULL`。

初始状态下，所有 next 指针都被设置为 `NULL`。

示例 1：

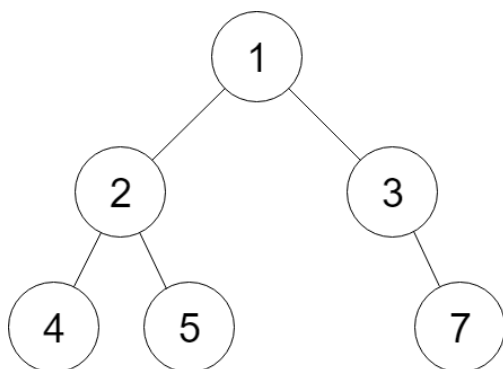


Figure A

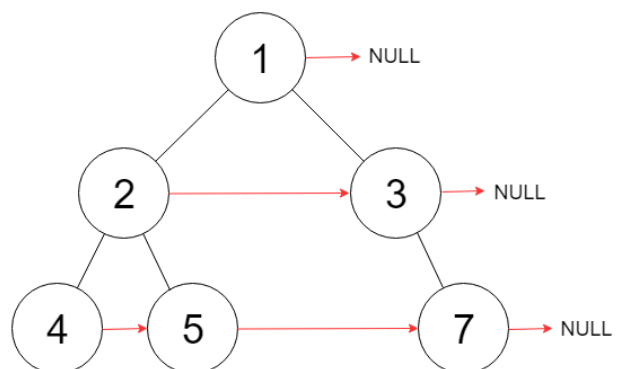


Figure B

输入: `root = [1,2,3,4,5,null,7]`

输出: `[1,#,2,3,#,4,5,7,#]`

解释: 给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化输出按层序遍历顺序（由 next 指针连接），'#' 表示每层的末尾。

示例 2:

输入: root = []
输出: []

题解

本题既可以使用深度优先搜索,也可以使用广度优先搜索(个人感觉更容易理解)。但因为放在这一章,所以介绍深度优先搜索方法。

- 1.首先创建一个数组pre, pre[i]用于记录每一层的前一个节点。
- 2.构建dfs函数,递归输入为(节点root和深度depth),并按前序遍历方式遍历二叉树。
- 3.当depth和pre数组长度相等时,说明当前节点是这一层的最左侧节点,将root添加到pre的末尾。
- 4.否则,pre[depth]是root的左侧节点,将pre[depth]的next指针指向root,并将pre[depth]更新为root。
- 5.最后返回root即可。

```
"""
# Definition for a Node.
class Node:
    def __init__(self, val: int = 0, left: 'Node' = None, right: 'Node' = None,
next: 'Node' = None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next
"""

class Solution:
    def connect(self, root: 'Node') -> 'Node':
        pre = []
        def dfs(root, depth):
            if root == None:
                return None
            if depth == len(pre):
                pre.append(root)
            else:
                pre[depth].next = root
                pre[depth] = root
            dfs(root.left, depth+1)
            dfs(root.right, depth+1)
            return root
        return dfs(root, 0)
```

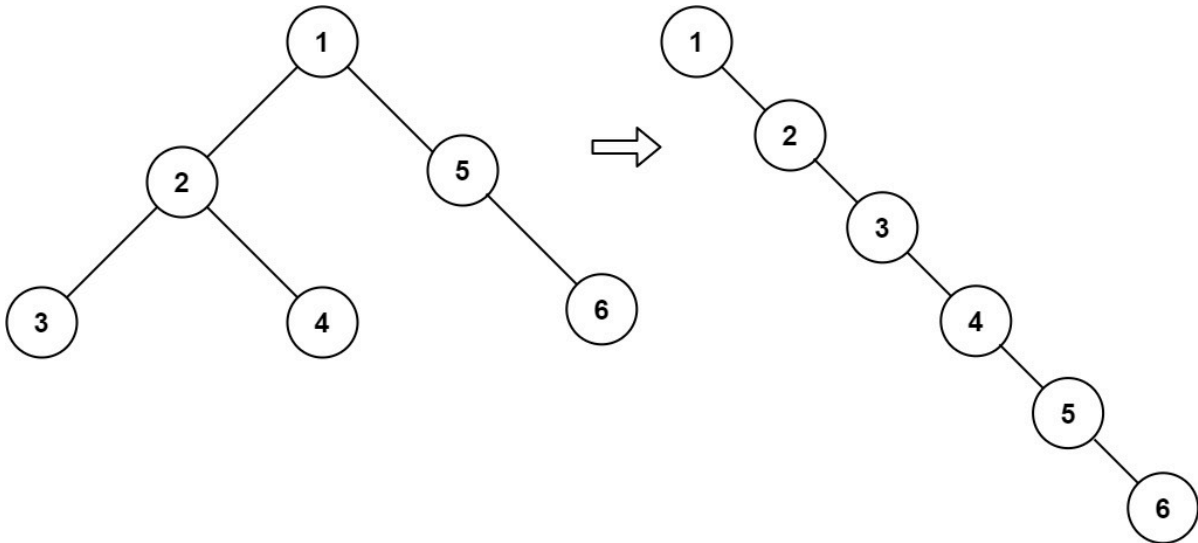
114. 二叉树展开为链表

题目描述

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。
- 展开后的单链表应该与二叉树 [先序遍历](#) 顺序相同。

示例 1：



输入: `root = [1,2,5,3,4,null,6]`

输出: `[1,null,2,null,3,null,4,null,5,null,6]`

示例 2：

输入: `root = []`

输出: `[]`

示例 3：

输入: `root = [0]`

输出: `[0]`

题解

本题首先利用先序遍历方式构建数组，其次根据数组构建一个单链表。

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

```
def dfs(root.left):
    dfs(root.right)
dfs(root)
n = len(pre_order)

for i in range(1,n):
    root.right = TreeNode(pre_order[i])
    root.left = None
    root = root.right
```

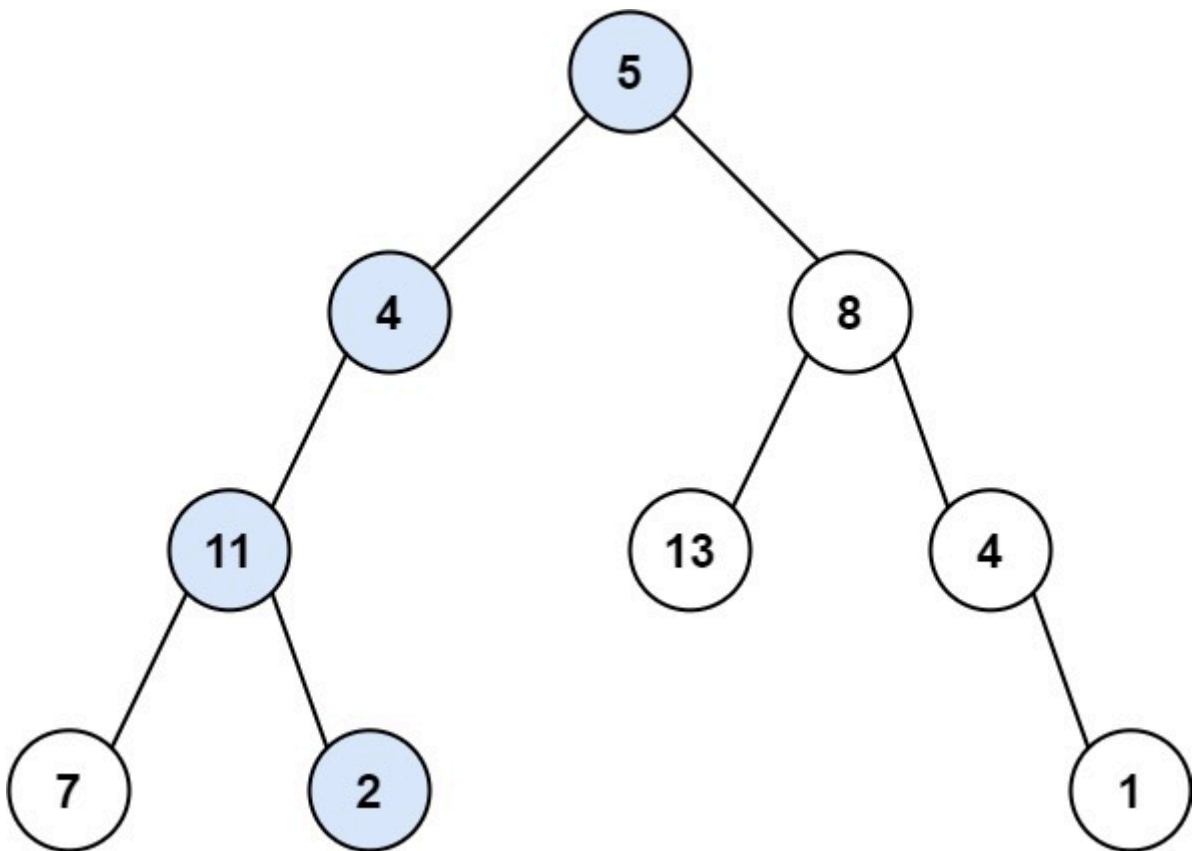
112. 路径总和

题目描述

给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum`。判断该树中是否存在 **根节点到叶子节点** 的路径，这条路径上所有节点值相加等于目标和 `targetSum`。如果存在，返回 `true`；否则，返回 `false`。

叶子节点 是指没有子节点的节点。

示例 1:

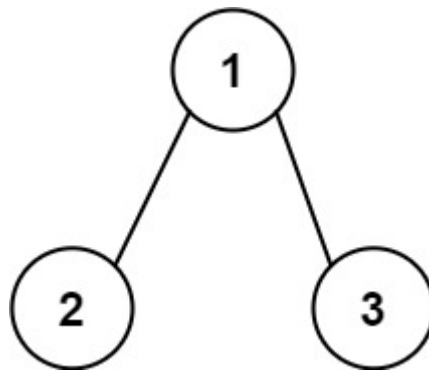


输入: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22`

输出: `true`

解释: 等于目标和的根节点到叶节点路径如上图所示。

示例 2:



输入: `root = [1,2,3]`, `targetSum = 5`

输出: `false`

解释: 树中存在两条根节点到叶子节点的路径:

(1 --> 2): 和为 3

(1 --> 3): 和为 4

不存在 `sum = 5` 的根节点到叶子节点的路径。

示例 3:

输入: `root = []`, `targetSum = 0`

输出: `false`

解释: 由于树是空的, 所以不存在根节点到叶子节点的路径。

题解

本题在遍历过程中, 可以记录根节点到当前节点的和, 从而避免重复计算。因此dfs函数的输入为 `root`和`targetSum - root.val`。在遍历到叶子节点时, 只需要一个叶子结点满足要求, 就返回`True`。

```
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        if root == None:
            return False
        if not root.left and not root.right:
            if targetSum == root.val:
                return True
        return self.hasPathSum(root.left, targetSum-root.val) or \
            self.hasPathSum(root.right, targetSum-root.val)
```

129. 求根节点到叶节点数字之和

题目描述

给你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。

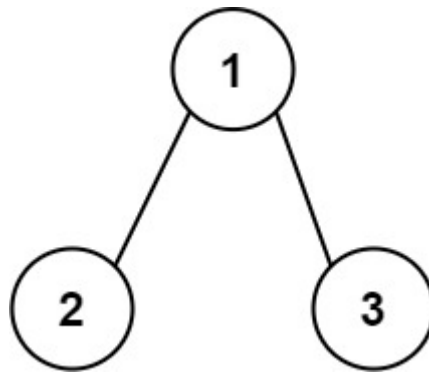
每条从根节点到叶节点的路径都代表一个数字:

- 例如, 从根节点到叶节点的路径 1 -> 2 -> 3 表示数字 123 。

计算从根节点到叶节点生成的 **所有数字之和** 。

叶节点 是指没有子节点的节点。

示例 1:



输入: `root = [1,2,3]`

输出: 25

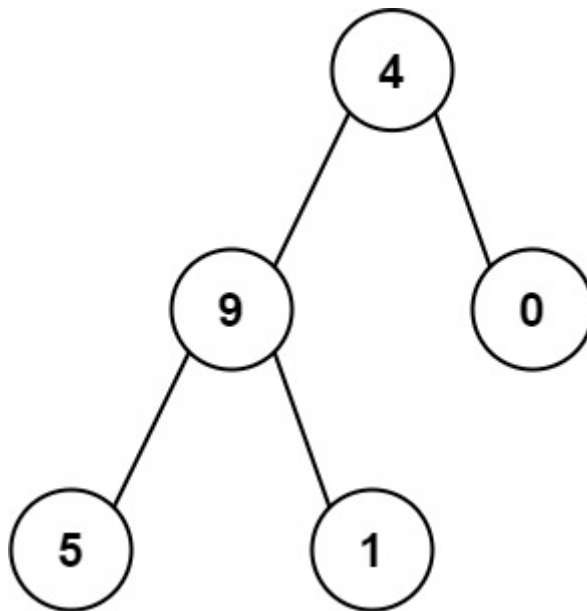
解释:

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

题解

本题在需要通过dfs函数计算每个叶子节点代表的数字, 并将他们相加。而叶子节点代表的数字等于父节点代表的数字*10+叶子结点的val。因此可以用pre_num表示父节点代表的数字, 并不断递推叶子节点代表的数字。

```
class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        def dfs(root, pre_ans):
            if root == None:
                return 0
            current_ans = pre_ans*10 + root.val
            if not root.left and not root.right:
                return current_ans
            return dfs(root.left, current_ans) + dfs(root.right, current_ans)
        return dfs(root, 0)
```

173. 二叉树搜索迭代器

题目描述

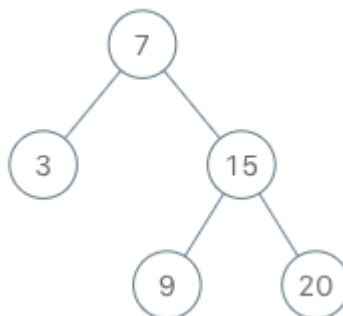
实现一个二叉搜索树迭代器类 `BSTIterator`，表示一个按中序遍历二叉搜索树（BST）的迭代器：

- `BSTIterator(TreeNode root)` 初始化 `BSTIterator` 类的一个对象。BST 的根节点 `root` 会作为构造函数的一部分给出。指针应初始化为一个不存在于 BST 中的数字，且该数字小于 BST 中的任何元素。
- `boolean hasNext()` 如果向指针右侧遍历存在数字，则返回 `true`；否则返回 `false`。
- `int next()` 将指针向右移动，然后返回指针处的数字。

注意，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。

你可以假设 `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。

示例：



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]
```

解释

```
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next();      // 返回 3
bSTIterator.next();      // 返回 7
bSTIterator.hasNext();   // 返回 True
```

```
bSTIterator.next();    // 返回 9
bSTIterator.hasNext(); // 返回 True
bSTIterator.next();    // 返回 15
bSTIterator.hasNext(); // 返回 True
bSTIterator.next();    // 返回 20
bSTIterator.hasNext(); // 返回 False
```

题解

本题感觉不是考算法，暂时只给出代码

```
class BSTIterator:
    def __init__(self, root: Optional[TreeNode]):
        self.queue = collections.deque()#初始化一个队列
        self.inOrder(root)#基于队列构建中序遍历的二叉树
    def inOrder(self, root):#中序遍历
        if not root: return
        self.inOrder(root.left)
        self.queue.append(root.val)
        self.inOrder(root.right)

    def next(self) -> int:
        return self.queue.popleft()

    def hasNext(self) -> bool:
        return len(self.queue) > 0
```

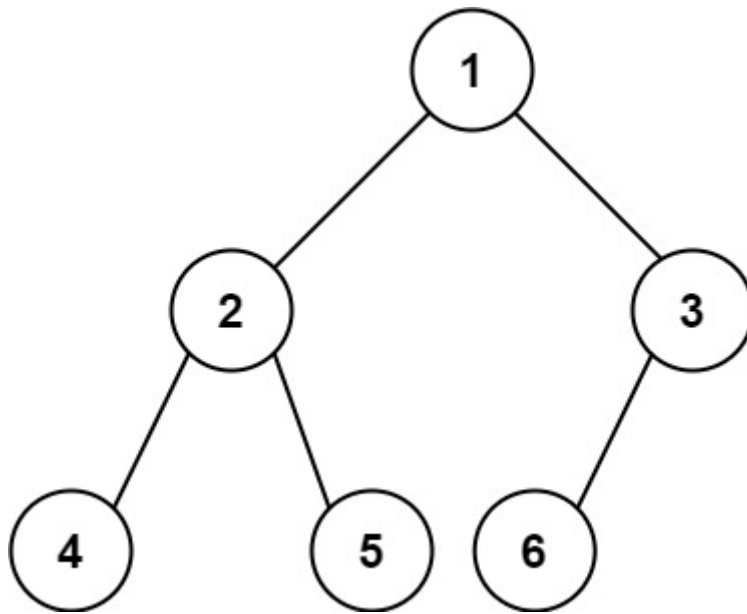
222. 完全二叉树的节点个数

题目描述

给你一棵 **完全二叉树** 的根节点 `root`，求出该树的节点个数。

完全二叉树 的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层（从第 0 层开始），则该层包含 $1 \sim 2^h$ 个节点。

示例 1：



输入: root = [1,2,3,4,5,6]
输出: 6

示例 2:

输入: root = []
输出: 0

示例 3:

输入: root = [1]
输出: 1

题解

本题直接用朴素的遍历方式解决。

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

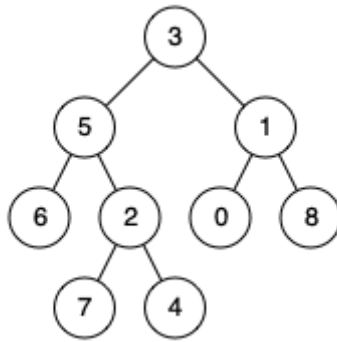
236. 二叉树的最近公共祖先

题目描述

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

[百度百科](#)中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

示例 1:

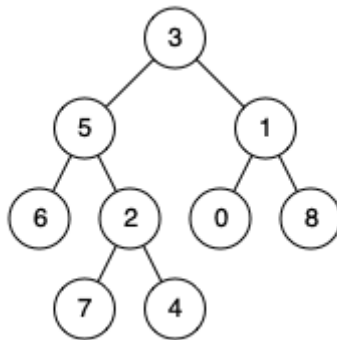


输入: `root = [3,5,1,6,2,0,8,null,null,7,4]`, `p = 5`, `q = 1`

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:



输入: `root = [3,5,1,6,2,0,8,null,null,7,4]`, `p = 5`, `q = 4`

输出: 5

解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

示例 3:

输入: `root = [1,2]`, `p = 1`, `q = 2`

输出: 1

题解

在遍历过程中, 符合条件的最近公共祖先`root`必须是以下两种情况之一: 1.`p,q`分别在`root`的左右子树中; 2.`p == root` 或者`q == root`。因此递归函数中的分类讨论情况

1. 判断`root`是否等于`p`或`q`, 若满足则返回`root`。`root`为`None`时也返回`root`。
2. 遍历左子树, 若左子树中不存在`p`或`q`, 则`p`, `q`只能在右子树中, 返回右子树中的最近公共祖先。
3. 遍历右子树, 若右子树中不存在`p`或`q`, 则`p`, `q`只能在左子树中, 返回左子树中的最近公共祖先。
4. 否则, `p`和`q`分别存在于左、右子树中, 则当前节点就是最近公共祖先。

`lowestCommonAncestord()`的返回值可以理解为当前二叉树中的最近公共祖先。

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q:
'TreeNode') -> 'TreeNode':

        if root == None or root == p or root == q:
            return root
        left = self.lowestCommonAncestor(root.left, p,q)
        right = self.lowestCommonAncestor(root.right,p,q)
        if not left: return right
        if not right: return left
        return root
```

4.3 题目（图）

200. 岛屿数量

题目描述

给你一个由 '1'（陆地）和 '0'（水）组成的二维网格，请你计算网格中岛屿的数量。岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。此外，你可以假设该网格的四条边均被水包围。

示例 1:

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

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
```

题解

该题属于典型的网格问题，深度优先搜索（dfs）方法的基本思路如下：1.对于主函数，我们扫描整个网格，如果一个节点为1，那就以此节点开始进行深度优先搜索；2.对于dfs函数，遍历的相邻节点有上下左右四个，而遍历的终止条件（base case）是相邻节点不存在或者已被遍历。本题的dfs函数将遍历过的节点置为0，防止重复遍历，使得在dfs函数遍历结束后，一座岛屿只有一个位置为“1”。因此主函数中遍历到节点状态为“1”的次数，即为岛屿的数量。

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
```

```

rows,cols = len(grid),len(grid[0])
visited = [[False] * cols for _ in range(rows)]
directions = [(0,1),(0,-1),(1,0),(-1,0)]
ans = 0
def dfs(m,n):
    if grid[m][n] == "1":
        grid[m][n] = 0
        visited[m][n] = 1
        for dm,dn in directions:
            new_m,new_n = m + dm, n + dn
            if 0<=new_m<rows and 0<=new_n<cols and not visited[new_m]
[new_n]:
                dfs(new_m,new_n)

    for m in range(rows):
        for n in range(cols):
            if grid[m][n] == "1":
                ans += 1
                dfs(m,n)
    return ans

```

130. 被围绕的区域

题目描述

给你一个 $m \times n$ 的矩阵 `board`，由若干字符 `'x'` 和 `'o'` 组成，**捕获** 所有 **被围绕的区域**：

- **连接**：一个单元格与水平或垂直方向上相邻的单元格连接。
- **区域**：连接所有 `'o'` 的单元格来形成一个区域。
- **围绕**：如果您可以用 `'x'` 单元格 **连接这个区域**，并且区域中没有任何单元格位于 `board` 边缘，则该区域被 `'x'` 单元格围绕。

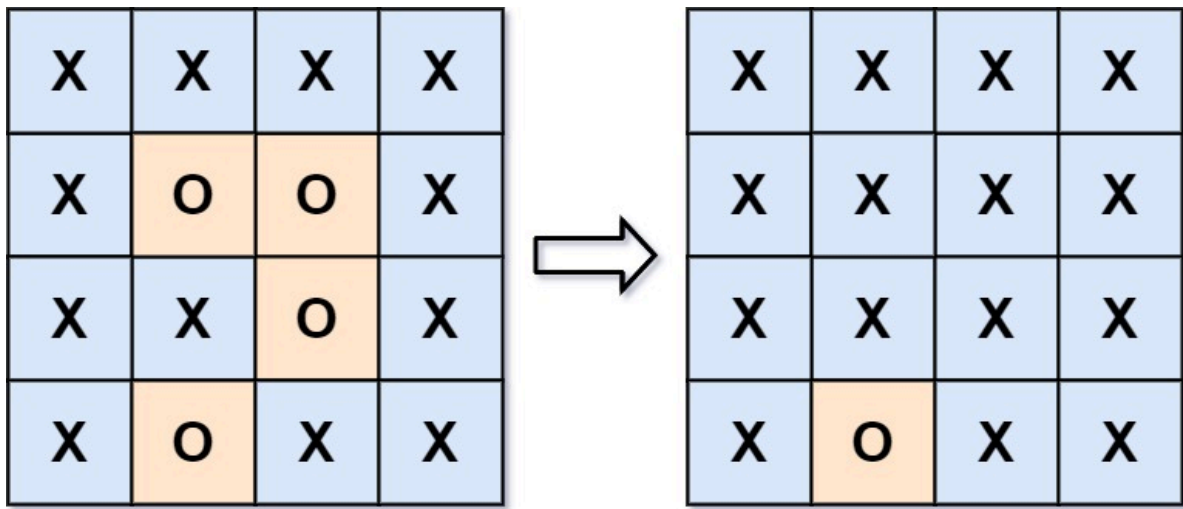
通过 **原地** 将输入矩阵中的所有 `'o'` 替换为 `'x'` 来 **捕获被围绕的区域**。你不需要返回任何值。

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：



在上图中，底部的区域没有被捕获，因为它在 board 的边缘并且不能被围绕。

示例 2:

输入: board = [["X"]]

输出: [["X"]]

题解

从题目可知，处于边缘的 'o' 以及与其相连接的 'o' 所构成的区域是不会被捕获的，因此，本题可以从边缘的 'o' 开始，dfs搜索不会被捕获的区域，并且用 'A' 进行标记。最后将剩余的 'o' 改成 'x'。因为本题用 'A' 进行了标记，所以不需要visited矩阵记录节点是否已访问。

```
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        rows, cols = len(board), len(board[0])
        directions = [(0,1),(0,-1),(1,0),(-1,0)]
        def dfs(m,n):
            if board[m][n] == 'O':
                board[m][n] = 'A'
                for dm, dn in directions:
                    new_m, new_n = m + dm, n + dn
                    if 0<=new_m<rows and 0<=new_n<cols:
                        dfs(new_m,new_n)

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

        for m in range(rows):
            for n in range(cols):
                if board[m][n] == 'O':
                    board[m][n] = 'X'
                if board[m][n] == 'A':
                    board[m][n] = 'O'
```


133. 克隆图

题目描述

给你无向 [连通](#) 图中一个节点的引用，请你返回该图的 [深拷贝](#)（克隆）。

图中的每个节点都包含它的值 `val`（`int`）和其邻居的列表（`list[Node]`）。

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

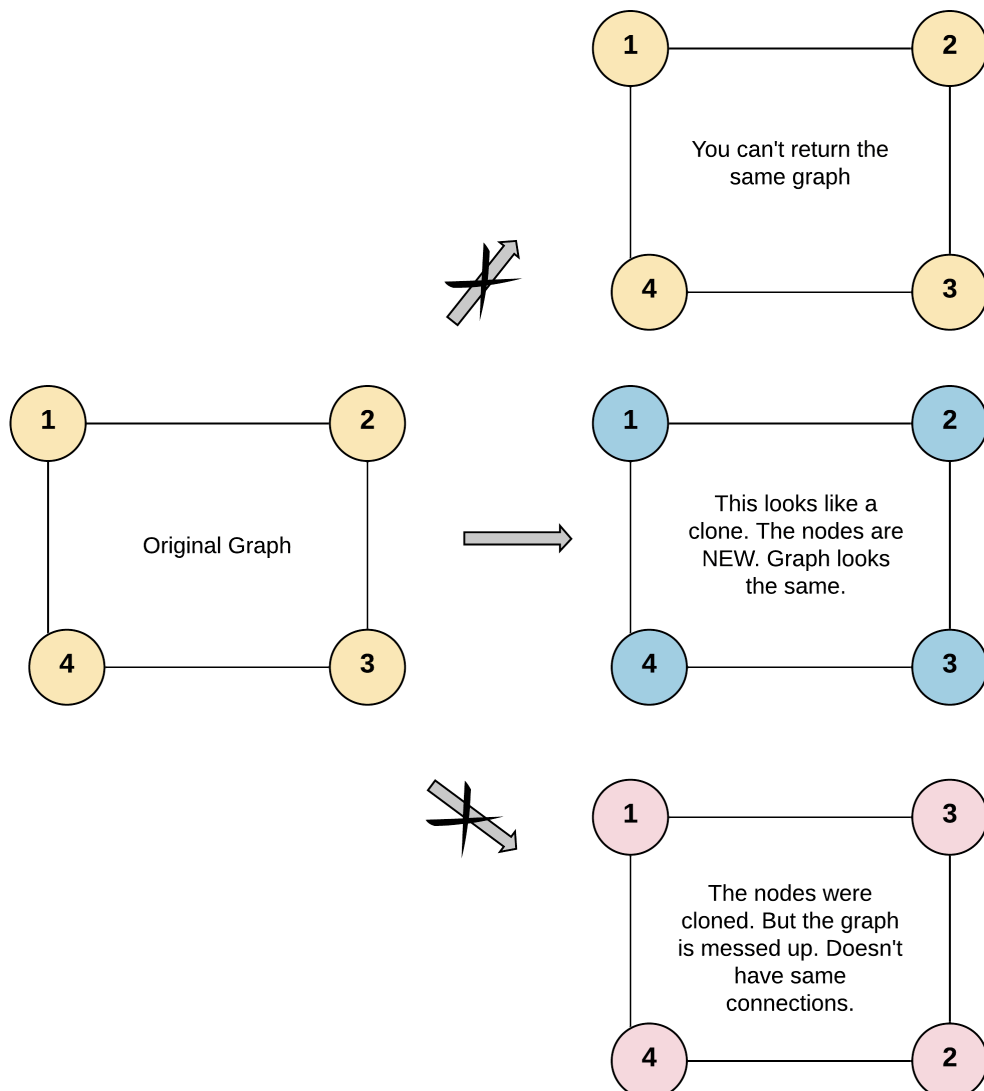
测试用例格式：

简单起见，每个节点的值都和它的索引相同。例如，第一个节点值为 1（`val = 1`），第二个节点值为 2（`val = 2`），以此类推。该图在测试用例中使用邻接列表表示。

邻接列表 是用于表示有限图的无序列表的集合。每个列表都描述了图中节点的邻合集。

给定节点将始终是图中的第一个节点（值为 1）。你必须将 **给定节点的拷贝** 作为对克隆图的引用返回。

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 = [[]]`

输出: `[[]]`

解释: 输入包含一个空列表。该图仅仅只有一个值为 1 的节点，它没有任何邻居。

示例 3:

输入: `adjList = []`

输出: `[]`

解释: 这个图是空的，它不含任何节点。

题解

深拷贝需要通过遍历原来的图来构建一张新的无向连通图，具体步骤如下：

1. 为了防止无向连通图多次遍历同一节点，因此构建一个哈希表 `visited` 来记录已遍历的节点
2. 对于 `dfs` 函数，如果当前节点已遍历，则返回其克隆节点，否则创建克隆节点并放置于哈希表中
3. 对于新的克隆节点，递归地遍历其邻接节点，并将返回值构成一个列表。

```
"""
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""

from typing import Optional
class Solution:
    def cloneGraph(self, node: Optional['Node']) -> Optional['Node']:
        visited = dict()
        if not node:
            return node
        def dfs(node):
            if node in visited:
```

```

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

```

399. 除法求值

题目描述

给你一个变量对数组 `equations` 和一个实数值数组 `values` 作为已知条件，其中 `equations[i] = [Ai, Bi]` 和 `values[i]` 共同表示等式 $A_i / B_i = \text{values}[i]$ 。每个 `Ai` 或 `Bi` 是一个表示单个变量的字符串。

另有一些以数组 `queries` 表示的问题，其中 `queries[j] = [Cj, Dj]` 表示第 `j` 个问题，请你根据已知条件找出 $C_j / D_j = ?$ 的结果作为答案。

返回 **所有问题的答案**。如果存在某个无法确定的答案，则用 `-1.0` 替代这个答案。如果问题中出现了给定的已知条件中没有出现的字符串，也需要用 `-1.0` 替代这个答案。

注意：输入总是有效的。你可以假设除法运算中不会出现除数为 0 的情况，且不存在任何矛盾的结果。

注意：未在等式列表中出现的变量是未定义的，因此无法确定它们的答案。

示例 1：

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

示例 2：

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

示例 3：

输入: `equations = [["a","b"]]`, `values = [0.5]`, `queries = [["a","b"],["b","a"],["a","c"],["x","y"]]`
 输出: `[0.50000,2.00000,-1.00000,-1.00000]`

题解

本题首先要构造一个图，然后进行深度优先搜索，最后求解。步骤如下：

1. 把每一个变量看做是一个节点，把除法运算看做是一条边，且商是边的权重，基于此利用dict结构来构造图。
2. 定义dfs函数，输入为节点值和累乘的权重。本题从queries[i][0]开始，深度遍历搜索整个图，若能找到queries[i][1]，则返回累乘的权重。
3. 如果遍历完所有节点，仍没有找到querie[i][1]，则说明没有答案，返回-1。这个过程中用visited矩阵来记录所有已访问的节点。

```
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float],
queries: List[List[str]]) -> List[float]:
        graph = {}
        for (s,e), v in zip(equations, values):
            if s not in graph:
                graph[s] = {}
            graph[s][e] = v
            if e not in graph:
                graph[e] = {}
            graph[e][s] = 1.0/v
            graph[s][s] = 1.0
            graph[e][e] = 1.0
        n = len(queries)
        ans = [-1.0]*n

        def dfs(qx,mul,qy):
            if qx == qy:
                return mul
            visited.add(qx)
            res = -1
            for neighbor, weight in graph[qx].items():
                if neighbor not in visited:
                    res = dfs(neighbor, mul*weight,qy)
                    if res != -1.0:
                        break
            return res

        for i, (qx,qy) in enumerate(queries):
            if qx not in graph or qy not in graph: continue
            visited = set([qx])
            ans[i] = dfs(qx,1.0,qy)
        return ans
```

207. 课程表

题目描述

你这个学期必须选修 `numCourses` 门课程，记为 `0` 到 `numCourses - 1`。

在选修某些课程之前需要一些先修课程。先修课程按数组 `prerequisites` 给出，其中 `prerequisites[i] = [ai, bi]`，表示如果要学习课程 `ai` 则 **必须** 先学习课程 `bi`。

- 例如，先修课程对 `[0, 1]` 表示：想要学习课程 `0`，你需要先完成课程 `1`。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true`；否则，返回 `false`。

示例 1:

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

输出: `true`

解释: 总共有 2 门课程。学习课程 1 之前，你需要完成课程 0。这是可能的。

示例 2:

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

输出: `false`

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

题解

本题实际上是判断图中是否有环，如果有环则返回错误，否则返回True。

根据灵神的思路，首先对于每个节点x，都定义三种颜色值（状态值）：

0：节点x尚未被访问到。

1：节点x正在访问中，dfs(x)尚未结束（注：节点还有可能和其他节点构成环）

2：节点x已经完全访问完毕，dfs(x)已返回。

算法流程

1.建图：把每个`prerequisites[i] = [a,b]`看成一条有向边`b—>a`，构建一个有向图。

2.创建长为`numCourses`的颜色数组`colors`，所有元素初始为0，用于记录节点状态

3.遍历每一个节点，如果对应的`colors[i]=0`，则调用递归函数`dfs(i)`，判断从他出发是否有环

4.执行dfs (i):

a.首先标记`color[i] = 1`,表示节点x正在访问中。

b.遍历i的邻居j，如果`color[j] = 1`，则找到了环，返回

c.如果没有找到环，则标记`colors[i] = 2`，表示i已经完全访问完毕，并返回false

5.如果dfs(i)返回true，则找到了环，返回false

6.如果遍历完所有节点都没有找到环，返回true。

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
```

```

#构建图
g = [[] for _ in range(numCourses)]
for a,b in prerequisites:
    g[b].append(a)
#设置状态
colors = [0] * numCourses
#遍历周围的点，检查是否有环路
def dfs(i):
    colors[i] = 1
    for j in g[i]:
        if colors[j] == 1 or colors[j] == 0 and dfs(j):
            return True
    colors[i] = 2
    return False
#从未访问过的点开始，检查是否有环
for i,c in enumerate(colors):
    if c == 0 and dfs(i):
        return False
return True

```

210. 课程表II

现在你总共有 `numCourses` 门课需要选，记为 `0` 到 `numCourses - 1`。给你一个数组 `prerequisites`，其中 `prerequisites[i] = [ai, bi]`，表示在选修课程 `ai` 前 **必须** 先选修 `bi`。

- 例如，想要学习课程 `0`，你需要先完成课程 `1`，我们用一个匹配来表示：`[0,1]`。

返回你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序，你只要返回 **任意一种** 就可以了。如果不可能完成所有课程，返回 **一个空数组**。

示例 1:

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

输出: `[0,1]`

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

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]`。

示例 3:

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

输出: `[0]`

题解

本题和前一题基本相同，只是要输出具体的一个课程顺序，这里可以在dfs遍历过程中记录已完全访问的节点，只是最后要逆序返回。

```
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        #构建图
        g = [[] for _ in range(numCourses)]
        for a,b in prerequisites:
            g[b].append(a)
        #设置状态
        colors = [0] * numCourses
        ans = []
        #遍历周围的点，检查是否有环路
        def dfs(i):
            colors[i] = 1
            for j in g[i]:
                if colors[j] == 1 or colors[j] == 0 and dfs(j):
                    return True
            colors[i] = 2
            ans.append(i)
            return False
        #从未访问过的点开始，检查是否有环
        for i,c in enumerate(colors):
            if c == 0 and dfs(i):
                return []
        ##逆序输出
        n = len(ans)
        ans_1 = []
        for i in range(n-1,-1,-1):
            ans_1.append(ans[i])
        return ans_1
        # 逆序输出也可以返回ans[::-1]
```

五、广度优先搜索

5.1 算法简介

本章对应的题目为“二叉树层次遍历”、“图的广度优先搜索”。

(1) 基本定义

广度优先搜索算法（Breadth - First Search，简称 BFS）是一种从层级方向遍历树或图的算法。它从起始节点开始，逐层地对节点进行访问，即先访问距离起始节点最近的所有节点，然后再依次访问距离更远一层的节点，直到访问完所有可达节点。

广度优先搜索是先入先出模式，因此使用队列结构。Python 中可以使用 `collections.deque` 来实现高效的队列操作。（力扣中collections可省略）

(2) 应用场景

二叉树：层次遍历等。

图：最短路径问题、连通分量测量等等。

(3) python模板

广度优先搜索一般使用while循环来遍历队列，并且记录每一层的节点数量来区分不同的层，下面给出一些经典应用场景的python模板。

二叉树——层次遍历

```
#from collections import deque
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
def bfs_binary_tree(root):
    if not root:
        return []
    result = [] #最终结果
    queue = deque([root]) #构建队列
    while queue:
        n = len(queue) #当前层的节点个数，区分不同层数，避免重复遍历
        temp = [] #保存当前层的节点值，也可以是其他操作
        for _ in range(n):
            node = queue.popleft() #取出当前层的节点
            ###队列中加入下一层的节点
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(temp) #所有层的节点值
    return result
```

图——二维网格的BFS模板

```
#from collections import deque

def bfs_matrix(matrix, start):
    rows, cols = len(matrix), len(matrix[0])
    # 定义四个方向：上、下、左、右
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # 初始化队列，队列元素为（行索引，列索引），
    queue = deque([start])
    # 初始化已访问矩阵
    visited = [[False] * cols for _ in range(rows)]
    # 标记起始节点为已访问
    visited[start[0]][start[1]] = True

    while queue:
        # 从队列中取出当前节点
        x, y = queue.popleft()
        # 遍历四个方向
```



```

for dx, dy in directions:
    new_x, new_y = x + dx, y + dy
    # # 检查节点位置是否在矩阵内以及是否已访问
    if 0 <= new_x < rows and 0 <= new_y < cols and not visited[new_x]
[new_y]:
    # 标记新节点为已访问
    visited[new_x][new_y] = True
    # 根据题目进行操作
    # . . . . .
    # 将新节点加入队列
    queue.append((new_x, new_y))
return visited

```

5.2 题目（二叉树层次遍历）

199. 二叉树的右视图

题目描述

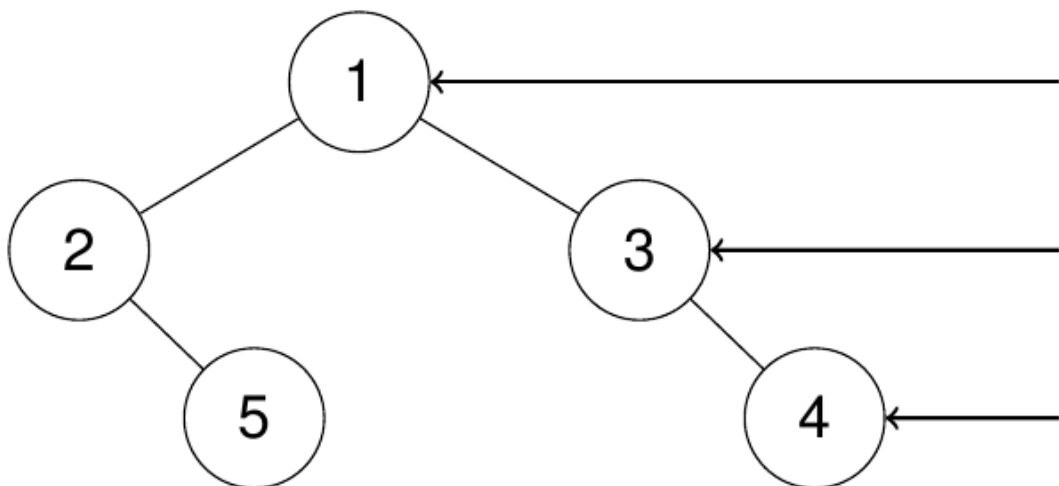
给定一个二叉树的 **根节点** `root`，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例 1:

输入: `root = [1,2,3,null,5,null,4]`

输出: `[1,3,4]`

解释:

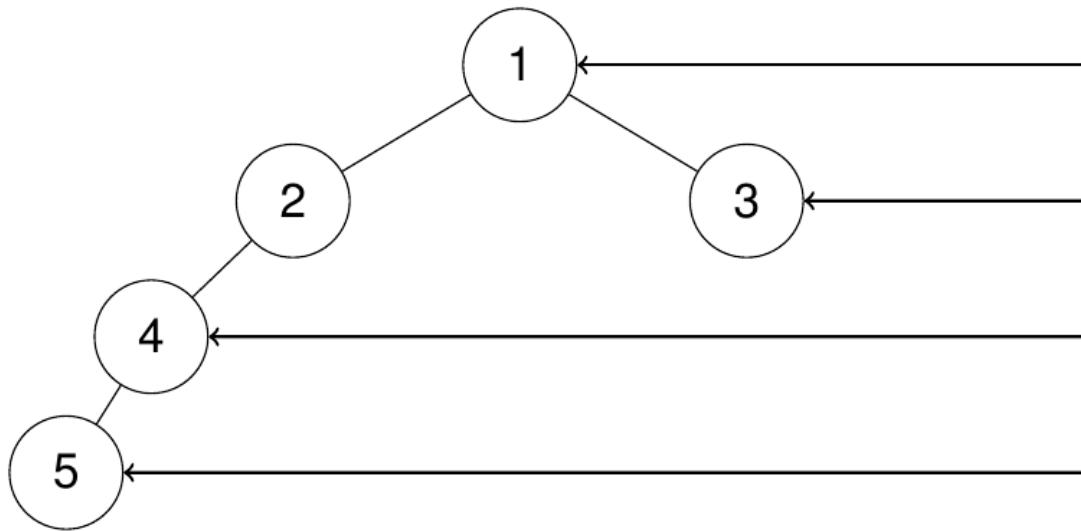


示例 2:

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

输出: `[1,3,4,5]`

解释:



示例 3:

输入: root = [1,null,3]

输出: [1,3]

示例 4:

输入: root = []

输出: []

题解

采用模板中层次遍历的方法遍历二叉树，并在将每一层最后一个节点值添加到result中。

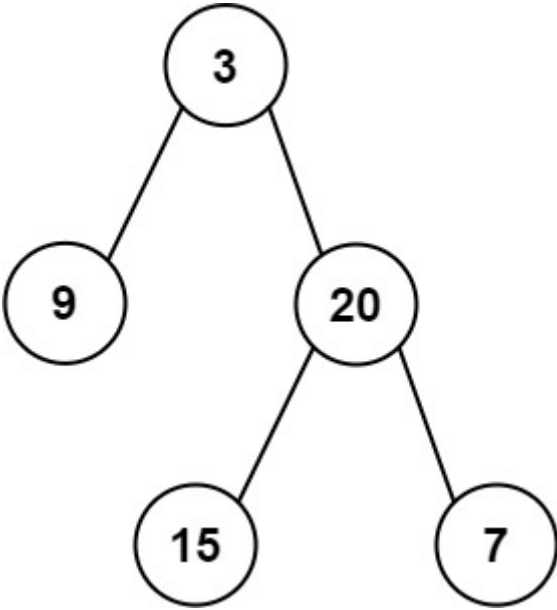
```
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        result = []
        queue = deque([root])
        while queue:
            n = len(queue)
            for i in range(n):
                node = queue.popleft()
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(node.val)
        return result
```

637. 二叉树的层平均值

题目描述

给定一个非空二叉树的根节点 `root`，以数组的形式返回每一层节点的平均值。与实际答案相差 10^{-5} 以内的答案可以被接受。

示例 1:

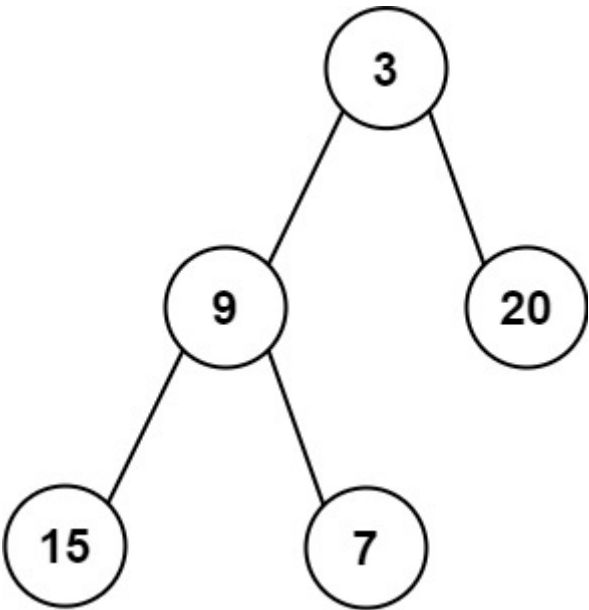


输入: `root = [3,9,20,null,null,15,7]`

输出: `[3.00000,14.50000,11.00000]`

解释: 第 0 层的平均值为 3, 第 1 层的平均值为 14.5, 第 2 层的平均值为 11。
因此返回 `[3, 14.5, 11]`。

示例 2:



输入: `root = [3,9,20,15,7]`

输出: `[3.00000,14.50000,11.00000]`

题解

本题也是根据模板进行层次遍历，只是在遍历每一层时记录其平均值。

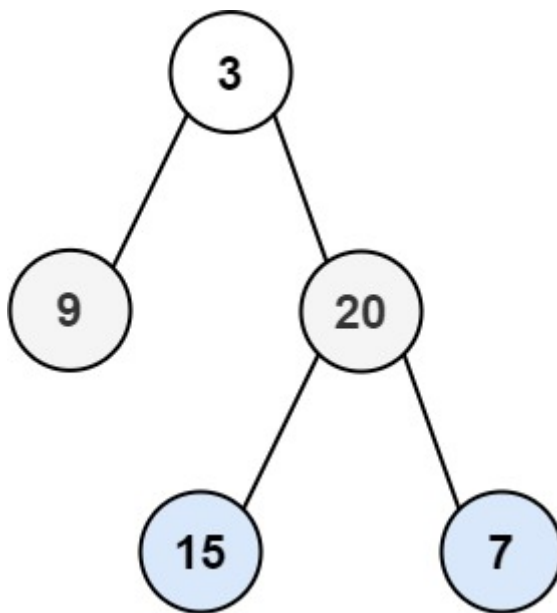
```
class Solution:
    def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:
        average = []
        queue = deque([root])
        while queue:
            n = len(queue)
            total = 0
            for _ in range(n):
                node = queue.popleft()
                total += node.val
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            average.append(total/n)
        return average
```

102. 二叉树的层次遍历

题目描述：

给你二叉树的根节点 `root`，返回其节点值的 **层序遍历**。（即逐层地，从左到右访问所有节点）。

示例 1：



输入: `root = [3,9,20,null,null,15,7]`

输出: `[[3],[9,20],[15,7]]`

示例 2：

输入: `root = [1]`

输出: `[[1]]`

示例 3:

输入: `root = []`
输出: `[]`

题解

本题是标准的层次遍历题目，用temp记录每一层的节点值，output记录所有层的节点值。

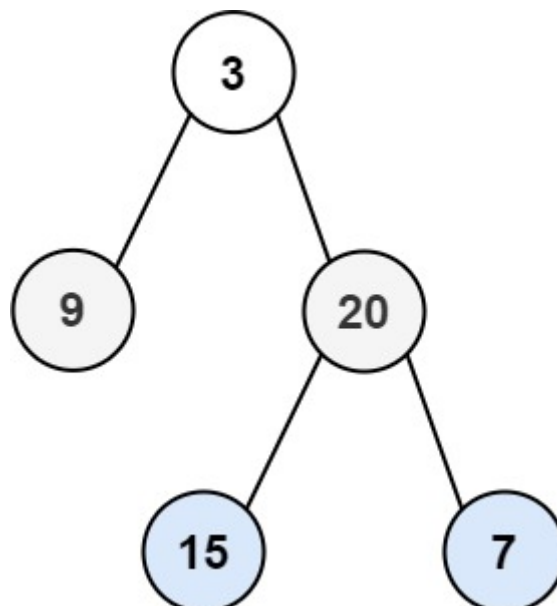
```
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        output = []
        queue = deque([root])
        while queue:
            n = len(queue)
            temp = []
            for _ in range(n):
                node = queue.popleft()
                temp.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            output.append(temp)
        return output
```

103. 二叉树的锯齿形层序遍历

题目描述

给你二叉树的根节点 `root`，返回其节点值的 **锯齿形层序遍历**。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

示例 1:



输入: root = [3,9,20,null,null,15,7]
输出: [[3],[20,9],[15,7]]

示例 2:

输入: root = [1]
输出: [[1]]

示例 3:

输入: root = []
输出: []

题解

本题在层次遍历的基础上，添加了“先从左到右，再从右到左遍历”的变化，可以用depth记录奇偶层，对于奇数层，使用append添加每层的节点值，对于偶数层，使用appendleft添加每层的节点值。这里只改变temp的append方式，不改层次遍历的顺序。

```
class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        queue = deque([root])
        result = []
        depth = 0
        while queue:
            n = len(queue)
            depth += 1
            temp = deque([])
            for _ in range(n):
                node = queue.popleft()
                if depth%2 == 0:
                    temp.appendleft(node.val)
                else:
                    temp.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            result.append(list(temp))
        return result
```

5.3 题目（图的广度优先搜索）

909. 蛇梯棋

题目描述

给你一个大小为 $n \times n$ 的整数矩阵 `board`，方格按从 1 到 n^2 编号，编号遵循 [转行交替方式](#)，**从左下角开始**（即，从 `board[n - 1][0]` 开始）的每一行改变方向。

你一开始位于棋盘上的方格 1。每一回合，玩家需要从当前方格 `curr` 开始出发，按下述要求前进：

- 选定目标方格 `next`，目标方格的编号在范围 $[curr + 1, \min(curr + 6, n^2)]$ 。
 - 该选择模拟了掷 **六面体骰子** 的情景，无论棋盘大小如何，玩家最多只能有 6 个目的地。
- 传送玩家：如果目标方格 `next` 处存在蛇或梯子，那么玩家会传送到蛇或梯子的目的地。否则，玩家传送到目标方格 `next`。
- 当玩家到达编号 n^2 的方格时，游戏结束。

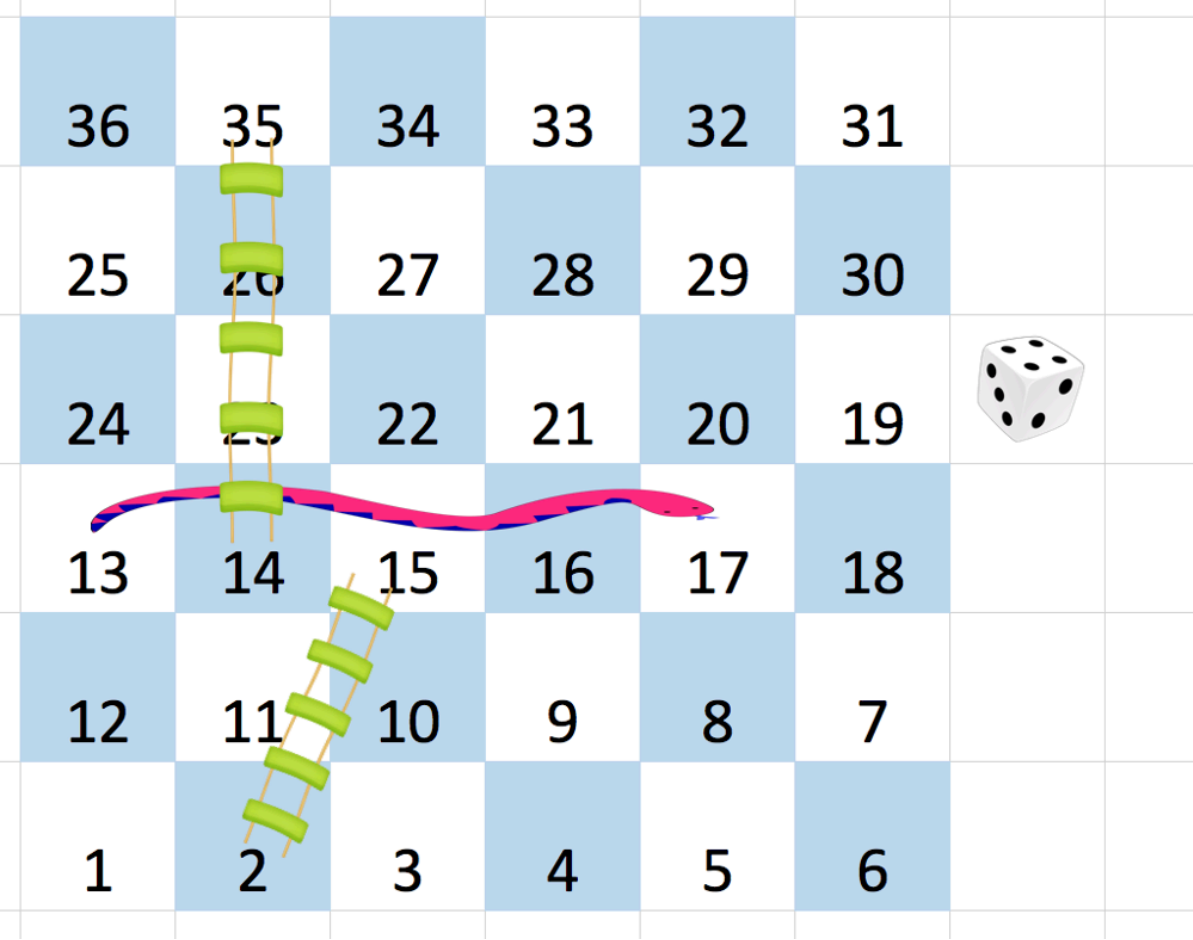
如果 `board[r][c] != -1`，位于 r 行 c 列的棋盘格中可能存在“蛇”或“梯子”。那个蛇或梯子的目的地将会是 `board[r][c]`。编号为 1 和 n^2 的方格不是任何蛇或梯子的起点。

注意，玩家在每次掷骰的前进过程中最多只能爬过蛇或梯子一次：就算目的地是另一条蛇或梯子的起点，玩家也 **不能** 继续移动。

- 举个例子，假设棋盘是 `[[-1,4],[-1,3]]`，第一次移动，玩家的目标方格是 2。那么这个玩家将会顺着梯子到达方格 3，但 **不能** 顺着方格 3 上的梯子前往方格 4。（简单来说，类似飞行棋，玩家掷出骰子点数后移动对应格数，遇到单向的路径（即梯子或蛇）可以直接跳到路径的终点，但如果多个路径首尾相连，也不能连续跳多个路径）

返回达到编号为 n^2 的方格所需的最少掷骰次数，如果不可能，则返回 -1。

示例 1：



输入: board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],
[-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]

输出: 4

解释:

首先, 从方格 1 [第 5 行, 第 0 列] 开始。

先决定移动到方格 2 , 并必须爬过梯子移动到到方格 15 。

然后决定移动到方格 17 [第 3 行, 第 4 列], 必须爬过蛇到方格 13 。

接着决定移动到方格 14 , 且必须通过梯子移动到方格 35 。

最后决定移动到方格 36 , 游戏结束。

可以证明需要至少 4 次移动才能到达最后一个方格, 所以答案是 4 。

示例 2:

输入: board = [[-1,-1],[-1,3]]

输出: 1

题解

本体首先需要编写坐标转换函数id2rc(id), 将方格编号转换为矩阵坐标, 需要注意的是最大的号码(示例1中的36) 对应于[0][0], 并且列数需要根据奇偶行数进行变换。

其次通过广度优先搜索方法进行遍历, 本题在队列的元素中, 需要添加step, 用于记录最短步数。在遍历过程中, 需要用用一个哈希表记录节点是否已访问, 已访问的节点不添加到队列。因为重复遍历节点相当于绕路到同一节点, 肯定不是最小步数, 所以不需要再添加该节点。

```
class Solution:
    def snakesAndLadders(self, board: List[List[int]]) -> int:
        n = len(board)
        visted = set()
        def id2rc(id):
            r,c=(id-1)//n, (id-1)%n
            if r%2==1:
                c = n-1-c
            return n-1-r,c

        queue = deque([(1,0)])
        while queue:
            idx,step = queue.popleft()
            for i in range(1,1+6):
                idx_next = idx + i
                if idx_next > n*n:
                    break
                x_next,y_next = id2rc(idx_next)
                if board[x_next][y_next] != -1:
                    idx_next = board[x_next][y_next]
                if idx_next == n*n:
                    return step + 1
                if idx_next not in visted:
                    visted.add(idx_next)
                    queue.append((idx_next,step+1))

        return -1
```


433. 最小基因变化

题目描述

基因序列可以表示为一条由 8 个字符组成的字符串，其中每个字符都是 'A'、'C'、'G' 和 'T' 之一。

假设我们需要调查从基因序列 `start` 变为 `end` 所发生的基因变化。一次基因变化就意味着这个基因序列中的一个字符发生了变化。

- 例如，`"AACCGGTT" --> "AACCGGTA"` 就是一次基因变化。

另有一个基因库 `bank` 记录了所有有效的基因变化，只有基因库中的基因才是有效的基因序列。（变化后的基因必须位于基因库 `bank` 中）

给你两个基因序列 `start` 和 `end`，以及一个基因库 `bank`，请你找出并返回能够使 `start` 变化为 `end` 所需的最少变化次数。如果无法完成此基因变化，返回 `-1`。

注意：起始基因序列 `start` 默认是有效的，但是它并不一定会出现在基因库中。

示例 1：

```
输入：start = "AACCGGTT", end = "AACCGGTA", bank = ["AACCGGTA"]
输出：1
```

示例 2：

```
输入：start = "AACCGGTT", end = "AAACGGTA", bank =
["AACCGGTA", "AACCGCTA", "AAACGGTA"]
输出：2
```

示例 3：

```
输入：start = "AAAAACCC", end = "AACCCCCC", bank =
["AAAACCCC", "AAACCCCC", "AACCCCCC"]
输出：3
```

题解

本题将基因当作一个节点，对他进行广度优先搜索的方式为：

- 1.对基因中每个字母尝试替换为"A""G""C""T"，
- 2.判断变换后的基因是否属于bank，属于bank的基因可以作为下一个节点添加到队列中。在添加到队列后，bank中需要移除该基因，因为到该基因最小步数已找到。
- 3.如果变换后的基因是endGene，则返回最小步数。

```
class Solution:
    def minMutation(self, startGene: str, endGene: str, bank: List[str]) -> int:
        if len(startGene) != len(endGene):
            return -1
        if endGene not in bank:
            return -1
        queue = deque([(startGene,0)])
        bank = set(bank)
```

```

while queue:
    gene, step = queue.popleft()
    for i, x in enumerate(gene):
        for y in "AGCT":
            if x != y:
                new_gene = gene[:i] + y + gene[i+1:]
                if new_gene in bank:
                    if new_gene == endGene:
                        return step + 1
                    else:
                        queue.append((new_gene, step+1))
                        bank.remove(new_gene)

return -1

```

六、动态规划

6.1 算法简介

本章对应的题目为“一维动态规划”、“二维动态规划”、“Kadane算法”。

(1) 基本定义

动态规划（Dynamic Programming，简称 DP）是一种用于解决优化问题的算法策略，它通过将复杂问题分解为相对简单的子问题，并保存子问题的解以避免重复计算，从而达到降低时间复杂度的目的。它主要利用了问题的**最优子结构**性质和**重叠子问题**性质。

Kadane算法基于动态规划的思想，采用贪心策略来解决最大子数组和问题，因此也放于这一章。

(2) 应用场景

资源分配问题、文本编辑距离问题、股票买卖问题等

(3) python模板

动态规划一般分为三步：1.状态定义；2.设计状态转移方程；3.处理边界条件以及初始值。

一维动态规划——斐波那契数列

```

def fibonacci(n):
    # 边界条件：n等于0或者1时，直接返回n
    if n == 0 or n == 1:
        return n
    # 状态定义：dp[i]定义为斐波那契数列每一个位置的值
    dp = [0] * (n + 1)
    # 初始化前两个元素
    dp[0] = 0
    dp[1] = 1

    for i in range(2, n + 1):
        # 状态转移方程：斐波那契数列当前元素值等前两个元素值的和
        dp[i] = dp[i - 1] + dp[i - 2]
    # 返回第n个dp元素
    return dp[n]

```

多维动态规划——二维网格中的路径总数问题

```
def uniquePaths(m, n):  
    # 状态定义: dp[i][j] 表示机器人到达网格中第 i 行第 j 列位置的不同路径数量。  
    dp = [[0 for _ in range(n)] for _ in range(m)]  
    # 边界条件: 初始化第一行和第一列, 到达第一行和第一列的每个位置分别只能往右和往下走, 所以路径  
    # 数都为 1  
    for j in range(n):  
        dp[0][j] = 1  
    for i in range(m):  
        dp[i][0] = 1  
    # 从第二行第二列开始遍历网格  
    for i in range(1, m):  
        for j in range(1, n):  
            # 状态转移方程: 到达当前位置的路径数等于到达上方位置和左方位置的路径数之和。  
            # 根据题目变化, 例如添加障碍。  
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]  
    # 返回到达右下角位置的不同路径数量  
    return dp[m - 1][n - 1]
```

6.2 题目（一维动态规划）

70. 爬楼梯

题目描述

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢

示例 1:

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

示例 2:

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

题解

爬楼梯问题实际上是斐波那契数列问题。考虑爬到最后一个台阶时可能爬了1个台阶或者2个台阶, 因此有 $dp[n]=dp[n-1]+dp[n-2]$

#常规写法

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

用迭代的形式写dp问题，可以将空间复杂度降为O(1)。

空间优化形式

```
class Solution:
    def climbStairs(self, n: int) -> int:
        a,b=1,1
        for i in range(n-1):
            a ,b = b, a+b
        return b
```

198 打家劫舍

题目描述

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

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

示例 1:

输入: [1,2,3,1]

输出: 4

解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。

偷窃到的最高金额 = 1 + 3 = 4 。

示例 2:

输入: [2,7,9,3,1]

输出: 12

解释: 偷窃 1 号房屋 (金额 = 2)，偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。

偷窃到的最高金额 = 2 + 9 + 1 = 12 。

题解

本题的状态dp[k]定义为偷窃k间房屋时，能偷窃到的最高金额。dp[k]的计算分两种情况：

- 1.偷窃第k间房屋，此时 $dp[k]=dp[k-2]+nums[k-1]$ ，这里 $nums[k-1]$ 是第k间房屋的金额；
- 2.不偷窃第k间房屋，此时 $dp[k]=dp[k-1]$ 。

最高金额为两种情况的最大值，由此可得状态方程 $dp[k] = \max(dp[k-1], dp[k-2]+nums[k-1])$ 。

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

139. 单词拆分

题目描述

给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

示例 1：

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

示例 2：

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

示例 3：

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

题解

本题的状态 $dp[i]$ 定义为字符串前 i 个字符是否能利用字符串中出现的一个或多个单词拼接而成。 $dp[i]$ 的状态转移方程如下：

$$dp[i] = dp[i] \&\& \text{check}(s[i..j-1])$$

其中 $\text{check}(s[i..j-1])$ 表示子串 $s[i..j-1]$ 是否出现在字典中。 $dp[i]$ 为True时表示字符串 s 前 i 个字符可以用`wordDict`表示，若此时子串 $s[i..j-1]$ 出现在字典中，则字符串 s 的前 j 个字符可以用`wordDict`表示。

本题将状态初始化为False，其中 $dp[0]$ 初始化为True，然后遍历所有子串。

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

322. 零钱兑换

题目描述

给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。

计算并返回可以凑成总金额所需的 **最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

你可以认为每种硬币的数量是无限的。

示例 1:

输入: `coins = [1, 2, 5]`, `amount = 11`
 输出: 3
 解释: $11 = 5 + 5 + 1$

示例 2:

输入: `coins = [2]`, `amount = 3`
 输出: -1

示例 3:

输入: `coins = [1]`, `amount = 0`
 输出: 0

题解

本题的状态 $dp[i]$ 定义为组成金额 i 需要的最少的硬币个数， $dp[i]$ 的状态转移方程如下：

$$dp[i] = \min_{j=0,1,\dots,n-1} dp[i - c_j] + 1$$

其中， c_j 表示硬币 j 的金额。假设我们使用的最后一枚硬币金额为 c_j ，那么金额 i 的状态需要从金额 $i - c_j$ 中转移过来，即组成金额 $i - c_j$ 的最小硬币数量再加1（最后一枚硬币 c_j ）。代码将 dp 状态初始化为一个极大值，如果 dp 值不变则表示当前的金额无法组成，返回-1。

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [1000000000000]*(amount+1)
        dp[0] = 0
        for coin in coins:
            for x in range(coin, amount+1):
                dp[x] = min(dp[x], dp[x-coin] + 1)
        return dp[amount] if dp[amount] != 1000000000000 else -1
```

300. 最长递增子序列

题目描述

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

子序列 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列

示例 1:

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

示例 2:

输入: `nums = [0,1,0,3,2,3]`
输出: 4

示例 3:

输入: `nums = [7,7,7,7,7,7,7]`
输出: 1

题解

本题的状态 $dp[i]$ 定义为以元素 $nums[i]$ 结尾的最大子序列长度。 $dp[i]$ 的状态转移方程如下：

设 $j \in [0, i)$ ，在计算 $dp[i]$ 时需要遍历 $dp[j]$ ，分为两种情况：

1. 当 $nums[i] > nums[j]$ 时， $nums[i]$ 可以接在 $nums[j]$ 之后，因此 $dp[i] = \max(dp[i], dp[j] + 1)$
2. 当 $nums[i] \leq nums[j]$ 时， $nums[i]$ 不可以接在 $nums[j]$ 之后， $dp[i]$ 不大于 $dp[j]$ ，这种情况可以直接跳过

最后返回 dp 的最大值，即可得到最长严格递增子序列的长度。

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

6.3 题目（多维动态规划）

120. 三角形最小路径和

题目描述

给定一个三角形 `triangle`，找出自顶向下的最小路径和。

每一步只能移动到下一行中相邻的结点上。**相邻的结点**在这里指的是 **下标与上一层结点的下标相同或者等于上一层结点的下标 + 1** 的两个结点。也就是说，如果正位于当前行的下标 `i`，那么下一步可以移动到下一行的下标 `i` 或 `i + 1`。

示例 1:

输入: `triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]`

输出: 11

解释: 如下面简图所示:

```

    2
   3 4
  6 5 7
 4 1 8 3
```

自顶向下的最小路径和为 11（即， $2 + 3 + 5 + 1 = 11$ ）。

示例 2:

输入: `triangle = [[-10]]`

输出: -10

题解

本题的状态 $dp[i][j]$ 定义为到达第 i 行，下标为 j 处 (i,j) 的最小路径和。 $dp[i][j]$ 的状态转移方程如下：

$$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-1]) + triangle[i][j]$$

因为每一步只能移动到下一行的相邻节点处，所以走到 $dp[i][j]$ 处的前一个位置只能是 $[i-1][j]$ 和 $[i-1][j-1]$ 。因此最小路径和是前一步最小路径和加上当前位置的路径。

本题的边界条件要考虑两种情况。对于 $j=0$ 时的情况，此时前一个位置只有 $[i-1][j]$ ， $[i-1][j-1]$ 不存在。状态转移方程可改为：

$$dp[i][0] = dp[i-1][0] + triangle[i][0]$$

对于 $dp[i][i]$ ，因为 $triangle[i-1][i]$ 不存在，所以 $dp[i-1][i]$ 是 0，此时的状态转移方程可改为：

$$dp[i][i] = dp[i-1][i-1] + triangle[i][i]$$

最后返回最后一行的最小值即可。

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        n = len(triangle)
        dp = [[0]*n for _ in range(n)]
        dp[0][0] = triangle[0][0]
        for i in range(1,n):
            dp[i][0] = dp[i-1][0] + triangle[i][0]
            for j in range(1,i+1):
                dp[i][j] = min(dp[i-1][j-1],dp[i-1][j]) + triangle[i][j]
            dp[i][i] = dp[i-1][i-1] + triangle[i][i]
        return min(dp[-1])
```

64. 最小路径和

给定一个包含非负整数的 $m \times n$ 网格 `grid`，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例 1：

1	3	1
1	5	1
4	2	1

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

输出: 7

解释: 因为路径 `1→3→1→1→1` 的总和最小。

示例 2：

输入: `grid = [[1,2,3],[4,5,6]]`

输出: 12

题解

本题的状态 $dp[i][j]$ 定义为到达位置 $[i][j]$ 处的路径最小之和。 $dp[i][j]$ 的状态转移方程如下：

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$$

因为每次只能向下或者向右移动一步，所以走到 $dp[i][j]$ 处的前一个位置只能是 $[i-1][j]$ 和 $[i][j-1]$ 。因此最小路径和是前一步最小路径和加上当前位置的路径。

本题的边界条件需要考虑两种情况：对于j=0的情况，此时前一个位置只能是[i-1][j]，状态转移方程可改为：

$$dp[i][0] = dp[i-1][0] + grid[i][0]$$

同样地，对于i=0的情况，此时前一个位置只能是[0][j-1]，状态转移方程可改为：

$$dp[0][j] = dp[0][j-1] + grid[0][j]$$

最后返回右下角元素的dp值即可。

```
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        m = len(grid)
        n = len(grid[0])

        dp = [[0]*n for _ in range(m)]
        dp[0][0] = grid[0][0]
        for i in range(1,m):
            dp[i][0] = dp[i-1][0] + grid[i][0]
        for j in range(1,n):
            dp[0][j] = dp[0][j-1] + grid[0][j]

        for i in range(1,m):
            for j in range(1,n):
                dp[i][j] = min(dp[i-1][j],dp[i][j-1]) + grid[i][j]
        return dp[-1][-1]
```

63. 不同路径II

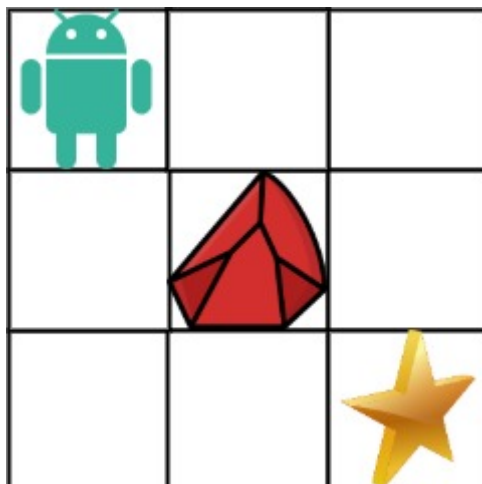
题目描述

给定一个 $m \times n$ 的整数数组 `grid`。一个机器人初始位于 **左上角**（即 `grid[0][0]`）。机器人尝试移动到 **右下角**（即 `grid[m - 1][n - 1]`）。机器人每次只能向下或者向右移动一步。

网格中的障碍物和空位置分别用 `1` 和 `0` 来表示。机器人的移动路径中不能包含 **任何** 有障碍物的方格。

返回机器人能够到达右下角的不同路径数量

示例 1:



输入: `obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]`

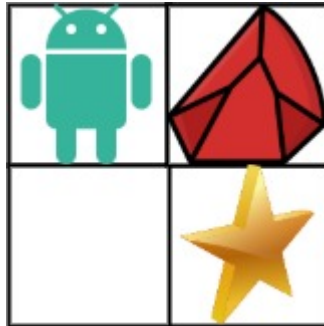
输出: 2

解释: 3x3 网格的正中间有一个障碍物。

从左上角到右下角一共有 2 条不同的路径:

1. 向右 -> 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右 -> 向右

示例 2:



输入: `obstacleGrid = [[0,1],[0,0]]`

输出: 1

题解

本题的状态 $dp[i][j]$ 定义为到达位置 $[i][j]$ 处的不同路径数量。 $dp[i][j]$ 的状态转移方程如下:

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

与不同路径I相比, 本题多了障碍物。在移动过程中, 如果碰到障碍物, 则当前位置的路径数量为 0, 可以忽略, 直接考虑无障碍物的情况, 即 $obstacleGrid[i][j] = 0$ 的情况。

和前一题相同, 本题的边界条件也是考虑 $i=0$ 和 $j=0$ 的情况。

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m = len(obstacleGrid)
        n = len(obstacleGrid[0])
        dp = [[0]*n for _ in range(m)]
        #初始化
        if obstacleGrid[0][0] == 0:
            dp[0][0] = 1
        else:
            return 0
        #边界
        for i in range(1,m):
            if obstacleGrid[i][0] == 0:
                dp[i][0] = dp[i-1][0]
        for j in range(1,n):
            if obstacleGrid[0][j] == 0:
                dp[0][j] = dp[0][j-1]
        #状态转移
        for i in range(1,m):
            for j in range(1,n):
                if obstacleGrid[i][j] == 0:
                    dp[i][j] = dp[i-1][j] + dp[i][j-1]
        return dp[-1][-1]
```

5. 最长回文子串

题目描述

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

示例 1:

输入: `s = "babad"`

输出: `"bab"`

解释: `"aba"` 同样是符合题意的答案。

示例 2:

输入: `s = "cbbd"`

输出: `"bb"`

题解

本题的状态 $dp[i][j]$ 定义为子串 $s[i:j]$ 是否为回文子串。 $dp[i][j]$ 的状态转移方程如下:

$$\text{当 } s[i] == s[j] \text{ 时, } dp[i][j] = dp[i+1][j-1]$$

本题只有在当前子串去掉首位字母后，剩下的子串是回文子串，且首位字母相同时，当前子串是回文子串。换句话说，当前子串的首位字母相同时，它的状态和去掉首位字母后的子串状态相同，其他情况下当前子串不是回文子串。

本题的边界条件需要考虑两种，一种是 $i > j$ ，子串不成立；另一种是 $i = j$ ，此时只有一个字符，子串是回文子串，即 $dp[i][i] = \text{True}$ 。

在遍历过程中，我们需要根据子串的长度进行遍历，如果当前子串是回文子串，则比较它和当前最长子串的长度，记录最长子串的长度以及起始位置，最后返回最长子串。

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        max_len = 1
        start = 0
        dp = [[False]*n for _ in range(n)]
        for i in range(n):
            dp[i][i] = True
        if n < 2:
            return s

        for L in range(2, n+1):
            for i in range(n):
                j = i+L-1
                if j >= n:
                    break
                if s[i] != s[j]:
                    continue
                else:
                    if L == 2:
```

```

        dp[i][j] = True
    else:
        dp[i][j] = dp[i+1][j-1]
    if dp[i][j] and L > max_len:
        start = i
        max_len = L
    return s[start:start+max_len]

```

97. 交错字符串

题目描述

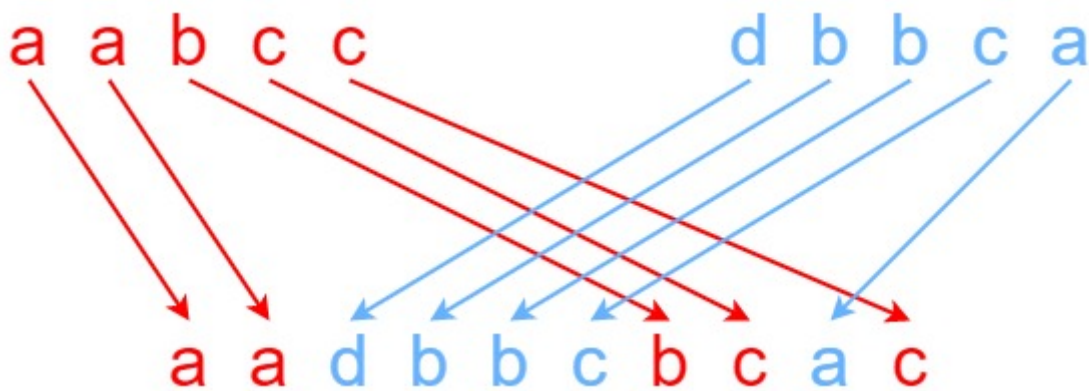
给定三个字符串 `s1`、`s2`、`s3`，请你帮忙验证 `s3` 是否是由 `s1` 和 `s2` **交错** 组成的。

两个字符串 `s` 和 `t` **交错** 的定义与过程如下，其中每个字符串都会被分割成若干 **非空** 子字符串：

- $s = s_1 + s_2 + \dots + s_n$
- $t = t_1 + t_2 + \dots + t_m$
- $|n - m| \leq 1$
- **交错** 是 $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + \dots$ 或者 $t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + \dots$

注意： `a + b` 意味着字符串 `a` 和 `b` 连接。

示例 1:



输入: `s1 = "aabcc", s2 = "dbbca", s3 = "aadbcbcbcac"`
 输出: `true`

示例 2:

输入: `s1 = "aabcc", s2 = "dbbca", s3 = "aadbcbaccc"`
 输出: `false`

示例 3:

输入: `s1 = "", s2 = "", s3 = ""`
 输出: `true`

题解

本题的状态 $dp[i][j]$ 定义为 $s1$ 的前 i 个元素和 $s2$ 的前 j 个元素是否能组成 $s3$ 的前 $i+j$ 个元素。如果 $s1$ 的第 i 个元素等于 $s3$ 的第 $i+j$ 个元素，则 $dp[i][j]$ 取决于 $dp[i-1][j]$ ，如果 $s2$ 的第 j 个元素等于 $s3$ 的第 $i+j$ 个元素时，则 $dp[i][j]$ 取决于 $dp[i][j-1]$ 。因此状态转移方程为：

$$\begin{aligned} \text{if } s1[i-1] == s3[i+j-1], dp[i][j] &= dp[i-1][j] \\ \text{if } s2[j-1] == s3[i+j-1], dp[i][j] &= dp[i][j-1] \end{aligned}$$

初始化 $dp[0][0]$ 为True。为了避免 dp 值通过 $s1$ 计算为True后，又被 $s2$ 计算为False，因此代码在 $s2$ 处添加了一个 $or dp[i][j]$ 。

```
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        m = len(s1)
        n = len(s2)
        t = len(s3)

        if m+n != t:
            return False

        dp = [[False]*(n+1) for _ in range(m+1)] #两个字符串的dp题一般长度为n+1和m+1,
        此时的dp[0][0]表示两个空字符串的操作。
        dp[0][0] = True

        for i in range(m+1):
            for j in range(n+1):
                if i>0 and s1[i-1] == s3[i+j-1]:
                    dp[i][j] = dp[i-1][j] #可以在此处也可以添加 or dp[i][j], 结果无影响
                if j>0 and s2[j-1] == s3[i+j-1]:
                    dp[i][j] = dp[i][j-1] or dp[i][j]
        return dp[-1][-1]
```

72. 编辑距离

题目描述

给你两个单词 `word1` 和 `word2`，请返回将 `word1` 转换成 `word2` 所使用的最少操作数。

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

示例 1：

```
输入: word1 = "horse", word2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2:

```
输入: word1 = "intention", word2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

题解

本题的状态 $dp[i][j]$ 表示将word1前i个字符转化为word2前j个字符所需要的最少操作数（编辑距离）。当word1的第i个字符和word2的第j个字符相同时，由于最后一个字符不需要操作，因此 $dp[i][j]$ 也与 $dp[i-1][j-1]$ 相等。当两个字符不同时，只需要替换word1最后一个字符转变为 $dp[i-1][j-1]$ ；或删除掉word1一个字符转变为 $dp[i-1][j]$ ；或删除掉word2一个字符转变为 $dp[i][j-1]$ ，最少操作数在三者的最小值中+1即可。状态转移方程如下：

$$\begin{aligned} &\text{if } s1[i-1] == s2[j-1], dp[i][j] = dp[i-1][j-1] \\ &\text{else, } dp[i][j] = \min(dp[i-1][j-1], dp[i][j-1], dp[i-1][j]) + 1 \end{aligned}$$

本题的边界条件主要考虑两种，一种是 $i = 0$ 时，从word1到word2的最少操作数等于word2的长度（对word1进行插入操作）。另一种是 $j = 0$ 时，从word1到word2的最少操作数等于word1的长度（对word1进行删除操作）。本题的初始化条件 $dp[0][0]$ 为0。

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m = len(word1)
        n = len(word2)
        dp = [[0]*(n+1) for _ in range(m+1)]

        for i in range(m+1):
            dp[i][0] = i
        for j in range(n+1):
            dp[0][j] = j

        for i in range(1,m+1):
            for j in range(1,n+1):
                if word1[i-1] == word2[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                else:
                    dp[i][j] = min(dp[i-1][j-1], dp[i][j-1], dp[i-1][j]) + 1

        return dp[-1][-1]
```

221. 最大正方形

题目描述

在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

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

示例 3:

输入: matrix = `[["0"]]`
输出: 0

题解

本题的状态 $dp[i][j]$ 定义为以 (i,j) 为右下角时，只包含1的最大正方形的边长。当 $matrix[i][j]$ 为0时，正方形不存在，边长为0。当 $matrix[i][j]$ 为1时， $dp[i][j]$ 值由上方，左方，左上方三个相邻位置的 dp 值决定，即相邻位置 dp 最小值加1。状态转移方程如下：

$$dp[i][j] = \min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1$$

以下提供三个3乘3的矩阵供大家理解为什么要取三个相邻位置的最小值。（有点类似于木桶的最短边问题）

```
0 1 1    1 1 0    1 1 1
1 1 1    1 1 1    1 1 1
1 1 1    1 1 1    0 1 1
```

对应的 dp 值如下：

```
0 1 1    1 1 0    1 1 1
1 1 2    1 2 1    1 2 2
1 2 2    1 2 2    0 1 2
```

本题的边界条件主要考虑两种，一种是 $i=0$ ，另一种是 $j=0$ 。在 $matrix[i][j]$ 为1的条件下，他们的 dp 值都为1。

```
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        m = len(matrix)
        n = len(matrix[0])
        dp = [[0]*n for _ in range(m)]
        max_len = 0
        for i in range(m):
            for j in range(n):
                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_len = max(max_len, dp[i][j])

        return max_len*max_len
```

6.4 题目（Kadane算法）

53. 最大子数组和

题目描述

给你一个整数数组 `nums`，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。**子数组**是数组中的一个连续部分。

示例 1：

输入: `nums = [-2,1,-3,4,-1,2,1,-5,4]`
输出: 6
解释: 连续子数组 `[4,-1,2,1]` 的和最大, 为 6。

示例 2:

输入: `nums = [1]`
输出: 1

示例 3:

输入: `nums = [5,4,-1,7,8]`
输出: 23

题解

本题状态`dp[i]`定义为以`nums[i]`为结尾的最大子数组和, 状态转移方程为:

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

初始化`dp[0] = nums[0]`。最后返回`dp` 的最大值即可。

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

本题常用迭代的形式解答。首先用`current_max`记录当前子数组和, `global_max`维护最大子数组和。然后状态转移方程如下。当前子数组和要么以`nums[i]`作为新的子数组的开始, 要么将`nums[i]`加入到之前的子数组中。

```
current_max = max(nums[i], current_max + nums[i]);
global_max = max(global_max, current_max)
```

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

        for i in range(1,n):
            current_max = max(nums[i], nums[i]+current_max) #记录当前子数组和
            global_max = max(global_max, current_max) #维护最大子数组和
        return global_max
```

918. 环形子数组的最大和

题目描述

给定一个长度为 n 的环形整数数组 `nums`，返回 `nums` 的非空子数组的最大可能和。

环形数组意味着数组的末端将会与开头相连呈环状。形式上，`nums[i]` 的下一个元素是 `nums[(i + 1) % n]`，`nums[i]` 的前一个元素是 `nums[(i - 1 + n) % n]`。

子数组最多只能包含固定缓冲区 `nums` 中的每个元素一次。形式上，对于子数组 `nums[i]`，`nums[i + 1]`， \dots ，`nums[j]`，不存在 $i \leq k_1, k_2 \leq j$ 其中 $k_1 \% n == k_2 \% n$ 。

示例 1:

输入: `nums = [1,-2,3,-2]`
输出: 3
解释: 从子数组 `[3]` 得到最大和 3

示例 2:

输入: `nums = [5,-3,5]`
输出: 10
解释: 从子数组 `[5,5]` 得到最大和 $5 + 5 = 10$

示例 3:

输入: `nums = [3,-2,2,-3]`
输出: 3
解释: 从子数组 `[3]` 和 `[3,-2,2]` 都可以得到最大和 3

题解

本题是53.最大子数组和的进阶版，最大子数组分为非环形和环形两种情况：

1. 非环形情况：最大子数组不跨越数组首尾，即普通的最大子数组和问题
2. 环形情况：最大子数组跨越数组首尾，可以用数组减去最小非环形子数组得到。但是当数组所有元素都小于0时，环形最大子数组和会得到0，此时直接返回非环形最大子数组和即可。

```
class Solution:
    def maxSubarraySumCircular(self, nums: List[int]) -> int:
        n = len(nums)
        dp_max = [0]*n
        dp_max[0] = nums[0]
        for i in range(1,n):
            dp_max[i] = max(nums[i], dp_max[i-1]+nums[i])

        non_circur_max = max(dp_max)
        dp_min = [0]*n
        dp_min[0] = nums[0]
        for i in range(1,n):
            dp_min[i] = min(nums[i], dp_min[i-1]+nums[i])
        circur_max = sum(nums) - min(dp_min)

        if circur_max == 0 and non_circur_max < 0: #数组所有元素为负数的情况
```

```
        return non_circur_max
    return max(non_circur_max, circur_max)
```

另一种计算环形最大子数组和的解法是，首先计算左侧nums[0:i]的最大和，然后从右往左遍历，计算右侧子数组nums[j:n]的和，并加上左侧nums[0:i]的和所得的最大值。如下所示。个人感觉没有第一种思路容易理解。

```
class Solution:
    def maxSubarraySumCircular(self, nums: List[int]) -> int:
        n = len(nums)
        leftMax = [0] * n
        leftMax[0], leftSum = nums[0], nums[0]
        pre, res = nums[0], nums[0]
        for i in range(1, n):
            pre = max(pre + nums[i], nums[i])
            res = max(res, pre) # 非环形最大子数组和
            leftSum += nums[i]
            leftMax[i] = max(leftMax[i-1], leftSum) # 左侧[0:i]最大子数组和
        rightsum = 0
        for i in range(n-1, 0, -1):
            rightsum += nums[i]
            res = max(res, rightsum + leftMax[i-1]) # 取非环形和环形的最大值
        return res
```

七、分治

7.1 算法简介

本章对应的题目为“分治”。

(1) 基本定义

分治算法 (Divide - and - Conquer) 是一种多分支递归算法。它将一个分解为若干个规模较小的子问题，这些子问题相互独立且与原问题类型相同，然后递归的解决这些子问题，最后将子问题的解合并得到原问题的解。

(2) 应用场景

归并排序、凸包问题、二叉搜索树、大数据处理等

(3) python模板

分治算法一般先“分”：递归分解，然后“治”：融合。以归并排序为模板。

```
class Solution:
    def merge_sort(self, nums):
        # 分解：如果数组长度小于等于 1，认为数组已经有序，直接返回
        if len(nums) < 2:
            return nums
        # 找到数组的中间位置
        mid = len(nums) // 2
        # 递归地对左半部分数组进行排序
        left = self.merge_sort(nums[:mid])
```

```

# 递归地对右半部分数组进行排序
right = self.merge_sort(nums[mid:])
# 合并两个有序的子数组
return self.merge(left, right)

def merge(self, left, right):
    result = []
    left_idx = 0
    right_idx = 0
    # 比较左右子数组的元素，将较小的元素依次添加到结果数组中
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] < right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1
    # 将左子数组中剩余的元素添加到结果数组中
    result += left[left_idx:]
    # 将右子数组中剩余的元素添加到结果数组中
    result += right[right_idx:]
    return result
def sortArray(self, nums: List[int]) -> List[int]:
    return self.merge_sort(nums)

```

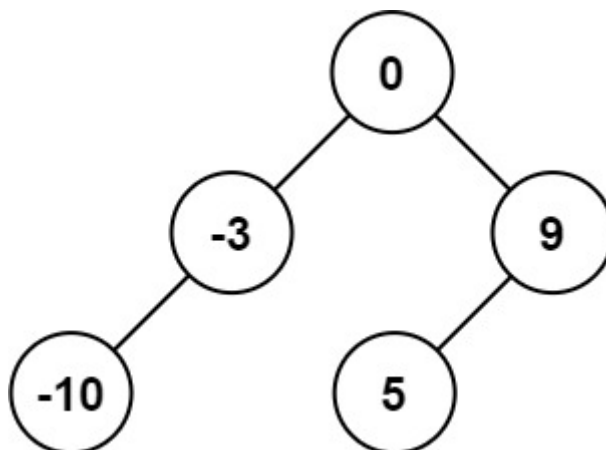
7.2 题目（分治）

108. 将有序数组转换为二叉搜索树

题目描述

给你一个整数数组 `nums`，其中元素已经按 **升序** 排列，请你将其转换为一棵平衡二叉搜索树。

示例 1:

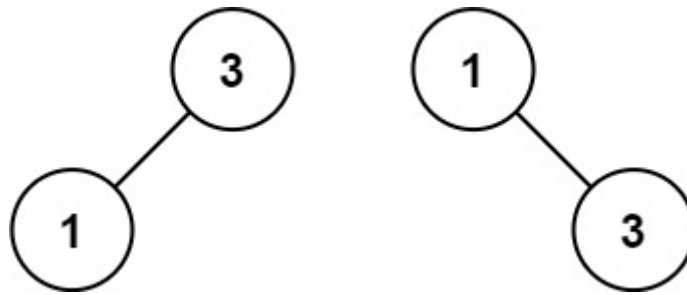


输入: `nums = [-10,-3,0,5,9]`

输出: `[0,-3,9,-10,null,5]`

解释: `[0,-10,5,null,-3,null,9]` 也将被视为正确答案:

示例 2:



输入: `nums = [1,3]`

输出: `[3,1]`

解释: `[1,null,3]` 和 `[3,1]` 都是高度平衡二叉搜索树。

题解

二叉搜索树（左子节点值小于根节点，右子节点值大于根节点）的中序遍历是升序数组。题目中给的升序数组即是二叉搜索树的中序遍历。为了获取平衡二叉搜索树，需要将数组的中间节点作为根节点。然后不断递归即可。本题的分治思想在于将问题转为递归左子树和右子树两个子问题，然后合并成一颗二叉搜索树。

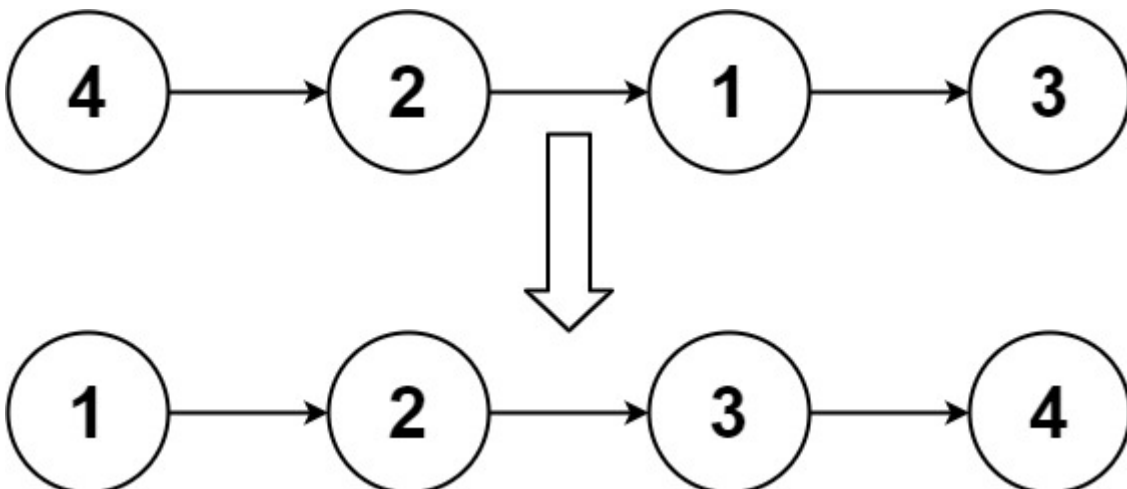
```
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        n = len(nums)
        def helper(left, right):
            if left > right:
                return None
            mid = left + (right - left) // 2
            root = TreeNode(nums[mid])
            root.left = helper(left, mid - 1) #子问题
            root.right = helper(mid + 1, right) #子问题
            return root
        return helper(0, n - 1)
```

148. 排序链表

题目描述

给你链表的头结点 `head`，请将其按 **升序** 排列并返回 **排序后的链表**。

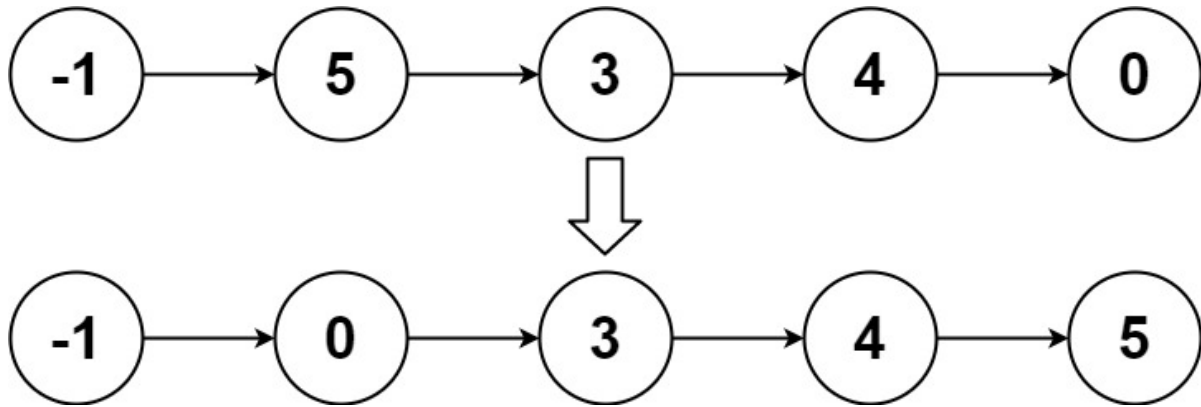
示例 1:



输入: head = [4,2,1,3]

输出: [1,2,3,4]

示例 2:



输入: head = [-1,5,3,4,0]

输出: [-1,0,3,4,5]

示例 3:

输入: head = []

输出: []

题解

本题采用归并排序方法，基本思路类似于模板，主要难点在于寻找链表的中点。这里采用快慢指针的方式来寻找中点，快指针移动速度是慢指针两倍，当快指针到达尾部时，慢指针到达中点。

```
class Solution:
    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        def merge_sort(head, tail):
            if not head:
                return head
            if head.next == tail:
                head.next = None #对于left，删除最右侧的节点，因为left最后节点是right的第一节点
                #对于right，删除最后一个空节点。
                return head
            slow = head
            fast = head
            while fast != tail:
                slow = slow.next
                fast = fast.next
                if fast != tail:
                    fast = fast.next
            mid = slow
            left = merge_sort(head, mid)
            right = merge_sort(mid, tail)
            return merge(left, right)

        def merge(left, right):
```

```

begin = ListNode(0)
temp = begin
while left and right:
    if left.val < right.val:
        temp.next = left
        left = left.next
    else:
        temp.next = right
        right = right.next
    temp = temp.next
if left:
    temp.next = left
if right:
    temp.next = right
return begin.next
return merge_sort(head, None)

```

427. 建立四叉树

题目描述

给你一个 $n * n$ 矩阵 `grid`，矩阵由若干 0 和 1 组成。请你用四叉树表示该矩阵 `grid`。

你需要返回能表示矩阵 `grid` 的四叉树的根结点。

四叉树数据结构中，每个内部节点只有四个子节点。此外，每个节点都有两个属性：

- `val`：储存叶子结点所代表的区域的值。1 对应 **True**，0 对应 **False**。注意，当 `isLeaf` 为 **False** 时，你可以把 **True** 或者 **False** 赋值给节点，两种值都会被判题机制 **接受**。
- `isLeaf`：当这个节点是一个叶子结点时为 **True**，如果它有 4 个子节点则为 **False**。

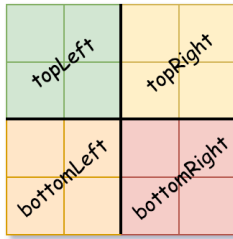
```

class Node {
    public boolean val;
    public boolean isLeaf;
    public Node topLeft;
    public Node topRight;
    public Node bottomLeft;
    public Node bottomRight;
}

```

我们可以按以下步骤为二维区域构建四叉树：

1. 如果当前网格的值相同（即，全为 0 或者全为 1），将 `isLeaf` 设为 **True**，将 `val` 设为网格相应的值，并将四个子节点都设为 **Null** 然后停止。
2. 如果当前网格的值不同，将 `isLeaf` 设为 **False**，将 `val` 设为任意值，然后如下图所示，将当前网格划分为四个子网格。
3. 使用适当的子网格递归每个子节点。



如果你想了解更多关于四叉树的内容，可以参考 [wiki](#)。

四叉树格式：

你不需要阅读本节来解决这个问题。只有当你想了解输出格式时才会这样做。输出为使用层序遍历后四叉树的序列化形式，其中 `null` 表示路径终止符，其下面不存在节点。

它与二叉树的序列化非常相似。唯一的区别是节点以列表形式表示 `[isLeaf, val]`。

如果 `isLeaf` 或者 `val` 的值为 `True`，则表示它在列表 `[isLeaf, val]` 中的值为 `1`；如果 `isLeaf` 或者 `val` 的值为 `False`，则表示值为 `0`。

示例 1：

0	1
1	0

输入: `grid = [[0,1],[1,0]]`
输出: `[[0,1],[1,0],[1,1],[1,1],[1,0]]`
解释: 此示例的解释如下:
请注意，在下面四叉树的图示中，0 表示 `false`，1 表示 `True`。

示例 2：

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

输入: `grid = [[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,1,1,1,1],`
`[1,1,1,1,1,1,1,1],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],`
`[1,1,1,1,0,0,0,0]]`
输出: `[[0,1],[1,1],[0,1],[1,1],[1,0],null,null,null,null,[1,0],[1,0],[1,1],[1,1]]`
解释: 网格中的所有值都不相同。我们将网格划分为四个子网格。
`topLeft`, `bottomLeft` 和 `bottomRight` 均具有相同的值。
`topRight` 具有不同的值, 因此我们将其再分为 4 个子网格, 这样每个子网格都具有相同的值。
解释如下图所示:

题解

本题主要是对一个正方形区域进行赋值, 分为两种情况。1.若所有节点值相等, 则他是叶子节点, 将`isleaf`赋值为`True`, `val`取决于每个节点的值。2.若节点值不相等, 则将`isleaf`赋值为`False`, `val`的值可以随意, 然后其划分为四个部分继续遍历。

```
class Solution:
    def construct(self, grid: List[List[int]]) -> 'Node':
        def dfs(r0,c0,r1,c1):
            if all(grid[r0][c0] == grid[r][c] for r in range(r0,r1) for c in
range(c0,c1)):
                return Node(grid[r0][c0]==1,True)
            else:
                return Node(
                    False, #也可以是True
                    False,
                    dfs(r0,c0,(r0+r1)//2,(c0+c1)//2),
                    dfs(r0,(c0+c1)//2,(r0+r1)//2,c1),
                    dfs((r0 + r1) // 2, c0, r1, (c0 + c1) // 2),
                    dfs((r0 + r1) // 2, (c0 + c1) // 2, r1, c1),
                )
        return dfs(0,0,len(grid),len(grid[0]))
```

八、回溯

8.1 算法简介

本章对应的题目为“回溯”。

(1) 基本定义

回溯算法 (backtrack) 是一种用于搜索问题解空间的通用算法策略。它采用深度优先搜索 (DFS) 的方式遍历解空间。在搜索过程中, 当发现当前路径不可能得到解时, 就回溯到上一个状态, 尝试其他可能的路径。

(2) 应用场景

组合问题、排列问题、N皇后问题

(3) python模板

以组合问题构建模板（从 n 个数字中选取 k 个数字的组合），回溯一般包含一个条件语句，用于添加结果；以及一个用于回溯的for循环语句，用于遍历所有结果。有时可以加入剪枝提高效率。

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        ans = [] #初始化结果列表，存储所有符合条件的组合
        path = [] #初始化路径列表（当前组合），在回溯过程中存储当前组合

        def backtrack(start): #回溯函数
            #剪枝：如果当前路径的剩余长度小于题目要求的剩余长度，直接返回
            if n - start + 1 < k - len(path):
                return
            #将当前组合的内容添加到结果列表中
            if len(path) == k:
                ans.append(path[:])
            else:
                #遍历从start到n的数字，
                for i in range(start, n+1):
                    #将当前数字加入path中
                    path.append(i)
                    #调用回溯函数，此处传入i+1，说明下一个数字从i+1开始
                    backtrack(i+1)
                    #将最后一个数字删除，搜索下一个数字
                    path.pop()

        #从数字1开始调用
        backtrack(1)
        return ans
```

8.2 题目（回溯）

17. 电话号码的字母组合

题目描述

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。



示例 1:

```
输入: digits = "23"
输出: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

示例 2:

```
输入: digits = ""  
输出: []
```

示例 3:

```
输入: digits = "2"  
输出: ["a","b","c"]
```

题解

本题首先要将数字和对应的字母用字典进行表示，然后不断回溯每个字母的组合，本题的每个数字代表不同的字母，因此在回溯过程中不需要考虑重复字母的问题。

```
class Solution:  
    def letterCombinations(self, digits: str) -> List[str]:  
        if not digits:  
            return []  
        phone = {  
            '2': ['a', 'b', 'c'],  
            '3': ['d', 'e', 'f'],  
            '4': ['g', 'h', 'i'],  
            '5': ['j', 'k', 'l'],  
            '6': ['m', 'n', 'o'],  
            '7': ['p', 'q', 'r', 's'],  
            '8': ['t', 'u', 'v'],  
            '9': ['w', 'x', 'y', 'z'],  
        }  
  
        def backtrack(start):  
            if len(path) == len(digits):  
                ans.append("".join(path))  
            else:  
                for letter in phone[digits[start]]:  
                    path.append(letter)  
                    backtrack(start+1)  
                    path.pop()  
  
        ans = []  
        path = []  
        backtrack(0)  
        return ans
```

77. 组合

题目描述

给定两个整数 n 和 k ，返回范围 $[1, n]$ 中所有可能的 k 个数的组合。你可以按 **任何顺序** 返回答案。

示例 1:

输入: $n = 4, k = 2$

输出:

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

示例 2:

输入: $n = 1, k = 1$

输出: `[[1]]`

题解

本题是模板题。

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        ans = []
        path = []

        def backtrack(start):
            if n - start + 1 < k - len(path):
                return
            if len(path) == k:
                ans.append(path[:])
            else:
                for i in range(start, n+1):
                    path.append(i)
                    backtrack(i+1)
                    path.pop()

        backtrack(1)
        return ans
```

46. 全排列

题目描述

给定一个不含重复数字的数组 `nums`，返回其 *所有可能的全排列*。你可以 **按任意顺序** 返回答案。

示例 1:

输入: `nums = [1,2,3]`

输出: `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`

示例 2:

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

示例 3:

输入: `nums = [1]`
输出: `[[1]]`

题解

相比于电话号码的字母组合，本题需要考虑重复数字的问题，因此在回溯过程中，需要删除重复字母的情况

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        if not nums:
            return []
        def backtrack(start): #本题start参数没有派上用场，可以删除
            if len(path) == len(nums):
                ans.append(path[:])
            else:
                for number in nums:
                    if number not in path:
                        path.append(number)
                        backtrack(start+1)
                        path.pop()

        ans = []
        path = []
        backtrack(1)
        return ans
```

39. 组合总和

题目描述

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

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

对于给定的输入，保证和为 `target` 的不同组合数少于 150 个。

示例 1:

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

示例 2:

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

示例 3:

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

题解

本题将目标值作为参数传入到回溯函数中, 若目标值-当前数字小于0, 则不需要再继续回溯。同时由于数字可以重复, 因此递归回溯函数的输入仍然是i, 不是i+1。

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        n = len(candidates)
        candidates.sort()

        def backtrack(start, target):
            if target == 0:
                ans.append(path[:])
            else:
                for i in range(start, n):
                    if target - candidates[i] < 0:
                        break
                    path.append(candidates[i])
                    backtrack(i, target - candidates[i]) # 由于数字可以重复, 输入仍然是i
                    path.pop()

        ans = []
        path = []
        backtrack(0, target)
        return ans
```

22. 括号生成

题目描述

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

示例 1:

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

示例 2:

输入: $n = 1$
输出: ["()"]

题解

本题左括号和右括号的使用数量是独立的，因此需要用left和right分别记录使用了多少个括号。回溯思路是：如果左括号数量不大于n，我们可以放一个左括号。如果右括号数量小于左括号的数量，我们可以放一个右括号。

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

        def backtrack(left, right):
            if len(path) == 2*n:
                ans.append("".join(path))
            if left < n:
                path.append("(")
                backtrack(left+1, right)
                path.pop()
            if right < left:
                path.append(")")
                backtrack(left, right+1)
                path.pop()

        ans = []
        path = []
        backtrack(0, 0)
        return ans
```

79. 单词搜索

题目描述

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

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

示例 1:

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

输入: board = `[["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]]`, word = `"ABCCED"`
输出: true

示例 2:

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

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

示例 3:

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

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

题解

本题需要遍历网络中的所有点，以每个点作为起点，向相邻格进行回溯搜索，并且判断是否搜索到目标单词word。对于回溯函数backtrack(i,j,k)，用i,j表示搜索的位置，k表示搜索到word中第几个字母。当board[i][j]!=word[k]时，这个点已经无法回复搜索到word，可以直接返回False；当k == len(word)-1时，表示搜索完毕，返回True。对于for循环，只要相邻节点在网格内，就需要回溯搜索。为了避免重复搜索，用visited矩阵记录是否已搜索。

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        m = len(board)
        n = len(board[0])
        directions = [(0,1),(0,-1),(-1,0),(1,0)]
        def backtrack(i,j,k):
            if board[i][j] != word[k]:
```

```

        return False
    if k == len(word) - 1:
        return True
    visited.add((i,j))
    result = False
    for di,dj in directions:
        new_i, new_j = i+di, j+dj
        if 0<=new_i<m and 0<=new_j<n:
            if (new_i,new_j) not in visited:
                if backtrack(new_i,new_j,k+1):
                    result = True

    visited.remove((i,j))
    return result
visited = set()
for i in range(m):
    for j in range(n):
        if backtrack(i,j,0):
            return True
return False

```

第二部分 数学

九、数学

本章对应的题目为“数学”，

9.1 基本定义

本章主要考察数学知识，因此不做相关简介，直接给出题目和题解

9.2 题目（数学）

9. 回文数

题目描述

给你一个整数 `x`，如果 `x` 是一个回文整数，返回 `true`；否则，返回 `false`。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

- 例如，`121` 是回文，而 `123` 不是。

示例 1:

输入: `x = 121`
输出: `true`

示例 2:

输入: `x = -121`
输出: `false`
解释: 从左向右读, 为 `-121` 。 从右向左读, 为 `121-` 。因此它不是一个回文数。

示例 3:

输入: $x = 10$

输出: `false`

解释: 从右向左读, 为 `01`。因此它不是一个回文数。

题解

如果是负数, 或者非零且能被10整除, 肯定不是回文数, 可以直接返回`False`。对于其他情况, 反转一半的数, 判断反转后的后半段数和前半段数是否相等即可。

```
class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0 or (x!=0 and x%10==0):
            return False
        reverse = 0 # 存储反转后一半的数
        while x > reverse:
            reverse = reverse * 10 + x % 10
            x = x//10
        return reverse == x or reverse//10 == x
```

66. 加一

题目描述

给定一个由 **整数** 组成的 **非空** 数组所表示的非负整数, 在该数的基础上加一。

最高位数字存放在数组的首位, 数组中每个元素只存储**单个**数字。

你可以假设除了整数 0 之外, 这个整数不会以零开头。

示例 1:

输入: `digits = [1,2,3]`

输出: `[1,2,4]`

解释: 输入数组表示数字 123。

示例 2:

输入: `digits = [4,3,2,1]`

输出: `[4,3,2,2]`

解释: 输入数组表示数字 4321。

示例 3:

输入: `digits = [9]`

输出: `[1,0]`

解释: 输入数组表示数字 9。

加 1 得到了 $9 + 1 = 10$ 。

因此, 结果应该是 `[1,0]`。

题解

本题用tmp表示当前位数是否需要加一，然后遍历所有位置的数字，进行加法操作即可。

```
class Solution:
    def plusOne(self, digits: List[int]) -> List[int]:
        tmp = 1
        n = len(digits)
        for i in range(len(digits)-1,-1,-1):
            digits[i] += tmp
            if digits[i] == 10:
                tmp = 1
                digits[i] = 0
            else:
                tmp = 0

        if tmp == 1:
            return [1]+digits
        else:
            return digits
```

172. 阶乘后的零

题目描述

给定一个整数 n ，返回 $n!$ 结果中尾随零的数量。

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

示例 1:

输入: $n = 3$
输出: 0
解释: $3! = 6$ ，不含尾随 0

示例 2:

输入: $n = 5$
输出: 1
解释: $5! = 120$ ，有一个尾随 0

示例 3:

输入: $n = 0$
输出: 0

题解

0一般是用5*2构成的，因此 $n!$ 尾零的数量取决于质因子2的数量和质因子5的数量，其中2的数量大于5的数量，因此只要考虑5的数量即可。

```
class Solution:
    def trailingZeroes(self, n: int) -> int:
        ans = 0
        for i in range(5, n+1, 5):
            while i%5==0:
                ans += 1
                i //= 5
        return ans
```

69. x 的平方根

题目描述

给你一个非负整数 x ，计算并返回 x 的 **算术平方根**。

由于返回类型是整数，结果只保留 **整数部分**，小数部分将被 **舍去**。

注意：不允许使用任何内置指数函数和算符，例如 `pow(x, 0.5)` 或者 `x ** 0.5`。

示例 1：

输入：x = 4
输出：2

示例 2：

输入：x = 8
输出：2
解释：8 的算术平方根是 $2.82842\dots$ ，由于返回类型是整数，小数部分将被舍去。

题解

二分查找，并且移动左右指针，判断其平方和是否大于x。

```
class Solution:
    def mySqrt(self, x: int) -> int:
        l, r = 0, x
        while l <= r:
            mid = (l+r)//2

            if mid * mid <= x:
                ans = mid
                l = mid + 1
            else:
                r = mid - 1
        return ans
```

50. Pow(x,n)

题目描述

实现 `pow(x, n)`，即计算 `x` 的整数 `n` 次幂函数（即，`xn`）。

示例 1:

输入: `x = 2.00000, n = 10`
输出: `1024.00000`

示例 2:

输入: `x = 2.10000, n = 3`
输出: `9.26100`

示例 3:

输入: `x = 2.00000, n = -2`
输出: `0.25000`
解释: $2^{-2} = 1/2^2 = 1/4 = 0.25$

题解

本题使用递归法，将N次幂拆解为 $(N/2, N/2)$ ， $(N/4, N/4)$ ，直到幂为0。

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        def quickly(N):
            if N == 0:
                return 1.0
            y = quickly(N//2)
            return y*y if N%2 == 0 else y*y*x
        return quickly(n) if n>0 else 1/quickly(-n)
```

十、位运算

10.1 算法简介

(1) 基本定义

位运算是一种直接对二进制位进行操作的运算

(2) 应用场景

数据压缩与加密、状态标志位、快速计算

(3) 常用代码 (python)

```
# 定义两个整数用于位运算
a = 5 # 二进制表示: 0101
b = 3 # 二进制表示: 0011

# 按位与 (&)
bitwise_and = a & b
# 按位或 (|)
bitwise_or = a | b
# 按位取反 (~)
bitwise_not_a = ~a
# 左移 (<<)
left_shift = a << 2
# 右移 (>>)
right_shift = a >> 1
```

10.2 题目 (位运算)

67. 二进制之和

题目描述

给你两个二进制字符串 `a` 和 `b` , 以二进制字符串的形式返回它们的和。

示例 1:

```
输入: a = "11", b = "1"
输出: "100"
```

示例 2:

```
输入: a = "1010", b = "1011"
输出: "10101"
```

题解

本题将两个字符串用0补齐位数, 然后按位置相加即可, 主要考虑进位的问题。

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        while len(a)>len(b):
            b = "0" + b
        while len(a)<len(b):
            a = "0" + a
        tmp = 0
        a, b = list(a), list(b)
        for i in range(len(a)-1, -1, -1):
            if int(a[i]) + int(b[i]) + tmp == 3:
                tmp = 1
                b[i] = "1"
            elif int(a[i]) + int(b[i]) + tmp == 2:
```

```

        tmp = 1
        b[i] = "0"
    elif int(a[i]) + int(b[i]) + tmp == 1:
        tmp = 0
        b[i] = "1"
    else:
        tmp = 0
        b[i] = "0"
if tmp == 1:
    return "".join(["1"] + b)
else:
    return "".join(b)

```

190. 颠倒二进制位

题目描述

颠倒给定的 32 位无符号整数的二进制位。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用 [二进制补码](#) 记法来表示有符号整数。因此，在 **示例 2** 中，输入表示有符号整数 `-3`，输出表示有符号整数 `-1073741825`。

示例 1：

输入：n = 00000010100101000001111010011100

输出：964176192 (00111001011110000010100101000000)

解释：输入的二进制串 00000010100101000001111010011100 表示无符号整数 43261596，因此返回 964176192，其二进制表示形式为 00111001011110000010100101000000。

示例 2：

输入：n = 11111111111111111111111111111101

输出：3221225471 (10111111111111111111111111111111)

解释：输入的二进制串 11111111111111111111111111111101 表示无符号整数 4294967293，因此返回 3221225471 其二进制表示形式为 10111111111111111111111111111111。

题解

本题不断利用 $(n \& 1)$ 提取最右位数字，然后放到答案中，接着左移这个数字，最后重复32二次可以获得颠倒的32位无符号数。


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

191. 位1的个数

题目描述

给定一个正整数 n ，编写一个函数，获取一个正整数的二进制形式并返回其二进制表达式中设置位的个数（也被称为[汉明重量](#)）。

示例 1:

输入: $n = 11$
 输出: 3
 解释: 输入的二进制串 1011 中，共有 3 个设置位。

示例 2:

输入: $n = 128$
 输出: 1
 解释: 输入的二进制串 10000000 中，共有 1 个设置位。

示例 3:

输入: $n = 2147483645$
 输出: 30
 解释: 输入的二进制串 11111111111111111111111111101 中，共有 30 个设置位。

题解

本题也是利用 $n \& 1$ 判断最右边的数是否为1，即是否为设置位，最后统计设置位数字即可。

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

136. 只出现一次的数字

题目描述

给你一个 **非空** 整数数组 `nums`，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

你必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

示例 1：

输入: `nums = [2,2,1]`
输出: `1`

示例 2：

输入: `nums = [4,1,2,1,2]`
输出: `4`

示例 3：

输入: `nums = [1]`
输出: `1`

题解

本题将所有数字做亦或[^]操作，相同数字亦或操作的结果是0,0和任何数字做亦或等于该数字

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        x = 0
        for num in nums:
            x ^= num
        return x
```

137. 只出现一次的数字II

给你一个整数数组 `nums`，除某个元素仅出现 **一次** 外，其余每个元素都恰出现 **三次**。请你找出并返回那个只出现了一次的元素。

你必须设计并实现线性时间复杂度的算法且使用常数级空间来解决此问题。

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`

题解

考虑数字的二进制形式，对于出现三次的数字，各二进制位出现的次数都是3的倍数，因此统计所有二进制位中的1的出现次数，并对3求余，就可以得到只出现一次的数。

由于各二进制位运算规则相同，因此只需要考虑1位即可。对于某个二进制位1的个数，对3取余有三种状态0, 1, 2，而状态变化分为两种：

1. 若输入为1，则状态按0-1-2-0的规律变化
2. 若输入为0，则状态不变

本题可以用两个二进制位表示三种状态，00-01-10-00，假设这两位是two、one，可以推导这两位的变化公式。

对于one：

```
if two == 0:
    if n==0:
        one = one
    if n==1:
        one = ~one
if two == 1:
    one = 0
#可简化为:
one = one ^ n & ~two
```

对于two：

```
two = two ^ n & ~ one
```

由于每个二进制位最后的状态必然是 00 或者 01，因此返回one即可。

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        one = 0
        two = 0
        for n in nums:
            one = one ^ n & ~ two
            two = two ^ n & ~ one
        return one
```

201. 数字范围按位与

题目描述

给你两个整数 `left` 和 `right`，表示区间 `[left, right]`，返回此区间内所有数字 **按位与** 的结果（包含 `left`、`right` 端点）。

示例 1：

```
输入: left = 5, right = 7
输出: 4
```

示例 2:

输入: left = 0, right = 0
输出: 0

示例 3:

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

题解

对所有数字按位与的最终答案是公开前缀再用0补上剩余位。本题可以不断右移left和right，当他们相等时，就可以得到公共前缀，此时记录下右移位数，然后再将他们往左移相应位数即可。

```
class Solution:
    def rangeBitwiseAnd(self, left: int, right: int) -> int:
        shift = 0
        while left < right:
            left >>= 1
            right >>= 1
            shift += 1
        return left << shift
```

第三部分 数据结构

十一、数组/字符串

11.1 数据结构简介

本章对应的题目为“数组/字符串”。

(1) 基本定义

数组：数组是一种线性数据结构，它由相同类型的元素组成，这些元素在内存中连续存储。每个元素都可以通过一个索引来访问，索引通常从 0 开始。在python中常用列表（list）表示。

字符串：字符串是由字符组成的序列，它可以看作是一种特殊的数组，其中的元素是字符。字符串通常用双引号或单引号括起来表示

(2) 应用场景

数组：用于存储向量、矩阵等数据结构，便于数据存储、以及各种数学运算。

字符串：用于文本处理和用户输入和输出。

(3) 常用代码 (python)

数组

```
# arr = [0,1,2,3,4]
```

```

# 1. 获取一维数组长度
n = len(arr)
# 2. 获取二维数组尺寸
m = len(arr) # 行数
n = len(arr[0]) # 列数
# 3. 创建一维数组（列表）
arr = [0] * n
# 4. 创建二维数组
arr = [[0] * n for _ in range(m)]
# 5. 数组切片
sliced_arr = arr[1:4] # 获取索引1到3的元素，（:左边能取到，右边取不到）
# 6. 数组从小到大排序
arr.sort()
# 7. 数组添加元素
arr.append(20) #
# 8. 数组删除元素
del arr[0]

```

字符串

```

# str1 = "hello"
# str2 = "world"

# 1. 获取字符串长度
n = len(str1)
# 2. 字符串拼接
combined_str = str1 + " " + str2
# 3. 字符串访问
char = combined_str[4]
# 4. 字符串切片
sliced_str = combined_str[1:4]
# 5. 查找子串
index = combined_str.find("world") # 返回第一个字母的索引
# 6. 替换子串
new_str = combined_str.replace("world", "python")
# 7. 字符串分割
words = combined_str.split(" ")
# 8. 大小写转换
upper_str = combined_str.upper() # 转大写
lower_str = combined_str.lower() # 转小写
# 9. 去除首尾空格
str_with_space = "  Hello world  "
trimmed_str = str_with_space.strip()

```

11.2 题目（数组/字符串）

88. 合并两个有序数组

给你两个按 **非递减顺序** 排列的整数数组 `nums1` 和 `nums2`，另有两个整数 `m` 和 `n`，分别表示 `nums1` 和 `nums2` 中的元素数目。

请你 **合并** `nums2` 到 `nums1` 中，使合并后的数组同样按 **非递减顺序** 排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 `nums1` 中。为了应对这种情况，`nums1` 的初始长度为 `m + n`，其中前 `m` 个元素表示应合并的元素，后 `n` 个元素为 `0`，应忽略。`nums2` 的长度为 `n`。

示例 1：

输入: `nums1 = [1,2,3,0,0,0]`, `m = 3`, `nums2 = [2,5,6]`, `n = 3`
输出: `[1,2,2,3,5,6]`
解释: 需要合并 `[1,2,3]` 和 `[2,5,6]`。
合并结果是 `[1,2,2,3,5,6]`，其中斜体加粗标注的为 `nums1` 中的元素。

示例 2：

输入: `nums1 = [1]`, `m = 1`, `nums2 = []`, `n = 0`
输出: `[1]`
解释: 需要合并 `[1]` 和 `[]`。
合并结果是 `[1]`。

示例 3：

输入: `nums1 = [0]`, `m = 0`, `nums2 = [1]`, `n = 1`
输出: `[1]`
解释: 需要合并的数组是 `[]` 和 `[1]`。
合并结果是 `[1]`。
注意，因为 `m = 0`，所以 `nums1` 中没有元素。`nums1` 中仅存的 `0` 仅仅是为了确保合并结果可以顺利存放到 `nums1` 中。

题解

本题需要将`nums2`合并到`nums1`中，可以采用双指针的方式，利用`p1`和`p2`两个指针遍历两个数组。由于从小到大遍历数组并将更小的元素存储到`nums1`中时，会覆盖原来的元素。因此，改变思路，采用逆向双指针的方式，从大到小遍历两个数组，并从数组末尾开始，不断向左填充更大的元素。

为什么从大到小（从右到左）遍历不会产生覆盖的情况呢？考虑到`nums1`的长度为`m+n`，可以想象两种极端情况。

1. `nums2`所有元素都大于`nums1`，则`nums2`的元素直接放置在长度为`n`的位置，不影响`num1`的元素
2. `nums2`所有元素都小于`nums1`，则每将`nums1`的右侧元素放于数组尾部时，可以认为`nums1`少了一个元素，因此不会产生重叠覆盖的情况。

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
```

```

p1 = m-1
p2 = n-1
tail = m + n -1
while p1>=0 or p2>=0:
    if p1<0:
        nums1[0:tail+1] = nums2[0:p2+1]
        p2 = -1
    elif p2<0:
        nums1[0:tail+1] = nums1[0:p1+1]
        p1 = -1
    elif nums1[p1]<nums2[p2]:
        nums1[tail] = nums2[p2]
        p2 -= 1
    else:
        nums1[tail] = nums1[p1]
        p1 -= 1
    tail -= 1

```

27. 移除元素

题目描述

给你一个数组 `nums` 和一个值 `val`，你需要 **原地** 移除所有数值等于 `val` 的元素。元素的顺序可能发生改变。然后返回 `nums` 中与 `val` 不同的元素的数量。

假设 `nums` 中不等于 `val` 的元素数量为 `k`，要通过此题，您需要执行以下操作：

- 更改 `nums` 数组，使 `nums` 的前 `k` 个元素包含不等于 `val` 的元素。`nums` 的其余元素和 `nums` 的大小并不重要。
- 返回 `k`。

用户评测：

评测机将使用以下代码测试您的解决方案：

```

int[] nums = [...]; // 输入数组
int val = ...; // 要移除的值
int[] expectedNums = [...]; // 长度正确的预期答案。
                               // 它以不等于 val 的值排序。

int k = removeElement(nums, val); // 调用你的实现

assert k == expectedNums.length;
sort(nums, 0, k); // 排序 nums 的前 k 个元素
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}

```

如果所有的断言都通过，你的解决方案将会 **通过**。

示例 1：

输入: `nums = [3,2,2,3]`, `val = 3`
输出: `2`, `nums = [2,2,-,-]`
解释: 你的函数应该返回 `k = 2`, 并且 `nums` 中的前两个元素均为 `2`。
你在返回的 `k` 个元素之外留下了什么并不重要 (因此它们并不计入评测)。

示例 2:

输入: `nums = [0,1,2,2,3,0,4,2]`, `val = 2`
输出: `5`, `nums = [0,1,4,0,3,-,-,-]`
解释: 你的函数应该返回 `k = 5`, 并且 `nums` 中的前五个元素为 `0,0,1,3,4`。
注意这五个元素可以任意顺序返回。
你在返回的 `k` 个元素之外留下了什么并不重要 (因此它们并不计入评测)。

题解

本题采用快慢指针的方式, 若不等于`val`的元素数量为`k`, 利用快指针`fast`遍历数组查找值不等于`val`的元素,

1. 当找到之后, 将其复制到慢指针`slow`处, 并同时移动快慢指针继续遍历。
2. 否则利用快指针继续遍历。

```
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        slow = 0
        for fast in range(len(nums)):
            if nums[fast] != val:
                nums[slow] = nums[fast]
                slow += 1
        return slow
```

26. 删除有序数组中的重复项

题目描述

给你一个 **非严格递增排列** 的数组 `nums`, 请你 **原地** 删除重复出现的元素, 使每个元素 **只出现一次**, 返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致**。然后返回 `nums` 中唯一元素的个数。

考虑 `nums` 的唯一元素的数量为 `k`, 你需要做以下事情确保你的题解可以被通过:

- 更改数组 `nums`, 使 `nums` 的前 `k` 个元素包含唯一元素, 并按照它们最初在 `nums` 中出现的顺序排列。 `nums` 的其余元素与 `nums` 的大小不重要。
- 返回 `k`。

判题标准:

系统会用下面的代码来测试你的题解:


```
int[] nums = [...]; // 输入数组
int[] expectedNums = [...]; // 长度正确的期望答案

int k = removeDuplicates(nums); // 调用

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

如果所有断言都通过，那么您的题解将被 **通过**。

示例 1:

输入: `nums = [1,1,2]`
 输出: `2, nums = [1,2,_]`
 解释: 函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。不需要考虑数组中超出新长度后面的元素。

示例 2:

输入: `nums = [0,0,1,1,1,2,2,3,3,4]`
 输出: `5, nums = [0,1,2,3,4]`
 解释: 函数应该返回新的长度 `5`，并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。不需要考虑数组中超出新长度后面的元素。

题解

本题也是采用快慢指针的方法，快指针`fast`不断向右遍历，直到快慢指针对应的元素不相等时，将`fast`对应的元素复制到`slow`位置。

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        n = len(nums)
        slow = 0
        for fast in range(n):
            if nums[slow] != nums[fast]:
                slow += 1
                nums[slow] = nums[fast]
        return slow + 1
```

80. 删除有序数组中的重复项

题目描述

给你一个有序数组 `nums`，请你 **原地** 删除重复出现的元素，使得出现次数超过两次的元素 **只出现两次**，返回删除后数组的新长度。

不要使用额外的数组空间，你必须在 **原地** 修改输入数组 并在使用 $O(1)$ 额外空间的条件下完成。

说明:

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

示例 1：

输入：nums = [1,1,1,2,2,3]

输出：5, nums = [1,1,2,2,3]

解释：函数应返回新长度 length = 5，并且原数组的前五个元素被修改为 1, 1, 2, 2, 3。不需要考虑数组中超出新长度后面的元素。

示例 2：

输入：nums = [0,0,1,1,1,1,2,3,3]

输出：7, nums = [0,0,1,1,2,3,3]

解释：函数应返回新长度 length = 7，并且原数组的前七个元素被修改为 0, 0, 1, 1, 2, 3, 3。不需要考虑数组中超出新长度后面的元素。

题解

本题仍然使用快慢指针的方式，但因为一个数字可以出现两次，所以前两个元素不用变化，直接将快慢指针都初始化为2。当遍历快指针时，因为slow为止的数组中所有数字最多出现两次，所以我们只需要考虑nums[fast]和nums[slow-2]是否相同即可。

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        n = len(nums)
        if n <= 2:
            return n
        slow = 2
        for fast in range(2, n):
            if nums[slow-2] != nums[fast]:
                nums[slow] = nums[fast]
                slow += 1
        return slow
```

169. 多数元素

题目描述

给定一个大小为 n 的数组 `nums`，返回其中的多数元素。多数元素是指在数组中出现次数大于 $\lfloor n/2 \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1:

输入: `nums = [3,2,3]`
输出: 3

示例 2:

输入: `nums = [2,2,1,1,1,2,2]`
输出: 2

题解

对于一个有序数组， $\lfloor n/2 \rfloor$ 处的元素一定是多数元素，因此只需要先排序再返回 `nums[$\lfloor n/2 \rfloor$]` 即可

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        nums.sort()
        return nums[len(nums)//2]
```

189. 轮转数组

题目描述

给定一个整数数组 `nums`，将数组中的元素向右轮转 k 个位置，其中 k 是非负数。

示例 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]`

题解

本题向右轮转 k 个位置，意味着 $\text{nums}[(i+k)\%n] = \text{nums}[i]$ 。由于本题需要再原数组上进行操作，因此可以创建一个新数组，再将变换后的值赋值到原数组。

```
class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        n = len(nums)
        new_nums = [0] * n
        for i in range(n):
            new_nums[(i+k)%n] = nums[i]
        nums[:] = new_nums[:]
```

121. 买卖股票的最佳时机

题目描述

给定一个数组 `prices`，它的第 i 个元素 `prices[i]` 表示一支给定股票第 i 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0`。

示例 1:

输入: `[7,1,5,3,6,4]`

输出: `5`

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = $6 - 1 = 5$ 。

注意利润不能是 $7 - 1 = 6$ ，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2:

输入: `prices = [7,6,4,3,1]`

输出: `0`

解释: 在这种情况下，没有交易完成，所以最大利润为 0。

题解

本题本质上是一道动态规划题目，基本思路是计算在第 i 天卖出时的最大收益， i 属于 $(0,n)$ ，然后返回其最大值。这里用 $\text{price}[i] - \text{minprice}$ 来计算第 i 天卖出时的最大收益。

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        min_price = prices[0]
        max_Profit = 0
        for i in range(n):
            min_price = min(min_price, prices[i])
            max_Profit = max(max_Profit, prices[i] - min_price)
        return max_Profit
```

122. 买卖股票的最佳时机II

题目描述

给你一个整数数组 `prices`，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以先购买，然后在 **同一天** 出售。

返回 **你能获得的最大利润**。

示例 1:

输入: `prices = [7,1,5,3,6,4]`

输出: 7

解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = $6 - 3 = 3$ 。

最大总利润为 $4 + 3 = 7$ 。

示例 2:

输入: `prices = [1,2,3,4,5]`

输出: 4

解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5 - 1 = 4$ 。

最大总利润为 4。

示例 3:

输入: `prices = [7,6,4,3,1]`

输出: 0

解释: 在这种情况下，交易无法获得正利润，所以不参与交易可以获得最大利润，最大利润为 0。

题解

相比于前一题，本题只需要计算每天卖出的收益是否为正，将所有为正的收益相加即可

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        max_Profit = 0
        n = len(prices)
        for i in range(1,n):
            if prices[i] - prices[i-1]>=0:
                max_Profit += prices[i] - prices[i-1]
        return max_Profit
```

55. 跳跃游戏

题目描述

给你一个非负整数数组 `nums`，你最初位于数组的 **第一个下标**。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标，如果可以，返回 `true`；否则，返回 `false`。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: `true`

解释: 可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

示例 2:

输入: `nums = [3,2,1,0,4]`

输出: `false`

解释: 无论如何，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远不可能到达最后一个下标。

题解

本题采取贪心的思路，遍历数组并用reached记录当前能到达的最大下标，如果最大下标小于当前下标，则说明永远到不了当前下标，返回False，否则返回True。

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        n = len(nums)
        reached = 0
        for i in range(n):
            if reached < i:
                return False
            reached = max(reached,nums[i]+i)
        return True
```

45 跳跃游戏II

题目描述

给定一个长度为 n 的 **0 索引** 整数数组 `nums`。初始位置为 `nums[0]`。

每个元素 `nums[i]` 表示从索引 i 向后跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处：

- $0 \leq j \leq \text{nums}[i]$
- $i + j < n$

返回到达 `nums[n - 1]` 的最小跳跃次数。生成的测试用例可以到达 `nums[n - 1]`。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。

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

示例 2:

输入: `nums = [2,3,0,1,4]`

输出: 2

题解

本题同样采取贪心算法的思路。为了用最小跳跃次数到达 $n-1$ 处，我们希望在当前可选的跳跃位置中，下一次跳跃能到达更远的位置。例如当前在 i 位置，可选择的跳跃位置为 j ， j 的范围是 $[i+0, i+\text{nums}[i]]$ ，我们选取下一步跳跃最大位置 `nums[j]` 所对应的 j 值，作为当前要跳跃到的位置。

在写代码时，将每一步能到达的最远位置当成一个区间的右端点，用 `current_pos` 记录，将从该区间出发能到达的最远位置当成下一个区间的右端点，用 `next_max_pos` 记录。在遍历过程中，每达到一个区间的右端点，就将步数+1，并更新下一个区间的右端点。

当 $n-2 == \text{current_pos}$ 时，此时步数+1，并且下一个区间肯定能到达 $n-1$ ，当 $n-2 < \text{current_pos}$ 时，说明当前区间已经能到到 $n-1$ ，因此遍历到 $n-2$ 即可。

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        n = len(nums)
        next_max_pos, current_pos, step = 0, 0, 0
        for i in range(n-1):
            next_max_pos = max(next_max_pos, i + nums[i]) #遍历从第一个区间内所有位置出发可以到达的最远位置，并获得下一个区间的右端点
            if i == current_pos: #当遍历完第一个区间，更新下一个区间右端点
                current_pos = next_max_pos
                step += 1
        return step
```

274. H指数

题目描述

给你一个整数数组 `citations`，其中 `citations[i]` 表示研究者的第 `i` 篇论文被引用的次数。计算并返回该研究者的 **h 指数**。

根据维基百科上 [h 指数的定义](#)：**h** 代表“高引用次数”，一名科研人员的 **h 指数** 是指他（她）至少发表了 **h** 篇论文，并且 **至少** 有 **h** 篇论文被引用次数大于等于 **h**。如果 **h** 有多种可能的值，**h 指数** 是其中最大的那个。

示例 1:

输入: `citations = [3,0,6,1,5]`

输出: 3

解释: 给定数组表示研究者总共有 5 篇论文，每篇论文相应的被引用了 3, 0, 6, 1, 5 次。

由于研究者有 3 篇论文每篇 至少 被引用了 3 次，其余两篇论文每篇被引用 不多于 3 次，所以她的 h 指数是 3。

示例 2:

输入: `citations = [1,3,1]`

输出: 1

题解

本题首先对引用次数数组 `citations` 进行排序，然后再判断当前元素是否大于等于剩余长度（即引用次数大于等于当前元素的论文数量），第一个满足条件的剩余长度就是 h 指数。

```
class Solution:
    def hIndex(self, citations: List[int]) -> int:
        citations.sort()
        n = len(citations)
        for i in range(n):
            if citations[i] >= n-i:
                return n-i
        return 0
```

380. O(1)时间插入、删除和获取随机元素

题目描述

实现 `RandomizedSet` 类:

- `RandomizedSet()` 初始化 `RandomizedSet` 对象
- `bool insert(int val)` 当元素 `val` 不存在时，向集合中插入该项，并返回 `true`；否则，返回 `false`。
- `bool remove(int val)` 当元素 `val` 存在时，从集合中移除该项，并返回 `true`；否则，返回 `false`。
- `int getRandom()` 随机返回现有集合中的一项（测试用例保证调用此方法时集合中至少存在一个元素）。每个元素应该有 **相同的概率** 被返回。

你必须实现类的所有函数，并满足每个函数的 **平均** 时间复杂度为 $O(1)$ 。

示例：

输入

```
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]
```

```
[[], [1], [2], [2], [], [1], [2], []]
```

输出

```
[null, true, false, true, 2, true, false, 2]
```

解释

```
RandomizedSet randomizedSet = new RandomizedSet();
randomizedSet.insert(1); // 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomizedSet.remove(2); // 返回 false ，表示集合中不存在 2 。
randomizedSet.insert(2); // 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomizedSet.getRandom(); // getRandom 应随机返回 1 或 2 。
randomizedSet.remove(1); // 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomizedSet.insert(2); // 2 已在集合中，所以返回 false 。
randomizedSet.getRandom(); // 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
```

题解

本题利用一个数组和一个哈希表来实现 $O(1)$ 复杂度的操作

对于插入一个元素：首先判断元素是否在哈希表中，如果不在则将其添加到哈希表以及数组的末尾

对于移除一个元素：首先判断元素是否在哈希表中，如果在的话就将数组尾部的元素移到该元素位置，并pop出末尾的元素，与此同时，哈希表中也要删除相应元素

对于随机取出元素，只需要随机一个位置*i*，返回相应的数组值即可，但因为存在随机性，可能会判错。

```
class RandomizedSet:

    def __init__(self):
        self.nums = []
        self.index = {}

    def insert(self, val: int) -> bool:
        if val in self.index:
            return False
        self.index[val] = len(self.nums)
        self.nums.append(val)
        return True

    def remove(self, val: int) -> bool:
        if val not in self.index:
            return False
        idx = self.index[val]
        self.nums[idx] = self.nums[-1]
        self.nums.pop()
        del self.index[val]
        return True

    def getRandom(self) -> int:
```

```
i = random.randint(0, len(self.nums)-1)
return self.nums[i]
```

280. 除自身以外数组的乘积

给你一个整数数组 `nums`，返回 数组 `answer`，其中 `answer[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积。

题目数据 **保证** 数组 `nums` 之中任意元素的全部前缀元素和后缀的乘积都在 **32 位** 整数范围内。

请 **不要使用除法**，且在 $O(n)$ 时间复杂度内完成此题。

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]
```

题解

这是一道前后缀分解题。定义`left[i]`为从`nums[0]`到`nums[i-1]`的乘积，`right[i]`为从`nums[i+1]`到`nums[n-1]`的乘积。最后将他们相乘得到最终的答案。

```
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        left = [1] * n
        right = [1] * n
        ans = [1] * n
        for i in range(1,n):
            left[i] = nums[i-1]*left[i-1]
        for i in range(n-2,-1,-1):
            right[i] = nums[i+1]*right[i+1]
        for i in range(n):
            ans[i] = left[i]*right[i]
        return ans
```

134. 加油站

题目描述

在一条环路上有 `n` 个加油站，其中第 `i` 个加油站有汽油 `gas[i]` 升。

你有一辆油箱容量无限的汽车，从第 `i` 个加油站开往第 `i+1` 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组 `gas` 和 `cost`，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 `-1`。如果存在解，则 **保证** 它是 **唯一** 的。

示例 1:

输入: `gas = [1,2,3,4,5]`, `cost = [3,4,5,1,2]`

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发,可获得 4 升汽油。此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 4 号加油站,此时油箱有 $4 - 1 + 5 = 8$ 升汽油

开往 0 号加油站,此时油箱有 $8 - 2 + 1 = 7$ 升汽油

开往 1 号加油站,此时油箱有 $7 - 3 + 2 = 6$ 升汽油

开往 2 号加油站,此时油箱有 $6 - 4 + 3 = 5$ 升汽油

开往 3 号加油站,你需要消耗 5 升汽油,正好足够你返回到 3 号加油站。

因此,3 可为起始索引。

示例 2:

输入: `gas = [2,3,4]`, `cost = [3,4,3]`

输出: -1

解释:

你不能从 0 号或 1 号加油站出发,因为没有足够的汽油可以让你行驶到下一个加油站。

我们从 2 号加油站出发,可以获得 4 升汽油。 此时油箱有 $= 0 + 4 = 4$ 升汽油

开往 0 号加油站,此时油箱有 $4 - 3 + 2 = 3$ 升汽油

开往 1 号加油站,此时油箱有 $3 - 3 + 3 = 3$ 升汽油

你无法返回 2 号加油站,因为返程需要消耗 4 升汽油,但是你的油箱只有 3 升汽油。

因此,无论如何,你都不可能绕环路行驶一周

题解

最简单的方法是不断遍历,从每一个位置出发,判断总的油量是否大于总消耗。这里介绍灵神的思路:“已经在谷底了,怎么走都是向上。”

假设油量可以为负,我们可以从0号加油站开始,计算到达每一个加油站时的油量变化。其中油量最低时所处的加油站就是出发的加油站。

此外,只要总油量大于等于总消耗,就一定能环绕一圈

```
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        current_gas = 0
        min_gas = 0
        ans = 0
        n = len(gas)
        for i in range(n):
            current_gas += gas[i] - cost[i]
            if current_gas < min_gas:
                ans = i + 1 #油量最低时所处的加油站
                min_gas = current_gas
        if current_gas < 0:
            return -1
        else:
            return ans
```

135. 分发糖果

题目描述

n 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。

你需要按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 1 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。

请你给每个孩子分发糖果，计算并返回需要准备的 **最少糖果数目**。

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。

题解

本题虽然是困难题，但其实也只需要两次贪心算法的遍历即可。

1. 从左到右遍历，当`ratings[i]>ratings[i-1]`时,只需要让评分更高的孩子多一颗糖果即可，`nums[i] = nums[i-1]+1`
2. 从右到左遍历，当`ratings[i]<ratings[i+1]`时，需要先判断评分更高的孩子的糖果数量是否已经更高，如果没有更高就要加一颗糖果。

最后返回所有的糖果数量

```
class Solution:
    def candy(self, ratings: List[int]) -> int:
        n = len(ratings)
        nums = [1]*n
        for i in range(1,n):
            if ratings[i]>ratings[i-1]:
                nums[i] = nums[i-1]+1
        for i in range(n-1,0,-1):
            if ratings[i-1]>ratings[i]:
                nums[i-1] = max(nums[i]+1,nums[i-1])

        return sum(nums)
```

13. 罗马数字转整数

题目描述

罗马数字包含以下七种字符: **I**, **V**, **X**, **L**, **C**, **D** 和 **M**。

字符	数值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

例如，罗马数字 2 写做 **II**，即为两个并列的 1。12 写做 **XII**，即为 **X** + **II**。27 写做 **XXVII**，即为 **XX** + **V** + **II**。

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 **IIII**，而是 **IV**。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4。同样地，数字 9 表示为 **IX**。这个特殊的规则只适用于以下六种情况：

- **I** 可以放在 **V** (5) 和 **X** (10) 的左边，来表示 4 和 9。
- **X** 可以放在 **L** (50) 和 **C** (100) 的左边，来表示 40 和 90。
- **C** 可以放在 **D** (500) 和 **M** (1000) 的左边，来表示 400 和 900。

给定一个罗马数字，将其转换成整数。

示例 1:

输入: `s = "III"`
输出: 3

示例 2:

输入: `s = "IV"`
输出: 4

示例 3:

输入: `s = "IX"`
输出: 9

示例 4:

输入: `s = "LVIII"`
输出: 58
解释: `L = 50`, `V = 5`, `III = 3`。

示例 5:

输入: s = "MCMXCIV"
输出: 1994
解释: M = 1000, CM = 900, XC = 90, IV = 4.

题解

本题需要将每个罗马字符对应的值相加, 如果当前罗马字符的值小于下个罗马字符, 根据规则, 则需要将其添加一个“-”。

```
class Solution:
    def romanToInt(self, s: str) -> int:
        symbol_value = {
            "I":1,
            "V":5,
            "X":10,
            "L":50,
            "C":100,
            "D":500,
            "M":1000,
        }
        n = len(s)
        num = 0
        for i in range(n-1):
            if symbol_value[s[i]] < symbol_value[s[i+1]]:
                num -= symbol_value[s[i]]
            else:
                num += symbol_value[s[i]]
        num += symbol_value[s[-1]]
        return num
```

12. 整数转罗马数字

题目描述

七个不同的符号代表罗马数字, 其值如下:

符号	值
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

罗马数字是通过添加从最高到最低的小数位值的转换而形成的。将小数位值转换为罗马数字有以下规则:

- 如果该值不是以 4 或 9 开头，请选择可以从输入中减去的最大值的符号，将该符号附加到结果，减去其值，然后将其余部分转换为罗马数字。
- 如果该值以 4 或 9 开头，使用 **减法形式**，表示从以下符号中减去一个符号，例如 4 是 5 (V) 减 1 (I): IV，9 是 10 (X) 减 1 (I): IX。仅使用以下减法形式：4 (IV)，9 (IX)，40 (XL)，90 (XC)，400 (CD) 和 900 (CM)。
- 只有 10 的次方 (I, X, C, M) 最多可以连续附加 3 次以代表 10 的倍数。你不能多次附加 5 (V)，50 (L) 或 500 (D)。如果需要将符号附加 4 次，请使用 **减法形式**。

给定一个整数，将其转换为罗马数字。

示例 1:

输入: num = 3749

输出: "MMMDCCLXIX"

解释:

3000 = MMM 由于 1000 (M) + 1000 (M) + 1000 (M)
700 = DCC 由于 500 (D) + 100 (C) + 100 (C)
40 = XL 由于 50 (L) 减 10 (X)
9 = IX 由于 10 (X) 减 1 (I)
注意: 49 不是 50 (L) 减 1 (I) 因为转换是基于小数位

示例 2:

输入: num = 58

输出: "LVIII"

解释:

50 = L
8 = VIII

示例 3:

输入: num = 1994

输出: "MCMXCIV"

解释:

1000 = M
900 = CM
90 = XC
4 = IV

题解

本题把num拆解成个位数、十位数、百位数、以及千位数，然后构建不同位数对应的字符表，用字符串将不同位数对应的罗马字符连接起来，就是最终答案。

```
class Solution:
    def intToRoman(self, num: int) -> str:
        thousands = ["", "M", "MM", "MMM"]
        hundreds = ["", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"]
        tens = ["", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"]
        ones = ["", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"]

        return thousands[num//1000] + hundreds[num%1000//100] + tens[num%100//10]
        + ones[num%10]
```

58. 最后一个单词的长度

题目描述

给你一个字符串 `s`，由若干单词组成，单词前后用一些空格字符隔开。返回字符串中 **最后一个** 单词的长度。

单词 是指仅由字母组成、不包含任何空格字符的最大子字符串。

示例 1:

输入: `s = "Hello world"`
 输出: 5
 解释: 最后一个单词是“world”，长度为 5。

示例 2:

输入: `s = " fly me to the moon "`
 输出: 4
 解释: 最后一个单词是“moon”，长度为 4。

示例 3:

输入: `s = "luffy is still joyboy"`
 输出: 6
 解释: 最后一个单词是长度为 6 的“joyboy”。

题解

对于经常使用python的同学，直接分割字符串吧。不然就是暴力搜索。

```
class Solution:
    def lengthOfLastWord(self, s: str) -> int:
        return len(s.strip().split(" ")[-1])
```


14. 最长公共前缀

题目描述

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

示例 1:

输入: strs = ["flower","flow","flight"]
输出: "fl"

示例 2:

输入: strs = ["dog","racecar","car"]
输出: ""
解释: 输入不存在公共前缀。

题解

本题首先需要构建一个计算两个字符串最长公共前缀的函数，这里直接遍历即可；其次，遍历字符串数组，每一次都用当前的最长公共前缀与下一个字符串计算新的最长公共前缀。

```
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        def lcp(str1, str2):
            n = min(len(str1), len(str2))
            index = 0
            while index < n:
                if str1[index] == str2[index]:
                    index += 1
                else:
                    break
            return str1[:index]

        ans = strs[0]
        for i in range(1, len(strs)):
            if not ans:
                break
            else:
                ans = lcp(ans, strs[i])
        return ans
```

151. 反转字符串中的单词

题目描述

给你一个字符串 `s`，请你反转字符串中 **单词** 的顺序。

单词 是由非空格字符组成的字符串。`s` 中使用至少一个空格将字符串中的 **单词** 分隔开。

返回 **单词** 顺序颠倒且 **单词** 之间用单个空格连接的结果字符串。

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

示例 1：

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

示例 2：

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 3：

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。

题解

首先将字符串进行以“ ”为标志符进行分割，其次将其倒序输出

```
class Solution:
    def reversewords(self, s: str) -> str:
        s = s.strip().split()
        reversed_s = ""
        for i in range(len(s)-1,-1,-1):
            reversed_s += s[i] + " "
        return reversed_s.strip()
```

6. Z 字形变换

题目描述

将一个给定字符串 `s` 根据给定的行数 `numRows`，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 `"PAYPALISHIRING"` 行数为 `3` 时，排列如下：

```
P   A   H   N
A P L S I I G
Y   I   R
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如: `"PAHNAPLSIIGYIR"`。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1：

输入: `s = "PAYPALISHIRING", numRows = 3`
输出: `"PAHNAPLSIIGYIR"`

示例 2:

输入: `s = "PAYPALISHIRING", numRows = 4`
输出: `"PINALSIGYAHRPI"`
解释:
P I N
A L S I G
Y A H R
P I

示例 3:

输入: `s = "A", numRows = 1`
输出: `"A"`

题解

本题可以构建多个数组来管理字符串, 一个数组代表一行。按顺序遍历字符串, 将每个字符添加到数组中。具体添加方式如下:

1. 计算周期: 当我们填写字符时, 先向下填写`numRows`个字符, 然后向右上填写`numRows-2`个字符, 因此一个周期`t`是`numRows*2-2`。
2. 添加字符: 我们使用`x`来表示当前字符需要添加到第`x`行的数组中,
 1. 当`i % t < numRows-1`时, `x += 1`,
 2. 当`i % t > numRows`时, `x -= 1`
 3. 不断循环, 直到`i`遍历结束。

```
class Solution:
    def convert(self, s: str, numRows: int) -> str:
        n = len(s)
        if numRows == 1 or numRows == len(s):
            return s
        mat = [[] for _ in range(numRows)]
        t = numRows*2 - 2
        x = 0
        for i, ch in enumerate(s):
            mat[x].append(ch)
            if i % t < numRows-1:
                x += 1
            else:
                x -= 1
        new_s = ""
        for i in range(numRows):
            new_s += "".join(mat[i])
        return new_s
```

28. 找出字符串中第一个匹配项的下标

题目描述

给你两个字符串 `haystack` 和 `needle`，请在 `haystack` 字符串中找出 `needle` 字符串的第一个匹配项的下标（下标从 0 开始）。如果 `needle` 不是 `haystack` 的一部分，则返回 `-1`。

示例 1:

输入: `haystack = "sadbutsad"`, `needle = "sad"`

输出: `0`

解释: `"sad"` 在下标 `0` 和 `6` 处匹配。

第一个匹配项的下标是 `0`，所以返回 `0`。

示例 2:

输入: `haystack = "leetcode"`, `needle = "leeto"`

输出: `-1`

解释: `"leeto"` 没有在 `"leetcode"` 中出现，所以返回 `-1`。

题解

本题使用双指针算法，分别用 `l1`, `l2` 指针遍历两个字符串：

1. 当两个指针指向的元素相等时，指针同时向右移
2. 当 `needle` 字符串遍历完时，表示已找到，返回第一个匹配项的下标
3. 当两个指针指向的元素不相等时，重置 `l2` 指针，并将 `l1` 转回第一个匹配项的下一个位置。

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        l1, l2 = 0, 0
        n1, n2 = len(haystack), len(needle)
        while l1 < n1:
            if l2 < n2 and haystack[l1] == needle[l2]:
                l1 += 1
                l2 += 1
            elif l2 == n2:
                return l1 - n2
            else:
                l1 += 1 - l2
                l2 = 0
        if l2 == n2:
            return l1 - n2
        return -1
```

十二、矩阵

12.1 数据结构简介

本章对应的题目为“矩阵”。

(1) 基本定义

矩阵是由数按照一定顺序排列成的矩形阵列。一般用大写字母表示。组成矩阵的每个数称为矩阵的元素。在算法题中，一般用二维数组表示，如表示`board[i][j]`矩阵中第*i*行第*j*列的元素。

(2) 应用场景

图的邻接矩阵表示、背包问题、矩阵快速幂。

(3) 常用代码 (python)

```
# 1. 计算矩阵行数和列数
m = len(board)
n = len(board[0])
# 2. 构建相同尺寸的初始化矩阵
board = [[0]*n for _ in range(m)]
# 3. 矩阵遍历方向
directions = [(0,1),(0,-1),(1,0),(-1,0)]
```

12.2 题目（矩阵）

36. 有效的数独

题目描述

请你判断一个 `9 x 9` 的数独是否有效。只需要 **根据以下规则**，验证已经填入的数字是否有效即可。

- 数字 `1-9` 在每一行只能出现一次。
- 数字 `1-9` 在每一列只能出现一次。
- 数字 `1-9` 在每一个以粗实线分隔的 `3x3` 宫内只能出现一次。（请参考示例图）

注意：

- 一个有效的数独（部分已被填充）不一定是可解的。
- 只需要根据以上规则，验证已经填入的数字是否有效即可。
- 空白格用 `'.'` 表示。

示例 1：

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

输入: board =

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

输出: true

示例 2:

输入: board =

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

输出: false

解释: 除了第一行的第一个数字从 5 改为 8 以外, 空格内其他数字均与 示例1 相同。 但由于位于左上角的 3x3 宫内有两个 8 存在, 因此这个数独是无效的。

题解

本题初始化三个9*9的数组来记录每一行 (rows) ,每一列 (cols) 以及每一个宫 (sub_board) 内 1-9各个数字出现的次数。遍历矩阵中每一个元素, 当任意数组中显示某个元素已出现过, 就返回False。

本题主要的难点在于将[i][j]坐标转换为第b个宫。计算方式为 $b = (i//3)*3 + j//3$

```
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        rows = [[0] * 9 for _ in range(9)]
        cols = [[0] * 9 for _ in range(9)]
        sub_board = [[0] * 9 for _ in range(9)]
```

```

for i in range(9):
    for j in range(9):
        if board[i][j] != ".":
            num = int(board[i][j])-1
            b = (i//3)*3 + j//3
            if rows[i][num] or cols[j][num] or sub_board[b][num] :
                return False
            rows[i][num] = 1
            cols[j][num] = 1
            sub_board[b][num] = 1
return True

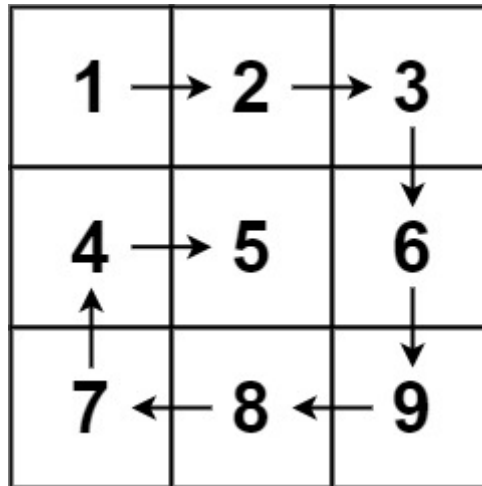
```

54. 螺旋矩阵

题目描述

给你一个 m 行 n 列的矩阵 `matrix`，请按照 **顺时针螺旋顺序**，返回矩阵中的所有元素。

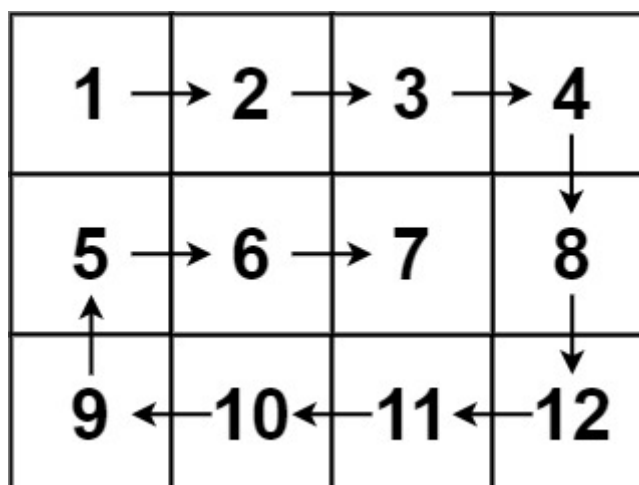
示例 1:



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

输出: `[1,2,3,6,9,8,7,4,5]`

示例 2:



输入: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]

题解

本题模拟旋转矩阵的路径，初始位置是矩阵左上角，每当下一个位置超出矩阵范围，或者已经被遍历（使用辅助矩阵），就要改变遍历方向directions。

由于每个位置都会被遍历，因此当路径长度和矩阵元素数量相同时，就遍历结束。

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        if not matrix or not matrix[0]:
            return []
        rows,cols = len(matrix),len(matrix[0])
        visted = [[0]*cols for _ in range(rows)]
        total = rows*cols
        m,n=0,0
        ans = []
        directions = [(0,1),(1,0),(0,-1),(-1,0)]
        d = 0
        for i in range(total):
            ans.append(matrix[m][n])
            visted[m][n] = 1
            next_m = m + directions[d][0]
            next_n = n + directions[d][1]
            if not (0<=next_m<rows and 0<=next_n<cols) or visted[next_m][next_n]
== 1:
                d = (d+1)%4
            m += directions[d][0]
            n += directions[d][1]

        return ans
```

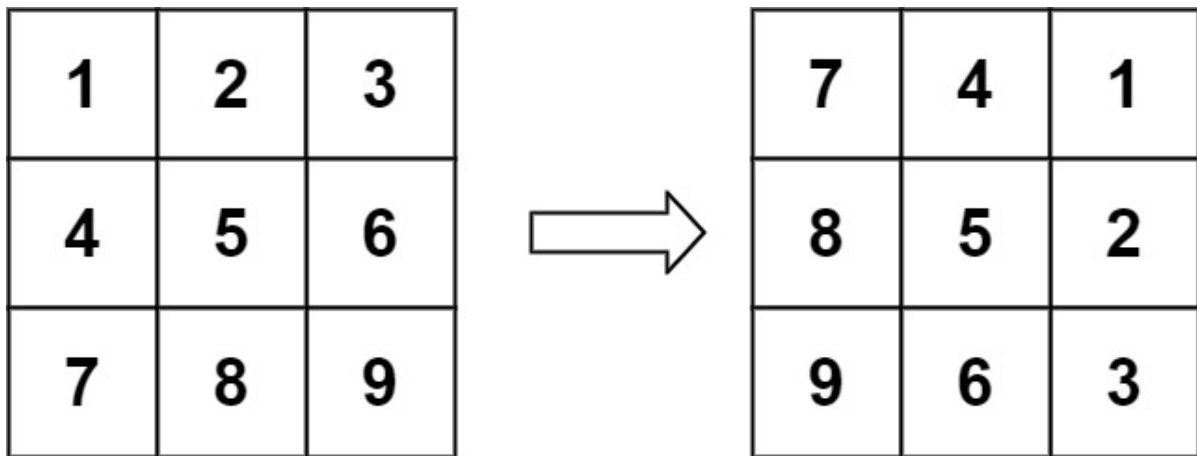
48. 旋转图像

题目描述

给定一个 $n \times n$ 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。

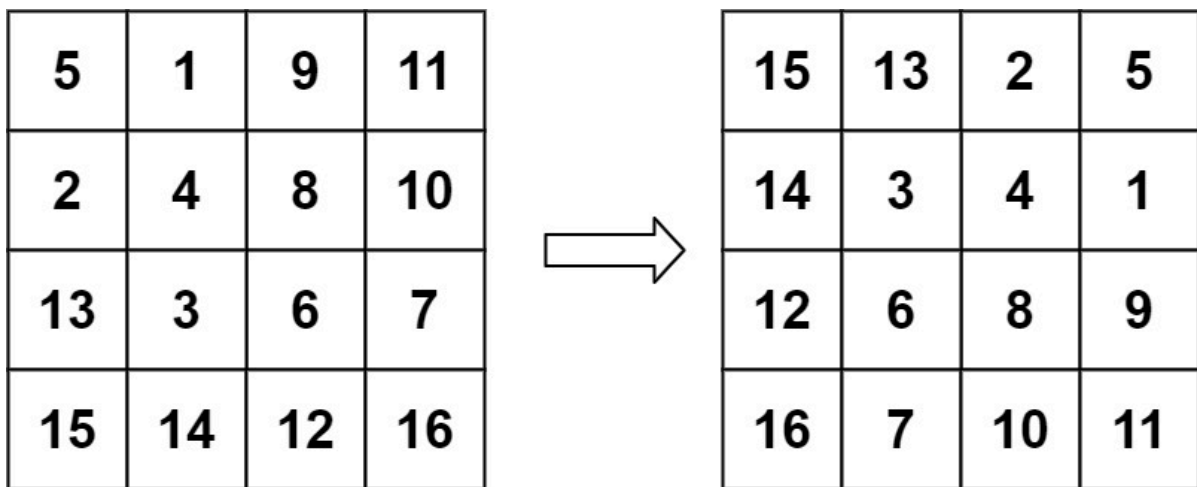
你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。**请不要** 使用另一个矩阵来旋转图像。

示例 1:



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

示例 2:



输入: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
输出: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

题解

在经过90度翻转后，矩阵元素的变化关系为： $matrix[i][j] \rightarrow matrix[j][n-i] \rightarrow matrix[n-i][n-j] \rightarrow matrix[n-j][i] \rightarrow matrix[i][j]$ 。我们可以把一开始的 $matrix[i][j]$ 保存为temp，后续不断的进行位置交换即可。这里遍历左上的区域就行。

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)
        for i in range(n//2):
            for j in range((n+1)//2):
                temp = matrix[i][j]
                matrix[i][j] = matrix[n-j-1][i]
                matrix[n-j-1][i] = matrix[n-1-i][n-1-j]
                matrix[n-1-i][n-1-j] = matrix[j][n-1-i]
                matrix[j][n-1-i] = temp
```


73. 矩阵置零

题目描述

给定一个 $m \times n$ 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用原地算法。

示例 1:

1	1	1
1	0	1
1	1	1



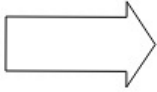
1	0	1
0	0	0
1	0	1

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

输出: [[1,0,1],[0,0,0],[1,0,1]]

示例 2:

0	1	2	0
3	4	5	2
1	3	1	5



0	0	0	0
0	4	5	0
0	3	1	0

输入: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]

输出: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

题解

本题使用第一行和第一列两个矩阵来记录矩阵中的元素是否有0出现，（行和列都从1开始遍历）。但是这样会导致第一行和第一列是否有0的信息被覆盖。因此，首先使用两个标记参数来记录第一行和第一列是否有0，如果有，那相应的第一行和第一列上的元素都是0，如果没有，则不用进一步修改。

在记录完毕后，再次遍历矩阵，根据记录来修改元素。

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        m = len(matrix)
```

```

n = len(matrix[0])
flag_row0 = any(matrix[0][j]==0 for j in range(n))
flag_col0 = any(matrix[i][0]==0 for i in range(m))

for i in range(1,m):
    for j in range(1,n):
        if matrix[i][j] == 0:
            matrix[i][0] = 0
            matrix[0][j] = 0
for i in range(1,m):
    for j in range(1,n):
        if matrix[i][0] == 0 or matrix[0][j] == 0:
            matrix[i][j] = 0

if flag_row0:
    for j in range(n):
        matrix[0][j] = 0
if flag_col0:
    for i in range(m):
        matrix[i][0] = 0

```

289. 生命游戏

题目描述

根据 [百度百科](#)，**生命游戏**，简称为**生命**，是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机。

给定一个包含 $m \times n$ 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞都具有一个初始状态：**1** 即为 **活细胞**（live），或 **0** 即为 **死细胞**（dead）。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

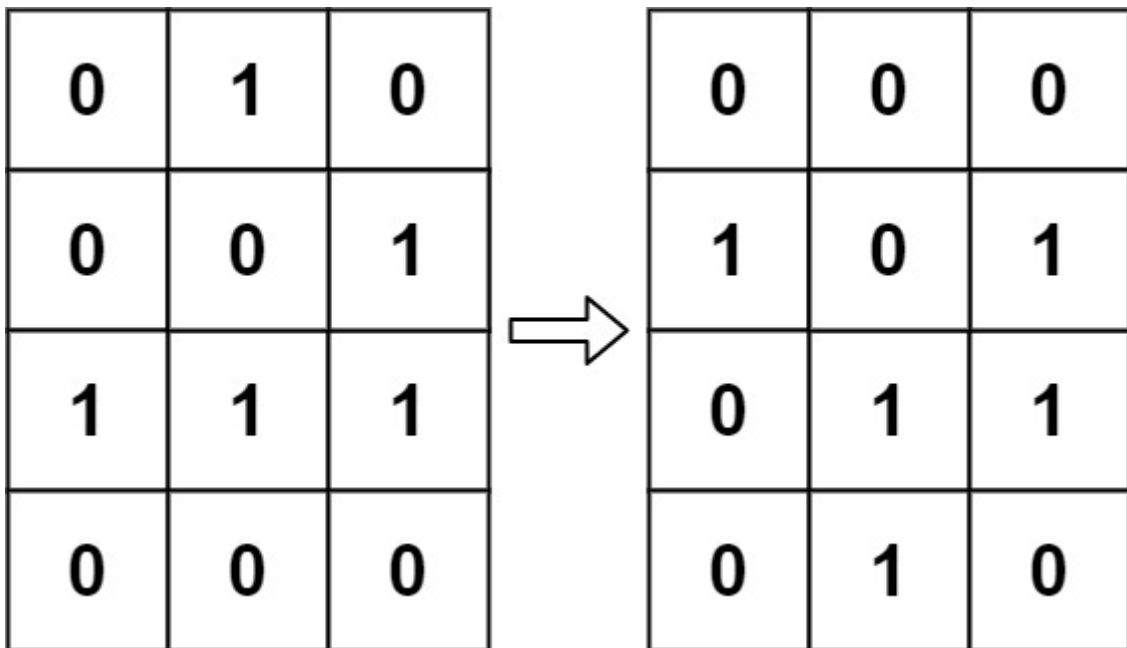
1. 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；
2. 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；
3. 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；
4. 如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

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。给你 $m \times n$ 网格面板 `board` 的当前状态，返回下一个状态。

给定当前 `board` 的状态，**更新** `board` 到下一个状态。

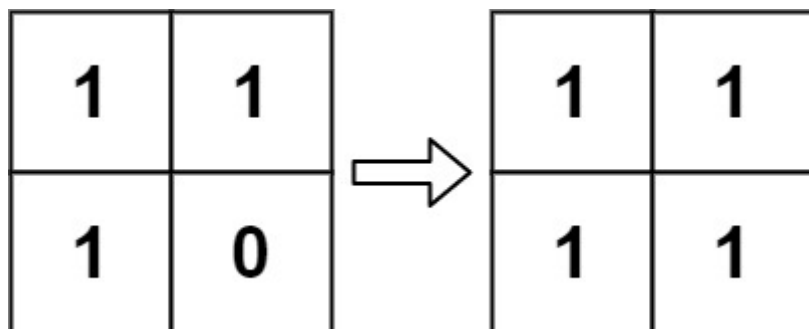
注意 你不需要返回任何东西。

示例 1：



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:



输入: board = [[1,1],[1,0]]
 输出: [[1,1],[1,1]]

题解

由于细胞的出生和死亡是 **同时** 发生的，因此直观的思路是复制一个矩阵，避免先前元素的变化对其他元素产生影响。但为了降低空间复杂度，可以通过添加额外的状态来记录元素的变化。

当细胞从死->活时，状态设为-1

当细胞从死->死时，状态设为0

当细胞从活->活时，状态设为1

当细胞从活->死时，状态设为2

当遍历完所有元素之后，将-1改0,2改为1即可。

```
class Solution:
    def gameOfLife(self, board: List[List[int]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        m = len(board)
```

```

n = len(board[0])
directions = [(1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1), (0,1),
(1,1)]
for i in range(m):
    for j in range(n):
        live = 0
        for direction in directions:
            new_i = i + direction[0]
            new_j = j + direction[1]
            if 0<=new_i<m and 0<=new_j<n and board[new_i][new_j] >= 1:
                live += 1
        if board[i][j] ==1:
            if live < 2 or live>3:
                board[i][j] = 2
        else:
            if live == 3:
                board[i][j] = -1
for i in range(m):
    for j in range(n):
        if board[i][j] == 2:
            board[i][j] = 0
        elif board[i][j] == -1:
            board[i][j] = 1

```

十三、哈希表

13.1 数据结构简介

本章对应的题目为“哈希表”。

(1) 基本定义

哈希表 (Hash Table) 是根据关键码值 (key value) 而直接进行访问的数据结构。他通过一个数据结构将键值映射到一个固定大小的数组中的特定位置, 这个位置被称为哈希桶 (Hash Bucket) 。

在python中, 常用字典dict()以及j集合set()来实现哈希表。其中集合set只存储键, 不存储值。

(2) 应用场景

数据库索引、缓存系统、去除列表的重复元素。

(3) 常用代码 (python)

****字典****

```

# hash_table = {'apple': 1, 'banana': 2, 'cherry': 3}

# 1. 创建空字典 (哈希表)
hash_table = {}
hash_table = dict()
# 2. 插入新的键值对 或更新已存在键的值
hash_table['banana'] = 2

```

```

# 3. 通过键查找值
value = hash_table.get('banana')
# 4. 检查键是否存在
if 'apple' in hash_table:
# 5. 使用 del 关键字删除键值对
del hash_table['banana']
# 6. 使用 pop() 方法删除键值对，并返回被删除的值
value = hash_table.pop('apple')
# 7. 遍历键
for key in hash_table.keys():
# 8. 遍历值
for value in hash_table.values():
# 9. 遍历键值对
for key, value in hash_table.items():
# 10. 获取字典长度
n = len(hash_table)

```

集合

```

# 1. 创建空集合
empty_set = set()
# 2. 创建包含初始元素的集合
fruits = {'apple', 'banana', 'cherry'}
# 3. 插入单个元素
fruits.add('cherry')
# 4. 插入多个元素（使用 update 方法）
new_fruits = {'date', 'elderberry'}
fruits.update(new_fruits)
# 5. 检查元素是否存在
if 'apple' in fruits:
# 6. 删除元素
fruits.remove('banana') # 如果元素不存在会抛出 KeyError
fruits.discard('date') # 如果元素不存在不会抛出异常
popped_fruit = fruits.pop() # 随机删除一个元素

# 集合运算
# set1 = {'apple', 'banana', 'cherry'}
# set2 = {'cherry', 'date', 'elderberry'}
union_set = set1.union(set2) # 并集
intersection_set = set1.intersection(set2) # 交集
difference_set = set1.difference(set2) # 差集

```

13.2 题目（哈希表）

383. 赎金信

题目描述

给你两个字符串：ransomNote 和 magazine，判断 ransomNote 能不能由 magazine 里面的字符构成。

如果可以，返回 true；否则返回 false。

magazine 中的每个字符只能在 ransomNote 中使用一次。

示例 1:

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

示例 2:

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

示例 3:

```
输入: ransomNote = "aa", magazine = "aab"  
输出: true
```

题解

本题将magazine转化为键为字母，值为出现次数的字典，然后遍历ransomNote中出现的字母，判断在其出现次数是否大于字典中的出现次数。

```
class Solution:  
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:  
        cnt = collections.Counter(magazine)  
        for c in ransomNote:  
            cnt[c] -= 1  
            if cnt[c] < 0:  
                return False  
        return True
```

205. 同构字符串

题目描述

给定两个字符串 `s` 和 `t`，判断它们是否是同构的。

如果 `s` 中的字符可以按某种映射关系替换得到 `t`，那么这两个字符串是同构的。

每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，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。

示例 1:

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

示例 2:

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

示例 3:

输入: s = "paper", t = "title"
输出: true

题解

本题要求两个字符串的字符——对应，因此需要构建两个哈希表来维护映射关系。随后，遍历字符串元素，判断映射关系是否出现矛盾。

```
class Solution:
    def isIsomorphic(self, s: str, t: str) -> bool:

        s2t = {}
        t2s = {}
        n = len(s)
        for i in range(n):
            if s[i] in s2t and s2t[s[i]] != t[i]:
                return False
            if t[i] in t2s and t2s[t[i]] != s[i]:
                return False
            s2t[s[i]] = t[i]
            t2s[t[i]] = s[i]
        return True
```

290. 单词规律

题目描述

给定一种规律 `pattern` 和一个字符串 `s`，判断 `s` 是否遵循相同的规律。

这里的 **遵循** 指完全匹配，例如，`pattern` 里的每个字母和字符串 `s` 中的每个非空单词之间存在着双向连接的对应规律。

示例1:

输入: pattern = "abba", s = "dog cat cat dog"
输出: true

示例 2:

输入: pattern = "abba", s = "dog cat cat fish"
输出: false

示例 3:

输入: pattern = "aaaa", s = "dog cat cat dog"
输出: false

题解

本题的思路和205同构字符串是一样的，只不过字符和字符的映射关系转为字符和单词的映射关系。

```
class Solution:
    def wordPattern(self, pattern: str, s: str) -> bool:
        s = s.strip().split()
        if len(s) != len(pattern):
            return False
        p2s = {}
        s2p = {}
        n = len(pattern)
        for i in range(n):
            if pattern[i] in p2s and p2s[pattern[i]] != s[i]:
                return False
            if s[i] in s2p and s2p[s[i]] != pattern[i]:
                return False
            p2s[pattern[i]] = s[i]
            s2p[s[i]] = pattern[i]

        return True
```

242. 有效的字母异位词

题目描述

给定两个字符串 `s` 和 `t`，编写一个函数来判断 `t` 是否是 `s` 的字母异位词。

示例 1:

输入: `s = "anagram", t = "nagaram"`
输出: `true`

示例 2:

输入: `s = "rat", t = "car"`
输出: `false`

题解

本题也是判断两个字符串中，各个字母出现的次数是否相同，可以直接使用`collections.Counter`函数进行判断。

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False
        if not collections.Counter(s) == collections.Counter(t):
            return False
        else:
            return True
```

49. 字母异位词分组

题目描述

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

字母异位词 是由重新排列源单词的所有字母得到的一个新单词。

示例 1:

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

示例 2:

```
输入: strs = [""]
输出: [[""]]
```

示例 3:

```
输入: strs = ["a"]
输出: [["a"]]
```

题解

当我们把字母异位词重新排序后，可以得到相同的单词。因此可以把排序后的单词当作key，异位词组合当作值，构建哈希表。

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        mp = collections.defaultdict(list)
        for s in strs:
            mp[''.join(sorted(s))].append(s)
        return list(mp.values())
```

1. 两数之和

题目描述

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

你可以假设每种输入只会对应一个答案，并且你不能使用两次相同的元素。

你可以按任意顺序返回答案。

示例 1:

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

示例 2:

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

示例 3:

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

题解

本题构建一个哈希表，用于记录已经出现的数字，如果target-num已经出现，就返回num和target-num两个数字对应的位置即可。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        hash_table = {}
        for i, num in enumerate(nums):
            if target-num in hash_table:
                return [i, hash_table[target-num]]
            hash_table[num] = i
        return []
```

202. 快乐数

题目描述

编写一个算法来判断一个数 `n` 是不是快乐数。

「快乐数」定义为：

- 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。
- 如果这个过程 **结果为 1**，那么这个数就是快乐数。

如果 `n` 是 快乐数 就返回 `true`；不是，则返回 `false`。

示例 1:

输入: n = 19
输出: true
解释:
 $1^2 + 9^2 = 82$
 $8^2 + 2^2 = 68$
 $6^2 + 8^2 = 100$
 $1^2 + 0^2 + 0^2 = 1$

示例 2:

输入: n = 2
输出: false

题解

如果一个数不是快乐数，那么他在不断替换的过程中会进入循环，因此，只需要对一个数进行不断的替换，并用哈希表记录他，如果出现重复数字，他就不是快乐数，否则终究会变成1。

```
class Solution:
    def isHappy(self, n: int) -> bool:
        def get_next(n):
            next_n = 0
            while(n!=0):
                next_n += (n%10)**2
                n = n//10
            return next_n
        hash_table = set()
        while(n!=1):
            n = get_next(n)
            if n in hash_table:
                return False
            else:
                hash_table.add(n)
        return True
```

219. 存在重复元素II

题目描述

给你一个整数数组 `nums` 和一个整数 `k`，判断数组中是否存在两个 **不同的索引** `i` 和 `j`，满足 `nums[i] == nums[j]` 且 `abs(i - j) <= k`。如果存在，返回 `true`；否则，返回 `false`。

示例 1:

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

示例 2:

输入: `nums = [1,0,1,1]`, `k = 1`
输出: `true`

示例 3:

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

题解

本题使用一个哈希表记录各个数字出现的最大位置，当新出现的相同数字和之前最大位置之间的范围小于等于`k`，则返回`True`，否则跟新最大出现位置。

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        loc = {}
        for i,num in enumerate(nums):
            if num in loc and i - loc[num] <= k:
                return True
            else:
                loc[num] = i
        return False
```

128、最长连续序列

题目描述

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

请你设计并实现时间复杂度为 $O(n)$ 的算法解决此问题。

示例 1:

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

示例 2:

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

题解

本题遍历每一个数字 x ，来枚举 $x+1$ 是否存在，如果存在，则最大长度加一。这里可以通过构建哈希表来实现。此外，如果 $x-1$ 存在于哈希表中，则说明 x 已经被枚举过，因此不需要再枚举。

```
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        length = 0
        num_set = set(nums)
        for num in num_set:
            if num - 1 not in num_set:
                current_num = num
                current_length = 1
                while current_num + 1 in num_set:
                    current_num += 1
                    current_length += 1
                length = max(length, current_length)
        return length
```

十四、区间

14.1 数据结构简介

本章对应的题目为“区间”。

(1) 基本定义

区间是由两个端点界定的一段范围，这两个端点可以是实数、整数或其他具有顺序关系的元素。一般用列表或元组表示： $[a,b]$, (a,b) 。

(2) 应用场景

任务调度，数据统计与分析、几何问题

(3) 常用代码 (python)

```
# 1. 区间合并
def merge_intervals(interval1, interval2):
    if interval1[1] < interval2[0] or interval2[1] < interval1[0]:
        return None
    new_start = min(interval1[0], interval2[0])
    new_end = max(interval1[1], interval2[1])
    return [new_start, new_end]

# 2. 区间交集
def intersection(interval1, interval2):
    start = max(interval1[0], interval2[0])
    end = min(interval1[1], interval2[1])
    if start <= end:
        return [start, end]
    return None

# 3. 区间列表合并
def merge_interval_list(intervals):
    if not intervals:
        return []
    intervals.sort(key=lambda x: x[0])
    result = [intervals[0]]
    for interval in intervals[1:]:
        if interval[0] <= result[-1][1]:
            result[-1][1] = max(result[-1][1], interval[1])
        else:
            result.append(interval)
    return result
```

14.2 题目（区间）

228. 汇总区间

题目描述

给定一个 **无重复元素** 的 **有序** 整数数组 `nums` 。

返回 **恰好覆盖数组中所有数字** 的 **最小有序** 区间范围列表。也就是说，`nums` 的每个元素都恰好被某个区间范围所覆盖，并且不存在属于某个范围但不属于 `nums` 的数字 `x` 。

列表中的每个区间范围 `[a,b]` 应该按如下格式输出：

- `"a->b"` , 如果 `a != b`
- `"a"` , 如果 `a == b`

示例 1:

```
输入: nums = [0,1,2,4,5,7]
输出: ["0->2","4->5","7"]
解释: 区间范围是:
[0,2] --> "0->2"
[4,5] --> "4->5"
[7,7] --> "7"
```

示例 2:

```
输入: nums = [0,2,3,4,6,8,9]
输出: ["0","2->4","6","8->9"]
解释: 区间范围是:
[0,0] --> "0"
[2,4] --> "2->4"
[6,6] --> "6"
[8,9] --> "8->9"
```

题解

本题向右便利，用`start`和`end`记录区间的开始和终止位置，当下一个元素和当前元素差值为1时，则继续扩张区间，否则就将该区间添加到`ans`中，继续遍历下一区间。

```
class Solution:
    def summaryRanges(self, nums: List[int]) -> List[str]:
        n = len(nums)
        ans = []
        i=0
        while i<n:
            start = i
            i+=1
            while i<n and nums[i] - nums[i-1] == 1:
                i +=1
            end = i -1
            if start < end:
                temp = str(nums[start]) + "->" + str(nums[end])
            else:
                temp = str(nums[start])

            ans.append(temp)
        return ans
```

56. 合并区间

题目描述

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`。请你合并所有重叠的区间，并返回 一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1:

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

示例 2:

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

题解

本题首先将当前区间按左端点进行排序，然后不断遍历区间。如果新区间的左端点小于当前合并区间的右端点，则将他们合并，否则添加新的合并区间。

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda x:x[0])
        ans = []
        while interval in intervals:
            if interval[0] <= ans[-1][1] and ans:
                ans[-1][1] = max(ans[-1][1], interval[1])
            else:
                ans.append(interval)
        return ans
```

57. 插入区间

题目描述

给你一个 **无重叠的**，按照区间起始端点排序的区间列表 `intervals`，其中 `intervals[i] = [starti, endi]` 表示第 `i` 个区间的开始和结束，并且 `intervals` 按照 `starti` 升序排列。同样给定一个区间 `newInterval = [start, end]` 表示另一个区间的开始和结束。

在 `intervals` 中插入区间 `newInterval`，使得 `intervals` 依然按照 `starti` 升序排列，且区间之间不重叠（如果有必要的话，可以合并区间）。

返回插入之后的 `intervals`。

注意 你不需要原地修改 `intervals`。你可以创建一个新数组然后返回它。

示例 1:

输入: intervals = [[1,3],[6,9]], newInterval = [2,5]
输出: [[1,5],[6,9]]

示例 2:

输入: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
输出: [[1,2],[3,10],[12,16]]
解释: 这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。

题解

本题不断遍历区间列表，并比较当前区间和新区间起始端点的大小，分为三种情况：

1. 如果当前区间的右端点小于新区间的左端点，则直接添加当前区间
2. 如果当前区间的左端点大于新区间的右端点，就看新区间是否已经放进去了，如果没有放进去，就先放新区间，然后把当前区间放进去
3. 其他情况表示当前区间和新区间有交集，可以把这两个区间进行合并，合并区间作为新区间。

最后如果一直没有放进去，则最后需要将新区间放进去

```
class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        left, right = newInterval
        ans = []
        placed = False
        for li, ri in intervals:
            if ri < left:
                ans.append([li, ri])
            elif li > right:
                if not placed:
                    ans.append([left, right])
                    placed = True
                ans.append([li, ri])
            else:
                left = min(li, left)
                right = max(ri, right)
        if not placed:
            ans.append([left, right])
        return ans
```

452. 用最少数量的箭引爆气球

题目描述

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 `points`，其中 `points[i] = [xstart, xend]` 表示水平直径在 `xstart` 和 `xend` 之间的气球。你不知道气球的确切 y 坐标。

一支弓箭可以沿着 x 轴从不同点 **完全垂直** 地射出。在坐标 `x` 处射出一支箭，若有一个气球的直径的开始和结束坐标为 `xstart`, `xend`，且满足 `xstart ≤ x ≤ xend`，则该气球会被 **引爆**。可以射出的弓箭的数量 **没有限制**。弓箭一旦被射出之后，可以无限地前进。

给你一个数组 `points`，返回引爆所有气球所必须射出的 **最小** 弓箭数。

示例 1:

输入: `points = [[10,16],[2,8],[1,6],[7,12]]`
输出: 2
解释: 气球可以用2支箭来爆破:
- 在 $x = 6$ 处射出箭，击破气球 `[2,8]` 和 `[1,6]`。
- 在 $x = 11$ 处发射箭，击破气球 `[10,16]` 和 `[7,12]`。

示例 2:

输入: `points = [[1,2],[3,4],[5,6],[7,8]]`
输出: 4
解释: 每个气球需要射出一支箭，总共需要4支箭。

示例 3:

输入: `points = [[1,2],[2,3],[3,4],[4,5]]`
输出: 2
解释: 气球可以用2支箭来爆破:
- 在 $x = 2$ 处发射箭，击破气球 `[1,2]` 和 `[2,3]`。
- 在 $x = 4$ 处射出箭，击破气球 `[3,4]` 和 `[4,5]`。

题解

本题需要将区间按右端点进行排序。我们可以将射箭位置定在当前的右端点，从而保证尽可能多的覆盖更多的区间。

```
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if not points:
            return 0
        points.sort(key=lambda x:x[1])
        pos = points[0][1]
        ans = 1
        for point in points:
            if point[0] > pos:
                ans += 1
                pos = point[1]
        return ans
```

十五、栈和链表

15.1 数据结构简介

本章对应的题目为“栈”，“链表”。

(1) 基本定义

栈：栈是一种特殊的线性表，它只能在表的一端（称为栈顶）进行插入和删除操作。这使得栈具有“后进先出”（Last In First Out, LIFO）的特性，就像一摞盘子，最后放上去的盘子总是最先被取下来。在python中常用列表实现。

链表：链表由一系列节点（Node）组成。每个节点包含数据域和指针域。数据域用于存储具体的数据，指针域则存储指向下一个节点的引用。链表通过指针将各个节点连接起来，形成一个链式结构。包含单链表、双向链表，循环链表。

(2) 应用场景

栈：函数调用和递归，表达式求值，

链表：实现栈和队列，内存管理，哈希表的冲突处理

(3) 常用代码 (python)

栈

```
# 1. 初始化一个空栈
stack = []
# 2. 入栈操作
stack.append(1)
# 3. 出栈操作
popped_element = stack.pop()
# 4. 查看栈顶元素
top_element = stack[-1]
# 5. 获取栈的大小
stack_size = len(stack)
```

链表

```
# 1. 定义单链表节点类
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# 2. 头部插入节点
new_node = ListNode(val)
new_node.next = self.head
self.head = new_node
# 3. 尾部插入节点
new_node = ListNode(val)
if not self.head:
    self.head = new_node
current = self.head
while current.next:
    current = current.next
current.next = new_node
# 4. 添加一个哑巴节点dummy, 它的next指针指向头结点，从而不需要对头结点进行特殊的判断
dummy = ListNode(0, head) # dummy.next = head
```

15.2 题目（栈）

20. 有效的括号

题目描述

给定一个只包括 '('、')'、'{'、'}'、'['、']' 的字符串 `s`，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。
3. 每个右括号都有一个对应的相同类型的左括号。

示例 1：

输入：s = "()"

输出：true

示例 2：

输入：s = "()[]{}"

输出：true

示例 3：

输入：s = "[]"

输出：false

示例 4：

输入：s = "([])"

输出：true

题解

本题可以构建一个栈，以及括号对应关系的哈希表，然后遍历字符串中的字母。如果是左括号，就入栈，如果是右括号，就要判断当前栈顶元素是否是对应的左括号。如果字母和栈顶元素成功配对，就将栈顶元素出栈。

```
class Solution:
    def isValid(self, s: str) -> bool:

        stack = []
        hash_table = {
            ")": "(",
            "}": "{",
            "]": "[",
        }

        for ch in s:
            if ch in hash_table:
                if not stack or hash_table[ch] != stack[-1]:
```

```
        return False
    stack.pop()
else:
    stack.append(ch)
return not stack
```

71. 简化路径

题目描述

给你一个字符串 `path`，表示指向某一文件或目录的 Unix 风格 **绝对路径**（以 `'/'` 开头），请你将其转化为 **更加简洁的规范路径**。

在 Unix 风格的文件系统中规则如下：

- 一个点 `'.'` 表示当前目录本身。
- 此外，两个点 `'..'` 表示将目录切换到上一级（指向父目录）。
- 任意多个连续的斜杠（即，`'///'` 或 `'////'`）都被视为单个斜杠 `'/'`。
- 任何其他格式的点（例如，`'...'` 或 `'....'`）均被视为有效的文件/目录名称。

返回的 **简化路径** 必须遵循下述格式：

- 始终以斜杠 `'/'` 开头。
- 两个目录名之间必须只有一个斜杠 `'/'`。
- 最后一个目录名（如果存在）**不能** 以 `'/'` 结尾。
- 此外，路径仅包含从根目录到目标文件或目录的路径上的目录（即，不含 `'.'` 或 `'..'`）。

返回简化后得到的 **规范路径**。

示例 1：

输入：path = `"/home/"`

输出：`"/home"`

解释：

应删除尾随斜杠。

示例 2：

输入：path = `"/home//foo/"`

输出：`"/home/foo"`

解释：

多个连续的斜杠被单个斜杠替换。

示例 3：

输入: path = "/home/user/Documents/../Pictures"

输出: "/home/user/Pictures"

解释:

两个点 `".."` 表示上一级目录 (父目录) 。

示例 4:

输入: path = "/../"

输出: "/"

解释:

不可能从根目录上升一级目录。

示例 5:

输入: path = "/.../a/../b/c/../d../"

输出: "/.../b/d"

解释:

`"..."` 在这个问题中是一个合法的目录名。

题解

本题首先将字符串以 '/' 为分隔符进行切割, 并构建栈来保存路径名字, 然后遍历切割后的数字

1. 当碰到 '..' 时, 需要将栈顶元素移除, 也就是返回更上级的目录
2. 当路径不等于 '.' 且非空时, 可以入栈
3. 其他情况无需操作, 最后将栈的元素用 '/' 连接

```
class Solution:
    def simplifyPath(self, path: str) -> str:
        names = path.strip().split('/')
        stack = []
        for name in names:
            if name == "..":
                if stack:
                    stack.pop()
            elif name and name != '.':
                stack.append(name)
        return "/" + "/".join(stack)
```

155. 最小栈

题目描述

设计一个支持 `push` , `pop` , `top` 操作，并能在常数时间内检索到最小元素的栈。

实现 `MinStack` 类:

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

示例 1:

输入:

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

输出:

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

解释:

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

题解

本题在构建栈的同时，构建一个最小栈来辅助记录最小值。栈按照正常方式编写代码。当有元素入栈时，最小栈需要入的是栈顶元素和当前元素的最小值，而出栈时则栈和最小栈都需要移除栈顶元素，从而使最小栈的栈顶元素一直保持是栈的最小值。

```
class MinStack:  
  
    def __init__(self):  
        self.stack=[]  
        self.min_stack = [1000000000000]  
    def push(self, val: int) -> None:  
        self.stack.append(val)  
        self.min_stack.append(min(val,self.min_stack[-1]))  
    def pop(self) -> None:  
        self.stack.pop()  
        self.min_stack.pop()  
    def top(self) -> int:  
        return self.stack[-1]  
    def getMin(self) -> int:  
        return self.min_stack[-1]
```

150. 逆波兰表达式求值

题目描述

给你一个字符串数组 `tokens`，表示一个根据 [逆波兰表示法](#) 表示的算术表达式。

请你计算该表达式。返回一个表示表达式值的整数。

注意：

- 有效的算符为 `'+'`、`'-'`、`'*'` 和 `'/'`。
- 每个操作数（运算对象）都可以是一个整数或者另一个表达式。
- 两个整数之间的除法总是 **向零截断**。
- 表达式中不含除零运算。
- 输入是一个根据逆波兰表示法表示的算术表达式。
- 答案及所有中间计算结果可以用 **32 位** 整数表示。

示例 1：

输入：`tokens = ["2","1","+","3","*"]`

输出：9

解释：该算式转化为常见的中缀算术表达式为： $((2 + 1) * 3) = 9$

示例 2：

输入：`tokens = ["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 3：

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
```

题解

本题构建栈来存储字符：

- 当字符是数字时，直接入栈
- 当字符是运算符时，出栈两个元素，并使用运算符进行计算，将计算结果入栈。最后栈里只有一个元素，就是答案。

```
class Solution:
```



```
def evalRPN(self, tokens: List[str]) -> int:
    stack = []

    op = {
        "+": add,
        "-": sub,
        "*": mul,
        "/": lambda x,y: int(x/y),
    }
    for token in tokens:
        if token in op:
            num2 = stack.pop()
            num1 = stack.pop()
            stack.append(op[token](num1,num2))
        else:
            stack.append(int(token))
    return stack[-1]
```

15.3 题目（链表）

141. 环形链表

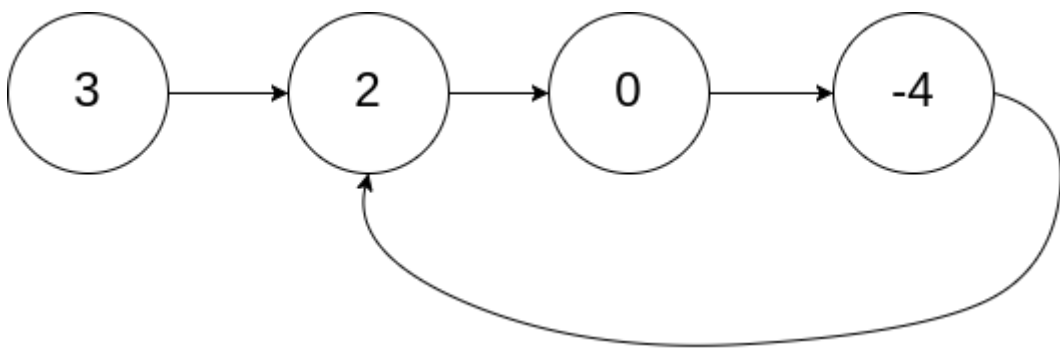
题目描述

给你一个链表的头节点 `head`，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。**注意：**`pos` 不作为参数进行传递。仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true`。否则，返回 `false`。

示例 1：

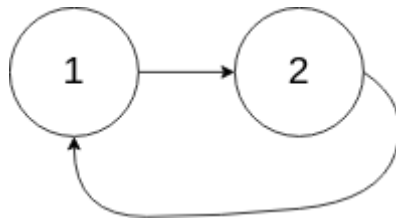


输入: `head = [3,2,0,-4]`, `pos = 1`

输出: `true`

解释: 链表中有一个环，其尾部连接到第二个节点。

示例 2：



输入: head = [1,2], pos = 0

输出: true

解释: 链表中有一个环, 其尾部连接到第一个节点。

示例 3:



输入: head = [1], pos = -1

输出: false

解释: 链表中没有环。

题解

本题用一个哈希表记录已访问的节点, 如果遍历到的节点已访问, 则有环。

```
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        seen = set()
        while head:
            if head in seen:
                return True
            seen.add(head)
            head = head.next
        return False
```

2. 两数相加

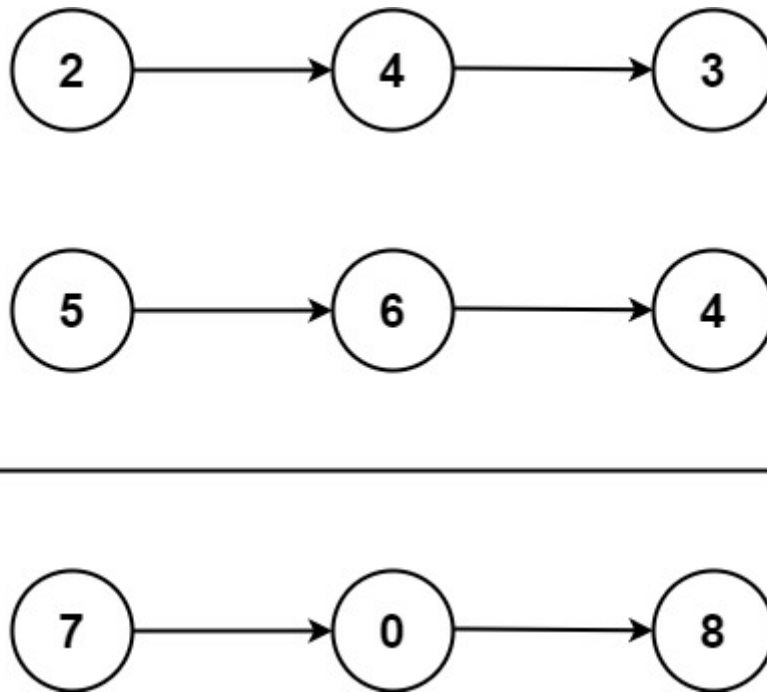
题目描述

给你两个 **非空** 的链表, 表示两个非负的整数。它们每位数字都是按照 **逆序** 的方式存储的, 并且每个节点只能存储 **一位** 数字。

请你将两个数相加, 并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外, 这两个数都不会以 0 开头。

示例 1:



输入: l1 = [2,4,3], l2 = [5,6,4]

输出: [7,0,8]

解释: 342 + 465 = 807.

示例 2:

输入: l1 = [0], l2 = [0]

输出: [0]

示例 3:

输入: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

输出: [8,9,9,9,0,0,0,1]

题解

本题同时遍历两个链表，逐位计算他们的和，并且加上进位数字（carry），如果两个链表长度不一样，就认为长度短的链表后面都是0。最后如果进位大于0，则l3再加一个节点，值为carry。

```
class Solution:
    def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
        curr = ListNode()
        head = curr
        carry = 0
        val = 0
        while carry or l1 or l2:
            val = carry

            # carry, val = divmod(val,10)
            # curr.next = ListNode(val)

            if l1:
                val = val + l1.val
                l1 = l1.next
```

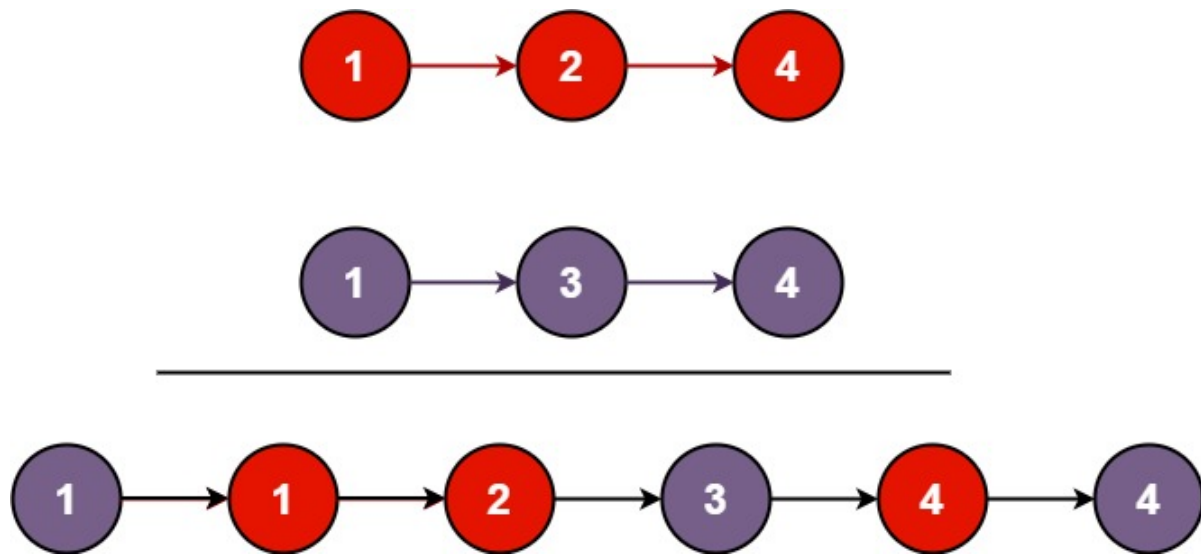
```
if l2:
    val = val + l2.val
    l2 = l2.next
    carry, val = divmod(val,10)
    curr.next = ListNode(val)
    curr = curr.next
return head.next
```

21. 合并两个有序链表

题目描述

将两个升序链表合并为一个新的 **升序** 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1:



输入: l1 = [1,2,4], l2 = [1,3,4]
输出: [1,1,2,3,4,4]

示例 2:

输入: l1 = [], l2 = []
输出: []

示例 3:

输入: l1 = [], l2 = [0]
输出: [0]

题解

本题同时遍历两个链表，并比较两个链表的节点值，将值更小的节点放入新链表中。如果list1已遍历完，则直接将list2的剩余节点放入新链表中，反之亦然。

```
class Solution:
```

```
def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode])
-> Optional[ListNode]:
    list3 = ListNode()
    head = list3
    while list1 or list2:
        if not list1:
            list3.next = list2
            return head.next
        if not list2:
            list3.next = list1
            return head.next
        if list1.val < list2.val:
            list3.next = ListNode(list1.val)
            list1 = list1.next
            list3 = list3.next
        else:
            list3.next = ListNode(list2.val)
            list2 = list2.next
            list3 = list3.next

    return head.next
```

138. 随机链表的复制

题目描述

给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 `random`，该指针可以指向链表中的任何节点或空节点。

构造这个链表的 [深拷贝](#)。深拷贝应该正好由 n 个 **全新** 节点组成，其中每个新节点的值都设为其对应的原节点的值。新节点的 `next` 指针和 `random` 指针也都应指向复制链表中的新节点，并使原链表和复制链表中的这些指针能够表示相同的链表状态。**复制链表中的指针都不应指向原链表中的节点。**

例如，如果原链表中有 x 和 y 两个节点，其中 $x.random \rightarrow y$ 。那么在复制链表中对应的两个节点 x 和 y ，同样有 $x.random \rightarrow y$ 。

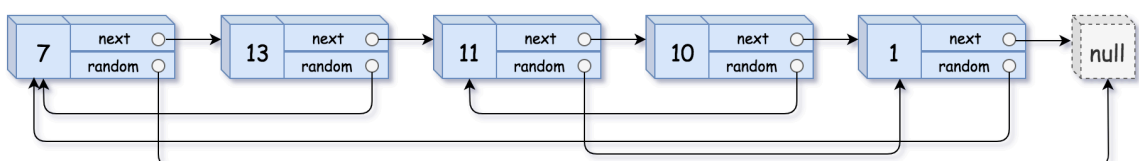
返回复制链表的头节点。

用一个由 n 个节点组成的链表来表示输入/输出中的链表。每个节点用一个 `[val, random_index]` 表示：

- `val`：一个表示 `Node.val` 的整数。
- `random_index`：随机指针指向的节点索引（范围从 `0` 到 `n-1`）；如果不指向任何节点，则为 `null`。

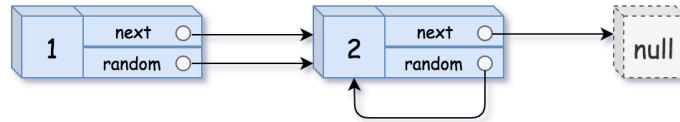
你的代码 **只** 接受原链表的头节点 `head` 作为传入参数。

示例 1：



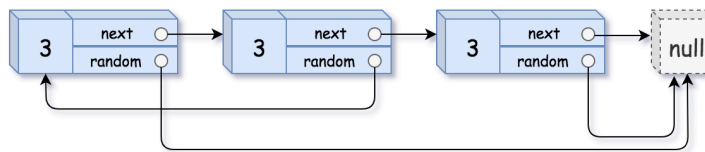
输入: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出: [[7,null],[13,0],[11,4],[10,2],[1,0]]

示例 2:



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

示例 3:



输入: head = [[3,null],[3,0],[3,null]]
输出: [[3,null],[3,0],[3,null]]

题解

本题分为三步:

1. 根据遍历的原节点创建对应的新节点, 每个新节点放在原节点后面, 例如 (1->1'->2->2'->3->3')
2. 设置新链表的随机指针: 例如原节点1的随机指针指向原节点3, 新节点1'的随机指针指向的是原节点3的next'
3. 分离两个链表, 返回新链表的表头

```
class Solution(object):
    def copyRandomList(self, head):
        if not head:
            return None
        p = head
        # 第一步, 在每个原节点后面创建一个新节点
        while p:
            new_node = Node(p.val)
            new_node.next = p.next
            p.next = new_node
            p = new_node.next
        p = head
        # 第二步, 设置新节点的随机节点
        while p:
            if p.random:
                p.next.random = p.random.next
            p = p.next.next
        # 第三步, 将两个链表分离
        p = head
```

```

dummy = Node(-1)
cur = dummy
while p:
    cur.next = p.next
    cur = cur.next
    p.next = cur.next
    p = p.next
return dummy.next

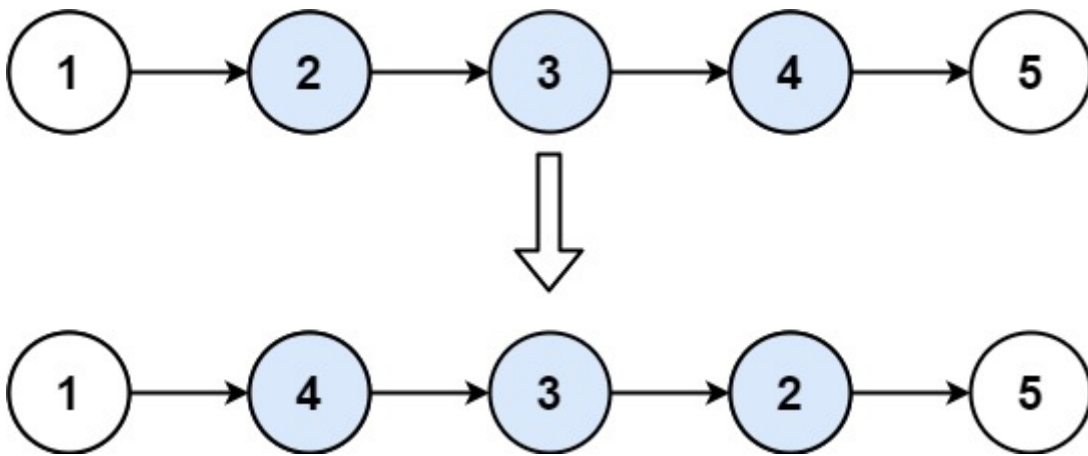
```

92. 反转链表II

题目描述

给你单链表的头指针 `head` 和两个整数 `left` 和 `right`，其中 `left <= right`。请你反转从位置 `left` 到位置 `right` 的链表节点，返回 **反转后的链表**。

示例 1:



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

输出: `[1,4,3,2,5]`

示例 2:

输入: `head = [5]`, `left = 1`, `right = 1`

输出: `[5]`

题解

```

class Solution:
    def reverseBetween(self, head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:
        def reverse_linked_list(head:ListNode):
            pre = None
            # cur = head
            while head:
                next_node = head.next
                head.next = pre
                pre = head
                head = next_node
            return pre

```

```

dummy_node = ListNode(-1)
dummy_node.next = head
pre = dummy_node
for _ in range(left-1):
    pre = pre.next
left_node = pre.next
right_node = pre
for _ in range(right-left+1):
    right_node = right_node.next
succ = right_node.next

pre.next = None
right_node.next = None
reverse = reverse_linked_list(left_node)
pre.next = reverse
left_node.next = succ

return dummy_node.next

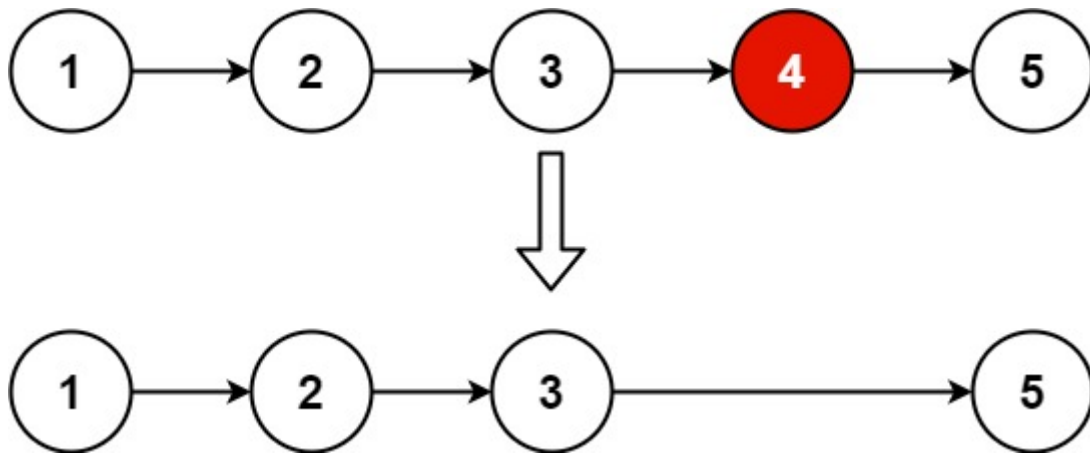
```

19. 删除链表的倒数第N个结点

题目描述

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

示例 1:



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

示例 2:

输入: head = [1], n = 1
输出: []

示例 3:

输入: head = [1,2], n = 1
输出: [1]

题解

```
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:

        def getLength(head: ListNode) -> int:
            length = 0
            while head:
                length += 1
                head = head.next
            return length

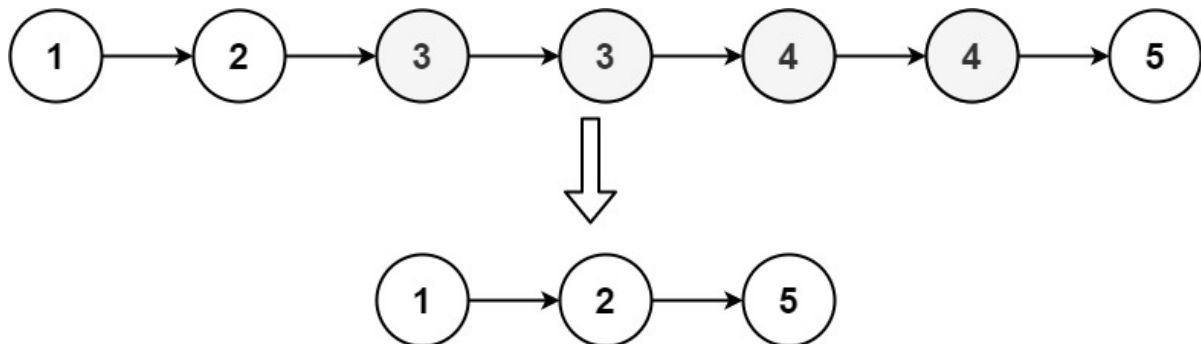
        dummy = ListNode(0, head)
        length = getLength(head)
        cur = dummy
        for i in range(length-n):
            cur = cur.next
        cur.next = cur.next.next
        return dummy.next
```

82. 删除排序链表中的重复元素II

题目描述

给定一个已排序的链表的头 `head`，删除原始链表中所有重复数字的节点，只留下不同的数字。返回已排序的链表。

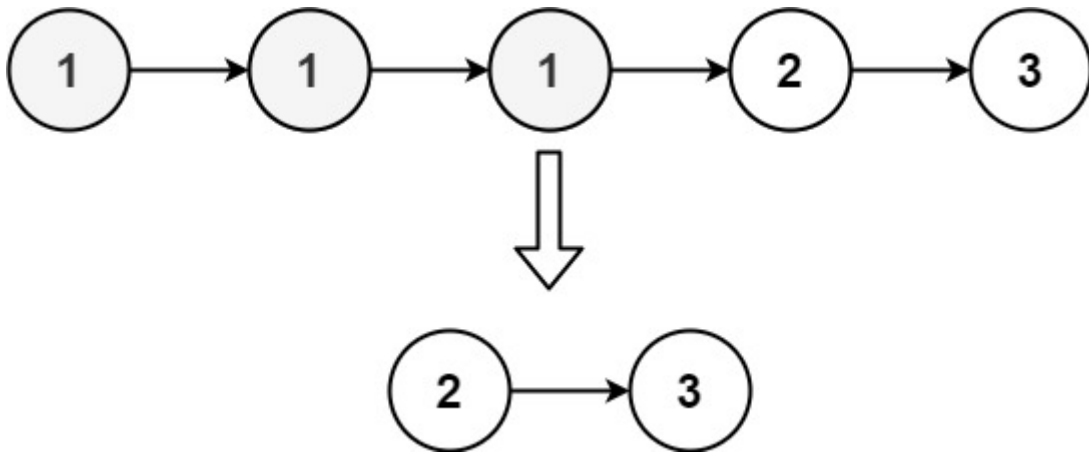
示例 1:



输入: `head = [1,2,3,3,4,4,5]`

输出: `[1,2,5]`

示例 2:



输入: head = [1,1,1,2,3]

输出: [2,3]

题解

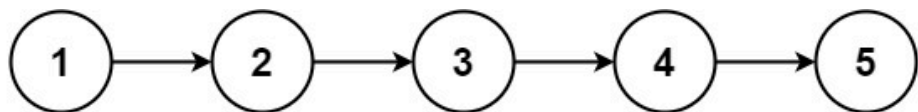
```
class Solution:
    def deleteDuplicates(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy_node = ListNode(-1, head)
        cur = dummy_node
        while(dummy_node.next and dummy_node.next.next):
            if dummy_node.next.val == dummy_node.next.next.val:
                x = dummy_node.next.val
                while dummy_node.next and dummy_node.next.val == x:
                    dummy_node.next = dummy_node.next.next
            else:
                dummy_node = dummy_node.next
        return cur.next
```

61. 旋转链表

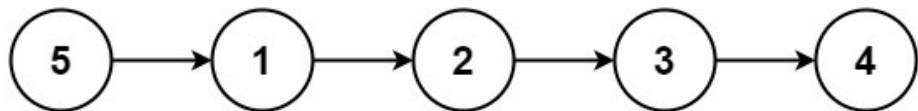
题目描述

给你一个链表的头节点 `head`，旋转链表，将链表每个节点向右移动 `k` 个位置。

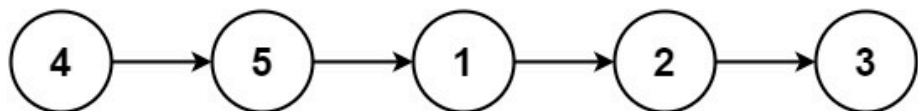
示例 1:



rotate 1

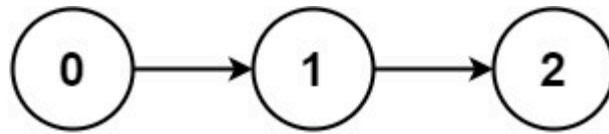


rotate 2

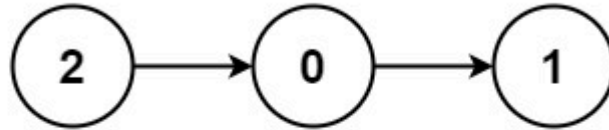


输入: head = [1,2,3,4,5], k = 2
输出: [4,5,1,2,3]

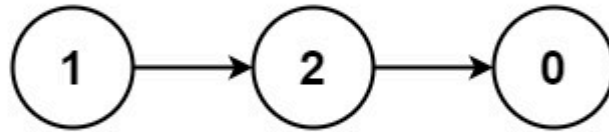
示例 2:



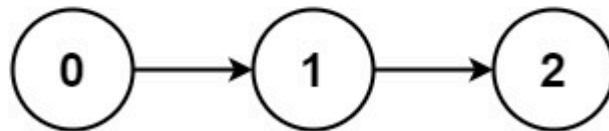
rotate 1



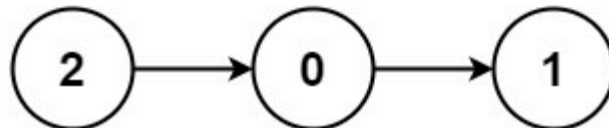
rotate 2



rotate 3



rotate 4



输入: head = [0,1,2], k = 4
输出: [2,0,1]

题解

```
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        if head == None:
            return None
        n = 1
        cur = head
        while(head.next):
            n=n+1
            head=head.next

        head.next = cur

        add = (n-k)%n
```

```

while(add):
    head = head.next
    add-=1

ret = head.next
head.next = None
return ret

```

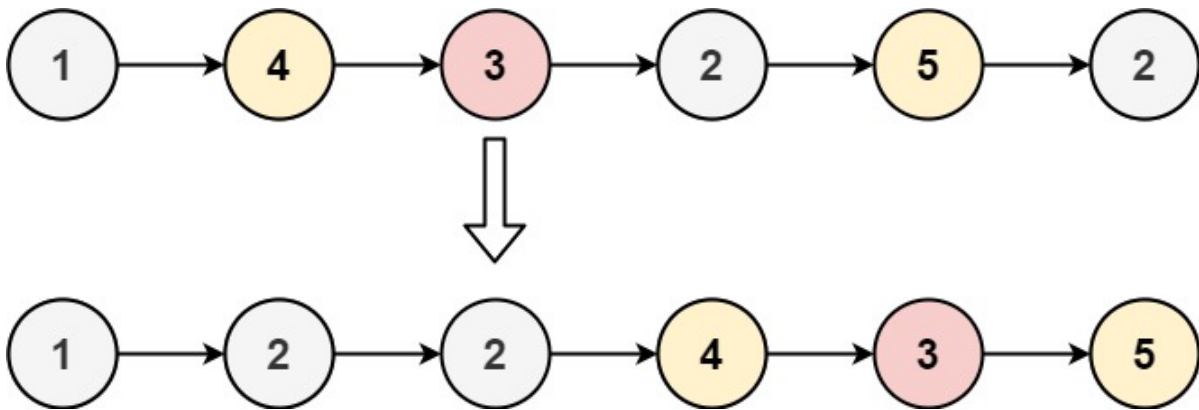
86. 分隔链表

题目描述

给你一个链表的头节点 `head` 和一个特定值 `x`，请你对链表进行分隔，使得所有 **小于** `x` 的节点都出现在 **大于或等于** `x` 的节点之前。

你应当 **保留** 两个分区中每个节点的初始相对位置。

示例 1:



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

示例 2:

输入: `head = [2,1]`, `x = 2`
 输出: `[1,2]`

题解

```

class Solution:
    def partition(self, head: Optional[ListNode], x: int) -> Optional[ListNode]:
        small_pro = small_head = ListNode()

        big_pro = big_head = ListNode()

        while(head):
            if head.val < x:
                small_head.next = head
                small_head = small_head.next
            else:
                big_head.next = head
                big_head = big_head.next

```

```
        head = head.next
    big_head.next = None

    small_head.next = big_pro.next
    return small_pro.next
```

146. LRU缓存

题目描述

请你设计并实现一个满足 [LRU \(最近最少使用\) 缓存](#) 约束的数据结构。

实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以 **正整数** 作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1`。
- `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值 `value`；如果不存在，则向缓存中插入该组 `key-value`。如果插入操作导致关键字数量超过 `capacity`，则应该**逐出**最久未使用的关键字。

函数 `get` 和 `put` 必须以 $O(1)$ 的平均时间复杂度运行。

示例：

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

输出
[null, null, null, 1, null, -1, null, -1, 3, 4]
```

解释

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

题解

```
class DLinkedNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
```

```

def __init__(self, capacity: int):
    self.cache = dict()
    # 使用伪头部和伪尾部节点
    self.head = DLinkedNode()
    self.tail = DLinkedNode()
    self.head.next = self.tail
    self.tail.prev = self.head
    self.capacity = capacity
    self.size = 0

def get(self, key: int) -> int:
    if key not in self.cache:
        return -1
    # 如果 key 存在, 先通过哈希表定位, 再移到头部
    node = self.cache[key]
    self.moveToHead(node)
    return node.value

def put(self, key: int, value: int) -> None:
    if key not in self.cache:
        # 如果 key 不存在, 创建一个新的节点
        node = DLinkedNode(key, value)
        # 添加进哈希表
        self.cache[key] = node
        # 添加至双向链表的头部
        self.addToHead(node)
        self.size += 1
        if self.size > self.capacity:
            # 如果超出容量, 删除双向链表的尾部节点
            removed = self.removeTail()
            # 删除哈希表中对应的项
            self.cache.pop(removed.key)
            self.size -= 1
    else:
        # 如果 key 存在, 先通过哈希表定位, 再修改 value, 并移到头部
        node = self.cache[key]
        node.value = value
        self.moveToHead(node)

def addToHead(self, node):
    node.prev = self.head
    node.next = self.head.next
    self.head.next.prev = node
    self.head.next = node

def removeNode(self, node):
    node.prev.next = node.next
    node.next.prev = node.prev

def moveToHead(self, node):
    self.removeNode(node)
    self.addToHead(node)

def removeTail(self):
    node = self.tail.prev

```

```
self.removeNode(node)
return node
```

十六、堆

16.1 数据结构简介

本章对应的题目为“堆”。

(1) 基本定义

堆是一种特殊的完全二叉树，完全二叉树是指除了最后一层外，每一层都被完全填充，并且最后一层的节点都尽可能靠左排列。在堆中，每个节点都满足特定的顺序性质。

大根堆：在大根堆中，每个节点的值都大于或等于其子节点的值。这意味着堆的根节点存储的是堆中所有元素的最大值

小根堆：与大根堆相反，小根堆中每个节点的值都小于或等于其子节点的值，所以堆的根节点存储的是堆中所有元素的最小值。

(2) 应用场景

优先队列、堆排序、图算法

(3) 常用代码 (python)

`heapq` 模块提供了一系列用于堆操作的函数，下面是一些常见操作的代码示例：

```
import heapq

# 1. 初始化一个列表
nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

# 2. 将列表转换为最小堆（原地操作）
heapq.heapify(nums)

# 3. 向堆中插入元素
heapq.heappush(nums, 0)

# 4. 从堆中弹出并返回最小元素
min_element = heapq.heappop(nums)

# 5. 获取最小的 n 个元素
smallest_n = heapq.nsmallest(3, nums)

# 6. 获取最大的 n 个元素
largest_n = heapq.nlargest(3, nums)
```

不适用`heapq`模块实现的堆排序算法

```
def heapify(self, nums, n, i): # 下浮操作，构建大根堆
    largest = i # 当前元素下标
    left = 2*i+1 # 左子节点下标
    right = 2*i+2 # 右子节点下标
    if left < n and nums[left] > nums[largest]: # 如果左子节点存在且大于根节点
        largest = left
    if right < n and nums[right] > nums[largest]: # 如果右子节点存在且大于根节点
```

```

        largest = right
    if largest != i: # 如果最大值节点不是当前根节点
        nums[i], nums[largest] = nums[largest], nums[i]
        self.heapify(nums, n, largest) # 递归调整受影响的子树

def heap_sort(self, nums):
    n = len(nums)
    for i in range(n//2-1, -1, -1): # 构建最大堆
        self.heapify(nums, n, i)
    for i in range(n-1, -1, -1): # 将堆顶元素放于底部，并且出堆，
        nums[i], nums[0] = nums[0], nums[i]
        self.heapify(nums, i, 0) # 将剩下的元素整理成大根堆
    return nums

```

16.2 题目（堆）

215. 数组中的第K个最大元素

题目描述

给定整数数组 `nums` 和整数 `k`，请返回数组中第 `**k**` 个最大的元素。

请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

你必须设计并实现时间复杂度为 `O(n)` 的算法解决此问题。

示例 1:

输入: [3,2,1,5,6,4], k = 2
输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6], k = 4
输出: 4

题解:

(添加一个快排算法)

```

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        pivot_val = random.choice(nums)

        larger, equal, smaller = [], [], []
        for num in nums:
            if num > pivot_val:
                larger.append(num)
            elif num < pivot_val:
                smaller.append(num)
            else:
                equal.append(num)

        if k <= len(larger):

```



```

        return self.findKthLargest(larger,k)
    if k > len(larger) + len(equal):
        return self.findKthLargest(smaller, k - len(larger)-len(equal))

    return pivot_val

```

(这是堆排序算法)

本题可以构建一个大根堆，进行堆排序后，返回nums[-k]，（还有一种方法是返回第k个堆顶元素，复杂度更低，可以思考一下）

```

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        new_nums = self.heap_sort(nums)
        return new_nums[-k]
    def heapify(self,nums,n,i): #下浮操作，构建大根堆
        largest = i # 当前元素下标
        left = 2*i+1 # 左子节点下标
        right = 2*i+2 # 右子节点下标
        if left < n and nums[left] > nums[largest]:# 如果左子节点存在且大于根节点
            largest = left
        if right < n and nums[right] > nums[largest]:# 如果右子节点存在且大于根节点
            largest = right
        if largest != i: # 如果最大值节点不是当前根节点
            nums[i],nums[largest] = nums[largest], nums[i]
            self.heapify(nums, n, largest) # 递归调整受影响的子树

    def heap_sort(self,nums):
        n = len(nums)
        for i in range(n//2-1,-1,-1): # 构建最大堆
            self.heapify(nums,n,i)
        for i in range(n-1,-1,-1):
            nums[i],nums[0] = nums[0], nums[i]
            self.heapify(nums,i,0)
        return nums

```

373. 查找和最小的K对数字

题目描述

给定两个以 **非递减顺序排列** 的整数数组 `nums1` 和 `nums2`，以及一个整数 `k`。

定义一对值 `(u,v)`，其中第一个元素来自 `nums1`，第二个元素来自 `nums2`。

请找到和最小的 `k` 个数对 `(u1,v1)`，`(u2,v2)` ... `(uk,vk)`。

示例 1:

输入: `nums1 = [1,7,11]`, `nums2 = [2,4,6]`, `k = 3`

输出: `[1,2]`, `[1,4]`, `[1,6]`

解释: 返回序列中的前 3 对数:

`[1,2]`, `[1,4]`, `[1,6]`, `[7,2]`, `[7,4]`, `[11,2]`, `[7,6]`, `[11,4]`, `[11,6]`

示例 2:

输入: nums1 = [1,1,2], nums2 = [1,2,3], k = 2

输出: [1,1],[1,1]

解释: 返回序列中的前 2 对数:

[1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

题解

本题可以借助最小堆, 堆中保存下标对 (i,j), 即可能成为下一个数对的a的下标i和b的下标j, 堆顶是最小的a[i]+b[j]。初始把 (0,0) 入堆, 每次出堆时, 可能成为下一个数对的是(i+1, j)和 (i, j+1) 这两个入堆。

```
class Solution:
    def kSmallestPairs(self, nums1: List[int], nums2: List[int], k: int) -> List[List[int]]:
        m, n = len(nums1), len(nums2)
        ans = []
        pq = [(nums1[i]+nums2[0], i, 0) for i in range(min(k, m))]
        while pq and len(ans) < k:
            _, i, j = heappop(pq)
            ans.append([nums1[i], nums2[j]])
            if j+1 < n:
                heappush(pq, (nums1[i]+nums2[j+1], i, j+1))
        return ans
```

十七、特殊的二叉树

17.1 数据结构简介

本题对应的题目是“二叉搜索树”和“字典树”

(1) 基本定义

二叉搜索树: 二叉搜索树 (排序树、查找树) 是一种二叉树, 他满足以下条件:

1. 若它的左子树不为空, 则左子树上所有节点的值均小于它的根节点的值。
1. 若它的右子树不为空, 则右子树上所有节点的值均大于它的根节点的值。
1. 它的左、右子树也分别为二叉搜索树。

二叉搜索树的性质:

1. 对二叉搜索树进行中序遍历, 得到的节点值是一个升序序列
1. 二叉搜索树每个节点的值都是唯一的

字典树: (比较少见, 目前只能先记住这两个题的代码)

节点结构: 字典树的每个节点都包含若干个指向其他节点的指针, 通常以字符为索引。每个节点还可能包含一个标记位, 用于表示从根节点到该节点所形成的字符串是否是一个完整的单词。

树的结构：根节点不存储任何字符，从根节点开始，每一条从根到叶子节点或部分中间节点的路径都代表一个字符串。路径上的字符连接起来就是对应的字符串。例如，若从根节点经过节点“a”“p”“p”“l”“e”，则表示字符串“apple”。

(2) 应用场景

二叉搜索树：数据搜索、排序、实现关联数组

字典树：单词拼写和检查、自动补全、IP地址查找

(3) 常用代码

二叉搜索树

```
# 构建二叉搜索树及基本操作
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
class BinarySearchTree:
    def __init__(self): #
        self.root = None

    def insert(self, val): #插入节点
        if self.root is None:
            self.root = TreeNode(val)
        else:
            self._insert_recursive(self.root, val)

    def _insert_recursive(self, node, val):
        if val < node.val:
            if node.left is None:
                node.left = TreeNode(val)
            else:
                self._insert_recursive(node.left, val)
        else:
            if node.right is None:
                node.right = TreeNode(val)
            else:
                self._insert_recursive(node.right, val)

    def search(self, val):
        return self._search_recursive(self.root, val)

    def _search_recursive(self, node, val):
        if node is None or node.val == val:
            return node
        elif val < node.val:
            return self._search_recursive(node.left, val)
        else:
            return self._search_recursive(node.right, val)
```

字典树

字典树构建以及基本操作

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
```

17.2 题目（二叉搜索树）

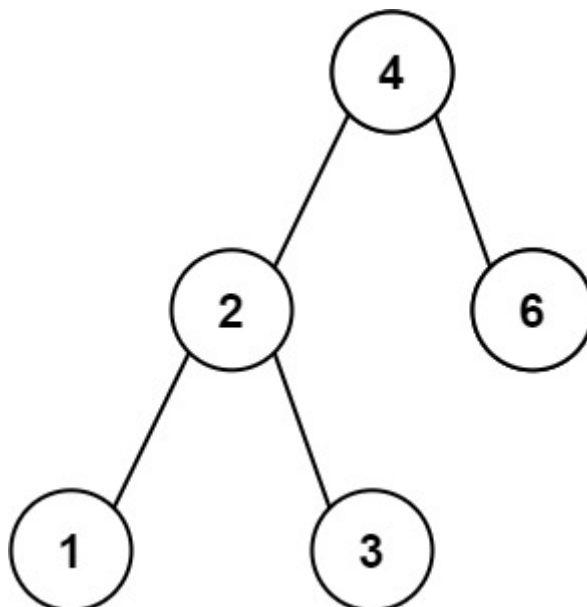
530.二叉搜索树的最小绝对差

题目描述

给你一个二叉搜索树的根节点 `root`，返回 树中任意两不同节点值之间的最小差值。

差值是一个正数，其数值等于两值之差的绝对值。

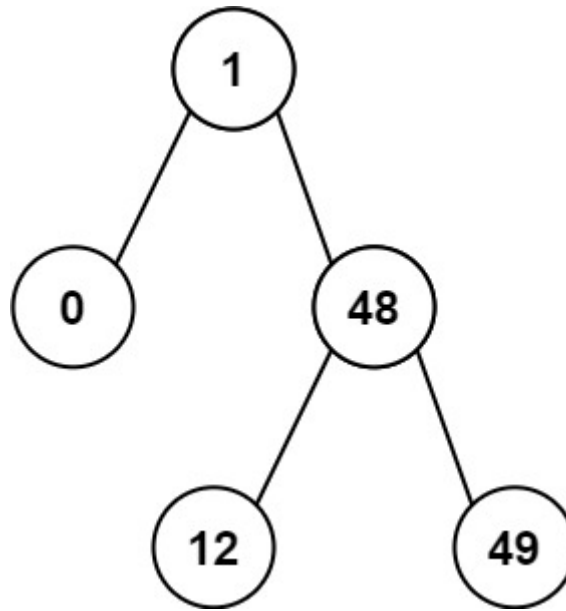
示例 1:



输入: root = [4,2,6,1,3]

输出: 1

示例 2:



输入: root = [1,0,48,null,null,12,49]

输出: 1

题解:

中序遍历二叉搜索树可以得到升序序列, 因此本题先中序遍历, 然后计算升序序列的相邻元素最小差值。

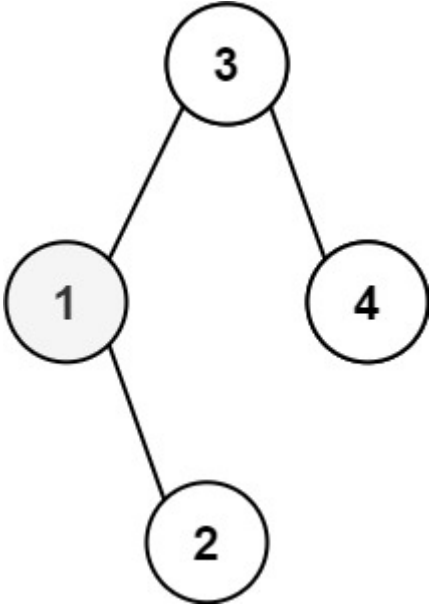
```
class Solution(object):
    def getMinimumDifference(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: int
        """
        res = []
        def inorder(root):
            if not root:
                return
            inorder(root.left)
            res.append(root.val)
            inorder(root.right)
        inorder(root)
        min_val = float('inf')
        for i in range(len(res)-1):
            min_val = min(min_val, res[i+1]-res[i])
        return min_val
```

230. 二叉搜索树中第K小的元素

题目描述

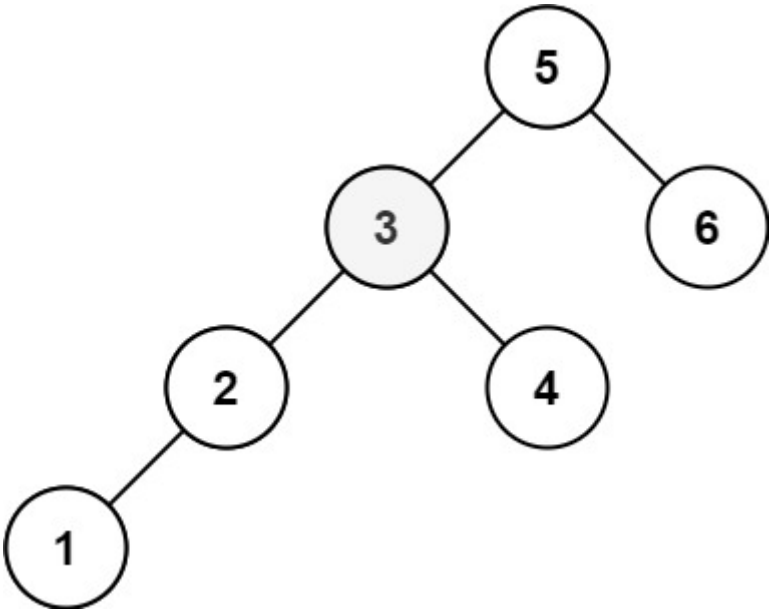
给定一个二叉搜索树的根节点 `root`， 和一个整数 `k`， 请你设计一个算法查找其中第 `k` 小的元素（从 1 开始计数）。

示例 1:



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

示例 2:



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

题解

中序遍历，然后返回第k小的数字。

```
class Solution(object):
    def kthSmallest(self, root, k):
        """
        :type root: Optional[TreeNode]
        :type k: int
        :rtype: int
        """
        res = []
        def inorder(root):
            if not root:
                return
            inorder(root.left)
            res.append(root.val)
            inorder(root.right)
        inorder(root)
        return res[k-1]
```

98. 验证二叉搜索树

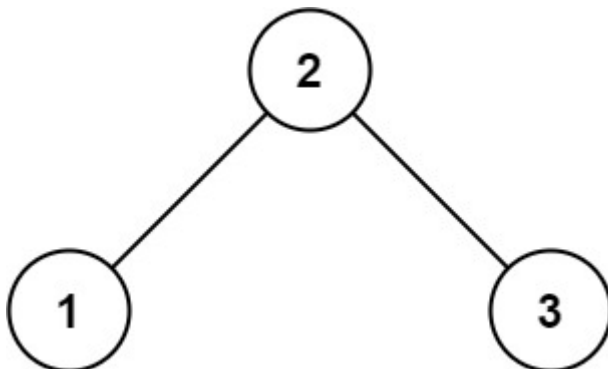
题目描述

给你一个二叉树的根节点 `root`，判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

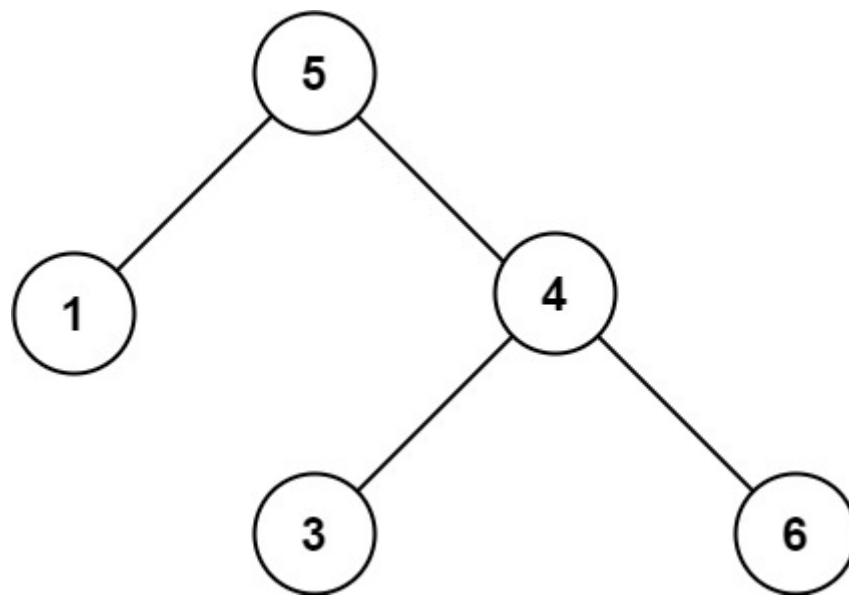
- 节点的左子树只包含小于 当前节点的数。
- 节点的右子树只包含 **大于** 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1：



输入: `root = [2,1,3]`
输出: `true`

示例 2：



输入: root = [5,1,4,null,null,3,6]

输出: false

解释: 根节点的值是 5 , 但是右子节点的值是 4 。

题解

中序遍历, 然后判断中序遍历后是否是升序序列

```
class Solution(object):
    def isValidBST(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: bool
        """
        res = []
        def inorder(root):
            if not root:
                return
            inorder(root.left)
            res.append(root.val)
            inorder(root.right)
        inorder(root)
        for i in range(len(res)-1):
            if res[i+1]<=res[i]:
                return False
        return True
```

17.3 题目 (字典树)

208 实现Trie(前缀树)

题目描述

Trie（发音类似 "try"）或者说 **前缀树** 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补全和拼写检查。

请你实现 Trie 类：

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word`。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true`（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`。

示例：

输入

```
["Trie", "insert", "search", "search", "startswith", "insert", "search"]  
[[], ["apple"], ["apple"], ["app"], ["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
```

提示：

- `1 <= word.length, prefix.length <= 2000`
- `word` 和 `prefix` 仅由小写英文字母组成
- `insert`、`search` 和 `startswith` 调用次数 **总计** 不超过 `3 * 104` 次

题解

本题参考灵神思路，构建Node类，相比于官方题解更容易理解。Trie的每个节点需要包含所有可选的字母，每一次遍历相当于选择一个字母。在本题中，可选的字母为'a'-'z'，共有二十六个，因此实际上是构建的一个二十六叉树。

1. 初始化：可以使用一个26维的指针数组children来表示子节点；以及一个布尔字段isEnd来表示字符串是否结束。
2. 插入字符串：遍历字符串word，对于当前字符对应的子节点存在两种情况：
 1. 子节点存在，沿着指针移动到子节点，继续搜索下一个字符是否存在，即是否有对应的子节点。
 2. 子节点不存在，创建一个新的子节点，记录在children数组的对应位置上，然后沿着指针移动到子节点上，继续搜索下一个子节点
3. 查找前缀：从字典树的根开始，查找前缀，对于当前字符对应的子节点存在两种情况：
 1. 子节点存在，沿着指针移动到子节点，继续搜索下一个字符

2. 子节点不存在, 说明不包含该前缀, 返回空指针

```
class Node(object):
    def __init__(self):
        self.son = [None] * 26
        self.end = False

class Trie(object):
    def __init__(self):
        self.root = Node()

    def insert(self, word: str) -> None:
        cur = self.root
        for c in word:
            c_index = ord(c) - ord('a')
            if cur.son[c_index] is None:
                cur.son[c_index] = Node()
            cur = cur.son[c_index]
        cur.end = True

    def find(self, word: str) -> int:
        cur = self.root
        for c in word:
            c_index = ord(c) - ord('a')
            if cur.son[c_index] is None:
                return 0
            cur = cur.son[c_index]
        return 2 if cur.end else 1

    def search(self, word: str) -> bool:
        return self.find(word) == 2

    def startswith(self, prefix: str) -> bool:
        return self.find(prefix) != 0
```

211. 添加与搜索单词 - 数据结构设计

题目描述

请你设计一个数据结构, 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。

示例:

输入:

```
["wordDictionary", "addword", "addword", "addword", "search", "search", "search", "search"]
```

```
[[], ["bad"], ["dad"], ["mad"], ["pad"], ["bad"], [".ad"], ["b.."]]
```

输出:

```
[null, null, null, null, false, true, 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
```

题解

本题插入单词的函数没有变化 (addWord()函数和前一题的insert()函数相同), 而搜索单词在前一题的基础上, 多了对"."的判断。如果当前字符是点号, 由于点号可以表示任何字母, 因此需要对当前结点的所有非空子结点继续搜索下一个字符。

```
class Node:
    def __init__(self):
        self.son = [None] * 26
        self.end = False

class wordDictionary:

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

    def addWord(self, word: str) -> None:
        cur = self.root
        for c in word:
            c_index = ord(c) - ord('a')
            if cur.son[c_index] is None:
                cur.son[c_index] = Node()
            cur = cur.son[c_index]
        cur.end = True

    def search(self, word: str) -> bool:
        def dfs(index, tri_node):
            if index == len(word):
                return tri_node.end
            c = word[index]
            if c != '.':
                son_node = tri_node.son[ord(c)-ord('a')]
                if son_node is not None and dfs(index+1, son_node):
                    return True
            else:
                for son_node in tri_node.son:
                    if son_node is not None and dfs(index+1, son_node):
                        return True
            return False
        return dfs(0, self.root)
```

```
return dfs(0,self.root)
```