

目录

一、哈希	1
第 1 题 两数之和	1
第 49 题 字母异位词	3
第 128 题 最长连续序列	5
二、双指针	7
第 283 题 移动零	7
第 11 题 盛最多水的容器	9
第 15 题 三数之和	11
第 42 题 接雨水	13
三、滑动窗口	16
第 3 题 无重复字符的最长子串	16
第 438 题 找到字符串中所有字母异位词	18
四、子串	20
第 560 题 和为 K 的子数组	20
第 239 题 滑动窗口最大值	23
第 76 题 最小覆盖子串	24
五、普通数组	25
第 53 题 最大子数组和	25
第 56 题 合并区间	27
第 189 题 轮转数组	29
第 238 题 除自身以外数组的乘积	32

第 41 题 缺失的第一个正数	34
六、矩阵	35
第 73 题 矩阵置零	35
第 54 题 螺旋矩阵	38
第 48 题 旋转图像	45
第 240 题 搜索二维矩阵 II	48
七、链表	51
第 141 题 环形链表	51
第 160 题 相交链表	54
第 206 题 反转链表	57
第 92 题 反转链表 II	62
第 234 题 回文链表	63
第 21 题 合并两个有序链表	65
第 142 题 环形链表 II	67
第 2 题 两数相加	71
第 19 题 删除链表的倒数第 N 个节点	73
第 24 题 两两交换链表中的节点	75
第 25 题 K 个一组翻转链表	78
第 138 题 随机链表的复制	79
第 148 题 排序链表	81
第 23 题 合并 K 个升序链表	82
第 146 题 LRU 缓存	83

八、二叉树	92
第 94 题 二叉树的中序遍历	92
第 104 题 二叉树的最大深度	95
第 226 题 翻转二叉树	97
第 101 题 对称二叉树	99
第 543 题 二叉树的直径	104
第 102 题 二叉树的层序遍历	108
第 103 题 二叉树的锯齿形层序遍历	113
第 108 题 将有序数组转换为二叉搜索树	118
第 98 题 验证二叉搜索树	122
第 230 题 二叉搜索树中第 K 个大的元素	123
第 199 题 二叉树的右视图	124
第 114 题 二叉树展开为链表	125
第 105 题 从前序与中序遍历序列构造二叉树	126
第 437 题 路经总和III	127
第 236 题 二叉树的最近公共祖先	128
第 124 题 二叉树中的最大路径和	129
九、图论	130
第 200 题 岛屿数量	130
十、回溯	135
第 46 题 全排列	135
第 47 题 全排列 II	141

第 78 题 子集	143
第 51 题 N 皇后	145
十一、二分查找	146
第 35 题 搜索插入位置	146
第 74 题 搜索二维矩阵	148
第 34 题 在排序数组中查找元素的第一个和最后一个位置 .	152
第 33 题 搜索旋转排序数组	154
第 153 题 寻找旋转排序数组中的最小值	157
十二、栈	159
第 20 题 有效的括号	159
第 155 题 最小栈	162
第 394 题 字符串解码	165
第 739 题 每日温度	168
十三、堆	171
第 215 题 数组中的第 K 个最大元素	171
第 347 题 前 K 个高频元素	172
十四、贪心算法	173
第 121 题 买卖股票的最佳时机	173
第 122 题 买卖股票的最佳时机 II	175
第 55 题 跳跃游戏	176
第 45 题 跳跃游戏 II	178
第 763 题 划分字母区间	180

十五、动态规划	184
第 70 题 爬楼梯	184
第 118 题 杨辉三角	186
第 198 题 打家劫舍	189
第 279 题 完全平方数	192
第 494 题 目标和	196
第 322 题 零钱兑换	199
第 518 题 零钱兑换 II	202
十六、多维动态规划	205
第 62 题 不同路径	205
第 64 题 最小路径和	208
十七、技巧	210
第 136 题 只出现一次的数字	210
第 169 题 多数元素	211
十八、CodeTop 补充	215
第 88 题 合并两个有序数组	215
第 232 题 用栈实现队列	217
第 225 题 用队列实现栈	224
第 912 题 排序数组	227
十大排序总结:	236
第 5 题 最长回文子串	237
第 300 题 最长递增子序列	240

第 415 题 字符串相加	241
第 143 题 重排链表	244
第 72 题 编辑距离	245
第 93 题 复原 IP 地址	246
第 1143 题 最长公共子序列	247
第 4 题 寻找两个正序数组的中位数	248
第 82 题 删除排序链表中的重复元素 II	250
第 704 题 二分查找	251
第 31 题 下一个排列	253
第 22 题 括号生成	254
第 8 题 字符串转换整数 (atoi)	255
第 69 题 x 的平方根	256
第 165 题 比较版本号	258
第 32 题 最长有效括号	259
第 LCR 140 题 训练计划 II (与 19 题同理)	260
第 43 题 字符串相乘	262
第 151 题 反转字符串中的单词	265
第 129 题 求根节点到叶节点数字之和	268
第 144 题 二叉树的前序遍历	269
第 110 题 平衡二叉树	270
第 39 题 组合总和	271
第 470 题 用 Rand7() 实现 Rand10()	272

第 221 题 最大正方形	273
第 162 题 寻找峰值	274
第 113 题 路径总和 II	275
第 14 题 最长公共前缀	276
第 662 题 二叉树最大宽度	277
第 112 题 路径总和	278
第 152 题 乘积最大子数组	279
第 695 题 岛屿的最大面积	280
第 179 题 最大数	281
第 227 题 基本计算器 II	282
第 718 题 最长重复子数组	283
第 139 题 单词拆分	284
第 209 题 长度最小的子数组	285

一、哈希

第1题 两数之和

1. 两数之和

已解答

[简单](#) [相关标签](#) [相关企业](#) [提示](#) [A_x](#)

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]`

Python3 智能模式

```
1 class Solution:
2     def twoSum(self, nums: List[int], target: int) -> List[int]:
3
4         dict={}
5
6         for i in range(len(nums)):
7             if target - nums[i] not in dict:
8                 dict[nums[i]]=i
9             else:
10                 return [i,dict[target-nums[i]]]
11
```

思路 2：哈希表

哈希表中键值对信息为 $\$target - \text{nums}[i] : i$, 其中 i 为下标。

1. 遍历数组，对于每一个数 $\text{nums}[i]$:

1. 先查找字典中是否存在 $\text{target} - \text{nums}[i]$, 存在则输出 $\text{target} - \text{nums}[i]$ 对应的下标和当前数组的下标 i 。
2. 不存在则在字典中存入 $\text{target} - \text{nums}[i]$ 的下标 i 。

思路 2：代码

```
1 def twoSum(self, nums: List[int], target: int) -> List[int]:  
2     numDict = dict()  
3     for i in range(len(nums)):  
4         if target-nums[i] in numDict:  
5             return numDict[target-nums[i]], i  
6         numDict[nums[i]] = i  
7     return [0]
```

py

思路 2：复杂度分析

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

第 49 题 字母异位词

题目描述 | 题解 | 提交记录

49. 字母异位词分组

中等 相关标签 相关企业 A+

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"]]
```

```
</> 代码 | 笔记 X
Python3 ▾ 智能模式
1 class Solution:
2     def groupAnagrams(self, strs: List[str]) -> List[List[str]]
```



Stream

发布于 北京 · (编辑过) · 2024.11.15

人话都不会说。【把字母数量相同的字符串分到一组】很难吗？示例：
["nat", "tan"]都是一个a, 一个n, 一个t, 所以分到一组。["ate", "eat", "tea"]都是一个a, 一个e, 一个t, 所以分到一组。

1 20 隐藏回复 回复 分享 ...



Stream

发布于 北京 2024.11.15

["nat", "tan"]都是一个a, 一个n, 一个t, 所以分到一组。
["ate", "eat", "tea"]都是一个a, 一个e, 一个t, 所以分到一组。

1 0 回复

```
Pythons v  目录树
```

```
1 class Solution:
2     def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
3         if len(strs) == 1:
4             return [strs]
5
6         ans = {}
7         for ss in strs:
8             s = str(sorted(ss))      #注意是字符串str
9             if s not in ans:
10                 ans[s] = [ss]    #注意ss是列表
11             else:
12                 ans[s].append(ss)
13
14         return list(ans.values())
15
```

为什么 `return [ans.values()]` 会报错?

假设你有以下的字典 `ans` :

```
python
```

```
ans = {
    "['a', 'e', 't']": ['eat', 'tea', 'ate'],
    "['a', 'n', 't']": ['tan', 'nat'],
    "['a', 'b', 't']": ['bat']
}
```

复制代码

`ans.values()` 会返回字典中所有的值 (即每个键对应的列表)。在上面的例子中, `ans.values()` 会返回:

```
python
```

```
dict_values([['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']])
```

复制代码

这时候:

- `ans.values()` 是一个 `dict_values` 对象, 它是一个可迭代对象, 可以通过 `list()` 转换为列表。
- 如果你使用 `return [ans.values()]`, 你实际上是将整个 `dict_values` 对象作为一个元素, 放入一个新的列表中, 这会导致你得到一个包含 `dict_values` 对象的列表, 而不是你想要的原始列表。

举个例子，`return [ans.values()]` 实际上返回的是：

python

复制代码

```
[dict_values([['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']])]
```

这意味着，你的返回值是一个包含 `dict_values` 对象的列表，而不是直接包含原始字符串列表的列表。

正确的做法：`return list(ans.values())`

`ans.values()` 返回一个 `dict_values` 对象，它是一个可迭代的集合。要把它转换成列表，你应该使用 `list(ans.values())`，这样可以直接得到字典的所有值作为一个列表。

例如，`return list(ans.values())` 会返回：

python

复制代码

```
[['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
```

这样是你想要的一个列表，其中每个元素都是一个包含原始字符串的列表。

总结：

- `return [ans.values()]` 会返回一个包含 `dict_values` 对象的列表，导致结果不符合预期。
- `return list(ans.values())` 才能将字典中的值提取出来，正确地返回一个列表，包含了所有的字符串列表。

第 128 题 最长连续序列

128. 最长连续序列

中等 相关标签 相关企业 A

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

The screenshot shows a code editor window with Python 3 selected. The code defines a class `Solution` with a method `longestConsecutive` that takes a list of integers `nums` and returns an integer. The code has been saved and is at line 1, column 1.

```
Python3 智能模式
1 class Solution:
2     def longestConsecutive(self, nums: List[int]) -> int:
3
```

思路 1：哈希表

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

思路 1：代码

```
1 class Solution:
2     def longestConsecutive(self, nums: List[int]) -> int:
3         ans = 0
4         nums_set = set(nums)
5         for num in nums_set:
6             if num - 1 not in nums_set:
7                 curr_num = num
8                 curr_streak = 1
9
10            while curr_num + 1 in nums_set:
11                curr_num += 1
12                curr_streak += 1
13            ans = max(ans, curr_streak)
14
15        return ans
```

py

思路 1：复杂度分析

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



二、双指针

第 283 题 移动零

283. 移动零

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

请注意，必须在不复制数组的情况下原地对数组进行操作。

示例 1:

输入: `nums = [0,1,0,3,12]`
输出: `[1,3,12,0,0]`

示例 2:

输入: `nums = [0]`
输出: `[0]`

```
Python3 v 智能模式
1 class Solution:
2     def moveZeroes(self, nums: List[int]) -> None:
3         """
4             Do not return anything, modify nums in-place instead.
5         """
6
```

```
</> 代码 | 笔记 x
Python3 v 智能模式
1 class Solution:
2     def moveZeroes(self, nums: List[int]) -> None:
3         """
4             Do not return anything, modify nums in-place instead.
5         """
6         i= 0
7         for num in nums:
8             if num !=0:
9                 nums[i]=num
10                i+=1
11         for j in range(i,len(nums)):
12             nums[j]=0
```

思路 2：快慢指针

1. 使用两个指针 $slow$, $fast$ 。 $slow$ 指向处理好的非 0 数字数组的尾部, $fast$ 指针指向当前待处理元素。
2. 不断向右移动 $fast$ 指针, 每次移动到非零数, 则将左右指针对应的数交换, 交换同时将 $slow$ 右移。
3. 此时, $slow$ 指针左侧均为处理好的非零数, 而从 $slow$ 指针指向的位置开始, $fast$ 指针左边为止都为 0。

遍历结束之后, 则所有 0 都移动到了右侧, 且保持了非零数的相对位置。

思路 2：代码

```
1 class Solution:
2     def moveZeroes(self, nums: List[int]) -> None:
3         slow = 0
4         fast = 0
5         while fast < len(nums):
6             if nums[fast] != 0:
7                 nums[slow], nums[fast] = nums[fast], nums[slow]
8                 slow += 1
9             fast += 1
```

py

思路 2：复杂度分析

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



第 11 题 盛最多水的容器

11. 盛最多水的容器

中等 相关标签 相关企业 提示 A+

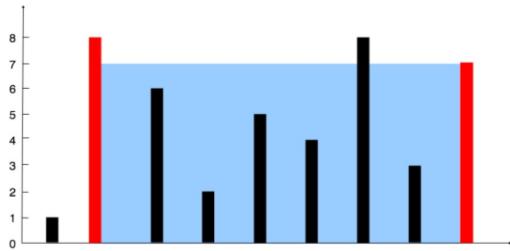
给定一个长度为 n 的整数数组 $height$ 。有 n 条垂线，第 i 条线的两个端点是 $(i, 0)$ 和 $(i, height[i])$ 。

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

Python3 智能模式

```
1 class Solution:
2     def maxArea(self, height: List[int]) -> int:
3         pass
```

已存储 行 3, 列 9

测试用例 > 测试结果

Case 1 Case 2 +

height =

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

思路 1：对撞指针

从示例中可以看出，如果确定好左右两端的直线，容纳的水量是由 左右两端直线中较低直线的高度 * 两端直线之间的距离 所决定的。所以我们应该使得 **较低直线的高度尽可能的高**，这样才能使盛水面积尽可能的大。

可以使用对撞指针求解。移动较低直线所在的指针位置，从而得到不同的高度和面积，最终获取其中最大的面积。具体做法如下：

1. 使用两个指针 `left`, `right`。`left` 指向数组开始位置, `right` 指向数组结束位置。
2. 计算 `left` 和 `right` 所构成的面积值，同时维护更新最大面积值。
3. 判断 `left` 和 `right` 的高度值大小。
 1. 如果 `left` 指向的直线高度比较低，则将 `left` 指针右移。
 2. 如果 `right` 指向的直线高度比较低，则将 `right` 指针左移。
4. 如果遇到 `left == right`，跳出循环，最后返回最大的面积。

思路 1：代码

```
1 class Solution:
2     def maxArea(self, height: List[int]) -> int:
3         left = 0
4         right = len(height) - 1
5         ans = 0
6         while left < right:
7             area = min(height[left], height[right]) * (right-left)
8             ans = max(ans, area)
9             if height[left] < height[right]:
10                 left += 1
11             else:
12                 right -= 1
13         return ans
```

解释：

- **初始化：** 左指针 `left` 指向数组的第一个元素，右指针 `right` 指向数组的最后一个元素，`max_area` 用于记录最大面积。
- **计算面积：** 每次计算当前左指针和右指针之间的面积，并更新最大面积。
- **移动指针：** 由于容器的面积由较小的高度决定，因此我们总是选择移动较小高度的指针，试图找到一个更高的线，从而可能增大容器的面积。
- **终止条件：** 当左指针与右指针相遇时，停止循环。

第 15 题 三数之和

15. 三数之和

中等 相关标签 相关企业 提示

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 $i \neq j$ 、 $i \neq k$ 且 $j \neq k$ ，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：

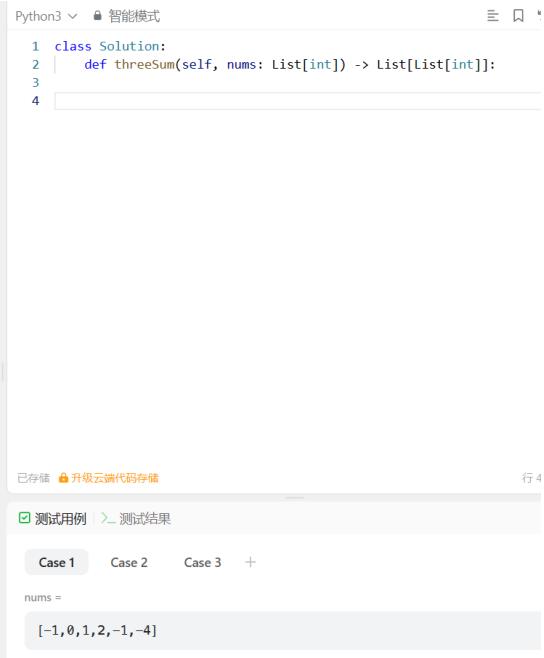
输入: `nums = [-1, 0, 1, 2, -1, -4]`
输出: `[[-1, -1, 2], [-1, 0, 1]]`
解释:
`nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0`。
`nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0`。
`nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0`。
不同的三元组是 `[-1, 0, 1]` 和 `[-1, -1, 2]`。
注意，输出的顺序和三元组的顺序并不重要。

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。



The screenshot shows a Python code editor interface. At the top, it says "Python3" and "智能模式". The code area contains the following Python code:

```
1 class Solution:
2     def threeSum(self, nums: List[int]) -> List[List[int]]:
3
4
```

Below the code, there is a note: "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 $i \neq j$ 、 $i \neq k$ 且 $j \neq k$ ，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。

On the right side of the editor, there is a sidebar with the following sections:

- 已存储 (Upgraded to cloud code storage)
- 行 4
- 测试用例 (Test Case)
 - Case 1 (selected)
 - Case 2
 - Case 3
 - +
- nums = [-1, 0, 1, 2, -1, -4]

解题思路

思路 1：对撞指针

直接三重遍历查找 a 、 b 、 c 的时间复杂度是: $O(n^3)$ 。我们可以通过一些操作来降低复杂度。

先将数组进行排序，以保证按顺序查找 a 、 b 、 c 时，元素值为升序，从而保证所找到的三个元素是不重复的。同时也方便下一步使用双指针减少一重遍历。时间复杂度为: $O(n \times \log n)$ 。

第一重循环遍历 a ，对于每个 a 元素，从 a 元素的下一个位置开始，使用对撞指针 $left$ 、 $right$ 。 $left$ 指向 a 元素的下一个位置， $right$ 指向末尾位置。先将 $left$ 右移、 $right$ 左移去除重复元素，再进行下边的判断。

- 如果 $nums[a] + nums[left] + nums[right] == 0$ ，则得到一个解，将其加入答案数组中，并继续将 $left$ 右移， $right$ 左移；
- 如果 $nums[a] + nums[left] + nums[right] > 0$ ，说明 $nums[right]$ 值太大，将 $right$ 向左移；
- 如果 $nums[a] + nums[left] + nums[right] < 0$ ，说明 $nums[left]$ 值太小，将 $left$ 右移。

思路 1：代码

```
1  class Solution:
2      def threeSum(self, nums: List[int]) -> List[List[int]]:
3          n = len(nums)
4          nums.sort()
5          ans = []
6
7          for i in range(n):
8              if i > 0 and nums[i] == nums[i - 1]:
9                  continue
10             left = i + 1
11             right = n - 1
12             while left < right:
13                 while left < right and left > i + 1 and nums[left] == nums[left - 1]:
14                     left += 1
15                 while left < right and right < n - 1 and nums[right + 1] == nums[right]:
16                     right -= 1
17                 if left < right and nums[i] + nums[left] + nums[right] == 0:
18                     ans.append([nums[i], nums[left], nums[right]])
19                     left += 1
20                     right -= 1
21                 elif nums[i] + nums[left] + nums[right] > 0:
22                     right -= 1
23                 else:
24                     left += 1
25
26         return ans
```

第 42 题 接雨水

42. 接雨水

困难 相关标签 相关企业 A+

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

示例 1：



输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

输出：6

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

示例 2：

输入：height = [4,2,0,3,2,5]

输出：9

```
1 class Solution:
2     def trap(self, height: List[int]) -> int:
3         pass
```

已存储 行 3, 列 5
测试用例 测试结果

思路 1：单调栈

1. 遍历高度数组 `height`。
2. 如果当前柱体高度较小，小于等于栈顶柱体的高度，则将当前柱子高度入栈。
3. 如果当前柱体高度较大，大于栈顶柱体的高度，则一直出栈，直到当前柱体小于等于栈顶柱体的高度。
4. 假设当前柱体为 `C`，出栈柱体为 `B`，出栈之后新的栈顶柱体为 `A`。则说明：
 1. 当前柱体 `C` 是出栈柱体 `B` 向右找到的第一个大于当前柱体高度的柱体，那么以出栈柱体 `B` 为中心，可以向右将宽度扩展到当前柱体 `C`。
 2. 新的栈顶柱体 `A` 是出栈柱体 `B` 向左找到的第一个大于当前柱体高度的柱体，那么以出栈柱体 `B` 为中心，可以向左将宽度扩展到当前柱体 `A`。
5. 出栈后，以新的栈顶柱体 `A` 为左边界，以当前柱体 `C` 为右边界，以左右边界与出栈柱体 `B` 的高度差为深度，计算可以接到雨水的面积。然后记录并更新累积面积。

思路 1：代码

```
1  class Solution:
2      def trap(self, height: List[int]) -> int:
3          ans = 0
4          stack = []
5          size = len(height)
6          for i in range(size):
7              while stack and height[i] > height[stack[-1]]:
8                  cur = stack.pop(-1)
9                  if stack:
10                     left = stack[-1] + 1
11                     right = i - 1
12                     high = min(height[i], height[stack[-1]]) - height[cur]
13                     ans += high * (right - left + 1)
14                 else:
15                     break
16             stack.append(i)
17         return ans
```

思路 1：复杂度分析

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

灵茶山艾府

发布于 浙江 · (编辑过) · 2023.07.22

知名 hard 题，经典面试题。如果不理解原理，脑中没有一副清晰的图像，刷多少遍都记不住。

为此我精心制作了一期视频，带你直观地理解接雨水的原理！请看[【基础算法精讲 02】](#)。

```
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = left = pre_max = suf_max = 0
        right = len(height) - 1
        while left < right:
            pre_max = max(pre_max, height[left])
            suf_max = max(suf_max, height[right])
            if pre_max < suf_max:
                ans += pre_max - height[left]
                left += 1
            else:
                ans += suf_max - height[right]
                right -= 1
        return ans
```

97 隐藏回复 回复 分享 ...



人民没有币

发布于 山西 2023.07.23

终于找到个用相向指针的了。题解里都是dp和单调栈搞得我都怀疑自己了

三、滑动窗口

第3题 无重复字符的最长子串

3. 无重复字符的最长子串

中等 相关标签 相关企业 提示 Ax

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 = "bbbbb"

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。

The screenshot shows a Python code editor interface with the following details:

- Editor title: Python3
- Mode: 智能模式
- Code content:

```
1 class Solution:
2     def lengthOfLongestSubstring(self, s: str) -> int:
3
```
- Status bar: 已解答 (Solved)
- Bottom status bar: 行 (Line)

解题思路

思路 1：滑动窗口（不定长度）

用滑动窗口 window 来记录不重复的字符个数，window 为哈希表类型。

1. 设定两个指针：left、right，分别指向滑动窗口的左右边界，保证窗口中没有重复字符。
2. 一开始，left、right 都指向 0。
3. 向右移动 right，将最右侧字符 s[right] 加入当前窗口 window 中，记录该字符个数。
4. 如果该窗口中该字符的个数多于 1 个，即 $window[s[right]] > 1$ ，则不断右移 left，缩小滑动窗口长度，并更新窗口中对应字符的个数，直到 $window[s[right]] \leq 1$ 。
5. 维护更新无重复字符的最长子串长度。然后继续右移 right，直到 $right \geq len(nums)$ 结束。
6. 输出无重复字符的最长子串长度。

思路 1：代码

```
1 class Solution:
2     def lengthOfLongestSubstring(self, s: str) -> int:
3         left = 0
4         right = 0
5         window = dict()
6         ans = 0
7
8         while right < len(s):
9             if s[right] not in window:
10                 window[s[right]] = 1
11             else:
12                 window[s[right]] += 1
13
14             while window[s[right]] > 1:
15                 window[s[left]] -= 1
16                 left += 1
17
18             ans = max(ans, right - left + 1)
19             right += 1
20
21         return ans
```



第 438 题 找到字符串中所有字母异位词

438. 找到字符串中所有字母异位词

中等 相关标签 相关企业 A+

给定两个字符串 s 和 p ，找到 s 中所有 p 的 [异位词](#) 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

示例 1：

输入： $s = "cbaebabacd"$, $p = "abc"$
输出： $[0, 6]$

解释：
起始索引等于 0 的子串是 "cba"，它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac"，它是 "abc" 的异位词。

示例 2：

输入： $s = "abab"$, $p = "ab"$
输出： $[0, 1, 2]$

解释：
起始索引等于 0 的子串是 "ab"，它是 "ab" 的异位词。
起始索引等于 1 的子串是 "ba"，它是 "ab" 的异位词。
起始索引等于 2 的子串是 "ab"，它是 "ab" 的异位词。

```
Python3 智能模式
1 class Solution:
2     def findAnagrams(self, s: str, p: str) -> List[int]:
3         pass
```

已存储 行 3, 列 9

解题思路

思路 1：滑动窗口

维护一个固定长度为 $\text{len}(p)$ 的滑动窗口。于是问题的难点变为了如何判断 s 的子串和 p 是异位词。可以使用两个字典来分别存储 s 的子串中各个字符个数和 p 中各个字符个数。如果两个字典对应的键值全相等，则说明 s 的子串和 p 是异位词。但是这样每一次比较的操作时间复杂度是 $O(n)$ ，我们可以通过在滑动数组中逐字符比较的方式来减少两个字典之间相互比较的复杂度，并用 $valid$ 记录经过验证的字符个数。整个算法步骤如下：

- 使用哈希表 $need$ 记录 p 中各个字符出现次数。使用字典 $window$ 记录 s 的子串中各个字符出现的次数。使用数组 res 记录答案。使用 $valid$ 记录 s 的子串中经过验证的字符个数。使用 $window_size$ 表示窗口大小，值为 $\text{len}(p)$ 。使用两个指针 $left$ 、 $right$ 。分别指向滑动窗口的左右边界。
- 一开始， $left$ 、 $right$ 都指向 0。
- 如果 $s[right]$ 出现在 $need$ 中，将最右侧字符 $s[right]$ 加入当前窗口 $window$ 中，记录该字符个数。并验证该字符是否和 $need$ 中对应字符个数相等。如果相等则验证的字符个数加 1，即 $valid += 1$ 。
- 如果该窗口字符长度大于等于 $window_size$ 个，即 $right - left + 1 \geq window_size$ 。则不断右移 $left$ ，缩小滑动窗口长度。
 - 如果验证字符个数 $valid$ 等于窗口长度 $window_size$ ，则 $s[left, right + 1]$ 为 p 的异位词，所以将 $left$ 加入到答案数组中。
 - 如果 $s[left]$ 在 $need$ 中，则更新窗口中对应字符的个数，同时维护 $valid$ 值。
- 右移 $right$ ，直到 $right \geq \text{len}(p)$ 结束。
- 输出答案数组 res 。

思路 1：代码

```
1 class Solution:
2     def findAnagrams(self, s: str, p: str) -> List[int]:
3         need = collections.defaultdict(int)
4         for ch in p:
5             need[ch] += 1
6
7         window = collections.defaultdict(int)
8         window_size = len(p)
9         res = []
10        left, right = 0, 0
11        valid = 0
12        while right < len(s):
13            if s[right] in need:
14                window[s[right]] += 1
15                if window[s[right]] == need[s[right]]:
16                    valid += 1
17
18            if right - left + 1 >= window_size:
19                if valid == len(need):
20                    res.append(left)
21                if s[left] in need:
22                    if window[s[left]] == need[s[left]]:
23                        valid -= 1
24                    window[s[left]] -= 1
25                left += 1
26            right += 1
27        return res
```



四、子串

第 560 题 和为 K 的子数组

560. 和为 K 的子数组

尝试过 

中等

相关标签

相关企业

提示

A*

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

子数组是数组中元素的连续非空序列。

示例 1:

输入: `nums = [1,1,1]`, `k = 2`
输出: 2

示例 2:

输入: `nums = [1,2,3]`, `k = 3`
输出: 2

Python3 ▾ 智能模式

```
1 class Solution:  
2     def subarraySum(self, nums: List[int], k: int) -> int:  
3         pass
```

已存储

思路 1：枚举算法（超时）

先考虑暴力做法，外层两重循环，遍历所有连续子数组，然后最内层再计算一下子数组的和。部分代码如下：

```
1  for i in range(len(nums)):
2      for j in range(i + 1):
3          sum = countSum(i, j)
```

py

这样下来时间复杂度就是 $O(n^3)$ 了。下一步是想办法降低时间复杂度。

对于以 i 开头，以 j 结尾 ($i \leq j$) 的子数组 $nums[i] \dots nums[j]$ 来说，我们可以通过顺序遍历 j ，逆序遍历 i 的方式（或者前缀和的方式），从而在 $O(n^2)$ 的时间复杂度内计算出子数组的和，同时使用变量 cnt 统计出和为 k 的子数组个数。

但这样提交上去超时了。

思路 1：代码

```
1  class Solution:
2      def subarraySum(self, nums: List[int], k: int) -> int:
3          cnt = 0
4          for j in range(len(nums)):
5              sum = 0
6              for i in range(j, -1, -1):
7                  sum += nums[i]
8                  if sum == k:
9                      cnt += 1
10
11      return cnt
```

py

思路 2：前缀和 + 哈希表

先用一重循环遍历数组，计算出数组 $nums$ 中前 j 个元素的和（前缀和），保存到一维数组 pre_sum 中，那么对于任意 $nums[i] \dots nums[j]$ 的子数组的和为 $pre_sum[j] - pre_sum[i - 1]$ 。这样计算子数组和的时间复杂度降为了 $O(1)$ 。总体时间复杂度为 $O(n^2)$ 。

但是还是超时了。。

由于我们只关心和为 k 出现的次数，不关心具体的解，可以使用哈希表来加速运算。

$pre_sum[i]$ 的定义是前 i 个元素和，则 $pre_sum[i]$ 可以由 $pre_sum[i - 1]$ 递推而来，即：
 $pre_sum[i] = pre_sum[i - 1] + num[i]$ 。
[$i..j$] 子数组和为 k 可以转换为： $pre_sum[j] - pre_sum[i - 1] == k$ 。

综合一下，可得： $pre_sum[i - 1] == k$ 。

所以，当我们考虑以 j 结尾和为 k 的连续子数组个数时，只需要统计有多少个前缀和为 $pre_sum[j] - k$ （即 $pre_sum[i - 1]$ ）的个数即可。具体做法如下：

- 使用 pre_sum 变量记录前缀和（代表 $pre_sum[j]$ ）。
- 使用哈希表 pre_dic 记录 $pre_sum[j]$ 出现的次数。键值对为 $pre_sum[j] : pre_sum_count$ 。
- 从左到右遍历数组，计算当前前缀和 pre_sum 。
- 如果 $pre_sum - k$ 在哈希表中，则答案个数累加上 $pre_dic[pre_sum - k]$ 。
- 如果 pre_sum 在哈希表中，则前缀和个数累加 1，即 $pre_dic[pre_sum] += 1$ 。
- 最后输出答案个数。

思路 2：代码

```
1 class Solution:
2     def subarraySum(self, nums: List[int], k: int) -> int:
3         pre_dic = {0: 1}
4         pre_sum = 0
5         count = 0
6         for num in nums:
7             pre_sum += num
8             if pre_sum - k in pre_dic:
9                 count += pre_dic[pre_sum - k]
10            if pre_sum in pre_dic:
11                pre_dic[pre_sum] += 1
12            else:
13                pre_dic[pre_sum] = 1
14        return count
```

思路 2：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

239. 滑动窗口最大值

困难

相关标签

相关企业

提示

A₂

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回 滑动窗口中的最大值。

示例 1:

输入: `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`

输出: `[3,3,5,5,6,7]`

解释:

滑动窗口的位置

最大值

[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

第 76 题 最小覆盖子串

76. 最小覆盖子串

困难

相关标签

相关企业

提示

A₂

给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 \dots 。

注意：

- 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
- 如果 s 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

输入: $s = "ADOBECODEBANC"$, $t = "ABC"$

输出: "BANC"

解释: 最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。

示例 2：

输入: $s = "a"$, $t = "a"$

输出: "a"

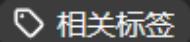
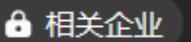
解释: 整个字符串 s 是最小覆盖子串。

五、普通数组

第 53 题 最大子数组和

53. 最大子数组和

已解答 

 中等  相关标签  相关企业 

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

```
Python3 ▼ 智能模式

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

具体步骤：

1. 初始化：

- `nums[i] += max(nums[i-1], 0)` 这一行的作用是：更新当前元素 `nums[i]` 的值为：当前元素的值 + 前一个元素的最大和，如果前一个元素的和是负数，就不加前一个元素，而是取 0。这样可以保证如果加上前一个元素后结果变得更小（负数），则直接从当前元素重新开始新的子数组。

2. 解释：

- `max(nums[i-1], 0)` 的含义是：如果 `nums[i-1]` 是负数，则当前元素的最大和将从 0 开始。也就是说，当前的子数组不包括前一个元素。
- 如果 `nums[i-1]` 是正数，当前的最大子数组和将会包括前一个元素，即 `nums[i-1]`。

3. 返回最大值：

- 经过循环更新后的 `nums` 数组，`nums[i]` 代表的是以 `i` 位置为结尾的最大子数组和。因此 `max(nums)` 取整个数组的最大值，即为最大子数组和。

第 56 题 合并区间

56. 合并区间

中等

相关标签

相关企业

A+

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。

思路 1：排序

1. 设定一个数组 `ans` 用于表示最终不重叠的区间数组，然后对原始区间先按照区间左端点大小从小到大进行排序。
2. 遍历所有区间。
3. 先将第一个区间加入 `ans` 数组中。
4. 然后依次考虑后边的区间：
 1. 如果第 `i` 个区间左端点在前一个区间右端点右侧，则这两个区间不会重合，直接将该区间加入 `ans` 数组中。
 2. 否则的话，这两个区间重合，判断一下两个区间的右区间值，更新前一个区间的右区间值为较大值，然后继续考虑下一个区间，以此类推。
5. 最后返回数组 `ans`。

思路 1：代码

```
1 class Solution:
2     def merge(self, intervals: List[List[int]]) -> List[List[int]]:
3         intervals.sort(key=lambda x: x[0])
4
5         ans = []
6         for interval in intervals:
7             if not ans or ans[-1][1] < interval[0]:
8                 ans.append(interval)
9             else:
10                 ans[-1][1] = max(ans[-1][1], interval[1])
11
12         return ans
```

思路 1：复杂度分析

- **时间复杂度：** $O(n \times \log_2 n)$ 。其中 n 为区间数量。
- **空间复杂度：** $O(n)$ 。

- **条件解释：**

- `not ans`：表示结果列表为空，即当前是第一个区间，直接加入结果列表。
- `ans[-1][1] < interval[0]`：当前区间的起始值大于结果列表中最后一个区间的结束值，说明它们不重叠，需要将当前区间加入结果列表。

- **作用：**如果当前区间和结果列表中最后一个区间有重叠，则合并它们。

- `ans[-1][1]`：结果列表中最后一个区间的结束值。
- `interval[1]`：当前区间的结束值。
- `max(ans[-1][1], interval[1])`：取两个区间的结束值中较大的值，作为合并后的结束值。

第 189 题 轮转数组

189. 轮转数组

中等

相关标签

相关企业

提示

A

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]`

1	2	3	4	5	6	7
---	---	---	---	---	---	---

$k = 3$

7	6	5	4	3	2	1
---	---	---	---	---	---	---

反转整个数组

5	6	7	4	3	2	1
---	---	---	---	---	---	---

反转前 k 个

5	6	7	1	2	3	4
---	---	---	---	---	---	---

反转后 $n - k$ 个

核心思路

右旋 k 次等价于：

1. 将数组的最后 k 个元素移到数组开头。
2. 其余元素顺序接在后面。

通过 **反转数组** 来实现这一效果：

1. **反转整个数组。**
2. **反转前 k 个元素。**
3. **反转剩余部分。**

```
1 class Solution:
2     def rotate(self, nums: List[int], k: int) -> None:
3         """
4             Do not return anything, modify nums in-place instead.
5         """
6
7         def reverse(i, j):
8             while i < j:
9                 nums[i], nums[j] = nums[j], nums[i]
10                i += 1
11                j -= 1
12
13        n = len(nums)
14        k %= n # 轮转 k 次等于轮转 k%n 次
15        reverse(0, n - 1)
16        reverse(0, k - 1) ←
17        reverse(k, n - 1)
18
```

举例

- 输入数组: `nums = [1, 2, 3, 4, 5]`
- 如果 `k = 7`, 数组右旋转 7 次:
 - 结果与右旋转 $7 \% 5 = 2$ 次相同。
 - 旋转 2 次后, 结果为 `[4, 5, 1, 2, 3]`。

第 238 题 除自身以外数组的乘积

238. 除自身以外数组的乘积

中等

相关标签

相关企业

提示

Ax

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]

思路 1：两次遍历

1. 构造一个答案数组 `res`，长度和数组 `nums` 长度一致。
2. 先从左到右遍历一遍 `nums` 数组，将 `nums[i]` 左侧的元素乘积累积起来，存储到 `res` 数组中。
3. 再从右到左遍历一遍，将 `nums[i]` 右侧的元素乘积累积起来，再乘以原本 `res[i]` 的值，即为 `nums` 中除了 `nums[i]` 之外的其他所有元素乘积。

思路 1：代码

```
1 class Solution:
2     def productExceptSelf(self, nums: List[int]) -> List[int]:
3         size = len(nums)
4         res = [1 for _ in range(size)]
5
6         left = 1
7         for i in range(size):
8             res[i] *= left
9             left *= nums[i]
10
11        right = 1
12        for i in range(size-1, -1, -1):
13            res[i] *= right
14            right *= nums[i]
15
16        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

python

复制代码

```
for i in range(size-1, -1, -1):
```

这句话的意思是从索引 `size-1` 开始，逐步递减到 `0`，包括 `0`。

解释

- `range(start, stop, step)`：
 - `start`：循环开始的值。
 - `stop`：循环终止条件（不包括 `stop` 的值）。
 - `step`：步长，表示每次循环变量的增量（可以为负数表示递减）。

在这段代码中：

- `size-1`：表示数组的最后一个索引（数组的长度为 `size`，最后一个索引是 `size-1`）。
- `-1`：表示循环结束在索引 `-1` 前，实际上循环到 `0` 为止（因为 `-1` 是 `stop`，不会包括它）。
- `-1`：步长为 `-1`，表示每次索引递减 1。

第 41 题 缺失的第一个正数

41. 缺失的第一个正数

困难

相关标签

相关企业

提示

A_文

给你一个未排序的整数数组 `nums`，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为 $O(n)$ 并且只使用常数级别额外空间的解决方案。

示例 1:

输入: `nums = [1, 2, 0]`

输出: 3

解释: 范围 [1, 2] 中的数字都在数组中。

示例 2:

输入: `nums = [3, 4, -1, 1]`

输出: 2

解释: 1 在数组中，但 2 没有。

示例 3:

输入: `nums = [7, 8, 9, 11, 12]`

输出: 1

解释: 最小的正数 1 没有出现。

六、矩阵

第 73 题 矩阵置零

73. 矩阵置零

中等 相关标签 相关企业 提示 Ax

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]]

```
1 class Solution:
2     def setZeroes(self, matrix: List[List[int]]) -> None:
3         """
4             Do not return anything, modify matrix in-place instead.
5         """
6
```

解题思路

思路 1：使用标记变量

直观上可以使用两个数组来标记行和列出现 0 的情况，但这样空间复杂度就是 $O(m + n)$ 了，不符合题意。

考虑使用数组原本的元素进行记录出现 0 的情况。

1. 设定两个变量 $flag_row0$ 、 $flag_col0$ 来标记第一行、第一列是否出现了 0。
2. 接下来我们使用数组第一行、第一列来标记 0 的情况。
3. 对数组除第一行、第一列之外的每个元素进行遍历，如果某个元素出现 0 了，则使用数组的第一行、第一列对应位置来存储 0 的标记。
4. 再对数组除第一行、第一列之外的每个元素进行遍历，通过对第一行、第一列的标记 0 情况，进行置为 0 的操作。
5. 最后再根据 $flag_row0$ 、 $flag_col0$ 的标记情况，对第一行、第一列进行置为 0 的操作。

思路 1：代码

```
1  class Solution:
2      def setZeroes(self, matrix: List[List[int]]) -> None:
3          m = len(matrix)
4          n = len(matrix[0])
5          flag_col0 = False
6          flag_row0 = False
7          for i in range(m):
8              if matrix[i][0] == 0:
9                  flag_col0 = True
10                 break
11
12          for j in range(n):
13              if matrix[0][j] == 0:
14                  flag_row0 = True
15                  break
16
17          for i in range(1, m):
18              for j in range(1, n):
19                  if matrix[i][j] == 0:
20                      matrix[i][0] = matrix[0][j] = 0
21
22          for i in range(1, m):
23              for j in range(1, n):
24                  if matrix[i][0] == 0 or matrix[0][j] == 0:
25                      matrix[i][j] = 0
26
27          if flag_col0:
28              for i in range(m):
29                  matrix[i][0] = 0
30
31          if flag_row0:
32              for j in range(n):
33                  matrix[0][j] = 0
```

第 54 题 螺旋矩阵

54. 螺旋矩阵

中等

相关标签

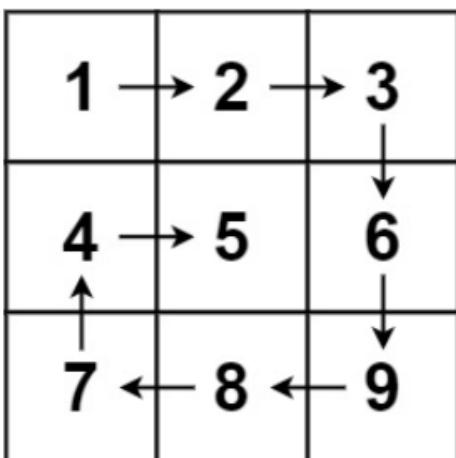
相关企业

提示

Ax

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]`

Python3 ▾ 智能模式

```
1 class Solution:
2     def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
3
4         result = []
5         while matrix:
6             result += matrix.pop(0) # 取矩阵第一行并删除
7             matrix = list(zip(*matrix))[:-1] # 旋转矩阵
8
9         return result
10
```

第一次循环

1. 取出第一行并删除:

- `matrix.pop(0)` 会返回 `[1, 2, 3]` 并删除它。
- 更新后的 `matrix` 变为：

```
python
[
    [4, 5, 6],
    [7, 8, 9]
]
```

 复制代码

- 更新后的 `result` 变为：

```
python
[1, 2, 3]
```

 复制代码

2. 旋转矩阵:

- `matrix = list(zip(*matrix))[::-1]` 这一行的作用是转置矩阵并反转每一列。
- 先转置矩阵：

```
python
list(zip(*matrix)) # 结果是 [(4, 7), (5, 8), (6, 9)]
```

 复制代码

- 然后反转每一列（即 `[::-1]`）：

```
python
[(6, 9), (5, 8), (4, 7)]
```

 复制代码

- 转换为列表后，`matrix` 更新为：

```
python
[
    [6, 9],
    [5, 8],
    [4, 7]
]
```

 复制代码

第二次循环

1. 取出第一行并删除:

- `matrix.pop(0)` 会返回 `[6, 9]` 并删除它。
- 更新后的 `matrix` 变为:

```
python
[
    [5, 8],
    [4, 7]
]
```

 复制代码

- 更新后的 `result` 变为:

```
python
[1, 2, 3, 6, 9]
```

 复制代码

2. 旋转矩阵:

- `matrix = list(zip(*matrix))[::-1]` 这一行的作用是再次转置矩阵并反转每一列。
- 先转置矩阵:

```
python
list(zip(*matrix)) # 结果是 [(5, 4), (8, 7)]
```

 复制代码

- 然后反转每一列:

```
python
[(8, 7), (5, 4)]
```

 复制代码

- 转换为列表后, `matrix` 更新为:

```
python
[
    [8, 7],
    [5, 4]
]
```

 复制代码

第三次循环

1. 取出第一行并删除：

- `matrix.pop(0)` 会返回 `[8, 7]` 并删除它。
- 更新后的 `matrix` 变为：

```
python
```

 复制代码

```
[  
    [5, 4]  
]
```

- 更新后的 `result` 变为：

```
python
```

 复制代码

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

2. 旋转矩阵：

- `matrix = list(zip(*matrix))[::-1]` 这一行的作用是再次转置矩阵并反转每一列。
- 先转置矩阵：

```
python
```

 复制代码

```
list(zip(*matrix)) # 结果是 [(5,)]
```

- 然后反转：

```
python
```

 复制代码

```
[(5,)]
```

- 转换为列表后，`matrix` 更新为：

```
python
```

 复制代码

```
[  
    [5]  
]
```

第四次循环

1. 取出第一行并删除:

- `matrix.pop(0)` 会返回 `[5]` 并删除它。
- 更新后的 `matrix` 变为：

```
python
```

```
[]
```

 复制代码

- 更新后的 `result` 变为：

```
python
```

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

 复制代码

结束循环

- 此时 `matrix` 已经为空，所以 `while matrix:` 条件不再满足，循环结束。
- 最终结果是：

```
python
```

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

 复制代码



`zip()` 函数会按列组合这些行中的元素，返回一个由元组组成的迭代器，每个元组包含对应位置的元素。具体来说，`zip()` 会将 **第 1 列**（即每个子列表的第一个元素）组合在一起，**第 2 列**（即每个子列表的第二个元素）组合在一起，依此类推。

对于 `zip([1, 2, 3], [4, 5, 6], [7, 8, 9])`，`zip()` 会返回：

```
python
```

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

 复制代码

这些元组是 **按列** 组合的，即每个元组代表矩阵的一个新“行”。

在 `list(zip(*matrix))[::-1]` 这一行代码中，操作顺序是先做 `zip(*matrix)`，然后做 `list()`，最后才做 `[::-1]`。

具体来说，顺序是：

1. `zip(*matrix)`:

这个操作首先会对 `matrix` 进行解包，`*matrix` 代表将 `matrix` 里的每一行作为独立的参数传递给 `zip()` 函数。`zip()` 会将这些行元素按列进行组合，生成一个新的“元组”。

比如：

```
python
```

复制代码

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

调用 `zip(*matrix)` 后，会得到以下的元组：

```
python
```

复制代码

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

2. `list(zip(*matrix))`:

`list()` 将 `zip(*matrix)` 生成的“元组”结果转换为列表，因此变成：

```
python
```

复制代码

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

3. `[::-1]`:

最后，`[::-1]` 对列表进行反转，即按列顺序反转列表中的每一行。所以反转后的列表为：

```
python
```

复制代码

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

总结

- `zip(*matrix)`：把矩阵的行“解包”并按列组合成元组。
- `list(zip(*matrix))`：将组合后的元组转换为列表。
- `[::-1]`：对列表进行反转（反转行的顺序）。

这样，矩阵就“旋转”了 90 度。



3. 为什么不是 `[:, :, -1]` 来反转序列?

`[::-1]` 和 `[:, :, -1]` 操作的主要区别在于：

- `[::-1]` 是针对 **单维数组** (如列表、字符串) 进行反向操作，它将序列中的所有元素按相反顺序排列。
- `[:, :, -1]` 是对 **多维数组** 进行切片操作，特别是 **选择最后一层 (第三维)**。

`[::-1]` 适用于所有类型的序列 (如列表、字符串)，将其反转；而 `[:, :, -1]` 则是针对 **多维数组** 的切片，它选取的是最后一个元素，而不是反向排列整个序列。所以，`[::-1]` 与 `[:, :, -1]` 不能互换使用。

总结

- `[::-1]`：适用于任何可切片的序列 (如列表、字符串等)，表示反向切片，将元素顺序反转。
 - `[:, :, -1]`：适用于 **多维数组** (如 NumPy 数组)，表示取最后一个维度的元素。它不是反转操作，而是选择最后一列、最后一层的数据。
-
- `matrix.pop(0)` 用来获取并删除矩阵的 **第一行**，而接下来的操作会继续处理剩余的部分。
 - `matrix.pop()` 在其他场景下可以用来删除列表的 **最后一个元素**，但在这个问题中，通常不是我们需要的操作，因为我们需要的是逐行访问矩阵。

第 48 题 旋转图像

48. 旋转图像

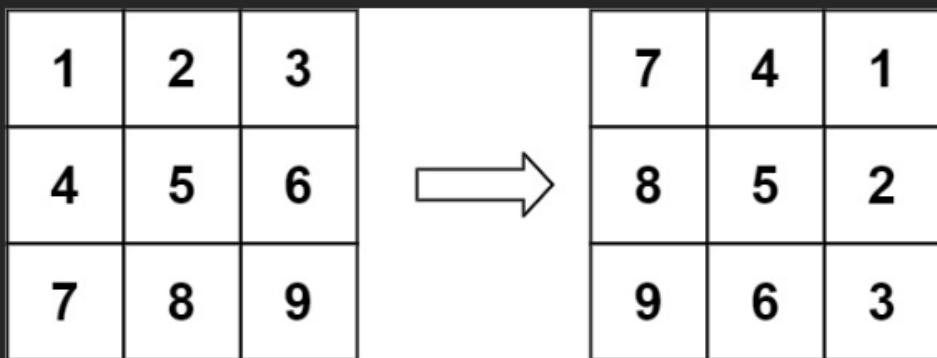
已解答 

中等  相关标签  相关企业 

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：



1	2	3
4	5	6
7	8	9

→

7	4	1
8	5	2
9	6	3

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

输出: `[[7,4,1],[8,5,2],[9,6,3]]`

Python3  智能模式

```
1 class Solution:
2     def rotate(self, matrix: List[List[int]]) -> None:
3         """
4             Do not return anything, modify matrix in-place instead.
5             """
6         matrix[:]=[list(tup)[::-1] for tup in zip(*matrix)]
```

操作步骤

1. `zip(*matrix)`

- `*matrix` 是解包操作，表示将 `matrix` 的每一行作为参数传递给 `zip`。
- `zip` 函数会将矩阵的每一列组合成元组，效果是**将矩阵转置**。

例子：

python

复制代码

```
zip(*matrix) # 对上述 matrix 操作后结果是:  
# [  
#   (1, 4, 7),  
#   (2, 5, 8),  
#   (3, 6, 9)  
# ]
```

2. `[list(tup)[::-1] for tup in zip(*matrix)]`

- 对 `zip(*matrix)` 生成的每个元组 `tup`：
 1. 使用 `list(tup)` 将元组转换成列表。
 2. 使用 `[::-1]` 对列表进行反转。
- 这一步完成了将矩阵的列反转。

例子：

python

复制代码

```
[list(tup)[::-1] for tup in zip(*matrix)]  
# [  
#   [7, 4, 1],  
#   [8, 5, 2],  
#   [9, 6, 3]  
# ]
```

3. `matrix[:] = [...]`

- `matrix[:]` 表示对原矩阵的内容进行**原地替换**。
- 将 `[list(tup)[::-1] for tup in zip(*matrix)]` 的结果赋值给 `matrix`。

为什么 `matrix[:]` 是原地替换？

1. 列表切片操作的特点

`matrix[:]` 表示对列表 `matrix` 的所有元素进行切片，并返回这些元素的视图（不是新的列表）。换句话说：

- `matrix[:]` 是对 `matrix` 的直接引用，不是重新创建一个新的列表。
- 对 `matrix[:]` 的赋值会改变原列表的内容，而不是生成新的列表。

例如：

```
python
```

复制代码

```
matrix = [[1, 2], [3, 4]]  
  
# 原地替换内容  
matrix[:] = [[5, 6], [7, 8]]  
  
print(matrix)  
# 输出: [[5, 6], [7, 8]]
```

2. 与直接赋值的区别

如果用 `matrix = ...`，就会创建一个新的列表对象并将其赋值给 `matrix`，而不会修改原矩阵的内容。

例子：

```
python
```

复制代码

```
matrix = [[1, 2], [3, 4]]  
  
# 直接赋值给 matrix  
matrix = [[5, 6], [7, 8]]  
  
# 这是一个新的列表，旧的列表内容不受影响  
print(matrix)  
# 输出: [[5, 6], [7, 8]]
```

如果有其他变量引用 `matrix` 的原列表时，`matrix = ...` 会导致其他变量仍然引用旧的列表，而不会受到新赋值的影响。

第 240 题 搜索二维矩阵 II

240. 搜索二维矩阵 II

中等

相关标签

相关企业

A₂

编写一个高效的算法来搜索 $m \times n$ 矩阵 `matrix` 中的一个目标值 `target`。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例 1：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

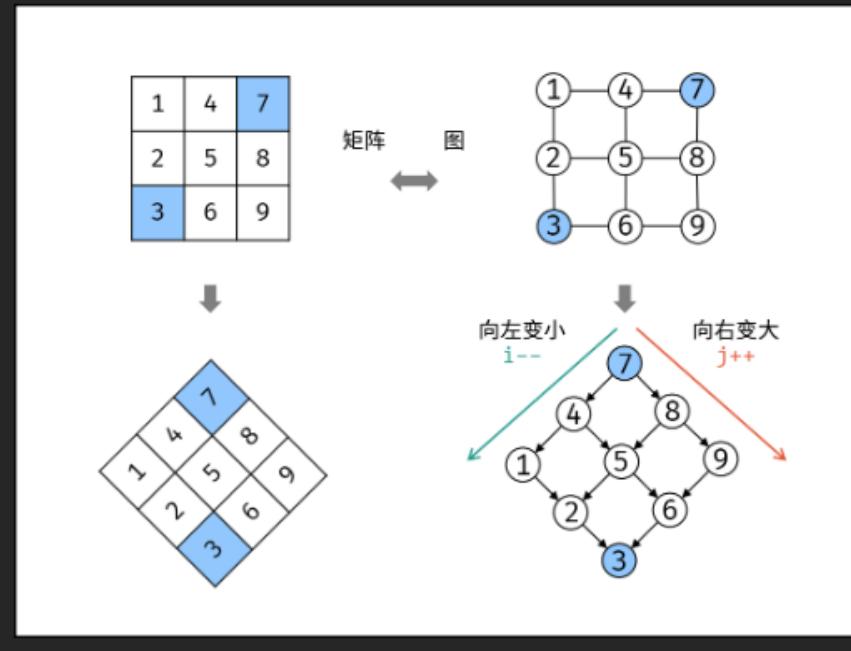
输入: `matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]`, `target = 5`

输出: `true`

解题思路：

若使用暴力法遍历矩阵 `matrix`，则时间复杂度为 $O(NM)$ 。暴力法未利用矩阵“**从上到下递增、从左到右递增**”的特点，显然不是最优解法。

如下图所示，我们将矩阵逆时针旋转 45° ，并将其转化为图形形式，发现其类似于**二叉搜索树**，即对于每个元素，其左分支元素更小、右分支元素更大。因此，通过从“根节点”开始搜索，遇到比 `target` 大的元素就向左，反之向右，即可找到目标值 `target`。



“根节点”对应的是矩阵的“左下角”和“右上角”元素，本文称之为**标志数**，以 `matrix` 中的**左下角元素**为标志数 `flag`，则有：

1. 若 `flag > target`，则 `target` 一定在 `flag` 所在**行的上方**，即 `flag` 所在行可被消去。
2. 若 `flag < target`，则 `target` 一定在 `flag` 所在**列的右方**，即 `flag` 所在列可被消去。

算法流程：

1. 从矩阵 `matrix` 左下角元素（索引设为 (i, j) ）开始遍历，并与目标值对比：

- 当 `matrix[i][j] > target` 时，执行 `i--`，即消去第 i 行元素。
- 当 `matrix[i][j] < target` 时，执行 `j++`，即消去第 j 列元素。
- 当 `matrix[i][j] = target` 时，返回 `true`，代表找到目标值。

2. 若行索引或列索引越界，则代表矩阵中无目标值，返回 `false`。

每轮 i 或 j 移动后，相当于生成了“消去一行（列）的新矩阵”，索引 (i, j) 指向新矩阵的左下角元素（标志数），因此可重复使用以上性质消去行（列）。

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

`target = 8`

`flag = 18`

`i = 4, j = 0`

`if flag > target`

`i--, 消去行`

Python | Java | C++

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        i, j = len(matrix) - 1, 0
        while i >= 0 and j < len(matrix[0]):
            if matrix[i][j] > target: i -= 1
            elif matrix[i][j] < target: j += 1
            else: return True
        return False
```

七、链表

第 141 题 环形链表

题目描述

141. 环形链表

给你一个链表的头节点 head，判断链表中是否有环。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，仅仅是用于标记链表的实际情况。

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

```
Python3
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        if head == None:
            return False
        slow = head
        fast = head.next
        while fast != None and fast.next != None:
            if slow == fast:
                return True
            slow = slow.next
            fast = fast.next.next
        return False
```

思路 1：哈希表

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

思路 1：代码

```
1 class Solution:
2     def hasCycle(self, head: ListNode) -> bool:
3         nodeset = set()
4
5         while head:
6             if head in nodeset:
7                 return True
8             nodeset.add(head)
9             head = head.next
10
11         return False
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

思路 2：快慢指针 (Floyd 判圈算法)

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

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

```
Python3 v 智能模式
1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, x):
4 #         self.val = x
5 #         self.next = None
6
7 class Solution:
8     def hasCycle(self, head: Optional[ListNode]) -> bool:
9         slow, fast = head, head
10        while fast is not None and fast.next is not None:
11            fast = fast.next.next
12            slow = slow.next
13            if fast == slow:
14                return True
15        return False
```



```
Python3 v 智能模式
1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, x):
4 #         self.val = x
5 #         self.next = None
6
7 class Solution:
8     def hasCycle(self, head: Optional[ListNode]) -> bool:
9         slow = head
10        fast = head
11        if head is None:
12            return False
13        while fast is not None and fast.next is not None:
14            slow = slow.next
15            fast = fast.next.next
16            if slow == fast:
17                return True
18        return False
```

布尔值的规则

在 Python 中，以下值会被视为 **False**：

- `None`
- `False`
- `0` (包括整数、浮点数等数值类型的 0)
- 空的序列或集合 (如 `[]`, `()`, `{}`, `""`, `set()`, `dict()`)
- 自定义对象的 `__bool__()` 或 `__len__()` 方法返回 `False` 或 `0`

而其他所有值都被视为 **True**。

第 160 题 相交链表

160. 相交链表

简单

相关标签

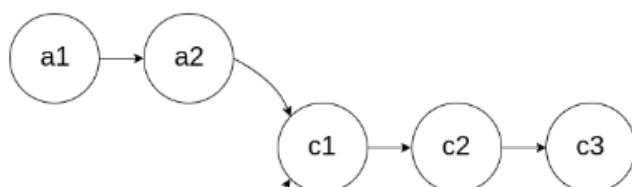
相关企业

A+

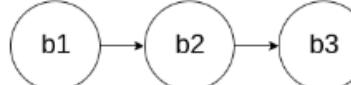
给你两个单链表的头节点 `headA` 和 `headB`，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 `null`。

图示两个链表在节点 `c1` 开始相交：

A:



B:



题目数据 **保证** 整个链式结构中不存在环。

注意，函数返回结果后，链表必须 **保持其原始结构**。

自定义评测：

评测系统 的输入如下（你设计的程序 **不适用** 此输入）：

```
pA=headA  
pB=headB  
while pA!=pB:  
    pA=pA.next if pA.next != None else pB  
    pB=pB.next if pB.next != None else pA  
return pA
```



这段代码有问题，无法正确实现两个链表的交点检测。主要问题在于以下两点：

1. `pA.next` 和 `pB.next` 的判断错误：

- 使用 `pA.next` 或 `pB.next` 判断是否到达链表尾部不合适，因为当指针到达 `None` 时，`pA.next` 会导致报错。

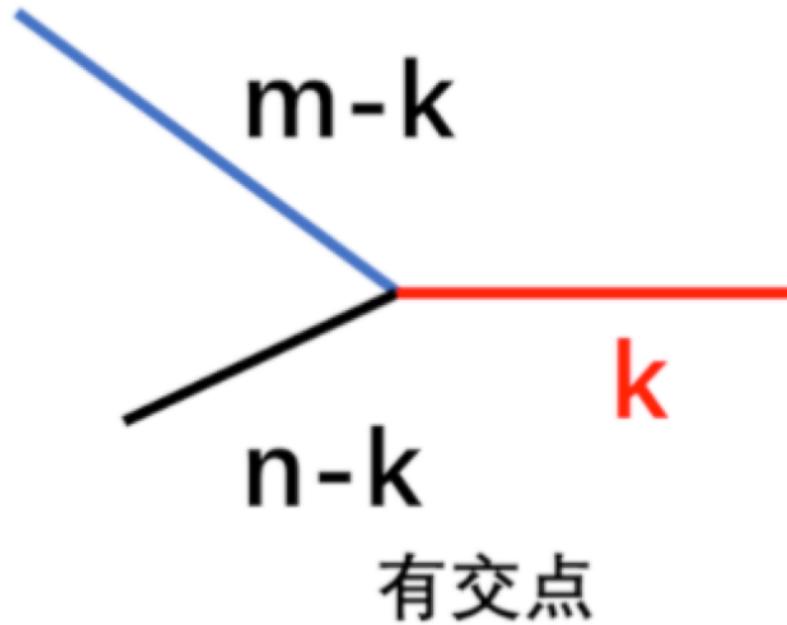
- 正确的做法是直接判断 `pA` 和 `pB` 是否为 `None`，而不是判断 `pA.next` 或 `pB.next`。

2. 逻辑错误：

- 在 `pA` 为 `None` 时，代码会试图访问 `pA.next`，这会导致 `AttributeError`。
- 即便修复上述错误，逻辑仍然不符合算法要求。

思路 1：双指针

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

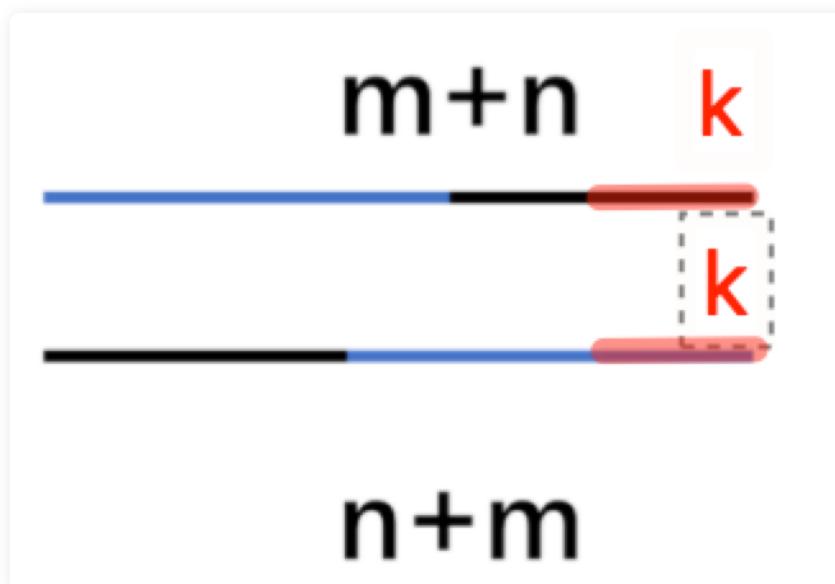


现在问题是如何找到 $m - k$ 或者 $n - k$ 的位置。

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

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

否则走到两个链表末尾，返回 `None`。



思路 1：代码

```
1 class Solution:
2     def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
3         if headA == None or headB == None:
4             return None
5         pA = headA
6         pB = headB
7         while pA != pB:
8             pA = pA.next if pA != None else headB
9             pB = pB.next if pB != None else headA
10        return pA
```

思路 1：复杂度分析

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

第 206 题 反转链表

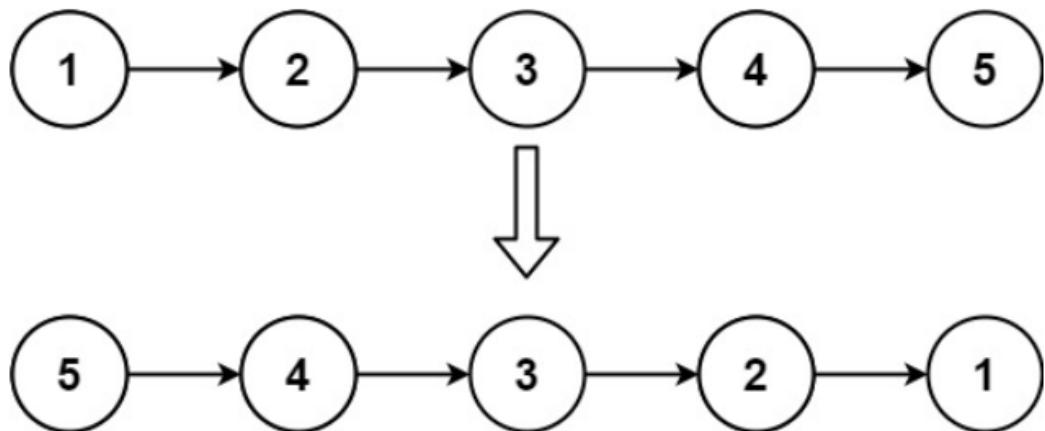
206. 反转链表

已解答

[简单](#) [相关标签](#) [相关企业](#) [A+](#)

给你单链表的头节点 `head`，请你反转链表，并返回反转后的链表。

示例 1：



输入: `head = [1, 2, 3, 4, 5]`

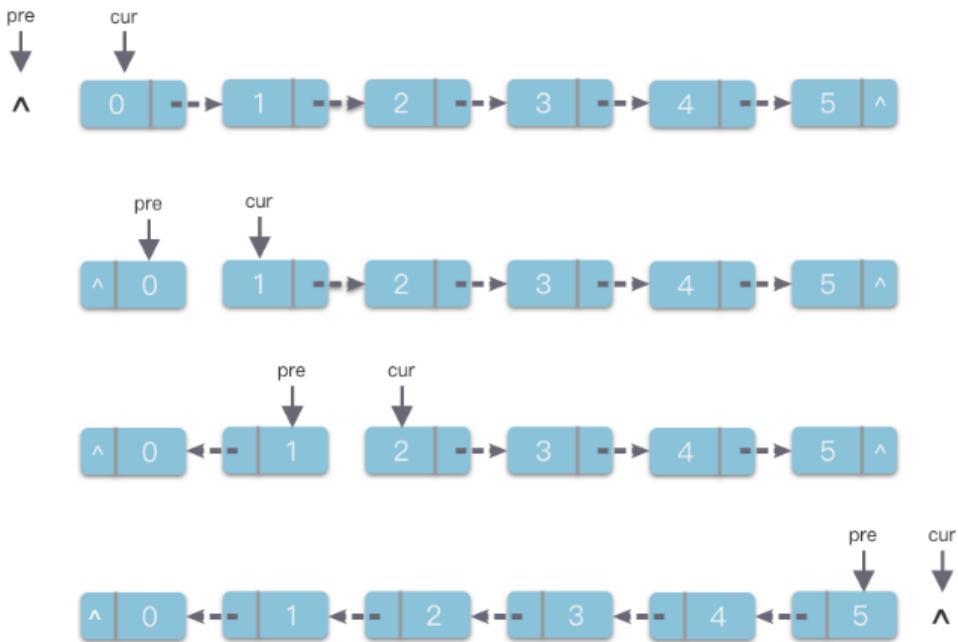
输出: `[5, 4, 3, 2, 1]`

思路 1：迭代

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

使用迭代法反转链表的示意图如下所示：

反转链表（迭代）



思路 1：代码

```
1 class Solution:
2     def reverseList(self, head: ListNode) -> ListNode:
3         pre = None
4         cur = head
5         while cur != None:
6             next = cur.next
7             cur.next = pre
8             pre = cur
9             cur = next
10        return pre
```

py

思路 1：复杂度分析

- 时间复杂度: $O(n)$ 。
- 空间复杂度: $O(1)$ 。

python

复制代码

```
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        pre = None # 用于保存前一个节点, 初始化为 None
        cur = head # 当前节点指向链表的头节点

        while cur != None: # 当当前节点不为空时, 持续反转
            next = cur.next # 临时保存当前节点的下一个节点
            cur.next = pre # 当前节点的指针反向, 指向前一个节点
            pre = cur # 更新前一个节点为当前节点
            cur = next # 移动到下一个节点
        return pre # 返回反转后的新头节点 (原链表的尾节点)
```

示例

假设原始链表是：

rust

复制代码

```
1 -> 2 -> 3 -> 4 -> 5
```

- **初始化：**

- `pre = None`
- `cur = head` (指向节点 1)

- **第一次循环：**

- `next = cur.next` (保存节点 2)
- `cur.next = pre` (节点 1 的 `next` 变为 `None`)
- `pre = cur` (`pre` 移动到节点 1)
- `cur = next` (`cur` 移动到节点 2)

当前链表变为：

rust

复制代码

```
1 <- 2 -> 3 -> 4 -> 5
```

- 第二次循环：

- `next = cur.next` (保存节点 3)
- `cur.next = pre` (节点 2 的 `next` 指向节点 1)
- `pre = cur` (pre 移动到节点 2)
- `cur = next` (cur 移动到节点 3)

当前链表变为：

rust

复制代码

1 <- 2 <- 3 -> 4 -> 5

- 继续循环，直到 `cur` 变为 `None`，最终链表反转为：

rust

复制代码

5 -> 4 -> 3 -> 2 -> 1

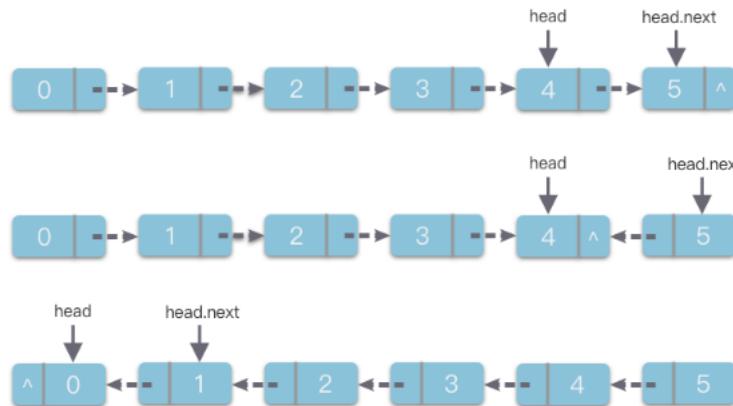
思路 2：递归

具体做法如下：

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

使用递归法反转链表的示意图如下所示：

反转链表（递归）



思路 2：代码

```
1 class Solution:
2     def reverseList(self, head: ListNode) -> ListNode:
3         if head == None or head.next == None:
4             return head
5         new_head = self.reverseList(head.next)
6         head.next.next = head
7         head.next = None
8         return new_head
```

思路 2：复杂度分析

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

递归过程说明

假设链表为 `1 -> 2 -> 3 -> 4 -> 5`，递归过程会像这样进行：

1. 递归到最深的节点：

- `reverseList(1)` 调用 `reverseList(2)`
- `reverseList(2)` 调用 `reverseList(3)`
- `reverseList(3)` 调用 `reverseList(4)`
- `reverseList(4)` 调用 `reverseList(5)`
- `reverseList(5)` 发现 `head.next == None`，返回 `5`，即链表的最后一个节点。

2. 开始返回：

- 在返回的过程中，逐步进行链表的反转。

3. 反转每对节点：

- `head = 4, head.next = 5`，我们让 `head.next.next = head`，即 `5.next = 4`，然后将 `4.next = None`，反转完成这一对节点。返回新的链表头 `5 -> 4`。
- `head = 3, head.next = 4`，我们让 `head.next.next = head`，即 `4.next = 3`，然后将 `3.next = None`，反转完成这一对节点。返回新的链表头 `5 -> 4 -> 3`。
- `head = 2, head.next = 3`，我们让 `head.next.next = head`，即 `3.next = 2`，然后将 `2.next = None`，反转完成这一对节点。返回新的链表头 `5 -> 4 -> 3 -> 2`。
- `head = 1, head.next = 2`，我们让 `head.next.next = head`，即 `2.next = 1`，然后将 `1.next = None`，反转完成这一对节点。返回新的链表头 `5 -> 4 -> 3 -> 2 -> 1`。

第 92 题 反转链表 II

92. 反转链表 II

中等

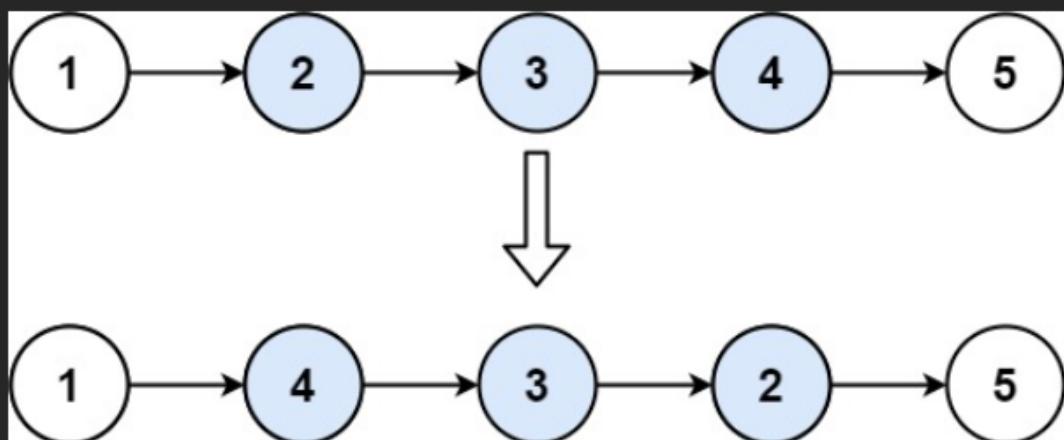
相关标签

 相关企业

A文

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]

第 234 题 回文链表

234. 回文链表

简单

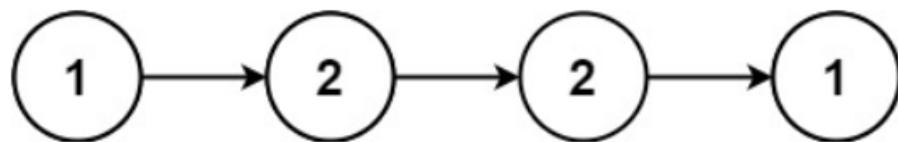
相关标签

相关企业

A+

给你一个单链表的头节点 `head`，请你判断该链表是否为回文链表。如果是，返回 `true`；否则，返回 `false`。

示例 1：



输入: `head = [1,2,2,1]`

输出: `true`

Python3 ▾ 智能模式

```
1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, val=0, next=None):
4 #         self.val = val
5 #         self.next = next
6 class Solution:
7     def isPalindrome(self, head: Optional[ListNode]) -> bool:
8
```

思路 1：利用数组 + 双指针

1. 利用数组，将链表元素依次存入。
2. 然后再使用两个指针，一个指向数组开始位置，一个指向数组结束位置。
3. 依次判断首尾对应元素是否相等，如果都相等，则为回文链表。如果不相等，则不是回文链表。

思路 1：代码

```
1 class Solution:
2     def isPalindrome(self, head: ListNode) -> bool:
3         nodes = []
4         p1 = head
5         while p1 != None:
6             nodes.append(p1.val)
7             p1 = p1.next
8         return nodes == nodes[::-1]
```

py

思路 1：复杂度分析

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

第 21 题 合并两个有序链表

21. 合并两个有序链表

简单

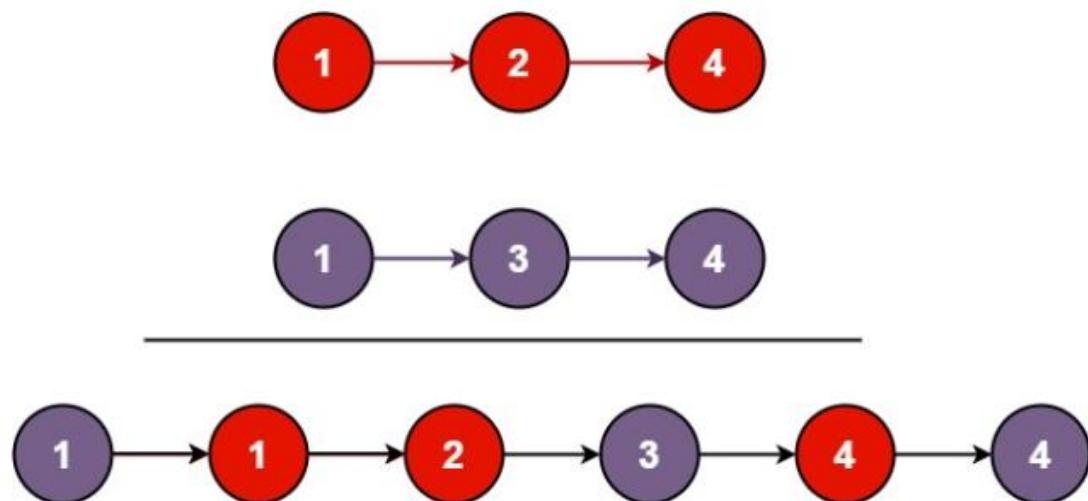
相关标签

相关企业

A+

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：



Python3 ▼ 智能模式

三 口 ⌂

```
1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, val=0, next=None):
4 #         self.val = val
5 #         self.next = next
6 class Solution:
7     def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
8         pass
```

思路 1：归并排序

利用归并排序的思想，具体步骤如下：

1. 使用哑节点 `dummy_head` 构造一个头节点，并使用 `curr` 指向 `dummy_head` 用于遍历。
2. 然后判断 `list1` 和 `list2` 头节点的值，将较小的头节点加入到合并后的链表中。并向后移动该链表的头节点指针。
3. 然后重复上一步操作，直到两个链表中出现链表为空的情况。
4. 将剩余链表链接到合并后的链表中。
5. 将哑节点 `dummy_head` 的下一个链节点 `dummy_head.next` 作为合并后有序链表的头节点返回。

思路 1：代码

```
1  class Solution:
2      def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[L
3          dummy_head = ListNode(-1)
4
5          curr = dummy_head
6          while list1 and list2:
7              if list1.val <= list2.val:
8                  curr.next = list1
9                  list1 = list1.next
10             else:
11                 curr.next = list2
12                 list2 = list2.next
13             curr = curr.next
14
15         curr.next = list1 if list1 is not None else list2
16
17     return dummy_head.next
```

思路 1：复杂度分析

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

第 142 题 环形链表 II

142. 环形链表 II

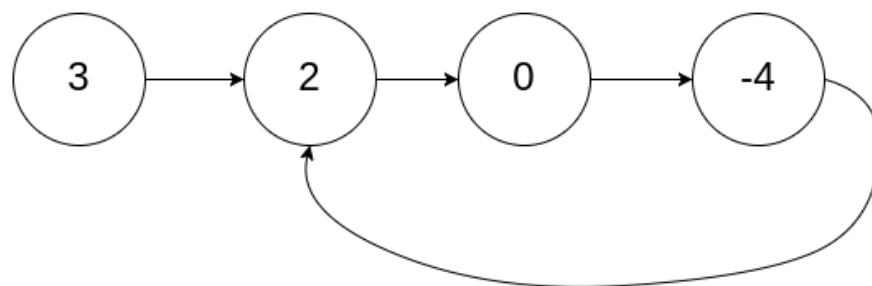
中等 相关标签 相关企业 Ax

给定一个链表的头节点 `head`，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 -1，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

不允许修改 链表。

示例 1：



输入：`head = [3, 2, 0, -4], pos = 1`

输出：返回索引为 1 的链表节点

解释：链表中有一个环，其尾部连接到第二个节点。

Python3 ▼ 智能模式

```
1 # Definition for singly-linked list.
2 # class ListNode:
3 #     def __init__(self, x):
4 #         self.val = x
5 #         self.next = None
6
7 class Solution:
8     def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
9
```

思路 1：快慢指针（Floyd 判圈算法）

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

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

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

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

思路 1：代码

```
1 class Solution:
2     def detectCycle(self, head: ListNode) -> ListNode:
3         fast, slow = head, head
4         while True:
5             if not fast or not fast.next:
6                 return None
7             fast = fast.next.next
8             slow = slow.next
9             if fast == slow:
10                 break
11
12         ans = head
13         while ans != slow:
14             ans, slow = ans.next, slow.next
15         return ans
```

思路 1：复杂度分析

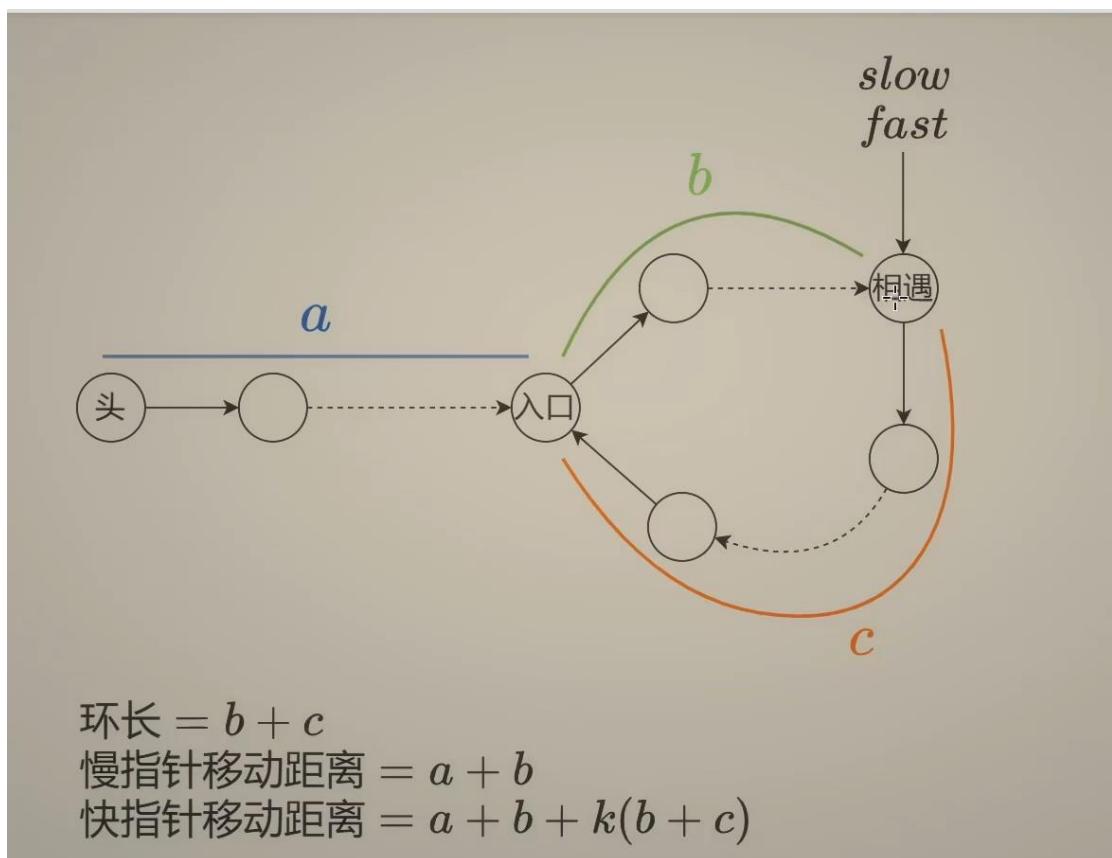
- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(1)$ 。

```

5     self.next = None
6
7 class Solution:
8     def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
9         slow, fast = head, head
10        while fast != None and fast.next != None:
11            slow = slow.next
12            fast = fast.next.next
13            if slow == fast:
14                break
15        else:
16            return None
17        ans = head
18        while ans != slow:
19            ans = ans.next
20            slow = slow.next
21        return ans

```

思路二：



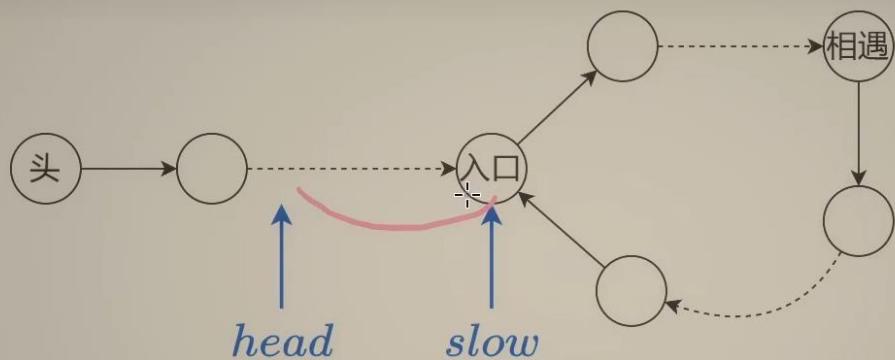
快指针移动距离是慢指针的两倍

$$\Rightarrow 2(a + b) = a + b + k(b + c)$$

$$\Rightarrow 2a + 2b = \underline{a + b + b + c} + (k - 1)(b + c)$$

$$\Rightarrow \cancel{a - c} = (k - 1)(b + c)$$

这意味着什么？



slow 从相遇点出发

head 从头节点出发

走 c 步后, *slow* 在入口

head 到入口的距离也恰好是环长的倍数

继续走, 两者必然会在入口相遇

```
6
7 # Java/C++/Go 等语言的实现, 见视频简介
8 class Solution:
9     def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
10         slow = head
11         fast = head
12         while fast and fast.next:
13             slow = slow.next
14             fast = fast.next.next
15             if fast is slow:
16                 while slow is not head:
17                     slow = slow.next
18                     head = head.next
19                 return slow
20         return None
```

第 2 题 两数相加

2. 两数相加

已解答 

中等

相关标签

相关企业

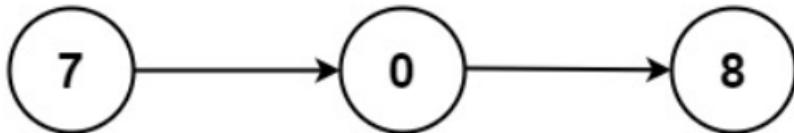
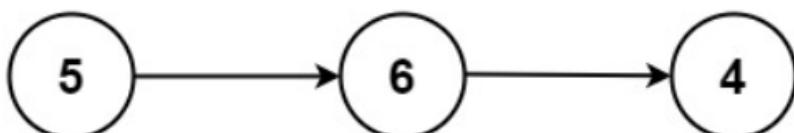
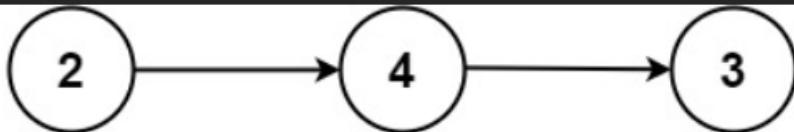
A+

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: $342 + 465 = 807$.

```

1  class Solution:
2      def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
3          head = curr = ListNode(0)
4          carry = 0
5          while l1 or l2 or carry:
6              if l1:
7                  num1 = l1.val
8                  l1 = l1.next
9              else:
10                 num1 = 0
11             if l2:
12                 num2 = l2.val
13                 l2 = l2.next
14             else:
15                 num2 = 0
16
17             sum = num1 + num2 + carry
18             carry = sum // 10
19
20             curr.next = ListNode(sum % 10)
21             curr = curr.next
22
23     return head.next

```

思路 1：复杂度分析

- 时间复杂度： $O(\max(m, n))$ 。其中， m 和 n 分别是链表 `l1` 和 `l2` 的长度。
- 空间复杂度： $O(1)$ 。

1. 为什么返回 `head.next`？

`head` 是链表的哑节点 (dummy node)，它的作用只是帮助初始化和操作链表的头部。在构建链表时，实际有效的结果链表从 `head.next` 开始。

python

复制代码

```
head = curr = ListNode(0)
```

这一步创建了一个哑节点，`head` 和 `curr` 都指向这个哑节点。但随着我们在循环中操作 `curr`：

python

复制代码

```
curr.next = ListNode(sum % 10)
curr = curr.next
```

- `curr` 是一个动态指针，始终指向当前链表的尾部，用来新增节点。
- `head` 一直保持不动，指向整个链表的起始位置。

最终，结果链表是从 `head.next` 开始的，因为 `head` 自身是一个哑节点，值是 `0`，不是我们计算的结果。

第 19 题 删除链表的倒数第 N 个节点

19. 删除链表的倒数第 N 个结点

中等

相关标签

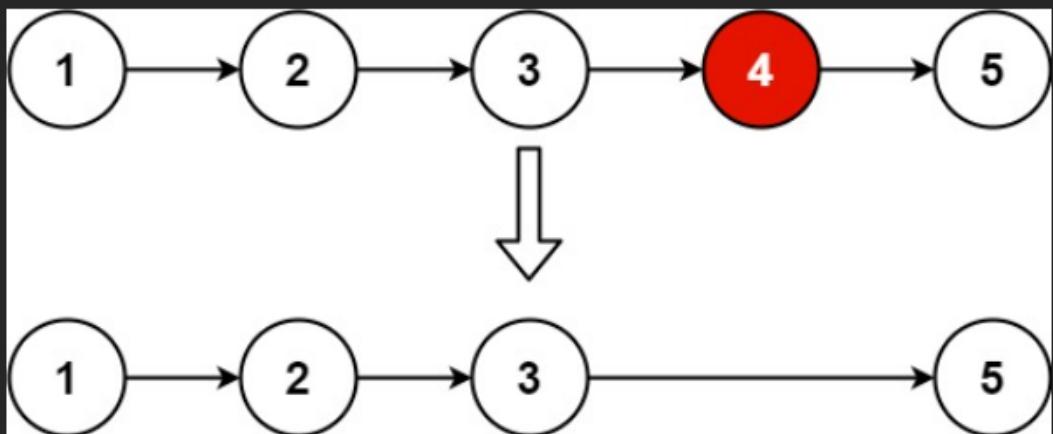
相关企业

提示

A_x

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: []

```
6 class Solution:
7     def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
8         # 由于可能会删除链表头部, 用哨兵节点简化代码
9         slow=fast=dummy=ListNode(next=head)
10        for _ in range(n):
11            fast=fast.next # 快指针先向右走 n 步
12        while fast.next:
13            slow=slow.next # 快慢指针一起走
14            fast=fast.next
15        slow.next=slow.next.next # 慢指针的下一个节点就是倒数第 n 个节点
16        return dummy.next
```

区别总结:

1. `while fast.next::`

- 判断的是 `fast` 指向的节点是否有下一个节点 (即 `fast.next` 是否不为 `None`)。
- 如果 `fast` 指向的是链表的倒数第二个节点, `fast.next` 就是链表的最后一个节点, 此时 `while fast.next:` 仍然会继续执行, 直到 `fast` 移动到最后一个节点, 才会退出。
- 适用于需要保证在操作过程中 `fast` 至少指向一个有效节点并且有下一个节点可访问的情况。

2. `while fast::`

- 判断的是 `fast` 是否指向 `None`。
- 如果 `fast` 在循环开始时指向 `None`, 循环就不会执行 (直接跳过)。如果 `fast` 是有效的节点 (即不为 `None`) , 则循环继续。
- 适用于你只关心 `fast` 是否为 `None` , 而不关心其后面是否还有节点的情况。

第 24 题 两两交换链表中的节点

24. 两两交换链表中的节点

中等

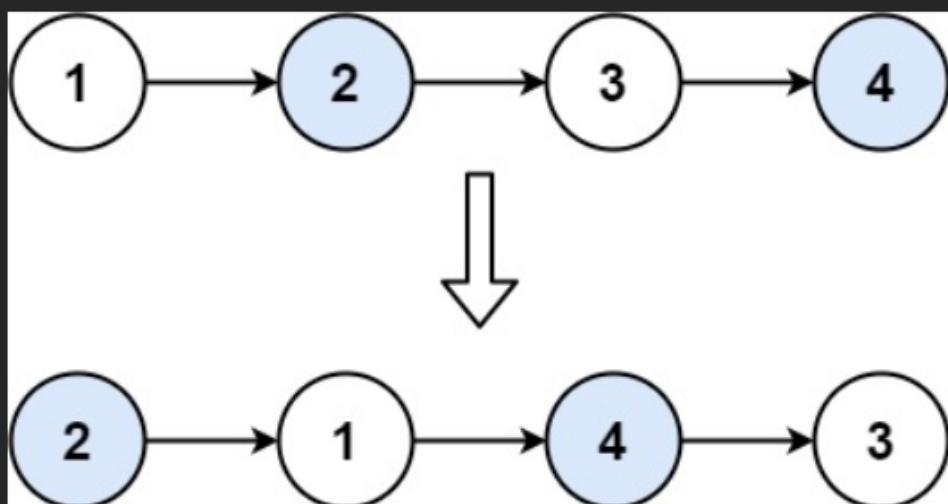
相关标签

相关企业

A+

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

示例 1：



输入: head = [1,2,3,4]

输出: [2,1,4,3]

思路 1：迭代

1. 创建一个哑节点 `new_head`，令 `new_head.next = head`。
2. 遍历链表，并判断当前链表后两位节点是否为空。如果后两个节点不为空，则使用三个指针：`curr` 指向当前节点，`node1` 指向下一个节点，`node2` 指向下面第二个节点。
3. 将 `curr` 指向 `node2`，`node1` 指向 `node2` 后边的节点，`node2` 指向 `node1`。则节点关系由 `curr → node1 → node2` 变为了 `curr → node2 → node1`。
This step is highlighted with a red arrow pointing to the `node2` node in the original text.
4. 依次类推，最终返回哑节点连接的后一个节点。

思路 1：代码

```
1 class Solution:
2     def swapPairs(self, head: ListNode) -> ListNode:
3         new_head = ListNode(0)
4         new_head.next = head
5         curr = new_head
6         while curr.next and curr.next.next:
7             node1 = curr.next
8             node2 = curr.next.next
9             curr.next = node2
10            node1.next = node2.next
11            node2.next = node1
12            curr = node1
13        return new_head.next
```



思路 1：复杂度分析

- 时间复杂度： $O(n)$ ，其中 n 为链表的节点数量。
- 空间复杂度： $O(n)$ 。

• 交换步骤：

- `curr.next = node2`：将 `curr.next` 指向 `node2`，即让 `curr` 的 `next` 指向 `2`，这时链表变为：

rust

复制代码

```
curr -> 0 -> 2 -> 1 -> 3 -> 4 -> 5
```

- `node1.next = node2.next`：将 `node1.next`（即 `1` 的 `next`）指向 `node2.next`（即 `3`）。这样 `1` 就指向了 `3`，链表变为：

rust

复制代码

```
curr -> 0 -> 2 -> 1 -> 3 -> 4 -> 5
```

- `node2.next = node1`：将 `node2.next`（即 `2` 的 `next`）指向 `node1`（即 `1`）。这样 `2` 就指向了 `1`，链表变为：

rust

复制代码

```
curr -> 0 -> 2 -> 1 -> 3 -> 4 -> 5
```

- `curr = node1`：更新 `curr` 为 `node1`（即 `1`），准备处理下一对节点。

2. 分析交换步骤的核心:

交换相邻节点的关键是改变 `curr`、`node1` 和 `node2` 的指向。你可以分为以下几个小步骤：

步骤 1：更新 `curr` 的 `next` 指向 `node2`。

- `curr.next = node2`：将 `curr` 的 `next` 指向 `node2`，将 `node2` 移到前面。
- 这就意味着交换后的链表开始部分变成了 `node2 -> node1`。

步骤 2：更新 `node1` 的 `next` 指向 `node2.next`。

- `node1.next = node2.next`：这一步确保 `node1` 不再指向 `node2`，而是指向 `node2` 后面的节点。如果 `node2` 是链表的最后一个节点，则 `node1` 的 `next` 会指向 `None`。

步骤 3：更新 `node2` 的 `next` 指向 `node1`。

- `node2.next = node1`：这一步将 `node2` 的 `next` 指向 `node1`，完成了交换。

步骤 4：更新 `curr` 为 `node1`。

- `curr = node1`：因为 `node1` 现在已经是交换后的第一个节点，我们继续遍历下一个相邻节点对，所以 `curr` 移动到 `node1` 继续交换。

3. 如何记住操作：

有一个简单的方法可以帮助你记住链表交换的逻辑，那就是通过 **三个小问题**：

1. 如何交换它们？

你需要更新 `curr` 的 `next`，让它指向第二个节点（`node2`）。这将让 `node2` 移到前面。

2. 如何确保剩下的链表不丢失？

你需要让第一个节点（`node1`）指向 `node2.next`，这样后续的部分才不会丢失。

3. 如何完成交换？

最后，`node2` 需要指向 `node1`，确保它变成第二个节点。

第 25 题 K 个一组翻转链表

25. K 个一组翻转链表

困难

相关标签

相关企业

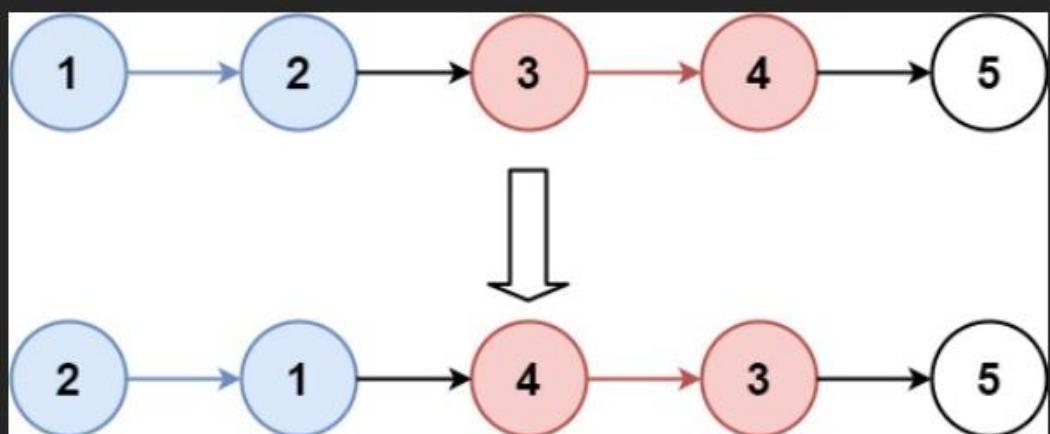
A*

给你链表的头节点 `head`，每 `k` 个节点一组进行翻转，请你返回修改后的链表。

`k` 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 `k` 的整数倍，那么请将最后剩余的节点保持原有顺序。

你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

示例 1：



输入: `head = [1,2,3,4,5], k = 2`

输出: `[2,1,4,3,5]`

138. 随机链表的复制

中等

相关标签

相关企业

提示

A

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 --> Y`。那么在复制链表中对应的两个节点 `x` 和 `y`，同样有 `x.random --> y`。

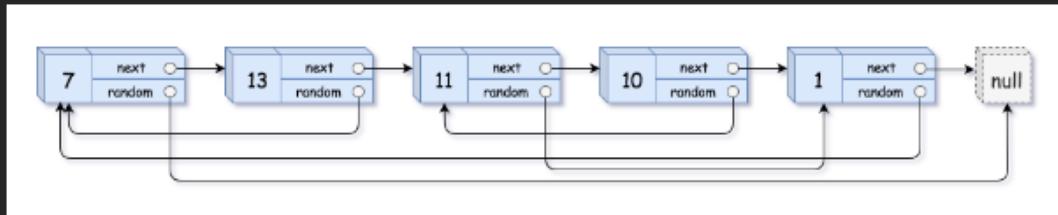
返回复制链表的头节点。

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。

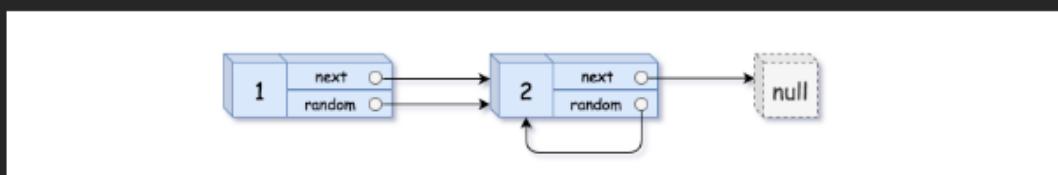
示例 1:



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]]

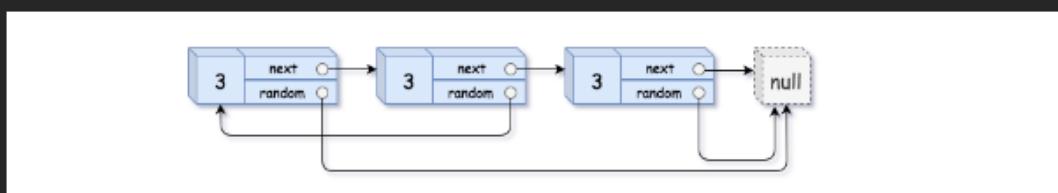
示例 2:



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]]

第 148 题 排序链表

148. 排序链表

中等

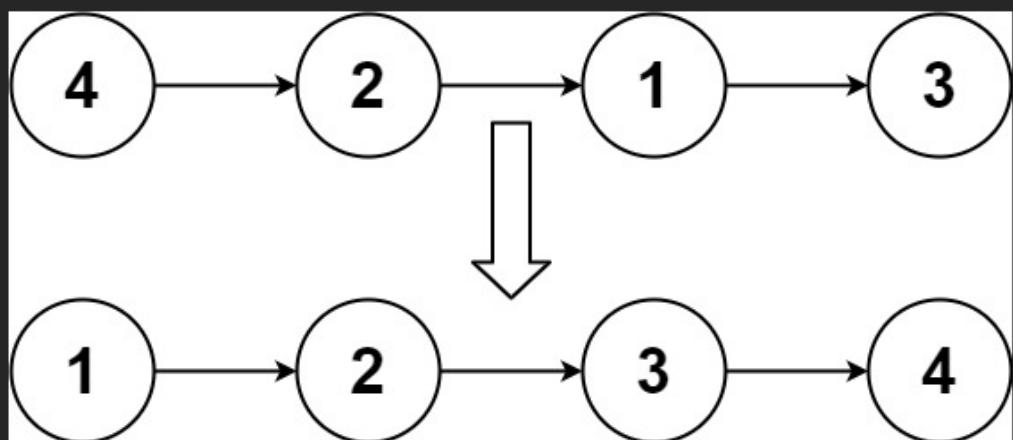
相关标签

相关企业

A+

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]`

23. 合并 K 个升序链表

困难

相关标签

相关企业

A+

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入: lists = [[1,4,5],[1,3,4],[2,6]]

输出: [1,1,2,3,4,4,5,6]

解释: 链表数组如下:

```
[  
    1->4->5,  
    1->3->4,  
    2->6
```

]

将它们合并到一个有序链表中得到。

1->1->2->3->4->4->5->6

示例 2：

输入: lists = []

输出: []

146. LRU 缓存

中等

相关标签

相关企业

A+

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]
```

LRU缓存 (Least Recently Used Cache, 最近最少使用缓存) 是一种常见的缓存管理策略，用于在内存有限的情况下优先淘汰那些最近最少使用的数据，从而为新的数据腾出空间。

LRU缓存的核心思想

LRU缓存基于以下假设：

- 最近使用的数据更有可能再次被使用。
- 较长时间未使用的数据更可能不再被使用。

因此，当缓存容量达到上限时，LRU策略会淘汰最久未使用的数据。

数据结构设计

1. 双向链表：

- 双向链表用于维护缓存中元素的访问顺序。
- 头节点 (`head`) 之后的节点表示最久未使用的缓存，尾节点 (`tail`) 之前的节点表示最近访问的缓存。
- 当访问或插入新元素时，会将该元素移到链表末尾（即最新位置）。
- 删除最久未使用的缓存时，直接删除链表的头节点之后的节点即可。

2. 哈希表：

- 哈希表用于快速查找键对应的缓存节点。
- 键为缓存的 `key`，值为该 `key` 在双向链表中对应的节点（包含 `key` 和 `value`）。

3. 容量控制：

- `capacity` 限定了缓存的大小。
- 如果缓存已满，则删除最久未使用的元素（链表头部的节点）。

```

1  class ListNode:
2      def __init__(self, key=None, value=None):
3          self.key = key
4          self.value = value
5          self.prev = None
6          self.next = None
7
8  class LRUCache:
9      def __init__(self, capacity: int):
10         self.capacity = capacity
11         self.hashmap = {}
12         # 新建两个节点 head 和 tail
13         self.head = ListNode()
14         self.tail = ListNode()
15         # 初始化链表为 head <-> tail
16         self.head.next = self.tail
17         self.tail.prev = self.head

```

```

18     # 因为get与put操作都可能需要将双向链表中的某个节点移到末尾，所以定义一个方法
19     def move_node_to_tail(self, key):
20         # 先将哈希表key指向的节点拎出来，为了简洁起名node
21         #           hashmap[key]                               hashmap[key]
22         #           |                                         |
23         #           V                                         -->
24         # prev <-> node <-> next           pre <-> next ... node
25         node = self.hashmap[key]
26         node.prev.next = node.next
27         node.next.prev = node.prev

```

```

28     # 之后将node插入到尾节点前
29         #           hashmap[key]                               hashmap[key]
30         #           |                                         |
31         #           V                                         -->
32         # prev <-> tail ... node           prev <-> node <-> tail
33         node.prev = self.tail.prev
34         node.next = self.tail
35         self.tail.prev.next = node
36         self.tail.prev = node

```

```
37     def get(self, key: int) -> int:
38         if key in self.hashmap:
39             # 如果已经在链表中了久把它移到末尾（变成最新访问的）
40             self.move_node_to_tail(key)
41             res = self.hashmap.get(key, -1)
42             if res == -1:
43                 return res
44             else:
45                 return res.value
46
47     def put(self, key: int, value: int) -> None:
48         if key in self.hashmap:
49             # 如果key本身已经在哈希表中了就不需要在链表中加入新的节点
50             # 但是需要更新字典该值对应节点的value
51             self.hashmap[key].value = value
52             # 之后将该节点移到末尾
53             self.move_node_to_tail(key)
54         else:
55             if len(self.hashmap) == self.capacity:
56                 # 去掉哈希表对应项
57                 self.hashmap.pop(self.head.next.key)
58                 # 去掉最久没有被访问过的节点，即头节点之后的节点
59                 self.head.next = self.head.next.next
60                 self.head.next.prev = self.head
61             # 如果不在的话就插入到尾节点前
62             new = ListNode(key, value)
63             self.hashmap[key] = new
64             new.prev = self.tail.prev
65             new.next = self.tail
66             self.tail.prev.next = new
67             self.tail.prev = new
```

双向链表节点类

python

复制代码

```
class ListNode:  
    def __init__(self, key=None, value=None):  
        self.key = key  
        self.value = value  
        self.prev = None  
        self.next = None
```

- 每个节点包含三个部分：
 - `key`：缓存的键。
 - `value`：缓存的值。
 - `prev` 和 `next`：指向前后节点的指针，用于维护链表的双向结构。

缓存类 LRUCache

初始化

python

复制代码

```
def __init__(self, capacity: int):  
    self.capacity = capacity  
    self.hashmap = {} # 存储 key 和链表节点的映射关系  
    self.head = ListNode() # 虚拟头节点  
    self.tail = ListNode() # 虚拟尾节点  
    self.head.next = self.tail  
    self.tail.prev = self.head
```

- 创建一个容量为 `capacity` 的 LRU 缓存。
- `hashmap` 用于快速定位缓存中的节点。
- 初始化双向链表时，设置 `head` 和 `tail` 两个虚拟节点，它们不存储数据，只用来标记链表的起点和终点。
- 更新链表指针时，一定要先操作新节点的指针，再修改链表中其他节点的指针。

缓存操作

get 方法

python

复制代码

```
def get(self, key: int) -> int:
    if key in self.hashmap:
        # 将节点移到尾部
        self.move_node_to_tail(key)
    res = self.hashmap.get(key, -1)
    if res == -1:
        return res # 不存在返回 -1
    else:
        return res.value # 返回对应的值
```

1. 检查 `key` 是否在 `hashmap` 中：

- 如果存在，将对应节点移到链表尾部（表示最近访问）。
- 如果不存在，返回 `-1`。

2. 时间复杂度：

- 查找哈希表是 $O(1)$ ，移动节点在双向链表中也是 $O(1)$ 。

为什么最久未使用的节点放在前面？



1. 访问顺序维护

- 每次访问 (`get`) 或更新 (`put`) 某个节点时，会把这个节点移动到链表末尾（尾节点 `tail` 的前面）。
- 这样，靠近头节点 `head` 的节点就是最久未使用的，靠近尾节点 `tail` 的节点就是最近使用的。



2. 淘汰策略

- 当缓存容量满时，直接删除头节点后面的节点 (`head.next`)，因为它是最久未使用的节点。
- 这种设计避免了遍历整个链表来寻找最久未使用的节点，操作复杂度是 $O(1)$ 。

示例说明

假设容量为 3，操作如下：

1. 插入三个键值对：`put(1, 1)`, `put(2, 2)`, `put(3, 3)`。

2. 缓存状态：

- 链表：`head <-> 1 <-> 2 <-> 3 <-> tail`
 - 节点 `1` 是最久未使用的，`3` 是最近使用的。
 - 哈希表：`{1: Node(1), 2: Node(2), 3: Node(3)}`
-

操作：访问 `get(2)`

1. 访问 `key = 2` 的值：

- 链表中节点 `2` 移动到链表末尾。

2. 缓存状态：

- 链表：`head <-> 1 <-> 3 <-> 2 <-> tail`
- 节点 `1` 是最久未使用的，`2` 是最近使用的。
- 哈希表不变：`{1: Node(1), 2: Node(2), 3: Node(3)}`

操作：插入 `put(4, 4)`

1. 容量已满，移除最久未使用的节点：

- 删除链表中的 `head.next` (节点 `1`)。
- 从哈希表中删除 `key = 1`。

2. 插入新节点 `4` 到链表末尾。

3. 缓存状态：

- 链表：`head <-> 3 <-> 2 <-> 4 <-> tail`
- 节点 `3` 是最久未使用的，`4` 是最近使用的。
- 哈希表：`{2: Node(2), 3: Node(3), 4: Node(4)}`

易错点：

问题 1：`self.tail.pre = node` 与 `self.tail.pre.next = node` 顺序错误

在 `move_node_to_tail` 方法中，以下两行的顺序是错误的：

python

复制代码

```
self.tail.pre = node  
self.tail.pre.next = node
```

- 问题原因：这会导致 `self.tail.pre` 被更新为 `node`，使得 `self.tail.pre.next` 指向自己而不是 `node`。
- 修正方式：应先更新 `self.tail.pre.next`，再更新 `self.tail.pre`。

问题 4：`get` 方法的逻辑不简洁

以下代码中多余地调用了两次 `hashmap.get()`：

python

复制代码

```
res = self.hashmap.get(key, -1)  
if res == -1:  
    return res  
else:  
    return res.value
```

- 问题：在 `key` 不存在时，会查询两次 `hashmap`，浪费性能。
- 修正方式：直接使用 `if key in self.hashmap` 判断是否存在。

修正后的代码：

python

复制代码

```
def get(self, key: int) -> int:  
    if key in self.hashmap:  
        self.move_node_to_tail(key)  
        return self.hashmap[key].value  
    return -1
```

插入新节点 B 的错误操作

如果插入 B 时，顺序错误：

python

复制代码

```
self.tail.prev = B # 错误：先更新尾节点的 prev  
self.tail.prev.next = B # 这里 self.tail.prev 已经是 B, B.next 会指向自己！  
B.prev = self.tail.prev # 这里 B.prev 会指向自己，而不是旧的最后一个节点  
B.next = self.tail
```

执行顺序导致问题：

1. `self.tail.prev = B` 使 `self.tail.prev` 指向新节点 B。
2. 然后执行 `self.tail.prev.next = B` 时，等价于 `B.next = B`，导致 B 的 `next` 指向了自己，链表结构出现循环错误。

总结

正确的插入顺序非常关键：

1. **先更新新节点的前后指向**，确保它完整地连接到链表中。
2. **再更新旧节点和尾节点的指向**，完成新节点的插入。

这种顺序可以避免尾节点或新节点的引用被破坏，确保链表的完整性。

必须要先更新插入节点的前、后指向；

然后再更新插入前节点的后指向、后节点的指向。

删除节点时必须要：

先更新指向 (`next`)，移除旧节点逻辑；

再更新引用 (`prev`)，修复链表的前向引用。

八、二叉树

第 94 题 二叉树的中序遍历

94. 二叉树的中序遍历

简单

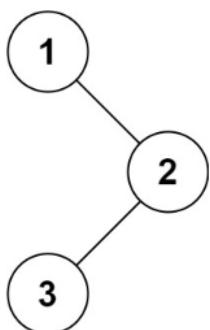
相关标签

相关企业

A+

给定一个二叉树的根节点 `root`，返回 它的 中序 遍历。

示例 1：



输入: `root = [1,null,2,3]`

输出: `[1,3,2]`

Python3 ▾ 🔒 智能模式

☰ ⌂ ⌂

```
1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
9
```

总结表格

遍历方法	访问顺序	实现方式
前序遍历	根 → 左子树 → 右子树	递归 / 栈
中序遍历	左子树 → 根 → 右子树	递归 / 栈
后序遍历	左子树 → 右子树 → 根	递归 / 栈
层序遍历	按层级从上到下、从左到右	队列 (BFS)

思路 1：递归遍历

二叉树的前序遍历递归实现步骤为：

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

思路 1：代码

```
1 class Solution:
2     def inorderTraversal(self, root: TreeNode) -> List[int]:
3         res = []
4         def inorder(root):
5             if not root:
6                 return
7             inorder(root.left)
8             res.append(root.val)
9             inorder(root.right)
10
11         inorder(root)
12         return res
```

py

思路 1：复杂度分析

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

思路 2：模拟栈迭代遍历

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

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

二叉树的前序遍历显式栈实现步骤如下：

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

思路 2：代码

```
1 class Solution:
2     def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
3         if not root: # 二叉树为空直接返回
4             return []
5
6         res = []
7         stack = []
8
9         while root or stack: # 根节点或栈不为空
10            while root:
11                stack.append(root) # 将当前树的根节点入栈
12                root = root.left # 找到最左侧节点
13
14            node = stack.pop() # 遍历到最左侧，当前节点无左子树时，将最左侧节点弹出
15            res.append(node.val) # 访问该节点
16            root = node.right # 尝试访问该节点的右子树
17
18        return res
```

思路 2：复杂度分析

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

104. 二叉树的最大深度

简单

相关标签

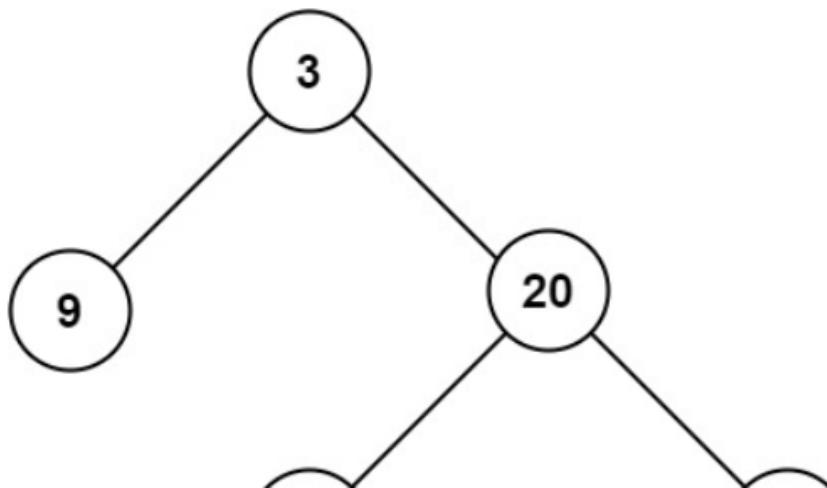
相关企业

A+

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：



</> 代码 | 笔记 ×

Python3 ▾ 智能模式

```
1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def maxDepth(self, root: Optional[TreeNode]) -> int:
9         |
```

解题思路

思路 1：递归算法

根据递归三步走策略，写出对应的递归代码。

1. 写出递推公式：当前二叉树的最大深度 = $\max(\text{当前二叉树左子树的最大深度}, \text{当前二叉树右子树的最大深度}) + 1$ 。
 - 即：先得到左右子树的高度，在计算当前节点的高度。
2. 确定终止条件：当前二叉树为空。
3. 翻译为递归代码：
 1. 定义递归函数：`maxDepth(self, root)` 表示输入参数为二叉树的根节点 `root`，返回结果为该二叉树的最大深度。
 2. 书写递归主体：`return max(self.maxDepth(root.left) + self.maxDepth(root.right))`。
 3. 确定递归终止条件：`if not root: return 0`

思路 1：代码

```
1 class Solution:
2     def maxDepth(self, root: Optional[TreeNode]) -> int:
3         if not root:
4             return 0
5
6         return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1
```

思路 1：复杂度分析

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

第 226 题 翻转二叉树

226. 翻转二叉树

已解答

简单

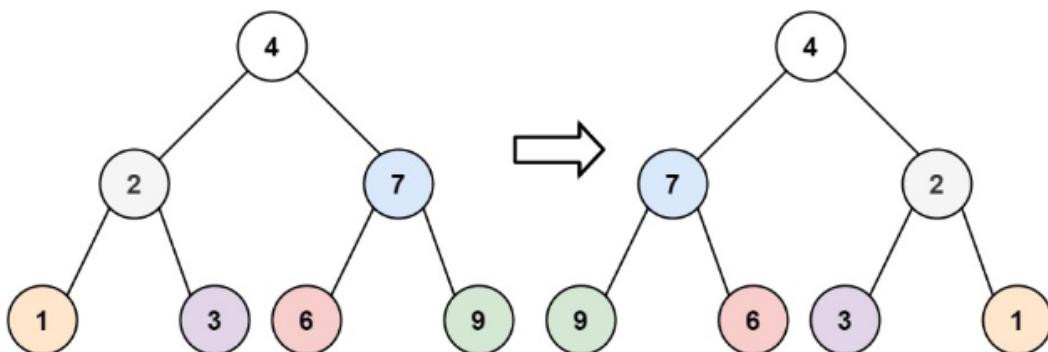
相关标签

相关企业

A+

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]`

代码 笔记

Python3 智能模式

```
1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
9         |
```

思路 1：递归遍历

根据我们的递推三步走策略，写出对应的递归代码。

1. 写出递推公式：

1. 递归遍历翻转左子树。
2. 递归遍历翻转右子树。
3. 交换当前根节点 `root` 的左右子树。

2. 明确终止条件：当前节点 `root` 为 `None`。

3. 翻译为递归代码：

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

2. 书写递归主体：

```
1 left = self.invertTree(root.left)
2 right = self.invertTree(root.right)
3 root.left = right
4 root.right = left
5 return root
```

py

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

4. 返回根节点 `root`。

思路 1：代码

```
1 class Solution:
2     def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
3         if not root:
4             return None
5         left = self.invertTree(root.left)
6         right = self.invertTree(root.right)
7         root.left = right
8         root.right = left
9         return root
```

py

思路 1：复杂度分析

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

第 101 题 对称二叉树

题目描述



101. 对称二叉树

简单

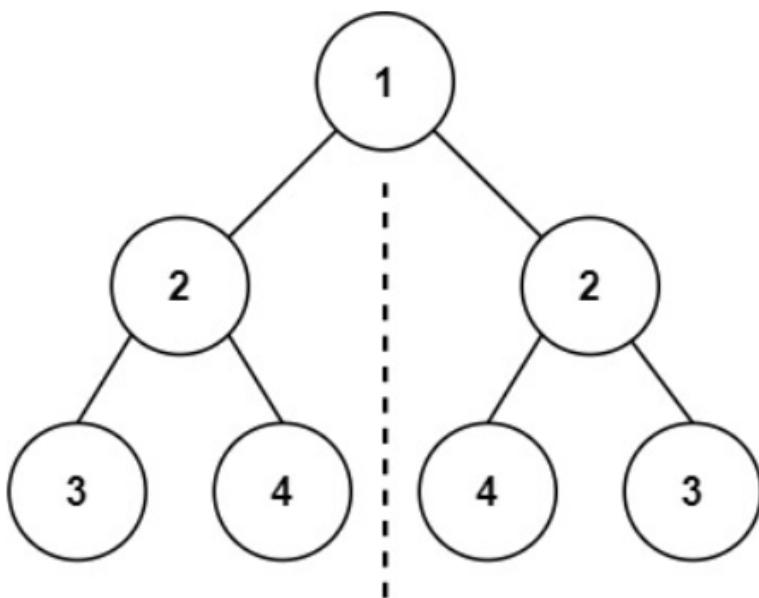
相关标签

相关企业

A+

给你一个二叉树的根节点 `root`，检查它是否轴对称。

示例 1：



代码 | 笔记 ×

Python3 ▾ 智能模式

```
1 # Definition for a binary tree node.
2 # class TreeNode:
3 #     def __init__(self, val=0, left=None, right=None):
4 #         self.val = val
5 #         self.left = left
6 #         self.right = right
7 class Solution:
8     def isSymmetric(self, root: Optional[TreeNode]) -> bool:
9 
```

思路 1：递归遍历

如果一棵二叉树是对称的，那么其左子树和右子树的外侧节点的节点值应当是相等的，并且其左子树和右子树的内侧节点的节点值也应当是相等的。

那么我们可以通过递归方式，检查其左子树与右子树外侧节点和内测节点是否相等。即递归检查左子树的左子节点值与右子树的右子节点值是否相等（外侧节点值是否相等），递归检查左子树的右子节点值与右子树的左子节点值是否相等（内测节点值是否相等）。

具体步骤如下：

1. 如果当前根节点为 `None`，则直接返回 `True`。
2. 如果当前根节点不为 `None`，则调用 `check(left, right)` 方法递归检查其左右子树是否对称。
 1. 如果左子树节点为 `None`，并且右子树节点也为 `None`，则直接返回 `True`。
 2. 如果左子树节点为 `None`，并且右子树节点不为 `None`，则直接返回 `False`。
 3. 如果左子树节点不为 `None`，并且右子树节点为 `None`，则直接返回 `False`。
 4. 如果左子树节点值不等于右子树节点值，则直接返回 `False`。
 5. 如果左子树节点不为 `None`，并且右子树节点不为 `None`，并且左子树节点值等于右子树节点值，则：
 1. 递归检测左右子树的外侧节点是否相等。
 2. 递归检测左右子树的内测节点是否相等。
 3. 如果左右子树的外侧节点、内测节点值相等，则返回 `True`。

思路 1：代码

```
1  class Solution:
2      def isSymmetric(self, root: TreeNode) -> bool:
3          if root == None:
4              return True
5          return self.check(root.left, root.right)
6
7      def check(self, left: TreeNode, right: TreeNode):
8          if left == None and right == None:
9              return True
10         elif left == None and right != None:
11             return False
12         elif left != None and right == None:
13             return False
14         elif left.val != right.val:
15             return False
16
17         return self.check(left.left, right.right) and self.check(left.right, right.left)
```

思路 1：复杂度分析

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

(1) 前序遍历 (Preorder Traversal)

- 访问顺序: 根节点 → 左子树 → 右子树
- 示例:

markdown

复制代码

```
1
/
2   3
/
4   5
```

前序遍历的结果是: [1, 2, 4, 5, 3]

- 代码实现 (递归) :

python

复制代码

```
def preorderTraversal(root):
    res = []
    def preorder(node):
        if not node:
            return
        res.append(node.val) # 访问根节点
        preorder(node.left) # 递归访问左子树
        preorder(node.right) # 递归访问右子树
    preorder(root)
    return res
```



- 代码实现 (迭代) :

python

复制代码

```
def preorderTraversal(root):
    if not root: return []
    res = []
    stack = [root]
    while stack:
        node = stack.pop()
        res.append(node.val)
        if node.right:
            stack.append(node.right) # 先压入右子节点
        if node.left:
            stack.append(node.left) # 后压入左子节点
    return res
```

(2) 中序遍历 (Inorder Traversal)

- 访问顺序：左子树 → 根节点 → 右子树
- 示例：

markdown

复制代码

```
1
/
2   3
/
4   5
```

中序遍历的结果是： [4, 2, 5, 1, 3]

- 代码实现（递归）：

python

复制代码

```
def inorderTraversal(root):
    res = []
    def inorder(node):
        if not node:
            return
        inorder(node.left)  # 递归访问左子树
        res.append(node.val) # 访问根节点
        inorder(node.right) # 递归访问右子树
    inorder(root)
    return res
```



- 代码实现（迭代）：

python

复制代码

```
def inorderTraversal(root):
    res = []
    stack = []
    current = root
    while current or stack:
        while current:
            stack.append(current) # 左子树一路压栈
            current = current.left
        current = stack.pop()
        res.append(current.val) # 访问根节点
        current = current.right # 转向右子树
    return res
```

(3) 后序遍历 (Postorder Traversal)

- 访问顺序: 左子树 → 右子树 → 根节点
- 示例:

markdown

复制代码

```
1
/
2  3
/
4  5
```

后序遍历的结果是: [4, 5, 2, 3, 1]

- 代码实现 (递归) :

python

复制代码

```
def postorderTraversal(root):
    res = []
    def postorder(node):
        if not node:
            return
        postorder(node.left)    # 递归访问左子树
        postorder(node.right)   # 递归访问右子树
        res.append(node.val)    # 访问根节点
    postorder(root)
    return res
```



- 代码实现 (迭代) :

python

复制代码

```
def postorderTraversal(root):
    if not root: return []
    res = []
    stack = [root]
    while stack:
        node = stack.pop()
        res.append(node.val)
        if node.left:
            stack.append(node.left)    # 先压入左子节点
        if node.right:
            stack.append(node.right)   # 后压入右子节点
    return res[::-1]    # 最后将结果反转
```

第 543 题 二叉树的直径

543. 二叉树的直径

简单

相关标签

相关企业

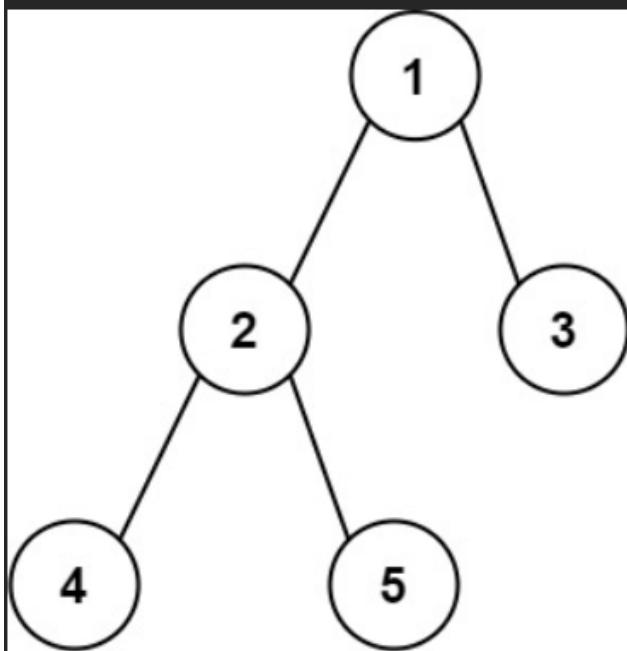
A+

给你一棵二叉树的根节点，返回该树的 **直径**。

二叉树的 **直径** 是指树中任意两个节点之间最长路径的 **长度**。这条路径可能经过也可能不经过根节点 `root`。

两节点之间路径的 **长度** 由它们之间边数表示。

示例 1：



输入: `root = [1,2,3,4,5]`

输出: 3

解释: 3，取路径 `[4,2,1,3]` 或 `[5,2,1,3]` 的长度。

```

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        self.ans = 1 # 初始化直径长度为1，因为直径是节点数而不是边数

        def depth(root):
            if not root: # 空节点深度为0
                return 0

            # 递归计算左、右子树的深度
            L = depth(root.left)
            R = depth(root.right)

            # 更新最大直径。直径是左子树深度 + 右子树深度 + 当前节点 (1)。
            self.ans = max(self.ans, L + R + 1)

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

        depth(root) # 调用递归函数
        return self.ans - 1 # 因为题目要求返回边数，所以减1

```

关键点说明

1. 深度定义与直径计算：

- **深度**：从当前节点到叶子节点的最长路径上的节点数。
- **直径**：经过某节点时的最大路径长度，计算公式为：`左子树深度 + 右子树深度 + 1`。

2. 递归过程：

- 递归计算左右子树的深度。
- 在每次递归中，更新 `self.ans` 为当前节点的直径与之前记录的最大直径的较大值。
- 返回当前节点的深度给上一级调用。

3. 最终结果减 1：

- 递归中用节点数计算直径，但题目要求边数，因此最后减 1。

1. `self.ans = 1`

- **含义:**

- 这是一个**实例变量**（也叫对象属性），属于 `Solution` 类的实例。
- `self` 表示当前类的实例，`self.ans` 是实例的一个属性，它的值可以在类的其他方法中被访问和修改。

- **作用域:**

- `self.ans` 在类的所有方法中都可见，只要用 `self.ans` 就可以访问该变量。
- 不会因为离开方法作用域而消失。

- **场景:**

- 当你需要在多个方法之间共享或保存状态时，可以用 `self.ans`。



2. `ans = 1`

- **含义:**

- 这是一个**局部变量**，它仅存在于定义它的方法的作用域内。
- 一旦方法执行完毕，局部变量 `ans` 就会被销毁，不能在类的其他方法中访问。

- **作用域:**

- 仅在定义它的方法内有效，其他方法无法直接访问。

- **场景:**

- 当变量只在单个方法中使用，且不需要在类的其他方法中共享时，可以用 `ans`。

在二叉树中，节点的**深度**是指从该节点到其最远叶子节点的路径长度。即使一个节点的左右子树深度均为 `0`，它本身也算作一个深度的贡献，因为这个节点是路径的一部分。

让我们详细解释为什么在左右子树深度均为 `0` 时，仍然返回深度 `1`。

深度的定义

- 一个**叶子节点**是没有任何子节点的节点。

- 叶子节点的深度计算为 `1`：

- 左右子树深度 `0`（没有子节点）。
- 自己加上自身的深度贡献： $\max(0, 0) + 1 = 1$ 。

对于非叶子节点，其深度是左右子树深度的**最大值**加 `1`。



为什么加 1?

加 1 是因为：

1. 当前节点本身需要计入深度。
2. 深度是从该节点到其叶子节点的路径长度。

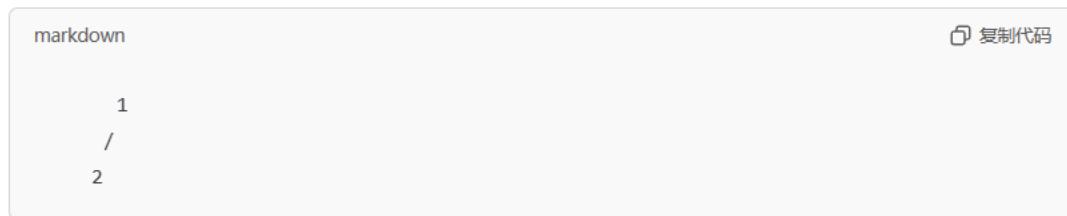


例如：

- 如果当前节点的左右子树深度都是 0，这表示它是一个叶子节点。
- 它的深度就是 1，因为路径仅包含它自身。

一个示例

示例树：



- 节点 2 (叶子节点) :

- 左子树深度 = 0
- 右子树深度 = 0
- 返回深度： $\max(0, 0) + 1 = 1$



- 节点 1 :

- 左子树深度 = 1 (节点 2 返回的深度)
- 右子树深度 = 0
- 返回深度： $\max(1, 0) + 1 = 2$



第 102 题 二叉树的层序遍历

102. 二叉树的层序遍历

中等

相关标签

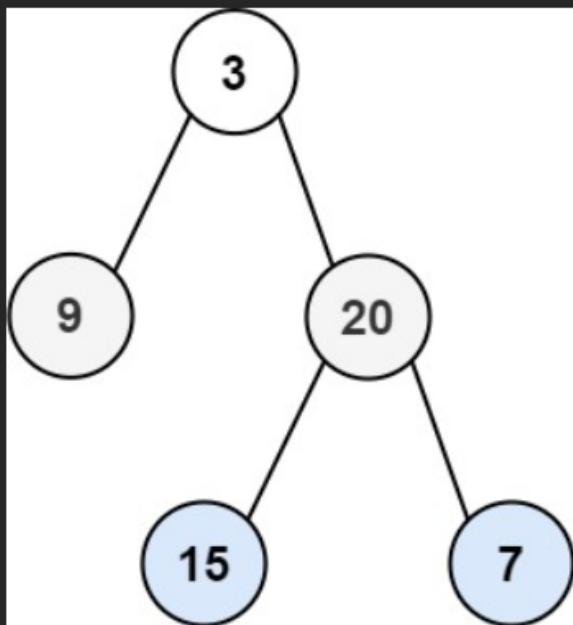
相关企业

提示

A+

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]]`

解题思路

思路 1：广度优先搜索

广度优先搜索，需要增加一些变化。普通广度优先搜索只取一个元素，变化后的广度优先搜索每次取出第 i 层上所有元素。

具体步骤如下：

1. 判断二叉树是否为空，为空则直接返回。
2. 令根节点入队。
3. 当队列不为空时，求出当前队列长度 s_i 。
4. 依次从队列中取出这 s_i 个元素，并对这 s_i 个元素依次进行访问。然后将其左右孩子节点入队，然后继续遍历下一层节点。
5. 当队列为空时，结束遍历。



思路 1：代码

```
1  class Solution:
2      def levelOrder(self, root: TreeNode) -> List[List[int]]:
3          if not root:
4              return []
5          queue = [root]
6          order = []
7          while queue:
8              level = []
9              size = len(queue)
10             for _ in range(size):
11                 curr = queue.pop(0)
12                 level.append(curr.val)
13                 if curr.left:
14                     queue.append(curr.left)
15                 if curr.right:
16                     queue.append(curr.right)
17             if level:
18                 order.append(level)
19         return order
```

思路 1：复杂度分析

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

`if not root:` 是一种 Pythonic 的方式，用于判断变量 `root` 是否为假值。它在许多场景中用于判断变量是否为空，尤其是在树的节点或链表节点处理中。

原因

在 Python 中，以下值会被认为是“假值”：

- `None`
- 数值类型的 `0` 或 `0.0`
- 空序列（如 `[]`, `()`, `''`）
- 空集合（如 `{}`, `set()`）
- `False`

因此，`if not root:` 等价于 `if root is None:`，但更简洁。

`queue = [root]` 的初始化是正确的，因为它存储的是树的节点引用，而不是树的数组表示。以下解释为何如此设计：

1. `queue` 的设计目的

`queue` 是一个用于广度优先搜索（BFS）的队列，目的是按层遍历二叉树中的节点。

- 它存储的是节点对象，而不是原始数组 `[3, 9, 20, null, null, 15, 7]`。
- `root` 是树的根节点，即值为 `3` 的 `TreeNode` 对象。

2. 为什么不是数组 `[3, 9, 20, null, null, 15, 7]`

如果 `queue` 存储数组 `[3, 9, 20, null, null, 15, 7]`，算法就不能正确地访问每个节点的左右子节点（`left` 和 `right`），也无法动态扩展队列。

初始化队列

队列以根节点 `root` 为起点:

python

```
queue = [root]
```

复制代码

这意味着队列初始状态是:

arduino

```
queue = [TreeNode(3)]
```

复制代码

初始化:

1. 变量初始化:

- `queue = [3]` (包含根节点) ←
- `order = []` (存储每一层的值)

第一轮 while 循环:

1. 当前队列: `queue = [3]`

2. 队列长度: `size = 1`

3. 进入 for 循环:

• 处理节点 `3`:

- 从队列中取出: `curr = queue.pop(0)` → `curr.val = 3`
- 加入当前层级: `level = [3]`
- 添加左右子节点: `queue = [9, 20]` (9 是左子节点, 20 是右子节点)

4. 完成一层:

- 将当前层结果加入: `order = [[3]]`

第二轮 while 循环：

1. 当前队列： `queue = [9, 20]`
2. 队列长度： `size = 2`
3. 进入 for 循环：
 - 处理节点 `9`：
 - 从队列中取出： `curr = queue.pop(0) → curr.val = 9`
 - 加入当前层级： `level = [9]`
 - 没有子节点，不修改队列： `queue = [20]`
 - 处理节点 `20`：
 - 从队列中取出： `curr = queue.pop(0) → curr.val = 20`
 - 加入当前层级： `level = [9, 20]`
 - 添加左右子节点： `queue = [15, 7]`
4. 完成一层：
 - 将当前层结果加入： `order = [[3], [9, 20]]`

103. 二叉树的锯齿形层序遍历

中等

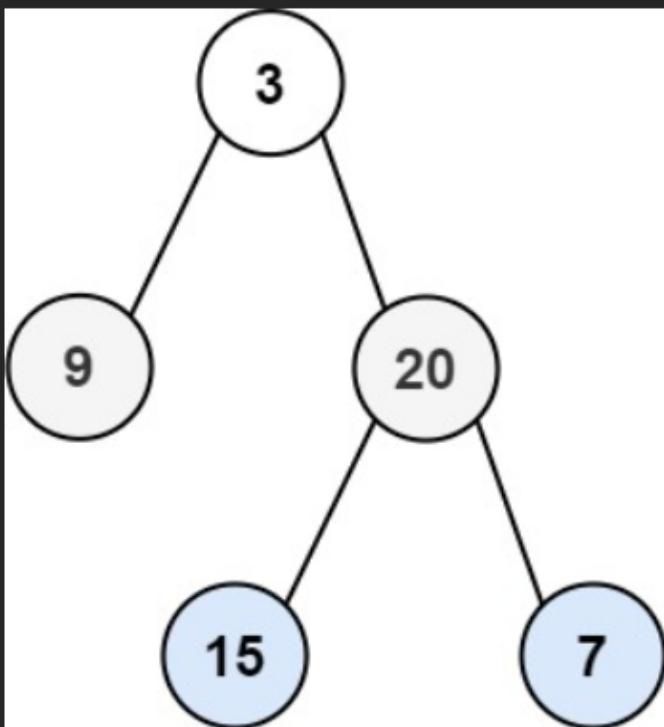
相关标签

相关企业

A+

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]]`

```

7  class Solution:
8      def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[int]:
9          if not root:
10              return []
11          res = []
12          deque = collections.deque([root])
13          while deque:
14              tmp = collections.deque()
15              for _ in range(len(deque)):
16                  node = deque.popleft()
17                  if len(res) % 2 == 0: # 奇数层 -> 插入队列尾部
18                      tmp.append(node.val)
19                  else: # 偶数层 -> 插入队列头部
20                      tmp.appendleft(node.val)
21                  if node.left: deque.append(node.left)
22                  if node.right: deque.append(node.right)
23              res.append(list(tmp))
24          return res

```

1. `if node.left: deque.append(node.left)`

- 含义:** 如果当前节点有左子节点 (`node.left` 不为空), 就将这个左子节点加入到队列 `deque` 的尾部。
- 目的:** 下一层遍历时会按照加入的顺序从队列中取出这些节点, 因此加入左子节点是为了在后续处理左子节点的值。

2. `if node.right: deque.append(node.right)`

- 含义:** 如果当前节点有右子节点 (`node.right` 不为空), 就将这个右子节点加入到队列 `deque` 的尾部。
- 目的:** 同样是为了在下一层遍历时处理右子节点。

队列的工作机制

队列 `deque` 作为广度优先搜索 (BFS) 的核心数据结构，遵循 **先进先出 (FIFO)** 的原则。

- 当前层的所有节点都会依次从队列中取出 (`popleft()`)。
- 当前层节点的子节点会按顺序加入队列 (通过 `append()`)，等待下一次处理。

假设我们有以下二叉树：

markdown

```
3
/ \
9  20
/ \
15  7
```

复制 编辑

初始化

- `res = []`：存储最终的结果。
- `dequeue = collections.deque([root])`：队列初始化为包含根节点 `3`。

第一层：`root` 节点 (值 3)

1. 进入 `while dequeue:` : `dequeue = [3]` (根节点)。

- 初始化 `tmp` :

python

```
tmp = collections.deque() # tmp = []
```

复制 编辑

2. 进入 `for _ in range(len(dequeue)):` , `len(dequeue) = 1`，所以只会循环一次。

3. 第一次循环：

- 从 `dequeue` 中弹出节点 `node = 3` :

python

```
node = dequeue.popleft() # node = 3
```

复制 编辑

4. 检查层数:

- 当前层是第 0 层 (`len(res) % 2 == 0`)，是奇数层，所以将节点值 `3` 加入 `tmp`：

```
python
```

复制 编辑

```
tmp.append(3) # tmp = [3]
```

5. 添加子节点到队列:

- 节点 `3` 有左右子节点 `9` 和 `20`，所以将它们加入 `dequeue`：

```
python
```

复制 编辑

```
dequeue.append(9) # dequeue = [9]
dequeue.append(20) # dequeue = [9, 20]
```

6. 结束 `for` 循环，将当前层的 `tmp` 转换为列表并加入到 `res`：

```
python
```

复制 编辑

```
res.append(list(tmp)) # res = [[3]]
```

第二层：节点 9 和 20

1. 进入 `while dequeue: , dequeue = [9, 20]`。
2. 初始化 `tmp`：

```
python
```

复制 编辑

```
tmp = collections.deque() # tmp = []
```

3. 进入 `for _ in range(len(dequeue)): , len(dequeue) = 2`，循环两次。

第一次循环（节点 9）：

- 从 `dequeue` 中弹出节点 `node = 9`：

```
python
```

复制 编辑

```
node = dequeue.popleft() # node = 9
```

- 当前层是第 1 层 (`len(res) % 2 == 1`)，是偶数层，所以将节点值 `9` 加入 `tmp` 的左边：

```
python
```

复制 编辑

```
tmp.appendleft(9) # tmp = [9]
```

- 节点 `9` 没有子节点，所以不做任何操作。

第二次循环（节点 20）：

- 从 `dequeue` 中弹出节点 `node = 20`：

```
python
```

复制 编辑

```
node = dequeue.popleft() # node = 20
```

- 当前层是第 1 层，依然是偶数层，所以将节点值 `20` 加入 `tmp` 的左边：

```
python
```

复制 编辑

```
tmp.appendleft(20) # tmp = [20, 9]
```

- 节点 `20` 有左右子节点 `15` 和 `7`，将它们加入 `dequeue`：

```
python
```

复制 编辑

```
dequeue.append(15) # dequeue = [15]  
dequeue.append(7) # dequeue = [15, 7]
```

- 结束 `for` 循环，将当前层的 `tmp` 转换为列表并加入到 `res`：

```
python
```

复制 编辑

```
res.append(list(tmp)) # res = [[3], [20, 9]]
```

`len(res) % 2 == 0`: 不应该是偶数层吗

你提到的 `len(res) % 2 == 0` 确实对应的是偶数层的问题。实际上，这是在判断当前的层数是否是偶数层（从 0 开始计数），这在很多情况下会让人感到混淆。我们来澄清一下这个判断是如何工作的。

1. 层数的计算：

- 在这段代码中，`len(res)` 记录的是结果列表 `res` 中已完成的层数（即已经处理过的层）。
- 初始时，`res` 是空的，所以 `len(res) = 0`，对应的是 第 0 层（即根节点层）。

2. 奇数层与偶数层的判断：

- 在 第 0 层（根节点所在的层），`len(res)` 为 0，`0 % 2 == 0`，因此 `len(res) % 2 == 0`。
- 在 第 1 层，`len(res)` 为 1，`1 % 2 == 1`，因此进入 `else` 分支，说明是 偶数层（按照代码的实际处理是偶数层，代码顺序从 0 开始）。

第 108 题 将有序数组转换为二叉搜索树

108. 将有序数组转换为二叉搜索树

简单

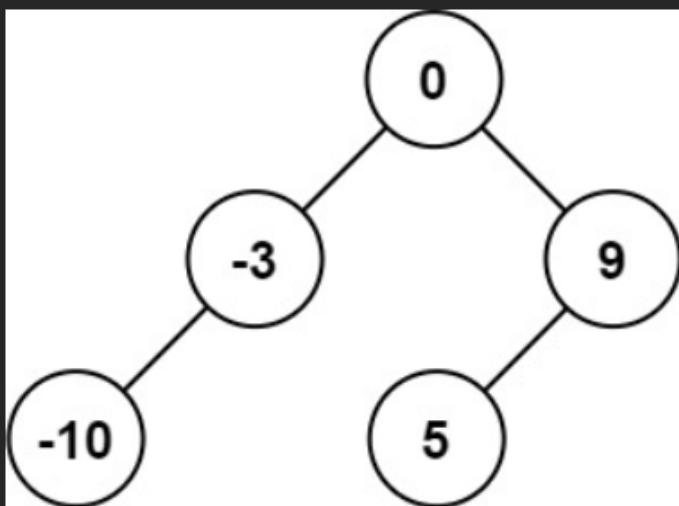
相关标签

相关企业

A+

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：

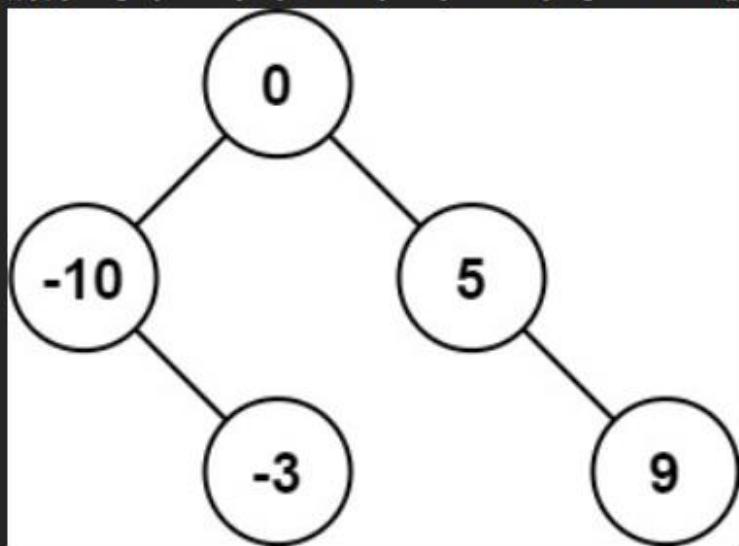


输入: `nums = [-10,-3,0,5,9]`

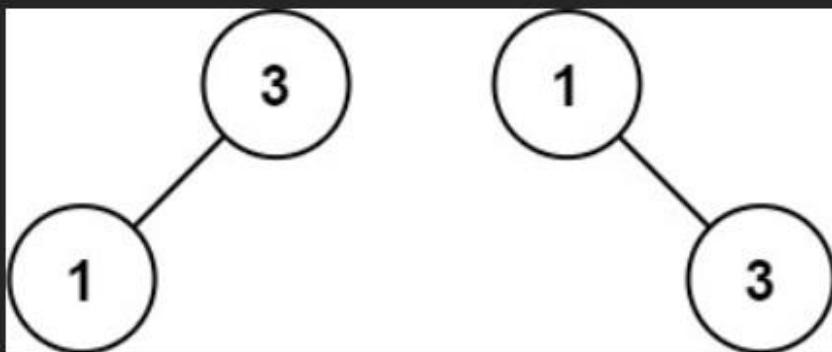
输出: `[0,-3,9,-10,null,5]`

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

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。

解题思路

思路 1: 递归遍历

直观上，如果把数组的中间元素当做根，那么数组左侧元素都小于根节点，右侧元素都大于根节点，且左右两侧元素个数相同，或最多相差 1 个。那么构建的树高度差也不会超过 1。

所以猜想到：如果左右子树越平均，树就越平衡。这样我们就可以每次取中间元素作为当前的根节点，两侧的元素作为左右子树递归建树，左侧区间 $[L, mid - 1]$ 作为左子树，右侧区间 $[mid + 1, R]$ 作为右子树。

思路 1：代码

```
1 class Solution:
2     def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
3         def build(left, right):
4             if left > right:
5                 return
6             mid = left + (right - left) // 2
7             root = TreeNode(nums[mid])
8             root.left = build(left, mid - 1)
9             root.right = build(mid + 1, right)
10            return root
11        return build(0, len(nums) - 1)
```

思路 1：复杂度分析

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

二叉搜索树的性质

1. 节点值的关系：

- 对于任意一个节点 Node:
 - 它左子树的所有节点值 **小于** Node.val。
 - 它右子树的所有节点值 **大于** Node.val。

2. 递归定义：

- 左子树和右子树本身也是二叉搜索树。
- 这种性质递归地适用于树中的每个节点。

3. 树的中序遍历：

- 对二叉搜索树进行 **中序遍历** (左-根-右) 时，节点值是按 **递增顺序排列** 的。

一个典型的二叉搜索树：

markdown

复制代码

```
8
/
3   10
/
1   6   14
/
4   7  13
```

验证性质：

- 对于节点 8：
 - 左子树是 [3, 1, 6, 4, 7]，所有值 < 8 。
 - 右子树是 [10, 14, 13]，所有值 > 8 。
- 对于节点 3：
 - 左子树是 [1]，值 < 3 。
 - 右子树是 [6, 4, 7]，值 > 3 。
- 对于节点 10：
 - 左子树为空（无节点）。
 - 右子树是 [14, 13]，值 > 10 。



98. 验证二叉搜索树

中等

相关标签

相关企业

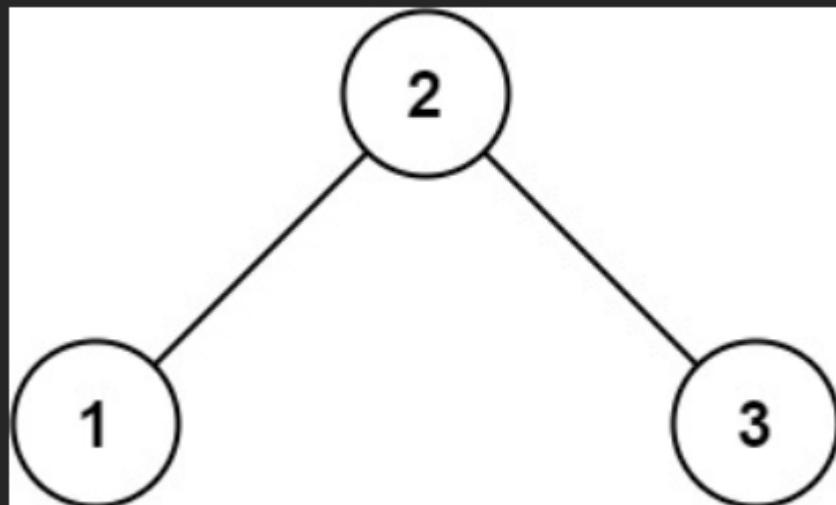
A_文

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`

第 230 题 二叉搜索树中第 K 个最小的元素

230. 二叉搜索树中第 K 小的元素

中等

相关标签

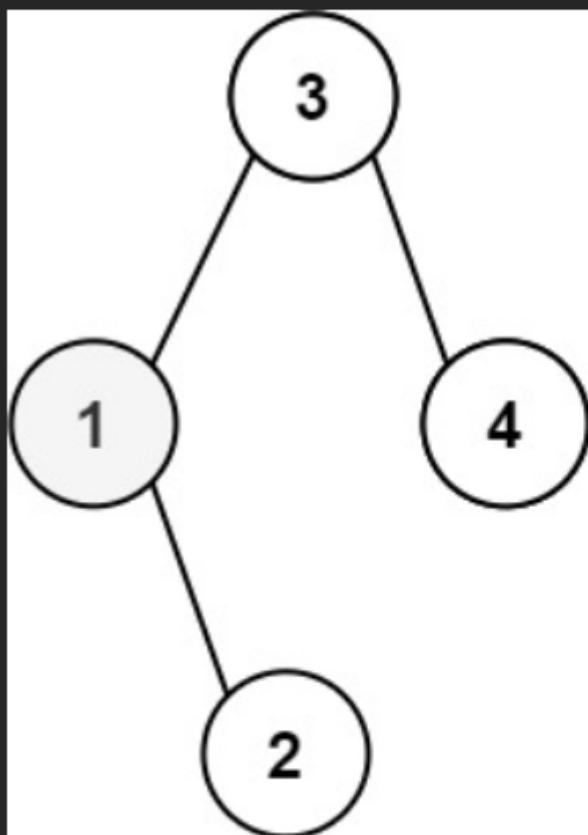
相关企业

提示

A₂

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

199. 二叉树的右视图

中等

相关标签

相关企业

A+

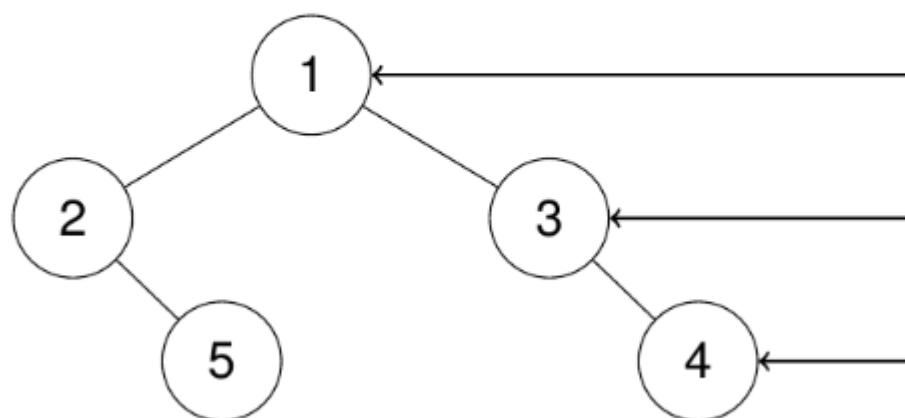
给定一个二叉树的 **根节点** `root`，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例 1：

输入： `root = [1,2,3,null,5,null,4]`

输出： `[1,3,4]`

解释：



114. 二叉树展开为链表

中等

相关标签

相关企业

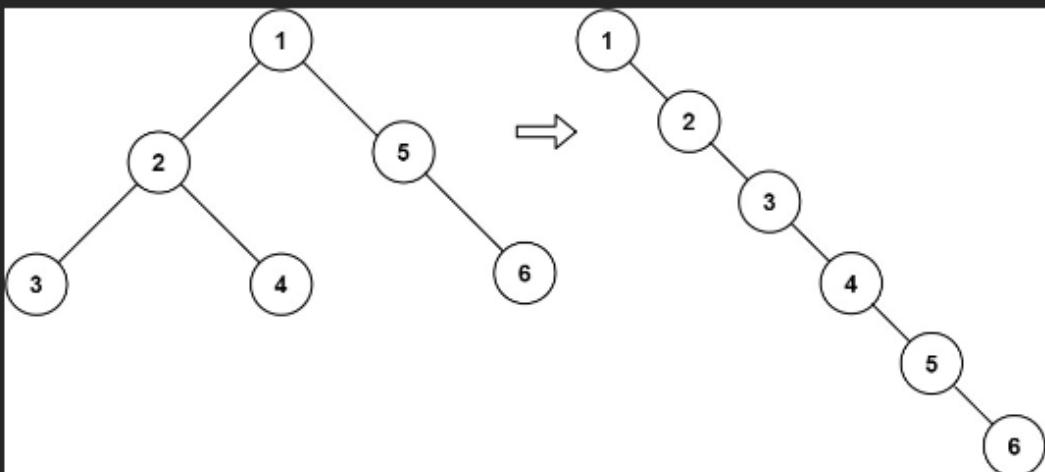
提示

A₂

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]`

第 105 题 从前序与中序遍历序列构造二叉树

105. 从前序与中序遍历序列构造二叉树

中等

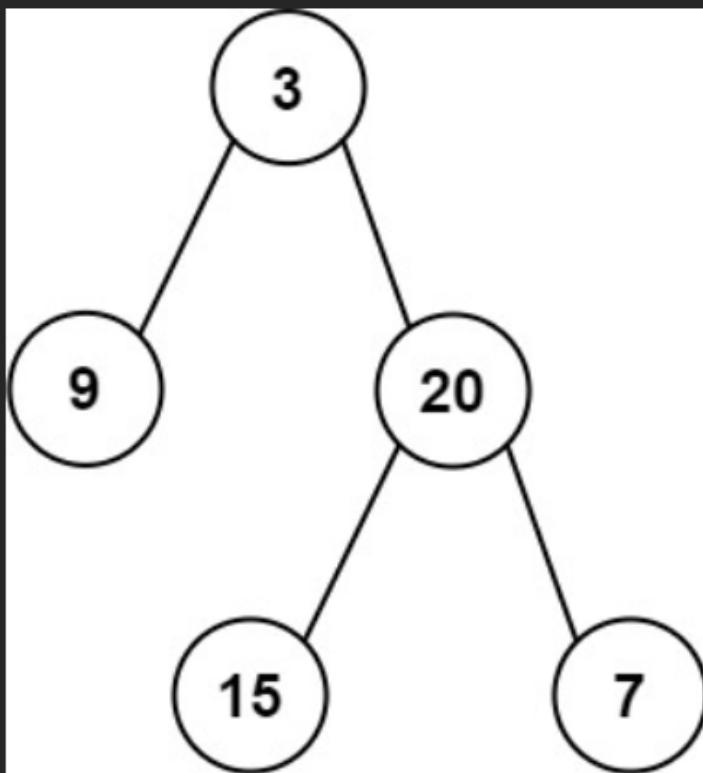
相关标签

相关企业

A₂

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]`

第 437 题 路径总和III

437. 路径总和 III

中等

相关标签

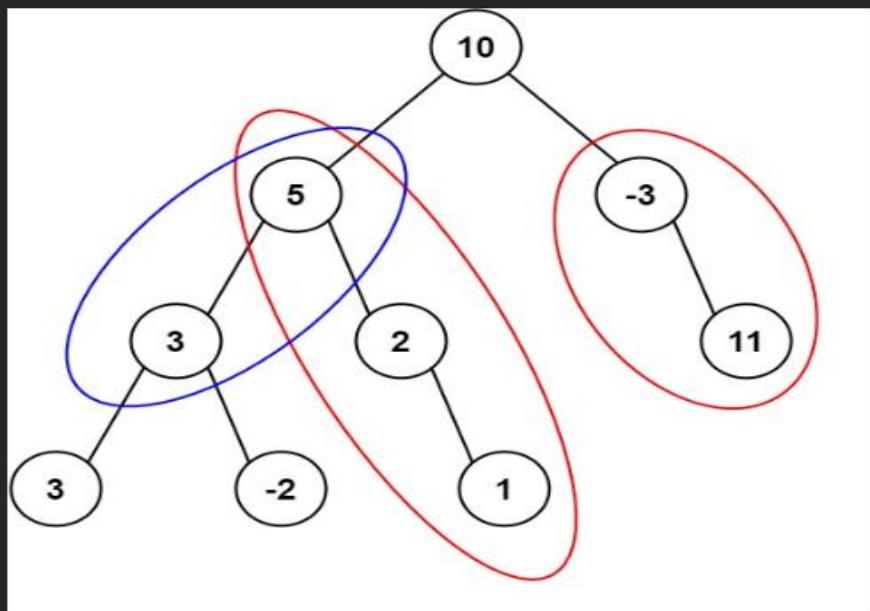
相关企业

A+

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

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

示例 1：



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

输出: 3

解释: 和等于 8 的路径有 3 条，如图所示。

示例 2：

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

输出: 3

236. 二叉树的最近公共祖先

中等

相关标签

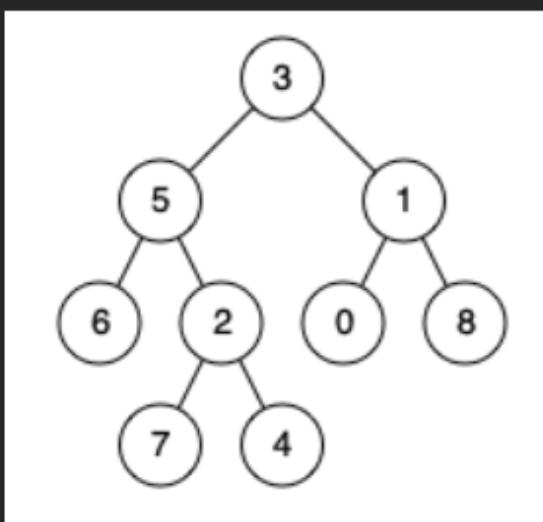
相关企业

A+

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。

124. 二叉树中的最大路径和

困难

相关标签

相关企业

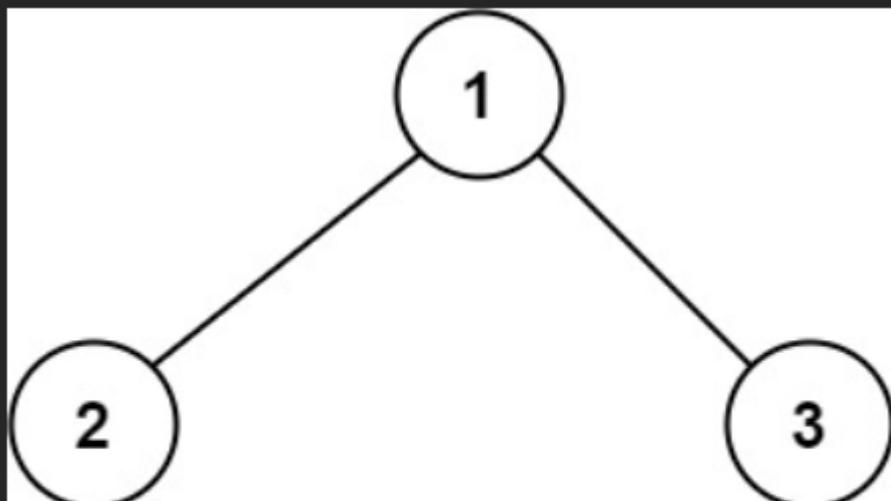
A₂

二叉树中的 **路径** 被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。同一个节点在一条路径序列中 **至多出现一次**。该路径 **至少包含一个** 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

给你一个二叉树的根节点 `root`，返回其 **最大路径和**。

示例 1：



输入: `root = [1, 2, 3]`

输出: 6

解释: 最优路径是 $2 \rightarrow 1 \rightarrow 3$ ，路径和为 $2 + 1 + 3 = 6$

九、图论

第 200 题 岛屿数量

200. 岛屿数量

中等

相关标签

相关企业

A_文

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
```

解题思路

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

使用深度优先搜索或者广度优先搜索都可以。

思路 1：深度优先搜索

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

```
class Solution:  
    def numIslands(self, grid: List[List[str]]) -> int:  
        count = 0  
        for i in range(len(grid)):  
            for j in range(len(grid[0])):  
                if grid[i][j] == "1": # 找到一个新的岛屿  
                    self.dfs(grid, i, j) # 深度优先搜索，标记整个岛屿  
                    count += 1 # 每次找到一个岛屿，计数增加  
        return count  
  
    def dfs(self, grid: List[List[str]], i: int, j: int):  
        m, n = len(grid), len(grid[0]) # 获取网格的行列数  
        # 如果越界或是水域 ('0')，则返回  
        if i < 0 or i >= m or j < 0 or j >= n or grid[i][j] == "0":  
            return  
  
        grid[i][j] = "0" # 将当前陆地标记为水域，防止重复访问  
        # 递归地搜索四个方向  
        self.dfs(grid, i + 1, j) # 向下  
        self.dfs(grid, i - 1, j) # 向上  
        self.dfs(grid, i, j + 1) # 向右  
        self.dfs(grid, i, j - 1) # 向左
```

使用 BFS 来解决（以点散面）：

```
class Solution:
    def bfs(self, grid, i, j):
        n = len(grid)
        m = len(grid[0])
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # 四个方向：右、下、左、上

        queue = deque([(i, j)]) # 初始化队列，将起始点加入队列
        grid[i][j] = '0' # 标记起始点为已访问

        while queue:
            x, y = queue.popleft() # 取出队首元素
            for dx, dy in directions: # 遍历四个方向
                nx, ny = x + dx, y + dy
                if 0 <= nx < n and 0 <= ny < m and grid[nx][ny] == '1':
                    grid[nx][ny] = '0' # 标记新访问的格子
                    queue.append((nx, ny)) # 将新格子加入队列

    def numIslands(self, grid: List[List[str]]) -> int:
        count = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1': # 遇到岛屿起点
                    self.bfs(grid, i, j) # 用 BFS 标记整个岛屿
                    count += 1 # 岛屿数量 +1
        return count
```

代码解读

BFS 方法 (bfs)

1. 初始化队列：

python

复制代码

```
queue = deque([(i, j)])
grid[i][j] = '0'
```

- 使用 `deque` (双端队列) 来存储待处理的格子。
- 将当前格子 `(i, j)` 入队并标记为 `'0'` (表示已访问)。

2. 逐层处理队列中的格子:

python

复制代码

```
while queue:  
    x, y = queue.popleft()
```

- 每次从队列中取出一个格子 `(x, y)`。

3. 访问四个方向:

python

复制代码

```
for dx, dy in directions:  
    nx, ny = x + dx, y + dy  
    if 0 <= nx < n and 0 <= ny < m and grid[nx][ny] == '1':  
        grid[nx][ny] = '0'  
        queue.append((nx, ny))
```

- 遍历当前格子的上下左右四个方向。
- 如果邻居格子是 `'1'` 且未访问过 (即在网格范围内) , 将其标记为 `'0'` 并加入队列。

在 Python 中, `queue = deque([(i, j)])` 使用的是列表 `[]` 包裹元组 `(i, j)`。这样做的目的是将一个元素 `(i, j)` 加入队列 `deque` 中, 而不是直接尝试将两个值 `i` 和 `j` 当作两个单独的元素放入队列。

原因

1. `deque` 是一个序列数据结构:

- `deque` 需要被初始化为一个序列 (如列表、元组、字符串等)。`deque([(i, j)])` 是将一个包含单个元组的列表传递给 `deque`。
- 如果你直接写 `deque((i, j))`, 那么 `deque` 会尝试将元组中的每个值分别作为单独的元素。例如:

python

复制代码

```
from collections import deque  
queue = deque((1, 2)) # 将 1 和 2 分别作为两个独立的元素  
print(queue) # 输出 deque([1, 2])
```

2. 表示一个坐标点 `(i, j)`:

- `[(i, j)]` 将 `(i, j)` 视为一个整体, 表示一个二维坐标点, 而不是拆解为两个值。

3. 队列的操作需要一致性:

- 在 BFS 的过程中, 队列的每个元素都是一个坐标 `(x, y)`。使用 `[(i, j)]` 初始化可以确保队列中的元素都是二维坐标形式, 这样后续处理时无需额外判断数据格式。

关键点总结

- 整体 vs 拆分：
 - `for item in directions`：不解包时，`item` 是一个完整的元组，例如 `(0, 1)`。
 - `for dx, dy in directions`：解包时，`(0, 1)` 被拆分为 `dx = 0` 和 `dy = 1`。
- Python 的灵活性：解包是 Python 的特性之一，它为开发者提供了简洁、高效的写法，特别适合像遍历坐标、键值对这样的场景。

为什么元组可以拆分？

因为在 Python 中，元组（或其他可迭代对象）是可以直接解包的。它的设计就是为了方便这种场景，比如遍历二维坐标、键值对等。以下是关键点：

1. 元组是可迭代对象：元组可以像列表一样迭代，所以 `for ... in ...` 能逐一取出元组中的元素。
2. 解包原理：当你写 `for dx, dy in directions` 时：
 - 每次从 `directions` 中取出的元素是一个元组（如 `(0, 1)`）。
 - Python 会将元组自动拆分，把第一个值赋给 `dx`，第二个值赋给 `dy`。
 - 这种解包机制对其他可迭代对象（如列表）同样适用。

DFS 与 BFS 的对比

特点	DFS	BFS
搜索顺序	深入一条路径，直到尽头，再回溯搜索其他路径	层层展开，每次处理一个层级的所有节点
实现方式	递归或显式栈	显式队列
空间复杂度	最坏为递归深度 ($O(n)$)	队列最大长度 ($O(n)$)
使用场景	需要快速到达叶子节点	需要按层遍历，或最短路径问题

DFS 和 BFS 的核心逻辑类似，具体选择取决于题目需求和实现偏好。

十、回溯

第 46 题 全排列

46. 全排列

中等

相关标签

相关企业

A_文

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]]`

思路 1：代码

```
1 class Solution:
2     def permute(self, nums: List[int]) -> List[List[int]]:
3         res = []      # 存放所有符合条件结果的集合
4         path = []    # 存放当前符合条件的结果
5         def backtracking(nums):           # nums 为选择元素列表
6             if len(path) == len(nums):    # 说明找到了一组符合条件的结果
7                 res.append(path[:])       # 将当前符合条件的结果放入集合中
8                 return
9
10            for i in range(len(nums)):   # 枚举可选元素列表
11                if nums[i] not in path:   # 从当前路径中没有出现的数字中选择
12                    path.append(nums[i]) # 选择元素
13                    backtracking(nums) # 递归搜索
14                    path.pop()          # 撤销选择
15
16            backtracking(nums)
17        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(n \times n!)$ ，其中 n 为数组 `nums` 的元素个数。
- 空间复杂度： $O(n)$ 。

优化版本：



The screenshot shows a Python code editor with the following code:

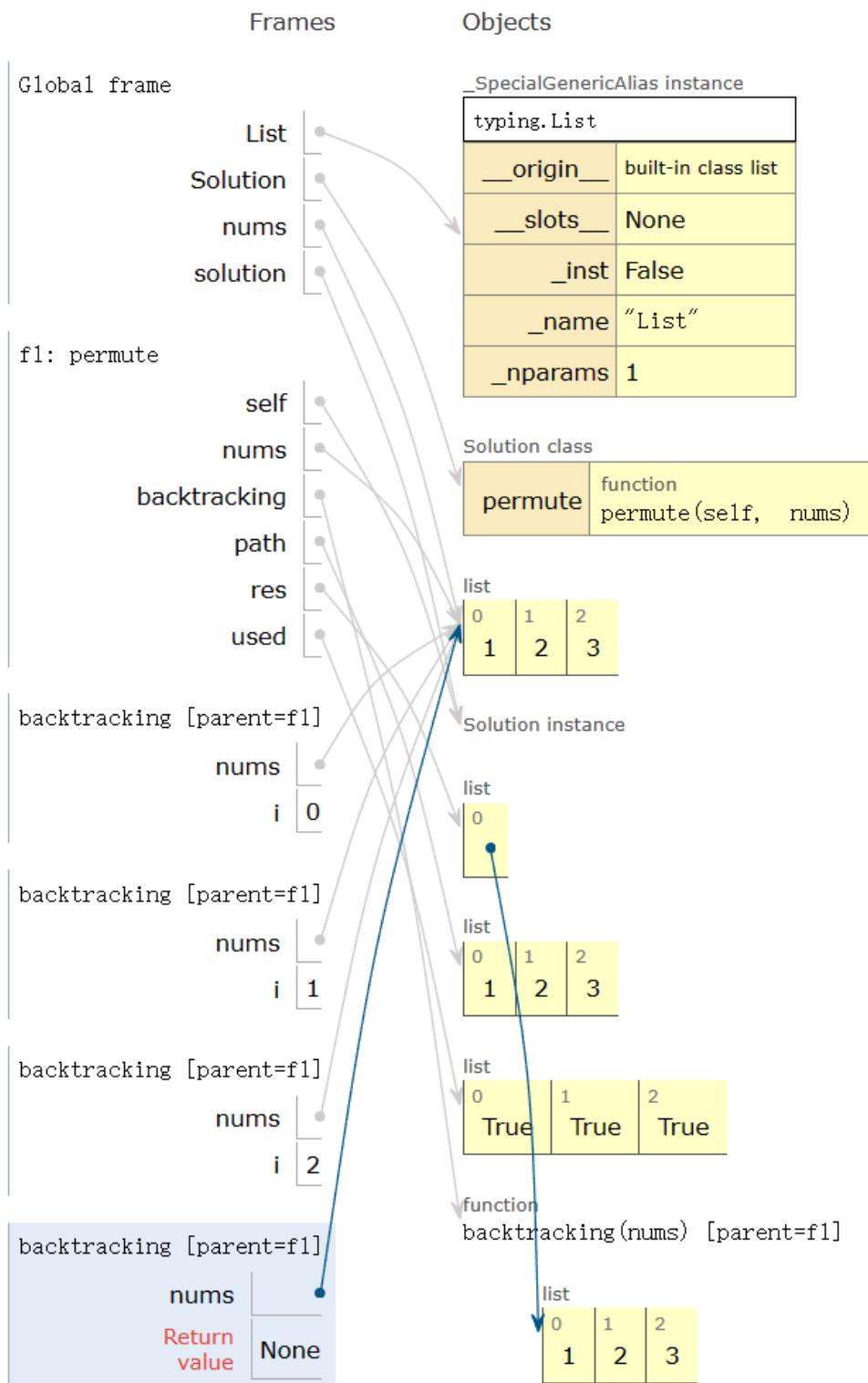
```
Python3 ⓘ 智能模式
1 class Solution:
2     def permute(self, nums: List[int]) -> List[List[int]]:
3         res = []
4         path = []
5         used = [False] * len(nums) # 标记数字是否已使用
6
7         def backtracking(nums):
8             if len(path) == len(nums):
9                 res.append(path[:])
10                return
11            for i in range(len(nums)):
12                if not used[i]: # 检查是否已使用
13                    path.append(nums[i])
14                    used[i] = True
15                    backtracking(nums)
16                    path.pop()
17                    used[i] = False # 撤销标记
18
19         backtracking(nums)
20     return res
```

2. 回溯函数 backtracking(nums):

python

复制代码

```
def backtracking(nums):
    if len(path) == len(nums): # 如果当前排列的长度等于 nums 的长度，说明找到一个完整的排列
        res.append(path[:]) # 将当前排列添加到结果集。使用 `path[:]` 是为了避免引用传递。
        return
    for i in range(len(nums)): # 遍历 nums 中的每个数字
        if not used[i]: # 如果当前数字未被使用
            path.append(nums[i]) # 将当前数字添加到 path 中
            used[i] = True # 标记该数字已被使用
            backtracking(nums) # 递归构造下一个数字
            path.pop() # 撤销选择（回溯）
            used[i] = False # 恢复标记，表示该数字可以在之后的选择中使用
```



回溯：

- 在第三次递归中，我们已经选择了 `1, 2, 3`，所以现在需要回溯。
 - 撤销对 `3` 的选择：`path.pop() → path = [1, 2]`。
 - 恢复 `used[2] = False`。
 - 继续回溯到第二次递归。

4. 继续回溯：

- 在第二次递归中，我们选择了 `1, 2`，现在需要回溯。
 - 撤销对 `2` 的选择：`path.pop() → path = [1]`。
 - 恢复 `used[1] = False`。
 - 继续回溯到第一次递归。

第一层回溯（路径：`[1]`）：

5. 选择其他未选数字：

- 在第一次递归中，撤销对 `1` 的选择：`path.pop() → path = []`。
- 恢复 `used[0] = False`，并选择其他数字，继续构造其他排列。

6. 选择第二个数字：

- 跳过 `1`，选择 `nums[1] = 2`，加入 `path = [2]`，标记 `used[1] = True`，然后继续递归。

完成全排列：

- 根据这个流程，程序会继续构建其他的排列：`[1, 3, 2]`，`[2, 1, 3]`，`[2, 3, 1]`，`[3, 1, 2]` 和 `[3, 2, 1]`。

你提到的这段代码：

python

复制代码

```
if len(path) == len(nums):  
    res.append(path[:])  
    return
```

这里的 `path[:]` 与 `path` 的区别是：

- `path[:]` 是一个**切片操作**，它会创建 `path` 列表的一个**浅拷贝**。这意味着 `res.append(path[:])` 添加的是 `path` 当前的**副本**，而不是原始的 `path` 列表。
- `path` 则是原始的列表引用，如果你直接将 `path` 添加到 `res` 中，之后 `path` 的改变（如 `pop()` 或其他操作）会影响到已经添加到 `res` 中的内容，因为它们都引用同一个列表。

所以，**为什么用 `path[:]` :**

在回溯算法中，我们通常需要保证每次递归的结果是独立的，即保存的是**当前路径的副本**。如果不使用 `path[:]`，而直接使用 `path`，你在之后的递归中对 `path` 的修改将会影响到已经保存到 `res` 中的内容。为了避免这个问题，我们使用 `path[:]` 来创建一个副本。

关键区别

特性	<code>return 0</code>	<code>return</code>
返回值	返回 <code>0</code>	返回 <code>None</code> (默认)
用途	返回具体值给调用者	仅终止函数执行
使用场景	需要明确返回值的函数	不需要返回值的函数

第 47 题 全排列 II

47. 全排列 II

中等

相关标签

相关企业

A+

给定一个可包含重复数字的序列 `nums`，按任意顺序 返回所有不重复的全排列。

示例 1：

输入: `nums = [1,1,2]`

输出:

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

示例 2：

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]]`

```
1 class Solution:  
2     def permuteUnique(self, nums: List[int]) -> List[List[int]]:  
3         res = set() # 使用集合来存储结果，自动去重 ←  
4         path = []  
5         used = [False] * len(nums)  
6  
7         def backtracking(nums):  
8             if len(path) == len(nums): ←  
9                 res.add(tuple(path)) # 将列表转为元组存入集合，自动去重  
10                return  
11            for i in range(len(nums)):  
12                if not used[i]: # 检查该元素是否被使用  
13                    path.append(nums[i])  
14                    used[i] = True  
15                    backtracking(nums)  
16                    path.pop()  
17                    used[i] = False  
18  
19            backtracking(nums) ←  
20            return [list(t) for t in res] # 将集合中的元组转为列表返回  
21
```

在 Python 中，`set` 是一个无序、不可重复的集合类型。它只能存储 **可哈希的对象**，而列表（`list`）是 **可变的对象**，因此 **不可哈希，不能直接存入 `set`**。

另一方面，**元组（tuple）是不可变的对象**，因此是可哈希的，可以作为 `set` 的元素。这就是为什么在 `res.add(tuple(path))` 中需要将列表 `path` 转换为元组来实现去重。

为什么列表不可哈希？

1. 列表是可变的，元素可以被增删或修改。
2. `set` 的核心依赖于哈希值（`hash()`），哈希值要求对象的内容不能改变。
3. 因此，列表（可变）不满足 `set` 的要求，而元组（不可变）满足。

为什么元组可以用来去重？

1. 元组一旦创建，其内容不可更改。
2. 元组可以计算哈希值，因此可以存入 `set`。
3. 将 `path` 转为元组后，每次生成的排列（元组）会被 `set` 自动去重。

第 78 题 子集

78. 子集

中等

相关标签

相关企业

A+

给你一个整数数组 `nums`，数组中的元素 **互不相同**。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1：

输入: `nums = [1, 2, 3]`

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

示例 2：

输入: `nums = [0]`

输出: `[[], [0]]`

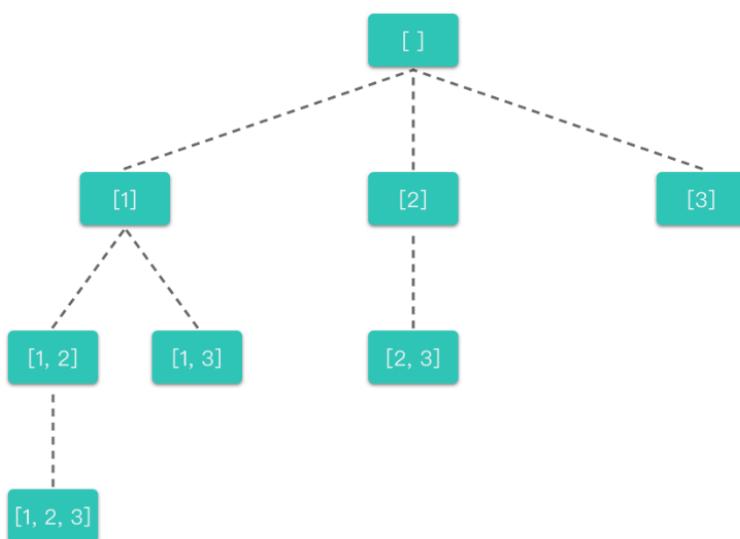
思路 1：回溯算法

数组的每个元素都有两个选择：选与不选。

我们可以通过向当前子集数组中添加可选元素来表示选择该元素。也可以在当前递归结束之后，将之前添加的元素从当前子集数组中移除（也就是回溯）来表示不选择该元素。

下面我们根据回溯算法三步走，写出对应的回溯算法。

1. 确明所有选择：根据数组中每个位置上的元素选与不选两种选择，画出决策树，如下图所示。



2. 明确终止条件：

- 当遍历到决策树的叶子节点时，就终止了。即当前路径搜索到末尾时，递归终止。

3. 将决策树和终止条件翻译成代码：

1. 定义回溯函数：

- `backtracking(nums, index)`: 函数的传入参数是 `nums` (可选数组列表) 和 `index` (代表目前正在考虑元素是 `nums[i]`)，全局变量是 `res` (存放所有符合条件结果的集合数组) 和 `path` (存放当前符合条件的结果)。
- `backtracking(nums, index)`: 函数代表的含义是：在选择 `nums[index]` 的情况下，递归选择剩下的元素。

2. 书写回溯函数主体 (给出选择元素、递归搜索、撤销选择部分)。

- 从目前正在考虑元素，到数组结束为止，枚举出所有可选的元素。对于每一个可选元素：
 - 约束条件：之前选过的元素不再重复选用。每次从 `index` 位置开始遍历而不是从 `0` 位置开始遍历就是为了避免重复。集合跟全排列不一样，子集中 `{1, 2}` 和 `{2, 1}` 是等价的。为了避免重复，我们之前考虑过的元素，就不再重复考虑了。
 - 选择元素：将其添加到当前子集数组 `path` 中。
 - 递归搜索：在选择该元素的情况下，继续递归考虑下一个位置上的元素。
 - 撤销选择：将该元素从当前子集数组 `path` 中移除。

```
1 for i in range(index, len(nums)): # 枚举可选元素列表
2     path.append(nums[i]) # 选择元素
3     backtracking(nums, i + 1) # 递归搜索
4     path.pop() # 撤销选择
```

py

3. 明确递归终止条件 (给出递归终止条件，以及递归终止时的处理方法)。

- 当遍历到决策树的叶子节点时，就终止了。也就是当正在考虑的元素位置到达数组末尾 (即 `start >= len(nums)`) 时，递归停止。
- 从决策树中也可以看出，子集需要存储的答案集合应该包含决策树上所有的节点，应该需要保存递归搜索的所有状态。所以无论是否达到终止条件，我们都应该将当前符合条件的结果放入到集合中。

思路 1：代码

```
1 class Solution:
2     def subsets(self, nums: List[int]) -> List[List[int]]:
3         res = [] # 存放所有符合条件结果的集合
4         path = [] # 存放当前符合条件的结果
5         def backtracking(nums, index): # 正在考虑可选元素列表中第 index 个元素
6             res.append(path[:]) # 将当前符合条件的结果放入集合中
7             if index >= len(nums): # 遇到终止条件 (本题)
8                 return
9
10            for i in range(index, len(nums)): # 枚举可选元素列表
11                path.append(nums[i]) # 选择元素
12                backtracking(nums, i + 1) # 递归搜索
13                path.pop() # 撤销选择
14
15        backtracking(nums, 0)
16        return res
```

思路 1：复杂度分析

- 时间复杂度：** $O(n \times 2^n)$ ，其中 n 指的是数组 `nums` 的元素个数， 2^n 指的是所有状态数。每种状态需要 $O(n)$ 的时间来构造子集。
- 空间复杂度：** $O(n)$ ，每种状态下构造子集需要使用 $O(n)$ 的空间。

第 51 题 N 皇后

51. N 皇后

困难

相关标签

相关企业

A₂

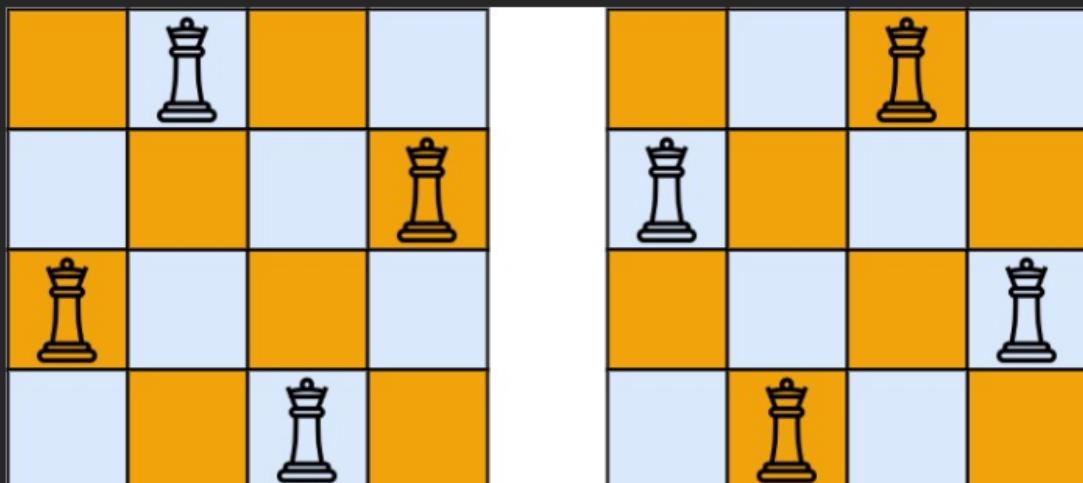
按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

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

给你一个整数 n ，返回所有不同的 **n 皇后问题** 的解决方案。

每一种解法包含一个不同的 **n 皇后问题** 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

示例 1：



输入: $n = 4$

输出: `[["Q..","...Q","Q...","..Q."],
["..Q.","Q...","...Q",".Q.."]]`

解释: 如上图所示，4 皇后问题存在两个不同的解法。

十一、二分查找

第 35 题 搜索插入位置

35. 搜索插入位置

已解答 ✓

Python3 智能模式

```
1 class Solution:
2     def searchInsert(self, nums: List[int], target: int) -> int:
3         pass
```

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

2.4K 4.6K 已存储 行 3, 列 79 人在线

思路 1：二分查找

设定左右节点为数组两端，即 `left = 0`, `right = len(nums) - 1`，代表待查找区间为 $[left, right]$ (左闭右闭)。

取两个节点中心位置 mid ，先比较中心位置值 $nums[mid]$ 与目标值 $target$ 的大小。

- 如果 $target == nums[mid]$ ，则当前中心位置为待插入数组的位置。
- 如果 $target > nums[mid]$ ，则将左节点设置为 $mid + 1$ ，然后继续在右区间 $[mid + 1, right]$ 搜索。
- 如果 $target < nums[mid]$ ，则将右节点设置为 $mid - 1$ ，然后继续在左区间 $[left, mid - 1]$ 搜索。

直到查找到目标值返回待插入数组的位置，或者等到 $left > right$ 时停止查找，此时 $left$ 所在位置就是待插入数组的位置。

注意 `mid` 是要一直变化得，所以必须在循环体内

思路 1：二分查找代码

```
1 class Solution:
2     def searchInsert(self, nums: List[int], target: int) -> int:
3         size = len(nums)
4         left, right = 0, size - 1
5
6         while left <= right:
7             mid = left + (right - left) // 2
8             if nums[mid] == target:
9                 return mid
10            elif nums[mid] < target:
11                left = mid + 1
12            else:
13                right = mid - 1
14
15        return left
```

思路 1：复杂度分析

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

第 74 题 搜索二维矩阵

74. 搜索二维矩阵

中等

相关标签

相关企业

A₂

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

例如示例 1，三行拼在一起得

$$a = [1, 3, 5, 7, 10, 11, 16, 20, 23, 30, 34, 60]$$

由于这是一个有序数组，我们可以用二分查找判断 $target$ 是否在 $matrix$ 中。

代码实现时，并不需要真的拼成一个长为 mn 的数组 a ，而是将 $a[i]$ 转换成矩阵中的行号和列号。例如示例 1， $i = 9$ 对应的 $a[i] = 30$ ，由于矩阵有 $n = 4$ 列，所以 $a[i]$ 在 $\left\lfloor \frac{i}{n} \right\rfloor = 2$ 行，在 $i \bmod n = 1$ 列。

一般地，有

$$a[i] = matrix[\lfloor i/n \rfloor][i \bmod n]$$

```
Python3 ✓ 智能模式 ✓ 口
1 class Solution:
2     def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
3         m = len(matrix)
4         n = len(matrix[0]) ←
5         left = 0
6         right = m * n - 1
7         while left <= right:
8             mid = (left + right) // 2 ←
9             value = matrix[mid // n][mid % n] # 与中间值做比较
10            if value == target:
11                return True
12            elif value < target:
13                left = mid + 1
14            else:
15                right = mid - 1 ←
16        return False
17
```

方法二：排除法

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

从右上角开始，寻找 11。

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

$7 < 11$ ，这意味着第一排的所有数都小于 11，排除。

10	11	16	20
23	30	34	60

在剩余矩阵中，比较右上角元素和 11 的大小。 $20 > 11$ ，这意味着最后一列的所有数都大于 11，排除。

10	11	16
23	30	34

在剩余矩阵中，比较右上角元素和 11 的大小。 $16 > 11$ ，这意味着最后一列的所有数都大于 11，排除。

10	11
23	30

右上角元素等于 11，找到。

Py3 | Java | C++ | C | Go | JS | Rust

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m, n = len(matrix), len(matrix[0])
        i, j = 0, n - 1
        while i < m and j >= 0: # 还有剩余元素
            if matrix[i][j] == target:
                return True # 找到 target
            if matrix[i][j] < target:
                i += 1 # 这一行剩余元素全部小于 target, 排除
            else:
                j -= 1 # 这一列剩余元素全部大于 target, 排除
        return False
```

34. 在排序数组中查找元素的第一个和最后一个位置

中等

相关标签

相关企业

A+

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`

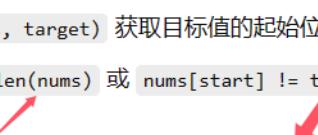
输出: `[-1,-1]`

```

1 class Solution:
2     def searchRange(self, nums: List[int], target: int) -> List[int]:
3         start=self.low_bound(nums,target)
4         if start==len(nums) or nums[start]!=target:
5             return [-1,-1]
6         # 如果 start 存在, 那么 end 必定存在
7         end=self.low_bound(nums,target+1)-1
8         return [start,end]
9     #low_bound 返回最小的满足 nums[i] >= target 的下标 i
10    def low_bound(self,nums,target):
11        left,right=0,len(nums)-1
12        while left<=right:
13            mid=left+(right-left)//2
14            if nums[mid]>=target:
15                right=mid-1
16            else:
17                left=mid+1
18        return left
19    # 循环结束后 left = right+1
20    # 此时 nums[left-1] < target 而 nums[left] = nums[right+1] >= target
21    # 所以 left 就是第一个 >= target 的元素下标

```

步骤:

1. 寻找起始位置: 使用 `lower_bound(nums, target)` 获取目标值的起始位置 `start`。
2. 检查目标值是否存在: 如果 `start == len(nums)` 或 `nums[start] != target`, 则目标值不存在于数组中, 返回 `[-1, -1]`。

3. 寻找结束位置: 使用 `lower_bound(nums, target + 1)` 来找到比目标值大的第一个元素的位置, 再减去 1 即为目标值的结束位置 `end`。

33. 搜索旋转排序数组

中等

相关标签

相关企业

A_x

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

```

1 class Solution:
2     def search(self, nums: List[int], target: int) -> int:
3         left,right=0,len(nums)-1
4         while left<=right:
5             mid=left+(right-left)//2
6             if nums[mid]==target:
7                 return mid
8             if nums[mid]>=nums[left]: #确定前半段是升序
9                 if nums[mid]>target and target>=nums[left]:
10                     right=mid-1
11                 else:
12                     left=mid+1
13             else:
14                 if nums[mid]<target and target<=nums[right]:
15                     left=mid+1
16                 else:
17                     right=mid-1
18         return -1

```

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
13																
14																
15					left		mid									right
16					4	5	6	7	8	9	1	2	3			
17																
18																
19																
20					left			mid								right
21					7	8	9	1	2	3	4	5	6			
22																
23																
24																
25																

在判断条件 `nums[mid] >= nums[left]` 中，使用 `>=` 而不是 `>` 是为了确保正确处理 **有序数组中没有旋转点的情况** 和 **只有两个元素的数组**。

详细原因：

1. 数组完全有序时：

- 如果数组本身是完全有序的（没有旋转点），例如 `nums = [1, 2, 3, 4, 5]`，中间元素 `nums[mid]` 必然大于等于左边界 `nums[left]`。
- 如果使用 `>`，当 `nums[mid] == nums[left]` 时，算法可能错误地认为右半部分是有序的，而实际上整个数组都是有序的，这可能会导致不正确的结果。

2. 边界条件处理：

- 当数组只有两个元素时，比如 `nums = [2, 3]`，如果目标值是 `2` 或 `3`，`nums[mid] >= nums[left]` 能正确处理这类情况。
- 若使用 `>`，当 `nums[mid] == nums[left]` 时会跳过这部分逻辑，可能导致错误。

3. 支持重复元素：

- 在某些扩展问题中（如允许重复元素的旋转排序数组），`nums[mid] == nums[left]` 的情况更常见，例如 `nums = [2, 2, 3, 4, 5]`。使用 `>=` 可以更稳健地判断当前段是否有序。

核心总结

无论左右两侧是否都局部有序，重点在于：

1. 只有一个区间会包含目标值。
2. 利用旋转数组的特性和有序区间，可以快速判断目标值可能所在的区间，缩小查找范围。

即使左右都各自有序，我们依然必须通过判断，来决定“查左还是查右”。

153. 寻找旋转排序数组中的最小值

中等

相关标签

相关企业

提示

A_文

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

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

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

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

你必须设计一个时间复杂度为 $O(\log n)$ 的算法解决此问题。

示例 1：

输入: `nums = [3,4,5,1,2]`

输出: 1

解释: 原数组为 `[1,2,3,4,5]`，旋转 3 次得到输入数组。

示例 2：

输入: `nums = [4,5,6,7,0,1,2]`

输出: 0

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

- 初始化左右指针 `left` 和 `right` 分别指向数组的起始和结束位置。
- 每次计算中间位置 `mid`，根据 `nums[mid]` 与 `nums[right]` 的比较来确定最小值可能在哪一部分。
- 若 `nums[mid] > nums[right]`，说明最小值在右半部分，将 `left` 更新为 `mid + 1`。
- 若 `nums[mid] <= nums[right]`，说明最小值在左半部分或者就是 `nums[mid]` 本身，将 `right` 更新为 `mid`。
- 最终当 `left == right` 时，即找到了最小值的位置。

思路 1：代码

```

1  class Solution:
2      def findMin(self, nums: List[int]) -> int:
3          left = 0
4          right = len(nums) - 1
5          while left < right:
6              mid = left + (right - left) // 2
7              if nums[mid] > nums[right]:
8                  left = mid + 1
9              else:
10                  right = mid
11      return nums[left]

```

思路 1：复杂度分析

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

十二、栈

第 20 题 有效的括号

20. 有效的括号

已解答

简单 相关标签 相关企业 提示 文

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

代码 | 笔记

Python3 智能模式

```
1 class Solution:  
2     def isValid(self, s: str) -> bool:  
3         pass
```

Python3 ▼ 智能模式

```
1 class Solution:
2     def isValid(self, s: str) -> bool:
3         while("()" in s or "{}" in s or "[]" in s):
4             s = s.replace("()", "")
5             s = s.replace("{}", "")
6             s = s.replace("[]", "")
7         return s == ''
```

性能问题

虽然这种方法能够正确处理问题，但它的时间复杂度比较高。`replace` 操作在每次循环中都会遍历整个字符串，这样可能导致在处理长字符串时性能较差。具体来说，假设字符串的长度为 n ，那么 `replace` 操作的时间复杂度是 $O(n)$ ，并且循环会执行多次（可能接近 $n/2$ 次），因此总的时间复杂度大约是 $O(n^2)$ 。

更优化的方法

通过使用栈，可以在 $O(n)$ 时间复杂度内完成这个问题。栈可以帮助我们实时追踪开放的括号，并且只在遇到匹配的闭括号时进行弹栈操作。以下是优化后的实现：

```
python
```

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        mapping = {')': '(', '}': '{', ']': '['}

        for char in s:
            if char in mapping: # 如果是闭括号
                top_element = stack.pop() if stack else '#'
                if mapping[char] != top_element:
                    return False
            else: # 如果是开括号
                stack.append(char)

        return not stack # 如果栈为空，说明括号有效
```

复制代码

举个例子：

假设栈为空，当前字符是 ')'。

- 代码会执行 `top_element = stack.pop() if stack else '#'`。
- 因为栈为空，`stack.pop()` 不会被执行，`top_element` 会被赋值为 `#`。
- 之后，程序会检查 `#` 是否与 `')'` 匹配。由于 `#` 不可能匹配任何有效的括号，所以 `mapping[')']` 不会是 `#`，最终返回 `False`。

例子说明：

假设我们有一个字符串 `s = "()"`，并且已经从左到右遍历到第一个 `)`：

1. 栈中有 `(`：

- 此时栈不是空的，`top_element = stack.pop()` 会弹出栈顶的 `(`。
- 比较 `mapping[')']` 和 `(`，它们匹配，所以继续检查下一个字符。

2. 栈为空时遇到 `)`：

- 如果栈为空（比如 `s = ")"`），`top_element` 会被赋值为 `#`，这样我们就能知道此时没有配对的开括号，从而返回 `False`。

总结：

`top_element = stack.pop() if stack else '#'` 的目的是确保在栈为空时不会出现错误，并使用一个无效的占位符 `#` 来避免不合法的匹配，进而保证代码的健壮性。

第 155 题 最小栈

155. 最小栈

中等 相关标签 相关企业 提示 A_z

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","getM
in"]
[[],[-2],[0],[-3],[],[],[],[]]
```

输出:

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

解释:

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

解题思路

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

思路 1：辅助栈

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

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

思路 1：代码

```
1 class MinStack:  
2  
3     def __init__(self):  
4         self.stack = []  
5         self.minstack = []  
6  
7     def push(self, val: int) -> None:  
8         if not self.stack:  
9             self.stack.append(val)  
10            self.minstack.append(val)  
11        else:  
12            self.stack.append(val)  
13            self.minstack.append(min(val, self.minstack[-1]))  
14  
15    def pop(self) -> None:  
16        self.stack.pop()  
17        self.minstack.pop()  
18  
19    def top(self) -> int:  
20        return self.stack[-1]  
21  
22    def getMin(self) -> int:  
23        return self.minstack[-1]
```

思路 1：复杂度分析

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

为什么要同步弹出呢？

- **保证 `mystack` 始终是当前栈的最小值：** `mystack` 保存的是一个辅助的最小值栈，它的栈顶始终是当前栈中最小的元素。如果不同步弹出，当你从 `stack` 中移除元素时，`mystack` 可能会保存一个已经不在 `stack` 中的值，导致 `getMin()` 返回错误的结果。
- **正确的最小值查询：** 假设我们没有同步弹出 `mystack`，当栈中剩下的元素并不是栈中最小值时，`mystack` 仍然会保存一个过时的最小值。如果我们调用 `getMin()`，返回的将不是当前栈的最小值。

示例：

假设我们执行以下操作：

1. `push(5)` → `stack = [5]`, `mystack = [5]`
2. `push(3)` → `stack = [5, 3]`, `mystack = [5, 3]`
3. `push(7)` → `stack = [5, 3, 7]`, `mystack = [5, 3, 3]`
4. `pop()` → `stack = [5, 3]`, `mystack = [5, 3]`
5. **如果不同步弹出：** `pop()` → `stack = [5]`, `mystack = [5, 3]` (错误, `mystack` 不再正确)
 - 如果我们调用 `getMin()`，会返回 `3`，但栈中只有 `5`，这个值显然不正确。

如果我们正确地同步弹出：

1. `pop()` → `stack = [5]`, `mystack = [5]` (正确)

最终，`mystack` 中的栈顶元素总是当前栈的最小元素，从而保证了 `getMin()` 总是返回正确的结果。

394. 字符串解码

中等

相关标签

相关企业

A+

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: $k[encoded_string]$ ，表示其中方括号内部的 $encoded_string$ 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 $3a$ 或 $2[4]$ 的输入。

示例 1:

输入: $s = "3[a]2[bc]"$

输出: "aaabcbc"

示例 2:

输入: $s = "3[a2[c]]"$

输出: "accaccacc"

示例 3:

输入: $s = "2[abc]3[cd]ef"$

输出: "abcabccdccdcdef"

示例 4:

输入: $s = "abc3[cd]xyz"$

输出: "abccdcdcdxyz"

思路 1：栈

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

思路 1：代码

```
1 class Solution:
2     def decodeString(self, s: str) -> str:
3         stack1 = []
4         stack2 = []
5         num = 0
6         res = ""
7         for ch in s:
8             if ch.isdigit():
9                 num = num * 10 + int(ch) ←
10            elif ch == '[':
11                stack1.append(res)
12                stack2.append(num)
13                res = ""
14                num = 0
15            elif ch == ']':
16                cur_res = stack1.pop()
17                cur_num = stack2.pop()
18                res = cur_res + res * cur_num ←
19            else:
20                res += ch ←
21        return res
```

思路 1：复杂度分析

- 时间复杂度： $O(n)$ 。
- 空间复杂度： $O(n)$ 。

示例：

考虑输入字符串 `s = "3[a2[c]]"`：

1. 读取 `3`：`num` 变成 `3`。
2. 读取 `[`：将当前的 `res`（为空）和 `num`（3）推入 `stack1` 和 `stack2`，然后重置 `res` 和 `num`。
3. 读取 `a`：将 `a` 添加到 `res`，此时 `res = "a"`。
4. 读取 `2`：`num` 变成 `2`。
5. 读取 `[`：将当前的 `res`（"a"）和 `num`（2）推入 `stack1` 和 `stack2`，然后重置 `res` 和 `num`。
6. 读取 `c`：将 `c` 添加到 `res`，此时 `res = "c"`。
7. 读取 `]`：从 `stack1` 和 `stack2` 中弹出 `cur_res = "a"` 和 `cur_num = 2`，然后 `res = "a" + "c" * 2 = "acc"`。
8. 读取 `]`：从 `stack1` 和 `stack2` 中弹出 `cur_res = ""` 和 `cur_num = 3`，然后 `res = "" + "acc" * 3 = "accaccacc"`。

最终，返回结果 `"accaccacc"`。

具体操作：

以 `s = "3[a2[c]]"` 为例：

1. `push(3)`：`num = 3`
2. `push([])`：将 `res = ""` 和 `num = 3` 入栈，重置 `res = ""` 和 `num = 0`。
3. `push(a)`：`res = "a"`
4. `push(2)`：`num = 2`
5. `push([])`：将 `res = "a"` 和 `num = 2` 入栈，重置 `res = ""` 和 `num = 0`。
6. `push(c)`：`res = "c"`
7. `push([])`：弹出 `stack1` 和 `stack2`，得到 `"a"` 和 `2`，然后将 `"c"` 重复 2 次，得到 `"cc"`，最终 `res = "acc"`。
8. `push([])`：弹出 `stack1` 和 `stack2`，得到 `""` 和 `3`，然后将 `"acc"` 重复 3 次，得到 `"accaccacc"`，最终结果。

739. 每日温度

中等

相关标签

相关企业

提示

A_文

给定一个整数数组 `temperatures`，表示每天的温度，返回一个数组 `answer`，其中 `answer[i]` 是指对于第 `i` 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

示例 1:

输入: `temperatures` =
[73, 74, 75, 71, 69, 72, 76, 73]
输出: [1, 1, 4, 2, 1, 1, 0, 0]

示例 2:

输入: `temperatures` = [30, 40, 50, 60]
输出: [1, 1, 1, 0]

示例 3:

输入: `temperatures` = [30, 60, 90]
输出: [1, 1, 0]

思路 1：单调栈

- 首先，将答案数组 `ans` 全部赋值为 0。然后遍历数组每个位置元素。
- 如果栈为空，则将当前元素的下标入栈。
- 如果栈不为空，且当前数字大于栈顶元素对应数字，则栈顶元素出栈，并计算下标差。
- 此时当前元素就是栈顶元素的下一个更高值，将其下标差存入答案数组 `ans` 中保存起来，判断栈顶元素。
- 直到当前数字小于或等于栈顶元素，则停止出栈，将当前元素下标入栈。
- 最后输出答案数组 `ans`。

```
class Solution:  
    def dailyTemperatures(self, T: List[int]) -> List[int]:  
        n = len(T) # 获取温度数组的长度  
        stack = [] # 单调递减栈，存储索引  
        ans = [0 for _ in range(n)] # 结果数组，初始值全部为 0  
  
        for i in range(n): # 遍历每一天的温度  
            while stack and T[i] > T[stack[-1]]: # 栈不为空，且当前温度比栈顶索引的温度高  
                index = stack.pop() # 弹出栈顶索引  
                ans[index] = (i - index) # 计算等待天数  
                stack.append(i) # 当前索引入栈  
  
        return ans # 返回结果数组
```

代码执行过程

我们通过 `T = [73, 74, 75, 71, 69, 72, 76, 73]` 具体分析代码执行过程。

Step 1：初始化

- `n = 8`
- `stack = []`
- `ans = [0, 0, 0, 0, 0, 0, 0, 0]`

Step 2：遍历 `T`

`i = 0, T[0] = 73`

- `stack` 为空，直接入栈
- `stack = [0]`

`i = 1, T[1] = 74`

- `T[1] > T[stack[-1]]`，即 `74 > 73`，说明 `T[0]` 找到了更高的温度
- `index = stack.pop()`, `index = 0`
- `ans[0] = 1 - 0 = 1`
- `stack = []`
- `stack.append(1)`
- `stack = [1]`
- 更新 `ans`: `[1, 0, 0, 0, 0, 0, 0, 0]`**

单调栈的分类

单调栈主要有两种类型：

1. 单调递增栈 (Monotonically Increasing Stack)

- 栈内元素单调递增 (从栈底到栈顶)
- 遇到更大的元素时，栈顶元素会出栈
- 适用于 找“下一个更大元素”的问题，例如 每日温度问题

2. 单调递减栈 (Monotonically Decreasing Stack)

- 栈内元素单调递减 (从栈底到栈顶)
- 遇到更小的元素时，栈顶元素会出栈
- 适用于 找“下一个更小元素”的问题

单调栈的应用场景

1. 下一个更大/更小元素问题

- 例子：给定数组，求每个元素的右侧第一个更大/更小的数

2. 股票问题

- 例子：求每一天股票价格比当前高的最近一天

3. 直方图最大矩形面积问题

- 例子：计算最大矩形面积 (单调递增栈)

4. 每日温度问题 (Daily Temperatures)

- 例子：求每一天需要等多少天才能遇到更高的温度

十三、堆

第 215 题 数组中的第 K 个最大元素

215. 数组中的第K个最大元素

已解答

中等

相关标签

相关企业

A+

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

```
Python3 ▼ 智能模式
1 class Solution:
2     def findKthLargest(self, nums: List[int], k: int) -> int:
3
4
5         nums.sort(reverse=True)
6         return nums[k-1]
7
8         # res=sorted(nums,reverse=True)
9         # return res[k-1]
10
```

第 347 题 前 K 个高频元素

347. 前 K 个高频元素

已解答

中等 相关标签 相关企业

给你一个整数数组 `nums` 和一个整数 `k`，请你返回其中出现频率前 `k` 高的元素。你可以按 **任意顺序** 返回答案。

示例 1:

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

示例 2:

输入: `nums = [1]`, `k = 1`
输出: `[1]`

```
Python3 智能模式
```

```
1 class Solution:
2     def topKFrequent(self, nums: List[int], k: int) -> List[int]:
3         # return [item for item, freq in Counter(nums).most_common(k)]
4         nums_dict={}
5         for num in nums:
6             if num in nums_dict:
7                 nums_dict[num]+=1
8             else:
9                 nums_dict[num]=1
10        sorted_nums_dict=dict(sorted(nums_dict.items(),key=lambda item:item[1],reverse=True))
11        result=list(sorted_nums_dict.keys())[:k] #列表的切片操作
12        return result
```

十四、贪心算法

第 121 题 买卖股票的最佳时机

121. 买卖股票的最佳时机

[简单](#) [相关标签](#) [相关企业](#) [Ax](#)

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

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

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

示例 1:

输入: `[7,1,5,3,6,4]`

输出: `5`

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

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

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。

[代码](#) | [笔记](#) ×

Python3 ▾ [智能模式](#)

```
1 class Solution:
2     def maxProfit(self, prices: List[int]) -> int:
3         minprice=float("inf")
4         maxprofit=0
5         for price in prices:
6             if price<minprice:
7                 minprice=price
8             elif price-minprice>maxprofit:
9                 maxprofit=price-minprice
10            return maxprofit
```

更新最低价格

python

复制代码

```
if price < minprice:  
    minprice = price
```

- 如果当前价格 `price` 小于已知的最低价格 `minprice`，更新 `minprice` 为当前价格。
- 这一逻辑确保我们总是用最便宜的价格作为买入价。

计算并更新最大利润

python

复制代码

```
elif price - minprice > maxprofit:  
    maxprofit = price - minprice
```

- 如果当前价格 `price` 与最低价格 `minprice` 的差值大于 `maxprofit`，更新 `maxprofit` 为新的差值。
- 这一逻辑确保我们总是记录最大的利润。 

思路 1：递推

- 设置两个变量 `minprice`（用来记录买入的最小值）、`maxprofit`（用来记录可获取的最大利润）。
- 从左到右进行遍历数组 `prices`。
- 如果遇到当前价格比 `minprice` 还要小的，就更新 `minprice`。
- 如果遇到当前价格大于或者等于 `minprice`，则判断一下以当前价格卖出的话能卖多少，如果比 `maxprofit` 还要大，就更新 `maxprofit`。
- 最后输出 `maxprofit`。

思路 1：代码

```
1 class Solution:  
2     def maxProfit(self, prices: List[int]) -> int:  
3         minprice = 10010  
4         maxprofit = 0  
5         for price in prices:  
6             if price < minprice:  
7                 minprice = price  
8             elif price - minprice > maxprofit:  
9                 maxprofit = price - minprice  
10        return maxprofit
```

复制代码

思路 1：复杂度分析

- 时间复杂度：** $O(n)$ ，其中 n 是数组 `prices` 的元素个数。
- 空间复杂度：** $O(1)$ 。

122. 买卖股票的最佳时机 II

中等

相关标签

相关企业

A+

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。

第 55 题 跳跃游戏

55. 跳跃游戏

中等

相关标签

相关企业

A_x

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

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

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: `true`

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

示例 2:

输入: `nums = [3,2,1,0,4]`

输出: `false`

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

思路 1：贪心算法

如果我们能通过前面的某个位置 j ，到达后面的某个位置 i ，则我们一定能到达区间 $[j, i]$ 中所有的点 ($j \leq i$)。

而前面的位置 j 肯定也是通过 j 前面的点到达的。所以我们可以通过贪心算法来计算出所能到达的最远位置。具体步骤如下：

1. 初始化能到达的最远位置 max_i 为 0。
2. 遍历数组 `nums`。
3. 如果能到达当前位置，即 $max_i \leq i$ ，并且当前位置 + 当前位置最大跳跃长度 > 能到达的最远位置，即 $i + nums[i] > max_i$ ，则更新能到达的最远位置 max_i 。
4. 遍历完数组，最后比较能到达的最远位置 max_i 和数组最远距离 `size - 1` 的关系。如果 $max_i \geq len(nums)$ ，则返回 `True`，否则返回 `False`。

思路 1：代码

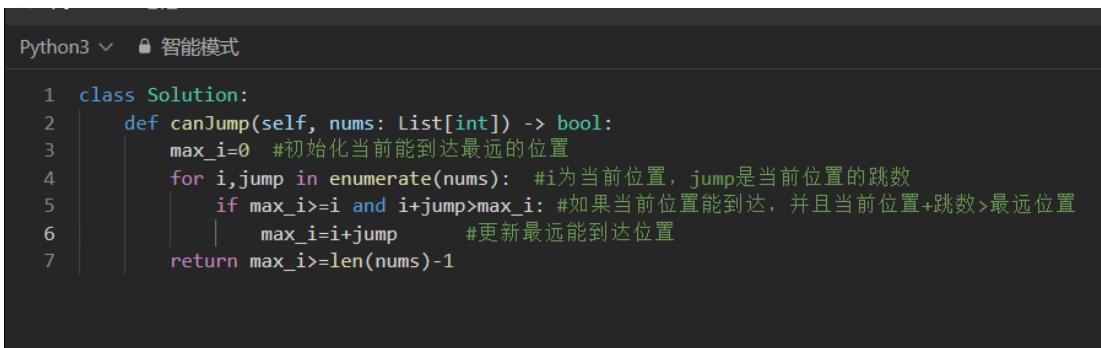
```
1 class Solution:
2     def canJump(self, nums: List[int]) -> bool:
3         size = len(nums)
4         max_i = 0
5         for i in range(size):
6             if max_i >= i and i + nums[i] > max_i:
7                 max_i = i + nums[i]
8
9     return max_i >= size - 1
```

思路 1：复杂度分析

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

本题的重点是：

到达最远的位置，就一定能到达它前面的位置



```
Python3 ▾ 智能模式
1 class Solution:
2     def canJump(self, nums: List[int]) -> bool:
3         max_i=0 #初始化当前能到达最远的位置
4         for i,jump in enumerate(nums): #i为当前位置, jump是当前位置的跳数
5             if max_i>=i and i+jump>max_i: #如果当前位置能到达, 并且当前位置+跳数>最远位置
6                 max_i=i+jump #更新最远能到达位置
7         return max_i>=len(nums)-1
```

条件判断

python

复制代码

```
if max_i >= i and i + jump >= max_i:
```

- `max_i >= i`: 当前索引 `i` 必须在可以到达的范围内。如果 `i` 超过了 `max_i`，说明当前位置不可达，应停止进一步检查。
- `i + jump >= max_i`: 如果从当前位置 `i` 的跳跃距离可以超过当前的最远位置 `max_i`，更新 `max_i` 的值。

第 45 题 跳跃游戏 II

45. 跳跃游戏 II

中等

相关标签

相关企业

Ax

给定一个长度为 n 的 0 索引整数数组 nums 。初始位置为 $\text{nums}[0]$ 。

每个元素 $\text{nums}[i]$ 表示从索引 i 向前跳转的最大长度。换句话说，如果你在 $\text{nums}[i]$ 处，你可以跳转到任意 $\text{nums}[i + j]$ 处：

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

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

示例 1:

输入: $\text{nums} = [2, 3, 1, 1, 4]$

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: $\text{nums} = [2, 3, 0, 1, 4]$

输出: 2

核心思路

这段代码采用了贪心策略，每次只关心当前能跳跃到的最远位置，并在必须跳跃时更新跳跃范围。贪心的关键点是：

1. 在每一步中，动态更新最远可达的位置。
2. 只有在当前步数无法覆盖更多的位置时，才增加跳跃次数。

思路 2：贪心算法

如果第 i 个位置所能跳到的位置为 $[i + 1, i + \text{nums}[i]]$ ，则：

- 第 0 个位置所能跳到的位置就是 $[0 + 1, 0 + \text{nums}[0]]$ ，即 $[1, \text{nums}[0]]$ 。
- 第 1 个位置所能跳到的位置就是 $[1 + 1, 1 + \text{nums}[1]]$ ，即 $[2, 1 + \text{nums}[1]]$ 。
-

对于每一个位置 i 来说，所能跳到的所有位置都可以作为下一个起跳点，为了尽可能使用最少的跳跃次数，所以我们应该使得下一次起跳所能达到的位置尽可能的远。简单来说，就是每次在「可跳范围」内选择可以使下一次跳的更远的位置。这样才能获得最少跳跃次数。具体做法如下：

1. 维护几个变量：当前所能达到的最远位置 end ，下一步所能跳到的最远位置 max_pos ，最少跳跃次数 steps 。
2. 遍历数组 nums 的前 $\text{len}(\text{nums}) - 1$ 个元素：
3. 每次更新第 i 位置下一步所能跳到的最远位置 max_pos 。
4. 如果索引 i 到达了 end 边界，则：更新 end 为新的当前位置 max_pos ，并令步数 steps 加 1。
5. 最终返回跳跃次数 steps 。

思路 2：贪心算法代码

```
1 class Solution:
2     def jump(self, nums: List[int]) -> int:
3         end, max_pos = 0, 0
4         steps = 0
5         for i in range(len(nums) - 1):
6             max_pos = max(max_pos, nums[i] + i)
7             if i == end:
8                 end = max_pos
9                 steps += 1
10            return steps
```

py

思路 2：复杂度分析

- **时间复杂度**: $O(n)$ 。一重循环遍历的时间复杂度是 $O(n)$ ，所以总体时间复杂度为 $O(n)$ 。
- **空间复杂度**: $O(1)$ 。只用到了常数项的变量，所以总体空间复杂度为 $O(1)$ 。

第 763 题 划分字母区间

763. 划分字母区间

中等 相关标签 相关企业 提示 A_z

给你一个字符串 s 。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。

注意，划分结果需要满足：将所有划分结果按顺序连接，得到的字符串仍然是 s 。

返回一个表示每个字符串片段的长度的列表。

示例 1：

输入: $s = "ababcbacadefegdehijhklij"$

输出: [9, 7, 8]

解释:

划分结果为 "ababcaca"、"defegde"、"hijhklij"。

每个字母最多出现在一个片段中。

像 "ababcbacadefegde"，"hijhklij" 这样的划分是错误的，因为划分的片段数较少。

示例 2：

输入: $s = "eccbbbdec"$

输出: [10]

8 回复



jml

发布于 北京 2022.08.10

@维瓦尔第小迷弟：即最多能砍 s 多少刀？使得同一字母都出现在一个片段，且片段最多。

2 回复 分享 ...

```

1 class Solution:
2     def partitionLabels(self, s: str) -> List[int]:
3         dic = {s: index for index, s in enumerate(s)}    #存储某个字母对应地最后出现位置
4         num = 0    #直接计数
5         result = []
6         j = dic[s[0]]    #第一个字符的最后位置
7
8         for i in range(len(s)):    #逐个遍历
9             num += 1    #找到一个就加1个长度
10            if dic[s[i]] > j:   #如果最后位置比刚才的大，就更新最后位置
11                j = dic[s[i]]
12            if i == j:    #找到这一段的结束了，就说明当前位置的index和这个字母在字典里是一样的
13                result.append(num)    #加入result
14                num = 0    #归0
15        return result

```

思路

1. 遍历 `s` 记录每个字母最后出现的位置
2. 维护一个当前子串的最远边界 `j`
3. 遍历 `s`, 不断更新 `j`, 当 `i == j` 时, 找到一个子串
4. 存入 `result`, 继续计算下一个子串

关键点

- 贪心策略**: 确保每个字符都出现在一个子串中, 不提前截断
- 哈希表存储字符最后位置**: 快速查找子串的边界
- 单次遍历计算**: $O(n)$ 复杂度, 效率极高

为什么记录的是最后一次出现的位置?

理解 `enumerate(s)`

`enumerate(s)` 会遍历字符串 `s`, 返回 (索引, 字符) 对:

python

复制 编辑

```
S = "ababcbacadefegdehijhkllij"
```

`enumerate(s)` 产生的序列: 

arduino

复制 编辑

```
(0, 'a'), (1, 'b'), (2, 'a'), (3, 'b'), (4, 'c'), (5, 'b'), (6, 'a'),  
(7, 'c'), (8, 'a'), (9, 'd'), (10, 'e'), (11, 'f'), (12, 'e'), (13, 'g'),  
(14, 'd'), (15, 'e'), (16, 'h'), (17, 'i'), (18, 'j'), (19, 'h'),  
(20, 'k'), (21, 'l'), (22, 'i'), (23, 'j')
```

如何构建 `dic`

字典 `dic` 使用字典推导式:

python

复制 编辑

```
dic = {s: index for index, s in enumerate(S)}
```

相同的字符会被多次赋值, 每次都会覆盖之前的值, 最终 `dic` 只会保留 每个字符最后一次出现的索引。

示例: 构造 `dic`

遍历过程中: 

- 遇到 `a = 0`, 记录 `dic['a'] = 0`
- 遇到 `a = 2`, 更新 `dic['a'] = 2`
- 遇到 `a = 6`, 更新 `dic['a'] = 6`
- 遇到 `a = 8`, 最终 `dic['a'] = 8` 

最终 `dic` 结果:

python

复制 编辑

```
{  
    'a': 8, 'b': 5, 'c': 7, 'd': 14, 'e': 15, 'f': 11,  
    'g': 13, 'h': 19, 'i': 22, 'j': 23, 'k': 20, 'l': 21  
}
```

这里可以看到 每个字符只存了最后出现的位置。 

第三步：遍历 s

i	s[i]	num	dic[s[i]]	j 更新	i == j ?	片段长度 result
0	'a'	1	8	8	✗	[]
1	'b'	2	5	8	✗	[]
2	'a'	3	8	8	✗	[]
3	'b'	4	5	8	✗	[]
4	'c'	5	7	8	✗	[]
5	'b'	6	5	8	✗	[]
6	'a'	7	8	8	✗	[]
7	'c'	8	7	8	✗	[]
8	'a'	9	8	8 ✓	✓	[9]
9	'd'	1	14	14	✗	[9]
10	'e'	2	15	15	✗	[9]
11	'f'	3	11	15	✗	[9]
12	'e'	4	15	15	✗	[9]
13	'g'	5	13	15	✗	[9]
14	'd'	6	14	15	✗	[9]
15	'e'	7	15	15 ✓	✓	[9, 7]
16	'h'	1	19	19	✗	[9, 7]
17	'i'	2	22	22	✗	[9, 7]
18	'j'	3	23	23	✗	[9, 7]
19	'h'	4	19	23	✗	[9, 7]
20	'k'	5	20	23	✗	[9, 7]
21	'l'	6	21	23	✗	[9, 7]
22	'i'	7	22	23	✗	[9, 7]
23	'j'	8	23	23 ↓	✓	[9, 7, 8]

十五、动态规划

第 70 题 爬楼梯

70. 爬楼梯

尝试过

[简单](#) [相关标签](#) [相关企业](#) [提示](#) [A*](#)

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

思路 2：动态规划

1. 划分阶段

按照台阶的层数进行划分为 $0 \sim n$ 。

2. 定义状态

定义状态 $dp[i]$ 为：爬到第 i 阶台阶的方案数。

3. 状态转移方程

根据题目大意，每次只能爬 1 或 2 个台阶。则第 i 阶楼梯只能从第 $i - 1$ 阶向上爬 1 阶上来，或者从第 $i - 2$ 阶向上爬 2 阶上来。所以可以推出状态转移方程为 $dp[i] = dp[i - 1] + dp[i - 2]$ 。

4. 初始条件

- 第 0 层台阶方案数：可以看做 1 种方法（从 0 阶向上爬 0 阶），即 $dp[0] = 1$ 。
- 第 1 层台阶方案数：1 种方法（从 0 阶向上爬 1 阶），即 $dp[1] = 1$ 。
- 第 2 层台阶方案数：2 种方法（从 0 阶向上爬 2 阶，或者从 1 阶向上爬 1 阶）。

5. 最终结果

根据状态定义，最终结果为 $dp[n]$ ，即爬到第 n 阶台阶（即楼顶）的方案数为 $dp[n]$ 。

思路 2：代码

```
1 class Solution:
2     def climbStairs(self, n: int) -> int:
3         dp = [0 for _ in range(n + 1)]
4         dp[0] = 1
5         dp[1] = 1
6         for i in range(2, n + 1):
7             dp[i] = dp[i - 1] + dp[i - 2]
8
9     return dp[n]
```

思路 2：复杂度分析

- **时间复杂度：** $O(n)$ 。一重循环遍历的时间复杂度为 $O(n)$ 。
- **空间复杂度：** $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。因为 $dp[i]$ 的状态只依赖于 $dp[i - 1]$ 和 $dp[i - 2]$ ，所以可以使用 3 个变量来分别表示 $dp[i]$ 、 $dp[i - 1]$ 、 $dp[i - 2]$ ，从而将空间复杂度优化到 $O(1)$ 。

图示理解

以 $n = 5$ 为例：

- `dp[0] = 1` (地面, 不动)
 - `dp[1] = 1` (一步到达第 1 阶)
 - `dp[2] = dp[1] + dp[0] = 1 + 1 = 2` (两种方法: 从地面迈两步, 或从第 1 阶迈一步)
 - `dp[3] = dp[2] + dp[1] = 2 + 1 = 3` (三种方法)
 - `dp[4] = dp[3] + dp[2] = 3 + 2 = 5` (五种方法)
 - `dp[5] = dp[4] + dp[3] = 5 + 3 = 8` (八种方法)

第 118 题 杨辉三角

118. 杨辉三角

简单 相关标签 相关企业 A_x

给定一个非负整数 $\textit{ numRows}$ ，生成「杨辉三角」的前 $\textit{ numRows}$ 行。

在「杨辉三角」中，每个数是它左上方和右上方的数的和。



示例 1：

输入: numRows = 5

输出: [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]

示例 2：

输入: numRows = 1

输出： [[1]]

把杨辉三角的每一排左对齐：

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

设要计算的二维数组是 c ，计算方法如下：

- 每一排的第一个数和最后一个数都是 1，即 $c[i][0] = c[i][i] = 1$ 。
- 其余数字，等于左上方的数，加上正上方的数，即 $c[i][j] = c[i - 1][j - 1] + c[i - 1][j]$ 。例如 $4 = 1 + 3$, $6 = 3 + 3$ 等。

Python3 | Java | C++ | C | Go | JavaScript | Rust

```
class Solution:
    def generate(self, numRows: int) -> List[List[int]]:
        c = [[1] * (i + 1) for i in range(numRows)]
        for i in range(2, numRows):
            for j in range(1, i):
                # 左上方的数 + 正上方的数
                c[i][j] = c[i - 1][j - 1] + c[i - 1][j]
        return c
```

复杂度分析

- 时间复杂度： $\mathcal{O}(numRows^2)$ 。
- 空间复杂度： $\mathcal{O}(1)$ 。返回值不计入。

1. `dp=[[1]*(i+1) for i in range(numRows)]`

- 初始化杨辉三角的 `dp`，每一行的元素数量为 `i+1`，初始值都为 1。
- 例如，当 `numRows=4` 时，`dp` 初始化为：

csharp

复制代码

```
[  
    [1],      # 第 0 行  
    [1, 1],   # 第 1 行  
    [1, 1, 1], # 第 2 行  
    [1, 1, 1] # 第 3 行  
]
```

2. `for i in range(1, numRows):`

- 外层循环控制行数，从第 1 行 (`i=1`) 开始，因为第 0 行已经初始化为 `[1]`，无需再处理。

3. `for j in range(1, i):`

- 内层循环控制列数，从第 1 列 (`j=1`) 到第 `i-1` 列（不包括最后一列 `j=i`），因为首尾元素固定为 1，无需修改。
- 例如，在第 3 行 (`i=3`) 时，`j` 的取值范围是 `[1, 2]`。

198. 打家劫舍

[中等](#) [相关标签](#) [相关企业](#) [A+](#)

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。

思路 1：动态规划

1. 划分阶段

按照房屋序号进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：前 i 间房屋所能偷窃到的最高金额。

3. 状态转移方程

i 间房屋的最后一个房子是 $nums[i - 1]$ 。

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

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

然后这两种状态取最大值即可，即状态转移方程为：

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

4. 初始条件

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

5. 最终结果

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

The screenshot shows a Python code editor with the following code:

```
1 class Solution:
2     def rob(self, nums: List[int]) -> int:
3         dp=[0 for _ in range(len(nums)+1)]
4         dp[1]=nums[0]
5         for i in range(2,len(nums)+1):
6             dp[i]=max(dp[i-1],dp[i-2]+nums[i-1])
7         return max(dp)
```

The code defines a class `Solution` with a method `rob` that takes a list of integers `nums` and returns an integer. The method initializes a list `dp` of size `len(nums)+1` with all elements set to 0. It then sets `dp[1]` to `nums[0]`. A loop iterates from index 2 to `len(nums)`, calculating `dp[i]` as the maximum of `dp[i-1]` and `dp[i-2] + nums[i-1]`. Finally, it returns the maximum value in `dp`.

思路 1：代码

```
1 class Solution:
2     def rob(self, nums: List[int]) -> int:
3         size = len(nums)
4         if size == 0:
5             return 0
6
7         dp = [0 for _ in range(size + 1)]
8         dp[0] = 0
9         dp[1] = nums[0]
10
11        for i in range(2, size + 1):
12            dp[i] = max(dp[i - 2] + nums[i - 1], dp[i - 1])
13
14        return dp[size]
```

py

思路 1：复杂度分析

- **时间复杂度**: $O(n)$ 。一重循环遍历的时间复杂度为 $O(n)$ 。
- **空间复杂度**: $O(n)$ 。用到了一维数组保存状态，所以总体空间复杂度为 $O(n)$ 。

状态转移的关键点

对于第 i 个房屋，有两种选择：

1. 不抢第 i 个房屋：

- 那么抢劫的金额就是前 $i - 1$ 个房屋的最大金额，即 $dp[i - 1]$ 。

2. 抢第 i 个房屋：

- 如果选择抢劫第 i 个房屋，那么第 $i - 1$ 个房屋就不能抢，因此抢劫的金额为 $dp[i - 2] + nums[i - 1]$ (其中 $nums[i - 1]$ 是第 i 个房屋的金额)。

状态转移方程

根据以上两种情况，取较大值即可：

$$dp[i] = \max(dp[i - 1], dp[i - 2] + nums[i - 1])$$

第 279 题 完全平方数

279. 完全平方数

中等

相关标签

相关企业

Ax

给你一个整数 n ，返回 和为 n 的完全平方数的最少数量。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

示例 1：

输入： $n = 12$

输出：3

解释： $12 = 4 + 4 + 4$

示例 2：

输入： $n = 13$

输出：2

解释： $13 = 4 + 9$

解题思路

1. 状态定义：

- 用 $dp[i]$ 表示将整数 i 表示为若干完全平方数之和所需要的最少数量。

2. 转移方程：

- 如果 i 是完全平方数（例如 1, 4, 9, 16 等），那么 $dp[i] = 1$ ，因为它本身就是一个完全平方数。
- 如果 i 不是完全平方数，枚举所有小于等于 \sqrt{i} 的完全平方数 j^2 ，然后根据转移方程更新 $dp[i]$ ：

$$dp[i] = 1 + \min(dp[i - j^2]) \quad \text{其中 } j^2 \leq i$$

3. 边界条件：

- $dp[0] = 0$ ：表示数字 0 需要 0 个完全平方数。

python

复制代码

```
class Solution:
    def numSquares(self, n: int) -> int:
        dp = [0] * (n + 1) # 初始化 dp 数组, 大小为 n+1, 全为 0
        for i in range(1, n + 1): # 从 1 遍历到 n
            if i ** 0.5 % 1 == 0: # 如果 i 是一个完全平方数
                dp[i] = 1 # 直接赋值为 1
            else:
                # 枚举所有可能的完全平方数 j^2, j 从 1 到 sqrt(i)
                dp[i] = 1 + min([dp[i - j * j] for j in range(1, int(i ** 0.5) + 1)])
        return dp[-1] # 返回 dp[n] 的值
```

思路分析

1. 选择完全平方数:

- 若某个数 j 是完全平方数，那么 $j * j$ 是一个可以使用的选项。
- 对于给定的数字 i ，可能使用的平方数最多到 $\text{int}(i^{0.5})$ 。

2. 子问题的最优解:

- 假设 $dp[x]$ 表示将数字 x 表示为完全平方数的最少数量。
- 如果我们选择用 $j * j$ ，那么剩下的数字就是 $i - j * j$ 。
- 将 $i - j * j$ 表示为完全平方数之和的最少数量为 $dp[i - j * j]$ 。

3. 状态转移方程的构造:

- 如果我们选择平方数 $j * j$ ，则数字 i 的结果是：使用当前的 $j * j$ （这已经是一个平方数，数量为 1）加上剩余部分 $dp[i - j * j]$ 。
- 遍历所有可能的平方数 $j * j$ ，选出 $dp[i - j * j]$ 的最小值，即：

$$dp[i] = 1 + \min(dp[i - j * j] \text{ for } j \text{ in range}(1, \text{int}(\sqrt{i}) + 1))$$

4. 初始状态:

- 当 $i = 0$ 时，不需要任何平方数， $dp[0] = 0$ 。
- 当 $i = 1$ 时，最优解为 1，因为 1 是平方数。
- 当 $i = 4$ 时，最优解为 1，因为 4 ↓ 平方数。

为什么要用 \min ？

因为我们希望找到最少的完全平方数个数，所以需要从所有可能的选择中找到最小的。

直观理解

你可以将 $dp[i]$ 看成是“到达目标数字 i 所需的最少步数”。每一步都尝试减去一个平方数，并递归考虑剩余部分的最优解。通过这种方式，我们能够确保得到最优解。

1. 第一种写法

python

复制代码

```
dp[i] = 1 + min(dp[i - j**2] for j in range(1, int(i**0.5) + 1))
```



- 特点:

- 这里使用的是 **生成器表达式**。
- `[dp[i - j**2]` 的计算会在每次迭代时动态生成，而不是提前生成整个列表。

- 优点:

- 内存效率更高，因为不需要创建整个列表。
- 对于大规模输入时，可以节省一定的内存消耗。

- 缺点:

- 由于每次迭代都动态计算，会比列表生成稍微慢一点（通常可以忽略不计）。

2. 第二种写法

python

复制代码

```
dp[i] = 1 + min([dp[i - j * j] for j in range(1, int(i ** 0.5) + 1)])
```



- 特点:

- 这里使用的是 **列表生成式**。
- `[dp[i - j * j] for j in range(1, int(i ** 0.5) + 1)]` 会先生成整个列表，存储在内存中。

- 优点:

- 对小规模数据来说，可能稍微快一点，因为 Python 的 `min()` 函数直接操作列表效率高。

- 缺点:

- 内存效率较低，特别是当范围很大时（比如 i 非常大），会占用更多内存。

总结对比

特性	第一种写法（生成器表达式）	第二种写法（列表生成式）
内存使用	更高效（动态生成，不占用额外内存）	需要更多内存（生成整个列表）
计算效率	可能稍慢（动态计算每个值）	可能稍快（直接操作列表）
适用场景	数据规模较大时，内存敏感任务	数据规模较小时，计算效率优先的任务

推荐

- 优先使用第一种写法（生成器表达式），尤其是在动态规划中，节省内存可以使代码更鲁棒。

在公式：

python

复制代码

```
dp[i] = 1 + min(dp[i - j * j] for j in range(1, int(i ** 0.5) + 1))
```

前面要加 1 的原因是 `dp[i]` 表示组成数字 `i` 所需的最少平方数个数，而我们在状态转移时，考虑了当前选取的一个平方数 j^2 ，这一步需要额外加上一个平方数。

详细解释

1. 问题描述

我们要求用最少的完全平方数（如 1, 4, 9, 16, ...）组成数字 `i`。

2. 状态转移含义

- $dp[i - j^2]$ ：表示从 $i - j^2$ 这个较小的数字组成所需的最少平方数个数。
- 选取平方数 j^2 后，数字 `i` 中已经包含了这个平方数，所以需要 **加上 1** 来计数。

3. 加 1 的意义

- $dp[i - j^2]$ ：是从较小的问题 $i - j^2$ 的结果。
- **加 1** 表示我们选择了 j^2 这个平方数，作为一个新的组成部分。

494. 目标和

中等

相关标签

相关企业

A₂

给你一个非负整数数组 `nums` 和一个整数 `target`。

向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个 **表达式**：

- 例如，`nums = [2, 1]`，可以在 `2` 之前添加 `'+'`，在 `1` 之前添加 `'-'`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同 **表达式** 的数目。

示例 1：

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

输出: 5

解释: 一共有 5 种方法让最终目标和为 3。

`-1 + 1 + 1 + 1 + 1 = 3`
`+1 - 1 + 1 + 1 + 1 = 3`
`+1 + 1 - 1 + 1 + 1 = 3`
`+1 + 1 + 1 - 1 + 1 = 3`
`+1 + 1 + 1 + 1 - 1 = 3`

思路 3：动态规划

假设数组中所有元素和为 sum , 数组中所有符号为 $+$ 的元素为 sum_x , 符号为 $-$ 的元素和为 sum_y 。则 $target = sum_x - sum_y$ 。

而 $sum_x + sum_y = sum$ 。根据两个式子可以求出 $2 \times sum_x = target + sum$, 即 $sum_x = (target + sum)/2$ 。

那么这道题就变成了, 如何在数组中找到一个集合, 使集合中元素和为 $(target + sum)/2$ 。这就变为了「0-1 背包问题」中求装满背包的方案数问题。



1. 定义状态

定义状态 $dp[i]$ 表示为: 填满容量为 i 的背包, 有 $dp[i]$ 种方法。

2. 状态转移方程

填满容量为 i 的背包的方法数来源于:

1. 不使用当前 num : 只使用之前元素填满容量为 i 的背包的方法数。
2. 使用当前 num : 填满容量 $i - num$ 的包的方法数, 再填入 num 的方法数。

则动态规划的状态转移方程为: $dp[i] = dp[i] + dp[i - num]$ 。

3. 初始化

初始状态下, 默认填满容量为 0 的背包有 1 种办法 (什么也不装)。即 $dp[0] = 1$ 。

4. 最终结果

根据状态定义, 最后输出 $dp[size]$ (即填满容量为 $size$ 的背包, 有 $dp[size]$ 种方法) 即可, 其中 $size$ 为数组 $nums$ 的长度。

思路 3：代码

```
1 class Solution:
2     def findTargetSumWays(self, nums: List[int], target: int) -> int:
3         sum_nums = sum(nums)
4         if abs(target) > abs(sum_nums) or (target + sum_nums) % 2 == 1:
5             return 0
6         size = (target + sum_nums) // 2
7         dp = [0 for _ in range(size + 1)]
8         dp[0] = 1
9         for num in nums:
10             for i in range(size, num - 1, -1):
11                 dp[i] = dp[i] + dp[i - num]
12         return dp[size]
```

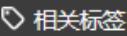
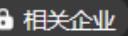


思路 3：复杂度分析

- **时间复杂度**: $O(n)$, 其中 n 为数组 $nums$ 的长度。
- **空间复杂度**: $O(n)$ 。

第 322 题 零钱兑换

322. 零钱兑换

中等   A*

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`

```
Python3 ▼ 智能模式

1 class Solution:
2     def coinChange(self, coins: List[int], amount: int) -> int:
3         dp = [(amount + 1) for _ in range(amount + 1)]
4         dp[0] = 0
5         for coin in coins:
6             for i in range(coin, amount + 1):
7                 dp[i] = min(dp[i], dp[i - coin] + 1)
8         if dp[amount] != amount + 1:
9             return dp[amount]
10        return -1
11
```

1. 初始化 `dp` 数组:

python

复制代码

```
dp = [(amount + 1) for _ in range(amount + 1)]
```

这行代码初始化一个 `dp` 数组，其中 `dp[i]` 表示用最少的硬币组成金额 `i` 所需的硬币数。初始化时，我们将所有的 `dp[i]` 设置为 `amount + 1`，这个值实际上是一个不可能的值，用来表示该金额无法通过硬币组合得到。

2. 初始化 `dp[0] = 0`:

python

复制代码

```
dp[0] = 0
```

由于组成金额 0 所需的硬币数为 0，所以我们将 `dp[0]` 设置为 0。

3. 动态规划的主循环：

```
python
```

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

复制代码

- 外层循环遍历所有的硬币 `coin`。
- 内层循环遍历从 `coin` 到 `amount` 的所有金额 `i`，对于每个金额 `i`，我们尝试通过当前硬币 `coin` 来更新 `dp[i]`。具体地，`dp[i]` 会是当前的值 `dp[i]` 和 `dp[i - coin] + 1` 的较小值，表示要么不使用当前硬币，要么使用当前硬币来更新最少硬币数。

4. 检查结果：

```
python
```

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

复制代码

- 最后检查 `dp[amount]` 是否依然是初始化的值 `amount + 1`，如果是，说明无法用给定的硬币组成目标金额，返回 `-1`。
- 如果 `dp[amount]` 不等于 `amount + 1`，说明找到了最小的硬币数，返回这个值。

为什么还要最后检查 `dp[amount]`

最后检查 `dp[amount]` 的目的是确认是否存在一种硬币组合能够恰好组成目标金额 `amount`。

具体原因：

在动态规划过程中，我们初始化了 `dp` 数组，并将 `dp[i]`（表示组成金额 `i` 所需的最小硬币数）初始化为 `amount + 1`，这个值实际上是一个不可能的值，表示初始状态下我们无法用硬币组成该金额。因此，`dp[i]` 表示一个可行的解时，`dp[i]` 的值一定小于或等于 `amount`。

在动态规划过程中，当我们遍历每个硬币和每个可能的金额时，我们尝试逐步更新 `dp[i]` 的值。如果某个金额 `i` 能够通过某个硬币组合得到，那么 `dp[i]` 会被更新为一个较小的值。

检查 `dp[amount]`：

- 如果 `dp[amount]` 的值仍然是初始化时的 `amount + 1`，说明没有找到一种硬币组合能够组成 `amount`，即目标金额是不可达的，因此返回 `-1`。
- 如果 `dp[amount]` 的值小于或等于 `amount`，说明存在一种硬币组合可以组成 `amount`，并且 `dp[amount]` 存储的是最少的硬币数。

518. 零钱兑换 II

中等

相关标签

相关企业

A₂

给你一个整数数组 `coins` 表示不同面额的硬币，另给一个整数 `amount` 表示总金额。

请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 `0`。

假设每一种面额的硬币有无限个。

题目数据保证结果符合 32 位带符号整数。

示例 1：

输入: `amount = 5, coins = [1, 2, 5]`

输出: 4

解释: 有四种方式可以凑成总金额:

$5=5$

$5=2+2+1$

$5=2+1+1+1$

$5=1+1+1+1+1$

思路 1：动态规划

这道题可以转换为：有 n 种不同的硬币， $\text{coins}[i]$ 表示第 i 种硬币的面额，每种硬币可以无限次使用。请问凑成总金额为 $amount$ 的背包，一共有多少种方案？

这就变成了完全背包问题。「[322. 零钱兑换](#)」中计算的是凑成总金额的最少硬币个数，而这道题计算的是凑成总金额的方案数。

1. 划分阶段

按照当前背包的载重上限进行阶段划分。

2. 定义状态

定义状态 $dp[i]$ 表示为：凑成总金额为 i 的方案总数。

3. 状态转移方程

凑成总金额为 i 的方案数 = 「不使用当前 $coin$, 只使用之前硬币凑成金额 i 的方案数」 + 「使用当前 $coin$ 凑成金额 $i - coin$ 的方案数」。即状态转移方程为： $dp[i] = dp[i] + dp[i - coin]$ 。

4. 初始条件

- 凑成总金额为 0 的方案数为 1，即 $dp[0] = 1$ 。

5. 最终结果

根据我们之前定义的状态， $dp[i]$ 表示为：凑成总金额为 i 的方案总数。所以最终结果为 $dp[amount]$ 。

思路 1：代码

```
1 class Solution:
2     def change(self, amount: int, coins: List[int]) -> int:
3
4         dp = [0 for _ in range(amount + 1)]
5         dp[0] = 1
6         for coin in coins:
7             for i in range(coin, amount + 1):
8                 dp[i] += dp[i - coin]
9
10    return dp[amount]
```



思路 1：复杂度分析

- **时间复杂度：** $O(n \times amount)$ ，其中 n 为数组 coins 的元素个数， $amount$ 为总金额。
- **空间复杂度：** $O(amount)$ 。

3. 状态转移方程 $dp[i] += dp[i - coin]$:

这个公式的含义是：如果我们知道如何组成金额 $i - coin$ 的所有不同组合数，那么通过加入一个硬币 $coin$ ，我们就能够得到组成金额 i 的组合数。因此，我们通过更新 $dp[i]$ 来累积所有使用硬币 $coin$ 后形成的金额 i 的组合数。

具体来说：

- $dp[i - coin]$ 表示组成金额 $i - coin$ 的不同组合数。
- 如果我们使用硬币 $coin$ 来组成金额 i ，那么我们将所有组成金额 $i - coin$ 的组合加上这个硬币 $coin$ ，从而得出组成金额 i 的新组合。

4. 直观理解：

假设我们有一组硬币 $[1, 2, 5]$ 和目标金额 5 。我们希望知道有多少种不同的组合可以组成 5 元。

举个简单的例子：

如果我们已经知道如何组成金额 4 ，那么通过在组合中加一个面值为 1 的硬币，我们就能得到组成金额 5 的所有组合。这就是 $dp[5] += dp[4]$ 的含义。

- 如果 $dp[4] = 2$ （表示组成 4 元有 2 种组合），那么通过加入一个硬币 1 ，就会得到 2 种新的组成 5 元的方式。
- 因此， $dp[5]$ 应该加上 $dp[4]$ ，也就是 $\downarrow dp[5] += dp[4]$ 。

十六、多维动态规划

第 62 题 不同路径

62. 不同路径

已解答 

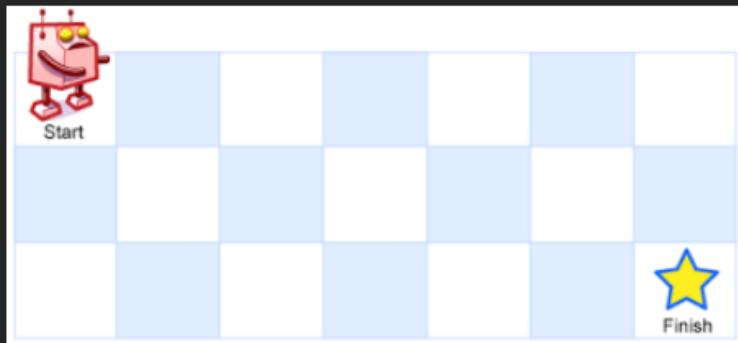
 中等  相关标签  相关企业 

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径？

示例 1：



输入: $m = 3, n = 7$

输出: 28

示例 2：

输入: $m = 3, n = 2$

输出: 3

解释:

从左上角开始，总共有 3 条路径可以到达右下角。

1. 向右 \rightarrow 向下 \rightarrow 向下

4.3.3 解题思路

1. 划分阶段

按照路径的结尾位置（行位置、列位置组成的二维坐标）进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：从左上角到达 (i, j) 位置的路径数量。

3. 状态转移方程

因为我们每次只能向右、或者向下移动一步，因此想要走到 (i, j) ，只能从 $(i - 1, j)$ 向下走一步走过来；或者从 $(i, j - 1)$ 向右走一步走过来。所以可以写出状态转移方程为： $dp[i][j] = dp[i - 1][j] + dp[i][j - 1]$ ，此时 $i > 0, j > 0$ 。

4. 初始条件

- 从左上角走到 $(0, 0)$ 只有一种方法，即 $dp[0][0] = 1$ 。
- 第一行元素只有一条路径（即只能通过前一个元素向右走得到），所以 $dp[0][j] = 1$ 。
- 同理，第一列元素只有一条路径（即只能通过前一个元素向下走得到），所以 $dp[i][0] = 1$ 。

5. 最终结果

根据状态定义，最终结果为 $dp[m - 1][n - 1]$ ，即从左上角到达右下角 $(m - 1, n - 1)$ 位置的路径数量为 $dp[m - 1][n - 1]$ 。

```
Python3 ▾  智能模式

1 class Solution:
2     def uniquePaths(self, m: int, n: int) -> int:
3         dp=[[0 for _ in range(n)] for _ in range(m)]
4
5         for i in range(m):
6             dp[i][0]=1
7         for j in range(n):
8             dp[0][j]=1
9
10        for i in range(1,m):
11            for j in range(1,n):
12                dp[i][j]=dp[i-1][j]+dp[i][j-1]
13
14        return dp[m-1][n-1]
```

1. 外层列表推导:

python

复制代码

```
[ [...] for _ in range(m)]
```

- 这部分表示创建一个长度为 m 的列表。
- 每个元素是通过内层列表推导生成的子列表。
- `for _ in range(m)` 的意思是重复执行 m 次。变量 $_$ 通常用作占位符，表示不关心具体的值。

2. 内层列表推导:

python

复制代码

```
[0 for _ in range(n)]
```

- 这部分表示创建一个长度为 n 的列表。
- 每个元素都是 0 。
- `for _ in range(n)` 的意思是重复生成 n 个 0 。

3. 最终结果:

- 外层列表将内层列表重复 m 次，从而形成一个 $m \times n$ 的二维列表。

第 64 题 最小路径和

64. 最小路径和

中等

相关标签

相关企业

A+

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

思路 1：动态规划

1. 划分阶段

按照路径的结尾位置（行位置、列位置组成的二维坐标）进行阶段划分。

2. 定义状态

定义状态 $dp[i][j]$ 为：从左上角到达 (i, j) 位置的最小路径和。

3. 状态转移方程

当前位置 (i, j) 只能从左侧位置 $(i, j - 1)$ 或者上方位置 $(i - 1, j)$ 到达。为了使得从左上角到达 (i, j) 位置的最小路径和最小，应从 $(i, j - 1)$ 位置和 $(i - 1, j)$ 位置选择路径和最小的位置达到 (i, j) 。

即状态转移方程为： $dp[i][j] = \min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j]$ 。

4. 初始条件

- 当左侧和上方是矩阵边界时（即 $i = 0, j = 0$ ）， $dp[i][j] = grid[i][j]$ 。
- 当只有左侧是矩阵边界时（即 $i \neq 0, j = 0$ ），只能从上方到达， $dp[i][j] = dp[i - 1][j] + grid[i][j]$ 。
- 当只有上方是矩阵边界时（即 $i = 0, j \neq 0$ ），只能从左侧到达， $dp[i][j] = dp[i][j - 1] + grid[i][j]$ 。

5. 最终结果

根据状态定义，最后输出 $dp[rows - 1][cols - 1]$ （即从左上角到达 $(rows - 1, cols - 1)$ 位置的最小路径和）即可。其中 $rows$ 、 $cols$ 分别为 $grid$ 的行数、列数。

思路 1：代码

```
1 class Solution:
2     def minPathSum(self, grid: List[List[int]]) -> int:
3         rows, cols = len(grid), len(grid[0])
4         dp = [[0 for _ in range(cols)] for _ in range(rows)]
5
6         dp[0][0] = grid[0][0]
7
8         for i in range(1, rows):
9             dp[i][0] = dp[i - 1][0] + grid[i][0]
10
11        for j in range(1, cols):
12            dp[0][j] = dp[0][j - 1] + grid[0][j]
13
14        for i in range(1, rows):
15            for j in range(1, cols):
16                dp[i][j] = min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j]
17
18        return dp[rows - 1][cols - 1]
```

思路 1：复杂度分析

- 时间复杂度：** $O(m * n)$ ，其中 m 、 n 分别为 $grid$ 的行数和列数。
- 空间复杂度：** $O(m * n)$ 。

十七、技巧

第 136 题 只出现一次的数字

136. 只出现一次的数字 已解答

[简单](#) [相关标签](#) [相关企业](#) [提示](#) [A_x](#)

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

Python3 智能模式

```
1 class Solution:  
2     def singleNumber(self, nums: List[int]) -> int:  
3         return reduce(xor,nums)
```

reduce() 函数

`reduce()` 是 Python `functools` 模块中的一个函数，用于将某个二元操作（函数）依次应用到序列的元素上，最终归约为一个结果。

语法

```
python 复制代码
from functools import reduce

reduce(function, iterable[, initializer])
```

- `function`: 二元函数（接受两个输入的函数）。
- `iterable`: 一个可迭代对象（如列表、元组等）。
- `initializer` (可选): 初始值。如果提供，`reduce` 从初始值开始计算。

第 169 题 多数元素

169. 多数元素

已解答

[简单](#) [相关标签](#) [相关企业](#) [A+](#)

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

思路 1：哈希表

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

思路 1：代码

```
1 class Solution:
2     def majorityElement(self, nums: List[int]) -> int:
3         numDict = dict()
4         for num in nums:
5             if num in numDict:
6                 numDict[num] += 1
7             else:
8                 numDict[num] = 1
9         max = float('-inf')
10        max_index = -1
11        for num in numDict:
12            if numDict[num] > max:
13                max = numDict[num]
14                max_index = num
15        return max_index
```

```
1 class Solution:
2     def majorityElement(self, nums: List[int]) -> int:
3         nums_dict={}
4         for i in nums:
5             if i in nums_dict:
6                 nums_dict[i]+=1
7             else:
8                 nums_dict[i]=1
9         max=0 #次数
10        max_index=0 #值
11        for j in nums_dict:
12            if nums_dict[j]>max:
13                max=nums_dict[j]
14                max_index=j
15        return max_index
16
```

```
Python3 ▼ 智能模式
1 class Solution:
2     def majorityElement(self, nums: List[int]) -> int:
3         nums_dict={}
4
5         for num in nums:
6             if num in nums_dict:
7                 nums_dict[num]+=1
8             else:
9                 nums_dict[num]=1
10
11         sort_nums_dict=sorted(nums_dict.items(), key=lambda item:item[1])
12         return sort_nums_dict[-1][0]
13
```

原因分析

1. `sorted_nums_dict` 是什么类型?

- `sorted(nums_dict.items(), key=lambda item: item[1], reverse=True)`
返回的是一个列表，其中每个元素是一个元组，这个元组包含字典的键和值。
- 例如：

```
python
nums_dict = {3: 2, 2: 1}
sorted_nums_dict = sorted(nums_dict.items(), key=lambda item: item[1], reverse=True)
print(sorted_nums_dict)
# 输出: [(3, 2), (2, 1)]
```

这里的 `sorted_nums_dict[0]` 是元组 `(3, 2)`。

2. `sorted_nums_dict[0]` 是什么?

- `sorted_nums_dict[0]` 取出的是排序后的第一个元组 `(3, 2)`。
- 这个元组的结构是 `(key, value)`，即 `(3, 2)`，其中 `3` 是键，`2` 是值。

3. 为什么是 `[0][0]` ?

- `sorted_nums_dict[0]` 是一个元组 `(3, 2)`。
- 要取出元组的第一个元素（即键 `3`），需要再加一个索引 `[0]`。
- 即：`sorted_nums_dict[0][0]` → 第一个元组的第一个元素（键）。

4. 如果只写 `sorted_nums_dict[0]` 会得到什么？

- 只写 `sorted_nums_dict[0]`，得到的是元组 `(3, 2)`，而我们只需要其中的键 `3`，所以需要进一步通过 `[0]` 索引。

十八、CodeTop 补充

第 88 题 合并两个有序数组

88. 合并两个有序数组

已解答

[简单](#) [相关标签](#) [相关企业](#) [提示](#) [A_x](#)

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。

python

复制代码

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        p1, p2, p = m - 1, n - 1, m + n - 1 # 初始化三个指针
        while p2 >= 0: # nums2 还有要合并的元素
            # 如果 p1 < 0, 那么说明 nums1 中没有剩余的有效元素了, 直接将 nums2 中的元素填入 nums1
            if p1 >= 0 and nums1[p1] > nums2[p2]:
                nums1[p] = nums1[p1] # 将 nums1[p1] 填入 nums1[p]
                p1 -= 1 # 移动 p1 指针
            else:
                nums1[p] = nums2[p2] # 将 nums2[p2] 填入 nums1[p]
                p2 -= 1 # 移动 p2 指针
            p -= 1 # 移动填充位置指针 p
```

逐行解释：

1. 初始化指针：

- `p1 = m - 1`: `p1` 指向 `nums1` 的有效元素的最后一个位置 (即 `nums1` 中第 `m` 个元素的索引)。
- `p2 = n - 1`: `p2` 指向 `nums2` 的最后一个元素 (即 `nums2[n-1]`)。
- `p = m + n - 1`: `p` 指向 `nums1` 的最后一个位置 (`nums1` 的总长度是 `m + n`, 此时需要填充的位置在数组末尾)。

2. while 循环：

- `while p2 >= 0`: 说明 `nums2` 中还有元素需要合并。当 `p2` 小于 0 时, `nums2` 中的所有元素已经被处理完。

3. 判断条件：

- `if p1 >= 0 and nums1[p1] > nums2[p2]`: 如果 `nums1[p1]` 大于 `nums2[p2]`, 说明 `nums1[p1]` 应该放在合并后的数组中 `nums2[p2]` 之前。此时将 `nums1[p1]` 填入 `nums1[p]` 位置, 并将 `p1` 和 `p` 各自减 1。
- `else`: 如果 `nums1[p1]` 小于等于 `nums2[p2]` 或者 `p1` 已经越界 (即 `p1 < 0`), 则将 `nums2[p2]` 填入 `nums1[p]` 位置, 并将 `p2` 和 `p` 各自减 1。

4. 填充并更新指针：

- 每次合并一个元素后, `p` 指向下一个待填充的位置, 需要更新 `p`。
↓
`p1` 和 `p2` 也需要根据条件各自向前移动。

第 232 题 用栈实现队列

232. 用栈实现队列

简单 相关标签 相关企业 A_文

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：

实现 MyQueue 类：

- void push(int x) 将元素 x 推到队列的末尾
- int pop() 从队列的开头移除并返回元素
- int peek() 返回队列开头的元素
- boolean empty() 如果队列为空，返回 true；否则，返回 false

说明：

- 你只能 使用标准的栈操作 —— 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque (双端队列) 来模拟一个栈，只要是标准的栈操作即可。

自我总结：主要是通过两个 stack, 首先是判断 outstack 是否为空，为空的话就没法模拟出队列的操作（没有元素），然后就是将 instack 的末尾元素逐个加入到 outstack 中，然后 top 就是 outstack[-1]

思路 1：双栈

使用两个栈，inStack 用于输入，outStack 用于输出。

- push 操作：将元素压入 inStack 中。
- pop 操作：如果 outStack 输出栈为空，将 inStack 输入栈元素依次取出，按顺序压入 outStack 栈。这样 outStack 栈的元素顺序和之前 inStack 元素顺序相反，outStack 顶层元素就是要取出的队头元素，将其移出，并返回该元素。如果 outStack 输出栈不为空，则直接取出顶层元素。
- peek 操作：和 pop 操作类似，只不过最后一步不需要取出顶层元素，直接将其返回即可。
- empty 操作：如果 inStack 和 outStack 都为空，则队列为空，否则队列不为空。

思路 1：代码

```
1  class MyQueue:
2
3      def __init__(self):
4          self.inStack = []
5          self.outStack = []
6          """
7              Initialize your data structure here.
8          """
9
10
11     def push(self, x: int) -> None:
12         self.inStack.append(x)
13         """
14             Push element x to the back of queue.
15         """
16
17
18     def pop(self) -> int:
19         if(len(self.outStack) == 0):
20             while(len(self.inStack) != 0):
21                 self.outStack.append(self.inStack[-1])
22                 self.inStack.pop()
23             top = self.outStack[-1]
24             self.outStack.pop()
25             return top
26         """
27             Removes the element from in front of queue and returns that element.
28         """
29
30
31     def peek(self) -> int:
32         if (len(self.outStack) == 0):
33             while (len(self.inStack) != 0):
34                 self.outStack.append(self.inStack[-1])
35                 self.inStack.pop()
36             top = self.outStack[-1]
37             return top
38         """
39             Get the front element.
40         """
41
42
43     def empty(self) -> bool:
44         return len(self.outStack) == 0 and len(self.inStack) == 0
45         """
46             Returns whether the queue is empty.
47         """
```

思路 1：复杂度分析

- **时间复杂度：**`push` 和 `empty` 为 $O(1)$, `pop` 和 `peek` 为均摊 $O(1)$ 。
- **空间复杂度：** $O(n)$ 。

代码解析：

`__init__` 方法:

python

复制代码

```
def __init__(self):
    self.inStack = []
    self.outStack = []
```

- 初始化两个空栈，`inStack` 用来存放入队的元素，`outStack` 用来存放出队的元素。

`push(x)` 方法:

python

复制代码

```
def push(self, x: int) -> None:
    self.inStack.append(x)
```

- 将元素 `x` 添加到 `inStack` 中，模拟队列的入队操作。

`pop()` 方法:

python

复制代码

```
def pop(self) -> int:
    if len(self.outStack) == 0:
        while len(self.inStack) != 0:
            self.outStack.append(self.inStack[-1])
            self.inStack.pop()
    top = self.outStack[-1]
    self.outStack.pop()
    return top
```



- 当我们要从队列中移除一个元素时，首先检查 `outStack` 是否为空。
 - 如果 `outStack` 为空，将 `inStack` 中的所有元素逐个弹出并压入 `outStack` 中（这将使得 `outStack` 中的顺序反转，模拟队列的 FIFO 顺序）。
 - 然后，从 `outStack` 中弹出栈顶元素，这就是队列的队头元素。

peek() 方法:

python

复制代码

```
def peek(self) -> int:  
    if (len(self.outStack) == 0):  
        while (len(self.inStack) != 0):  
            self.outStack.append(self.inStack[-1])  
            self.inStack.pop()  
    top = self.outStack[-1]  
    return top
```



- 和 `pop()` 方法类似，只是我们不弹出元素，只是返回 `outStack` 的栈顶元素，模拟查看队头元素。
- 如果 `outStack` 为空，仍然需要将 `inStack` 中的所有元素转移到 `outStack` 中。

empty() 方法:

python

复制代码

```
def empty(self) -> bool:  
    return len(self.outStack) == 0 and len(self.inStack) == 0
```

- 如果 `inStack` 和 `outStack` 都为空，则队列为空，返回 `True`；否则返回 `False`。

详细解释:

1. `self.outStack.append(self.inStack[-1]):`

- `self.inStack[-1]` 取出 `inStack` 栈的 **栈顶元素**，即 `inStack` 中的最后一个元素（栈是后进先出的数据结构，所以最后进入的元素是栈顶元素）。
- `self.outStack.append(self.inStack[-1])` 将这个栈顶元素压入 `outStack` 中。这样，`outStack` 中就会按照队列的顺序存放元素（最早进入队列的元素在 `outStack` 的栈底，最晚进入队列的元素在栈顶）。

2. `self.inStack.pop():`

- `self.inStack.pop()` 移除并返回 `inStack` 的栈顶元素，也就是将刚才添加到 `outStack` 中的那个元素从 `inStack` 中弹出。
- 这一步的作用是，从 `inStack` 中移除元素，确保元素不再留在 `inStack` 中，从而模拟出队列操作。

..

转移过程示意：

假设初始时：

- `inStack = [1, 2, 3]` (栈顶是 3，栈底是 1)
- `outStack = []` (空)

转移的过程是：

1. `self.outStack.append(self.inStack[-1])` 将 3 从 `inStack` 压入 `outStack`，`inStack` 变成 `[1, 2]`，`outStack` 变成 `[3]`。
2. `self.inStack.pop()` 从 `inStack` 弹出 3，`inStack` 变成 `[1, 2]`，`outStack` 变成 `[3]`。
3. 继续执行相同的操作，直到 `inStack` 中所有元素都转移到 `outStack` 中：
 - `self.outStack.append(self.inStack[-1])` 将 2 压入 `outStack`，`inStack` 变成 `[1]`，`outStack` 变成 `[3, 2]`。
 - `self.inStack.pop()` 弹出 2，`inStack` 变成 `[1]`，`outStack` 变成 `[3, 2]`。
 - `self.outStack.append(self.inStack[-1])` 将 1 压入 `outStack`，`inStack` 变成 `[]`，`outStack` 变成 `[3, 2, 1]`。
 - `self.inStack.pop()` 弹出 1，`inStack` 变成 `[]`，`outStack` 变成 `[3, 2, 1]`。

现在，`outStack` 中的元素按照队列的顺序排列 $\downarrow [3, 2, 1]$ ，最早进入队列的元素 1 在最上面，最晚进入的元素 3 在最下面。

1. `top = self.outStack[-1]`：

- 这一行代码取出 `outStack` 的栈顶元素，即队列的 **前端元素**。栈是后进先出 (LIFO, Last-In-First-Out) 的数据结构，所以 `outStack[-1]` 取的是栈顶的元素。
- **为什么是队列的前端元素？** 因为我们在之前的步骤中，将 `inStack` 中的元素转移到 `outStack`。转移的过程中，最先进入队列的元素被最后转移到 `outStack`，而转移后的 `outStack` 就按照队列的顺序排列。

假设转移后的 `outStack` 是 `[3, 2, 1]`，其中：

- 3 是最后进入队列的元素（最晚的），它会在 `outStack` 的栈底。
- 1 是最早进入队列的元素，它会在 `outStack` 的栈顶。

所以，`outStack[-1]` 取到的是 1，即队列的前端元素。

2. `self.outStack.pop()`：

- `pop()` 操作会移除并返回栈顶元素，因此这行代码将 **移除 `outStack` 中的栈顶元素**，即队列的 **前端元素**。
- **为什么移除的是队列的前端元素？** 因为在转移操作中，我们将 `inStack` 中的元素按队列顺序（先进先出）转移到 `outStack`。因此，`outStack` 的栈顶元素就是队列的前端元素（最早入队的元素），执行 `pop()` 后就移除并返回这个元素。

区别：

- `pop()`：
 - 在栈中：移除并返回栈顶元素（LIFO 顺序）。
 - 在队列中：移除并返回队列的前端元素（FIFO 顺序）。
- `peek()`：
 - 在栈中：返回栈顶元素，但不移除它。
 - 在队列中：返回队列的前端元素，但不移除它。

pop()函数跟peek()函数主要是什么作用

`pop()` 函数和 `peek()` 函数是栈（stack）和队列（queue）常用的操作，它们有不同的作用，下面分别解释它们在栈和队列中的作用：

1. `pop()` 函数：

在栈（Stack）中的作用：

- `pop()` 操作用于 **移除并返回** 栈顶元素（最后进入栈的元素）。栈是后进先出（LIFO, Last-In-First-Out）结构，所以 `pop()` 总是操作栈顶元素。
- **功能：**移除并返回栈顶元素。

示例：

```
python  
  
stack = [1, 2, 3]  
top_element = stack.pop() # pop() 会移除栈顶元素 3，并返回 3  
print(top_element) # 输出 3  
print(stack) # 输出 [1, 2], 栈顶元素被移除
```

复制代码

在队列 (Queue) 中的作用：

- 在队列中，`pop()` 操作通常与队列的 `front` (前端元素) 相关，它移除并返回 **最前面** 的元素 (最早进入队列的元素)。
- **功能：**移除并返回队列的前端元素。

示例（使用两个栈模拟队列）：

python

复制代码

```
queue = MyQueue() # 假设已经有一个 MyQueue 实现
queue.push(1)
queue.push(2)
front_element = queue.pop() # pop() 会移除最前面的元素 1，并返回 1
print(front_element) # 输出 1
```

2. `peek()` 函数：

在栈 (Stack) 中的作用：

- `peek()` 操作用于 **查看栈顶元素**，但不移除它。这个操作通常用于我们想要知道栈顶的内容，但不想改变栈的状态时。
- **功能：**返回栈顶元素，但不移除它。

示例：

python

复制代码

```
stack = [1, 2, 3]
top_element = stack[-1] # peek() 会查看栈顶元素 3，但不会移除它
print(top_element) # 输出 3
print(stack) # 输出 [1, 2, 3]，栈的内容没有改变
```

在队列 (Queue) 中的作用：

- `peek()` 操作用于 **查看队列的前端元素**，但不移除它。这个操作通常用于我们想要知道队列的前端元素，但不想改变队列状态时。
- **功能：**返回队列的前端元素，但不移除它。

示例（使用两个栈模拟队列）：

python

复制代码

```
queue = MyQueue() # 假设已经有一个 MyQueue 实现
queue.push(1)
queue.push(2)
front_element = queue.peek() # peek() 会查看队列前端元素 1，但不会移除它
print(front_element) # 输出 1
```

第 225 题 用队列实现栈

225. 用队列实现栈

简单 相关标签 相关企业 A₂

请你仅使用两个队列实现一个后入先出 (LIFO) 的栈，并支持普通栈的全部四种操作 (`push`、`top`、`pop` 和 `empty`)。

实现 `MyStack` 类：

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

自我总结：通过两个 `dequeue()` 来模拟，主要是判断 `popQueue` 里是否有元素，通过交换 `pushQueue` 跟 `popQueue` 来将元素反向，并且始终保持操作后的 `pushQueue` 为空，这样新加入元素时，才能在第一个

解题思路

思路 1：双队列

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

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



思路 1：代码

```
1  class MyStack:
2
3      def __init__(self):
4          """
5              Initialize your data structure here.
6          """
7          self.pushQueue = collections.deque()
8          self.popQueue = collections.deque()
9
10
11     def push(self, x: int) -> None:
12         """
13             Push element x onto stack.
14         """
15         self.pushQueue.append(x)
16         while self.popQueue:
17             self.pushQueue.append(self.popQueue.popleft())
18         self.pushQueue, self.popQueue = self.popQueue, self.pushQueue
19
20     def pop(self) -> int:
21         """
22             Removes the element on top of the stack and returns that element.
23         """
24         return self.popQueue.popleft()
25
26
27     def top(self) -> int:
28         """
29             Get the top element.
30         """
31         return self.popQueue[0]
32
33
34     def empty(self) -> bool:
35         """
36             Returns whether the stack is empty.
37         """
38         return not self.popQueue
39
40
```



思路 1：复杂度分析

- **时间复杂度：**入栈操作的时间复杂度为 $O(n)$ 。出栈、取栈顶元素、判断栈是否为空的时间复杂度为 $O(1)$ 。
- **空间复杂度：** $O(n)$ 。

工作原理总结：

- `push(x)`：将元素 `x` 添加到 `pushQueue`，然后将 `popQueue` 中的元素移动到 `pushQueue`，并交换两者的位置。这个过程确保了栈的顺序。
- `pop()`：直接从 `popQueue` 中移除并返回栈顶元素。每次 `push` 后，`popQueue` 保证了栈顶元素的顺序。
- `top()`：返回 `popQueue` 中的栈顶元素。
- `empty()`：检查 `popQueue` 是否为空，如果为空则栈为空。

示例：

假设我们依次执行以下操作：

1. `push(1)`

- `pushQueue = [1]`, `popQueue = []` (交换后, `popQueue = [1]`, `pushQueue = []`)

2. `push(2)`

- `pushQueue = [2]`, `popQueue = [1]` (交换后, `popQueue = [2, 1]`, `pushQueue = []`)

3. `push(3)`

- `pushQueue = [3]`, `popQueue = [2, 1]` (交换后, `popQueue = [3, 2, 1]`, `pushQueue = []`)

4. `pop()`

- 返回 `popQueue.popleft()`，即 `3`。
- `popQueue = [2, 1]`, `pushQueue = []`

5. `top()`

- 返回 `popQueue[0]`，即 `2`。 

912. 排序数组

尝试过 

中等

相关标签

相关企业

A₂

给你一个整数数组 `nums`，请你将该数组升序排列。

你必须在 **不使用任何内置函数** 的情况下解决问题，时间复杂度为 $O(n \log(n))$ ，并且空间复杂度尽可能小。

示例 1：

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

示例 2：

输入: `nums = [5, 1, 1, 2, 0, 0]`
输出: `[0, 0, 1, 1, 2, 5]`

解题思路

这道题是一道用来复习排序算法，测试算法时间复杂度的好题。我试过了十种排序算法。得到了如下结论：

- 超时算法（时间复杂度为 $O(n^2)$ ）：冒泡排序、选择排序、插入排序。
- 通过算法（时间复杂度为 $O(n \times \log n)$ ）：希尔排序、归并排序、快速排序、堆排序。
- 通过算法（时间复杂度为 $O(n)$ ）：计数排序、桶排序。
- 解答错误算法（普通基数排序只适合非负数）：基数排序。

思路 5：归并排序（通过）

归并排序（Merge Sort）基本思想：采用经典的分治策略，先递归地将当前数组平均分成两半，然后将有序数组两两合并，最终合并成一个有序数组。

假设数组的元素个数为 n 个，则归并排序的算法步骤如下：

1. **分解过程：**先递归地将当前数组平均分成两半，直到子数组长度为 1。
 1. 找到数组中心位置 mid ，从中心位置将数组分成左右两个子数组 $left_nums$ 、 $right_nums$ 。
 2. 对左右两个子数组 $left_nums$ 、 $right_nums$ 分别进行递归分解。
 3. 最终将数组分解为 n 个长度均为 1 的有序子数组。
2. **归并过程：**从长度为 1 的有序子数组开始，依次将有序数组两两合并，直到合并成一个长度为 n 的有序数组。
 1. 使用数组变量 $nums$ 存放合并后的有序数组。
 2. 使用两个指针 $left_i$ 、 $right_i$ 分别指向两个有序子数组 $left_nums$ 、 $right_nums$ 的开始位置。
 3. 比较两个指针指向的元素，将两个有序子数组中较小元素依次存入到结果数组 $nums$ 中，并将指针移动到下一位置。
 4. 重复步骤 3，直到某一指针到达子数组末尾。
 5. 将另一个子数组中的剩余元素存入到结果数组 $nums$ 中。
 6. 返回合并后的有序数组 $nums$ 。

思路 5：代码

```
1 class Solution:
2     # 合并过程
3     def merge(self, left_nums: [int], right_nums: [int]):
4         nums = []
5         left_i, right_i = 0, 0
6         while left_i < len(left_nums) and right_i < len(right_nums):
7             # 将两个有序子数组中较小元素依次插入到结果数组中
8             if left_nums[left_i] < right_nums[right_i]:
9                 nums.append(left_nums[left_i])
10                left_i += 1
11            else:
12                nums.append(right_nums[right_i])
13                right_i += 1
14
15        # 如果左子数组有剩余元素，则将其插入到结果数组中
16        while left_i < len(left_nums):
17            nums.append(left_nums[left_i]) ← 理论可以，但本题不能改写成
18            left_i += 1
19
20        # 如果右子数组有剩余元素，则将其插入到结果数组中
21        while right_i < len(right_nums):
22            nums.append(right_nums[right_i]) ← 因为频繁的切片操作会消耗更多的时间
23            right_i += 1
24
25        # 返回合并后的结果数组
26        return nums
27
```

```

27     # 分解过程
28     def mergeSort(self, nums: [int]) -> [int]:
29         # 数组元素个数小于等于 1 时，直接返回原数组
30         if len(nums) <= 1:
31             return nums
32
33
34         mid = len(nums) // 2
35         left_nums = self.mergeSort(nums[0: mid])
36         right_nums = self.mergeSort(nums[mid:])
37         return self.merge(left_nums, right_nums)
38
39     def sortArray(self, nums: [int]) -> [int]:
40         return self.mergeSort(nums)

```

逻辑解析：

1. 输入参数：

- `left_nums`：左边的有序子数组。
- `right_nums`：右边的有序子数组。

2. 合并过程：

- 使用两个指针 `left_i` 和 `right_i` 来分别指向两个子数组的当前位置。
- 比较两个子数组当前指针位置的元素，将较小的元素添加到 `nums` 数组中，并移动对应的指针。
- 当其中一个子数组元素全部加入到 `nums` 后，将剩余的另一个子数组的元素直接加入到 `nums`。

3. 返回值：

- 返回合并后的有序数组 `nums`。

逻辑解析：

1. 递归分解：

- 如果数组的长度小于等于 1，则返回原数组，表示已经有序。
- 否则，将数组从中间 `mid` 分为左右两部分，递归地对左半部分 `left_nums` 和右半部分 `right_nums` 进行排序。

2. 合并：

- 使用 `merge` 方法将排序后的 `left_nums` 和 `right_nums` 合并为一个有序数组，并返回该数组。

使用快速排序的方法：

快速排序使用**分治法** (Divide and conquer) 策略来把一个序列 (list) 分为较小和较大的 2 个子序列，然后递归地排序两个子序列。

其基本步骤为 [摘自@维基百科]：

1. **挑选基准值**：从数列中挑出一个元素，称为“基准” (pivot)；
2. **分割** (partition)：重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（与基准值相等的数可以到任何一边）。在这个分割结束之后，对基准值的排序就已经完成；
3. **递归排序子序列**：递归地将小于基准值元素的子序列和大于基准值元素的子序列排序。

递归到底部的判断条件是数列的大小是零或一，此时该数列显然已经有序。

选取基准值pivot也有多种方式，且选取pivot的方法对排序的时间性能有着决定性的影响。例如，对于一个逆序数组，如果每次选取数组中的第一个元素为pivot，那么将其正序排列的过程将会变得非常慢，时间复杂度为 $O(n^2)$ 。因此，在具体实现中考虑随机化选择基准值 pivot也是非常有必要的。

```

1  class Solution:
2      def sortArray(self, nums: List[int]) -> List[int]:
3          self.quick_sort(nums, 0, len(nums) - 1) # 调用快排函数对nums进行排序
4          return nums
5
6      def quick_sort(self, arr, low, high):
7          if low >= high: # 递归结束
8              return
9
10         pivot_idx = random.randint(low, high) # 随机选择pivot
11         arr[low], arr[pivot_idx] = arr[pivot_idx], arr[low] # pivot放置到最左边
12         pivot = arr[low] # 选取最左边为pivot
13
14         left, right = low, high # 双指针
15         while left < right:
16
17             while (
18                 left < right and arr[right] > pivot
19             ): # 找到右边第一个<=pivot的元素【需考虑重复元素问题】
20                 right -= 1
21             if left < right:
22                 arr[left], arr[right] = arr[right], arr[left] # 并将两者交换
23                 left += 1 # left指向下一个待排序的元素
24
25             while (
26                 left < right and arr[left] < pivot
27             ): # 找到左边第一个>=pivot的元素【需考虑重复元素问题】
28                 left += 1
29             if left < right:
30                 arr[right], arr[left] = arr[left], arr[right] # 并将两者交换
31                 right -= 1 # right指向下一个待排序的元素
32
33         arr[left] = pivot # pivot放置到中间left=right处
34         mid = left # 以mid=left=right为分割点
35         self.quick_sort(arr, low, mid - 1) # 递归对mid两侧元素进行排序
36         self.quick_sort(arr, mid + 1, high)

```

步骤 1：随机选择 pivot_idx

使用 `random.randint(low, high)` 在 `[0, 4]` 范围内随机生成一个索引。例如，假设随机生成的 `pivot_idx = 2`。

步骤 2：交换 arr[low] 和 arr[pivot_idx]

交换后数组变为：

- 原来的 `arr[low]` (即 `arr[0] = 4`) 和 `arr[pivot_idx]` (即 `arr[2] = 7`) 互换。

新的数组：

python

复制代码

```
arr = [7, 2, 4, 1, 3]
```

步骤 3：设置 pivot

选择 `arr[low]` (即 `arr[0] = 7`) 作为当前分区的基准值 `pivot`。

结果：

python

复制代码

```
pivot = 7
```

```
while (  
    left < right and arr[right] > pivot  
):  
    right -= 1
```

这里的`arr[right] > pivot`不是看的是右边的大于pivot基准值吗，怎么是要找小于它的，找到右边第一个`<= pivot`的元素：`arr[4] = 1`

你的观察是对的，`arr[right] > pivot` 的确是在寻找 **右侧大于基准值的元素**。但重点是 **循环会停止在第一个不满足 `arr[right] > pivot` 的位置**，也就是第一个 `arr[right] <= pivot` 的元素。

流程解释

在分区过程中，`while left < right and arr[right] > pivot` 是这样工作的：

1. 如果当前右指针元素（`arr[right]`）**大于基准值 pivot**，那么右指针 `right` 向左移动（`right -= 1`），继续检查前面的元素。
2. 如果遇到一个**小于等于基准值 pivot**的元素，循环停止。这时，`arr[right]` 是分区中**需要被移动到左侧的元素**。

示例说明

以数组 `arr = [6, 3, 5, 8, 2]`，`pivot = 6` 为例，说明双指针的移动和操作：

初始状态：

plaintext

复制代码

```
arr = [6, 3, 5, 8, 2]
```

```
pivot = 6, left = 0, right = 4
```

第一次循环（移动右指针）：

- `arr[right] = 2`, `2 <= 6`, 停止。
- `arr[left] = 6` 和 `arr[right] = 2` 交换：

plaintext

复制代码

```
arr = [2, 3, 5, 8, 6]
```

```
left = 1, right = 4
```

第二次循环（移动左指针）：

- `arr[left] = 3`, $3 < 6$, 继续。
- `arr[left] = 5`, $5 < 6$, 继续。
- `arr[left] = 8`, $8 \geq 6$, 停止。
- `arr[left] = 8` 和 `arr[right] = 6` 交换:

plaintext

复制代码

```
arr = [2, 3, 5, 6, 8]
```

```
left = 3, right = 3
```

外层循环结束:

- `left == right`, 双指针相遇, 完成一次分区。

`arr[left] = pivot`

`mid = left`

这两行具体的含义, 以及用示例说明

这两行代码的作用可以结合 **快速排序的分区操作** 来理解。它们的目的是:

1. `arr[left] = pivot`

把基准值 (`pivot`) 放置到最终的分区点 (`left` 或者 `right`, 这里两个指针已经重合)。

- 基准值的左侧都是小于它的元素。
 - 基准值的右侧都是大于等于它的元素。
-

2. `mid = left`

确定分区点 (`mid`)。在快速排序中, 分区点是用来递归分治的关键, 它把数组划分为两个部分:

- 左子数组: 所有元素小于基准值。
- 右子数组: 所有元素大于等于基准值。

对比：

方法	特点	优点	缺点
单指针法	遍历数组时只用一个指针记录小于基准值的分区边界	实现简单，适合初学者	可能性能不如双指针法，尤其在处理较大数据集时
双指针法	两个指针从两端向中间收缩，交换不符合的元素	更高效，适合处理大数据集，避免额外的数据移动	实现复杂度较高
随机化快速排序	在单指针或双指针的基础上引入随机化基准值	避免退化到 $O(n^2)$ 的最差情况，性能更稳定	实现时需要额外引入随机数逻辑



结论：

快速排序没有硬性规定是否使用双指针，选择哪种方法取决于需求和数据分布。双指针法是一种更经典和高效的实现，但对简单场景或教学目的，单指针法也完全够用。

十大排序总结：

三、总结表格

排序算法	时间复杂度 (最坏)	空间复杂度	稳定性	适用场景
冒泡排序	$O(n^2)$	$O(1)$	稳定	小规模数据，接近有序的数据
选择排序	$O(n^2)$	$O(1)$	不稳定	小规模数据
插入排序	$O(n^2)$	$O(1)$	稳定	小规模数据，部分有序数据
归并排序	$O(n \log n)$	$O(n)$	稳定	大规模数据排序
快速排序	$O(n^2)$	$O(\log n)$	不稳定	大规模数据排序
堆排序	$O(n \log n)$	$O(1)$	不稳定	大规模数据，稳定的排序
计数排序	$O(n + k)$	$O(k)$	稳定	数字范围较小的整数排序
桶排序	$O(n + k)$	$O(n)$	稳定	元素分布均匀的情况
基数排序	$O(nk)$	$O(n)$	稳定	整数或字符串排序

四、总结

- 比较排序**: 包括冒泡、选择、插入、归并、快速、堆等，通常适用于大多数通用排序需求。其最坏时间复杂度为 $O(n \log n)$ 。
- 非比较排序**: 包括计数排序、桶排序、基数排序等，能够打破比较排序的 $O(n \log n)$ 时间复杂度，但这些算法有额外的条件限制（如数据范围、数据分布等）。

选择排序算法时，需要考虑输入数据的特点，如数据规模、数据类型、是否有序等，才能选择合适的排序算法。



插帽龟 纹颈鸡 恩姓长者

中文名称	英文名称	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性
选择排序	Selection	n^2	n^2	n^2	1	不稳
冒泡排序	Bubble	n^2	n^2	n	1	稳
插入排序	Insertion	n^2	n^2	n	1	稳
堆排序	heap	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	1	不稳
希尔排序	Shell	$n^{1.3}$	n^2	n	1	不稳
归并排序	Merge	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	n	稳
快速排序	Quick	$n \log_2 n$	n^2	$n \log_2 n$	$n \log_2 n$	不稳
桶排序	Bucket	$n + k$	n^2	n	$n + k$	稳
计数排序	Counting	$n + k$	$n + k$	$n + k$	$n + k$	稳
基数排序	Radix	$n * k$	$n * k$	$n * k$	$n + k$	稳

5. 最长回文子串

中等

相关标签

相关企业

提示

A_文

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"

```

1 class Solution:
2     def longestPalindrome(self, s: str) -> str:
3         def expand_from_center(left,right):
4             # 从中心向两侧扩展，寻找最长回文子串
5             while left>=0 and right<len(s) and s[left]==s[right]:
6                 left-=1
7                 right+=1
8                 return s[left+1:right]
9             res=""
10            for i in range(len(s)):
11                # 奇数长度的回文（中心是一个字符）
12                odd_palind=expand_from_center(i,i)
13                # 偶数长度的回文（中心是两个字符之间的空隙）
14                even_palind=expand_from_center(i,i+1)
15                # 更新最长回文子串
16                res=max(res,odd_palind,even_palind,key=len)
17            return res

```

为什么需要 `left + 1` ?

- 因为在扩展停止时，`left` 指针已经超出了回文子串的左边界，需要右移回到有效的起点。
- 假设当前找到的回文子串为 `"aba"`，在停止时。
 - `left = -1` (多左移了一位)。
 - `right = 3` (此时指向 `"aba"` 右边多右移了一位)。
 - 因此，实际的回文子串范围是 `[left + 1, right)`。

为什么 `right` 不需要 `+1` ?

- Python 的字符串切片 `s[start:end]` 是左闭右开的：
 - 包括 `start` 位置的字符。
 - 不包括 `end` 位置的字符。
- 由于 `right` 已经超出回文子串的终点，所以直接使用它刚好切出完整的回文子串。

python

复制代码

```
odd_palindrome = expand_from_center(i, i)
```

- 奇数长度的回文：

- 以当前字符为中心 (`left = right = i`) 向两侧扩展，找到的回文子串存储在 `odd_palindrome` 中。
- 例如，字符串 `s = "babad"`，当 `i = 2` 时，`expand_from_center(2, 2)` 会找到回文 `aba`。

python

复制代码

```
even_palindrome = expand_from_center(i, i + 1)
```

- 偶数长度的回文：

- 以当前字符和下一个字符之间的空隙为中心 (`left = i, right = i + 1`) 向两侧扩展，找到的回文子串存储在 `even_palindrome` 中。
- 例如，字符串 `s = "cbbd"`，当 `i = 1` 时，`expand_from_center(1, 2)` 会找到回文 `bb`。

总结

`max()` 的参数组合：

1. `max(iterable, key=None)`
2. `max(arg1, arg2, *args, key=None)`

`key` 的用途：

- 决定如何比较元素（如长度、绝对值、某字段等）。

300. 最长递增子序列

中等

相关标签

相关企业

A+

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

415. 字符串相加

简单

相关标签

相关企业

A_文

给定两个字符串形式的非负整数 `num1` 和 `num2`，计算它们的和并同样以字符串形式返回。

你不能使用任何内建的用于处理大整数的库（比如 `BigInteger`），也不能直接将输入的字符串转换为整数形式。

示例 1：

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

示例 2：

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

示例 3：

输入: `num1 = "0"`, `num2 = "0"`
输出: "0"

思路 1：双指针

需要用字符串的形式来模拟大数加法。

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

模拟加法的做法是：

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

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

思路 1：代码

```
1  class Solution:
2      def addStrings(self, num1: str, num2: str) -> str:
3          # num1 位数
4          digit1 = len(num1) - 1
5          # num2 位数
6          digit2 = len(num2) - 1
7
8          # 进位
9          carry = 0
10         # sum 存储反向结果
11         sum = []
12         # 逆序相加
13         while carry > 0 or digit1 >= 0 or digit2 >= 0:
14             # 获取对应位数上的数字
15             num1_d = int(num1[digit1]) if digit1 >= 0 else 0
16             num2_d = int(num2[digit2]) if digit2 >= 0 else 0
17             digit1 -= 1
18             digit2 -= 1
19             # 计算结果，存储，进位
20             num = num1_d+num2_d+carry
21             sum.append('%d' % (num%10)) ←
22             carry = num // 10
23
24         # 返回计算结果
25         return "" .join(sum[::-1])
```

思路 1：复杂度分析

- **时间复杂度：** $O(\max(m + n))$ 。其中 m 是字符串 `num1` 的长度， n 是字符串 `num2` 的长度。
- **空间复杂度：** $O(\max(m + n))$ 。

在 `'%d' % (num % 10)` 中，`'%d'` 是格式字符串，而 `%` 是操作符，两者共同配合完成了将整数转为字符串的功能。



分解说明：

1. `'%d'` :

- 这是一个格式化字符串，其中 `%d` 是占位符，表示将一个整数填入这个位置。
- `d` 代表 "decimal" (十进制整数) 。

2. `%` 操作符:

- 这是 Python 的旧式字符串格式化操作符，用于将后面的值插入到格式字符串中。
- 这里 `(num % 10)` 是要插入到 `%d` 位置的值。

3. 完整过程:

- `'%d' % (num % 10)` 的含义是：将表达式 `(num % 10)` 的结果（一个整数）填入 `%d` 的占位符中，并返回一个字符串。



现代替代方法：

虽然 `'%d' % num` 可以完成转换，但它是旧式字符串格式化的用法，Python 3 更推荐以下方法：

1. 使用 `.format()` :

```
python
num = 123
last_digit_str = '{}'.format(num % 10)
print(last_digit_str) # 输出: '3'
```

复制 编辑

2. 使用 f-string (推荐，Python 3.6+) :

```
python
num = 123
last_digit_str = f'{num % 10}'
print(last_digit_str) # 输出: '3'
```

复制 编辑

2. 为什么需要判断 `carry > 0` ?

- 在处理完 `num1` 和 `num2` 的所有位数后（即 `digit1 < 0` 且 `digit2 < 0`），仍然可能存在未处理的进位。

143. 重排链表

中等

相关标签

相关企业

Ax

给定一个单链表 L 的头节点 $head$ ，单链表 L 表示为：

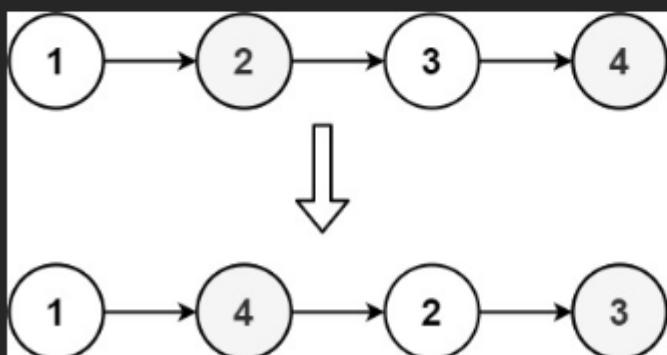
$L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$

请将其重新排列后变为：

$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1：



输入： $head = [1, 2, 3, 4]$

输出： $[1, 4, 2, 3]$

72. 编辑距离

中等

相关标签

相关企业

A_文

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` -> `orse` (将 'h' 替换为 'r')

`orse` -> `ose` (删除 'r')

`ose` -> `os` (删除 'e')

93. 复原 IP 地址

中等

相关标签

相关企业

A_x

有效 IP 地址 正好由四个整数（每个整数位于 0 到 255 之间组成，且不能含有前导 0），整数之间用 ‘.’ 分隔。

- 例如： "0.1.2.201" 和 "192.168.1.1" 是 **有效** IP 地址，但是 "0.011.255.245"、"192.168.1.312" 和 "192.168@1.1" 是 **无效** IP 地址。

给定一个只包含数字的字符串 s，用以表示一个 IP 地址，返回所有可能的 **有效 IP 地址**，这些地址可以通过在 s 中插入 ‘.’ 来形成。你 **不能** 重新排序或删除 s 中的任何数字。你可以按 **任何** 顺序返回答案。

示例 1：

输入： s = "25525511135"

输出： ["255.255.11.135", "255.255.111.35"]

示例 2：

输入： s = "0000"

输出： ["0.0.0.0"]

1143. 最长公共子序列

中等

相关标签

相关企业

提示

A_文

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列**，返回 `0`。

一个字符串的 **子序列** 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

- 例如，`"ace"` 是 `"abcde"` 的子序列，但 `"aec"` 不是 `"abcde"` 的子序列。

两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列。

示例 1：

输入: `text1 = "abcde"`, `text2 = "ace"`

输出: 3

解释: 最长公共子序列是 `"ace"`，它的长度为 3。

示例 2：

输入: `text1 = "abc"`, `text2 = "abc"`

输出: 3

解释: 最长公共子序列是 `"abc"`，它的长度为 3。

4. 寻找两个正序数组的中位数

困难

相关标签

相关企业

A_文

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

算法的时间复杂度应该为 $O(\log(m+n))$ 。

示例 1：

输入: `nums1 = [1,3]`, `nums2 = [2]`

输出: `2.00000`

解释: 合并数组 = `[1,2,3]`，中位数 `2`

示例 2：

输入: `nums1 = [1,2]`, `nums2 = [3,4]`

输出: `2.50000`

解释: 合并数组 = `[1,2,3,4]`，中位数 $(2 + 3) / 2 = 2.5$

第 83 题 删除排序链表中的重复元素

83. 删除排序链表中的重复元素

简单

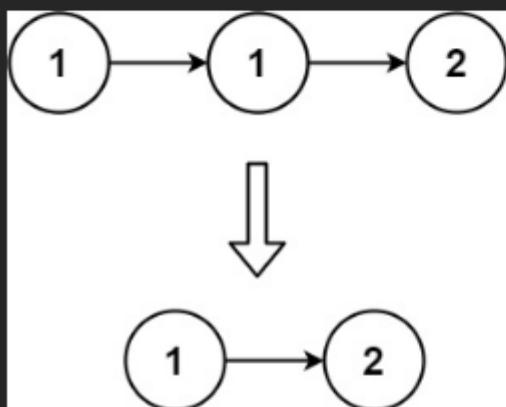
相关标签

相关企业

A_文

给定一个已排序的链表的头 `head`，删除所有重复的元素，使每个元素只出现一次。返回 已排序的链表。

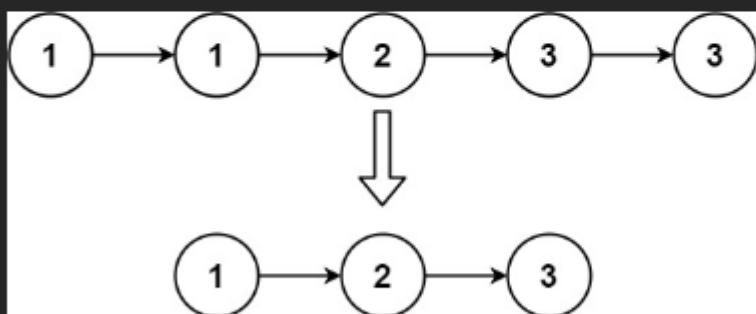
示例 1：



输入: `head = [1,1,2]`

输出: `[1,2]`

示例 2：

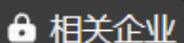


输入: `head = [1,1,2,3,3]`

输出: `[1,2,3]`

82. 删除排序链表中的重复元素 II

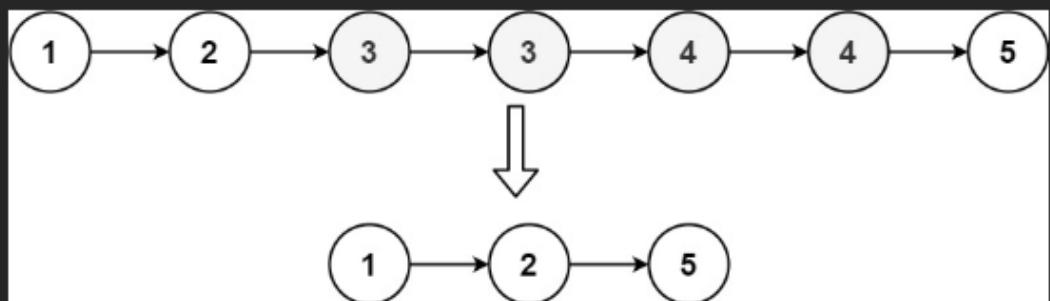
中等



A文

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。

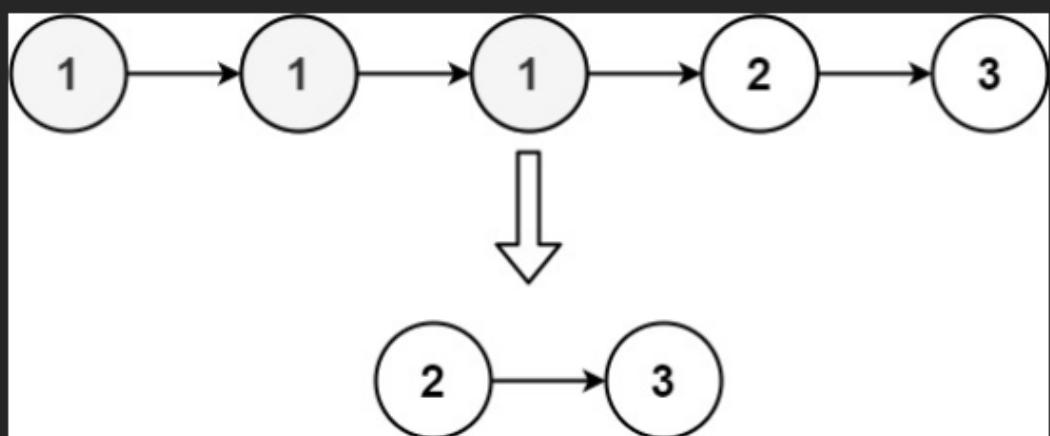
示例 1：



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]

704. 二分查找

简单

相关标签

相关企业

A_文

给定一个 n 个元素有序的 (升序) 整型数组 nums 和一个目标值 target ，写一个函数搜索 nums 中的 target ，如果目标值存在返回下标，否则返回 -1 。

示例 1:

输入: $\text{nums} = [-1, 0, 3, 5, 9, 12]$, $\text{target} = 9$

输出: 4

解释: 9 出现在 nums 中并且下标为 4

示例 2:

输入: $\text{nums} = [-1, 0, 3, 5, 9, 12]$, $\text{target} = 2$

输出: -1

解释: 2 不存在 nums 中因此返回 -1

```
1 class Solution:
2     def search(self, nums: List[int], target: int) -> int:
3         left,right=0,len(nums)-1
4         while left<=right:
5             mid=left+(right-left)//2
6             if nums[mid]==target:
7                 return mid
8             if nums[mid]<target:
9                 left=mid+1
10            else:
11                right=mid-1
12        return -1
```

已存储

测试用例 | [测试结果](#)

通过 执行用时: 0 ms

• Case 1 • Case 2 • Case 3

输入

nums =
[5]

target =
5

31. 下一个排列

中等

相关标签

相关企业

A_x

整数数组的一个 **排列** 就是将其所有成员以序列或线性顺序排列。

- 例如，`arr = [1,2,3]`，以下这些都可以视作 `arr` 的排列：
`[1,2,3]`、`[1,3,2]`、`[3,1,2]`、`[2,3,1]`。

整数数组的 **下一个排列** 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 **下一个排列** 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

- 例如，`arr = [1,2,3]` 的下一个排列是 `[1,3,2]`。
- 类似地，`arr = [2,3,1]` 的下一个排列是 `[3,1,2]`。
- 而 `arr = [3,2,1]` 的下一个排列是 `[1,2,3]`，因为 `[3,2,1]` 不存在一个字典序更大的排列。

给你一个整数数组 `nums`，找出 `nums` 的下一个排列。

必须 **原地** 修改，只允许使用额外常数空间。

题干的意思是：找出这个数组排序出的所有数中，刚好比当前数大的那个数

比如当前 `nums = [1,2,3]`。这个数是123，找出1, 2, 3这3个数字排序可能的所有数，排序后，比123大的那个数 也就是132

如果当前 `nums = [3,2,1]`。这就是1, 2, 3所有排序中最大的那个数，那么就返回1, 2, 3排序后所有数中最小的那个，也就是1, 2, 3 -> [1,2,3]

22. 括号生成

中等

相关标签

相关企业

A+

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： `["()"]`

8. 字符串转换整数 (atoi)

中等

相关标签

相关企业

A_文

请你来实现一个 `myAtoi(string s)` 函数，使其能将字符串转换成一个 32 位有符号整数。

函数 `myAtoi(string s)` 的算法如下：

1. **空格**: 读入字符串并丢弃无用的前导空格 (" ")
2. **符号**: 检查下一个字符 (假设还未到字符末尾) 为 '-' 还是 '+'. 如果两者都不存在，则假定结果为正。
3. **转换**: 通过跳过前置零来读取该整数，直到遇到非数字字符或到达字符串的结尾。如果没有读取数字，则结果为0。
4. **舍入**: 如果整数数超过 32 位有符号整数范围 $[-2^{31}, 2^{31} - 1]$ ，需要截断这个整数，使其保持在这个范围内。具体来说，小于 -2^{31} 的整数应该被舍入为 -2^{31} ，大于 $2^{31} - 1$ 的整数应该被舍入为 $2^{31} - 1$ 。

返回整数作为最终结果。

69. x 的平方根

简单

相关标签

相关企业

提示

A₂

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...，由于返回类型是整数，小数部分将被舍去。

解题思路

思路 1：二分查找

因为求解的是 x 开方的整数部分。所以我们可以从 $0 \sim x$ 的范围进行遍历，找到 $k^2 \leq x$ 的最大结果。

为了减少算法的时间复杂度，我们使用二分查找的方法来搜索答案。

```
1 class Solution:
2     def mySqrt(self, x: int) -> int:
3         left=0
4         right=x
5         ans=-1
6         while left<=right:
7             mid=left+(right-left)//2
8             if mid*mid<=x:
9                 ans=mid
10                left=mid+1
11            else:
12                right=mid-1
13        return ans
```

不过，在 Python 中，整数没有固定的大小限制，所以使用 `mid = (left + right) // 2` 不会导致溢出。Python 自动处理大整数，因此 `mid = left + (right - left) // 2` 只是在某些语言中（特别是 C/C++）的最佳实践。

结论：

- 在 Python 中，使用 `mid = (left + right) // 2` 和 `mid = left + (right - left) // 2` 都是安全的，并且没有性能差异。
- 在其他语言中，建议使用 `mid = left + (right - left) // 2` 来避免溢出问题，尤其是在 `left` 和 `right` 的值非常大的情况下。

165. 比较版本号

中等

相关标签

相关企业

提示

A_z

给你两个 **版本号字符串** `version1` 和 `version2`，请你比较它们。版本号由被点 `.` 分开的修订号组成。**修订号的值** 是它 **转换为整数** 并忽略前导零。

比较版本号时，请按 **从左到右的顺序** 依次比较它们的修订号。如果其中一个版本字符串的修订号较少，则将缺失的修订号视为 `0`。

返回规则如下：

- 如果 `version1 < version2` 返回 `-1`,
- 如果 `version1 > version2` 返回 `1`,
- 除此之外返回 `0`。

示例 1：

输入： `version1 = "1.2"`, `version2 = "1.10"`

输出： `-1`

解释：

`version1` 的第二个修订号为 `"2"`, `version2` 的第二个修订号为 `"10"`:
`2 < 10`, 所以 `version1 < version2`。

32. 最长有效括号

困难

相关标签

相关企业

A

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

示例 1：

输入：s = "(()"

输出：2

解释：最长有效括号子串是 "(())"

示例 2：

输入：s = ")()())"

输出：4

解释：最长有效括号子串是 "(()())"

示例 3：

输入：s = " " "

输出：0

LCR 140. 训练计划 II

简单

相关标签

相关企业

A+

给定一个头节点为 `head` 的链表用于记录一系列核心肌群训练项目编号，请查找并返回倒数第 `cnt` 个训练项目编号。

示例 1：

输入: `head = [2, 4, 7, 8]`, `cnt = 1`
输出: 8

算法流程：

- 初始化：** 前指针 `former`、后指针 `latter`，双指针都指向头节点 `head`。
- 构建双指针距离：** 前指针 `former` 先向前走 `cnt` 步（结束后，双指针 `former` 和 `latter` 间相距 `cnt` 步）。
- 双指针共同移动：** 循环中，双指针 `former` 和 `latter` 每轮都向前走一步，直至 `former` 走过链表 **尾节点** 时跳出（跳出后，`latter` 与尾节点距离为 `cnt - 1`，即 `latter` 指向倒数第 `cnt` 个节点）。
- 返回值：** 返回 `latter` 即可。

Python | Java | C++

```
class Solution:
    def trainingPlan(self, head: ListNode, cnt: int):
        former, latter = head, head
        for _ in range(cnt):
            former = former.next
        while former:
            former, latter = former.next, latter
        return latter
```

本题没有 $cnt >$ 链表长度的测试样例，因此不用考虑越界。考虑越界问题的代码如下：

Python | Java | C++

```
class Solution:
    def trainingPlan(self, head: ListNode, cnt: int):
        former, latter = head, head
        for _ in range(cnt):
            if not former: return
            former = former.next
        while former:
            former, latter = former.next, latter
        return latter
```

第 43 题 字符串相乘

43. 字符串相乘

中等

相关标签

相关企业

A+

给定两个以字符串形式表示的非负整数 `num1` 和 `num2`，返回 `num1` 和 `num2` 的乘积，它们的乘积也表示为字符串形式。

注意：不能使用任何内置的 `BigInteger` 库或直接将输入转换为整数。

示例 1：

输入： `num1 = "2"`, `num2 = "3"`

输出： "6"

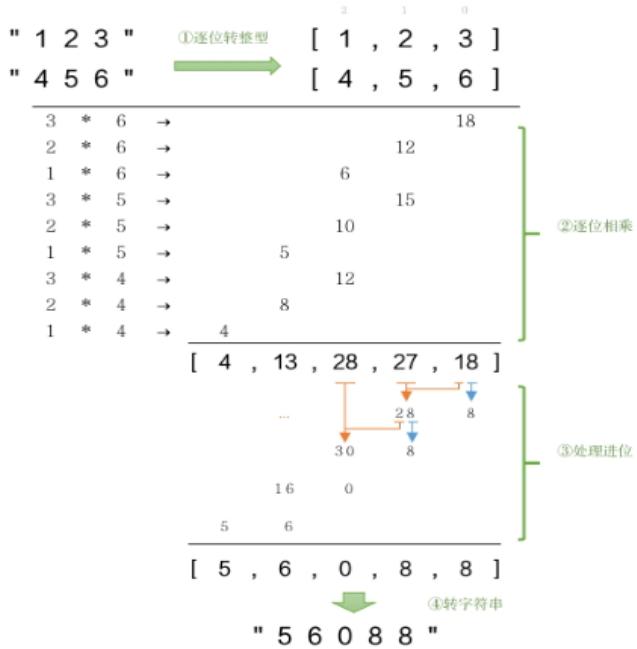
示例 2：

输入： `num1 = "123"`, `num2 = "456"`

输出： "56088"

示例：

计算： "1 2 3" * "4 5 6" = "5 6 0 8 8"



```

class Solution:
    def multiply(self, num1: str, num2: str) -> str:
        if num1 == "0" or num2 == "0":
            return "0"

        len1, len2 = len(num1), len(num2)
        nums = [0] * (len1 + len2) ←

        # 模拟逐位相乘
        for i in range(len1 - 1, -1, -1):
            for j in range(len2 - 1, -1, -1):
                mul = int(num1[i]) * int(num2[j])
                pos1, pos2 = i + j, i + j + 1
                total = mul + nums[pos2]

                nums[pos2] = total % 10
                nums[pos1] += total // 10 ←

        # 拼接结果并去除前导零
        result = ''.join(map(str, nums)).lstrip('0')
        return result

```

为什么乘积的位数最多是 $\text{len1} + \text{len2}$?

数学推导:

1. 假设 num1 和 num2 的长度分别是 len1 和 len2 , 它们的最大值是:

- num1 的最大值是 $10^{\text{len1}} - 1$ (全是 9)
- num2 的最大值是 $10^{\text{len2}} - 1$ (全是 9)

2. 两个数字相乘的最大值是:

$$(10^{\text{len1}} - 1) \times (10^{\text{len2}} - 1) = 10^{\text{len1}+\text{len2}} - 10^{\text{len1}} - 10^{\text{len2}} + 1$$

3. 从上式中可以看出, 乘积的位数最多是 $\text{len1} + \text{len2}$ (当乘积达到 $10^{\text{len1}} + 10^{\text{len2}} - 1$ 时, 结果有 $\text{len1} + \text{len2}$ 位) 。

这一步 `result = ''.join(map(str, nums)).lstrip('0')` 是将 `nums` 数组中的每个数字拼接成一个字符串，并去除前导零。我们可以分步分析每个操作的意义：

1. `map(str, nums)`

`map(str, nums)` 将 `nums` 列表中的每个数字元素转换为字符串形式。

- `nums` 是一个整数数组，每个元素表示一个数字。
- `map(str, nums)` 会对 `nums` 中的每个元素应用 `str()` 函数，将其转换为字符串。

假设 `nums = [0, 5, 5, 3, 5]`，执行 `map(str, nums)` 后，会得到：

css

复制 编辑

```
['0', '5', '5', '3', '5']
```

2. `'''.join(map(str, nums))`

`'''.join()` 是将 `map(str, nums)` 生成的字符串列表中的每个元素连接成一个单一的字符串。

- `join()` 方法是一个字符串方法，它将传入的可迭代对象（这里是 `map(str, nums)`）中的元素按照指定的分隔符连接起来。
- 在这里，使用空字符串 `''` 作为分隔符，表示将每个字符串直接连接。

假设 `map(str, nums)` 的结果是 `['0', '5', '5', '3', '5']`，那么：

bash

复制 编辑

```
'''.join(['0', '5', '5', '3', '5']) = '05535'
```

3. `.lstrip('0')`

`lstrip('0')` 是字符串的一个方法，它会去除字符串左侧的所有 `'0'` 字符。

- `lstrip('0')` 表示从字符串的左侧开始，移除所有的 `'0'`，直到遇到非 `'0'` 字符为止。

例如，如果字符串是 `'05535'`，调用 `lstrip('0')` 后，会去除左侧的 `'0'`，得到：

arduino

复制 编辑

```
'5535'
```

如果字符串本身没有 `'0'` 或已经没有左侧的 `'0'`，则原样返回。

151. 反转字符串中的单词

中等

相关标签

相关企业

A_文

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。

```
s = " h e l l o     w o r l d "
```

1. 删除首尾空格

```
s = " h e l l o     w o r l d "
```

2. 分割字符串

```
strs = ["hello", "world"]
```

3. 翻转单词列表

```
strs = ["world", "hello"]
```

4. 拼接并返回

返回值为 "world hello"

Python

```
class Solution:  
    def reverseWords(self, s: str) -> str:  
        s = s.strip()          # 删除首尾空格  
        strs = s.split()        # 分割字符串  
        这里是空格  
        strs.reverse()         # 翻转单词列表  
        return ''.join(strs)   # 拼接为字符串并返回
```

```
class Solution: / 这里是空格  
    def reverseWords(self, s: str) -> str:  
        return '' .join(s.strip().split()[:-1])
```

方法一：双指针

算法解析：

- 倒序遍历字符串 s ，记录单词左右索引边界 i, j 。
- 每确定一个单词的边界，则将其添加至单词列表 res 。
- 最终，将单词列表拼接为字符串，并返回即可。

```
class Solution:  
    def reverseWords(self, s: str) -> str:  
        s = s.strip()                                     # 删除首尾空格  
        i = j = len(s) - 1  
        res = []  
        while i >= 0:  
            while i >= 0 and s[i] != ' ': i -= 1 # 搜索首个空格  
            res.append(s[i + 1: j + 1])           # 添加单词  
            while i >= 0 and s[i] == ' ': i -= 1 # 跳过单词间空格  
            j = i                               # j 指向下个单词的尾部  
        return ' '.join(res)                         # 拼接并返回
```

第 129 题 求根节点到叶节点数字之和

129. 求根节点到叶节点数字之和

中等

相关标签

相关企业

A+

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。

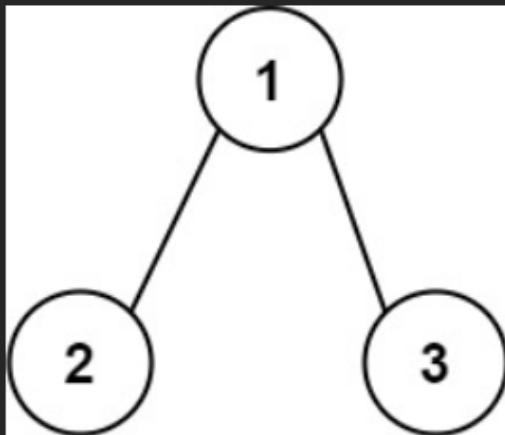
每条从根节点到叶节点的路径都代表一个数字：

- 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`

144. 二叉树的前序遍历

简单

相关标签

相关企业

Ax

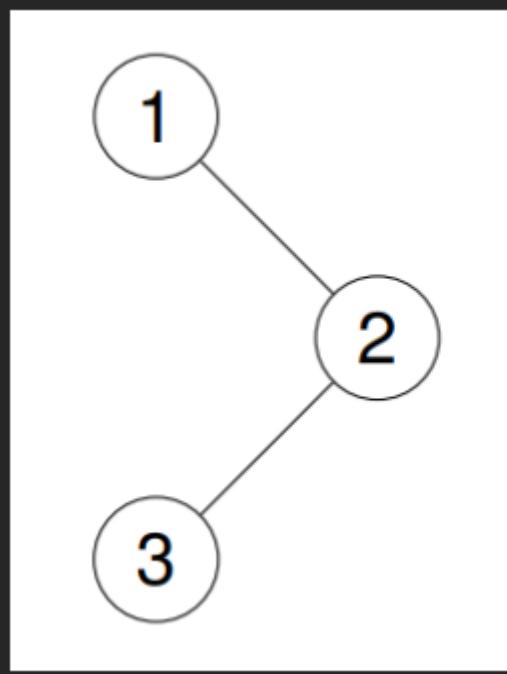
给你二叉树的根节点 `root`，返回它节点值的 前序 遍历。

示例 1：

输入： `root = [1,null,2,3]`

输出： `[1,2,3]`

解释：



第 110 题 平衡二叉树

110. 平衡二叉树

简单

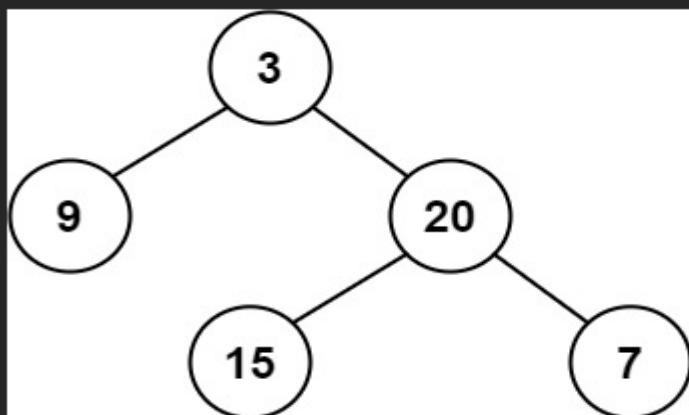
相关标签

相关企业

Ax

给定一个二叉树，判断它是否是 平衡二叉树

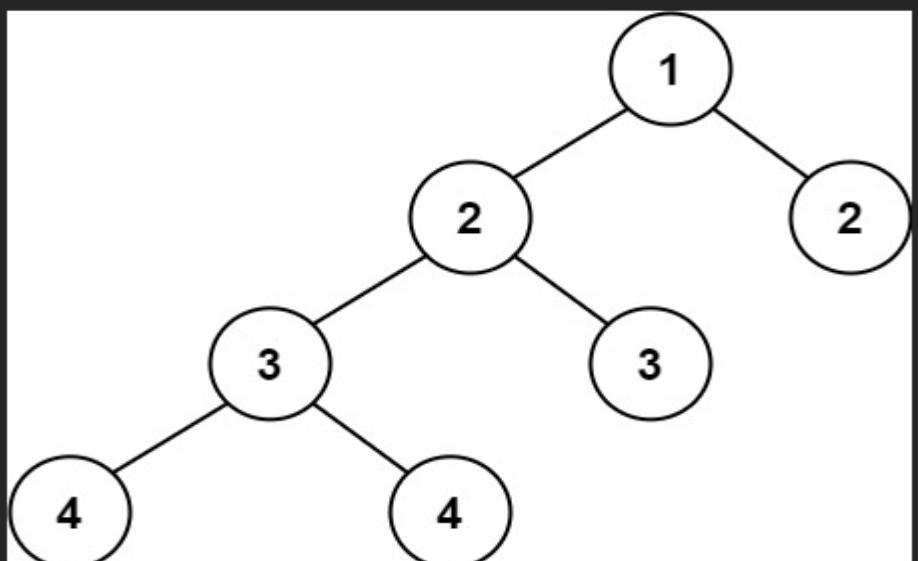
示例 1：



输入: root = [3,9,20,null,null,15,7]

输出: true

示例 2：



输入: root = [1,2,2,3,3,null,null,4,4]

输出: false

第 39 题 组合总和

39. 组合总和

中等

相关标签

相关企业

A+

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: `[]`

第 470 题 用 Rand7() 实现 Rand10()

470. 用 Rand7() 实现 Rand10()

中等

相关标签

相关企业

A+

给定方法 `rand7()` 可生成 `[1, 7]` 范围内的均匀随机整数，试写一个方法 `rand10()` 生成 `[1, 10]` 范围内的均匀随机整数。

你只能调用 `rand7()` 且不能调用其他方法。请不要使用系统的 `Math.random()` 方法。

每个测试用例将有一个内部参数 `n`，即你实现的函数 `rand10()` 在测试时将被调用的次数。请注意，这不是传递给 `rand10()` 的参数。

示例 1:

输入: 1

输出: [2]

示例 2:

输入: 2

输出: [2, 8]

示例 3:

输入: 3

输出: [3, 8, 10]

221. 最大正方形

中等

相关标签

相关企业

A_文

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

第 162 题 寻找峰值

162. 寻找峰值

中等

相关标签

相关企业

A_文

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。

第 113 题 路径总和 II

113. 路径总和 II

中等

相关标签

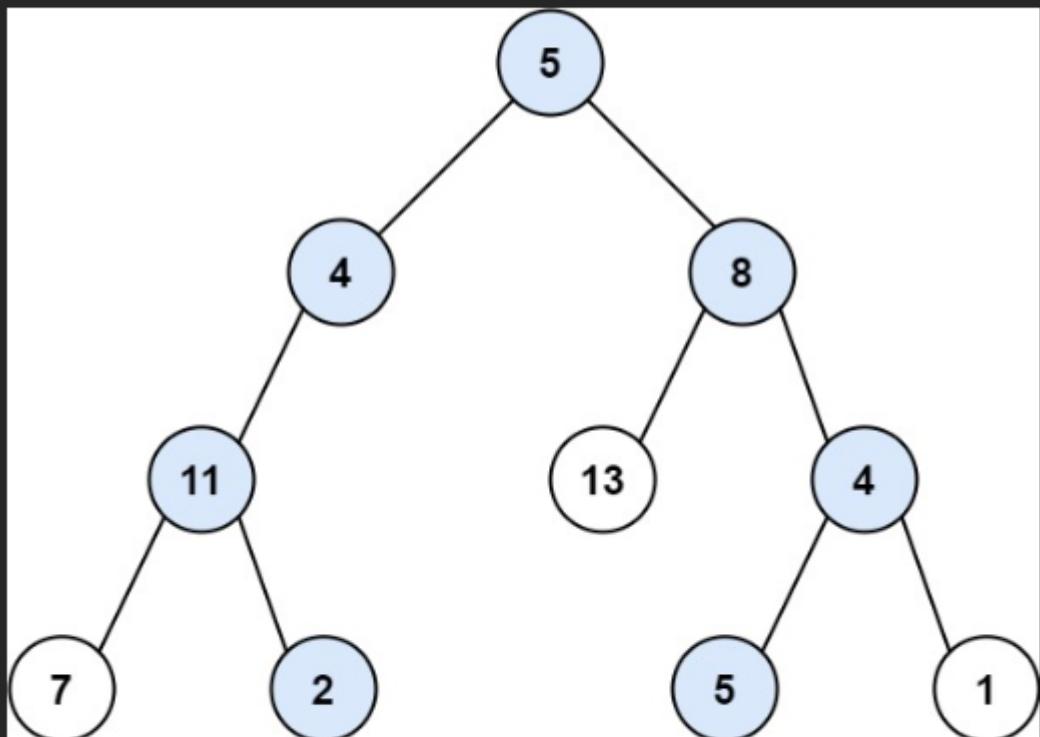
相关企业

A+

给你二叉树的根节点 `root` 和一个整数目标和 `targetSum`，找出所有 **从根节点到叶子节点** 路径总和等于给定目标和的路径。

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,5,1], targetSum = 22`
输出: `[[5,4,11,2],[5,8,4,5]]`

14. 最长公共前缀

简单

相关标签

相关企业

A+

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。

第 662 题 二叉树最大宽度

662. 二叉树最大宽度

中等

相关标签

相关企业

A+

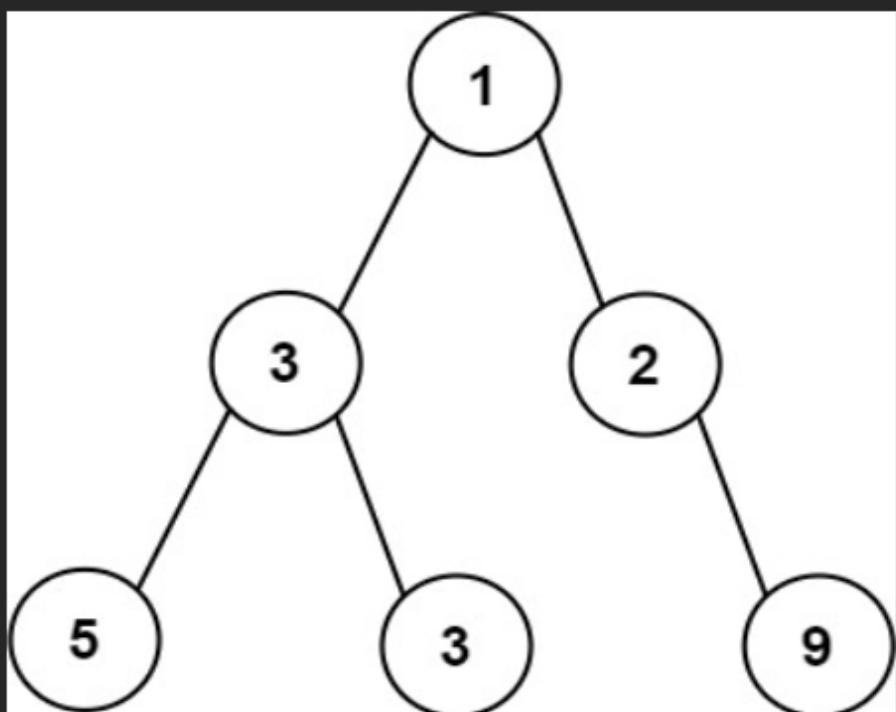
给你一棵二叉树的根节点 `root`，返回树的最大宽度。

树的最大宽度是所有层中最大的宽度。

每一层的宽度被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 `null` 节点，这些 `null` 节点也计入长度。

题目数据保证答案将会在 32 位带符号整数范围内。

示例 1：



输入： `root = [1,3,2,5,3,null,9]`

输出： 4

解释： 最大宽度出现在树的第 3 层，宽度为 4 (`5,3,null,9`)

112. 路径总和

简单

相关标签

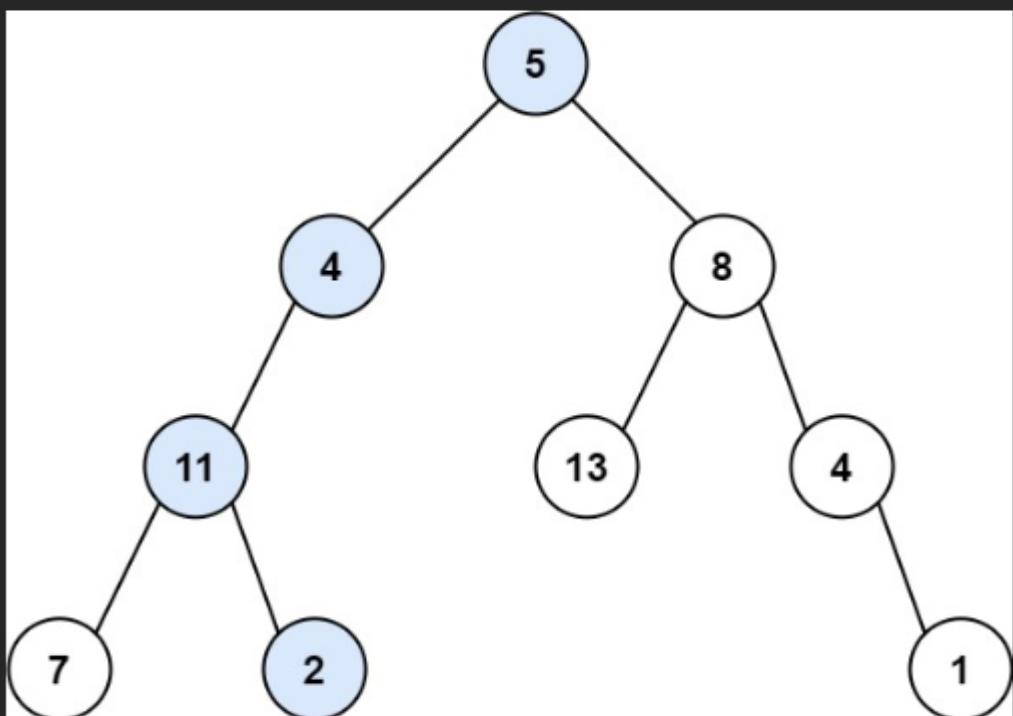
相关企业

A+

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。

152. 乘积最大子数组

中等

相关标签

相关企业

A_文

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

测试用例的答案是一个 **32-位 整数**。

示例 1:

输入: `nums = [2,3,-2,4]`

输出: `6`

解释: 子数组 `[2,3]` 有最大乘积 `6`。

示例 2:

输入: `nums = [-2,0,-1]`

输出: `0`

解释: 结果不能为 `2`，因为 `[-2,-1]` 不是子数组。

第 695 题 岛屿的最大面积

695. 岛屿的最大面积

中等

相关标签

相关企业

A+

给你一个大小为 $m \times n$ 的二进制矩阵 `grid`。

岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在 水平或者竖直的四个方向上 相邻。你可以假设 `grid` 的四个边缘都被 0 (代表水) 包围着。

岛屿的面积是岛上值为 1 的单元格的数目。

计算并返回 `grid` 中最大的岛屿面积。如果没有岛屿，则返回面积为 0。

示例 1：

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

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

输出: 6

解释: 答案不应该是 11，因为岛屿只能包含水平或垂直这四个方向上的 1。

179. 最大数

中等

相关标签

相关企业

A₂

给定一组非负整数 `nums`，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

注意：输出结果可能非常大，所以你需要返回一个字符串而不是整数。

示例 1：

输入：`nums = [10, 2]`

输出：`"210"`

示例 2：

输入：`nums = [3, 30, 34, 5, 9]`

输出：`"9534330"`

227. 基本计算器 II

中等

相关标签

相关企业

Ax

给你一个字符串表达式 `s`，请你实现一个基本计算器来计算并返回它的值。

整数除法仅保留整数部分。

你可以假设给定的表达式总是有效的。所有中间结果将在 $[-2^{31}, 2^{31} - 1]$ 的范围内。

注意：不允许使用任何将字符串作为数学表达式计算的内置函数，比如 `eval()`。

示例 1：

输入: `s = "3+2*2"`

输出: 7

示例 2：

输入: `s = " 3/2 "`

输出: 1

示例 3：

输入: `s = " 3+5 / 2 "`

输出: 5

718. 最长重复子数组

中等

相关标签

相关企业

提示

A+

给两个整数数组 `nums1` 和 `nums2`，返回 两个数组中 公共的、长度最长的子数组的长度。

示例 1：

输入: `nums1 = [1,2,3,2,1]`, `nums2 = [3,2,1,4,7]`

输出: 3

解释: 长度最长的公共子数组是 `[3,2,1]`。

示例 2：

输入: `nums1 = [0,0,0,0,0]`, `nums2 = [0,0,0,0,0]`

输出: 5

139. 单词拆分

中等

相关标签

相关企业

A₂

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 = "catsandog"`, `wordDict = ["cats", "dog", "sand", "and", "cat"]`

输出： `false`

209. 长度最小的子数组

中等

相关标签

相关企业

A+

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 $[nums_l, nums_{l+1}, \dots, nums_{r-1}, nums_r]$ ，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