

9.25.2024

704 二分法：

二分法听起来简单，在做起来时候会经常遇到边界错误，原因则在于对于边界条件判定的不一致

左闭右开时，left index 在正常判断时必然小于 right index，

举个例子：

[0, 1, 2, 3, 4]， 我们若搜索 4 的时候， 第一步骤将 array 分成[0, 2) , [3, 5)

但实际上我们永远不会再取到 2 了。

所以当其不满足判断条件时候，即为失败。

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # 左闭右开
        left = 0
        right = len(nums)
        while left < right:
            mid = (left + right) // 2
            # 判定
            # 在右边
            if nums[mid] < target:
                left = mid + 1
            # 在左边
            elif nums[mid] > target:
                right = mid
            else:
                return mid
        return -1
```

同理，左闭右闭

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        # 左闭右闭
        left = 0
        right = len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            # 判定
            # 在右边
            if nums[mid] < target:
                left = mid + 1
            # 在左边
            elif nums[mid] > target:
                right = mid - 1
            else:
                return mid
        return -1
```

27 移除元素

暴力法 $O(n^2)$:

外层找值，内层移动数据

```
def removeElement(self, nums: List[int], val: int) -> int:
    # brute force
    left, right = 0, len(nums)
    while left < right:
        if nums[left] == val:
            for i in range(left+1, right):
                nums[i - 1] = nums[i]
                right -= 1
            else:
                left += 1
    return left
```

如果 left 所在值等于 val，移除之后在下个循环仍要继续判断 left，因为需要确定新的 left 是否满足条件

双指针法只需要 $O(n)$ 的时间复杂度

```
# 双指针法
slow, fast = 0, 0
while fast < len(nums):
    if nums[fast] != val:
        nums[slow] = nums[fast]
        slow += 1
        fast += 1
    else:
        fast += 1
return slow
```

确保 slow 所在的位置即是更新后数组最后位置即可

977.有序数组的平方

最简单的方法便是算出平方后的值并且重新进行排序，时间复杂度为 $O(n\log n)$

更有效率的方法是使用双指针，因为大头永远在两端(绝对值最大的元素)

```
def sortedSquares(self, nums: List[int]) -> List[int]:
    res = [0] * len(nums)
    left, right, res_index = 0, len(nums) - 1, len(nums) - 1
    while left <= right:
        if pow(nums[left], 2) < pow(nums[right], 2):
            res[res_index] = pow(nums[right], 2)
            right -= 1
        else:
            res[res_index] = pow(nums[left], 2)
            left += 1
        res_index -= 1
    return res
```

9.26.2024

209 长度最小的子数组

首先可以想到的是暴力法，时间复杂度为 $O(n)$

其次会想到可以用两个指针来确定最小和子序列的边界，长度则为 $right - left + 1$ ，由此可以想到使用滑动窗口来解决这道题目。

```
def minSubArrayLen(self, target: int, nums: List[int]) -> int:
    left, right, res = 0, 0, float('inf')
    cur_sum = 0
    while right < len(nums):
        cur_sum += nums[right]
        while cur_sum >= target:
            res = min(res, right - left + 1)
            cur_sum -= nums[left]
            left += 1
        right += 1
    return res if right - left != len(nums) else 0
```

使用滑动窗口所需要的步骤:

- ① 定义需要维护的变量
- ② 定义窗口的左端和右端
- ③ 不断更新需要维护的变量
- ④ 最重要的来了，如何确定左端口的位置呢？

如果窗口长度固定：则用 if 判断即可，保持窗口长度不变

如果窗口长度不定：则用 while 来不断判断左端口边界，从而保证最终结果满足条件

在 step4 所有操作中，更新所有维护的变量

- ⑤ 返回答案

59.螺旋矩阵 II

```
res = generateMatrix(3)
print_matrix(res)
✓ 0.0s
```

1	2	3
8	9	4
7	6	5

```
res = generateMatrix(4)
print_matrix(res)
✓ 0.0s
```

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

这道题目感觉更像找规律：从左到右，从右到下，从右到左，从下到上为一个循环。



举个例子，

当 $n=3$ 的时候，第一层会更新 8 个元素，每个循环更新 2 个，2 次停止。

$n=4$ 的时候，第一层会更新 12 个元素，每个循环更新 3 个

第二层会更新 4 个元素，每个循环更新 1 个，2 次停止

$N=5$ 的时候，第一层会更新 16 个元素，每个循环更新 4 个

第二层会更新 8 个元素，每个循环更新 2 个，3 次停止

.....

由此发现规律，每个循环(左->右，右->下.....)，第一层偏移 $n-1$ 个，第二层偏移 $n-3$ 个.....因为每一层更新过后，单独拿出每个循环，向内缩进的过程中都有两个元素被填充了。

所以很容易得出，当对于从左到右的时候，左边界不断+1，有边界不断-1。直到 $\text{ceil}(n // 2)$ 次停止。但同时我们发现当 n 为奇数的时候，循环并不能处理到

最终的中点位置，所以为了一致性，我们单独处理中点。

那么此时 $n=3$ ，一次； $n=4$ ，两次； $n=5$ ，循环两次.... 循环次数为 $n // 2$

我们以左上点为每次循环的起点，来不断循环。代码如下：

```
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        res_matrix = [[0 for _ in range(n)] for _ in range(n)]
        loop_number = n // 2
        x, y = 0, 0
        cur_number = 1
        for offset in range(1, loop_number + 1):
            # ->
            for change_index in range(y, n - offset):
                res_matrix[x][change_index] = cur_number
                cur_number += 1
            # 下
            for change_index in range(x, n - offset):
                res_matrix[change_index][n - offset] = cur_number
                cur_number += 1
            # <-
            for change_index in range(n - offset, y, -1):
                res_matrix[n - offset][change_index] = cur_number
                cur_number += 1
            # 上
            for change_index in range(n - offset, x, -1):
                res_matrix[change_index][y] = cur_number
                cur_number += 1
            x += 1
            y += 1

        if n % 2 != 0:
            res_matrix[n // 2][n // 2] = cur_number
        return res_matrix
```

注意，每次循环结束左上点坐标都需要更新。

58. 区间和

```
import sys

input = sys.stdin.read

def main():
    data = input().split()
    index = 0
    n = int(data[index])
    index += 1
    sum_arr = []
    for i in range(n):
        sum_arr.append(int(data[index + i]))
    index += n

    for i in range(1, n):
        sum_arr[i] += sum_arr[i-1]

    # 这里是找到要输入的两个index
    while index < len(data):
        left = int(data[index])
        right = int(data[index + 1])
        index += 2
        res = sum_arr[right] - sum_arr[left - 1] if left > 0 else sum_arr[right]
        print(res)

if __name__ == "__main__":
    main()
```

这道题的难点主要是如何利用输入流来获取信息

```
5
1
2
3
4
5
0 2
2 4
^D
^Z
6
12
```

需要使用 ctrl+Z 来完成输入

44. 开发商购买土地

题目需要利用到前缀和

首先要求出列、行的总和

然后遍历可能的行切分、列切分来找到最小的差值

逻辑想通，代码不难

```
def main():  
    input = sys.stdin.read  
    data = input().split()  
    index = 0  
    row_num = int(data[index])  
    index += 1  
    col_num = int(data[index])  
    index += 1  
  
    matrix = []  
  
    for row in range(row_num):  
        row_arr = []  
        for col in range(col_num):  
            row_arr.append(int(data[index]))  
            index += 1  
        matrix.append(row_arr)  
  
    row_sum, col_sum = [0] * row_num, [0] * col_num  
  
    for i in range(row_num):  
        for j in range(col_num):  
            row_sum[i] += matrix[i][j]  
  
    for j in range(col_num):  
        for i in range(row_num):  
            col_sum[j] += matrix[i][j]  
  
    res = float('inf')  
  
    # 遍历横切和纵切  
    temp_sum = 0  
    for i in range(len(row_sum)):  
        temp_sum += row_sum[i]  
        other_temp_sum = sum(row_sum) - temp_sum  
        res = min(res, abs(other_temp_sum - temp_sum))  
  
    temp_sum = 0  
    for i in range(len(col_sum)):  
        temp_sum += col_sum[i]  
        other_temp_sum = sum(col_sum) - temp_sum  
        res = min(res, abs(other_temp_sum - temp_sum))  
  
    print(res)
```


9.27.2024

203 移除链表元素

设置虚拟头节点使得操作一致性

```
class Solution:
    def removeElements(self, head: Optional[ListNode], val: int) -> Optional[ListNode]:
        dummyhead = ListNode(0, head)
        cur = dummyhead
        while cur.next:
            if cur.next.val != val:
                cur = cur.next
            else:
                cur.next = cur.next.next
        return dummyhead.next
```

非常简单

接下来的方法不再设置虚拟头节点，直接对头节点进行操作，确保头节点的值

不再等于 val

```
if not head:
    return head
while head and head.val == val:
    head = head.next
cur = head
# 此时head.val != val
while cur and cur.next:
    if cur.next.val == val:
        cur.next = cur.next.next
    else:
        cur = cur.next
return head
```

注意：要先判断当前指针是否存在

707.设计链表

本题难点在于许多边界的判断，在遇到空值的时候应该如何处理

首先最重要的是 **index 从 0 开始!!!**

初始值规定了我们在进行 get、addAtIndex、delete 操作中边界的判断问题

```
def get(self, index: int) -> int:
    if index >= self.size or index < 0:
        return -1
    cur = self.dummyNode.next
    for i in range(index):
        cur = cur.next
    return cur.val

def addAtIndex(self, index: int, val: int) -> None:
    if index > self.size or index < 0:
        return

    node = ListNode(val)
    cur = self.dummyNode
    for i in range(index):
        cur = cur.next
    node.next = cur.next
    cur.next = node
    self.size += 1
    return

def deleteAtIndex(self, index: int) -> None:
    if index < 0 or index >= self.size:
        return

    cur = self.dummyNode
    for i in range(index):
        cur = cur.next

    cur.next = cur.next.next
    self.size -= 1
    return
```

可以看到，在以上三个操作中都出现了边界值的判断问题。

查找是位于 index，而增删操作需要找到前一个 index 的 node

206.反转链表

```
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        pre, cur, nextnode = None, head, None

        while cur:
            nextnode = cur.next
            cur.next = pre
            pre = cur
            cur = nextnode

        return pre
```

较为简单的双指针法

9.28.2024

24. 两两交换链表中的节点

```
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
7 class Solution:
8     def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
9         dummyNode = ListNode(0, head)
10        cur = dummyNode
11        while cur.next and cur.next.next:
12            temp_one = cur.next
13            temp_third = cur.next.next.next
14
15            cur.next = temp_one.next
16            cur.next.next = temp_one
17            temp_one.next = temp_third
18
19            cur = cur.next.next
20
21        return dummyNode.next
```

虚拟头节点真好用嘻嘻

19.删除链表的倒数第 N 个节点

第一时间想到的是先遍历一遍，第二遍遍历找到倒数第 N 个节点的前一个节点。

简单做法是使用双指针，倒数第 n 个节点，可以让 fast 指针先走 n 步，这样和 slow 的距离就有 n，当 fast 继续移动到 NULL 的时候，此时 slow 指向的指针就是待删除节点。可以画图来判断具体行进步数。



我们需要删除的是节点 3，所以 fast 从 dumminode 要先走 n+1 步(因为要找到的是待删除节点的前一个节点)

```
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
7 class Solution:
8     def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
9         dummyNode = ListNode(0, head)
10        slow = fast = dummyNode
11        for i in range(n + 1):
12            fast = fast.next
13
14        while fast:
15            fast = fast.next
16            slow = slow.next
17
18        slow.next = slow.next.next
19        return dummyNode.next
```

双指针法，时间复杂度 $O(n)$

160.链表相交

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> Optional[ListNode]:
        lenA, lenB = 0, 0
        cur = headA
        while cur:
            lenA += 1
            cur = cur.next
        cur = headB
        while cur:
            lenB += 1
            cur = cur.next

        if lenB > lenA:
            headA, headB = headB, headA
            lenA, lenB = lenB, lenA

        curA, curB = headA, headB
        for i in range(lenA - lenB):
            curA = curA.next

        while curA:
            if curA == curB:
                return curA
            else:
                curA = curA.next
                curB = curB.next
        return None
```

双指针法，时间复杂度 $O(n)$

142.环形链表 II

```
class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        slow = fast = head
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next

            if slow == fast:
                slow = head
                while slow != fast:
                    slow = slow.next
                    fast = fast.next
                return slow
        return None
```

数学问题

明确：相遇点一定在圈内，因为 fast 比 slow 快

从相遇节点 再到环形入口节点节点数为 z 。如图所示：



那么相遇时：slow指针走过的节点数为 $x + y$ ，fast指针走过的节点数为 $x + y + n(y + z)$ ， n 为fast指针在环内走了 n 圈才遇到slow指针， $(y+z)$ 为一圈内节点的个数 A 。

因为fast指针是一步走两个节点，slow指针一步走一个节点，所以fast指针走过的节点数 = slow指针走过的节点数 * 2：

$$(x + y) * 2 = x + y + n(y + z)$$

两边消掉一个 $(x+y)$ ： $x + y = n(y + z)$

因为要找环形的入口，那么要求的是 x ，因为 x 表示头结点到环形入口节点的距离。

所以要求 x ，将 x 单独放在左边： $x = n(y + z) - y$ ，

再从 $n(y+z)$ 中提出一个 $(y+z)$ 来，整理公式之后为如下公式： $x = (n - 1)(y + z) + z$ 注意这里 n 一定是大于等于1的，因为fast指针至少要多走一圈才能相遇slow指针。

9.30.2024

可以用三种数据结构实现哈希：

- ① 数组
- ② 集合
- ③ 映射 map

在C++中，set 和 map 分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	$O(\log n)$	$O(\log n)$
std::multiset	红黑树	有序	是	否	$O(\log n)$	$O(\log n)$
std::unordered_set	哈希表	无序	否	否	$O(1)$	$O(1)$

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	$O(\log n)$	$O(\log n)$
std::multimap	红黑树	key有序	key可重复	key不可修改	$O(\log n)$	$O(\log n)$
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	$O(1)$	$O(1)$

都是使用了空间换时间以迅速查找

242.有效的字母异位词

```
from collections import defaultdict

class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        s_dict = defaultdict(int)
        for x in s:
            s_dict[x] += 1

        for x in t:
            if x not in s_dict.keys():
                return False
            else:
                s_dict[x] -= 1
                if s_dict[x] == 0:
                    del s_dict[x]

        return len(s_dict) == 0
```

使用 defaultdict

349. 两个数组的交集

使用数组来做哈希的题目，是因为题目都限制了数值的大小。

本题中没有限制数值大小

```
from collections import defaultdict

class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:
        dic = defaultdict(int)
        for num in nums1:
            dic[num] += 1
        res = set()
        for num in nums2:
            if num in dic.keys():
                res.add(num)

        return res
```

202. 快乐数

这道题非常巧妙的运用了哈希算法，如果各位平方和相加之后的数等于之前出现过的，那么便是循环，返回 False。

```
class Solution:
    def isHappy(self, n: int) -> bool:
        appear_value = set()
        while n not in appear_value:
            appear_value.add(n)

            temp = 0
            for char in str(n):
                temp += int(char) ** 2
            if temp == 1:
                return True
            else:
                n = temp
        return False
```

1. 两数之和

首先想到的是暴力法，时间复杂度是 $O(n^2)$

但感觉肯定存在 $O(n)$ 的方法，使用字典

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        record = dict()
        for index, num in enumerate(nums):
            if target - num in record:
                return [record[target - num], index]
            record[num] = index
        return
```

使用 set(), 和上面一个意思

```
record = set()
for index, num in enumerate(nums):
    if target - num in record:
        return [nums.index(target - num), index]
    record.add(num)
return
```

双指针来了, 时间复杂度为 $O(n\log n)$

```
nums_inorder = sorted(nums)
left, right = 0, len(nums) - 1
while left <= right:
    temp = nums_inorder[left] + nums_inorder[right]
    if temp == target:
        left_index = nums.index(nums_inorder[left])
        right_index = nums.index(nums_inorder[right])
        if left_index == right_index:
            right_index = nums[left_index+1:].index(nums_inorder[right]) + left_index + 1
            break
    elif temp > target:
        right -= 1
    else:
        left += 1
return [left_index, right_index]
```

注意: 因为题目中要求我们每个数据只能用一次并且, 可能含有重复的元素

所以, 我们需要确认每个相等的元素各自在列表中的位置。

注意:

+ left_index + 1: 由于 .index() 返回的是相对于子列表的索引, 因此需要加

上 left_index + 1 来转换为相对于原始列表 nums 的索引

建议写个例子看看, 一写就明白。

10.05.2024

435. 无重叠区间

一开始一直没有想到办法，只感觉要先排序，但是不清楚排序之后应该如何确定哪块是应该保存的。

Carl 大佬使用的贪心：以右边界排序，右边界越小，代表能保存的越多（因为留给右侧的空间越大），一开始久久不能理解。但后来才想通这是贪心算法的真谛。局部最优：右边界最小；全局最优：保留的越多

```
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        if not intervals:
            return 0
        not_overlap = 1
        intervals.sort(key=lambda x : x[1])
        right = intervals[0][1]

        for i in range(1, len(intervals)):
            if intervals[i][0] >= right:
                not_overlap += 1
                right = intervals[i][1]

        return len(intervals) - not_overlap
```

感觉贪心像是脑筋急转弯，是个连环锁头。只有明确了眼下局部的正确，才能以此为判断全局。但是需要注意的是，并不是局部正确一定能得到全局正确。

10.06.2024

454.四数相加 II

```
def fourSumCount(self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]) -> int:
    dic = dict()
    for num1 in nums1:
        for num2 in nums2:
            if num1 + num2 not in dic:
                dic[num1 + num2] = 1
            else:
                dic[num1 + num2] += 1
    res = 0
    for num3 in nums3:
        for num4 in nums4:
            if -num3 - num4 in dic:
                res += dic[-num3 - num4]
    return res
```

之后会把三数之和和四数之和都添加进来，使用不同的办法来做为对比。

15.三数之和

本题目中不再使用不同的数组来存储元素，而是使用同一个数组。那么此时如何使用哈希呢，或者说还可以用哈希吗？我的想法：手动将数组分成三组。

```
from collections import defaultdict
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        dic = defaultdict(list)
        split = len(nums) // 3
        nums1 = nums[:split]
        nums2 = nums[split:split*2]
        nums3 = nums[split*2:]
        for num1 in nums1:
            for num2 in nums2:
                temp = [num1, num2]
                dic[num1 + num2].append(temp)
        res = []
        for num3 in nums3:
            if -num3 in dic:
                for group in dic[-num3]:
                    temp = group.copy()
                    temp.append(num3)
                    if temp not in res:
                        res.append(temp)
        return res
```

这样的写法会错过需要可能的结果，比如[-2,0,1,1,2]这组例子

还有个思路：先用两层循环确定，最后再用哈希，并且使用去重算法。时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(n)$

最简单的方法，使用双指针

```
from collections import defaultdict
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        res = []
        nums.sort()
        if len(nums) == 0:
            return []
        if nums[0] > 0:
            return []
        for i in range(len(nums)):
            if i > 0 and nums[i] == nums[i-1]:
                continue
            left = i + 1
            right = len(nums) - 1
            while left < right:
                temp = nums[i] + nums[left] + nums[right]
                if temp > 0:
                    right -= 1
                elif temp < 0:
                    left += 1
                else:
                    res.append([nums[i], nums[left], nums[right]])
                    while right > left and nums[right] == nums[right-1]:
                        right -= 1
                    while right > left and nums[left] == nums[left + 1]:
                        left += 1
                    left += 1
                    right -= 1
        return res
```

难点在于相同元素的去除。排序后对于相同算法去除的难度降低，只需要确保每一次符合条件的元素不一样即可，例如: [-2, -1, 0, 2, 3] 答案为 [-2, -1, 3], [-2, 0, 2]。

其实不一样!

都是和 `nums[i]` 进行比较, 是比较它的前一个, 还是比较它的后一个。

如果我们的写法是 这样:

```
1   if (nums[i] == nums[i + 1]) { // 去重操作
2       continue;
3   }
```

那我们就把 三元组中出现重复元素的情况直接pass掉了。例如 `{-1, -1, 2}` 这组数据, 当遍历到第一个 `-1` 的时候, 判断 下一个也是 `-1`, 那这组数据就pass了。

我们要做的是 不能有重复的三元组, 但三元组内的元素是可以重复的!

所以这里是有两个重复的维度。

那么应该这么写:

```
1   if (i > 0 && nums[i] == nums[i - 1]) {
2       continue;
3   }
```

这么写就是当前使用 `nums[i]`, 我们判断前一位是不是一样的元素, 在看 `{-1, -1, 2}` 这组数据, 当遍历到第一个 `-1` 的时候, 只要前一位没有 `-1`, 那么 `{-1, -1, 2}` 这组数据一样可以收录到 结果集里。

这是一个非常细节的思考过程。

以上这段话非常非常非常重要。做到不忽略任何可能性, 那我们需要做到让事情尽可能的早发生, 而不是在之后发生。

```
if temp > 0:
    right -= 1
elif temp < 0:
    left += 1
else:
    res.append([nums[i], nums[left], nums[right]])
    while right > left and nums[right] == nums[right-1]:
        right -= 1
    while right > left and nums[left] == nums[left + 1]:
        left += 1
    left += 1
    right -= 1
```

这段代码的核心在于去重, 首先满足条件的时候将其加入最终结果, while 一定会使最终的 `left` 和 `right` 位于等于上一个值的位置。最终对 `left` 和 `right` 进行同步变化来维持一致性。

18.四数之和

模仿上题思路，双指针+两层 for 循环嵌套

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        res = []
        nums.sort()
        for i in range(len(nums)):
            if i > 0 and nums[i-1] == nums[i]:
                continue
            for j in range(i+1, len(nums)):
                if nums[j-1] == nums[j] and j > i + 1:
                    continue
                left, right = j + 1, len(nums) - 1
                while left < right:
                    temp = nums[i] + nums[j] + nums[left] + nums[right]
                    if temp > target:
                        right -= 1
                    elif temp < target:
                        left += 1
                    else:
                        res.append([nums[i], nums[j], nums[left], nums[right]])
                        while left < right and nums[left] == nums[left + 1]:
                            left += 1
                        while left < right and nums[right] == nums[right - 1]:
                            right -= 1
                        left += 1
                        right -= 1
                return res
```

总结：

在使用双指针的过程中，可以将时间复杂度降低一个数量级。什么时候可以使用双指针？对于不需要返回索引的问题。因为排序后会打乱索引。双指针定位的过程中注意去重的问题——边界的定位

383. 赎金信

```
from collections import defaultdict
class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        dic = defaultdict(int)
        for char in magazine:
            dic[char] += 1
        for char in ransomNote:
            if char not in dic:
                return False
            if dic[char] > 0:
                dic[char] -= 1
                if dic[char] == 0:
                    del dic[char]
        return True
```

这道题目还是比较简单的


```
def canConstruct(self, ransomNote: str, magazine: str) -> bool:
    # dic = defaultdict(int)
    # for char in magazine:
    #     dic[char] += 1
    # for char in ransomNote:
    #     if char not in dic:
    #         return False
    #     if dic[char] > 0:
    #         dic[char] -= 1
    #     if dic[char] == 0:
    #         del dic[char]
    # return True

    for char in ransomNote:
        if ransomNote.count(char) <= magazine.count(char):
            continue
        else:
            return False
    return True
```

10.07.2024

344. 反转字符串

较为简单

```
def reverseString(self, s: List[str]) -> None:
    """
    Do not return anything, modify s in-place instead.
    """
    left, right = 0, len(s) - 1
    while left < right:
        temp = s[right]
        s[right] = s[left]
        s[left] = temp
        left += 1
        right -= 1
```

稍微有技巧性的方法:

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

541. 反转字符串 II

```
class Solution:
    def reverseStr(self, s: str, k: int) -> str:

        s = list(s)

        def reverse(s):
            n = len(s)
            for i in range(n // 2):
                s[i], s[n - i - 1] = s[n - i - 1], s[i]
            return s

        if len(s) < k:
            s = s[::-1]
            return s

        for i in range(0, len(s), 2 * k):
            s[i:i+k] = reverse(s[i:i+k])

        return "".join(str(x) for x in s)
```

很惭愧，开始把 str 和 list 很多 reverse 内置函数弄晕了。

```

import math

class Solution:
    def reverseStr(self, s: str, k: int) -> str:
        s = list(s)

        def reverse(s):
            n = len(s)
            for i in range(n // 2):
                s[i], s[n - i - 1] = s[n - i - 1], s[i]
            return s

        if len(s) < k:
            s = s[::-1]
            return s

        times = math.ceil(len(s) / 2)

        for i in range(0, times):
            s[2 * k * i : 2 * k * i + k] = reverse(s[2 * k * i : 2 * k * i + k])

        return "".join(str(x) for x in s)

class Solution:
    def reverseStr(self, s: str, k: int) -> str:
        cur = 0
        while cur <= len(s) - 1:
            next = cur + k
            s = s[:cur] + s[cur:next][::-1] + s[next:]
            cur += 2 * k
        return s

```

在字符串中，当：切片超过了字符串的长度时候，会截取到最后一个元素。列表同理。

卡码网：54.替换数字

数组填充类问题，第一步先给数组扩容，其次由后向前填充。

```

s = input()
res = "".join(['number' if x.isdigit() else x for x in s])
print(res)

```

10.08.2024

151.翻转字符串里的单词

- 当 sep 为 None 时，split() 会自动去除字符串开头和结尾的空白字符，并以任意长度的空白字符作为分隔符。

```
def reverseWords(self, s: str) -> str:
    words = s.split()
    left, right = 0, len(words) - 1
    while left < right:
        words[left], words[right] = words[right], words[left]
        left += 1
        right -= 1

    return " ".join(words)
```

人生苦短，我用 python

#理论上，本体难点之一在于去除多余空格

```
while fastindex < len(s):
    if fastindex > 0 and s[fastindex - 1] == s[fastindex] and s[fastindex] == " ":
        fastindex += 1
    else:
        s[slowindex] = s[fastindex]
        slowindex += 1
        fastindex += 1
```

不过 split 函数已经帮我们解决了这个问题。

459.重复的子字符串

```
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        leng = len(s)
        for i in range(1, leng // 2 + 1):
            if leng % i == 0:
                subs = s[:i]
                if subs * (leng // i) == s:
                    return True
        return False
```

卡码网：55.右旋转字符串

```
1 def main():
2     count = int(input())
3     str_ = input()
4     str_ = str_[::-1]
5     res = str_[:count][::-1] + str_[count:][::-1]
6     print(res)
7
8 main()
```

写的有些麻烦，实际上直接 `str_[-n:] + str_[:-n]` 就行

28 寻找子字符串出现的 index

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        if haystack == needle:
            return 0
        left, right = 0, len(needle) - 1
        while right < len(haystack):
            if haystack[left : right + 1] == needle:
                return left
            right += 1
            left += 1
        return -1
```

双指针算法。今天的题目写起来都是感觉到了 python 的便捷性，但同时相应的，对底层算法的理解不足。

总结：

字符串不能变，转化为列表

双指针可以降低一个等级的时间复杂度，在字符串和数组、链表中非常常用。

10.09.2024

232.用栈实现队列

```
class MyQueue:

    def __init__(self):
        self.instack = []
        self.outstack = []

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

    def pop(self) -> int:
        if self.empty():
            return None

        if self.outstack:
            return self.outstack.pop()

        else:
            for i in range(len(self.instack)):
                self.outstack.append(self.instack.pop())
            return self.outstack.pop()

    def peek(self) -> int:
        res = self.pop()
        self.outstack.append(res)
        return res

    def empty(self) -> bool:
        return not (self.instack or self.outstack)
```

数列 1 append 进入，pop 为出模仿输入栈

数列 2 由数列 1 的 pop 进入，再由自己的 pop 模仿输出栈

将两个数列看作一个整体，一起空才为空

225. 用队列实现栈

数列一需要把除了最后一个数据的其他数据移动到数列二，以实现出栈功能，

再将数据移动回来。实际上用一个数列也可以

```

class MyStack:

    def __init__(self):
        self.queue = deque()

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

    def pop(self) -> int:
        if self.empty():
            return None
        else:
            for i in range(len(self.queue) - 1):
                self.queue.append(self.queue.popleft())
            return self.queue.popleft()

    def top(self) -> int:
        if self.empty():
            return None
        else:
            for i in range(len(self.queue) - 1):
                self.queue.append(self.queue.popleft())
            res = self.queue.popleft()
            self.queue.append(res)
            return res

    def empty(self) -> bool:
        return not self.queue

```

20. 有效的括号

三种 False 的情况

- ① 匹配数组已经为空
- ② 不对应
- ③ 当所有匹配字符结束时数组不为空

```

class Solution:
    def isValid(self, s: str) -> bool:
        char_ls = []

        for char in s:
            if char == '{':
                char_ls.append('}')
            elif char == '(':
                char_ls.append(')')
            elif char == '[':
                char_ls.append(']')
            elif not char_ls or char != char_ls[-1]:
                return False
            else:
                char_ls.pop()

        return False if char_ls else True

```

1047. 删除字符串中的所有相邻重复项

```

class Solution:
    def removeDuplicates(self, s: str) -> str:
        temp = []
        for char in s:
            temp.append(char)
            if len(temp) > 1:
                temp_cur = temp.pop()
                temp_pre = temp.pop()
                if temp_pre == temp_cur:
                    continue
                else:
                    temp.append(temp_pre)
                    temp.append(temp_cur)
        return "".join(temp)

```

比较简单，一定是有一个 char 先添加进来，才能比较一对是否相同。

简单写法：

```

class Solution:
    def removeDuplicates(self, s: str) -> str:
        temp = []
        for char in s:
            if temp and temp[-1] == char:
                temp.pop()
            else:
                temp.append(char)
        return "".join(temp)

```


没有想到双指针：

```
class Solution:
    def removeDuplicates(self, s: str) -> str:
        slow, fast = 0, 0
        leng = len(s)
        res = list(s)
        while fast < leng:
            res[slow] = res[fast]
            if slow > 0 and res[slow] == res[slow - 1]:
                slow -= 1
            else:
                slow += 1
            fast += 1
        return "".join(res[0:slow])
```

在相同的时候往后退一个保证了没有相邻的是相同的。

10.10.2024

150. 逆波兰表达式求值

显然这是一个栈完成的例子，数字压栈，符号出栈

```
def evalRPN(self, tokens: List[str]) -> int:
    char_symbol = ['+', '-', '*', '/']
    process_stack = []
    for char in tokens:
        if char in char_symbol:
            cur = process_stack.pop()
            pre = process_stack.pop()
            if char == "+":
                temp = cur + pre
            elif char == "-":
                temp = pre - cur
            elif char == "*":
                temp = pre * cur
            else:
                temp = int(pre / cur)
            process_stack.append(temp)
        else:
            process_stack.append(int(char))
    return process_stack[0]
```

二叉树的后序遍历，左右中

239. 滑动窗口最大值

本题目中所说的栈不是非寻常意义上的栈，只代表一个容器（deque 是双端队列）

Hard 难度：需要自定义队列，确保队列元素单调递减，这样才能保证一直取到当前窗口内最大值。在压栈的时候要比较新入栈的元素是不是会更大。

Push：因为在入栈的时候我们要保证顺序关系，确保单调递减。那么当该值大于之前的值时之前所有比他小的都要出栈，以确保能更新窗口内大小顺序。

Pop: 为了确保栈的更新速度与滑动窗口的滑动速度一致, 当左窗口移动时并且 pass 过那个最大值的时候, 才会 drop 掉。

```
from collections import deque
class decreasingQueue:
    def __init__(self):
        self.deque = deque()

    def pop(self, x):
        if self.deque and self.deque[0] == x:
            self.deque.popleft()

    def push(self, x):
        while self.deque and self.deque[-1] < x:
            self.deque.pop()
        self.deque.append(x)

    def getfront(self):
        return self.deque[0]

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        stack = decreasingQueue()
        res = []
        for i in range(k):
            stack.push(nums[i])
            res.append(stack.getfront())

        for i in range(k, len(nums)):
            stack.pop(nums[i - k])
            stack.push(nums[i])
            res.append(stack.getfront())

        return res
```

347.前 K 个高频元素

对最小堆用法不熟练

需要用哈希遍历出现次数, 用最小堆保留 k 个出现频率最大的并且返回。

最初我想用大顶堆, 但是听了 carl 说的才知道了自己的无知。举个例子:

一开始的大顶堆[2, 4, 5], k=3。此时进入元素 2, 我们 pop 掉 5, 那么现在[2, 4, 2], 最大的已经被排除了, 那么结果怎么返回呢?

```
from collections import defaultdict
import heapq

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        map_ = defaultdict(int)
        for num in nums:
            map_[num] += 1

        min_que = []

        for key, value in map_.items():
            heapq.heappush(min_que, (value, key))
            if len(min_que) > k:
                heapq.heappop(min_que)

        res = [0] * k
        for i in range(k-1, -1, -1):
            res[i] = heapq.heappop(min_que)[1]
        return res
```

10.11.2024

144.二叉树的前序遍历

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

```
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return
        stack = []
        res = []
        stack.append(root)
        while stack:
            temp_node = stack.pop()
            res.append(temp_node.val)
            if temp_node.right:
                stack.append(temp_node.right)
            if temp_node.left:
                stack.append(temp_node.left)
        return res
```

前序是最简单的，后面的后序和中序不再写递归，只写迭代算法

145. 二叉树的后序遍历

逻辑法：

在整个过程中，如果左孩子右孩子都存在，那就需要走过两次父节点。需要用 pre 指针来判定右子孩子有没有遍历过，不然就无限循环了。

```
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        stack, res = [], []
        pre = None
        while root or stack:
            if root:
                stack.append(root)
                root = root.left
            else:
                cur = stack[-1]
                if not cur.right or cur.right == pre:
                    cur = stack.pop()
                    pre = cur
                    root = None
                    res.append(cur.val)
                else:
                    root = cur.right
        return res
```

规律法：

后序遍历顺序：左右中

前序遍历顺序：中左右 → 中右左 → 左右中

```
if not root:
    return
stack, res = [], []
stack.append(root)
while stack:
    cur = stack.pop()
    res.append(cur.val)
    if cur.left:
        stack.append(cur.left)
    if cur.right:
        stack.append(cur.right)
return res[::-1]
```

94.二叉树的中序遍历

```
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        stack, res = [], []
        while root or stack:
            if root:
                stack.append(root)
                root = root.left
            else:
                root = stack.pop()
                res.append(root.val)
                root = root.right
        return res
```

102.二叉树的层序遍历

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

这里有一个比较有意思的就是在 python 中，for 循环中 len(que)的值是固定的，而在 C++中，这个值在每次循环的时候会发生变化。

107.二叉树的层次遍历 II

```

class Solution:
    def levelOrderBottom(self, root: Optional[TreeNode]) -> List[List[int]]:
        res, que = [], deque()
        if not root:
            return
        que.append(root)
        while que:
            temp = []
            for i in range(len(que)):
                cur = que.popleft()
                temp.append(cur.val)
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
            res.append(temp)
        return res[::-1]

```

199.二叉树的右视图

Deque 不支持切片操作

```

class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        res, que = [], deque()
        if not root:
            return
        que.append(root)
        while que:
            leng = len(que)
            for i in range(leng):
                cur = que.popleft()
                if i == leng - 1:
                    res.append(cur.val)
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
        return res

```

637.二叉树的层平均值


```

class Solution:
    def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:
        res, que = [], deque()
        if root:
            que.append(root)
        while que:
            tempsum, leng = 0, len(que)
            for i in range(len(que)):
                cur = que.popleft()
                tempsum += cur.val
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
            res.append(tempsum / leng)

```

429.N 叉树的层序遍历

```

class Solution:
    def levelOrder(self, root: 'Node') -> List[List[int]]:
        res, que = [], deque()
        if root:
            que.append(root)
        while que:
            temp = []
            for i in range(len(que)):
                cur = que.popleft()
                temp.append(cur.val)
                for child in cur.children:
                    que.append(child)
            res.append(temp)
        return res

```

515.在每个树行中找最大值

```

class Solution:
    def largestValues(self, root: Optional[TreeNode]) -> List[int]:
        res, que = [], deque()
        if root:
            que.append(root)
        while que:
            max_value = -float("inf")
            for i in range(len(que)):
                cur = que.popleft()
                max_value = max(cur.val, max_value)
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
            res.append(max_value)
        return res

```

116.填充每个节点的下一个右侧节点指针

```

class Solution:
    def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':
        que = collections.deque()
        if root:
            que.append(root)
        while que:
            pre = None
            for i in range(len(que)):
                cur = que.popleft()

                if not pre:
                    pre = cur
                else:
                    pre.next = cur
                    pre = cur

                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)

            return root

```

117.填充每个节点的下一个右侧节点指针 II

```

class Solution:
    def connect(self, root: 'Node') -> 'Node':
        if not root:
            return
        que = collections.deque([root])
        while que:
            pre = None
            for i in range(len(que)):
                cur = que.popleft()
                if pre:
                    pre.next = cur
                pre = cur

                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
            return root

```

104.二叉树的最大深度

```

class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0
        height, que = 0, collections.deque([root])
        while que:
            height += 1
            for i in range(len(que)):
                cur = que.popleft()
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
            return height

```

111.二叉树的最小深度

[illegible]

10.11.2024

226.翻转二叉树

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

一开始在写的时候忘记+self 了一直报错，裂开。

单纯翻转不涉及其他内容，所以迭代法、层序法都可以

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

```

class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        stack = []
        pre = None
        node = root
        while node or stack:
            if node:
                stack.append(node)
                node = node.left
            else:
                cur = stack[-1]
                if not cur.right or cur.right == pre:
                    cur = stack.pop()
                    pre = cur
                    node = None
                    cur.left, cur.right = cur.right, cur.left
                else:
                    node = cur.right
        return root

```

101. 对称二叉树

是比较两个子树是否相同，而不是比较每一棵树的左右节点是否相同。

```

class Solution:

    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True
        return self.method(root.left, root.right)
    def method(self, left, right):
        if not left and not right:
            return True
        elif (left and not right) or (not left and right):
            return False
        elif left.val != right.val:
            return False
        outside = self.method(left.left, right.right)
        inside = self.method(left.right, right.left)
        return outside and inside

```

104. 二叉树的最大深度

昨天写过用层序遍历求最大深度的方法。

今天使用的是后续递归算法。

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

前序递归

```
class Solution:
    def __init__(self):
        self.result = 0

    def getdepth(self, root, depth):
        self.result = max(self.result, depth)
        if root.left:
            self.getdepth(root.left, depth + 1)
        if root.right:
            self.getdepth(root.right, depth + 1)
        return

    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0
        self.getdepth(root, 1)
        return self.result
```

111. 二叉树的最小深度

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。注意是叶子节点。注意!!!

```
class Solution:
    def getdepth(self, root):
        if not root:
            return 0

        leftheight = self.getdepth(root.left)
        rightheight = self.getdepth(root.right)
        if root.left and not root.right:
            return 1 + leftheight
        if not root.left and root.right:
            return 1 + rightheight

        return 1 + min(leftheight, rightheight)

    def minDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0
        return self.getdepth(root)
```


10.14.2024

110.平衡二叉树

平衡肯定要用到高度的比较，用后序遍历

```
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True
        tag = self.height(root)
        return True if tag != -1 else False

    def height(self, node):
        if not node:
            return 0

        leftheight = self.height(node.left)
        rightheight = self.height(node.right)

        if leftheight == -1 or rightheight == -1 or abs(leftheight - rightheight) > 1:
            return -1

        else:
            return 1 + max(leftheight, rightheight)
```

为什么要用两个函数呢？因为在求平衡树的过程中返回的是 int，而题目最终要判断的是 BOOL

257. 二叉树的所有路径

很显然需要用到前序遍历，回溯算法

```
class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        res = []
        def singleTreePath(root, temppath: List[int]):
            if not root:
                return

            temppath.append(root.val)
            if not root.left and not root.right:
                res.append("->".join(str(x) for x in temppath))
            else:
                singleTreePath(root.left, temppath)
                singleTreePath(root.right, temppath)

            temppath.pop()

        temppath = []
        singleTreePath(root, temppath)
        return res
```

注意，在回溯过程中，在判断逻辑中写出返回条件，而不要在段中写，会造成逻辑上的混淆。

例如：

```
def singleTreePath(root, temppath: List[int]):  
    if not root:  
        return  
  
    temppath.append(root.val)  
    if not root.left and not root.right:  
        res.append("->".join(str(x) for x in temppath))  
        return  
    else:  
        singleTreePath(root.left, temppath)  
        singleTreePath(root.right, temppath)  
  
    temppath.pop()
```

此处的 return 导致少执行了 pop 操作，会造成元素的重复。

Output

```
["1->2->5", "1->2->3"]
```

Expected

```
["1->2->5", "1->3"]
```

```

class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:
        res = []
        def singleTreePath(root, temppath:str):
            if not root:
                return

            temppath += str(root.val)
            if not root.left and not root.right:
                res.append(temppath)
            else:
                singleTreePath(root.left, temppath + "->")
                singleTreePath(root.right, temppath + "->")

        temppath = ""
        singleTreePath(root, temppath)
        return res

```

在 python 中，字符串是不可修改的变量。所以每次递归的过程中，都会有一个新的字符串生成，省去了 pop 的过程。

404.左叶子之和

之前写过迭代方法，本次用递归

```

class Solution:
    def sumOfLeftLeaves(self, root: Optional[TreeNode]) -> int:
        res = 0

        def traverseTree(root):
            nonlocal res
            if not root:
                return

            if root.left and not root.left.left and not root.left.right:
                res += root.left.val
            traverseTree(root.left)
            traverseTree(root.right)

        traverseTree(root)
        return res

```

整数也是不可变类型，所以需要生命 nonlocal

222.完全二叉树的节点个数

之前写过层序遍历方法，本题还可以使用前序中序后序，这三种方法的迭代和递归均可。因为不涉及到具体逻辑的判定，只需要计算节点个数。

先用后序迭代写一个

```
class Solution:
    def countNodes(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0
        stack = []
        count = 0
        pre, node = None, root
        while stack or node:
            if node:
                stack.append(node)
                node = node.left
            else:
                cur = stack[-1]
                if not cur.right or cur.right == pre:
                    pre = cur
                    cur = stack.pop()
                    count += 1
                else:
                    node = cur.right
        return count
```

前序递归

```
def countNodes(self, root: Optional[TreeNode]) -> int:
    if not root:
        return 0

    leftcount = self.countNodes(root.left)
    rightcount = self.countNodes(root.right)
    return 1 + leftcount + rightcount
```

在**完全二叉树**中，如果递归向左遍历的深度等于递归向右遍历的深度，那说明就是满二叉树。

```
class Solution:
    def countNodes(self, root: TreeNode) -> int:
        if not root:
            return 0

        left, right = root.left, root.right
        leftdepth, rightdepth = 0, 0
        while left:
            leftdepth += 1
            left = left.left
        while right:
            rightdepth += 1
            right = right.right
        if leftdepth == rightdepth:
            return (2 << leftdepth) - 1
        return 1 + self.countNodes(root.left) + self.countNodes(root.right)
```

在完全二叉树中，树可以由很多满二叉树组成。所以该任务就是将所有满二叉树加起来。

10.15.2024

513 找树左下角的值

```
class Solution:
    def findBottomLeftValue(self, root: Optional[TreeNode]) -> int:
        self.temp_depth = float("-inf")
        self.res = 0
        def traversal(root, depth):
            if not root.left and not root.right:
                if depth > self.temp_depth:
                    self.temp_depth = depth
                    self.res = root.val
            if root.left:
                traversal(root.left, depth + 1)
            if root.right:
                traversal(root.right, depth + 1)
            return
        traversal(root, 0)
        return self.res
```

在遍历的过程中，只要深度相同，那么保存的只会是最左侧的值。

112.路径总和

```
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        if not root:
            return False
        self.targetSum = targetSum
        def traversal(root, temp):
            if temp == self.targetSum and not root.left and not root.right:
                return True
            if root.left:
                if traversal(root.left, temp + root.left.val):
                    return True
            if root.right:
                if traversal(root.right, temp + root.right.val):
                    return True
            return False
        return traversal(root, root.val)
```

113.路径总和 ii

```
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:
        if not root:
            return []
        self.res = []

        def travelsum(root, tempsum, templs):
            if not root.left and not root.right and tempsum == targetSum:
                self.res.append(list(templs))
            return

        if root.left:
            templs.append(root.left.val)
            travelsum(root.left, tempsum + root.left.val, templs)
            templs.pop()

        if root.right:
            templs.append(root.right.val)
            travelsum(root.right, tempsum + root.right.val, templs)
            templs.pop()

        travelsum(root, root.val, [root.val])
        return self.res
```

与上题思路相同，代码相似。但是要注意的是我们在参数中直接传入了列表，而列表是可变对象，在之后的 pop 中会导致 self.res 中的答案改变。所以要注意在最后添加进入的时候要 list()重新指定一个单独的列表。

105.从前序与中序遍历序列构造二叉树

用 python 简介很多，依稀记得当时用 C++ 的痛。Python 的列表切片功能太强大了。

```

class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if len(inorder) == 0:
            return None

        root_value = preorder[0]
        root = TreeNode(root_value)

        split_node = inorder.index(root_value)
        leftinorder, rightinorder = inorder[:split_node], inorder[split_node+1:]
        leftpre, rightpre = preorder[1:1+len(leftinorder)], preorder[1+len(leftinorder):]

        root.left = self.buildTree(leftpre, leftinorder)
        root.right = self.buildTree(rightpre, rightinorder)
        return root

```

106.从中序与后序遍历序列构造二叉树

```

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

        root_node = postorder[-1]
        root = TreeNode(root_node)

        split_node = inorder.index(root_node)
        inleft, inright = inorder[:split_node], inorder[split_node+1:]
        postleft, postright = postorder[:len(inleft)], postorder[len(inleft):-1]

        root.left = self.buildTree(inleft, postleft)
        root.right = self.buildTree(inright, postright)
        return root

```