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。

```python
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

```python
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(nlogn)

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)

```python
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。

```python
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

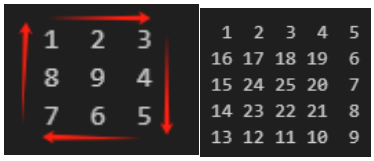
⑤ 返回答案

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 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：

```python
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

```python
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 crtl+Z 来完成输入

44. 开发商购买土地

题目需要利用到前缀和

首先需要求出列、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

```python
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

```python
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

```python
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

```python
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

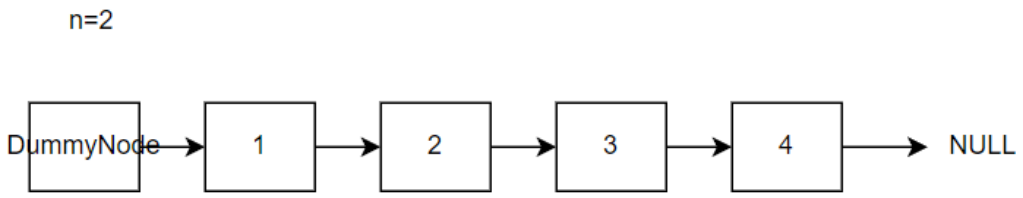
```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummyNode = ListNode(0, head)
        cur = dummyNode
        while cur.next and cur.next.next:
            temp_one = cur.next
            temp_third = cur.next.next.next

            cur.next = temp_one.next
            cur.next.next = temp_one
            temp_one.next = temp_third

            cur = cur.next.next

        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 dummynode 要先走 n+1 步(因为要找到的是待删除节点的前一个节点)

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        dummyNode = ListNode(0, head)
        slow = fast = dummyNode
        for i in range(n + 1):
            fast = fast.next

        while fast:
            fast = fast.next
            slow = slow.next

        slow.next = slow.next.next
        return dummyNode.next
```

双指针法，时间复杂度 O(n)

160.链表相交

```python
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)

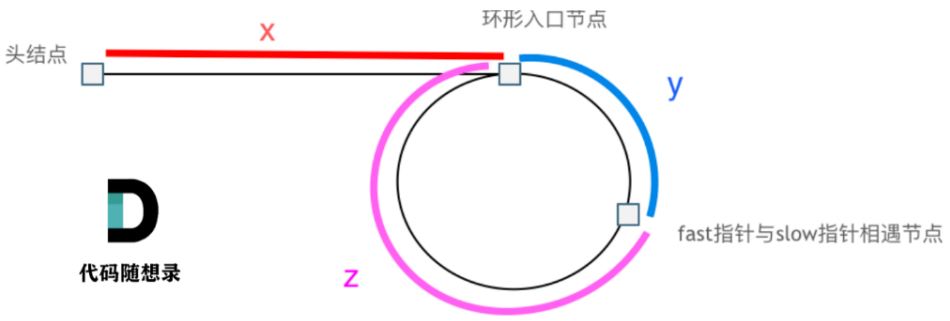
142.环形链表 II

```python
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： `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 环形入口节点的的距离。

所以要求x ，将x单独放在左面： `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(logn)	O(logn)
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(logn)	O(logn)
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

```python
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

```python
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。

```python
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

```python
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(nlogn)

```
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

```python
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

```python
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。

```python
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

```python
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：

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

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：

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

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

```python
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

```python
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

```python
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。

```python
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)
```

```python
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。

```python
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

```python
def main():
    count = int(input())
    str_ = input()
    str_ = str_[::-1]
    res = str_[:count][::-1] + str_[count:][::-1]
    print(res)

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

```python
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

```python
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

```python
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

```python
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

```python
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:

```python
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:

```python
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？

```python
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

```python
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
```

```python
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

```python
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

```python
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

```
#         self.right = right
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

```python
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

```python
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

```python
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

```python
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

```python
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

```python
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

```python
class Solution:
    def minDepth(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 not cur.left and not cur.right:
                    return height
                if cur.left:
                    que.append(cur.left)
                if cur.right:
                    que.append(cur.right)
        return
```

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
```

```python
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。

```python
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!!!

```python
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

```python
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

```python
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：

```python
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"]
```

```python
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

```python
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。

```python
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

```python
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
                return

            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

```python
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。

```python
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

```python
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
```

10.16.2024

654.最大二叉树

```python
class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> Optional[TreeNode]:
        if len(nums) == 0:
            return None

        node_index = nums.index(max(nums))
        node = TreeNode(nums[node_index])
        node.left = self.constructMaximumBinaryTree(nums[:node_index])
        node.right = self.constructMaximumBinaryTree(nums[node_index + 1:])
        return node
```

本题目思路是非常明确的。同样可以使用下表法

```python
class Solution:
    def constructMaximumBinaryTree(self, nums: List[int]) -> Optional[TreeNode]:
        def travelsal(nums, leftindex, rightindex):
            if leftindex >= rightindex:
                return

            max_index = nums.index(max(nums[leftindex:rightindex]))
            node = TreeNode(nums[max_index])
            node.left = travelsal(nums, leftindex, max_index)
            node.right = travelsal(nums, max_index+1, rightindex)
            return node
        leftindex, rightindex = 0, len(nums)
        return travelsal(nums, leftindex, rightindex)
```

在上面这个写法中，需要注意 leftindex 和 rightindex 的判定。因为本题目中使用了左闭右开的区间，所以 leftindex==rightindex 的时候意味着遍历的结束。同时在遍历时，注意 leftindex 是能取到的左边界，而 rightindex 不能取到。

617.合并二叉树

使用前序递归，主要问题在于条件的判定。

```python
class Solution:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> Optional[TreeNode]:
        if root1 and not root2:
            return root1
        elif not root1 and root2:
            return root2
        elif not root1 and not root2:
            return None

        root1.val += root2.val
        root1.left = self.mergeTrees(root1.left, root2.left)
        root1.right = self.mergeTrees(root1.right, root2.right)
        return root1
```

终止条件就是两点的存在与否

700.二叉搜索树中的搜索

前序递归

```python
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if not root:
            return None

        if val == root.val:
            return root
        elif val > root.val:
            return self.searchBST(root.right, val)
        else:
            return self.searchBST(root.left, val)
```

层序遍历法

```python
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if not root:
            return None
        que = deque([root])
        while que:
            top = que.popleft()
            if top.val == val:
                return top
            elif val > top.val:
                if top.right:
                    que.append(top.right)
            else:
                if top.left:
                    que.append(top.left)
        return None
```

```python
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        while root:
            if val == root.val:
                return root
            elif val > root.val:
                root = root.right
            else:
                root = root.left
```

98.验证二叉搜索树

题目也说了验证，那我们需要从中序遍历入手，确保每一棵子树都是顺序的。

最好理解的方法是将中序遍历结果入队，顺序比较，空间复杂度 O(n)。

也可以递归时候直接比较

最初我用后续遍历写出了这个代码

```python
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True

        lefttag = self.isValidBST(root.left)
        righttag = self.isValidBST(root.right)
        if (not root.left or root.left.val < root.val) and (not root.right or root.right.val > root.val):
            return True
        else:
            return False
        return lefttag and righttag
```

但实际上这样并不能确保中间的节点一定大于左子树所有节点或者小于右子树所有节点，只能确保每一个有三个节点的子树是有序的。

还是得用中序遍历（递归）：

```python
class Solution:
    def __init__(self):
        self.minvalue = float("-inf")

    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True

        lefttag = self.isValidBST(root.left)
        if root.val > self.minvalue:
            self.minvalue = root.val
        else:
            return False
        righttag = self.isValidBST(root.right)

        return lefttag and righttag
```

中序遍历（迭代）：

```python
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        stack = []
        pre = None
        while root or stack:
            if root:
                stack.append(root)
                root = root.left
            else:
                top = stack.pop()
                if not pre or pre.val < top.val:
                    pre = top
                    root = top.right
                else:
                    return False
        return True
```

10.17.2024

530.二叉搜索树的最小绝对差

通过中序遍历得到一个有序数组，求相邻元素差的绝对值最小值

```
class Solution:
    def __init__(self):
        self.res = float("inf")
        self.pre = None
    def getMinimumDifference(self, root: Optional[TreeNode]) -> int:
        if not root:
            return

        self.getMinimumDifference(root.left)
        if self.pre:
            self.res = min(self.res, abs(root.val - self.pre.val))
        self.pre = root
        self.getMinimumDifference(root.right)

        return self.res
```

因为中序遍历中，数组是有序的，所以可以直接改变 pre 而不用考虑回溯的问题。

```
class Solution:
    def getMinimumDifference(self, root: Optional[TreeNode]) -> int:
        stack = []
        pre = None
        res = float("inf")
        while root or stack:
            if root:
                stack.append(root)
                root = root.left
            else:
                top = stack.pop()
                if pre:
                    res = min(res, abs(pre.val - top.val))
                pre = top
                root = top.right
```

501.二叉搜索树中的众数

一般来说既然是求众数，那我们可以用消除法来做，最后保留的数就是众数。

此时的树是二叉搜索树，那我们可以用 pre 指针来表示当前的值，值相同，count++。反之重新计数

```python
class Solution:
    def __init__(self):
        self.res = []
        self.pre = None
        self.count = float("-inf")
        self.tempcount = 0
    def findMode(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return

        self.findMode(root.left)

        if not self.pre:
            self.tempcount = 1

        elif self.pre.val == root.val:
            self.tempcount += 1

        elif self.pre.val != root.val:
            self.tempcount = 1

        self.pre = root
        if self.tempcount > self.count:
            self.res = [root.val]
            self.count = self.tempcount
        elif self.tempcount == self.count:
            self.res.append(root.val)

        self.findMode(root.right)
        return self.res
```

236. 二叉树的最近公共祖先

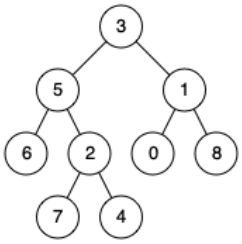
这个看到题目确实没什么思路，只知道肯定需要用到后序。

看了 carl 的思路后，才恍然大悟可以用返回值来标志该点的孩子(包括自己)是否有点 p 或者 q，如果两边子树分别有这两个节点，那么则是祖先节点。

```python
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if root == p or root == q or not root:
            return root
        print(root.val)
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        if left and right:
            return root

        elif left and not right:
            return left

        elif not left and right:
            return right

        else:
            return None
```

我们以此样本为例子

Example 2:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself
according to the LCA definition.
```

输出是

Stdout

```
3
1
0
8
```

可以看到我们的输出是根节点，但是没有输出 5，说明在后序的过程中，因为发现了 5 是 p，所以直接返回。这也是一开始我误解的地方。

实际上，因为我们要求的是公共祖先节点，而这个点的出现本身就意味着公共祖先节点只能是它或者它上面的节点。因此我们甚至不用判断另一个节点到底

是否存在，如果在别的树中发现，那说明他们的祖先节点仍然在上面。反之，说明在没有遍历的树内（因为已经确保了有解）

10.18.2024

235. 二叉搜索树的最近公共祖先

因为二叉搜索树是有序树，所以当第一个节点如果小于 q，大于 p，那他就是祖先节点。

```python
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if not root:
            return

        if root.val < p and root.val < q:
            left  = self.lowestCommonAncestor(root.right, p, q)
            if left:
                return left

        elif root.val > p and root.val > q:
            right = self.lowestCommonAncestor(root.left, p, q)
            if right:
                return right

        return root
```

701.二叉搜索树中的插入操作

本题目最简单的方式肯定是将插入节点都放到叶子节点。那么凡是涉及到叶子节点的问题并且没有返回值的时候，过程中肯定要同时判断他们的父节点与孩子节点，这样才能有明确的指向。

```python
class Solution:
    def insertIntoBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if not root:
            return TreeNode(val)
        def traversal(root):
            if root.val > val and not root.left:
                root.left = TreeNode(val)
                return
            elif root.val < val and not root.right:
                root.right = TreeNode(val)
                return

            if root.val > val:
                traversal(root.left)
            elif root.val < val:
                traversal(root.right)

        traversal(root)
        return root
```

那么有返回值应该是什么样子呢?

```python
class Solution:
    def insertIntoBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if not root:
            return TreeNode(val)

        if root.val > val:
            root.left = self.insertIntoBST(root.left, val)
        elif root.val < val:
            root.right = self.insertIntoBST(root.right, val)

        return root
```

可以说是更简单了

450.删除二叉搜索树中的节点

在删除之后，我们既可以用左子树代替，也可以用右子树代替，按照题目说法统一用右子树替代。那么就涉及到了树结构变化的问题，其中最难的部分是左右子树都存在时，像保证二叉搜索树的性质，一定要将待删除节点的左子树插入到右子树的最左孩子上

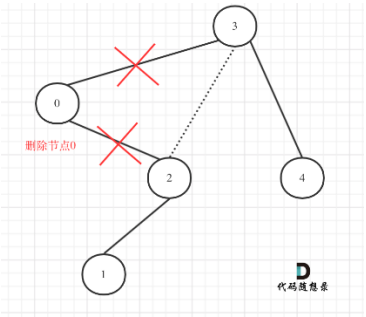
```python
class Solution:
    def deleteNode(self, root: Optional[TreeNode], key: int) -> Optional[TreeNode]:
        if not root:
            return None
        if root.val == key:
            if not root.left and not root.right:
                return None
            elif root.left and not root.right:
                return root.left
            elif not root.left and root.right:
                return root.right
            elif root.left and root.right:
                temp = root.right
                while temp.left:
                    temp = temp.left
                temp.left = root.left
                return root.right

        elif key > root.val:
            root.right = self.deleteNode(root.right, key)
        elif key < root.val:
            root.left = self.deleteNode(root.left, key)

        return root
```

10.18.2024

669. 修剪二叉搜索树

```
class Solution:
    def trimBST(self, root: Optional[TreeNode], low: int, high: int) -> Optional[TreeNode]:
        if not root:
            return None
        print(root.val)
        if root.val > high:
            left = self.trimBST(root.left, low, high)
            return left
        elif root.val < low:
            right = self.trimBST(root.right, low, high)
            return right
        root.left = self.trimBST(root.left, low, high)
        root.right = self.trimBST(root.right, low, high)
        return root
```

最开始我想到的解决思路是：对不满足条件的节点进行删除，用昨天完成的删除节点代码。但是这样做相比起直接链接起来更加麻烦



迭代法

```
class Solution:
    def trimBST(self, root: Optional[TreeNode], low: int, high: int) -> Optional[TreeNode]:
        if not root:
            return None

        while root and (root.val < low or root.val > high):
            if root.val < low:
                root = root.right
            else:
                root = root.left

        cur = root

        while cur:
            while cur.left and cur.left.val < low:
                cur.left = cur.left.right
            cur = cur.left
        cur = root
        while cur:
            while cur.right and cur.right.val > high:
                cur.right = cur.right.left
            cur = cur.right
        return root
```

108.将有序数组转换为二叉搜索树

```python
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        if not nums:
            return None

        split = len(nums) // 2
        root = TreeNode(nums[split])
        root.left = self.sortedArrayToBST(nums[:split])
        root.right = self.sortedArrayToBST(nums[split + 1:])
        return root
```

得益于强大的数组切片，我们并不需要做出很多边界条件的判定。

538.把二叉搜索树转换为累加树

右中左，这道题还是相当简单的。利用中序遍历和前序指针，不断累加。

```python
class Solution:
    def __init__(self):
        self.pre = None

    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return

        root.right = self.convertBST(root.right)
        if self.pre:
            root.val += self.pre.val
        self.pre = root
        root.left = self.convertBST(root.left)

        return root
```

迭代法

```python
def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
    if not root:
        return

    stack = []
    pre = None
    top = root
    while top or stack:
        if top:
            stack.append(top)
            top = top.right
        else:
            top = stack.pop()
            if pre:
                top.val += pre.val
            pre = top
            top = top.left
    return root
```

10.21.2024

回溯算法:

在回溯过程中，我们需要的可能是找到一个可能的结果，或者遍历所有情况，在这个过程中，我们将符合条件的结果保存下来。所以一般用 void 来进行遍历更为合适。因为回溯的时间复杂度很高，所以我们可以在此基础上进行优化——剪枝。

77. Combinations

第一道回溯题目比较简单，需要注意的是在向 res 保存 path 的时候需要单独存 path 的副本，这样之后 path 的改动不会影响 res 中的值。

```
class Solution:
    def __init__(self):
        self.res = []

    def combine(self, n: int, k: int) -> List[List[int]]:
        def travel(index, k, n, path):
            if len(path) == k:
                self.res.append(path[:])
                return
            for i in range(index, n+1):
                path.append(i)
                print(path)
                travel(i + 1, k, n, path)
                path.pop()
        travel(1, k, n, [])
        return self.res
```

只要提到回溯，那优化的时候必须考虑到剪枝的问题。

本题目中剪枝需要考虑的是

```
for i in range(index, n - (k - len(path)) + 2):
```

这是看了 carl 的想法才写出来的。当剩余的数不能满足长度 k 的数列时， 就没

必要继续遍历了。

216. Combination Sum III

和上题思路一致

```
class Solution:
    def __init__(self):
        self.res = []
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        def travel(k, n, path, tempcount, startindex):
            if len(path) == k and tempcount == n:
                self.res.append(path[:])
                return

            for i in range(startindex, 10):
                path.append(i)
                travel(k, n, path, tempcount + i, i + 1)
                path.pop()

        travel(k, n, [], 0, 1)
        return self.res
```

本题目中剪枝有两个方向

一个是 res 中列表长度，另一个是当数列和超过目标

```
class Solution:
    def __init__(self):
        self.res = []
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        def travel(k, n, path, tempcount, startindex):
            if len(path) == k and tempcount == n:
                self.res.append(path[:])
                return
            if tempcount > n:
                return

            for i in range(startindex, 9 - (k - len(path)) + 2):
                path.append(i)
                travel(k, n, path, tempcount + i, i + 1)
                path.pop()

        travel(k, n, [], 0, 1)
        return self.res
```

17.电话号码的字母组合

在本题中，并没有显示的回溯过程，例如，append,pop。因为 tempstr 是字符串，字符串不可更改，所以每次相当于一个新的变量进行传递，而旧的变量保持不变，因此不用在此基础上进行变化。

```
class Solution:
    def __init__(self):
        self.res = []
        self.dic = {'0':'', '1':'','2':'abc', '3':'def', '4':'ghi', '5':'jkl', '6':'mno', '7':'pqrs', '8':'tuv',
'9':'wxyz'}
    def letterCombinations(self, digits: str) -> List[str]:
        if len(digits) == 0:
            return []
        def travel(digits, index, tempstr):
            if index == len(digits):
                self.res.append(tempstr)
                return

            chars = self.dic[digits[index]]
            for c in chars:
                temps = tempstr + c
                travel(digits, index + 1, temps)

        travel(digits, 0, "")
        return self.res
```

用最简单的写法写时间复杂度最高的算法，这就是回溯。

10.22.2024

39. 组合总和

既然所有数都能重复选取，那么在终止条件上，不再是长度，而是当和大于 target 的时候返回。

```
class Solution:
    def __init__(self):
        self.res = []

    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def travel(candidates, startindex, tempres, tempsum, target):
            if tempsum > target:
                return
            elif tempsum == target:
                self.res.append(tempres[:])
                return

            for i in range(startindex, len(candidates)):
                tempres.append(candidates[i])
                travel(candidates, i, tempres, tempsum + candidates[i], target)
                tempres.pop()

        travel(candidates, 0, [], 0, target)
        return self.res
```

个人认为本题目中剪枝的影响没有那个大，在进行下一次回溯前进行剪枝

```
class Solution:
    def __init__(self):
        self.res = []

    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def travel(candidates, startindex, tempres, tempsum):
            if tempsum == 0:
                self.res.append(tempres[:])
                return

            for i in range(startindex, len(candidates)):
                if tempsum - candidates[i] < 0:
                    break
                tempres.append(candidates[i])
                travel(candidates, i, tempres, tempsum - candidates[i])
                tempres.pop()

        travel(candidates, 0, [], target)
        return self.res
```

在正常思考的过程中，潜意识往往相加向 target 靠拢，但实际上用减法会加快运行速度和节省空间。

40.组合总和 II

在用 set 去重的过程中，会超时。所以在回溯的过程中，就要处理。

因为每个元素只能使用一次，所以以 startindex 和当前 index 为比较保证此条件满足。比如[1,1,2]， 当 target 是 4 是的时候，第一个 res 是[1,1,2]，此时每个 1 已经被使用过。所以当回溯到第二个 1 的时候，循环直接跳过。

```python
class Solution:
    def __init__(self):
        self.res = []

    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        def travel(candidates, target, tempres, tempsum, startindex):
            if tempsum == target:
                self.res.append(tempres[:])
                return

            for i in range(startindex, len(candidates)):
                if i > startindex and candidates[i-1] == candidates[i]:
                    continue

                if tempsum + candidates[i] > target:
                    return
                tempres.append(candidates[i])
                travel(candidates, target, tempres, tempsum + candidates[i], i + 1)
                tempres.pop()

        candidates = sorted(candidates)
        travel(candidates, target, [], 0, 0)
        return self.res
```

131.分割回文串

首先需要一个函数来判断字符串是否是回文

分割和组合问题比较像。组合问题是不断向[]中添加 val，而分割问题是直接对字符串进行切片

```python
class Solution:
    def __init__(self):
        self.res = []

    def isparlin(self, s:str):
        i, j = 0, len(s) - 1
        while i < j:
            if s[i] != s[j]:
                return False
            i += 1
            j -= 1
        return True

    def partition(self, s: str) -> List[List[str]]:
        def travel(s, startindex, tempres):
            if startindex == len(s):
                self.res.append(tempres[:])
                return

            for i in range(startindex + 1, len(s)):
                if self.isparlin(s[startindex : i]):
                    tempres.append(s[startindex : i])
                    travel(s, i, tempres)
                    tempres.pop()
        travel(s, 0, [])
        return self.res
```

目前来看，回溯方法比较简单。首先需要确定推出循环的条件，其次通过 for 循环进行每一种可能性的回溯。剪枝方面，在不要求顺序的情况下，我们可以先对列表、字符串进行排序，从而在不满足条件下提前 break 循环，节省资源。

10.23.2024

93.复原 IP 地址

本质上也是对字符串的切分，但是多了不少判断条件

若切分不在 0 与 255 之内的区间，或者从 0 开始但是长度不为 1，那就说明不符合条件。

在终止条件上，不光需要保证只有 4 段字符串，还需要保证这个 s 字符串刚好用完。

```
class Solution:
    def __init__(self):
        self.res = []

    def is_valid(self, s, startindex, endindex):
        if endindex - startindex > 3:
            return False
        if s[startindex] == '0' and endindex > startindex:
            return False
        if int(s[startindex : endindex + 1]) > 255:
            return False
        return True

    def restoreIpAddresses(self, s: str) -> List[str]:
        def travel(s, startindex, path):
            if len(path) == 4 and startindex == len(s):
                self.res.append(".".join(path))
                return
            if len(path) > 4:
                return

            for i in range(startindex, len(s)):
                if self.is_valid(s, startindex, i):
                    path.append(s[startindex : i + 1])
                    travel(s, i + 1, path)
                    path.pop()
        travel(s, 0, [])
        return self.res
```

78.子集

在子集问题中，需要遍历整棵树，所以不需要终止条件，直接+就完事了。

```python
class Solution:
    def __init__(self):
        self.res = []

    def subsets(self, nums: List[int]) -> List[List[int]]:
        def travel(nums, tempres, startindex):
            self.res.append(tempres[:])

            for i in range(startindex, len(nums)):
                tempres.append(nums[i])
                travel(nums, tempres, i + 1)
                tempres.pop()

        travel(nums, [], 0)
        return self.res
```

90.子集 II

本题是 40 题，组合 II 和上题目的集合，不允许有重复的子集。

可以看到，需要对当前数和之前数值进行比较，来判断在之前的回溯中，结果中是否已经有相对应的答案了。这里和 40 题中都没有使用卡哥的 used 列表方法，因为我认为这种思路更符合我的理解。

```python
class Solution:
    def __init__(self):
        self.res = []

    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        def travel(nums, startindex, tempres):
            self.res.append(tempres[:])

            for i in range(startindex, len(nums)):
                if i > startindex and nums[i] == nums[i - 1]:
                    continue
                tempres.append(nums[i])
                travel(nums, i + 1, tempres)
                tempres.pop()
        nums = sorted(nums)
        travel(nums, 0, [])
        return self.res
```

10.24.2024

491.递增子序列

本题难点在于如何在非有序的数列中避免重复，如[4, 7, 6, 7]。

所以需要额外的空间来使用 set 对列表中已经走过的元素进行记录。

并且因为要遍历整棵树，所以不需要返回节点。

```python
class Solution:
    def __init__(self):
        self.res = []

    def findSubsequences(self, nums: List[int]) -> List[List[int]]:
        def travel(nums, startindex, tempres):
            if len(tempres) >= 2:
                self.res.append(tempres[:])

            tag = set()
            for i in range(startindex, len(nums)):
                if (tempres and nums[i] < tempres[-1]) or nums[i]  in tag:
                    continue
                tag.add(nums[i])
                tempres.append(nums[i])
                travel(nums, i + 1, tempres)
                tempres.pop()
        travel(nums, 0, [])
        return self.res
```

46.全排列

全排列中没有 index 的限制，需要遍历到每个元素但无所谓顺序。

所以需要全局有一个 tag 来进行标记，同样这也是回溯的一部分。

上个题目中是需要在当前层对状态分别进行记录

```
class Solution:
    def __init__(self):
        self.res = []

    def permute(self, nums: List[int]) -> List[List[int]]:
        def travel(nums, tag, tempres):
            if len(tempres) == len(nums):
                self.res.append(tempres[:])
                return

            for i in range(0, len(nums)):
                if tag[i]:
                    continue
                tag[i] = True
                tempres.append(nums[i])
                travel(nums, tag, tempres)
                tempres.pop()
                tag[i] = False

        tag = [False] * len(nums)
        travel(nums, tag, [])
        return self.res
```

47.全排列 II

数字重复，结果不重复，并且任意顺序。这说明我们的第一步需要先排序。因为要全排列，所以我们同样需要 tag 数组来标记哪些数字已经用了那些没有用。其次，因为不需要重复的，所以在排序之后我们需要比较相邻两个数之间是否值一样。

```python
class Solution:
    def __init__(self):
        self.res = []

    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        def travel(nums, tag, tempres):
            if len(tempres) == len(nums):
                self.res.append(tempres[:])
                return

            for i in range(0, len(nums)):
                if (i > 0 and nums[i] == nums[i-1] and not tag[i - 1]) or tag[i]:
                    continue
                tag[i] = True
                tempres.append(nums[i])
                travel(nums, tag, tempres)
                tempres.pop()
                tag[i] = False


        nums = sorted(nums)
        tag = [False] * len(nums)
        travel(nums, tag, [])
        return self.res
```

这里一定要注意：实际上在回溯之后前点是没有再用过的， 所以是判断前点，而不是此点。Or 前面的判断条件是为了去重， or 后面的判断条件是为了遍历每个回溯过程中所有节点。

332 重新安排行程

下次一定

51 N 皇后

感觉 N 皇后的难度主要集中在判断皇后位置是否符合条件而不在于回溯本身(毕竟回溯是个笨办法)，判断的时候只要判断之前的行就可以了。

Tips：在对行进行修改的时候，不能直接修改字符串的值，又忘了这个事了，还是的切分对列表进行修改

```python
class Solution:
    def __init__(self):
        self.res = []

    def validposition(self, matrix, row, col):
        for i in range(row):
            if matrix[i][col] == 'Q':
                return False

        pre_row, pre_col = row - 1, col + 1
        while pre_col < len(matrix) and pre_row >= 0:
            if matrix[pre_row][pre_col] == 'Q':
                return False
            pre_row -= 1
            pre_col += 1

        pre_row, pre_col = row - 1, col - 1
        while pre_row >= 0 and pre_col >= 0:
            if matrix[pre_row][pre_col] == 'Q':
                return False
            pre_row -= 1
            pre_col -= 1
        return True

    def solveNQueens(self, n: int) -> List[List[str]]:
        matrix = [("." * n) for _ in range(n)]

        def travel(matrix, row, n):
            if row == n:
                self.res.append(matrix.copy())
                return

            for col in range(n):
                if self.validposition(matrix, row, col):
                    matrix[row] = matrix[row][:col] + 'Q' + matrix[row][col + 1:]
                    travel(matrix, row + 1, n)
                    matrix[row] = matrix[row][:col] + '.' + matrix[row][col + 1:]

        travel(matrix, 0, n)
        return self.res
```

37 解数独

下次一定

10.25.2024

贪心算法：每一次的局部最优，不断推导出全局最优

455.分发饼干

```python
class Solution:
    def findContentChildren(self, g: List[int], s: List[int]) -> int:
        g, s = sorted(g), sorted(s)
        startindex, i = 0, 0
        res = 0
        while startindex < len(g) and i < len(s):
            if g[startindex] <= s[i]:
                startindex += 1
                res += 1
            i += 1
        return res
```

376. 摆动序列

找到峰值，峰顶和峰谷

```python
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        if len(nums) <= 1:
            return len(nums)

        pre, cur, res = 0, 0, 1
        for i in range(len(nums) - 1):
            cur = nums[i + 1] - nums[i]
            if cur * pre <= 0 and cur != 0:
                res += 1
                pre = cur
        return res
```

首先的问题是，最右侧的点自动当作峰值，所以 res 要从 1 开始。

其次再判断的时候 cur 不能等于 0， 这说明还没有波动。

53. 最大子序和

也是经常能在面经上看到的题，手撕

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        res, temp = float("-inf"), 0
        for i in range(len(nums)):
            temp += nums[i]
            res = max(res, temp)
            if temp < 0:
                temp = 0
        return res
```

动态规划， 我认为这题动态规划可能更好理解一些。

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        dp = [0] * len(nums)
        dp[0] = nums[0]
        res = dp[0]

        for i in range(1, len(nums)):
            dp[i] = max(dp[i-1] + nums[i], nums[i])
            res = max(res, dp[i])
        return res
```

10.26.2024

这些题目之前做过了，就少写些注释了嘿嘿

122.买卖股票的最佳时机 II

怎么赚最大的钱？每天赚一点点喽。赔钱就不卖喽

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

55. 跳跃游戏

找到目前为止跳跃可达的边界。

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        reach_dis = 0
        for i in range(len(nums)):
            if i > reach_dis:
                return False
            reach_dis = max(reach_dis, nums[i] + i)
            if reach_dis >= len(nums) - 1:
                return True
```

45.跳跃游戏 II

虽然是第三次写了但我仍然写错了，错误代码如下，这样相当于直接续命，而不是遍历的过程中找边界

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return 0
        res = 0
        reach_dis = 0
        for i in range(len(nums)):
            if i == reach_dis:
                res += 1
            reach_dis = max(reach_dis, i + nums[i])
            if reach_dis >= len(nums) - 1:
                break
        return res
```

本体贪心思想是尽可能的边界靠右，到达边界后跳一下

```python
class Solution:
    def jump(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return 0
        res = 0
        reach_dis = 0
        cur_dis = 0
        for i in range(len(nums)):
            reach_dis = max(reach_dis, nums[i] + i)
            if i == cur_dis:
                res += 1
                cur_dis = reach_dis
                if cur_dis >= len(nums) - 1:
                    break
        return res
```

我思考了一下，这道题不能像上一题一样在遍历的过程中找到最远边界的原因是需要计算跳数。而一个 reach_dis 的作用是找到最远边界，这肯定是没有问题的。但是我们还需要一个变量可以记录什么时候跳，或者说，在哪一段范围应该跳。而这个变量，记录的不是 reach_dis 每一次发生变化的时候，而是记录在达到最远点的时候发生的变化（贪心：靠右）思考：[2, 3, 2, 4, 1], [2, 3, 4, 4, 1]

1005.K 次取反后最大化的数组和

```python
class Solution:
    def largestSumAfterKNegations(self, nums: List[int], k: int) -> int:
        nums.sort()
        for i in range(len(nums)):
            if nums[i] < 0 and k > 0:
                nums[i] *= -1
                k -= 1

        if k % 2 == 1:
            nums[0] *= -1

        return sum(nums)
```

简单不多说了

10.28.2024

134. 加油站

如果当前剩余总油量是负的，那很明显起始点不能满足绕一圈条件，就得换。

这里的思想是一个数学问题。如果在[0, i]区间的总油是<0 的，但是我们已经提前确定了总里程油量>0，那么[i+1:]就是正的，并且一定比[0, i]要大。从 I + 1 走，一定可以保证能够走完这个过程。

```python
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        residual = [gas[i] - cost[i] for i in range(len(gas))]
        if sum(residual) < 0:
            return -1
        res = 0
        cur = 0
        index = 0
        for i in range(len(residual)):
            cur += residual[i]
            if cur < 0:
                index = i + 1
                cur = 0
        return index
```

135. 分发糖果

只能说牛逼了，每一轮实际上都用到了贪心算法，但是第一轮是隐式的，第二轮是显示的。

```python
class Solution:
    def candy(self, ratings: List[int]) -> int:
        if len(ratings) == 1:
            return 1
        res = [1] * len(ratings)
        # from left to right, we need to consider from one order firstly
        for i in range(1, len(ratings)):
            if ratings[i] > ratings[i-1]:
                res[i] = res[i-1] + 1
        # from right to left, make sure that the children has the most candies compared with left and right children.
        # Hence, we need to use the greedy to ensure every child has candies that satisfies the conditions.
        for i in range(len(ratings) - 2, -1, -1):
            if ratings[i] > ratings[i+1]:
                res[i] = max(res[i], res[i + 1] + 1)
        return sum(res)
```

860.柠檬水找零

这个确实很简单，逻辑直译

```python
from collections import defaultdict
class Solution:
    def lemonadeChange(self, bills: List[int]) -> bool:
        cash = defaultdict(int)
        for item in bills:
            if item == 5:
                cash['5'] += 1
            elif item == 10:
                if cash['5'] <= 0:
                    return False
                cash['10'] += 1
                cash['5'] -= 1
            elif item == 20:
                if cash['5'] > 0 and cash['10'] > 0:
                    cash['10'] -= 1
                    cash['5'] -= 1
                    cash['20'] += 1
                elif cash['5'] >= 3:
                    cash['5'] -= 3
                    cash['20'] += 1
                else:
                    return False
        return True
```

406.根据身高重建队列

```python
class Solution:
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        people.sort(key=lambda x : (-x[0], x[1]))
        que = []

        for p in people:
            que.insert(p[1], p)

        return que
```

10.29.2024

重叠区间问题需要先想好按照什么样的方式排序，用哪个边界来进行贪心。

452. 用最少数量的箭引爆气球

竖着射箭，考虑重叠的最大子集就可以了。只需遍历一遍，只要有前后项不重合的时候，就会多用一只箭。

```python
class Solution:
    def findMinArrowShots(self, points: List[List[int]]) -> int:
        if len(points) == 1:
            return 1
        res = 1
        points.sort()
        for i in range(1, len(points)):
            if points[i-1][1] < points[i][0]:
                res += 1
            else:
                points[i][1] = min(points[i-1][1], points[i][1])
        return res
```

435. 无重叠区间

这道题我的想法是，贪心要确保的右边界尽可能的小。一旦重叠，就去除，保留右边界最小的那个。所以首先对右边界排序，我们可以画图看出左边界根本不重要

```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x : x[1])
        left = intervals[0]
        no_overlap = 1
        for i in range(1, len(intervals)):
            if intervals[i][0] >= left[1]:
                no_overlap += 1
                left = intervals[i]
        return len(intervals) - no_overlap
```

这样写的时候虽然思想是贪心的，但是感觉呈现上面并没有贪心的感觉，像是 min，max

```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x : x[0])
        res = 0
        for i in range(1, len(intervals)):
            if intervals[i][0] < intervals[i-1][1]:
                res += 1
                intervals[i][1] = min(intervals[i][1], intervals[i-1][1])
        return res
```

这样就是最符合贪心写法的模式，不过此时我们对左边界排序，直接判断移除数量

763.划分字母区间

不会，看的 carl 代码

先找到每一个 char 可以出现的最后位置

在遍历的过程中，确定每个不重复的字符串最大长度。

```python
class Solution:
    def partitionLabels(self, s: str) -> List[int]:
        last_position = {}
        for index, char in enumerate(s):
            last_position[char] = index

        res = []
        l, r = 0, 0
        for index, char in enumerate(s):
            r = max(r, last_position[char])
            if index == r:
                res.append(r - l + 1)
                l = r + 1

        return res
```

做前两题觉得重复区间贪心好简单，最后一题感觉好难。

10.30.2024

56. 合并区间

和昨天的一样

```python
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort()
        res = [intervals[0]]
        for i in range(1, len(intervals)):
            if res[-1][1] >= intervals[i][0]:
                res[-1][-1] = max(res[-1][-1], intervals[i][1])
            else:
                res.append(intervals[i])
        return res
```

738.单调递增的数字

没绕过来

```python
class Solution:
    def monotoneIncreasingDigits(self, n: int) -> int:
        num = list(str(n))

        for i in range(len(num) - 1, 0, -1):
            if num[i - 1] > num[i]:
                num[i - 1] = str(int(num[i - 1]) - 1)
                for back in range(i, len(num)):
                    num[back] = '9'

        return int("".join(num))
```

968.监控二叉树 （可跳过）

没看明白，不会

10.31.2024

开始动态规划：在求解过程中出现很多重叠子问题，后一个状态由前一个状态导出。0

步骤:

① 确定 DP 数组含义

② 确定递推公式

③ 初始化

④ 确定遍历顺序（前向后，后向前）

⑤ 举例子!!!!

509. 斐波那契数

虽然这道题目没有用到 dp 数组，但也是这个思路。Dp 代表现在位置的值，由前两个之和得到。初始是前两个值，从前向后遍历

```
class Solution:
    def fib(self, n: int) -> int:
        if n <= 1:
            return n
        else:
            pre_pre = 0
            pre = 1
            count = 2
            cur = 0
            while count <= n:
                cur = pre_pre + pre
                pre_pre = pre
                pre = cur
                count += 1
            return cur
```

70. 爬楼梯

dp = dp[i-2] + dp[i-1]

图中题目说 n>=1，实际上 n=1 和 n=2 我们都可以直接写出来当作初始条件，从 N=3 开始进行递推，也符合 dp 的公式条件。因为 0 没有意义，所以直接置 0 就行。

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        if n <= 2:
            return n
        step_2, step_1 = 1, 2
        cur = 0
        for _ in range(3, n + 1):
            cur = step_2 + step_1
            step_2, step_1 = step_1, cur
        return cur
```

746. 使用最小花费爬楼梯

确定 dp 数组意味着到达当前需要消耗的最小体力

dp = min(dp[i-1] + cost[i-1], dp[i-2] + cost[i-2])

根据题目所说，前两步都是 0

从前向后遍历

```python
class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        if len(cost) <= 2:
            return 0
        dp = [0] * (len(cost) + 1)
        for i in range(2, len(cost) + 1):
            dp[i] = min(dp[i-2] + cost[i-2], dp[i-1] + cost[i-1])
            print(dp)
        return dp[-1]
```

11.01.2024

62.不同路径

五部曲，简单明了

```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [[0 for _ in range(n)] for _ in range(m)]

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

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

        print(dp)
        return dp[-1][-1]
```

63. 不同路径 II

同样是五部曲，需要增加对障碍物的判断。

```python
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]) -> int:
        m, n = len(obstacleGrid), len(obstacleGrid[0])
        dp = [[0 for _ in range(n)] for _ in range(m)]

        if obstacleGrid[0][0] == 1:
            return 0

        dp[0][0] = 1
        for i in range(1, n):
            if obstacleGrid[0][i] == 0:
                dp[0][i] = 1
            else:
                for j in range(i, n):
                    dp[0][j] = 0
                break
        for i in range(1, m):
            if obstacleGrid[i][0] == 0:
                dp[i][0] = 1
            else:
                for j in range(i, m):
                    dp[j][0] = 0
                break

        for i in range(1, m):
            for j in range(1, n):
                if obstacleGrid[i][j] == 1:
                    dp[i][j] = 0
                else:
                    dp[i][j] = dp[i-1][j] + dp[i][j-1]

        print(dp)
        return dp[-1][-1]
```

343. 整数拆分

本题初始应该如何定义 dp 呢

Dp[i]表示当正整数为 i 的时候的最大乘积

我们可以直接从 2 开始定义(1+1)

(i-j) * j 代表两数乘积，dp[i-j] * j 代表多数乘积

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

        for i in range(3, n + 1):
            for j in range(1, i // 2 + 1):
                dp[i] = max(dp[i], (i - j) * j, dp[i-j] * j)

        return dp[-1]
```

96. 不同的二叉搜索树

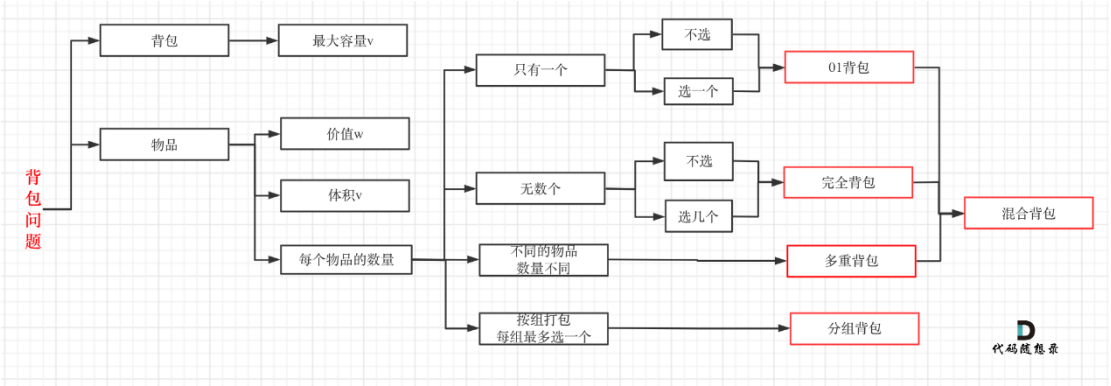
代码是真简单，但是真想不出来

11.02.2024

01 背包问题 二维

如果没有动态规划，想一想用贪心是怎么样的。按照单位价值最大的排序,假设背包大小为 3，大小:[1, 3, 3]，价值:[3, 5, 7]

那么必然得不到最优解，所以对于 01 背包问题(不能拆碎)证明是不能使用贪心的



按照 dp 的五部曲:

Dp[j]， 毫无疑问的代表容纳量为 j 的时候最大价值是多少。

Dp[i][j] ，代表的是当前 0-i 的物品可以任意选择，在背景容量为 j 的时候总背包价值是多少。

初始化 dp[0][j]，递推规律是 dp[i][j] =max(dp[i-1][j-space[i]] + value[i], dp[i-1][j])。满足条件是所剩容量大于 space[i]。一个是本次放入了这个物体，或者没放入（这步可以理解为，当前容量不满足放入这个物体，因为在之前的过程中包含了其他物体）。

```python
def main():
    input_content = input().split()
    M, N = int(input_content[0]), int(input_content[1])
    space = list(map(int, input().split()))
    value = list(map(int, input().split()))

    dp = [[0 for _ in range(N + 1)] for _ in range(M)]
    for i in range(0, N + 1):
        if i >= space[0]:
            dp[0][i] = value[0]


    for i in range(1, M):
        for j in range(1, N + 1):
            if j < space[i]:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-space[i]] + value[i])

    print(dp[-1][-1])

main()
```

01 背包问题 一维

一维背包问题难度++，首先不难看出，二维 dp 更新都是根据上一行进行更新。那么直接将当前操作在上一行的基础上完成呢？只是后引出了另一个问题。二维中我们的遍历顺序是左上到右下，在一维中我们的遍历顺序是从左到右。那么如果是从左到右进行遍历，肯定会改变先前的值，从而使后面的值更新的时候收到前面的影响。这对吗？肯定不对。解决方案就是从后向前进行更新。

```python
def main():
    input_content = input().split()
    M, N = int(input_content[0]), int(input_content[1])
    space = list(map(int, input().split()))
    value = list(map(int, input().split()))

    # 二维
    # dp = [[0 for _ in range(N + 1)] for _ in range(M)]
    # for i in range(0, N + 1):
    #     if i >= space[0]:
    #         dp[0][i] = value[0]


    # for i in range(1, M):
    #     for j in range(1, N + 1):
    #         if j < space[i]:
    #             dp[i][j] = dp[i-1][j]
    #         else:
    #             dp[i][j] = max(dp[i-1][j], dp[i-1][j-space[i]] + value[i])

    # 一维
    dp = [0 for _ in range(N + 1)]
    dp[0] = 0
    for i in range(M):
        for j in range(N, space[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - space[i]] + value[i])


    print(dp[-1])

main()
```

416. 分割等和子集

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        total = sum(nums)
        if total % 2 != 0:
            return  False
        target = total // 2

        dp = [0 for _ in range(target + 1)]
        for i in range(len(nums)):
            for j in range(target, nums[i] - 1, -1):
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i])

        return dp[target] == target
```

回溯法也可以

11.04.2024

今天不是很舒服，头有点停转了。这部分明天之前会重新看一遍

1049. 最后一块石头的重量 II

没想到应该如何定义 DP

Carl 将两落石块的分的尽可能一样。假设一堆石块是 x， 另一堆是 y，那么最终结果也就是|x – y|

```
class Solution:
    def lastStoneWeightII(self, stones: List[int]) -> int:
        target = sum(stones) // 2
        dp = [0 for _ in range(target + 1)]
        for i in range(0, len(stones)):
            for j in range(target, stones[i] - 1, -1):
                dp[j] = max(dp[j], dp[j - stones[i]] + stones[i])
        return sum(stones) - 2 * dp[-1]
```

494. 目标和

本题要如何使表达式结果为target，

既然为target，那么就一定有 left组合 - right组合 = target。

left + right = sum，而sum是固定的。right = sum - left

left - (sum - left) = target 推导出 left = (target + sum)/2 。

target是固定的，sum是固定的，left就可以求出来。

此时问题就是在集合nums中找出和为left的组合。

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        target_sum = (target + sum(nums)) // 2
        dp = [[0] * (target_sum + 1) for _ in range(len(nums) + 1)]

        dp[0][0] = 1

        for i in range(1, len(nums) + 1):
            for j in range(0, target_sum + 1):
                dp[i][j] = dp[i - 1][j]
                if j >= nums[i - 1]:
                    dp[i][j] += dp[i - 1][j - nums[i - 1]]

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

一维 DP

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        target_sum = (target + sum(nums)) // 2
        if abs(target) > sum(nums):
            return 0
        if (target + sum(nums)) % 2 == 1:
            return 0
        dp = [0] * (target_sum + 1)

        dp[0] = 1

        for i in range(len(nums)):
            for j in range(target_sum, nums[i] - 1, -1):
                dp[j] += dp[j - nums[i]]
        return dp[-1]
```

474. 一和零

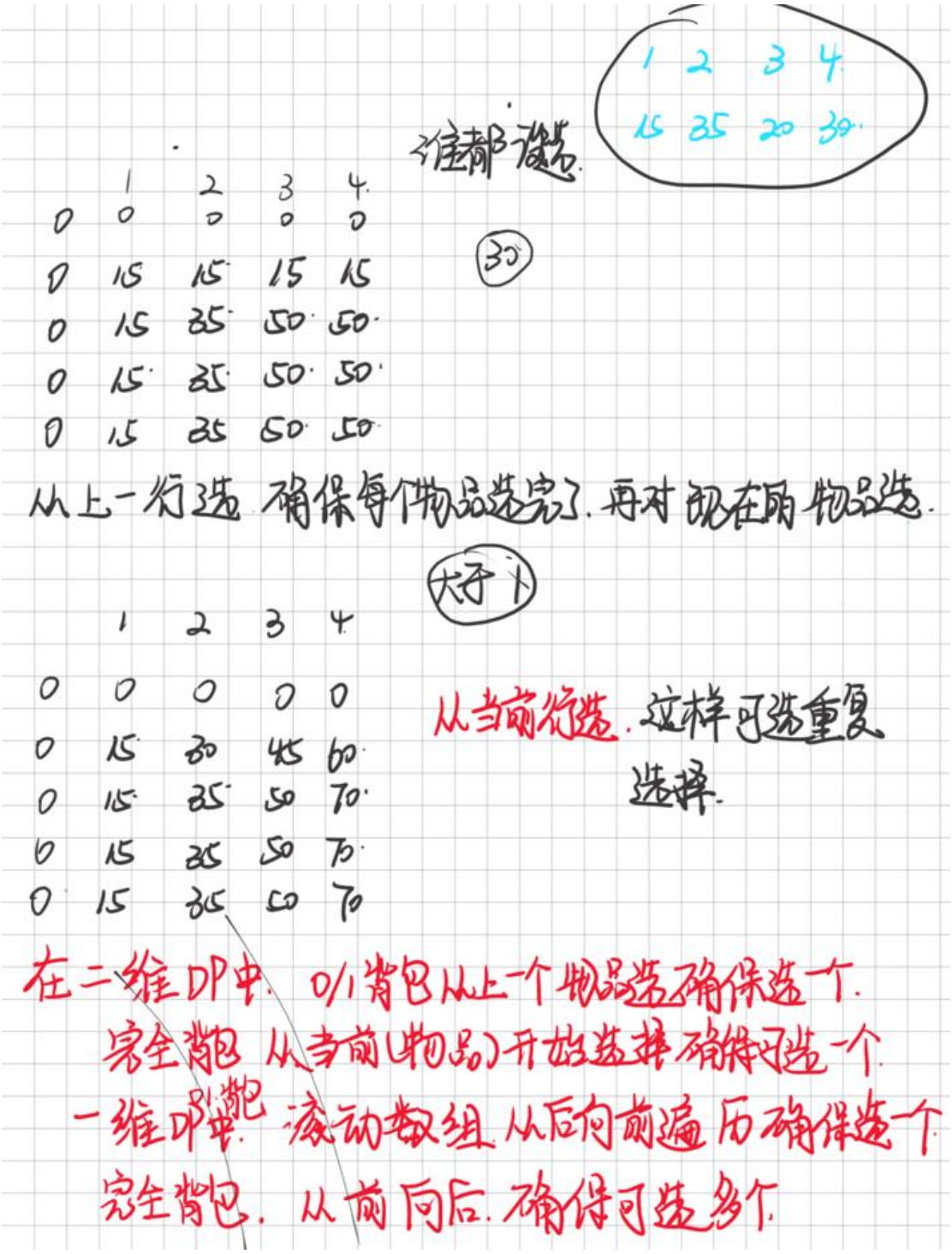
```python
class Solution:
    def findMaxForm(self, strs: List[str], m: int, n: int) -> int:
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        for s in strs:
            ones = s.count('1')
            zeros = s.count('0')
            for i in range(m, zeros - 1, -1):
                for j in range(n, ones - 1, -1):
                    dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1)
        return dp[m][n]
```

本题为什么两个 for 循环中也要用从后向前遍历呢，难道不是只有一维滚动数组才是吗？ 因为本题中要开始遍历 n 个字符串，遍历过程中涉及到之前的字符串。

11.05.2024

4 号内容已重看，重写

今日内容：**完全背包**问题



52. 携带研究材料

每个物品可以无限的重复选择，所以从遍历循序上，即使滚动数组也可以一直正向。

二维 DP

```python
N, V = map(int, input().split())
weights = []
values = []

for i in range(N):
    weight, value = map(int, input().split())
    weights.append(weight)
    values.append(value)

dp = [[0 for _ in range(V + 1)] for i in range(N + 1)]

# 初始化
for i in range(1, N + 1):
    for j in range(1, V + 1):
        dp[i][j] = dp[i-1][j]
        if j >= weights[i - 1]:
            dp[i][j] = max(dp[i][j], dp[i][j - weights[i-1]] + values[i-1])

print(dp[-1][-1])
```

一维 DP

```python

N, V = map(int, input().split())
weights = []
values = []

for i in range(N):
    weight, value = map(int, input().split())
    weights.append(weight)
    values.append(value)

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

# 初始化
for i in range(0, N):
    for j in range(weights[i], V + 1):
        dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

print(dp[-1])
```

518. 零钱兑换 II

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0 for _ in range(amount + 1)]
        dp[0] = 1
        for i in range(len(coins)):
            for j in range(coins[i], amount + 1):
                dp[j] += dp[j - coins[i]]
        return dp[-1]
```

Stdout

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

当我改变遍历顺序，先遍历总金额的时候

```python
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0 for _ in range(amount + 1)]
        dp[0] = 1
        for i in range(1, amount + 1):
            for j in range(len(coins)):
                dp[i] += dp[i - coins[j]]
        print(dp)
        return dp[-1]
```

Stdout

```
[1, 1, 2, 3, 5, 9]
```

本题目求的是组合问题，所以在先谁后谁的问题上有了要求。

如果是先钢锄再空间，那可以确保在 coin1,5 同时出现的时候，1 永远在 5 之前。而如果先空间再钢锄，那可能 coin5 先比 1 出现，也存在 1 在 5 之间的这种情况。

377. 组合总和 IV

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

    def combinationSum4(self, nums: List[int], target: int) -> int:

        def backtracking(temp_sum, target, temp_path, nums):
            if target == temp_sum:
                self.res += 1
                return

            for i in range(len(nums)):
                if temp_sum + nums[i] <= target:
                    temp_path.append(nums[i])
                    backtracking(temp_sum + nums[i], target, temp_path, nums)
                    temp_path.pop()

        backtracking(0, target, [], nums)
        return self.res
```

很不幸，回溯法超过时间限制了。

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

        for i in range(1, target + 1):
            for j in range(len(nums)):
                if i - nums[j] >= 0:
                    dp[i] += dp[i - nums[j]]

        return dp[-1]
```

70. 爬楼梯 （进阶）

```python
n, m = map(int, input().split())

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

print(dp[-1])
```

今日总结：在组合问题(不同序但相同元素组成视为一个)中，先元素再背包。

排列问题（不同序各视为一个），先背包再元素。

11.06.2024

322. 零钱兑换

我个人感觉的是在求最值问题的时候，并不需要在意遍历的顺序，因为最终“殊途同归”。但是在求几种方式的时候，就要注意题目中阐述的到底是排列组合哪个问题。本题的思路不难想，稍微难点在于 dp 的初始化。因为涉及到求最小，所以初始化应该为最大值，这样才能被覆盖。

一维写法

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # dp represents the number of coins.
        dp = [float("inf") for _ in range(amount + 1)]
        dp[0] = 0

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

        return dp[-1] if dp[-1] != float("inf") else -1
```

二维写法

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [[float("inf") for _ in range(amount + 1)] for _ in range(len(coins) + 1)]
        for i in range(len(coins) + 1):
            dp[i][0] = 0
        for i in range(1, len(coins) + 1):
            for j in range(1, amount + 1):
                dp[i][j] = dp[i-1][j]
                if j >= coins[i-1]:
                    dp[i][j] = min(dp[i][j], dp[i][j-coins[i-1]] + 1)
        return dp[-1][-1] if dp[-1][-1] != float("inf") else -1
```

279.完全平方数

和上题一样的思路

本题物品 i 也就是完全平方的项，只需要小于根号下 n 就可以。

```python
class Solution:
    def numSquares(self, n: int) -> int:
        # dp represents the least number of res
        dp = [float("inf") for _ in range(n + 1)]
        dp[0] = 0

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

139.单词拆分

本题难度大于上两题，首先要明确 dp 的意义：dp[i]代表从开始到 i 处的字符串能否被分割。那么在初始化上，最开始一定为 1 或者 True。在每个单词组成的字符串上，这是有序的，所以是排列问题，而不是组合问题。

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

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

也可以使用回溯法

11.07.2024

198.打家劫舍

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        # dp[i] represents the greatest value in the position i
        # dp[i] = max(dp[i-1], dp[i-2] + nums[i]), which ensures that you cannot choose the values constantly
        # initialization : dp[0] = nums[0], dp[1] = max(dp[0], dp[1])
        if len(nums) <= 2:
            return nums[0] if len(nums) == 1 else max(nums[0], nums[1])

        dp = [0 for _ in range(len(nums))]
        dp[0], dp[1] = nums[0], max(nums[0], nums[1])

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

        return dp[-1]
```

一开始我有疑惑的是：即使直到 dp[i]代表目前 i 处最优值，但是在递推逻辑的时候如何确保至少隔一个抢劫一个呢，甚至是隔两个三个….

但其实是不可能隔两个或者三个的。因为 dp 代表的是抢劫的最大值，这是一个单调不减数组，所以当前值永远只能从前两个中选。那么问题就好解决了

Dp = max(dp[i-1], dp[i-2] + nums[i]), 不选上一个，选上上个和当前的；或者选上一个，不选当前的。这样的逻辑，既保证了不会选择重复值，也会选择最大抢劫总和。

213.打家劫舍 II

这是我一开始的思路，将末尾移到最前面，继续沿用 198 思路。

反例：[2, 3, 2] 改变后变成[2, 2, 3]结果为 5。思路错误

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        nums = [nums[-1]] + nums[:-1]

        if len(nums) <= 2:
            return nums[0] if len(nums) == 1 else max(nums[0], nums[1])

        dp = [0 for _ in range(len(nums))]
        dp[0], dp[1] = nums[0], max(nums[0], nums[1])

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

        return dp[-1]
```

Carl 思路：既然第一个和最后一个不能同时存在，那我们直接分成两个数组分别进行抢劫，找到其中最大的那个就好了，TMD 天才，我好蠢。

```
class Solution:
    def rob(self, nums: List[int]) -> int:
        if len(nums) <= 2:
            return nums[0] if len(nums) == 1 else max(nums[0], nums[1])

        def rob_require(nums):
            dp = [0 for _ in range(len(nums))]
            dp[0], dp[1] = nums[0], max(nums[0], nums[1])
            for i in range(2, len(nums)):
                dp[i] = max(dp[i-1], dp[i-2] + nums[i])
            return dp[-1]

        nums1 = nums[:-1]
        nums2 = nums[1:]
        return max(rob_require(nums1), rob_require(nums2))
```

337.打家劫舍 III

树形 DP

虽然想到了后序遍历，但是具体遍历的逻辑还有 dp 代表什么及如何初始化还是没有想出来。

```
class Solution:
    def rob(self, root: Optional[TreeNode]) -> int:
        # dp数组（dp table）以及下标的含义：
        # 1. 下标为 0 记录 **不偷该节点** 所得到的的最大金钱
        # 2. 下标为 1 记录 **偷该节点** 所得到的的最大金钱
        dp = self.traversal(root)
        return max(dp)

    # 要用后序遍历，因为要通过递归函数的返回值来做下一步计算
    def traversal(self, node):

        # 递归终止条件，就是遇到了空节点，那肯定是不偷的
        if not node:
            return (0, 0)

        left = self.traversal(node.left)
        right = self.traversal(node.right)

        # 不偷当前节点，偷子节点
        val_0 = max(left[0], left[1]) + max(right[0], right[1])

        # 偷当前节点，不偷子节点
        val_1 = node.val + left[0] + right[0]

        return (val_0, val_1)
```

返回二维数组，代表抢劫该节点或者其子节点，真心佩服，难以想到。

二叉树后序遍历算法，没有用到 DP， 以字典来减少重复计算次数

```python
class Solution:
    def __init__(self):
        self.memory = {}
    def rob(self, root: TreeNode) -> int:
        if root is None:
            return 0
        if root.left is None and root.right  is None:
            return root.val
        if self.memory.get(root) is not None:
            return self.memory[root]

        val1 = root.val
        if root.left:
            val1 += (self.rob(root.left.left) + self.rob(root.left.right))
        if root.right:
            val1 += (self.rob(root.right.left) + self.rob(root.right.right))

        val2 = self.rob(root.left) + self.rob(root.right)
        self.memory[root] = max(val1, val2)
        return self.memory[root]
```

11.08.2024

121. 买卖股票的最佳时机

暴力会超过时间复杂度、贪心可以，时间为 O(n)。

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        res = 0
        buy = float("inf")

        for price in prices:
            if price < buy:
                buy = price
            res = max(res, price - buy)

        return res
```

DP:

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        dp = [[0, 0] for _ in range(len(prices))]
        dp[0][0] = -prices[0]
        for i in range(1, len(prices)):
            dp[i][0] = max(dp[i-1][0], -prices[i])
            dp[i][1] = max(dp[i-1][1], prices[i] + dp[i-1][0])
        print(dp)
        return dp[-1][1]
```

一开始没有想到应该用二维进行 DP 的设计，分别表示持有和卖出。

Dp[i][0], dp[i][1]分别表示在 i 处持有(买入)，在 i 处不持有(卖出)。可以参考上述贪心的想法，dp[i][0]意味着我们需要找到最便宜的那只股票(例如，-1 > -3)这也代表着他的价值，而 dp[i][1]代表着我们应当什么时候卖出所获得的最大价值。二维的 DP 既保证了可以知晓买入的最佳时机，也可以通过买入的最佳时机来判断是否是卖出的最佳时机。

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # dp = [[0, 0] for _ in range(len(prices))]
        # dp[0][0] = -prices[0]
        # for i in range(1, len(prices)):
        #     dp[i][0] = max(dp[i-1][0], -prices[i])
        #     dp[i][1] = max(dp[i-1][1], prices[i] + dp[i-1][0])
        # print(dp)
        # return dp[-1][1]

        # 一维DP
        buy_optimal = -prices[0]
        dp = 0
        for i in range(1, len(prices)):
            dp = max(dp, buy_optimal + prices[i])
            buy_optimal = max(buy_optimal, -prices[i])

        return dp
```

122.买卖股票的最佳时机 II

之前已经写过贪心算法

现在继续 DP

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # res = 0
        # for i in range(1, len(prices)):
        #     res += max(prices[i] - prices[i - 1], 0)
        # return res

        dp = [[0, 0] for _ in range(len(prices))]
        dp[0][0] = -prices[0]
        dp[0][1] = 0
        for i in range(1, len(prices)):
            dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i])
            dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i])
        return dp[-1][1]
```

可以和贪心算法对照着看，因为可以无数次卖出，所以我们的利润是可以叠加的。因此，我们的第 i 天持有后获得的现金应该算上前一天的利润。

123.买卖股票的最佳时机 III

这道题目没思路，最初的想法是：121 题目中的就是买卖一次，那这部分一定要在代码中表示出来，那第二次买卖如何表现呢？没有想法

想了一会直接看 carl

1. 确定dp数组以及下标的含义

一天一共就有五个状态，

0. 没有操作 （其实我们也可以不设置这个状态）
1. 第一次持有股票
2. 第一次不持有股票
3. 第二次持有股票
4. 第二次不持有股票

dp[i][j]中 i表示第i天，j为 [0 - 4] 五个状态，dp[i][j]表示第i天状态j所剩最大现金。

这里我想，没有操作应该包括在了第一次持有股票中，因为如果一直是单调不增数列，那么最终结果就是 0

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 0 第一次持有股票
        # 1 第一次不持有股票
        # 2 第二次持有股票
        # 3 第二次不持有股票

        dp = [[0] * 4 for _ in range(len(prices))]
        # 如何初始化
        dp[0][0] = -prices[0]
        # 下面非常重要
        dp[0][2] = -prices[0]

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

        return max(dp[-1][1], dp[-1][3])
```

最初我对 dp[0][2]的定义是错的，我的想法是 dp[0][2]因为没有经过第一次的买卖，所以谈不上第二次的买卖，所以必定为 0。

```
[[-1, 0, 0, 0], [-1, 1, 0, 2], [-1, 2, 0, 3], [-1, 3, 0, 4], [-1, 4, 0, 5]]
```

prices =

```
[1,2,3,4,5]
```

Stdout

```
[[-1, 0, -1, 0], [-1, 1, -1, 1], [-1, 2, -1, 2], [-1, 3, -1, 3], [-1, 4, -1, 4]]
```

第二次买入依赖于第一次卖出的状态，其实相当于第0天第一次买入了，第一次卖出了，然后再买入一次（第二次买入），那么现在手头上没有现金，只要买入，现金就做相应的减少。

所以第二次买入操作，初始化为：dp[0][3] = -prices[0];

同理第二次卖出初始化dp[0][4] = 0;

对于第二次持有的初始化的解释非常巧妙，也符合了 dp 推导的逻辑。如果 dp[0][3]=0 的时候，说明没有发生任何交易，甚至谈不上第一次持有，那么哪里来的到第二次呢?

或者我们反着想一下，既然第一次已经发生发生了交易，产生了利润上的变动，那么第二次怎么还是 0 呢？所以说第二次持有的初始化要在第一次的基础上进行变动。

11.09.2024

188.买卖股票的最佳时机 IV

本次可以卖 k 次，所以需要找到一个更巧妙的方法，并且这个方法可以涵盖之前三题。所以是否可以用二维数组 dp[i][j]， j=k 来判断呢。

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 0 第一次持有股票
        # 1 第一次不持有股票
        # 2 第二次持有股票
        # 3 第二次不持有股票

        dp = [[0] * 4 for _ in range(len(prices))]
        # 如何初始化
        dp[0][0] = -prices[0]
        # 下面非常重要
        dp[0][2] = -prices[0]

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

        return max(dp[-1][1], dp[-1][3])
```

昨天题目中，因为不操作的可能性归并到未持有股票中，所以不需要做特殊处理。今天涉及到 K 次，要对循环做统一化处理，所以需要初始化未操作 dp。

```python
class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        dp = [[0] * (2 * k + 1) for _ in range(len(prices))]

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

        for i in range(1, len(prices)):
            for j in range(1, 2 * k + 1, 2):
                # 第k次持有
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-1] - prices[i])
                # 第k次不持有
                dp[i][j+1] = max(dp[i-1][j+1], dp[i-1][j] + prices[i])

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

309.最佳买卖股票时机含冷冻期

版本一看着有点晕晕的。

版本二倒是很好直观的理解，我的思路在注释中。

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        dp = [[0] * 3 for _ in range(len(prices))]

        dp[0][0] = -prices[0]

        for i in range(1, len(prices)):
            # 第i天持有
            dp[i][0] = max(dp[i-1][0], dp[i-1][2] - prices[i])
            # 第i天处于冷静期，第i-1天卖出
            # 在不持有股票的前提条件下，交易只发生在冷静期的时候
            dp[i][1] = dp[i-1][0] + prices[i]
            # 第i天不处于冷静期，可以买卖
            # 不处于冷静期是怎么来的，要么前一天就不是冷静期，要么前一天是冷静期。但不管那种情况，交易只发生在之前而不是现在。
            dp[i][2] = max(dp[i-1][2], dp[i-1][1])

        return max(dp[-1][1], dp[-1][2])
```

714.买卖股票的最佳时机含手续费

```python
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        dp = [[0] * 2 for _ in range(len(prices))]

        # dp[i][0]为持有股票，dp[i][1]为不持有股票
        dp[0][0] = -prices[0]

        for i in range(1, len(prices)):
            dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i])
            dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i] - fee)

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

一个完整的交易过程，也就是卖出的时候，需要增加一笔手续费。

11.11.2024

300.最长递增子序列

Dp[i]为 nums[i]所在的位置最长子序列是多少

初始化肯定都是-1

递推公式 dp[i] = max(dp[i], dp[j] + 1) j 是比 i 小的所有数字

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

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

        return max(dp)
```

本题目可以用 res 记录优化，减少 O(n)的时间

674. 最长连续递增序列

本题目比上题少了一维，因为必须考虑连续而不是分开的子序列了。

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

        for i in range(1, len(nums)):
            if nums[i] > nums[i-1]:
                dp[i] = dp[i-1] + 1
                res = dp[i] if dp[i] > res else res

        return res
```

用贪心的话更加简单，省去了 O(n)的空间复杂度

718. 最长重复子数组

这题没想出来，看了 carl 的

结合以下两个 dp 来看

第一个 dp[i][j]的定义为在 nums1[i-1]和 nums2[j-1]最大重复子数组

```python
class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        dp = [[0] * (len(nums2) + 1) for _ in range(len(nums1) + 1)]
        res = 0

        for i in range(1, len(nums1) + 1):
            for j in range(1, len(nums2) + 1):
                if nums1[i-1] == nums2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                    res = dp[i][j] if dp[i][j] > res else res

        return res
```

第二个 dp[i][j]的定义为在 nums1[i]和 nums2[j]最大重复子数组，相对更加符合我们的定义的直觉。

不过初始化的定义要稍微复杂一些.

```python
class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        dp = [[0] * (len(nums2)) for _ in range(len(nums1))]
        res = 0

        for i in range(len(nums1)):
            if nums1[i] == nums2[0]:
                dp[i][0] = 1

        for i in range(len(nums2)):
            if nums1[0] == nums2[i]:
                dp[0][i] = 1

        for i in range(0, len(nums1)):
            for j in range(0, len(nums2)):
                if i > 0 and j > 0 and nums1[i] == nums2[j]:
                    dp[i][j] = dp[i-1][j-1] + 1
                res = dp[i][j] if dp[i][j] > res else res

        return res
```

一维 DP 写法:

```python
class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        dp = [0 for _ in range(len(nums1) + 1)]
        res = 0

        for i in range(1, len(nums2) + 1):
            for j in range(len(nums1), 0, -1):
                if nums1[j - 1] == nums2[i - 1]:
                    dp[j] = dp[j - 1] + 1
                else:
                    dp[j] = 0
                res = dp[j] if dp[j] > res else res

        return res
```

也需要从后向前遍历来确保当前遍历的长度不会受到之前影响。

11.11.2024

1143.最长公共子序列

这道题的解决思路不难想出来，遍历方向出了问题，为什么一定要是从上到下呢？

想了一会才明白，我们更新时候肯定需要所有的更新子状态已经更新完毕。如果从左到右，那么在更新的时候会忽略到 dp[i-1][j]的状态。

遍历方向的重要性

```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:

        dp = [[0 for _ in range(len(text1) + 1)] for _ in range(len(text2) + 1)]

        for i in range(1, len(text2) + 1):
            for j in range(1, len(text1) + 1):
                if text1[j-1] == text2[i-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
        return dp[-1][-1]
```

1035.不相交的线

没啥思路，看完 carl 恍然大悟，跪了

和最长公共子序列问题一样，因为我们确保两个序列相等数字相对位置不变，就可以保证不会重复

二维写法：

```python
class Solution:
    def maxUncrossedLines(self, nums1: List[int], nums2: List[int]) -> int:
        dp = [[0 for _ in range(len(nums2) + 1)] for _ in range(len(nums1) + 1)]

        for i in range(1, len(nums1) + 1):
            for j in range(1, len(nums2) + 1):
                if nums1[i-1] == nums2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])

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

一维写法：

主要是两个问题，本次的滚动数组涉及到同层滚动，应该如何进行？

之前的滚动数组大多都是从后向前，为什么这次可以从前向后呢？

这两个问题一起看，滚动数组从后向前是因为避免影响前面结果影响后面的。

而这次的滚动数组本身就需要比较前向的，所以需要从前向后进行。

```python
class Solution:
    def maxUncrossedLines(self, nums1: List[int], nums2: List[int]) -> int:
        prev = [0 for _ in range(len(nums2) + 1)]

        for i in range(1, len(nums1) + 1):
            dp = [0 for _ in range(len(nums2) + 1)]
            for j in range(1, len(nums2) + 1):
                if nums1[i-1] == nums2[j-1]:
                    dp[j] = prev[j-1] + 1
                else:
                    dp[j] = max(prev[j], dp[j-1])
            prev = dp
        return dp[-1]
```

53. 最大子序和

之前做过一次

这道题感觉直观来说还是贪心更简单一些。

但是贪心需要处理的问题是当前最值小于 0 则变成 0 这个弯

```python
class Solution:
    def maxSubArray(self, nums):
        result = float('-inf')  # 初始化结果为负无穷大
        count = 0
        for i in range(len(nums)):
            count += nums[i]
            if count > result:  # 取区间累计的最大值（相当于不断确定最大子序终止位置）
                result = count
            if count <= 0:  # 相当于重置最大子序起始位置，因为遇到负数一定是拉低总和
                count = 0
        return result
```

DP:

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

392.判断子序列

和最长公共子序列是一样的情况，最后只需要判断最长子序列是否等于两个序列长度较小者。

```python
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        if not t and s:
            return False
        elif t and not s:
            return True
        elif not t and not s:
            return True


        dp = [[0 for _ in range(len(t))] for _ in range(len(s))]


        for i in range(0, len(t)):
            if t[i] == s[0]:
                for j in range(i, len(t)):
                    dp[0][j] = 1

        for i in range(0, len(s)):
            if s[i] == t[0]:
                for j in range(i, len(s)):
                    dp[j][0] = 1

        for i in range(1, len(s)):
            for j in range(1, len(t)):
                if s[i] == t[j]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
        return dp[-1][-1] == len(s)
```

可以看到，如果初始化 dp[i][j]的意义是位于 s[i]和 t[j]位置处最大相同子序列长度，那么会复杂很多。