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

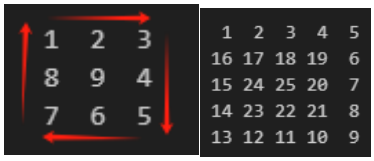
⑤ 返回答案

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

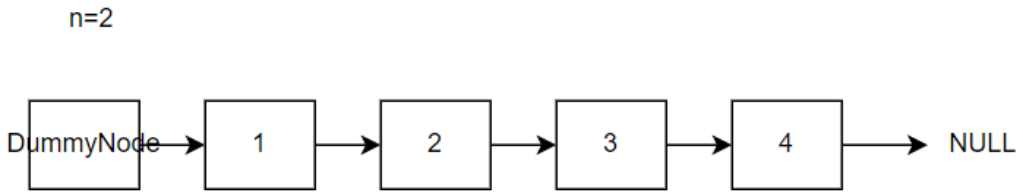
```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

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)

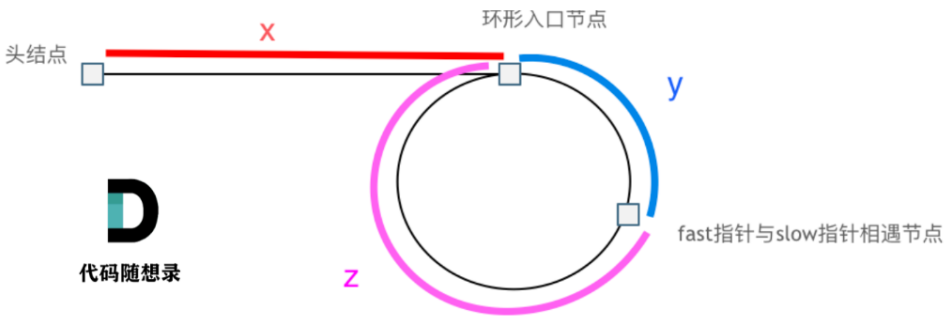
142.环形链表 II

```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。