Basic Data Structures

链表(Linked list)

链表的定义：

```
class ListNode:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next
```

虚拟头结点：

链表的一大问题就是操作当前节点必须要找前一个节点才能操作。这就造成了，头结点的尴尬，因为头结点没有前一个节点了。

每次对应头结点的情况都要单独处理，所以使用虚拟头结点的技巧，就可以解决这个问题。

```
dummy_head = ListNode(next=head) #添加一个虚拟节点
```

链表的增删改查

经典题

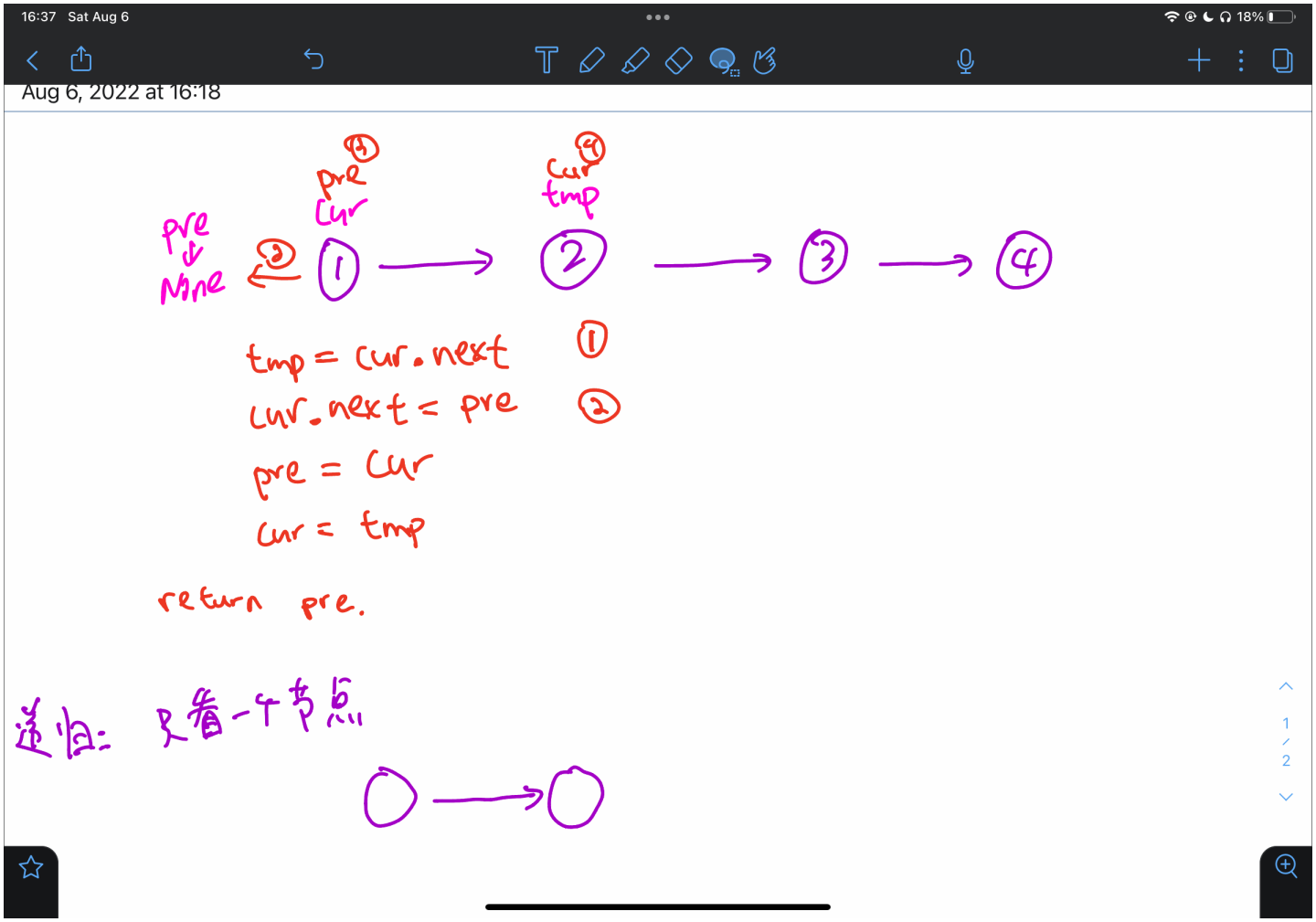
203.Remove Linked List Elements

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

206.Reverse Linked List

原：1->2->3->空

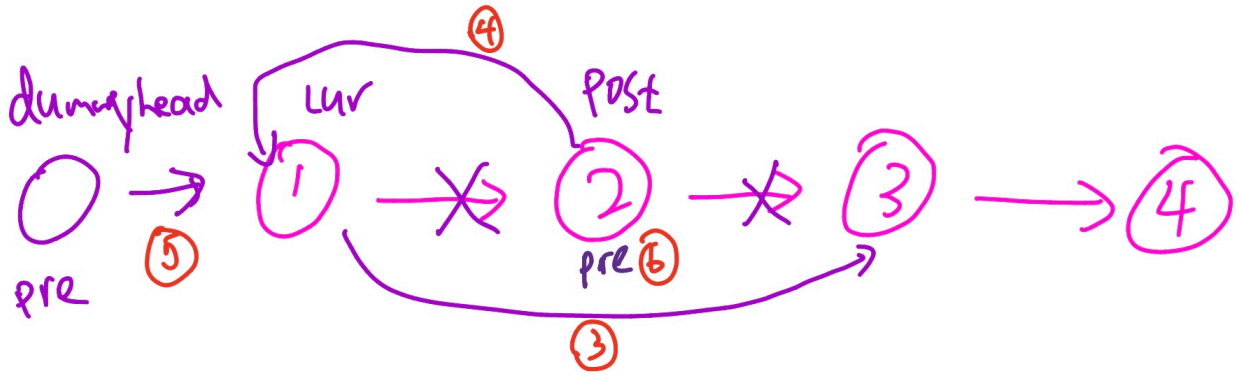
现在：空->3->2->1




```
#迭代(双指针)
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        pre = None
        cur = head
        while cur:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
        return pre
```

```python
#递归
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        def reverse(pre,cur):
            if not cur:
                return pre

            tmp = cur.next
            cur.next = pre

            return reverse(cur,tmp) #重点在这里

        return reverse(None,head)
```

24. Swap Nodes in Pairs



cur = pre.next

post = pre.next.next

cur.next = post.next ③

post.next = cur ④

pre.next = post ⑤

pre = pre.next.next ⑥

```python
class Solution:
    def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        #pre用来控制跳一格 巧妙
        dummyhead = ListNode(next=head)
        pre = dummyhead
        while pre.next and pre.next.next:
            cur = pre.next
            post = pre.next.next
            cur.next = post.next
            post.next = cur
            pre.next = post
            pre = pre.next.next
        return dummyhead.next
```

19.Remove Nth Node From End of List

快慢指针。需要注意的是n的边界问题，是大于等于还是大于。

```python
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        #快慢指针 快指针先走n步 当快指针到达末尾时 慢指针的next就是要删除的
        dummyhead = ListNode(next=head)
        slow = dummyhead
        fast = dummyhead
        while n>0:
            fast = fast.next
            n -= 1
        while fast.next:
            slow = slow.next
            fast = fast.next
        slow.next = slow.next.next
        return dummyhead.next
```

160.Intersection of Two Linked Lists

```python
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> Optional[ListNode]:
        # 难度在于两个链的长度不相等
        # 遍历2次，第一次到达结尾后去到另一条链遍历，这样就能弥补长度差
        p = headA
        q = headB
        while p!=q:
            if not p:
                p = headB
```

```
            else:
                p = p.next
            if not q:
                q = headA
            else:
                q = q.next

        '''
        while p!=q:
          p = p.next if p else headB
          q = q.next if q else headA
        '''
        return p
```

141.Linked List Cycle

```
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        #快慢指针
        #快指针一次2步，慢指针一次一步，如果能相遇则有还
        dummyhead = ListNode(next=head)
        slow= dummyhead
        fast = dummyhead
        while fast and fast.next:
            fast = fast.next.next
            slow = slow.next
            if slow == fast:
                return True
        return False
```

142.Linked List Cycle II（求入口）

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

        #找到相交的地方
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                break

        #无环 返回None
```

```
        if not fast or not fast.next:
            return None

        #快指针留下 慢指针重新开始 最终他们会在环的开头相交
        #为什么会这样? 看图
        slow = head
        while slow != fast:
            slow = slow.next
            fast = fast.next
        return slow
```

练习

21. Merge Two Sorted Lists

```
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) ->
Optional[ListNode]:
        #不在原链上修改 很复杂
        #新开一个头结点 接在后面即可

        dummyhead = ListNode(next = None)
        p = dummyhead
        headA = list1
        headB = list2
        while headA and headB:
            if headA.val>headB.val:
                p.next = headB
                headB = headB.next
            else:
                p.next = headA
                headA = headA.next
            p = p.next

        if headA:
            p.next = headA

        if headB:
            p.next = headB

        return dummyhead.next
```

86. Partition List

```python
class Solution:
    def partition(self, head: Optional[ListNode], x: int) -> Optional[ListNode]:
        # 新开一个头结点small 比x小的插入
        # 新开一个头结点big 比x大的插入
        # big插到small后面即可
        small = ListNode(next=None)
        smallhead = small
        big = ListNode(next=None)
        bighead = big

        #这样写会断链，但是无所谓，我们只关心head后面的结点，他们会以此存放到small或者big
        while head:
            if head.val < x:
                smallhead.next = head
                smallhead = smallhead.next
            else:
                bighead.next = head
                bighead = bighead.next
            head = head.next

        bighead.next = None
        smallhead.next = big.next
        return small.next
```

876.Middle of the Linked List

```python
class Solution:
    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        #快慢指针 快指针两倍速走 慢指针指向的就是中间
        fast = head
        slow = head
        #fast and fast.next处理单双数问题
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        return slow
```

143. Reorder List

143. Reorder List

同时考察206, 21，876. 非常好的题目

```
class Solution:
    def reorderList(self, head: Optional[ListNode]) -> None:
        """
        Do not return anything, modify head in-place instead.
        """
        #思路
        '''
        1、快慢指针找到中间节点
        2、反转后半段链表
        3、合并前半段链表和后半段链表
        '''

        def find_mid(head):
            slow = head
            fast = head
            while fast.next and fast.next.next:
                slow = slow.next
                fast = fast.next.next
            return slow

        def reverse_link(head):
            pre = None
            cur = head
            while cur:
                tmp = cur.next
                cur.next = pre
                pre = cur
                cur = tmp
            return pre

        #因为链表长度只相差1或者0，所以可以这样搞。但是对于长度差距大于1的链表需要用21的方法
        def merge_list(l1,l2):
            while l1 and l2:
                tmp1 = l1.next
                tmp2 = l2.next

                l1.next = l2
                l1 = tmp1

                l2.next = l1
                l2 = tmp2

        #找到中间
```

```
        #找到中间
        if not head:
            return None
        mid = find_mid(head)
        l1 = head
        l2 = mid.next
        mid.next=None
        l2 = reverse_link(l2)
        merge_list(l1,l2)
```

328. Odd Even Linked List

```
class Solution:
    def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        '''
        结点1作为奇数链的头  结点2作为偶数链的头
        从第3个点开始遍历，依次轮流附在奇、偶链的后面
        遍历完后，奇数链的尾连向偶链的头，偶链的尾为空，  返回奇数链的头
        '''
        if not head:
            return None
        odd = head
        even = head.next
        even_head = head.next

        while even and even.next:
            odd.next = even.next
            odd = odd.next
            even.next = odd.next
            even = even.next
        odd.next = even_head
        return head
```

92. Reverse Linked List II

```
class Solution:
    def reverseBetween(self, head: Optional[ListNode], left: int, right: int) ->
Optional[ListNode]:
        #难点在于边界
        # 题目给的left right是真实的下标+1
        dummyhead = ListNode(next=head)
        pre = dummyhead
        for _ in range(1,left):
```

```
        for _ in range(1,left):
            pre = pre.next

        cur = pre.next
        for _ in range(left,right):
            tmp = cur.next
            cur.next = tmp.next
            tmp.next = pre.next
            pre.next = tmp

        return dummyhead.next
```

237.Delete Node in a Linked List

题目限制 不能访问head

把下一node的值复制到当前node，然后跳过下一节点

```
class Solution:
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        node.val = node.next.val
        node.next = node.next.next
```

234. Palindrome Linked List

注意：当长度是奇数的时候，mid应该是中间节点再右移一位

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        # 找到中间
        # 将后面的翻转看是否等于前面
        slow = head
        fast = head
        while fast.next and fast.next.next:
            slow = slow.next
```

```
            slow = slow.next
            fast = fast.next.next
        if fast != None:
            slow = slow.next

        def reverse(head):
            if not head:
                return None
            pre = None
            cur = head
            while cur:
                tmp = cur.next
                cur.next = pre
                pre = cur
                cur = tmp
            return pre

        left = head
        right = reverse(slow)
        while right:
            if right.val != left.val:
                return False
            right = right.next
            left = left.next

        return True
```

148. Sort List

考点：合并两个有序列表，找中间节点，归并排序

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        # 链表的归并排序
        # 考点：合并两个有序列表，找中间节点，归并排序
        if not head or not head.next:
            return head
        mid = self.find_mid(head)
        right_head = mid.next

        mid.next = None
```

```
        mid.next = None
        left_head = head

        left = self.sortList(left_head)
        right = self.sortList(right_head)
        new_list = self.merge(left,right)
        return new_list

    def find_mid(self,head):
        if not head or not head.next:
            return None
        slow = head
        fast = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next
        return slow

    def merge(self,list1,list2):
        dummyhead = ListNode(next = None)
        p = dummyhead
        headA = list1
        headB = list2
        while headA and headB:
            if headA.val>headB.val:
                p.next = headB
                headB = headB.next
            else:
                p.next = headA
                headA = headA.next
            p = p.next

        if headA:
            p.next = headA

        if headB:
            p.next = headB

        return dummyhead.next
```

61. Rotate List

~~61. Rotate List~~

配合画图

```
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        #不停地把最后一个节点放在头结点前面
        if not head:
            return None

        #统计链表有多少个结点
        total = 1
        p = head
        while p.next:
            p = p.next
            total += 1

        #真正的k
        real_k = k % total
        if real_k == 0:
            return head

        # p此时在链表末尾 将收尾相连
        p.next = head
        for _ in range(total-real_k):
            p = p.next

        newhead = p.next
        p.next = None
        return newhead
```

82. Remove Duplicates from Sorted List II

```
class Solution:
    def deleteDuplicates(self, head: Optional[ListNode]) -> Optional[ListNode]:
        #把有重复项的元素全部删掉
        #画图

        dummyhead = ListNode(next=head)
        pre = dummyhead
        cur = dummyhead.next
        while cur:
            while cur.next and cur.val == cur.next.val:

                cur = cur.next
```

```
            if pre.next == cur:
                pre = pre.next
            else:
                pre.next = cur.next

            cur = cur.next
        return dummyhead.next
```

```
# 递归
class Solution(object):
    def deleteDuplicates(self, head):
        if not head or not head.next:
            return head
        if head.val != head.next.val:
            head.next = self.deleteDuplicates(head.next)
        else:
            move = head.next
            while move and head.val == move.val:
                move = move.next
            return self.deleteDuplicates(move)
        return head
```

25. Reverse Nodes in k-Group

递归：对于有序的才可以

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseKGroup(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        # 递归
        if not head:
            return None

        a,b = head,head
        for _ in range(k):
            if not b:  #不够则不用翻转
                return head
            b = b.next
```

```
        #翻转a，b之间的元素
        newhead = self.reverse(a,b)
        a.next = self.reverseKGroup(b,k)
        return newhead

    #左闭右开的翻转
    def reverse(self,a,b):
        if not a:
            return None
        pre = None
        cur = a
        tmp = a
        while cur != b:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
        return pre
```

数组(array)

二分查找(Binary search algorithm)

704. Binary Search(模版)

```
# 普通二分查找
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
          #'left + right' may cause the Integer Overflow, meaning that left+right >
2147483647
            middle = left + (right-left) // 2
            if nums[middle] < target:
                left = middle + 1
            elif nums[middle] > target:
                right = middle - 1

            elif nums[middle]==target:
```

```
                return middle
        return -1

# 寻找左边界的二分查找
        def search_left(left,right):
            while left <= right:
                mid = left + (right-left)//2
                if nums[mid] < target:
                    left = mid + 1
                elif nums[mid] > target:
                    right = mid - 1
                elif nums[mid] == target: #区别
                    right = mid - 1
            if left>=len(nums) or nums[left] != target:  #若越界或匹配不到 返回-1
                return -1
            return left

# 寻找右边界的二分查找
        def search_right(left,right):
            while left <= right:
                mid = left + (right-left)//2
                if nums[mid] < target:
                    left = mid + 1
                elif nums[mid] > target:
                    right = mid - 1
                elif nums[mid] == target:#区别
                    left = mid + 1
            if right < 0 or nums[right]!=target: #若越界或匹配不到 返回-1
                return -1
            return right
```

注意：只有数组是单调函数才能用二分查找！

69. Sqrt(x)

```
```

```python
class Solution:
    def mySqrt(self, x: int) -> int:
        #二分查找 寻找左边界版
        l, r, ans = 0, x, -1
        while l <= r:
            mid = (l + r) // 2
            if mid * mid <= x:
                ans = l
                l = mid + 1
            else:
                r = mid - 1
        return ans
```

双指针(快慢指针 & 左右指针)

26. Remove Duplicates from Sorted Array

```python
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        #快慢指针，快指针比慢指针快1步
        slow = 0
        fast = 1
        while fast<len(nums):
            if nums[fast] != nums[slow]:
                nums[slow+1] = nums[fast]
                slow += 1
            fast += 1
        return slow+1 #返回数组长度
```

写法二

```python
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        j = 1
        for i in range(1, len(nums)):
            if nums[i]!=nums[j-1]:
                nums[j] = nums[i]
                j += 1
        return j
```

80. Remove Duplicates from Sorted Array II

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

27. Remove Element

由于题目要求删除数组中等于 val 的元素，因此输出数组的长度一定小于等于输入数组的长度，我们可以把输出的数组直接写在输入数组上。可以使用双指针：右指针 $right$ 指向当前将要处理的元素，左指针 $left$ 指向下一个将要赋值的位置。

- 如果右指针指向的元素不等于 val，它一定是输出数组的一个元素，我们就将右指针指向的元素复制到左指针位置，然后将左右指针同时右移；
- 如果右指针指向的元素等于 val，它不能在输出数组里，此时左指针不动，右指针右移一位。

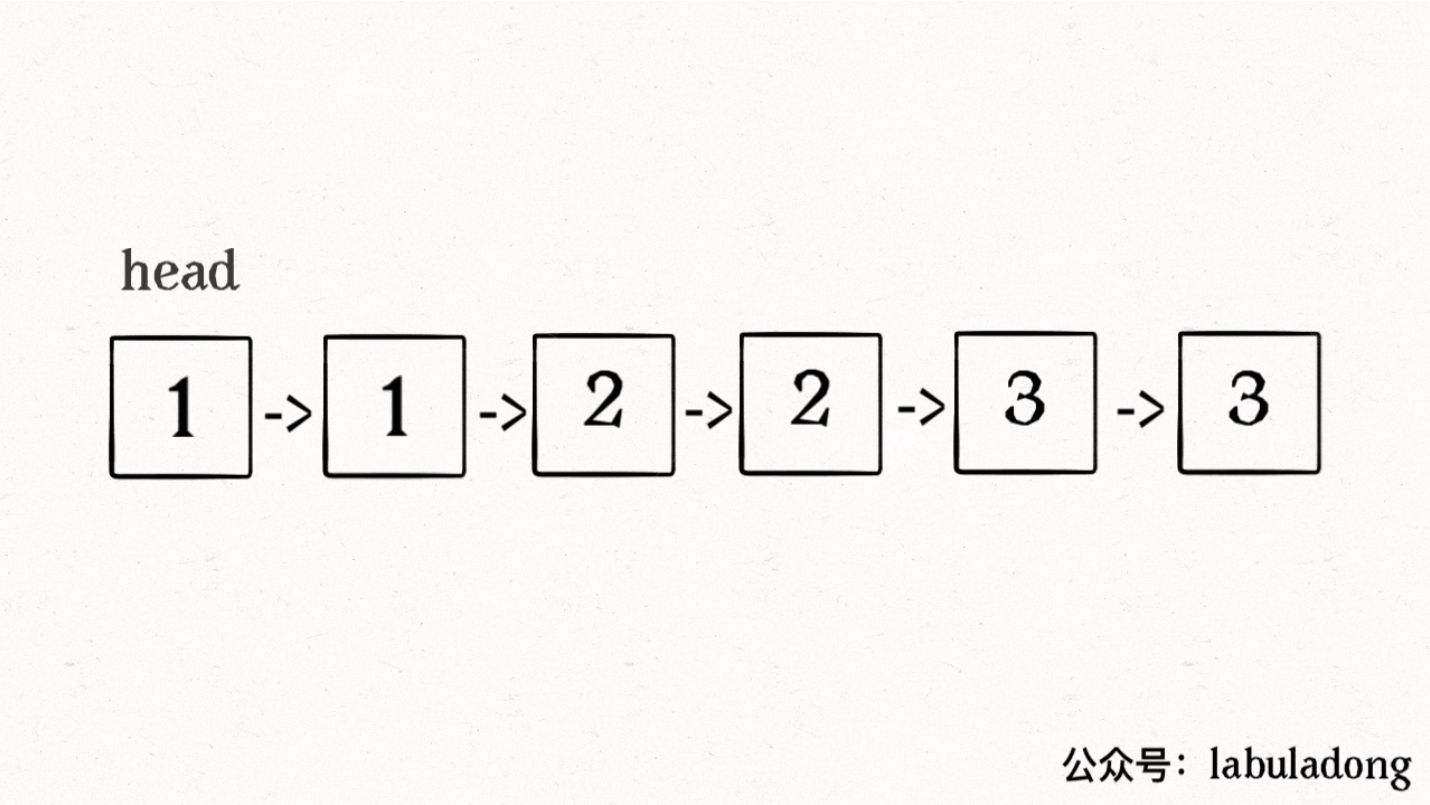
整个过程保持不变的性质是：区间 $[0, left)$ 中的元素都不等于 val。当左右指针遍历完输入数组以后，$left$ 的值就是输出数组的长度。

这样的算法在最坏情况下（输入数组中没有元素等于 val），左右指针各遍历了数组一次。

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

83. Remove Duplicates from Sorted List(同26，链表版)

把元素交换操作改成指针操作




```
class Solution:
    def deleteDuplicates(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head:
            return None
        slow, fast = head, head
        while fast:
            if fast.val != slow.val:
                slow.next = fast
                slow = slow.next
            fast = fast.next
        slow.next = None
        return head
```

283. Move Zeroes

```
class Solution(object):
  def moveZeroes(self, nums):
    """
    :type nums: List[int]

    :rtype: None Do not return anything, modify nums in-place instead.
```

```
    """
    if not nums:
      return 0
    # 两个指针i和j
    j = 0
    for i in range(len(nums)):
      # 当前元素!=0，就把其交换到左边，等于0就不动，相当于交换到右边
      if nums[i]!=0:
        nums[j],nums[i] = nums[i],nums[j]
        j += 1
```

75. Sort Colors

思想同283，只不过这题要使用3指针

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        #三指针
        if len(nums) < 2:
            return
        i = 0 #用来交换的
        j = len(nums)-1 #用来交换的
        k = 0  #用来遍历的
        #元素是0 换到左边 元素是2 换到右边
        while k<=j:
            if nums[k]==0:
                nums[k],nums[i] = nums[i],nums[k]
                i += 1
                k += 1
            elif nums[k]==1:
                k += 1
            else:
                nums[k],nums[j] = nums[j],nums[k]
                j -= 1
```

977. Squares of a Sorted Array

```python
class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        #思路一 先平方后排序 直接做
        #思路二 见代码 双指针

        i = 0
        j = len(nums)-1
        p = len(nums)-1
        res = [0] * len(nums)

        while i <= j:
            if abs(nums[i]) > abs(nums[j]):
                res[p] = nums[i] * nums[i]
                i += 1
            else:
                res[p] = nums[j] * nums[j]
                j -= 1
            p -= 1

        return res
```

88. Merge Sorted Array

```python
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        # 三指针
        # 从前面往后会导致覆盖，所以选择从后往前
        i = m-1
        j = n-1
        p = len(nums1)-1

        while i >= 0 and j >= 0:
            if nums1[i] > nums2[j]:
                nums1[p] = nums1[i]
                i -= 1
            else:
                nums1[p] = nums2[j]
                j -= 1
            p -= 1
```

```
        # 因为元素本身存放在nums1，只需考虑nums2是否遍历完
        while j >= 0:
            nums1[p] = nums2[j]
            j -= 1
            p -= 1
```

11. Container With Most Water

贪心问题：为什么要移动较低的那一边？因为移动会底边会减小，只有当高增大可以抵消边的缩小才能面积变大。所以移动较少的那边，尝试找到一个可以高到抵消到移动减小边的高度。

```
class Solution:
    def maxArea(self, height: List[int]) -> int:
        # 面积等于 (j-i) * min(i,j)
        res = -1
        left = 0
        right = len(height)-1
        while left < right:
            area = (right-left) * min(height[right],height[left])
            res = max(res,area)
            #移动较低一边，尝试找到比当前更高的
            if height[left] < height[right]:
                left += 1
            else:
                right -= 1
        return res
```

42. Trapping Rain Water

```
class Solution:
    def trap(self, height: List[int]) -> int:
        # 对于i 能装的水为 min(左边最高,右边最高)-height[i]
        # 双指针 边遍历边更新

        i = 0
        j = len(height) - 1
        max_left = 0
        max_right = 0
        res = 0
        while i < j:
            max_left = max(height[i],max_left)

            max_right = max(height[j],max_right)
```

```
            #res = min(max_left,max_right) - height[i]
            # 更新小的 思想同11
            #将上式分成两部
            if max_left < max_right:
                res += max_left - height[i]
                i += 1
            else:
                res += max_right - height[j]
                j -= 1
        return res
```

滑动窗口(Sliding window)

常用于字符串问题

所谓滑动窗口，**就是不断的调节子序列的起始位置和终止位置，从而得出我们要想的结果。**

滑动窗口算法的思路是这样：

1、我们在字符串 `S` 中使用双指针中的左右指针技巧，初始化 `left = right = 0`，把索引**左闭右开**区间 `[left, right)` 称为一个「窗口」。

> PS：理论上你可以设计两端都开或者两端都闭的区间，但设计为左闭右开区间是最方便处理的。因为这样初始化 `left = right = 0` 时区间 `[0, 0)` 中没有元素，但只要让 `right` 向右移动（扩大）一位，区间 `[0, 1)` 就包含一个元素 `0` 了。如果你设置为两端都开的区间，那么让 `right` 向右移动一位后开区间 `(0, 1)` 仍然没有元素；如果你设置为两端都闭的区间，那么初始区间 `[0, 0]` 就包含了一个元素。这两种情况都会给边界处理带来不必要的麻烦。

2、我们先不断地增加 `right` 指针扩大窗口 `[left, right)`，直到窗口中的字符串符合要求（包含了 `T` 中的所有字符）。

3、此时，我们停止增加 `right`，转而不断增加 `left` 指针缩小窗口 `[left, right)`，直到窗口中的字符串不再符合要求（不包含 `T` 中的所有字符了）。同时，每次增加 `left`，我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 `right` 到达字符串 `S` 的尽头。

这个思路其实也不难，**第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解**，也就是最短的覆盖子串。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动，这就是「滑动窗口」这个名字的来历。

所需要的变量：

哈希表`needs` 和 `window` 相当于计数器，分别记录 `T` (要匹配的字符)中字符出现次数和「窗口」中的相应字符的出

现次数。

valid : 合法字符数量。当window['a']达到某个数值时，valid+=1

left，right： 左右指针，用于全局遍历

start，length：用于记录符合条件的区间【start: start+length】

现在开始套模板，只需要思考以下几个问题：

1、什么时候应该移动 `right` 扩大窗口？窗口加入字符时，应该更新哪些数据?

2、什么时候窗口应该暂停扩大，开始移动 `left` 缩小窗口？从窗口移出字符时，应该更新哪些数据?

3、我们要的结果应该在扩大窗口时还是缩小窗口时进行更新?

如果一个字符进入窗口，应该增加 `window` 计数器；如果一个字符将移出窗口的时候，应该减少 `window` 计数器；当 `valid` 满足 `need` 时应该收缩窗口；应该在收缩窗口的时候更新最终结果。

76. Minimum Window Substring(模版)

```python
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if len(s) < len(t):
            return ''
        need = defaultdict(int)
        for char in t:
            need[char] += 1
        window = defaultdict(int) #统计窗口内每个字符串的个数
        left,right = 0,0
        valid = 0 #合法字符数量 如need[a] = 3,当window[a] = 3时,  valid +=1
        start, length = 0, inf
        while right < len(s):
            c = s[right]  #c是移入窗口的字符
            right += 1    #扩大窗口
            #如果进去的是需要的字符，就进行更新(哈希表window和合法字符数量valid的更新)
            if need.get(c):
                window[c] += 1
                if window[c] == need[c]:
                    valid += 1

            # 判断左边窗口是否要收缩
            while valid == len(need): #收缩窗口的条件(全部字符都满足要求了)
                #更新start和length
                if right - left < length:
                    start = left

                    # 由于每次扩大窗口都对right+1了 所以此时right是目标区间的后一个元素下标
```

```
                    # 对应切片左闭右开 此处不用+1
                    length = right - left

                #缩小窗口
                d = s[left]
                left += 1
                #如果出去的是需要的字符，就进行更新(哈希表window和合法字符数量valid的更新)
                if need.get(d):
                    if window[d] == need[d]:
                        valid -=1
                    window[d] -= 1

        return '' if length == inf else s[start:start+length]
```

3. Longest Substring Without Repeating Characters

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        #哈希表记录滑动窗口内子串
        window = defaultdict(int)
        start, length = 0, 0
        left,right = 0, 0
        res = 0

        while right < len(s):
            #先扩大窗口再移入
            c = s[right]
            right += 1
            window[c] += 1
            #收缩
            while window[c] > 1:
                d = s[left]
                left += 1
                window[d] -= 1
            res = max(res,right-left)

        return res
```

\438. Find All Anagrams in a String

```python
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        need = defaultdict(int)
        window = defaultdict(int)
        res = []
        start = 0
        left,right = 0, 0
        valid = 0
        for ss in p:
            need[ss] += 1

        while right < len(s):
            c = s[right]
            right += 1
            if need.get(c):
                window[c] += 1
                if window[c] == need[c]: #字符在窗口出现的次数和需要的次数一样 valid才加
                    valid += 1
            while right - left >= len(p): #左边收缩条件
                if valid == len(need):
                    res.append(left)
                d = s[left]
                left += 1
                #注意和右边扩展有区别
                #先-valid再-window
                if need.get(d):
                    if window[d] == need[d]:
                        valid -= 1
                    window[d] -= 1
        return res
```

\567. Permutation in String

基本同438

```python
class Solution:
    def checkInclusion(self, s1: str, s2: str) -> bool:
        left, right = 0, 0
        need = defaultdict(int)
        for s in s1:
            need[s] += 1
        window = defaultdict(int)

        valid = 0
```

```
        res = False
        while right < len(s2):
            c = s2[right]
            right += 1
            if need.get(c):
                window[c] += 1
                if window[c] == need[c]:
                    valid += 1
            while right - left >= len(s1):
                if valid == len(need):
                    return True
                d = s2[left]
                left += 1
                if need.get(d):
                    if window[d] == need[d]:
                        valid -= 1
                    window[d] -= 1
        return False
```

\239. Sliding Window Maximum

最直接的做法，但是超时

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        left, right = 0, k-1
        res = []
        while right < len(nums):
            max_value = max(nums[left:right+1])
            res.append(max_value)
            right += 1
            left += 1
        return res
```

使用单调队列:**所有队列里的元素都是按递增（递减）的顺序队列，这个队列的头是最小（最大）的元素。**

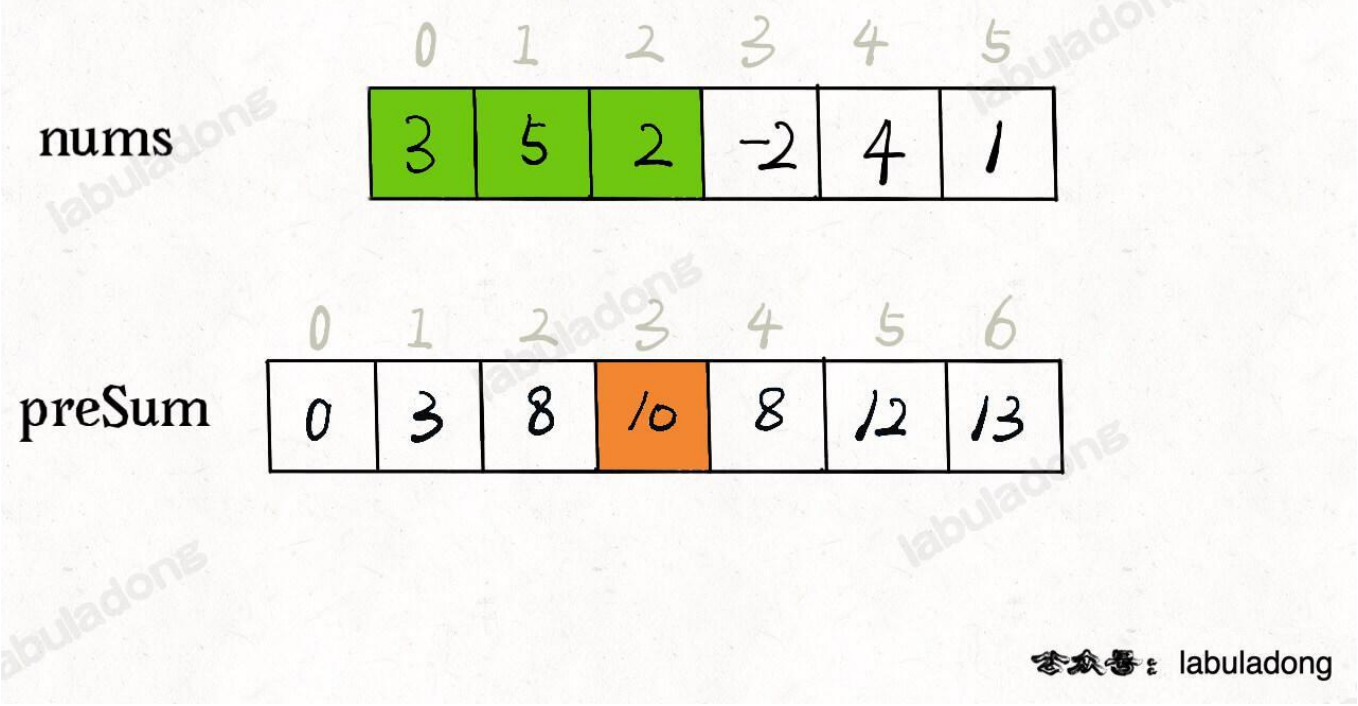
队列储存的是数组的下标！！！

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        # 初始化队列和结果，队列存储数组的下标

        queue = []
```

```
        res = []

        for i in range(len(nums)):
            # 如果当前队列最左侧存储的下标等于 i-k 的值，代表目前队列已满。
            # 但是新元素需要进来，所以列表最左侧的下标出队列
            if queue and queue[0] == i - k:
                queue.pop(0)

            # 对于新进入的元素，如果队列前面的数比它小，那么前面的都出队列
            # 目的是维护第一个数永远是最大的（单调队列的定义）
            while queue and nums[queue[-1]] < nums[i]:
                queue.pop()
            # 新元素入队列,进来的是下标
            queue.append(i)

            # 当前的大值加入到结果数组中
            if i >= k-1:
                res.append(nums[queue[0]])

        return res
```

前缀和数组(Prefix sum)

前缀和技巧适用于快速、频繁地计算一个索引区间内的元素之和。



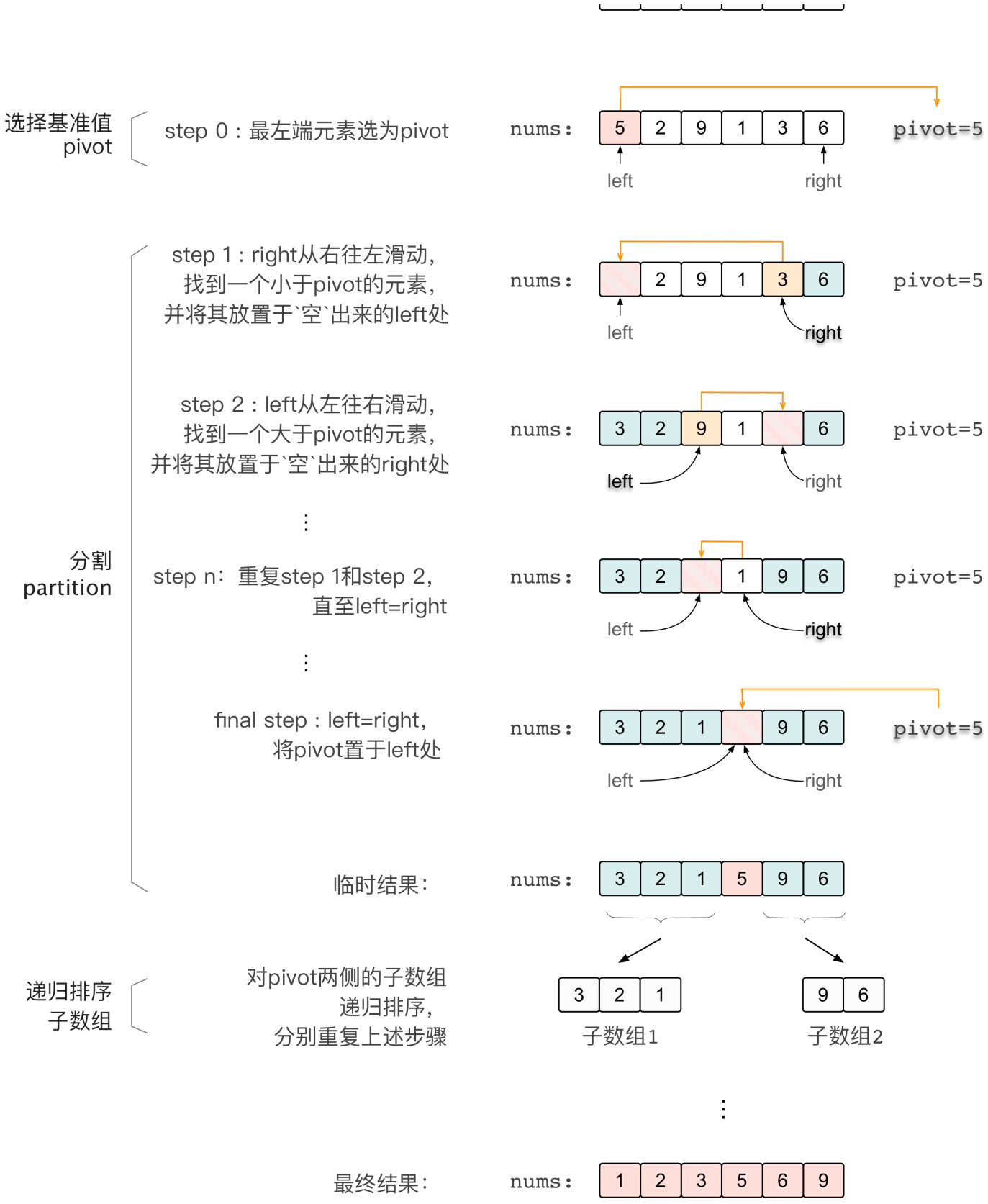
303. Range Sum Query - Immutable

```python
class NumArray:

    def __init__(self, nums: List[int]):
        # self.preSum里先放一个0，即列表里总共放n+1个元素
        self.preSum = [0]
        for num in nums:
            self.preSum.append(self.preSum[-1] + num)

    def sumRange(self, left: int, right: int) -> int:
        # 查询闭区间的累加和
        # 在self.preSum里，index=right + 1时，是加了index=right这个元素的值
        # 因此闭区间[left, right]的累加和等于下面
        return self.preSum[right + 1] - self.preSum[left]
```

排序

快速排序

nums: | 5 | 2 | 9 | 1 | 3 | 6 |




```
#quick_sort

class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
```

```
        def partition(arr, low, high):
            pivot = arr[low]  #固定pivot
            left = low
            right = high
            while left < right:
                #从右边往左边找到第一个小于pivot的元素 放在left位置
                while left<right and arr[right] >= pivot:   #注意这里是大于等于
                    right -= 1
                arr[left] = arr[right]
                #从左边往右边找到第一个大于pivot的元素 放在right位置
                while left<right and arr[left] <= pivot:    #注意这里是小于等于
                    left += 1
                arr[right] = arr[left]
            #最终将pivot元素放在中间 即left==right处
            arr[left] = pivot
            return left

        def random_select_pivot(arr,low,high):
            pivot_idx = random.randint(low,high)
            arr[low], arr[pivot_idx] = arr[pivot_idx], arr[low]
            return partition(arr,low,high)

        def quicksort(arr,low,high):
            if(low >= high): #注意这里是大于等于
                return
            #mid = partition(arr,low,high)
            mid = random_select_pivot(arr,low,high)
            quicksort(arr,low,mid-1)
            quicksort(arr,mid+1,high)

        quicksort(nums,0,len(nums)-1)
        return nums

# quick selection 快速选择(用于topK split问题)
#quick select, find a pivot, move all nums that is smaller than it to it's left, all nums
> pivot to it's right. check how many on the left. if count == k, return, if count > k, it
must be on the left, otherwise on the right.
```

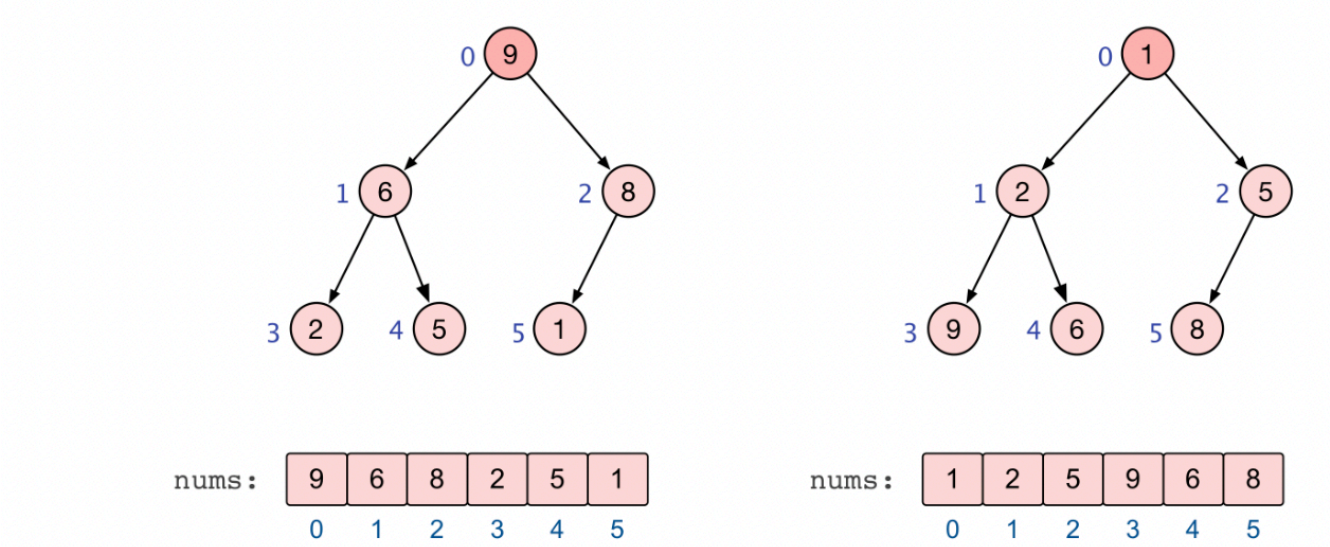
归并排序

```python
# merge sort
#思想比较简单
class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        def merge_sort(arr,low,high):
            if low >= high:
                return
            mid = low + (high-low)//2
            merge_sort(arr,low,mid) #右边到mid
            merge_sort(arr,mid+1,high)

            left, right = low, mid+1
            tmp = [] #记录结果
            while left <= mid and right <= high:
                if arr[left] <= arr[right]:
                    tmp.append(arr[left])
                    left += 1
                else:
                    tmp.append(arr[right])
                    right += 1

            #比较完了 还没放入tmp的直接放入
            while left <= mid:
                tmp.append(arr[left])
                left += 1

            while right <= high:
                tmp.append(arr[right])
                right += 1

            arr[low:high+1] = tmp #注意是浅复制

        merge_sort(nums,0,len(nums)-1)
        return nums
```

堆排序

- 基本定义:大根堆和小根堆

大根堆　　小根堆



- 大根堆 / 大顶堆：每个节点的值均大于等于其左右孩子节点的值；
- 小根堆 / 小顶堆：每个节点的值均小于等于其左右孩子节点的值。

- 如何进行堆排序?

\215. Kth Largest Element in an Array

Hash Table

- 用数组作为哈希表
- 用dict作为哈希表{key:value}

python中： collections.defaultdict 和 collections.Counter的使用

https://docs.python.org/3/library/collections.html#counter-objects

242. Valid Anagram

用两个dict，分别插入字符串，比较两个dict是否相等即可。

```
class Solution:
```

```
    def isAnagram(self, s: str, t: str) -> bool:
        a = defaultdict(int)
        b = defaultdict(int)
        for char in s:
            a[char] += 1
        for char in t:
            b[char] += 1
        if a == b:
            return True
        return False
```

1002. Find Common Characters

难点：

1. Counter 的定义和用法

```
from collections import Counter
s = 'evergfsafcage'
h = Counter(s)
print(h)
#Counter({'e': 3, 'g': 2, 'f': 2, 'a': 2, 'v': 1, 'r': 1, 's': 1, 'c': 1})

#elements()
print(list(h.elements()))

#['e', 'e', 'e', 'v', 'r', 'g', 'g', 'f', 'f', 's', 'a', 'a', 'c']
```

2. python中集合取交集的取法: &=

```
class Solution:
    def commonChars(self, words: List[str]) -> List[str]:
        res = Counter(words[0])
        for i in words:
            res &= Counter(i)
        return list(res.elements())
```

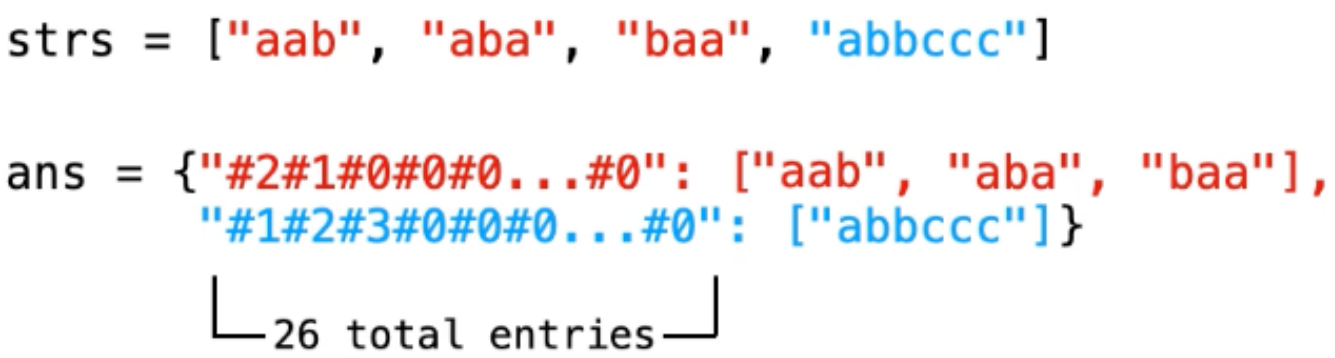
349. Intersection of Two Arrays

```python
class Solution:
    def intersection(self, nums1: List[int], nums2: List[int]) -> List[int]:

        return set(nums1) & set(nums2)
```

49. Group Anagrams

哈希表的使用

将每个字母出现的次数使用字符串表示，作为哈希表的键




```python
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        mp = defaultdict(list)
        for s in strs: #遍历字符串
            count = [0] * 26
            for i in s: #遍历每个字符
                count[ord(i)-ord('a')] += 1
            mp[tuple(count)].append(s) #需要将 list 转换成 tuple 才能进行哈希
        return list(mp.values())
```

73. Set Matrix Zeroes

集合的使用，比较简单粗暴

注意第二次循环使用的技巧 in

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

```
        #先找到所有0的下标 在删
        zero_row = set()
        zero_col = set()
        m = len(matrix)
        n = len(matrix[0])
        for i in range(m):
            for j in range(n):
                if matrix[i][j]==0:
                    zero_row.add(i)
                    zero_col.add(j)

        for i in range(m):
            for j in range(n):
                if i in zero_row or j in zero_col:
                    matrix[i][j] = 0
```

N sum 问题

1. Two Sum

难点

1. 使用 `for idx,val in enumerate(num):` 同时遍历编号和数值
2. 一边遍历一边加入字典

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        #遍历列表同时查字典

        #字典{数值：编号} 因为要输出的是编号
        res = dict()
        for idx,val in enumerate(nums):
            tar = target - val
            if tar in res:
                return [res[tar],idx]
            else:
                res[val] = idx
```

本题还可以用双指针做,但是排序会造成序号的改变, 所以我们需要记住排序中记住序号

```
# 记住当前序号并排序
```

```
nums = sorted(enumerate(nums), key= lambda x: x[1])
# res:  [(4, 1), (3, 2), (1, 3), (0, 4), (2, 6)]
```

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        return self.solve(nums, target)

    def solve(self, nums, target):
        nums = sorted(enumerate(nums), key= lambda x: x[1])
        l, r = 0, len(nums)-1

        while l < r:
            twosum = nums[l][1] + nums[r][1]
            if twosum == target:
                return [nums[l][0], nums[r][0]]
            elif twosum < target:
                l += 1
            elif twosum > target:
                r -= 1
```

15. 3Sum

关键在于如何去重：无论是对于nums[i],left,right，要是当前的元素和前一个元素相等，直接跳过！

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        #双指针
        res = []
        nums.sort()
        for i in range(len(nums)):
            #left 和 right需要写在循环里面，因为是不断变化的
            left = i+1
            right = len(nums)-1
            if nums[i]>0: #left和right都在nums[i]右边
                break
            if i >= 1 and nums[i] == nums[i - 1]:  # 当前元素和前一个元素相等，会出现重复，跳过
                continue
            while left<right:
                s = nums[i]+nums[left]+nums[right]
                if s < 0:
                    left += 1
                elif s > 0:
                    right -= 1
                else:
                    res.append([nums[i],nums[left],nums[right]])
                    #去重 无论是left和right，要是当前元素和前一个元素相等，都跳过
```

```
                    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
```

16. 3Sum Closest

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        #不用去重
        res = inf
        nums.sort()
        for i in range(len(nums)):
            left = i+1
            right = len(nums)-1
            while left < right :
                cur_sum = nums[i] + nums[left] + nums[right]
                if abs(target - cur_sum) < abs(target - res):
                    res = cur_sum
                if cur_sum < target:
                    left += 1
                elif cur_sum > target:
                    right -= 1
                else:
                    return res
        return res
```

18. 4Sum

```
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        #照样是双指针
        #比3sum增加一层循环
        res = []
        nums.sort()
        n = len(nums)
        for i in range(n):
            #去重
            if i>0 and nums[i] == nums[i-1]:
                continue
            for j in range(i+1,n):
```

```
                #去重
                if j>i+1 and nums[j] == nums[j-1]:
                    continue

                left = j+1
                right = n-1

                while left < right:
                    s = nums[i] + nums[j] + nums[left] + nums[right]
                    if s < target:
                        left += 1
                    elif s > target:
                        right -= 1
                    else:
                        res.append([nums[i],nums[j],nums[left],nums[right]])
                        #去重
                        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
```

字符串(string)

python 常用字符串方法(纯字符串题目)

对于字符串s

```
#1. 翻转
s = s[::-1]

#2. 切片

#3. 先用list变成列表，再用''.join(s)恢复成字符串(如151题)
res = list(i for i in s.split(' ') if len(i)>0)

#4. 替换(用xx替换空格或类似的)
'xx'.join(s.split(' '))

#5. 先局部翻转再全部翻转 & 先整体翻转再局部翻转

#6. KMP算法（比较难）
```

344. Reverse String

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

最简单的方法如下：

需要注意的需要用s[:]浅拷贝。

浅拷贝是在另一块地址中创建一个新的变量或容器，但是容器内的元素的地址均是源对象的元素的地址的拷贝。也就是说新的容器中指向了旧的元素（新瓶装旧酒）。

深拷贝是在另一块地址中创建一个新的变量或容器，同时容器内的元素的地址也是新开辟的，仅仅是值相同而已，是完全的副本。也就是说（新瓶装新酒）。

```
class Solution:
    def reverseString(self, s: List[str]) -> None:
        """
        Do not return anything, modify s in-place instead.
        """
        s[:] = s[::-1]
```

541. Reverse String II

使用上一题的函数进行翻转

要注意的是把字符串弄成列表操作，然后再把它变回字符串。

注意range的第三个参数是步长

```
class Solution:

    def reverseStr(self, s: str, k: int) -> str:
        def reverse_substring(text):
```

```
            left, right = 0, len(text) - 1
            while left < right:
                text[left], text[right] = text[right], text[left]
                left += 1
                right -= 1
            return text

        res = list(s)

        for cur in range(0, len(s), 2 * k):
            # 如果k是2，则cur分别是0,2,4,6，。。。
            res[cur: cur + k] = reverse_substring(res[cur: cur + k])

        return ''.join(res)
```

151. Reverse Words in a String

```
class Solution:
    def reverseWords(self, s: str) -> str:
        #转成列表
        #翻转列表
        #恢复成字符串
        res = list(i for i in s.split(' ') if len(i)>0)
        res = res[::-1]
        return ' '.join(res)
```

栈和队列(stack and queue)

20. Valid Parentheses

非常经典的使用栈的问题！

思路：遍历字符串，当遇到左括号时，往栈中插入相应的右括号；如果遇到右括号，比较此时栈中最顶元素是否为同一个右括号，不是直接返回false，此外，当字符串还没被遍历完，栈已经空了，说明连续出现了两个一样的右括号，返回false。当字符串被遍历完时，如果栈也为空，则返回true，否则返回false。

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []
        for item in s:

            if item == '(':
                stack.append(')')
```

```
            elif item == '[':
                stack.append(']')
            elif item == '{':
                stack.append('}')
            elif not stack or item != stack[-1]:
                return False
            else:
                stack.pop()
        return True if not stack else False
```

1047. Remove All Adjacent Duplicates In String

```
class Solution:
    def removeDuplicates(self, s: str) -> str:
        #对对碰
        stack = []
        for item in s:
            if stack and item == stack[-1]:
                stack.pop()
            else:
                stack.append(item)

        return ''.join(stack)
```

150. Evaluate Reverse Polish Notation

Matrix模拟

48. Rotate Image

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        # 有点智力题的意思
        # 先上下翻转 再沿着45度对角线翻转
        row = len(matrix)

        col = len(matrix[0])
```

```
        # 上下翻转
        i = 0
        j = row-1
        while i<j:
            matrix[i][:], matrix[j][:] = matrix[j][:], matrix[i][:]
            i += 1
            j -= 1

        # 对角线
        for i in range(row):
            for j in range(col):
                if i==j:
                    break
                matrix[i][j],matrix[j][i] = matrix[j][i], matrix[i][j]

        return matrix
```

54. Spiral Matrix

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        #设计上下左右边界 当初碰到边界时
        row = len(matrix)
        col = len(matrix[0])

        left_bound = 0
        right_bound = col-1
        up_bound = 0
        down_bound = row-1

        nums = []
        while len(nums) < row*col:
            # 在最上边从左到右
            if up_bound <= down_bound:
                for j in range(left_bound,right_bound+1):
                    nums.append(matrix[up_bound][j])
                up_bound += 1

            # 在最右边从上到下
            if right_bound >= left_bound:
                for j in range(up_bound,down_bound+1):
                    nums.append(matrix[j][right_bound])
                right_bound -= 1
```

```
            # 在最下面从右到左
            if down_bound >= up_bound:
                for j in range(right_bound,left_bound-1,-1):
                    nums.append(matrix[down_bound][j])
                down_bound -= 1

            #在最左边从下到上
            if left_bound <= right_bound:
                for j in range(down_bound,up_bound-1,-1):
                    nums.append(matrix[j][left_bound])
                left_bound += 1

        return nums
```

59. Spiral Matrix II

54的代码稍微修改即可

```
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        #和54题同理 这次是填入数字


        matrix = [[0 for _ in range(n)] for _ in range(n)]

        left_bound = 0
        right_bound = n-1
        up_bound = 0
        down_bound = n-1

        nums = 1
        while nums <= n*n:
            # 在最上边从左到右
            if up_bound <= down_bound:
                for j in range(left_bound,right_bound+1):
                    matrix[up_bound][j] = nums
                    nums += 1
                up_bound += 1

            # 在最右边从上到下
            if right_bound >= left_bound:
                for j in range(up_bound,down_bound+1):

                    matrix[j][right_bound] = nums
                    nums += 1
```

```
                right_bound -= 1

            # 在最下面从右到左
            if down_bound >= up_bound:
                for j in range(right_bound,left_bound-1,-1):
                    matrix[down_bound][j] = nums
                    nums += 1
                down_bound -= 1

            #在最左边从下到上
            if left_bound <= right_bound:
                for j in range(down_bound,up_bound-1,-1):
                    matrix[j][left_bound] = nums
                    nums += 1
                left_bound += 1

        return matrix
```

73. Set Matrix Zeroes

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

        #先找到所有0的下标 在删
        zero_row = set()
        zero_col = set()
        m = len(matrix)
        n = len(matrix[0])
        for i in range(m):
            for j in range(n):
                if matrix[i][j]==0:
                    zero_row.add(i)
                    zero_col.add(j)

        for i in range(m):
            for j in range(n):
                if i in zero_row or j in zero_col:
                    matrix[i][j] = 0
```

74. Search a 2D Matrix

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # 有序 想到二分查找 可以转成1维搜
        # 也可以从右上角开始搜
        row = 0
        col = len(matrix[0])-1
        while row <= len(matrix)-1 and col >= 0:
            if matrix[row][col] < target:
                row += 1
            elif matrix[row][col] > target:
                col -= 1
            elif matrix[row][col] == target:
                return True
        return False
```

240. Search a 2D Matrix II

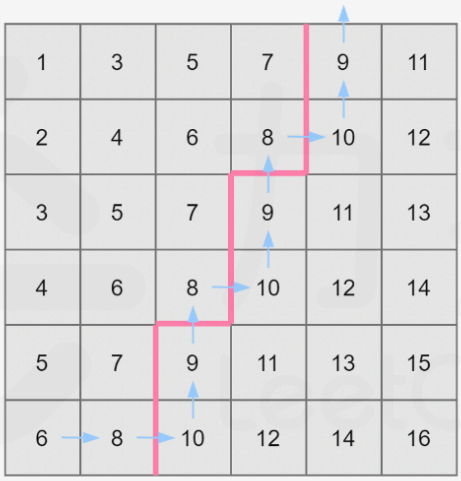
```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        # 每一行递增
        # 每一列递增

        # 还是从右上角开始遍历 大了往左移动 小了往右移动
        row = 0
        col = len(matrix[0])-1
        while row < len(matrix) and col >= 0:
            if matrix[row][col] > target:
                col -= 1
            elif matrix[row][col] < target:
                row += 1
            elif matrix[row][col] == target:
                return True
        return False
```

378. Kth Smallest Element in a Sorted Matrix

215题是一维，这里是二维

思想：和74,240差不多，也是从左下或者右上开始走

走法演示如下，依然取 $mid = 8$：



可以这样描述走法：

- 初始位置在 $matrix[n-1][0]$（即左下角）；
- 设当前位置为 $matrix[i][j]$。若 $matrix[i][j] \leq mid$，则将当前所在列的不大于 mid 的数的数量（即 $i+1$）累加到答案中，并向右移动，否则向上移动；
- 不断移动直到走出格子为止。

我们发现这样的走法时间复杂度为 $O(n)$，即我们可以线性计算对于任意一个 mid，矩阵中有多少数不大于它。这满足了二分查找的性质。

不妨假设答案为 x，那么可以知道 $l \leq x \leq r$，这样就确定了二分查找的上下界。

每次对于「猜测」的答案 mid，计算矩阵中有多少数不大于 mid：

- 如果数量不少于 k，那么说明最终答案 x 不大于 mid；
- 如果数量少于 k，那么说明最终答案 x 大于 mid。

这样我们就可以计算出最终的结果 x 了。

```python
class Solution:
    def kthSmallest(self, matrix: List[List[int]], k: int) -> int:
        n = len(matrix)


        def check(mid):
            # nums >= k 目标值<=k
            # nums < k  目标值>k
```

```
        # nums < k ,目标值>k
        i, j = n - 1, 0
        num = 0
        while i >= 0 and j < n:
            if matrix[i][j] <= mid:
                num += i + 1
                j += 1
            else:
                i -= 1
        return num >= k

    left, right = matrix[0][0], matrix[-1][-1]
    while left <= right:
        mid = (left + right) // 2
        if check(mid):
            right = mid - 1
        else:
            left = mid + 1

    return left
```