面试中的算法问题，有很多并不需要复杂的数据结构支撑。就是用数组，就能考察出很多东西了。其实，经典的排序问题，二分搜索等等问题，就是在数组这种最基础的结构中处理问题的，今天主要介绍 LeetCode 中典型的数组类问题，主要介绍这类问题的一些常用解法：做好初始定、基础算法思想应用、对撞指针、滑动窗口法等。

总结：array这类题目，需要考虑巧妙设计指针，即指针的意义是什么(非零值，最后一个不同的值，目前出现最小的值)，以及指针的摆放位置(两个指针均从头开始滑动，或者一头一尾相向滑动)，以及指针的移动方式（指针在什么条件下移动，一般有一个指针做遍历，另一个指针根据条件移动）。另外还顺便了解了两个NlogN的排序算法，归并排序(middle, mergesort(left, middle-1), mergesort(middle+1,right), merge two order array)，以及快速排序算法(pi = partition, quicksort(pi+1, right), quicksort(left, pi -1))。

考点一

如何利用双指针来对数组中的元素进行剔除。

做数组类算法问题的时候，我们常常需要定义一个变量，明确该变量的定义，并且在书写整个逻辑的时候，要不停的维护住这个变量的意义。也特别需要注意初始值和边界的问题。

283. Move Zeroes

Easy

Given an array `nums`, write a function to move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Example:

```
Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Note:

1. You must do this **in-place** without making a copy of the array.
2. Minimize the total number of operations.

思路：

利用双指针问题，一个指针a用来遍历数组，另外一个指针b在遍历时，用来记录最后一个非零元素的位置。因为a>=b, 遍历时，遇到非零元素，交换数组两个位置的值即可把非零元素置前。

解题：

```
class Solution:
    def moveZeroes(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """

        NonZeroPoint = 0


        for CurPoint in range(len(nums)):
            if nums[CurPoint] != 0:
                nums[CurPoint], nums[NonZeroPoint] = nums[NonZeroPoint], nums[CurPoint]
                NonZeroPoint = NonZeroPoint + 1
```

283. Remove Element

Easy

1. Given an array *nums* and a value *val*, remove all instances of that value in-place and return the new length.

   Do not allocate extra space for another array, you must do this by **modifying the input array in-place** with O(1) extra memory.

   The order of elements can be changed. It doesn't matter what you leave beyond the new length.

   Example 1:

   ```
   Given nums = [3,2,2,3], val = 3,

   Your function should return length = 2, with the first two elements of nums being
   2.

   It doesn't matter what you leave beyond the returned length.
   ```

   Example 2:

   ```
   Given nums = [0,1,2,2,3,0,4,2], val = 2,

   Your function should return length = 5, with the first five elements of nums
   containing 0, 1, 3, 0, and 4.

   Note that the order of those five elements can be arbitrary.

   It doesn't matter what values are set beyond the returned length.
   ```

思路：

可以像上一个题目一样，只不过这里的零变成了某个特定的元素罢了

解题：

```
class Solution:
    def removeElement(self, nums, val):
        """
        :type nums: List[int]
        :type val: int
        :rtype: int
        """

        nz_point = 0
        for cur_point in range(len(nums)):
            if nums[cur_point] != val:
                nums[cur_point], nums[nz_point] = nums[nz_point], nums[cur_point]
                nz_point = nz_point + 1
        return nz_point
```

更为简洁的思路：

```
```

26. Remove Duplicates from Sorted Array

Easy

Given a sorted array *nums*, remove the duplicates in-place such that each element appear only *once* and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array in-place** with O(1) extra memory.

Example 1:

```
Given nums = [1,1,2],

Your function should return length = 2, with the first two elements of nums being 1 and
2 respectively.

It doesn't matter what you leave beyond the returned length.
```

Example 2:

```
Given nums = [0,0,1,1,1,2,2,3,3,4],

Your function should return length = 5, with the first five elements of nums being
modified to 0, 1, 2, 3, and 4 respectively.

It doesn't matter what values are set beyond the returned length.
```

思路：

还是利用双指针，不同点在于我们对于第二个指针unique_point的累加方式变得不同了。具体累加方式可见下算法中，我们这里的判断条件由if nums[cur_point] != val: 变为if nums[cur_point] != last_value:。 last_value记录了上个不同的值，因此只有当出现新的不同的值，我们才会数组操作，对值进行交换。

解题：

```python
 class Solution:
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        unique_point = 0
        last_value = None

        for cur_point in range(len(nums)):
            if nums[cur_point] != last_value:
                last_value = nums[cur_point]
                nums[cur_point], nums[unique_point] = nums[unique_point],
nums[cur_point]
                unique_point = unique_point + 1
        return unique_point
```

更为简洁的方案：

```
class Solution:
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        i = 0

        for n in nums:
            if i <1 or n > nums[i-1]:
                nums[i] = n
                i = i +1
        return i
```

80. Remove Duplicates from Sorted Array II

Medium

Given a sorted array *nums*, remove the duplicates in-place such that duplicates appeared at most *twice* and return the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array in-place** with O(1) extra memory.

Example 1:

```
Given nums = [1,1,1,2,2,3],

Your function should return length = 5, with the first five elements of nums being 1,
1, 2, 2 and 3 respectively.

It doesn't matter what you leave beyond the returned length.
```

Example 2:

```
Given nums = [0,0,1,1,1,1,2,3,3],

Your function should return length = 7, with the first seven elements of nums being
modified to 0, 0, 1, 1, 2, 3 and 3 respectively.

It doesn't matter what values are set beyond the returned length.
```

思路：

仍然是一个双指针问题。快指针对数组进行正常遍历，慢指针记录修改数组的位置。在前边，慢指针的移动方式是：根据要求不能有重复的值，如果当前快指针与满指针指的值不同，则慢指针向前移动，并触发修改数组的方式。

而本题中，要求中是每个数字最多出现两次，则慢指针移动方式为： 快指针与慢指针所指的值不同 or times<2。其中times<2的要求是为了满足，除了出现新的数字慢指针需要移动，当一个数字出现的次数小于两次时，慢指针也需要移动。

解题：

```python
class Solution:
    def removeDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if nums == []:
            return None


        slow_point = 0
        times = 0
        last_value = None
        for cur_point in range(0,len(nums)):
            # print(nums[cur_point], last_value, nums[slow_point], times)
            if last_value != nums[cur_point]:
                last_value = nums[cur_point]
                nums[slow_point] = nums[cur_point]
                slow_point = slow_point + 1
                times = 1
                # print('_', last_value, nums[cur_point], times)

            elif last_value == nums[cur_point] and times < 2:
                nums[slow_point] = nums[cur_point]
                slow_point = slow_point + 1
                times = times +1
                # print(nums[cur_point], times)

        return slow_point
```

更为简洁的方案：仍然是两个point，只是判断条件在这里变得异常简洁。因为允许一个数字重复两次，所以前两个step，我们直接移动两个pointer。因为允许一个数字重复两次，所以只要 n > nums[i-2] 等价于n != nums[i-2]，说明当前数字n没有出现过两次，可以同时移动两个pointer。

```
class Solution:
    def removeDuplicates(self, nums):
        i = 0
        for n in nums:
            if i < 2 or n > nums[i-2]:
                nums[i] = n
                i += 1
        return i
```

考点二

典型的排序算法思想、二分查找思想在解 LeetCode 题目时很有用。

75. Sort Colors

Medium

Given an array with *n* objects colored red, white or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note: You are not suppose to use the library's sort function for this problem.

Example:

```
Input: [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Follow up:

- A rather straight forward solution is a two-pass algorithm using counting sort. First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.
- Could you come up with a one-pass algorithm using only constant space?

思路

一共只会有三种颜色，而我们的目的是为了让所有的0都在这个数组的最左边，所有的2都在数组的最右边。所有的1在中间，我们可以通过两个指针来完成和一个迭代器来完成这样的功能。一个指针负责指向数组左边最后一个0的位置，另外一个指针指向数组右边最靠前的一个2的位置。其中第一个指针的用法，类似于第一个考点中，用一个指针记录我们想要输出的位置。本题目中因为要考虑数组的头和尾位置的元素，所以考虑使用两个指针。

这种方法好像也叫做三路快速排序方法。

解题

```
class Solution:
    def sortColors(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        start = 0
        end = len(nums) -1

        iter = 0

        while iter <=end:
            if nums[iter] == 0:
                nums[iter], nums[start] = nums[start], nums[iter]
                iter = iter + 1
                start = start +1
            elif nums[iter] == 2:
                nums[iter], nums[end] = nums[end], nums[iter]
                end = end - 1
            else:
                iter = iter + 1
```

215. Kth Largest Element in an Array

Medium

- Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

  Example 1:

  ```
  Input: [3,2,1,5,6,4] and k = 2
  Output: 5
  ```

  Example 2:

  ```
  Input: [3,2,3,1,2,4,5,5,6] and k = 4
  Output: 4
  ```

  Note: You may assume k is always valid, 1 ≤ k ≤ array's length.

思路

参考思路：https://leetcode.com/problems/kth-largest-element-in-an-array/discuss/167837/Python-or-tm

如果这个array的大小很小，我们可以尝试使用快速排序对数组进行排序，这一步的操作为 $Time: O(nlogn) | Space: O(1)$，之后再从排序好的数组中取出对应数即可，所以关键在于排序，快速排序。geek

方案一：所以如下为基于快速排序的实现方法：partition + quickSort + quickSort

```
class Solution:
    def findKthLargest(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        self.quickSort(nums, 0, len(nums) -1)
        # nums.sort()
        return nums[-k]


    def quickSort(self, arr, low, high):
        if low< high:
            pi = self.partition(arr, low, high)
            self.quickSort(arr, low, pi - 1)
            self.quickSort(arr, pi + 1, high)


    def partition(self, arr, low, high):
        # pivot (Element to be placed at right position)
        pivot = arr[high]

        # index of smaller elment
        i = low

        # index of iterator
        iter = low

        while iter <= high:

            if arr[iter] <= pivot:
                arr[iter], arr[i] = arr[i], arr[iter]
                i = i + 1
            iter = iter + 1
        return i - 1
```

方案二：同样我们可以基于归并排序 from geek, mergeSort + mergeSort + merge

```
import heapq
class Solution:
    def findKthLargest(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        self.MergeSort(nums, 0, len(nums) - 1)

        return(nums[-k])
    def MergeSort(self, arr, low, high):
        if low < high:

            middle = (high - low )//2 + low

            self.MergeSort(arr, low, middle)
            self.MergeSort(arr, middle + 1, high)

            self.merge(arr, low, middle, high)

    def merge(self, arr, low, middle, high):

        # construct two auxliary array

        left = [arr[i] for i in range(low, middle + 1)]
        right = [arr[i] for i in range(middle+1, high+1)] # sapce left+right = O(N)

        # replace original array
        i, j, k = 0, 0, low

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i = i + 1
            else:
                arr[k] = right[j]
                j = j + 1
            k = k + 1

        # consider the remain element

        while i < len(left):
            arr[k] = left[i]
            i = i + 1
            k = k + 1

        while j < len(right):
            arr[k] = right[j]
            j = j + 1
            k = k + 1
```

算法	稳定性	时间复杂度	空间复杂度	备注
选择排序	×	N2	1	
冒泡排序	√	N2	1	
插入排序	√	N ~ N2	1	时间复杂度和初始顺序有关
希尔排序	×	N 的若干倍乘于递增序列的长度	1	改进版插入排序
快速排序	×	NlogN	logN	
三向切分快速排序	×	N ~ NlogN	logN	适用于有大量重复主键
归并排序	√	NlogN	N	
堆排序	×	NlogN	1	无法利用局部性原理

但是当数组的长度n特别大的时候，上述算法时间复杂度过高。可以使用堆来实现更快速的算法。堆的性质， 堆在python中的常用命令

python内置有heapq的库，其为min heap，即每个根节点要比子节点的值要小。堆常用的操作的时间复杂度为：

- 将长度为N的list初始化为堆，$O(N)$

```
data2 = [1,5,3,2,9,5]
heapq.heapify(data2)
```

- 元素个数为k的min heap每次pop，将会排出堆当前的最小值，$O(k)$

```
heapq.heappop(data2)
```

- 元素个数为k的min heap每次insert，$O(k)$

```
heapq.heappush(data2,i)
```

- 返回数组nums中 k个最大的值 time $O(Nlog(k))$ + space $O(k)$

```
heapq.nlargest(k, nums)
```

方案三：min-heap 算法 时间复杂度为 $Time: O(2n + klog(n))$, 空间复杂度为 $ Space: O(n)$

```
import heapq
class Solution:
    def findKthLargest(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        nums = [-num for num in nums] # time O(n)
        heapq.heapify(nums)           # time O(n) + space O(n)

        res = None
        for i in range(k):
            res = heapq.heappop(nums) # time O(log n)
        return -res
```

min-heap

设置一个大小为k的min-heap，用数组的前k个值对其初始化，然后对数组剩下的数值遍历，每次当这个元素比heap中的最小值大的时候，就把heap中的最小值弹出，并把当前值压入。当对所有的数值进行遍历后，最大的数值会在树底。如果我们想要弹出最大值，需要弹k次。需要弹出第k个值时，仅需要弹一次。

方案四：复杂度为 $Time: O(k + (n-k) log k) $, 空间复杂度为 $ Space: O(k)$

```
import heapq
class Solution:
    def findKthLargest(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        min_heap = nums[:k]
        heapq.heapify(min_heap) # space O(k) + time O(k)

        for i in range(k,len(nums)): # time o((n-K) log k)
            num = nums[i]
            if  nums[i]> min_heap[0]:
                heapq.heappop(min_heap)
                heapq.heappush(min_heap, num)

        return min_heap[0]
```

方案五：

直接使用heapq.nlargets函数，原理如下：

The idea is to init a heap "the smallest element first", and add all elements from the array into this heap one by one keeping the size of the heap always less or equal to `k`. That would results in a heap containing `k` largest elements of the array.

The head of this heap is the answer, i.e. the kth largest element of the array.

The time complexity of adding an element in a heap of size `k` is \mathcal{O}(\log(k))O(log(k)), and we do it `N` times that means \mathcal{O}(N \log(k))O(Nlog(k)) time complexity for the algorithm.

In Python there is a method `nlargest` in `heapq` library which has the same \mathcal{O}(N \log(k))O(Nlog(k)) time complexity and reduces the code to one line.

This algorithm improves time complexity, but one pays with \mathcal{O}(k)O(k) space complexity.

```
```

88. Merge Sorted Array

Easy

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

Note:

- The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.
- You may assume that *nums1* has enough space (size that is greater or equal to *m*+ *n*) to hold additional elements from *nums2*.

Example:

```
Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

Output: [1,2,2,3,5,6]
```

思路：

考虑使用双指针，一个用来记录数组1中的起始位置，一用来记录数组2中的起始位置，让两个逐个比较，较小的值放在数组1中，但是这样会使得数组1中的值均往后移，很难实现。**但是指针只能从数组的起始部位开始移动吗**，答案是不。

我们设置三个指针，tail1, tail2, end. tail1指向数组1的有值的末尾，即赋值为 m - 1。tail2指向数组2的末尾，end指针指向两个数组合并后数组的末尾，即赋值为 m+n+1。这样的话我们就可以像 mergeSort中 merge两个sort array那样来合并数组，只不过在mergeSort中，两个数组的长度均没有冗余的，两个数组的值会合并到另外一个数组中，且顺序是从左到右，小的放在前，具体见上题中的mergesort中的merge方法。。而本题中因为数组1的特殊性我们考虑从后往前，大的放在后。

这种从后往前的插入到num1的方法叫做尾插法。

解题：

```
class Solution:
    def merge(self, nums1, m, nums2, n):
        """
        :type nums1: List[int]
        :type m: int
        :type nums2: List[int]
        :type n: int
        :rtype: void Do not return anything, modify nums1 in-place instead.
        """
        tail1 = m - 1
        tail2 = n - 1
        end = m + n -1

        while tail1 >=0 and tail2 >=0:

            if nums1[tail1] > nums2[tail2]:
                nums1[end] = nums1[tail1]
                tail1 = tail1 - 1
            else:
                nums1[end] = nums2[tail2]
                tail2 = tail2 - 1
            end = end - 1

        if tail2>=0:
            nums1[:end+1] = nums2[:tail2+1]
```

同样的思路，更为简洁的写法如下：

```
def merge(self, nums1, m, nums2, n):
        while m > 0 and n > 0:
            if nums1[m-1] >= nums2[n-1]:
                nums1[m+n-1] = nums1[m-1]
                m -= 1
            else:
                nums1[m+n-1] = nums2[n-1]
                n -= 1
        if n > 0:
            nums1[:n] = nums2[:n]
```

考点三

常用思想：两个指针，一个指向数组头部，一个指向尾部，通过两个指针的对撞来实现算法

有一些 LeetCode 题目，我们可以采用对撞指针进行求解：指针 i 和 j 分别指向数组的第一个元素和最后一个元素，然后指针 i 不断向前， 指针 j 不断递减，知道 i = j（当然具体的逻辑操作根据题目的变化而变化）

167. Two Sum II - Input array is sorted

Easy

Given an array of integers that is already **sorted in ascending order**, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2.

Note:

- Your returned answers (both index1 and index2) are not zero-based.
- You may assume that each input would have *exactly* one solution and you may not use the *same* element twice.

Example:

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore index1 = 1, index2 = 2.
```

思路

最简单的思路就是两个遍历循环....，但是很慢，时间复杂度为$O(n^2)$如下：

```
class Solution:
    def twoSum(self, numbers, target):
        """
        :type numbers: List[int]
        :type target: int
        :rtype: List[int]
        """

        for i in range(len(numbers)):
            numbers1 = numbers[i]
            if numbers1 > target:
                break
            for j in range(i+1,len(numbers)):
                numbers2 = numbers[j]

                if numbers2 == target - numbers1:
                    return i+1, j+1
                elif numbers2 > target - numbers1:
                    break
```

另外一种思路，可以考虑使用两个指针，一个指向头部，一个指向尾部，因为数组是事先排列好的，如果两个指针指向的元素之和小于target，则指向尾部的指针需要向前移动，反之第一个指针向后移动，逐渐往中间靠拢。这样的时间复杂度$O(n)$. 如下双指针解题方法所示

解题

双指针解法

```python
class Solution:
    def twoSum(self, numbers, target):
        """
        :type numbers: List[int]
        :type target: int
        :rtype: List[int]
        """

        left = 0
        right = len(numbers) - 1

        while left < right:
            if numbers[left] + numbers[right] < target:
                left = left + 1
            elif numbers[left] + numbers[right] > target:
                right = right - 1
            else:
                return [left+1, right+1]
```

125. Valid Palindrome

Easy

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Note: For the purpose of this problem, we define empty string as valid palindrome.

Example 1:

```
Input: "A man, a plan, a canal: Panama"
Output: true
```

Example 2:

```
Input: "race a car"
Output: false
```

思路

设置两个指针，往中间靠拢，如果指针指向的不是数字或者字母，则往中间靠拢。当两者指向均为数字或者字母时，进行比较，如果不等，return FALSE，否则继续往中间靠拢。

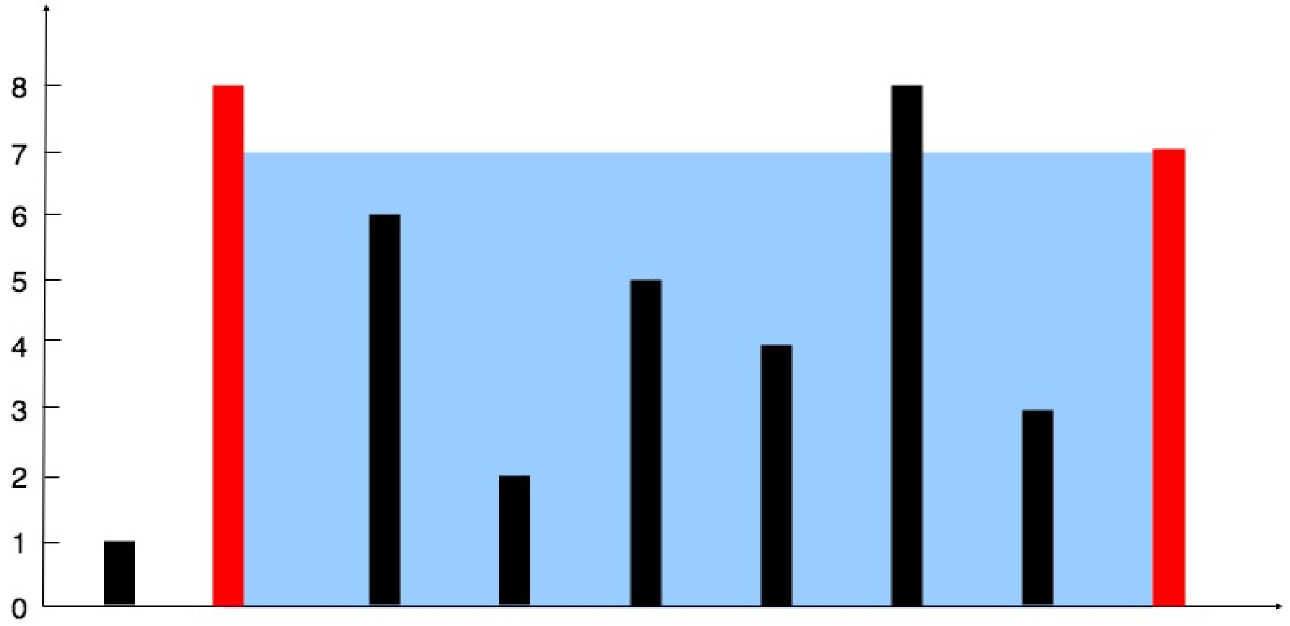
```
class Solution:
    def isPalindrome(self, s):
        """
        :type s: str
        :rtype: bool
        """
        s = s.upper()
        left = 0
        right = len(s) - 1

        while left < right:

            while not s[left].isalnum() and left < right:
                left = left + 1
            while not s[right].isalnum() and left < right:
                right = right - 1

            if s[left] == s[right]:
                left = left + 1
                right = right -1
            else:
                return False


        return True
```

11. Container With Most Water

Medium

Given *n* non-negative integers *a1*, *a2*, ..., *an* , where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and *n* is at least 2.



The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example:

```
Input: [1,8,6,2,5,4,8,3,7]
Output: 49
```

思路

很明显是一道对撞指针的问题。我们设置左指针left对应左壁，right指针对应右壁，那么容器内的盛水量为 (right - left) * min (nums[left], nums[right])。可以看到left和right离得越远，水量越多，并且木桶原理， 较小的壁限制了水量。 我们把left对应数组起始，right对应数组末尾。每次计算水量，并且移动较小的边，来尝试获得更大的容水量。 如果指针从中间开始移动，到两头也是可以的，不过一般习惯对撞指针的做法。

解题：

$TC ~O(n) | SC ~O(1)$

```
class Solution:
    def maxArea(self, height):
        """
        :type height: List[int]
        :rtype: int
        """
        left = 0
        right = len(height) - 1 # SC O(1)
        max_water = 0

        while left < right: # O(n)
            w = right -left
            if height[left] >= height[right]:
                h = height[right]
                right = right - 1
            else:
                h = height[left]
                left = left + 1
            max_water = max_water if w*h< max_water else w*h

        return max_water
```

考点三

双索引技巧 - 滑动窗口

一些题目用滑动窗口方法解题，可以将时间复杂度控制在 O(n) 级别，最重要的是定义好滑动窗口，明确它要表达的意思，当然边界和初始值非常重要。

209. Minimum Size Subarray Sum

Medium

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum ≥ **s**. If there isn't one, return 0 instead.

Example:

```
Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: the subarray [4,3] has the minimal length under the problem constraint.
```

Follow up:

If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*).

思路

如果考虑使用双指针，可以用双指针做一个滑动窗口。left和right指针均指向数组的起始位置，并求sum[left:right+1]，如果sum >=s，则计算当前长度，并累加一次满足条件的次数，并且left指针应当向右移动，减小sum。如果sum<s，则right指针应当向右移动增大sum。

$TC ~O(n*n) | SC ~O(1)$

```
class Solution:
    def minSubArrayLen(self, s, nums):
        """
        :type s: int
        :type nums: List[int]
        :rtype: int
        """
        n = float('inf')
        m = 0
        left = right = 0

        while right < len(nums): # TC O(N)

            accum = sum(nums[left:right+1]) # TC O(N)

            if accum >= s:
                if right - left + 1<n:
                    n = right - left + 1
                m = m +1
                left = left + 1

            if accum < s:
                right = right + 1



        return m if m==0 else n
```

每次都求sum[left:right]将会引入 $O(N)$的时间复杂度，因此我们考虑换一种方式。思路相同，只不过我们不再每次移动指针后，都对nums进行累加，而是当累加值大于s时，减去left指向的值，并把left向后移动一步。当累加值小于s时，加上right指向的值，并把right向后移动一位。**另外需要注意边界问题，如果用while right< len(nums)**，则最后一位nums被累加后，无法对其再次进行判断，所以用while right< len(nums) or accum >= s:。

$TC ~O(n) | SC ~O(1)$

```
class Solution:
    def minSubArrayLen(self, s, nums):
        """
        :type s: int
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0


        left = 0
        right = 0
        min_len = float('inf')
        m = 0
        accum = 0
        while right < len(nums) or accum >=s: # TC O(N)

            if accum >=s:

                accum = accum - nums[left]
                left = left + 1

                length = right - left + 1
                if length <= min_len:
                    min_len = length
                m = m + 1

            else:

                accum = accum + nums[right]
                right = right + 1

        return min_len if m else 0
```

相同的想法，更为简洁的实现方式：

```
class Solution:
    def minSubArrayLen(self, s, nums):
        """
        :type s: int
        :type nums: List[int]
        :rtype: int
        """
        left = right = 0
        accum = 0
        ans = float('inf')

        while right < len(nums):
            accum = accum + nums[right]
            while accum >= s:
                ans = min(ans, right - left + 1)
                accum = accum - nums[left]
                left = left + 1
            right = right + 1
        return ans if ans != float('inf') else 0
```

介绍一种基于二分法的方法，虽然SC和TC比刚才的方法要高，就权当学习了。index of value = binary_search(order_array, low, high, key_value)是一个用来对有序数组中寻找特定值索引的搜索方法，其时间复杂度为$O(log N)$, SC为 $O(1)$。但是在本题当中数组不是有序数组，但是数组内元素均为正值，如果我们建立一个数组，用来记录累加到每个位置的和，那么这个记录累加值的数组为order array。 而两个累加值之差，sum(0~5) - sum(0~3)就会得到 sum(4~5)即连续数组之和，因此我们可以从累加的order array中，以第i位的值+s作为特定值来进行binary search。

java中有现成的 binary search，下边的算法没能通过，原因是原本的binary search在搜索arrary中没有元素时，会返回为空，这点我暂时还没处理好。

```
class Solution:
    def minSubArrayLen(self, s, nums):
        """
        :type s: int
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0

        accum_arr = [0]
        min_len = float('inf')
        for i, num in enumerate(nums):
            accum_arr.append(accum_arr[i] + num)
        print(accum_arr)
        for i, accum in enumerate(accum_arr):
            index = self.binarySearch1(i, len(accum_arr)-1, accum + s, accum_arr)
            print(i,index,accum_arr[index], accum)
            index = index + 1 if accum_arr[index] < accum + s else index

            # the last sum is less than s, just break
            if index == len(accum_arr) - 1:
                break

            min_len = min_len if min_len < index - i else index - i
        return 0 if min_len == float('inf') else min_len
        # accum_arr = [0, 2, 5, 6, 8, 12, 15]
        # s = 7
        # print(self.binarySearch1(0, len(accum_arr)-1, s, accum_arr))



    # recursively method
    def binarySearch1(self, low, high, value, orderArr):

        if low >= high:
            return high
        middle = (low + high) //2
        # print(middle, orderArr[low], orderArr[high])
        if orderArr[middle] < value:
            return self.binarySearch1(middle + 1, high, value, orderArr)

        if orderArr[middle] > value:
            return self.binarySearch1(low, middle-1, value, orderArr)
        if orderArr[middle] == value:
            return middle

    # iterative method
    def binarySearch2(self,low, high, value, orderArr):

        while low != high:
            middle = (low+high) //2


            if orderArr[middle] < value:
                low = middle + 1

            if orderArr[middle] > value:
                high = middle -1

            if orderArr[middle] == value:
                return middle
        return high
```