叙言

从本章开始我们进入算法珠峰部分，本部分我们主要学习四大经典的算法思想：滑动窗口、贪心、回溯、动态规划、图和宽度优先问题。以这几种典型的思想为主线可以拓展出大量的问题，而且很多问题都非常热门。但是这些问题又不会直接告诉你要用什么方法解决，所以很多题目不会，不是你数组、链表不行，而是你需要懂得更高级的思想。

那如何掌握这些思想呢？在讲义里，你会经常看到”模板“两个字，是让你背下来吗？不是的，我们在第一章就一直在说，算法要按照专题来刷，因为每个专题都有很多共性问题，我们将其不断抽象和总结，并通过大量相关的题目来练习，最后就形成该类型题目的解题模板。这也是为什么学生时代很多人不那么能拼也能考好的原因，这也是为什么说题目越做越少的原因。

任何学习都是这样的，越是高级的算法，解决的问题越典型，适用的场景越单一，因此每种思想的解题套路越明确。我们学习的过程中，要时刻思考为什么会有这种思想，能解决什么场景的问题，有哪些典型的题目，如何变换一下来解决其他问题，只有这些问题都清楚了，才算学到家了。

还有就是要深入到什么程度才算可以呢，算法题目很多，但是常见的面试题数量是有限的，要知道面试官也是刷题过来的，现在刷的也基本是他曾经刷过的，所以我们将大部分常见的题目搞清楚就够了，也不必在遇到某个难题时恐慌。你不会因为考不上清华北大就不高考了吧，也不会因为娶不到杨幂就孤独终老吧。

本章我们先来研究双指针和滑动窗口思想，在一维数组和链表部分我们已经介绍了很多双指针的题目，这里是其进一步拓展。

【学习目标】

1.复习一维数组，对数组进行多轮插入或者删除时会频繁移动数据，理解双指针是如何避免该问题的。

2.理解滑动窗口的原理和适用场景。

3.掌握常见的滑动窗口算法题。

1.基本思想

在数组章节我们说过很多算法会大量移动数组中的元素，而且会频繁移动，这导致执行效率低下、执行超时。使用两个变量能比较好的解决很多相关问题，在《一维数组》和《链表》章节我们介绍了很多典型例子，于是这种方式就慢慢演化成了“双指针思想”。在很多应用中将其进一步完善，便形成了滑动窗口思想，学过计算机网络的同学都知道滑动窗口协议（Sliding Window Protocol），该协议是TCP实现流量控制等的核心策略之一。事实上在与流量控制、熔断、限流、超时等场景下都会首先从滑动窗口的角度来思考问题，例如hystrix、sentinel等框架都使用了这种思想。

滑动窗口的思想非常简单，如下图所示，假如窗口的大小是3，当不断有新数据来时，我们会维护一个大小为3的一个区间，超过3的就将新的放入老的移走。这个过程有点像火车在铁轨上跑，原始数据可能保存在一个很大的空间里，但是我们标记的小区间就像一列火车，一直向前走。

有了区间，那我们就可以造题了，例如让你找序列上三个连续数字的最大和是多少，或者子数组平均数是多少（LeetCode643）等等。

从上面的图可以看到，所谓窗口就是建立两个索引，left和right，并且保持right-left=3，然后一边遍历序列，一边寻找，每改变一次就标记一下当前区间的最大值就行了。

这个例子已经告诉我们了什么是窗口、什么是窗口的滑动：

- **窗口：**窗口其实就是两个变量left和right之间的元素，也可以理解为一个区间。窗口大小可能固定，也可能变化，如果是固定大小的，那么自然要先确定窗口是否越界，再执行逻辑处理。如果不是固定的，就要先判断是否满足要求，再执行逻辑处理。
- **滑动：**说明这个窗口是移动的，事实上移动的仍然是left和right两个变量，而不是序列中的元素。当变量移动的时，其中间的元素必然会发生变化，因此就有了这种不断滑动的效果。

这个思路其实非常简单，固定窗口的滑动就是火车高铁行驶这种大小不变的移动 。而可变的窗口就像两个老师带着一队学生外出，一个负责开路，一个负责断后，中间则是小朋友。两个指针轮流前进，窗口大小增增减减，窗口不断向右滑动。

根据窗口大小是否固定，可以造出两种类型的题。如果是固定的，则一般会让你求哪个窗口的元素最大、最小、平均值、和最大、和最小等等类型的问题。如果窗口是变的，则一般会让你求一个序列里最大、最小窗口是什么。

很多题目要处理的区间不算大，因此可以采用暴力搜索、Hash、集合等来操作。如果再让你找区间里的最大最小等问题，你会想到什么？堆！我们在排序章节介绍过，堆结构非常适合在流数据中找固定区间内的最大、最小等问题。因此滑动窗口经常和堆一起使用可以完美解决很多复杂的问题，详细看“滑动窗口与堆结合的问题”一节。

最后一个问题，那双指针和滑动窗口啥区别呢？根据性质我们可以看到，滑动窗口是双指针的一种类型，主要关注两个指针之间元素的情况，因此范围更小一些，而双指针的应用范围更大。

2.滑动窗口热点问题

接下来我们就看一下滑动窗口的典型题目。

2.1 两个入门题

本小节，我们看一下固定窗口和窗口不固定的典型问题。

2.1.1.子数组最大平均数

LeetCode643 给定 `n` 个整数，找出平均数最大且长度为 `k` 的连续子数组，并输出该最大平均数。

其中 1<=k<=nums.length<=10^5。

```
输入：[1,12,-5,-6,50,3], k = 4
输出：12.75
解释：最大平均数 (12-5-6+50)/4 = 51/4 = 12.75
```

这是典型的滑动窗口，大小都规定了，就是K，那我们只要先读取k个，然后逐步让窗口向前走就可以了，图示与上一节的基本一样。直接看代码：

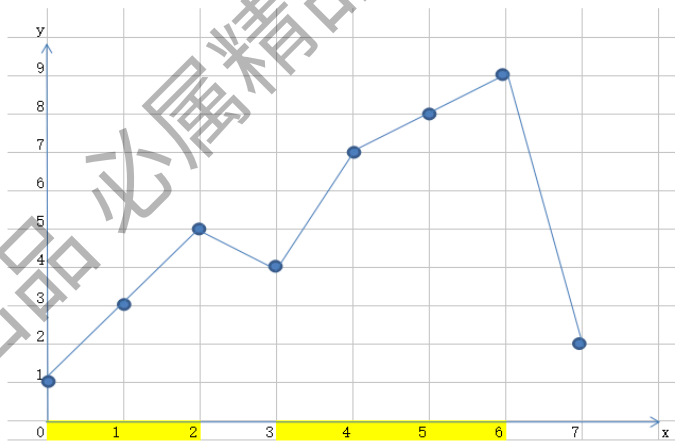
```
    public static double findMaxAverage(int[] nums, int k) {
         int len = nums.length;
         int widowSum = 0;
         if (k > nums.length || nums.length < 1 || k < 1) {
             return 0;
         }
         //   第一步 先求第一个窗口的和
         for (int i = 0; i < k; i++) {
             widowSum += nums[i];
         }

         //  第二步 ： 遍历，每次右边增加一个，左边减去一个，重新计算窗口最大值
         int res = widowSum;
         for (int right = k; right < len; right++) {
             widowSum += nums[right] - nums[right - k];
             res = Math.max(res, widowSum);
         }
         return (double) res / k;
     }
```

2.1.2最长连续递增序列

我们再看一个窗户变化的情况。LeetCode674.给定一个未经排序的整数数组，找到最长且连续递增的子序列**，并返回该序列的长度。

```
输入：nums = [1,3,5,4,7]
输出：3
解释：最长连续递增序列是 [1,3,5], 长度为3。
尽管 [1,3,5,7] 也是升序的子序列, 但它不是连续的，因为 5 和 7 在原数组里被 4 隔开。
```

为了方便演示，我们将示例序列再增加几个元素{1,3,5,4,7,8,9,2}，则图示如下，题目要求找到最长的连续递增子序列。



可以看到，最长递增子序列为{4,7,8,9}所以应该返回4。所以在遍历的时候，我们可以从第 2 个元素开始，先定义[left,right)的区间来表示当前的递增区间，执行如下操作：

- 如果当前遍历到的元素比它左边的那一个元素要严格大，right就增加；
- 否则就将left跳到right的起始位置，重新开始计算。

```
public static int findLengthOfLCIS(int[] nums) {
    int left = 0, right = 0;
    int res = 0;
    while (right < nums.length) {
        //右侧的新元素比左侧的小，则重新开始记录left位置
        //该问题本质就是快慢指针，left就是slow指针
        if (right > 0 && nums[right - 1] >= nums[right]) {
            left = right;
        }
        right++;
        res = Math.max(res, right - left);
    }
    return res;
}
```

上面代码中，序列在[left..right) 严格单调递增，区间的长度为right - left。

本题还有多种解法，另外一种简易的思路是一边遍历，一边统计每个递增区间的长度，如果长度超过之前所有区间的长度，就将其保留，代码如下：

```
    public static int findLengthOfLCIS(int[] nums) {
        int curLen = 1;//当前连续递增区间的长度
        int res = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i - 1] >= nums[i]) {
                //不满足要求，重新开始计数
                curLen = 1;
            } else {
                curLen++;
            }
            res = Math.max(curLen, res);
        }
        return res;
    }
```

如果不知道滑动窗口，本题也能做，这里只是将其明确了一下，所以滑动窗口就是个名字，不要被这些概念吓到。

2.2 最长子串专题

先来看一道高频算法题：无重复字符的最长子串。具体要求是给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。例如，输入: s = "abcabcbb" 则输出3，因为无重复字符的最长子串是 "abc"，所以其长度为3。

怎么做后面再说，如果再变一下要求，至多包含两个不同字符的最长子串，该怎么做呢?

再变一下要求，至多包含 K 个不同字符的最长子串，该怎么做呢?

到这里是否感觉，这不在造题吗？是的！上面就分别是LeetCode3、159、340题，而且这几道题都可以用滑动窗口来解决。学会之后，我们就总结出滑动窗口的解题模板了。

接下来，我们就一道一道看。

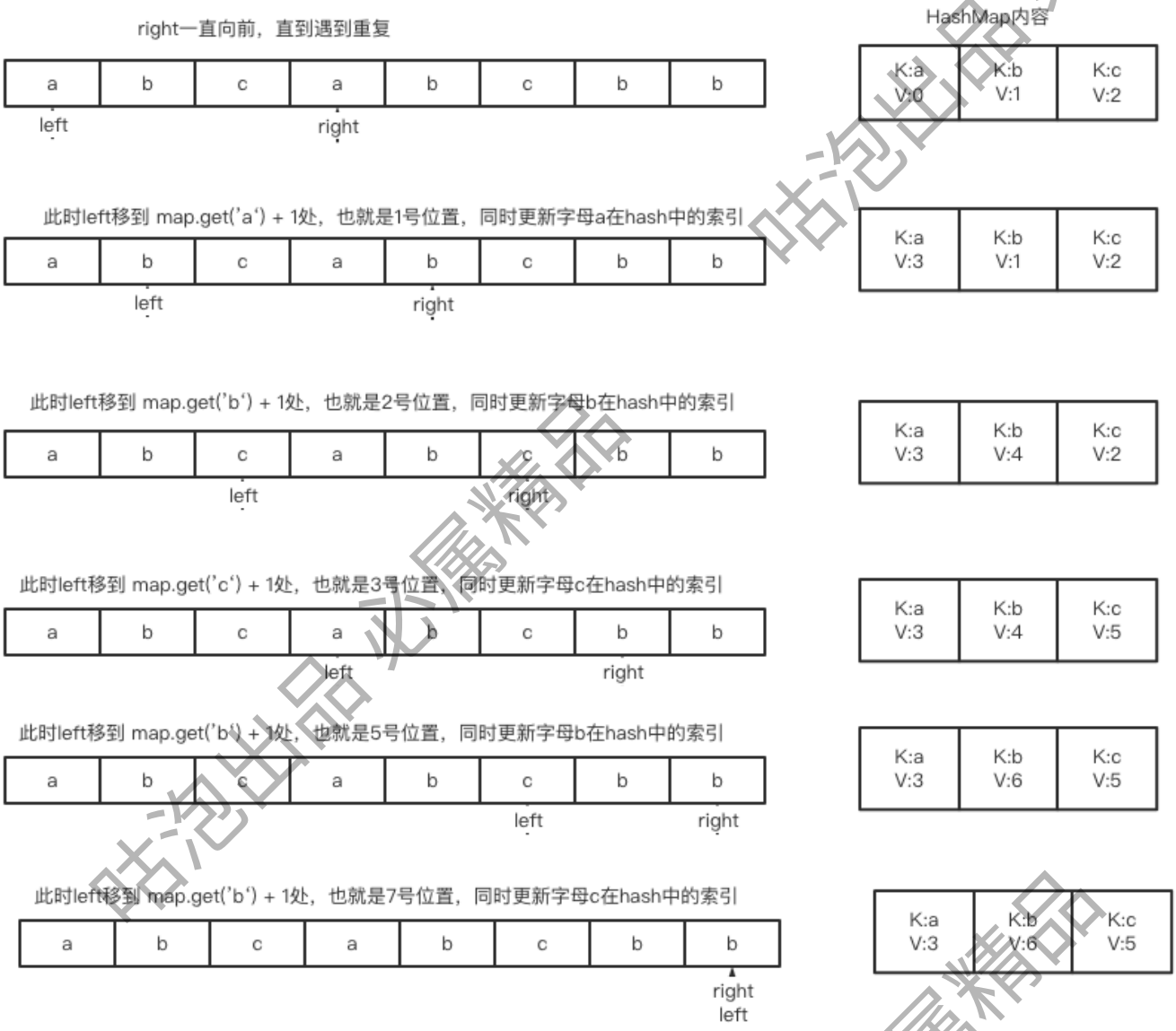
2.2.1 无重复字符的最长子串

LeetCode3 给定一个字符串 s ，请你找出其中不含有重复字符的最长子串的长度。例如：

```
输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

要找最长子串，必然要知道无重复字符串的首和尾，然后再从中确定最长的那个，因此至少两个指针才可以，这就想到了滑动窗口思想。即使使用滑动窗口，深入分析会发现具体处理起来有多种方式。这里介绍一种经典的使用Map的思路。

我们定义一个K-V形式的map，key表示的是当前正在访问的字符串，value是其下标索引值。我们每访问一个新元素，都将其下标更新成对应的索引值。具体过程如下图：



如果是已经出现过的，例如上述示例中的 abcabc，当第二次遇到a时，我们就更新left成为第一个b所在的位置，此时惊奇的发现left要移动的位置恰好就是map.get('a') + 1=1，我们将'a'用序列来表示，放在一起就是map.get(s.charAt(i)) + 1。其他情况可以参考图示依次类推。

有一种特殊情况我们需要考虑，例如abba，我们第二次访问b时，left=map.get('b') + 1=2。然后继续访问第二个a，此时left=map.get('a') + 1=1，也就是left后退了，显然不对。我们应该让left在2的基础上继续向前，那该怎么办呢？和原来的对比一下，将最大的加1就可以了，也就是：

```
left = Math.max(left,map.get(s.charAt(i)) + 1);
```

完整的代码如下：

```
public  int lengthOfLongestSubstring2(String s) {
      if (s.length() == 0) return 0;
      HashMap<Character, Integer> map = new HashMap<Character, Integer>();
      int max = 0;
      int left = 0;
      for (int right = 0; right < s.length(); right++) {
          if (map.containsKey(s.charAt(right))) {
```

```
                left = Math.max(left, map.get(s.charAt(right)) + 1);
            }
            map.put(s.charAt(right), right);
            max = Math.max(max, right - left + 1);
        }
        return max;
    }
```

除了上述方法，不用Hash存储索引，也可以用滑动窗口思想来解决，感兴趣的可以研究一下。

2.2.2 至多包含两个不同字符的最长子串

给定一个字符串 s ，找出 至多 包含两个不同字符的最长子串 t ，并返回该子串的长度，这就是LeetCode159题。例如：

```
输入：“eceba”
输出：3
解释：t 是 “ece”，长度为3。
```

我们仍然使用left和right来锁定一个窗口，然后一边向右移动一边分析。我们用一个序列来看一下：aabbcccd。

我们接下来需要解决两个问题，一个是怎么判断只有2个元素，另一个是移除的时候怎么知道移除谁，以及移除之后left是什么。

要判断只有2个元素，还是Hash好用，每一个时刻，这个 hashmap 包括不超过 3 个元素。这里还要考虑到要移除谁，所以我们要设计一下Hash的Key-Value的含义。我们把字符串里的字符都当做键，在窗口中的最右边的字符位置作为值。此时使用下面的代码就可以确定要删除谁，以及窗口left的新位置：

```
del_idx = Collections.min(hashmap.values());
left = del_idx + 1;
```

为什么呢？我们还是画图看一下：

对应上的【1】，此时应该删除a，对应就是hash里val最小的a，
left恰好等于val+1=2

对应上的【3】，此时应该删除b，对应就是hash里val最小的b，
left恰好等于val+1=4

所以我们可以充分利用Map的工具来解决该问题：

```
public  int lengthOfLongestSubstringTwoDistinct(String s) {

     if (s.length() < 3) {
         return s.length();
     }
     int left = 0, right = 0;
     HashMap<Character, Integer> hashmap = new HashMap<>();
     int maxLen = 2;

     while (right < s.length()) {

         if (hashmap.size() < 3)
             hashmap.put(s.charAt(right), right++);

         // 如果大小达到了3个
         if (hashmap.size() == 3) {
             // 最左侧要删除的位置
             int del_idx = Collections.min(hashmap.values());
             hashmap.remove(s.charAt(del_idx));
             // 窗口left的新位置
             left = del_idx + 1;
         }

         maxLen = Math.max(maxLen, right - left);
     }
     return maxLen;
 }
```

2.2.3 至多包含 K 个不同字符的最长子串

如果再提高一下难度， 至多包含 K 个不同字符的最长子串该怎么办呢？这就是LeetCode340题。

题目的完整要求是：给定一个字符串 s，找出 **至多** 包含 k 个不同字符的最长子串T。示例：

```
输入：s = "eceba", k = 2
输出：3
解释：则 T 为 "ece"，所以长度为 3。
```

本题与上面的题几乎没有区别，只要将判断hash大小为2改成k就可以，超过2就是k+1。一分钟实现：

```
    public  int lengthOfLongestSubstringKDistinct(String s, int k) {
         if (s.length() < k + 1) {
             return s.length();
         }

         int left = 0, right = 0;
         HashMap<Character, Integer> hashmap = new HashMap<>();
         int maxLen = k;

         while (right < s.length()) {

             if (hashmap.size() < k + 1)
                 hashmap.put(s.charAt(right), right++);

             // 如果大小达到了k个
             if (hashmap.size() == k + 1) {
                 //
                 int del_idx = Collections.min(hashmap.values());
                 hashmap.remove(s.charAt(del_idx));
                 // 窗口left的新位置
                 left = del_idx + 1;
             }

             maxLen = Math.max(maxLen, right - left);
         }
         return maxLen;
     }
```

2.3 长度最小的子数组

LeetCode209.长度最小的子数组,给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其和 ≥ target 的长度最小的 连续子数组 [numsl, numsl+1, ..., numsr-1, numsr] ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

```
输入：target = 7, nums = [2,3,1,2,4,3]
输出：2
解释：子数组 [4,3] 是该条件下的长度最小的子数组。
```

本题可以使用双指针来解决，也可以视为队列法，基本思路是先让元素不断入队，当入队元素和等于target时就记录一下此时队列的容量，如果队列元素之和大于target则开始出队， 直到小于target则再入队。

如果出现等于target的情况，则记录一下此时队列的大小，之后继续先入队再出队。每当出现元素之和等于target时我们就保留容量最小的那个。

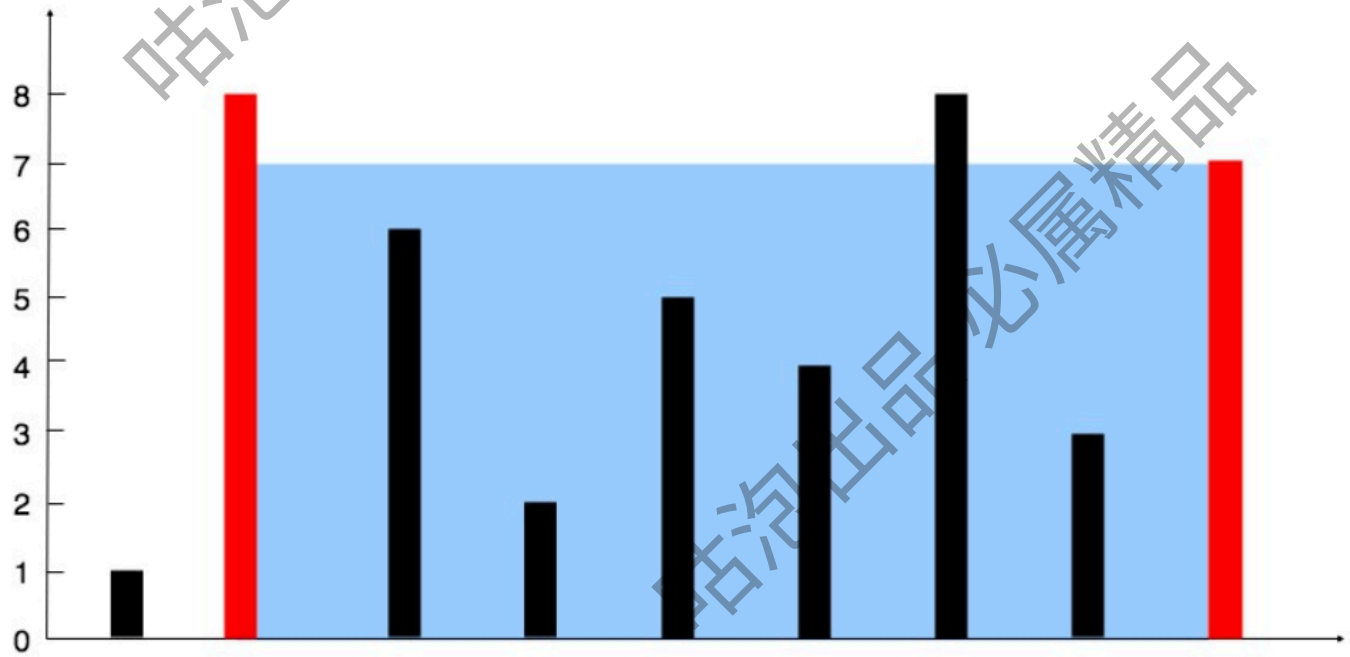
实现代码如下：

```
    public int minSubArrayLen(int target, int[] nums) {
        int left = 0, right = 0, sum = 0, min = Integer.MAX_VALUE;
        while (right < nums.length) {
            sum += nums[right++];
            while (sum >= target) {
                min = Math.min(min, right - left);
                sum -= nums[left++];
            }
        }
        return min == Integer.MAX_VALUE ? 0 : min;
    }
```

2.4 盛水最多的容器

LeetCode11.给定一个长度为 n 的整数数组 height 。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i]) 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。返回容器可以储存的最大水量。

示例：




```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。
```

本题看似复杂，但其实简单的很。设两指针 i , j ，指向的水槽板高度分别为 h[i] , h[j] ，此状态下水槽面积为S(i,j) 。由于可容纳水的高度由两板中的 短板 决定，因此可得如下面积公式：

$$S(i,j)=min(h[i],h[j])\times(j-i)$$

在每个状态下，无论长板或短板向中间收窄一格，都会导致水槽底边宽度−1 变短：

- 若向内移动短板 ，水槽的短板min(h[i],h[j]) 可能变大，因此下个水槽的面积可能增大 。
- 若向内移动长板 ，水槽的短板min(h[i],h[j]) 不变或变小，因此下个水槽的面积一定变小 。

因此，只要初始化双指针分列水槽左右两端，循环每轮将短板向内移动一格，并更新面积最大值，直到两指针相遇时跳出；即可获得最大面积。

```
class Solution {
    public int maxArea(int[] height) {
        int i = 0, j = height.length - 1, res = 0;
        while(i < j) {
            res = height[i] < height[j] ?
                Math.max(res, (j - i) * height[i++]):
                Math.max(res, (j - i) * height[j--]);
        }
        return res;
    }
}
```

2.5 寻找子串异位词（排列）

如果两个字符串仅仅是字母出现的位置不一样，则称两者相互为对方的一个排列，也称为异位词。

如果判断两个字符串是否互为排列，是字符串的一个基本算法。现在我们给增加难度。看LeetCode567和438两个题。

2.5.1 字符串的排列

LeetCode567.给你两个字符串s1和s2 ，写一个函数来判断 s2是否包含 s1的排列。如果是，返回 true ；否则，返回 false 。换句话说，s1 的排列之一是 s2 的子串 。其中s1和s2都只包含小写字母。

```
示例:
输入: s1 = "ab" s2 = "eidbaooo"
输出: true
解释: s2 包含 s1 的排列之一 ("ba").
```

本题因为字符串s1的异位词长度一定是和s2字符串的长度一样的，所以很自然的想到可以以s1.length()为大小截图一个固定窗口，然后窗口一边向右移动，一边比较就行了。此时可以将窗口内的元素和s1先做一个排序，然后再比较即可，但是这样做的问题是排序代价太高了，我们需要考虑性能更优的方法。

所谓的异位词不过两点：字母类型一样，每个字母出现的个数也是一样的。题目说s1和s2都仅限小写字母，因此我们可以创建一个大小为26的数组，每个位置就存储从a到z的个数，为了方便操作，索引我们使用

```
index=s1.charAt(i) - 'a'
```

来表示，这是处理字符串的常用技巧。

此时窗口的right向右移动就是执行：

```
charArray2[s2.charAt(right) - 'a']++;
```

而left向右移动就是执行：

```
 int left = right - sLen1;
 charArray2[s2.charAt(left) - 'a']--;
```

所以，完整代码如下：

```
public boolean checkInclusion(String s1, String s2) {
        int sLen1 = s1.length(), sLen2 = s2.length();
        if (sLen1 > sLen2) {
            return false;
        }
        int[] charArray1 = new int[26];
        int[] charArray2 = new int[26];
        //先读最前面的一段来判断。
        for (int i = 0; i < sLen1; ++i) {
            ++charArray1[s1.charAt(i) - 'a'];
            ++charArray2[s2.charAt(i) - 'a'];
        }

        if (Arrays.equals(charArray1, charArray2)) {
            return true;
        }
        for (int right = sLen1; right < sLen2; ++right) {
            charArray2[s2.charAt(right) - 'a']++;
            int left = right - sLen1;
            charArray2[s2.charAt(left) - 'a']--;
            if (Arrays.equals(charArray1, charArray2)) {
                return true;
            }
        }
        return false;
    }
```

上面只是判断有没有，那如果让你确定一下有几个呢？有或者如果有的话，将异位词的开始位置输出出来怎么做呢？这就是LeetCode438题。

2.5.2 找到字符串中所有字母异位

LeetCode438.找到字符串中所有字母异位词，给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。注意s和p仅包含小写字母。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。例如：

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

本题的思路和实现与上面几乎一模一样，唯一不同的是需要用一个List，如果出现异位词，还要记录其开始位置，那直接将其add到list中就可以了。完整代码：

```
    public List<Integer> findAnagrams(String s, String p) {
       int sLen = s.length(), pLen = p.length();
        if (sLen < pLen) {
            return new ArrayList<Integer>();
        }
        List<Integer> ans = new ArrayList<Integer>();
        int[] sCount = new int[26];
        int[] pCount = new int[26];
        //先分别初始化两个数组
        for (int i = 0; i < pLen; i++) {
            sCount[s.charAt(i) - 'a']++;
            pCount[p.charAt(i) - 'a']++;
        }
        if (Arrays.equals(sCount, pCount)) {
            ans.add(0);
        }

        for (int left = 0; left < sLen - pLen; left++) {
            sCount[s.charAt(left) - 'a']--;
            int right=left + pLen;
            sCount[s.charAt(right) - 'a']++;

            if (Arrays.equals(sCount, pCount)) {
            //上面left多减了一次，所以
                ans.add(left + 1);
            }
        }
        return ans;
    }
```

2.6 滑动窗口与堆结合的问题

我们在《堆》一章解释过堆的大小一般是有限的，而且能直接返回当前位置下的最大值或者最小值。而该特征与滑动窗口结合，碰撞出的火花可以非常方便的解决一些特定场景的问题。

LeetCode239 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位，返回滑动窗口中的最大值。

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

这种方法我们在基础算法的堆部分介绍过。对于最大值、K个最大这种场景，优先队列（堆）是首先应该考虑的思路。大根堆可以帮助我们实时维护一系列元素中的最大值。

本题初始时，我们将数组 nums 的前 k个元素放入优先队列中。每当我们向右移动窗口时，我们就可以把一个新的元素放入优先队列中，此时堆顶的元素就是堆中所有元素的最大值。然而这个最大值可能并不在滑动窗口中，在这种情况下，这个值在数组 nums 中的位置出现在滑动窗口左边界的左侧。因此，当我们后续继续向右移动窗口时，这个值就永远不可能出现在滑动窗口中了，我们可以将其永久地从优先队列中移除。

我们不断地移除堆顶的元素，直到其确实出现在滑动窗口中。此时，堆顶元素就是滑动窗口中的最大值。为了方便判断堆顶元素与滑动窗口的位置关系，我们可以在优先队列中存储二元组 (num,index)，表示元素num 在数组中的下标为index。

```
public int[] maxSlidingWindow(int[] nums, int k) {
      int n = nums.length;
      PriorityQueue<int[]> pq = new PriorityQueue<int[]>(new Comparator<int[]>() {
          public int compare(int[] pair1, int[] pair2) {
              return pair1[0] != pair2[0] ? pair2[0] - pair1[0] : pair2[1] - pair1[1];
          }
      });
      for (int i = 0; i < k; ++i) {
          pq.offer(new int[]{nums[i], i});
      }
      int[] ans = new int[n - k + 1];
      ans[0] = pq.peek()[0];
      for (int i = k; i < n; ++i) {
          pq.offer(new int[]{nums[i], i});
          while (pq.peek()[1] <= i - k) {
              pq.poll();
          }
          ans[i - k + 1] = pq.peek()[0];
      }
```

```
        return ans;
    }
```

本题除了堆，直接比较也是可以的，只是效率低，逼格低，除此之外，还可以使用双向队列、单调队列来解决，后面再研究。

我们再拓展一下，如果要找中位数怎么办呢？可以使用两个堆，感兴趣的同学可以研究一下LeetCode480. 滑动窗口中位数。

2.7 颜色分类（荷兰国旗问题）

这个也是非常经典的算法问题，LeetCode75，也称为荷兰国旗问题。给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。必须在不使用库的sort函数的情况下解决这个问题。

```
示例：
输入：nums = [2,0,2,1,1,0]
输出：[0,0,1,1,2,2]
```

如果你查一下会发现一个有趣的现象，很多几个国家的国旗如下：

你是否感觉这些号称创新力很高的国家竟然这么敷衍，这也太像了吧！这道题你是否感觉叫法国国旗或者意大利国旗更合适啊，那为什么叫荷兰国旗呢？因为这个题的发明者正是大名鼎鼎的荷兰计算机科学家Dijkstra，在图算法中我们已经认识他了。

这个题是非常经典的双指针问题，而且还可以使用多种方式的双指针。这里我们分析两种方法，一种与冒泡排序非常类似，一种与快速排序非常类似。

1.基于冒泡排序的双指针（快慢指针）

冒泡排序我们都知道，就是根据大小逐步和后面的比较，慢慢调整到整体有序。这种方法还是稳定的排序方法。

我们可以考虑对数组进行两次遍历。在第一次遍历，我们将数组中所有的 0 交换到数组的头部，这样第二次遍历只需要处理1和2的问题就行了，而这两次寻找本身又是非常漂亮的双指针。代码如下：

```
public  void sortColors(int[] nums) {
        int n = nums.length;
        int left = 0;
        //将所有的0交换到数组的最前面
        for (int right = 0; right < n; right++) {
            if (nums[right] == 0) {
                int temp = nums[right];
```

```
                nums[right] = nums[left];
                nums[left] = temp;
                left++;
            }
        }

        //将所有的1交换到2的前面
        for (int right = left; right < n; ++right) {
            if (nums[right] == 1) {
                int temp = nums[right];
                nums[right] = nums[left];
                nums[left] = temp;
                ++left;
            }
        }
    }
```

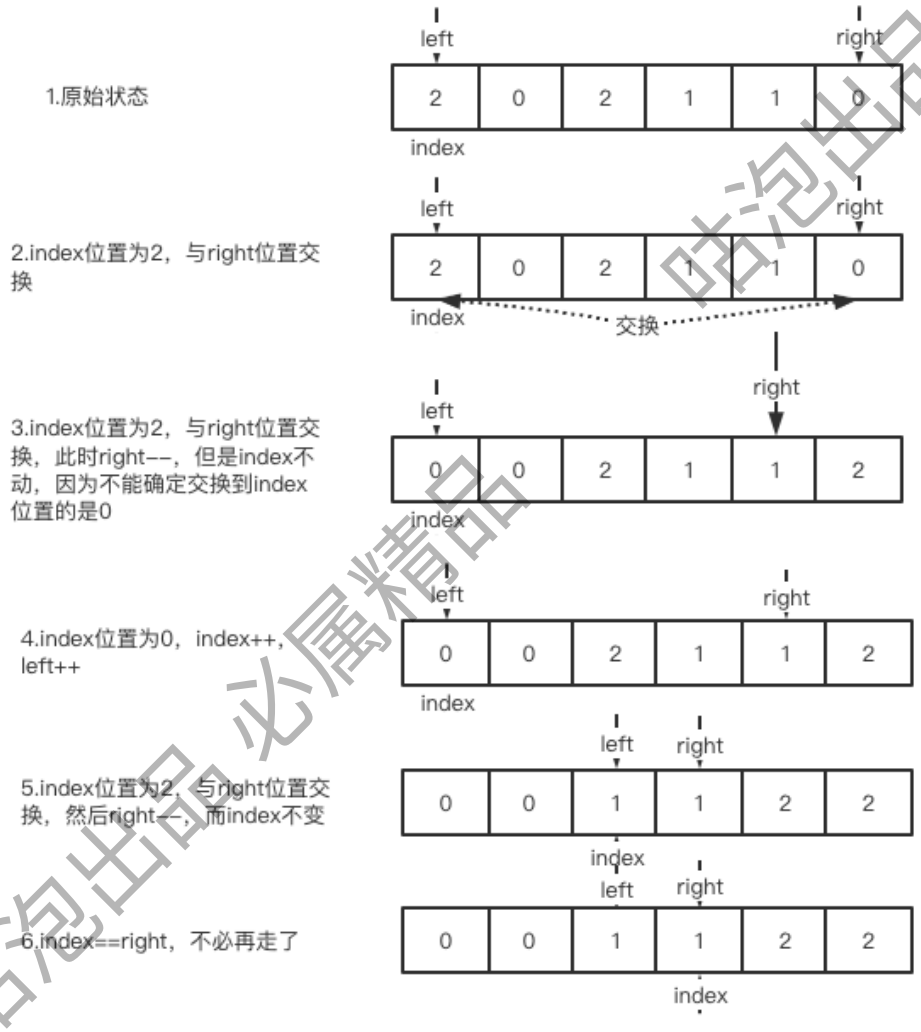
上面的方式能解决问题，而且效率还不错。但是面试官可能又给你出幺蛾子，能否将两次遍历变成一次搞定？**这个稍微有些难，如果感觉太烧脑，可以暂时放下，头发长齐了再看。**

如果要求只用一次遍历就要解决问题，该怎么办呢？我们隐约感觉到要使用三个指针才行：

- left指针，表示left左侧的元素都是0
- right指针，表示right右侧的元素都是2
- index指针，从头到尾遍历数组，根据nums[index]是0还是2决定与left交换还是与right交换。

index位置上的数字代表着我们当前需要处理的数字。当index为数字1的时候，我们什么都不需要做，直接+1即可。如果是0，我们放到左边，如果是2，放到右边。如果index=right，则可以停止。

我们看一下图示：



这里的重点和难点index位置为2进行交换后为什么只进行right--，而不用index++呢？这是因为我们right位置交换过来的元素可能是0，也可能是1。如果是0自然没问题，但是如果是1则执行index++就将1跳过了无法处理了。所以我们先不动index，在下一次循环时继续判断这个index位置元素是不是0。

那为啥index位置是0的时候执行swap就可以index++了呢，这是因为如果index前面位置如果存在位置都会被swap到right位置去了，这里只需要处理0和1的情况就可以了。

代码如下：

```
public void sortColors(int[] nums) {
        int left=0,right=nums.length-1;
        int index=0;
        while(index<=right){
            if(nums[index]==0)
                swap(nums,index++,left++);
            else if(nums[index]==2)
                swap(nums,index,right--);
            else
                index++;
        }
    }
    private void swap(int[] nums,int i,int j){
        int temp=nums[i];
        nums[i]=nums[j];
```

```
        nums[j]=temp;
    }
```

3. 总结

本章我们介绍了第一个常见的算法思想——滑动窗口。

解题模板：

1. 我们在序列S中使用两个变量(指针)，初始化 left = right = 0，把索引闭区间 [left, right] 称为一个「窗口」。
2. 我们先不断地增加 right，扩大窗口 [left, right]，直到窗口满足或者不满足要求（根据题目要求判断）。
3. 之后不断调整 left 指针，直到窗口中的元素不满足或者满足符合要求(根据题目要求确定)，然后执行一些逻辑处理。
4. 重复第 2 和第 3 步，直到 right到达元素尽头。
5.

在一维数组中，我们介绍过简单的双指针，这里进一步拓展成滑动窗口思想，并讲解了一些经典的问题。如果还想继续练习滑动窗口思想，在LeetCode中还有很多，以下滑动窗口的题可以继续练习：

LeetCode219 存在重复元素 II

LeetCode 220 存在重复元素 III

LeetCode713 乘积小于K的子数组

LeetCode 904 水果成篮

LeetCode1052 爱生气的书店老板

LeetCode1695 删除子数组的最大得分

下面这几个滑动窗口问题难度稍大，搞一下?

LeetCode424 替换后的最长重复字符

LeetCode187 DNA的问题

leetcode 718 最长重复子数组

LeetCode 487 最大连续1的个数 II

LeetCode1004 最大连续1的个数 III

LeetCode76 最小覆盖子串