滑动窗口

注意连续的话，移动可以直接移动很远的。

双串的是否包含，覆盖的问题

滑动窗口的算法设计技巧思路非常简单，就是维护一个窗口，不断滑动，然后更新答案。

算法逻辑如下：

```
int left = 0, right = 0;

while (right < s.size()) {`
    // 增大窗口
    window.add(s[right]);
    right++;

    while (window needs shrink) {
        // 缩小窗口
        window.remove(s[left]);
        left++;
    }
}
```

困扰是不是算法思路，而是各种细节问题。比如说如何向窗口添加元素，如何缩小窗口，在窗口滑动的哪个阶段更新结果。

```
/* 滑动窗口算法框架 */
void slidingWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 右移窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 左移窗口
```

```
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
    }
}
```

其中两处 `...` 表示的更新窗口数据的地方，到时候你直接往里面填就行了。

Leetcode76 最小覆盖子串

给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。

注意：如果 s 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"

示例 2：

输入：s = "a", t = "a"
输出："a"

如果我们使用暴力解法，代码大概是这样的：

```
for (int i = 0; i < s.size(); i++)
    for (int j = i + 1; j < s.size(); j++)
        if s[i:j] 包含 t 的所有字母:
            更新答案
```

> 解题思路：用一个数组记录 t中出现的字母以及频率；滑动窗口也进行记录；为了知道都记录到了需要加个判断条件来进行if判断。

```
class Solution {
    public String minWindow(String s, String t) {
        // 两个数组，一个变量
        int[] need = new int[128];
        int[] window = new int[128];
        // 对其遍历 记录字母以及频率次数
        for(char c:t.toCharArray()){
            need[c]++;
        }
        // 记录满足次数
        int left = 0;
        int right = 0;
        // 记录长度
        int minLength = s.length()+1;
        int count = 0;
        // 记录空字符串
        String res = "";
```

```
        while(right<s.length()){
            char c = s.charAt(right);
            window[c]++;
            // 符合就判断记录
            if(need[c]>0&&need[c]>=window[c]){
                count++;
            }
            // 开始缩放
            while(count==t.length()){
                // 左边
                c = s.charAt(left);
                // 再次判断如果在其中就停止这个while循环了
                if(need[c]>0&&need[c]>=window[c]){
                    count--;
                }
                // 记录长度
                if(right-left+1<minLength){
                    minLength = right-left+1;
                    res = s.substring(left,right+1);
                }
                // 窗口
                window[c]--;
                left++;
            }

            // 继续走
            right++;
        }
        return res;

    }
}
```

Leetcode567 判断s2是否包含s1的排列

给定两个字符串 s1 和 s2，写一个函数来判断 s2 是否包含 s1 的排列。

换句话说，第一个字符串的排列之一是第二个字符串的 子串 。

示例 1：

输入: s1 = "ab" s2 = "eidbaooo"
输出: True
解释: s2 包含 s1 的排列之一 ("ba").
示例 2：

输入: s1= "ab" s2 = "eidboaoo"
输出: False

> 与上述的题目，类似。不过缩放的条件变为了长度的限制，多做了一层判断

```
class Solution {
    public boolean checkInclusion(String s1, String s2) {
```

```
        // 与之前的解法类似，只不过left的移动时机是right-left>size 控制是通过长度，后续在判断是否单词都存在了
        // 两个数组，一个维护need 一个维护windows
        // 只包含小写字母
        int[] need = new int[128];
        int[] windows = new int[128];
        // 统计s1 统计频率表
        for(char c:s1.toCharArray()){
            need[c]++;
        }
        // 需要记录是否都有了
        int count = 0;
        // 滑动窗口的思路
        int left = 0;
        int right = 0;

        while(right<s2.length()){
            // 开始滑动记录
            char c = s2.charAt(right);
            windows[c]++;
            // 记录是否
            if(need[c]>0&&need[c]>=windows[c]){
                count++;
            }

            // 开始缩放 查看长度
            while(right-left+1>=s1.length()){
                // 如果此时单词数满足
                if(count==s1.length()){
                    return true;
                }
                // 不符合条件开始缩
                c = s2.charAt(left);
                // 是否记录了
                if(need[c]>0&&need[c]>=windows[c]){
                    count--;
                }
                windows[c]--;
                left++;

            }
            // 继续走
            right++;
        }
        return false;
    }
}
```

Leetcode438 找到字符串中所有字母异位词

给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。

说明：

字母异位词指字母相同，但排列不同的字符串。
不考虑答案输出的顺序。
示例 1:

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" 的字母异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。

> 解题思路：还是类似于上面的

```
class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        //一个need数组 一个windows窗口 一个判断是否都包含了
        int[] need = new int[128];
        int[] windows = new int[128];
        int count = 0;
        // 开始记录
        for(char c:p.toCharArray()){
            need[c]++;
        }
        // 滑动窗口
        int left = 0;
        int right = 0;
        // 结果记录
        List<Integer> res  = new ArrayList<>();
        // 开始
        while(right<s.length()){
            char c = s.charAt(right);
            // 记录
            windows[c]++;
            // 是否有了
            if(need[c]>0&&need[c]>=windows[c]){
                count++;
            }
            // 什么条件缩放，长度限制
            while(right-left+1>=p.length()){
                // 看是否满足都在的元素
                if(count==p.length()){
                    res.add(left);
                }
                // 开始排出元素
                c = s.charAt(left);
                // 是否满足条件
                if(need[c]>0&&need[c]>=windows[c]){
                    count--;
                }
                windows[c]--;
                left++;
            }
            right++;
        }
        return res;
```

```
    }
}
```

连续单个字符串的问题最长长度问题

Leetcode003 无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1:

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。
示例 2:

输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

> 解题思路：用hashmap存储下标索引

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        // hashmap存储下标索引
        HashMap<Character,Integer> hashMap = new HashMap<>();
        int left = 0;
        int right = 0;
        int minLength = 0;
        while(right<s.length()){
            // 判断是否出现重复了
            if(hashMap.containsKey(s.charAt(right))){
                left = Math.max(left,hashMap.get(s.charAt(right))+1);
            }
            hashMap.put(s.charAt(right),right);
            minLength = Math.max(minLength,right-left+1);
            right++;
        }
        return minLength;
    }
}
```

Leetcode674 最长连续递增序列

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。

连续递增的子序列 可以由两个下标 `l` 和 `r`（`l < r`）确定，如果对于每个 `l <= i < r`，都有 `nums[i] < nums[i + 1]` ，那么子序列 `[nums[l], nums[l + 1], ..., nums[r - 1], nums[r]]` 就是连续递增子序列。

示例 1：

```
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。
```

```java
class Solution {
    public int findLengthOfLCIS(int[] nums) {
        if(nums.length==0){
            return 0;
        }
        int left = 0;
        int right = 1;
        int n = nums.length;
        int maxLength = 1;
        while(right<n){
            // 开始缩不符合条件
            if(nums[right]<=nums[right-1]){
                left = right;
            }
            maxLength = Math.max(maxLength,right-left+1);
            right++;
        }
        return maxLength;
    }
}
```

Leetcode485 最大连续1的个数

给定一个二进制数组， 计算其中最大连续 1 的个数。

示例：

输入：[1,1,0,1,1,1]
输出：3
解释：开头的两位和最后的三位都是连续 1 ，所以最大连续 1 的个数是 3.

> 连续1的个数

```java
class Solution {
    public int findMaxConsecutiveOnes(int[] nums) {
        int left = 0;
        int right = 0;
        int n = nums.length;
        // 记录连续的最长长度
        int maxLength = 0;
        while(right<n){
            // 开始缩
            if(nums[right]!=1){
                left=right+1;
            }
            maxLength = Math.max(maxLength,right-left+1);
            right++;
        }
        return maxLength;
    }
```

```
}
```

连续单个字符可以给定操作次数来求最长长度

Leetcode1004 最大连续1的个数III

给定一个由若干 0 和 1 组成的数组 A，我们最多可以将 K 个值从 0 变成 1 。

返回仅包含 1 的最长（连续）子数组的长度。

示例 1：

输入：A = [1,1,1,0,0,0,1,1,1,1,0], K = 2
输出：6
解释：
[1,1,1,0,0,1,1,1,1,1,1]
粗体数字从 0 翻转到 1，最长的子数组长度为 6。
示例 2：

输入：A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3
输出：10
解释：
[0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
粗体数字从 0 翻转到 1，最长的子数组长度为 10。

```
class Solution {
    public int longestOnes(int[] A, int K) {
        // 只有0和1的替换
        int left = 0;
        int right = 0;
        // 将0替换成1
        int n = A.length;
        // 记录最长长度
        int maxLength = 0;
        int zeros = 0;
        while(right<n){
            // 添加一个记录0
            if(A[right]==0){
                zeros++;
            }
            // 达到一定次数进行缩放
            while(zeros>K){
                // 左边开始排出去0
                if(A[left]==0){
                    zeros--;
                }
                left++;
            }

            maxLength = Math.max(maxLength,right-left+1);
            right++;
        }
        return maxLength;
```

```
    }
}
```

Leetcode1493 删掉一个元素以后全为1的最长子数组

给你一个二进制数组 nums ，你需要从中删掉一个元素。

请你在删掉元素的结果数组中，返回最长的且只包含 1 的非空子数组的长度。

如果不存在这样的子数组，请返回 0 。

提示 1：

输入：nums = [1,1,0,1]
输出：3
解释：删掉位置 2 的数后，[1,1,1] 包含 3 个 1 。

示例 2：

输入：nums = [0,1,1,1,0,1,1,0,1]
输出：5
解释：删掉位置 4 的数字后，[0,1,1,1,1,1,0,1] 的最长全 1 子数组为 [1,1,1,1,1] 。

> 滑动窗口，删掉可以想象成替换，无非就是最后求得时候再加上。对0的个数上的限制

```
class Solution {
    public int longestSubarray(int[] nums) {
        int left = 0;
        int right = 0;
        int maxLength = 0;
        int zeros  = 0;
        while(right<nums.length){
            if(nums[right]==0){
                zeros++;
            }
            // 判断
            while(zeros>1){
                if(nums[left]==0){
                    zeros--;
                }
                left++;
            }

            // 替换了之后，但是必须要删掉一个元素
            maxLength = Math.max(right-left,maxLength);
            right++;
        }
        return maxLength;
    }
}
```

替换之后的升级版，不止是对0和1替换，而是对数组中的字母。替换后的最长重复字符

给你一个仅由大写英文字母组成的字符串，你可以将任意位置上的字符替换成另外的字符，总共可最多替换 k 次。在执行上述操作后，找到包含重复字母的最长子串的长度。

注意：字符串长度 和 k 不会超过 104。

示例 1：

输入：s = "ABAB", k = 2
输出：4
解释：用两个'A'替换为两个'B',反之亦然。

示例 2：

输入：s = "AABABBA", k = 1
输出：4
解释：
将中间的一个'A'替换为'B',字符串变为 "AABBBBA"。
子串 "BBBB" 有最长重复字母, 答案为 4。

> 替换字符

```
class Solution {
    public int characterReplacement(String s, int k) {
        // 替换跟数字那道题很相似
        // 记录当前窗口出现的字母频率数
        int[] windows = new int[128];
        // 滑动窗口
        int left = 0;
        int right = 0;
        char[] arr = s.toCharArray();
        // 记录之前出现的最大
        int cur = 0;
        int minLength = 0;
        while(right<arr.length){
            //记录窗口中字母出现的频率数
            windows[arr[right]]++;
            // 比较之前出现的，记录出现的最大频率数
            cur = Math.max(cur,windows[arr[right]]);
            // 用当前窗口的大小-减去出现的最大频率字母=k
            while(right-left+1-cur>k){
                // 开始缩
                windows[arr[left]]--;
                left++;
            }

            // 记录
            minLength = Math.max(minLength,right-left+1);
            right++;
        }
        return minLength;
    }
}
```

连续数组不超过代价的问题

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 。

示例 1：

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。

示例 2：

输入：target = 4, nums = [1,4,4]
输出：1

示例 3：

输入：target = 11, nums = [1,1,1,1,1,1,1,1]
输出：0

> 解题思路：连续子数组，其中的长度不能超过target，求其长度

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int left = 0;
        int right = 0;
        int minLength = nums.length+1;
        // 加个限制条件
        int temp_sum = 0;
        while(right<nums.length){
            temp_sum += nums[right];
            // 查看是否符合条件了
            while(temp_sum>=target){
                // 记录
                minLength = Math.min(minLength,right-left+1);

                // 开始移动左边
                temp_sum -= nums[left];
                left++;
            }
            right++;
        }
        return minLength==nums.length+1?0:minLength;
    }
}
```

Leetcode1208 尽可能使字符串相等

给你两个长度相同的字符串，s 和 t。

将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。

用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。

如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。

如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。

示例 1：

输入：s = "abcd", t = "bcdf", maxCost = 3
输出：3
解释：s 中的 "abc" 可以变为 "bcd"。开销为 3，所以最大长度为 3。

> 滑动窗口，增加代价，对应判断即可了

```
class Solution {
    public int equalSubstring(String s, String t, int maxCost) {
        // 计算代价
        char[] s_arr = s.toCharArray();
        char[] t_arr = t.toCharArray();
        int len1 = s.length();
        int len2 = t.length();
        if(len1!=len2){
            return 0;
        }
        int[] cost = new int[len1];
        for(int i=0;i<len1;i++){
            cost[i] = Math.abs(s_arr[i]-t_arr[i]);
        }
        // 滑动窗口计算
        int left = 0;
        int right = 0;
        // 计算代价，
        int temp_cost = 0;
        // 计算长度
        int maxLength = 0;
        while(right<len1){
            temp_cost += cost[right];
            // 判断是否缩放
            while(temp_cost>maxCost){
                temp_cost-=cost[left];
                left++;
            }
            maxLength = Math.max(maxLength,right-left+1);
            right++;
        }
        return maxLength;

    }
}
```

Leetcode1052 爱生气的书店老板

今天，书店老板有一家店打算试营业 customers.length 分钟。每分钟都有一些顾客（customers[i]）会进入书店，所有这些顾客都会在那一分钟结束后离开。

在某些时候，书店老板会生气。 如果书店老板在第 i 分钟生气，那么 grumpy[i] = 1，否则 grumpy[i] = 0。 当书店老板生气时，那一分钟的顾客就会不满意，不生气则他们是满意的。

书店老板知道一个秘密技巧，能抑制自己的情绪，可以让自己连续 X 分钟不生气，但却只能使用一次。

请你返回这一天营业下来，最多有多少客户能够感到满意的数量。

示例：

输入：customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], X = 3
输出：16
解释：
书店老板在最后 3 分钟保持冷静。
感到满意的最大客户数量 = 1 + 1 + 1 + 1 + 7 + 5 = 16.

> 解题思路：先计算不生气，后计算生气中的窗口内的最大值

```
class Solution {
    public int maxSatisfied(int[] customers, int[] grumpy, int X) {
        // 先计算正常的
        int sum1 = 0;
        // 遍历
        for(int i=0;i<customers.length;i++){
            // 如果此时不生气
            if(grumpy[i]==0){
                sum1 += customers[i];
                // 计算完成将其顾客数量设置为0
                customers[i] = 0;
            }
        }
        // 继续计算
        int left = 0;
        int right = 0;
        int sum_2 = 0;
        // 维护一个窗口内的值
        int temp_sum = 0;
        while(right<customers.length){
            // 开始计算
            temp_sum += customers[right];
            // 长度超过限制了
            while(right-left+1>X){
                temp_sum -= customers[left];
                left++;
            }
            // 维护
            sum_2 = Math.max(sum_2,temp_sum);
            right++;
        }
        return sum1+sum_2;
    }
}
```