滑动窗口

循环不变量

循环不变量

循环前、中、后保持不变
「循环不变量」不是很高深的概念，在「算法」和「数据结构」的世界里，到处都有它的身影。

「循环不变量」是指我们在编写代码的过程中，要一直循序不变的性质，这样的性质是根据要解决的问题，由我们自己定义的。「循环不变量」是我们写对一个问题的基础，保证了在「初始化」「循环遍历」「结束」这三个阶段相同的性质，使得一个问题能够被正确解决。

例题

删除有序数组中的重复项

最长连续递增序列（滑动窗口、DP）

> 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。

思路分析：题目要求我们找的子序列是 连续 的，并且子序列里的元素要求 严格单调递增。在遍历的时候，从第 2 个元素开始；

如果当前遍历到的元素比它左边的那一个元素要严格大，「连续递增」的长度就加 1；
否则「连续递增」的起始位置就需要重新开始计算。

总结

区间不同的定义决定了不同的初始化逻辑、遍历过程中的逻辑。 [i, j]和[i, j)

练习

移除元素

删除排序数组中的重复项 II

移动零

使用循环不变量写对代码

颜色分类

Dijsktra快速排序

对于pivot将元素划分成<pivot的部分 >pivot的部分 和==pivot的部分

```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        quickSort(nums, 0, nums.size()-1);
        return nums;
    }

    void quickSort(vector<int> &nums, int low, int high){
        if (low < high){
            int pos = partition(nums, low, high);
            quickSort(nums, low, pos-1);
            quickSort(nums, pos+1, high);
        }
    }

    int partition(vector<int> &nums, int low, int high){
        int rand_pos = rand() % (high-low+1) + low;
        swap(nums[low], nums[rand_pos]);
        int pivot = nums[low];
        /*
        *   Dijsktra快速排序
        *   nums[low...l] <pivot, nums[r...high]  >pivot, nums(l, i) == pivot
        *   nums[i...r) hasn't visited
        */
        int l = low-1, r = high+1, i = low;
        while (i < r){
            if (nums[i] < pivot){
                swap(nums[++l], nums[i]);
                i++;
            }else if (nums[i] == pivot){
                i++;
            }else{
                swap(nums[--r], nums[i]);
            }
        }
        return i-1;
    }
};
```

1、闭区间

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        //nums[0...lo] -> 0, nums[hi...n-1] -> 2, nums(lo...i) ->1, nums[i...hi]
还未遍历
        int lo = -1, hi = nums.size(), i = 0;
        while (i < hi){
            if (nums[i] == 0){
                swap(nums[++lo], nums[i]);
                i++;
```

```
            }else if (nums[i] == 1){
                i++;
            }else{
                swap(nums[--hi], nums[i]);
            }
        }
    }
};
```

2、开区间

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        //nums[0...lo) -> 0, nums(hi...n-1] -> 2, nums[lo...i) ->1, nums[i...hi)
还未遍历
        int lo = 0, hi = nums.size()-1, i = 0;
        while (i <= hi){
            if (nums[i] == 0){
                swap(nums[lo++], nums[i]);
                i++;
            }else if (nums[i] == 1){
                i++;
            }else{
                swap(nums[hi--], nums[i]);
            }
        }
    }
};
```

例题:

数组中的第 K 个最大元素

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        int n = nums.size();
        return partition(nums, 0, n-1, n-k);
    }

    int partition(vector<int> &nums, int low, int high, int k){
        int temp_low = low, temp_high = high;
        int rand_pos = rand() % (high-low + 1) + low;
        swap(nums[low], nums[rand_pos]);
        int pivot = nums[low];
        /*
        *   Dijsktra快速排序
        *   nums[low...l] <pivot, nums[r...high]  >pivot, nums(l, i) == pivot
        *   nums[i...r] hasn't visited
        */
        int l = low-1, r = high+1, i = low;
        while (i < r){
            if (nums[i] < pivot){
                swap(nums[++l], nums[i]);
                i++;
            }else if (nums[i] == pivot){
```

```
                i++;
            }else{
                swap(nums[--r], nums[i]);
            }
        }
        if (i-1 == k)   return  pivot;
        else if (i-1 > k)   return partition(nums, temp_low, i-2, k);
        else    return partition(nums, i, temp_high, k);
    }
};
```

总结

循环不变量是人为定义的，无需记忆。

只要我们在编码的开始明确了我们对变量和区间的定义，写对代码就是水到渠成的事情了。

滑动窗口 1：同向交替移动的两个变量

例题

[子数组最大平均数 I]

> 给你一个由 `n` 个元素组成的整数数组 `nums` 和一个整数 `k` 。
>
> 请你找出平均数最大且 **长度为** `k` 的连续子数组，并输出该最大平均数。
>
> 任何误差小于 `10-5` 的答案都将被视为正确答案。
>
> **示例 1：**
>
> ```
> 输入：nums = [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
> ```

[爱生气的书店老板]

> 有一个书店老板，他的书店开了 n 分钟。每分钟都有一些顾客进入这家商店。给定一个长度为 n 的整数数组 customers ，其中 customers[i] 是在第 i 分钟开始时进入商店的顾客数量，所有这些顾客在第 i 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，若老板不生气则顾客是满意的。
>
> 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 minutes 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 。

示例 1：

输入：customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], minutes = 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.

可获得的最大点数

几张卡牌 排成一行，每张卡牌都有一个对应的点数。点数由整数数组 cardPoints 给出。

每次行动，你可以从行的开头或者末尾拿一张卡牌，最终你必须正好拿 k 张卡牌。

你的点数就是你拿到手中的所有卡牌的点数之和。

给你一个整数数组 cardPoints 和整数 k，请你返回可以获得的最大点数。

示例 1：

输入：cardPoints = [1,2,3,4,5,6,1], k = 3
输出：12
解释：第一次行动，不管拿哪张牌，你的点数总是 1 。但是，先拿最右边的卡牌将会最大化你的可获得点数。最优策略是拿右边的三张牌，最终点数为 1 + 6 + 5 = 12 。

定长子串中元音的最大数目

给你字符串 s 和整数 k 。

请返回字符串 s 中长度为 k 的单个子字符串中可能包含的最大元音字母数。

英文中的 元音字母 为（a, e, i, o, u）。

示例 1：

输入：s = "abciiidef", k = 3
输出：3
解释：子字符串 "iii" 包含 3 个元音字母。

将 x 减到 0 的最小操作数

给你一个整数数组 nums 和一个整数 x 。每一次操作时，你应当移除数组 nums 最左边或最右边的元素，然后从 x 中减去该元素的值。请注意，需要 修改 数组以供接下来的操作使用。

如果可以将 x 恰好 减到 0 ，返回 最小操作数 ；否则，返回 -1 。

示例 1：

输入：nums = [1,1,4,2,3], x = 5
输出：2
解释：最佳解决方案是移除后两个元素，将 x 减到 0 。

滑动窗口 2：不定长度的滑动窗口

有一类数组上的问题，需要使用两个指针变量（我们称为左指针和右指针），同向、交替向右移动完成任务。这样的过程像极了一个窗口在平面上滑动的过程，因此我们将解决这一类问题的算法称为「滑动窗口」问题。

掌握好这一类「滑动窗口」的问题，需要先从「暴力解法」开始分析，「滑动窗口」利用了问题本身的特点，在两个指针同向、交替向右移动的过程中，少考虑了很多「暴力解法」需要考察了情况，将时间复杂度降到了线性级别 $O(N)$

（这里 N 是数组的长度），如下图所示。

写对「滑动窗口」除了要弄清楚为什么可以使用滑动窗口，还需要明白代码编写过程中的「循环不变量」，这样才不会在初始化和一些边界问题上出错。我们会在例题讲解的部分和大家进行说明。

例题

[最小覆盖子串（hard）]

> 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：
>
> 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
> 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"
> 解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。

滑动窗口

一开始的时候，left 和 right 都位于下标 0 的位置。right 向右移动，直至包含 T 的所有字母。由于我们要求的是最小子串，因此，以 left 开头的子串 `[left..right + 1]`、`[left..right + 2]`、……、`[left..len - 1]` 一定不符合要求，因此这些区间可以不用判断；然后考虑 left 如何移动。此时 left 不能向左移动，向左移动只能让子串更长，我们要求最小子串，因此 left 只能右移，移到恰恰好 [left..right] 区间里面的字符不包含 T 所有字母的最小子串；
然后 right 继续向右移动，直到包含 T 所有字母的最小子串。
重复这样的过程，直到 right 到达 S 的末尾。

「滑动窗口」算法有下面的特点：

right 先向右移动，移到不能再移动的时候，left 再向右移动；
right 右移使得滑动窗口边长，刚好满足条件，left 右移使得滑动窗口变短到刚好不满足条件，然后 right 变长刚好满足条件，如此循环下去，直到 right 到达末尾。这里的条件是指：[left, right) 包含 T 所有字母。
那么如何判断区间 [left, right] 内包含 T 所有字母呢？由于我们并不关心字母的顺序，**因此我们采用的是对比频数数组的方式。**

先对 T 做频数统计，然后设置一个变量 **distance** 表示 T 中一共有多少个不同的字母；
left 和 right 在动的时候，只对 T 中出现的字母做统计；
right 移动的时候，频数增加，加到刚刚好和 T 对应字母相等的时候，distance - 1，表示滑动窗口内的字母种类与 T 的差距减少了 1，当这个差距为 0 的时候，滑动窗口内包含 T 所有字母的最小子串。此时考虑移动 left；
left 移动的时候，做减法，减少到刚刚好比 T 中对应字符个数少 1 的时候，就说明「平衡」被打破，此时应该 right 继续向右移动。

1、在最后一个用例超时

```
class Solution {
public:
    string minWindow(string s, string p) {
        string res = "";
        int hs[128] = {0}, hp[128] = {0};
        for (char ch : p)   hp[ch]++;
        int distance =  0;
        for (int i = 0; i < 128; i++){
            if (hp[i])  distance++;
        }
        int match = 0;
        for (int i = 0, j = 0; j < s.size(); j++){
            int idx = s[j];
            if (++hs[s[j]] == hp[s[j]])
                match++;
            while (match == distance){
                if (res.empty() || j-i+1 < res.size())
                    res = s.substr(i, j-i+1);
                if (--hs[s[i]] < hp[s[i]])
                    match--;
                i++;
            }
        }
        return res;
    }
};
```

match维护的是`s[i...j]`中和t匹配的字符数

2、当`hs[s[i]] > hp[s[i]]`时将i不断右移

在这里需要注意的是，如果先判断`s[i]`是否出现在p中，再将`hp[s[i]]++`会导致问题

因为没在p中出现的字符对应的`hs[s[i]]`值为0，那么当前字符只出现在s中而未出现在p中，i也应该继续向右移动

修改`hs[s[i]]++`的原则，无论`s[i]`是否在p中出现，我们都将`hs[s[i]]++`

```
class Solution {
public:
    string minWindow(string s, string t) {
        string res = "";
        int n = s.size(), m = t.size();
        int hs[128] = {0}, ht[128] = {0};
        for (char ch : t)   ht[ch]++;
        int distance = 0;   //统计distance中出现的字符数
        for (int i : ht) {
            if (i) distance++;
        }
        int match = 0;      //字符串s中[i...j]中和t匹配的字符数
        for (int i = 0, j = 0; j < n; j++){
            int idx = s[j];
            hs[idx]++;
            //当字符在t中出现并且在s[i...j]中出现的次数等于在t中出现的次数
            if (ht[idx] && hs[idx] == ht[idx])
                match++;
            //不断将滑动窗口的左边界向右移动
            while (i < j && hs[s[i]] > ht[s[i]]){
                hs[s[i]]--;
                i++;
            }
            if (match == distance){
                if (res.empty() || j-i+1 < res.size())  //更新res
                    res = s.substr(i, j-i+1);
            }
        }
        return res;
    }
};
```

match维护到的是s[o...j]中和t匹配的字符数，但是每次不断将i进行右移，使得i是符合匹配的最右下标

替换后的最长重复字符

> 给你一个字符串 s 和一个整数 k 。你可以选择字符串中的任一字符，并将其更改为任何其他大写英文字符。该操作最多可执行 k 次。
>
> 在执行上述操作后，返回包含相同字母的最长子字符串的长度。
>
> **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。

滑动窗口

```
class Solution {
public:
    int characterReplacement(string s, int k) {
        int cntMax = 0, res = 0;  //cntMax统计在滑动窗口内字符的最多出现次数
        int count[26] = {0};
        for (int i = 0, j = 0; j < s.size(); j++){
            count[s[j] - 'A']++;
            cntMax = max(cntMax, count[s[j] - 'A']);
            while (j-i+1 > cntMax+k){  //滑动窗口内最多可以容纳cntMax个相同的字符和k个不同的字符
                count[s[i] - 'A']--;
                i++;
            }
            res = max(res, j-i+1);
        }
        return res;
    }
};
```

总结

滑动窗口最重要的就是循环不变量的定义，如何在遍历过程中去维护滑动窗口

「滑动窗口」是一类通过使用两个变量在数组上同向交替移动解决问题的算法。这一类问题的思考路径通常是：先思考暴力解法，分析暴力解法的缺点（一般而言暴力解法的缺点是重复计算，包括哪些不会是正确答案的解也会计算），然后 结合问题的特点，使用「双指针」技巧对暴力解法进行剪枝。因此，思考算法设计的合理性是更关键的，这一点适用于所有算法问题。

left 和 right 同方向移动；
定义条件，即我们需要时刻检测的一件事情；
原理：充分利用本题本身的特点，以减少不必要的计算；
利用「循环不变量」保证代码边界正确；
不要记忆代码模板，应该结合具体问题分析出什么时候滑动窗口最长，什么时候滑动窗口最短；
掌握处理字符串的技巧。

基础练习

无重复字符的最长子串

长度最小的子数组

删除子数组的最大得分

下面三题题目描述和实现起来差不多，但有一些细节不同

找到字符串中所有字母异位词

（1）match统计s[0...j]中匹配的字符数

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
```

```
        vector<int> res;
        int hs[26] = {0}, hp[26] = {0};
        for (char ch : p)   hp[ch-'a']++;
        int distance = 0, match = 0;
        for (int i : hp){
            if (i)  distance++;
        }
        for (int i = 0, j = i; j < s.size(); j++){
            if (++hs[s[j]-'a'] == hp[s[j]-'a'])
                match++;
            while (i < j && hs[s[i]-'a'] > hp[s[i]-'a']){
                --hs[s[i++]-'a'];
            }
            if (match == distance){
                if (j-i+1 == p.size())
                    res.push_back(i);
            }
        }
        return res;
    }
};
```

（2）math统计s[i...j]中匹配的字符数

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        vector<int> res;
        int hs[26] = {0}, hp[26] = {0};
        for (char ch : p)   hp[ch-'a']++;
        int distance = 0, match = 0;
        for (int i : hp){
            if (i)  distance++;
        }
        for (int i = 0, j = i; j < s.size(); j++){
            if (++hs[s[j]-'a'] == hp[s[j]-'a'])
                match++;
            while (i < j && hs[s[i]-'a'] > hp[s[i]-'a']){
                if (--hs[s[i]-'a'] < hp[s[i]-'a'])
                    match--;
                i++;
            }
            if (match == distance){
                if (j-i+1 == p.size())
                    res.push_back(i);
            }
        }
        return res;
    }
};
```

类似题目：**最小覆盖子串**、**最小窗口子序列**

[字符串的排列]

[最大连续1的个数 II]

[最大连续 1 的个数 III]

[尽可能使字符串相等]

[删掉一个元素以后全为 1 的最长子数组]

[爱生气的书店老板]

[最长湍流子数组]

> 给定一个整数数组 `arr` ，返回 `arr` 的 最大湍流子数组的**长度** 。
>
> 如果比较符号在子数组中的每个相邻元素对之间翻转，则该子数组是 **湍流子数组** 。
>
> 更正式地来说，当 `arr` 的子数组 `A[i], A[i+1], ..., A[j]` 满足仅满足下列条件时，我们称其为湍流子数组：
>
> - 若 i <= k < j ：
>   - 当 `k` 为奇数时， `A[k] > A[k+1]`，且
>   - 当 `k` 为偶数时，`A[k] < A[k+1]`；
> - 或 若 i <= k < j：
>   - 当 `k` 为偶数时，`A[k] > A[k+1]` ，且
>   - 当 `k` 为奇数时， `A[k] < A[k+1]`。
>
> **示例 1：**
>
> ```
> 输入：arr = [9,4,2,10,7,8,8,1,9]
> 输出：5
> 解释：arr[1] > arr[2] < arr[3] > arr[4] < arr[5]
> ```
>
> **示例 2：**
>
> ```
> 输入：arr = [4,8,12,16]
> 输出：2
> ```
>
> **示例 3：**
>
> ```
> 输入：arr = [100]
> 输出：1
> ```

分类讨论 + 滑动窗口

```
class Solution {
public:
    int maxTurbulenceSize(vector<int>& arr) {
        if (arr.size() == 1)
            return 1;
        int n = arr.size();
        int res = 0, pre = 2;
        //这里假设arr[j] > == < arr[j-1]时，pre的值分别问1 0 -1
        for (int i = 0, j = 1; j < n; j++){
```

```
            if (arr[j] == arr[j-1])   {
                i = j;
                pre = 0;
            }else if (arr[j] < arr[j-1]){
                if (pre == -1)   i = j-1;
                pre = -1;
            }else if (arr[j] > arr[j-1]){
                if (pre == 1)   i = j-1;
                pre = 1;
            }
            res = max(res, j-i+1);
        }
        return res;
    }
};
```

进阶练习

[K 连续位的最小翻转次数]

> 给定一个二进制数组 `nums` 和一个整数 `k` 。
>
> **k位翻转** 就是从 `nums` 中选择一个长度为 `k` 的 **子数组** ，同时把子数组中的每一个 `0` 都改成 `1` ，把子数组中的每一个 `1` 都改成 `0` 。
>
> 返回数组中不存在 `0` 所需的最小 **k位翻转** 次数。如果不可能，则返回 `-1` 。
>
> **子数组** 是数组的 **连续** 部分。
>
> **示例 1：**
>
> ```
> 输入：nums = [0,1,0], K = 1
> 输出：2
> 解释：先翻转 A[0]，然后翻转 A[2]。
> ```
>
> **示例 2：**
>
> ```
> 输入：nums = [1,1,0], K = 2
> 输出：-1
> 解释：无论我们怎样翻转大小为 2 的子数组，我们都不能使数组变为 [1,1,1]。
> ```
>
> **示例 3：**
>
> ```
> 输入：nums = [0,0,0,1,0,1,1,0], K = 3
> 输出：3
> 解释：
> 翻转 A[0],A[1],A[2]: A变成 [1,1,1,1,0,1,1,0]
> 翻转 A[4],A[5],A[6]: A变成 [1,1,1,1,1,0,0,0]
> 翻转 A[5],A[6],A[7]: A变成 [1,1,1,1,1,1,1,1]
> ```
>
> **提示：**

- $1 <= nums.length <= 10^5$
- `1 <= k <= nums.length`

（1）贪心模拟翻转

每遇到一个0就将以0为开始的k个字符进行翻转

```
class Solution {
public:
    int minKBitFlips(vector<int>& nums, int k) {
        int res = 0;
        for (int i = 0; i < nums.size()-k+1; i++){
            if (nums[i] == 0){
                for (int j = i; j < i+k; j++){
                    nums[j] ^= 1;
                }
                res++;
            }
        }
        for (int i = 0; i < nums.size(); i++){
            if (!nums[i])   return -1;
        }
        return res;
    }
};
```

时间复杂度O(n * k)，对于某个数，可能被前面的k-1个数翻转了多次

（2）滑动窗口

翻转的性质：

- 对于`nums[i]`而言，只受到前面k-1个元素的影响
- `nums[i]`翻转偶数次，元素值不变，而奇数次后，`nums[i] ^= 1`

如果可以确定前面k-1个元素中有多少个元素发生了翻转，那么在根据`nums[i]`的值来确定当前元素是否需要翻转

用队列q来存储`i`前面需要翻转的元素个数

```
if (q.size() % 2)
    nums[i]被翻转了（nums[i] == 0时元素不需要翻转，nums[i] == 1时，收到前面翻转的影响当前变成了0，那么当前元素需要翻转）
else
    nums[i]的值不变nums[i] == 1时元素不需要翻转，nums[i] == 0时，当前元素需要翻转）
```

综上，当`q.size() == nums[i]`时元素需要翻转

```
class Solution {
public:
    int minKBitFlips(vector<int>& nums, int k) {
        queue<int> q;
        int res = 0;
        for (int i = 0; i < nums.size(); i++){
            while (q.size() && i-q.front() >= k)
                q.pop();
```

```
            if (q.size()%2 == nums[i]){
                if (i+k > nums.size())    return -1;
                q.push(i);
                res++;
            }
        }
        return res;
    }
};
```

（3）利用前缀和

```
class Solution {
public:
    int minKBitFlips(vector<int>& nums, int k) {
        /*
        1、贪心模拟翻转，时间复杂度O(n * k)
        2、滑动窗口
        首先利用队列q来记录需要翻转的元素，对于nums[i]而言，如果q里有偶数个元素，那么当前nums[i]经过翻转之后元素值不变（即nums[i] == 0时需要加入队列，进行翻转），否则经过反转之后得元素值会改变（即nums[i] == 1是，需要进入队列）
        将上面两个写成一个：q.size()%2 == nums[i]时需要进行翻转
        对于需要翻转的元素nums[i]，如果i+k > nums.size()（后面没有k-1个元素进行翻转，返回-1）
        3、前缀和
        f[i]表示f[i...i+k-1]需要进行翻转
        对于nums[i]而言，是否需要进行翻转取决于前面nums[i-k+1...i-1]个元素的翻转次数，如果前面的翻转次数是奇数并且nums[i] == 1，或者前面的翻转次数是偶数nums[i] == 0，当前nums[i]需要进行翻转
        对于区间求和问题，可以利用前缀和数组在O(1)的时间内进行计算
        */
        int n = nums.size();
        vector<int> f(n+1, 0), sum(n+1, 0);
        int res = 0;
        for (int i = 1; i+k <= n+1; i++){
            int t = sum[i-1] - (i>=k ? sum[i-k] : 0);    //t是f[i-k+1...i-1]的前缀和
            if ((t+nums[i-1])%2 == 0){                   //需要进行翻转
                res++;
                f[i] = 1;
            }
            sum[i] = sum[i-1] + f[i];
        }
        for (int i = n-k+2; i <= n; i++){
            int t = sum[i-1] - (i>=k ? sum[i-k] : 0);    //t是f[i-k+1...i-1]的前缀和
            if ((t+nums[i-1])%2 == 0){                   //最后一组元素也需要进行翻转
                return -1;
            }
            sum[i] = sum[i-1] + f[i];
        }
        return res;
    }
};
```

[最小窗口子序列]

和76最小覆盖子串很像，但题目要求字母按顺序出现

```
class Solution {
public:
    string minWindow(string s1, string s2) {
        string res = "";
        for (int i = 0, j = 0, k = 0; j < s1.size(); j++){
            if (s1[j] == s2[k]) k++;
            if (k == s2.size()){ //窗口的左边界右移
                //收缩窗口的右边界，从s2末尾开始匹配s1
                i = j;
                for (k = s2.size()-1; k >= 0; i--){
                    if (s1[i] == s2[k]) k--;
                    if (k < 0) break;
                }
                k = 0;
                // cout << i << " " << j << endl;
                if (res.empty() || j-i+1 < res.size())
                    res = s1.substr(i, j-i+1);
                j = i+1;    //如果缺少这行代码会出错，窗口的右边界左移？？？
            }
        }
        return res;
    }
};
```

[长度为 K 的无重复字符子串（固定长度的滑动窗口）]

[最少交换次数来组合所有的 1]

[健身计划评估]

滑动窗口 3：计数问题选讲

例题

[至多包含两个不同字符的最长子串]

[至多包含 K 个不同字符的最长子串]

[区间子数组个数]

> 给你一个整数数组 `nums` 和两个整数：`left` 及 `right` 。找出 `nums` 中连续、非空且其中最大元素在范围 `[left, right]` 内的子数组，并返回满足条件的子数组的个数。
>
> 生成的测试用例保证结果符合 **32-bit** 整数范围。
>
> **示例 1：**

```
输入：nums = [2,1,4,3], left = 2, right = 3
输出：3
解释：满足条件的三个子数组：[2], [2, 1], [3]
```

示例 2：

```
输入：nums = [2,9,2,5,6], left = 2, right = 8
输出：7
```

提示：

- $1 <= nums.length <= 10^5$
- $0 <= nums[i] <= 10^9$
- $0 <= left <= right <= 10^9$

求最大值位于[left, right]的子数组个数，可以将问题转换成求最大值<=right的子区间的个数 - 最大值<left-1的子区间个数

```cpp
class Solution {
public:
    int numSubarrayBoundedMax(vector<int>& nums, int left, int right) {
        //类似于前缀和，在[left...right]的区间数 = ( ≤right的区间数 - ≤ left-1的区间数)

        return numSubarray(nums, right) - numSubarray(nums, left-1);
    }

    int numSubarray(vector<int> &nums, int k){
        //利用滑动窗口求nums中<=k的区间数
        int res = 0;
        for (int i = 0, j = 0; j < nums.size(); j++){
            if (nums[j] > k){
                i = j+1;
                continue;
            }
            res += j-i+1;
        }
        return res;
    }
};
```

K 个不同整数的子数组

和上面一题类似，首先需要将问题转换

练习

乘积小于 K 的子数组

注意一些special case：[1 1 1] 1，[1 2 3] 0

水果成篮

包含所有三种字符的子字符串数目

环绕字符串中唯一的子字符串 dp

23. **合并K个升序链表**

378. 有序矩阵中第 K 小的元素

找出第 k 小的距离对

```
class Solution {
public:
    int smallestDistancePair(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        int lo = 0, hi = nums[n-1] - nums[0];
        while (lo < hi){
            int mid = lo + (hi - lo) / 2;
            if (getCount(nums, mid) >= k)   hi = mid;
            else    lo = mid+1;
        }
        return lo;
    }

    int getCount(vector<int> &nums, int mid){
        //利用双指针计算nums中<=mid的元素个数，维持窗口内的元素差值<=mid
        int cnt = 0;
        for (int i = 0, j = 1; j < nums.size(); j++){
            while (nums[j] - nums[i] > mid)
                i++;
            cnt += j-i;    //计算数对
        }
        return cnt;
    }
};
```

滑动窗口 4：使用数据结构维护窗口性质

滑动窗口最大值（最小值）

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int N = 1e6+10;
int nums[N];
```

```
int main(){
    int n, k;
    cin >> n >> k;
    for (int i = 0; i < n; i++)
        cin >> nums[i];
    deque<int> q;   //维护一个严格单调递增的滑动窗口（求最小值）
    for (int i = 0; i < k; i++){
        while (q.size() && nums[i] <= nums[q.back()])
            q.pop_back();
        q.push_back(i);
    }
    cout << nums[q.front()] << " ";
    for (int i = k; i < n; i++){
        while (q.size() && nums[i] <= nums[q.back()])
            q.pop_back();
        while (q.size() && i-q.front() >= k)
            q.pop_front();
        q.push_back(i);
        cout << nums[q.front()] << " ";
    }
    cout << endl;
    q.clear();

    //求窗口内的最小值，维护一个严格递减的滑动窗口
    for (int i = 0; i < k; i++){
        while (q.size() && nums[i] >= nums[q.back()])
            q.pop_back();
        q.push_back(i);
    }
    cout << nums[q.front()] << " ";
    for (int i = k; i < n; i++){
        while (q.size() && nums[i] >= nums[q.back()])
            q.pop_back();
        while (q.size() && i-q.front() >= k)
            q.pop_front();
        q.push_back(i);
        cout << nums[q.front()] << " ";
    }
    cout << endl;
    return 0;
}
```

295. 数据流的中位数

例题：滑动窗口中位数

```
class Solution {
    priority_queue<int> queMin; //默认是大根堆
    priority_queue<int, vector<int>, greater<int>> queMax;
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        unordered_map<int, int> mp;
        vector<double> res(n-k+1);
        for (int i = 0; i < k; i++) queMin.push(nums[i]);
        for (int i = 0; i < k/2; i++){
            queMax.push(queMin.top());
```

```
            queMin.pop();
        }
        res[0] = getMedian(k);
        for (int i = k; i < n; i++){
            int le = nums[i-k];    //需要被移出滑动窗口的元素
            mp[le]++;

            //为了之中让queMin.size() == queMax.size() ||   queMin.size() ==
queMax.size()+1
            int balance = 0;
            if (queMin.size() && le <= queMin.top())
                balance--;
            else
                balance++;
            if (queMax.size() && nums[i] >= queMax.top()){
                queMax.push(nums[i]);
                balance--;
            }else{
                queMin.push(nums[i]);
                balance++;
            }
            //balance原本是0经过上述调整之后balance只可能是0、2、-2这三个数
            if (balance > 0){   //左边比右边多两个元素
                queMax.push(queMin.top());
                queMin.pop();
            }else if (balance < 0){
                queMin.push(queMax.top());
                queMax.pop();
            }

            while (queMin.size() && mp.find(queMin.top()) != mp.end()){
                if (--mp[queMin.top()] == 0)
                    mp.erase(queMin.top());
                queMin.pop();
            }
            while (queMax.size() && mp.find(queMax.top()) != mp.end()){
                if (--mp[queMax.top()] == 0)
                    mp.erase(queMax.top());
                queMax.pop();
            }
            res[i-k+1] = getMedian(k);
        }
        return res;
    }

    double getMedian(int k){
        if (k%2 == 0){
            return ((long long)queMin.top() + queMax.top()) / 2.0;
        }else
            return queMin.top();
    }
};
```

220. 存在重复元素 III

K 个关闭的灯泡

绝对差不超过限制的最长连续子数组

1、滑动窗口+set

2、滑动窗口+单调队列

```
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        deque<int> queMax, queMin;
        int res = 0, n = nums.size();
        for (int i = 0, j = 0; j < nums.size(); j++){
            while (queMax.size() && nums[queMax.back()] < nums[j])  //queMax维护
一个单调递减的队列，队首元素就是最大值
                queMax.pop_back();
            while (queMin.size() && nums[queMin.back()] > nums[j])  //queMin维护
一个单调递增的队列，队首元素就是最小值
                queMin.pop_back();
            queMin.push_back(j);
            queMax.push_back(j);

            while (queMin.size() && queMax.size() && nums[queMax.front()] -
nums[queMin.front()] > limit){
                if (i == queMin.front())  queMin.pop_front();
                if (i == queMax.front())  queMax.pop_front();
                i++;
            }
            res = max(res, j-i+1);
        }
        return res;
    }
};
```

双指针

链表中的双指针问题

解决链表中的一些问题有些时候需要一些脑洞，并没有那么容易想到。好在这些问题只需要掌握这些常见的技巧就可以了。其中最典型的技巧就是「快慢指针」，也称为「同步指针」。事实上，解决它们都是在链表中使用了两个变量，因此也称为「双指针」技巧。

例题

环形链表

删除链表的倒数第 N 个结点

链表的中间结点

环形链表 II

对于链表题细节比较重要

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        /*
        *   双指针slow每次走一步，fast每次走两步
        *   假设链表的头结点和环的入口相差a个节点，环内的长度是b个节点
        *   第一次相遇时f = 2*s = s + n*b   (f时fast走的步数，s时slow走的步数，两个节点相遇fast比slow多走了n圈即n*b步)
        *   --->       f = 2 * n *b, s = n * b
        *   如果让fast从头走a步，s也走a步，那么f = a, s = a + n*b他们会在环的入口处相遇
        */
        ListNode *slow = head, *fast = head;
        while (fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) break;
        }
        //退出循环时，fast为最后一个节点（链表长度为奇数）,fast为空节点（链表长度为偶数）
        if (!fast || !fast->next)   return NULL;
        fast = head;
        while (fast && fast->next){
            if (slow == fast)   break;
            slow = slow->next;
            fast = fast->next;

        }
        return slow;
    }
};
```

相交链表

双指针：相向交替移动的两个变量

「双指针」是指通过两个变量交替相向移动完成任务的算法，具体来说，可以使用两个变量 i 和 j ，初始的时候，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 不断向右移动， 指针 j 不断向左移动，直到它们相遇。这样设计的算法少考虑了很多暴力解法需要考虑的情况，如下图所示。

例题

盛最多水的容器

两数之和 II - 输入有序数组

三数之和

排序+双指针

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> res;
        int n = nums.size();
        for (int i = 0; i+2 < n; i++){
            if (i > 0 && nums[i] == nums[i-1])  continue;   //不加上这个会在最后一个用例超时
            if (nums[i] + nums[i+1] + nums[i+2] > 0)    break;
            for (int j = i+1, k = nums.size()-1; j < k; ){

                int sum = nums[i] + nums[j] + nums[k];
                if (sum == 0)   res.push_back({nums[i], nums[j++], nums[k--]});
                else if (sum > 0)   k--;
                else   j++;
            }
        }
        sort(res.begin(), res.end());
        res.erase(unique(res.begin(), res.end()), res.end());
        return res;
    }
};
```

排序+二分

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> res;
        int n = nums.size();
        for (int i = 0; i+2 < n; i++){
            if (i > 0 && nums[i] == nums[i-1])  continue;
            if (nums[i] + nums[i+1] + nums[i+2] > 0)    break;
            for (int j = i+1; j+1 < n; j++){
                int taregt = -(nums[i] + nums[j]);
                int pos = lower_bound(nums.begin() + j+1, nums.end(), taregt) - nums.begin();
```

```
                if (pos < n && nums[pos] + nums[i] + nums[j] == 0)
                    res.push_back({nums[i], nums[j], nums[pos]});
            }
        }
        sort(res.begin(), res.end());
        res.erase(unique(res.begin(), res.end()), res.end());
        return res;
    }
};
```

验证回文串

反转字符串

反转字符串中的元音字母

练习

最接近的三数之和

```
class Solution {
public:
    int threeSumClosest(vector<int>& nums, int target) {
        int res = 0x3f3f3f3f, distance = 0x3f3f3f3f, n = nums.size();
        sort(nums.begin(), nums.end());
        for (int i = 0; i < n; i++){
            for (int j = i+1, k = n-1; j < k; ){
                int sum = nums[i] + nums[j] + nums[k];
                if (sum < target)   j++;
                else if (sum > target)  k--;
                else return target;
                if (abs(sum - target) < distance){
                    res = sum;
                    distance = abs(sum - target);
                }
            }
        }
        return res;
    }
};
```

四数之和 （数字之和超出int的范围）

接雨水

按列算雨水

```
class Solution {
public:
    int trap(vector<int>& height) {
        int res = 0, n = height.size();
        vector<int> left(n, 0), right(n, 0);
        left[0] = 0, right[n-1] = n-1;
        for (int i = 1; i < n; i++){
            left[i] = height[i] > height[left[i-1]] ? i : left[i-1];
        }
```

```
        for (int i = n-2; i >= 0; i--){
            right[i] = height[i] > height[right[i+1]] ? i : right[i+1];
        }
        for (int i = 1; i < n-1; i++){
            res += max(0, min(height[left[i]], height[right[i]]) - height[i]);
        }
        return res;
    }
};
```

使用了两个数组

改进：只使用一个数组

找到 K 个最接近的元素

```
class Solution {
public:
    vector<int> findClosestElements(vector<int>& arr, int k, int x) {
        /*
        *   x < arr[0]
        *   x > arr[n-1]
        *   x介于数组元素
        */
        vector<int> res;
        int pos = lower_bound(arr.begin(), arr.end(), x) - arr.begin(), n =
arr.size();
        if (pos == 0){
            for (int i = 0; i < k; i++) res.push_back(arr[i]);
        }else if (pos == n){
            for (int i = n-k; i < n; i++){
                res.push_back(arr[i]);
            }
        }else{
            for (int i = pos-1, j = pos; i >= 0 || j < n; ){
                if (res.size() >= k)    break;
                if (i >= 0 && j < arr.size()){
                    if (abs(arr[i] - x) <= abs(arr[j] - x)){
                        res.push_back(arr[i--]);
                    }else
                        res.push_back(arr[j++]);
                }else if (i >= 0){
                    res.push_back(arr[i--]);
                }else
                    res.push_back(arr[j++]);
            }
        }
        sort(res.begin(), res.end());
        return res;
    }
};
```

较小的三数之和

```
class Solution {
public:
    int threeSumSmaller(vector<int>& nums, int target) {
        sort(nums.begin(), nums.end());
        int res = 0, n = nums.size();
        for (int i = 0; i+2 < n; i++){
            if (nums[i] + nums[i+1] + nums[i+2] > target)
                break;
            int j = i+1, k= n-1;
            while (j < k){
                if (nums[i] + nums[j] + nums[k] < target){
                    res += k-j;
                    j++;
                }else
                    k--;
            }
        }
        return res;
    }
};
```

有序转化数组

数学题

```
class Solution {
public:
    vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {
        int n = nums.size();
        vector<int> res(n);
        int idx = a >= 0 ? n-1 : 0;
        auto fx = [&](const auto& x) {
            return a * x * x + b * x + c;
        };
        int i = 0, j = n-1;
        while (i <= j){
            if (a >= 0)
                res[idx--] = fx(nums[i]) >= fx(nums[j]) ? fx(nums[i++]) :
fx(nums[j--]);
            else
                res[idx++] = fx(nums[i]) <= fx(nums[j]) ? fx(nums[i++]) :
fx(nums[j--]);
        }
        return res;
    }


};
```

比较含退格的字符串

使用双指针空间复杂度O(1)，时间复杂度O(N + M)

利用skip来标记当前字符是否需要跳过，如果当前字符为#则skip++，i--

否则判断当前字符是否需要跳过，if skip > 0，那么skip--，i--

否则从当前字符开始匹配

```
class Solution {
public:
    bool backspaceCompare(string s, string t) {
        int i = s.size()-1, j = t.size()-1;
        int skipS = 0, skipT = 0;
        while (i >= 0 || j >= 0){
            while (i >= 0){
                if (s[i] == '#'){
                    skipS++, i--;
                }else if (skipS > 0){
                    skipS--, i--;
                }else
                    break;
            }

            while (j >= 0){
                if (t[j] == '#'){
                    skipT++, j--;
                }else if (skipT > 0){
                    skipT--, j--;
                }else
                    break;
            }

            if (i >= 0 && j >= 0){
                if (s[i] != t[j])   return false;
                // i--, j--;
            }else{
                if (i >= 0 || j >= 0)
                    return  false;
            }
            i--, j--;
        }
        return true;
    }
};
```

数组中的最长山脉

长按键入

有序数组的平方

小于 K 的两数之和

安排会议日程