**3. 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。

输入: "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。

输入: "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。

输入: "pwwkew"

输出: 3

解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

|  |
| --- |
| class Solution {  public:  int lengthOfLongestSubstring(string s) {  int res = 0;  int left = 0;  int right = 0;  map<char, int> window;  while (right < s.size()) {  char c = s[right];  window[c]++;  right++;  while (window[c] > 1) {  window[s[left]]--;  left++;  }  res = max(res, (right - left));  }  return res;  }  };  维护一个不含重复字符的滑动窗口。需要记录每个字符最后出现的位置，当遇到重复字符的时候，就把窗口首部调到上一次出现这个字符的下一个位置。 |

**76. 最小覆盖子串**

给你一个字符串 S、一个字符串 T 。请你设计一种算法，可以在 O(n) 的时间复杂度内，从字符串 S 里面找出：包含 T 所有字符的最小子串。

输入：S = "ADOBECODEBANC", T = "ABC"

输出："BANC"

|  |
| --- |
| class Solution {  public:  string minWindow(string s, string t) {  map<char, int> needed;  for (char c : t) {  needed[c]++;  }  int left = 0;  int right = 0;  int start = 0;  int windowSize = INT\_MAX;  int matched = 0;  map<char, int> window;  while (right < s.size()) {  char c = s[right];  if (needed.find(c) != needed.end()) {  window[c]++;  if (window[c] == needed[c]) {  matched++;  }  }  right++;  while (matched == needed.size()) {  if (right - left < windowSize) {  start = left;  windowSize = right - left;  }  char c = s[left];  left++;  if (window.find(c) != window.end()) {  window[c]--;  if (window[c] < needed[c]) {  matched--;  }  }  }  }  return windowSize == INT\_MAX ? "" : s.substr(start, windowSize);  }  };  滑动窗口题。维护一个包含t中所有字符的最小滑动窗口，首先用一个hashmap记录所有t中的字符和出现次数，在s中每遇到一次计数器加一，找到了符合条件的窗口后，尝试向右移动窗口左指针，直到恰好能够满足条件为止。更新当前最小滑动窗口。 |

**209. 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 s ，找出该数组中满足其和 ≥ s 的长度最小的 连续 子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

输入：s = 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。

|  |
| --- |
| class Solution {  public:  int minSubArrayLen(int s, vector<int>& nums) {  int left = 0;  int right = 0;  long long sum = 0;  int res = INT\_MAX;  while (right < nums.size()) {  sum += nums[right];  right++;  while (sum >= s) {  res = min(res, right - left);  sum -= nums[left];  left++;  }  }  return res == INT\_MAX ? 0 : res;  }  };  维护一个大于等于sum的最小滑动窗口。 |

**239. 滑动窗口最大值**

给定一个数组 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。

进阶：

你能在线性时间复杂度内解决此题吗？

输入: nums = [1,3,-1,-3,5,3,6,7], 和 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

思路：维护窗口，向右移动时左侧超出窗口的值弹出，因为需要的是窗口内的最大值，所以只要保证窗口内的值是递减的即可，小于新加入的值全部弹出。最左端即为窗口最大值

|  |
| --- |
| python解法：  class Solution(object):  def maxSlidingWindow(self, nums, k):  win, ret = [], []  for i, v in enumerate(nums):  if i >= k and win[0] <= i - k: win.pop(0)  while win and nums[win[-1]] <= v: win.pop() // 单调递减的队列  win.append(i)  if i >= k - 1: ret.append(nums[win[0]])  return ret |

|  |
| --- |
| class Solution {  public:  vector<int> maxSlidingWindow(vector<int>& nums, int k) {  vector<int> res;  deque<int> deque;  for (int i = 0; i < nums.size(); i++) {  if (!deque.empty() && deque.front() == i-k) deque.pop\_front();  while (!deque.empty() && nums[i] > nums[deque.back()]) deque.pop\_back();  deque.push\_back(i);  if (i >= k-1) res.push\_back(nums[deque.front()]);  }  return res;  }  }; |

**424. 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。

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。

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。

|  |
| --- |
| class Solution {  public:  int characterReplacement(string s, int k) {  vector<int> counts(26, 0); //记录当前窗口字母出现的个数  int left=0, res=0, maxCnt=0; // maxCnt记录字符出现次数最多字符的次数  for(int i=0; i< s.size(); i++){  counts[s[i]-'A']++;  maxCnt = max(maxCnt, counts[s[i]-'A']); // 比较之前最大数量和当前字符的数量  while(i-left+1-maxCnt > k){ // 若当前窗口大小 减去 窗口中最多相同字符的个数 大于 k 时  counts[s[left]-'A']--; // 将窗口最左边的字符 在计数数组中减1  left++; // 滑动窗口  }  res = max(res, i-left+1);  }  return res;  }  };  维护一个最多包含k个额外字符的滑动窗口。需要记录当前出现次数最多字符的出现次数来判断窗口是否合法，如果超过了，就把首指针向后挪一位，同时更新最多出现次数。对每个合法窗口，取其中的最大值。 |

**438. 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。

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。

|  |
| --- |
| class Solution {  public:  vector<int> findAnagrams(string s, string p) {  vector<int> pCounter(26, 0);  for(auto &c : p){  pCounter[c-'a']++;  }  vector<int> res;  vector<int> sCounter(26, 0);  int l = 0;  for(int r = 0; r < s.length(); r++){  sCounter[s[r]-'a']++;  if(r < p.length()-1) continue;  **if(pCounter == sCounter)** res.push\_back(l);  sCounter[s[l]-'a']--;  l++;  }  return res;  }  };  主要思路是维护两个hashmap,一个记录期望出现的字符，一个记录当前出现的字符。当前出现的字符随着窗口滚动不停更新，每次移动窗口后，都判断当前窗口是否满足条件。 |

**480. 滑动窗口中位数**

中位数是有序序列最中间的那个数。如果序列的大小是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。

例如：

[2,3,4]，中位数是 3

[2,3]，中位数是 (2 + 3) / 2 = 2.5

给你一个数组 nums，有一个大小为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

给出 nums = [1,3,-1,-3,5,3,6,7]，以及 k = 3。

窗口位置 中位数

--------------- -----

[1 3 -1] -3 5 3 6 7 1

1 [3 -1 -3] 5 3 6 7 -1

1 3 [-1 -3 5] 3 6 7 -1

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] 6

因此，返回该滑动窗口的中位数数组 [1,-1,-1,3,5,6]。

提示：

你可以假设 k 始终有效，即：k 始终小于输入的非空数组的元素个数。

与真实值误差在 10 ^ -5 以内的答案将被视作正确答案。

|  |
| --- |
| class Solution {  public:  vector<double> medianSlidingWindow(vector<int>& nums, int k) {  vector<double> res;  deque<int> deq;  int left = 0;  for (size\_t i = 0; i < nums.size(); i++)  {  // 二分查找 有序插入  if(i < k - 1)  {  int index = upper\_bound(deq.begin(), deq.end(), nums[i]) - deq.begin();  deq.insert(deq.begin() + index , nums[i]); // 在index位置插入 nums[i]  continue;  }  // 当窗口再插入一个刚好满的时候，每进来一个数做有序插入 计算中位数  int index = upper\_bound(deq.begin(), deq.end(), nums[i]) - deq.begin();  deq.insert(deq.begin() + index , nums[i]);  res.push\_back( ((double)deq[(k-1)/2] + (double)deq[k/2]) / 2 );  // 从队列中删除左边的第一个数  int leftindex = lower\_bound(deq.begin(), deq.end() ,nums[left]) - deq.begin();  deq.erase(deq.begin() + leftindex);  left++;  }  return res;  }  }; |

**904. 水果成篮**

在一排树中，第 i 棵树产生 tree[i] 型的水果。

你可以从你选择的任何树开始，然后重复执行以下步骤：

1. 把这棵树上的水果放进你的篮子里。如果你做不到，就停下来。
2. 移动到当前树右侧的下一棵树。如果右边没有树，就停下来。

请注意，在选择一颗树后，你没有任何选择：你必须执行步骤 1，然后执行步骤 2，然后返回步骤 1，然后执行步骤 2，依此类推，直至停止。

你有两个篮子，每个篮子可以携带任何数量的水果，但你希望每个篮子只携带一种类型的水果。

用这个程序你能收集的水果树的最大总量是多少？

**输入：[1,2,1]**

输出：3

解释：我们可以收集 [1,2,1]。

输入：[0,1,2,2]

输出：3

解释：我们可以收集 [1,2,2]

如果我们从第一棵树开始，我们将只能收集到 [0, 1]。

输入：[1,2,3,2,2]

输出：4

解释：我们可以收集 [2,3,2,2]

如果我们从第一棵树开始，我们将只能收集到 [1, 2]。

输入：[3,3,3,1,2,1,1,2,3,3,4]

输出：5

解释：我们可以收集 [1,2,1,1,2]

如果我们从第一棵树或第八棵树开始，我们将只能收集到 4 棵水果树。

|  |
| --- |
| class Solution {  public:  int totalFruit(vector<int>& tree) {  int left = 0;  int right = 0;  int max\_res = 0;  unordered\_map<int, int> map;  for (right = 0; right < tree.size(); right++) {  map[tree[right]]++;  while(map.size() > 2) {  max\_res = max(max\_res, right - left);  map[tree[left]]--;  if(map[tree[left]] == 0) map.erase(tree[left]);  left++;  }  }  max\_res = max(max\_res, right - left);  return max\_res;  }  }; |

**978. 最长湍流子数组**

当 A 的子数组 A[i], A[i+1], ..., A[j] 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]。

也就是说，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。

返回 A 的最大湍流子数组的长度。

输入：[9,4,2,10,7,8,8,1,9]

输出：5

解释：(A[1] > A[2] < A[3] > A[4] < A[5])

输入：[4,8,12,16]

输出：2

输入：[100]

输出：1

|  |
| --- |
| class Solution {  public:  int maxTurbulenceSize(vector<int>& A) {  int len = A.size();  int up = 1, down = 1;  int ans = 1;  for (int i = 1; i < len; i++) {  if (A[i - 1] < A[i]) { up = down + 1; down = 1; } // 本次上升, 总长度=下降长度+1 下降长度=1  else if (A[i - 1] > A[i]) { down = up + 1; up = 1; } // 本次下降, 总长度=上升长度+1 上升长度=1  else { up = down = 1; } // 重置到位置i  ans = max(ans, max(up, down));  }  return ans;  }  };  // 和奇偶没关系 只要前后大于等于号相反就可以  **判断连续上升或者下降的波段个数， 似乎就不难理解了。** |

**376. 摆动序列**

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如， [1,7,4,9,2,5] 是一个摆动序列，因为差值 (6,-3,5,-7,3) 是正负交替出现的。相反, [1,4,7,2,5] 和 [1,7,4,5,5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。 通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

输入: [1,7,4,9,2,5]

输出: 6

解释: 整个序列均为摆动序列。

输入: [1,17,5,10,13,15,10,5,16,8]

输出: 7

解释: 这个序列包含几个长度为 7 摆动序列，其中一个可为[1,17,10,13,10,16,8]。

输入: [1,2,3,4,5,6,7,8,9]

输出: 2

class Solution {

|  |
| --- |
| public int wiggleMaxLength(int[] nums) {  int n = nums.length;  if (n < 2) {  return n;  }  int up = 1;  int down = 1;  for (int i = 1; i < n; i++) {  if (nums[i] > nums[i - 1]) {  up = down + 1;  }  if (nums[i] < nums[i - 1]) {  down = up + 1;  }  }  return Math.max(up, down);  }  } |

**992. K 个不同整数的子数组**

给定一个正整数数组 A，如果 A 的某个子数组中不同整数的个数恰好为 K，则称 A 的这个连续、不一定独立的子数组为好子数组。

（例如，[1,2,3,1,2] 中有 3 个不同的整数：1，2，以及 3。）

返回 A 中好子数组的数目。

输入：A = [1,2,1,2,3], K = 2

输出：7

解释：恰好由 2 个不同整数组成的子数组：[1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2].

输入：A = [1,2,1,3,4], K = 3

输出：3

解释：恰好由 3 个不同整数组成的子数组：[1,2,1,3], [2,1,3], [1,3,4].

|  |
| --- |
| class Solution {  public:  int subarraysWithKDistinct(vector<int>& A, int K) {  int left = 0;  int right = 0;  map<int, int> window;  int n = A.size();  int count = 0;  while (right < n) {  int cur = A[right];  window[cur]++;  // 超出K个，左侧往右移动，删除，直到K  while (window.size() > K) {  int toDel = A[left];  window[toDel]--;  if (window[toDel] == 0) {  window.erase(toDel);  }  left++;  }  // 等于K时，固定右侧的端点，左侧往右移动，判断是否能够保持K个  int leftPos = left;  while (window.size() == K) {  **count++;**  int toDel = A[left];  window[toDel]--;  if (window[toDel] == 0) {  window.erase(toDel);  }  left++;  }  // 左侧又往左移动到原来的左边的位置，因为下面右边会后挪1个，不会重复  while (leftPos < left) {  left--;  window[A[left]]++;  }  right++;  }  return count;  }  }; |

**1004. 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。

输入：A = [1,1,1,0,0,0,1,1,1,1,0], K = 2

输出：6

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

|  |
| --- |
| class Solution {  public:  int longestOnes(vector<int>& A, int K) {  int left = 0;  int right = 0;  int zeroCount = 0;  int res = 0;  while (right < A.size()) {  if (A[right] == 0) {  zeroCount++;  }  right++;  while (zeroCount > K) {  if (A[left] == 0) {  zeroCount--;  }  left++;  }  res = max(res, right - left);  }  return res;  }  };  维护最多包含k个0的记录，一旦超过了k个0，把队首的0 pop出来。不断更新当前滑动窗口中0的个数，并取最大值返回即可。 **与424相同** |

**5. 最长回文子串**

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

输入: "babad"

输出: "bab"

注意: "aba" 也是一个有效答案。

输入: "cbbd"

输出: "bb"

|  |
| --- |
| class Solution {  public:  string PalindromeExpend(string& s, int l, int r) {  while (l >= 0 && r < s.size() && s[l] == s[r]) {  l--;r++;  }  return s.substr(l + 1, r - l - 1);// r - 1 - (l + 1) + 1 = r - 1 -l  }  string longestPalindrome(string s) {  string res;  for (int i = 0; i < s.size(); i++) {  string s1 = PalindromeExpend(s, i, i);  string s2 = PalindromeExpend(s, i, i+1);  res = res.size() > s1.size() ? res : s1;  res = res.size() > s2.size() ? res : s2;  }  return res;  }  };  // **中心扩展法**：以i为中心向两侧扩散，判断是否为对称的子串 |

**1109. 航班预订统计**

这里有 n 个航班，它们分别从 1 到 n 进行编号。

我们这儿有一份航班预订表，表中第 i 条预订记录 bookings[i] = [i, j, k] 意味着我们在从 i 到 j 的每个航班上预订了 k 个座位。

请你返回一个长度为 n 的数组 answer，按航班编号顺序返回每个航班上预订的座位数。

输入：bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5

输出：[10,55,45,25,25]

|  |
| --- |
| class Solution {  public:  vector<int> corpFlightBookings(vector<vector<int>>& bookings, int n) {  vector<int> ans(n, 0);  int s, e, k;  for (int i = 0; i < bookings.size(); i++) {  s = bookings[i][0];  e = bookings[i][1];  k = bookings[i][2];  ans[s - 1] += k;  if (e < n) ans[e] -= k;  }  for (int i = 1; i < n; i++) ans[i] += ans[i - 1];  return ans;  }  }; |

可以想象有一个从左至右的等值的无限长的数列，世界是平的，然后由于bookings的加入起了波澜，假设用sum记录的是从左到右的数值变化（求导），例如[1,3,10]，它的变化记录是 +10, 0, 0, -10, 0 ；[2,4,10]，变化记录是0,+10,0,0,-10。 把所有bookings一路求导相加，得出一个总的从左到右变化的记录。 如果要获得每个点的绝对值，那就从前到后加起来（积分）。

|  |
| --- |
| class Solution {  public:  vector<int> corpFlightBookings(vector<vector<int>>& bookings, int n)  {  vector<int> res(n, 0);  for (const auto& book : bookings) {  int l = book[0] - 1;  int r = book[1];  int seats = book[2];  res[l] += book[2];  if (r < n) {  res[r] += -book[2];  }  }  for (int i = 1; i < n; ++i) {  res[i] = res[i - 1] + res[i];  }  return res;  }  }; |