宫水三叶的刷题日记

滑动窗口

Author ：宫水三叶
Date ：2021/10/07
QQ Group: 703311589
WeChat ：oaoaya

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「滑动窗口」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「滑动窗口」即可获取最新下载链接。

💡**下面介绍使用本合集的最佳使用实践：**

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「滑动窗口」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **3. 无重复字符的最长子串** ，难度为 **中等**。

Tag : 「哈希表」、「双指针」、「滑动窗口」

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:

宫水三叶的刷题日记
公众号: 宫水三叶的刷题日记

```
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。
```

示例 3:

```
输入: s = "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。
```

示例 4:

```
输入: s = ""
输出: 0
```

提示：

- 0 <= s.length <= 5 * 10^4
- s 由英文字母、数字、符号和空格组成

双指针 + 哈希表

定义两个指针 `start` 和 `end`，表示当前处理到的子串是 `[start,end]`。

`[start,end]` 始终满足要求：无重复字符。

从前往后进行扫描，同时维护一个哈希表记录 `[start,end]` 中每个字符出现的次数。

遍历过程中，`end` 不断自增，将第 `end` 个字符在哈希表中出现的次数加一。

令 `right` 为 下标 `end` 对应的字符，当满足 `map.get(right) > 1` 时，代表此前出现过第 `end` 位对应的字符。

此时更新 `start` 的位置（使其右移），直到不满足 `map.get(right) > 1`（代表 `[start,end]` 恢复满足无重复字符的条件）。同时使用 `[start,end]` 长度更新答案。

代码：

宫水三叶 刷题日记 公众号: 宫水三叶的刷题日记

```java
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int ans = 0;
        for (int start = 0, end = 0; end < s.length(); end++) {
            char right = s.charAt(end);
            map.put(right, map.getOrDefault(right, 0) + 1);
            while (map.get(right) > 1) {
                char left = s.charAt(start);
                map.put(left, map.get(left) - 1);
                start++;
            }
            ans = Math.max(ans, end - start + 1);
        }
        return ans;
    }
}
```

- 时间复杂度：虽然有两层循环，但每个字符在哈希表中最多只会被插入和删除一次，复杂度为 $O(n)$
- 空间复杂度：使用了哈希表进行字符记录，复杂度为 $O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **30. 串联所有单词的子串** ，难度为 **困难**。

Tag：「哈希表」、「滑动窗口」

给定一个字符串 s 和一些长度相同的单词 words。

找出 s 中恰好可以由 words 中所有单词串联形成的子串的起始位置。

注意子串要与 words 中的单词完全匹配，中间不能有其他字符，但不需要考虑 words 中单词串联的顺序。

示例 1：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```
输入：
  s = "barfoothefoobarman",
  words = ["foo","bar"]

输出：[0,9]

解释：
从索引 0 和 9 开始的子串分别是 "barfoo" 和 "foobar" 。
输出的顺序不重要, [9,0] 也是有效答案。
```

示例 2：

```
输入：
  s = "wordgoodgoodgoodbestword",
  words = ["word","good","best","word"]

输出：[]
```

朴素哈希表

令 `n` 为字符串 `s` 的长度，`m` 为数组 `words` 的长度（单词的个数），`w` 为单个单词的长度。

由于 `words` 里面每个单词长度固定，而我们要找的字符串只能恰好包含所有的单词，所有我们要找的目标子串的长度为 $m * w$。

那么一个直观的思路是：

1. 使用哈希表 `map` 记录 `words` 中每个单词的出现次数
2. 枚举 `s` 中的每个字符作为起点，往后取得长度为 $m * w$ 的子串 `sub`
3. 使用哈希表 `cur` 统计 `sub` 每个单词的出现次数（每隔 `w` 长度作为一个单词）
4. 比较 `cur` 和 `map` 是否相同

注意：在步骤 3 中，如果发现 `sub` 中包含了 `words` 没有出现的单词，可以直接剪枝。

剪枝处使用了带标签的 `continue` 语句直接回到外层循环进行。

代码：

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        List<Integer> ans = new ArrayList<>();
        if (words.length == 0) return ans;

        int n = s.length(), m = words.length, w = words[0].length();

        Map<String, Integer> map = new HashMap<>();
        for (String word : words) {
            map.put(word, map.getOrDefault(word, 0) + 1);
        }

        out:for (int i = 0; i + m * w <= n; i++) {
            Map<String, Integer> cur = new HashMap<>();
            String sub = s.substring(i, i + m * w);
            for (int j = 0; j < sub.length(); j += w) {
                String item = sub.substring(j, j + w);
                if (!map.containsKey(item)) continue out;
                cur.put(item, cur.getOrDefault(item, 0) + 1);
            }
            if (cmp(cur, map)) ans.add(i);
        }

        return ans;
    }
    boolean cmp(Map<String, Integer> m1, Map<String, Integer> m2) {
        if (m1.size() != m2.size()) return false;
        for (String k1 : m1.keySet()) {
            if (!m2.containsKey(k1) || !m1.get(k1).equals(m2.get(k1))) return false;
        }
        for (String k2 : m2.keySet()) {
            if (!m1.containsKey(k2) || !m1.get(k2).equals(m2.get(k2))) return false;
        }
        return true;
    }
}
```

- 时间复杂度：将 `words` 中的单词存入哈希表，复杂度为 $O(m)$；然后第一层循环枚举 `s` 中的每个字符作为起点，复杂度为 $O(n)$；在循环中将 `sub` 划分为 `m` 个单词进行统计，枚举了 `m - 1` 个下标，复杂度为 $O(m)$；每个字符串的长度为 `w`。整体复杂度为 $O(n*m*w)$
- 空间复杂度：$O(m*w)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

滑动窗口 & 哈希表

事实上，我们可以**优化这个枚举起点的过程**。

我们可以将起点根据 **当前下标与单词长度的取余结果** 进行分类，这样我们就不用频繁的建立新的哈希表和进行单词统计。

代码：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```java
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        List<Integer> ans = new ArrayList<>();
        if (words.length == 0) return ans;

        int n = s.length(), m = words.length, w = words[0].length();

        // 统计 words 中「每个目标单词」的出现次数
        Map<String, Integer> map = new HashMap<>();
        for (String word : words) {
            map.put(word, map.getOrDefault(word, 0) + 1);
        }

        for (int i = 0; i < w; i++) {
            // 构建一个当前子串对应 map，统计当前子串中「每个目标单词」的出现次数
            Map<String, Integer> curMap = new HashMap<>();
            // 滑动窗口的大小固定是 m * w
            // 每次将下一个单词添加进 cur，上一个单词移出 cur
            for (int j = i; j + w <= n; j += w) {
                String cur = s.substring(j, j + w);
                if (j >= i + (m * w)) {
                    int idx = j - m * w;
                    String prev = s.substring(idx, idx + w);
                    if (curMap.get(prev) == 1) {
                        curMap.remove(prev);
                    } else {
                        curMap.put(prev, curMap.get(prev) - 1);
                    }
                }
                curMap.put(cur, curMap.getOrDefault(cur, 0) + 1);
                // 如果当前子串对应 map 和 words 中对应的 map 相同，说明当前子串包含了「所有的目标单词
                if (map.containsKey(cur) && curMap.get(cur).equals(map.get(cur)) && cmp(ma
                    ans.add(j - (m - 1) * w);
                }
            }
        }

        return ans;
    }
    // 比较两个 map 是否相同
    boolean cmp(Map<String, Integer> m1, Map<String, Integer> m2) {
        if (m1.size() != m2.size()) return false;
        for (String k1 : m1.keySet()) {
            if (!m2.containsKey(k1) || !m1.get(k1).equals(m2.get(k1))) return false;
        }
        for (String k2 : m2.keySet()) {
```

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

```
                if (!m1.containsKey(k2) || !m1.get(k2).equals(m2.get(k2))) return false;
            }
            return true;
        }
    }
```

- 时间复杂度：将 `words` 中的单词存入哈希表，复杂度为 $O(m)$；然后枚举了取余的结果，复杂度为 $O(w)$；每次循环最多处理 `n` 长度的字符串，复杂度为 $O(n)$。整体复杂度为 $O(m + w * n)$
- 空间复杂度：$O(m * w)$

**🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **220. 存在重复元素 III** ，难度为 **中等**。

Tag：「滑动窗口」、「二分」、「桶排序」

给你一个整数数组 nums 和两个整数 k 和 t 。

请你判断是否存在 两个不同下标 i 和 j，使得 abs(nums[i] - nums[j]) <= t ，同时又满足 abs(i - j) <= k 。

如果存在则返回 true，不存在返回 false。

示例 1：

```
输入：nums = [1,2,3,1], k = 3, t = 0

输出：true
```

示例 2：

```
输入：nums = [1,0,1,1], k = 1, t = 2

输出：true
```

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

示例 3：

```
输入：nums = [1,5,9,1,5,9], k = 2, t = 3

输出：false
```

提示：

- 0 <= nums.length <= 2 * 10^4
- -2^{31} <= nums[i] <= 2^{31} - 1
- 0 <= k <= 10^4
- 0 <= t <= 2^{31} - 1

滑动窗口 & 二分

根据题意，对于任意一个位置 `i`（假设其值为 `u`），我们其实是希望在下标范围为 $[max(0, i-k), i)$ 内找到值范围在 $[u-t, u+t]$ 的数。

一个朴素的想法是每次遍历到任意位置 `i` 的时候，往后检查 `k` 个元素，但这样做的复杂度是 $O(nk)$ 的，会超时。

显然我们需要优化「检查后面 `k` 个元素」这一过程。

我们希望使用一个「有序集合」去维护长度为 `k` 的滑动窗口内的数，该数据结构最好支持高效「查询」与「插入/删除」操作：

- **查询：能够在「有序集合」中应用「二分查找」，快速找到「小于等于 u 的最大值」和「大于等于 `u` 的最小值」（即「有序集合」中的最接近 `u` 的数）。**
- **插入/删除：在往「有序集合」添加或删除元素时，能够在低于线性的复杂度内完成（维持有序特性）。**

或许你会想到近似 $O(1)$ 操作的 `HashMap`，但注意这里我们需要找的是符合 $abs(nums[i], nums[j]) \leqslant t$ 的两个值，`nums[i]` 与 `nums[j]` 并不一定相等，而 `HashMap` 无法很好的支持「范围查询」操作。

我们还会想到「树」结构。

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

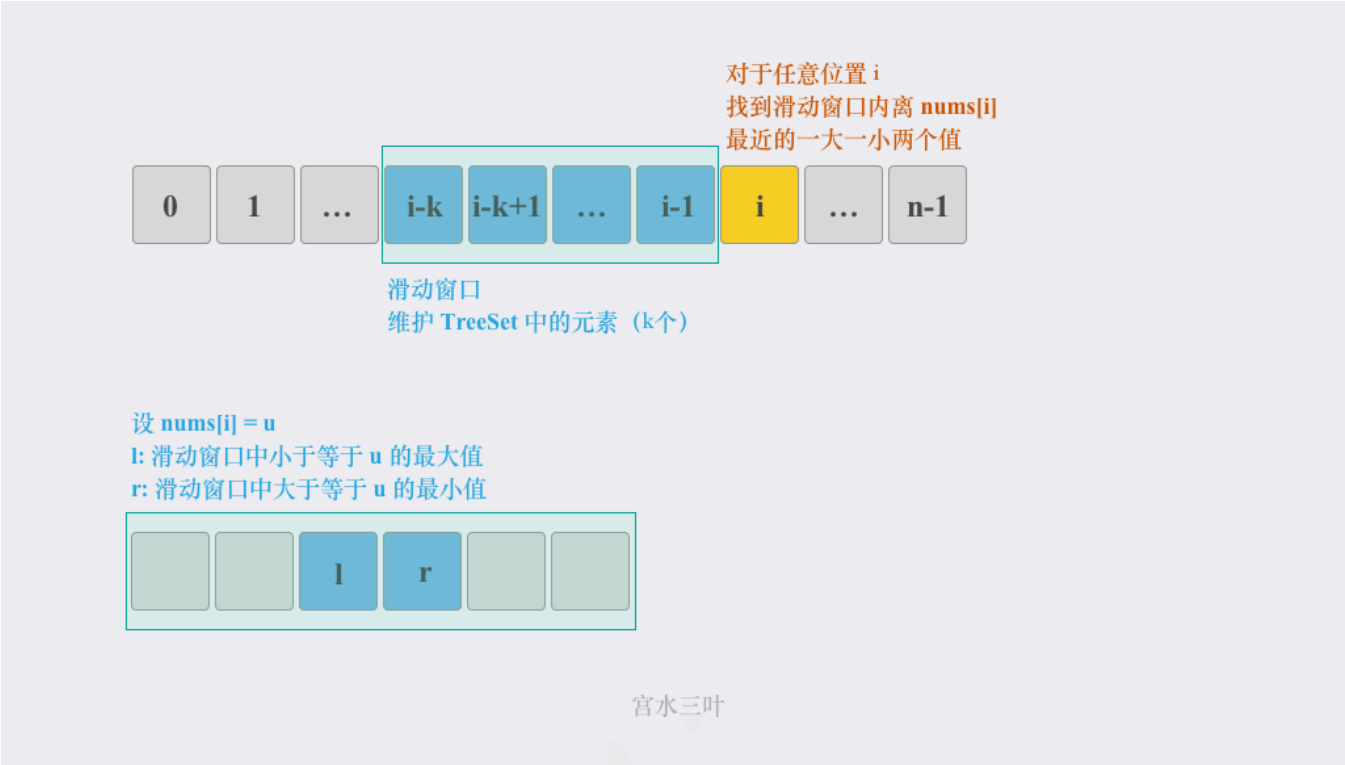
例如 AVL，能够让我们在最坏为 $O(\log k)$ 的复杂度内取得到最接近 `u` 的值是多少，但本题除了「查询」以外，还涉及频繁的「插入/删除」操作（随着我们遍历 `nums` 的元素，滑动窗口不断右移，我们需要不断的往「有序集合」中删除和添加元素）。

简单采用 AVL 树，会导致每次的插入删除操作都触发 AVL 的平衡调整，一次平衡调整会伴随着若干次的旋转。

而红黑树则很好解决了上述问题：将平衡调整引发的旋转的次数从「若干次」限制到「最多三次」。

因此，当「查询」动作和「插入/删除」动作频率相当时，更好的选择是使用「红黑树」。

也就是对应到 Java 中的 `TreeSet` 数据结构（基于红黑树，查找和插入都具有折半的效率）。



其他细节：由于 `nums` 中的数较大，会存在 `int` 溢出问题，我们需要使用 `long` 来存储。

代码：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        int n = nums.length;
        TreeSet<Long> ts = new TreeSet<>();
        for (int i = 0; i < n; i++) {
            Long u = nums[i] * 1L;
            // 从 ts 中找到小于等于 u 的最大值（小于等于 u 的最接近 u 的数）
            Long l = ts.floor(u);
            // 从 ts 中找到大于等于 u 的最小值（大于等于 u 的最接近 u 的数）
            Long r = ts.ceiling(u);
            if(l != null && u - l <= t) return true;
            if(r != null && r - u <= t) return true;
            // 将当前数加到 ts 中，并移除下标范围不在 [max(0, i - k), i) 的数（维持滑动窗口大小为 k
            ts.add(u);
            if (i >= k) ts.remove(nums[i - k] * 1L);
        }
        return false;
    }
}
```

- 时间复杂度：`TreeSet` 基于红黑树，查找和插入都是 $O(\log k)$ 复杂度。整体复杂度为 $O(n \log k)$
- 空间复杂度：$O(k)$

桶排序

上述解法无法做到线性的原因是：我们需要在大小为 `k` 的滑动窗口所在的「有序集合」中找到与 `u` 接近的数。

如果我们能够将 `k` 个数字分到 k 个桶的话，那么我们就能 $O(1)$ 的复杂度确定是否有 $[u-t, u+t]$ 的数字（检查目标桶是否有元素）。

具体的做法为：令桶的大小为 $size = t + 1$，根据 `u` 计算所在桶编号：

- 如果已经存在该桶，说明前面已有 $[u-t, u+t]$ 范围的数字，返回 `true`
- 如果不存在该桶，则检查相邻两个桶的元素是有 $[u-t, u+t]$ 范围的数字，如有返回 `true`
- 建立目标桶，并删除下标范围不在 $[max(0, i-k), i)$ 内的桶

代码：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    long size;
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        int n = nums.length;
        Map<Long, Long> map = new HashMap<>();
        size = t + 1L;
        for (int i = 0; i < n; i++) {
            long u = nums[i] * 1L;
            long idx = getIdx(u);
            // 目标桶已存在（桶不为空），说明前面已有 [u - t, u + t] 范围的数字
            if (map.containsKey(idx)) return true;
            // 检查相邻的桶
            long l = idx - 1, r = idx + 1;
            if (map.containsKey(l) && u - map.get(l) <= t) return true;
            if (map.containsKey(r) && map.get(r) - u <= t) return true;
            // 建立目标桶
            map.put(idx, u);
            // 移除下标范围不在 [max(0, i - k), i) 内的桶
            if (i >= k) map.remove(getIdx(nums[i - k] * 1L));
        }
        return false;
    }
    long getIdx(long u) {
        return u >= 0 ? u / size : ((u + 1) / size) - 1;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(k)$

【重点】如何理解 `getIdx()` 的逻辑

1. 为什么 `size` 需要对 `t` 进行 `+1` 操作？

目的是为了确保差值小于等于 `t` 的数能够落到一个桶中。

举个 🌰，假设 `[0,1,2,3]`，`t = 3`，显然四个数都应该落在同一个桶。

如果不对 `t` 进行 `+1` 操作的话，那么 `[0,1,2]` 和 `[3]` 会被落到不同的桶中，那么为了解决这种错误，我们需要对 `t` 进行 `+1` 作为 `size`。

宫水三叶 刷题日记 公众号：宫水三叶的刷题日记

这样我们的数轴就能被分割成：

`0 1 2 3 | 4 5 6 7 | 8 9 10 11 | 12 13 14 15 | …`

总结一下，令 `size = t + 1` 的本质是因为差值为 `t` 两个数在数轴上相隔距离为 `t + 1`，它们需要被落到同一个桶中。

当明确了 `size` 的大小之后，对于正数部分我们则有 `idx = nums[i] / size`。

2. 如何理解负数部分的逻辑？

由于我们处理正数的时候，处理了数值 `0`，因此我们负数部分是从 `-1` 开始的。

还是我们上述 🌰，此时我们有 `t = 3` 和 `size = t + 1 = 4`。

考虑 `[-4,-3,-2,-1]` 的情况，它们应该落在一个桶中。

如果直接复用 `idx = nums[i] / size` 的话，`[-4]` 和 `[-3,-2,-1]` 会被分到不同的桶中。

根本原因是我们处理整数的时候，已经分掉了数值 `0`。

这时候我们需要先对 `nums[i]` 进行 `+1` 操作（即将负数部分在数轴上进行整体右移），即得到 `(nums[i] + 1) / size`。

这样一来负数部分与正数部分一样，可以被正常分割了。

但由于 0 号桶已经被使用了，我们还需要在此基础上进行 `-1`，相当于将负数部分的桶下标（`idx`）往左移，即得到 `((nums[i] + 1) / size) - 1`。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **424. 替换后的最长重复字符** ，难度为 **中等**。

Tag：「双指针」、「滑动窗口」

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。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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 10^4。

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

令 `l` 为符合条件的子串的左端点，`r` 为符合条件的子串的右端点。

使用 `cnt` 统计 `[l,r]` 范围的子串中每个字符串出现的次数。

对于合法的子串而言，必然有 `sum(所有字符的出现次数) - max(出现次数最多的字符的出现次数) = other(其他字符的出现次数) <= k`。

当找到这样的性质之后，我们可以对 `s` 进行遍历，每次让 `r` 右移并计数，如果符合条件，更新最大值；如果不符合条件，让 `l` 右移，更新计数，直到符合条件。

代码：

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

```java
class Solution {
    public int characterReplacement(String s, int k) {
        char[] cs = s.toCharArray();
        int[] cnt = new int[26];
        int ans = 0;
        for (int l = 0, r = 0; r < s.length(); r++) {
            // cnt[cs[r] - 'A']++;
            int cur = cs[r] - 'A';
            cnt[cur]++;
            // while (!check(cnt, k)) cnt[cs[l++] - 'A']--;
            while (!check(cnt, k)) {
                int del = cs[l] - 'A';
                cnt[del]--;
                l++;
            }
            ans = Math.max(ans, r - l + 1);
        }
        return ans;
    }
    boolean check(int[] cnt, int k) {
        int max = 0, sum = 0;
        for (int i = 0; i < 26; i++) {
            max = Math.max(max, cnt[i]);
            sum += cnt[i];
        }
        return sum - max <= k;
    }
}
```

- 时间复杂度：使用 `l` 和 `r` 指针对 `s` 进行单次扫描，复杂度为 $O(n)$；`check` 方法是对长度固定的数组进行扫描，复杂度为 $O(1)$。整体复杂度为 $O(n)$。
- 空间复杂度：使用了固定长度的数组进行统计。复杂度为 $O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **480. 滑动窗口中位数** ，难度为 **困难**。

Tag：「滑动窗口」、「堆」、「优先队列」

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

中位数是有序序列最中间的那个数。如果序列的长度是偶数，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，有一个长度为 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。

示例：

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 始终小于等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。

朴素解法

一个直观的做法是：对每个滑动窗口的数进行排序，获取排序好的数组中的第 `k / 2` 和 `(k - 1) / 2` 个数（避免奇偶数讨论），计算中位数。

我们大概分析就知道这个做法至少 $O(n * k)$ 的，算上排序的话应该是 $O(n * (k + k \log k))$

宫水三叶 刷题日记 公众号：宫水三叶的刷题日记

。

比较无奈的是，这道题没有给出数据范围。我们无法根据判断这样的做法会不会超时。

PS. 实际上这道题朴素解法是可以过的，有蓝桥杯内味了 ~

朴素做法通常是优化的开始，所以我还是提供一下朴素做法的代码

代码：

```
class Solution {
    public double[] medianSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        int cnt = n - k + 1;
        double[] ans = new double[cnt];
        int[] t = new int[k];
        for (int l = 0, r = l + k - 1; r < n; l++, r++) {
            for (int i = l; i <= r; i++) t[i - l] = nums[i];
            Arrays.sort(t);
            ans[l] = (t[k / 2] / 2.0) + (t[(k - 1) / 2] / 2.0);
        }
        return ans;
    }
}
```

- 时间复杂度：最多有 `n` 个窗口需要滑动计算。每个窗口，需要先插入数据，复杂度为 $O(k)$，插入后需要排序，复杂度为 $O(k \log k)$。整体复杂度为 $O(n * (k + k \log k))$
- 空间复杂度：使用了长度为 `k` 的临时数组。复杂度为 $O(k)$

优先队列（堆）解法

从朴素解法中我们可以发现，其实我们需要的就是滑动窗口中的第 `k / 2` 小的值和第 `(k - 1) / 2` 小的值。

我们知道滑动窗口求最值的问题，可以使用优先队列来做。

但这里我们求的是第 `k` 小的数，而且是需要两个值。还能不能使用优先队列来做呢？

我们可以维护两个堆：

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

- 一个大根堆维护着滑动窗口中一半较小的值（此时堆顶元素为滑动窗口中的第 `(k - 1) / 2` 小的值）
- 一个小根堆维护着滑动窗口中一半较大的值（此时堆顶元素为滑动窗口中的第 `k / 2` 小的值）

滑动窗口的中位数就是两个堆的堆顶元素的平均值。

实现细节：

1. 初始化时，先让 `k` 个元素直接入 `right`，再从 `right` 中倒出 `k / 2` 个到 `left` 中。这时候可以根据 `left` 和 `right` 得到第一个滑动窗口的中位值。
2. 开始滑动窗口，每次滑动都有一个待添加和待移除的数：
   2.1 根据与右堆的堆顶元素比较，决定是插入哪个堆和从哪个堆移除
   2.2 之后调整两堆的大小（确保只会出现 `left.size() == right.size()` 或 `right.size() - left.size() == 1`，对应了窗口长度为偶数或者奇数的情况）
   2.3 根据 `left` 堆 和 `right` 堆得到当前滑动窗口的中位值

代码：

宫水三叶 の 刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    public double[] medianSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        int cnt = n - k + 1;
        double[] ans = new double[cnt];
        // 如果是奇数滑动窗口，让 right 的数量比 left 多一个
        PriorityQueue<Integer> left  = new PriorityQueue<>((a,b)->Integer.compare(b,a)); //
        PriorityQueue<Integer> right = new PriorityQueue<>((a,b)->Integer.compare(a,b)); //
        for (int i = 0; i < k; i++) right.add(nums[i]);
        for (int i = 0; i < k / 2; i++) left.add(right.poll());
        ans[0] = getMid(left, right);
        for (int i = k; i < n; i++) {
            // 人为确保了 right 会比 left 多，因此，删除和添加都与 right 比较（left 可能为空）
            int add = nums[i], del = nums[i - k];
            if (add >= right.peek()) {
                right.add(add);
            } else {
                left.add(add);
            }
            if (del >= right.peek()) {
                right.remove(del);
            } else {
                left.remove(del);
            }
            adjust(left, right);
            ans[i - k + 1] = getMid(left, right);
        }
        return ans;
    }
    void adjust(PriorityQueue<Integer> left, PriorityQueue<Integer> right) {
        while (left.size() > right.size()) right.add(left.poll());
        while (right.size() - left.size() > 1) left.add(right.poll());
    }
    double getMid(PriorityQueue<Integer> left, PriorityQueue<Integer> right) {
        if (left.size() == right.size()) {
            return (left.peek() / 2.0) + (right.peek() / 2.0);
        } else {
            return right.peek() * 1.0;
        }
    }
}
```

- 时间复杂度：调整过程中堆大小最大为 `k`，堆操作中的指定元素删除复杂度为 $O(k)$；窗口数量最多为 `n`。整体复杂度为 $O(n*k)$
- 空间复杂度：最多有 `n` 个元素在堆内。复杂度为 $O(n)$

宫水三叶 刷题日记 公众号：宫水三叶的刷题日记

注意点 (2021-02-04 更新)

今天的题解发到 LeetCode 后，针对一些同学的评论。

我觉得有一定的代表性，所以拿出来讲讲 ~

- (问)某同学：为什么 `new PriorityQueue<>((x,y)->(y-x))` 的写法会有某些案例无法通过？和 `new PriorityQueue<>((x,y)->Integer.compare(y,x))` 写法有何区别？
- (答)三叶：`(x,y)->(y-x)` 的写法逻辑没有错，AC 不了是因为 int 溢出。
  在 Java 中 Integer.compare 的实现是 `(x < y) ? -1 : ((x == y) ? 0 : 1)`。只是单纯的比较，不涉及运算，所以不存在溢出风险。
  而直接使用 `y - x`，当 `y = Integer.MAX_VALUE`, `x = Integer.MIN_VALUE` 时，到导致溢出，返回的是 **负数** ，而不是逻辑期望的 **正数**

同样具有溢出问题的还有计算第 `k / 2` 小的数和第 `(k - 1) / 2` 小的数的平均值时。

我是使用 `(a / 2.0) + (b / 2.0)` 的形式，而不是采用 `(a + b) / 2.0` 的形式。后者有相加溢出的风险。

注意点 (2021-02-05 更新)

JDK 中 `PriorityQueue` 的 `remove(Object o)` 实现是先调用 `indexOf(Object o)` 方法进行线性扫描找到下标（复杂度为 $O(n)$），之后再调用 `removeAt(int i)` 进行删除（复杂度为 $O(\log n)$）。

对于本题而言，如果需要实现 $O(\log n)$ 的 `remove(Object o)`， 只能通过引入其他数据结构（如哈希表）来实现快速查找元素在对堆数组中的下标。

对于本题，可以使用元素在原数组中的下标作为 key，在堆数组中的真实下标作为 val。

通过哈希表可以 $O(1)$ 的复杂度找到下标，之后的删除只需要算堆调整的复杂度即可（最多 down 一遍，up 一遍，复杂度为 $O(\log n)$）。

至于 JDK 没有这样做的原因，猜测是因为基本类型的包装类型存在小数缓存机制，导致无法很好的使用哈希表来对应一个插入元素的下标。

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

举个🌰，我们调用三次 `add(10)`， 会有 3 个 10 在堆内，但是由于小数（默认范围为【-128,127】）包装类型存在缓存机制，使用哈希表继续记录的话，只会有 { Integer.valueOf(10) : 移动过程中最后一次访问的数组下标 } 这样一条记录（add 进去的 10 均为同一对象）。这时候删除一个 10 之后，哈希表无法正确指导我们找到下一个 10 的位置。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **567. 字符串的排列** ，难度为 **中等**。

Tag :「滑动窗口」

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
```

提示：

- 输入的字符串只包含小写字母
- 两个字符串的长度都在 [1, 10,000] 之间

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

滑动窗口

由于是 `s2` 中判断是否包含 `s1` 的排列，而且 `s1` 和 `s2` 均为小数。

可以使用数组先对 `s1` 进行统计，之后使用滑动窗口进行扫描，每滑动一次检查窗口内的字符频率和 `s1` 是否相等 ~

以下代码，可以作为滑动窗口模板使用：

PS. 你会发现以下代码和 643. 子数组最大平均数 I 和 1423. 可获得的最大点数 代码很相似，因为是一套模板。

1. 初始化将滑动窗口压满，取得第一个滑动窗口的目标值
2. 继续滑动窗口，每往前滑动一次，需要删除一个和添加一个元素

代码：

```
class Solution {
    public boolean checkInclusion(String s1, String s2) {
        int m = s1.length(), n = s2.length();
        if (m > n) return false;
        int[] cnt = new int[26];
        for (char c : s1.toCharArray()) cnt[c - 'a']++;
        int[] cur = new int[26];
        for (int i = 0; i < m; i++) cur[s2.charAt(i) - 'a']++;
        if (check(cnt, cur)) return true;
        for (int i = m; i < n; i++) {
            cur[s2.charAt(i) - 'a']++;
            cur[s2.charAt(i - m) - 'a']--;
            if (check(cnt, cur)) return true;
        }
        return false;
    }
    boolean check(int[] cnt1, int[] cnt2) {
        for (int i = 0; i < 26; i++) {
            if (cnt1[i] != cnt2[i]) return false;
        }
        return true;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

经过了前面几天的「滑动窗口」，相信你做这题已经很有感觉了。

但其实这是某道困难题的简化版，本题根据「字符」滑动，而 30. 串联所有单词的子串 则是根据「单词」来。但基本思路都是一样的，强烈建议你来试试 ~

相关链接

30. 串联所有单词的子串：【刷穿LC】朴素哈希表解法 + 滑动窗口解法

643. 子数组最大平均数 I：滑动窗口裸题 （含模板）~

1423. 可获得的最大点数：详解滑动窗口基本思路（含模板）~

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **643. 子数组最大平均数 I** ，难度为 **简单**。

Tag : 「滑动窗口」

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。

示例：

```
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
```

提示：

- 1 <= k <= n <= 30,000。
- 所给数据范围 [-10,000，10,000]。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

滑动窗口

这是一道道滑动窗口裸题。

以下代码，可以作为滑动窗口模板使用：

1. 初始化将滑动窗口压满，取得第一个滑动窗口的目标值
2. 继续滑动窗口，每往前滑动一次，需要删除一个和添加一个元素

代码：

```
class Solution {
    public double findMaxAverage(int[] nums, int k) {
        double ans = 0, sum = 0;
        for (int i = 0; i < k; i++) sum += nums[i];
        ans = sum / k;
        for (int i = k; i < nums.length; i++) {
            sum = sum + nums[i] - nums[i - k]; // int add = nums[i], del = nums[i - k];
            ans = Math.max(ans, sum / k);
        }
        return ans;
    }
}
```

- 时间复杂度：每个元素最多滑入和滑出窗口一次。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **992. K 个不同整数的子数组** ，难度为 **困难**。

Tag：「双指针」、「滑动窗口」

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、不一定不同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。）

宫水三叶的刷题日记
公众号: 宫水三叶的刷题日记

返回 A 中好子数组的数目。

示例 1：

```
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]
```

示例 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]
```

提示：

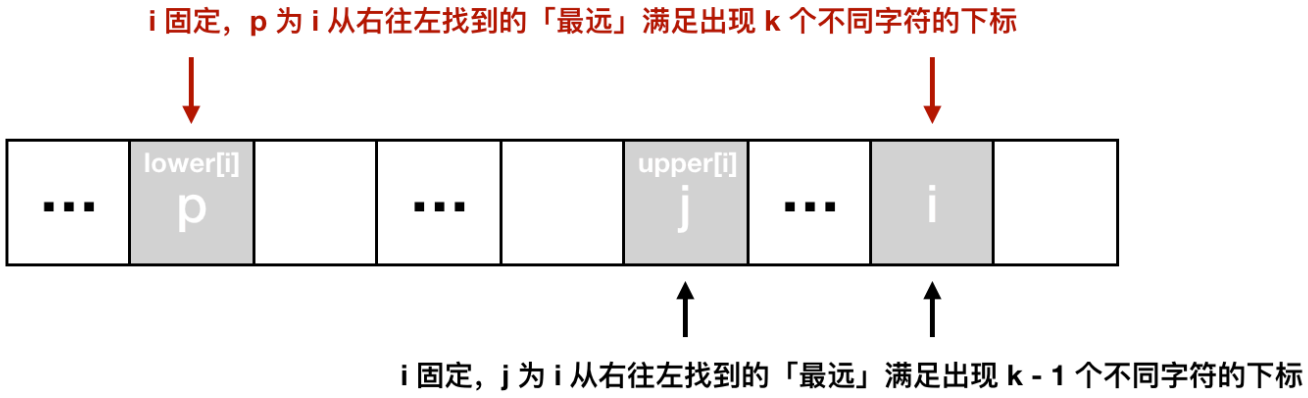
- 1 <= A.length <= 20000
- 1 <= A[i] <= A.length
- 1 <= K <= A.length

双指针

对原数组 `nums` 的每一个位置 `i` 而言：

1. 找到其左边「最远」满足出现 `k` 个不同字符的下标，记为 `p`。这时候形成的区间为 `[p, i]`
2. 找到其左边「最远」满足出现 `k - 1` 个不同字符的下标，记为 `j`。这时候形成的区间为 `[j, i]`
3. **那么对于 `j - p` 其实就是代表以 nums[i] 为右边界（必须包含 num[i]），不同字符数量「恰好」为 k 的子数组数量**

宫水三叶 の 刷题日记
公众号：宫水三叶的刷题日记



我们使用 `lower` 数组存起每个位置的 `k`；使用 `upper` 数组存起每个位置的 `j`。

累积每个位置的 `upper[i] - lower[i]` 就是答案。

计算 `lower` 数组 和 `upper` 数组的过程可以使用双指针：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    public int subarraysWithKDistinct(int[] nums, int k) {
        int n = nums.length;
        int[] lower = new int[n], upper = new int[n];
        find(lower, nums, k);
        find(upper, nums, k - 1);
        // System.out.println(Arrays.toString(lower));
        // System.out.println(Arrays.toString(upper));
        int ans = 0;
        for (int i = 0; i < n; i++) ans += upper[i] - lower[i];
        return ans;
    }
    void find(int[] arr, int[] nums, int k) {
        int n = nums.length;
        int[] cnt = new int[n + 1];
        for (int i = 0, j = 0, sum = 0; i < n; i++) {
            int right = nums[i];
            if (cnt[right] == 0) sum++;
            cnt[right]++;
            while (sum > k) {
                int left = nums[j++];
                cnt[left]--;
                if (cnt[left] == 0) sum--;
            }
            arr[i] = j;
        }
    }
}
```

- 时间复杂度：对数组进行常数次扫描。复杂度为 $O(n)$。
- 空间复杂度：$O(n)$

其他

这里的 `lower` 和 `upper` 其实可以优化掉，但也只是常数级别的优化，空间复杂度仍为 $O(n)$。

推荐大家打印一下 `lower` 和 `upper` 来看看，加深对 「 **`upper[i] - lower[i]` 代表了考虑 `nums[i]` 为右边界，不同字符数量「恰好」为 k 的子数组数量** 」 这句话的理解。

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1004. 最大连续1的个数 III** ，难度为 **中等**。

Tag：「双指针」、「滑动窗口」、「二分」、「前缀和」

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。
```

提示：

- 1 <= A.length <= 20000
- 0 <= K <= A.length
- A[i] 为 0 或 1

动态规划（TLE）

看到本题，其实首先想到的是 DP，但是 DP 是 $O(nk)$ 算法。

看到了数据范围是 10^4，那么时空复杂度应该都是 10^8。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

空间可以通过「滚动数组」优化到 10^4，但时间无法优化，会超时。

PS. 什么时候我们会用 DP 来解本题？通过如果 K 的数量级不超过 1000 的话，DP 应该是最常规的做法。

定义 $f[i,j]$ 代表考虑前 i 个数（并以 $A[i]$ 为结尾的），最大翻转次数为 j 时，连续 1 的最大长度。

- 如果 $A[i]$ 本身就为 1 的话，无须消耗翻转次数，$f[i][j] = f[i-1][j] + 1$。
- 如果 $A[i]$ 本身不为 1 的话，由于定义是必须以 $A[i]$ 为结尾，因此必须要选择翻转该位置，$f[i][j] = f[i-1][j-1] + 1$。

代码：

```
class Solution {
    public int longestOnes(int[] nums, int k) {
        int n = nums.length;
        //
        int[][] f = new int[2][k + 1];
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= k; j++) {
                if (nums[i - 1] == 1) {
                    f[i & 1][j] = f[(i - 1) & 1][j] + 1;
                } else {
                    f[i & 1][j] = j == 0 ? 0 : f[(i - 1) & 1][j - 1] + 1;
                }
                ans = Math.max(ans, f[i & 1][j]);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(nk)$
- 空间复杂度：$O(k)$

前缀和 + 二分

从数据范围上分析，**平方级别**的算法过不了，往下优化就应该是**对数级别**的算法。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

因此，很容易我们就会想到「二分」。

当然还需要我们对问题做一下等价变形。

最大替换次数不超过 `k` 次，可以将问题转换为找出连续一段区间 `[l,r]`，使得区间中出现 0 的次数不超过 `k` 次。

我们可以枚举区间 左端点/右端点 ，然后找到其满足「出现 0 的次数不超过 `k` 次」的最远右端点/最远左端点。

为了快速判断 `[l,r]` 之间出现 0 的个数，我们需要用到前缀和。

假设 `[l,r]` 的区间长度为 `len`，区间和为 `tot`，那么出现 0 的格式为 `len - tol`，再与 `k` 进行比较。

由于数组中不会出现负权值，因此前缀和数组具有「单调性」，那么必然满足「其中一段满足 $len - tol <= k$，另外一段不满足 $len - tol <= k$」。

因此，对于某个确定的「左端点/右端点」而言，以「其最远右端点/最远左端点」为分割点的前缀和数轴，具有「二段性」。可以通过二分来找分割点。

代码：

宫水三叶 の 刷题日记
公众号: 宫水三叶的刷题日记

```Java
class Solution {
    public int longestOnes(int[] nums, int k) {
        int n = nums.length;
        int ans = 0;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + nums[i - 1];
        for (int i = 0; i < n; i++) {
            int l = 0, r = i;
            while (l < r) {
                int mid = l + r >> 1;
                if (check(sum, mid, i, k)) {
                    r = mid;
                } else {
                    l = mid + 1;
                }
            }
            if (check(sum, r, i, k)) ans = Math.max(ans, i - r + 1);
        }
        return ans;
    }
    boolean check(int[] sum, int l, int r, int k) {
        int tol = sum[r + 1] - sum[l], len = r - l + 1;
        return len - tol <= k;
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(n)$

关于二分结束后再次 `check` 的说明：由于「二分」本质是找满足某个性质的分割点，通常我们的某个性质会是「非等值条件」，不一定会取得 `=`。

例如我们很熟悉的：从某个非递减数组中找目标值，找到返回下标，否则返回 -1。

当目标值不存在，「二分」找到的应该是数组内比目标值小或比目标值大的最接近的数。因此二分结束后先进行 `check` 再使用是一个好习惯。

双指针

由于我们总是比较 `len`、`tot` 和 `k` 三者的关系。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

因此我们可以使用「滑动窗口」的思路，动态维护一个左右区间 `[j, i]` 和维护窗口内和 `tot`。

右端点一直右移，左端点在窗口不满足「`len - tol <= k`」的时候进行右移。

即可做到线程扫描的复杂度：

```
class Solution {
    public int longestOnes(int[] nums, int k) {
        int n = nums.length;
        int ans = 0;
        for (int i = 0, j = 0, tot = 0; i < n; i++) {
            tot += nums[i];
            while ((i - j + 1) - tot > k) tot -= nums[j++];
            ans = Math.max(ans, i - j + 1);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

总结

除了掌握本题解法以外，我还希望你能理解这几种解法是如何被想到的（特别是如何从「动态规划」想到「二分」）。

根据数据范围（复杂度）调整自己所使用的算法的分析能力，比解决该题本身更加重要。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1052. 爱生气的书店老板** ，难度为 **中等**。

Tag：「滑动窗口」、「双指针」

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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.
```

提示：

- 1 <= X <= customers.length == grumpy.length <= 20000
- 0 <= customers[i] <= 1000
- 0 <= grumpy[i] <= 1

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

滑动窗口

由于「技巧」只会将情绪将「生气」变为「不生气」，不生气仍然是不生气。

1. 我们可以先将原本就满意的客户加入答案，同时将对应的 `customers[i]` 变为 0。
2. 之后的问题转化为：在 `customers` 中找到连续一段长度为 `x` 的子数组，使得其总和最大。这部分就是我们应用技巧所得到的客户。

```
class Solution {
    public int maxSatisfied(int[] cs, int[] grumpy, int x) {
        int n = cs.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (grumpy[i] == 0) {
                ans += cs[i];
                cs[i] = 0;
            }
        }
        int max = 0, cur = 0;
        for (int i = 0, j = 0; i < n; i++) {
            cur += cs[i];
            if (i - j + 1 > x) cur -= cs[j++];
            max = Math.max(max, cur);
        }
        return ans + max;
    }
}
```

- 时间复杂度：$O(n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

- 空间复杂度：$O(1)$

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1208. 尽可能使字符串相等** ，难度为 **中等**。

Tag : 「前缀和」、「二分」、「滑动窗口」

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。
```

示例 2：

```
输入：s = "abcd", t = "cdef", maxCost = 3

输出：1

解释：s 中的任一字符要想变成 t 中对应的字符，其开销都是 2。因此，最大长度为 1。
```

示例 3：

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

输入：s = "abcd", t = "acde", maxCost = 0

输出：1

解释：a -> a, cost = 0，字符串未发生变化，所以最大长度为 1。

提示：

- 1 <= s.length, t.length <= 10^5
- 0 <= maxCost <= 10^6
- s 和 t 都只含小写英文字母。

前缀和 + 二分 + 滑动窗口

给定了长度相同的 `s` 和 `t`，那么对于每一位而言，修改的成本都是相互独立而确定的。

我们可以先预处理出修改成本的前缀和数组 `sum`。

当有了前缀和数组之后，对于任意区间 `[i,j]` 的修改成本，便可以通过 `sum[j] - sum[i - 1]` 得出。

那么接下来我们只需要找出成本不超过 `maxCost` 的最大长度区间，这个长度区间其实就是滑动窗口长度，滑动窗口长度的范围为 `[1, n]` (n 为字符串的长度)。

通过枚举来找答案可以吗？

我们可以通过数据范围大概分析一下哈，共有 `n` 个滑动窗口长度要枚举，复杂度为 $O(n)$，对于每个滑动窗口长度，需要对整个前缀和数组进行滑动检查，复杂度为 $O(n)$。也就是整体复杂度是 $O(n^2)$ 的。

数据范围是 10^5，那么单个样本的计算量是 10^{10}，计算机单秒肯定算不完，会超时 ~

所以我们直接放弃通过枚举的朴素做法。

那么如何优化呢？其实有了对于朴素解法的分析之后，无非就是两个方向：

1. 优化第一个 $O(n)$：减少需要枚举的滑动窗口长度
2. 优化第二个 $O(n)$：实现不完全滑动前缀和数组，也能确定滑动窗口长度是否合法

宫水三叶的刷题日记
公众号: 宫水三叶的刷题日记

事实上第 2 点是无法实现的，我们只能「减少需要枚举的滑动窗口长度」。

一个 $O(n)$ 的操作往下优化，通常就是优化成 $O(\log n)$，$O(\log n)$ 基本上我们可以先猜一个「二分」查找。

然后我们再来分析是否可以二分：假设我们有满足要求的长度 `ans`，那么在以 `ans` 为分割点的数轴上（数轴的范围是滑动窗口长度的范围：`[1, n]`）：

1. 所有满足 `<= ans` 的点的修改成本必然满足 `<= maxCost`
2. 所有满足 `> ans` 的点的修改成本必然满足 `> maxCost`（否则 `ans` 就不会是答案）

因此 `ans` 在数轴 `[1, n]` 上具有二段性，我们可以使用「二分」找 `ans`。得证「二分」的合理性。

可见二分的本质是二段性，而非单调性。

编码细节：

1. 为了方便的预处理前缀和和减少边界处理，我会往字符串头部添加一个空格，使之后的数组下标从 1 开始
2. 二分出来滑动窗口长度，需要在返回时再次检查，因为可能没有符合条件的有效滑动窗口长度

代码：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```java
class Solution {
    public int equalSubstring(String ss, String tt, int max) {
        int n = ss.length();
        ss = " " + ss;
        tt = " " + tt;
        char[] s = ss.toCharArray();
        char[] t = tt.toCharArray();
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + Math.abs(s[i] - t[i]);
        int l = 1, r = n;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (check(sum, mid, max)) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        return check(sum, r, max) ? r : 0;
    }
    boolean check(int[] nums, int mid, int max) {
        for (int i = mid; i < nums.length; i++) {
            int tot = nums[i] - nums[i - mid];
            if (tot <= max) return true;
        }
        return false;
    }
}
```

- 时间复杂度：预处理出前缀和的复杂度为 $O(n)$；二分出「滑动窗口长度」的复杂度为 $O(\log n)$，对于每个窗口长度，需要扫描一遍数组进行检查，复杂度为 $O(n)$，因此二分的复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：使用了前缀和数组。复杂度为 $O(n)$

**🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1423. 可获得的最大点数** ，难度为 **中等**。

Tag：「滑动窗口」

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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
```

示例 2：

```
输入：cardPoints = [2,2,2], k = 2

输出：4

解释：无论你拿起哪两张卡牌，可获得的点数总是 4 。
```

示例 3：

```
输入：cardPoints = [9,7,7,9,7,7,9], k = 7

输出：55

解释：你必须拿起所有卡牌，可以获得的点数为所有卡牌的点数之和。
```

示例 4：

```
输入：cardPoints = [1,1000,1], k = 1

输出：1

解释：你无法拿到中间那张卡牌，所以可以获得的最大点数为 1 。
```

示例 5：

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

```
输入：cardPoints = [1,79,80,1,1,1,200,1], k = 3

输出：202
```

提示：

- 1 <= cardPoints.length <= 10^5
- 1 <= cardPoints[i] <= 10^4
- 1 <= k <= cardPoints.length

滑动窗口

简单的推导：

从两边选择卡片，选择 `k` 张，卡片的总数量为 `n` 张，即有 `n - k` 张不被选择。

所有卡边的总和 `sum` 固定，要使选择的 `k` 张的总和最大，反过来就是要让不被选择的 `n - k` 张的总和最小。

可以使用滑动窗口来计算 `n - k` 张卡片的最小总和 `min`，最终答案为 `sum - min`。

以下代码，可以作为滑动窗口模板使用：

PS. 你会发现以下代码和 643. 子数组最大平均数 I 代码很相似，因为是一套模板，所以说这道其实是道简单题，只是多了一个小学奥数难度的等式推导过程 ~

1. 初始化将滑动窗口压满，取得第一个滑动窗口的目标值
2. 继续滑动窗口，每往前滑动一次，需要删除一个和添加一个元素

代码：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```java
class Solution {
    public int maxScore(int[] nums, int k) {
        int n = nums.length, len = n - k;
        int sum = 0, cur = 0;
        for (int i = 0; i < n; i++) sum += nums[i];
        for (int i = 0; i < len; i++) cur += nums[i];
        int min = cur;
        for (int i = len; i < n; i++) {
            cur = cur + nums[i] - nums[i - len];
            min = Math.min(min, cur);
        }
        return sum - min;
    }
}
```

- 时间复杂度：每个元素最多滑入和滑出窗口一次。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

**🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1438. 绝对差不超过限制的最长连续子数组** ，难度为 **中等**。

Tag：「滑动窗口」、「单调队列」、「二分」

给你一个整数数组 nums ，和一个表示限制的整数 limit，请你返回最长连续子数组的长度，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit 。

如果不存在满足条件的子数组，则返回 0 。

示例 1：

输入：nums = [8,2,4,7], limit = 4
输出：2
解释：所有子数组如下：
[8] 最大绝对差 |8-8| = 0 <= 4.
[8,2] 最大绝对差 |8-2| = 6 > 4.
[8,2,4] 最大绝对差 |8-2| = 6 > 4.
[8,2,4,7] 最大绝对差 |8-2| = 6 > 4.
[2] 最大绝对差 |2-2| = 0 <= 4.
[2,4] 最大绝对差 |2-4| = 2 <= 4.
[2,4,7] 最大绝对差 |2-7| = 5 > 4.
[4] 最大绝对差 |4-4| = 0 <= 4.
[4,7] 最大绝对差 |4-7| = 3 <= 4.
[7] 最大绝对差 |7-7| = 0 <= 4.
因此，满足题意的最长子数组的长度为 2 。

示例 2：

输入：nums = [10,1,2,4,7,2], limit = 5
输出：4
解释：满足题意的最长子数组是 [2,4,7,2]，其最大绝对差 |2-7| = 5 <= 5 。

示例 3：

输入：nums = [4,2,2,2,4,4,2,2], limit = 0
输出：3

提示：

- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^9
- 0 <= limit <= 10^9

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

二分 + 滑动窗口

数据范围是 10^5，因此只能考虑「对数解法」和「线性解法」。

对数解法很容易想到「二分」。

在给定 `limit` 的情况下，倘若有「恰好」满足条件的区间长度为 `len`，必然存在满足条件且长度小于等于 `len` 的区间，同时必然不存在长度大于 `len` 且满足条件的区间。

因此长度 `len` 在数轴中具有「二段性」。

问题转化为「如何判断 `nums` 中是否有长度 `len` 的区间满足绝对值不超过 `limit`」

我们可以枚举区间的右端点 `r`，那么对应的左端点为 `r - len + 1`，然后使用「单调队列」来保存区间的最大值和最小值。

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

```Java
class Solution {
    public int longestSubarray(int[] nums, int limit) {
        int n = nums.length;
        int l = 1, r = n;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (check(nums, mid, limit)) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        return r;
    }
    boolean check(int[] nums, int len, int limit) {
        int n = nums.length;
        Deque<Integer> max = new ArrayDeque<>(), min = new ArrayDeque<>();
        for (int r = 0, l = r - len + 1; r < n; r++, l = r - len + 1) {
            if (!max.isEmpty() && max.peekFirst() < l) max.pollFirst();
            while (!max.isEmpty() && nums[r] >= nums[max.peekLast()]) max.pollLast();
            max.addLast(r);
            if (!min.isEmpty() && min.peekFirst() < l) min.pollFirst();
            while (!min.isEmpty() && nums[r] <= nums[min.peekLast()]) min.pollLast();
            min.addLast(r);
            if (l >= 0 && Math.abs(nums[max.peekFirst()] - nums[min.peekFirst()]) <= limit
        }
        return false;
    }
}
```

- 时间复杂度：枚举长度的复杂度为 $O(\log n)$，对于每次 `check` 而言，每个元素最多入队和出队常数次，复杂度为 $O(n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

双指针

上述解法我们是在对 `len` 进行二分，而事实上我们可以直接使用「双指针」解法找到最大值。

始终让右端点 `r` 右移，当不满足条件时让 `l` 进行右移。

同时，还是使用「单调队列」保存我们的区间最值，这样我们只需要对数组进行一次扫描即可得到答案。

```
class Solution {
    public int longestSubarray(int[] nums, int limit) {
        int n = nums.length;
        int ans = 0;
        Deque<Integer> max = new ArrayDeque<>(), min = new ArrayDeque<>();
        for (int r = 0, l = 0; r < n; r++) {
            while (!max.isEmpty() && nums[r] >= nums[max.peekLast()]) max.pollLast();
            while (!min.isEmpty() && nums[r] <= nums[min.peekLast()]) min.pollLast();
            max.addLast(r);
            min.addLast(r);
            while (Math.abs(nums[max.peekFirst()] - nums[min.peekFirst()]) > limit) {
                l++;
                if (max.peekFirst() < l) max.pollFirst();
                if (min.peekFirst() < l) min.pollFirst();
            }
            ans = Math.max(ans, r - l + 1);
        }
        return ans;
    }
}
```

- 时间复杂度：每个元素最多入队和出队常数次，复杂度为 $O(n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

- 空间复杂度：$O(n)$

**🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1838. 最高频元素的频数** ，难度为 **中等**。

Tag：「枚举」、「哈希表」、「排序」、「前缀和」、「二分」、「滑动窗口」、「双指针」

元素的**频数**是该元素在一个数组中出现的次数。

给你一个整数数组 $nums$ 和一个整数 k 。

在一步操作中，你可以选择 $nums$ 的一个下标，并将该下标对应元素的值增加 1 。

执行最多 k 次操作后，返回数组中最高频元素的**最大可能频数**。

示例 1：

```
输入：nums = [1,2,4], k = 5

输出：3

解释：对第一个元素执行 3 次递增操作，对第二个元素执 2 次递增操作，此时 nums = [4,4,4] 。
4 是数组中最高频元素，频数是 3 。
```

示例 2：

```
输入：nums = [1,4,8,13], k = 5

输出：2

解释：存在多种最优解决方案：
- 对第一个元素执行 3 次递增操作，此时 nums = [4,4,8,13] 。4 是数组中最高频元素，频数是 2 。
- 对第二个元素执行 4 次递增操作，此时 nums = [1,8,8,13] 。8 是数组中最高频元素，频数是 2 。
- 对第三个元素执行 5 次递增操作，此时 nums = [1,4,13,13] 。13 是数组中最高频元素，频数是 2 。
```

示例 3：

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

```
输入：nums = [3,9,6], k = 2

输出：1
```

提示：

- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^5
- 1 <= k <= 10^5

枚举

一个朴素的做法是，先对原数组 $nums$ 进行排序，然后枚举最终「频数对应值」是哪个。

利用每次操作只能对数进行加一，我们可以从「频数对应值」开始往回检查，从而得出在操作次数不超过 k 的前提下，以某个值作为「频数对应值」最多能够凑成多少个。

算法整体复杂度为 $O(n^2)$，Java 2021/07/19 可过。

代码：

宫水三叶 の 刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    public int maxFrequency(int[] nums, int k) {
        int n = nums.length;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i : nums) map.put(i, map.getOrDefault(i, 0) + 1);
        List<Integer> list = new ArrayList<>(map.keySet());
        Collections.sort(list);
        int ans = 1;
        for (int i = 0; i < list.size(); i++) {
            int x = list.get(i), cnt = map.get(x);
            if (i > 0) {
                int p = k;
                for (int j = i - 1; j >= 0; j--) {
                    int y = list.get(j);
                    int diff = x - y;
                    if (p >= diff) {
                        int add = p / diff;
                        int min = Math.min(map.get(y), add);
                        p -= min * diff;
                        cnt += min;
                    } else {
                        break;
                    }
                }
            }
            ans = Math.max(ans, cnt);
        }
        return ans;
    }
}
```

- 时间复杂度：得到去重后的频数后选集合复杂度为 $O(n)$；最坏情况下去重后仍有 n 个频数，且判断 k 次操作内某个频数最多凑成多少复杂度为 $O(n)$。整体复杂度为 $O(n^2)$
- 空间复杂度：$O(n)$

排序 + 前缀和 + 二分 + 滑动窗口

先对原数组 $nums$ 进行从小到大排序，如果存在真实最优解 len，意味着至少存在一个大小为 len 的区间 $[l, r]$，使得在操作次数不超过 k 的前提下，区间 $[l, r]$ 的任意值 $nums[i]$ 的值调整为 $nums[r]$。

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

这引导我们利用「数组有序」&「前缀和」快速判断「某个区间 $[l, r]$ 是否可以在 k 次操作内将所有值变为 $nums[r]$」：

具体的，我们可以二分答案 len 作为窗口长度，利用前缀和我们可以在 $O(1)$ 复杂度内计算任意区间的和，同时由于每次操作只能对数进行加一，即窗口内的所有数最终变为 $nums[r]$ ，最终目标区间和为 $nums[r] * len$，通过比较目标区间和和真实区间和的差值，我们可以知道 k 次操作是否能将当前区间变为 $nums[r]$。

上述判断某个值 len 是否可行的 `check` 操作复杂度为 $O(n)$，因此算法复杂度为 $O(n \log n)$。

代码：

```
class Solution {
    int[] nums, sum;
    int n, k;
    public int maxFrequency(int[] _nums, int _k) {
        nums = _nums;
        k = _k;
        n = nums.length;
        Arrays.sort(nums);
        sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + nums[i - 1];
        int l = 0, r = n;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (check(mid)) l = mid;
            else r = mid - 1;
        }
        return r;
    }
    boolean check(int len) {
        for (int l = 0; l + len - 1 < n; l++) {
            int r = l + len - 1;
            int cur = sum[r + 1] - sum[l];
            int t = nums[r] * len;
            if (t - cur <= k) return true;
        }
        return false;
    }
}
```

- 时间复杂度：排序的复杂度为 $O(n \log n)$；计算前缀和数组复杂度为 $O(n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

；`check` 函数的复杂度为 $O(n)$，因此二分复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$

- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡**更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。**

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「滑动窗口」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

宫水三叶 の 刷题日记
公众号: 宫水三叶的刷题日记

"给作者手机充个电"

YOLO 的赞赏码

版权声明：任何形式的转载请保留出处 Wiki。