宫水三叶的刷题日记

双指针

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「双指针」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「双指针」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「双指针」；
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 **11. 盛最多水的容器** ，难度为 **中等**。

Tag：「双指针」、「贪心」

给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。

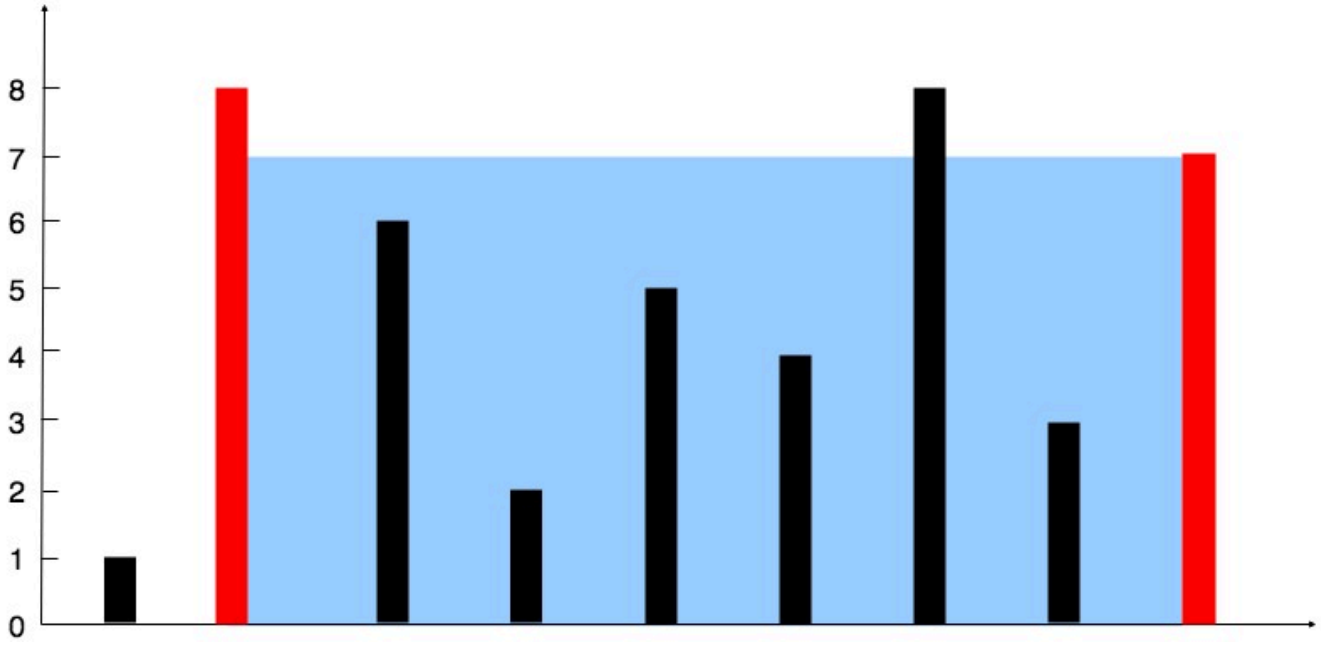
在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0) 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

说明：你不能倾斜容器。

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

示例 2：

输入：height = [1,1]
输出：1

示例 3：

输入：height = [4,3,2,1,4]
输出：16

示例 4：

输入：height = [1,2,1]
输出：2

提示：

- n = height.length
- 2 <= n <= 3 * 10^4
- 0 <= height[i] <= 3 * 10^4

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

朴素解法

我们可以直接枚举所有的情况：先枚举确定左边界，再往右枚举确定右边界。

然后再记录枚举过程中的最大面积即可：

代码：

```
class Solution {
    public int maxArea(int[] height) {
        int n = height.length;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int w = j - i;
                int h = Math.min(height[i], height[j]);
                ans = Math.max(w * h, ans);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(1)$

双指针 + 贪心

先用两个指针 `i` 和 `j` 指向左右边界，然后考虑指针应该怎么移动。

由于构成矩形的面积，取决于 `i` 和 `j` 之间的距离（记为 `w`） 和 `i` 和 `j` 下标对应的高度的最小值（记为 `h`）。

首先无论是 `i` 指针往右移动还是 `j` 指针往左移动都会导致 `w` 变小，所以想要能够枚举到更大的面积，我们应该让 `h` 在指针移动后变大。

不妨假设当前情况是 `height[i] < heigth[j]`（此时矩形的高度为 `height[i]`），然后分情况讨论：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

- 让 `i` 和 `j` 两者高度小的指针移动，即 `i` 往右移动：
    - 移动后，i 指针对应的高度变小，即 `height[i] > height[i + 1]`：`w` 和 `h` 都变小了，面积一定变小
    - 移动后，i 指针对应的高度不变，即 `height[i] = height[i + 1]`：`w` 变小，`h` 不变，面积一定变小
    - 移动后，i 指针对应的高度变大，即 `height[i] < height[i + 1]`：`w` 变小，`h` 变大，面积可能会变大
- 让 `i` 和 `j` 两者高度大的指针移动，即 `j` 往左移动：
    - 移动后，j 指针对应的高度变小，即 `height[j] > height[j - 1]`：`w` 变小，`h` 可能不变或者变小（当 `height[j - 1] >= height[i]` 时，`h` 不变；当 `height[j - 1] < height[i]` 时，`h` 变小），面积一定变小
    - 移动后，j 指针对应的高度不变，即 `height[j] = height[j - 1]`：`w` 变小，`h` 不变，面积一定变小
    - 移动后，j 指针对应的高度变大，即 `height[j] < height[j - 1]`：`w` 变小，`h` 不变，面积一定变小

综上所述，我们只有将高度小的指针往内移动，才会枚举到更大的面积：

代码：

```
class Solution {
    public int maxArea(int[] height) {
        int n = height.length;
        int i = 0, j = n - 1;
        int ans = 0;
        while (i < j) {
            ans = Math.max(ans, (j - i) * Math.min(height[i], height[j]));
            if (height[i] < height[j]) {
                i++;
            } else {
                j--;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：会对整个数组扫描一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

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 **

题目描述

这是 LeetCode 上的 **15. 三数之和** ，难度为 **中等**。

Tag：「双指针」、「排序」、「n 数之和」

给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？

请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
```

示例 2：

```
输入：nums = []
输出：[]
```

示例 3：

```
输入：nums = [0]
输出：[]
```

提示：

- 0 <= nums.length <= 3000
- -10^5 <= nums[i] <= 10^5

排序 + 双指针

对数组进行排序，使用三个指针 `i` 、 `j` 和 `k` 分别代表要找的三个数。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

1. 通过枚举 `i` 确定第一个数，另外两个指针 `j`，`k` 分别从左边 `i + 1` 和右边 `n - 1` 往中间移动，找到满足 `nums[i] + nums[j] + nums[k] == 0` 的所有组合。
2. `j` 和 `k` 指针的移动逻辑，分情况讨论 `sum = nums[i] + nums[j] + nums[k]`：
    - `sum` > 0：`k` 左移，使 `sum` 变小
    - `sum` < 0：`j` 右移，使 `sum` 变大
    - `sum` = 0：找到符合要求的答案，存起来

由于题目要求答案不能包含重复的三元组，所以在确定第一个数和第二个数的时候，要跳过数值一样的下标（在三数之和确定的情况下，确保第一个数和第二个数不会重复，即可保证三元组不重复）。

代码：

```
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        int n = nums.length;
        List<List<Integer>> ans = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            int j = i + 1, k = n - 1;
            while (j < k) {
                while (j > i + 1 && j < n && nums[j] == nums[j - 1]) j++;
                if (j >= k) break;
                int sum = nums[i] + nums[j] + nums[k];
                if (sum == 0) {
                    ans.add(Arrays.asList(nums[i], nums[j], nums[k]));
                    j++;
                } else if (sum > 0) {
                    k--;
                } else if (sum < 0) {
                    j++;
                }
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序的复杂度为 $O(logN)$，对于每个 `i` 而言，最坏的情况 `j` 和 `k` 都要扫描一遍数组的剩余部分，复杂度为 $O(n^2)$。整体复杂度为 $O(n^2)$

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：$O(n^2)$

> **🍭 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 **16. 最接近的三数之和** ，难度为 **中等**。

Tag：「双指针」、「排序」、「n 数之和」

给定一个包括 n 个整数的数组 nums 和 一个目标值 target。

找出 nums 中的三个整数，使得它们的和与 target 最接近。

返回这三个数的和。

假定每组输入只存在唯一答案。

示例：

```
输入：nums = [-1,2,1,-4], target = 1

输出：2

解释：与 target 最接近的和是 2 (-1 + 2 + 1 = 2) 。
```

提示：

- 3 <= nums.length <= 10^3
- -10^3 <= nums[i] <= 10^3
- -10^4 <= target <= 10^4

排序 + 双指针

这道题的思路和「15. 三数之和（中等）」区别不大。

对数组进行排序，使用三个指针 `i` 、 `j` 和 `k` 分别代表要找的三个数。

1. 通过枚举 `i` 确定第一个数，另外两个指针 `j` ， `k` 分别从左边 `i + 1` 和右边

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

`n - 1` 往中间移动，找到满足 `nums[i] + nums[j] + nums[k]` 最接近 `target` 的唯一解。

2. `j` 和 `k` 指针的移动逻辑，分情况讨论 `sum = nums[i] + nums[j] + nums[k]` ：
    - `sum` > `target` ：`k` 左移，使 `sum` 变小
    - `sum` < `target` ：`j` 右移，使 `sum` 变大
    - `sum` = `target` ：找到最符合要求的答案，直接返回

为了更快找到答案，对于相同的 `i` ，可以直接跳过下标。

代码：

```
class Solution {
    public int threeSumClosest(int[] nums, int t) {
        Arrays.sort(nums);
        int ans = nums[0] + nums[1] + nums[2];
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            int j = i + 1, k = n - 1;
            while (j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                if (Math.abs(sum - t) < Math.abs(ans - t)) ans = sum;
                if (ans == t) {
                    return t;
                } else if (sum > t) {
                    k--;
                } else if (sum < t) {
                    j++;
                }
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序的复杂度为 $O(logN)$，对于每个 `i` 而言，最坏的情况 `j` 和 `k` 都要扫描一遍数组的剩余部分，复杂度为 $O(n^2)$。整体复杂度为 $O(n^2)$
- 空间复杂度：$O(n^2)$

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

宫水三叶 刷题日记 公众号：宫水三叶的刷题日记

题目描述

这是 LeetCode 上的 **18. 四数之和** ，难度为 **中等**。

Tag：「双指针」、「排序」、「n 数之和」

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？

找出所有满足条件且不重复的四元组。

注意：答案中不可以包含重复的四元组。

示例 1：

```
输入：nums = [1,0,-1,0,-2,2], target = 0

输出：[[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]
```

示例 2：

```
输入：nums = [], target = 0

输出：[]
```

提示：

- 0 <= nums.length <= 200
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9

排序 + 双指针

这道题的思路和「15. 三数之和（中等）」、「16. 最接近的三数之和（中等）」类似。

对数组进行排序，使用四个指针 `i` 、 `j` 、 `k` 和 `p` 分别代表要找的四个数。

1. 通过枚举 `i` 确定第一个数，枚举 `j` 确定第二个数，另外两个指针 `k` 和 `p` 分别

公众号：宫水三叶的刷题日记

从左边 `j + 1` 和右边 `n - 1` 往中间移动，找到满足 `nums[i] + nums[j] + nums[k] + nums[p] == t` 的所有组合。

2. `k` 和 `p` 指针的移动逻辑，分情况讨论 `sum = nums[i] + nums[j] + nums[k] + nums[p]` ：
    - `sum` > `target` ：`p` 左移，使 `sum` 变小
    - `sum` < `target` ：`k` 右移，使 `sum` 变大
    - `sum` = `target` ：将组合加入结果集，`k` 右移继续进行检查

题目要求不能包含重复元素，所以我们要对 `i` 、 `j` 和 `k` 进行去重，去重逻辑是对于相同的数，只使用第一个。

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<List<Integer>> fourSum(int[] nums, int t) {
        Arrays.sort(nums);
        int n = nums.length;
        List<List<Integer>> ans = new ArrayList<>();
        for (int i = 0; i < n; i++) { // 确定第一个数
            if (i > 0 && nums[i] == nums[i - 1]) continue; // 对第一个数进行去重（相同的数只取第
            for (int j = i + 1; j < n; j++) { // 确定第二个数
                if (j > i + 1 && nums[j] == nums[j - 1]) continue; // 对第二个数进行去重（相同
                // 确定第三个数和第四个数
                int k = j + 1, p = n - 1;
                while (k < p) {

                    // 对第三个数进行去重（相同的数只取第一个）
                    while (k > j + 1 && k < n && nums[k] == nums[k - 1]) k++;
                    // 如果 k 跳过相同元素之后的位置超过了 p，本次循环结束
                    if (k >= p) break;

                    int sum = nums[i] + nums[j] + nums[k] + nums[p];
                    if (sum == t) {
                        ans.add(Arrays.asList(nums[i], nums[j], nums[k], nums[p]));
                        k++;
                    } else if (sum > t) {
                        p--;
                    } else if (sum < t) {
                        k++;
                    }
                }
            }
        }
        return ans;
    }
}
```

- 时间复杂度：`i` 和 `j` 是直接枚举确定，复杂度为 $O(n^2)$，当确定下来 `i` 和 `j` 之后，通过双指针确定 `k` 和 `p` ，也就是对于每一组 `i` 和 `j` 而言复杂度为 $O(n)$。总的复杂度为 $O(n^3)$
- 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 **19. 删除链表的倒数第 N 个结点** ，难度为 **中等**。

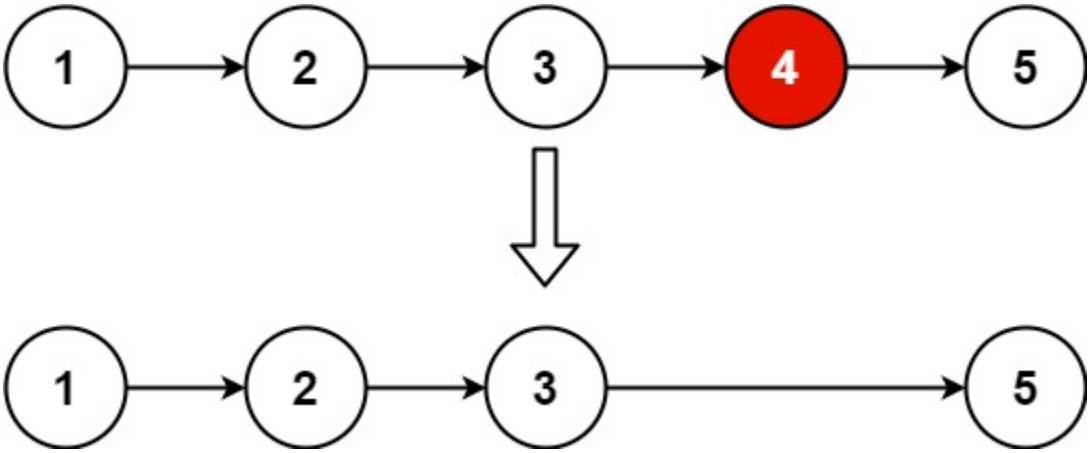
宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

Tag：「链表」、「快慢指针」、「双指针」

给你一个链表，删除链表的倒数第 `n` 个结点，并且返回链表的头结点。

进阶：你能尝试使用一趟扫描实现吗？

示例 1：




```
输入：head = [1,2,3,4,5], n = 2

输出：[1,2,3,5]
```

示例 2：

```
输入：head = [1], n = 1

输出：[]
```

示例 3：

```
输入：head = [1,2], n = 1

输出：[1]
```

提示：

- 链表中结点的数目为 sz
- 1 <= sz <= 30
- 0 <= Node.val <= 100

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

- 1 <= n <= sz

快慢指针

删除链表的倒数第 `n` 个结点，首先要确定倒数第 `n` 个节点的位置。

我们可以设定两个指针，分别为 `slow` 和 `fast`，刚开始都指向 head。

然后先让 `fast` 往前走 `n` 步，`slow` 指针不动，这时候两个指针的距离为 `n`。

再让 `slow` 和 `fast` 同时往前走（保持两者距离不变），直到 `fast` 指针到达结尾的位置。

这时候 `slow` 会停在待删除节点的前一个位置，让 `slow.next = slow.next.next` 即可。

但这里有一个需要注意的边界情况是：如果链表的长度是 `L`，而我们恰好要删除的是倒数第 `L` 个节点（删除头节点），这时候 `fast` 往前走 `n` 步之后会变为 null，此时我们只需要让 `head = slow.next` 即可删除。

代码：

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        if (head.next == null) return null;

        ListNode slow = head;
        ListNode fast = head;
        while (n-- > 0) fast = fast.next;

        if (fast == null) {
            head = slow.next;
        } else {
            while (fast.next != null) {
                slow = slow.next;
                fast = fast.next;
            }
            slow.next = slow.next.next;
        }
        return head;
    }
}
```

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

- 时间复杂度：需要扫描的长度为链表的长度。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 **26. 删除有序数组中的重复项** ，难度为 **简单**。

Tag：「数组」、「双指针」、「数组移除元素问题」

给你一个有序数组 nums ，请你 原地 删除重复出现的元素，使每个元素 只出现一次 ，返回删除后数组的新长度。

不要使用额外的数组空间，你必须在 原地 修改输入数组 并在使用 O(1) 额外空间的条件下完成。

说明：

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

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：nums = [1,1,2]

输出：2, nums = [1,2]

解释：函数应该返回新的长度 2 ，并且原数组 nums 的前两个元素被修改为 1, 2 。不需要考虑数组中超出新长度后面的元素。

示例 2：

输入：nums = [0,0,1,1,1,2,2,3,3,4]

输出：5, nums = [0,1,2,3,4]

解释：函数应该返回新的长度 5 ， 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4 。不需要考虑数组中超出新长度后面的元素

提示：

- 0 <= nums.length <= 3 * 10^4
- -10^4 <= nums[i] <= 10^4
- nums 已按升序排列

双指针解法

一个指针 `i` 进行数组遍历，另外一个指针 `j` 指向有效数组的最后一个位置。

只有当 `i` 所指向的值和 `j` 不一致（不重复），才将 `i` 的值添加到 `j` 的下一位置。

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 removeDuplicates(int[] nums) {
        int n = nums.length;
        int j = 0;
        for (int i = 0; i < n; i++) {
            if (nums[i] != nums[j]) {
                nums[++j] = nums[i];
            }
        }
        return j + 1;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

通用解法

为了让解法更具有一般性，我们将原问题的「最多保留 1 位」修改为「最多保留 k 位」。

对于此类问题，我们应该进行如下考虑：

- 由于是保留 `k` 个相同数字，**对于前 `k` 个数字，我们可以直接保留。**
- 对于后面的任意数字，能够保留的前提是：**与当前写入的位置前面的第 `k` 个元素进行比较，不相同则保留。**

举个🌰，我们令 `k=1`，假设有样例：`[3,3,3,3,4,4,4,5,5,5]`

0. 设定变量 `idx`，指向待插入位置。`idx` 初始值为 `0`，目标数组为 `[]`
1. 首先我们先让第 `1` 位直接保留（性质 1）。`idx` 变为 `1`，目标数组为 `[3]`
2. 继续往后遍历，能够保留的前提是与 `idx` 的前面 `1` 位元素不同（性质 2），因此我们会跳过剩余的 `3`，将第一个 `4` 追加进去。`idx` 变为 `2`，目标数组为 `[3,4]`
3. 继续这个过程，跳过剩余的 `4`，将第一个 `5` 追加进去。`idx` 变为 `3`，目标数组为 `[3,4,5]`
4. 当整个数组被扫描完，最终我们得到了目标数组 `[3,4,5]` 和 答案 `idx` 为 `3`。

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 int removeDuplicates(int[] nums) {
        return process(nums, 1);
    }
    int process(int[] nums, int k) {
        int idx = 0;
        for (int x : nums) {
            if (idx < k || nums[idx - k] != x) nums[idx++] = x;
        }
        return idx;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

基于上述解法我们还能做一点小剪枝：**利用目标数组的最后一个元素必然与原数组的最后一个元素相同进行剪枝，从而确保当数组有超过 `k` 个最大值时，数组不会被完整扫描。**

但需要注意这种「剪枝」同时会让我们单次循环的常数变大，所以仅作为简单拓展。

代码：

```Java
class Solution {
    public int removeDuplicates(int[] nums) {
        int n = nums.length;
        if (n <= 1) return n;
        return process(nums, 1, nums[n - 1]);
    }
    int process(int[] nums, int k, int max) {
        int idx = 0;
        for (int x : nums) {
            if (idx < k || nums[idx - k] != x) nums[idx++] = x;
            if (idx - k >= 0 && nums[idx - k] == max) break;
        }
        return idx;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

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 **

题目描述

这是 LeetCode 上的 **27. 移除元素** ，难度为 **简单**。

Tag：「数组」、「双指针」、「数组移除元素问题」

给你一个数组 nums 和一个值 val，你需要「原地」移除所有数值等于 val 的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 $O(1)$ 额外空间并「原地」修改输入数组。

元素的顺序可以改变。

你不需要考虑数组中超出新长度后面的元素。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度, 它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

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：nums = [3,2,2,3], val = 3

输出：2, nums = [2,2]

解释：函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。你不需要考虑数组中超出新长度后面的元素。例如，函数返回的新长度
```

示例 2：

```
输入：nums = [0,1,2,2,3,0,4,2], val = 2

输出：5, nums = [0,1,4,0,3]

解释：函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。注意这五个元素可为任意顺序。你不需要考虑数组中超出
```

提示：

- 0 <= nums.length <= 100
- 0 <= nums[i] <= 50
- 0 <= val <= 100

双指针解法

本解法的思路与 【题解】26. 删除排序数组中的重复项 中的「双指针解法」类似。

根据题意，我们可以将数组分成「前后」两段：

- 前半段是有效部分，存储的是不等于 `val` 的元素。
- 后半段是无效部分，存储的是等于 `val` 的元素。

最终答案返回有效部分的结尾下标。

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 removeElement(int[] nums, int val) {
        int j = nums.length - 1;
        for (int i = 0; i <= j; i++) {
            if (nums[i] == val) {
                swap(nums, i--, j--);
            }
        }
        return j + 1;
    }
    void swap(int[] nums, int i, int j) {
        int tmp = nums[i];
        nums[i] = nums[j];
        nums[j] = tmp;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

通用解法

本解法的思路与 【题解】26. 删除排序数组中的重复项 中的「通用解法」类似。

先设定变量 `idx`，指向待插入位置。`idx` 初始值为 0

然后从题目的「要求/保留逻辑」出发，来决定当遍历到任意元素 `x` 时，应该做何种决策：

- 如果当前元素 `x` 与移除元素 `val` 相同，那么跳过该元素。
- 如果当前元素 `x` 与移除元素 `val` 不同，那么我们将其放到下标 `idx` 的位置，并让 `idx` 自增右移。

最终得到的 `idx` 即是答案。

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 removeElement(int[] nums, int val) {
        int idx = 0;
        for (int x : nums) {
            if (x != val) nums[idx++] = x;
        }
        return idx;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

总结

对于诸如「相同元素最多保留 `k` 位元素」或者「移除特定元素」的问题，更好的做法是从题目本身性质出发，利用题目给定的要求提炼出具体的「保留逻辑」，将「保留逻辑」应用到我们的遍历到的每一个位置。

> **🍭 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 **45. 跳跃游戏 II** ，难度为 **中等**。

Tag：「贪心」、「线性 DP」、「双指针」

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

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: [2,3,1,1,4]

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。
     从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
```

示例 2:

```
输入: [2,3,0,1,4]

输出: 2
```

提示:

- 1 <= nums.length <= 1000
- 0 <= nums[i] <= 10^5

BFS

对于这一类问题，我们一般都是使用 BFS 进行求解。

本题的 BFS 解法的复杂度是 $O(n^2)$，数据范围为 10^3，可以过。

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 jump(int[] nums) {
        int n = nums.length;
        int ans = 0;
        boolean[] st = new boolean[n];
        Deque<Integer> d = new ArrayDeque<>();
        st[0] = true;
        d.addLast(0);
        while (!d.isEmpty()) {
            int size = d.size();
            while (size-- > 0) {
                int idx = d.pollFirst();
                if (idx == n - 1) return ans;
                for (int i = idx + 1; i <= idx + nums[idx] && i < n; i++) {
                    if (!st[i]) {
                        st[i] = true;
                        d.addLast(i);
                    }
                }
            }
            ans++;
        }
        return ans;
    }
}
```

- 时间复杂度：如果每个点跳跃的距离足够长的话，每次都会将当前点「后面的所有点」进行循环入队操作（由于 st 的存在，不一定都能入队，但是每个点都需要被循环一下）。复杂度为 $O(n^2)$
- 空间复杂度：队列中最多有 $n-1$ 个元素。复杂度为 $O(n)$

双指针 + 贪心 + 动态规划

本题数据范围只有 10^3，所以 $O(n^2)$ 勉强能过。

如果面试官要将数据范围出到 10^6，又该如何求解呢？

我们需要考虑 $O(n)$ 的做法。

其实通过 10^6 这个数据范围，就已经可以大概猜到是道 DP 题。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

我们定义 $f[i]$ 为到达第 i 个位置所需要的最少步数，那么答案是 $f[n-1]$。

学习过 路径 DP 专题 的同学应该知道，通常确定 DP 的「状态定义」有两种方法。

- 一种是根据经验猜一个状态定义，会结合题目给定的维度，和要求的答案去猜。
- 另外一种则是通过设计一个合理的 `DFS` 方法签名来确定状态定义。

这里我是采用第一种方法。

至于如何确定「状态定义」是否可靠，关键是看使用这个状态定义能否推导出合理的「状态转移方程」，来覆盖我们所有的状态。

不失一般性的考虑 $f[n-1]$ 该如何转移：

我们知道最后一个点前面可能会有很多个点能够一步到达最后一个点。

宫水三叶

也就是有 $f[n-1] = min(f[n-k], ..., f[n-3], f[n-2]) + 1$。

然后我们再来考虑集合 $f[n-k], ..., f[n-3], f[n-2]$ 有何特性。

不然发现其实必然有 $f[n-k] <= ... <= f[n-3] <= f[n-2]$。

推而广之，不止是经过一步能够到达最后一个点的集合，其实任意连续的区间都有这个性质。

举个🌰，比如我经过至少 5 步到达第 i 个点，那么必然不可能出现使用步数少于 5 步就能达到第 $i+1$ 个点的情况。到达第 $i+1$ 个点的至少步数必然是 5 步或者 6 步。

搞清楚性质之后，再回头看我们的状态定义：$f[i]$ *为到达第 i 个位置所需要的最少步数。*

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

因此当我们要求某一个 $f[i]$ 的时候，我们需要找到最早能够经过一步到达 i 点的 j 点。

即有状态转移方程：$f[i] = f[j] + 1$。

也就是我们每次都贪心的取离 i 点最远的点 j 来更新 $f[i]$。

而这个找 j 的过程可以使用双指针来找。

因此这个思路其实是一个「双指针 + 贪心 + 动态规划」的一个解法。

代码：

```
class Solution {
    public int jump(int[] nums) {
        int n = nums.length;
        int[] f = new int[n];
        for (int i = 1, j = 0; i < n; i++) {
            while (j + nums[j] < i) j++;
            f[i] = f[j] + 1;
        }
        return f[n - 1];
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

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

题目描述

这是 LeetCode 上的 **88. 合并两个有序数组** ，难度为 **简单**。

Tag：「双指针」、「排序」

给你两个有序整数数组 nums1 和 nums2，请你将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

你可以假设 nums1 的空间大小等于 m + n，这样它就有足够的空间保存来自 nums2 的元素。

示例 1：

```
输入：nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

输出：[1,2,2,3,5,6]
```

示例 2：

```
输入：nums1 = [1], m = 1, nums2 = [], n = 0

输出：[1]
```

提示：

- nums1.length == m + n
- nums2.length == n
- 0 <= m, n <= 200
- 1 <= m + n <= 200
- -10^9 <= nums1[i], nums2[i] <= 10^9

双指针（额外空间）

一个简单的做法是，创建一个和 $nums1$ 等长的数组 arr，使用双指针将 $num1$ 和 $nums2$

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

的数据迁移到 arr。

最后再将 arr 复制到 $nums1$ 中。

代码：

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int total = m + n;
        int[] arr = new int[total];
        int idx = 0;
        for (int i = 0, j = 0; i < m || j < n;) {
            if (i < m && j < n) {
                arr[idx++] = nums1[i] < nums2[j] ? nums1[i++] : nums2[j++];
            } else if (i < m) {
                arr[idx++] = nums1[i++];
            } else if (j < n) {
                arr[idx++] = nums2[j++];
            }
        }
        System.arraycopy(arr, 0, nums1, 0, total);
    }
}
```

- 时间复杂度：$O(m + n)$
- 空间复杂度：$O(m + n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

先合并再排序

我们还可以将 $nums2$ 的内容先迁移到 $nums1$ 去，再对 $nums1$ 进行排序。

代码：

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        System.arraycopy(nums2, 0, nums1, m, n);
        Arrays.sort(nums1);
    }
}
```

- 时间复杂度：$O((m+n)log(m+n))$
- 空间复杂度：$O(1)$

PS. Java 中的 sort 排序是一个综合排序。包含插入/双轴快排/归并/timsort，这里假定 `Arrays.sort` 使用的是「双轴快排」，并忽略递归带来的空间开销。

宫水三叶 の 刷题日记
公众号：宫水三叶的刷题日记

原地合并（从前往后）

也可以直接在 $nums1$ 进行合并操作，但是需要确保每次循环开始，$nums2$ 的指针指向的必然是最小的元素。

因此，我们需要对 $nums2$ 执行局部排序。

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 void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = 0, j = 0;
        while (j < n) {
            if (i >= m) {
                nums1[i] = nums2[j++];
            } else {
                int a = nums1[i], b = nums2[j];
                if (a > b) swap(nums1, i, nums2, j);
                sort(nums2, j, n - 1);
            }
            i++;
        }
    }
    void sort(int[] nums, int l, int r) {
        if (l >= r) return;
        int x = nums[l], i = l - 1, j = r + 1;
        while (i < j) {
            do i++; while (nums[i] < x);
            do j--; while (nums[j] > x);
            if (i < j) swap(nums, i, nums, j);
        }
        sort(nums, l, j);
        sort(nums, j + 1, r);
    }
    void swap(int[] nums1, int i, int[] nums2, int j) {
        int tmp = nums1[i];
        nums1[i] = nums2[j];
        nums2[j] = tmp;
    }
}
```

- 时间复杂度：$O(n + m^2logm)$
- 空间复杂度：忽略递归开销，复杂度为 $O(1)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

原地合并（从后往前）

思路和「方法一」是类似的，将遍历方向由「从前往后」调整为「从后往前」即可做到 $O(1)$ 空间复杂度。

代码：

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i = m - 1, j = n - 1;
        int idx = m + n - 1;
        while (i >= 0 || j >= 0) {
            if (i >= 0 && j >= 0) {
                nums1[idx--] = nums1[i] >= nums2[j] ? nums1[i--] : nums2[j--];
            } else if (i >= 0) {
                nums1[idx--] = nums1[i--];
            } else {
                nums1[idx--] = nums2[j--];
            }
        }
    }
}
```

- 时间复杂度：$O(m + n)$
- 空间复杂度：$O(1)$

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

宫水三叶の刷题日记

公众号：宫水三叶的刷题日记

题目描述

这是 LeetCode 上的 **345. 反转字符串中的元音字母** ，难度为 **简单**。

Tag：「双指针」、「模拟」

编写一个函数，以字符串作为输入，反转该字符串中的元音字母。

示例 1：

```
输入："hello"

输出："holle"
```

示例 2：

```
输入："leetcode"

输出："leotcede"
```

提示：

- 元音字母不包含字母 “y” 。

双指针

一个朴素的做法是利用「双指针」进行前后扫描，当左右指针都是元音字母时，进行互换并移到下一位。

由于元音字母相对固定，因此我们可以使用容器将其存储，并使用 `static` 修饰，确保整个容器的创建和元音字母的填入在所有测试样例中只会发生一次。

我们期望该容器能在 $O(1)$ 的复杂度内判断是否为元音字母，可以使用语言自带的哈希类容器（$P2$ 代码）或是使用数组模拟（$P1$ 代码）。

> 一些细节：由于题目没有说字符串中只包含字母，因此在使用数组模拟哈希表时，我们需要用当前字符减去 ASCII 码的最小值（空字符），而不是 ‘A’

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

代码：

```
class Solution {
    static boolean[] hash = new boolean[128];
    static char[] vowels = new char[]{'a','e','i','o','u'};
    static {
        for (char c : vowels) {
            hash[c - ' '] = hash[Character.toUpperCase(c) - ' '] = true;
        }
    }
    public String reverseVowels(String s) {
        char[] cs = s.toCharArray();
        int n = s.length();
        int l = 0, r = n - 1;
        while (l < r) {
            if (hash[cs[l] - ' '] && hash[cs[r] - ' ']) {
                swap(cs, l++, r--);
            } else {
                if (!hash[cs[l] - ' ']) l++;
                if (!hash[cs[r] - ' ']) r--;
            }
        }
        return String.valueOf(cs);
    }
    void swap(char[] cs, int l, int r) {
        char c = cs[l];
        cs[l] = cs[r];
        cs[r] = c;
    }
}
```

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```Java
class Solution {
    static char[] vowels = new char[]{'a','e','i','o','u'};
    static Set<Character> set = new HashSet<>();
    static {
        for (char c : vowels) {
            set.add(c);
            set.add(Character.toUpperCase(c));
        }
    }
    public String reverseVowels(String s) {
        char[] cs = s.toCharArray();
        int n = s.length();
        int l = 0, r = n - 1;
        while (l < r) {
            if (set.contains(cs[l]) && set.contains(cs[r])) {
                swap(cs, l++, r--);
            } else {
                if (!set.contains(cs[l])) l++;
                if (!set.contains(cs[r])) r--;
            }
        }
        return String.valueOf(cs);
    }
    void swap(char[] cs, int l, int r) {
        char c = cs[l];
        cs[l] = cs[r];
        cs[r] = c;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：由于 `toCharArray` 会创建新数组，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 **395. 至少有 K 个重复字符的最长子串** ，难度为 **中等**。

Tag：「双指针」、「枚举」

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

给你一个字符串 s 和一个整数 k ，请你找出 s 中的最长子串， 要求该子串中的每一字符出现次数都不少于 k 。返回这一子串的长度。

示例 1：

```
输入：s = "aaabb", k = 3
输出：3
解释：最长子串为 "aaa" ，其中 'a' 重复了 3 次。
```

示例 2：

```
输入：s = "ababbc", k = 2
输出：5
解释：最长子串为 "ababb" ，其中 'a' 重复了 2 次， 'b' 重复了 3 次。
```

提示：

- 1 <= s.length <= 10^4
- s 仅由小写英文字母组成
- 1 <= k <= 10^5

枚举 + 双指针

其实看到这道题，我第一反应是「二分」，直接「二分」答案。

但是往下分析就能发现「二分」不可行，因为不具有二段性质。

也就是假设有长度 `t` 的一段区间满足要求的话，`t + 1` 长度的区间是否「一定满足」或者「一定不满足」呢？

显然并不一定，是否满足取决于 `t + 1` 个位置出现的字符在不在原有区间内。

举个🌰吧，假设我们已经画出来一段长度为 `t` 的区间满足要求（且此时 k > 1），那么当我们将长度扩成 `t + 1` 的时候（无论是往左扩还是往右扩）：

- 如果新位置的字符在原有区间**出现过**，那必然还是满足出现次数大于 k，这时候 **`t + 1` 的长度满足要求**
- 如果新位置的字符在原有区间**没出现过**，那新字符的出现次数只有一次，这时候

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

`t + 1` 的长度不满足要求

因此我们无法是使用「二分」，相应的也无法直接使用「滑动窗口」思路的双指针。

因为双指针其实也是利用了二段性质，当一个指针确定在某个位置，另外一个指针能够落在某个明确的分割点，使得左半部分满足，右半部分不满足。

那么还有什么性质可以利用呢？这时候要留意数据范围「数值小」的内容。

题目说明了只包含小写字母（26 个，为有限数据），**我们可以枚举最大长度所包含的字符类型数量，答案必然是 [1, 26]，即最少包含 1 个字母，最多包含 26 个字母。**

你会发现，**当确定了长度所包含的字符种类数量时，区间重新具有了二段性质。**

当我们使用双指针的时候：

1. 右端点往右移动必然会导致字符类型数量增加（或不变）
2. 左端点往右移动必然会导致字符类型数量减少（或不变）

当然，我们还需要记录有多少字符符合要求（出现次数不少于 k），当区间内所有字符都符合时更新答案。

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 longestSubstring(String s, int k) {
        int ans = 0;
        int n = s.length();
        char[] cs = s.toCharArray();
        int[] cnt = new int[26];
        for (int p = 1; p <= 26; p++) {
            Arrays.fill(cnt, 0);
            // tot 代表 [j, i] 区间所有的字符种类数量；sum 代表满足「出现次数不少于 k」的字符种类数量
            for (int i = 0, j = 0, tot = 0, sum = 0; i < n; i++) {
                int u = cs[i] - 'a';
                cnt[u]++;
                // 如果添加到 cnt 之后为 1，说明字符总数 +1
                if (cnt[u] == 1) tot++;
                // 如果添加到 cnt 之后等于 k，说明该字符从不达标变为达标，达标数量 + 1
                if (cnt[u] == k) sum++;
                // 当区间所包含的字符种类数量 tot 超过了当前限定的数量 p，那么我们要删除掉一些字母，即
                while (tot > p) {
                    int t = cs[j++] - 'a';
                    cnt[t]--;
                    // 如果添加到 cnt 之后为 0，说明字符总数-1
                    if (cnt[t] == 0) tot--;
                    // 如果添加到 cnt 之后等于 k - 1，说明该字符从达标变为不达标，达标数量 - 1
                    if (cnt[t] == k - 1) sum--;
                }
                // 当所有字符都符合要求，更新答案
                if (tot == sum) ans = Math.max(ans, i - j + 1);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：枚举 26 种可能性，每种可能性会扫描一遍数组。复杂度为 $O(n)$
- 空间复杂度：$O(n)$

总结 & 补充

【总结】：

「当确定了窗口内所包含的字符数量时，区间重新具有了二段性质」。这是本题的滑动窗口解法和迄今为止做的滑动窗口题目的最大不同，本题需要手动增加限制，即限制窗口内字符种类。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

【补充】这里解释一下「为什么需要先枚举 26 种可能性」：

首先我们知道「**答案子串的左边界左侧的字符以及右边界右侧的字符一定不会出现在子串中，否则就不会是最优解**」。

但**如果我们只从该性质出发的话，朴素解法应该是使用一个滑动窗口，不断的调整滑动窗口的左右边界，使其满足「左边界左侧的字符以及右边界右侧的字符一定不会出现在窗口中」**，这实际上就是双指针解法，但是如果不先敲定（枚举）出答案所包含的字符数量的话，**这里的双指针是不具有单调性的**。

换句话说，只利用这一性质是没法完成逻辑的。

这时候我们面临的问题是：**性质是正确的，但是还无法直接利用**。

因此我们需要**先利用字符数量有限性（可枚举）作为切入点，使得「答案子串的左边界左侧的字符以及右边界右侧的字符一定不会出现在子串中」这一性质在双指针的实现下具有单调性**。也就是题解说的「让区间重新具有二段性质」。

然后遍历 26 种可能性（答案所包含的字符种类数量），**对每种可能性应用滑动窗口（由上述性质确保正确），可以得到每种可能性的最大值（局部最优），由所有可能性的最大值可以得出答案（全局最优）**。

点评

这道题的突破口分析其实和 1178. 猜字谜 类似。

解决思路：当我们采用常规的分析思路发现无法进行时，**要去关注一下数据范围中「数值小」的值。因为数值小其实是代表了「可枚举」，往往是解题或者降低复杂度的一个重要（甚至是唯一）的突破口。**

**🍥 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 **413. 等差数列划分** ，难度为 **中等**。

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

Tag：「双指针」、「模拟」、「数学」

如果一个数列 至少有三个元素 ，并且任意两个相邻元素之差相同，则称该数列为等差数列。

例如，[1,3,5,7,9]、[7,7,7,7] 和 [3,-1,-5,-9] **都是等差数列。**

给你一个整数数组 nums **，返回数组** nums **中所有为等差数组的 子数组 个数。**

子数组 是数组中的一个连续序列。

示例 1：

```
输入：nums = [1,2,3,4]

输出：3

解释：nums 中有三个子等差数组：[1, 2, 3]、[2, 3, 4] 和 [1,2,3,4] 自身。
```

示例 2：

```
输入：nums = [1]

输出：0
```

提示：

- 1 <= nums.length <= 5000
- -1000 <= nums[i] <= 1000

双指针

具体的，我们可以枚举 i **作为差值为** d **的子数组的左端点，然后通过「双指针」的方式找到当前等差并最长的子数组的右端点** j**，令区间** $[i, j]$ **长度为** len**。**

那么显然，符合条件的子数组的数量为：

$$cnt = \sum_{k=3}^{len} countWithArrayLength(k)$$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

函数 `int countWithArrayLength(int k)` 求的是长度为 k 的子数组的数量。

不难发现，随着入参 k 的逐步减小，函数返回值逐步增大。

因此上述结果 cnt 其实是一个 **首项为 1，末项为 $len-3+1$，公差为 1 的等差数列的求和结果**。直接套用「等差数列求和」公式求解即可。

代码：

```
class Solution {
    public int numberOfArithmeticSlices(int[] nums) {
        int n = nums.length;
        int ans = 0;
        for (int i = 0; i < n - 2; ) {
            int j = i, d = nums[i + 1] - nums[i];
            while (j + 1 < n && nums[j + 1] - nums[j] == d) j++;
            int len = j - i + 1;
            // a1：长度为 len 的子数组数量；an：长度为 3 的子数组数量
            int a1 = 1, an = len - 3 + 1;
            // 符合条件（长度大于等于3）的子数组的数量为「差值数列求和」结果
            int cnt = (a1 + an) * an / 2;
            ans += cnt;
            i = j;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 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 **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

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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 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：

宫水三叶的刷题日记
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 **443. 压缩字符串** ，难度为 **中等**。

Tag：「模拟」、「双指针」、「字符串」

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

给你一个字符数组 chars ，请使用下述算法压缩：

从一个空字符串 s 开始。对于 chars 中的每组 连续重复字符 ：

- 如果这一组长度为 1 ，则将字符追加到 s 中。
- 否则，需要向 s 追加字符，后跟这一组的长度。

压缩后得到的字符串 s 不应该直接返回 ，需要转储到字符数组 chars 中。需要注意的是，如果组长度为 10 或 10 以上，则在 chars 数组中会被拆分为多个字符。

请在** 修改完输入数组后**，返回该数组的新长度。

你必须设计并实现一个只使用常量额外空间的算法来解决此问题。

示例 1：

```
输入：chars = ["a","a","b","b","c","c","c"]

输出：返回 6 ，输入数组的前 6 个字符应该是：["a","2","b","2","c","3"]

解释：
"aa" 被 "a2" 替代。"bb" 被 "b2" 替代。"ccc" 被 "c3" 替代。
```

示例 2：

```
输入：chars = ["a"]

输出：返回 1 ，输入数组的前 1 个字符应该是：["a"]

解释：
没有任何字符串被替代。
```

示例 3：

```
输入：chars = ["a","b","b","b","b","b","b","b","b","b","b","b","b"]

输出：返回 4 ，输入数组的前 4 个字符应该是：["a","b","1","2"]。

解释：
由于字符 "a" 不重复，所以不会被压缩。"bbbbbbbbbbbb" 被 “b12” 替代。
注意每个数字在数组中都有它自己的位置。
```

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

提示：

- 1 <= chars.length <= 2000
- chars[i] 可以是小写英文字母、大写英文字母、数字或符号

双指针

令输入数组 `cs` 长度为 n。

使用两个指针 `i` 和 `j` 分别指向「当前处理到的位置」和「答案待插入的位置」：

1. `i` 指针一直往后处理，每次找到字符相同的连续一段 $[i, idx)$，令长度为 cnt；
2. 将当前字符插入到答案，并让 `j` 指针后移：`cs[j++] = cs[i]`；
3. 检查长度 cnt 是否大于 1，如果大于 1，需要将数字拆分存储。由于简单的实现中，我们只能从个位开始处理 cnt，因此需要使用 `start` 和 `end` 记录下存储数字的部分，再处理完 cnt 后，将 $[start, end)$ 部分进行翻转，并更新 `j` 指针；
4. 更新 `i` 为 `idx`，代表循环处理下一字符。

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 compress(char[] cs) {
        int n = cs.length;
        int i = 0, j = 0;
        while (i < n) {
            int idx = i;
            while (idx < n && cs[idx] == cs[i]) idx++;
            int cnt = idx - i;
            cs[j++] = cs[i];
            if (cnt > 1) {
                int start = j, end = start;
                while (cnt != 0) {
                    cs[end++] = (char)((cnt % 10) + '0');
                    cnt /= 10;
                }
                reverse(cs, start, end - 1);
                j = end;
            }
            i = idx;
        }
        return j;
    }
    void reverse(char[] cs, int start, int end) {
        while (start < end) {
            char t = cs[start];
            cs[start] = cs[end];
            cs[end] = t;
            start++; end--;
        }
    }
}
```

- 时间复杂度：$O(n)$
- 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 **485. 最大连续 1 的个数** ，难度为 **简单**。

Tag：「双指针」

宫水三叶の刷题日记
公众号: 宫水三叶的刷题日记

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.
```

提示：

- 输入的数组只包含 0 和 1 。
- 输入数组的长度是正整数，且不超过 10,000。

双指针解法

使用 `i` 和 `j` 分别代表连续 1 的左右边界。

起始状态 `i == j`，当 `i` 到达第一个 1 的位置时，让 `j` 不断右移直到右边界。

更新 `ans`

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 findMaxConsecutiveOnes(int[] nums) {
        int n = nums.length;
        int ans = 0;
        for (int i = 0, j = 0; i < n; j = i) {
            if (nums[i] == 1) {
                while (j + 1 < n && nums[j + 1] == 1) j++;
                ans = Math.max(ans, j - i + 1);
                i = j + 1;
            } else {
                i++;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 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 **524. 通过删除字母匹配到字典里最长单词** ，难度为 **中等**。

Tag：「双指针」、「贪心」、「排序」

给你一个字符串 s 和一个字符串数组 dictionary 作为字典，找出并返回字典中最长的字符串，该字符串可以通过删除 s 中的某些字符得到。

如果答案不止一个，返回长度最长且字典序最小的字符串。如果答案不存在，则返回空字符串。

示例 1：

```
输入：s = "abpcplea", dictionary = ["ale","apple","monkey","plea"]

输出："apple"
```

示例 2：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```
输入：s = "abpcplea", dictionary = ["a","b","c"]

输出："a"
```

提示：

- 1 <= s.length <= 1000
- 1 <= dictionary.length <= 1000
- 1 <= dictionary[i].length <= 1000
- s 和 dictionary[i] 仅由小写英文字母组成

排序 + 双指针 + 贪心

根据题意，我们需要找到 `dictionary` 中为 `s` 的子序列，且「长度最长（优先级 1）」及「字典序最小（优先级 2）」的字符串。

数据范围全是 1000。

我们可以先对 `dictionary` 根据题意进行自定义排序：

1. 长度不同的字符串，按照字符串长度排倒序；
2. 长度相同的，则按照字典序排升序。

然后我们只需要对 `dictionary` 进行顺序查找，找到的第一个符合条件的字符串即是答案。

具体的，我们可以使用「贪心」思想的「双指针」实现来进行检查：

1. 使用两个指针 `i` 和 `j` 分别代表检查到 `s` 和 `dictionary[x]` 中的哪位字符；
2. 当 `s[i] != dictionary[x][j]`，我们使 `i` 指针右移，直到找到 `s` 中第一位与 `dictionary[x][j]` 对得上的位置，然后当 `i` 和 `j` 同时右移，匹配下一个字符；
3. 重复步骤 2，直到整个 `dictionary[x]` 被匹配完。

> 证明：对于某个字符 `dictionary[x][j]` 而言，选择 `s` 中 **当前** 所能选择的下标最小的位置进行匹配，对于后续所能进行选择方案，会严格覆盖不是选择下标最小的位置，因此结果不会变差。

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 String findLongestWord(String s, List<String> list) {
        Collections.sort(list, (a,b)->{
            if (a.length() != b.length()) return b.length() - a.length();
            return a.compareTo(b);
        });
        int n = s.length();
        for (String ss : list) {
            int m = ss.length();
            int i = 0, j = 0;
            while (i < n && j < m) {
                if (s.charAt(i) == ss.charAt(j)) j++;
                i++;
            }
            if (j == m) return ss;
        }
        return "";
    }
}
```

- 时间复杂度：令 `n` 为 `s` 的长度，`m` 为 `dictionary` 的长度。排序复杂度为 $O(m \log m)$；对 `dictionary` 中的每个字符串进行检查，单个字符串的检查复杂度为 $O(\min(n, dictionary[i])) \approx O(n)$。整体复杂度为 $O(m \log m + m * n)$
- 空间复杂度：$O(\log m)$

**🍭 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 **581. 最短无序连续子数组** ，难度为 **中等**。

Tag：「排序」、「双指针」

给你一个整数数组 nums ，你需要找出一个 连续子数组 ，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

请你找出符合题意的 最短 子数组，并输出它的长度。

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：nums = [2,6,4,8,10,9,15]

输出：5

解释：你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。

示例 2：

输入：nums = [1,2,3,4]

输出：0

示例 3：

输入：nums = [1]

输出：0

提示：

- 1 <= nums.length <= 10^4
- -10^5 <= nums[i] <= 10^5
- 进阶：**你可以设计一个时间复杂度为 O(n) 的解决方案吗？**

双指针 + 排序

最终目的是让整个数组有序，那么我们可以先将数组拷贝一份进行排序，然后使用两个指针 i 和 j 分别找到左右两端第一个不同的地方，那么 $[i, j]$ 这一区间即是答案。

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 findUnsortedSubarray(int[] nums) {
        int n = nums.length;
        int[] arr = nums.clone();
        Arrays.sort(arr);
        int i = 0, j = n - 1;
        while (i <= j && nums[i] == arr[i]) i++;
        while (i <= j && nums[j] == arr[j]) j--;
        return j - i + 1;
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(n)$

双指针 + 线性扫描

另外一个做法是，我们把整个数组分成三段处理。

起始时，先通过双指针 i 和 j 找到左右两次侧满足 **单调递增** 的分割点。

即此时 $[0, i]$ 和 $[j, n)$ 满足升序要求，而中间部分 (i, j) **不确保有序**。

然后我们对中间部分 $[i, j]$ 进行遍历：

- 发现 $nums[x] < nums[i - 1]$：由于对 $[i, j]$ 部分进行排序后 $nums[x]$ 会出现在 $nums[i - 1]$ 后，将不满足整体升序，此时我们需要调整分割点 i 的位置；
- 发现 $nums[x] > nums[j + 1]$：由于对 $[i, j]$ 部分进行排序后 $nums[x]$ 会出现在 $nums[j + 1]$ 前，将不满足整体升序，此时我们需要调整分割点 j 的位置。

> 一些细节：在调整 i 和 j 的时候，我们可能会到达数组边缘，这时候可以建立两个哨兵：数组左边存在一个足够小的数，数组右边存在一个足够大的数。

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 {
    int MIN = -100005, MAX = 100005;
    public int findUnsortedSubarray(int[] nums) {
        int n = nums.length;
        int i = 0, j = n - 1;
        while (i < j && nums[i] <= nums[i + 1]) i++;
        while (i < j && nums[j] >= nums[j - 1]) j--;
        int l = i, r = j;
        int min = nums[i], max = nums[j];
        for (int u = l; u <= r; u++) {
            if (nums[u] < min) {
                while (i >= 0 && nums[i] > nums[u]) i--;
                min = i >= 0 ? nums[i] : MIN;
            }
            if (nums[u] > max) {
                while (j < n && nums[j] < nums[u]) j++;
                max = j < n ? nums[j] : MAX;
            }
        }
        return j == i ? 0 : (j - 1) - (i + 1) + 1;
    }
}
```

- 时间复杂度：$O(n)$
- 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 **611. 有效三角形的个数** ，难度为 **中等**。

Tag：「排序」、「二分」、「双指针」

给定一个包含非负整数的数组，你的任务是统计其中可以组成三角形三条边的三元组个数。

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: [2,2,3,4]

输出: 3

解释:
有效的组合是:
2,3,4 (使用第一个 2)
2,3,4 (使用第二个 2)
2,2,3
```

注意:

1. 数组长度不超过1000。
2. 数组里整数的范围为 [0, 1000]。

基本分析

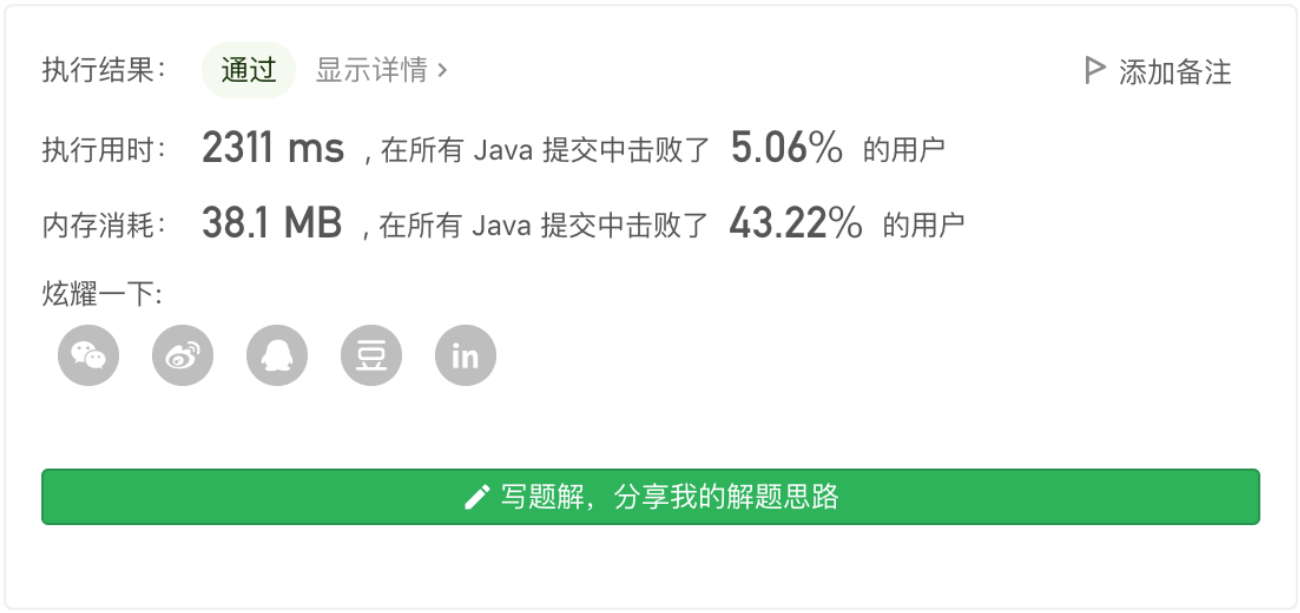
根据题意，是要我们统计所有符合 $nums[k] + nums[j] > nums[i]$ 条件的三元组 (k, j, i) 的个数。

为了防止统计重复的三元组，我们可以先对数组进行排序，然后采取「先枚举较大数；在下标不超过较大数下标范围内，找次大数；在下标不超过次大数下标范围内，找较小数」的策略。

排序 + 暴力枚举

根据「基本分析」，我们可以很容易写出「排序 + 三层循环」的实现。

宫水三叶 の 刷题日记
公众号: 宫水三叶的刷题日记



代码：

```java
class Solution {
    public int triangleNumber(int[] nums) {
        int n = nums.length;
        Arrays.sort(nums);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i - 1; j >= 0; j--) {
                for (int k = j - 1; k >= 0; k--) {
                    if (nums[j] + nums[k] > nums[i]) ans++;
                }
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序时间复杂度为 $O(n \log n)$；三层遍历找所有三元祖的复杂度为 $O(n^3)$。整体复杂度为 $O(n^3)$
- 空间复杂度：$O(\log n)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

排序 + 二分

根据我们以前讲过的的 优化枚举的基本思路，要找符合条件的三元组，其中一个切入点可以是「枚举三元组中的两个值，然后优化找第三数的逻辑」。

我们发现，在数组有序的前提下，**当枚举到较大数下标 i 和次大数下标 j 时，在 $[0, j)$ 范围内找的符合 $nums[k'] + nums[j] > nums[i]$ 条件的 k' 的集合时，以符合条件的最小下标 k 为分割点的数轴上具有「二段性」。**

令 k 为符合条件的最小下标，那么在 $nums[i]$ 和 $nums[j]$ 固定时，$[0, j)$ 范围内：

- 下标大于等于 k 的点集符合条件 $nums[k'] + nums[j] > nums[i]$；
- 下标小于 k 的点集合不符合条件 $nums[k'] + nums[j] > nums[i]$。

因此我们可以通过「二分」找到这个分割点 k，在 $[k, j)$ 范围内即是固定 j 和 i 时，符合条件的 k' 的个数。

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 triangleNumber(int[] nums) {
        int n = nums.length;
        Arrays.sort(nums);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i - 1; j >= 0; j--) {
                int l = 0, r = j - 1;
                while (l < r) {
                    int mid = l + r >> 1;
                    if (nums[mid] + nums[j] > nums[i]) r = mid;
                    else l = mid + 1;
                }
                if (l == r && nums[r] + nums[j] > nums[i]) ans += j - r;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序时间复杂度为 $O(n \log n)$；两层遍历加二分所有符合条件的三元组的复杂度为 $O(n^2 * \log n)$。整体复杂度为 $O(n^2 * \log n)$
- 空间复杂度：$O(\log n)$

排序 + 双指针

更进一步我们发现，当我们在枚举较大数下标 i，并在 $[0, i)$ 范围内逐步减小下标（由于数组有序，也就是逐步减少值）找次大值下标 j 时，符合条件的 k' 必然是从 0 逐步递增（这是由三角不等式 $nums[k] + nums[j] > nums[i]$ 所决定的）。

因此，我们可以枚举较大数下标 i 时，在 $[0, i)$ 范围内通过双指针，以逐步减少下标的方式枚举 j，并在遇到不满足条件的 k 时，增大 k 下标。从而找到所有符合条件三元组的个数。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

代码：

```Java
class Solution {
    public int triangleNumber(int[] nums) {
        int n = nums.length;
        Arrays.sort(nums);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i - 1, k = 0; k < j; j--) {
                while (k < j && nums[k] + nums[j] <= nums[i]) k++;
                ans += j - k;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序时间复杂度为 $O(n \log n)$，双指针找所有符合条件的三元组的复杂度为 $O(n^2)$。整体复杂度为 $O(n^2)$
- 空间复杂度：$O(\log n)$

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

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

题目描述

这是 LeetCode 上的 **633. 平方数之和** ，难度为 **中等**。

Tag：「数学」、「双指针」

给定一个非负整数 c ，你要判断是否存在两个整数 a 和 b，使得 $a^2 + b^2 = c$ 。

示例 1：

```
输入：c = 5

输出：true

解释：1 * 1 + 2 * 2 = 5
```

示例 2：

```
输入：c = 3

输出：false
```

示例 3：

```
输入：c = 4

输出：true
```

示例 4：

```
输入：c = 2

输出：true
```

示例 5：

```
输入：c = 1

输出：true
```

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

提示：

- 0 <= c <= 2^{31} - 1

基本分析

根据等式 $a^2 + b^2 = c$，可得知 `a` 和 `b` 的范围均为 $[0, \sqrt{c}]$。

基于此我们会有以下几种做法。

枚举

我们可以枚举 `a` ，边枚举边检查是否存在 `b` 使得等式成立。

这样做的复杂度为 $O(\sqrt{c})$。

代码：

```
class Solution {
    public boolean judgeSquareSum(int c) {
        int max = (int)Math.sqrt(c);
        for (int a = 0; a <= max; a++) {
            int b = (int)Math.sqrt(c - a * a);
            if (a * a + b * b == c) return true;
        }
        return false;
    }
}
```

- 时间复杂度：$O(\sqrt{c})$
- 空间复杂度：$O(1)$

双指针

由于 `a` 和 `b` 的范围均为 $[0, \sqrt{c}]$，因此我们可以使用「双指针」在 $[0, \sqrt{c}]$ 范围进行扫描：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

- $a^2 + b^2 == c$：找到符合条件的 `a` 和 `b`，返回 $true$
- $a^2 + b^2 < c$：当前值比目标值要小，`a++`
- $a^2 + b^2 > c$：当前值比目标值要大，`b--`

代码：

```
class Solution {
    public boolean judgeSquareSum(int c) {
        int a = 0, b = (int)Math.sqrt(c);
        while (a <= b) {
            int cur = a * a + b * b;
            if (cur == c) {
                return true;
            } else if (cur > c) {
                b--;
            } else {
                a++;
            }
        }
        return false;
    }
}
```

- 时间复杂度：$O(\sqrt{c})$
- 空间复杂度：$O(1)$

费马平方和

费马平方和：奇质数能表示为两个平方数之和的充分必要条件是该质数被 4 除余 1 。

翻译过来就是：**当且仅当一个自然数的质因数分解中，满足 `4k+3` 形式的质数次方数均为偶数时，该自然数才能被表示为两个平方数之和。**

因此我们对 `c` 进行质因数分解，再判断满足 `4k+3` 形式的质因子的次方数是否均为偶数即可。

代码：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```Java
public class Solution {
    public boolean judgeSquareSum(int c) {
        for (int i = 2, cnt = 0; i * i <= c; i++, cnt = 0) {
            while (c % i == 0 && ++cnt > 0) c /= i;
            if (i % 4 == 3 && cnt % 2 != 0) return false;
        }
        return c % 4 != 3;
    }
}
```

- 时间复杂度：$O(\sqrt{c})$
- 空间复杂度：$O(1)$

我猜你问

- 三种解法复杂度都一样，哪个才是最优解呀？

前两套解法是需要「真正掌握」的，而「费马平方和」更多的是作为一种拓展。

你会发现从复杂度上来说，其实「费马平方和」并没有比前两种解法更好，但由于存在对 `c` 除质因数操作，导致「费马平方和」实际表现效果要优于同样复杂度的其他做法。但这仍然不成为我们必须掌握「费马平方和」的理由。

三者从复杂度上来说，都是 $O(\sqrt{c})$ 算法，不存在最不最优的问题。

- 是否有关于「费马平方和」的证明呢？

想要看 莱昂哈德·欧拉 对于「费马平方和」的证明在 这里，我这里直接引用 费马 本人的证明：

> 我确实发现了一个美妙的证明，但这里空白太小写不下。

- 我就是要学「费马平方和」，有没有可读性更高的代码？

有的，在这里。喜欢的话可以考虑背过：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```java
public class Solution {
    public boolean judgeSquareSum(int c) {
        for (int i = 2; i * i <= c; i++) {
            int cnt = 0;
            while (c % i == 0) {
                cnt++;
                c /= i;
            }
            if (i % 4 == 3 && cnt % 2 != 0) return false;
        }
        return c % 4 != 3;
    }
}
```

> **🍭 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 **832. 翻转图像** ，难度为 **简单**。

Tag：「双指针」

给定一个二进制矩阵 A，我们想先水平翻转图像，然后反转图像并返回结果。

水平翻转图片就是将图片的每一行都进行翻转，即逆序。例如，水平翻转 [1, 1, 0] 的结果是 [0, 1, 1]。

反转图片的意思是图片中的 0 全部被 1 替换， 1 全部被 0 替换。例如，反转 [0, 1, 1] 的结果是 [1, 0, 0]。

示例 1：

```
输入：[[1,1,0],[1,0,1],[0,0,0]]
输出：[[1,0,0],[0,1,0],[1,1,1]]
解释：首先翻转每一行: [[0,1,1],[1,0,1],[0,0,0]]；
     然后反转图片: [[1,0,0],[0,1,0],[1,1,1]]
```

示例 2：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```
输入：[[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]]
输出：[[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]
解释：首先翻转每一行: [[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]]；
      然后反转图片: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]
```

提示：

- 1 <= A.length = A[0].length <= 20
- 0 <= A[i][j] <= 1

双指针代码

对于每行而言，我们都需要对其进行「翻转」和「反转」。

这两步可以到放到一遍循环里做：

- 翻转部分：使用双指针进行数字交换
- 反转部分：将数字存储进目标位置前，使用「异或」对 0 1 进行翻转

当前有一些「小细节」需要注意：

1. 题目要求我们对参数图像进行翻转，并返回新图像。因此我们不能对输入直接进行修改，而要先进行拷贝再处理
2. 由于我们将「翻转」和「反转」合成了一步，因此对于「奇数」图像，需要对中间一列进行特殊处理：仅「反转」

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

对于 Java 的基本类型拷贝，有三种方式进行拷贝：

1. System.arraycopy() : 底层的数组拷贝接口，具体实现与操作系统相关，调用的是系统本地方法。需要自己创建好目标数组进行传入，可指定拷贝长度，实现局部拷贝。
2. Arrays.copyOf() : 基于 `System.arraycopy()` 封装的接口，省去了自己目标数组这一步。但无法实现局部拷贝。
3. clone() : Object 的方法。会调用每个数组成员的 clone() 方法进行拷贝。因此对于一维数组而言，可以直接使用 clone() 得到「深拷贝数组」，而对于多维数组而言，得到的是「浅拷贝数组」。

```
class Solution {
    public int[][] flipAndInvertImage(int[][] a) {
        int n = a.length;
        int[][] ans = new int[n][n];
        for (int i = 0; i < n; i++) {
            // ans[i] = a[i].clone();
            // ans[i] = Arrays.copyOf(a[i], n);
            System.arraycopy(a[i], 0, ans[i], 0, n);
            int l = 0, r = n - 1;
            while (l < r) {
                int c = ans[i][r];
                ans[i][r--] = ans[i][l] ^ 1;
                ans[i][l++] = c ^ 1;
            }
            if (n % 2 != 0) ans[i][r] ^= 1;
        }
        return ans;
    }
}
```

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[][] flipAndInvertImage(int[][] a) {
        int n = a.length;
        int[][] ans = new int[n][n];

        // 遍历每一行进行处理
        for (int i = 0; i < n; i++) {
            // 对每一行进行拷贝（共三种方式）
            // ans[i] = a[i].clone();
            // ans[i] = Arrays.copyOf(a[i], n);
            System.arraycopy(a[i], 0, ans[i], 0, n);

            // 使用「双指针」对其进行数组交换，实现「翻转」
            // 并通过「异或」进行 0 1 翻转，实现「反转」
            int l = 0, r = n - 1;
            while (l < r) {
                int c = ans[i][r];
                ans[i][r--] = ans[i][l] ^ 1;
                ans[i][l++] = c ^ 1;
            }

            // 由于「奇数」矩形的中间一列不会进入上述双指针逻辑
            // 需要对其进行单独「反转」
            if (n % 2 != 0) ans[i][r] ^= 1;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：使用了同等大小的空间存储答案。复杂度为 $O(n^2)$

补充

Q: 那么 Arrays.copyOfRange() 与 System.arraycopy() 作用是否等同呢？

A: 不等同。

Arrays.copyOf() 和 Arrays.copyOfRange() 都会内部创建目标数组。前者是直接创建一个和源数组等长的数组，而后者则是根据传参 to 和 from 计算出目标数组长度进行创建。

它们得到的数组都是完整包含了要拷贝内容的，都无法实现目标数组的局部拷贝功能。

宫水三叶 刷题日记 公众号：宫水三叶的刷题日记

例如我要拿到一个长度为 10 的数组，前面 5 个位置的内容来源于「源数组」的拷贝，后面 5 个位置我希望预留给我后面自己做操作，它们都无法满足，只有 System.arraycopy() 可以。

> **🍭 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 **881. 救生艇** ，难度为 **中等**。

Tag：「贪心」、「双指针」

第 `i` 个人的体重为 `people[i]`，每艘船可以承载的最大重量为 `limit`。

每艘船最多可同时载两人，但条件是这些人的重量之和最多为 `limit`。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1：

```
输入：people = [1,2], limit = 3

输出：1

解释：1 艘船载 (1, 2)
```

示例 2：

```
输入：people = [3,2,2,1], limit = 3

输出：3

解释：3 艘船分别载 (1, 2), (2) 和 (3)
```

示例 3：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```
输入：people = [3,5,3,4], limit = 5

输出：4

解释：4 艘船分别载 (3), (3), (4), (5)
```

提示：

- 1 <= people.length <= 50000
- 1 <= people[i] <= limit <= 30000

贪心

一个直观的想法是：由于一个船要么载两人要么载一人，在人数给定的情况下，为了让使用的总船数最小，要当尽可能让更多船载两人，即尽可能多的构造出数量之和不超过 $limit$ 的二元组。

先对 $people$ 进行排序，然后使用两个指针 `l` 和 `r` 分别从首尾开始进行匹配：

- 如果 $people[l] + people[r] <= limit$，说明两者可以同船，此时船的数量加一，两个指针分别往中间靠拢；
- 如果 $people[l] + people[r] > limit$，说明不能成组，由于题目确保人的重量不会超过 $limit$，此时让 $people[r]$ 独立成船，船的数量加一，`r` 指针左移。

我们猜想这样「最重匹配最轻、次重匹配次轻」的做法能使双人船的重量之和尽可能平均，从而使双人船的数量最大化。

接下来，我们使用「归纳法」证明猜想的正确性。

假设最优成船组合中二元组的数量为 $c1$，我们贪心做法的二元组数量为 $c2$。

最终答案 = 符合条件的二元组的数量 + 剩余人数数量，而在符合条件的二元组数量固定的情况下，剩余人数也固定。因此我们只需要证明 $c1 = c2$ 即可。

通常使用「归纳法」进行证明，都会先从边界入手。

当我们处理最重的人 $people[r]$（此时 r 为原始右边界 $n - 1$）时：

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

- 假设其与 $people[l]$（此时 l 为原始左边界 0）之和超过 $limit$，说明 $people[r]$ 与数组任一成员组合都会超过 $limit$，即无论在最优组合还是贪心组合中，$people[r]$ 都会独立成船；
- 假设 $people[r] + people[l] <= limit$，说明数组中存在至少一个成员能够与 $people[l]$ 成船：
    - 假设在最优组合中 $people[l]$ 独立成船，此时如果将贪心组合 $(people[l], people[r])$ 中的 $people[l]$ 拆分出来独立成船，贪心二元组数量 $c2$ 必然不会变大（可能还会变差），即将「贪心解」调整成「最优解」结果不会变好；
    - 假设在最优组合中，$people[l]$ 不是独立成船，又因此当前 r 处于原始右边界，因此与 $people[l]$ 成组的成员 $people[x]$ 必然满足 $people[x] <= people[r]$。
      此时我们将 $people[x]$ 和 $people[r]$ 位置进行交换（将贪心组合调整成最优组合），此时带来的影响包括：
        - 与 $people[l]$ 成组的对象从 $people[r]$ 变为 $people[x]$，但因为 $people[x] <= people[r]$，即有 $people[l] + people[x] <= people[l] + people[r] <= limit$，仍为合法二元组，消耗船的数量为 1；
        - 原本位置 x 的值从 $people[x]$ 变大为 $people[r]$，如果调整后的值能组成二元组，那么原本更小的值也能组成二元组，结果没有变化；如果调整后不能成为组成二元组，那么结果可能会因此变差。

      综上，将 $people[x]$ 和 $people[r]$ 位置进行交换（将贪心组合调整成最优组合），贪心二元组数量 $c2$ 不会变大，即将「贪心解」调整成「最优解」结果不会变好。

对于边界情况，我们证明了从「贪心解」调整为「最优解」不会使得结果更好，因此可以保留当前的贪心决策，然后将问题规模缩减为 $n-1$ 或者 $n-2$，同时数列仍然满足升序特性，即归纳分析所依赖的结构没有发生改变，可以将上述的推理分析推广到每一个决策的回合（新边界）中。

至此，我们证明了将「贪心解」调整为「最优解」结果不会变好，即贪心解是最优解之一。

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 numRescueBoats(int[] people, int limit) {
        Arrays.sort(people);
        int n = people.length;
        int l = 0, r = n - 1;
        int ans = 0;
        while (l <= r) {
            if (people[l] + people[r] <= limit) l++;
            r--;
            ans++;
        }
        return ans;
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$；双指针统计答案复杂度为 $O(n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(\log n)$

**🍭 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 **930. 和相同的二元子数组** ，难度为 **中等**。

Tag：「前缀和」、「哈希表」、「双指针」

给你一个二元数组 nums ，和一个整数 goal ，请你统计并返回有多少个和为 goal 的 非空 子数组。

子数组 是数组的一段连续部分。

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：nums = [1,0,1,0,1], goal = 2

输出：4

解释：
如下面黑体所示，有 4 个满足题目要求的子数组：
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
```

示例 2：

```
输入：nums = [0,0,0,0,0], goal = 0

输出：15
```

提示：

- 1 <= nums.length <= 3 * 10^4
- nums[i] 不是 0 就是 1
- 0 <= goal <= nums.length

前缀和 + 哈希表

一个简单的想法是，先计算 $nums$ 的前缀和数组 sum，然后从前往后扫描 $nums$，对于右端点 r，通过前缀和数组可以在 $O(1)$ 复杂度内求得 $[0, r]$ 连续一段的和，根据容斥原理，我们还需要求得某个左端点 l，使得 $[0, r]$ 减去 $[0, l-1]$ 和为 t，即满足 $sum[r] - sum[l-1] = t$，这时候利用哈希表记录扫描过的 $sum[i]$ 的出现次数，可以实现 $O(1)$ 复杂度内求得满足要求的左端点个数。

该方法适用于 $nums[i]$ 值不固定为 0 和 1 的其他情况。

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 numSubarraysWithSum(int[] nums, int t) {
        int n = nums.length;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] + nums[i - 1];
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int r = sum[i + 1], l = r - t;
            ans += map.getOrDefault(l, 0);
            map.put(r, map.getOrDefault(r, 0) + 1);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

双指针

另外一个通用性稍差一点的做法则是利用 $nums[i]$ 没有负权值。

$nums[i]$ 没有负权值意味着前缀和数组必然具有（非严格）单调递增特性。

不难证明，在给定 t 的情况下，当我们右端点 r 往右移动时，满足条件的左端点 l 必然往右移动。

实现上，我们可以使用两个左端点 $l1$ 和 $l2$，代表在给定右端点 r 的前提下满足要求的左端点集合，同时使用 $s1$ 和 $s2$ 分别代表两个端点到 r 这一段的和。

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 numSubarraysWithSum(int[] nums, int t) {
        int n = nums.length;
        int ans = 0;
        for (int r = 0, l1 = 0, l2 = 0, s1 = 0, s2 = 0; r < n; r++) {
            s1 += nums[r];
            s2 += nums[r];
            while (l1 <= r && s1 > t) s1 -= nums[l1++];
            while (l2 <= r && s2 >= t) s2 -= nums[l2++];
            ans += l2 - l1;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 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。）

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]
```

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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]
```

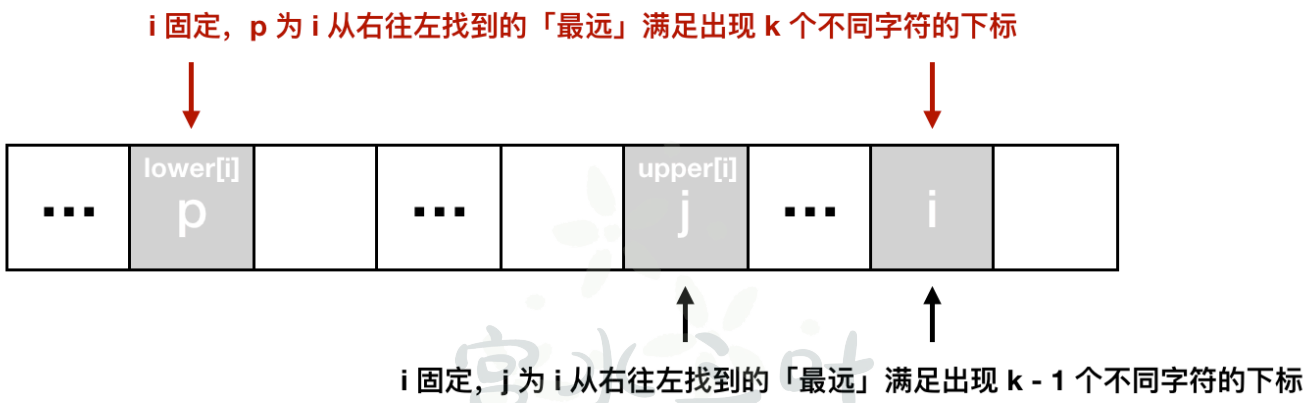
提示：

- 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**



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`。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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：

```
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**

宫水三叶 刷题日记 公众号：宫水三叶的刷题日记

`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。

> **🍭 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

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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。

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：$O(k)$

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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。

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 **1221. 分割平衡字符串** ，难度为 **简单**。

Tag：「贪心」、「双指针」

在一个 平衡字符串 中，‘L’ 和 ‘R’ 字符的数量是相同的。

给你一个平衡字符串 s，请你将它分割成尽可能多的平衡字符串。

注意：分割得到的每个字符串都必须是平衡字符串。

返回可以通过分割得到的平衡字符串的 最大数量 。

示例 1：

```
输入：s = "RLRRLLRLRL"

输出：4

解释：s 可以分割为 "RL"、"RRLL"、"RL"、"RL" ，每个子字符串中都包含相同数量的 'L' 和 'R' 。
```

示例 2：

```
输入：s = "RLLLLRRRLR"

输出：3

解释：s 可以分割为 "RL"、"LLLRRR"、"LR" ，每个子字符串中都包含相同数量的 'L' 和 'R' 。
```

示例 3：

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

```
输入：s = "LLLLRRRR"

输出：1

解释：s 只能保持原样 "LLLLRRRR".
```

示例 4：

```
输入：s = "RLRRRLLRLL"

输出：2

解释：s 可以分割为 "RL"、"RRRLLRLL" ，每个子字符串中都包含相同数量的 'L' 和 'R' 。
```

提示：

- 1 <= s.length <= 1000
- s[i] = ‘L’ 或 ‘R’
- s 是一个 平衡 字符串

基本分析

题目确保了 `s` 为一个平衡字符串，即必然能分割成若干个 `LR` 子串。

一个合法的 `LR` 子串满足 `L` 字符和 `R` 字符数量相等，常规检查一个字符串是否为合格的 `LR` 子串可以使用 $O(n)$ 的遍历方式，可以使用记录前缀信息的数据结构，而对于成对结构的元素统计，更好的方式是转换为数学判定，使用 `1` 来代指 `L` 得分，使用 `-1` 来代指 `R` 得分。

那么一个子串为合格 `LR` 子串的充要条件为 **整个 `LR` 子串的总得分为** 0。

这种方式最早应该在 (题解) 301. 删除无效的括号 详细讲过，可延伸到任意的成对结构元素统计题目里去。

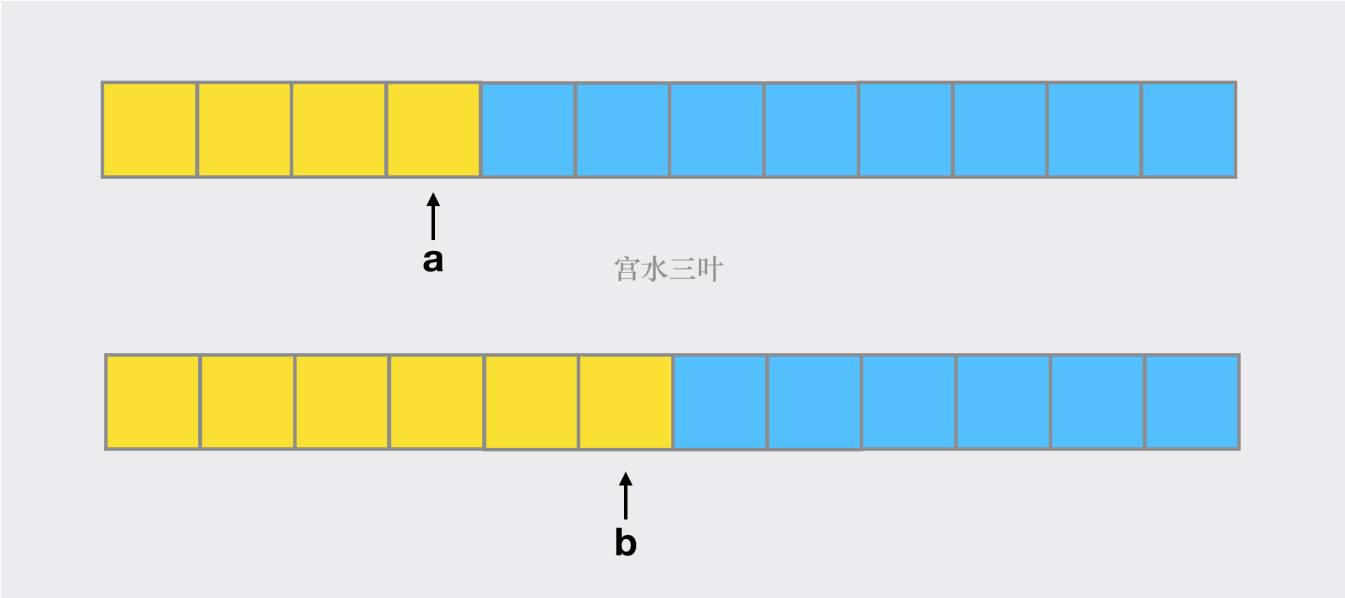
贪心

回到本题，题目要求分割的 `LR` 子串尽可能多，直观上应该是尽可能让每个分割串尽可能短。

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「归纳法」来证明该猜想的正确性。

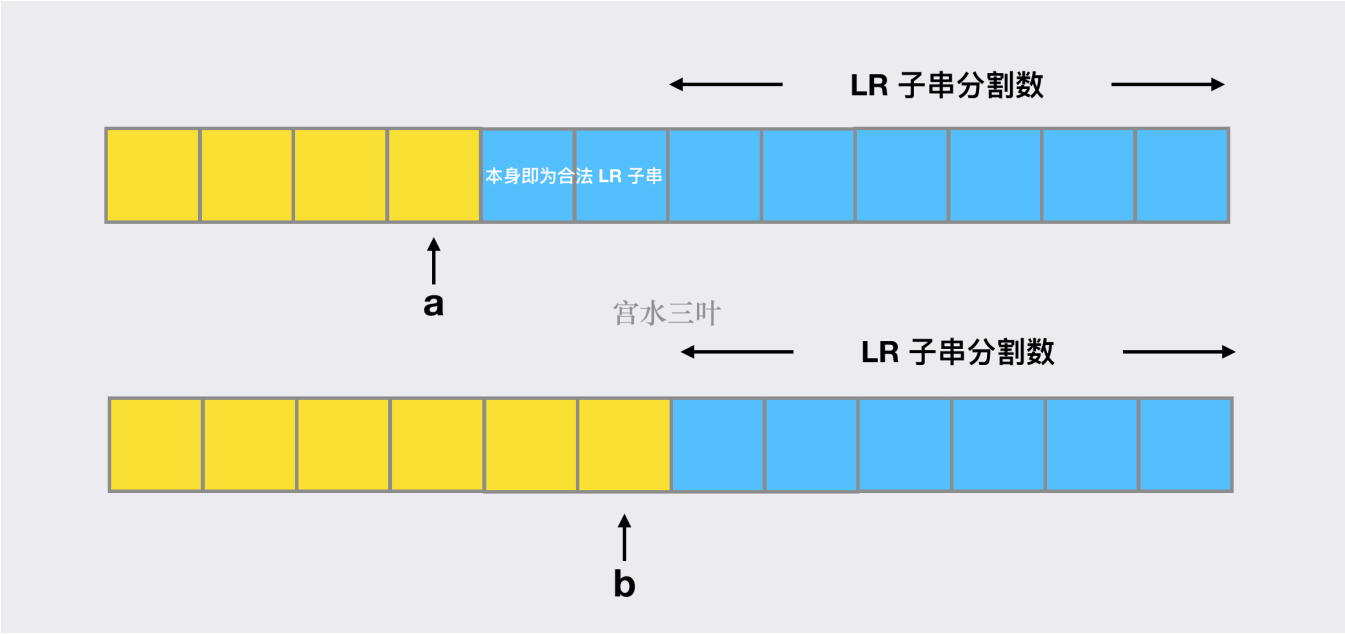
首先题目数据保证给定的 `s` 本身是合法的 `LR` 子串，假设从 $[0...a]$ 可以从 `s` 中分割出 **长度最小** 的 `LR` 子串，而从 $[0...b]$ 能够分割出 **长度更大** 的 `LR` 子串（即 `a <= b`）。



我们来证明起始时（第一次分割）「将从 `b` 分割点将 `s` 断开」调整为「从 `a` 分割点将 `s` 断开」结果不会变差：

1. 从 `b` 点首次分割调整为从 `a` 点首次分割，两种分割形式分割点两端仍为合法 `LR` 子串，因此不会从“有解”变成“无解”；
2. 从 `b` 分割后的剩余部分长度小于从 `a` 分割后的剩余部分，同时由 `b` 分割后的剩余部分会被由 `a` 分割后的剩余部分所严格覆盖，因此「对 `a` 分割的剩余部分再分割所得的子串数量」**至少** 与「从 `b` 点分割的剩余部分再分割所得的子串数量」相等（不会变少）。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记



至此，我们证明了对于首次分割，将任意合格分割点调整为最小分割点，结果不会变得更差（当 `a < b` 时还会更好）。

同时，由于首次分割后的剩余部分仍为合格的 `LR` 子串，因此归纳分析所依赖的结构没有发生改变，可以将上述的推理分析推广到每一个决策的回合（新边界）中。

至此，我们证明了只要每一次都从最小分割点进行分割，就可以得到最优解。

代码：

```
class Solution {
    public int balancedStringSplit(String s) {
        char[] cs = s.toCharArray();
        int n = cs.length;
        int ans = 0;
        for (int i = 0; i < n; ) {
            int j = i + 1, score = cs[i] == 'L' ? 1 : -1;
            while (j < n && score != 0) score += cs[j++] == 'L' ? 1 : -1;
            i = j;
            ans++;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：调用 `toCharArray` 会拷贝新数组进行返回（为遵循 `String` 的不可

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

变原则），因此使用 `toCharArray` 复杂度为 $O(n)$，使用 `charAt` 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 **1743. 从相邻元素对还原数组** ，难度为 **中等**。

Tag：「哈希表」、「双指针」、「模拟」

存在一个由 n 个不同元素组成的整数数组 nums ，但你已经记不清具体内容。好在你还记得 nums 中的每一对相邻元素。

给你一个二维整数数组 adjacentPairs ，大小为 n - 1 ，其中每个 adjacentPairs[i] = [ui, vi] 表示元素 ui 和 vi 在 nums 中相邻。

题目数据保证所有由元素 nums[i] 和 nums[i+1] 组成的相邻元素对都存在于 adjacentPairs 中，存在形式可能是 [nums[i], nums[i+1]] ，也可能是 [nums[i+1], nums[i]] 。这些相邻元素对可以 按任意顺序 出现。

返回 原始数组 nums 。如果存在多种解答，返回 其中任意一个 即可。

示例 1：

```
输入：adjacentPairs = [[2,1],[3,4],[3,2]]

输出：[1,2,3,4]

解释：数组的所有相邻元素对都在 adjacentPairs 中。
特别要注意的是，adjacentPairs[i] 只表示两个元素相邻，并不保证其 左-右 顺序。
```

示例 2：

宫水三叶 の 刷题日记
公众号：宫水三叶的刷题日记

```
输入：adjacentPairs = [[4,-2],[1,4],[-3,1]]

输出：[-2,4,1,-3]

解释：数组中可能存在负数。
另一种解答是 [-3,1,4,-2] ，也会被视作正确答案。
```

示例 3：

```
输入：adjacentPairs = [[100000,-100000]]

输出：[100000,-100000]
```

提示：

- nums.length == n
- adjacentPairs.length == n - 1
- adjacentPairs[i].length == 2
- 2 <= n <= 10^5
- -10^5 <= nums[i], ui, vi <= 10^5
- 题目数据保证存在一些以 adjacentPairs 作为元素对的数组

单向构造（哈希表计数）

根据题意，由于所有的相邻关系都会出现在 $nums$ 中，假设其中一个合法数组为 ans，长度为 n。

那么显然 $ans[0]$ 和 $ans[n-1]$ 在 $nums$ 中只存在一对相邻关系，而其他 $ans[i]$ 则存在两对相邻关系。

因此我们可以使用「哈希表」对 $nums$ 中出现的数值进行计数，找到“出现一次”的数值作为 ans 数值的首位，然后根据给定的相邻关系进行「单向构造」，为了方便找到某个数其相邻的数是哪些，我们还需要再开一个「哈希表」记录相邻关系。

宫水三叶の刷题日记
公众号：宫水三叶的刷题日记

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[] restoreArray(int[][] aps) {
        int m = aps.length, n = m + 1;
        Map<Integer, Integer> cnts = new HashMap<>();
        Map<Integer, List<Integer>> map = new HashMap<>();
        for (int[] ap : aps) {
            int a = ap[0], b = ap[1];
            cnts.put(a, cnts.getOrDefault(a, 0) + 1);
            cnts.put(b, cnts.getOrDefault(b, 0) + 1);
            List<Integer> alist = map.getOrDefault(a, new ArrayList<>());
            alist.add(b);
            map.put(a, alist);
            List<Integer> blist = map.getOrDefault(b, new ArrayList<>());
            blist.add(a);
            map.put(b, blist);
        }
        int start = -1;
        for (int i : cnts.keySet()) {
            if (cnts.get(i) == 1) {
                start = i;
                break;
            }
        }
        int[] ans = new int[n];
        ans[0] = start;
        ans[1] = map.get(start).get(0);
        for (int i = 2; i < n; i++) {
            int x = ans[i - 1];
            List<Integer> list = map.get(x);
            for (int j : list) {
                if (j != ans[i - 2]) ans[i] = j;
            }
        }
        return ans;
    }
}
```

- 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

双向构造（双指针）

在解法一中，我们通过「哈希表」计数得到 ans 首位元素作为起点，进行「单向构造」。

那么是否存在使用任意数值作为起点进行的双向构造呢？

答案是显然的，我们可以利用 ans 的长度为 $2 <= n <= 10^5$，构造一个长度 10^6 的数组 q（这里可以使用 `static` 进行加速，让多个测试用例共享一个大数组）。

> 这里 q 数组不一定要开成 $1e6$ 大小，只要我们 q 大小大于 ans 的两倍，就不会存在越界问题。

从 q 数组的 **中间位置** 开始，先随便将其中一个元素添加到中间位置，使用「双指针」分别往「两边拓展」（`l` 和 `r` 分别指向左右待插入的位置）。

当 `l` 指针和 `r` 指针之间已经有 n 个数值，说明整个 ans 构造完成，我们将 $[l+1, r-1]$ 范围内的数值输出作为答案即可。

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 {
    static int N = (int)1e6+10;
    static int[] q = new int[N];
    public int[] restoreArray(int[][] aps) {
        int m = aps.length, n = m + 1;
        Map<Integer, List<Integer>> map = new HashMap<>();
        for (int[] ap : aps) {
            int a = ap[0], b = ap[1];
            List<Integer> alist =  map.getOrDefault(a, new ArrayList<>());
            alist.add(b);
            map.put(a, alist);
            List<Integer> blist = map.getOrDefault(b, new ArrayList<>());
            blist.add(a);
            map.put(b, blist);
        }
        int l = N / 2, r = l + 1;
        int std = aps[0][0];
        List<Integer> list = map.get(std);
        q[l--] = std;
        q[r++] = list.get(0);
        if (list.size() > 1) q[l--] = list.get(1);
        while ((r - 1) - (l + 1) + 1 < n) {
            List<Integer> alist = map.get(q[l + 1]);
            int j = l;
            for (int i : alist) {
                if (i != q[l + 2]) q[j--] = i;
            }
            l = j;

            List<Integer> blist = map.get(q[r - 1]);
            j = r;
            for (int i : blist) {
                if (i != q[r - 2]) q[j++] = i;
            }
            r = j;
        }
        int[] ans = new int[n];
        for (int i = l + 1, idx = 0; idx < n; i++, idx++) {
            ans[idx] = q[i];
        }
        return ans;
    }
}
```

- 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

> **🍥 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 **1764. 通过连接另一个数组的子数组得到一个数组** ，难度为 **中等**。

Tag：「双指针」

给你一个长度为 n 的二维整数数组 groups ，同时给你一个整数数组 nums 。

你是否可以从 nums 中选出 n 个 不相交 的子数组，使得第 i 个子数组与 groups[i] （下标从 0 开始）完全相同，且如果 i > 0 ，那么第 (i-1) 个子数组在 nums 中出现的位置在第 i 个子数组前面。（也就是说，这些子数组在 nums 中出现的顺序需要与 groups 顺序相同）

如果你可以找出这样的 n 个子数组，请你返回 true ，否则返回 false 。

如果不存在下标为 k 的元素 nums[k] 属于不止一个子数组，就称这些子数组是 不相交 的。子数组指的是原数组中连续元素组成的一个序列。

示例 1：

```
输入：groups = [[1,-1,-1],[3,-2,0]], nums = [1,-1,0,1,-1,-1,3,-2,0]
输出：true
解释：你可以分别在 nums 中选出第 0 个子数组 [1,-1,0,1,-1,-1,3,-2,0] 和第 1 个子数组 [1,-1,0,1,-1,-1,3,-2,0] 。
这两个子数组是不相交的，因为它们没有任何共同的元素。
```

示例 2：

```
输入：groups = [[10,-2],[1,2,3,4]], nums = [1,2,3,4,10,-2]
输出：false
解释：选择子数组 [1,2,3,4,10,-2] 和 [1,2,3,4,10,-2] 是不正确的，因为它们出现的顺序与 groups 中顺序不同。
[10,-2] 必须出现在 [1,2,3,4] 之前。
```

示例 3：

```
输入：groups = [[1,2,3],[3,4]], nums = [7,7,1,2,3,4,7,7]
输出：false
解释：选择子数组 [7,7,1,2,3,4,7,7] 和 [7,7,1,2,3,4,7,7] 是不正确的，因为它们不是不相交子数组。
它们有一个共同的元素 nums[4] （下标从 0 开始）。
```

宫水三叶的刷题日记
公众号：宫水三叶的刷题日记

提示：

- groups.length == n
- 1 <= n <= 10^3
- 1 <= groups[i].length, sum(groups[i].length) <= 10^3
- 1 <= nums.length <= 10^3
- -10^7 <= groups[i][j], nums[k] <= 10^7

朴素解法

本质上是道模拟题。

使用 `i` 指针代表 `nums` 当前枚举到的位置；`j` 代表 `groups` 中枚举到哪个数组。

`cnt` 代表匹配的数组个数。

- 当 `i` 开头的连续一段和 `groups[j]` 匹配：`j` 指针右移一位（代表匹配下一个数组），`i` 指针右移 `groups[j].length` 长度。
- 当 `i` 开头的连续一段和 `groups[j]` 不匹配：`i` 右移一位。

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 canChoose(int[][] gs, int[] nums) {
        int n = nums.length, m = gs.length;
        int cnt = 0;
        for (int i = 0, j = 0; i < n && j < m;) {
            if (check(gs[j], nums, i)) {
                i += gs[j++].length;
                cnt++;
            } else {
                i++;
            }
        }
        return cnt == m;
    }
    boolean check(int[] g, int[] nums, int i) {
        int j = 0;
        for (;j < g.length && i < nums.length; j++, i++) {
            if (g[j] != nums[i]) return false;
        }
        return j == g.length;
    }
}
```

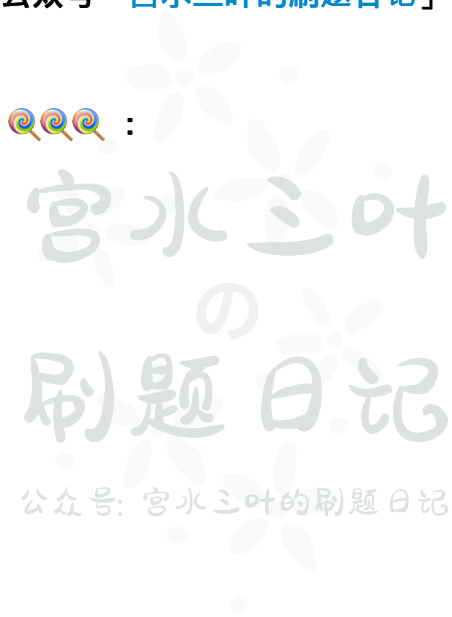
- 时间复杂度：$O(n * m)$
- 空间复杂度：$O(1)$

> **🍭 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「双指针」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：



"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。