宫水三叶的刷题日记

排序

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

> **🍭 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「排序」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「排序」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「排序」；
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 **41. 缺失的第一个正数** ，难度为 **困难**。

Tag : 「桶排序」

给你一个未排序的整数数组 nums ，请你找出其中没有出现的最小的正整数。

进阶：你可以实现时间复杂度为 O(n) 并且只使用常数级别额外空间的解决方案吗？

示例 1：

```
输入：nums = [1,2,0]

输出：3
```

示例 2：

```
输入：nums = [3,4,-1,1]

输出：2
```

示例 3：

```
输入：nums = [7,8,9,11,12]

输出：1
```

提示：

- 0 <= nums.length <= 300
- -2^{31} <= nums[i] <= 2^{31} - 1

桶排序

令数组长度为 `n`，那么答案必然在 `[1, n + 1]` 范围内。

因此我们可以使用「桶排序」的思路，将每个数放在其应该出现的位置上。

基本思路为：

1. 按照桶排序思路进行预处理：保证 1 出现在 `nums[0]` 的位置上，2 出现在 `nums[1]` 的位置上，...，`n` 出现在 `nums[n - 1]` 的位置上。不在 `[1, n]` 范围内的数不用动。

例如样例中 `[3,4,-1,1]` 将会被预处理成 `[1,-1,3,4]`。

2. 遍历 `nums`，找到第一个不在应在位置上的 `[1, n]` 的数。如果没有找到，说明数据连续，答案为 `n + 1`。

例如样例预处理后的数组 `[1,-1,3,4]` 中第一个 `nums[i] != i + 1` 的是数字 2（i = 1）。

代码：

```java
class Solution {
    public int firstMissingPositive(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            while (nums[i] >= 1 && nums[i] <= n && nums[i] != i + 1 && nums[i] != nums[num
                swap(nums, i, nums[i] - 1);
            }
        }
        for (int i = 0; i < n; i++) {
            if (nums[i] != i + 1) return i + 1;
        }
        return n + 1;
    }
    void swap(int[] nums, int a, int b) {
        int c = nums[a];
        nums[a] = nums[b];
        nums[b] = c;
    }
}
```

- 时间复杂度：每个数字应该被挪动的数都会被一次性移动到目标位置。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

实战技巧

还记得最早我在 4. 寻找两个正序数组的中位数 跟你说过，对于一些从文字上限制我们的题目，我们应该如何分析能否采用别的做法来 AC。

这对于比赛和机试，这种要求我们尽快 AC 的场景来说，尤为重要。

总的来说，你可以根据**直观解法的时空复杂度**、**文字限制的时空复杂度**和**数据范围**等几个方面来判断。

对于本题而言，我们可以很容易想到先排序，再遍历的做法：

排序的复杂度为 $O(n \log n)$；找答案的过程需要枚举 `[1, n]`，然后通过「二分」找当前的枚举值是否在排序数据中，复杂度为 $O(n \log n)$。因此整体复杂度为 $O(n \log n)$。

而文字要求我们使用 $O(n)$ 复杂度的解法。这时候我们基本上可以不看数据范围就可以确定

$O(n \log n)$ 能做。

因为 $O(n \log n)$ 和 $O(n)$ 并没有差多少，哪怕数据范围出到极限 10^7。$log10^7$ = 23，数值非常小，$O(n \log n)$ 可以等效看做一个常数计算为 23 的 $O(n)$ 解法。何况本题的数据范围只有 300，可以说怎么做都行。

$O(n \log n)$ 解法代码：

```
class Solution {
    public int firstMissingPositive(int[] nums) {
        Arrays.sort(nums);
        int n = nums.length;
        for (int i = 1; i <= n; i++) {
            if (Arrays.binarySearch(nums, i) < 0) return i;
        }
        return n + 1;
    }
}
```

你会发现，在 LeetCode 上 $O(n \log n)$ 解法和 $O(n)$ 解法的执行用时没有差别。

> **🍭 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
```

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。

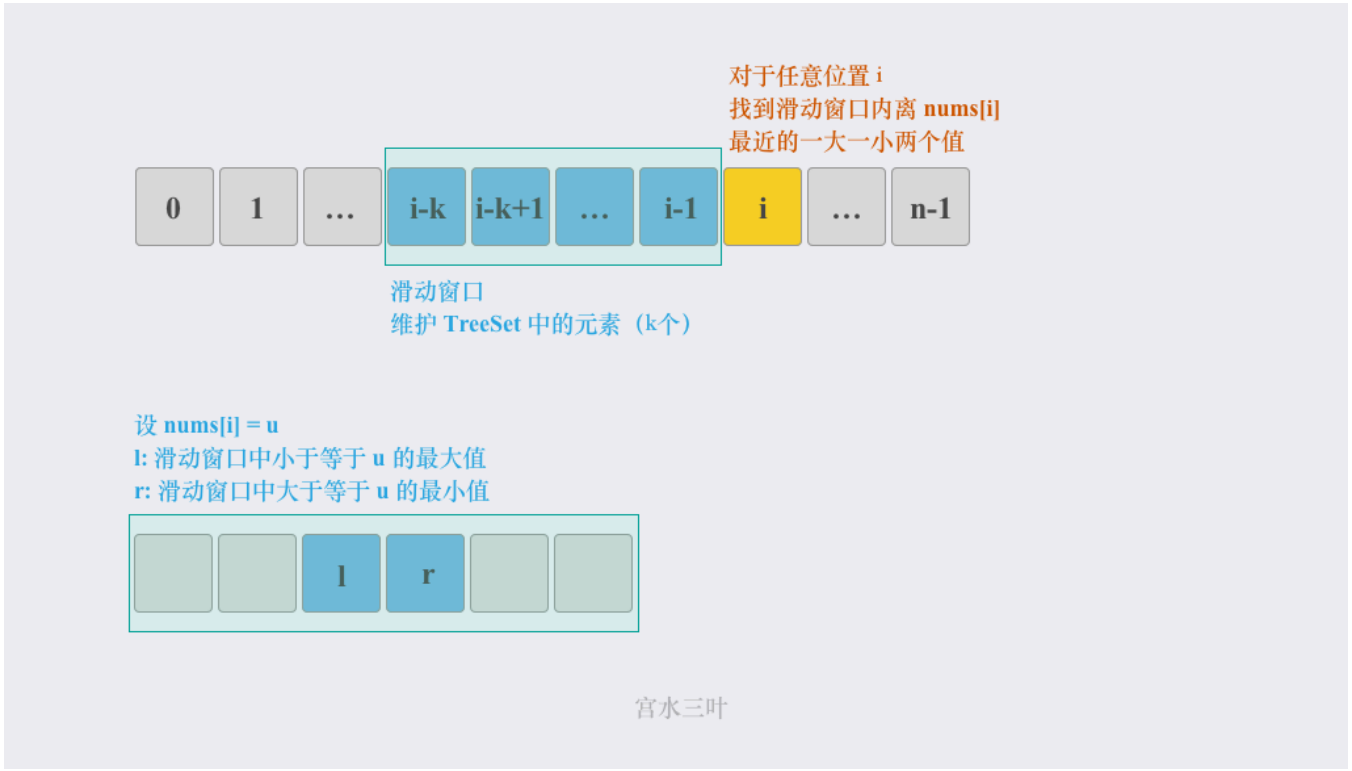
例如 AVL，能够让我们在最坏为 $O(\log k)$ 的复杂度内取得到最接近 `u` 的值是多少，但本题除了「查询」以外，还涉及频繁的「插入/删除」操作（随着我们遍历 `nums` 的元素，滑动窗口不断右移，我们需要不断的往「有序集合」中删除和添加元素）。

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：

```
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：

```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`。

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 **414. 第三大的数**，难度为 **简单**。

Tag：「排序」、「数组」、「模拟」

给你一个非空数组，返回此数组中 **第三大的数**。如果不存在，则返回数组中最大的数。

示例 1：

```
输入：[3, 2, 1]

输出：1

解释：第三大的数是 1 。
```

示例 2：

```
输入：[1, 2]

输出：2

解释：第三大的数不存在, 所以返回最大的数 2 。
```

示例 3：

```
输入：[2, 2, 3, 1]

输出：1

解释：注意，要求返回第三大的数，是指在所有不同数字中排第三大的数。
此例中存在两个值为 2 的数，它们都排第二。在所有不同数字中排第三大的数为 1 。
```

提示：

- 1 <= nums.length <= 10^4
- -2^{31} <= nums[i] <= $2^{31} - 1$

进阶：你能设计一个时间复杂度 O(n) 的解决方案吗？

Set 去重 + 排序

题目要求返回含重复元素的数组 $nums$ 中的第三大数。

一个朴素的做法是，先使用 `Set` 对重复元素进行去重，然后对去重后的元素进行排序，并返回第三大的元素。

代码：

```
class Solution {
    public int thirdMax(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for (int x : nums) set.add(x);
        List<Integer> list = new ArrayList<>(set);
        Collections.sort(list);
        return list.size() < 3 ? list.get(list.size() - 1) : list.get(list.size() - 3);
    }
}
```

- 时间复杂度：使用 `Set` 去重的复杂度为 $O(n)$；排序复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

有限变量 + 遍历

经典的找数组次大值的做法是使用两个变量 `a` 和 `b` 分别存储遍历过程中的最大值和次大值。

假设当前遍历到的元素为 x，当满足如下条件时，考虑更新 `a` 或者 `b`：

1. 当 $x > a$ 时，说明最大值被更新，同时原来的最大值沦为次大值。即有 $b = a; a = x;$
2. 在条件 1 不满足，且有$x > b$ 时，此时可以根据是否有「严格次大值」的要求，而决定是否要增加 $x < a$ 的条件：
    - 不要求为「严格次大值」：直接使用 x 来更新 `b`，即有 $b = x$；
    - 当要求为「严格次大值」：此时需要满足 $x < a$ 的条件，才能更新 `b`。

回到本题，同理我们可以使用 `a`、`b` 和 `c` 三个变量来代指「最大值」、「严格次大值」和「严格第三大值」。

从前往后遍历 $nums$，假设当前元素为 x，对是否更新三者进行分情况讨论（判断优先级从上往下）：

1. $x > a$，说明最大值被更新，将原本的「最大值」和「次大值」往后顺延为「次大值」和「第三大值」，并用 x 更新 `a`；

2. $x < a$ 且 $x > b$，说明次大值被更新，将原本的「次大值」往后顺延为「第三大值」，并用 x 更新 `b`；
3. $x < b$ 且 $x > c$，说明第三大值被更新，使用 x 更新 `c`。

起始时，我们希望使用一个足够小的数来初始化 `a`、`b` 和 `c`，但由于 $num[i]$ 的范围为 $[-2^{31}, 2^{31} - 1]$，因此需要使用 `long` 来进行代替。

返回时，通过判断第三大值是否为初始化时的负无穷，来得知是否存在第三大值。

代码：

```
class Solution {
    long INF = (long)-1e18;
    public int thirdMax(int[] nums) {
        long a = INF, b = INF, c = INF;
        for (int x : nums) {
            if (x > a) {
                c = b; b = a; a = x;
            } else if (x < a && x > b) {
                c = b; b = x;
            } else if (x < b && x > c) {
                c = x;
            }
        }
        return c != INF ? (int)c : (int)a;
    }
}
```

- 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 **448. 找到所有数组中消失的数字** ，难度为 **简单**。

Tag：「排序」

给定一个范围在 1 ≤ a[i] ≤ n (n = 数组大小) 的 整型数组，数组中的元素一些出现了两次，另一

些只出现一次。

找到所有在 [1, n] 范围之间没有出现在数组中的数字。

您能在不使用额外空间且时间复杂度为O(n)的情况下完成这个任务吗? 你可以假定返回的数组不算在额外空间内。

示例:

```
输入:
[4,3,2,7,8,2,3,1]

输出:
[5,6]
```

桶排序

题目规定了 `1 ≤ a[i] ≤ n` ，因此我们可以使用「桶排序」的思路，将每个数放在其应该出现的位置上。

基本思路为：

按照桶排序思路进行预处理：保证 1 出现在 nums[0] 的位置上，2 出现在 nums[1] 的位置上，...，n 出现在 nums[n - 1] 的位置上。不在 [1, n] 范围内的数不用动。

例如样例中 [4,3,2,7,8,2,3,1] 将会被预处理成 [1,2,3,4,3,2,7,8]。

遍历 nums，将不符合 `nums[i] != i + 1` 的数字加入结果集 ~

```java
class Solution {
    public List<Integer> findDisappearedNumbers(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            while (nums[i] != i + 1 && nums[nums[i] - 1] != nums[i]) swap(nums, i, nums[i]
        }
        List<Integer> ans = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (nums[i] != i + 1) ans.add(i + 1);
        }
        return ans;
    }
    void swap(int[] nums, int a, int b) {
        int c = nums[a];
        nums[a] = nums[b];
        nums[b] = c;
    }
}
```

- 时间复杂度：每个数字最多挪动一次。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

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：

```
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：

```
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：

```
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：

```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：

```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:

```
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]。

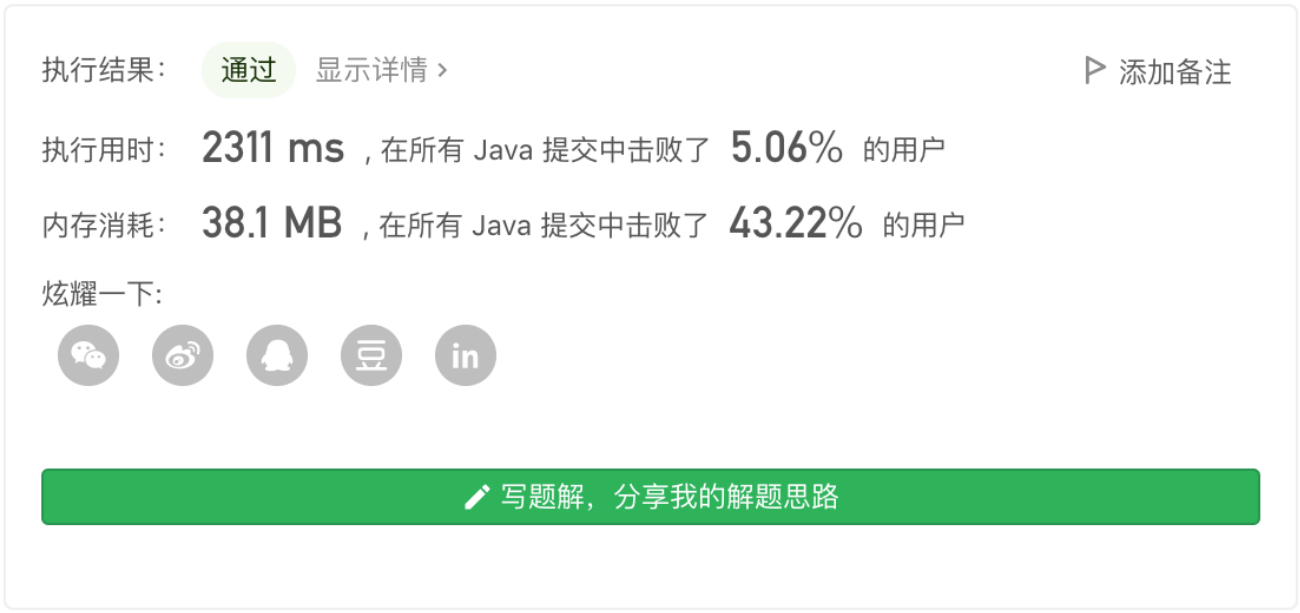
基本分析

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。



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：$O(\log n)$

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：

```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。

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：$O(\log n)$

题目描述

这是 LeetCode 上的 **645. 错误的集合** ，难度为 **简单**。

Tag：「模拟」、「哈希表」、「数学」、「桶排序」

集合 `s` 包含从 `1` 到 `n` 的整数。不幸的是，因为数据错误，导致集合里面某一个数字复制了成了集合里面的另外一个数字的值，导致集合 丢失了一个数字 并且 有一个数字重复 。

给定一个数组 `nums` 代表了集合 `S` 发生错误后的结果。

请你找出重复出现的整数，再找到丢失的整数，将它们以数组的形式返回。

示例 1：

```
输入：nums = [1,2,2,4]
输出：[2,3]
```

示例 2：

```
输入：nums = [1,1]
输出：[1,2]
```

提示：

- 2 <= nums.length <= 10^4
- 1 <= nums[i] <= 10^4

计数

一个朴素的做法是，使用「哈希表」统计每个元素出现次数，然后在 $[1, n]$ 查询每个元素的出现次数。

在「哈希表」中出现 2 次的为重复元素，未在「哈希表」中出现的元素为缺失元素。

由于这里数的范围确定为 $[1, n]$，我们可以使用数组来充当「哈希表」，以减少「哈希表」的哈希函数执行和冲突扩容的时间开销。

代码：

```Java
class Solution {
    public int[] findErrorNums(int[] nums) {
        int n = nums.length;
        int[] cnts = new int[n + 1];
        for (int x : nums) cnts[x]++;
        int[] ans = new int[2];
        for (int i = 1; i <= n; i++) {
            if (cnts[i] == 0) ans[1] = i;
            if (cnts[i] == 2) ans[0] = i;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

数学

我们还可以利用数值范围为 $[1, n]$，只有一个数重复和只有一个缺失的特性，进行「作差」求解。

- 令 $[1, n]$ 的求和为 tot，这部分可以使用「等差数列求和公式」直接得出：$tot = \frac{n(1+n)}{2}$；
- 令数组 $nums$ 的求和值为 sum，由循环累加可得；
- 令数组 $sums$ 去重求和值为 set，由循环配合「哈希表/数组」累加可得。

最终答案为 `(重复元素, 缺失元素) = (sum-set, tot-set)`。

代码：

```Java
class Solution {
    public int[] findErrorNums(int[] nums) {
        int n = nums.length;
        int[] cnts = new int[n + 1];
        int tot = (1 + n) * n / 2;
        int sum = 0, set = 0;
        for (int x : nums) {
            sum += x;
            if (cnts[x] == 0) set += x;
            cnts[x] = 1;
        }
        return new int[]{sum - set, tot - set};
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

桶排序

因为值的范围在 $[1, n]$，我们可以运用「桶排序」的思路，根据 $nums[i] = i + 1$ 的对应关系使用 $O(n)$ 的复杂度将每个数放在其应该落在的位置里。

然后线性扫描一遍排好序的数组，找到不符合 $nums[i] = i + 1$ 对应关系的位置，从而确定重复元素和缺失元素是哪个值。

代码：

```java
class Solution {
    public int[] findErrorNums(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            while (nums[i] != i + 1 && nums[nums[i] - 1] != nums[i]) {
                swap(nums, i, nums[i] - 1);
            }
        }
        int a = -1, b = -1;
        for (int i = 0; i < n; i++) {
            if (nums[i] != i + 1) {
                a = nums[i];
                b = i == 0 ? 1 : nums[i - 1] + 1;
            }
        }
        return new int[]{a, b};
    }
    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：$O(1)$

> **🍭 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 **703. 数据流中的第 K 大元素** ，难度为 **简单**。

Tag：「Top K」、「排序」、「堆」、「优先队列」

设计一个找到数据流中第 k 大元素的类（class）。注意是排序后的第 k 大元素，不是第 k 个不同的元素。

请实现 KthLargest 类：

- KthLargest(int k, int[] nums) 使用整数 k 和整数流 nums 初始化对象。

- int add(int val) 将 val 插入数据流 nums 后，返回当前数据流中第 k 大的元素。

示例：

```
输入：
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]
输出：
[null, 4, 5, 5, 8, 8]

解释：
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3);   // return 4
kthLargest.add(5);   // return 5
kthLargest.add(10);  // return 5
kthLargest.add(9);   // return 8
kthLargest.add(4);   // return 8
```

提示：

- 1 <= k <= 10^4
- 0 <= nums.length <= 10^4
- -10^4 <= nums[i] <= 10^4
- -10^4 <= val <= 10^4
- 最多调用 add 方法 10^4 次
- 题目数据保证，在查找第 k 大元素时，数组中至少有 k 个元素

冒泡排序 (TLE)

每次调用 `add` 时先将数装入数组，然后遍历 `k` 次，通过找 `k` 次最大值来找到 Top K。

代码：

```Java
class KthLargest {
    int k;
    List<Integer> list = new ArrayList<>(10009);
    public KthLargest(int _k, int[] _nums) {
        k = _k;
        for (int i : _nums) list.add(i);
    }
    public int add(int val) {
        list.add(val);
        int cur = 0;
        for (int i = 0; i < k; i++) {
            int idx = findMax(cur, list.size() - 1);
            swap(cur++, idx);
        }
        return list.get(cur - 1);
    }
    int findMax(int start, int end) {
        int ans = 0, max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int t = list.get(i);
            if (t > max) {
                max = t;
                ans = i;
            }
        }
        return ans;
    }
    void swap(int a, int b) {
        int c = list.get(a);
        list.set(a, list.get(b));
        list.set(b, c);
    }
}
```

- 时间复杂度：$O(nk)$
- 空间复杂度：$O(n)$

快速排序

上述的解法时间复杂度是 $O(nk)$ 的，当 `k` 很大的时候会超时。

我们可以使用快排来代替冒泡，将复杂度变为 $O(n \log n)$。

代码：

```
class KthLargest {
    int k;
    List<Integer> list = new ArrayList<>(10009);
    public KthLargest(int _k, int[] _nums) {
        k = _k;
        for (int i : _nums) list.add(i);
    }

    public int add(int val) {
        list.add(val);
        Collections.sort(list);
        return list.get(list.size() - k);
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(n)$

优先队列

使用优先队列构建一个容量为 `k` 的小根堆。

将 `nums` 中的前 `k` 项放入优先队列（此时堆顶元素为前 `k` 项的最大值）。

随后逐项加入优先队列：

- 堆内元素个数达到 `k` 个：
    - 加入项小于等于堆顶元素：加入项排在第 `k` 大元素的后面。直接忽略
    - 加入项大于堆顶元素：将堆顶元素弹出，加入项加入优先队列，调整堆
- 堆内元素个数不足 `k` 个，将加入项加入优先队列

将堆顶元素进行返回（数据保证返回答案时，堆内必然有 `k` 个元素）：

代码：

```java
class KthLargest {
    int k;
    PriorityQueue<Integer> queue;
    public KthLargest(int _k, int[] _nums) {
        k = _k;
        queue = new PriorityQueue<>(k, (a,b)->Integer.compare(a,b));
        int n = _nums.length;
        for (int i = 0; i < k && i < n; i++) queue.add(_nums[i]);
        for (int i = k; i < n; i++) add(_nums[i]);
    }
    public int add(int val) {
        int t = !queue.isEmpty() ? queue.peek() : Integer.MIN_VALUE;
        if (val > t || queue.size() < k) {
            if (!queue.isEmpty() && queue.size() >= k) queue.poll();
            queue.add(val);
        }
        return queue.peek();
    }
}
```

- 时间复杂度：最坏情况下，`n` 个元素都需要入堆。复杂度为 $O(n \log k)$
- 空间复杂度：$O(k)$

> **🍭 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 **987. 二叉树的垂序遍历** ，难度为 **困难**。

Tag：「数据结构运用」、「二叉树」、「哈希表」、「排序」、「优先队列」、「DFS」

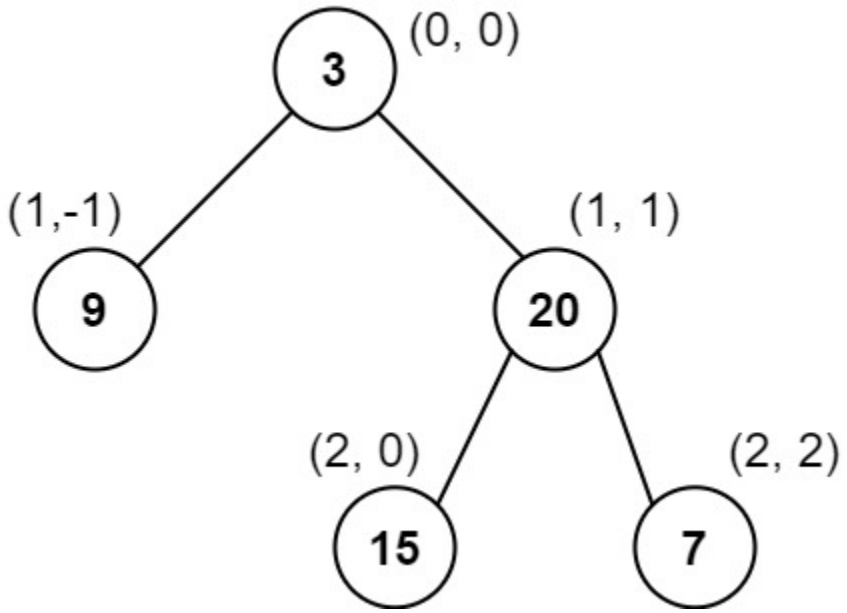
给你二叉树的根结点 root ，请你设计算法计算二叉树的 垂序遍历 序列。

对位于 (row, col) 的每个结点而言，其左右子结点分别位于 (row + 1, col - 1) 和 (row + 1, col + 1) 。树的根结点位于 (0, 0) 。

二叉树的 垂序遍历 从最左边的列开始直到最右边的列结束，按列索引每一列上的所有结点，形成一个按出现位置从上到下排序的有序列表。如果同行同列上有多个结点，则按结点的值从小到大进行排序。

返回二叉树的 垂序遍历 序列。

示例 1：

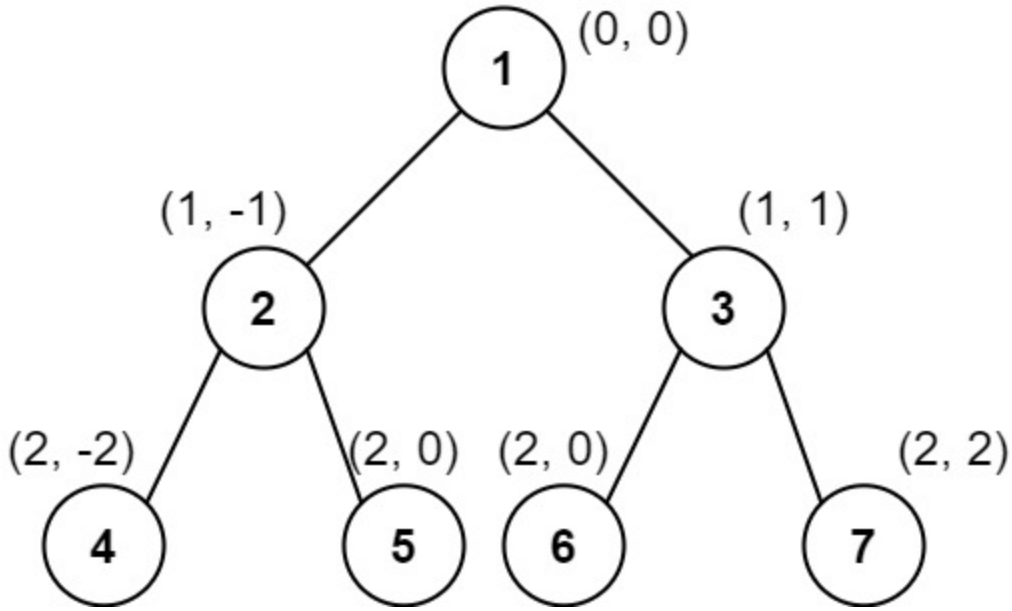



```
输入：root = [3,9,20,null,null,15,7]

输出：[[9],[3,15],[20],[7]]

解释：
列 -1 ：只有结点 9 在此列中。
列  0 ：只有结点 3 和 15 在此列中，按从上到下顺序。
列  1 ：只有结点 20 在此列中。
列  2 ：只有结点 7 在此列中。
```

示例 2：

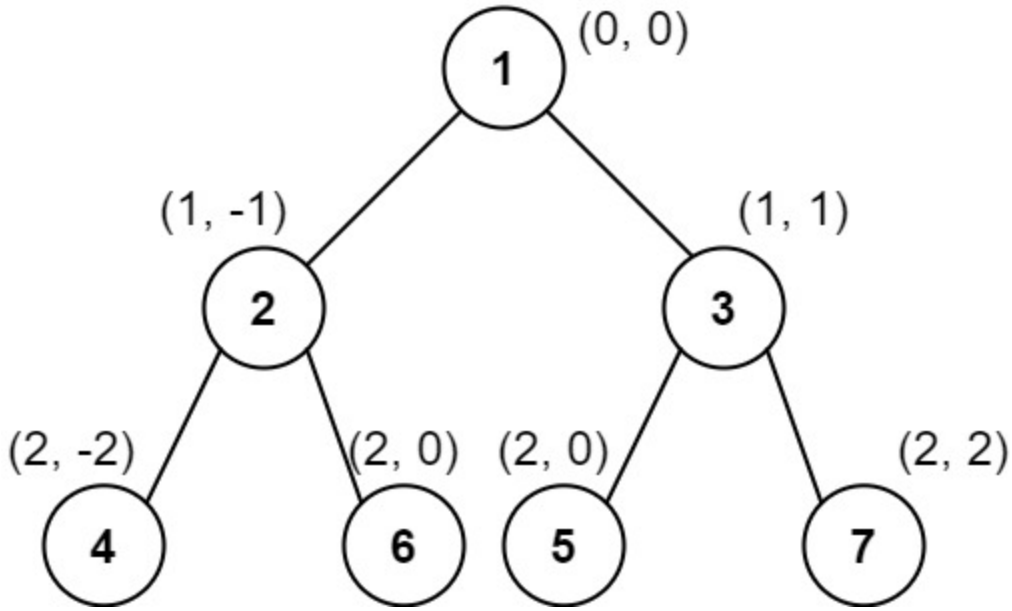



```
输入：root = [1,2,3,4,5,6,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：
列 -2 ：只有结点 4 在此列中。
列 -1 ：只有结点 2 在此列中。
列  0 ：结点 1 、5 和 6 都在此列中。
          1 在上面，所以它出现在前面。
          5 和 6 位置都是 (2, 0) ，所以按值从小到大排序，5 在 6 的前面。
列  1 ：只有结点 3 在此列中。
列  2 ：只有结点 7 在此列中。
```

示例 3：



输入：root = [1,2,3,4,6,5,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：
这个示例实际上与示例 2 完全相同，只是结点 5 和 6 在树中的位置发生了交换。
因为 5 和 6 的位置仍然相同，所以答案保持不变，仍然按值从小到大排序。

提示：

- 树中结点数目总数在范围 [1, 10]

DFS + 哈希表 + 排序

根据题意，我们需要按照优先级「**“列号从小到大”，对于同列节点，“行号从小到大”，对于同列同行元素，“节点值从小到大”**」进行答案构造。

因此我们可以对树进行遍历，遍历过程中记下这些信息 (col, row, val)，然后根据规则进行排序，并构造答案。

我们可以先使用「哈希表」进行存储，最后再进行一次性的排序。

代码：

```Java
class Solution {
    Map<TreeNode, int[]> map = new HashMap<>(); // col, row, val
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        map.put(root, new int[]{0, 0, root.val});
        dfs(root);
        List<int[]> list = new ArrayList<>(map.values());
        Collections.sort(list, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            if (a[1] != b[1]) return a[1] - b[1];
            return a[2] - b[2];
        });
        int n = list.size();
        List<List<Integer>> ans = new ArrayList<>();
        for (int i = 0; i < n; ) {
            int j = i;
            List<Integer> tmp = new ArrayList<>();
            while (j < n && list.get(j)[0] == list.get(i)[0]) tmp.add(list.get(j++)[2]);
            ans.add(tmp);
            i = j;
        }
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return ;
        int[] info = map.get(root);
        int col = info[0], row = info[1], val = info[2];
        if (root.left != null) {
            map.put(root.left, new int[]{col - 1, row + 1, root.left.val});
            dfs(root.left);
        }
        if (root.right != null) {
            map.put(root.right, new int[]{col + 1, row + 1, root.right.val});
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：令总节点数量为 n，填充哈希表时进行树的遍历，复杂度为 $O(n)$；构造答案时需要进行排序，复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

DFS + 优先队列（堆）

显然，最终要让所有节点的相应信息有序，可以使用「优先队列（堆）」边存储边维护有序性。

代码：

```
class Solution {
    PriorityQueue<int[]> q = new PriorityQueue<>((a, b)->{ // col, row, val
        if (a[0] != b[0]) return a[0] - b[0];
        if (a[1] != b[1]) return a[1] - b[1];
        return a[2] - b[2];
    });
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        int[] info = new int[]{0, 0, root.val};
        q.add(info);
        dfs(root, info);
        List<List<Integer>> ans = new ArrayList<>();
        while (!q.isEmpty()) {
            List<Integer> tmp = new ArrayList<>();
            int[] poll = q.peek();
            while (!q.isEmpty() && q.peek()[0] == poll[0]) tmp.add(q.poll()[2]);
            ans.add(tmp);
        }
        return ans;
    }
    void dfs(TreeNode root, int[] fa) {
        if (root.left != null) {
            int[] linfo = new int[]{fa[0] - 1, fa[1] + 1, root.left.val};
            q.add(linfo);
            dfs(root.left, linfo);
        }
        if (root.right != null) {
            int[] rinfo = new int[]{fa[0] + 1, fa[1] + 1, root.right.val};
            q.add(rinfo);
            dfs(root.right, rinfo);
        }
    }
}
```

- 时间复杂度：令总节点数量为 n，将节点信息存入优先队列（堆）复杂度为 $O(n\log n)$；构造答案复杂度为 $O(n\log n)$。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 **

题目描述

这是 LeetCode 上的 **1833. 雪糕的最大数量** ，难度为 **中等**。

Tag：「贪心」、「排序」

夏日炎炎，小男孩 Tony 想买一些雪糕消消暑。

商店中新到 `n` 支雪糕，用长度为 n 的数组 `costs` 表示雪糕的定价，其中 `costs[i]` 表示第 `i` 支雪糕的现金价格。

Tony 一共有 `coins` 现金可以用于消费，他想要买尽可能多的雪糕。

给你价格数组 `costs` 和现金量 `coins` ，请你计算并返回 Tony 用 `coins` 现金能够买到的雪糕的 最大数量 。

注意：Tony 可以按任意顺序购买雪糕。

示例 1：

```
输入：costs = [1,3,2,4,1], coins = 7

输出：4

解释：Tony 可以买下标为 0、1、2、4 的雪糕，总价为 1 + 3 + 2 + 1 = 7
```

示例 2：

```
输入：costs = [10,6,8,7,7,8], coins = 5

输出：0

解释：Tony 没有足够的钱买任何一支雪糕。
```

示例 3：

输入：costs = [1,6,3,1,2,5],

输出：6

解释：Tony 可以买下所有的雪糕，总价为 1 + 6 + 3 + 1 + 2 + 5 = 18 。

提示：

- costs.length == n
- 1 <= n <= 10^5
- 1 <= costs[i] <= 10^5
- 1 <= coins <= 10^8

基本分析

从题面看，是一道「01 背包」问题，每个物品的成本为 $cost[i]$，价值为 1。

但「01 背包」的复杂度为 $O(N * C)$，其中 N 为物品数量（数量级为 10^5），C 为背包容量(数量级为 10^8)。显然会 TLE。

换个思路发现，每个被选择的物品对答案的贡献都是 1，优先选择价格小的物品会使得我们剩余金额尽可能的多，将来能够做的决策方案也就相应变多。

因此一个直观的做法是，对物品数组进行「从小到大」排序，然后「从前往后」开始决策购买。

证明

直观上，这样的贪心思路可以使得最终选择的物品数量最多。

接下来证明一下该思路的正确性。

假定贪心思路取得的序列为 $[a1, a2, a3, ..., an]$（长度为 n），真实最优解所取得的序列为 $[b1, b2, b3, .., bm]$（长度为 m）。

两个序列均为「单调递增序列」。

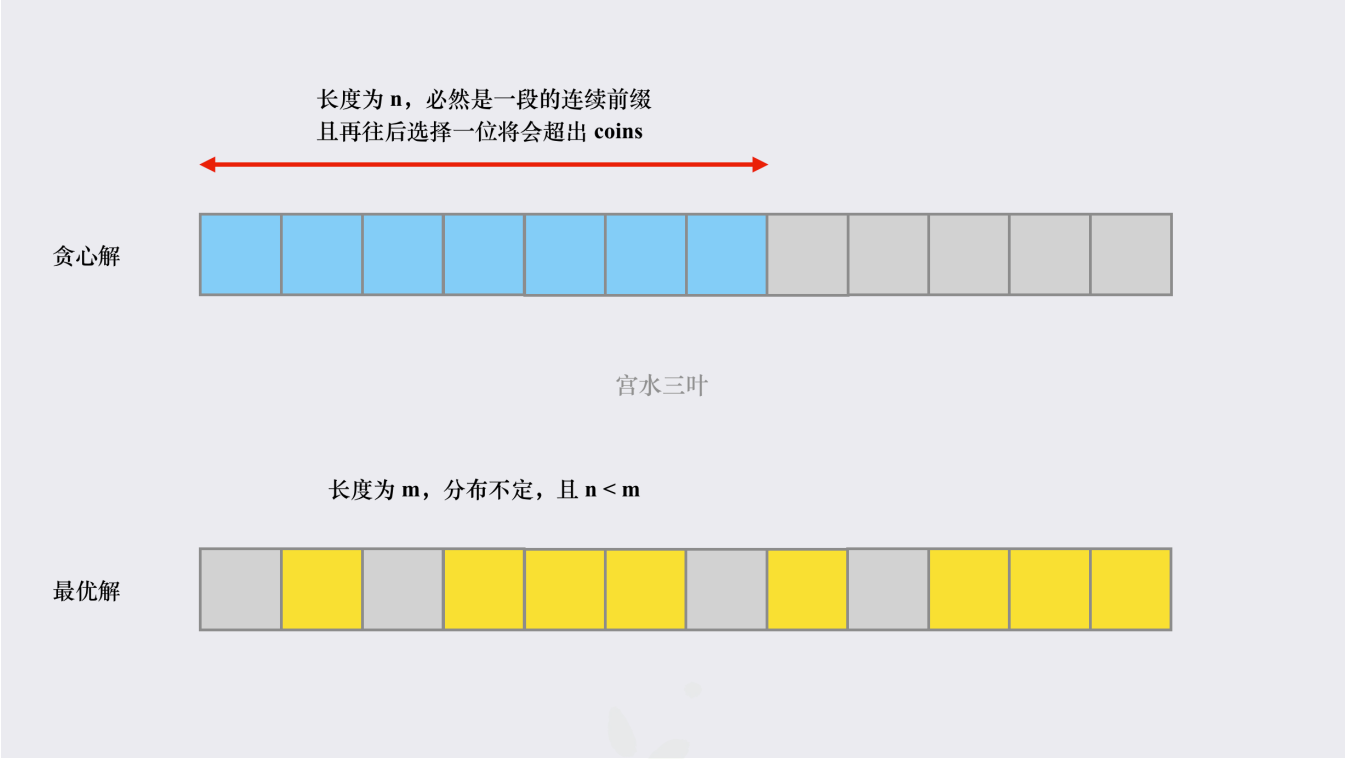
其中最优解所对应具体方案不唯一，即存在多种选择方案使得物品数量相同。

因此，**我们只需要证明两个序列长度一致即可**。

按照贪心逻辑，最终选择的方案总成本不会超过 $coins$，因此至少是一个合法的选择方案，天然有 $n \leq m$，只需要证明 $n \geq m$ 成立，即可得证 $n = m$。

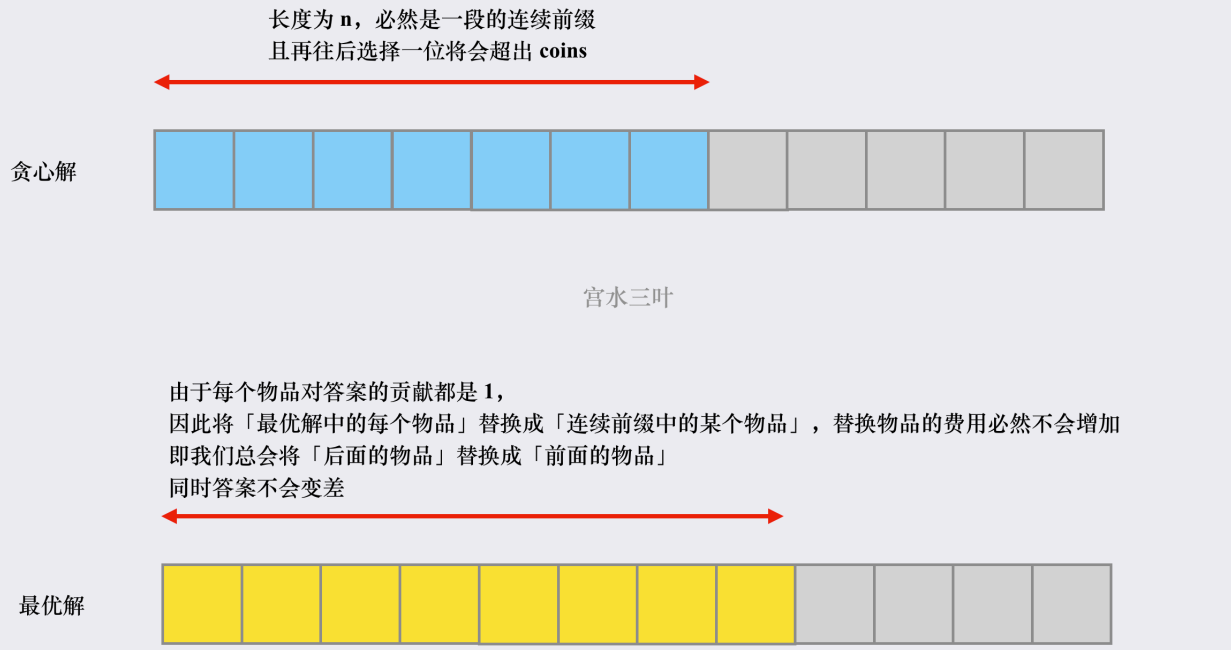
通过反证法证明 $n \geq m$ 成立，假设 $n \geq m$ 不成立，即有 $n < m$。

根据贪心决策，**我们选择的物品序列在「排序好的 $cost$ 数组」里，必然是一段连续的前缀。并且再选择下一物品将会超过总费用 $coins$；而真实最优解的选择方案在「排序好的 $cost$ 数组」里分布不定。**



这时候我们可以利用「**每个物品对答案的贡献均为 1，将最优解中的分布靠后的物品，替换为分布较前的物品，不会使得费用增加，同时答案不会变差**」。

从而将真实最优解也调整为某段连续的前缀。



综上，通过反证法得证 $n \geq m$ 成立，结合 $n \leq m$，可推出 $n = m$。

即贪心解必然能够取得与最优解一样的长度。

贪心

排序，从前往后决策，直到不能决策为止。

代码：

```Java
class Solution {
    public int maxIceCream(int[] cs, int t) {
        int n = cs.length;
        Arrays.sort(cs);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (t >= cs[i]) {
                ans++;
                t -= cs[i];
            }
        }
        return ans;
    }
}
```

- 时间复杂度：排序复杂度为 $O(n \log n)$；获取答案的复杂度为 $O(n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：排序复杂度为 $O(\log n)$。整体复杂度为 $O(\log n)$

PS. 这里假定 `Arrays.sort` 使用的是「双轴排序」的实现。

> **🍭 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 **1834. 单线程 CPU** ，难度为 **中等**。

Tag：「模拟」、「排序」、「优先队列」

给你一个二维数组 $tasks$，用于表示 n 项从 0 到 $n-1$ 编号的任务。

其中 $tasks[i] = [enqueueTime_i, processingTime_i]$ 意味着第 i 项任务将会于 $enqueueTime_i$ 时进入任务队列，需要 $processingTime_i$ 的时长完成执行。

现有一个单线程 CPU ，同一时间只能执行**最多一项**任务，该 CPU 将会按照下述方式运行：

- 如果 CPU 空闲，且任务队列中没有需要执行的任务，则 CPU 保持空闲状态。
- 如果 CPU 空闲，但任务队列中有需要执行的任务，则 CPU 将会选择 执行时间最短的任务开始执行。如果多个任务具有同样的最短执行时间，则选择下标最小的任务

开始执行。

- 一旦某项任务开始执行，CPU 在 执行完整个任务 前都不会停止。
- CPU 可以在完成一项任务后，立即开始执行一项新任务。

返回 CPU 处理任务的顺序。

示例 1：

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

输出：[0,2,3,1]

解释：事件按下述流程运行：
- time = 1 ，任务 0 进入任务队列，可执行任务项 = {0}
- 同样在 time = 1 ，空闲状态的 CPU 开始执行任务 0 ，可执行任务项 = {}
- time = 2 ，任务 1 进入任务队列，可执行任务项 = {1}
- time = 3 ，任务 2 进入任务队列，可执行任务项 = {1, 2}
- 同样在 time = 3 ，CPU 完成任务 0 并开始执行队列中用时最短的任务 2 ，可执行任务项 = {1}
- time = 4 ，任务 3 进入任务队列，可执行任务项 = {1, 3}
- time = 5 ，CPU 完成任务 2 并开始执行队列中用时最短的任务 3 ，可执行任务项 = {1}
- time = 6 ，CPU 完成任务 3 并开始执行任务 1 ，可执行任务项 = {}
- time = 10 ，CPU 完成任务 1 并进入空闲状态
```

示例 2：

```
输入：tasks = [[7,10],[7,12],[7,5],[7,4],[7,2]]

输出：[4,3,2,0,1]

解释：事件按下述流程运行：
- time = 7 ，所有任务同时进入任务队列，可执行任务项  = {0,1,2,3,4}
- 同样在 time = 7 ，空闲状态的 CPU 开始执行任务 4 ，可执行任务项 = {0,1,2,3}
- time = 9 ，CPU 完成任务 4 并开始执行任务 3 ，可执行任务项 = {0,1,2}
- time = 13 ，CPU 完成任务 3 并开始执行任务 2 ，可执行任务项 = {0,1}
- time = 18 ，CPU 完成任务 2 并开始执行任务 0 ，可执行任务项 = {1}
- time = 28 ，CPU 完成任务 0 并开始执行任务 1 ，可执行任务项 = {}
- time = 40 ，CPU 完成任务 1 并进入空闲状态
```

提示：

- tasks.length == n
- 1 <= n <= 10^5
- 1 <= $enqueueTime_i, processingTime_i$ <= 10^9

模拟 + 数据结构

先将 $tasks$ 按照「入队时间」进行升序排序，同时为了防止任务编号丢失，排序前需要先将二元组的 $tasks$ 转存为三元组，新增记录的是原任务编号。

然后可以按照「时间线」进行模拟：

1. 起始令 $time$ 从 1 开始进行递增，每次将到达「入队时间」的任务进行入队；
2. 判断当前队列是否有可以执行的任务：
    1. 如果没有，说明还没到达下一个入队任务的入队时间，直接将 $times$ 快进到下一个入队任务的入队时间；
    2. 如果有，从队列中取出任务执行，同时由于是单线程执行，在该任务结束前，不会有新任务被执行，将 $times$ 快进到该任务的结束时间。

代码：

```Java
class Solution {
    public int[] getOrder(int[][] ts) {
        int n = ts.length;
        // 将 ts 转存成 nts，保留任务编号
        int[][] nts = new int[n][3];
        for (int i = 0; i < n; i++) nts[i] = new int[]{ts[i][0], ts[i][1], i};
        // 根据任务入队时间进行排序
        Arrays.sort(nts, (a,b)->a[0]-b[0]);
        // 根据题意，先按照「持续时间」排序，再根据「任务编号」排序
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->{
            if (a[1] != b[1]) return a[1] - b[1];
            return a[2] - b[2];
        });
        int[] ans = new int[n];
        for (int time = 1, j = 0, idx = 0; idx < n; ) {
                // 如果当前任务可以添加到「队列」中（满足入队时间）则进行入队
            while (j < n && nts[j][0] <= time) q.add(nts[j++]);
            if (q.isEmpty()) {
                    // 如果当前「队列」没有任务，直接跳到下个任务的入队时间
                time = nts[j][0];
            } else {
                    // 如果有可执行任务的话，根据优先级将任务出队（记录下标），并跳到该任务完成时间点
                int[] cur = q.poll();
                ans[idx++] = cur[2];
                time += cur[1];
            }
        }
        return ans;
    }
}
```

- 时间复杂度：将 ts 转存成 nts 的复杂度为 $O(n)$；对 nts 排序复杂度为 $O(n \log n)$；模拟时间线，将任务进行入队出队操作，并构造最终答案复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

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：

```
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：

```
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 $nums[r]$。

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：

```Java
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 $O(n)$；`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「排序」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：