宫水三叶的刷题日记

堆

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「堆」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「堆」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「堆」；
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 **23. 合并K个升序链表** ，难度为 **中等**。

Tag : 「优先队列」、「堆」、「链表」

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]

输出：[1,1,2,3,4,4,5,6]

解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

示例 2：

```
输入：lists = []

输出：[]
```

示例 3：

```
输入：lists = [[]]

输出：[]
```

提示：

- k == lists.length
- 0 <= k <= 10^4
- 0 <= lists[i].length <= 500
- -10^4 <= lists[i][j] <= 10^4
- lists[i] **按 升序 排列**
- lists[i].length **的总和不超过** 10^4

优先队列（哨兵技巧）

哨兵技巧我们在前面的多道链表题讲过，让三叶来帮你回忆一下：

做有关链表的题目，有个常用技巧：添加一个虚拟头结点（哨兵），帮助简化边界情况的判断。

由于所有链表本身满足「升序」，一个直观的做法是，我们比较每条链表的头结点，选取值最小的节点，添加到结果中，然后更新该链表的的头结点为该节点的 next 指针。循环比较，直到所有的节点都被加入结果中。

对应到代码的话，相当于我们需要准备一个「集合」，将所有链表的头结点放入「集合」，然后每次都从「集合」中挑出最小值，并将最小值的下一个节点添加进「集合」（如果有的话），循环这个过程，直到「集合」为空（说明所有节点都处理完，进过集合又从集合中出来）。

而「堆」则是满足这样要求的数据结构。

代码：

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        ListNode dummy = new ListNode(-1), tail = dummy;
        PriorityQueue<ListNode> q = new PriorityQueue<>((a, b) -> a.val - b.val);
        for (ListNode node : lists) {
            if (node != null) q.add(node);
        }
        while (!q.isEmpty()) {
            ListNode poll = q.poll();
            tail.next = poll;
            tail = tail.next;
            if (poll.next != null) q.add(poll.next);
        }
        return dummy.next;
    }
}
```

- 时间复杂度：会将每个节点处理一遍。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

题目描述

这是 LeetCode 上的 **218. 天际线问题** ，难度为 **困难**。

Tag：「扫描线问题」、「优先队列」

城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。给你所有建筑物的位置和高度，请返回由这些建筑物形成的 天际线 。

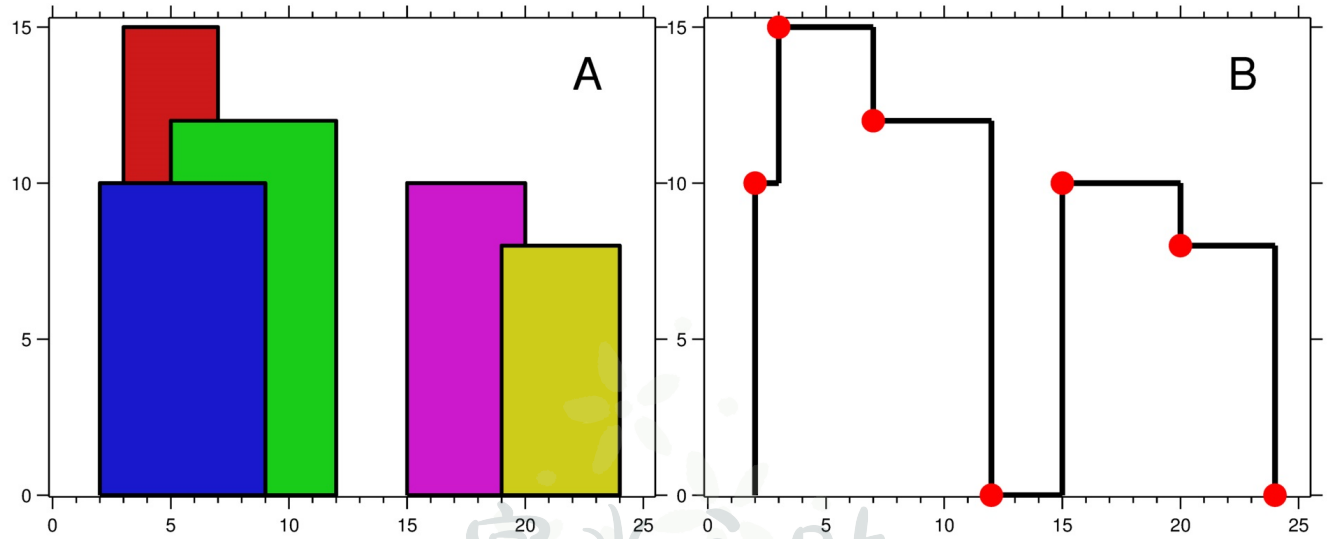
每个建筑物的几何信息由数组 `buildings` 表示，其中三元组 `buildings[i] = [lefti, righti, heighti]` 表示：

- `left[i]` 是第 i 座建筑物左边缘的 x 坐标。
- `right[i]` 是第 i 座建筑物右边缘的 x 坐标。
- `height[i]` 是第 i 座建筑物的高度。

天际线 应该表示为由 “关键点” 组成的列表，格式 `[[x1,y1],[x2,y2],...]` ，并按 x 坐标 进行排序 。关键点是水平线段的左端点。列表中最后一个点是最右侧建筑物的终点，y 坐标始终为 0 ，仅用于标记天际线的终点。此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。

注意：输出天际线中不得有连续的相同高度的水平线。例如 `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` 是不正确的答案；三条高度为 5 的线应该在最终输出中合并为一个：`[...[2 3], [4 5], [12 7], ...]`

示例 1：



```
输入：buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

输出：[[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]

解释：
图 A 显示输入的所有建筑物的位置和高度，
图 B 显示由这些建筑物形成的天际线。图 B 中的红点表示输出列表中的关键点。
```

示例 2：

```
输入：buildings = [[0,2,3],[2,5,3]]

输出：[[0,3],[5,0]]
```

提示：

- 1 <= buildings.length <= 10^4
- 0 <= lefti < righti <= 2^{31} - 1
- 1 <= heighti <= 2^{31} - 1
- buildings 按 lefti 非递减排序

基本分析

这是一题特别的扫描线问题 🤣🤣🤣

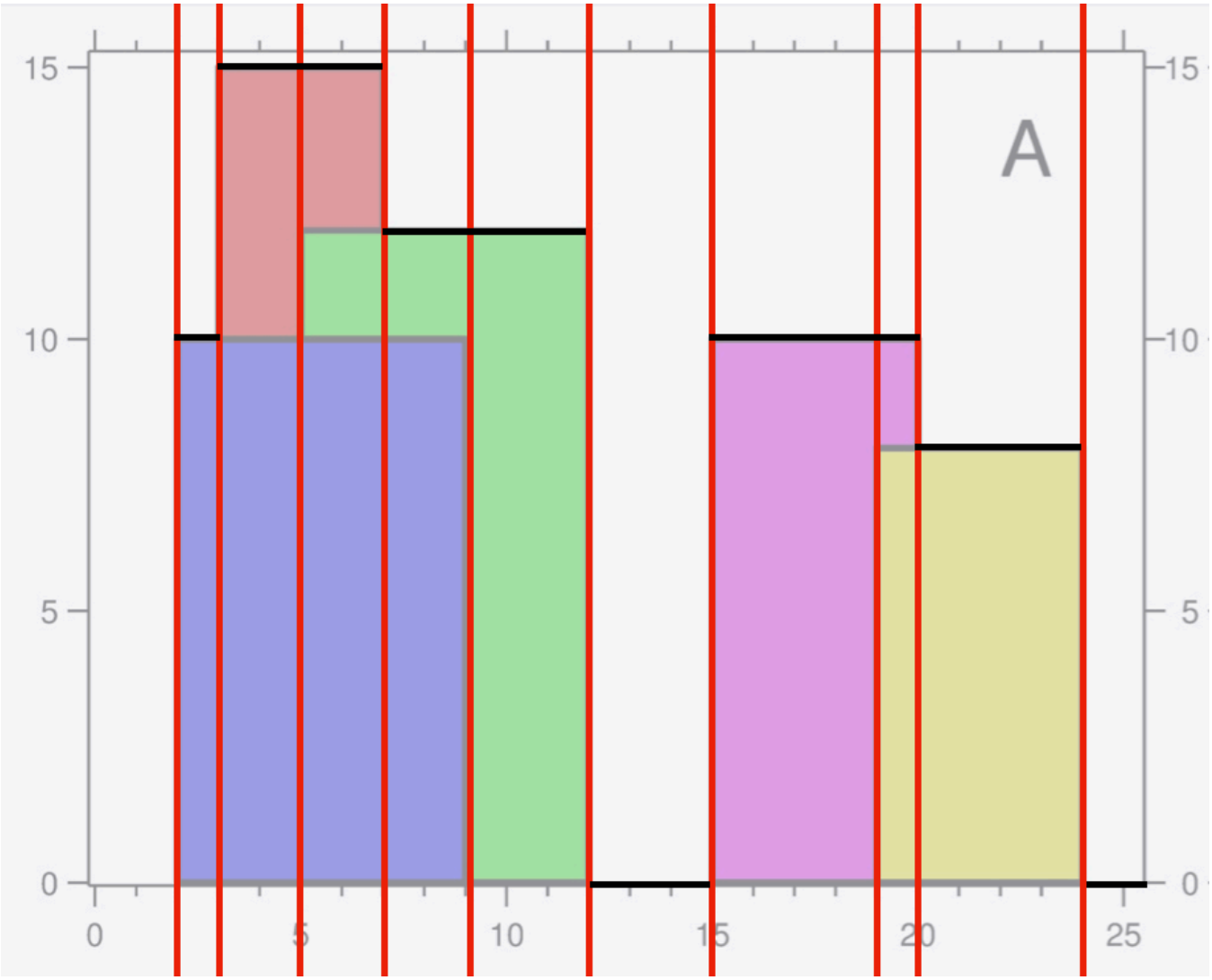
既不是求周长，也不是求面积，是求轮廓中的所有的水平线的左端点 🤣🤣🤣

所以这不是一道必须用「线段树」来解决的扫描线问题（因为不需要考虑区间查询问题）。

扫描线的核心在于 **将不规则的形状按照水平或者垂直的方式，划分成若干个规则的矩形。**

扫描线

对于本题，对应的扫描线分割形状如图：



不难发现，由相邻两个横坐标以及最大高度，可以确定一个矩形。

题目要我们 **输出每个矩形的“上边”的左端点，同时跳过可由前一矩形“上边”延展而来的那些边。**

因此我们需要实时维护一个最大高度，可以使用优先队列（堆）。

实现时，我们可以先记录下 $buildings$ 中所有的左右端点横坐标及高度，并根据端点横坐标进行从小到大排序。

在从前往后遍历处理时（遍历每个矩形），根据当前遍历到的点进行分情况讨论：

- 左端点：因为是左端点，必然存在一条从右延展的边，但不一定是需要被记录的边，因为在同一矩形中，我们只需要记录最上边的边。这时候可以将高度进行入队；

- 右端点：此时意味着之前某一条往右延展的线结束了，这时候需要将高度出队（代表这结束的线不被考虑）。

然后从优先队列中取出当前的最大高度，为了防止当前的线与前一矩形“上边”延展而来的线重合，我们需要使用一个变量 `prev` 记录上一个记录的高度。

代码：

```java
class Solution {
    public List<List<Integer>> getSkyline(int[][] bs) {
        List<List<Integer>> ans = new ArrayList<>();

        // 预处理所有的点，为了方便排序，对于左端点，令高度为负；对于右端点令高度为正
        List<int[]> ps = new ArrayList<>();
        for (int[] b : bs) {
            int l = b[0], r = b[1], h = b[2];
            ps.add(new int[]{l, -h});
            ps.add(new int[]{r, h});
        }

        // 先按照横坐标进行排序
        // 如果横坐标相同，则按照左端点排序
        // 如果相同的左/右端点，则按照高度进行排序
        Collections.sort(ps, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            return a[1] - b[1];
        });

        // 大根堆
        PriorityQueue<Integer> q = new PriorityQueue<>((a,b)->b-a);
        int prev = 0;
        q.add(prev);
        for (int[] p : ps) {
            int point = p[0], height = p[1];
            if (height < 0) {
                // 如果是左端点，说明存在一条往右延伸的可记录的边，将高度存入优先队列
                q.add(-height);
            } else {
                // 如果是右端点，说明这条边结束了，将当前高度从队列中移除
                q.remove(height);
            }

            // 取出最高高度，如果当前不与前一矩形“上边”延展而来的那些边重合，则可以被记录
            int cur = q.peek();
            if (cur != prev) {
                List<Integer> list = new ArrayList<>();
                list.add(point);
                list.add(cur);
                ans.add(list);
                prev = cur;
            }
        }
        return ans;
    }
```

```
}
```

- 时间复杂度：需要处理的矩阵数量与 n 正比，每个矩阵需要使用优先队列维护高度，其中 `remove` 操作需要先花费 $O(n)$ 复杂度进行查找，然后通过 $O(\log n)$ 复杂度进行移除，复杂度为 $O(n)$。整体复杂度为 $O(n^2)$
- 空间复杂度：$O(n)$

答疑

1. 将左端点的高度存成负数再进行排序是什么意思？

这里只是为了方便，所以采取了这样的做法，当然也能够多使用一位来代指「左右」。

只要最终可以达到如下的排序规则即可：

1. 先严格按照横坐标进行「从小到大」排序
2. 对于某个横坐标而言，可能会同时出现多个点，应当按照如下规则进行处理：
    1. 优先处理左端点，再处理右端点
    2. 如果同样都是左端点，则按照高度「从大到小」进行处理（将高度增加到优先队列中）
    3. 如果同样都是右端点，则按照高度「从小到大」进行处理（将高度从优先队列中删掉）

代码：

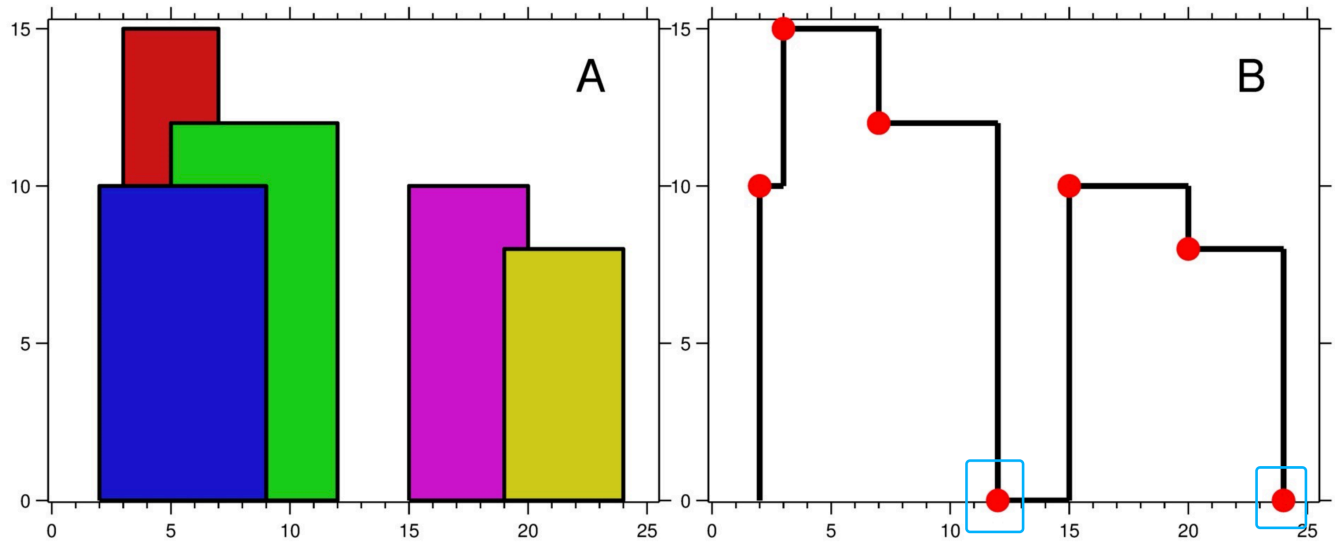
```java
class Solution {
    public List<List<Integer>> getSkyline(int[][] bs) {
        List<List<Integer>> ans = new ArrayList<>();
        List<int[]> ps = new ArrayList<>();
        for (int[] b : bs) {
            int l = b[0], r = b[1], h = b[2];
            ps.add(new int[]{l, h, -1});
            ps.add(new int[]{r, h, 1});
        }
        /**
         * 先严格按照横坐标进行「从小到大」排序
         * 对于某个横坐标而言，可能会同时出现多个点，应当按照如下规则进行处理：
         * 1. 优先处理左端点，再处理右端点
         * 2. 如果同样都是左端点，则按照高度「从大到小」进行处理（将高度增加到优先队列中）
         * 3. 如果同样都是右端点，则按照高度「从小到大」进行处理（将高度从优先队列中删掉）
         */
        Collections.sort(ps, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            if (a[2] != b[2]) return a[2] - b[2];
            if (a[2] == -1) {
                return b[1] - a[1];
            } else {
                return a[1] - b[1];
            }
        });
        PriorityQueue<Integer> q = new PriorityQueue<>((a,b)->b-a);
        int prev = 0;
        q.add(prev);
        for (int[] p : ps) {
            int point = p[0], height = p[1], flag = p[2];
            if (flag == -1) {
                q.add(height);
            } else {
                q.remove(height);
            }

            int cur = q.peek();
            if (cur != prev) {
                List<Integer> list = new ArrayList<>();
                list.add(point);
                list.add(cur);
                ans.add(list);
                prev = cur;
            }
        }
        return ans;
```

```
    }
}
```

2. 为什么在处理前，先往「优先队列」添加一个 0 ？

因为题目本身要求我们把一个完整轮廓的「右下角」那个点也取到，所以需要先添加一个 0。

也就是下图被圈出来的那些点：



3. 优先队列的 `remove` 操作成为了瓶颈，如何优化？

由于优先队列的 `remove` 操作需要先经过 $O(n)$ 的复杂度进行查找，再通过 $O(\log n)$ 的复杂度进行删除。因此整个 `remove` 操作的复杂度是 $O(n)$ 的，这导致了我们算法整体复杂度为 $O(n^2)$。

优化方式包括：使用基于红黑树的 `TreeMap` 代替优先队列；或是使用「哈希表」记录「执行了删除操作的高度」及「删除次数」，在每次使用前先检查堆顶高度是否已经被标记删除，如果是则进行 `poll` 操作，并更新删除次数，直到遇到一个没被删除的堆顶高度。

代码：

```Java
class Solution {
    public List<List<Integer>> getSkyline(int[][] bs) {
        List<List<Integer>> ans = new ArrayList<>();
        List<int[]> ps = new ArrayList<>();
        for (int[] b : bs) {
            int l = b[0], r = b[1], h = b[2];
            ps.add(new int[]{l, h, -1});
            ps.add(new int[]{r, h, 1});
        }
        /**
         * 先严格按照横坐标进行「从小到大」排序
         * 对于某个横坐标而言，可能会同时出现多个点，应当按照如下规则进行处理：
         * 1. 优先处理左端点，再处理右端点
         * 2. 如果同样都是左端点，则按照高度「从大到小」进行处理（将高度增加到优先队列中）
         * 3. 如果同样都是右端点，则按照高度「从小到大」进行处理（将高度从优先队列中删掉）
         */
        Collections.sort(ps, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            if (a[2] != b[2]) return a[2] - b[2];
            if (a[2] == -1) {
                return b[1] - a[1];
            } else {
                return a[1] - b[1];
            }
        });
        // 记录进行了删除操作的高度，以及删除次数
        Map<Integer, Integer> map = new HashMap<>();
        PriorityQueue<Integer> q = new PriorityQueue<>((a,b)->b-a);
        int prev = 0;
        q.add(prev);
        for (int[] p : ps) {
            int point = p[0], height = p[1], flag = p[2];
            if (flag == -1) {
                q.add(height);
            } else {
                map.put(height, map.getOrDefault(height, 0) + 1);
            }

            while (!q.isEmpty()) {
                int peek = q.peek();
                if (map.containsKey(peek)) {
                    if (map.get(peek) == 1) map.remove(peek);
                    else map.put(peek, map.get(peek) - 1);
                    q.poll();
                } else {
                    break;
```

```
                }
            }

            int cur = q.peek();
            if (cur != prev) {
                List<Integer> list = new ArrayList<>();
                list.add(point);
                list.add(cur);
                ans.add(list);
                prev = cur;
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n \log n)$
- 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 **264. 丑数 II** ，难度为 **中等**。

Tag：「多路归并」、「堆」、「优先队列」

给你一个整数 n ，请你找出并返回第 n 个 丑数 。

丑数 就是只包含质因数 2、3 和 5 的正整数。

示例 1：

```
输入：n = 10

输出：12

解释：[1, 2, 3, 4, 5, 6, 8, 9, 10, 12] 是由前 10 个丑数组成的序列。
```

示例 2：

输入：n = 1

输出：1

解释：1 通常被视为丑数。

提示：

- 1 <= n <= 1690

基本思路

根据丑数的定义，我们有如下结论：

- 1 是最小的丑数。
- 对于任意一个丑数 x，其与任意的质因数（2、3、5）相乘，结果（$2x$、$3x$、$5x$）仍为丑数。

优先队列（小根堆）解法

有了基本的分析思路，一个简单的解法是使用优先队列：

1. 起始先将最小丑数 1 放入队列
2. 每次从队列取出最小值 x，然后将 x 所对应的丑数 $2x$、$3x$ 和 $5x$ 进行入队。
3. 对步骤 2 循环多次，第 n 次出队的值即是答案。

为了防止同一丑数多次进队，我们需要使用数据结构 Set 来记录入过队列的丑数。

代码：

```Java
class Solution {
    int[] nums = new int[]{2,3,5};
    public int nthUglyNumber(int n) {
        Set<Long> set = new HashSet<>();
        Queue<Long> pq = new PriorityQueue<>();
        set.add(1L);
        pq.add(1L);
        for (int i = 1; i <= n; i++) {
            long x = pq.poll();
            if (i == n) return (int)x;
            for (int num : nums) {
                long t = num * x;
                if (!set.contains(t)) {
                    set.add(t);
                    pq.add(t);
                }
            }
        }
        return -1;
    }
}
```

- 时间复杂度：从优先队列中取最小值为 $O(1)$，往优先队列中添加元素复杂度为 $O(\log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

多路归并（多指针）解法

从解法一中不难发现，我们「往后产生的丑数」都是基于「已有丑数」而来（使用「已有丑数」乘上「质因数」2、3、5）。

因此，如果我们所有丑数的有序序列为 $a1, a2, a3, ..., an$ 的话，序列中的每一个数都必然能够被以下三个序列（中的至少一个）覆盖：

- 由丑数 * 2 所得的有序序列：$1 * 2$、$2 * 2$、$3 * 2$、$4 * 2$、$5 * 2$、$6 * 2$、$8 * 2$ ...
- 由丑数 * 3 所得的有序序列：$1 * 3$、$2 * 3$、$3 * 3$、$4 * 3$、$5 * 3$、$6 * 3$、$8 * 3$ ...
- 由丑数 * 5 所得的有序序列：$1 * 5$、$2 * 5$、$3 * 5$、$4 * 5$、$5 * 5$、$6 * 5$、$8 * 5$ ...

举个🌰，假设我们需要求得 $[1, 2, 3, ..., 10, 12]$ 丑数序列 arr 的最后一位，那么该序列可以看作以下三个有序序列归并而来：

- $1 * 2, 2 * 2, 3 * 2, ..., 10 * 2, 12 * 2$ ，将 2 提出，即 $arr * 2$
- $1 * 3, 2 * 3, 3 * 3, ..., 10 * 3, 12 * 3$ ，将 3 提出，即 $arr * 3$
- $1 * 5, 2 * 5, 3 * 5, ..., 10 * 5, 12 * 5$ ，将 5 提出，即 $arr * 5$

因此我们可以使用三个指针来指向目标序列 arr 的某个下标（下标 0 作为哨兵不使用，起始都为 1），使用 $arr[下标] * 质因数$ 代表当前使用到三个有序序列中的哪一位，同时使用 idx 表示当前生成到 arr 哪一位丑数。

代码：

```
class Solution {
    public int nthUglyNumber(int n) {
        // ans 用作存储已有丑数（从下标 1 开始存储，第一个丑数为 1）
        int[] ans = new int[n + 1];
        ans[1] = 1;
        // 由于三个有序序列都是由「已有丑数」*「质因数」而来
        // i2、i3 和 i5 分别代表三个有序序列当前使用到哪一位「已有丑数」下标（起始都指向 1）
        for (int i2 = 1, i3 = 1, i5 = 1, idx = 2; idx <= n; idx++) {
            // 由 ans[iX] * X 可得当前有序序列指向哪一位
            int a = ans[i2] * 2, b = ans[i3] * 3, c = ans[i5] * 5;
            // 将三个有序序列中的最小一位存入「已有丑数」序列，并将其下标后移
            int min = Math.min(a, Math.min(b, c));
            // 由于可能不同有序序列之间产生相同丑数，因此只要一样的丑数就跳过（不能使用 else if ）
            if (min == a) i2++;
            if (min == b) i3++;
            if (min == c) i5++;
            ans[idx] = min;
        }
        return ans[n];
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **295. 数据流的中位数** ，难度为 **困难**。

Tag：「优先队列」、「堆」

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。
  示例：

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

进阶:

- 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？
- 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

数据结构运用

这是一道经典的数据结构运用题。

具体的，**我们可以使用两个优先队列（堆）来维护整个数据流数据，令维护数据流左半边数据的优先队列（堆）为 `l`，维护数据流右半边数据的优先队列（堆）为 `r`。**

显然，**为了可以在 $O(1)$ 的复杂度内取得当前中位数，我们应当令 `l` 为大根堆，`r` 为小根堆，并人为固定 `l` 和 `r` 之前存在如下的大小关系：**

- **当数据流元素数量为偶数：`l` 和 `r` 大小相同，此时动态中位数为两者堆顶元素的平均值；**

- **当数据流元素数量为奇数：`l` 比 `r` 多一，此时动态中位数为 `l` 的堆顶原数。**

为了满足上述说的奇偶性堆大小关系，在进行 `addNum` 时，我们应当分情况处理：

- 插入前两者大小相同，说明插入前数据流元素个数为偶数，插入后变为奇数。我们期望操作完达到「`l` 的数量为 `r` 多一，同时双堆维持有序」，进一步分情况讨论：
    - 如果 `r` 为空，说明当前插入的是首个元素，直接添加到 `l` 即可；
    - 如果 `r` 不为空，且 `num <= r.peek()`，说明 `num` 的插入位置不会在后半部分（不会在 `r` 中），直接加到 `l` 即可；
    - 如果 `r` 不为空，且 `num > r.peek()`，说明 `num` 的插入位置在后半部分，此时将 `r` 的堆顶元素放到 `l` 中，再把 `num` 放到 `r`（相当于从 `r` 中置换一位出来放到 `l` 中）。
- 插入前两者大小不同，说明前数据流元素个数为奇数，插入后变为偶数。我们期望操作完达到「`l` 和 `r` 数量相等，同时双堆维持有序」，进一步分情况讨论（此时 `l` 必然比 `r` 元素多一）：
    - 如果 `num >= l.peek()`，说明 `num` 的插入位置不会在前半部分（不会在 `l` 中），直接添加到 `r` 即可。
    - 如果 `num < l.peek()`，说明 `num` 的插入位置在前半部分，此时将 `l` 的堆顶元素放到 `r` 中，再把 `num` 放入 `l` 中（相等于从 `l` 中替换一位出来当到 `r` 中）。

代码：

```java
class MedianFinder {
    PriorityQueue<Integer> l = new PriorityQueue<>((a,b)->b-a);
    PriorityQueue<Integer> r = new PriorityQueue<>((a,b)->a-b);

    public void addNum(int num) {
        int s1 = l.size(), s2 = r.size();
        if (s1 == s2) {
            if (r.isEmpty() || num <= r.peek()) {
                l.add(num);
            } else {
                l.add(r.poll());
                r.add(num);
            }
        } else {
            if (l.peek() <= num) {
                r.add(num);
            } else {
                r.add(l.poll());
                l.add(num);
            }
        }
    }

    public double findMedian() {
        int s1 = l.size(), s2 = r.size();
        if (s1 == s2) {
            return (l.peek() + r.peek()) / 2.0;
        } else {
            return l.peek();
        }
    }
}
```

- 时间复杂度：`addNum` 函数的复杂度为 $O(\log n)$；`findMedian` 函数的复杂度为 $O(1)$
- 空间复杂度：$O(n)$

进阶

- 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？

可以使用建立长度为 101 的桶，每个桶分别统计每个数的出现次数，同时记录数据流中总的元

素数量，每次查找中位数时，先计算出中位数是第几位，从前往后扫描所有的桶得到答案。

这种做法相比于对顶堆做法，计算量上没有优势，更多的是空间上的优化。

对顶堆解法两个操作中耗时操作复杂度为 $O(\log n)$，$\log$ 操作常数不会超过 3，在极限数据 10^7 情况下计算量仍然低于耗时操作复杂度为 $O(C)$（C 固定为 101）桶计数解法。

- 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

和上一问解法类似，对于 1% 采用哨兵机制进行解决即可，在常规的最小桶和最大桶两侧分别维护一个有序序列，即建立一个代表负无穷和正无穷的桶。

上述两个进阶问题的代码如下，但注意由于真实样例的数据分布不是进阶所描述的那样（不是绝大多数都在 $[0, 100]$ 范围内），所以会 TLE。

代码：

```java
class MedianFinder {

    TreeMap<Integer, Integer> head = new TreeMap<>(), tail = new TreeMap<>();
    int[] arr = new int[101];
    int a, b, c;

    public void addNum(int num) {
        if (num >= 0 && num <= 100) {
            arr[num]++;
            b++;
        } else if (num < 0) {
            head.put(num, head.getOrDefault(num, 0) + 1);
            a++;
        } else if (num > 100) {
            tail.put(num, tail.getOrDefault(num, 0) + 1);
            c++;
        }
    }

    public double findMedian() {
        int size = a + b + c;
        if (size % 2 == 0) return (find(size / 2) + find(size / 2 + 1)) / 2.0;
        return find(size / 2 + 1);
    }

    int find(int n) {
        if (n <= a) {
            for (int num : head.keySet()) {
                n -= head.get(num);
                if (n <= 0) return num;
            }
        } else if (n <= a + b) {
            n -= a;
            for (int i = 0; i <= 100; i++) {
                n -= arr[i];
                if (n <= 0) return i;
            }
        } else {
            n -= a + b;
            for (int num : tail.keySet()) {
                n -= tail.get(num);
                if (n <= 0) return num;
            }
        }
        return -1; // never
    }
```

```
}
```

> **🍭 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 **313. 超级丑数** ，难度为 **中等**。

Tag :「优先队列」、「多路归并」

超级丑数 是一个正整数，并满足其所有质因数都出现在质数数组 primes 中。

给你一个整数 n 和一个整数数组 primes ，返回第 n 个 超级丑数 。

题目数据保证第 n 个 超级丑数 在 32-bit 带符号整数范围内。

示例 1：

```
输入：n = 12, primes = [2,7,13,19]

输出：32

解释：给定长度为 4 的质数数组 primes = [2,7,13,19]，前 12 个超级丑数序列为：[1,2,4,7,8,13,14,16,19,26,28,32] 。
```

示例 2：

```
输入：n = 1, primes = [2,3,5]

输出：1

解释：1 不含质因数，因此它的所有质因数都在质数数组 primes = [2,3,5] 中。
```

提示：

- 1 <= n <= 10^6
- 1 <= primes.length <= 100
- 2 <= primes[i] <= 1000
- **题目数据 保证 primes[i] 是一个质数**

- primes 中的所有值都 互不相同 ，且按 递增顺序 排列

基本分析

类似的题目在之前的每日一题也出现过。

本题做法与 264. 丑数 II 类似，相关题解在 这里。

回到本题，根据丑数的定义，我们有如下结论：

- 1 是最小的丑数。
- 对于任意一个丑数 x，其与任意给定的质因数 $primes[i]$ 相乘，结果仍为丑数。

优先队列（堆）

有了基本的分析思路，一个简单的解法是使用优先队列：

1. 起始先将最小丑数 1 放入队列
2. 每次从队列取出最小值 x，然后将 x 所对应的丑数 $x * primes[i]$ 进行入队。
3. 对步骤 2 循环多次，第 n 次出队的值即是答案。

为了防止同一丑数多次进队，我们需要使用数据结构 Set 来记录入过队列的丑数。

代码：

```Java
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        Set<Long> set = new HashSet<>();
        PriorityQueue<Long> q = new PriorityQueue<>();
        q.add(1L);
        set.add(1L);
        while (n-- > 0) {
            long x = q.poll();
            if (n == 0) return (int)x;
            for (int k : primes) {
                if (!set.contains(k * x)) {
                    set.add(k * x);
                    q.add(k * x);
                }
            }
        }
        return -1; // never
    }
}
```

- 时间复杂度：令 $primes$ 长度为 m，需要从优先队列（堆）中弹出 n 个元素，每次弹出最多需要放入 m 个元素，堆中最多有 $n * m$ 个元素。复杂度为 $O(n * m \log (n * m))$
- 空间复杂度：$O(n * m)$

多路归并

从解法一中不难发现，我们「往后产生的丑数」都是基于「已有丑数」而来（使用「已有丑数」乘上「给定质因数」$primes[i]$）。

因此，如果我们所有丑数的有序序列为 $a1, a2, a3, ..., an$ 的话，序列中的每一个数都必然能够被以下三个序列（中的至少一个）覆盖（这里假设 $primes = [2, 3, 5]$）：

- 由丑数 * 2 所得的有序序列：$1 * 2$、$2 * 2$、$3 * 2$、$4 * 2$、$5 * 2$、$6 * 2$、$8 * 2$ ...
- 由丑数 * 3 所得的有序序列：$1 * 3$、$2 * 3$、$3 * 3$、$4 * 3$、$5 * 3$、$6 * 3$、$8 * 3$ ...
- 由丑数 * 5 所得的有序序列：$1 * 5$、$2 * 5$、$3 * 5$、$4 * 5$、$5 * 5$、$6 * 5$、$8 * 5$ ...

我们令这些有序序列为 arr，最终的丑数序列为 ans。

如果 $primes$ 的长度为 m 的话，我们可以使用 m 个指针来指向这 m 个有序序列 arr 的当前

下标。

显然，我们需要每次取 m 个指针中值最小的一个，然后让指针后移（即将当前序列的下一个值放入堆中），不断重复这个过程，直到我们找到第 n 个丑数。

当然，实现上，我们并不需要构造出这 m 个有序序列。

我们可以构造一个存储三元组的小根堆，三元组信息为 (val, i, idx)：

- val ：为当前列表指针指向具体值；
- i ：代表这是由 $primes[i]$ 构造出来的有序序列；
- idx：代表丑数下标，存在关系 $val = ans[idx] * primes[i]$。

起始时，我们将所有的 $(primes[i], i, 0)$ 加入优先队列（堆）中，每次从堆中取出最小元素，那么下一个该放入的元素为 $(ans[idx + 1] * primes[i], i, idx + 1)$。

另外，由于我们每个 arr 的指针移动和 ans 的构造，都是单调递增，因此我们可以通过与当前最后一位构造的 $ans[x]$ 进行比较来实现去重，而无须引用常数较大的 `Set` 结构。

代码：

```
class Solution {
    public int nthSuperUglyNumber(int n, int[] primes) {
        int m = primes.length;
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->a[0]-b[0]);
        for (int i = 0; i < m; i++) {
            q.add(new int[]{primes[i], i, 0});
        }
        int[] ans = new int[n];
        ans[0] = 1;
        for (int j = 1; j < n; ) {
            int[] poll = q.poll();
            int val = poll[0], i = poll[1], idx = poll[2];
            if (val != ans[j - 1]) ans[j++] = val;
            q.add(new int[]{ans[idx + 1] * primes[i], i, idx + 1});
        }
        return ans[n - 1];
    }
}
```

- 时间复杂度：需要构造长度为 n 的答案，每次构造需要往堆中取出和放入元素，堆中有 m 个元素，起始时，需要对 $primes$ 进行遍历，复杂度为 $O(m)$。整体复杂

度为 $O(\max(m, n\log m))$

- 空间复杂度：存储 n 个答案，堆中有 m 个元素，复杂度为 $O(n+m)$

> **🍭 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 **451. 根据字符出现频率排序** ，难度为 **中等**。

Tag：「模拟」、「桶排序」、「哈希表」、「数组」、「优先队列（堆）」

给定一个字符串，请将字符串里的字符按照出现的频率降序排列。

示例 1:

```
输入:
"tree"

输出:
"eert"

解释:
'e'出现两次，'r'和't'都只出现一次。
因此'e'必须出现在'r'和't'之前。此外，"eetr"也是一个有效的答案。
```

示例 2:

```
输入:
"cccaaa"

输出:
"cccaaa"

解释:
'c'和'a'都出现三次。此外，"aaaccc"也是有效的答案。
注意"cacaca"是不正确的，因为相同的字母必须放在一起。
```

示例 3:

输入：
"Aabb"

输出：

"bbAa"

解释：
此外，"bbaA"也是一个有效的答案，但"Aabb"是不正确的。
注意'A'和'a'被认为是两种不同的字符。

数据结构 + 模拟

这是一道考察数据结构运用的模拟题。

具体做法如下：

1. 先使用「哈希表」对词频进行统计；
2. 遍历统计好词频的哈希表，将每个键值对以 `{字符,词频}` 的形式存储到「优先队列（堆）」中。并规定「优先队列（堆）」排序逻辑为：
    - 如果 `词频` 不同，则按照 `词频` 倒序；
    - 如果 `词频` 相同，则根据 `字符字典序` 升序（由于本题采用 Special Judge 机制，这个排序策略随意调整也可以。但通常为了确保排序逻辑满足「全序关系」，这个地方可以写正写反，但理论上不能不写，否则不能确保每次排序结果相同）；
3. 从「优先队列（堆）」依次弹出，构造答案。

代码：

```Java
class Solution {
    class Node {
        char c;
        int v;
        Node(char _c, int _v) {
            c = _c; v = _v;
        }
    }
    public String frequencySort(String s) {
        char[] cs = s.toCharArray();
        Map<Character, Integer> map = new HashMap<>();
        for (char c : cs) {
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->{
            if (b.v != a.v) return b.v - a.v;
            return a.c - b.c;
        });
        for (char c : map.keySet()) {
            q.add(new Node(c, map.get(c)));
        }
        StringBuilder sb = new StringBuilder();
        while (!q.isEmpty()) {
            Node poll = q.poll();
            int k = poll.v;
            while (k-- > 0) sb.append(poll.c);
        }
        return sb.toString();
    }
}
```

- 时间复杂度：令字符集的大小为 C。使用「哈希表」统计词频的复杂度为 $O(n)$；最坏情况下字符集中的所有字符都有出现，最多有 C 个节点要添加到「优先队列（堆）」中，复杂度为 $O(C\log C)$；构造答案需要从「优先队列（堆）」中取出元素并拼接，复杂度为 $O(n)$。整体复杂度为 $O(\max(n, C\log C))$
- 空间复杂度：$O(n)$

数组实现 + 模拟

基本思路不变，将上述过程所用到的数据结构使用数组替代。

具体的，利用 ASCII 字符集共 128 位，预先建立一个大小为 128 的数组，利用「桶排序」的思路替代「哈希表」和「优先队列（堆）」的作用。

代码：

```
class Solution {
    public String frequencySort(String s) {
        int[][] cnts = new int[128][2];
        char[] cs = s.toCharArray();
        for (int i = 0; i < 128; i++) cnts[i][0] = i;
        for (char c : cs) cnts[c][1]++;
        Arrays.sort(cnts, (a, b)->{
            if (a[1] != b[1]) return b[1] - a[1];
            return a[0] - b[0];
        });
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 128; i++) {
            char c = (char)cnts[i][0];
            int k = cnts[i][1];
            while (k-- > 0) sb.append(c);
        }
        return sb.toString();
    }
}
```

- 时间复杂度：令字符集的大小为 C。复杂度为 $O(\max(n, C\log C))$
- 空间复杂度：$O(n + C + \log C)$

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

题目描述

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

Tag：「滑动窗口」、「堆」、「优先队列」

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

例如：

- [2,3,4]，中位数是 3
- [2,3]，中位数是 (2 + 3) / 2 = 2.5

给你一个数组 nums，有一个长度为 k 的窗口从最左端滑动到最右端。

窗口中有 k 个数，每次窗口向右移动 1 位。

你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

示例：

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

```
窗口位置                      中位数
---------------               -----
[1  3  -1] -3  5  3  6  7       1
 1 [3  -1  -3] 5  3  6  7      -1
 1  3 [-1  -3  5] 3  6  7      -1
 1  3  -1 [-3  5  3] 6  7       3
 1  3  -1  -3 [5  3  6] 7       5
 1  3  -1  -3  5 [3  6  7]      6
```

因此，返回该滑动窗口的中位数数组 `[1,-1,-1,3,5,6]`。

提示：

- 你可以假设 k 始终有效，即：k 始终小于等于输入的非空数组的元素个数。

- 与真实值误差在 10^{-5} 以内的答案将被视作正确答案。

朴素解法

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

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

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

PS. 实际上这道题朴素解法是可以过的，有蓝桥杯内味了 ~

朴素做法通常是优化的开始，所以我还是提供一下朴素做法的代码

代码：

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

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

优先队列（堆）解法

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

我们知道滑动窗口求最值的问题，可以使用优先队列来做。

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

我们可以维护两个堆：

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

滑动窗口的中位数就是两个堆的堆顶元素的平均值。

实现细节：

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

代码：

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

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

注意点 (2021-02-04 更新)

今天的题解发到 LeetCode 后，针对一些同学的评论。

我觉得有一定的代表性，所以拿出来讲讲 ~

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

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

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

注意点 (2021-02-05 更新)

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

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

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

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

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

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

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

题目描述

这是 LeetCode 上的 **502. IPO** ，难度为 **困难**。

Tag：「贪心」、「优先队列」、「堆」

假设 力扣（LeetCode）即将开始 IPO 。

为了以更高的价格将股票卖给风险投资公司，力扣 希望在 IPO 之前开展一些项目以增加其资本。

由于资源有限，它只能在 IPO 之前完成最多 k 个不同的项目。

帮助 力扣 设计完成最多 k 个不同项目后得到最大总资本的方式。

给你 n 个项目。对于每个项目 i ，它都有一个纯利润 profits[i] ，和启动该项目需要的最小资本 capital[i] 。

最初，你的资本为 w 。当你完成一个项目时，你将获得纯利润，且利润将被添加到你的总资本中。

总而言之，从给定项目中选择 最多 k 个不同项目的列表，以 最大化最终资本 ，并输出最终可获得的最多资本。

答案保证在 32 位有符号整数范围内。

示例 1：

```
输入：k = 2, w = 0, profits = [1,2,3], capital = [0,1,1]

输出：4

解释：
由于你的初始资本为 0，你仅可以从 0 号项目开始。
在完成后，你将获得 1 的利润，你的总资本将变为 1。
此时你可以选择开始 1 号或 2 号项目。
由于你最多可以选择两个项目，所以你需要完成 2 号项目以获得最大的资本。
因此，输出最后最大化的资本，为 0 + 1 + 3 = 4。
```

示例 2：

```
输入：k = 3, w = 0, profits = [1,2,3], capital = [0,1,2]

输出：6
```

提示：

- 1 <= k <= 10^5
- 0 <= w <= 10^9
- n == profits.length
- n == capital.length
- 1 <= n <= 10^5
- 0 <= profits[i] <= 10^4
- 0 <= capital[i] <= 10^9

贪心 + 优先队列（堆）

由于每完成一个任务都会使得总资金 `w` 增加或不变。因此对于所选的第 i 个任务而言，应该在所有「未被选择」且启动资金不超过 `w` 的所有任务里面选利润最大的。

可通过「归纳法」证明每次都在所有候选中选择利润最大的任务，可使得总资金最大。

对于第 i 次选择而言（当前所有的资金为 w），如果选择的任务利润为 cur，而实际可选的最大任务利润为 max（ $cur <= max$ ）。

将「选择 cur」调整为「选择 max」，结果不会变差：

1. 根据传递性，由 $cur <= max$ 可得 $w + cur <= w + max$，可推导出调整后的总资金不会变少；
2. 利用推论 1，由于总资金相比调整前没有变少，因此后面可选择的任务集合也不会变少。这意味着 **至少可以维持** 第 i 次选择之后的所有原有选择。

至此，我们证明了将每次的选择调整为选择最大利润的任务，结果不会变差。

当知道了「每次都应该在所有可选择的任务里选利润最大」的推论之后，再看看算法的具体流程。

由于每完成一个任务总资金都会 增大/不变，因此所能覆盖的任务集合数量也随之 增加/不变 。

因此算法核心为「**每次决策前，将启动资金不超过当前总资金的任务加入集合，再在里面取利润最大的任务**」。

「取最大」的过程可以使用优先队列（根据利润排序的大根堆），而「将启动资金不超过当前总资金的任务加入集合」的操作，可以利用总资金在整个处理过程递增，而先对所有任务进行预处理排序来实现。

具体的，我们可以按照如下流程求解：

1. 根据 `profits` 和 `capital` 预处理出总的任务集合二元组，并根据「启动资金」进行升序排序；
2. 每次决策前，将所有的启动资金不超过 w 的任务加入优先队列（根据利润排序的大根堆），然后从优先队列（根据利润排序的大根堆），将利润累加到 w；
3. 循环步骤 2，直到达到 k 个任务，或者队列为空（当前资金不足以选任何任务）。

代码：

```Java
class Solution {
    public int findMaximizedCapital(int k, int w, int[] profits, int[] capital) {
        int n = profits.length;
        List<int[]> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(new int[]{capital[i], profits[i]});
        }
        Collections.sort(list, (a,b)->a[0]-b[0]);
        PriorityQueue<Integer> q = new PriorityQueue<>((a,b)->b-a);
        int i = 0;
        while (k-- > 0) {
            while (i < n && list.get(i)[0] <= w) q.add(list.get(i++)[1]);
            if (q.isEmpty()) break;
            w += q.poll();
        }
        return w;
    }
}
```

- 时间复杂度：构造出二元组数组并排序的复杂度为 $O(n \log n)$；大根堆最多有 n 个元素，使用大根堆计算答案的复杂度为 $O(k \log n)$。整体复杂度为 $O(\max(n \log n, k \log n))$
- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **692. 前K个高频单词** ，难度为 **中等**。

Tag：「哈希表」、「优先队列」

给一非空的单词列表，返回前 k 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。

如果不同的单词有相同出现频率，按字母顺序排序。

示例 1：

```
输入: ["i", "love", "leetcode", "i", "love", "coding"], k = 2

输出: ["i", "love"]

解析: "i" 和 "love" 为出现次数最多的两个单词，均为2次。
    注意，按字母顺序 "i" 在 "love" 之前。
```

示例 2：

```
输入: ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4

输出: ["the", "is", "sunny", "day"]

解析: "the", "is", "sunny" 和 "day" 是出现次数最多的四个单词，
    出现次数依次为 4, 3, 2 和 1 次。
```

注意：

- **假定 k 总为有效值， 1 ≤ k ≤ 集合元素数。**
- **输入的单词均由小写字母组成。**

扩展练习：

- **尝试以 O(n log k) 时间复杂度和 O(n) 空间复杂度解决。**

哈希表 & 优先队列（堆）

这道题是在「优先队列（堆）」裸题的基础上增加了字典序大小的比较。

相应的，我们不能只根据「词频大小」构建小根堆来获取前 k 个元素，还需要结合字典序大小来做。

具体的，我们可以使用「哈希表」&「优先队列」进行求解：

1. **使用「哈希表」来统计所有的词频**
2. **构建大小为 k 按照「词频升序 + (词频相同)字典序倒序」的优先队列：**
    - **如果词频不相等，根据词频进行升序构建，确保堆顶元素是堆中词频最小的元素**

- 如果词频相等，根据字典序大小进行倒序构建，结合 2.1 可以确保堆顶元素是堆中「词频最小 & 字典序最大」的元素

3. 对所有元素进行遍历，尝试入堆：
    - 堆内元素不足 k 个：直接入堆
    - 词频大于堆顶元素：堆顶元素不可能是前 k 大的元素。将堆顶元素弹出，并将当前元素添加到堆中
    - 词频小于堆顶元素；当前元素不可能是前 k 大的元素，直接丢弃。
    - 词频等于堆顶元素：根据当前元素与堆顶元素的字典序大小决定（如果字典序大小比堆顶元素要小则入堆）
4. 输出堆内元素，并翻转

代码：

```java
class Solution {
    public List<String> topKFrequent(String[] ws, int k) {
        Map<String, Integer> map = new HashMap<>();
        for (String w : ws) map.put(w, map.getOrDefault(w, 0) + 1);
        PriorityQueue<Object[]> q = new PriorityQueue<>(k, (a, b)->{
            // 如果词频不同，根据词频升序
            int c1 = (Integer)a[0], c2 = (Integer)b[0];
            if (c1 != c2) return c1 - c2;
            // 如果词频相同，根据字典序倒序
            String s1 = (String)a[1], s2 = (String)b[1];
            return s2.compareTo(s1);
        });
        for (String s : map.keySet()) {
            int cnt = map.get(s);
            if (q.size() < k) { // 不足 k 个，直接入堆
                q.add(new Object[]{cnt, s});
            } else {
                Object[] peek = q.peek();
                if (cnt > (Integer)peek[0]) { // 词频比堆顶元素大，弹出堆顶元素，入堆
                    q.poll();
                    q.add(new Object[]{cnt, s});
                } else if (cnt == (Integer)peek[0]) { // 词频与堆顶元素相同
                    String top = (String)peek[1];
                    if (s.compareTo(top) < 0) { // 且字典序大小比堆顶元素小，弹出堆顶元素，入堆
                        q.poll();
                        q.add(new Object[]{cnt, s});
                    }
                }
            }
        }
        List<String> ans = new ArrayList<>();
        while (!q.isEmpty()) ans.add((String)q.poll()[1]);
        Collections.reverse(ans);
        return ans;
    }
}
```

- 时间复杂度：使用哈希表统计词频，复杂度为 $O(n)$；使用最多 n 个元素维护一个大小为 k 的堆，复杂度为 $O(n\log k)$；输出答案复杂度为 $O(k)$（同时 $k \leq n$）。整体复杂度为 $O(n\log k)$
- 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 **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：

```
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：

```
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：$O(k)$

题目描述

这是 LeetCode 上的 **726. 原子的数量** ，难度为 **困难**。

Tag：「模拟」、「数据结构运用」、「栈」、「哈希表」、「优先队列」

给定一个化学式 `formula`（作为字符串），返回每种原子的数量。

原子总是以一个大写字母开始，接着跟随0个或任意个小写字母，表示原子的名字。

如果数量大于 1，原子后会跟着数字表示原子的数量。如果数量等于 1 则不会跟数字。例如，`H2O` 和 `H2O2` 是可行的，但 `H1O2` 这个表达是不可行的。

两个化学式连在一起是新的化学式。例如 `H2O2He3Mg4` 也是化学式。

一个括号中的化学式和数字（可选择性添加）也是化学式。例如 `(H2O2)` 和 `(H2O2)3` 是化学式。

给定一个化学式，输出所有原子的数量。格式为：第一个（按字典序）原子的名子，跟着它的数量（如果数量大于 1），然后是第二个原子的名字（按字典序），跟着它的数量（如果数量大于 1），以此类推。

示例 1:

```
输入: formula = "H2O"

输出: "H2O"

解释: 原子的数量是 {'H': 2, 'O': 1}。
```

示例 2:

```
输入: formula = "Mg(OH)2"

输出: "H2MgO2"

解释: 原子的数量是 {'H': 2, 'Mg': 1, 'O': 2}。
```

示例 3:

```
输入: formula = "K4(ON(SO3)2)2"

输出: "K4N2O14S4"

解释: 原子的数量是 {'K': 4, 'N': 2, 'O': 14, 'S': 4}。
```

注意:

- 所有原子的第一个字母为大写，剩余字母都是小写。
- `formula` 的长度在[1, 1000]之间。
- `formula` 只包含字母、数字和圆括号，并且题目中给定的是合法的化学式。

数据结构 + 模拟

一道综合模拟题。

相比于（题解）227. 基本计算器 II 的表达式计算问题，本题设计模拟流程的难度要低很多，之所谓定位困难估计是使用到的数据结构较多一些。

为了方便，我们约定以下命名：

- 称一段完整的连续字母为「原子」
- 称一段完整的连续数字为「数值」
- 称 `(` 和 `)` 为「符号」

基本实现思路如下：

1. 在处理入参 `s` 的过程中，始终维护着一个哈希表 `map`，`map` 中 **实时维护** 着某个「原子」对应的实际「数值」（即存储格式为 `{H:2,S:1}`）；
   > 由于相同原子可以出在 `s` 的不同位置中，为了某个「数值」对「原子」的累乘效果被重复应用，我们这里应用一个”小技巧“：为每个「原子」增加一个”编号后缀“。即实际存储时为 `{H_1:2, S_2:1, H_3:1}`。
2. 根据当前处理到的字符分情况讨论：
   - 符号：直接入栈；
   - 原子：继续往后取，直到取得完整的原子名称，将完整原子名称入栈，同时在 `map` 中计数加 1；
   - 数值：继续往后取，直到取得完整的数值并解析，然后根据栈顶元素是否为 `)` 符号，决定该数值应用给哪些原子：
     - 如果栈顶元素不为 `)`，说明该数值只能应用给栈顶的原子
     - 如果栈顶元素是)，说明当前数值可以应用给「连续一段」的原子中
3. 对 `map` 的原子做 “合并” 操作：`{H_1:2, S_2:1, H_3:1}` => `{H:3, S:1}`；
4. 使用「优先队列（堆）」实现字典序排序（也可以直接使用 `List`，然后通过 `Collections.sort` 进行排序），并构造答案。

代码：

```Java
class Solution {
    class Node {
        String s; int v;
        Node (String _s, int _v) {
            s = _s; v = _v;
        }
    }
    public String countOfAtoms(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        Map<String, Integer> map = new HashMap<>();
        Deque<String> d = new ArrayDeque<>();
        int idx = 0;
        for (int i = 0; i < n; ) {
            char c = cs[i];
            if (c == '(' || c == ')') {
                d.addLast(String.valueOf(c));
                i++;
            } else {
                if (Character.isDigit(c)) {
                    // 获取完整的数字，并解析出对应的数值
                    int j = i;
                    while (j < n && Character.isDigit(cs[j])) j++;
                    String numStr = s.substring(i, j);
                    i = j;
                    int cnt = Integer.parseInt(String.valueOf(numStr));

                    // 如果栈顶元素是 )，说明当前数值可以应用给「连续一段」的原子中
                    if (!d.isEmpty() && d.peekLast().equals(")")) {
                        List<String> tmp = new ArrayList<>();

                        d.pollLast(); // pop )
                        while (!d.isEmpty() && !d.peekLast().equals("(")) {
                            String cur = d.pollLast();
                            map.put(cur, map.getOrDefault(cur, 1) * cnt);
                            tmp.add(cur);
                        }
                        d.pollLast(); // pop (

                        for (int k = tmp.size() - 1; k >= 0; k--) {
                            d.addLast(tmp.get(k));
                        }

                    // 如果栈顶元素不是 )，说明当前数值只能应用给栈顶的原子
                    } else {
                        String cur = d.pollLast();
```

```
                    map.put(cur, map.getOrDefault(cur, 1) * cnt);
                    d.addLast(cur);
                }
            } else {
                // 获取完整的原子名
                int j = i + 1;
                while (j < n && Character.isLowerCase(cs[j])) j++;
                String cur = s.substring(i, j) + "_" + String.valueOf(idx++);
                map.put(cur, map.getOrDefault(cur, 0) + 1);
                i = j;
                d.addLast(cur);
            }
        }
    }

    // 将不同编号的相同原子进行合并
    Map<String, Node> mm = new HashMap<>();
    for (String key : map.keySet()) {
        String atom = key.split("_")[0];
        int cnt = map.get(key);
        Node node = null;
        if (mm.containsKey(atom)) {
            node = mm.get(atom);
        } else {
            node = new Node(atom, 0);
        }
        node.v += cnt;
        mm.put(atom, node);
    }

    // 使用优先队列（堆）对 Node 进行字典序排序，并构造答案
    PriorityQueue<Node> q = new PriorityQueue<Node>((a,b)->a.s.compareTo(b.s));
    for (String atom : mm.keySet()) q.add(mm.get(atom));

    StringBuilder sb = new StringBuilder();
    while (!q.isEmpty()) {
        Node poll = q.poll();
        sb.append(poll.s);
        if (poll.v > 1) sb.append(poll.v);
    }

    return sb.toString();
}
}
```

- 时间复杂度：最坏情况下，每次处理数值都需要从栈中取出元素进行应用，处理 `s`

的复杂度为 $O(n^2)$；最坏情况下，每个原子独立分布，合并的复杂度为 $O(n)$；将合并后的内容存入优先队列并取出构造答案的复杂度为 $O(n \log n)$；整体复杂度为 $O(n^2)$

- 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 **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」

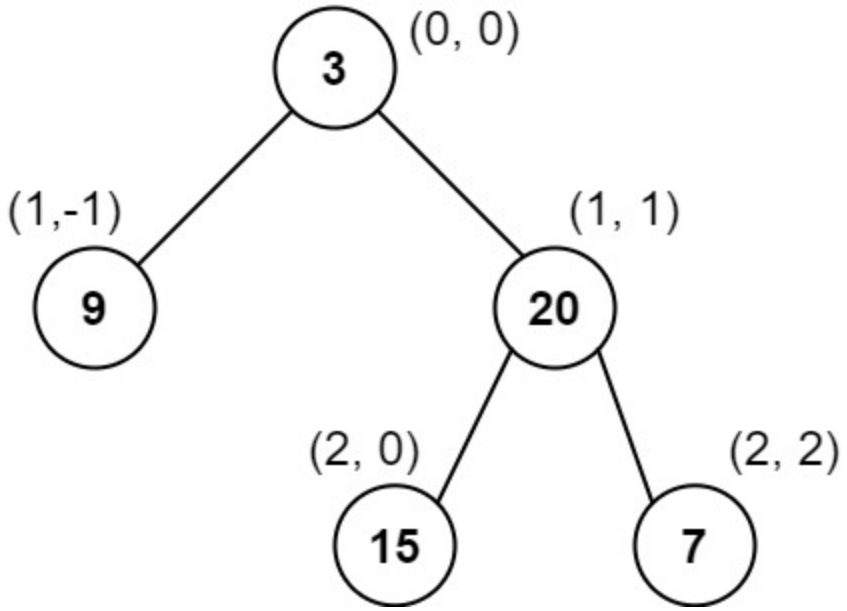
给你二叉树的根结点 root ，请你设计算法计算二叉树的 垂序遍历 序列。

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：

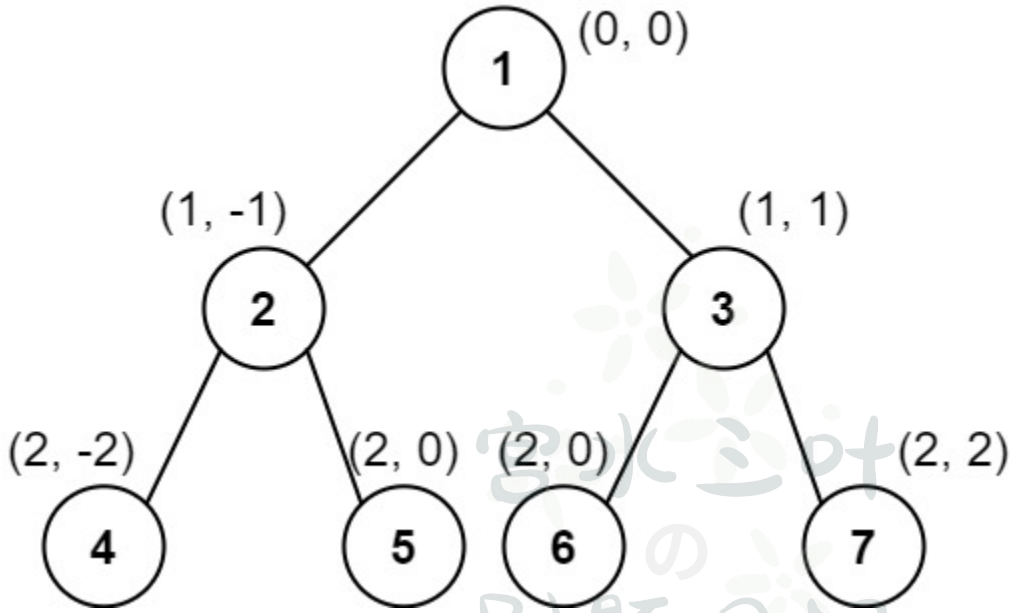


输入：root = [3,9,20,null,null,15,7]

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：

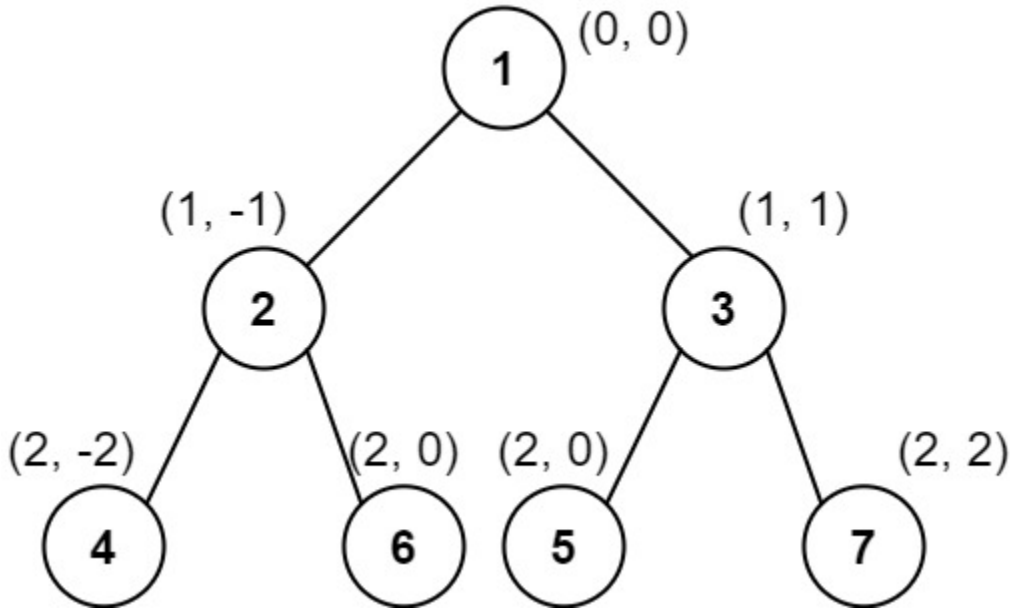


输入：root = [1,2,3,4,5,6,7]

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 **1337. 矩阵中战斗力最弱的 K 行** ，难度为 **简单**。

Tag：「优先队列」、「二分」

给你一个大小为 $m * n$ 的矩阵 mat，矩阵由若干军人和平民组成，分别用 1 和 0 表示。

请你返回矩阵中战斗力最弱的 k 行的索引，按从最弱到最强排序。

如果第 i 行的军人数量少于第 j 行，或者两行军人数量相同但 i 小于 j，那么我们认为第 i 行的战斗力比第 j 行弱。

军人 总是 排在一行中的靠前位置，也就是说 1 总是出现在 0 之前。

示例 1：

```
输入：mat =
[[1,1,0,0,0],
 [1,1,1,1,0],
 [1,0,0,0,0],
 [1,1,0,0,0],
 [1,1,1,1,1]],
k = 3

输出：[2,0,3]

解释：
每行中的军人数目：
行 0 -> 2
行 1 -> 4
行 2 -> 1
行 3 -> 2
行 4 -> 5
从最弱到最强对这些行排序后得到 [2,0,3,1,4]
```

示例 2：

```
输入：mat =
[[1,0,0,0],
 [1,1,1,1],
 [1,0,0,0],
 [1,0,0,0]],
k = 2

输出：[0,2]

解释：
每行中的军人数目：
行 0 -> 1
行 1 -> 4
行 2 -> 1
行 3 -> 1
从最弱到最强对这些行排序后得到 [0,2,3,1]
```

提示：

- m == mat.length
- n == mat[i].length
- 2 <= n, m <= 100
- 1 <= k <= m
- matrix[i][j] 不是 0 就是 1

朴素解法

一个朴素的做法是对矩阵进行遍历，统计每一行的军人数量，并以二元组 (cnt_i, idx_i) 的形式进行存储。

然后对所有行的战力进行排序，选出战力最小的 k 个下标即是答案。

代码：

```Java
class Solution {
    public int[] kWeakestRows(int[][] mat, int k) {
        int m = mat.length, n = mat[0].length;
        int[][] all = new int[m][1];
        for (int i = 0; i < m; i++) {
            int cur = 0;
            for (int j = 0; j < n; j++) cur += mat[i][j];
            all[i] = new int[]{cur, i};
        }
        Arrays.sort(all, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            return a[1] - b[1];
        });
        int[] ans = new int[k];
        for (int i = 0; i < k; i++) ans[i] = all[i][1];
        return ans;
    }
}
```

- 时间复杂度：遍历矩阵的复杂度为 $O(m * n)$；排序复杂度为 $O(m \log m)$；构造答案复杂度为 $O(k)$。整体复杂度为 $O(\max(m * n, m \log m))$
- 空间复杂度：$O(m)$ 空间用于存储所有的行战力；$O(\log m)$ 空间用于排序。整体复杂度为 $O(m + \log m)$

二分 + 优先队列（堆）

根据「军人总是排在靠前位置」的特性，我们可以通过「二分」找到每一行最后一个军人的位置，从而在 $O(\log n)$ 的复杂度内统计出每行的军人数量（战力情况）。

同时由于我们只需要「战力最弱」的 k 行数据，这引导我们可以建立一个「战力大根堆」来做，「战力大根堆」存放着数量最多为 k 的战力二元组 (cnt_i, idx_i)，堆顶元素为战力最大的数对。

每次通过「二分」得到当前行的战力值后，判断当前战力值与堆顶元素的战力大小关系：

- 如果当前战力值比堆顶的元素要大：直接丢弃当前战力值（不可能属于在第 k 小的集合中）；
- 如果当前战力值比堆顶的元素要小：将堆顶元素弹出，将当前行放入堆中。

> 一些细节：每次写二分都有同学会问， `check` 函数怎么写，可以重点看看 34 题题解。由于考虑某行没有军人的情况，我们需要二分完检查一下分割点是否符合 `check` 来决定军人数量。
> 另外，从堆弹出和往堆存入，需要与当前堆元素的数量有所关联。

代码：

```
class Solution {
    public int[] kWeakestRows(int[][] mat, int k) {
        int m = mat.length, n = mat[0].length;
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->{
            if (a[0] != b[0]) return b[0] - a[0];
            return b[1] - a[1];
        });
        for (int i = 0; i < m; i++) {
            int l = 0, r = n - 1;
            while (l < r) {
                int mid = l + r + 1 >> 1;
                if (mat[i][mid] >= 1) l = mid;
                else r = mid - 1;
            }
            int cur = mat[i][r] >= 1 ? r + 1 : r;
            if (q.size() == k && q.peek()[0] > cur) q.poll();
            if (q.size() < k) q.add(new int[]{cur, i});
        }
        int[] ans = new int[k];
        int idx = k - 1;
        while (!q.isEmpty()) ans[idx--] = q.poll()[1];
        return ans;
    }
}
```

- 时间复杂度：二分得到每行的战力情况，复杂度为 $O(m \log n)$；堆中最多有 k 个元素，将行信息存入堆中复杂度为 $O(m \log k)$；构造答案复杂度为 $O(k \log k)$。整体复杂度为 $O(m * (\log n + \log k))$
- 空间复杂度：$O(k)$

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：$O(n)$

接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：