z80160280 Update 0347.前K个高频元素.md History

4 contributors 

202 lines (144 sloc) | 8.04 KB

欢迎大家参与本项目，贡献其他语言版本的代码，拥抱开源，让更多学习算法的小伙伴们收益！

> 前K个大数问题，老生常谈，不得不谈

347.前 K 个高频元素

https://leetcode-cn.com/problems/top-k-frequent-elements/

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

示例 1: 输入: nums = [1,1,1,2,2,3], k = 2 输出: [1,2]

示例 2: 输入: nums = [1], k = 1 输出: [1]

提示： 你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。 你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。 题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。 你可以按任意顺序返回答案。

思路

这道题目主要涉及到如下三块内容：

1. 要统计元素出现频率
2. 对频率排序
3. 找出前K个高频元素

首先统计元素出现的频率，这一类的问题可以使用map来进行统计。

然后是对频率进行排序，这里我们可以使用一种 容器适配器就是**优先级队列**。

什么是优先级队列呢？

其实**就是一个披着队列外衣的堆**，因为优先级队列对外接口只是从队头取元素，从队尾添加元素，再无其他取元素的方式，看起来就是一个队列。

而且优先级队列内部元素是自动依照元素的权值排列。那么它是如何有序排列的呢？

缺省情况下priority_queue利用max-heap（大顶堆）完成对元素的排序，这个大顶堆是以vector为表现形式的complete binary tree（完全二叉树）。

什么是堆呢？

堆是一颗完全二叉树，树中每个结点的值都不小于（或不大于）其左右孩子的值。如果父亲结点是大于等于左右孩子就是大顶堆，小于等于左右孩子就是小顶堆。

所以大家经常说的大顶堆（堆头是最大元素），小顶堆（堆头是最小元素），如果懒得自己实现的话，就直接用priority_queue（优先级队列）就可以了，底层实现都是一样的，从小到大排就是小顶堆，从大到小排就是大顶堆。

本题我们就要使用优先级队列来对部分频率进行排序。

为什么不用快排呢， 使用快排要将map转换为vector的结构，然后对整个数组进行排序， 而这种场景下，我们其实只需要维护k个有序的序列就可以了，所以使用优先级队列是最优的。

此时要思考一下，是使用小顶堆呢，还是大顶堆？

有的同学一想，题目要求前 K 个高频元素，那么果断用大顶堆啊。

那么问题来了，定义一个大小为k的大顶堆，在每次移动更新大顶堆的时候，每次弹出都把最大的元素弹出去了，那么怎么保留下来前K个高频元素呢。

所以我们要用小顶堆，因为要统计最大前k个元素，只有小顶堆每次将最小的元素弹出，最后小顶堆里积累的才是前k个最大元素。

寻找前k个最大元素流程如图所示：（图中的频率只有三个，所以正好构成一个大小为3的小顶堆，如果频率更多一些，则用这个小顶堆进行扫描）

我们来看一下C++代码：

```
// 时间复杂度：O(nlogk)
// 空间复杂度：O(n)
class Solution {
public:
    // 小顶堆
    class mycomparison {
    public:
        bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs) {
            return lhs.second > rhs.second;
        }
    };
```

```
    vector<int> topKFrequent(vector<int>& nums, int k) {
        // 要统计元素出现频率
        unordered_map<int, int> map; // map<nums[i],对应出现的次数>
        for (int i = 0; i < nums.size(); i++) {
            map[nums[i]]++;
        }
                                                ???
        // 对频率排序
        // 定义一个小顶堆，大小为k
        priority_queue<pair<int, int>, vector<pair<int, int>>, mycomparison> pri_que;

        // 用固定大小为k的小顶堆，扫面所有频率的数值
        for (unordered_map<int, int>::iterator it = map.begin(); it != map.end(); it++) {
            pri_que.push(*it);
            if (pri_que.size() > k) { // 如果堆的大小大于了K，则队列弹出，保证堆的大小一直为k
                pri_que.pop();
            }
        }

        // 找出前K个高频元素，因为小顶堆先弹出的是最小的，所以倒叙来输出到数组
        vector<int> result(k);
        for (int i = k - 1; i >= 0; i--) {
            result[i] = pri_que.top().first;
            pri_que.pop();
        }
        return result;

    }
};
```

拓展

大家对这个比较运算在建堆时是如何应用的，为什么左大于右就会建立小顶堆，反而建立大顶堆比较困惑。

确实 例如我们在写快排的cmp函数的时候，`return left>right` 就是从大到小，`return left<right` 就是从小到大。

优先级队列的定义正好反过来了，可能和优先级队列的源码实现有关（我没有仔细研究），我估计是底层实现上优先队列队首指向后面，队尾指向最前面的缘故！

其他语言版本

Java：

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        int[] result = new int[k];
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int num : nums) {
            map.put(num, map.getOrDefault(num, 0) + 1);
        }
```

```
        Set<Map.Entry<Integer, Integer>> entries = map.entrySet();
        // 根据map的value值正序排，相当于一个小顶堆
        PriorityQueue<Map.Entry<Integer, Integer>> queue = new PriorityQueue<>((o1, o2) ->
        for (Map.Entry<Integer, Integer> entry : entries) {
            queue.offer(entry);
            if (queue.size() > k) {
                queue.poll();
            }
        }
        for (int i = k - 1; i >= 0; i--) {
            result[i] = queue.poll().getKey();
        }
        return result;
    }
}
```

Python：

```
#时间复杂度：O(nlogk)
#空间复杂度：O(n)
import heapq
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        #要统计元素出现频率
        map_ = {} #nums[i]:对应出现的次数
        for i in range(len(nums)):
            map_[nums[i]] = map_.get(nums[i], 0) + 1

        #对频率排序
        #定义一个小顶堆，大小为k
        pri_que = [] #小顶堆

        #用固定大小为k的小顶堆，扫面所有频率的数值
        for key, freq in map_.items():
            heapq.heappush(pri_que, (freq, key))
            if len(pri_que) > k: #如果堆的大小大于了K，则队列弹出，保证堆的大小一直为k
                heapq.heappop(pri_que)

        #找出前K个高频元素，因为小顶堆先弹出的是最小的，所以倒叙来输出到数组
        result = [0] * k
        for i in range(k-1, -1, -1):
            result[i] = heapq.heappop(pri_que)[1]
        return result
```

Go：
