题目地址(347. 前 K 个高频元素)

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

入选理由

- 统计频率是哈希表的一个应用。当然如果数据范围小，可以考虑使用数组，理论性能更好（复杂度不变）
- 推荐大家和今天的力扣每日一题结合练习。https://leetcode-cn.com/problems/degree-of-an-array/ 那道题考察了一个哈希表记录最近和最远位置的点，这个考点也很常见。

题目描述

给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按 任意顺序 返回答案。

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:

```
1 <= nums.length <= 10^5
k 的取值范围是 [1, 数组中不相同的元素的个数]
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
```

进阶：你所设计算法的时间复杂度 必须 优于 O(n log n) ，其中 n 是数组大小。

标签

- 堆
- 排序
- 哈希表

难度

- 中等

前置知识

- 哈希表
- 堆排序
- 快速选择

思考

直接根据题意，可以把问题化解两个小问题

- 计算每个数的频次
- 在生成的频次里取第 K 大的

计算每个数的频次的话，我们可以采用哈希表, key 为列表的数，value 为出现的频次。这在讲义中有过介绍。

生成的频次里取第 K 大的，也就是我们熟悉 TOP K 问题。一般来说，我们可以通过以下方式求取：

1. 排序后通过索引直接取第 K 大的，比如从大到小排序后取索引为 k - 1 的。
2. 建堆。利用最小堆每次都能取最小的特性，取 k 次最小即可
3. 快速选择。

下面我们分别来介绍。

哈希表+桶排序

思路

桶的 key 是值，value 是值等于 key 的列表。

1. 初始化一个桶 bucket，一个哈希表 counter 记录数值频次
2. 从最后一个桶开始遍历直到取出 K 个数

> 当然你也可以使用其他排序方法， 亦或者是直接调用系统的 sort 函数

```
class Solution {

    public int[] topKFrequent(int[] nums, int k) {

        List<Integer> res = new LinkedList<>();
        List<Integer>[] bucket = new List[nums.length + 1];
        Map<Integer, Integer> counter = new HashMap<>();

        for (int num: nums)
            counter.put(num, counter.getOrDefault(num, 0) + 1);

        for (Map.Entry<Integer, Integer> entry: counter.entrySet()) {
            int val = entry.getValue();
            if (bucket[val] == null)
                bucket[val] = new LinkedList<>();
            bucket[val].add(entry.getKey());
        }

        int kNum = 0;
        for (int i = bucket.length - 1; i >= 0; i--)
            if (bucket[i] != null)
                for (int elem: bucket[i]){

                    res.add(elem);
                    kNum++;
                }

        int[] ret = new int[k];
        for (int i = 0; i < ret.length; i++)
            ret[i] = res.get(i);

        return ret;
    }
}
```

- 时间复杂度: $O(N)$, N 为数组长度
- 空间复杂度: $O(N)$, N 为数组长度

哈希表+堆排序

思路

看到 求前 k 个 这样的描述自然会联想到用 堆 来进行排序。

- 用 大顶堆 的话，需要将所有 [数字, 次数] 元组都入堆，再进行 k 次取极值的操作。

- 用 小顶堆 的话，只需要维持堆的大小一直是 k 即可。

这里我们采用第二种思路。

1. 建立一个 size 为 K 的小顶堆，堆中存的是每个数的频次信息。堆初始化为空。

2. 对每个频次 C ，与堆顶 T 比较，如果 C > T, C 替换 T，维持小顶堆性质。

代码

代码支持：CPP，Java

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> counts;
        // 计算频次
        for(int i : nums) counts[i]++;
        // 最小堆
        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> q;
        // 堆中元素为 [频次，数值] 元组，并根据频次维护小顶堆特性
        for(auto it : counts) {
            if (q.size() != k) {
                q.push(make_pair(it.second, it.first));
            } else {
                if (it.second > q.top().first) {
                    q.pop();
                    q.push(make_pair(it.second, it.first));
                }
            }
        }
        vector<int> res;
        while(q.size()) {
            res.push_back(q.top().second);
            q.pop();
        }
        return vector<int>(res.rbegin(), res.rend());
    }
};
```

Java Code(from leetcode-cn)：

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

        // int[] 的第一个元素代表数组的值，第二个元素代表了该值出现的次数
        PriorityQueue<int[]> queue = new PriorityQueue<int[]>(new Comparator<int[]>() {
            public int compare(int[] m, int[] n) {
                return m[1] - n[1];
            }
        });
        for (Map.Entry<Integer, Integer> entry : occurrences.entrySet()) {
            int num = entry.getKey(), count = entry.getValue();
            if (queue.size() == k) {
                if (queue.peek()[1] < count) {
                    queue.poll();
                    queue.offer(new int[]{num, count});
                }
            } else {
                queue.offer(new int[]{num, count});
            }
        }
        int[] ret = new int[k];
        for (int i = 0; i < k; ++i) {
            ret[i] = queue.poll()[0];
        }
        return ret;
    }
}
```

- 时间复杂度: $O(N * \log K)$, N 为数组长度
- 空间复杂度: $O(N)$, N 为数组长度，主要为哈希表开销

思路 - 快速选择

快速排序变种，快速排序的核心是选出一个拆分点，将数组分为 `left` ， `right` 两个 part，对两个 part 内的元素分治处理，时间是 $O(n * \log n)$, 但是注意，我们只是需要找出前 K 个数，并不需要其有序，所有通过拆分出 K 个数，使得前 K 个数都大于后面 n - k 个数即可。

和快速排序的唯一不同是，快速选择每次不会递归访问 pivot 两侧，而是仅访问一侧。

和快速排序一样，最坏的情况时间复杂度是平方。这和 pivot 的选择有关，因此实际应用中更多的是检测到数组相对无序才会使用该算法。

代码

代码支持：C++，Java(from leetcode-cn)，JS

CPP Code:

```
class Solution {
public:
    void qsort(vector<pair<int, int>>& v, int start, int end, vector<int>& ret, int k) {
        int picked = rand() % (end - start + 1) + start;
        swap(v[picked], v[start]);

        int pivot = v[start].second;
        int index = start;
        for (int i = start + 1; i <= end; i++) {
            if (v[i].second >= pivot) {
                swap(v[index + 1], v[i]);
                index++;
            }
        }
        swap(v[start], v[index]);

        if (k <= index - start) {
            qsort(v, start, index - 1, ret, k);
        } else {
            for (int i = start; i <= index; i++) {
                ret.push_back(v[i].first);
            }
            if (k > index - start + 1) {
                qsort(v, index + 1, end, ret, k - (index - start + 1));
            }
        }
    }

    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> occurrences;
        for (auto& v: nums) {
            occurrences[v]++;
        }

        vector<pair<int, int>> values;
        for (auto& kv: occurrences) {
            values.push_back(kv);
        }
        vector<int> ret;
        qsort(values, 0, values.size() - 1, ret, k);
        return ret;
    }
};
```

Java Code:

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

        List<int[]> values = new ArrayList<int[]>();
        for (Map.Entry<Integer, Integer> entry : occurrences.entrySet()) {
            int num = entry.getKey(), count = entry.getValue();
            values.add(new int[]{num, count});
        }
        int[] ret = new int[k];
        qsort(values, 0, values.size() - 1, ret, 0, k);
        return ret;
    }

    public void qsort(List<int[]> values, int start, int end, int[] ret, int retIndex, int k) {
        int picked = (int) (Math.random() * (end - start + 1)) + start;
        Collections.swap(values, picked, start);

        int pivot = values.get(start)[1];
        int index = start;
        for (int i = start + 1; i <= end; i++) {
            if (values.get(i)[1] >= pivot) {
                Collections.swap(values, index + 1, i);
                index++;
            }
        }
        Collections.swap(values, start, index);

        if (k <= index - start) {
            qsort(values, start, index - 1, ret, retIndex, k);
        } else {
            for (int i = start; i <= index; i++) {
                ret[retIndex++] = values.get(i)[0];
            }
            if (k > index - start + 1) {
                qsort(values, index + 1, end, ret, retIndex, k - (index - start + 1));
            }
        }
    }
}
```

JavaScript Code：

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number[]}
 */
var topKFrequent = function (nums, k) {
  const counts = {};
  for (let num of nums) {
    counts[num] = (counts[num] || 0) + 1;
  }
  let pairs = Object.keys(counts).map((key) => [counts[key], key]);

  select(0, pairs.length - 1, k);
  return pairs.slice(0, k).map((item) => item[1]);

  // 快速选择
  function select(left, right, offset) {
    if (left >= right) {
      return;
    }
    const pivotIndex = partition(left, right);
    console.log({ pairs, pivotIndex });
    if (pivotIndex === offset) {
      return;
    }

    if (pivotIndex <= offset) {
      select(pivotIndex + 1, right, offset);
    } else {
      select(left, pivotIndex - 1);
    }
  }

  // 拆分数组为两个part
  function partition(left, right) {
    const [pivot] = pairs[right];
    let cur = left;
    let leftPartIndex = left;
    while (cur < right) {
      if (pairs[cur][0] > pivot) {
        swap(leftPartIndex++, cur);
      }
      cur++;
    }
    swap(right, leftPartIndex);
    return leftPartIndex;
  }

  function swap(x, y) {
```

```
    const term = pairs[x];
    pairs[x] = pairs[y];
    pairs[y] = term;
  }
};
```

- 时间复杂度: $O(N)$, 最坏能到 $O(N^2)$
- 空间复杂度: $O(N)$

上一页 下一页