top-k-frequent-elements

```
package algorithm.sort;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

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

 Given a non-empty array of integers, return the k most frequent elements.

 For example,
 Given [1,1,1,2,2,3] and k = 2, return [1,2].

 Note:
 You may assume k is always valid, 1 ≤ k ≤ number of unique elements.
 Your algorithm's time complexity must be better than O(n log n), where n i

 * @author xiaobaoqiu  Date: 16-5-18 Time:  下午10:03
 */
public class TopKFrequentElements {
    public static void main(String[] args) {
        int[] nums = new int[] {4,4,4,2,4,2,3};

        //2.
        List<Integer> ret = topKFrequent(nums, 2);
        for (int v : ret) System.out.print(v + ",");
    }

    /**
     * 39 ms
     * 整体时间复杂度：O(n)+O(n*logk)
     */
    public static List<Integer> topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> counter = new HashMap<Integer, Integer>();
        //O(n)
        for (int v : nums) {
            if (!counter.containsKey(v)) counter.put(v, 1);
            else counter.put(v, counter.get(v) + 1);
        }

        //小顶堆，大小为k
        PriorityQueue<Map.Entry<Integer, Integer>> queue = new PriorityQueu
            public int compare(Map.Entry<Integer, Integer> left, Map.Entry<
                return left.getValue() - right.getValue();
            }
        });
        //O(n*logk)
        for (Map.Entry<Integer, Integer> entry : counter.entrySet()) {
            queue.offer(entry);
```

```
            if (queue.size() > k) queue.poll();
        }

        List<Integer> res = new ArrayList<Integer>(k);
        while(!queue.isEmpty()) {
            res.add(queue.poll().getKey());
        }
        return res;
    }
}
```