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

Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

**Example 1:**

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

**Example 2:**

Input: nums = [1], k = 1

Output: [1]

**Constraints:**

1 <= nums.length <= 10^5

-10^4 <= nums[i] <= 10^4

k is in the range [1, the number of unique elements in the array].

It is **guaranteed** that the answer is **unique**.

 **Follow up:** Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

**Attempt 1: 2023-02-26**

**Solution 1: PriorityQueue + Hash Table (10 min)**

class Solution {

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

        Map<Integer, Integer> freq = new HashMap<Integer, Integer>();

        for(int num : nums) {

            freq.put(num, freq.getOrDefault(num, 0) + 1);

        }

        PriorityQueue<Integer> maxPQ = new PriorityQueue<Integer>((a, b) -> freq.get(b) - freq.get(a));

        for(int n : freq.keySet()) {

            maxPQ.offer(n);

        }

        int[] result = new int[k];

        for(int i = 0; i < k; i++) {

            result[i] = maxPQ.poll();

        }

        return result;

    }

}

Time complexity: O(Nlog⁡k), if k<N and O(N) in the particular case of N=k. That ensures time complexity to be better than O(Nlog⁡N).

Space complexity: O(N+k) to store the hash map with not more N elements and a heap with k elements.

**Refer to**

<https://leetcode.com/problems/top-k-frequent-elements/editorial/>

**Approach 1: Heap**

Let's start from the simple [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)) approach with O(Nlog⁡k) time complexity. To ensure that O(Nlog⁡k) is always less than O(Nlog⁡N), the particular case k=N could be considered separately and solved in O(N) time.

**Algorithm**

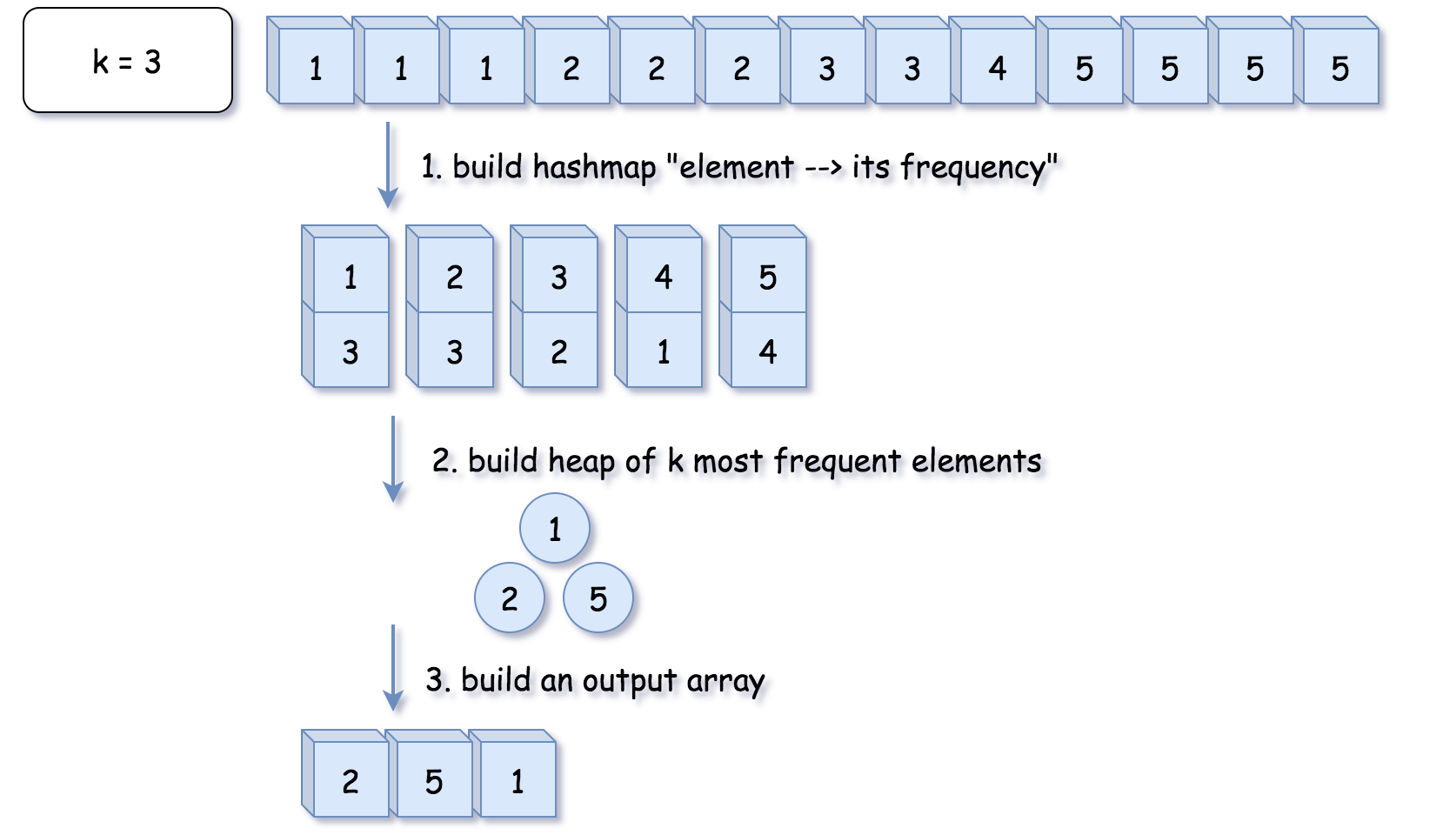
The first step is to build a hash map element -> its frequency. In Java, we use the data structure HashMap. Python provides dictionary subclass Counter to initialize the hash map we need directly from the input array.

This step takes O(N) time where Nis a number of elements in the list.

The second step is to build a heap of *size k using N elements*. To add the first k elements takes a linear time O(k) in the average case, and O(log⁡1+log⁡2+...+log⁡k)=O(logk!)=O(klog⁡k)in the worst case. It's equivalent to [heapify implementation in Python](https://hg.python.org/cpython/file/2.7/Lib/heapq.py#l16). After the first k elements we start to push and pop at each step, N - k steps in total. The time complexity of heap push/pop is O(log⁡k) and we do it N - k times that means O((N−k)log⁡k) time complexity. Adding both parts up, we get O(Nlog⁡k) time complexity for the second step.

The third and the last step is to convert the heap into an output array. That could be done in O(klog⁡k) time.

In Python, library heapq provides a method n largest, which [combines the last two steps under the hood](https://hg.python.org/cpython/file/2.7/Lib/heapq.py#l203) and has the same O(Nlog⁡k) time complexity.



class Solution {

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

        // O(1) time

        if (k == nums.length) {

            return nums;

        }

        // 1. build hash map : character and how often it appears

        // O(N) time

        Map<Integer, Integer> count = new HashMap();

        for (int n: nums) {

          count.put(n, count.getOrDefault(n, 0) + 1);

        }

        // init heap 'the less frequent element first'

        Queue<Integer> heap = new PriorityQueue<>((n1, n2) -> count.get(n1) - count.get(n2));

        // 2. keep k top frequent elements in the heap

        // O(N log k) < O(N log N) time

        for (int n: count.keySet()) {

          heap.add(n);

          if (heap.size() > k) heap.poll();

        }

        // 3. build an output array

        // O(k log k) time

        int[] top = new int[k];

        for(int i = k - 1; i >= 0; --i) {

            top[i] = heap.poll();

        }

        return top;

    }

}

**Complexity Analysis**

Time complexity : O(Nlog⁡k)if k<N and O(N) in the particular case of N=k. That ensures time complexity to be better than O(Nlog⁡N).

Space complexity : O(N+k) to store the hash map with not more N elements and a heap with k elements.

**Solution 2: Bucket Sort (30 min)**

class Solution {

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

        Map<Integer, Integer> freq = new HashMap<Integer, Integer>();

        for(int num : nums) {

            freq.put(num, freq.getOrDefault(num, 0) + 1);

        }

        // Array index represent frequency, list on certain index

        // represent values happen as that frequence (range 1 to n)

        // that's why increase length with 1 to include frequence = n

        // Note: Don't write as "new List<Integer>[nums.length + 1]",

        // will throw out generic array creation

        // Refer to

        // http://stackoverflow.com/questions/7131652/generic-array-creation-error

        List<Integer>[] bucket = new List[nums.length + 1];

        for(Map.Entry<Integer, Integer> entry : freq.entrySet()) {

            if(bucket[entry.getValue()] == null) {

                bucket[entry.getValue()] = new ArrayList<Integer>();

            }

            bucket[entry.getValue()].add(entry.getKey());

        }

        // Loop the bucket and print all bucket not null from highest freq

        // which equals to start from right index

        // Note: Don't forget limitation on k with "result.size() < k"

        List<Integer> result = new ArrayList<Integer>();

        for(int i = bucket.length - 1; i >= 0 && result.size() < k; i--) {

            if(bucket[i] != null) {

                result.addAll(bucket[i]);

            }

        }

        // Refer to

        // How to convert an ArrayList containing Integers to primitive int array?

        // https://stackoverflow.com/questions/718554/how-to-convert-an-arraylist-containing-integers-to-primitive-int-array

        return result.stream().mapToInt(i -> i).toArray();

    }

}

Time complexity: O(N)

Space complexity: O(N+k) to store the hash map with not more N elements and a heap with k elements.

**Refer to**

<https://leetcode.com/problems/top-k-frequent-elements/solutions/81602/java-o-n-solution-bucket-sort>

Idea is simple. Build an array of list to be buckets with length 1 to sort

public List<Integer> topKFrequent(int[] nums, int k) {

    List<Integer>[] bucket = new List[nums.length + 1];

    Map<Integer, Integer> frequencyMap = new HashMap<Integer, Integer>();

    for (int n : nums) {

        frequencyMap.put(n, frequencyMap.getOrDefault(n, 0) + 1);

    }

    for (int key : frequencyMap.keySet()) {

        int frequency = frequencyMap.get(key);

        if (bucket[frequency] == null) {

            bucket[frequency] = new ArrayList<>();

        }

        bucket[frequency].add(key);

    }

    List<Integer> res = new ArrayList<>();

    for (int pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {

        if (bucket[pos] != null) {

            res.addAll(bucket[pos]);

        }

    }

    return res;

}

Thanks for sharing, only one nitpick:

Think about the case when K=2,and you have 1 number that has max frequency, say 10 times.and you have 10 numbers that has 2nd max frequency, say 9 times.With your algo, the returned list will contain 11 numbers instead of 2.

Any easy fix:return res.subList(0,k);

(It seems the above scenario is not covered by the existing test cases.)

I tweaked the code a bit so it would give [1,2] for test case " [1,1,1,2,2,2,3,3,3] k=2" instead of [1,2,3].

public List<Integer> topKFrequent(int[] nums, int k) {

    Map<Integer, Integer> hm = new HashMap<>();

    List<Integer>[] bucket = new List[nums.length + 1];

    List<Integer> res = new ArrayList<>();

    for(int num: nums){

        hm.put(num, hm.getOrDefault(num, 0) + 1);

    }

    for(int key: hm.keySet()){

        int frequency = hm.get(key);

        if(bucket[frequency] == null)

            bucket[frequency] = new ArrayList<>();

        bucket[frequency].add(key);

    }

    for(int pos = bucket.length-1; pos >= 0; pos--){

        if(bucket[pos] != null){

            for(int i = 0; i < bucket[pos].size() && res.size() < k; i++)

                res.add(bucket[pos].get(i));

        }

    }

    return res;

}

**Refer to**

[L387.First Unique Character in a String (Ref.L451,L2351)](note://WEBb0eb8477294265e35391dc28dc47dd0b)

[L451.P14.5.Sort Characters By Frequency (Ref.L347)](note://512A6121D907491D8CF8FDD9119E5CC3)