Solution: Top K Frequent Elements

Let's solve the Top K Frequent Elements problem using the Top K Elements pattern.

We'll cover the following

Statement

Given an array of integers, `arr`, and an integer, `k`, return the k most frequent elements.

> **Note:** You can return the answer in any order.

Constraints:

- $1 \leq$ `arr.length` $\leq 10^3$
- $10^{-4} \leq$ `arr[i]` $\leq 10^4$
- $1 \leq$ `k` $\leq$ number of unique elements in an array.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach to finding the k most frequent elements involves iterating through each element in the input array and storing their frequencies in a hash map. For each element in the array, the algorithm checks if the element is already in the hash map. If it is not in the hash map, it is added with a frequency of 1. Otherwise, its frequency is incremented.

After counting the frequencies of each element in the array, the algorithm searches through the hash map for the element with the highest frequency. This involves iterating through the hash map, which takes $O(n)$ time. Once the element with the highest frequency is found, it is removed from the hash map and added to a final list of the k most frequent elements.

The time complexity for this approach is $O(k * n)$, and the space complexity is $O(n + k)$, where n is the length of the input array.

Optimized approach using top k elements

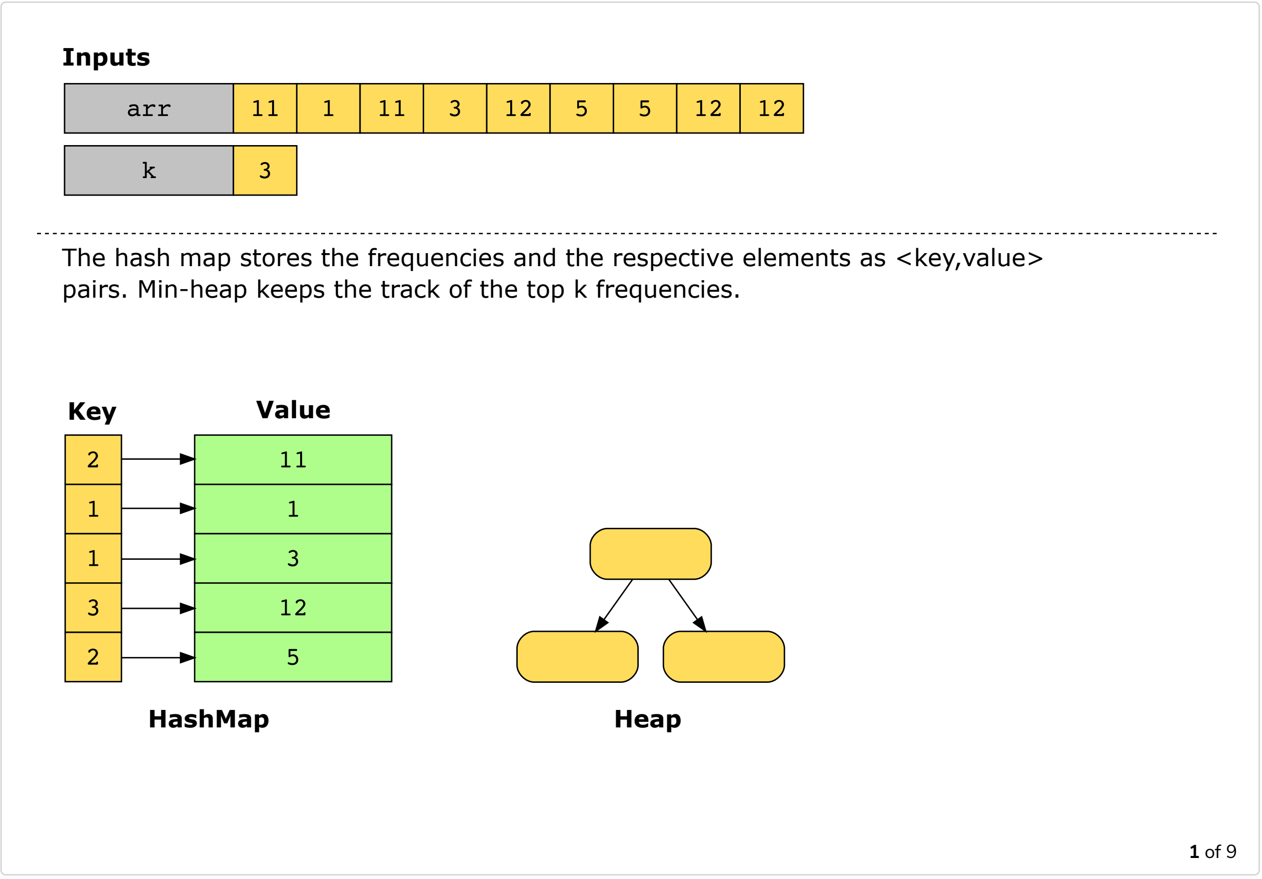
We can optimize the previous approach by using the top k elements pattern, which is a useful pattern for finding the top k elements of a sequence or collection based on a certain criterion. We first need to know the frequency of each element. We can do this using a hash map. Once we have the frequency map, the min-heap is the best data structure to keep track of the top `k` frequencies. After storing the frequencies of the elements, we'll iterate through the hash map, insert each element into our heap, and find the top `k` frequent elements.

The hash map will store the element as the key, and its corresponding frequency in the array as the value. When inserting elements from the hash map into the min-heap, the following steps are taken:

- We'll store a pair (a, b) as a tuple (this can be in the form of any data structure like an array or a tuple) in the heap where a will be the frequency of the element, and b will be the element itself. This ensures that the elements in the min-heap are always sorted by frequency.
- We'll make sure that if the size of the min-heap becomes greater than `k`, that is, there are more than `k` elements in the min-heap, we pop the pair with the lowest frequency element from the min-heap. This ensures that we always have the `k` maximum frequent elements in the min-heap.

Once we have added the pairs from the hash map to the min-heap, the min-heap will have the pairs with the top `k` frequencies. We will traverse the min-heap and add the second element of each pair from the min-heap to this new array. This is done since we only need the top `k` frequent elements, not their respective frequencies. Finally, we will return this new array.

The illustration below shows the whole process:



Inputs
arr
11
1
11
3
12
5
5
12
12
k
3
The hash map stores the frequencies and the respective elements as <key,value> pairs. Min-heap keeps the track of the top k frequencies.
Key
Value
2
11
1
1
1
3
3
12
2
5
HashMap
Heap
1 of 9

Let's look at the code for this solution below:

Java

```java
import java.util.*;
```

```java
class FrequentElements {
    public static List<Integer> topKFrequent(int[] arr, int k) {
        // Find the frequency of each number
        Map<Integer, Integer> numFrequencyMap = new HashMap<>();
        for (int n : arr)
            numFrequencyMap.put(n, numFrequencyMap.getOrDefault(n, 0) + 1);

        PriorityQueue<Map.Entry<Integer, Integer>> topKElements = new PriorityQueue<>(
                (e1, e2) -> e1.getValue() - e2.getValue()
        );

        // Go through all numbers of the numFrequencyMap and push them into topKElements, which will have
        // the top k frequent numbers. If the heap size is more than k, we remove the smallest (top) numb
        for (Map.Entry<Integer, Integer> entry : numFrequencyMap.entrySet()) {
            topKElements.add(entry);
            if (topKElements.size() > k) {
                topKElements.poll();
            }
        }

        // Create a list of top k numbers
        List<Integer> topNumbers = new ArrayList<>(k);
        while (!topKElements.isEmpty()) {
            topNumbers.add(topKElements.poll().getKey());
        }
        Collections.sort(topNumbers);
```

Top K Frequent Elements

1. Create a hash map to store the frequency of each number in the input array.
2. Create an empty min heap to store the top `k` frequent elements seen so far.
3. For each key-value pair (num, frequency) in the hash map, add the pair to the min-heap and keep the size of the heap to a maximum of `k` by popping the least frequent element when the size exceeds `k`.
4. Extract the top k numbers from the min heap by popping elements from the heap and inserting them in a new list.
5. Return the list, which contains the top k frequent numbers.

Time complexity

Let n be the number of elements in our list and k be the number of frequent elements we need to return. Since we iterate through a list of size n, inserting an element into the heap takes $O(\log(k))$ time. If $k < n$, the time complexity will be $O(n \times \log(k))$, and if $k = n$, the time complexity will be $O(n \times log(n))$.

Space complexity

We'll be using a heap and hash map to store the elements. Since the heap will be storing at most k elements and the hash map will contain n elements, the total space complexity is $O(n + k)$.

← Back
Top K Frequent Eleme...

Next →
Kth Largest Element i...

Mark as Completed