Maximum Distinct Elements (medium)

We'll cover the following

Problem Statement

Given an array of numbers and a number ‘K’, we need to remove ‘K’ numbers from the array such that we are left with maximum distinct numbers.

Example 1:

```
Input: [7, 3, 5, 8, 5, 3, 3], and K=2
Output: 3
Explanation: We can remove two occurrences of 3 to be left with 3 distinct numbers [7, 3, 8], w
e have
to skip 5 because it is not distinct and occurred twice.
Another solution could be to remove one instance of '5' and '3' each to be left with three
distinct numbers [7, 5, 8], in this case, we have to skip 3 because it occurred twice.
```

Example 2:

```
Input: [3, 5, 12, 11, 12], and K=3
Output: 2
Explanation: We can remove one occurrence of 12, after which all numbers will become distinc
t. Then
we can delete any two numbers which will leave us 2 distinct numbers in the result.
```

Example 3:

```
Input: [1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5], and K=2
Output: 3
Explanation: We can remove one occurrence of '4' to get three distinct numbers.
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const find_maximum_distinct_elements = function(nums, k) {
  // TODO: Write your code here
  return -1;
};

console.log(`Maximum distinct numbers after removing K numbers: ${find_maximum_distinct_elements([7, 3, 5
console.log(`Maximum distinct numbers after removing K numbers: ${find_maximum_distinct_elements([3, 5, 1
console.log(`Maximum distinct numbers after removing K numbers: ${find_maximum_distinct_elements([1, 2, 3
```

RUN SAVE RESET

Solution

This problem follows the Top ‘K’ Numbers pattern, and shares similarities with Top ‘K’ Frequent Numbers.

We can following a similar approach as discussed in Top ‘K’ Frequent Numbers problem:

1. First, we will find the frequencies of all the numbers.
2. Then, push all numbers that are not distinct (i.e., have a frequency higher than one) in a **Min Heap** based on their frequencies. At the same time, we will keep a running count of all the distinct numbers.
3. Following a greedy approach, in a stepwise fashion, we will remove the least frequent number from the heap (i.e., the top element of the min-heap), and try to make it distinct. We will see if we can remove all occurrences of a number except one. If we can, we will increment our running count of distinct numbers. We have to also keep a count of how many removals we have done.
4. If after removing elements from the heap, we are still left with some deletions, we have to remove some distinct elements.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```javascript
const Heap = require('./collections/heap'); //http://www.collectionsjs.com


function find_maximum_distinct_elements(nums, k) {
  distinctElementsCount = 0;
  if (nums.length <= k) {
    return distinctElementsCount;
  }

  // find the frequency of each number
  numFrequencyMap = {};
  nums.forEach((num) => {
    if (!(num in numFrequencyMap)) {
      numFrequencyMap[num] = 1;
    } else {
      numFrequencyMap[num]++;
    }
  });

  const minHeap = new Heap([], null, ((a, b) => b - a));
  // insert all numbers with frequency greater than '1' into the min-heap
  Object.keys(numFrequencyMap).forEach((num) => {
    if (numFrequencyMap[num] === 1) {
      distinctElementsCount += 1;
    } else {
      minHeap.push(numFrequencyMap[num]);
    }
  });
```

RUN | SAVE | RESET

Close

Output 4.444s

```
Maximum distinct numbers after removing K numbers: 3
Maximum distinct numbers after removing K numbers: 2
Maximum distinct numbers after removing K numbers: 3
```

Time complexity

Since we will insert all numbers in a **HashMap** and a **Min Heap**, this will take $O(N * logN)$ where 'N' is the total input numbers. While extracting numbers from the heap, in the worst case, we will need to take out 'K' numbers. This will happen when we have at least 'K' numbers with a frequency of two. Since the heap can have a maximum of 'N/2' numbers, therefore, extracting an element from the heap will take $O(logN)$ and extracting 'K' numbers will take $O(KlogN)$. So overall, the time complexity of our algorithm will be $O(N * logN + KlogN)$.

We can optimize the above algorithm and only push 'K' elements in the heap, as in the worst case we will be extracting 'K' elements from the heap. This optimization will reduce the overall time complexity to $O(N * logK + KlogK)$.

Space complexity

The space complexity will be $O(N)$ as, in the worst case, we need to store all the 'N' characters in the **HashMap**.

Back
'K' Closest Numbers (medium)

Next
Sum of Elements (medium)

Mark as Completed

Report an Issue | Ask a Question