

Maximum Distinct Elements (medium)

We'll cover the following ^

- Problem Statement
- Try it yourself
- Solution
- Code
 - Time complexity
 - Space complexity

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], we 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 distinct. 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++

```
1 import java.util.*;
2
3 class MaximumDistinctElements {
4
5     public static int findMaximumDistinctElements(int[] nums, int k) {
6         // TODO: Write your code here
7         return -1;
8     }
9
10    public static void main(String[] args) {
11        int result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 7, 3, 5, 8, 5, 3, 3 }, 2);
12        System.out.println("Maximum distinct numbers after removing K numbers: " + result);
13
14        result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 3, 5, 12, 11, 12 }, 3);
15        System.out.println("Maximum distinct numbers after removing K numbers: " + result);
16
17        result = MaximumDistinctElements.findMaximumDistinctElements(new int[] { 1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5 }, 2);
18        System.out.println("Maximum distinct numbers after removing K numbers: " + result);
19    }
20 }
```

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

```
1 import java.util.*;
2
3 class MaximumDistinctElements {
4
5     public static int findMaximumDistinctElements(int[] nums, int k) {
6         int distinctElementsCount = 0;
7         if (nums.length <= k)
8             return distinctElementsCount;
9
10        // find the frequency of each number
11        Map<Integer, Integer> numFrequencyMap = new HashMap<>();
12        for (int i : nums)
13            numFrequencyMap.put(i, numFrequencyMap.getOrDefault(i, 0) + 1);
14
15        PriorityQueue<Map.Entry<Integer, Integer>> minHeap = new PriorityQueue<Map.Entry<Integer, Integer>>((
16            (e1, e2) -> e1.getValue() - e2.getValue()));
17
18        // insert all numbers with frequency greater than '1' into the min-heap
19        for (Map.Entry<Integer, Integer> entry : numFrequencyMap.entrySet()) {
20            if (entry.getValue() == 1)
21                distinctElementsCount++;
22            else
23                minHeap.add(entry);
24        }
25
26        // following a greedy approach, try removing the least frequent numbers first from the min-heap
27        while (k > 0 && !minHeap.isEmpty()) {
28            Map.Entry<Integer, Integer> entry = minHeap.poll();
```

Run Save Reset ↺

Time complexity

Since we will insert all numbers in a **HashMap** and a **Min Heap**, this will take $O(N * \log N)$ 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 $O(\log N)$ and extracting 'K' numbers will take $O(K \log N)$. So overall, the time complexity of our algorithm will be $O(N * \log N + K \log N)$.

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 $O(N * \log K + K \log K)$.

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

Next →

'K' Closest Numbers (medium)

Sum of Elements (medium)

✓ Mark as Completed

