1481. Least Number of Unique Integers after K Removals

Medium Greedy Array Hash Table Counting Sorting

Leetcode Link

Problem Description

Given an array `arr` of integers and an integer `k`, the task is to determine the minimum number of unique integers that remain in the array after precisely `k` elements have been removed. The key to solving this problem lies in effectively selecting which elements to remove to achieve the least number of unique integers possible.

Intuition

To solve this problem, we should consider removing elements that appear more frequently first since this will not reduce the unique count as quickly. Ideally, we want to remove the numbers that occur the least amount of times last. We can apply a strategy that consists of the following steps:

1. Count the frequency of each unique integer in the array using a hash table (or `Counter` in Python).
2. Sort these unique integers by their frequency in ascending order. This way, the elements that appear less frequently are at the beginning of the sorted list.
3. Traverse the sorted list, and with each step, decrease `k` by the frequency of the current element. This simulates removing that element's occurrences from the array.
4. If `k` becomes negative, it means we can't remove all occurrences of the current element without exceeding the allowed `k` removals. Therefore, the current count of unique integers minus the number of elements we've been able to fully remove by this point gives us the answer.
5. If we go through the entire list without `k` becoming negative, it means we managed to remove all occurrences of certain elements, and we end up with 0 unique integers.

Having this strategy in place allows us to reach the solution in an efficient manner.

Solution Approach

The implementation of the solution follows a straightforward approach outlined in previous steps, which uses common data structures and algorithms:

1. **Hash Table (Counter):** We employ a hash table which is native to Python called `Counter` from the `collections` module. This structure automatically counts the frequency of each unique integer, which is essential to our strategy. It simplifies the process of determining how many times each integer appears in the array.

```
cnt = Counter(arr)
```

2. **Sorting:** After counting the occurrences of each integer, we sort these counts. This sorting is ascending, meaning that unique integers with fewer occurrences are considered first when we start removing elements.

```
sorted(cnt.values())
```

3. **Traversal and Subtraction:** With the sorted frequencies, we traverse the list. For each frequency value `v`, we subtract `k` by `v`, simulating the removal of `v` occurrences from the array.

```
for i, v in enumerate(sorted(cnt.values())):
    k -= v
    ...
```

   - If `k` becomes less than zero during the iteration, it indicates that we cannot remove all occurrences of the current element as it would exceed `k`. Hence, the minimal number of unique integers is obtained by subtracting the current index `i` from the total number of unique integers (the length of `cnt`):

```
if k < 0:
    return len(cnt) - i
```

4. **Return Result:** If we are able to traverse the entire sorted list of frequencies without `k` becoming negative, we have managed to remove all occurrences of certain elements leading to zero unique integers left.

```
return 0
```

The overall complexity of the solution is determined mainly by the sorting operation and the counting operation, with the rest being linear traversals and basic arithmetic operations.

Example Walkthrough

Let's use a small example to illustrate the solution approach:

Suppose `arr = [4, 3, 1, 1, 2, 3, 3]` and `k = 3`. Our goal is to remove `k` elements from `arr` in such a way that the number of unique integers remaining is minimal.

1. **Count Frequency:** We first count how many times each integer appears in the array:
   - 1 appears 2 times
   - 2 appears 1 time
   - 3 appears 3 times
   - 4 appears 1 time

   Using a hash table:

```
cnt = {1: 2, 2: 1, 3: 3, 4: 1}
```

2. **Sort by Frequency:** We sort these counts in ascending order based on frequency:

   Sorted counts: `[1, 1, 2, 3]`

   The number of unique integers is initially 4.

3. **Traverse and Remove:** We then traverse the sorted list and remove `k` elements.
   - We start with the first element (frequency of `1`). We remove one instance of the integer with a count of `1`, which decreases `k` to 2.
   - We move to the next element (another frequency of `1`). We remove one instance of another integer with a count of `1`, which decreases `k` to 1.
   - With the third element (frequency of `2`), we can remove both instances of the integer `1`, which would decrease `k` to -1. However, since we can't go negative, it means we can't remove all instances of `1`. At this point, we have removed two unique integers (those with the initial frequency of `1`), resulting in a remaining unique integer count of `2` (those with frequencies of `2` and `3` are still present).

   Here's the traversal in action:

```
cnt = {1: 2, 2: 1, 3: 3, 4: 1}
sorted_counts = [1, 1, 2, 3]
k = 3
for i, v in enumerate(sorted_counts):
    k -= v
    if k < 0:
        return 4 - i  # number of unique integers initially minus the index
```

4. **Return Result:** With the loop broken, we know we've removed instances of unique integers only until the array's `k` becomes negative. This means we return `4 - 2`, which equals `2`.

Therefore, the minimum number of unique integers remaining in the array `arr` after removing exactly `k` elements is `2`.

Python Solution

```
from collections import Counter
from typing import List

class Solution:
    def findLeastNumOfUniqueInts(self, arr: List[int], k: int) -> int:
        # Create a counter for all elements in the array
        counter = Counter(arr)

        # Sort the counts of each unique integer
        sorted_counts = sorted(counter.values())

        # Go through the counts starting from the smallest
        for index, value in enumerate(sorted_counts):
            # Reduce the count of deletable elements by the current count value
            k -= value

            # If k becomes negative, we can't delete anymore unique integers
            if k < 0:
                # Return the count of remaining unique integers
                return len(counter) - index

        # If k is not negative after trying to remove all, return 0
        # because all elements can be removed to achieve k deletion
        return 0
```

Java Solution

```
class Solution {
    public int findLeastNumOfUniqueInts(int[] arr, int k) {
        // Create a hashmap to store the frequency of each integer in the array
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        // Populate the frequency map
        for (int num : arr) {
            frequencyMap.merge(num, 1, Integer::sum); // Increment the count for each occurrence of a number
        }
        // Create a list to store the frequencies only
        List<Integer> frequencies = new ArrayList<>(frequencyMap.values());
        // Sort the frequencies in ascending order
        Collections.sort(frequencies);
        // Iterate over the list of frequencies
        for (int i = 0, totalUniqueNumbers = frequencies.size(); i < totalUniqueNumbers; ++i) {
            k -= frequencies.get(i); // Subtract the frequency from 'k'
            if (k < 0) {
                // If 'k' becomes negative, the current frequency can't be fully removed
                // so return the number of remaining unique integers
                return totalUniqueNumbers - i;
            }
        }
        // If all frequencies have been removed with 'k' operations, return 0 as there are no unique integers left
        return 0;
    }
}
```

C++ Solution

```
#include <vector>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    int findLeastNumOfUniqueInts(vector<int>& arr, int k) {
        // Create a hashmap to count the occurrence of each integer in the array
        unordered_map<int, int> frequencyMap;
        for (int number : arr) {
            ++frequencyMap[number];
        }

        // Extract the frequencies and sort them in ascending order
        vector<int> frequencies;
        for (auto& [number, count] : frequencyMap) {
            frequencies.push_back(count);
        }
        sort(frequencies.begin(), frequencies.end());

        // Determine the least number of unique integers by removing k occurrences
        int uniqueIntegers = frequencies.size();
        for (int i = 0; i < frequencies.size(); ++i) {
            // Subtract the frequency of the current number from k
            k -= frequencies[i];

            // If k becomes negative, we can't remove any more numbers
            if (k < 0) {
                return uniqueIntegers - i; // Return the remaining number of unique integers
            }
        }

        // If k is non-negative after all removals, we've removed all duplicates
        return 0;
    }
};
```

Typescript Solution

```
function findLeastNumOfUniqueInts(arr: number[], k: number): number {
    // Create a map to hold the frequency of each integer in the array
    const frequencyMap: Map<number, number> = new Map();
    // Iterate over the array and populate the frequency map
    for (const number of arr) {
        frequencyMap.set(number, (frequencyMap.get(number) || 0) + 1);
    }

    // Extract the frequency values from the map and store them in an array
    const frequencies: number[] = [];
    for (const frequency of frequencyMap.values()) {
        frequencies.push(frequency);
    }

    // Sort the frequencies array in ascending order
    frequencies.sort((a, b) => a - b);

    // Iterate over the sorted frequencies
    for (let i = 0; i < frequencies.length; ++i) {
        // Decrement k by the current frequency
        k -= frequencies[i];
        // If k becomes negative, we've used up k removals, so we return
        // the number of unique integers left, which is the length of the
        // frequencies array minus the current index
        if (k < 0) {
            return frequencies.length - i;
        }
    }

    // If we've processed all frequencies and haven't used up k removals,
    // all integers have been removed and 0 unique integers are left
    return 0;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n log n)`. This complexity arises from the sorting operation `sorted(cnt.values())` where `cnt.values()` represents the counts of unique integers in the array `arr`; sorting these counts requires `O(n log n)` time since sorting is typically done using comparison-based algorithms like quicksort or mergesort which have `O(n log n)` complexity in the average and worst case.

The space complexity of the code is `O(n)` because we are storing counts of the elements in the array in a dictionary `cnt`. In the worst case, if all elements are unique, it will contain `n` key-value pairs which relate directly to the size of the input array `arr`.