1224. Maximum Equal Frequency

Hard | Array | Hash Table — Leetcode Link

Problem Description

In this problem, we are given an array `nums` consisting of positive integers. The goal is to find the length of the longest prefix (an initial segment of the array) with the property that if you remove exactly one element from it, all the remaining numbers in the prefix will have the same frequency of occurrence.

This means that in the longest prefix, for all the numbers that have appeared thus far, either all numbers appear the same number of times, or removing one instance of a number can achieve this state of equal frequency.

An important part of the description specifies that if we end up removing one element from the prefix, and there are no elements left, this also counts as having all numbers appearing with the same frequency (since they all appear zero times).

Intuition

To solve this problem, we need to keep track of two things at each step:

1. The frequency of each number in the prefix (`cnt`).
2. The count of frequencies that we've seen (`ccnt`).

Using these two counters, we can determine at each step if the prefix can have equal frequency after removing a single element, and hence calculate the longest such prefix.

The thinking process is to iterate through the array, and at each step, update our two counters: one for the frequency of the current element (`cnt[v]`) and another for the frequency of this frequency (`ccnt[cnt[v]]`). While doing so, we maintain a variable `mx` that stores the maximum frequency of any number in the prefix so far.

We consider three cases where the prefix could potentially have all numbers appearing the same number of times after removing one element:

1. If the maximum frequency (`mx`) is 1, which means all elements are unique, any prefix is valid by removing any one of them.
2. If the highest frequency number (`mx`) appears exactly once (`ccnt[mx] == 1`) and the rest of the numbers have one less frequency (`mx - 1`), then by removing the single occurrence of the highest frequency number, all numbers would have the same frequency.
3. Likewise, if all the numbers have the same frequency (`ccnt[mx] * mx`), except for one number that is alone (`ccnt[1] == 1`), we can remove this single occurrence to match all the others.

We repeat these checks at each step and update the answer everytime we find a longer valid prefix. After iterating through the array, the value in `ans` will be the length of the longest possible prefix satisfying the problem condition.

Solution Approach

The solution uses a couple of key data structures along with some logical checks that help execute the strategy described in the intuition effectively.

Data Structures

1. **Counter `cnt`:** This is a dictionary that keeps track of the occurrences of each element in the prefix of the array `nums`. If `v` is an element in `nums`, then `cnt[v]` will tell us how many times `v` has appeared so far.

2. **Counter `ccnt`:** This is a dictionary that keeps track of the count of the frequencies themselves. It tells us how many numbers have a certain frequency. If `f` is a frequency, then `ccnt[f]` will tell us how many numbers have appeared exactly `f` times.

Algorithm

- Initialize the `cnt` and `ccnt` dictionaries (imported from `collections` class in Python). They are both initially empty.

- Start iterating through each element `v` in `nums`, keeping track of the index `i` (starting from 1 instead of 0).

- At each iteration, increase the count for `v` in `cnt`. If `v` has appeared before, decrease the counter for its old frequency in `ccnt` - since the frequency of `v` is about to increase by one.

- Update `ccnt` for the new frequency of `v`.

- Keep track of the maximum frequency seen so far in variable `mx`. This is updated to the maximum of its current value and the frequency of the current number `cnt[v]`.

- Perform the checks to see if we could make all frequencies equal by removing one element:

  - Check if `mx` is 1, meaning all numbers are unique, and any prefix is valid.
  - Check if `ccnt[mx]` is 1 and the total count of elements with frequency `mx` or `mx - 1` equals the current length of the array (`i`). This means we could remove the one element with frequency `mx` to even out the counts.
  - Check if the total count of elements with frequency `mx` plus 1 equals the current length of the array and there is only one number with the frequency of 1 (`ccnt[1] == 1`). We could remove the single occurrence of an element to achieve equal frequency.
- If any of the above conditions are true, then the current prefix (up to the index `i`) can be made to have equal frequencies by removing one element. We thus update `ans` to store the maximum prefix length satisfying the condition.

- After the loop terminates, `ans` will contain the length of the longest prefix which can achieve equal frequency by removing one element, and we return this value.

Using a combination of logical checks along with efficient tracking of frequencies of the numbers and their counts using hashmaps (`cnt` and `ccnt`), the solution is able to identify the longest prefix satisfying the problem's conditions without having to check each possible prefix repeatedly.

Example Walkthrough

Let's take an array `nums` with the values `[1, 2, 2, 3, 3, 3, 4, 4, 4, 4]` and walk through the solution step by step to understand how we apply the solution approach described above.

- We initialize our `cnt` and `ccnt` dictionaries empty and a variable `ans` for the answer.

- Start with `i = 1`, processing the first element `v = 1`.

  - Update `cnt[1]` from 0 to 1. Since 1 has not appeared before, there's no old frequency count to decrease in `ccnt`.
  - Update `ccnt[1]` from 0 to 1, since `cnt[1]` is now 1.
  - Set `mx` to 1, as it's the first and thus the highest frequency so far.
  - Since `mx` is 1, we update `ans` to 1, which is the length of the prefix now.
- Moving to `i = 2` with the element `v = 2`.

  - `cnt[2]` gets updated from 0 to 1.
  - `ccnt[1]` gets updated to 2 now, as there are two numbers with a frequency of 1.
  - `mx` is still 1, and thus the prefix of length 2 also satisfies the condition, so `ans` is updated to 2.
- Continue with `i = 3` with the element `v = 2`.

  - `cnt[2]` increases from 1 to 2.
  - Decrease `ccnt[1]` from 2 to 1 since the frequency of 2 has changed.
  - Update `ccnt[2]` from 0 to 1.
  - `mx` updates to 2.
  - Now, `ccnt[mx] * mx` + `ccnt[1]` is 3, which matches the length of the prefix, so `ans` is updated to 3.

In the interest of brevity, let's jump to `i = 10` with the element `v = 4`.

- We should have `cnt` showing `{1: 1, 2: 2, 3: 3, 4: 4}`, and `ccnt` showing `{1: 1, 2: 1, 3: 1, 4: 1}`.

- The `mx` now is 4 after processing all elements up to index 10.

  - Check if `mx` is 1 (false in this case).
  - Check if `ccnt[mx]` is 1 and the total count of elements with frequency `mx` or `mx - 1` equals the current index (10). This is false because `ccnt[4]` is 1, but `ccnt[3] * 3 + ccnt[4] * 4` is 9, not 10.
  - Check if `ccnt[mx] * mx + ccnt[1]` equals the current index (10) and `ccnt[1]` is 1. This is false because `ccnt[4] * 4 + ccnt[1]` is 17, not 10.
- Since no condition is met, we do not update `ans`.

After processing all elements, `ans` remains 3, which is the length of the longest prefix where one can remove exactly one element to make the frequency of the remaining elements the same. Therefore, the answer for our example array `[1, 2, 2, 3, 3, 3, 4, 4, 4, 4]` is 3.

Python Solution

```python
from collections import Counter

class Solution:
    def maxEqualFreq(self, nums):
        # Counts the frequency of each number in nums
        frequency_counter = Counter()
        # Counts the frequency of each frequency (!) in frequency_counter
        frequencies_count = Counter()
        # Initialize answer and max frequency seen so far
        answer = max_frequency = 0

        # Iterate over nums, tracking the index and value with i and num respectively
        for i, num in enumerate(nums, 1):
            # If the current number already has a frequency, decrement its frequency count
            if num in frequency_counter:
                frequencies_count[frequency_counter[num]] -= 1

            # Increment the frequency count of the current number and get the new count
            frequency_counter[num] += 1
            num_frequency = frequency_counter[num]

            # Update max_frequency if necessary
            max_frequency = max(max_frequency, num_frequency)

            # Increment the count of this new frequency
            frequencies_count[num_frequency] += 1

            # Check if a single frequency dominates (all numbers have the same frequency)
            if max_frequency == 1:
                answer = i
            # Check if, by removing one instance of the max frequency, all other numbers have the same frequency
            elif frequencies_count[max_frequency] * max_frequency + frequencies_count[max_frequency - 1] * (max_frequency - 1) == i a
                answer = i
            # Check if we can remove one number to satisfy the condition (all frequencies are the same except one)
            elif frequencies_count[max_frequency] * max_frequency + 1 == i and frequencies_count[1] == 1:
                answer = i

        return answer
```

Java Solution

```java
class Solution {
    // Initialize count arrays with enough space
    private static int[] frequencyCounter = new int[100010];
    private static int[] countOfFrequencyCounter = new int[100010];

    public int maxEqualFreq(int[] nums) {
        // Reset frequencies for each test case
        Arrays.fill(frequencyCounter, 0);
        Arrays.fill(countOfFrequencyCounter, 0);

        int longestPrefixLength = 0; // Stores the answer
        int maxFrequency = 0; // Maximum frequency of any element

        // Iterate through the array elements
        for (int i = 1; i <= nums.length; ++i) {
            int value = nums[i - 1];

            // Decrement the count of the current frequency for the value
            if (frequencyCounter[value] > 0) {
                --countOfFrequencyCounter[frequencyCounter[value]];
            }
            // Increment the frequency for the value and update the count of that frequency
            ++frequencyCounter[value];
            maxFrequency = Math.max(maxFrequency, frequencyCounter[value]);
            ++countOfFrequencyCounter[frequencyCounter[value]];

            // Check the conditions that need to be satisfied to consider the prefix

            // All elements have frequency one, can remove any one element
            if (maxFrequency == 1) {
                longestPrefixLength = i;
            }
            // Remove one instance of the element with maximum frequency
            else if (countOfFrequencyCounter[maxFrequency] * maxFrequency +
                     countOfFrequencyCounter[maxFrequency - 1] * (maxFrequency - 1) == i &&
                     countOfFrequencyCounter[maxFrequency] == 1) {
                longestPrefixLength = i;
            }
            // Only one element that can be removed to equalize the frequency
            else if (countOfFrequencyCounter[maxFrequency] * maxFrequency + 1 == i &&
                     countOfFrequencyCounter[1] == 1) {
                longestPrefixLength = i;
            }
        }
        // Return the length of the longest prefix where all elements can have the same frequency
        return longestPrefixLength;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int maxEqualFreq(vector<int>& nums) {
        unordered_map<int, int> frequencyMap; // Map to store the frequency of each number
        unordered_map<int, int> freqCountMap; // Map to store the count of particular frequencies
        int result = 0; // Variable to store the maximum length of subarray
        int maxFreq = 0; // Variable to keep track of the maximum frequency

        // Iterate through the array
        for (int i = 1; i <= nums.size(); ++i) {
            int value = nums[i - 1]; // Value of the current element

            // Decrease the count of the old frequency in the freqCountMap
            if (frequencyMap[value]) {
                --freqCountMap[frequencyMap[value]];
            }

            // Increase the frequency of the current element
            ++frequencyMap[value];
            // Update the maxFreq if necessary
            maxFreq = max(maxFreq, frequencyMap[value]);
            // Increase the count of the new frequency in the freqCountMap
            ++freqCountMap[frequencyMap[value]];

            // Check if all numbers have the same frequency
            if (maxFreq == 1) {
                result = i;
            }
            // Check if we can delete one element to make all frequencies equal
            else if (freqCountMap[maxFreq] * maxFreq + freqCountMap[maxFreq - 1] * (maxFreq - 1) == i && freqCountMap[maxFreq] == 1
                result = i;
            }
            // Check if we have one number of max frequency and all others are 1
            else if (freqCountMap[maxFreq] * maxFreq + 1 == i && freqCountMap[1] == 1) {
                result = i;
            }
        }

        return result; // Return the length of longest subarray with max equal frequency
    }
};
```

Typescript Solution

```typescript
function maxEqualFreq(nums: number[]): number {
    const length = nums.length;
    const frequencyMap = new Map<number, number>();

    // Fill the frequency map with the occurrences of each number in nums array
    for (const num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) ?? 0) + 1);
    }

    // Iterate backward to find the longest prefix of nums with all numbers having same frequency
    for (let index = length - 1; index > 0; index--) {
        for (const key of frequencyMap.keys()) {
            // Decrease the frequency of current key to check for equal frequency
            frequencyMap.set(key, frequencyMap.get(key)! - 1);

            let frequency = 0; // Store the frequency of the first non-zero occurrence.
            // Find the first non-zero frequency to compare with others.
            for (const value of frequencyMap.values()) {
                if (value !== 0) {
                    frequency = value;
                    break;
                }
            }

            let isValid = true; // Flag to check validity of equal frequency
            let sum = 1; // Start with 1 to consider the decreased frequency

            // Check if all non-zero frequencies are the same as the first non-zero frequency
            for (const value of frequencyMap.values()) {
                if (value !== 0 && value !== frequency) {
                    isValid = false;
                    break;
                }
                sum += value;
            }

            // If all nonzero frequencies are equal, return the sum as the max length
            if (isValid) {
                return sum;
            }

            // Restore the decreased frequency before moving to the next key
            frequencyMap.set(key, frequencyMap.get(key)! + 1);
        }

        // After checking all keys, decrease the frequency of the number at current index
        frequencyMap.set(nums[index], frequencyMap.get(nums[index])! - 1);
    }

    // If no solution found, return 1 (single number's frequency is always equal)
    return 1;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(N)` where `N` is the length of the `nums` list. This is because the code iterates through each element of the `nums` list exactly once, performing a constant amount of work for each element concerning updating the `cnt` and `ccnt` Counters, and computing the maximum frequency `mx`. Every operation inside the loop—such as checking if `v` is in `cnt`, updating counters, and the conditional checks—is done in constant time, assuming that the counter operations are O(1) on average due to the hashing mechanism used.

Space Complexity

The space complexity of the provided code is `O(N)` as well. The `cnt` and `ccnt` Counters will each store elements proportional to the number of distinct elements in `nums`. In the worst case, if all elements of `nums` are unique, the space taken by these counters will be linear in terms of the size of the input. Additionally, `nums`, `ans`, `mx`, and other variables use a constant amount of extra space, but this does not change the overall space complexity.