2910. Minimum Number of Groups to Create a Valid Assignment

Medium Greedy Array Hash Table Leetcode Link

Problem Description

In this problem, we're working with an array of integers called 'nums', and we've been tasked with organizing its indices into different groups. The challenge is to find the minimum number of groups needed to satisfy two specific criteria:

1. Each group can only contain indices that point to the same value in 'nums'. That means if 'nums[i]' equals 'nums[j]', then 'i' and 'j' can be in the same group.
2. Every group should be as close in size as possible. The size difference between any two groups must not be more than 1.

The goal is to achieve these conditions with the fewest number of groups possible.

Intuition

The solution strategy involves a clever use of counting and enumeration. Here's how we arrive at the solution:

1. We first count how frequently each number occurs in our 'nums' array using a hash table called 'cnt'. This helps us know the distribution of values within 'nums'.

2. Understanding that the groups can have sizes that are either 'k' or 'k+1', where 'k' is the minimum occurrence count of any value, we can enumerate possible group sizes starting from 'k' down to 1.

3. For each potential group size 'k', we look at every number's occurrence 'v' and check if it's possible to divide 'v' into groups of size 'k' or 'k+1' without violating our conditions (where the size of the groups does not differ by more than 1).

4. If for any occurrence 'v', it is not possible to make such groups (that is, if 'v/k' is less than 'v % k'), we know that dividing into groups of this size would not satisfy the condition. So, we skip to the next group size.

5. If it is possible to form groups for a given 'k', we compute the number of groups by dividing 'v' by 'k+1' and rounding up to ensure we count partially filled groups as a whole one (this is done to minimize the number of groups).

6. Since we are trying group sizes from largest to smallest, as soon as we find a valid grouping that satisfies our conditions, we can be assured that it's the optimal one with the minimum groups required.

By tackling it step-by-step, we ensure that we're checking all possible group sizes and finding the most economical way to distribute the indices in accordance with both conditions.

Solution Approach

The chosen approach to solve this problem can be broken down into a few strategic steps, employing common algorithms, data structures, and patterns.

First, let's detail the use of Python's `Counter` class for creating the frequency table, often referred to as a hash table, which is crucial for counting occurrences of each unique number from the array `nums`.

1. **Hash Table Creation**: Using `Counter(nums)`, we count the occurrences of each number in `nums`. It allows us to easily access the frequency `v` of each unique value in `nums`.

2. **Enumeration of Group Sizes**: We then attempt to find the minimum group sizes by starting from the minimum frequency `k` found in our hash table and decreasing towards 1. This takes advantage of the pattern that larger group sizes can potentially lead to fewer groups if such groupings are possible.

3. **Divisibility Check**: For each proposed group size `k`, we iterate through all frequencies `v` and check whether each value can be divided into groups of size `k` or `k+1` without exceeding the size difference constraint. This is done by checking if `floor(v/k) < v mod k`, where `floor` is implicit in integer division and `mod` is the modulo operation.
   - If this condition is true for any frequency `v`, it indicates that the size `k` cannot allow for a valid grouping, and we move to the next smaller size by breaking the current loop prematurely.
4. **Grouping Calculation**: If all frequencies can be grouped by the current `k`, it means that `k` is a valid group size. To ensure we use as few groups as possible, we want to create groups of `k+1` first and only use groups of size `k` if necessary. The calculation `(v + k) // (k + 1)` ensures we are creating as many full groups of size `k+1` as possible before resorting to any groups of size `k`.
   - This step is essential because we want the minimum number of groups, which means maximizing group size when possible while still respecting the rules.
5. **Optimal Solution**: Since we enumerate `k` from its maximum possible value down to 1, the first `k` for which a valid grouping exists will give us the minimum number of groups needed to satisfy the problem conditions.

This algorithm is both efficient and effective, employing a hash table for quick value access and enumeration to systematically check each group size. The first verified group size that holds the condition provides us with an optimal solution.

The formulae and concepts used:

- Hash table frequency count: `Counter(nums)`
- Enumeration from max to min: `for k in range(min(cnt.values()), 0, -1)`
- Divisibility check: `if v // k < v % k`
- Group calculation: `ans += (v + k) // (k + 1)`
- Optimal solution on first valid: `if ans: return ans`

This approach elegantly combines these elements to guarantee the most efficient grouping under the given constraints.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach with the array `nums = [3,3,3,3,1,1,1]`.

1. **Hash Table Creation**: First, we use `Counter(nums)` to create a frequency table.
   - We get `cnt = Counter({3: 4, 1: 3})`. This shows us that the number 3 occurs 4 times, and the number 1 occurs 3 times.
2. **Enumeration of Group Sizes**: Next, we look for the minimum group size starting from `min(cnt.values())`, which is 3 in this case, and decrement towards 1.
   - We iterate `k` from 3 down to 1.
3. **Divisibility Check**: For `k = 3`, we check each value in `cnt` to see if it can be divided into groups of size 3 or 4 (`k+1`) while respecting the size difference condition.
   - For value `v = 4` (number 3 in `nums`), we check if `4 // 3 < 4 % 3`.
     - This gives `1 < 1`, which is false, so a group size of 3 is valid for this occurrence.
   - For value `v = 3` (number 1 in `nums`), we check if `3 // 3 < 3 % 3`.
     - This gives `1 < 0`, which is false, so a group size of 3 is valid for this occurrence as well.
   - Since both checks passed, we move to the Grouping Calculation step.
4. **Grouping Calculation**: Both numbers can create groups with size 3 or 4. We calculate the number of groups for each:
   - For the number 3 in `nums` (`v = 4`): `ans += (4 + 3) // (4)` results in `ans += 2`.
   - For the number 1 in `nums` (`v = 3`): `ans += (3 + 3) // (4)` results in `ans += 1`.
   - Total `ans` now is `2 + 1 = 3`.
5. **Optimal Solution**: The first valid group size was 3, which required 3 groups. Since this is the first valid solution we've encountered as we decremented from `k = min(cnt.values())`, it is the optimal solution. Thus, we need a minimum of 3 groups to satisfy the problem conditions.

Putting it all together, we've found that the array `nums = [3,3,3,3,1,1,1]` can be organized into a minimum of 3 groups such that each group contains indices that point to the same value, and the sizes of the groups differ by at most 1. The groups can be visualized as: `[3, 3, 3]`, `[3]`, and `[1, 1, 1]`.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def minGroupsForValidAssignment(self, nums: List[int]) -> int:
        # Count the frequency of each number in the given list
        frequency_count = Counter(nums)

        # Iterate from the maximum frequency down to 1
        for group_size in range(max(frequency_count.values()), 0, -1):
            groups_needed = 0  # Initialize the number of groups needed for this group size

            # Iterate through each frequency value
            for frequency in frequency_count.values():
                # If the distribution of frequency across groups is invalid, reset groups and break
                if frequency // group_size < frequency % group_size:
                    groups_needed = 0
                    break

                # Calculate the number of groups needed for each number with its frequency
                groups_needed += -(-frequency // (group_size + 1))  # Same as ceil division

            # If groups are successfully calculated, return the result
            if groups_needed:
                return groups_needed

        # If unable to calculate number of groups, return 0
        return 0  # As per the original code (although this line is unnecessary since the function implicitly returns None if no retu

# Example usage:
# solution = Solution()
# print(solution.minGroupsForValidAssignment([1,2,3,3,3,4,4]))  # Replace with the actual numbers list

```

Java Solution

```java
class Solution {

    /**
     * Calculates the minimum number of groups for a valid assignment based on the input array.
     * The method counts the frequency of each number in the array and determines the smallest
     * number of groups such that the frequency of the numbers is proportionally distributed.
     * @param nums The input array containing numbers.
     * @return The minimum number of groups required.
     */
    public int minGroupsForValidAssignment(int[] nums) {
        // Create a map to store the frequency count of each unique number in nums
        Map<Integer, Integer> frequencyCount = new HashMap<>();
        for (int number : nums) {
            // Increment the frequency count for each number
            frequencyCount.merge(number, 1, Integer::sum);
        }

        // Initialize k as the number of elements in nums, the maximum possible frequency
        int k = nums.length;

        // Find the smallest value among the frequencies to identify the initial group size
        for (int frequency : frequencyCount.values()) {
            k = Math.min(k, frequency);
        }

        // Continuously try smaller values of k to optimize the number of groups
        while (k > 0) {
            int groupsNeeded = 0;
            for (int frequency : frequencyCount.values()) {
                // If the frequency divided by k leaves a remainder larger than the quotient,
                // the current value of k isn't a valid group size, break and try a smaller k
                if (frequency % k > frequency / k) {
                    groupsNeeded = 0;
                    break;
                }
                // Calculate the number of groups needed for the current value of k
                groupsNeeded += (frequency + k - 1) / k;
            }
            // If the number of needed groups is greater than zero, we've found a valid grouping
            if (groupsNeeded > 0) {
                return groupsNeeded;
            }
            // Decrement k and try again for a smaller group size
            k--;
        }

        // The code should never reach this point
        return -1; // This line is just for the sake of completeness; logically, it'll always return from the loop
    }
}

```

C++ Solution

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

class Solution {
public:
    int minGroupsForValidAssignment(vector<int>& nums) {
        // Create a map to store the frequency of each number in nums
        unordered_map<int, int> frequency;
        for (int num : nums) {
            frequency[num]++;
        }

        // Initialize k with an arbitrarily large number
        int k = 1e9;
        // Variable to store the minimum number of groups
        int minGroups = k;

        // Find the smallest frequency to initialize the minimum number of groups
        for (auto& element : frequency) {
            minGroups = min(minGroups, element.second);
        }

        // Decrease k until a valid configuration is found
        while (k > 0) {
            // Temporary variable to store count of groups for the current k
            int tempGroupCount = 0;

            // Check if the configuration is valid for current k
            for (auto& element : frequency) {
                int freq = element.second;

                // If at any point we cannot satisfy the condition, break out
                if (freq / k < freq % k) {
                    tempGroupCount = 0;
                    break;
                }

                // Otherwise, add the number of groups needed for this frequency
                tempGroupCount += (freq + k - 1) / k; // Use integer ceiling division
            }

            // If we found a valid group configuration, return it
            if (tempGroupCount > 0) {
                return tempGroupCount;
            }

            // Decrement k and try again
            --k;
        }

        // In case no valid configuration is found (which won't happen for valid input),
        // just return zero as a safe fallback.
        return 0;
    }
};

```

Typescript Solution

```typescript
function minGroupsForValidAssignment(nums: number[]): number {
    // A map to count the frequency of each number in the input array
    const frequencyMap: Map<number, number> = new Map();

    // Populating the frequency map
    for (const num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);
    }

    // Trying to find the minimum group size starting from the smallest frequency
    for (let groupSize = Math.min(...frequencyMap.values()); ; --groupSize) {
        let groupsNeeded = 0; // Variable to hold the number of groups needed

        // Calculate the number of groups needed for each unique number
        for (const [_, frequency] of frequencyMap) {

            // Calculate how many full groups can be formed with the current frequency
            const fullGroups = (frequency / groupSize) | 0;

            // If the number of full groups is less than the remainder, we cannot form a valid group
            if (fullGroups < frequency % groupSize) {
                groupsNeeded = 0; // Reset groups needed, as the current group size is invalid
                break;
            }

            // Increase the count of total groups needed
            groupsNeeded += Math.ceil(frequency / (groupSize + 1));
        }

        // If we found a valid number of groups, return it
        if (groupsNeeded) {
            return groupsNeeded;
        }

        // Note: there is no explicit break in this loop for when group size reaches 0
        // The loop relies on eventually finding a valid group size before that happens
    }
}

```

Time and Space Complexity

Time Complexity

The time complexity is actually not `O(n)` in general, it depends on both `n`, the length of `nums`, and also the range of unique values as well as their frequencies. The time complexity can be analyzed as follows:

- `Counter(nums)` has a complexity of `O(n)` since it goes through each element of `nums`.
- The outer loop runs at most `min(cnt.values())` times which depends on the minimum frequency of a number in `nums`.
- The inner loop runs `O(u)` times where `u` is the number of unique elements in `nums` because it iterates through all values in the counter.

So, the complexity is `O(n + min(min(cnt.values()), n/u) * u)` which is `O(n + min_count * u)` if we let `min_count` be `min(cnt.values())`.

Giving a final verdict on time complexity without constraints of input can lead to a misleading statement since it can vary. If `min_count` is small, it could be close to linear but could also go up to `O(n^2)` in the worst scenario when all elements are unique.

Space Complexity

The space complexity is `O(n)` for the counter dictionary that stores up to `n` unique values from `nums` where `n` is the length of `nums`. No other significant space is used.