781. Rabbits in Forest

Medium Greedy Array Hash Table Math Leetcode Link

Problem Description

In this problem, we have a forest that contains an unknown number of rabbits, and we have gathered some information from a few of them. Specifically, when we ask a rabbit how many other rabbits have the same color as itself, it gives us an integer answer. These answers are collected in an array called `answers`, where `answers[i]` represents the answer from the i-th rabbit.

Our task is to determine the minimum number of rabbits that could be in the forest based on these answers. It's important to realize that if a rabbit says there are `x` other rabbits with the same color, it means there is a group of `x + 1` rabbits of the same color (including the one being asked). However, it is possible that multiple rabbits from the same group have been asked, which we need to account for in our calculation. We are asked to find the smallest possible number of total rabbits that is consistent with the responses.

Intuition

To find a solution, we need to understand that rabbits with the same answer can form a group, and the size of each group should be one more than the answer (since the answer includes other rabbits only, not the one being asked). However, if we get more responses of a certain number than that number + 1, we know that there are multiple groups of rabbits with the same color.

We use a hashmap (or counter in Python), to count how many times each answer appears. Then for each unique answer `k`, the number of rabbits that have answered `k` (`v`), can form `ceil(v / (k + 1))` groups, and each group contains `k + 1` rabbits. We calculate the total number of these rabbits and sum them up for all different answers.

To determine the minimum number of rabbits that could be in the forest, we iterate through each unique answer in our counter, calculate the number of full groups for that answer, each group having `k + 1` rabbits, and sum them up. We make sure to round up to account for incomplete groups, as even a single rabbit's answer indicates at least one complete group of its color.

Hence, the summation of `ceil(v / (k + 1)) * (k + 1)` for all unique answers `k` gives us the minimum number of rabbits that could possibly be in the forest.

Solution Approach

To implement the solution, we primarily use the `Counter` class from Python's `collections` module to facilitate the counting of unique answers. This data structure helps us because it automatically creates a hashmap where each key is a unique answer and each value is the frequency of that answer in the `answers` array.

Here's a breakdown of the steps in the implementation:

1. Initialize the counter with `answers`, so we get a mapping of each answer to how many times it's been given.

```
counter = Counter(answers)
```

2. Iterate over the items (key-value pairs) in our counter:

```
for k, v in counter.items():
```

   - Here, `k` represents the number of other rabbits the rabbit claims have the same color, and `v` represents how many rabbits gave that answer.

3. For each unique answer `k`, calculate the minimum number of rabbits that could have given this answer by using the formula `ceil(v / (k + 1)) * (k + 1)`:

   - We divide `v` by `k + 1` since `k + 1` is the actual size of the group that the answer suggests. If `v` is not perfectly divisible by `k + 1`, we must round up since even one extra rabbit means there is at least one additional group of that color. This rounding up is done using `math.ceil`.

   - Then we multiply by `k + 1` to get the total number of rabbits in these groups.

   4. Sum up these values to get the overall minimum number of rabbits in the forest. The sum function combines the values for all unique answers, returning the final result.

   Implementation of the above steps:

```
return sum([math.ceil(v / (k + 1)) * (k + 1) for k, v in counter.items()])
```

The complete solution makes use of the hashmap pattern for efficient data access and the mathematical formula for rounding up to the nearest group size. This approach ensures that the number we calculate is the minimum possible while still being consistent with the given answers.

Example Walkthrough

Let's consider an example where the `answers` array given by rabbits is `[1, 1, 2]`.

- This means we have three rabbits who've given us answers:
  - Two rabbits say there is another rabbit with the same color as theirs.
  - One rabbit says there are two other rabbits with the same color as itself.

Using the steps from the solution approach, we proceed as follows:

1. Create a counter from the `answers` list:

   - The `Counter` would look like this: `{1: 2, 2: 1}`. This denotes that the answer '1' has appeared twice, and the answer '2' has appeared once.

2. Iterate over the items (key-value pairs) in our counter:

   - For the first key-value pair (`k=1`, `v=2`):
     - There are two rabbits that claim there is one other rabbit with the same color. As `k + 1` is `2`, we know that they are just in one group. So we don't need to round up; the group size is 2.
   - For the second key-value pair (`k=2`, `v=1`):
     - There is one rabbit that says there are two other rabbits with the same color. That indicates at least one group of `k + 1` which is `3`.

3. We sum up the group sizes to find the minimum number of rabbits that could have given these answers:

   - For the first group (`k=1`), since `v` is 2 and `k + 1` is 2, `ceil(v / (k + 1))` is `ceil(2 / 2)`, which is `1`. Thus, it accounts for `1 * (1 + 1)` which is `2` rabbits.

   - For the second group (`k=2`), since `v` is 1 and `k + 1` is 3, `ceil(v / (k + 1))` is `ceil(1 / 3)`, which is `1`. Thus, it accounts for `1 * (2 + 1)` which is `3` rabbits.

4. Adding the numbers together, `2 + 3`, the minimum number of rabbits in the forest is `5`.

By using this approach, we can efficiently calculate the minimum number of rabbits in the forest consistent with the given answers: `[1, 1, 2]` would lead us to conclude there are at least `5` rabbits in the forest.

Python Solution

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

class Solution:
    def numRabbits(self, answers: List[int]) -> int:
        # Count the occurrences of each answer
        answer_counter = Counter(answers)

        # Initialize total number of rabbits reported
        total_rabbits = 0

        # Iterate through each unique answer (number_of_other_rabbits) and its count
        for number_of_other_rabbits, count in answer_counter.items():
            # Each rabbit with the same answer (number_of_other_rabbits) forms a group.
            # The size of each group is number_of_other_rabbits + 1 (including itself).
            group_size = number_of_other_rabbits + 1

            # Calculate the number of full groups (possibly partial for the last group)
            # by dividing the count of rabbits by the group size and rounding up.
            # This gives the number of groups where each rabbit reports
            # number_of_other_rabbits other rabbits with the same color.
            number_of_groups = math.ceil(count / group_size)

            # Add to the total number of rabbits by multiplying the number of groups
            # by the size of the group.
            total_rabbits += number_of_groups * group_size

        # Return the total number of rabbits reported
        return total_rabbits

```

Java Solution

```java
class Solution {

    // Function to calculate the minimum probable number of rabbits in the forest
    public int numRabbits(int[] answers) {
        // Create a map to count the frequency of each answer
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        // Iterate over the array of answers given by the rabbits
        for (int answer : answers) {
            // Update the frequency of this particular answer
            frequencyMap.put(answer, frequencyMap.getOrDefault(answer, 0) + 1);
        }

        // Initialize the result variable to store the total number of rabbits
        int totalRabbits = 0;
        // Iterate over the entries in the map to calculate the total number of rabbits
        for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
            // key is the number of other rabbits the current rabbit claims exist
            int otherRabbits = entry.getKey();
            // value is the frequency of the above claim from the array of answers
            int frequencyOfClaim = entry.getValue();

            // Calculate the number of groups of rabbits with the same claim
            int groupsOfRabbits = (int) Math.ceil(frequencyOfClaim / ((otherRabbits + 1) * 1.0));
            // Add the total number of rabbits in these groups to the result
            totalRabbits += groupsOfRabbits * (otherRabbits + 1);
        }

        // Return the total number of rabbits calculated
        return totalRabbits;
    }
}

```

C++ Solution

```cpp
#include <cmath>        // Include cmath for using the ceil function
#include <map>          // Include map for using the map data structure
#include <vector>       // Include vector for using the vector data structure

class Solution {
public:
    // Function to calculate the minimum probable number of rabbits in the forest
    int numRabbits(std::vector<int>& answers) {
        // Create a map to count the frequency of each answer
        std::map<int, int> frequencyMap;
        // Iterate over the vector of answers given by the rabbits
        for (int answer : answers) {
            // Update the frequency of this particular answer
            frequencyMap[answer]++;
        }

        // Initialize the result variable to store the total number of rabbits
        int totalRabbits = 0;
        // Iterate over the entries in the map to calculate the total number of rabbits
        for (auto& entry : frequencyMap) {
            // key is the number of other rabbits the current rabbit claims exist
            int otherRabbits = entry.first;
            // value is the frequency of the above claim from the array of answers
            int frequencyOfClaim = entry.second;

            // Calculate the number of groups of rabbits with the same claim
            int groupsOfRabbits = static_cast<int>(std::ceil((double)frequencyOfClaim / (otherRabbits + 1)));
            // Add the total number of rabbits in these groups to the result
            totalRabbits += groupsOfRabbits * (otherRabbits + 1);
        }

        // Return the total number of rabbits calculated
        return totalRabbits;
    }
};

```

Typescript Solution

```typescript
// TypeScript code to calculate the minimum probable number of rabbits in the forest

// Define a method to calculate the minimum number of rabbits
function numRabbits(answers: number[]): number {

    // Create a map to hold the frequency of each answer given by the rabbits
    let frequencyMap: Map<number, number> = new Map();

    // Iterate over the array of answers given by the rabbits
    answers.forEach(answer => {
        // Update the frequency count of this particular answer
        frequencyMap.set(answer, (frequencyMap.get(answer) || 0) + 1);
    });

    // Initialize a variable to store the total number of rabbits
    let totalRabbits: number = 0;

    // Iterate over the entries in the map to cumulate the total number of rabbits
    frequencyMap.forEach((frequencyOfClaim, otherRabbits) => {
        // Calculate the number of groups of rabbits with the same claim
        let groupsOfRabbits: number = Math.ceil(frequencyOfClaim / (otherRabbits + 1));

        // Add the total number of rabbits in these groups to the cumulated result
        totalRabbits += groupsOfRabbits * (otherRabbits + 1);
    });

    // Return the calculated total number of rabbits
    return totalRabbits;
}

```

Time and Space Complexity

Time Complexity

The function `numRabbits` loops once through the `answers` array to create a `counter`, which is essentially a histogram of the answers. The time complexity of creating this counter is `O(n)`, where `n` is the number of elements in `answers`.

After that, it iterates over the items in the counter and performs a constant number of arithmetic operations for each distinct answer, in addition to calling the `math.ceil` function. Since the number of distinct answers is at most `n`, the time taken for this part is also `O(n)`.

Therefore, the overall time complexity of the function is `O(n)`.

Space Complexity

The main extra space used by this function is the `counter`, which in the worst case stores a count for each unique answer. In the worst case, every rabbit has a different answer, so the space complexity would also be `O(n)`.

Hence, the space complexity of the function is also `O(n)`.