1399. Count Largest Group

Easy Hash Table Math

Leetcode Link

Problem Description

In this problem, you are given an integer `n`. Your task is to count how numbers from `1` to `n` can be grouped based on the sum of their digits. Then, you need to return the count of the groups with the largest size. A group's size is determined by how many numbers share the same digit sum. For example, if `n` is 13, the numbers `11` (1+1=2) and `20` (2+0=2) would belong to the same group with a digit sum of 2. If the group with a digit sum of 2 has the most numbers in it, you would include that in your count of the largest groups.

Intuition

To solve this problem, the approach is to create a map that will keep track of how many times each digit sum appears across the numbers from `1` to `n`. The key intuition is that instead of directly grouping numbers and then comparing sizes, we can efficiently tally counts for each possible digit sum using a map or dictionary.

Here's the process to arrive at the solution:

1. We iterate through all numbers from `1` to `n`.
2. For each number, we calculate the sum of its digits.
3. We record the digit sum in a counter (dictionary) by incrementing the corresponding digit sum's count.
4. As we update the counter, we also keep track of the maximum count (size of the largest group) encountered so far.
5. If we encounter a digit sum with a count higher than our current maximum, we update the maximum and reset our answer to 1 since we have found a larger group size.
6. If we encounter a digit sum with a count equal to the maximum, we increment our answer because this is another group of the same largest size.

The final answer is the count of digit sum groups/counts that equal the maximum size we found throughout the iteration.

The provided Python solution follows this intuition and keeps track of the digit sums using the `Counter` class from the `collections` module, offering a clean and efficient way to manage the counts.

Solution Approach

The solution uses a hash table and some basic arithmetic operations to solve the problem. Specifically, it employs a `Counter` object from Python's `collections` module, which is a subclass of dictionary designed to count hashable objects.

Here's a detailed walk-through of the implementation:

1. A `Counter` called `cnt` is initialized to store the frequency of each possible digit sum. It maps each sum to the number of times it has appeared.
2. Two variables, `ans` and `mx`, are initialized to `0`. `ans` will store the final answer (the number of groups with the largest size), and `mx` will store the current maximum size encountered.
3. We iterate over every number from `1` to `n` inclusive, using a `for` loop.
4. Inside the loop, for each number `i`, we calculate the digit sum by repeatedly taking `i % 10` (which gives the last digit) and adding it to an accumulator variable `s`. After each step, we update `i` to `i // 10` which effectively removes the last digit from `i`. This process continues in a `while` loop until `i` becomes `0`.
5. We then update the Counter `cnt` by incrementing the count for the computed digit sum `s`.
6. After updating `cnt`, we compare the updated count `cnt[s]` to the current `mx`. If the new count is greater, this means we have a new largest group size, so we update `mx` to `cnt[s]`, and reset `ans` to `1` as this is the first group of this new size.
7. If the new count `cnt[s]` is equal to the current `mx`, then we have found another group of the current largest size, and we increment `ans` by `1`.
8. After the loop has finished executing, the variable `ans` holds the number of groups that have the largest size, and we return this value.

Mathematically, given `s` as the sum of digits of a number `i`, the operation to update `cnt` can be represented as:

```
cnt[s] += 1
```

And the comparisons to determine if we found a new largest group size or another group of that size is:

```
if mx < cnt[s]:
    mx = cnt[s]
    ans = 1
elif mx == cnt[s]:
    ans += 1
```

The provided code efficiently calculates the required group sizes and finds the number of groups with the largest size, adhering to the solution approach described.

Example Walkthrough

Let's walk through a small example using `n = 15` to illustrate the solution approach:

1. Initialize a `Counter` named `cnt` and two variables: `ans` set to `0` and `mx` also set to `0`.
2. We begin to iterate from `1` to `15`. Firstly, we take number `1`, the sum of its digits is `1`. We update `cnt[1]` to be `1`. Since `mx < cnt[1]`, `mx` is now set to `1` and `ans` is set to `1`.
3. Next, we consider number `2`, the sum of its digits is `2`. We update `cnt[2]` to be `1`. Now `mx` is still `1`, but `ans` is now `2`, as we have two groups (for digit sums `1` and `2`) with the same size, `1`.
4. This process continues for numbers `3`, `4`, `5`, `6`, `7`, `8`, and `9`, each forming a new group of size `1`. `ans` is now `9`.
5. Number `10` has a digit sum of `1`, updating `cnt[1]` to `2`. Now `mx < cnt[1]`, so we set `mx` to `2` and `ans` to `1`, since we have found a larger group size (group of digit sum `1` is now the largest with size `2`).
6. Numbers `11`, `12`, and `13` have digit sums of `2`, `3`, and `4` respectively:
   - For `11`, `cnt[2]` becomes `2`, tying with the current maximum size. This increments `ans` to `2` because there's another group with the same size as the largest found so far.
   - For `12`, `cnt[3]` becomes `2`, tying again, and `ans` goes up to `3`.
   - For `13`, `cnt[4]` becomes `2`, tying again, and `ans` goes up to `4`.
7. For Number `14`, the digit sum is `5`, and updating `cnt[5]` to `2` ties with the current max; `ans` is incremented to `5`.
8. For Number `15`, the digit sum is `6`, which also ties with the max after updating `cnt[6]` to `2`; `ans` is incremented to `6`.

After considering all numbers from `1` to `15`, we find there are `6` groups tied for the largest size, which is `2`. The groups are the digit sums of `1`, `2`, `3`, `4`, `5`, and `6`.

Therefore, the final answer that would be returned is `6`, representing the count of the largest groups by digit sum from `1` to `15`.

Python Solution

```python
from collections import Counter

class Solution:
    def countLargestGroup(self, n: int) -> int:
        # Initialize a counter to keep track of the sum of digits
        digit_sum_counter = Counter()
        # Initialize variables to keep track of the maximum count and the number of groups with that count
        max_count = 0
        num_groups_with_max_count = 0

        # Iterate over each integer from 1 to n
        for i in range(1, n + 1):
            sum_of_digits = 0
            current_number = i
            # Calculate the sum of the digits of the current number
            while current_number:
                sum_of_digits += current_number % 10
                current_number //= 10
            # Increment the count for the group represented by this sum of digits
            digit_sum_counter[sum_of_digits] += 1

            # Update max_count and num_groups_with_max_count based on the current sums
            if max_count < digit_sum_counter[sum_of_digits]:
                max_count = digit_sum_counter[sum_of_digits]
                num_groups_with_max_count = 1
            elif max_count == digit_sum_counter[sum_of_digits]:
                num_groups_with_max_count += 1

        # Return the number of groups that have the maximum size
        return num_groups_with_max_count
```

Java Solution

```java
class Solution {
    public int countLargestGroup(int n) {
        // Initialize a count array to keep track of the frequency of sums.
        int[] sumFrequency = new int[40]; // The maximum sum for 4 digits can be 9 + 4 = 36, thus 40 is safe.
        int largestGroupSizeCount = 0; // This will hold the count of groups with the largest size.
        int maxGroupSize = 0; // This will hold the maximum size of any group encountered.

        // Loop through each number from 1 to n.
        for (int x = 1; x <= n; ++x) {
            int sumOfDigits = 0; // This will hold the sum of digits of the current number.
            // Loop to calculate the sum of digits of the current number.
            for (int y = x; y > 0; y /= 10) {
                sumOfDigits += y % 10;
            }

            // Increase the frequency count of the current sum.
            ++sumFrequency[sumOfDigits];

            // If the current sum frequency is greater than the maxGroupSize, update maxGroupSize and reset largestGroupSizeCount.
            if (maxGroupSize < sumFrequency[sumOfDigits]) {
                maxGroupSize = sumFrequency[sumOfDigits];
                largestGroupSizeCount = 1;
            } else if (maxGroupSize == sumFrequency[sumOfDigits]) {
                // If the current sum frequency is equal to the maxGroupSize, increment the count.
                ++largestGroupSizeCount;
            }
        }

        // Return the count of the sums that have the largest group size.
        return largestGroupSizeCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int countLargestGroup(int n) {
        // Array to store counts of sum groups with a large enough size
        // to avoid the constant reallocation.
        int sumCounts[40] = {};

        // Variables to keep track of the largest group size and the
        // number of groups with this size.
        int largestGroupSizeCount = 0;
        int largestGroupSize = 0;

        // Iterate over all numbers from 1 to n.
        for (int i = 1; i <= n; ++i) {
            int digitSum = 0; // This will hold the sum of the digits.

            // Calculate the sum of digits of the current number.
            for (int x = i; x; x /= 10) {
                digitSum += x % 10;
            }

            // Increment the count of the found sum.
            ++sumCounts[digitSum];

            // If we've found a new maximum size for our digit groups.
            if (largestGroupSize < sumCounts[digitSum]) {
                largestGroupSize = sumCounts[digitSum]; // Update the largest group size.
                largestGroupSizeCount = 1; // Reset the count to reflect this new largest group size.
            } else if (largestGroupSize == sumCounts[digitSum]) {
                // If this group size is the same as the current largest, increment the count.
                ++largestGroupSizeCount;
            }
        }

        // Return the number of groups that have the largest size.
        return largestGroupSizeCount;
    }
};
```

Typescript Solution

```typescript
function countLargestGroup(n: number): number {
    // Initialize an array to store the count of each sum of digits
    const sumCounts: number[] = new Array(40).fill(0);
    let maxCount = 0;  // Variable to keep track of the maximum count of any sum
    let numMaxGroups = 0; // Variable to count how many groups have this maximum count

    // Iterate over each number from 1 to n
    for (let i = 1; i <= n; ++i) {
        let sum = 0; // Variable to store sum of digits of the current number i
        // Calculate the sum of digits of i
        for (let x = i; x > 0; x = Math.floor(x / 10)) {
            sum += x % 10;
        }
        // Increase the count of this sum in the 'sumCounts' array
        sumCounts[sum]++;
        // Check if the current sum has a new maximum count
        if (maxCount < sumCounts[sum]) {
            maxCount = sumCounts[sum]; // Update the max count
            numMaxGroups = 1; // Reset the number of max groups as a new max is found
        } else if (maxCount === sumCounts[sum]) {
            // If the current sum is equal to the maximum count,
            // increment the number of groups with max count
            numMaxGroups++;
        }
    }
    // Return the number of groups with the maximum count of sums
    return numMaxGroups;
}
```

Time and Space Complexity

The given Python code computes the size of the largest group of numbers based on the sum of the digits of each number from `1` to `n`.

Time Complexity:

To analyze the time complexity, let's examine the primary operations in the code:

1. The for loop runs from `1` to `n` which gives us `O(n)` complexity.
2. Inside the loop, we calculate the sum of digits of an integer `i`. Since in the worst case scenario integer `i` can have at most `O(log n)` digits (since every increase by an order of magnitude adds one digit), the inner while loop runs in `O(log n)` time for each number.
3. Updating the counter for sum `s` and comparing `mx` with `cnt[s]` for each number is done in constant time, `O(1)`.

The overall time complexity is thus the product of the complexity of the outer loop and the inner while loop, which gives us `O(n log n)`.

Space Complexity:

For space complexity, we need to consider:

1. The counter `cnt` will at most have an entry for each distinct sum of digits `s` that we can get from numbers `1` to `n`. In the worst case, this sum would not exceed `9 * log n` (assuming each digit of a number `n` is `9`). Therefore, the space complexity due to the `cnt` map is `O(log n)`.
2. The variables `s`, `i`, `mx`, and `ans` use constant space `O(1)`.

The space complexity of the algorithm is mainly determined by the `cnt` map. Hence, the overall space complexity is `O(log n)`.

The code you provided is already efficient with respect to big-O notation, and any further optimization would depend on specific constraints or requirements of the problem not detailed here.