1093. Statistics from a Large Sample

Medium Array Math Probability and Statistics Leetcode Link

Problem Description

You are provided with an array called `count` representing a large sample of integers where each element's index (`k`) corresponds to an integer, and the value at each index (`count[k]`) corresponds to how many times `k` occurs in the sample. The range of integers is from 0 to 255. Your task is to calculate various statistics about the sample: the minimum value (`minimum`), maximum value (`maximum`), mean, median, and mode of the sample. The mean is the sum of all the integer occurrences multiplied by their respective values divided by the total number of elements. The median is the middle value once the sample is sorted, or if the sample size is even, then it's the average of the two middle values. The mode is the value that appears most frequently in the sample, and it's given that there will be a single, unique mode. The result should be returned as an array of floating-point numbers containing the statistics in the order they were mentioned.

Intuition

The solution first involves finding the minimum and maximum values present in the sample. Since the `count` array represents how many times each integer in the range [0, 255] occurs, we iterate through this array and look for the first and last indexes that have a non-zero count which represents the minimum and maximum, respectively.

Next, we calculate the mean by summing up each integer multiplied by its occurrences (`k * count[k]`) and then dividing by the total number of elements in the sample (`cnt`).

Calculating the median is a bit trickier because we need to consider whether the total number of elements is odd or even. We use a helper function `find(i)` to find the i-th smallest value in the sample according to its sorted position. If the total number of elements is odd, we want the element that is in the middle position. If it is even, we average the two middle elements, which involves finding the `cnt // 2`-th and the `cnt // 2 + 1`-th elements if you sort the sample.

Lastly, the mode is the integer value that has the highest count in the `count` array. As we iterate through the `count` array to perform other operations, we also keep track of the mode by comparing the current integers' count.

Combining all these steps, the function calculates and returns the statistics as a list `[minimum, maximum, mean, median, mode]`.

Solution Approach

The `sampleStats` function in the provided solution uses a straightforward algorithm to calculate the various statistics for the given sample.

1. **Initialization of Variables:** The variables `mi` (minimum), `mx` (maximum), `s` (sum), `cnt` (count), and `mode` are initialized. The `inf` keyword in Python represents an infinite number, used here to make the initial minimum as high as possible. The `mx` is initialized to -1 to later find the maximum which will certainly be higher than -1. The sum `s` and count `cnt` are initialized to 0 to calculate the mean. The mode is initialized to 0.

2. **Finding Minimum, Maximum, Sum, Count, and Mode:** A single loop iterates through all possible integer values (0 to 255) as indices in the `count` array. If an index `k` has a non-zero count (`x`), the algorithm does several things:

   - Updates minimum and maximum using `min(mi, k)` and `max(mx, k)` functions.
   - Adds `k * x` to the sum `s`.
   - Increases the total count `cnt` by `x`.
   - Checks if the current count `x` is greater than the count of the current mode and if so, updates the mode.

3. **Finding the Median:** The median calculation depends on whether the total count `cnt` is odd or even. A helper nested function `find(i)` is defined which returns the `i`-th smallest value in the sample (if the sample were sorted). For an odd total count, it finds the middle value. For an even total count, it finds the average of the two middle values. This is done by calling the `find` function accordingly.

The helper function `find` works by iterating through the `count` array again. It accumulates the total number of elements seen so far in a temporary count `t` and checks if it is greater than or equal to the target `i`. When it reaches or passes the target, it returns the current integer value `k` as the `i`-th smallest value.

4. **Returning the Result:** Finally, the function returns a list with the calculated statistics: `[mi, mx, s / cnt, median, mode]`, corresponding to minimum, maximum, mean, median, and mode.

The pattern used here is mainly iterating through the count array to gather all the needed statistics, utilizing a single pass wherever possible to optimize performance and then leveraging aggregated data to derive mean and median.

Understanding this solution approach is mainly about recognizing that the array index itself represents the integer value in the sample, and the increment in the array represents its frequency or count in the sample.

Example Walkthrough

Let's consider a simple example to illustrate the solution approach. Suppose we have an array `count` which has non-zero values at indices 2, 3, and 4, which represent the integers 2, 3, and 4 in the sample. The `count` array looks like this:

```
count = [0, 0, 1, 2, 1, 0, 0, ... 0]
```

This means the integer 2 occurs once, 3 occurs twice, and 4 occurs once in our sample. Given that the rest of the `count` array contains zeros, we will ignore them for brevity. Now, let's walk through the solution approach:

1. **Initialization of Variables:** The variables `mi`, `mx`, `s`, `cnt`, and `mode` are initialized. We start with `mi` set to infinity, `mx` to -1, `s` and `cnt` to 0, and `mode` to 0.

2. **Finding Minimum, Maximum, Sum, Count, and Mode:** We iterate through the indices 0 to 255 of the `count` array. Here's the breakdown:

   - At index 2, `count[2]` is 1. We update `mi` to min(inf, 2) which is 2, `mx` to max(-1, 2) which is 2, add `2 * 1` to `s` to make it 2, add 1 to `cnt` to make it 1, and update the mode to 2 since 1 > 0.
   - At index 3, `count[3]` is 2. We update `mi` (remains 2), `mx` to max(2, 3) which is 3, add `3 * 2` to `s` to make it 8, add 2 to `cnt` to make it 3, and update mode to 3 since 2 > 1.
   - At index 4, `count[4]` is 1. We update `mi` (remains 2), `mx` to max(3, 4) which is 4, add `4 * 1` to `s` to make it 12, add 1 to `cnt` to make it 4, and mode remains 3 (since count of 4 is not greater than count of 3).

3. **Finding the Median:** The total count `cnt` is 4, which is even. We use our `find` function to locate the middle values. We need the average of the 2nd and 3rd smallest values.

   - Invoking `find(2)` will iterate through `count` and sum up the counts until it equals or surpasses 2. It surpasses 2 at index 3, thus the 2nd smallest value is 3.
   - Invoking `find(3)` will iterate again until the sum surpasses 3, which will happen at index 3 as well. Thus, the 3rd smallest value is also 3.
   - The average of these is 3, so the median is 3.

4. **Returning the Result:** We calculate the mean as `s / cnt` which is `12 / 4` equals 3. Our final statistics array is `[mi, mx, mean, median, mode]`, which in this case is `[2, 4, 3, 3, 3]`.

From this example, you see how the count array index's value is used as an integer from the sample and the array's value at that index is its frequency. By iterating through the count array, we can gather all the information we need for our statistics using well-organized loops and conditional statements.

Python Solution

```python
from math import inf
from typing import List

class Solution:
    def sampleStats(self, count: List[int]) -> List[float]:
        # Helper function to find the index of the i-th number in the data stream
        def find(i: int) -> int:
            total = 0
            for index, frequency in enumerate(count):
                total += frequency
                if total >= i:
                    return index

        # Initialize minimum and maximum to represent the range of the data stream
        minimum = inf
        maximum = -1
        # Initialize variables for sum and count to compute mean
        sum_values = 0
        total_count = 0
        # Initialize the mode value
        mode = 0

        # Loop through count array to compute min, max, sum, total_count, and mode
        for value, frequency in enumerate(count):
            if frequency:
                minimum = min(minimum, value)
                maximum = max(maximum, value)
                sum_values += value * frequency
                total_count += frequency
                # Update mode if the current frequency is greater than the max frequency found so far
                if frequency > count[mode]:
                    mode = value

        # Compute median
        if total_count & 1:  # If the total number of elements is odd
            median = find(total_count // 2 + 1)
        else:
            # If the number of elements is even, average the middle two elements
            median = (find(total_count // 2) + find(total_count // 2 + 1)) / 2

        # Compute mean
        mean = sum_values / total_count

        # Return a list containing the minimum, maximum, mean, median, and mode
        return [float(minimum), float(maximum), mean, median, float(mode)]

```

Java Solution

```java
class Solution {
    private int[] elementCounts; // This array holds the count for each number.

    public double[] sampleStats(int[] count) {
        this.elementCounts = count;
        int min = Integer.MAX_VALUE, max = Integer.MIN_VALUE;
        long sum = 0; // Used to calculate the mean
        int totalCount = 0; // Total number of elements
        int mode = 0; // The number that appears the most frequently

        // Iterate through the count array to find minimum, maximum, sum, total count, and mode
        for (int number = 0; number < elementCounts.length; ++number) {
            if (elementCounts[number] > 0) {
                min = Math.min(min, number);
                max = Math.max(max, number);
                sum += (long) number * elementCounts[number];
                totalCount += elementCounts[number];
                if (elementCounts[number] > elementCounts[mode]) {
                    mode = number;
                }
            }
        }

        // Calculate median
        double median = totalCount % 2 == 1
            ? findKthElement(totalCount / 2 + 1)
            : (findKthElement(totalCount / 2) +
               findKthElement(totalCount / 2 + 1)) / 2.0;

        // Calculate mean
        double mean = sum * 1.0 / totalCount;

        // Return the results as an array: [min, max, mean, median, mode]
        return new double[] {min, max, mean, median, mode};
    }

    // Helper function to find the kth element when the count array is treated like an expanded array
    private int findKthElement(int k) {
        int elementsSoFar = 0;

        // Iterate through elementCounts and keep adding until we reach or pass the kth element
        for (int number = 0;; ++number) {
            elementsSoFar += elementCounts[number];
            if (elementsSoFar >= k) {
                return number;
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<double> sampleStats(vector<int>& count) {
        // Lambda to find the value at the ith position when the array is considered sorted.
        auto findValueAtPosition = [&](int position) -> int {
            for (int value = 0, accumulated = 0;; ++value) {
                accumulated += count[value];
                if (accumulated >= position) {
                    return value;
                }
            }
        };

        // Initialize minimum and maximum values to extreme values.
        int minimumValue = INT_MAX, maximumValue = INT_MIN;

        // Initialize variables to calculate the sum and mode.
        long long sum = 0;
        int totalCount = 0, modeValue = 0;

        // Loop through the count array to find the minimum, maximum, total count and mode.
        for (int value = 0; value < count.size(); ++value) {
            if (count[value] > 0) {
                minimumValue = min(minimumValue, value);
                maximumValue = max(maximumValue, value);
                sum += static_cast<long long>(value) * count[value];
                totalCount += count[value];
                if (count[value] > count[modeValue]) {
                    modeValue = value;
                }
            }
        }

        // Calculate median using the findValueAtPosition lambda.
        double median;
        if (totalCount % 2 == 1) {
            // If the total count is odd, select the middle value directly.
            median = findValueAtPosition(totalCount / 2 + 1);
        } else {
            // If the total count is even, take the average of the two middle values.
            median = (findValueAtPosition(totalCount / 2) + findValueAtPosition(totalCount / 2 + 1)) / 2.0;
        }

        // Calculate mean by dividing the sum by the total count.
        double mean = sum / static_cast<double>(totalCount);

        // Store the stats results as a vector of doubles and return.
        return vector<double>{static_cast<double>(minimumValue),
                              static_cast<double>(maximumValue),
                              mean,
                              median,
                              static_cast<double>(modeValue)};
    }
};

```

Typescript Solution

```typescript
// Helper function to find the ith smallest number based on the cumulative frequency
function findIthNumber(numbersCount: number[], ith: number): number {
    for (let k = 0, total = 0; ; ++k) {
        total += numbersCount[k];
        if (total >= ith) {
            return k;
        }
    }
}

// Main function to calculate statistics from a sample provided as an array of counts for each value
function sampleStats(numbersCount: number[]): number[] {
    let min = Number.MAX_SAFE_INTEGER; // Initialize minimum to a large number
    let max = -1;                      // Initialize maximum to a small number
    let sum = 0;                       // Sum of all numbers
    let totalCount = 0;                // Total count of all numbers
    let mode = 0;                      // Mode value of the numbers

    // Iterate over each count index to calculate min, max, sum, and mode
    for (let k = 0; k < numbersCount.length; ++k) {
        if (numbersCount[k] > 0) {
            min = Math.min(min, k);
            max = Math.max(max, k);
            sum += k * numbersCount[k];
            totalCount += numbersCount[k];
            if (numbersCount[k] > numbersCount[mode]) {
                mode = k;
            }
        }
    }

    // Calculate median based on whether the count is odd or even
    let median =
        totalCount % 2 === 1
            ? findIthNumber(numbersCount, (totalCount >> 1) + 1)
            : (findIthNumber(numbersCount, totalCount >> 1) + findIthNumber(numbersCount, (totalCount >> 1) + 1)) / 2;

    // Return an array containing min, max, mean, median, mode
    return [min, max, sum / totalCount, median, mode];
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the code can be analyzed by looking at the number of operations performed relative to the number of elements `n` in the `count` list:

- The loop to find the minimum (`mi`) and maximum (`mx`) values, the total count (`cnt`), the sum (`s`), and the mode runs once for each of the `n` elements: `O(n)`.
- The `find` function is called either 2 or 3 times depending on whether `cnt` is odd or even. Each call to `find` runs in `O(n)` because, in the worst-case scenario, it iterates over the entire `count` list. Thus, the worst case for finding the median is `O(n)`.
- In summary, the overall time complexity for this piece of code is `O(n)` + `O(n)` for median calculation, which simplifies to `O(n)` since both are linear operations.

Space Complexity:

- The space complexity is `O(1)` because the space used does not depend on the input size `n`. The variables `mi`, `mx`, `s`, `cnt`, and `mode` use a constant amount of space.
- The `find` function uses a constant amount of space as well since the variables `t` and `k` are of fixed size.

In conclusion, the code exhibits a linear time complexity (`O(n)`) and constant space complexity (`O(1)`).