1365. How Many Numbers Are Smaller Than the Current Number

Easy Array Hash Table Counting Sorting

Leetcode Link

Problem Description

The problem is to find for each element in the array `nums`, how many elements are smaller than the current element. In other words, you walk through each item in the input list, and for each of these items, you need to count all other numbers in the list that are less than it. It's important to remember that you should not include the current element in the count. The goal is to create an output array of the same length as `nums` where each index contains this count of smaller numbers.

Example: If `nums = [8,1,2,2,3]`, the output would be `[4,0,1,1,3]`. Here, four numbers are smaller than 8, no number is smaller than 1, one number is smaller than the first occurrence of 2, and so on.

Intuition

To find a solution to this problem, we could consider the following:

1. Brute Force: We could use a nested loop to compare each pair of elements and count how many elements are smaller, but this would be O(n^2) in time complexity, which is not efficient for large arrays.

2. Sorting: We could sort the array and then map each element to its corresponding index, but this would lose the original order of the elements, which we need to preserve.

3. Counting Sort Principle: Since we know the elements in `nums` are integers and the problem typically has constraints that limit their range (not stated in the problem description here but usually up to 100), we can use a counting sort approach. This means we tally how many times each number occurs (the frequency), and then we can deduce how many numbers are smaller by looking at numbers (frequencies) to the left of the current number.

The intuition behind the provided solution is to build a frequency array `cnt` with an extra element so we can easily accumulate the counts. We use the `accumulate` function from Python's `itertools` to get a prefix sum of counts, effectively giving us a running total of how many numbers are smaller than each value up to 101 (assuming `nums` contains numbers up to 100 for this argument). By doing this, `s[x]` corresponds to the count of numbers less than `x` in the original array, as desired. Thus, `s` is used to create the output result conveniently and efficiently.

Solution Approach

The implementation of the solution makes use of a frequency array approach, which is a technique commonly used in counting sort algorithm. Here is a detailed walkthrough of the solution:

1. Initialize a frequency array `cnt` with a size of 102 to include all possible values from `0` to `100` along with an extra slot for ease of calculation. If `nums` contains numbers up to `maxValue`, the size of `cnt` should be `maxValue + 2`.

```
cnt = [0] * 102
```

2. Iterate over each element `x` in the input `nums` and increment the count of `x + 1` in the frequency array `cnt`. This shift by one is to allow us to later easily calculate the number of elements smaller than a given number using prefix sums.

```
for x in nums:
    cnt[x + 1] += 1
```

3. Calculate the prefix sum using Python's `accumulate` function from the `itertools` module. This operation will give us a new list `s` representing the cumulative frequency. Now `s[x]` will represent the total count of numbers that are less than `x`.

```
s = list(accumulate(cnt))
```

4. Iterate through the elements in `nums` again to create the results array. For each element `x` in `nums`, we look up `s[x]` which gives us the number of elements that are less than `x` and add this count to our result array.

```
return [s[x] for x in nums]
```

The solution thus effectively uses the counting sort algorithm, which is often preferred for small integer ranges due to its `O(n + k)` complexity, where `n` is the number of elements in the array and `k` is the range of input numbers. This is more efficient than comparison-based sorting methods when `k` is relatively small compared to `n`.

This approach leverages the `itertools.accumulate` function to avoid manually calculating prefix sums, which provides a clean and efficient way to obtain the cumulative frequency.

The beauty of this solution lies in its simplicity and efficiency, allowing us to conclude the counts of smaller numbers in a single pass while utilizing a non-comparison-based sorting concept.

Example Walkthrough

Let's illustrate the solution approach using a smaller example array, `nums = [4, 0, 2, 1]`. Here we want to find out how many elements are smaller than each element in `nums`.

1. We initialize the frequency array `cnt` with a size of 102 to cover all possible values in the range 0 to 100 inclusively, with an extra slot:

```
cnt = [0] * 102
```

2. We then iterate over each element in `nums` and increment the corresponding index in `cnt` (shifted by one):

```
nums = [4, 0, 2, 1]

After processing each element:
- cnt[4 + 1] = cnt[5] += 1  # increment at index 5
- cnt[0 + 1] = cnt[1] += 1  # increment at index 1
- cnt[2 + 1] = cnt[3] += 1  # increment at index 3
- cnt[1 + 1] = cnt[2] += 1  # increment at index 2

The frequency array ('cnt') looks like this after iteration:
cnt = [0, 1, 1, 1, 0, 1, 0, ..., 0]
```

3. We use Python's `accumulate` function to calculate the prefix sum, which gives us the cumulative count of smaller numbers:

```
from itertools import accumulate

s = list(accumulate(cnt))
# s now looks like this: [0, 1, 2, 3, 3, 4, 4, ..., 4]
# This array represents the cumulative count of numbers less than the index.
```

4. Lastly, we iterate through the elements in `nums` again and create our result array using the `s` array we just computed:

```
result = [s[x] for x in nums]
# result = [s[4], s[0], s[2], s[1]]
# result now should be: [3, 0, 2, 1]
```

The final `result` array indicates that:

- 3 numbers are smaller than 4 (`0`, `1`, and `2`)
- 0 numbers are smaller than 0
- 1 number is smaller than 2 (`0`)
- 1 number is smaller than 1 (`0`)

This example walkthrough demonstrates each step of the method described in the content above, yielding an efficient outcome that correctly matches each element in `nums` with the count of elements that are smaller than it.

Python Solution

```
from itertools import accumulate

class Solution:
    def smallerNumbersThanCurrent(self, nums: List[int]) -> List[int]:
        # Initialize a count array with additional space to accumulate counts.
        # We use 102 instead of 101 because we're using cnt[x+1] in the loop.
        count = [0] * 102

        # Count how many times each number (0-100) appears in nums.
        # We increment the count at index x+1 to later use this space for accumulate.
        for number in nums:
            count[number + 1] += 1

        # Calculate the prefix sum to determine how many numbers are
        # less than the current number. This would help in getting counts directly.
        # The 'accumulate' function is like a cumulative sum.
        sum_count = list(accumulate(count))

        # For each number in the original array, we now look up
        # how many numbers are smaller than it by referring to our sum_count.
        # The result for each number is simply sum_count at the index of that number.
        return [sum_count[number] for number in nums]
```

Java Solution

```
class Solution {
    public int[] smallerNumbersThanCurrent(int[] nums) {
        // Container to store the count of the number of smaller elements for each element in 'nums'.
        int[] count = new int[102]; // We use 102 instead of 101 to simplify the algorithm,
                                    // so we don't need to handle the case of 0 separately.

        // Populate the count array where index represents the number+1 and
        // the value at that index represents the count of that number in 'nums'.
        for (int num : nums) {
            ++count[num + 1];
        }

        // Convert the count array to a prefix sum array
        // where count[i] contains the number of elements less than i.
        for (int i = 1; i < count.length; ++i) {
            count[i] += count[i - 1];
        }

        // Length of the original 'nums' array.
        int n = nums.length;
        // Result array where each element will be replaced by the count of numbers less than it.
        int[] result = new int[n];

        // Fill the result array using the count array.
        for (int i = 0; i < n; ++i) {
            result[i] = count[nums[i]]; // nums[i] gives the number and count[nums[i]] gives
                                        // the count of elements less than that number.
        }

        // Return the result array.
        return result;
    }
}
```

C++ Solution

```
#include <vector>

class Solution {
public:
    // Function to find the number of elements smaller than each element in the array
    vector<int> smallerNumbersThanCurrent(vector<int>& nums) {
        // Array to store the count of each number plus one value ahead (prefix sum)
        int count[102] = {0}; // Initialize all elements to 0

        // Count how many times each number appears in the input vector
        for (int number : nums) {
            ++count[number + 1];
        }

        // Calculate the prefix sum to get the number of elements less than current
        for (int i = 1; i < 102; ++i) {
            count[i] += count[i - 1];
        }

        // Initialize the answer vector to store the results
        vector<int> answer;

        // Determine the number of smaller elements for each element in nums
        for (int number : nums) {
            // The count at the index 'number' of the prefix sum will be the
            // number of elements smaller than 'number'
            answer.push_back(count[number]);
        }

        // Return the complete answer
        return answer;
    }
};
```

Typescript Solution

```
function smallerNumbersThanCurrent(nums: number[]): number[] {
    // Initialize a frequency array with 102 elements (to cover numbers 0-100 inclusive and an extra one for indexing purposes).
    const frequencyArray: number[] = new Array(102).fill(0);

    // Populate the frequency array with the count of each number in the input array (shifted by one index).
    for (const num of nums) {
        ++frequencyArray[num + 1];
    }

    // Transform the frequency array into a prefix sum array.
    // Each element at index i now holds the total count of numbers less than i-1.
    for (let i = 1; i < frequencyArray.length; ++i) {
        frequencyArray[i] += frequencyArray[i - 1];
    }

    // The length of the input array.
    const length0fNums = nums.length;

    // Initialize an array to hold the answer.
    const result: number[] = new Array(length0fNums);

    // For each number in the input array, find the count of numbers smaller than itself using the prefix sum array.
    for (let i = 0; i < length0fNums; ++i) {
        result[i] = frequencyArray[nums[i]];
    }

    // Return the resulting array with counts for each number.
    return result;
}
```

Time and Space Complexity

The time complexity of the code is `O(n + m)`, where `n` is the number of elements in the input `nums` list and `m` is the range of numbers in `nums`. Here, the range `m` is fixed at 101 since we are using a count array `cnt` of size 102 (the extra one is to accommodate the 0 indexing and the fact that we started from `x + 1`). The time complexity breaks down as follows:

- `O(n)` for the first for-loop where we iterate over the `nums` list and update the `cnt` array.
- `O(m)` for the `accumulate` function which iterates over the `cnt` array of fixed size 102.
- `O(n)` for the list comprehension that constructs the result list by iterating over the `nums` list again.

Thus, the overall time complexity is dominated by the terms `O(n)` and `O(m)`, resulting in `O(n + m)`.

The space complexity of the code is `O(m)`, which comes from the `cnt` array of fixed size 102, and the `s` array, which will have the same size due to the use of the `accumulate` function on `cnt`. The space required for the output list is not counted towards the space complexity as it is required for the output of the function. Since `m` is a fixed constant (102), the space complexity can also be considered `O(1)`, constant space complexity.