923. 3Sum With Multiplicity

Medium | Array | Hash Table | Two Pointers | Counting | Sorting | Leetcode Link

Problem Description

The problem requires us to find the number of distinct triplets `(i, j, k)` in an integer array `arr` where the condition `i < j < k` is satisfied, and the sum of the elements at these indices is equal to a given `target` integer. Specifically, `arr[i] + arr[j] + arr[k] == target` should hold true for the counted triplets. Since the number of such triplets can be quite large, we are asked to return the answer modulo `10^9 + 7` to keep the output within integer value limits.

Intuition

The key to solving this problem is to consider how we can traverse the array to find all the valid triplets efficiently. We want to avoid checking all possible triplets due to the high time complexity that approach would imply.

The algorithm usually starts by counting the occurrences of each number in the array using a Counter, which is a special kind of dictionary in Python. This allows us to know how many times each number appears in the array without traversing it multiple times.

Once we have these counts, we iterate through the array for the second number of the triplet. For each possible second number `b` at index `j`, we decrease its count by 1 to ensure that we do not use the same instance of `b` when looking for the third number.

Next, we traverse the array again up to the second number's index `j` to find every possible first number `a`. With both `a` and `b` known, we calculate the required third number `c` by subtracting the sum of `a` and `b` from the `target`.

If `c` is present in our Counter (which means it's somewhere in the original array), we can form a triplet (`a`, `b`, `c`) that sums up to `target`. We then add the count of `c` from our Counter to our answer, since there are as many possibilities to form a triplet with `a` and `b` as the count of `c`. It's important to use modulo with `10^9 + 7` during each addition to keep the number within bounds.

The iteration is cleverly structured to ensure that each element is used according to its occurrence and that `i < j < k` always holds true by managing the index and counts carefully.

Solution Approach

The solution uses a combination of a hashmap (in Python, a `Counter`) and a two-pointer approach to find the valid triplets.

The algorithm goes as follows:

1. A `Counter` (which is a specialized hashmap/dictionary in Python) is initialized to count the occurrences of the elements in the given `arr`. This data structure allows for O(1) access to the count of each element, which is essential for efficient computation of the number of triplets.
2. We define a variable `ans` to keep the running total of the number of valid triplets.
3. The `mod` variable is set to `10**9 + 7` to ensure that we perform all our arithmetic operations modulo this number.
4. We loop through each element `b` of the array using its index `j`. This element is considered the second element of our potential triplet.
5. Before starting the inner loop, we decrease the count of `b` in the `Counter` by one. This ensures that we don't count the same element `b` twice when looking for the third element of the triplet.
6. An inner loop runs through the array up to the current index `j`, selecting each element `a` as a candidate for the first element of the triplet. The index `i` is implicit in this loop.
7. We then calculate the required third element `c` of the triplet by subtracting the sum of `a` and `b` from the `target`, i.e., `c = target - a - b`.
8. The total count of valid triplets is then incremented by the count of `c` from the Counter if `c` is present. We use modulo `mod` to keep the answer within the range of valid integers.
9. Finally, we return `ans` as the result.

The algorithm ensures that no element is used more often than it appears in the array, and the `i < j < k` condition is naturally upheld by the two nested loops and the management of the `Counter`.

The use of the `Counter` and looping through the array only once per element significantly reduces the computational complexity compared to checking all possible triplets directly. This pattern is a common approach in problems involving counting specific arrangements or subsets in an array, especially when the array elements are bound to certain conditions.

Example Walkthrough

Let's consider an example with `arr = [1, 1, 2, 2, 3, 3]` and `target = 6`. We want to find all unique triplets `i, j, k` (with `i < j < k`) such that `arr[i] + arr[j] + arr[k] == target`.

1. We begin by initializing a Counter to count the occurrences of each element in `arr`. The Counter will look like this: `{1:2, 2:2, 3:2}` which reflects that each number 1, 2, and 3 occurs twice in the array.
2. Set `ans = 0` to keep track of the total number of valid triplets and `mod = 10**9 + 7` for modulo operations.
3. We now look for the second number `b` for all triplets by iterating through `arr`. Consider `j=2` where `b = arr[j] = 2`.
4. We decrement the count of `b` in `Counter` by 1. Now the Counter will look like this: `{1:2, 2:1, 3:2}`.
5. We start the inner loop to select `a` as the first element of the triplet. We iterate from `start` to `j-1`, in this case, from `0` to `1`.
6. For `a = arr[0] = 1` at `i=0`, we calculate the needed `c = target - a - b = 6 - 1 - 2 = 3`.
7. The count of `c` in the Counter is 2, which means we can form two triplets `(1, 2, 3)` with `i=0, j=2`. We increment `ans` by the count of `c` in Counter. So, `ans = (ans + 2) % mod`.
8. For `a = arr[1] = 1` at `i=1`, we calculate `c = 6 - 1 - 2 = 3` again. Since `i < j`, it's valid and we have not used the same `a` as before. `ans` is updated to `ans = (ans + 2) % mod`.
9. Continue this process for every `j` from `0` to `len(arr)`, and you'll count all valid triplets.
10. After finishing, `ans` is the total number of valid triplets.

For this example, the possible triplets are two instances of (1, 2, 3) using the first 1 and first 2, two using the first 1 and second 2, two using the second 1 and first 2, and two using the second 1 and second 2. These are counted as four unique triplets because the pairs (1, 2) are distinct by their positions. There's no triplet using any two '3's because that would require `i` not to be less than `j`. Therefore, `ans = 4`.

Remember that in a real problem with a large `arr`, not all triples `(a, b, c)` will be unique because of duplicate values, and there will be a variable number of contributions to `ans` from each triplet depending on how many times each value occurs.

Python Solution

```python
from collections import Counter

class Solution:
    def threeSumMulti(self, arr: List[int], target: int) -> int:
        # Create a counter to keep track of occurrences of each number in the array
        count = Counter(arr)

        # Initialize answer to 0
        answer = 0

        # Define the modulo value for large numbers to handle overflow
        modulo = 10**9 + 7

        # Iterate over the array using index and value
        for index, second_value in enumerate(arr):
            # Decrement the count of the current element to avoid overcounting
            count[second_value] -= 1

            # Iterate over the elements up to the current index
            for i in range(index):
                first_value = arr[i]  # Get the current first value

                # Calculate the third value that would make the triplet sum to the target
                third_value = target - first_value - second_value

                # Add the number of occurrences of the third value to the answer
                # Use modulo to avoid overflow
                answer = (answer + count[third_value]) % modulo

        # Return the total number of triplets that sum up to the target
        return answer
```

Here's a breakdown of the changes made:

1. Imported `List` from `typing` to use type hints for list parameters.
2. Replaced `cnt` with `count` for more clarity that this variable represents occurrences of numbers.
3. Replaced `j` and `i` with `index` and `i` respectively for better readability in loops.
4. Renamed `a` and `b` to `first_value` and `second_value`, while `c` to `third_value`, to clearly distinguish the triplet elements.
5. Added comments explaining each portion of the code, outlining what each section does and why certain operations are performed, such as the purpose of the modulo operation.
6. Ensured that the method names and the logic of the function remained unchanged.

To complete this snippet, you'll need to add the following import if not already at the top of your file:

```python
from typing import List
```

Java Solution

```java
class Solution {
    // Define the modulo constant for taking modulus
    private static final int MOD = 1_000_000_007; // 1e9 + 7 is represented as 1000000007

    public int threeSumMulti(int[] arr, int target) {
        int[] count = new int[101]; // Array to store count of each number, considering constraint 0 <= arr[i] <= 100
        // Populate the count array with the frequency of each value in arr
        for (int num : arr) {
            ++count[num];
        }
        long ans = 0; // To store the result, using long to avoid integer overflow before taking the modulus

        // Iterate through all elements in arr to find triplets
        for (int j = 0; j < arr.length; ++j) {
            int second = arr[j]; // The second element in the triplet
            --count[second]; // Decrement count since this number is being used in the current triplet

            // Iterate from the start of the array to the current index 'j'
            for (int i = 0; i < j; ++i) {
                int first = arr[i]; // The first element in the triplet
                int third = target - first - second; // Calculate the third element

                // Check if third element is within range and add the count to the answer
                if (third >= 0 && third <= 100) {
                    ans = (ans + count[third]) % MOD; // Use the modulo to avoid overflow and get the correct result
                }
            }
        }
        // Cast and return the final answer as an integer
        return (int) ans;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Modulo constant for the problem
    const int MOD = 1e9 + 7;

    // Function to calculate the number of triplets that sum up to the target
    int threeSumMulti(vector<int>& arr, int target) {
        // Array to store the count of each number in the range [0, 100]
        int count[101] = {0};

        // Populating the count array
        for (int num : arr) {
            ++count[num];
        }

        // Variable to store the result
        long answer = 0;

        // Iterate through each element in the array to use it as the second element of the triplet
        for (int j = 0; j < arr.size(); ++j) {
            // Select the current element as the second element of the triplet
            int secondElement = arr[j];
            // Decrement the count as this element is considering for pairing
            --count[secondElement];

            // Iterate through elements up to the current second element to find the first element
            for (int i = 0; i < j; ++i) {
                int firstElement = arr[i];
                // Calculate the required third element to meet the target
                int thirdElement = target - firstElement - secondElement;

                // If the third element is within the valid range
                if (thirdElement >= 0 && thirdElement <= 100) {
                    // Add the count of the third element to the answer
                    answer += count[thirdElement];
                    // Ensure answer is within the MOD range
                    answer %= MOD;
                }
            }
        }

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

Typescript Solution

```typescript
// Constant for modulo operations
const MOD: number = 1e9 + 7;

// This function calculates the number of triplets in the array that sum up to the target value
function threeSumMulti(arr: number[], target: number): number {
    // Array to store the count of occurrence for each number within the range [0, 100]
    let count: number[] = new Array(101).fill(0);

    // Populate the count array with the number of occurrences of each element
    for (let num of arr) {
        count[num]++;
    }

    // Variable to store the result
    let answer: number = 0;

    // Iterate through each element in the array to use it as the second element of the triplet
    for (let j = 0; j < arr.length; j++) {
        // The current element is selected as the second element of the triplet
        let secondElement: number = arr[j];
        // Decrement the count for this element as it is being considered for pairing
        count[secondElement]--;

        // Iterate over all possible first elements up to the current second element
        for (let i = 0; i < j; i++) {
            let firstElement: number = arr[i];
            // Calculate the required third element to meet the target sum
            let thirdElement: number = target - firstElement - secondElement;

            // Check if the third element is within the valid range [0, 100]
            if (thirdElement >= 0 && thirdElement <= 100) {
                // Add the count of the third element to the answer
                answer += count[thirdElement];
                // Ensure answer remains within the range specified by MOD
                answer %= MOD;
            }
        }
    }

    // Return the final computed answer
    return answer;
}

// Example usage of the function
// let arrExample = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5];
// let targetExample = 8;
// let result = threeSumMulti(arrExample, targetExample);
// console.log(result); // Output the result
```

Time and Space Complexity

The time complexity of the provided Python code snippet is `O(n^2)` where `n` is the number of elements in the input list `arr`. This complexity arises because there is a nested for-loop where the outer loop runs through the elements of `arr` (after decrementing the count of the current element), and the inner loop iterates up to the current index `j` of the outer loop. For each pair (`a`, `b`), the code looks up the count of the third element `c` that is needed to sum up to the target. Even though the lookup in the counter is `O(1)`, the nested loops result in quadratic complexity.

The space complexity of the code is `O(m)` where `m` is the number of unique elements in the input list `arr`. This complexity is due to the use of a `Counter` to store frequencies of all unique elements in `arr`. The space taken by the counter will directly depend on the number of unique values.