2198. Number of Single Divisor Triplets

Medium Math Leetcode Link

Problem Description

You are tasked with finding the number of special triplets in a given array of positive integers, where the array is indexed starting at 0. A triplet consists of three distinct indices `(i, j, k)`. This triplet is considered special - termed as a *single divisor triplet* - if the sum of the numbers at these indices, specifically `nums[i] + nums[j] + nums[k]`, is divisible by exactly one of the three numbers `nums[i]`, `nums[j]`, or `nums[k]`. The goal is to count how many such triplets exist in the provided array.

Intuition

To solve this problem, we should first understand that checking every possible triplet would be quite inefficient if we did so naïvely, since that requires looking at all possible combinations of numbers in the array. However, by utilizing the `Counter` class from Python's `collections` module, we can improve the efficiency when certain numbers are repeated in `nums`.

The solution approach involves the following steps:

1. We count the occurrences of each number in the array utilizing the `Counter` class.
2. Once we have the counts of each number, we iterate through the array thrice – simulating the selection of three distinct elements, denoted by `a`, `b`, and `c`, for our triplet. If `a`, `b`, and `c` are the same number, we skip to avoid checking the same combination again. It's important to note that `a`, `b`, or `c` could be the same number but from different indices.
3. For each triplet, we check if the sum of `a`, `b`, and `c` (`s = a + b + c`) is divisible by only one of the three numbers. The `check` function helps in this regard by returning `True` if exactly one divisibility condition is met.
4. If the `check` function returns `True`, we must account for the occurrences of each number. Since the numbers might be duplicated in `nums`, we need to calculate the number of triplets that we can form with the given counts of `a`, `b`, and `c`. We do this by considering the different scenarios:
   - If two numbers are the same and the third is different, we use combinations like `cnt1 * (cnt1 - 1) * cnt3` to reflect the number of possible triplets.
   - If all three numbers are different, we simply multiply the counts: `cnt1 * cnt2 * cnt3`.
5. We sum up these counts to get the total number of single divisor triplets.

Using the `Counter` allows us to avoid unnecessary duplicate computations and vastly reduces the number of required operations, especially when dealing with an array containing many repeated numbers.

Solution Approach

The implementation of the solution leverages the `Counter` class from Python's `collections` module to efficiently track the number of times each number appears in the input array `nums`. This is crucial to avoid duplicate computations when numbers are repeated. Here's a step-by-step breakdown of the algorithm:

1. A `Counter` object is created for the array `nums` to get the count of each unique number. For example, if `nums` is `[1, 2, 2, 3]`, the `Counter` object would be `{1: 1, 2: 2, 3: 1}`.

2. We define a helper function `check(a, b, c)` which calculates the sum `s` of the parameters `a`, `b`, and `c`. It then checks if `s` is divisible by exactly one of the values `a`, `b`, or `c` by using the modulo operation `s % x` for each and counts the occurrences of `0` (true divisibility) using a generator expression. If this count is `1`, it returns `True`; otherwise, it returns `False`.

3. The solution then iterates through every possible combination of the counts of each unique number, looking for ways to form triplets. The outer loop picks a number, denoted as `a`, from the `Counter`. The nested loops pick additional numbers, `b` and `c`.

4. For each triplet `(a, b, c)`, it calls the `check` function to verify whether it's a single divisor triplet. If it is, the count of possible single divisor triplets that can be made with `a`, `b`, and `c` is calculated. This depends on whether `a`, `b`, and `c` are the same or different. For example:

   - If `a` is the same as `b` but different from `c`, the number of triplets is `cnt1 * (cnt1 - 1) / 2 * cnt3` because there are `cnt1 * (cnt1 - 1) / 2` ways to choose two numbers `a` and `b` from `cnt1` occurrences, and `cnt3` ways to choose `c`.
   - If all three numbers are different, the number of triplets will be `cnt1 * cnt2 * cnt3` since there are `cnt1` ways to choose `a`, `cnt2` ways to choose `b`, and `cnt3` ways to choose `c`.
5. It's essential to handle overcounting carefully; the solution ensures that each triplet of numbers is counted exactly once, with attention to the distinction between using combinations vs. permutations when selecting elements from counts.

6. As each valid triplet is found, its count is added to an accumulator variable, `ans`, which finally holds the total count of single divisor triplets.

7. After checking all possible combinations of numbers from the `Counter`, the final result is returned, which is the accumulated count of all valid triplets.

Thus, the algorithm makes smart use of combinatorics and the properties of divisibility to compute the desired count without exhaustively examining each distinct triplet in the original array.

Example Walkthrough

Let's walk through an example using the provided solution approach to get a clearer understanding of how the algorithm works.

Given Array

Consider the array `nums = [1, 2, 2, 3, 4]`.

Step 1: Counter Object Creation

We create a `Counter` object from `nums` which gives us `{1: 1, 2: 2, 3: 1, 4: 1}`.

Step 2: Define the `check` function

The `check(a, b, c)` function takes three numbers and returns `True` if the sum `s = a + b + c` is divisible by exactly one of `a`, `b`, or `c`.

Step 3: Iterate over unique combinations

We start iterating over every unique combination using the counts from the `Counter` object:

- First we pick `a = 1` from the counter.
- Then we pick `b = 2`. Since we have two 2's, `cnt2 = 2`.
- Next, we pick `c = 4`. Since there is only one 4, `cnt3 = 1`.

Step 4: Apply the `check` function and calculate the triplet count

Using the `check` function, we find that `1 + 2 + 4 = 7` is not divisible by any number exactly once, so we discard this triplet.

Next, we try `a = 2`, `b = 2`, and `c = 4` (where `a` and `b` are the same). The sum is `2 + 2 + 4 = 8`, and this sum is divisible by `4` only, which makes it a valid triplet. Since `a` and `b` are the same, there are `cnt2 * (cnt2 - 1) / 2` ways to pick `a` and `b`, and `cnt3` ways to pick `c`. This gives us a count of `(2 * (2 - 1) / 2) * 1 = 1`.

We continue this process for other possible combinations.

Step 5: Add valid triplets count to the accumulator

Each time we find a valid triplet combination, we increment our accumulator `ans` with the count of its occurrences.

Step 6: Final result

After iterating through all unique number combinations from the `Counter`, we sum up all valid triplet counts we have found. Suppose we found another valid triplet using different numbers, let's say we found `1` more valid triplet in the process. Our total count `ans` would be `1 + 1 = 2`.

Thus, for the given array `[1, 2, 2, 3, 4]`, our final result is `2`, meaning there are two single divisor triplets whose sum is divisible exactly by one of the three numbers.

Python Solution

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

class Solution:
    def singleDivisorTriplet(self, nums: List[int]) -> int:
        # A helper function to check if only one of a, b, or c is a divisor of their sum
        def is_single_divisor_triplet(a: int, b: int, c: int) -> bool:
            sum_triplet = a + b + c
            return sum(sum_triplet % x == 0 for x in [a, b, c]) == 1

        # Count the occurrences of each number in the list
        num_counter = Counter(nums)
        count_triplets = 0  # Initialize the count of valid triplets

        # Iterate over all possible combinations of a, b, and c
        for a, count_a in num_counter.items():
            for b, count_b in num_counter.items():
                for c, count_c in num_counter.items():
                    # Check if the current combination is a single divisor triplet
                    if is_single_divisor_triplet(a, b, c):
                        # If any two numbers are equal, adjust the count accordingly
                        if a == b and b == c:
                            # All numbers are the same
                            count_triplets += count_a * (count_a - 1) * (count_a - 2) // 6
                        elif a == b:
                            count_triplets += count_a * (count_a - 1) // 2 * count_c
                        elif a == c:
                            count_triplets += count_a * (count_a - 1) // 2 * count_b
                        elif b == c:
                            count_triplets += count_a * count_b * (count_b - 1) // 2
                        else:
                            # All numbers are different
                            count_triplets += count_a * count_b * count_c

        # Since each triplet is counted 6 times (all permutations), we divide by 6
        return count_triplets // 6

# The above code can be used as follows:
# solution = Solution()
# result = solution.singleDivisorTriplet([nums])

```

Java Solution

```java
class Solution {
    public long singleDivisorTriplet(int[] nums) {
        // Initialize a counter array to count the occurrences of each number up to 100.
        int[] count = new int[101];
        for (int num : nums) {
            count[num]++;
        }

        // Initialize a variable to store the answer.
        long answer = 0;

        // Iterate through all possible trios and calculate the sum.
        for (int i = 1; i <= 100; i++) {
            for (int j = 1; j <= 100; j++) {
                for (int k = 1; k <= 100; k++) {
                    // Get the count of each number in the trio.
                    int countI = count[i];
                    int countJ = count[j];
                    int countK = count[k];

                    int sum = i + j + k;

                    // Check if exactly one number divides the sum.
                    int divisorCount = 0;
                    divisorCount += sum % i == 0 ? 1 : 0;
                    divisorCount += sum % j == 0 ? 1 : 0;
                    divisorCount += sum % k == 0 ? 1 : 0;

                    // If not exactly one divisor, then continue to the next trio.
                    if (divisorCount != 1) {
                        continue;
                    }

                    // Calculate the number of valid configurations.
                    if (i == j && i == k) {
                        // All three numbers are the same.
                        answer += (long) countI * (countI - 1) * (countI - 2) / 6;
                    } else if (i == j) {
                        // i and j are the same, but k is different.
                        answer += (long) countI * (countI - 1) / 2 * countK;
                    } else if (i == k) {
                        // i and k are the same, but j is different.
                        answer += (long) countI * (countI - 1) / 2 * countJ;
                    } else if (j == k) {
                        // j and k are the same, but i is different.
                        answer += (long) countI * countJ * (countJ - 1) / 2;
                    } else {
                        // All three numbers are different.
                        answer += (long) countI * countJ * countK;
                    }
                }
            }
        }

        // Return the computed answer.
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    long long singleDivisorTriplet(vector<int>& nums) {
        // Counter array to keep the frequency of each number from 1 to 100
        vector<int> frequency(101, 0);
        // Fill the counter with frequencies of numbers in the input vector nums
        for (int number : nums) {
            frequency[number]++;
        }

        long long answer = 0; // Variable to store the final count of triplets

        // Check all possible combinations of three numbers (i, j, k)
        for (int i = 1; i <= 100; ++i) {
            for (int j = 1; j <= 100; ++j) {
                for (int k = 1; k <= 100; ++k) {
                    // Fetch the frequencies of the current combination
                    int freq_i = frequency[i];
                    int freq_j = frequency[j];
                    int freq_k = frequency[k];

                    // Calculate the sum of the triplet
                    int sum = i + j + k;
                    // Count the number of divisors the sum has from the triplet
                    int divisorCount = (sum % i == 0) + (sum % j == 0) + (sum % k == 0);

                    // We are only interested in triplets where exactly one of the numbers divides the sum
                    if (divisorCount != 1) continue;

                    // Now, handle the cases depending on the uniqueness of i, j, k
                    if (i == j && i == k) {
                        // All numbers are the same
                        answer += freq_i * (freq_i - 1) * (freq_i - 2) / 6; // Combination of nC3
                    } else if (i == j) {
                        // i and j are the same, k is different
                        answer += (long long) freq_i * (freq_i - 1) / 2 * freq_k; // Combination of nC2 times the count of k
                    } else if (i == k) {
                        // i and k are the same, j is different
                        answer += (long long) freq_i * (freq_i - 1) / 2 * freq_j; // Combination of nC2 times the count of j
                    } else if (j == k) {
                        // j and k are the same, i is different
                        answer += (long long) freq_j * (freq_j - 1) / 2 * freq_i; // Combination of nC2 times the count of i
                    } else {
                        // i, j, and k are all different
                        answer += (long long) freq_i * freq_j * freq_k; // Just the product of the counts
                    }
                }
            }
        }

        // Return the total count of single divisor triplets
        return answer;
    }
};

```

Typescript Solution

```typescript
const MAX_NUM = 100;

// Function to count single divisor triplets
function singleDivisorTriplet(nums: number[]): number {
    // Initialize a counter array to keep the frequency of each number from 1 to 100
    const frequency: number[] = new Array(MAX_NUM + 1).fill(0);

    // Fill the counter with frequencies of numbers in the input array nums
    nums.forEach(number => {
        frequency[number]++;
    });

    let answer: number = 0; // Variable to store the final count of triplets

    // Check all possible combinations of three numbers (i, j, k)
    for (let i = 1; i <= MAX_NUM; ++i) {
        for (let j = 1; j <= MAX_NUM; ++j) {
            for (let k = 1; k <= MAX_NUM; ++k) {
                // Fetch the frequencies of the current combination
                const freqI: number = frequency[i];
                const freqJ: number = frequency[j];
                const freqK: number = frequency[k];

                // Calculate the sum of the triplet
                const sum: number = i + j + k;
                // Count the number of divisors the sum has from the triplet
                const divisorCount: number = (sum % i === 0 ? 1 : 0)
                                           + (sum % j === 0 ? 1 : 0)
                                           + (sum % k === 0 ? 1 : 0);

                // We are only interested in triplets where exactly one of the numbers divides the sum
                if (divisorCount !== 1) continue;

                // Handle the cases depending on the uniqueness of i, j, k
                if (i === j && i === k) {
                    // All numbers are the same (choose 3 from freqI)
                    answer += freqI * (freqI - 1) * (freqI - 2) / 6;
                } else if (i === j) {
                    // i and j are the same, k is different (choose 2 from freqI and multiply by freqK)
                    answer += freqI * (freqI - 1) / 2 * freqK;
                } else if (i === k) {
                    // i and k are the same, j is different (choose 2 from freqI and multiply by freqJ)
                    answer += freqI * (freqI - 1) / 2 * freqJ;
                } else if (j === k) {
                    // j and k are the same, i is different (choose 2 from freqJ and multiply by freqI)
                    answer += freqJ * (freqJ - 1) / 2 * freqI;
                } else {
                    // i, j, and k are all different (multiply counts of freqI, freqJ, freqK)
                    answer += freqI * freqJ * freqK;
                }
            }
        }
    }

    // Return the total count of single divisor triplets
    return answer;
}

// Sample usage
const sampleNums: number[] = [1, 2, 3];
const sampleResult: number = singleDivisorTriplet(sampleNums);
console.log(sampleResult);  // Output the result

```

Time and Space Complexity

The given Python code aims to count the number of triplets in a list where exactly one of the three numbers is a divisor of the sum of the triplet.

Time Complexity:

The time complexity of the algorithm can be analyzed based on the nested loops and the operations performed within them.

We iterate over all unique elements in `nums` up to three times due to the nested loops. If there are `k` unique elements, then the triple-nested loop runs `O(k^3)` times. The `check` function is called inside the innermost loop, which operates in `O(1)` time because it simply performs a fixed series of modulo operations and comparisons.

For each unique triplet `(a, b, c)`, we calculate the number of such triplets considering their counts in the original list. The increments to `ans` involve multiplication, which is an `O(1)` operation.

Therefore, the overall time complexity is `O(k^3)`.

Space Complexity:

The space complexity can be determined by the additional space used by the algorithm, which is not part of the input.

- The `counter` variable is a Counter object, which stores the frequency of each unique number in `nums`. It occupies `O(k)` space, where `k` is the number of unique numbers in `nums`.

- Other than that, the space usage is constant `O(1)` because we only use a fixed number of extra variables (`ans`, `a`, `b`, `c`, `cnt1`, `cnt2`, `cnt3`, and space for the `check` function's return value).

Thus, the overall space complexity is `O(k)`.