2964. Number of Divisible Triplet Sums

Medium Array Hash Table

Problem Description

The challenge is to find out the number of unique triplets within an array `nums`, where each triplet consists of different indices `(i, j, k)` that follow a specific criterion. This criterion requires that the indices form a strictly increasing sequence (`i < j < k`) and the sum of the elements at these indices is divisible by a given integer `d`. In other words, `nums[i] + nums[j] + nums[k]` should be an integer multiple of `d`.

To tackle this problem, we need to identify combinations of three different array elements whose indices are in ascending order and check for the divisibility of their sum by `d`. It's a computational challenge that requires efficient enumeration of the possible triplets to avoid a brute force approach that would take too long.

Intuition

To optimize the process of finding these triplets, the solution leverages a hash table strategy to avoid redundant calculations. The main idea behind the solution is to use a hash table, denoted as `cnt`, to keep track of how many times each remainder (when elements of `nums` are divided by `d`) occurs up to the current index being considered. As we iterate through the array, we calculate what remainder we would need from the `nums[i]` (where `i < j`) to ensure that the sum of `nums[i]`, `nums[j]`, and `nums[k]` is divisible by `d`.

Here's the thinking process:

1. As we move through the array, with each `nums[j]`, we look ahead to all future `nums[k]` where `k > j`. For each of these pairs, we calculate the remainder that would be needed by a preceding `nums[i]` to make the sum of `nums[i] + nums[j] + nums[k]` divisible by `d`. We use the aforementioned hash table to quickly check the count of such potential `nums[i]` elements.
2. We then add the count from our hash table to our answer (accumulating the number of triplets that meet our criteria up to the current `j`).
3. After considering pairs of `nums[j]` and `nums[k]`, we increment the count of the remainder of `nums[j]` itself in the hash table before continuing to the next `j`.

Solution Approach

The solution to this LeetCode problem is centered around a clever use of a hash table, specifically a `defaultdict` from Python's collections module, which allows us to automatically initialize missing keys with an integer (initialized to `0` in this case). This helps us to track the frequency of remained parts of numbers modulo `d`.

The algorithm can be broken down into the following steps:

1. First, we iterate over the elements of the `nums` array while calculating the remainder when each element `nums[j]` is divided by `d` (i.e., `nums[j] % d`). We use this to determine what the corresponding `nums[i]`'s remainder should be in order to have `(nums[i] + nums[j] + nums[k]) % d == 0`.
2. For any given index `j`, we look ahead to the elements `nums[k]` for all `k` such that `k > j` and calculate `x`, which is equal to `(d - (nums[j] + nums[k]) % d) % d`. This represents the remainder we need from some `nums[i]` (where `i < j`) so that the sum of the three elements is divisible by `d`.
3. The calculated `x` is then used to check in our `cnt` hash table how many times we've seen such a remainder before index `j`. We sum up these occurrences in a variable `ans`, which ultimately holds the total number of triplets that satisfy the problem's condition.
4. Before we move on to the next `j`, we increase the count of `nums[j]`'s remainder in the hash table by 1, i.e., `cnt[nums[j] % d] += 1`, representing that we have seen another occurrence of this particular remainder.
5. Once we exhaust all possibilities for `j` and its corresponding `k`, the variable `ans` will hold the correct answer, which is then returned.

One of the clever patterns employed in this solution is the recognition that for triplets `(i, j, k)` to satisfy our condition, it is not necessary to track each `i` explicitly. By using remainders and counting their occurrences, we implicitly handle all possible `i` candidates while iterating over `j` and `k`. This avoids the need for a full, expensive three-level loop and thus significantly improves the time complexity.

The use of mathematics to track the needed remainder part instead of the raw sum also reduces the memory complexity as we only need to store counts for each possible remainder range from `0` to `d-1`. This is much smaller than the potential range of the sums if `d` is small relative to the values in `nums`.

This algorithm runs in `O(n^2)` time complexity, with `n` being the size of the array `nums`, which is much more efficient than the brute force `O(n^3)`. The space complexity is `O(d)` due to the hash table storing at most `d` different remainders.

Example Walkthrough

Let's illustrate the solution approach using an example:

Consider `nums = [2, 3, 5, 7, 11]` and `d = 5`.

We initiate a `defaultdict(int)` which will serve as the `cnt` hash table to store the counts of seen remainders.

As there are no values in `cnt` yet, all remainders start with a count of 0.

We start by iterating through the array:

1. For `nums[0] = 2`, the remainder when divided by `d` is 2.
   - There are no previous elements, so we just move to the next index.
   - Update `cnt[2]` to 1 (since 2 % 5 = 2).
2. At `nums[1] = 3`, the remainder is 3.
   - We are looking for a triplet that includes `nums[i]`, `nums[j]=3`, and some future `nums[k]`.
   - Update `cnt[3]` to 1 (3 % 5 = 3).
3. Moving on to `nums[2] = 5`, the remainder is 0.
   - Now, let's look ahead for future `k` values:
     - For a potential `k` with `nums[k] = 7` (next element), we need a remainder `x` (needed from some previous `nums[i]`), which is `(5 - (3 + 7) % 5) % 5 = 0`. We look into `cnt` and see that `cnt[0] = 0` (0 hasn't been seen yet as a remainder before index `2`), so no triplets can be formed here.
   - Update `cnt[0]` to 1 (5 % 5 = 0).
4. At `nums[3] = 7`, the remainder when divided by `d` is 2.
   - Let's look ahead:
     - For a future `k` with `nums[k] = 11` (next element), we need a remainder `x` from some previous `nums[i]` which is `(5 - (3 + 11) % 5) % 5 = 1`. We don't have any elements that left a remainder of 1 until now (`cnt[1]` is 0), so no triplet can be formed.
   - Update `cnt[2]` to 2.
5. Finally, at `nums[4] = 11`, the remainder is 1.
   - No need to look ahead because `11` is the last element.
   - Update `cnt[1]` to 1 (11 % 5 = 1).

Since there are no triplets that satisfy `(nums[i] + nums[j] + nums[k]) % 5 == 0`, our answer thus far is `0`.

In this example, we failed to find any valid triplets, but the process demonstrates how we would systematically check for them using the remainders and the `cnt` hash table to avoid redundancy. In cases where `nums` contains the right combinations, the `cnt` table would help us tally up the valid triplet counts quickly and efficiently.

Solution Implementation

Python

```python
from collections import defaultdict

class Solution:
    def divisibleTripletCount(self, nums: List[int], divisor: int) -> int:
        # Dictionary to store the frequency of remainders
        remainder_count = defaultdict(int)

        # Initialize the count of valid triplets
        valid_triplet_count = 0

        # Length of the input list
        n = len(nums)

        # Loop over the list to find valid triplets
        for j in range(n):
            for k in range(j + 1, n):
                # Compute the remainder needed from the third element
                # for the sum of the triplet to be divisible by 'divisor'
                needed_remainder = (divisor - (nums[j] + nums[k]) % divisor) % divisor

                # Add the count of numbers previously encountered with the needed remainder
                valid_triplet_count += remainder_count[needed_remainder]

            # Increment the count of the remainder for the current element
            remainder_count[nums[j] % divisor] += 1

        # Return the total count of valid triplets
        return valid_triplet_count
```

Java

```java
class Solution {

    /**
     * Counts the number of divisible triplets in the given array.
     * A treeplit is a sequence of three numbers (a, b, c), such that (a + b + c) % d == 0.
     *
     * @param nums The input array of integers.
     * @param d    The divisor for checking divisibility.
     * @return The count of divisible triplets.
     */
    public int divisibleTripletCount(int[] nums, int d) {
        // Map to store frequency counts of numbers modulo d
        Map<Integer, Integer> frequencyCounts = new HashMap<>();
        // The answer (count of divisible triplets)
        int answer = 0;
        // Length of the nums array
        int length = nums.length;

        // Iterate through the pairs (j, k) where j < k
        for (int j = 0; j < length; ++j) {
            for (int k = j + 1; k < length; ++k) {
                // Calculate the modulo of the negative sum of nums[j] + nums[k]
                // This is the number needed to complete triplet to be divisible by d
                int neededModulo = (d - (nums[j] + nums[k]) % d) % d;
                // Add to answer the count of numbers that have the neededModulo
                answer += frequencyCounts.getOrDefault(neededModulo, 0);
            }
            // Update the frequencyCounts map with the current number's modulo
            frequencyCounts.merge(nums[j] % d, 1, Integer::sum);
        }

        // Return the total count of divisible triplets
        return answer;
    }
}
```

C++

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

class Solution {
public:
    // Function to count the number of triplets such that sum of two elements is divisible by 'd'.
    int divisibleTripletCount(vector<int>& nums, int d) {
        // 'counts' is used to store the frequency of elements mod 'd'.
        unordered_map<int, int> counts;
        int answer = 0;
        int n = nums.size(); // Length of the array.

        // Iterate over all pairs of numbers in 'nums'.
        for (int j = 0; j < n; ++j) {
            for (int k = j + 1; k < n; ++k) {
                // Calculate the complement that would make the sum of a triplet divisible by 'd'.
                int complement = (d - (nums[j] + nums[k]) % d) % d;

                // Add the count of the complement to the answer.
                answer += counts[complement];
            }
            // For the number at position 'j', increment its frequency.
            counts[nums[j] % d]++;
        }
        return answer; // Return the total count of divisible triplets.
    }
};
```

TypeScript

```typescript
function divisibleTripletCount(nums: number[], divisor: number): number {
    const arrayLength = nums.length; // get the length of the nums array
    const remainderCount: Map<number, number> = new Map(); // map to count occurrences of each remainder
    let tripletCount = 0; // initialize triplet count to zero

    // Iterate over each pair of numbers in the array
    for (let middleIndex = 0; middleIndex < arrayLength; ++middleIndex) {
        for (let lastIndex = middleIndex + 1; lastIndex < arrayLength; ++lastIndex) {
            // calculate the required value to complete the triplet
            const requiredValue = (divisor - ((nums[middleIndex] + nums[lastIndex]) % divisor)) % divisor;
            // increase the count of valid triplets by the amount found in remainderCount
            tripletCount += remainderCount.get(requiredValue) || 0;
        }
        // update the remainder count for the current number
        const currentRemainder = nums[middleIndex] % divisor;
        remainderCount.set(currentRemainder, (remainderCount.get(currentRemainder) || 0) + 1);
    }

    return tripletCount; // return the total count of divisible triplets
}
```


```python
from collections import defaultdict

class Solution:
    def divisibleTripletCount(self, nums: List[int], divisor: int) -> int:
        # Dictionary to store the frequency of remainders
        remainder_count = defaultdict(int)

        # Initialize the count of valid triplets
        valid_triplet_count = 0

        # Length of the input list
        n = len(nums)

        # Loop over the list to find valid triplets
        for j in range(n):
            for k in range(j + 1, n):
                # Compute the remainder needed from the third element
                # for the sum of the triplet to be divisible by 'divisor'
                needed_remainder = (divisor - (nums[j] + nums[k]) % divisor) % divisor

                # Add the count of numbers previously encountered with the needed remainder
                valid_triplet_count += remainder_count[needed_remainder]

            # Increment the count of the remainder for the current element
            remainder_count[nums[j] % divisor] += 1

        # Return the total count of valid triplets
        return valid_triplet_count
```


Time and Space Complexity

The given Python code defines a method `divisibleTripletCount` within a `Solution` class that counts triplets in an array, where the sum of each triplet is divisible by a given integer `d`. The code primarily involves two nested loops: the outer loop iterates over each element `j` of the array, and the inner loop iterates over the elements following `j` (`k`). Here's a breakdown of the complexities:

Time Complexity:

The time complexity is determined by the number of nested iterations over the input list `nums`.

- The outer loop runs for `n` iterations, with `n` being the length of `nums`.
- For each iteration of the outer loop, the inner loop executes `n - j - 1` times, which results in an average case of `n/2` times per iteration of the outer loop.

This creates a total of around `n * (n/2)` comparisons, simplifying to `(n^2)/2` which is in the order of `O(n^2)`. This is because constants are ignored in Big O notation.

Therefore, the time complexity of the code is `O(n^2)`.

Space Complexity:

The space complexity refers to the amount of additional memory used by the program in relation to the input size.

- A `defaultdict(int)` is created to store counts of `nums[j] % d`. In the worst case, we would have to store a number for every unique value of `nums[j] % d`. However, since `% d` creates `d` possible remainders, the `defaultdict` will at most contain `d` entries.
- We also have a few integer variables (`ans`, `n`, `x`), but these do not scale with input size `n`, and thus contribute a constant amount to the space complexity.

The space complexity of the code is therefore dictated by the `defaultdict` size, along with a small constant for the variables used, so it is `O(d)`. However, given that `d` is a single integer value and not related to the size of the input array `nums`, the `d` in the space complexity could be considered a constant factor.

Thus, the space complexity of the code is `O(1)` if `d` is considered constant with respect to `n`. However, since the reference answer suggests that the space complexity is `O(n)`, it seems there might be a presumption that the array might contain up to `n` distinct values modulo `d`, binding the space complexity to the length of the array `nums`. Under this assumption, the space complexity would indeed be `O(n)`.

In conclusion, the time complexity of the code is `O(n^2)`, and the space complexity, depending on the context, is either `O(1)` or `O(n)` based on the aforementioned reasoning.