1534. Count Good Triplets

Easy Array Enumeration

Problem Description

In this problem, we are provided with an array of integers `arr` and three separate integers `a`, `b`, and `c`. Our goal is to determine how many triplets `(arr[i], arr[j], arr[k])` meet certain criteria, where `i`, `j`, and `k` are distinct indices into the array with `i < j < k`.

A triplet is considered "good" if it satisfies the following conditions:

- The absolute difference between `arr[i]` and `arr[j]` is less than or equal to `a`.
- The absolute difference between `arr[j]` and `arr[k]` is less than or equal to `b`.
- The absolute difference between `arr[i]` and `arr[k]` is less than or equal to `c`.

The problem asks us to return the total number of these good triplets.

Intuition

The straightforward way to find all good triplets is to check every possible triplet in the array to see if it meets the criteria. This involves using three nested loops to generate all possible combinations of `i`, `j`, and `k` where `0 <= i < j < k < arr.length`.

Within these loops, we will calculate the absolute differences between `arr[i]`, `arr[j]`, and `arr[k]` and check if they satisfy the conditions related to `a`, `b`, and `c`. If a triplet satisfies all these conditions, we increment a counter.

Solution Approach

The implementation of the solution provided in the reference code uses the brute-force approach, which is a common pattern when dealing with problems that require us to explore all possible combinations of elements to find a particular set that satisfies certain conditions.

Algorithm steps:

1. Initialize a counter `ans` to keep track of the number of good triplets found.
2. Calculate the length `n` of the input array `arr`.
3. Iterate over the array with the first pointer `i`, which runs from the beginning of the array to the third-to-last element.
4. For each `i`, iterate with the second pointer `j`, which runs from one element after `i` to the second-to-last element.
5. For each `j`, iterate with the third pointer `k`, which runs from one element after `j` to the last element.
6. Inside the innermost loop (where `k` is iterating), check if the current triplet `(arr[i], arr[j], arr[k])` is a good triplet by verifying the three conditions with the given `a`, `b`, and `c`:
   - `abs(arr[i] - arr[j]) <= a`
   - `abs(arr[j] - arr[k]) <= b`
   - `abs(arr[i] - arr[k]) <= c`
7. If all three conditions are satisfied, increment the counter `ans`.
8. After all iterations are complete, return the value of `ans`.

This approach does not use any specific data structures or complex algorithms, but rather relies on nested loops to examine each triplet one by one. The only operations involved are arithmetic operations (`-` and `abs`) and simple comparisons (`<=`).

While it is an exhaustive solution, it is not the most efficient in terms of time complexity. The time complexity of this approach is `O(n^3)`, as there are three nested loops each iterating over the array. This solution can be quite slow for larger arrays but is perfectly fine for smaller input sizes where a more sophisticated algorithm might not lead to significant improvements in runtime or when fast and simple programming is required over an optimized, complex solution.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following:

- `arr = [2, 1, 3]`
- `a = 1`
- `b = 1`
- `c = 1`

Now we want to find the number of good triplets that satisfy all the given conditions.

We start by initializing our counter `ans` to 0. Our array length `n` is 3.

1. We begin iterating with pointer `i` starting at 0.
2. For `i = 0` (`arr[i] = 2`), we move on to pointer `j`.
3. We let `j` iterate starting from `i + 1`, which is 1.
4. For `j = 1` (`arr[j] = 1`), we then use pointer `k`.
5. `k` starts iterating from `j + 1`, which is 2.
6. For `k = 2` (`arr[k] = 3`), we now check if `(arr[i], arr[j], arr[k]) = (2, 1, 3)` is a good triplet.
   - The absolute difference between `arr[i]` and `arr[j]` is `|2 - 1| = 1`, which is less than or equal to `a`.
   - The absolute difference between `arr[j]` and `arr[k]` is `|1 - 3| = 2`, so this is not less than or equal to `b` and we can ignore this triplet.

Since there are no other combinations to check, our count `ans` stays at 0. We conclude that there are no good triplets in the array `[2, 1, 3]` with the given `a`, `b`, and `c` values.

Applying the same approach in a larger array would involve more iterations, but the steps would be similar: iterate over each potential triplet and check if they satisfy the conditions.

Solution Implementation

Python

```python
from typing import List  # Import List from typing module for type annotations.

class Solution:
    def count_good_triplets(self, arr: List[int], a: int, b: int, c: int) -> int:
        # Initialize the count of good triplets.
        good_triplets_count = 0

        # Get the length of the array.
        array_length = len(arr)

        # Iterate through each element of the array for the first element of the triplet.
        for first_index in range(array_length):
            # For the second element, start from the next index of the first element.
            for second_index in range(first_index + 1, array_length):
                # For the third element, start from the next index of the second element.
                for third_index in range(second_index + 1, array_length):
                    # Check if the current triplet (arr[first_index], arr[second_index], arr[third_index])
                    # is a good triplet based on the provided conditions.
                    is_good_triplet = (
                        abs(arr[first_index] - arr[second_index]) <= a and
                        abs(arr[second_index] - arr[third_index]) <= b and
                        abs(arr[first_index] - arr[third_index]) <= c
                    )
                    # If it is a good triplet, increment the good_triplets_count.
                    good_triplets_count += is_good_triplet

        # After checking all possible triplets, return the total count of good triplets.
        return good_triplets_count
```

Java

```java
class Solution {
    public int countGoodTriplets(int[] arr, int maxDiffAB, int maxDiffBC, int maxDiffAC) {
        // The length of the input array
        int arrayLength = arr.length;
        // Counter to keep track of the number of "good" triplets
        int goodTripletsCount = 0;

        // Loop over each element for the first value of the triplet
        for (int i = 0; i < arrayLength; ++i) {
            // Loop over each element after the first value for the second value of the triplet
            for (int j = i + 1; j < arrayLength; ++j) {
                // Loop over each element after the second value for the third value of the triplet
                for (int k = j + 1; k < arrayLength; ++k) {
                    // Check the conditions to determine if the triplet (arr[i], arr[j], arr[k]) is "good"
                    if (Math.abs(arr[i] - arr[j]) <= maxDiffAB && // Condition for the difference between arr[i] and arr[j]
                        Math.abs(arr[j] - arr[k]) <= maxDiffBC && // Condition for the difference between arr[j] and arr[k]
                        Math.abs(arr[i] - arr[k]) <= maxDiffAC) { // Condition for the difference between arr[i] and arr[k]
                        // If all conditions are met, increment the count of "good" triplets
                        ++goodTripletsCount;
                    }
                }
            }
        }
        // Return the total number of "good" triplets found
        return goodTripletsCount;
    }
}
```

C++

```cpp
class Solution {
public:
    int countGoodTriplets(vector<int>& numbers, int limitA, int limitB, int limitC) {
        int size = numbers.size(); // Size of the input array
        int goodTripletsCount = 0; // Initialize count of good triplets

        // Iterate over all possible triplets
        for (int i = 0; i < size; ++i) {
            for (int j = i + 1; j < size; ++j) {
                // Check if the first condition is met to avoid unnecessary computations
                if (abs(numbers[i] - numbers[j]) <= limitA) {
                    for (int k = j + 1; k < size; ++k) {
                        // accumulate 1 to the count if all three conditions are satisfied
                        goodTripletsCount += (abs(numbers[i] - numbers[j]) <= limitA &&
                                              abs(numbers[j] - numbers[k]) <= limitB &&
                                              abs(numbers[i] - numbers[k]) <= limitC) ? 1 : 0;
                    }
                }
            }
        }
        // Return the total count of good triplets
        return goodTripletsCount;
    }
};
```

TypeScript

```typescript
// Define the function to count good triplets
function countGoodTriplets(numbers: number[], limitA: number, limitB: number, limitC: number): number {
    let size = numbers.length; // Size of the input array
    let goodTripletsCount = 0; // Initialize count of good triplets

    // Iterate over all possible triplets
    for (let i = 0; i < size; ++i) {
        for (let j = i + 1; j < size; ++j) {
            // Check if the first condition is met to avoid unnecessary computations
            if (Math.abs(numbers[i] - numbers[j]) <= limitA) {
                for (let k = j + 1; k < size; ++k) {
                    // Check if the current triplet (arr[first_index], arr[second_index], arr[third_index])
                    // is a good triplet based on the provided conditions.
                    is_good_triplet = (
                        abs(arr[first_index] - arr[second_index]) <= a and
                        abs(arr[second_index] - arr[third_index]) <= b and
                        abs(arr[first_index] - arr[third_index]) <= c
                    )
                    // If it is a good triplet, increment the good_triplets_count.
                    good_triplets_count += is_good_triplet

        // After checking all possible triplets, return the total count of good triplets.
        return good_triplets_count
```

Time and Space Complexity

The given Python code implements a function to count the number of good triplets in an array. A triplet `(arr[i], arr[j], arr[k])` is considered good if it satisfies all of the following conditions:

- `0 <= i < j < k < arr.length`
- `|arr[i] - arr[j]| <= a`
- `|arr[j] - arr[k]| <= b`
- `|arr[i] - arr[k]| <= c`

Where `|x - y|` denotes the absolute value of the difference between `x` and `y`.

Time Complexity

The time complexity of the function is determined by the three nested for-loops. The outermost loop runs `n` times, where `n` is the length of the array. The middle loop runs up to `n-1` times, and the innermost loop runs up to `n-2` times. Therefore, the total number of iterations is governed by the series of descending integers starting from `n` and decreasing to 1.

The time complexity can then be calculated as the sum of the series `Σ(i * (i-1) / 2)`, for `i` from `1` to `n-2`. This simplifies to `O(n^3)` since the loop counters are independent and the series is a sum of cubic numbers.

Space Complexity

The space complexity of the solution is `O(1)`. Other than the input array, only a fixed number of integer variables (`ans`, `n`, `i`, `j`, `k`) are used, and their number does not depend on the input size `n`.