2367. Number of Arithmetic Triplets

Easy Array Hash Table Two Pointers Enumeration

Problem Description

You are given an array of distinct integers which is in increasing order and a positive integer representing the difference value, `diff`. The goal is to find out how many unique triplets of indices `(i, j, k)` within the array satisfy two conditions: first, `nums[j] - nums[i]` is equal to `diff`, and second, `nums[k] - nums[j]` is equal to `diff` as well, with the indices following the relationship of `i < j < k`. This is essentially looking for sequences of three numbers that form an arithmetic progression with a common difference of `diff`.

Intuition

The intuition behind the solution involves understanding that for an arithmetic triplet to exist for a number `x` (`nums[i]`), there must be two other numbers `x + diff` (`nums[j]`) and `x + diff * 2` (`nums[k]`) present in the array. The solution utilizes a set to store the numbers for constant-time lookups, which makes checking the existence of `x + diff` and `x + diff * 2` efficient.

The approach is as follows:

1. Convert the list of numbers into a set for faster lookup. This is beneficial because we want to be able to quickly check if a number `+ diff` exists in the original array.
2. Iterate over each number in the array (which is already in increasing order) and for each number `x`, check if both `x + diff` and `x + diff * 2` are present in the set.
3. Sum up the results of the checks for each element. If for a given number `x`, both `x + diff` and `x + diff * 2` are found, this indicates the existence of an arithmetic triplet, so we count it.
4. The sum gives the total count of such unique triplets in the array.

This method is efficient because we capitalize on the properties of sets and the given ordered nature of the array to avoid unnecessary computations. By doing a single pass through the array and checking for the presence of the other two elements in the triplet, we achieve a solution that is simple and effective.

Solution Approach

The solution is implemented using a set data structure and a single for-loop iteration through the array. Here is the breakdown of how the solution works step by step:

1. Convert the `nums` array into a set called `vis` which stands for visited or seen. The conversion to a set is critical because it allows for `O(1)` time complexity lookups to check if an element exists within the array.

```
vis = set(nums)
```

2. Use list comprehension combined with the `sum` function to iterate over each number `x` in `nums`. The iteration results in a boolean expression for each number which checks if both the next two numbers in the arithmetic sequence are present in the set `vis`. For each `x` in `nums`, if `x + diff` and `x + diff * 2` exist in `vis`, the condition is `True` and contributes 1 to the sum, otherwise, it contributes 0.

```
sum(x + diff in vis and x + diff * 2 in vis for x in nums)
```

3. For each iteration, the algorithm checks for the two required conditions:
   - `x + diff` is in `vis`: This checks if there is another number ahead in the array that is `diff` greater than the current number `x`. This is looking for the `j` index such that `nums[j] - nums[i] == diff`.
   - `x + diff * 2` is in `vis`: This checks if there is a number that is twice the `diff` greater than the current number `x`. This is looking for the `k` index such that `nums[k] - nums[j] == diff`.
4. The `sum` function then adds up the results from the list comprehension. Each `True` represents a valid arithmetic triplet, and the sum is therefore the total count of unique arithmetic triplets in the array.

The pattern used here is effectively checking each possible starting point of an arithmetic triplet and rightly assuming that due to the strictly increasing nature of the inputs, if an `x + diff` and `x + diff * 2` exist, they will be part of a valid arithmetic triplet. The use of set for constant-time lookups and list comprehension for concise code makes the implementation both efficient and readable.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose our array of distinct integers is `nums = [1, 3, 5, 7, 9]`, and the given difference value is `diff = 2`.

Following the steps outlined in the solution approach:

1. We first convert the `nums` array into a set `vis` to make element lookups more efficient:

```
nums = [1, 3, 5, 7, 9]
vis = set(nums)  # vis is now {1, 3, 5, 7, 9}
```

2. Next, we use list comprehension and the `sum` function to iterate over each number in `nums`. For each number `x`, we check if `x + diff` and `x + diff * 2` are present in the set `vis`.

   Here's what happens for each element of `nums`:

   - For `x = 1`: Check if `1 + 2` and `1 + 4` are in `vis`. This is `True` because both `3` and `5` are in `vis`.
   - For `x = 3`: Check if `3 + 2` and `3 + 4` are in `vis`. This is `True` because both `5` and `7` are in `vis`.
   - For `x = 5`: Check if `5 + 2` and `5 + 4` are in `vis`. This is `True` because both `7` and `9` are in `vis`.
   - For `x = 7`: Check if `7 + 2` and `7 + 4` are in `vis`. This is `False` since `9` is in `vis` but `11` is not.
   - For `x = 9`: Check if `9 + 2` and `9 + 4` are in `vis`. This is `False` since neither `11` nor `13` is in `vis`.

   The above checks can be summed up with the following line of code:

```
sum(x + diff in vis and x + diff * 2 in vis for x in nums)
```

3. As we iterate over the array, we find that `x = 1`, `x = 3`, and `x = 5` satisfy both conditions of being a valid starting point for an arithmetic triplet with a common difference of `diff`. Therefore, for these elements, the corresponding boolean will be `True`.
4. The `sum` function will sum these `True` values. In our example, we have three `True` values corresponding to starting points `1`, `3`, and `5`, resulting in a sum of `3`.

Therefore, there are three unique triplets that form an arithmetic progression with a common difference of `2` in the array `nums = [1, 3, 5, 7, 9]`.

This walkthrough demonstrates the utility of the set data structure for lookups and the effectiveness of iterating through the ordered list to identify valid arithmetic triplets.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def arithmeticTriplets(self, nums: List[int], diff: int) -> int:
        # Create a set from the list for constant time look-up.
        seen_numbers = set(nums)

        # Count the number of valid arithmetic triplets
        # An arithmetic triplet is a sequence of three numbers where
        # the difference between consecutive numbers is equal to diff.
        # Here it checks if for a given number x in nums, both x + diff and
        # x + 2 * diff are present in nums. If so, that contributes to one valid triplet.
        triplet_count = sum(x + diff in seen_numbers and x + 2 * diff in seen_numbers for x in nums)

        # Return the total count of triplets found.
        return triplet_count
```

Java

```java
class Solution {
    // Function to find the number of arithmetic triplets in an array
    public int arithmeticTriplets(int[] nums, int diff) {
        // An array to keep track of the presence of elements up to the maximum possible value
        boolean[] seen = new boolean[301];

        // Mark the presence of each number in the 'seen' array
        for (int num : nums) {
            seen[num] = true;
        }

        // Initialize the count for arithmetic triplets
        int count = 0;

        // Iterate through the array to find arithmetic triplets
        for (int num : nums) {
            // Check if the two subsequent numbers (with the given difference 'diff') are present
            if (seen[num + diff] && seen[num + 2 * diff]) {
                // If both are present, we found an arithmetic triplet, increment the count
                ++count;
            }
        }

        // Return the total count of arithmetic triplets found
        return count;
    }
}
```

C++

```cpp
#include <vector>
#include <bitset>

class Solution {
public:
    // This function counts the number of arithmetic triplets in the array.
    // An arithmetic triplet is a sequence of three numbers such that
    // the difference between consecutive numbers is the same as 'diff'.
    int arithmeticTriplets(vector<int>& nums, int diff) {
        bitset<301> visitedNumbers; // Initialize a bitset where we will mark the numbers present in 'nums'.

        // Mark all the numbers present in the 'nums' vector within the bitset 'visitedNumbers'.
        for (int number : nums) {
            visitedNumbers[number] = 1;
        }

        int countTriplets = 0; // Initialize counter for arithmetic triplets.

        // Iterate through all the numbers in 'nums'.
        for (int number : nums) {
            // Check if there are two other numbers in the sequence that make an arithmetic triplet
            // with the current 'number' and the given 'diff' (difference).
            // Increases countTriplets when the two other numbers forming the triplet are present, i.e., when
            // both 'number + diff' and 'number + 2 * diff' are set in the 'visitedNumbers' bitset.
            countTriplets += visitedNumbers[number + diff] && visitedNumbers[number + 2 * diff];
        }

        return countTriplets; // Return the total number of arithmetic triplets found in 'nums'.
    }
};
```

TypeScript

```typescript
function arithmeticTriplets(nums: number[], diff: number): number {
    // Initialize an array that will keep track of the presence of numbers within 'nums'
    const visited: boolean[] = new Array(301).fill(false);

    // Populate the 'visited' array with true at indexes that exist in the 'nums' array
    for (const num of nums) {
        visited[num] = true;
    }

    // Initialize a counter for the number of arithmetic triplets found
    let tripletCount = 0;

    // Iterate through the 'nums' array to find arithmetic triplets
    for (const num of nums) {
        // Check if the two successors (num+diff and num+2*diff) are present in 'nums'
        if (visited[num + diff] && visited[num + 2 * diff]) {
            // Increment the counter if both successors are found, signifying an arithmetic triplet
            tripletCount++;
        }
    }

    // Return the total count of arithmetic triplets
    return tripletCount;
}
```

```python
from typing import List

class Solution:
    def arithmeticTriplets(self, nums: List[int], diff: int) -> int:
        # Create a set from the list for constant time look-up.
        seen_numbers = set(nums)

        # Count the number of valid arithmetic triplets
        # An arithmetic triplet is a sequence of three numbers where
        # the difference between consecutive numbers is equal to diff.
        # Here it checks if for a given number x in nums, both x + diff and
        # x + 2 * diff are present in nums. If so, that contributes to one valid triplet.
        triplet_count = sum(x + diff in seen_numbers and x + 2 * diff in seen_numbers for x in nums)

        # Return the total count of triplets found.
        return triplet_count
```

Time and Space Complexity

Time Complexity

The given code traverses through all the elements in the `nums` list once to construct the `vis` set, and again it iterates through all elements of `nums` in the return statement.

For each element `x` in `nums`, the code checks if `x + diff` and `x + diff * 2` are present in the `vis` set. Looking up an element in a set has an average time complexity of `O(1)` because sets in Python are implemented as hash tables.

Therefore, the overall time complexity is `O(n)` where `n` is the number of elements in `nums`, since the set lookup operation is constant time on average, and we are doing this operation twice for each element in `nums`.

Space Complexity

The space complexity of the code is determined by the additional data structures that are used. Here, a set `vis` is created based on the elements of `nums`. In the worst case, if all elements in `nums` are unique, the `vis` set will contain the same number of elements as `nums`.

So, the space complexity would be `O(n)`, where `n` is the number of elements in `nums`.