413. Arithmetic Slices

Medium Array Dynamic Programming Leetcode Link

Problem Description

The goal of the given problem is to find the number of contiguous subarrays within a given array of integers, where each subarray forms an arithmetic sequence. An **arithmetic sequence** is defined as a sequence of at least three numbers where the difference between consecutive elements is the same throughout.

For example, if the difference between the first and second elements is 2, then every subsequent pair of consecutive elements must also have a difference of 2 for the array to be considered arithmetic.

Our task is to count how many such arithmetic subarrays exist in the `nums` array.

Intuition

The intuition behind the solution comes from the realization that we can count each possible arithmetic subarray as we iterate through the array `nums` and keep a running count of the number of times we encounter consecutive pairs with the same difference.

If we find consecutive pairs with the same difference (`d`), this means we've found an additional element that can extend the arithmetic subarray formed so far. Each time we find a new pair with the same difference, it means we've formed an additional arithmetic subarray including the new element.

In detail, if we have a sequence of `n` consecutive elements that have the same difference `d` between them, the number of arithmetic subarrays that can be formed from these elements is equal to the sum of the first `n-2` positive integers. This is because an arithmetic subarray needs at least three elements.

So, to reach the solution, we keep variables for the current pair difference (`d`) and a counter (`cnt`) for consecutive pairs with the same difference found so far. We initialize `d` to a value outside any possible difference we could get from elements in `nums` to make sure we don't wrongly increment `cnt` at the start.

As we iterate through each pair of consecutive elements using the `pairwise` function, if the difference is the same as `d`, we increment the `cnt` because we've found an additional element that extends the arithmetic subarrays we've counted so far. We add the current `cnt` value to the answer (`ans`) as this is the number of new arithmetic subarrays formed by including this new element.

If the difference is not the same, it means we've reached the end of the current sequence of elements forming arithmetic subarrays, so we set `cnt` back to zero and set `d` to the current pair's difference.

By the end of the iteration, `ans` will contain the total number of arithmetic subarrays within `nums`.

Solution Approach

The given solution in Python takes advantage of the `pairwise` utility which creates an iterator that returns consecutive pairs of elements from the input `nums`. In other words, if `nums` is `[a, b, c, d, e]`, `pairwise(nums)` would yield `(a, b)`, `(b, c)`, `(c, d)`, and `(d, e)` consecutively.

Here's the step-by-step implementation of the solution:

1. **Initialize Variables:** Before looping through the `nums`, we initialize the `ans` variable to 0. This variable will accumulate the total count of arithmetic subarrays. We also initialize a `cnt` variable to 0; it keeps track of consecutive pairs with the same difference. The `d` variable, which holds the current common difference between pairs, is set to a number outside the valid range (`3000`) to handle the edge case at the beginning of the array.

2. **Loop through `pairwise(nums)`:** We iterate over each pair `(a, b)` in `pairwise(nums)`. Here, `a` and `b` represent consecutive elements in `nums`.

3. **Check the Difference and Update Counter:** We compare the difference `b - a` with the current `d`. If they're equal, increment `cnt` by 1, because this extends the current arithmetic subarray sequence by one element. If they're different, update `d` to the new difference `b - a` and reset `cnt` to 0, because we're starting to count a new set of arithmetic subarrays.

4. **Update the Answer:** Add the current `cnt` to `ans` in each iteration. By adding `cnt`, we're accounting for all the new arithmetic subarrays that end at the current element `b`. The reason adding `cnt` works is that for each extension of an arithmetic subarray by one element, we introduce exactly `cnt` new subarrays where `cnt` is the count of previous consecutive elements that were part of such subarrays.

5. **Return Result:** After the loop completes, `ans` represents the total number of arithmetic subarrays in `nums`, which is then returned as the final answer.

This algorithm is efficient, as it needs only a single pass through the array `nums`, making it an `O(n)` time complexity solution, where `n` is the length of `nums`. The space complexity is `O(1)` since it uses a constant amount of extra space.

Example Walkthrough

Let's demonstrate the solution approach with a small example. Consider the array `nums = [1, 3, 5, 7, 9, 15]`.

1. **Initialize Variables:** Set `ans = 0`, `cnt = 0`, and `d = 3000`.

2. **Loop through `pairwise(nums)`:** We'll be looking at the following pairs as we loop: `(1, 3)`, `(3, 5)`, `(5, 7)`, `(7, 9)`, and `(9, 15)`.

3. **Check the Difference and Update Counter:**

   - For the first pair `(1, 3)`, the difference `b - a` is `2`. Since `d` is initialized to `3000`, `d` does not equal `b - a`. So, update `d` to `2` and keep `cnt` at `0`.
   - Moving to the next pair `(3, 5)`, the difference is `2` which matches the current `d`. Increment `cnt` by `1`. Now, `cnt = 1` and `ans = 1`.
   - For `(5, 7)`, the difference is still `2`. Increment `cnt` again, `cnt = 2` and `ans = 1 + 2 = 3`.
   - The pair `(7, 9)` also has the same difference, so `cnt = 3` and `ans = 3 + 3 = 6`.
   - The final pair `(9, 15)` has a different difference of `6`. This ends the current sequence of arithmetic subarrays, so reset `cnt` to `0` and update `d` to `6`.
4. **Update the Answer:** We've updated `ans` throughout the steps as we went along.

5. **Return Result:** The value of `ans` after evaluating all pairs is `6`. This means there are 6 contiguous subarrays within `nums` that form an arithmetic sequence.

To conclude, given `nums = [1, 3, 5, 7, 9, 15]`, the arithmetic subarrays are `[1, 3, 5]`, `[3, 5, 7]`, `[5, 7, 9]`, `[1, 3, 5, 7]`, `[3, 5, 7, 9]`, and `[1, 3, 5, 7, 9]`, resulting in a count of 6.

Python Solution

```python
from itertools import pairwise

class Solution:
    def numberOfArithmeticSlices(self, nums: List[int]) -> int:
        total_slices = 0  # Initialize the count of arithmetic slices
        current_sequence_length = 0  # Tracks the length of the current arithmetic sequence

        # Set initial difference to a large number outside the possible range
        previous_difference = 3000

        # Iterate pairwise over the array to check differences between consecutive elements
        for current_num, next_num in pairwise(nums):
            current_difference = next_num - current_num

            # If the current difference equals the previous one, we extend the arithmetic sequence
            if current_difference == previous_difference:
                current_sequence_length += 1
            else:
                # If not, we start a new arithmetic sequence
                previous_difference = current_difference
                current_sequence_length = 0

            # The number of arithmetic slices ending at the current position can be added to the total
            total_slices += current_sequence_length

        return total_slices  # Return the total count of arithmetic slices

```

Java Solution

```java
class Solution {
    public int numberOfArithmeticSlices(int[] nums) {
        int arithmeticSliceCount = 0;  // To store the number of arithmetic slices found.
        int currentSliceLength = 0;    // To keep track of the current sequence length.
        int difference = 3000;         // Initialize with a value outside the problem constraints.

        // Iterate through the given array to find arithmetic slices.
        for (int i = 0; i < nums.length - 1; ++i) {
            // Check if the current pair of elements continue the arithmetic sequence.
            if (nums[i + 1] - nums[i] == difference) {
                // If they do, increment the count of arithmetic slices by increasing the current length.
                ++currentSliceLength;
            } else {
                // If not, update the common difference to the new pair's difference.
                difference = nums[i + 1] - nums[i];
                // Reset the current sequence length because a new arithmetic sequence starts.
                currentSliceLength = 0;
            }
            // Add the number of arithmetic slices ending at the current position to the total count.
            arithmeticSliceCount += currentSliceLength;
        }
        // Return the total count of arithmetic slices in the array.
        return arithmeticSliceCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int numberOfArithmeticSlices(vector<int>& nums) {
        int totalSlices = 0; // This will hold the total number of arithmetic slices
        int currentStreak = 0; // This keeps track of the current streak of arithmetic slices
        int previousDifference = 3000; // Initialize with a difference not likely to appear in the sequence

        for (int i = 0; i < nums.size() - 1; ++i) { // Loop through the vector, but not including the last element
            int currentDifference = nums[i + 1] - nums[i]; // Calculate the difference between two consecutive elements

            // Check if the current difference is the same as the previous difference
            if (currentDifference == previousDifference) {
                ++currentStreak; // If so, increment the current streak of consecutive slices
            } else {
                previousDifference = currentDifference; // Otherwise, update the difference
                currentStreak = 0; // And reset the current streak since a new difference has started
            }

            totalSlices += currentStreak; // Add the current streak to the total slices count
                                          // This works because an arithmetic slice of length n contributes
                                          // (n - 1) + (n - 2) + ... + 1 slices.
        }

        return totalSlices; // Return the total number of arithmetic slices found
    }
};

```

Typescript Solution

```typescript
function numberOfArithmeticSlices(nums: number[]): number {
    let totalSlices = 0; // Initialize count of arithmetic slices
    let currentStreak = 0; // Count of consecutive arithmetic pairs within a slice
    let previousDifference = 3000; // Large initial difference to ensure the first pair forms a new slice

    // Iterate through the array of numbers to detect arithmetic slices
    for (let i = 0; i < nums.length - 1; ++i) {
        const currentNum = nums[i]; // The current element in nums
        const nextNum = nums[i + 1]; // The next element in nums
        const currentDifference = nextNum - currentNum; // Difference between the current and next element

        // Check if the current pair continues the streak of the same differences
        if (currentDifference === previousDifference) {
            // If so, increment the streak count since it extends an arithmetic slice
            ++currentStreak;
        } else {
            // If not, reset the streak count and update the difference tracker
            previousDifference = currentDifference;
            currentStreak = 0;
        }

        // Accumulate the count of arithmetic slices
        // Each additional number in a streak adds to existing slices
        totalSlices += currentStreak;
    }

    // Return the total count of arithmetic slices found in the array
    return totalSlices;
}

```

Time and Space Complexity

The given Python function `numberOfArithmeticSlices` computes the number of continuous arithmetic slices (subarrays) in an array. The computation iterates over the array once, using a pairwise comparison of elements to determine if a current pair continues an arithmetic sequence or starts a new one.

Time Complexity:

The time complexity of the function is `O(n)`, where `n` is the length of the input list `nums`. This is because the function makes a single pass through the list using `pairwise(nums)` to compare consecutive elements.

The inner operations of the for loop (checking conditions, updating the count `cnt`, updating the difference `d`, and incrementing `ans`) are all constant time operations, which means they do not depend on the size of the input list. Thus, they do not change the overall linear time complexity.

Space Complexity:

The space complexity of the function is `O(1)` as only a fixed number of extra variables are used (`ans`, `cnt`, `d`). The space used does not scale with the input size, so it remains constant regardless of the length of `nums`.