Complete documentation

---

1. Problem Statement

The problem asks to find the number of arithmetic subarrays in a given integer array `nums`. An arithmetic subarray is defined as a contiguous subsequence of at least three elements where the difference between any two consecutive elements is the same.

Input:

- A list `nums` of integers, where `1 <= len(nums) <= 5000` and `-1000 <= nums[i] <= 1000`.

Output:

- Return the total number of arithmetic subarrays in `nums`.

---

2. Intuition

The idea is to efficiently track arithmetic subarrays by maintaining a dynamic count of valid subarrays as we iterate through the array. As we traverse through `nums`, we can detect if a new arithmetic subarray starts by comparing the difference between consecutive elements and extending existing arithmetic subarrays if the difference remains consistent.

---

3. Key Observations

- **Arithmetic Subarray**: A subarray of length 3 or more is arithmetic if the difference between each consecutive pair of elements is constant.
- **Dynamic Approach**: When an arithmetic subarray is found, extending it by one more element still forms an arithmetic subarray as long as the difference remains constant.
- **Efficiency**: The problem requires an efficient solution that works within the time limits for arrays up to length 5000.

---

4. Approach

Step-by-Step Approach:

1. **Initialize Variables**:
   - `total_slices`: Stores the total count of arithmetic subarrays.

    - `current_slices`: Tracks the number of arithmetic subarrays that end at the current index.
2. **Iterate through the Array**:
    - Start from index 2 (because an arithmetic subarray requires at least 3 elements).
    - Compare the difference between the current element and the previous one with the difference between the previous element and the one before it.
    - If the differences match, increment the `current_slices` counter.
    - If they don't match, reset `current_slices` to 0 because the arithmetic condition is broken.
3. **Update Total Slices**:
    - After processing each element, add `current_slices` to `total_slices`.
4. **Return Result**:
    - Finally, return `total_slices`, which gives the number of arithmetic subarrays.

---

5. Edge Cases

- **Small Array**: If the array has fewer than 3 elements, the answer should be 0 because no arithmetic subarray can exist.
- **Constant Array**: If the array consists of the same value repeated, every subarray of length 3 or more is arithmetic.
- **Non-Arithmetic Array**: If no arithmetic subarray exists, the result should be 0.
- **Array with Negative Numbers**: The solution handles negative integers and calculates differences correctly, so there is no need for special treatment of negative values.

---

6. Complexity Analysis

Time Complexity:

- **O(n)**, where `n` is the length of the input array `nums`. We are iterating through the array once, and at each step, we perform constant-time operations.

Space Complexity:

- **O(1)**. We only use a few integer variables (`total_slices` and `current_slices`), so the space usage does not depend on the size of the input array.

---

7. Alternative Approaches

Brute Force Approach:

A brute force approach would involve generating all possible subarrays of size at least 3 and checking if each one is arithmetic. This would take **O(n^3)** time, which is inefficient for large arrays (up to length 5000).

Optimized Dynamic Programming:

We could use dynamic programming to store results for previously calculated subarrays, but the optimized approach described in this solution is more efficient and straightforward.

---

8. Code Implementation

```
from typing import List

class Solution:
    def numberOfArithmeticSlices(self, nums: List[int]) -> int:
        n = len(nums)
        if n < 3:
            return 0

        # Variable to store the total number of arithmetic slices
        total_slices = 0

        # Variable to store the length of the current arithmetic slice
        current_slices = 0

        for i in range(2, n):
            # Check if the difference between nums[i] and nums[i-1] is the same as
nums[i-1] and nums[i-2]
            if nums[i] - nums[i-1] == nums[i-1] - nums[i-2]:
                # If they match, increment current_slices
                current_slices += 1
                # Add current_slices to total_slices
                total_slices += current_slices
            else:
                # Reset the current_slices if they don't match
                current_slices = 0

        return total_slices
```

---

9. Test Cases

Test Case 1:

Input: `[1, 2, 3, 4]`
Output: `3`
Explanation: The arithmetic slices are `[1, 2, 3]`, `[2, 3, 4]`, and `[1, 2, 3, 4]`.

Test Case 2:

Input: `[1]`
Output: `0`
Explanation: No subarray has at least 3 elements.

Test Case 3:

Input: `[1, 3, 5, 7, 9]`
Output: `6`

Explanation: The arithmetic slices are: `[1, 3, 5]`, `[3, 5, 7]`, `[5, 7, 9]`, `[1, 3, 5, 7]`, `[3, 5, 7, 9]`, `[1, 3, 5, 7, 9]`.

Test Case 4:

Input: `[1, 2, 4, 6, 8]`
Output: `4`
Explanation: The arithmetic slices are: `[1, 2, 4]`, `[2, 4, 6]`, `[4, 6, 8]`, `[1, 2, 4, 6]`.

---

10. Final Thoughts

The solution efficiently calculates the number of arithmetic subarrays using a dynamic programming approach. By keeping track of the difference between consecutive elements, we avoid the need to check every possible subarray manually, leading to a time complexity of **O(n)**. This approach is optimal for the problem constraints, ensuring that it can handle the largest input sizes effectively.