446. Arithmetic Slices II - Subsequence

Hard Array Dynamic Programming Leetcode Link

Problem Description

The problem is about finding the total count of arithmetic subsequences in a given array `nums`. A subsequence is defined as a sequence that can be derived from the original array by deleting some or no elements without changing the order of the remaining elements. The question specifies arithmetic subsequences which are described by two key characteristics:

1. They must contain **at least three elements**.
2. The difference between consecutive elements must be the same throughout the subsequence (this is known as the common difference in arithmetic sequences).

For example, `[1, 3, 5]` is an arithmetic subsequence with a common difference of 2, and `[1, 1, 1]` is also an arithmetic subsequence with a common difference of 0. However, a sequence like `[1, 2, 4]` is not arithmetic since the differences between the consecutive elements are not equal (1 and 2, respectively).

The goal is to find the number of such subsequences in the given array `nums`.

Intuition

When solving this type of problem, it's important to recognize that a straightforward approach to generating all possible subsequences and checking whether each is arithmetic would be inefficient, likely leading to an exponential time complexity. Instead, we should strive for a dynamic programming approach that builds on the solution of smaller subproblems.

The intuition is to use dynamic programming to keep track of the count of arithmetic subsequence slices ending at each index with a given common difference. We create an array `f` of dictionaries, with each dictionary at index `i` holding counts of subsequences that end at `nums[i]` categorized by their common differences.

As we iterate through `nums`, we consider each pair of indices `(i, j)` such that `i > j`. The common difference `d` is calculated as `nums[i] - nums[j]`. Then, we update our dynamic programming array `f` and count as follows:

1. The number of arithmetic subsequences ending at `i` with a common difference `d` is incremented by the count of subsequences ending at `j` with the same difference (found in `f[j][d]`) plus one (for the new subsequence formed by just including `nums[i]` and `nums[j]`).
2. The overall number of arithmetic subsequences (stored in `ans`) is increased by `f[j][d]`, since we can extend any subsequence ending at `j` with `nums[i]` to form a new valid subsequence.

The use of a dictionary enables us to keep track of different common differences efficiently, and by iterating over all pairs `(i, j)` where `i > j`, we ensure that we consider all valid subsequences that could be formed.

Solution Approach

The provided Python solution uses dynamic programming with an implementation that optimizes the process of counting arithmetic subsequences.

These are the key components of the implementation:

1. **Dynamic Programming Array (`f`):** An array of dictionaries is used to store the state of the dynamic programming algorithm. This array `f` has the same length as the input array `nums`, where each index `i` of `f` has a corresponding dictionary. In this dictionary, keys represent the common differences of arithmetic subsequences that end at `nums[i]`, and values represent the number of such subsequences.

2. **Counting Arithmetic Subsequences:** The algorithm iterates through the input array `nums` using two nested loops to consider each pair `(i, j)` where `i > j` as potential end points for an arithmetic subsequence. The difference `d = nums[i] - nums[j]` is computed for each pair.

3. **Updating State and Counting Slices (`ans`):** For each pair `(i, j)`, the algorithm does two things:

   - It updates the state in `f[i][d]` by adding `f[j][d] + 1` to it. The extra `+1` accounts for a new subsequence formed by including just `nums[i]` and `nums[j]`.
   - It increases the overall count (`ans`) by `f[j][d]`. Each entry in `f[j]` represents a valid subsequence that ends at `nums[j]` and can be extended by `nums[i]` to form a new subsequence.

In summary, the solution involves iterating over pairs of elements to calculate possible common differences and uses dynamic programming to store the count of arithmetic subsequences up to the current index, avoiding the inefficiency of checking each possible subsequence individually. The final variable `ans` holds the count of all valid arithmetic subsequences of length three or more across the entire array `nums`.

Since this approach avoids redundant calculations by reusing previously computed results, it allows for an efficient calculation of the total number of arithmetic subsequences in the input array.

Example Walkthrough

Let's illustrate the solution approach using a small example array `nums = [2, 4, 6, 8, 10]`.

1. **Initializing Dynamic Programming Array (`f`):** We start by creating an array of dictionaries, `f`, where each index will store dictionaries. Initially, all dictionaries are empty since no subsequences have been computed yet.

2. **Counting Arithmetic Subsequences:**

   - For `i = 1 (nums[1] = 4)`, we look backward for `j < i`:

     - At `j = 0 (nums[0] = 2)`, the common difference `d = 4 - 2 = 2`.
     - The count at `f[1][d]` is initially 0. We increment it by `f[0][2] + 1` (0 + 1 since `f[0][2]` does not exist yet) to account for the subsequence `[2, 4]`.
     - However, this subsequence is not yet long enough to increment our answer, as we need at least three elements for an arithmetic subsequence.
   - For `i = 2 (nums[2] = 6)`, we again look backward for `j < i`:

     - For `j = 1 (nums[1] = 4)`, `d = 6 - 4 = 2`. We increment `f[2][2]` by `f[1][2] + 1` which is `1 + 1 = 2` (the `[2, 4, 6]` subsequence and just `[4, 6]`).
     - For `j = 0 (nums[0] = 2)`, `d = 6 - 2 = 4`. This common difference does not align with the subsequences formed up to `i = 2`, so no update is made to `ans`.
     - Our overall count `ans` is increased by `f[1][2]` which is `1`, as `[2, 4, 6]` is a valid arithmetic subsequence.
   - For `i = 3 (nums[3] = 8)`, we look backward again for `j < i`:

     - For `j = 2 (nums[2] = 6)`, `d = 8 - 6 = 2`. Increment `f[3][2]` by `f[2][2] + 1` which becomes `2 + 1 = 3` (we can form `[4, 6, 8]`, `[2, 6, 8]`, and just `[6, 8]`).
     - For `j = 1`, `d = 2`, we again increment `f[3][2]` and `ans` by `f[1][2]`.
     - Our count `ans` is updated with the sum of `f` values for `d = 2` for `j = 1` and `j = 2` which gives us 3 more valid subsequences (`[2, 4, 8]`, `[4, 6, 8]`, `[2, 6, 8]`).
   - Continuing this process for `i = 4 (nums[4] = 10)`, we find the subsequences ending with `10` and update `f` and `ans` accordingly.

3. **Final Count (`ans`):** By the end of our iteration, `ans` would include the count of all valid arithmetic subsequences of at least three elements. In our case, the subsequences are `[2, 4, 6]`, `[2, 4, 6, 8]`, `[2, 4, 8]`, `[4, 6, 8]`, `[2, 6, 8]`, `[2, 4, 6, 8, 10]`, `[2, 4, 6, 10]`, `[2, 4, 8, 10]`, `[2, 6, 8, 10]`, `[4, 6, 8, 10]`, so `ans` would be 7.

By reusing previously computed results for overlapping subproblems, the efficient dynamic programming solution allows us not to recalculate every potential subsequence, leading to a more optimized approach for counting the total number of arithmetic subsequences.

Python Solution

```python
from typing import List
from collections import defaultdict

class Solution:
    def numberOfArithmeticSlices(self, nums: List[int]) -> int:
        # Initialize a list of dictionaries for each number in `nums`
        # Each dictionary will hold the count of arithmetic sequences ending with that number
        arithmetic_count = [defaultdict(int) for _ in nums]

        # Initialize the answer to 0
        total_count = 0

        # Iterate over each pair (i, x) where `i` is the index and `x` is the number
        for i, current_num in enumerate(nums):
            # Iterate over all the numbers before the current number
            for j, prev_num in enumerate(nums[:i]):
                # Calculate the difference between the current and previous number
                difference = current_num - prev_num

                # Add the count of arithmetic sequences ending at index `j` with the same difference to the answer
                total_count += arithmetic_count[j][difference]

                # Add or increment the count of arithmetic sequences ending at index `i` with the same difference
                # It will be 1 more than the count at index `j` since `i` extends the sequence from `j` by one more element
                arithmetic_count[i][difference] += arithmetic_count[j][difference] + 1

        # Return the total count of all arithmetic sequences found in `nums`
        return total_count

```

Java Solution

```java
class Solution {
    public int numberOfArithmeticSlices(int[] nums) {
        // Total number of elements in the input array.
        int n = nums.length;
        // Array of maps to store the count of arithmetic slices ending at each index.
        Map<Long, Integer>[] countMaps = new Map[n];
        // Initialize each map in the array.
        Arrays.setAll(countMaps, element -> new HashMap<>());
        // Variable to store the final answer.
        int totalCount = 0;

        // Iterate through each element in the array starting from the second element.
        for (int i = 0; i < n; ++i) {
            // For each element, check all previous elements to calculate the differences.
            for (int j = 0; j < i; ++j) {
                // Calculate the difference between the current and previous elements.
                Long diff = 1L * nums[i] - nums[j];
                // Get the current count of arithmetic slices with the same difference ending at index j.
                int count = countMaps[j].getOrDefault(diff, 0);
                // Accumulate the total number of found arithmetic slices.
                totalCount += count;
                // Update the countMap for the current element (index i).
                // Increment the count of the current difference by the count from the previous index plus 1 for the new slice.
                countMaps[i].merge(diff, count + 1, Integer::sum);
            }
        }

        // Return the accumulated count of all arithmetic slices found in the array.
        return totalCount;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>

class Solution {
public:
    int numberOfArithmeticSlices(std::vector<int>& numbers) {
        int count = numbers.size(); // Count of numbers in the vector.
        std::vector<std::unordered_map<long long, int>> arithmeticCount(count);
        // The vector 'arithmeticCount' will store maps to keep the count of
        // how many times a particular arithmetic difference has appeared up to
        // the current index.

        int totalSlices = 0; // This will hold the total number of arithmetic slices.

        for (int i = 0; i < count; ++i) {
            for (int j = 0; j < i; ++j) {
                // Compute the difference 'diff' of the current pair of numbers.
                long long diff = static_cast<long long>(numbers[i]) - numbers[j];
                // The number of sequences ending with 'j' that have a common difference 'diff'.
                int sequencesEndingWithJ = arithmeticCount[j][diff];
                // Increment the total number of arithmetic slices found so far by this number.
                totalSlices += sequencesEndingWithJ;
                // Increment the count of the number of sequences with difference 'diff'
                // ending at 'i' by the number of such sequences ending at 'j' + 1
                arithmeticCount[i][diff] += sequencesEndingWithJ + 1;
                // The '+1' is for the new sequence formed by 'j' and 'i' themselves.
            }
        }
        return totalSlices;
    }
};

```

Typescript Solution

```typescript
function numberOfArithmeticSlices(nums: number[]): number {
    // Length of the input array
    const length = nums.length;

    // Array of maps to store the count of arithmetic slice endings at each index with a certain difference
    const arithmeticMap: Map<number, number>[] = new Array(length).fill(0).map(() => new Map());

    // Final count of arithmetic slices
    let totalCount = 0;

    // Iterate over all the pairs of elements in the array
    for (let i = 0; i < length; ++i) {
        for (let j = 0; j < i; ++j) {
            // Compute the difference between the current pair of numbers
            const difference = nums[i] - nums[j];

            // Retrieve the count of slices ending at index j with the computed difference
            const count = arithmeticMap[j].get(difference) || 0;

            // Increment the total count by the number of slices found
            totalCount += count;

            // Update the map for the current index i, adding the count calculated above to the existing count
            // and account for the new slice formed by i and j
            arithmeticMap[i].set(difference, (arithmeticMap[i].get(difference) || 0) + count + 1);
        }
    }

    // Return the final count of arithmetic slices
    return totalCount;
}

```

Time and Space Complexity

The given Python code defines a method `numberOfArithmeticSlices`, which calculates the number of arithmetic slices in a list of numbers. It does this by using dynamic programming with a list of default dictionaries to track the arithmetic progressions.

Time Complexity:

The time complexity of the method can be determined by analyzing the nested loops and the operations performed within those loops. The outer loop runs for `n` iterations, where `n` is the length of the list `nums`. The inner loop runs `i` times for each iteration of the `i`th element in the outer loop:

- The outer loop: `O(n)`, iterating through each element of `nums`.
- The inner loop: Up to `O(n)`, iterating through elements up to `i`, which on average would be `O(n/2)` for each iteration of the outer loop.
- Within the inner loop:
  - The operation `d = x - y` is `O(1)`.
  - The lookup and update operations on the default dictionaries `f[j][d]` and `f[i][d]` are average-case `O(1)`, assuming hash table operations.

Therefore, the time complexity is the sum of the work done across all iterations of both loops, which is:

- `O(n * (1 + 2 + 3 + ... + (n - 1)))`
- This equals `O(n * (n(n - 1) / 2))`
- Hence, the time complexity simplifies to `O(n^3)`.

Space Complexity:

The space complexity can be analyzed by looking at the data structures used:

- The list `f` contains `n` defaultdicts, one for each element in `nums`.
- Each defaultdict can have up to `i` different keys, where `i` is the index of the outer loop, leading to `O(n^2)` in the worst case because each arithmetic progression can have a different common difference.

Thus, the space complexity is `O(n^2)` based on the storage used by the `f` list.