2036. Maximum Alternating Subarray Sum

Medium Array Dynamic Programming

Leetcode Link

Problem Description

In this problem, we are given an array of integers, where the array is 0-indexed, meaning indexing starts from 0. A subarray is defined as a contiguous non-empty sequence of elements within the array. The alternating subarray sum is a special kind of sum where we alternately add and subtract the elements of the subarray starting with addition. For example, if the subarray starts at index `i` and ends at index `j`, then the alternating sum would be calculated as `nums[i] - nums[i+1] + nums[i+2] - ... +/- nums[j]`.

The objective is to find the maximum alternating subarray sum that can be obtained from any subarray within the given integer array `nums`.

Intuition

To solve this problem, we need to take into consideration that the sum we are looking to maximize can switch from positive to negative numbers and vice versa because of the alternating addition and subtraction. A brute force approach would involve checking all possible subarrays, but this would not be efficient, especially for large arrays.

Consequently, we turn to dynamic programming (DP) to optimize our process. The solution builds on the concept that the maximum alternating sum ending at any element in the array (which is either added or subtracted) can be based on the maximum sum we've computed up to the previous element.

We keep two accumulators, `f` and `g`, at each step of our iteration through the array. `f` represents the maximum alternating subarray sum where the last element is added, and `g` is where the last element is subtracted. At each iteration, `f` can either be the previous `g` (plus the current element, if `g` was at least 0, indicating a previous alternating subarray contribution), or it can directly be the current element if including the previous sequence doesn't yield a larger sum. Concurrently, `g` becomes the previous `f` minus the current element.

By maintaining and updating these two values at each step, we can ensure that the maximum alternating subarray sum can be computed in a single pass, thus significantly reducing the time complexity from what a brute force method would entail. The answer is the highest value achieved by either `f` or `g` during the iteration.

Solution Approach

The solution employs a dynamic programming (DP) approach to solve the problem efficiently. The key insight of the DP approach is to define two states that represent the maximum alternating subarray sum at each position `i` in the array `nums`.

- `f` is defined to be the maximum alternating subarray sum ending with `nums[i]` being added.
- `g` is defined to be the maximum alternating subarray sum ending with `nums[i]` being subtracted.

The solution initiates `f` and `g` with the value `-inf` to represent that initially, there's no subarray, hence the alternating sum is the smallest possible (negative infinity). As the array is traversed, `f` and `g` will be updated for each element.

Here's how `f` and `g` get updated for every element in `nums`:

1. `f` gets updated to the maximum of `g + nums[i]` or 0 plus `nums[i]`. In essence, this step decides whether to extend the alternating subarray by adding the current number to the previously best alternating sum ending with a subtracted number, or to start fresh with the current number if no profitable alternating subarray exists before it (in which case `g` would be non-positive).
2. `g` gets updated to `f - nums[i]` from before the update in step 1. This step effectively chooses to continue the alternating subarray by transitioning from addition to subtraction at this element.

The overall answer, `ans`, will be the maximum of all such `f` and `g` values computed. This ensures that at the end of the iteration through all elements in `nums`, `ans` will store the maximum alternating subarray sum of any potential subarray within `nums`.

The mathematical formulas involved in updating `f` and `g` are as follows:

`f = max(g + nums[i], nums[i])` which equates to `f = max(g, 0) + nums[i]` since adding a negative `g` would be less beneficial than just starting fresh from `nums[i]`.

`g = f_previous - nums[i]`

Lastly, for each iteration, `ans` is updated to the larger of itself, `f` or `g`. At the end of the loop, `ans` will hold the final maximum alternating subarray sum, which the function returns.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with the array `nums = [3, -1, 4, -3, 5]`.

Initially, we set `f` and `g` to negative infinity, representing no sum because we haven't started the process, and `ans` to negative infinity to track the maximum sum we discover.

Starting with the first element (3), we update `f` since no previous sum exists. So, `f` becomes `max(-inf, 0) + 3 = 3`. We update `ans` with the maximum of `ans`, `f` and `g`, so `ans = max(-inf, 3, -inf) = 3`. No need to update `g` yet since we only have one element.

For the second element (-1), we now have a subarray to consider:

- `g` gets updated to the previous value of `f` minus the current element, so `g = 3 - (-1) = 4`.
- `f` gets updated to the max of current `g` plus the element or just the element, so `f = max(0 + -1, 4 + -1) = 3`.
- `ans` now becomes `max(3, 4, 3) = 4`.

Moving on to the third element (4):

- `g` gets updated to `f` from the previous step minus the current element, so `g = 3 - 4 = -1`.
- `f` is updated to be `max(0 + 4, -1 + 4) = 4` because the previous `g` was negative.
- `ans` is updated to `max(4, -1, 4) = 4`.

For the fourth element (-3):

- `g` gets updated to `f` from the previous step minus the current element, so `g = 4 - (-3) = 7`.
- `f` is not any larger when updating with the current `g`, so `f` remains `max(0 + -3, 7 + -3) = 4`.
- `ans` is updated to `max(4, 7, 4) = 7`.

Finally, for the last element (5):

- `g` gets updated, `g = 4 - 5 = -1`.
- `f` gets updated since `g` is negative, `f = max(0 + 5, -1 + 5) = 5`.
- `ans` is updated to `max(7, -1, 5) = 7`.

After completing the iteration, `ans` holds the value of `7`, which is the maximum alternating subarray sum of any subarray within `nums`. The subarray contributing to this max sum in our example is `[-1, 4, -3]`, resulting in an alternating sum of `-1 + 4 - (-3) = 7`.

Python Solution

```python
from typing import List

class Solution:
    def maximumAlternatingSubarraySum(self, nums: List[int]) -> int:
        # Initialize variables to store the maximum sum found,
        # and temporary sums for even and odd indexed elements.
        max_sum = float('-inf')  # Use float('-inf') for negative infinity
        sum_even_idx = float('-inf')  # Sum at an even index of the alternated subarray
        sum_odd_idx = float('-inf')  # Sum at an odd index of the alternated subarray

        # Loop through each number in the input array
        for num in nums:
            # Update the sum at an even index. It is the greater of
            # the previous sum at an odd index (if it was a valid subarray sum) plus the current number,
            # or starting a new subarray from the current number.
            sum_even_idx, sum_odd_idx = max(sum_odd_idx, 0) + num, sum_even_idx - num

            # Update the maximum sum found so far by comparing with the new sums
            # obtained from the even and odd indexed subarrays.
            max_sum = max(max_sum, sum_even_idx, sum_odd_idx)

        # Return the maximum alternating subarray sum that was found.
        return max_sum
```

Java Solution

```java
class Solution {
    public long maximumAlternatingSubarraySum(int[] nums) {
        // Initialize a large value as "infinity".
        final long INF = 1L << 60;

        // Initialize `maxSum` to negative infinity which will store
        // the final result, the maximum alternating subarray sum.
        long maxSum = -INF;

        // Initialize variables to keep track of alternate subarray
        // sums during the iteration.
        long evenPositionSum = -INF;
        long oddPositionSum = -INF;

        // Iterate through the `nums` array.
        for (int num : nums) {
            // Calculate temporary sum for the subarray ending at an even position.
            long tempEvenSum = Math.max(oddPositionSum, 0) + num;

            // Update the sum for the subarray ending at an odd position.
            oddPositionSum = evenPositionSum - num;

            // Update `evenPositionSum` with the new calculated value.
            evenPositionSum = tempEvenSum;

            // Update `maxSum` to be the maximum of itself, `evenPositionSum`, and `oddPositionSum`.
            maxSum = Math.max(maxSum, Math.max(evenPositionSum, oddPositionSum));
        }

        // Return the computed maximum alternating subarray sum.
        return maxSum;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For using the max function

class Solution {
public:
    // Function to find the maximum alternating subarray sum.
    long long maximumAlternatingSubarraySum(vector<int>& nums) {
        using ll = long long; // Define long long as ll for easy use
        const ll negativeInfinity = 1LL << 60; // Define a very large negative number to compare with

        ll maxSum = -negativeInfinity; // Initialize maxSum with a very large negative value
        ll sumEven = -negativeInfinity; // Sum ending at even index (alternating subarray sum starting with positive term)
        ll sumOdd = -negativeInfinity; // Sum ending at odd index (alternating subarray sum starting with negative term)

        // Loop through each number in the array to build alternating sums
        for (int num : nums) {
            ll newSumEven = max(sumOdd, 0LL) + num; // Calculate new even index sum
            sumOdd = sumEven - num; // Calculate new odd index sum
            sumEven = newSumEven; // Update even index sum

            // Update the maxSum with the maximum value among maxSum, sumEven, and sumOdd
            maxSum = max({maxSum, sumEven, sumOdd});
        }

        return maxSum; // Return the maximum alternating subarray sum
    }
};
```

Typescript Solution

```typescript
function maximumAlternatingSubarraySum(nums: number[]): number {
    // Initialize maxSum to the smallest number to ensure it gets updated
    // f represents the maximum sum of an alternating subarray ending with a positive term
    // g represents the maximum sum of an alternating subarray ending with a negative term
    let maxSum = Number.MIN_SAFE_INTEGER, positiveTermSum = Number.MIN_SAFE_INTEGER, negativeTermSum = Number.MIN_SAFE_INTEGER;

    for (const num of nums) {
        // Update positiveTermSum: either start a new subarray from current num (if g is negative) or add num to the previous subarra
        positiveTermSum = Math.max(negativeTermSum, 0) + num;
        // Update negativeTermSum: subtract current num from the previous subarray to alternate the sign
        negativeTermSum = positiveTermSum - num;
        // Update maxSum to the maximum value of maxSum, positiveTermSum, and negativeTermSum at each iteration
        maxSum = Math.max(maxSum, positiveTermSum, negativeTermSum);
    }

    return maxSum;
}
```

Time and Space Complexity

The given Python code defines a method `maximumAlternatingSubarraySum` which calculates the maximum sum of any non-empty subarray from `nums`, where the sign of the elements in the subarray alternates.

Time Complexity

To understand the time complexity, let's go through the code:

1. The function iterates through the array `nums` once using a for loop (line 4).
2. Within each iteration of the for loop, it performs a constant number of operations: updating `f`, `g`, and `ans` (line 5).
3. No additional loops or recursive calls are present.

Since the number of operations within the loop does not depend on the size of the array, and the loop runs `n` times where `n` is the number of elements in `nums`, the time complexity is linear with respect to the length of the array, which we represent as `O(n)`.

Space Complexity

Now, let's examine the space complexity:

1. The solution uses a fixed number of extra variables `ans`, `f`, and `g` (line 3), independent of the input size.
2. No additional data structures dependent on the input size are utilized.
3. No recursive calls are made that would add to the call stack and therefore increase space complexity.

Taking the above points into consideration, the space complexity of the algorithm is constant, denoted as `O(1)` since the additional space used does not scale with the size of the input.