1911. Maximum Alternating Subsequence Sum

Medium Array Dynamic Programming

Problem Description

The problem defines an *alternating sum* of an array as the sum of elements at even indices minus the sum of elements at odd indices. For instance, in an array `[4,2,5,3]`, the alternating sum is (4 + 5) - (2 + 3) = 4.

The task is to find the **maximum alternating sum** of any subsequence in a given array `nums`. A *subsequence* can be obtained by deleting some elements (possibly none) from the original array but keeping the order of the remaining elements.

Intuition

To get the maximum alternating sum, we have to choose a subsequence where the difference between the sum of elements at even indices and the sum of elements at odd indices is maximized. This implies we want to include larger numbers at even indices and smaller numbers at odd indices, if possible.

To find the solution, we use dynamic programming to keep track of two values while iterating through the array: `f` and `g`. Here `f` represents the maximum alternating sum ending with an element at an even index, and `g` represents the maximum alternating sum ending with an element at an odd index.

1. When we are at a new number `x`, we have two choices: either include `x` in our subsequence or not. If the last included number was at an even index (and we are at an odd index now), `f` is updated by subtracting `x` from it, because we assume that including `x` would contribute to an alternating sum pattern. On the other hand, if we decide not to include `x`, then `f` remains the same. We choose the maximum of these two options to get the new value of `f`.
2. Similarly, when updating `g` (representing an odd index), we consider adding `x` (since we are now at an even index) to the previous `g` or keeping `g` as is. We select the maximum of these two choices.
3. As we loop through the array, `f` and `g` are updated in an alternating manner, reflecting the inclusion or exclusion of elements in the subsequence to maintain the pattern.
4. Finally, after considering all elements in `nums`, the maximum of `f` and `g` will be our answer, as it will represent the maximum alternating sum attainable by any subsequence of the original array.

Solution Approach

The solution to this problem is elegantly handled with dynamic programming, where two variables `f` and `g` represent the current best alternating sums ending on an even index and an odd index, respectively. Here's a step-by-step breakdown of how the algorithm is implemented:

1. **Initialization**: We initialize two variables `f` and `g` to zero. Here, `f` will be used to track the maximum alternating sum ending with an even index, and `g` will track the maximum alternating sum ending with an odd index.
2. **Iterating through nums**: We loop through each element `x` in the provided array `nums`. For each element, we have to decide whether including it in the subsequence would lead to a higher alternating sum.
3. **Dynamic Programming Transition**:
   - To update `f`, we take the maximum between the current value of `f` (which includes not picking `x`) and `g - x` (which accounts for picking `x` and adhering to the alternating sum rule). The equation can be written as `f = max(f, g - x)`.
   - To update `g`, we take the maximum between the current value of `g` (which also includes not picking `x`) and `f + x` (which accounts for picking `x` and maintaining the alternating pattern). The equation is `g = max(g, f + x)`.
4. **Maximizing the Result**: With each element processed, the variables `f` and `g` are updated to always reflect the highest possible alternating sums up to that point.
5. **Returning the Result**: After the loop is finished, the larger of `f` and `g` will represent the largest alternating sum of a subsequence that can be formed from the array `nums`. Since the subsequence can end with either an even or odd indexed element, we take `max(f, g)` as the final result.

Using only two variables to keep track of the state at each step makes the solution space-efficient. The dynamic programming technique utilized here is especially useful as it avoids the need to consider all possible subsequences explicitly, which would be computationally expensive. The given implementation thus runs in O(n) time, where n is the number of elements in `nums`, since it processes each element only once.

Example Walkthrough

Let's consider a small example using the array `nums = [3, 1, 6, 4]` to illustrate how the dynamic programming solution works with the given approach.

1. **Initialization**:
   - Initialize `f` and `g` to zero.
   - `f` (ends with even index) = 0
   - `g` (ends with odd index) = 0
2. **Element 1 (3 at index 0 which is even)**:
   - To update `f`, we choose max between `f` (which is 0) and `g - x` (which is also 0 because `g` is 0).
   - To update `g`, it remains 0, since 0 + 3 is less than `f`.
   - Updated values: `f = 3`, `g = 0`
3. **Element 2 (1 at index 1 which is odd)**:
   - To update `f`, we choose max between `f` (which is 3) and `g - x` (which is -1, because `g` is 0 and `x` is 1).
   - `f` doesn't change because 3 is greater than -1.
   - To update `g`, we choose max between `g` (which is 0) and `f + x` (which is 4, because `f` is 3 and `x` is 1).
   - Updated values: `f = 3`, `g = 4`
4. **Element 3 (6 at index 2 which is even)**:
   - To update `f`, we choose max between `f` (which is 3) and `g - x` (which is -2, because `g` is 4 and `x` is 6).
   - `f` doesn't change because 3 is greater than -2.
   - To update `g`, we choose max between `g` (which is 4) and `f + x` (which is 9, because `f` is 3 and `x` is 6).
   - Updated values: `f = 3`, `g = 9`
5. **Element 4 (4 at index 3 which is odd)**:
   - To update `f`, we choose max between `f` (which is 3) and `g - x` (which is 5, because `g` is 9 and `x` is 4).
   - Updated values: `f = 5`, `g` still 9, as `f + x` (which is 7) is less than `g`.

After considering all elements in `nums`, we have two final values for `f` and `g`. `f` holds the maximum alternating sum when we end on an even index and `g` when we end on an odd index.

- `f` is 5
- `g` is 9

The maximum between `f` and `g` is `g`. Therefore, the **maximum alternating sum** of any subsequence of the given array `nums` is 9.

This walkthrough captures the gradual update process of the dynamic programming approach where variables `f` and `g` facilitates the capture of the highest sums possible with alternating subsequence selection at every step without considering all subsequence combinations.

Solution Implementation

Python

```python
from typing import List  # Import the List type from typing module for type hints.

class Solution:
    def max_alternating_sum(self, nums: List[int]) -> int:
        even_index_sum = 0  # Initialize the max sum when considering even-indexed elements.
        odd_index_sum = 0  # Initialize the max sum when considering odd-indexed elements.

        for num in nums:
            # Calculate the new even_index_sum by considering the previous odd_index_sum
            # and subtracting the current number (if it leads to a larger value).
            new_even_index_sum = max(odd_index_sum - num, even_index_sum)

            # Calculate the new odd_index_sum by considering the previous even_index_sum
            # and adding the current number (if it leads to a larger value).
            new_odd_index_sum = max(even_index_sum + num, odd_index_sum)

            # Update the sums for the next iteration.
            even_index_sum, odd_index_sum = new_even_index_sum, new_odd_index_sum

        # Return the maximum of both sums, determining the max alternating sum.
        return max(even_index_sum, odd_index_sum)
```

Java

```java
class Solution {
    public long maxAlternatingSum(int[] nums) {
        // Initialize the variables 'evenSum' and 'oddSum'.
        // 'evenSum' tracks the maximum alternating sum ending with an element at an even index.
        // 'oddSum' tracks the maximum alternating sum ending with an element at an odd index.
        long evenSum = 0, oddSum = 0;

        // Iterate through the 'nums' array to calculate maximum alternating sums.
        for (int num : nums) {
            // 'nextEvenSum' will be the maximum of current 'oddSum' minus current 'num',
            // or it will remain the same as the current 'evenSum'.
            long nextEvenSum = Math.max(oddSum - num, evenSum);

            // 'nextOddSum' will be the maximum of current 'evenSum' plus current 'num',
            // or it will remain the same as the current 'oddSum'.
            long nextOddSum = Math.max(evenSum + num, oddSum);

            // Update 'evenSum' and 'oddSum' for the next iteration.
            evenSum = nextEvenSum;
            oddSum = nextOddSum;
        }

        // Return the maximum of 'evenSum' and 'oddSum' as the result.
        // It represents the maximum alternating sum that can be obtained.
        return Math.max(evenSum, oddSum);
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for 'max' function

class Solution {
public:
    // Calculates the maximum alternating sum of an array.
    long long maxAlternatingSum(vector<int>& nums) {
        long long evenIndexSum = 0; // Sum when considering elements at even indices
        long long oddIndexSum = 0;  // Sum when considering elements at odd indices

        // Loop through all elements in the array
        for (int x : nums) {
            // Update the sums at even and odd indices
            long long newEvenIndexSum = max(oddIndexSum - x, evenIndexSum);
            long long newOddIndexSum = max(evenIndexSum + x, oddIndexSum);

            // Assign the updated sums back to the variables for next iteration
            evenIndexSum = newEvenIndexSum;
            oddIndexSum = newOddIndexSum;
        }

        // Return the maximum of the two sums
        return max(evenIndexSum, oddIndexSum);
    }
};
```

TypeScript

```typescript
function maxAlternatingSum(nums: number[]): number {
    // Introduce variables to keep track of alternating sums.
    // oddSum: Sum when considering odd-indexed elements in the sequence.
    // evenSum: Sum when considering even-indexed elements in the sequence.
    let [evenSum, oddSum] = [0, 0];

    // Iterate over each number in the nums array.
    for (const num of nums) {
        // Temporarily store the current state of evenSum,
        // so we can update evenSum based on the prior value of oddSum.
        let tempEvenSum = evenSum;

        // Update evenSum: Choose the maximum between the current evenSum
        // and (oddSum - current number) to simulate the effect of "adding" an even-indexed number.
        evenSum = Math.max(oddSum - num, evenSum);

        // Update oddSum by reversing the roles: Choose the maximum between the current oddSum
        // and (tempEvenSum + current number) to simulate the effect of "adding" an odd-indexed number.
        oddSum = Math.max(tempEvenSum + num, oddSum);
    }

    // Return the maximum sum obtained by considering even-indexed elements.
    // It represents the max alternating sum for the array.
    return oddSum;
}
```

```python
from typing import List  # Import the List type from typing module for type hints.

class Solution:
    def max_alternating_sum(self, nums: List[int]) -> int:
        even_index_sum = 0  # Initialize the max sum when considering even-indexed elements.
        odd_index_sum = 0  # Initialize the max sum when considering odd-indexed elements.

        for num in nums:
            # Calculate the new even_index_sum by considering the previous odd_index_sum
            # and subtracting the current number (if it leads to a larger value),
            new_even_index_sum = max(odd_index_sum - num, even_index_sum)

            # Calculate the new odd_index_sum by considering the previous even_index_sum
            # and adding the current number (if it leads to a larger value).
            new_odd_index_sum = max(even_index_sum + num, odd_index_sum)

            # Update the sums for the next iteration.
            even_index_sum, odd_index_sum = new_even_index_sum, new_odd_index_sum

        # Return the maximum of both sums, determining the max alternating sum.
        return max(even_index_sum, odd_index_sum)
```

Time and Space Complexity

The given Python code snippet defines a function that calculates the maximum alternating sum of an array. To analyze its time and space complexity, consider the following:

Time Complexity:

- The function iterates through the list of numbers once. Let `n` be the length of `nums`.
- Within this single loop, it performs constant-time operations such as computing the maximum value between two numbers and summing or subtracting `x` from the variables `f` and `g`.
- There are no nested loops or recursive calls that would increase the complexity.
- Therefore, the time complexity is $O(n)$.

Space Complexity:

- The function uses a fixed number of integer variables `f`, and `g` irrespective of the input size.
- No additional data structures are created that grow with the size of the input.
- Hence, the space complexity of the function is $O(1)$.