1749. Maximum Absolute Sum of Any Subarray

Medium Array Dynamic Programming

Problem Description

You are given an array of integers called `nums`. The task is to find the maximum absolute sum of any subarray of this array. Let's clarify a few things first. A subarray is a contiguous part of the original array, and it can be as small as zero elements or as large as the entire array. The absolute sum is the total sum in which we consider a negative total as its positive counterpart (that is, we apply the absolute value operation to the sum). We can denote it mathematically as `abs(sum)` where `sum` is the sum of elements in the subarray. Now, the maximum absolute sum is the highest value of `abs(sum)` that you can get from all possible subarrays of `nums`.

Intuition

The intuition behind the solution comes from dynamic programming, a method where we build up to the solution by solving subproblems and using those solutions to solve larger problems. Here, we aim to track two values while iterating through the array: the maximum subarray sum so far and the minimum subarray sum so far. By keeping track of both, we can ensure that we consider both positive and negative numbers effectively because the maximum absolute sum could come from a subarray with a very negative sum due to the absolute value operation.

We define two variables, `f` and `g`, that will represent the maximum and minimum subarray sum ending at the current position, respectively. As we move through the array, we keep updating these values. If `f` drops below zero, we reset it to zero because a subarray with a negative sum would not contribute to a maximum absolute sum; we should instead start a new subarray from the next element. Similarly, if `g` becomes positive, we reset it to zero, starting a new subarray for the minimum sum calculation.

`f` keeps track of the maximum sum achieved so far (without taking the absolute value), and `g` does the same for the minimum sum. Notice that for `g`, we take the absolute value to compare with the max value `f`, because we are interested in the maximum absolute sum. Our final answer is the maximum between `f` and the absolute value of `g` throughout the entire pass. This is because for a negative subarray to contribute to the maximum absolute sum, its absolute value should be taken which might be larger than the maximum positive subarray sum.

The brilliance of this method lies in the realization that we don't need to maintain a full array of sums, but simply update our two tracking variables as we iterate through `nums`. This simplifies the process and reduces our space complexity to `O(1)`.

Solution Approach

The solution uses a simple yet powerful approach based on dynamic programming, which enables us to keep track of the maximum and minimum sums of subarrays that we have encountered so far. The dynamic programming is apparent in the way we update our running accumulations `f` and `g`, and in how they depend solely on the values at the previous index.

Let's go over the logic in detail:

- We initialize two variables, `f` and `g`, to store the current subarray's maximum and minimum sum, respectively.
- We also have a variable `ans` to store the global maximum absolute sum that we will eventually return.
- As we iterate over each element `x` in `nums`, we update `f` and `g` for every position:
  - `f` is updated to be the maximum of `f` reset to 0 (if it was negative) plus the current element `x`. This effectively means, if accumulating a sum including `x` leads to a sum less than zero, we are better off starting a new subarray at the next position.

  For `f`, the state transition can be written as:

```
f[i] = max(f[i - 1], 0) + nums[i]
```

  - `g` is updated to be the minimum of `g` reset to 0 (if it was positive) plus the current element `x`. This keeps track of the negative sums, which, when their absolute value is taken, may contribute to the absolute sum.

  The state transition for `g` is:

```
g[i] = min(g[i - 1], 0) + nums[i]
```

- After updating `f` and `g` with the current element, we update our answer `ans` to be the maximum of `ans`, `f`, and the absolute value of `g`. This ensures that `ans` always holds the highest value of either the current maximum subarray sum, or the absolute value of the current minimum subarray sum.
- After the loop completes, `ans` holds the maximum absolute sum possible from all the subarrays of `nums`.

The beauty of this algorithm lies in the fact that it computes the answer in a single pass through the input array with `O(n)` time complexity, where `n` is the length of `nums`, and it does so with `O(1)` additional space complexity, since it doesn't require storing an array of sums.

Remember, the original problem allowed for the subarray to be empty, which would mean an absolute sum of 0. However, this scenario is naturally covered in our implementation because the initialization of `ans` starts at 0, and `f` and `g` can only increase from there.

The solution could also be seen as an adaptation of the famous Kadane's algorithm, which is used to find the maximum sum subarray, here cleverly tweaked to also account for negative sums by the introduction of `g` and maximizing the absolute values.

Example Walkthrough

Let's consider an example `nums` array to illustrate the solution approach:

```
nums = [2, -1, -2, 3, -4]
```

We want to find the maximum absolute sum of any subarray of this array.

- Initialize `f` and `g` to 0. `f` is the maximum subarray sum so far, and `g` is the minimum subarray sum so far. Also, initialize `ans` to 0, which will hold the final answer.
- Start iterating over the `nums` array.

1. First element is 2:
   - Update `f`: `f = max(0, 0) + 2 = 2`
   - Update `g`: `g = min(0, 0) + 2 = 2`
   - Update `ans`: `ans = max(0, 2, abs(2)) = 2`
2. Second element is -1:
   - Update `f`: `f = max(2, 0) - 1 = 1`
   - Update `g`: `g = min(2, 0) - 1 = -1`
   - Update `ans`: `ans = max(2, 1, abs(-1)) = 2`
3. Third element is -2:
   - Update `f`: `f = max(1, 0) - 2 = -1` (But since `f` is now less than 0, we reset it to 0)
   - Update `g`: `g = min(-1, 0) - 2 = -3`
   - Update `ans`: `ans = max(2, 0, abs(-3)) = 3`
4. Fourth element is 3:
   - Update `f`: `f = max(0, 0) + 3 = 3`
   - Update `g`: `g = min(-3, 0) + 3 = 0` (But since `g` would become positive, we reset it to 0)
   - Update `ans`: `ans = max(3, 3, abs(0)) = 3`
5. Fifth element is -4:
   - Update `f`: `f = max(3, 0) - 4 = -1` (Again, reset `f` to 0 since it's less than 0)
   - Update `g`: `g = min(0, 0) - 4 = -4`
   - Update `ans`: `ans = max(3, 0, abs(-4)) = 4`

- After iterating through the entire array, the final answer (`ans`) is 4, which is the maximum absolute sum of any subarray in `nums`.

Thus, in this example, the maximum absolute sum is 4, obtained from the subarray `[-4]` whose absolute sum is `abs(-4) = 4`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
        """
        This function finds the maximum absolute sum of any non-empty subarray of the given array.

        Args:
        nums: List[int] - the list of integers over which we want to find the maximum absolute sum.

        Returns:
        The maximum absolute subarray sum.
        """

        # f_max keeps track of the maximum subarray sum ending at the current position.
        # g_min keeps track of the minimum subarray sum ending at the current position.
        f_max = g_min = 0

        # ans stores the maximum absolute sum found so far.
        ans = 0

        # Iterate through each number in nums.
        for num in nums:
            # Update the maximum subarray sum ending at the current position.
            # Reset to current number if the max sum becomes negative.
            f_max = max(f_max, 0) + num

            # Update the minimum subarray sum ending at the current position.
            # Reset to current number if the min sum becomes positive.
            g_min = min(g_min, 0) + num

            # Update the ans with the larger value between the current ans,
            # the current maximum subarray sum, and the absolute value of
            # the current minimum subarray sum.
            ans = max(ans, f_max, abs(g_min))

        # Return the maximum absolute subarray sum found.
        return ans
```

Java

```java
class Solution {
    public int maxAbsoluteSum(int[] nums) {
        int maxEndingHere = 0; // Represents the maximum subarray sum ending at the current position
        int minEndingHere = 0; // Represents the minimum subarray sum ending at the current position
        int maxAbsoluteSum = 0; // Keeps track of the maximum absolute subarray sum

        // Traverse the array
        for (int num : nums) {
            // Calculate the maximum subarray sum ending here by taking the maximum of
            // the current maximum subarray sum (extended by the current number) and 0
            maxEndingHere = Math.max(maxEndingHere + num, 0);

            // Calculate the minimum subarray sum ending here by taking the minimum of
            // the current minimum subarray sum (extended by the current number) and 0
            minEndingHere = Math.min(minEndingHere + num, 0);

            // Calculate the current maximum absolute subarray sum by taking the maximum of
            // the current maximum absolute sum, the current maximum subarray sum ending here,
            // and the absolute value of the current minimum subarray sum ending here
            maxAbsoluteSum = Math.max(maxAbsoluteSum, Math.max(maxEndingHere, Math.abs(minEndingHere)));
        }

        // Return the overall maximum absolute subarray sum found
        return maxAbsoluteSum;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the maximum absolute sum of any subarray of nums
    int maxAbsoluteSum(vector<int>& nums) {
        // Initialize the variables to keep track of the current max and min subarray sum
        int currentMaxSum = 0;
        int currentMinSum = 0;

        // Variable to store the final maximum absolute sum
        int maxAbsoluteSum = 0;

        // Iterate over the numbers in the vector
        for (int& num : nums) {

            // Update currentMaxSum: reset to zero if it becomes negative, then add the current element
            currentMaxSum = max(0, currentMaxSum) + num;

            // Update currentMinSum: reset to zero if it is positive, then add the current element
            currentMinSum = min(0, currentMinSum) + num;

            // Find the maximum between currentMaxSum and the absolute value of currentMinSum
            // Update maxAbsoluteSum if necessary
            maxAbsoluteSum = max({maxAbsoluteSum, currentMaxSum, abs(currentMinSum)});
        }

        // Return the final maximum absolute sum
        return maxAbsoluteSum;
    }
};
```

TypeScript

```typescript
function maxAbsoluteSum(nums: number[]): number {
    // Initialize local variables to track the maximum subarray sum and the minimum subarray sum
    let maxSubarraySum = 0; // f represents the maximum potential subarray sum at the current index
    let minSubarraySum = 0; // g represents the minimum potential subarray sum at the current index
    let maxAbsoluteSumResult = 0; // ans is the final answer tracking the maximum absolute sum found

    // Iterate through the array of numbers
    for (const num of nums) {
        // Update the maximum subarray sum: reset to 0 if negative, or add current number
        maxSubarraySum = Math.max(maxSubarraySum, 0) + num;

        // Update the minimum subarray sum: reset to 0 if positive, or add current number
        minSubarraySum = Math.min(minSubarraySum, 0) + num;

        // Update the maximum absolute sum found so far by comparing it with the
        // current maximum subarray sum and the absolute value of the current minimum subarray sum
        maxAbsoluteSumResult = Math.max(maxAbsoluteSumResult, maxSubarraySum, -minSubarraySum);
    }

    // Return the maximum absolute sum of any subarray
    return maxAbsoluteSumResult;
}
```

```python
from typing import List

class Solution:
    def maxAbsoluteSum(self, nums: List[int]) -> int:
        """
        This function finds the maximum absolute sum of any non-empty subarray of the given array.

        Args:
        nums: List[int] - the list of integers over which we want to find the maximum absolute sum.

        Returns:
        The maximum absolute subarray sum.
        """

        # f_max keeps track of the maximum subarray sum ending at the current position.
        # g_min keeps track of the minimum subarray sum ending at the current position.
        f_max = g_min = 0

        # ans stores the maximum absolute sum found so far.
        ans = 0

        # Iterate through each number in nums.
        for num in nums:
            # Update the maximum subarray sum ending at the current position.
            # Reset to current number if the max sum becomes negative.
            f_max = max(f_max, 0) + num

            # Update the minimum subarray sum ending at the current position.
            # Reset to current number if the min sum becomes positive.
            g_min = min(g_min, 0) + num

            # Update the ans with the larger value between the current ans,
            # the current maximum subarray sum, and the absolute value of
            # the current minimum subarray sum.
            ans = max(ans, f_max, abs(g_min))

        # Return the maximum absolute subarray sum found.
        return ans
```

Time and Space Complexity

Time complexity

The time complexity of the provided code is `O(n)` since it passes through the array `nums` once, performing a constant amount of work for each element in the array. No nested loops or recursive calls are present that would increase the complexity. Here `n` is the length of the array `nums`.

Space complexity

The space complexity of the code is `O(1)` because the space used does not grow with the size of the input array. Only a fixed number of variables `f`, `g`, and `ans` are used, irrespective of the input size.