1567. Maximum Length of Subarray With Positive Product

Medium Prefix Array Dynamic Programming

Leetcode Link

Problem Description

The given problem presents us with an array of integers, `nums`. Our task is to find the length of the longest subarray where the product of all the elements is positive. A subarray is defined as a contiguous part of the original array and can contain zero or more elements. The output should be the length of the longest such subarray.

To solve this problem, we must consider that the product of elements in a subarray is positive if there are an even number of negative numbers (including zero) because the product of two negatives is positive. We must also remember that the presence of a zero resets the product because any number multiplied by zero is zero.

Intuition

The intuition behind the solution is based on dynamic programming. We can keep track of the maximum subarray length with a positive product in two situations:

- The current number is positive
- The current number is negative

We do this through two variables, `f1` and `f2`, where `f1` represents the length of the subarray ending at the current position with a positive product, and `f2` (if non-zero) represents the length of a subarray ending at the current position with a negative product.

Here's the approach to arrive at the solution:

1. Initialize `f1` and `f2`. If the first number is positive, `f1` starts at 1, and if it's negative, `f2` starts at 1. This sets up our initial condition.
2. Iterate through the array, starting from the second element, and update `f1` and `f2` based on the current number:
   - If the current number is positive:
     - Increment `f1` because a positive number won't change the sign of the product.
     - If `f2` is non-zero, increment it as well, as a positive number won't affect the sign of an already negative product.
   - If the current number is negative:
     - Assign `f1` the value of `f2 + 1`, because a negative number changes the sign of the product.
     - Assign `f2` the value of `f1` (before updating `f1`) + 1, again because a negative number changes the sign.
   - If the current number is zero, reset both `f1` and `f2` to zero because the product would be zero regardless of the preceding values.
3. At each step, compare `f1` with our current maximum length (`res`) and update `res` if `f1` is larger.
4. Once we've iterated through the array, return `res` as it contains the maximum length of a subarray with a positive product.

Thus, using this approach, we are able to avoid calculating the product of subarray elements and can find the maximum length of a subarray with a positive product efficiently.

Solution Approach

The solution uses a dynamic programming approach that revolves around two main concepts: tracking the longest subarray length where the product is positive (`f1`), and tracking the longest subarray length where the product is negative (`f2`). To implement this, we use a simple linear scan of the input array with constant space complexity, updating both `f1` and `f2` based on the sign and value of the current element.

Here is a step-by-step explanation of the algorithm:

1. Initialize `f1` and `f2`:
   - `f1` is set to 1 if the first element of `nums` is positive, indicating the subarray starts with a positive product.
   - `f2` is set to 1 if the first element of `nums` is negative, indicating the subarray starts with a negative product.
   - `res` is initialized to `f1` to keep track of the maximum length.
2. Iterate over the array starting from the second element:
   - For each element, there are three cases to consider: a. The current element is positive:
     - Increment `f1` as the positive number will not change the sign of the product.
     - Increment `f2` only if it is positive, meaning there was a previous negative number and this positive number extends the subarray with a now positive product. b. The current element is negative:
     - Swap `f1` and `f2` and increment `f2`, as the negative number makes a previously positive product negative.
     - Only increment `f1` if `f2` was originally positive, as this indicates we had a negative product and now have made it positive. c. The current element is zero:
     - Reset `f1` and `f2` to 0 as any subarray containing a zero has a product of zero, which is neither positive nor negative.
3. After considering the current number, update the maximum length `res` with the current value of `f1` if `f1` is greater than the current `res`.
4. Continue the loop until the end of the array.
5. Return the final value of `res`, which contains the maximum length of the subarray with a positive product.

The code uses constant extra space (only needing `f1`, `f2`, and `res`), and it runs in O(n) time, where n is the length of the input array `nums`. This is an efficient solution because it avoids the need to calculate products directly and uses the properties of positive and negative numbers to infer the sign of the subarray products.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the following array of integers:

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

Our goal is to find the length of the longest subarray where the product of all elements is positive.

1. Initialize `f1`, `f2`, and `res`:
   - The first element is positive, so `f1 = 1`.
   - There are no negative elements yet, so `f2 = 0`.
   - Initialize `res` with the value of `f1`, hence `res = 1`.
2. Iterate over the array:
   a. Second element, `-2` (current element is negative):
   - Since `f1 = 1` and `f2 = 0`, we update `f2` to `f1 + 1`, resulting in `f2 = 2`.
   - There is no previously recorded negative subarray (`f2` was 0), so `f1` remains the same.
   - No update to `res` as `f1` has not increased.
   b. Third element, `-3` (current element is negative):
   - Swap `f1` and `f2`. Before the swap, `f1 = 1` and `f2 = 2`. After the swap, `f1` is now 2.
   - Increment `f2` to become `f1(pre-swap) + 1`, hence `f2 = 2`.
   - Update `res` with the current value of `f1` which is now 2, as it is greater than the previous `res`.
   c. Fourth element, `4` (current element is positive):
   - Increment `f1` to `f1 + 1`, so `f1` becomes 3.
   - Increment `f2` as well since it's non-zero, so `f2` becomes 3.
   - Update `res` with the current value of `f1` which is 3 as it is greater than the previous `res`.
3. After iterating through the array, the maximum length `res` is 3. We have found the longest subarray `[1, -2, -3, 4]`, which indeed has a positive product and its length is 3.

Therefore, the length of the longest subarray with a positive product is 3.

Python Solution

```python
from typing import List

class Solution:
    def getMaxLen(self, nums: List[int]) -> int:
        # f_positive is the length of the longest subarray with a positive product ending at the current position
        f_positive = 1 if nums[0] > 0 else 0

        # f_negative is the length of the longest subarray with a negative product ending at the current position
        f_negative = 1 if nums[0] < 0 else 0

        # res is the length of the longest subarray with a positive product found so far
        res = f_positive

        # Iterate through the array starting from the second element
        for num in nums[1:]:
            # Store previous f_positive and f_negative before updating
            prev_f_positive, prev_f_negative = f_positive, f_negative

            # When the current number is positive
            if num > 0:
                # Extend the subarray with a positive product
                f_positive += 1
                # If there was a subarray with a negative product, extend it too; otherwise, reset to 0
                f_negative = prev_f_negative + 1 if prev_f_negative > 0 else 0

            # When the current number is negative
            elif num < 0:
                # The new subarray with a negative product becomes the previous positive subarray plus the current negative number
                f_negative = prev_f_positive + 1
                # If there was a subarray with a negative product, it becomes positive now; otherwise, reset to 0
                f_positive = prev_f_negative + 1 if prev_f_negative > 0 else 0

            # When the current number is zero, reset both counts to 0
            else:
                f_positive = 0
                f_negative = 0

            # Update res to be the max of itself and the current positive subarray length
            res = max(res, f_positive)

        # After iterating through the array, return the result
        return res
```

Java Solution

```java
class Solution {

    public int getMaxLen(int[] nums) {
        int positiveCount = nums[0] > 0 ? 1 : 0; // Initialize the length of positive product subarray
        int negativeCount = nums[0] < 0 ? 1 : 0; // Initialize the length of negative product subarray
        int maxLength = positiveCount; // Store the maximum length of subarrays with positive product

        // Iterate over the array starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] > 0) {
                // If the current number is positive, increase the length of positive product subarray
                ++positiveCount;
                // If there was a negative product subarray, increase its length too
                negativeCount = negativeCount > 0 ? negativeCount + 1 : 0;
            } else if (nums[i] < 0) {
                // If the current number is negative, swap the lengths of positive and negative product subarrays
                int previousPositiveCount = positiveCount;
                int previousNegativeCount = negativeCount;
                // The length of negative product subarray is now positive
                negativeCount = previousPositiveCount > 0 ? previousNegativeCount + 1 : 0;
                positiveCount = previousNegativeCount + 1;
            } else {
                // If the current number is zero, reset the lengths as any sequence will be discontinued by zero
                positiveCount = 0;
                negativeCount = 0;
            }
            // Update the maximum length if the current positive product subarray is longer
            maxLength = Math.max(maxLength, positiveCount);
        }
        return maxLength; // Return the maximum length found
    }
}
```

C++ Solution

```cpp
#include <vector> // Include necessary header for vector usage
#include <algorithm> // Include for the max() function

class Solution {
public:
    int getMaxLen(vector<int>& nums) {
        // Initialize lengths of subarrays with positive product (positiveLen) and negative product (negativeLen)
        int positiveLen = nums[0] > 0 ? 1 : 0;
        int negativeLen = nums[0] < 0 ? 1 : 0;
        int result = positiveLen; // This will store the maximum length of subarray with positive product

        // Iterate through the array starting from the second element
        for (int i = 1; i < nums.size(); ++i) {
            if (nums[i] > 0) {
                // If the current number is positive, increment positive length
                ++positiveLen;
                // If there was a negative product, increment negative length, otherwise reset to 0
                negativeLen = negativeLen > 0 ? negativeLen + 1 : 0;
            } else if (nums[i] < 0) {
                // Store the previous lengths before updating
                int prevPositiveLen = positiveLen;
                int prevNegativeLen = negativeLen;

                // If the current number is negative, the new negative length is
                // the previous positive length + 1, as it changes the sign
                negativeLen = prevPositiveLen + 1;

                // The new positive length would be the previous negative length + 1
                // if there has been no negative number yet, reset positive length to 0
                positiveLen = prevNegativeLen > 0 ? prevNegativeLen + 1 : 0;
            } else { // If the current number is 0, reset lengths as the product is broken
                positiveLen = 0;
                negativeLen = 0;
            }

            // Update result if the current positive length is greater
            result = std::max(result, positiveLen);
        }

        return result; // Return the maximum length of subarray with positive product
    }
};
```

Typescript Solution

```typescript
function getMaxLen(nums: number[]): number {
    // Initialize counts for positive sequence (posSeqCount) and negative sequence (negSeqCount)
    let posSeqCount = nums[0] > 0 ? 1 : 0;
    let negSeqCount = nums[0] < 0 ? 1 : 0;

    // Initialize the answer (maxLen) with the count of the positive sequence
    let maxLen = posSeqCount;

    // Loop through the array starting from index 1
    for (let i = 1; i < nums.length; ++i) {
        let current = nums[i]; // Current number in the array

        if (current === 0) {
            // If current number is 0, reset the counts
            posSeqCount = 0;
            negSeqCount = 0;
        } else if (current > 0) {
            // If current number is positive, increment the positive sequence count
            ++posSeqCount;
            // Increment the negative counts, increment it, otherwise set to 0
            negSeqCount = negSeqCount > 0 ? negSeqCount + 1 : 0;
        } else {
            // If current number is negative, swap the counts after incrementing
            let tempPosSeq = posSeqCount;
            let tempNegSeq = negSeqCount;
            // Increment negative sequence count if there's a positive sequence; otherwise set to 0
            posSeqCount = tempNegSeq > 0 ? tempNegSeq + 1 : 0;
            // Always increment the negative sequence count since current number is negative
            negSeqCount = tempPosSeq + 1;
        }

        // Update maxLen to the greater of maxLen or the current positive sequence count
        maxLen = Math.max(maxLen, posSeqCount);
    }

    // Return the maximum length of subarray of positive product
    return maxLen;
}
```

Time and Space Complexity

The provided code maintains two variables `f1` and `f2` to keep track of the length of the longest subarray with a positive product and the length of the longest subarray with a negative product, respectively.

Time Complexity:

The function iterates through the list `nums` once. For each element in `nums`, it performs a constant number of computations: updating `f1`, `f2`, and `res` based on the sign of the current element. Therefore, the time complexity is `O(n)`, where `n` is the number of elements in `nums`.

Space Complexity:

The space complexity is `O(1)` because the code uses a fixed amount of space: variables `f1`, `f2`, `res`, `pf1`, and `pf2`. The space used does not grow with the size of the input array.