1567. Maximum Length of Subarray With Positive Product

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:

```
1  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.
3. After iterating through the array, the maximum length `res` is 3 as it is greater than the previous `res`. 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
 1   from typing import List
 2
 3   class Solution:
 4       def getMaxLen(self, nums: List[int]) -> int:
 5           # f_positive is the length of the longest subarray with a positive product ending at the current position
 6           f_positive = 1 if nums[0] > 0 else 0
 7
 8           # f_negative is the length of the longest subarray with a negative product ending at the current position
 9           f_negative = 1 if nums[0] < 0 else 0
10
11           # res is the length of the longest subarray with a positive product found so far
12           res = f_positive
13
14           # Iterate through the array starting from the second element
15           for num in nums[1:]:
16               # Store previous f_positive and f_negative before updating
17               prev_f_positive, prev_f_negative = f_positive, f_negative
18
19               # When the current number is positive
20               if num > 0:
21                   # Extend the subarray with a positive product
22                   f_positive += 1
23                   # If there was a subarray with a negative product, extend it too; otherwise, reset to 0
24                   f_negative = prev_f_negative + 1 if prev_f_negative > 0 else 0
25
26               # When the current number is negative
27               elif num < 0:
28                   # The new subarray with a negative product becomes the previous positive subarray plus the current negative number
29                   f_negative = prev_f_positive + 1
30                   # If there was a subarray with a negative product, it becomes positive now; otherwise, reset to 0
31                   f_positive = prev_f_negative + 1 if prev_f_negative > 0 else 0
32
33               # When the current number is zero, reset both counts to 0
34               else:
35                   f_positive = 0
36                   f_negative = 0
37
38               # Update res to be the max of itself and the current positive subarray length
39               res = max(res, f_positive)
40
41           # After iterating through the array, return the result
42           return res
```

Java Solution

```java
 1   class Solution {
 2
 3       public int getMaxLen(int[] nums) {
 4           int positiveCount = nums[0] > 0 ? 1 : 0; // Initialize the length of positive product subarray
 5           int negativeCount = nums[0] < 0 ? 1 : 0; // Initialize the length of negative product subarray
 6           int maxLength = positiveCount; // Store the maximum length of subarray with positive product
 7
 8           // Iterate over the array starting from the second element
 9           for (int i = 1; i < nums.length; ++i) {
10               if (nums[i] > 0) {
11                   // If the current number is positive, increase the length of positive product subarray
12                   ++positiveCount;
13                   // If there was a negative product subarray, increase its length too
14                   negativeCount = negativeCount > 0 ? negativeCount + 1 : 0;
15               } else if (nums[i] < 0) {
16                   // If the current number is negative, swap the lengths of positive and negative product subarrays
17                   int previousPositiveCount = positiveCount;
18                   positiveCount = negativeCount > 0 ? negativeCount + 1 : 0;
19                   negativeCount = previousPositiveCount + 1;
20               } else {
21                   // If the current number is zero, reset the lengths as any sequence will be discontinued by zero
22                   positiveCount = 0;
23                   negativeCount = 0;
24               }
25               // Update the maximum length if the current positive product subarray is longer
26               maxLength = Math.max(maxLength, positiveCount);
27           }
28           return maxLength; // Return the maximum length found
29       }
30   }
```

C++ Solution

```cpp
 1   #include <vector> // Include necessary header for vector usage
 2   #include <algorithm> // Include for the max() function
 3
 4   class Solution {
 5   public:
 6       int getMaxLen(std::vector<int>& nums) {
 7           // Initialize lengths of subarrays with positive product (positiveLen) and negative product (negativeLen)
 8           int positiveLen = nums[0] > 0 ? 1 : 0;
 9           int negativeLen = nums[0] < 0 ? 1 : 0;
10           int result = positiveLen; // This will store the maximum length of subarray with positive product
11
12           // Iterate through the array starting from the second element
13           for (int i = 1; i < nums.size(); ++i) {
14               if (nums[i] > 0) {
15                   // If the current number is positive, increment positive length
16                   ++positiveLen;
17                   // If there was a negative product, increment negative length; otherwise set to 0
18                   negativeLen = negativeLen > 0 ? negativeLen + 1 : 0;
19               } else if (nums[i] < 0) {
20                   // Store the previous lengths before updating
21                   int prevPositiveLen = positiveLen;
22                   int prevNegativeLen = negativeLen;
23
24                   // If the current number is negative, the new negative length is
25                   // the previous positive length + 1, as it changes the sign
26                   negativeLen = prevPositiveLen + 1;
27
28                   // The new positive length would be the previous negative length + 1
29                   // If there has been no negative number yet, reset positive length to 0
30                   positiveLen = prevNegativeLen > 0 ? prevNegativeLen + 1 : 0;
31               } else { // If the current number is 0, reset lengths as the product is broken
32                   positiveLen = 0;
33                   negativeLen = 0;
34               }
35
36               // Update result if the current positive length is greater
37               result = std::max(result, positiveLen);
38           }
39
40           return result; // Return the maximum length of subarray with positive product
41       }
42   };
```

Typescript Solution

```typescript
 1   function getMaxLen(nums: number[]): number {
 2       // Initialize counts for positive sequence (posSeqCount) and negative sequence (negSeqCount)
 3       let posSeqCount = nums[0] > 0 ? 1 : 0;
 4       let negSeqCount = nums[0] < 0 ? 1 : 0;
 5
 6       // Initialize the answer (maxLen) with the count of the positive sequence
 7       let maxLen = posSeqCount;
 8
 9       // Loop through the array starting from index 1
10       for (let i = 1; i < nums.length; ++i) {
11           let current = nums[i]; // Current number in the array
12
13           if (current === 0) {
14               // If current number is 0, reset the counts
15               posSeqCount = 0;
16               negSeqCount = 0;
17           } else if (current > 0) {
18               // If current number is positive, increment the positive sequence count
19               posSeqCount++;
20               // If there are negative counts, increment it, otherwise set to 0
21               negSeqCount = negSeqCount > 0 ? negSeqCount + 1 : 0;
22           } else {
23               // If current number is negative, swap the counts after incrementing
24               let tempPosSeq = posSeqCount;
25               let tempNegSeq = negSeqCount;
26               // Increment negative sequence count if there's a positive sequence; otherwise set to 0
27               posSeqCount = tempNegSeq > 0 ? tempNegSeq + 1 : 0;
28               // Always increment the negative sequence count since current number is negative
29               negSeqCount = tempPosSeq + 1;
30           }
31
32           // Update maxLen to the greater of maxLen or the current positive sequence count
33           maxLen = Math.max(maxLen, posSeqCount);
34       }
35
36       // Return the maximum length of subarray of positive product
37       return maxLen;
38   }
```

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.