1822. Sign of the Product of an Array

Easy Array Math [Leetcode Link]

Problem Description

The problem gives us a `signFunc(x)` function that returns `1` if `x` is positive, `-1` if `x` is negative, and `0` if `x` is zero. We are also provided with an integer array `nums`, and we need to calculate the product of all the values in this array. The final goal is not to return the actual product but the sign of this product as determined by `signFunc`.

To put it simply, we must determine whether the product of all numbers in the array is positive, negative, or zero, without actually multiplying the numbers (as this might cause overflow with large values).

Intuition

To arrive at the solution approach, we recognize that the sign of a product of numbers is determined by the following rules:

1. If any number in the product is `0`, the product is `0`.
2. If there is an even number of negative numbers in the product, the product is positive.
3. If there is an odd number of negative numbers in the product, the product is negative.

We do not need to calculate the actual product because we are only interested in the sign. Therefore, we can use these rules to determine the sign by iterating over the array and keeping track of two things: whether we have encountered a zero, and the count of negative numbers.

The solution code demonstrates this approach efficiently:

- It initializes a variable `ans` to `1`, which will be used to keep track of the sign (positive or negative).
- It iterates over each number `v` in the array `nums`:
  - If `v` is `0`, the function immediately returns `0` since the product would be `0`.
  - If `v` is negative, `ans` is multiplied by `-1`, effectively flipping the sign of `ans`.
- After the loop, the algorithm returns the value of `ans`.

This approach avoids unnecessary calculations and potential integer overflow, directly giving us the sign of the product as required by the problem definition.

Solution Approach

The solution is straightforward and uses a simple linear traversal algorithm. It does not depend on any complex data structures or patterns. Instead, it leverages basic variables and control-flow statements to determine the final sign. Here's a detailed walk-through of the implementation:

1. **Initialization**: A variable `ans` is set to `1`. This variable will hold our "sign accumulator". Instead of accumulating the actual product, we will only track changes in its sign as we iterate through the numbers in the array.

2. **Iteration through `nums`**: The program enters a loop where it examines each value `v` in the array `nums`. There are two cases when `v` affects `ans`:

   - If `v` is `0`: We directly return `0` since a zero in the product will always result in zero.
   - If `v` is less than `0`: This indicates a negative number. Each negative number flips the sign of the final product, so `ans` is multiplied by `-1` which has the effect of toggling its value between `1` and `-1`.

3. **Sign Determination**: The loop will skip positive numbers since they do not affect the sign of the product. After the loop, we are left with `ans` that correctly represents the sign of the product (positive or negative), or we would have already returned `0` if a zero was found.

4. **Return Result**: The function concludes by returning `ans`, which by the end of the process reflects the sign of the product according to the sign function definition provided.

This method effectively eliminates the need for any product calculation, and instead relies solely on sign modification, which is both time and space-efficient since it uses constant extra space (`ans`) and linear time in proportion to the length of `nums`.

Example Walkthrough

Let's illustrate the solution approach using an example array `nums`.

Suppose `nums = [-1, 2, 0, 3, -2]`. Following the steps outlined in the solution approach:

1. **Initialization**: We begin by setting `ans` to `1`. This will be our sign accumulator.

2. **Iteration through `nums`**:

   - First, we examine `-1`. Since it's negative, we multiply `ans` by `-1`. Now, `ans = -1`.
   - Next is `2`. It's positive, so it doesn't affect the sign of the product. `ans` remains `-1`.
   - Then comes `0`. As per our rules, if the product includes a zero, the entire product is `0`. Hence, we immediately return `0`.
   - No need to check `3` and `-2` because we have already encountered a zero and returned the result.

3. **Sign Determination**: As we encountered a zero, the sign determination step is not reached in this example.

4. **Return Result**: We have already returned `0` after encountering the zero. Therefore, for the given array, the result is `0`.

This example shows how the algorithm efficiently concludes at the presence of zero without considering all the elements. If there were no zero, the algorithm would continue until the end of the array to determine the sign based on the count of negative numbers.

Python Solution

```python
from typing import List

class Solution:
    def arraySign(self, nums: List[int]) -> int:
        # Initialize the sign of the product of the array elements as positive (1)
        product_sign = 1

        # Iterate over each value in the numbers list
        for value in nums:
            # If a zero is found, the product is zero, so return 0 immediately
            if value == 0:
                return 0
            # If a negative number is found, flip the sign of the product
            if value < 0:
                product_sign *= -1

        # Return the sign of the product of the array elements
        return product_sign

```

Java Solution

```java
class Solution {

    /**
     * Determines the sign of the product of an array of numbers.
     * The result is 1 if the product is positive, -1 if negative, and 0 if any number is 0.
     *
     * @param nums the array of integers
     * @return the sign of the product of the input array
     */
    public int arraySign(int[] nums) {
        // Initialize the sign as positive (1)
        int productSign = 1;

        // Iterate over each value in the array
        for (int value : nums) {
            // If any number is zero, the product is zero, so return 0
            if (value == 0) {
                return 0;
            }
            // If the number is negative, flip the current sign
            if (value < 0) {
                productSign *= -1;
            }
        }

        // Return the sign of the product
        return productSign;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function returns the sign of the product of all numbers in a vector
    int arraySign(vector<int>& nums) {
        // Initialize the sign as positive
        int sign = 1;

        // Loop through each number in the vector
        for (int num : nums) {
            // If the current number is zero, the product will be zero
            if (num == 0) return 0;

            // If the current number is negative, flip the sign
            if (num < 0) sign *= -1;
        }

        // Return the sign of the product of all numbers
        return sign;
    }
};

```

Typescript Solution

```typescript
/**
 * Determines the sign of the product of an array of numbers.
 * - If the product is positive, returns 1.
 * - If the product is negative, returns -1.
 * - If any element is zero, returns 0 immediately as the product is zero.
 *
 * @param {number[]} nums - The array of numbers to determine the sign of the product.
 * @return {number} - The sign of the product as 1, -1, or 0.
 */
function arraySign(nums: number[]): number {
    let productSign: number = 1; // Represents the sign of the product, initialized to positive.

    for (const value of nums) {
        if (value === 0) {
            return 0; // If any number is 0, the product is 0.
        }

        if (value < 0) {
            productSign *= -1; // Flip the sign when encountering a negative number.
        }
    }

    return productSign; // Return the sign of the product.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of elements in the input list `nums`. This is because the code iterates through each element of the list exactly once to determine the sign of the product.

Space Complexity

The space complexity of the given code is `O(1)`. This is constant space complexity because the code only uses a fixed number of extra variables (`ans`) regardless of the input size. There are no additional data structures used that scale with the input size.