238. Product of Array Except Self

Prefix Sum

Medium <u>Array</u>

Problem Description

The problem provided relates to an array transformation challenge. Specifically, you are given an integer array named nums and the task is to generate a new array named answer. For each index i in the new array, answer[i] should be the product of all the elements in nums except for nums[i]. For example, if nums = [1, 2, 3, 4], then answer = [24, 12, 8, 6] because: • answer[0] should be 2*3*4 = 24

- answer[1] should be 1*3*4 = 12
- answer[2] should be 1*2*4 = 8 • answer[3] should be 1*2*3 = 6
- Important constraints to note in this problem include:

nums[i] for each answer[i]).

element, respectively.

numbers to the right of the current index.

- The resultant product of any index will fit within a 32-bit integer, meaning we won't have integer overflow issues for the given range of input.
 - array. • The division operation cannot be used, which would have been the straightforward approach (multiplying all elements and then dividing by each

• The desired algorithm should run with a time complexity of O(n), which implies a solution should be achievable in a single pass through the

These constraints guide us to find a more creative solution that involves using multiplication only and finding a way to accumulate and distribute the products without dividing.

Intuition The solution to this problem involves accumulating products from both ends of the array. Since we can't use division, we need to

think about how we can get the product of all elements except for the current one. A smart way to do this is by multiplying two separate products: one accumulated from the beginning of the array up to the element right before the current one, and another

from the end of the array to the element right after the current one. To elaborate: We initialize two variables, left and right, to 1. They will hold the products of elements to the left and right of the current

We traverse the array from the left; each ans [i] gets the current product on its left (initialized with 1 for the first element as

- there are no elements to its left), then we update left to include nums[i]. So during this pass, left accumulates the product of all numbers to the left of the current index. Next, we traverse the array from the right; we multiply each ans [i] with the current product on its right (again, starting with
- 1), and after the multiplication, we update right to include nums[i]. During this pass, right accumulates the product of all

nums [i], effectively accumulating the product up to the current element.

We create an array ans to store our results, starting with default values (often zeroes).

- By the end of these two passes, each ans [i] will have been multiplied by both the products of the numbers to its left and right, effectively giving us the product of all other numbers except for nums [i].
- two passes over the array.
- products. The algorithm does not use complex data structures or patterns; instead, it builds on the simple principle of

Initialize an array ans to hold our results, filled with zeros, and ensuring that it has the same length as the input array nums.

Two variables, left and right, are initialized to 1 to hold the running product of the elements to the left and right of the

The first loop iterates over the elements of nums from left to right. For each element at index i, we set ans [i] to the current

value of left since left contains the product of all elements before nums[i]. Then, we update left by multiplying it with

This first loop is described by the following code snippet:

ans[i] *= right

right *= nums[i]

for i, x in enumerate(nums):

current element.

- ans[i] = left left *= x
- the first pass. Now, for each element at index i, we multiply the current value in ans[i] (the left product) with right, which
- accumulating the product from the end of the array towards the beginning. The second loop corresponds to the following code snippet: for i in range(n - 1, -1, -1):

```
    Initialize the ans array with the same length as nums, in this case, ans = [0, 0, 0, 0] for storing the results.

    Initialize variables left and right with the value 1 to accumulate products from the left and right respectively.

First Pass (Left to Right):
```

ans[0] = left (which is 1 initially).

■ ans now becomes [1, 0, 0, 0].

ans is updated to [1, 2, 0, 0].

■ Update left to left * nums[0], now left = 2.

■ Update left to left * nums[1], now left = 6.

Step-by-Step Walkthrough

Initialization:

Iteration details:

1. i = 0:

2. i = 1: ans[1] = left (which is 2 now).

```
3.i = 2:
    ans[2] = left (which is 6).
```

ans[3] = left (which is 24). left is no longer updated since it's the last element. ■ Final ans after the first pass: [1, 2, 6, 24].

Iterate over nums from left to right, and calculate the left product excluding the current element.

- Update right to right * nums[3], now right = 5. 2.i = 2: ans[2] = ans[2] * right (6 * 5), so ans[2] becomes 30.
- right update is not needed since it's the final element. **Final Result:**

This matches what we expect as:

- ans [0] = 3 * 4 * 5 = 60

 \blacksquare ans [1] = 2 * 4 * 5 = 40

Solution Implementation

from typing import List

left = 1

right = 1

num_length = len(nums)

answer = $[1] * num_length$

for i in range(num_length):

for i in range(num_length - 1, -1, -1):

// Method to calculate the product of elements except self

// Initialize the answer array with the same length

// Loop through the array from left to right

// Loop through the array from right to left

// Return the final product except self array

for (int $i = length - 1; i >= 0; i--) {$

result[i] *= rightProduct;

rightProduct *= nums[i];

public int[] productExceptSelf(int[] nums) {

// Get the length of the input array

int[] result = new int[length];

for (int i = 0; i < length; i++) {</pre>

result[i] = leftProduct;

leftProduct *= nums[i];

answer[i] = left

left *= nums[i]

answer[i] *= right

right *= nums[i]

int length = nums.length;

int leftProduct = 1;

int rightProduct = 1;

return result;

class Solution:

Python

 \blacksquare ans [2] = 2 * 3 * 5 = 30 \blacksquare ans [3] = 2 * 3 * 4 = 24

return answer Java

// Initialize 'leftProduct' to 1, to represent the product of elements to the left of the current index

// Multiply 'leftProduct' by the current element in nums for the next iteration (prefix product)

// Initialize 'rightProduct' to 1, to represent the product of elements to the right of the current index

// Multiply 'rightProduct' by the current element in nums for the next iteration (suffix product)

Multiply the existing value in 'answer' by the 'right' value which is the product of elements to the right of 'i'.

C++

// Multiply the current element in the result array by 'rightProduct'

// Initialize the right running product as 1 int rightProduct = 1; // Calculate the running product from the right for each element for (int index = size - 1; index >= 0; --index) {

const length = nums.length; // Total number of elements in input array

// Forward pass: Calculate the product of all elements to the left of each index for (let i = 0, productToLeft = 1; i < length; i++) {</pre> result[i] = productToLeft; // Set the product (initially 1) productToLeft *= nums[i]; // Update the productToLeft for the next index for (let i = length - 1, productToRight = 1; $i \ge 0$; i--) {

function productExceptSelf(nums: number[]): number[] {

from typing import List class Solution: def productExceptSelf(self, nums: List[int]) -> List[int]: # Determine the length of the input list 'nums'. num_length = len(nums) # Initialize the answer list with 1's of the same length as 'nums'. answer = $[1] * num_length$

'left' will represent the cumulative product of elements to the left of the current element.

'right' will represent the cumulative product of elements to the right of the current element.

Store the cumulative product in the 'answer' list at the current index 'i'.

- # Multiply the existing value in 'answer' by the 'right' value which is the product of elements to the right of 'i'. answer[i] *= right # Update 'right' to include the product of the current element. right *= nums[i]
- Time and Space Complexity

The time complexity of the code is O(n) because there are two separate for-loops that each run through the list of n elements exactly once. The first loop calculates the product of all elements to the left of the current element, and the second loop calculates the product of all elements to the right. Since each loop operates independently and sequentially, the procedures do not multiply in complexity but rather add, resulting in 2n, which simplifies to 0(n).

This approach cleverly bypasses the need for division and adheres to the time complexity requirement since we are only making Solution Approach The implementation of the solution leverages a single-dimensional array for storage and two variables for the accumulation of accumulation and iteration over the input array. Here's a step-by-step breakdown:

The second loop iterates over the elements of nums from right to left. We've already captured the products on the left side in now represents the product of elements after nums [i]. Subsequently, we update right by multiplying it with nums [i],

constraints by maintaining a linear time complexity O(n) as we traverse the input array exactly twice, no matter its size. **Example Walkthrough** Let's walk through a small example to illustrate the solution approach. Consider the array nums = [2, 3, 4, 5].

By maintaining two pointers that accumulate the product of elements to the left and right of the current index, and by updating

our result array in-place, we manage to get the desired products without ever using division. Moreover, we meet the complexity

■ Update left to left * nums[2], now left = 24. So, ans becomes [1, 2, 6, 0].

Iteration details:

4. i = 3:

1.i = 3:

3. i = 1:

- **Second Pass (Right to Left):** • Iterate over nums from right to left, and calculate the right product excluding the current element.
- ans[1] = ans[1] * right (2 * 20), thus ans[1] becomes 40. Update right to right * nums[1], which makes right = 60. 4.i = 0:

■ Update right to right * nums[2], now right = 20.

ans [3] is multiplied by right (1 initially), so ans [3] remains 24.

ans[0] = ans[0] * right (1 * 60), resulting in ans[0] being 60.

• The resulting ans array after both passes reflects the products of all elements for each position excluding the element itself: ans = [60, 40, 30, 24].

passes through the array twice regardless of its size.

def productExceptSelf(self, nums: List[int]) -> List[int]:

Determine the length of the input list 'nums'.

This walkthrough indicates how the solution approach effectively computes the desired output without the need for division, and it successfully avoids any overflow issues within a 32-bit signed integer range. The time complexity of the solution is 0(n), as it

Initialize the answer list with 1's of the same length as 'nums'.

Update 'left' to include the product of the current element.

Update 'right' to include the product of the current element.

// Set the current element in the result array to 'leftProduct'

'left' will represent the cumulative product of elements to the left of the current element.

'right' will represent the cumulative product of elements to the right of the current element.

Store the cumulative product in the 'answer' list at the current index 'i'.

Return the 'answer' list which now contains the products except self. class Solution {

```
class Solution {
public:
   vector<int> productExceptSelf(vector<int>& nums) {
        int size = nums.size(); // Store the size of the input vector
       vector<int> output(size); // Initialize the output vector with the same size as the input
       // Initialize the left running product as 1
       int leftProduct = 1;
       // Calculate the running product from the left for each element
        for (int index = 0; index < size; ++index) {</pre>
            output[index] = leftProduct; // Set the current output element as the product so far from the left
            leftProduct *= nums[index]; // Update the leftProduct to include the current number
            output[index] *= rightProduct; // Multiply the current output element by the product so far from the right
            rightProduct *= nums[index]; // Update the rightProduct to include the current number
        return output; // Return the final output vector containing the product of all elements except self for each position
};
TypeScript
```

// Backward pass: Calculate the product of all elements to the right of each index result[i] *= productToRight; // Multiply with the already stored product to the left productToRight *= nums[i]; // Update the productToRight for the previous index return result; // Return the final result array

const result: number[] = new Array(length); // Initialize an array to store the result

right = 1for i in range(num_length - 1, -1, -1): # Return the 'answer' list which now contains the products except self.

Update 'left' to include the product of the current element.

return answer

constant space complexity.

for i in range(num_length):

answer[i] = left

left *= nums[i]

left = 1

The space complexity is 0(1), excluding the output array ans. This is because the algorithm uses a constant number of extra variables (left, right, i, x). The space taken by these variables does not scale with the size of the input array nums, hence