LeetCode 238 — Product of Array Except Self

1. Problem Title & Link

- **Title:** LeetCode 238 — Product of Array Except Self
- **Link:** https://leetcode.com/problems/product-of-array-except-self/

2. Problem Statement (Short Summary)

You are given an integer array nums.
For each index i, create an output array where:

output[i] = product of all elements except nums[i]

⚠️ Without using division.

⚠️ Must run in O(n) time.

3. Examples (Input → Output)

Example 1

Input: nums = [1,2,3,4]

Output: [24,12,8,6]

Example 2

Input: nums = [-1,1,0,-3,3]

Output: [0,0,9,0,0]

4. Constraints

- 2 <= nums.length <= 10^5
- -30 <= nums[i] <= 30
- No division allowed.
- Must be O(n) time and O(1) extra space (result excluded).

5. Core Concept (Pattern / Topic)

Prefix Product + Suffix Product

We compute:

- **Prefix product**: product of elements to the **left**
- **Suffix product**: product of elements to the **right**

Then multiply them for each index.

This is a classic **prefix–suffix array pattern**.

6. Thought Process (Step-by-Step Explanation)

❌ **Brute Force (O(n²))**

For each index, multiply all elements except itself → too slow.

❌ Using full prefix[] + suffix[] arrays (O(n) space)

Works, but not optimal.

✅ Optimal O(n) time, O(1) space

We do it in **two scans**:

1️⃣ Prefix scan (left → right)

- Build product of all left elements
- Store it directly in result array

2️⃣ Suffix scan (right → left)

- Multiply suffix (product of right elements) into result array

This gives:

result[i] = (product of nums[0..i-1]) * (product of nums[i+1..end])

Perfect and efficient.

7. Visual / Intuition Diagram

Given:

nums = [1, 2, 3, 4]

Prefix products:

[1, 1, 2, 6]

Suffix products:

[24, 12, 4, 1]

Final result:

multiply prefix × suffix → [24, 12, 8, 6]

8. Pseudocode

```
initialize result array with 1s
prefix = 1

for i in 0..n-1:
    result[i] = prefix
    prefix = prefix * nums[i]

suffix = 1

for i in n-1..0:
    result[i] = result[i] * suffix
    suffix = suffix * nums[i]

return result
```

9. Code Implementation

✅ Python

```
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        res = [1] * n

        prefix = 1
        for i in range(n):
            res[i] = prefix
            prefix *= nums[i]

        suffix = 1
        for i in range(n - 1, -1, -1):
            res[i] *= suffix
            suffix *= nums[i]

        return res
```

✅ Java

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] res = new int[n];

        int prefix = 1;
        for (int i = 0; i < n; i++) {
            res[i] = prefix;
            prefix *= nums[i];
        }

        int suffix = 1;
        for (int i = n - 1; i >= 0; i--) {
            res[i] *= suffix;
            suffix *= nums[i];
        }

        return res;
    }
}
```

10. Time & Space Complexity

Metric	Complexity
Time	O(n)
Space	O(1) extra (output array not counted)

Efficient and meets all constraints.

11. Common Mistakes / Edge Cases

❌ Using division (not allowed)

❌ Forgetting to reset prefix/suffix

❌ Incorrect suffix order (must go backward)

❌ Handling arrays with zeros incorrectly in brute force

Edge case:

- Arrays with one or two zeros → prefix–suffix logic handles naturally.

12. Detailed Dry Run (Step-by-Step Table)

Input:

nums = [1,2,3,4]

🔵 Prefix Scan (left → right)

i	nums[i]	prefix	res[i]
0	1	1	1
1	2	1	1
2	3	2	2
3	4	6	6

After prefix pass:

res = [1,1,2,6]

🔴 Suffix Scan (right → left)

i	nums[i]	suffix	res[i] = res[i] × suffix
3	4	1	6 × 1 = 6
2	3	4	2 × 4 = 8
1	2	12	1 × 12 = 12

0	1	24	1 × 24 = 24

Final result:

[24, 12, 8, 6]

13. Common Use Cases

- Financial calculations (exclude current day)
- Multiplicative prefix analysis
- Array transformation tasks
- Interview questions on prefix/suffix logic

14. Common Traps

- Trying to store prefix/suffix arrays separately
- Using extra space unnecessarily
- Forgetting backwards loop for suffix
- Division-based thinking

15. Builds To (Related Problems)

- **LC 152** — Maximum Product Subarray
- **LC 334** — Increasing Triplet Subsequence
- **LC 560** — Subarray Sum Equals K
- **LC 53** — Kadane’s Algorithm

16. Alternate Approaches + Comparison

Approach	Time	Space	Notes
Brute Force	$O(n^2)$	O(1)	Too slow
Division	O(n)	O(1)	Not allowed
Prefix + Suffix	O(n)	O(1)	✔ Optimal

17. Why This Solution Works (Short Intuition)

Because prefix gives product of all elements before i,
suffix gives product of all elements after i,
and multiplying them gives product of array except self — without division.

18. Variations / Follow-Up Questions

What if division was allowed?
What if array contained extremely large numbers?
What if negative numbers dominate?
How to modify for sum instead of product? (prefix sum)