Topic Introduction

Today, we're diving into a family of problems that challenge your intuition about **products in arrays**. While adding things up is second nature, multiplying across an array brings its own set of surprises—especially when you have to avoid division, keep track of negatives, or answer questions about subarrays on the fly.

Core Concept: Array Product Manipulation

Definition:
Given an array of numbers, array product problems ask you to compute the result of multiplying elements together, often with specific constraints (like skipping elements, maximizing the result, or keeping the product below a threshold).

How it works:
At its simplest, the product of an array is just multiplying every number together. But interviews rarely ask for the obvious! Instead, you'll be challenged to:

- Compute the product for every element except itself
- Find the subarray (contiguous slice) with the largest product
- Count how many subarrays have a product less than a target

Why is this useful?
These problems pop up a lot because they test your ability to handle:

- Edge cases (zeroes, negatives, large ranges)
- Non-commutative operations (order matters!)
- Efficient traversals (often with prefix/suffix arrays, dynamic programming, or sliding windows)

Simple Example (not one of our main problems):
Suppose you have `[2, 3, 4]`. The product is `2 * 3 * 4 = 24`.
But what if you need to skip one index at a time? For index 1: `2 * 4 = 8`.
Now imagine doing this for every element, or for every possible subarray—this is where things get interesting!

Let's look at three real interview problems that stretch this basic idea in different directions:

Product of Array Except Self

LeetCode 238

Maximum Product Subarray

LeetCode 152

Subarray Product Less Than K

LeetCode 713

Why are these grouped together?

All three problems deal with calculating products over arrays in clever ways.

- **Product of Array Except Self**: Compute all products except at the current position, without using division.
- **Maximum Product Subarray**: Find the contiguous slice with the biggest product, handling negatives and zeroes.
- **Subarray Product Less Than K**: Count how many contiguous subarrays have a product less than a given number, using a sliding window.

Let's start with the Product of Array Except Self, which lays the groundwork for the others.

Problem 1: Product of Array Except Self

LeetCode 238

Problem Statement (in plain English):
Given an integer array `nums`, return an array `output` such that `output[i]` is the product of all elements in `nums` except `nums[i]`.
Do not use division, and aim for O(n) time.

Example:
Input: `[1, 2, 3, 4]`
Output: `[24, 12, 8, 6]`
Explanation:

- At index 0: `2 * 3 * 4 = 24`
- At index 1: `1 * 3 * 4 = 12`
- …and so on.

Try this one yourself:
Input: `[2, 5, 7]`
What should the output be?

Brute-force approach:
For each index, multiply every other number (skip `i`). This takes O(n^2) time—not good enough for interviews.

Optimal approach:
We can **avoid division** and process everything in O(n) time by using two passes:

Key insight:

- The product for position `i` = product of all numbers to the left of `i` * product of all numbers to the right of `i`.
- Build two arrays (or reuse the output array to save space):
    - `left[i]`: Product of everything left of `i`
    - `right[i]`: Product of everything right of `i`

Step-by-step logic:

- **First pass (left products):**

For every index, store the product of all numbers before it.

- **Second pass (right products):**

Traverse from the end, multiplying with the running product of all numbers after it.

Python Solution

```
def productExceptSelf(nums):
    n = len(nums)
    output = [1] * n  # Output array, initialized to 1

    # First pass: accumulate products to the left of each index
    left_product = 1
    for i in range(n):
        output[i] = left_product
        left_product *= nums[i]

    # Second pass: accumulate products to the right, multiply in-place
    right_product = 1
    for i in range(n - 1, -1, -1):
        output[i] *= right_product
        right_product *= nums[i]

    return output
```

Time Complexity: O(n)

Space Complexity: O(1) extra (if we ignore output array), otherwise O(n)

Code Walkthrough:

- `output` starts as all 1s.
- Left pass:
  - At each `i`, set `output[i]` to the product of everything before `i`.
- Right pass:
  - At each `i` (from end to start), multiply `output[i]` by the product of everything after `i`.

Trace Example:

Input: `[1, 2, 3, 4]`

- Left pass:
  - output = [1, 1, 2, 6]
- Right pass (starting from the end):
  - output = [24, 12, 8, 6]

Try this input yourself:

`[3, 6, 2, 9]`

Take a moment to solve this on your own before jumping into the solution!

Did you know?

If you're allowed to use division and there are no zeroes, you could just multiply everything and divide by `nums[i]`. But real interviewers want you to avoid that shortcut!

Problem 2: Maximum Product Subarray

LeetCode 152

Problem Statement:

Given an array of integers, find the contiguous subarray (of at least one number) that has the largest product, and return that product.

Example:

Input: `[2, 3, -2, 4]`

Output: `6`

Explanation: The subarray `[2, 3]` has the max product.

Try this input:

`[-2, 0, -1]`

What will your code return?

How is this similar or different?

Like the previous problem, we're dealing with products, but now we need to consider **subarrays** and the impact of negative numbers (since two negatives make a positive). This is a classic dynamic programming pattern.

Brute-force:

Check every possible subarray, multiply its elements, and track the maximum. This is O(n^2) or worse.

Optimal approach:

Key insight:

- Because multiplying by a negative flips the sign, we need to track both the maximum and minimum product ending at each position.
- At each position, the max product is either:
  - The current number
  - The current number times the previous max product
  - The current number times the previous min product

Step-by-step logic:

- Initialize:
  - `max_prod` = nums[0] (best so far)
  - `curr_max`, `curr_min` = nums[0] (max/min ending here)
- For each number from the second onward:

- If the number is negative, swap `curr_max` and `curr_min`
- Update `curr_max` = max(num, curr_max * num)
- Update `curr_min` = min(num, curr_min * num)
- Update `max_prod` if `curr_max` is higher

Pseudocode

```
max_prod = nums[0]
curr_max = nums[0]
curr_min = nums[0]
for i from 1 to end:
    if nums[i] < 0:
        swap(curr_max, curr_min)
    curr_max = max(nums[i], curr_max * nums[i])
    curr_min = min(nums[i], curr_min * nums[i])
    max_prod = max(max_prod, curr_max)
return max_prod
```

Example trace:

Input: `[2, 3, -2, 4]`

- Start: max_prod = curr_max = curr_min = 2
- i=1 (3):
  - curr_max = max(3, 2*3) = 6
  - curr_min = min(3, 2*3) = 3
  - max_prod = 6
- i=2 (-2):
  - swap curr_max and curr_min
  - curr_max = max(-2, 3*-2) = max(-2, -6) = -2
  - curr_min = min(-2, 6*-2) = min(-2, -12) = -12
  - max_prod remains 6
- i=3 (4):
  - curr_max = max(4, -2*4) = 4
  - curr_min = min(4, -12*4) = -48
  - max_prod remains 6

Another test case for you:

Input: `[0, 2, -5, -2, -4, 3]`

Try to work through this step by step.

Time Complexity: O(n)

Space Complexity: O(1)

Problem 3: Subarray Product Less Than K

LeetCode 713

Problem Statement:

Given an array of positive integers `nums` and an integer `k`, return the number of contiguous subarrays where the product of all the elements is less than `k`.

Example:

Input: `nums = [10, 5, 2, 6]`, `k = 100`

Output: `8`

Explanation: The subarrays with product less than 100 are:

[10], [5], [2], [6], [10,5], [5,2], [2,6], [5,2,6]

Try this input:

`nums = [1, 2, 3], k = 0`

What should the output be?

How is this similar or different?

Unlike the previous problems, this one asks *how many* subarrays (not the value), and the array is all positive. The trick is to count all valid subarrays efficiently.

Brute-force:

Check every subarray and count if its product is less than k. This is O(n^2).

Optimal approach:

Key insight:

- Since all numbers are positive, as we extend a window to the right, the product only increases.
- Use a **sliding window**:
  - Start with `left = 0`, `prod = 1`
  - For each `right`, multiply in `nums[right]`
  - While product >= k, move `left` forward (dividing out nums[left])
  - For each position, the number of valid subarrays ending at `right` is `right - left + 1`

Pseudocode

```
if k <= 1: return 0
prod = 1
result = 0
left = 0
for right in 0..n-1:
    prod *= nums[right]
    while prod >= k and left <= right:
        prod //= nums[left]
        left += 1
    result += right - left + 1
```

```
return result
```

Example trace:

Input: `[10, 5, 2, 6]`, `k=100`

- right=0: prod=10 <100, result=1
- right=1: prod=50 <100, result=3 (subarrays: [5], [10,5])
- right=2: prod=100==100, move left, prod=10, result=5
- right=3: prod=60 <100, result=8

Try this test case:

`nums = [2, 3, 4], k = 10`

Time Complexity: O(n)

Space Complexity: O(1)

Summary and Next Steps

We grouped these three problems because they each ask you to think beyond just multiplying all the numbers in an array. Sometimes you need to avoid certain elements, sometimes you handle negatives and zeroes, and sometimes you count how many subarrays meet a product constraint.

Key patterns to remember:

- Prefix/suffix products for skipping elements without division
- Tracking both min and max when negative numbers are involved
- Sliding window for counting subarrays when all numbers are positive

Common mistakes:

- Forgetting to reset or swap min/max when negatives are present
- Not handling zero or one correctly in the sliding window
- Trying to use division where it's not allowed

Action List

- Try implementing all three problems yourself, even if you've seen the code here.
- For Problems 2 and 3, challenge yourself to solve them with a different approach.
- Study solutions for edge cases: zeroes, negatives, single elements.
- Practice explaining your logic out loud or to a friend.
- Review similar problems (like subarray sum variants) to deepen your pattern recognition.

If you get stuck, that's part of the journey! Keep at it, and these product patterns will soon feel like second nature. Happy coding!