The Problem:

"Given an array `nums`, return an array `output` such that `output[i]` is the product of all the elements of `nums` **except** `nums[i]`. You must solve it **without division** and in **O(n)** time."

Q1: What's the tricky part of this problem?

- 🚫 **No division allowed.**
  - Division-based solutions are trivial, but **interviewers expect an optimized non-division solution**.
- ⏳ Time complexity must be **O(n)**.

Q2: Can't I just brute force?

Brute force = loop over `nums`, calculate product of everything except the current element for every index. 😬
Runtime: O(n²) – interviewers will frown.

Q3: What's the clever approach?

Key insight: Split the array's product computation into two **prefix arrays**:

- `prefix[i]`: Product of all elements **to the left** of index `i`.
- `suffix[i]`: Product of all elements **to the right** of index `i`.
  Combine these two arrays to get the result without multiplying `nums[i]`.

Q4: Let's optimize further.

Instead of explicitly storing `prefix` and `suffix`, we can compute **on the fly** to save space.
Space Complexity = O(1) if we exclude the output array.

Optimal Solution: Two Passes with Constant Space

python
Copier le code

```
def product_except_self(nums):
    n = len(nums)
```

```
    output = [1] * n  # Step 1: Initialize output array with 1's

    # Step 2: Calculate prefix products and store in output array
    prefix = 1
    for i in range(n):
        output[i] = prefix
        prefix *= nums[i]

    # Step 3: Calculate suffix products and combine with prefix in
the output array
    suffix = 1
    for i in range(n - 1, -1, -1):
        output[i] *= suffix
        suffix *= nums[i]

    return output

# Example
print(product_except_self([1, 2, 3, 4]))  # Output: [24, 12, 8, 6]
```

Q5: Why two loops work magic?

1. **First pass** → Fills `output[i]` with prefix products up to `i-1`.
2. **Second pass** → Combines suffix product at `i+1` with the prefix already in `output[i]`.

Each index gets `prefix * suffix` directly. Efficient AF. 🚀

Q6: Edge Cases to Know

1. **Contains zeros:**
    - One `0`: Output will be zero everywhere except for the index of the zero.
    - Multiple `0`s: Entire output is zero.

Example:
python
Copier le code

```
nums = [0, 4, 5]
output = [20, 0, 0]
```

2.
3. **Length 1 Array:**
   Corner case for array length = 1 → Invalid per problem constraints.

Key Concepts from This Problem

Arrays

1. **Power of Prefix + Suffix:** Many array problems can be solved by splitting left-side and right-side computations.
2. Arrays are contiguous memory; leveraging index-based pre-calculations makes them fast.

Multiplication Principles

- Combine prefix and suffix products to handle exclusions.
- **Avoid division-based logic** unless constraints allow.

Interview Follow-Ups & Variations

🔄 With Division Allowed

Simple math approach:

python
Copier le code

```python
def product_with_division(nums):
    total_product = 1
    zero_count = 0

    for num in nums:
        if num == 0:
            zero_count += 1
        else:
            total_product *= num

    if zero_count > 1:
        return [0] * len(nums)
    elif zero_count == 1:
        return [0 if num != 0 else total_product for num in nums]
    else:
        return [total_product // num for num in nums]
```

🟡 Product of K Consecutive Numbers

Given `k`, find the product of `k` consecutive numbers in an array:

python
Copier le code

```
def product_k_consecutive(nums, k):
    product = 1
    res = []

    for i in range(len(nums)):
        product *= nums[i]
        if i >= k - 1:
            res.append(product)
            product //= nums[i - k + 1]

    return res
```

Facts About Arrays + Prefix Products

1. **Prefix + Suffix = Superpower:** Split the array logic into two sides for efficiency. #DataStructures
2. **O(n)** without division? Use prefix/suffix + avoid duplication. Trust me, it’s elegant.
3. Calculating things **"except the current index"** is an evergreen trick for arrays. Hashmaps too.
4. Efficient doesn’t always mean using less space—trade-offs exist! #Arrays