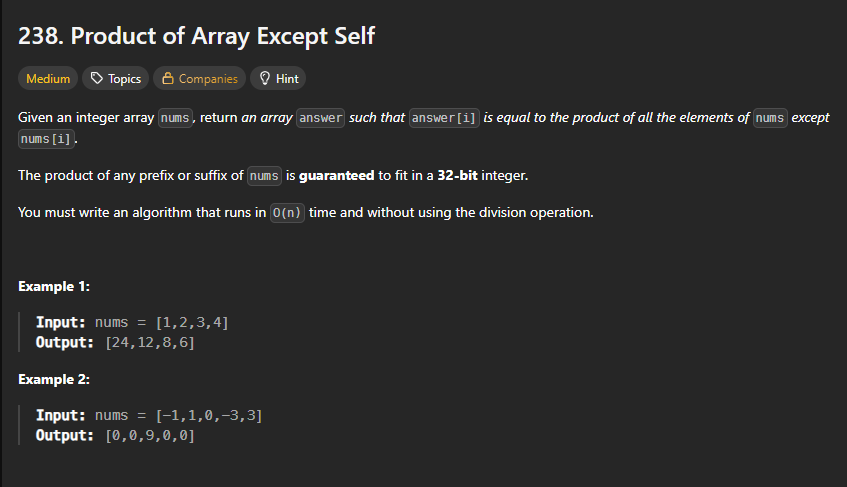
**Leet Code Problem**



**Code O(n)**

class Solution {

public:

    vector<int> productExceptSelf(vector<int>& nums) {

        //      intuition based approach

        int totalProduct = 1;

        for(int i = 0; i < nums.size(); i++){

            totalProduct \*= nums[i];

        }

        vector<int> outputArray[nums.size()];

        for(int i = 0; i < nums.size(); i++){

            outputArray[i] = totalProduct / nums[i];

        }

        return outputArray;

    }

};

**Code O(n^2)**

class Solution {

public:

    vector<int> productExceptSelf(vector<int>& nums) {

        //      intuition based approach

        int totalProduct = 1;

        vector<int> outputArray[nums.size()];

        for(int i = 0; i < nums.size(); i++){

            for(int j=0; j < nums.size(); j++)

            totalProduct \*= nums[i];

        }

        return outputArray;

    }

};

**Optimised Code Approach**

// Time Complexity: O(n)

// Space Complexity: O(1) (excluding output array)

class Solution {

public:

    vector<int> productExceptSelf(vector<int>& nums) {

        // optimised approach

        // step 1 : find prefix product Array

        // step 2 : find suffix product Array

        // step 3 : multiply elements of both array

        // create a vector

        vector<int> outputArray(nums.size());

        //take a variable to store suffix

        int suffix = 1;

        // initialize outputArray 's starting index value as 1'

        outputArray[0] = 1;

        outputArray[nums.size() -1] = 1;

        //run a loop which traverse from starting till end -1 to do prefix calculation.

        for(int leftIdx = 1; leftIdx < nums.size(); leftIdx++ ){

            outputArray[leftIdx] = outputArray[leftIdx -1] \* nums[leftIdx - 1];

        }

        // run a loop traversing from (right -1 ) end to beginiing calculating suffix as well final output as moving progressive.

        for(int rightIdx = nums.size() - 2; rightIdx >= 0; rightIdx--){

            suffix \*= nums[rightIdx + 1];

            outputArray[rightIdx] \*= suffix;

        }

        return outputArray;

    }

};

**KeyLearnings**

You're on the right track with the prefix-suffix approach, Sahitya! But there's a **critical bug** in your code: this line—

vector<int> outputArray[nums.size()];

—is **not valid C++**. You're trying to declare a variable-length array of vectors, which is a Clang extension and not standard C++. What you actually want is a single vector of integers with size nums.size():

vector<int> outputArray(nums.size());

Let me walk you through the corrected version with your logic intact and a few refinements:

**✅ Corrected Code**

class Solution {

public:

vector<int> productExceptSelf(vector<int>& nums) {

int n = nums.size();

vector<int> outputArray(n, 1); // Initialize with 1s

// Step 1: Prefix product

for (int i = 1; i < n; ++i) {

outputArray[i] = outputArray[i - 1] \* nums[i - 1];

}

// Step 2: Suffix product and final result

int suffix = 1;

for (int i = n - 2; i >= 0; --i) {

suffix \*= nums[i + 1];

outputArray[i] \*= suffix;

}

return outputArray;

}

};

**🔍 Key Fixes & Notes**

* vector<int> outputArray(n, 1); creates a vector of size n initialized with 1s—perfect for prefix multiplication.
* Removed manual initialization of outputArray[0] and outputArray[n-1] since the vector is already filled with 1s.
* Variable Suffix should be lowercase (suffix) to match C++ naming conventions and avoid confusion.

If you're documenting this for your DSA repo, you might want to add a comment like:

// Time Complexity: O(n)

// Space Complexity: O(1) (excluding output array)