<https://leetcode.com/problems/product-of-array-except-self/>

Given an integer array nums, return *an array* answer *such that* answer[i] *is equal to the product of all the elements of* nums *except* nums[i].

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

**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]

**Constraints:**

2 <= nums.length <= 10^5

-30 <= nums[i] <= 30

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

**Follow up:** Can you solve the problem in O(1) extra space complexity? (The output array **does not** count as extra space for space complexity analysis.)

**Attempt 1: 2023-02-26**

**Solution 1:  Brute Force (10 min, TLE)**

class Solution {

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

        int[] result = new int[nums.length];

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

            int product = 1;

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

                if(i != j) {

                    product \*= nums[j];

                }

            }

            result[i] = product;

        }

        return result;

    }

}

Time complexity: O(n^2)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/product-of-array-except-self/solutions/1342916/3-minute-read-mimicking-an-interview>

**1. Brute Force**

So, the first and foremost, the simplest method that comes to mind is, I can loop through the complete array using a pointer, say j, for every index i, and multiply all the elements at index j except when i == j. This would ensure that I skip the element at index i from being multiplied.The Time Complexity for this solution would be O(n^2).

Below is the Java Code for this approach.

class Solution {

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

        int n = nums.length;

        int ans[] = new int[n];

        for(int i = 0; i < n; i++) {

            int pro = 1;

            for(int j = 0; j < n; j++) {

                if(i == j) continue;

                pro \*= nums[j];

            }

            ans[i] = pro;

        }

        return ans;

    }

}

To this, the interviewer would surely say to optimize the time complexity of the solution.

Then a bit of acting to think

**Solution 2: Dividing the product of array with the number (10 min, error out because of product = 0)**

class Solution {

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

        int[] result = new int[nums.length];

        int product = 1;

        for(int num : nums) {

            product \*= num;

        }

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

            result[i] = product / nums[i];

        }

        return result;

    }

}

Time complexity: O(n)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/product-of-array-except-self/solutions/1342916/3-minute-read-mimicking-an-interview>

What we would do is, we would find the product of all the numbers of our Array and then divide the product with each element of the array to get the new element for that position in our final answer array. Now after presenting the interviewer with this solution, here is our one more chance to shine out in the interview. **We would specifically tell the interviewer that one major *con* in going with this method is when we have an element as 0 in our array. The problem is that, we can't perform a division by 0, as a result, we won't be able to get corresponding values in our final answer array for the indices having 0 in our initial array at that position.**

The Time Complexity of this approach would be O(n).

Below is the Java Code for the above approach.

class Solution {

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

        int n = nums.length;

        int ans[] = new int[n];

        int pro = 1;

        for(int i : nums) {

            pro \*= i;

        }

        for(int i = 0; i < n; i++) {

            ans[i] = pro / nums[i];

        }

        return ans;

    }

}

To which the interviewer would now say to overcome this problem which we face when having 0.

Again a bit of acting acting to think 🤔 (because that's very important 😜). Then we would say that, 0 is our problem only when we are performing division, with multiplication, we have no such problem with 0, so we would need to think of a way using multiplication.

**Solution 3: Finding Prefix Product and Suffix Product (10 min)**

class Solution {

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

        int len = nums.length;

        int[] prefix\_product = new int[len];

        int[] suffix\_product = new int[len];

        prefix\_product[0] = 1;

        suffix\_product[len - 1] = 1;

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

            prefix\_product[i] = prefix\_product[i - 1] \* nums[i - 1];

        }

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

            suffix\_product[i] = suffix\_product[i + 1] \* nums[i + 1];

        }

        int[] result = new int[len];

        for(int i = 0; i < len; i++) {

            result[i] = prefix\_product[i] \* suffix\_product[i];

        }

        return result;

    }

}

Time complexity: O(n)

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/product-of-array-except-self/solutions/1342916/3-minute-read-mimicking-an-interview>

Similar to finding Prefix Sum Array, here we would intend to find the Prefix Product Array and Suffix Product Array for our original array, i.e.

pre[i] = pre[i - 1] \* a[i - 1] (yes, we multiply with a[i - 1] and not with a[i] on purpose) and similarly suff[i] = suff[i + 1] \* a[i + 1].Now, at any index

i our final answer ans[i] would be given by ans[i] = pre[i] \* suff[i]. Why? Because the pre[i] \* suff[i] contains product of every element before

i and every element after i but not the element at index i (and that is the reason why we excluded a[i] in our prefix and suffix product).

The Time Complexity would be O(n), but we are now using Auxiliary Space of O(n) (excluding the final answer array).

Below is the Java Code for this approach

class Solution {

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

        int n = nums.length;

        int pre[] = new int[n];

        int suff[] = new int[n];

        pre[0] = 1;

        suff[n - 1] = 1;

        for(int i = 1; i < n; i++) {

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

        }

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

            suff[i] = suff[i + 1] \* nums[i + 1];

        }

        int ans[] = new int[n];

        for(int i = 0; i < n; i++) {

            ans[i] = pre[i] \* suff[i];

        }

        return ans;

    }

}

Now the interviewer might ask you to optimizer the space complexity of the program. Again a bit of thinking 🤔.

**Solution 4: Directly store the product of prefix and suffix into the final answer array (30 min)**

class Solution {

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

        int len = nums.length;

        int[] result = new int[len];

        result[0] = 1;

        int prefix\_product = 1;

        int suffix\_product = 1;

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

            prefix\_product = prefix\_product \* nums[i - 1];

            result[i] = prefix\_product;

        }

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

            suffix\_product = suffix\_product \* nums[i + 1];

            result[i] \*= suffix\_product;

        }

        return result;

    }

}

Time complexity: O(n)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/product-of-array-except-self/solutions/65622/simple-java-solution-in-o-n-without-extra-space/comments/67603>

Given numbers [2, 3, 4, 5], regarding the third number 4, the product of array except 4 is 2\*3\*5 which consists of two parts: left 2\*3 and right 5. The product is left\*right. We can get lefts and rights:

Numbers:    2    3    4    5

Lefts:            2  2\*3 2\*3\*4

Rights:  3\*4\*5  4\*5    5

Let’s fill the empty with 1:

Numbers:    2    3    4    5

Lefts:      1    2  2\*3 2\*3\*4

Rights:  3\*4\*5  4\*5    5    1

We can calculate lefts and rights in 2 loops. The time complexity is O(n).

We store lefts in result array. If we allocate a new array for rights. The space complexity is O(n). To make it O(1), we just need to store it in a variable which is right in @lycjava3’s code.

Clear code with comments:

public class Solution {

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

        int n = nums.length;

        int[] res = new int[n];

        // Calculate lefts and store in res.

        int left = 1;

        for (int i = 0; i < n; i++) {

            if (i > 0)

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

            res[i] = left;

        }

        // Calculate rights and the product from the end of the array.

        int right = 1;

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

            if (i < n - 1)

                right = right \* nums[i + 1];

            res[i] \*= right;

        }

        return res;

    }

}

**Refer to**

<https://leetcode.com/problems/product-of-array-except-self/solutions/1342916/3-minute-read-mimicking-an-interview/>

The logic is, we don't actually need separate array to store prefix product and suffix products, we can do all the approach discussed in method 3 directly onto our final answer array.

The Time Complexity would be O(n) but now, the Auxiliary Space is O(1) (excluding the final answer array).

Below is the Java Code for the above algorithm.

class Solution {

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

        int n = nums.length;

        int ans[] = new int[n];

        Arrays.fill(ans, 1);

        int curr = 1;

        for(int i = 0; i < n; i++) {

            ans[i] \*= curr;

            curr \*= nums[i];

        }

        curr = 1;

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

            ans[i] \*= curr;

            curr \*= nums[i];

        }

        return ans;

    }

}

**Refer to**

[L845.Longest Mountain in Array (Ref.L821)](note://WEB927061d55e7df1331b08a8ea60797df9)

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)