Previous Next

238. Product of Array Except Self

May 13, 2019 | 296.9K views

★★★★★
Average Rating: 4.64 (177 votes)

Given an array `nums` of *n* integers where *n* > 1, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Example:

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

Constraint: It's guaranteed that the product of the elements of any prefix or suffix of the array (including the whole array) fits in a 32 bit integer.

Note: Please solve it **without division** and in O(*n*).

Follow up:
Could you solve it with constant space complexity? (The output array **does not** count as extra space for the purpose of space complexity analysis.)

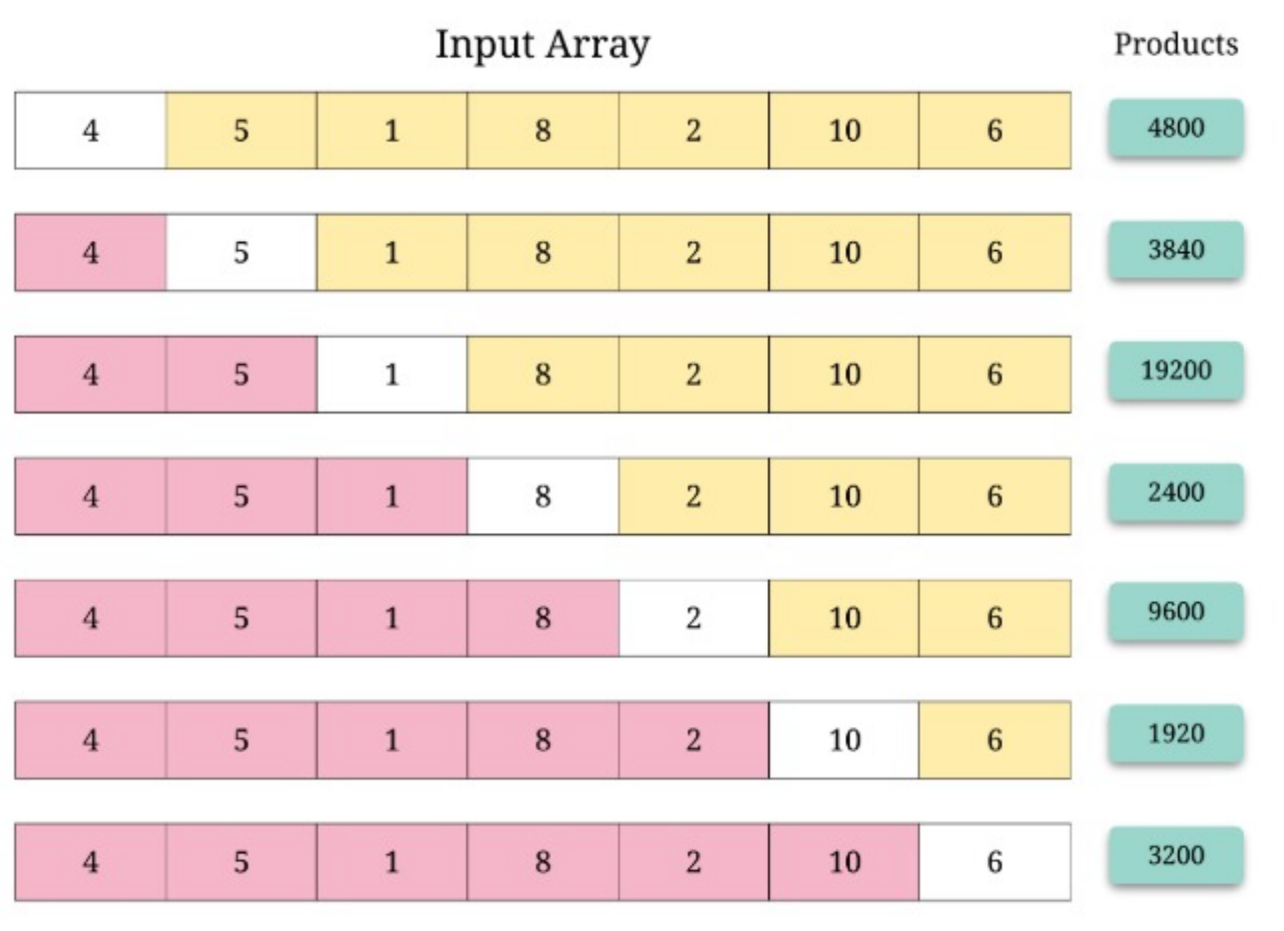
Solution

From the looks of it, this seems like a simple enough problem to solve in linear time and space. We can simply take the product of all the elements in the given array and then, for each of the elements x of the array, we can simply find `product of array except self` value by dividing the product by x.

Doing this for each of the elements would solve the problem. However, there's a note in the problem which says that we are not allowed to use division operation. That makes solving this problem a bit harder.

Approach 1: Left and Right product lists

It's much easier to build an intuition for solving this problem without division once you visualize how the different `products except self` look like for each of the elements. So, let's take a look at an example array and the different products.



Looking at the figure about we can figure another way of computing those different product values.

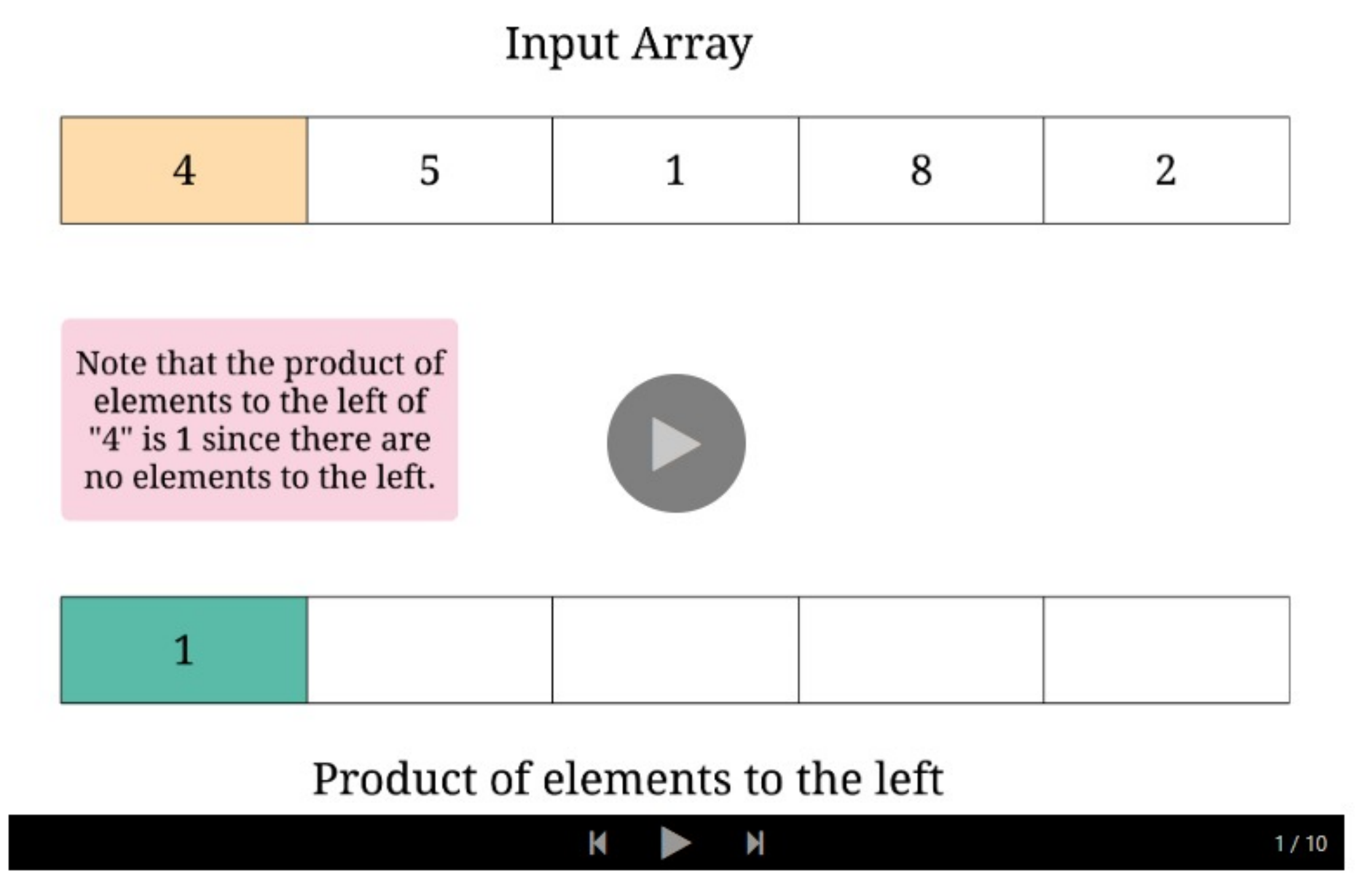
> Instead of dividing the product of all the numbers in the array by the number at a given index to get the corresponding product, we can make use of the product of all the numbers to the left and all the numbers to the right of the index. Multiplying these two individual products would give us the desired result as well.

For every given index, i, we will make use of the product of all the numbers to the left of it and multiply it by the product of all the numbers to the right. This will give us the product of all the numbers except the one at the given index i. Let's look at a formal algorithm describing this idea more concretely.

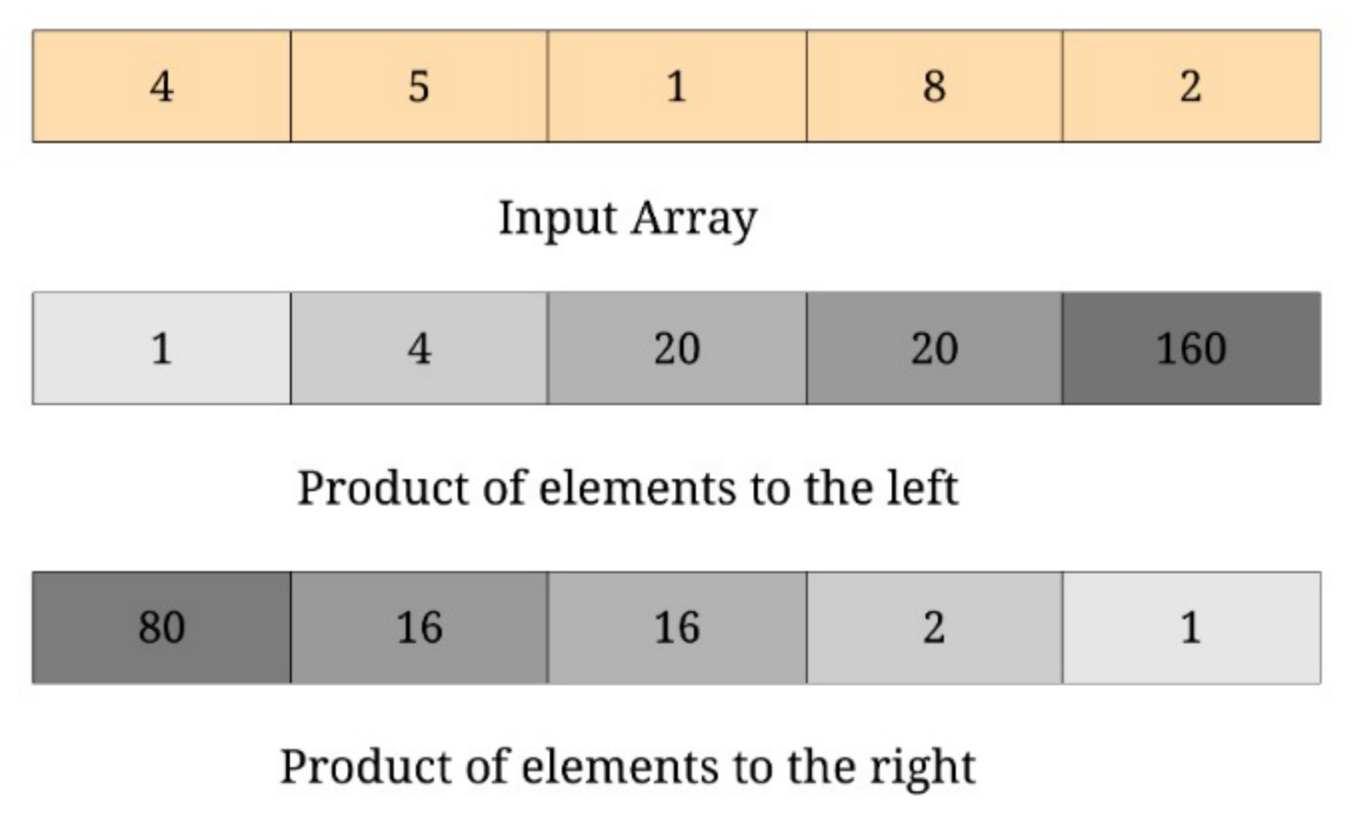
Algorithm

1. Initialize two empty arrays, `L` and `R` where for a given index `i`, `L[i]` would contain the product of all the numbers to the left of `i` and `R[i]` would contain the product of all the numbers to the right of `i`.
2. We would need two different loops to fill in values for the two arrays. For the array `L`, $L[0]$ would be `1` since there are no elements to the left of the first element. For the rest of the elements, we simply use $L[i] = L[i-1] * nums[i-1]$. Remember that `L[i]` represents product of all the elements *to the left of element at index i*.
3. For the other array, we do the same thing but in reverse i.e. we start with the initial value of `1` in $R[length-1]$ where $length$ is the number of elements in the array, and keep updating `R[i]` in reverse. Essentially, $R[i] = R[i+1] * nums[i+1]$. Remember that `R[i]` represents product of all the elements *to the right of element at index i*.
4. Once we have the two arrays set up properly, we simply iterate over the input array one element at a time, and for each element at index `i`, we find the `product except self` as $L[i] * R[i]$.

Let's go over a simple run of the algorithm that clearly depicts the construction of the two intermediate arrays and finally the answer array.



For the given array $[4, 5, 1, 8, 2]$, the `L` and `R` arrays would finally be:



Java Python Copy

```python
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:

        # The length of the input array
        length = len(nums)

        # The left and right arrays as described in the algorithm
        L, R, answer = [0]*length, [0]*length, [0]*length

        # L[i] contains the product of all the elements to the left
        # Note: for the element at index '0', there are no elements to the left,
        # so the L[0] would be 1
        L[0] = 1
        for i in range(1, length):

            # L[i - 1] already contains the product of elements to the left of 'i - 1'
            # Simply multiplying it with nums[i - 1] would give the product of all
            # elements to the left of index 'i'
            L[i] = nums[i - 1] * L[i - 1]

        # R[i] contains the product of all the elements to the right
        # Note: for the element at index 'length - 1', there are no elements to the right,
        # so the R[length - 1] would be 1
        R[length - 1] = 1
        for i in reversed(range(length - 1)):

            # R[i + 1] already contains the product of elements to the right of 'i + 1'
```

Complexity analysis

- Time complexity : $O(N)$ where N represents the number of elements in the input array. We use one iteration to construct the array L, one to construct the array R and one last to construct the $answer$ array using L and R.
- Space complexity : $O(N)$ used up by the two intermediate arrays that we constructed to keep track of product of elements to the left and right.

Approach 2: O(1) space approach

Although the above solution is good enough to solve the problem since we are not using division anymore, there's a follow-up component as well which asks us to solve this using constant space. Understandably so, the output array *does not* count towards the space complexity. This approach is essentially an extension of the approach above. Basically, we will be using the output array as one of `L` or `R` and we will be constructing the other one on the fly. Let's look at the algorithm based on this idea.

Algorithm

1. Initialize the empty `answer` array where for a given index `i`, `answer[i]` would contain the product of all the numbers to the left of `i`.
2. We construct the `answer` array the same way we constructed the `L` array in the previous approach. These two algorithms are exactly the same except that we are trying to save up on space.
3. The only change in this approach is that we don't explicitly build the `R` array from before. Instead, we simply use a variable to keep track of the running product of elements to the right and we keep updating the `answer` array by doing $answer[i] = answer[i] * R$. For a given index `i`, `answer[i]` contains the product of all the elements to the left and `R` would contain product of all the elements to the right. We then update `R` as $R = R * nums[i]$

Java Python Copy

```python
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:

        # The length of the input array
        length = len(nums)

        # The answer array to be returned
        answer = [0]*length

        # answer[i] contains the product of all the elements to the left
        # Note: for the element at index '0', there are no elements to the left,
        # so the answer[0] would be 1
        answer[0] = 1
        for i in range(1, length):

            # answer[i - 1] already contains the product of elements to the left of 'i - 1'
            # Simply multiplying it with nums[i - 1] would give the product of all
            # elements to the left of index 'i'
            answer[i] = nums[i - 1] * answer[i - 1]

        # R contains the product of all the elements to the right
        # Note: for the element at index 'length - 1', there are no elements to the right,
        # so the R would be 1
        R = 1;
        for i in reversed(range(length)):

            # For the index 'i', R would contain the
```

Complexity analysis

- Time complexity : $O(N)$ where N represents the number of elements in the input array. We use one iteration to construct the array L, one to update the array $answer$.
- Space complexity : $O(1)$ since don't use any additional array for our computations. The problem statement mentions that using the $answer$ array doesn't add to the space complexity.

Rate this article: ★★★★★

Previous Next

Comments: 69 Sort By

Type comment here... (Markdown is supported)

Preview Post

spooja__ ★ 388 October 26, 2019 12:03 PM

How in the world can I ever think of this in an interview

358 Share Reply

SHOW 26 REPLIES

anubhakushwaha ★ 150 May 16, 2019 6:50 PM

We can utilize properties of log, if input array is [a, b, c, d] then
precalculated_val = log(a* b* c* d) = log(a) + log(b) + log(c) + log(d)
So for output array it would be { antilog [precalculated_val - log(arr[i])] }

143 Share Reply

SHOW 9 REPLIES

mike_zamini ★ 80 June 17, 2019 5:53 AM

Don't you think that the second approach is NOT in O(1) space complexity? We are still using ONE intermediate array (answer) to make the L array. So it is O(N), right?

78 Share Reply

SHOW 8 REPLIES

stalasila ★ 23 May 17, 2019 12:26 PM

Good One !!!

23 Share Reply

mhelvens ★ 688 May 16, 2019 3:05 PM

There's more to the simple solution using division than this article suggests. I challenge people to try to implement that solution. I predict that most will not get it right on their first try.
(If you'd like to try, refrain from opening / reading the replies to this comment.)

23 Share Reply

SHOW 7 REPLIES

stanislaw89 ★ 27 February 3, 2020 4:58 AM

The description should contain a note that there are not 0 in the array, otherwise, the solution above won't work.

11 Share Reply

SHOW 1 REPLY

deleted_user ★ 198 September 12, 2019 1:27 AM

This is not a constant space solution. Constant space would imply using only variables instead of an array.

9 Share Reply

SHOW 1 REPLY

al-abbasi ★ 12 July 26, 2019 8:13 PM

It's slightly quicker if you don't use reversed (faster than 99% of submissions)

```
class Solution(object):
def productExceptSelf(self, nums):
```

Read More

9 Share Reply

SHOW 1 REPLY

Pkoiralap ★ 13 July 11, 2019 1:39 PM

I did it this way:

1. If there is a single zero in the list, every other elements will be zero expect for that
2. If there are more than one zero in the list, every element will be zero

Read More

13 Share Reply

SHOW 2 REPLIES

Hogwarts ★ 16 August 22, 2019 9:20 AM

Is the space complexity of the second approach O(1)? I think it's O(n)

9 Share Reply

SHOW 1 REPLY

‹ 1 2 3 4 5 6 7 ›