Problem 238: Product of Array Except Self

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

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 input is generated such that

answer[i]

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.)

Code Snippets

C++:

```
class Solution {
public:
```

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

}
};
```

Java:

```
class Solution {
public int[] productExceptSelf(int[] nums) {

}
}
```

Python3:

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

Python:

```
class Solution(object):
def productExceptSelf(self, nums):
"""
:type nums: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @return {number[]}
*/
var productExceptSelf = function(nums) {

};
```

TypeScript:

```
function productExceptSelf(nums: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] ProductExceptSelf(int[] nums) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* productExceptSelf(int* nums, int numsSize, int* returnSize) {

}
```

Go:

```
func productExceptSelf(nums []int) []int {

}
```

Kotlin:

```
class Solution {
fun productExceptSelf(nums: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func productExceptSelf(_ nums: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn product_except_self(nums: Vec<i32>) -> Vec<i32> {
}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {Integer[]}
def product_except_self(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function productExceptSelf($nums) {

}
}
```

Dart:

```
class Solution {
List<int> productExceptSelf(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def productExceptSelf(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec product_except_self(nums :: [integer]) :: [integer]
def product_except_self(nums) do

end
end
```

Erlang:

```
-spec product_except_self(Nums :: [integer()]) -> [integer()].
product_except_self(Nums) ->
.
```

Racket:

```
(define/contract (product-except-self nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Product of Array Except Self
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<int> productExceptSelf(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Product of Array Except Self
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Product of Array Except Self
Difficulty: Medium
Tags: array

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def productExceptSelf(self, nums: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def productExceptSelf(self, nums):
"""
:type nums: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Product of Array Except Self
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} nums
* @return {number[]}
*/
var productExceptSelf = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Product of Array Except Self
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function productExceptSelf(nums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Product of Array Except Self
* Difficulty: Medium
* Tags: array
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int[] ProductExceptSelf(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Product of Array Except Self
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* productExceptSelf(int* nums, int numsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Product of Array Except Self
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func productExceptSelf(nums []int) []int {
}
```

Kotlin Solution:

```
class Solution {
fun productExceptSelf(nums: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func productExceptSelf(_ nums: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Product of Array Except Self
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn product_except_self(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {Integer[]}
def product_except_self(nums)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function productExceptSelf($nums) {

}
}
```

Dart Solution:

```
class Solution {
List<int> productExceptSelf(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def productExceptSelf(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec product_except_self(nums :: [integer]) :: [integer]
def product_except_self(nums) do

end
end
```

Erlang Solution:

```
-spec product_except_self(Nums :: [integer()]) -> [integer()].
product_except_self(Nums) ->
.
```

Racket Solution:

```
(define/contract (product-except-self nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```