You have an array of integers, and for each index you want to find the product of *every integer except the integer at that index.*

Write a function `getProductsOfAllIntsExceptAtIndex()` that takes an array of integers and returns an array of the products.

For example, given:

```
[1, 7, 3, 4]
```

your function would return:

```
[84, 12, 28, 21]
```

by calculating:

```
[7 * 3 * 4,  1 * 3 * 4,  1 * 7 * 4,  1 * 7 * 3]
```

Here's the catch: **You can't use division in your solution!**

Gotchas

Does your function work if the input array contains zeroes? Remember—no division.

We can do this in $O(n)$ time and $O(n)$ space!

We only need to allocate *one* new array of size n.

Breakdown

A brute force approach would use two loops to multiply the integer at every `index` by the integer at every `nestedIndex`, unless `index == nestedIndex`.

This would give us a runtime of $O(n^2)$. Can we do better?

Well, we're wasting a lot of time **doing the same calculations.** As an example, let's take:

```
// input array
[1, 2, 6, 5, 9]

// array of the products of all integers
// except the integer at each index:
[540, 270, 90, 108, 60]  // [2 * 6 * 5 * 9,  1 * 6 * 5 * 9,  1 * 2 * 5 * 9,  1 * 2 * 6 * 9,
```

We're doing some of the same multiplications *two or three times*!

`[2*6*5*9, 1*6*5*9, 1*2*5*9, 1*2*6*9, 1*2*6*5]`

Or look at this pattern:

`[2*6*5*9, 1*6*5*9, 1*2*5*9, 1*2*6*9, 1*2*6*5]`

We're redoing multiplications when instead we could be storing the results! This would be a great time to use a greedy approach. We could store the results of each multiplication highlighted in blue, then just multiply by *one new* integer each time.

So in the last highlighted multiplication, for example, we wouldn't have to multiply $1 * 2 * 6$ again. If we stored that value (12) from the previous multiplication, we could just multiply $12 * 5$.

Can we break our problem down into subproblems so we can use a greedy approach?

Let's look back at the last example:

`[2*6*5*9, 1*6*5*9, 1*2*5*9, 1*2*6*9, 1*2*6*5]`

What do all the highlighted multiplications have in common?

They are all the integers that are *before each index* in the input array ($[1, 2, 6, 5, 9]$). For example, the highlighted multiplication at index 3 ($1 * 2 * 6$) is all the integers before index 3 in the input array.

```
[1, 2, 6, 5, 9]

[2*6*5*9, 1*6*5*9, 1*2*5*9, 1*2*6*9, 1*2*6*5]
```

Do all the multiplications that *aren't highlighted* have anything in common?

Yes, they're all the integers that are *after* each index in the input array!

Knowing this, can we break down our original problem to use a greedy approach?

The product of *all* the integers except the integer at each index can be broken down into two pieces:

1. the product of all the integers *before* each index, and
2. the product of all the integers *after* each index.

To start, let's just get the product of all the integers **before each index**.

How can we do this? Let's take another example:

```
// input array
[3, 1, 2, 5, 6, 4]

// multiplication of all integers before each index
// (we give index 0 a value of 1 since it has no integers before it)
[1, 3,  3 * 1,  3 * 1 * 2,  3 * 1 * 2 * 5,  3 * 1 * 2 * 5 * 6]

// final array of the products of all the integers before each index
[1, 3, 3, 6, 30, 180]
```

Notice that we're always adding *one* new integer to our multiplication for each index!

So to get the products of all the integers before each index, we could greedily store each product *so far* and multiply that by the *next* integer. Then we can store that *new* product so far and keep going.

So how can we apply this to our input array?

Let's make an array `productsOfAllIntsBeforeIndex`:

```
int productsOfAllIntsBeforeIndex[arrayLength];
int productSoFar;
size_t i;

// for each integer, find the product of all the integers
// before it, storing the total product so far each time
productSoFar = 1;
for (i = 0; i < arrayLength; i++) {
    productsOfAllIntsBeforeIndex[i] = productSoFar;
    productSoFar *= intArray[i];
}
```

So we solved the subproblem of finding the products of all the integers *before* each index. **Now, how can we find the products of all the integers *after* each index?**

It might be tempting to make a new array of all the values in our input array in **reverse**, and just use the same function we used to find the products before each index.

Is this the best way?

This method will work, but:

1. **We'll need to make a whole new array** that's basically the same as our input array. That's another $O(n)$ memory cost!
2. To keep our indices aligned with the original input array, **we'd have to reverse the array of products we return**. That's two reversals, or two $O(n)$ operations!

Is there a cleaner way to get the products of all the integers after each index?

We can just *walk through our array backwards*! So instead of reversing the values of the array, we'll just reverse the *indices* we use to iterate!

```
int productsOfAllIntsAfterIndex[arrayLength];
size_t i;
int productSoFar = 1;

for (i = arrayLength; i > 0; i--) {
    productsOfAllIntsAfterIndex[i - 1] = productSoFar;
    productSoFar *= intArray[i - 1];
}
```

Now we've got `productsOfAllIntsAfterIndex`, but we're starting to build a lot of new arrays. And we still need our final array of the *total* products. **How can we save space?**

Let's take a step back. Right now we'll need three arrays:

1. `productsOfAllIntsBeforeIndex`
2. `productsOfAllIntsAfterIndex`
3. `productsOfAllIntsExceptAtIndex`

To get the first one, we keep track of the total product so far going *forwards*, and to get the second one, we keep track of the total product so far going *backwards*. How do we get the third one?

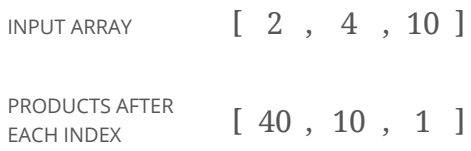
Well, we want the product of all the integers *before* an index **and** the product of all the integers *after* an index. We just need to multiply every integer in `productsOfAllIntsBeforeIndex` with the integer *at the same index* in `productsOfAllIntsAfterIndex`!

Let's take an example. Say our input array is $[2, 4, 10]$:

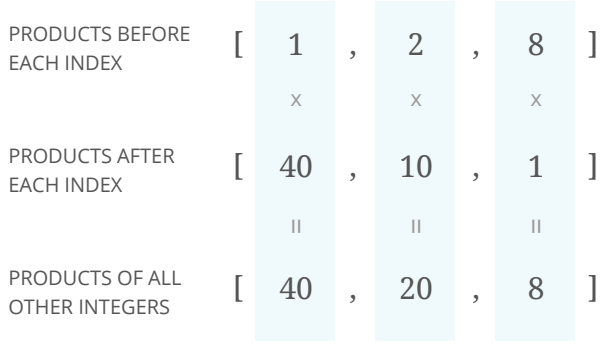
We'll calculate `productsOfAllIntsBeforeIndex` as:

INPUT ARRAY	[2 , 4 , 10]
PRODUCTS BEFORE EACH INDEX	[1 , 2 , 8]

And we'll calculate `productsOfAllIntsAfterIndex` as:



If we take these arrays and multiply the integers at the same indices, we get:



And this gives us what we're looking for—the products of all the integers except the integer at each index.

Knowing this, can we eliminate any of the arrays to reduce the memory we use?

Yes, instead of building the second array `productsOfAllIntsAfterIndex`, we could take the product we would have stored and just multiply it by the matching integer in `productsOfAllIntsBeforeIndex`!

So in our example above, when we calculated our first (well, "0th") "product after index" (which is 40), we'd just multiply that by our first "product before index" (1) instead of storing it in a new array.

How many arrays do we need now?

Just one! We create an array, populate it with the products of all the integers *before* each index, and then multiply those products with the products *after* each index to get our final result!

`productsOfAllIntsBeforeIndex` now contains the products of all the integers *before and after* every index, so we can call it `productsOfAllIntsExceptAtIndex`!

Almost done! Are there any edge cases we should test?

What if the input array contains zeroes? What if the input array only has *one* integer?

We'll be fine with zeroes.

But what if the input array has fewer than two integers?

Well, there won't be any products to return because at any index there are no “other” integers. So let's abort.

Solution

To find the products of all the integers except the integer at each index, we'll go through our array greedily *twice*. First we get the products of all the integers **before** each index, and then we go *backwards* to get the products of all the integers **after** each index.

When we multiply all the products before and after each index, we get our answer—the products of all the integers except the integer at each index!

```
C
void getProductsOfAllIntsExceptAtIndex(
    const int *intArray,
    size_t intArrayLength,
    int *productsOfAllIntsExceptAtIndexOutput,
    size_t productsOfAllIntsExceptAtIndexOutputLength)
{
    int productSoFar;
    size_t i;

    // getting the product of numbers at other indices requires at least 2 numbers
    assert(intArrayLength >= 2);

    // make sure output is large enough
    assert(productsOfAllIntsExceptAtIndexOutputLength >= intArrayLength);

    // for each integer, we find the product of all the integers
    // before it, storing the total product so far each time
    productSoFar = 1;
    for (i = 0; i < intArrayLength; i++) {
        productsOfAllIntsExceptAtIndexOutput[i] = productSoFar;
        productSoFar *= intArray[i];
    }

    // for each integer, we find the product of all the integers
    // after it. since each index in products already has the
    // product of all the integers before it, now we're storing
    // the total product of all other integers
    productSoFar = 1;
    for (i = intArrayLength; i > 0; i--) {
        productsOfAllIntsExceptAtIndexOutput[i - 1] *= productSoFar;
        productSoFar *= intArray[i - 1];
    }
}
```

Complexity

$O(n)$ time and $O(n)$ space. We make two passes through our input an array, and the array we build always has the same length as the input array.

Bonus

What if you *could* use division? Careful—watch out for zeroes!

What We Learned

Another question using a greedy approach. The tricky thing about this one: we couldn't actually solve it in *one* pass. But we could solve it in *two* passes!

This approach probably wouldn't have been obvious if we had started off trying to use a greedy approach.

Instead, we started off by coming up with a slow (but correct) brute force solution and trying to improve from there. We looked at *what our solution actually calculated*, step by step, and found some *repeat work*. Our final answer came from brainstorming ways to avoid doing that repeat work.

So that's a pattern that can be applied to other problems:

Start with a brute force solution, look for *repeat work* in that solution, and modify it to only do that work once.

← course home (/table-of-contents)

Next up: Cafe Order Checker ➡ (/question/cafe-order-checker?course=fc1&section=greedy)