9 The maximum-product subarray

Given an array of integers, return the product of the contiguous subarray having the largest product.

Example 1: for `[2, 3, 4]`, the product is 24 for the subarray `[2, 3, 4]`.

Example 2: for `[-2, 3, 4]`, the product is 12 for the subarray `[3, 4]`.

Example 3: for `[-2, 3, -4]`, the product is 24 for the subarray `[-2, 3, -4]`.

Clarification questions

Q: What result should be returned for an empty array?
A: The result should be 0.

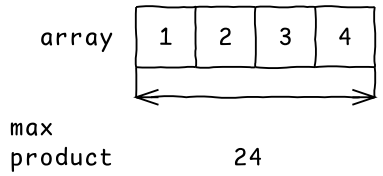
Q: Can some elements be zero?
A: Yes.

Q: What is the maximum length of the input?
A: 10,000 elements.

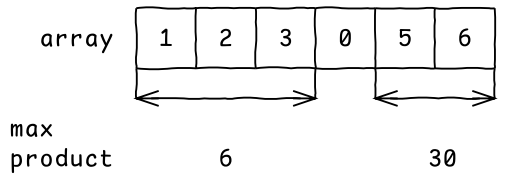
Solution 1: greedy, two-pass, $O(n)$ time

Before we try to propose an algorithm, let's consider some examples to identify general cases depending on the sign of the elements in the array.

Firstly, if the array contains only positive numbers, the subarray with the maximum product is the whole array. This is since the numbers are integers, so positive numbers are ≥ 1:

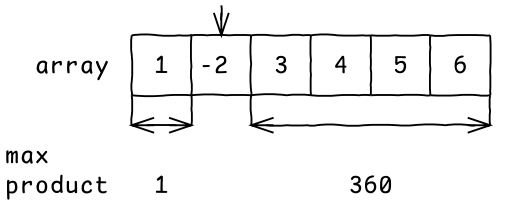


Secondly, we need to handle elements that are 0. Including any such element will cause the product to drop to zero, so we should avoid them. Consider an example:



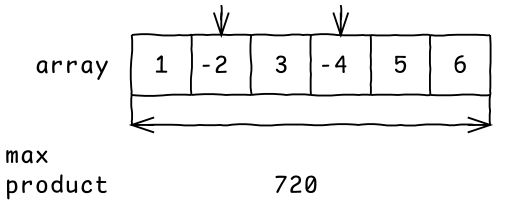
We could say that the 0 acts like a separator: the array can be split into subarrays at any point where it contains a zero. We can search for the solution in the resulting subarrays independently, after which we choose the one having the maximum product as the final solution of the problem.

Finally, we have to handle negative elements. Such elements complicate the search, since they can flip the sign of the product. Let's consider a few examples to understand this effect.



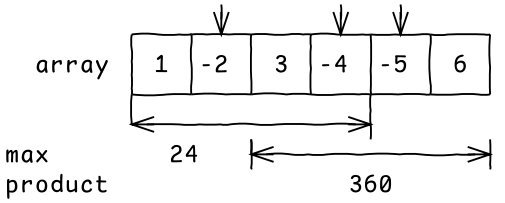
Here we see that we cannot include number -2 in the subarray, since the product will be negative. We have to choose either the left subarray of positive numbers having product 1, or the right subarray with positive numbers having product 360. The solution is the right subarray, since its product is larger.

Let's add a second negative number to see what happens:



With 2 negative numbers the logic changes: the two negative signs cancel each other out, so we can safely use both in the subarray, to obtain a product of 720.

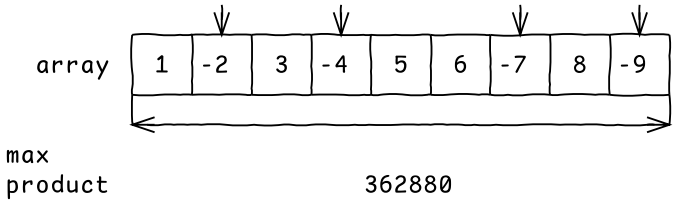
Let's add a third negative number:



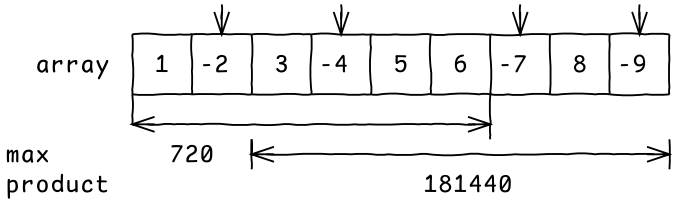
In this case, we can extend the subarray to include two of the negative numbers, so that the signs cancel out. The third negative number cannot be used, since the product would be negative.

We have two choices: either choose the left subarray, ending at -4, or the right subarray, starting after -2. We choose the right one since it has the largest product: 360.

In general, if we have an even number of negative numbers, we can choose the entire array (assuming that it does not contain any zeroes):



If we have an odd number `k` of negative numbers, we choose either the left subarray or the right subarray that contains `k-1` negatives (note that `k-1` is even, so the product is positive):



To summarize what we have learned so far:

- Every time we encounter a zero, we have to start a new search;
- If the number of negative numbers is even, we use all elements;
- If the number of negative numbers is odd, we use the biggest of the left or the right subarray that contains an even number of negatives and is maximal.

Let's sketch the implementation:

```
def find_max_prod_subarray(array):
    def find_non_zero_chunks(array):
        # TODO

    def find_max_prod_in_chunk(chunk):
        # TODO

    max_product = 0
    for chunk in find_non_zero_chunks(array):
        max_product = max(max_product, find_max_prod_in_chunk(chunk))
    return max_product
```

We first extract the chunks of the array that do not contain zeroes. Then we search each chunk for the maximum product subarray. We return the overall maximum.

Let's implement the helper function `find_non_zero_chunks`:

```
def find_non_zero_chunks(array):
    chunk = []
    for value in array:
        if value:
            chunk.append(value)
```

```
        elif chunk:
            yield chunk
            chunk = []
    if chunk:
        yield chunk
```

We iterate over the values in the array, appending non-zero values to the current chunk. Each time we find a zero, we yield the current chunk and empty it. We are careful to return the last chunk as well, in case the array does not end with 0 so there are leftover values at the end.

The code can be simplified slightly by removing the need for the final check, by padding the array with a 0 at the end:

```
def find_non_zero_chunks(array):
    chunk = []
    for value in array + [0]:
        if value:
            chunk.append(value)
        elif chunk:
            yield chunk
            chunk = []
```

Let's now implement `find_max_prod_in_chunk`:

```
def find_max_prod_forward_pass(chunk):
    max_product = 0
    product = 1
    for value in chunk:
        product *= value
        max_product = max(max_product, product)
    return max_product

def find_max_prod_in_chunk(chunk):
    return max(find_max_prod_forward_pass(chunk),
               find_max_prod_forward_pass(chunk[::-1]))
```

To handle the case where there are `k` negative numbers, with `k` odd, so that we need to keep only `k-1` negatives, we must make two passes over the chunk: one in which we use the first `k-1` negatives; and another one in which we use the last `k-1` negatives. The two passes are symmetrical, but the logic is otherwise identical.

We take advantage of the symmetry to implement the logic in a single function: `find_max_prod_forward_pass`. This computes the maximum product by taking the maximum of the running product as we iterate over the array from left to right. In case of `k` negative numbers with `k` odd, it will use the first `k-1` negatives.

To handle the case of choosing the last `k-1` negatives in the chunk, we call the helper function again with the chunk reversed.

Thus we make two passes over the chunk, one in the forward direction by passing `chunk`, and the other in the reverse direction by passing `chunk[::-1]` (chunk[::-1] is the chunk reversed).

The way this works can be seen in the following example:

array	1	-2	3	-4	-5	6
product	1	-2	-6	24	-120	-720
max	1	1	1	24	24	24

Notice how keeping track of the maximum product allows us to avoid negative products.

The code is correct also when the number of negatives is even:

array	1	-2	3	-4	-5	-6
product	1	-2	-6	24	-120	720
max	1	1	1	24	24	720

Here is the full implementation:

```
def find_max_prod_subarray(array):
    def find_non_zero_chunks(array):
        chunk = []
        for value in array + [0]:
            if value:
                chunk.append(value)
            elif chunk:
                yield chunk
                chunk = []

    def find_max_prod_forward_pass(chunk):
        max_product = 0
        product = 1
        for value in chunk:
            product *= value
            max_product = max(max_product, product)
        return max_product

    def find_max_prod_in_chunk(chunk):
        return max(find_max_prod_forward_pass(chunk),
                   find_max_prod_forward_pass(chunk[::-1]))
```

```
    max_product = 0
    for chunk in find_non_zero_chunks(array):
        max_product = max(max_product, find_max_prod_in_chunk(chunk))
    return max_product
```

This code can be shortened by removing the split into non-zero chunks, and adding a zero product check in the forward pass:

```
def find_max_prod_subarray(array):
    def find_max_prod_forward_pass(array):
        max_product = 0
        product = 1
        for value in array:
            if not product:
                # Reset the search any time the running product is zero.
                product = value
            else:
                product *= value
            max_product = max(max_product, product)
        return max_product
    return max(find_max_prod_forward_pass(array),
               find_max_prod_forward_pass(array[::-1]))
```

Let's see the forward pass in action on an example:

array	1	2	3	0	-5	-6
product	1	2	6	0	-5	30
max	1	2	6	6	6	30

The new implementation is shorter than before and uses less memory. The disadvantage is that the line `max_product = max(max_product, product)` now has dual purpose:

- It keeps track of the maximum across non-zero chunks;
- It keeps track of the maximum when the running chunk product goes below zero.

This is a tricky point that can make future changes to the code more difficult, so it can be seen as a disadvantage. The longer code may be preferrable, even though it is somewhat slower.

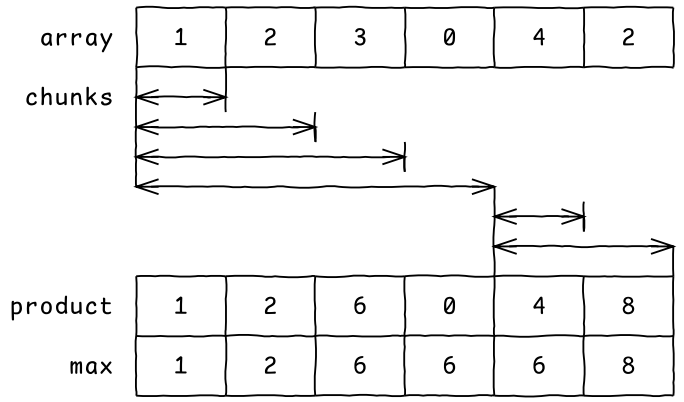
> **Compact and clever code vs. longer but simpler**
>
> Making the code compact and clever may come at the expense of readability. Competitive programming resources are often biased towards short and clever implementations. In a software engineering position, these are usually frowned upon, since the code may

become hard to understand and maintain. Avoid writing the code too cryptic/clever during the interview, since it may be seen as a negative point.

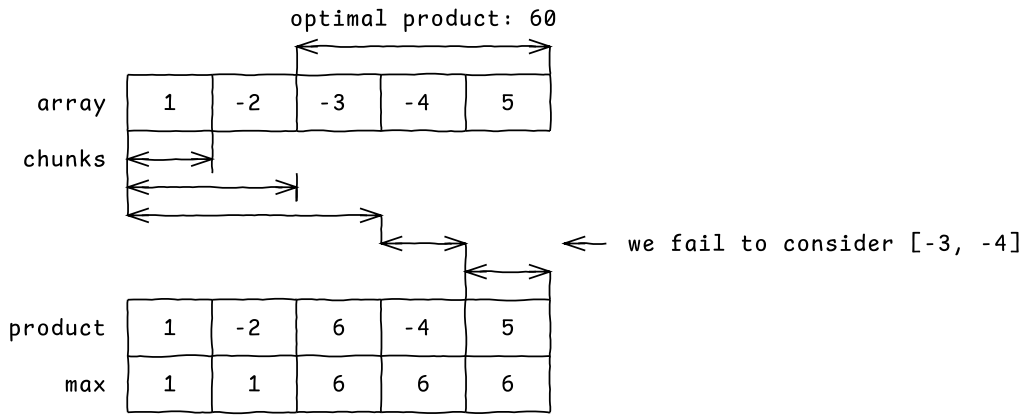
Solution 2: dynamic programming, one-pass, $O(n)$ time

We can convert the greedy solution into a one-pass dynamic programming algorithm using an approach similar to the solution for the maximum-sum subarray: for each index `i`, we compute the maximum product that can be obtained by a subarray ending at index `i`. To do that, we consider two choices: either extending the subarray ending at index `i-1`, or starting a new subarray consisting only of the element at index `i`.

This approach works well for arrays containing non-negative elements, including zeroes:



However it fails when there are negative elements:



This is because we did not handle a case: when the element at index `i` is negative, we can also extend an array ending at `i-1` that has negative product, to form an array with a positive product. To maximize the product, we must consider the array ending at `i-1` that has the *minimum* possible *negative* product.

Therefore we need to keep track of not just the maximum-product subarray ending at index `i`, but also of the minimum-negative-product subarray.

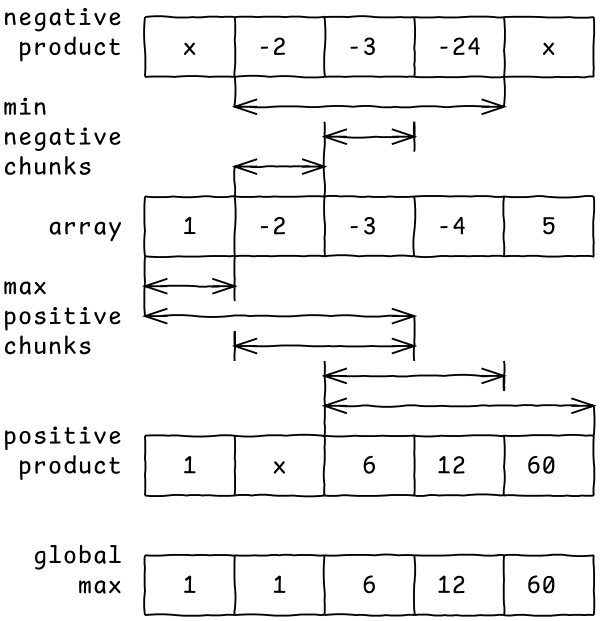
The maximum-product subarray ending at index `i` is computed as the best option between the following choices:

- Starting a new subarray containing only the element at index `i`;
- Extending the maximum-product subarray ending at index `i - 1`, if it has a positive value and the current value is also positive;
- Extending the minimum-product subarray ending at index `i - 1`, if it has a negative value and the current value is also negative;
- If not possible to obtain a product > 0, we set it to an invalid value, such as $-\infty$.

The minimum-negative-product subarray ending at index `i` is computed as the best option between the following choices:

- Starting a new subarray containing only the element at index `i`;
- Extending the minimum-product subarray ending at index `i - 1`, if it has a negative value and the current value is positive;
- Extending the maximum-product subarray ending at index `i - 1`, if it has a positive value and the current value is negative;
- If not possible to obtain a product < 0, we set it to an invalid value, such as ∞.

Here is an example of the strategy in action:



This can be implemented as:

```
def find_max_prod_subarray(array):
    max_chunk = []
```

```
    min_chunk = []
    global_max = 0
    for value in array:
        # Compute candidates for the max-product subarray
        # ending with the current element:
        # Candidate 1: start new subarray
        max_new_chunk = value
        # Candidate 2: continue positive subarray
        if max_chunk and max_chunk[-1] > 0 and value > 0:
            max_continue_positive = max_chunk[-1] * value
        else:
            max_continue_positive = -float('inf')
        # Candidate 3: flip sign of negative subarray
        if min_chunk and min_chunk[-1] < 0 and value < 0:
            max_flip_negative = min_chunk[-1] * value
        else:
            max_flip_negative = -float('inf')

        # Compute candidates for the min-product subarray
        # ending with the current element:
        # Candidate 1: start new subarray
        min_new_chunk = value
        # Candidate 2: continue negative subarray
        if min_chunk and min_chunk[-1] < 0 and value > 0:
            min_continue_negative = min_chunk[-1] * value
        else:
            min_continue_negative = float('inf')
        # Candidate 3: flip sign of positive subarray
        if max_chunk and max_chunk[-1] > 0 and value < 0:
            min_flip_positive = max_chunk[-1] * value
        else:
            min_flip_positive = float('inf')

        # Choose the best candidate
        max_chunk.append(max(max_new_chunk,
                             max_continue_positive,
                             max_flip_negative))
        min_chunk.append(min(min_new_chunk,
                             min_continue_negative,
                             min_flip_positive))
        global_max = max(global_max, max_chunk[-1])
    return global_max
```

Notice how at each step we first compute all the new candidates, then update the `max_chunk` and the `min_chunk` arrays. We cannot update `max_chunk` before computing the candidates

for `min_chunk`, as we would compute the wrong value.

The time complexity is linear, and the space complexity is linear as well. We can reduce the space complexity to constant by noticing that we do not need to store the entire array `max_chunk` and `min_chunk`, but only their last value:

```
def find_max_prod_subarray(array):
    max_chunk = -float('inf')
    min_chunk = float('inf')
    global_max = 0
    for value in array:
        # Compute candidates for the max-product subarray
        # ending with the current element:
        # Candidate 1: start new subarray
        max_new_chunk = value
        # Candidate 2: continue positive subarray
        if max_chunk > 0 and value > 0:
            max_continue_positive = max_chunk * value
        else:
            max_continue_positive = -float('inf')
        # Candidate 3: flip sign of negative subarray
        if min_chunk < 0 and value < 0:
            max_flip_negative = min_chunk * value
        else:
            max_flip_negative = -float('inf')

        # Compute candidates for the min-product subarray
        # ending with the current element:
        # Candidate 1: start new subarray
        min_new_chunk = value
        # Candidate 2: continue negative subarray
        if min_chunk < 0 and value > 0:
            min_continue_negative = min_chunk * value
        else:
            min_continue_negative = float('inf')
        # Candidate 3: flip sign of positive subarray
        if max_chunk > 0 and value < 0:
            min_flip_positive = max_chunk * value
        else:
            min_flip_positive = float('inf')

        # Choose the best candidate
        max_chunk = max(max_new_chunk,
                        max_continue_positive,
                        max_flip_negative)
        min_chunk = min(min_new_chunk,
```

```
                        min_continue_negative,
                        min_flip_positive)
        global_max = max(global_max, max_chunk)
    return global_max
```

This solution is very fast, since it makes a single pass over the array. The disadvantage is that it is much less readable and more error-prone compared to the greedy solution. In a real situation, it should only be used if it is part of a critical path (so its performance matters) and if there are measurements showing that the greedy solution is performing significantly worse.

Unit tests

For testing, we start with simple test cases that cover:

- only positive numbers;
- even count of negative numbers;
- odd count of negative numbers;
- only zero or positive numbers;
- zeroes and positives and negative numbers (odd and even counts)

We also add some performance tests with large inputs, to benchmark the implementation as well as to catch performance regressions.

The unit tests can be written as:

```
class TestMaxProdSubarray(unittest.TestCase):
    def test_1_positives(self):
        self.assertEqual(find_max_prod_subarray([1, 2, 3]), 6)

    def test_2_even_count_negatives(self):
        self.assertEqual(find_max_prod_subarray([1, -2, -3]), 6)
        self.assertEqual(find_max_prod_subarray([1, -2, -3, -4, -5]), 120)

    def test_3_odd_count_negatives(self):
        self.assertEqual(find_max_prod_subarray([1, 2, -3]), 2)
        self.assertEqual(find_max_prod_subarray([1, -2, -3, -4, 5]), 60)

    def test_4_zeroes_positives(self):
        self.assertEqual(find_max_prod_subarray([1, 2, 3, 0, 4, 1]), 6)
        self.assertEqual(find_max_prod_subarray([1, 2, 3, 0, 4, 2]), 8)
        self.assertEqual(find_max_prod_subarray([0]), 0)
        self.assertEqual(find_max_prod_subarray([]), 0)

    def test_5_zeroes_even_count_negatives(self):
        self.assertEqual(find_max_prod_subarray([-2, -2, 0, -5, -1]), 5)
        self.assertEqual(find_max_prod_subarray([-2, -3, 0, -4, -1]), 6)
```

```
    def test_6_zeroes_odd_count_negatives(self):
        self.assertEqual(find_max_prod_subarray(
                         [-2, -2, 0, -3, -1, -2]), 4)
        self.assertEqual(find_max_prod_subarray(
                         [-2, -2, -2, 0, -3, -2, -2]), 6)

    def test_7_perf(self):
        n = 10000
        self.assertEqual(find_max_prod_subarray([2, 3] * n), 6 ** n)
        self.assertEqual(find_max_prod_subarray([-2, -3] * n), 6 ** n)
        self.assertEqual(find_max_prod_subarray([-2, 3] * n), 6 ** n)
        self.assertEqual(find_max_prod_subarray([-7] + [-2, 3] * n),
                         7 * 6 ** (n - 1))
        self.assertEqual(find_max_prod_subarray([-2, 3] * n + [-7]),
                         7 * 3 * 6 ** (n - 1))
```