Maximum Product Subarray in an Array

Problem Statement: Given an array that contains both negative and positive integers, find the maximum product subarray.

Examples

```
Example 1:
Input:
 Nums = [1,2,3,4,5,0]
Output:
 120
Explanation:
 In the given array, we can see 1×2×3×4×5 gives maximum product value.


Example 2:
Input:
 Nums = [1,2,-3,0,-4,-5]
Output:
 20
Explanation:
 In the given array, we can see (-4)×(-5) gives maximum product value.
```

Brute Force Approach
Algorithm / Intuition

Approach:

Find all possible subarrays of the given array. Find the product of each subarray. Return the maximum of all them.

Following are the steps for the approach:-

- Run a loop on the array to choose the start point for each subarray.
- Run a nested loop to get the end point for each subarray.
- Multiply elements present in the chosen range.

Dry Run:

Code

```
import java.util.*;
public class Main
{
        static int maxProductSubArray(int arr[]) {
            int result = Integer.MIN_VALUE;
            for(int i=0;i<arr.length-1;i++)
                for(int j=i+1;j<arr.length;j++) {
                    int prod = 1;
                    for(int k=i;k<=j;k++)
                        prod *= arr[k];
                    result = Math.max(result,prod);
                }
           return result;
        }
        public static void main(String[] args) {
                int nums[] = {1,2,-3,0,-4,-5};
                int answer = maxProductSubArray(nums);
                System.out.print("The maximum product subarray is: "+answer);
        }
}
```

Output: The maximum product subarray: 20

Complexity Analysis

Time Complexity: O(N3)

Reason: We are using 3 nested loops for finding all possible subarrays and their product.

Space Complexity: O(1)

Reason: No extra data structure was used

Better Approach
Algorithm / Intuition

Approach:

We can optimize the brute force by making 3 nested iterations to 2 nested iterations

Following are the steps for the approach:

- Run a loop to find the start of the subarrays.
- Run another nested loop
- Multiply each element and store the maximum value of all the subarray.

Dry Run:

Code

```
import java.util.*;

public class Main
{
        static int maxProductSubArray(int arr[]) {
            int result = arr[0];
            for(int i=0;i<arr.length-1;i++) {
                int p = arr[i];
                for(int j=i+1;j<arr.length;j++) {
                    result = Math.max(result,p);
                    p *= arr[j];
                }
                result = Math.max(result,p);
            }
           return result;
        }
        public static void main(String[] args) {
                int nums[] = {1,2,-3,0,-4,-5};
                int answer = maxProductSubArray(nums);
                System.out.print("The maximum product subarray is: "+answer);
        }
}
```

Output: The maximum product subarray: 20

Complexity Analysis

Time Complexity: O(N2)

Reason: We are using two nested loops

Space Complexity: O(1)

Reason: No extra data structures are used for computation

Optimal Approach 1:
Algorithm / Intuition

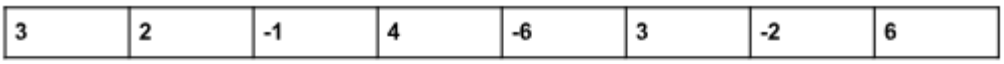
We will optimize the solution through some observations.

Observations:

1. **If the given array only contains positive numbers:** If this is the case, we can confidently say that the maximum product subarray will be the entire array itself.
2. **If the given also array contains an even number of negative numbers:** As we know, an even number of negative numbers always results in a positive number. So, also, in this case, the answer will be the entire array itself.

If the given array also contains an odd number of negative numbers: Now, an odd number of negative numbers when multiplied result in a negative number. Removal of 1 negative number out of the odd number of negative numbers will leave us with an even number of negatives. Hence the idea is to remove 1 negative number from the result. Now we need to decide which 1 negative number to remove such that the remaining subarray yields the maximum product.

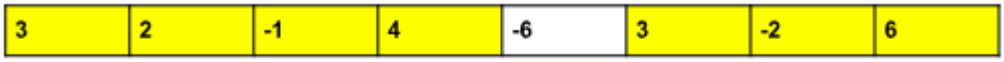
For example, the given array is: {3, 2, -1, 4, -6, 3, -2, 6}
We will try to remove each possible negative number and check in which case the subarray yields the maximum product.



Case 1: If we remove -1, the subarrays will be the following:

Product = 6 Product = 864

Case 2: If we remove -6, the subarrays will be the following:



Product = -24 Product = -36

Case 3: If we remove -2, the subarrays will be the following:

Product = 432 Product = 6

1. Upon observation, we notice that each chosen negative number divides the array into two parts.
2. The answer will either be the prefix or the suffix of that negative number.
3. To find the answer, we will check all possible prefix subarrays (starting from index 0) and all possible suffix subarrays (starting from index n-1).
4. The maximum product obtained from these prefix and suffix subarrays will be our final answer.
5. **If the array contains 0's as well:** We should never consider 0's in our answer(*as considering 0 will always result in 0*) and we want to obtain the maximum possible product. So, we will divide the given array based on the location of the 0's and apply the logic of case 3 for each subarray.

   For example, the given array is: {-2, 3, 4, -1, 0, -2, 3, 1, 4, 0, 4, 6, -1, 4}.
   1. In this case, we will divide the array into 3 different subarrays based on the 0's locations. So, the subarrays will be {-2, 3, 4, -1}, {-2, 3, 1, 4}, and {4, 6, -1, 4}.
   2. In these 3 subarrays, we will apply the logic discussed in case 3. We will get 3 different answers for 3 different subarrays.
   3. The maximum one among those 3 answers will be the final answer.

Summary: In real-life problems, we will not separate out the cases as we did in the observations. Instead, we can directly apply the logic discussed in the 4th observation to any given subarray, and it will automatically handle all the other cases.

Algorithm:

1. We will first declare 2 variables i.e. 'pre'(*stores the product of the prefix subarray*) and 'suff'(*stores the product of the suffix subarray*). They both will be initialized with 1(*as we want to store the product*).

2. Now, we will use a loop(say **i**) that will run from 0 to n-1.
3. We have to check 2 cases to handle the presence of 0:
    1. **If pre = 0:** This means the previous element was 0. So, we will consider the current element as a part of the new subarray. So, we will set ‘pre’ to 1.
    2. **If suff = 0:** This means the previous element was 0 in the suffix. So, we will consider the current element as a part of the new suffix subarray. So, we will set ‘suff’ to 1.
4. Next, we will multiply the elements from the starting index with ‘pre’ and the elements from the end with ‘suff’. To incorporate both cases inside a single loop, we will do the following:
    1. We will multiply arr[i] with ‘pre’ i.e. pre *= arr[i].
    2. We will multiply arr[n-i-1] with ‘suff’ i.e. suff *= arr[n-i-1].
5. After each iteration, we will consider the maximum among the previous answer, ‘pre’ and ‘suff’ i.e. max(previous_answer, pre, suff).
6. Finally, we will return the maximum product.

Code

```
import java.util.*;

public class Main {
    public static int maxProductSubArray(int[] arr) {
        int n = arr.length; //size of array.

        int pre = 1, suff = 1;
        int ans = Integer.MIN_VALUE;
        for (int i = 0; i < n; i++) {
            if (pre == 0) pre = 1;
            if (suff == 0) suff = 1;
            pre *= arr[i];
            suff *= arr[n - i - 1];
            ans = Math.max(ans, Math.max(pre, suff));
        }
        return ans;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, -3, 0, -4, -5};
        int answer = maxProductSubArray(arr);
        System.out.println("The maximum product subarray is: " + answer);
    }
}
```

Output: The maximum product subarray: 20

Complexity Analysis

Time Complexity: O(N), N = size of the given array.
Reason: We are using a single loop that runs for N times.

Space Complexity: O(1) as No extra data structures are used for computation.

Optimal Approach 2
Algorithm / Intuition

Approach:

The following approach is motivated by Kandane’s algorithm. To know Kadane’s Algorithm follow **Kadane's Algorithm**

The pick point for this problem is that we can get the maximum product from the product of two negative numbers too.

Following are the steps for the approach:

- Initially store 0th index value in prod1, prod2 and result.
- Traverse the array from 1st index.
- For each element, update prod1 and prod2.
- Prod1 is maximum of current element, product of current element and prod1, product of current element and prod2
- Prod2 is minimum of current element, product of current element and prod1, product of current element and prod2
- Return maximum of result and prod1

Codea

```
import java.util.*;
public class Main
{
        static int maxProductSubArray(int arr[]) {
    int prod1 = arr[0],prod2 = arr[0],result = arr[0];

    for(int i=1;i<arr.length;i++) {
        int temp = Math.max(arr[i],Math.max(prod1*arr[i],prod2*arr[i]));
        prod2 = Math.min(arr[i],Math.min(prod1*arr[i],prod2*arr[i]));
        prod1 = temp;

        result = Math.max(result,prod1);
    }

    return result;
        }
        public static void main(String[] args) {
                int nums[] = {1,2,-3,0,-4,-5};
                int answer = maxProductSubArray(nums);
                System.out.print("The maximum product subarray is: "+answer);
        }
}
```

Output: The maximum product subarray: 20

Complexity Analysis

Time Complexity: O(N)

Reason: A single iteration is used.

Space Complexity: O(1)

Reason: No extra data structure is used for computation