# Maximum Product Subarray

**Maximum Product Subarray**

Difficulty: **Medium** Accuracy: **18.09%** Submissions: **434K+** Points: **4**

Given an array **arr[]** that contains positive and negative integers (may contain 0 as well). Find the **maximum** product that we can get in a subarray of **arr[]**.

Note: It is guaranteed that the output fits in a 32-bit integer.

**Examples**

**Input:** arr[] = [-2, 6, -3, -10, 0, 2]
**Output:** 180
**Explanation:** The subarray with maximum product is {6, -3, -10} with product = 6 * (-3) * (-10) = 180.

**Input:** arr[] = [-1, -3, -10, 0, 6]
**Output:** 30
**Explanation:** The subarray with maximum product is {-3, -10} with product = (-3) * (-10) = 30.

**Input:** arr[] = [2, 3, 4]
**Output:** 24
**Explanation:** For an array with all positive elements, the result is product of all elements.

**Constraints:**
$1 \leq arr.size() \leq 10^6$
$-10 \leq arr[i] \leq 10$

[Naive Approach] By using two nested loops - O(n^2) time & O(1) space :-

The idea is to traverse over every contiguous subarray, find the product of each of these subarrays & return the maximum product among all the subarrays.

C++ (g++ 5.4) Start Timer

```cpp
// } Driver Code Ends

// User function Template for C++
class Solution {
  public:
    // Function to find maximum product subarray
    int maxProduct(vector<int> &arr) {
        // Your Code Here
        int n=arr.size();
        int result=arr[0];
        for(int i=0;i<n;i++){
            int mul=1;
            for(int j=i;j<n;j++){
                mul*=arr[j];
                result=max(result, mul);
            }
        }
        return result;
    }
};
```

Time Complexity : O(n^2), where n is the size of array

Auxiliary Space : O(1).

[Expected Approach 1] Using minimum & maximum product ending at any index - O(n) Time & O(1) space

Let's assume that the input array has only positive elements. Then, we can simply iterate from left to right keeping track of the

we can simply iterate from left to right keeping track of the maximum running product ending at any index. The maximum product would be the product ending at the last index. The problem arises when we encounter zero or a negative element.

- If we encounter zero, then all the subarrays containing this zero will have product = 0, so zero simply resets the product of the subarray.
- If we encounter a negative number, we need to keep track of the minimum product as well as the maximum product ending at the previous index. This is because when we multiply the minimum product with a negative number, it can give us the maximum product. So keeping track of minimum product ending at any index is important as it can lead to the maximum product on encountering a negative number.

**Step-by-step algorithm:**

- Create 3 variables, **currMin**, **currMax** and **maxProd** initialized to the first element of the array.
- Iterate the indices 0 to N-1 and update the variables:
  - currMax = maximum(arr[i], currMax * arr[i], currMin * arr[i])
  - currMin= minimum(arr[i], currMax * arr[i], currMin * arr[i])
  - update the maxProd with the maximum value for each index.
- Return maxProd as the result.

```cpp
// User function Template for C++
class Solution {
  public:
    // Function to find maximum product subarray
    int maxProduct(vector<int> &arr) {
        // Your Code Here
        int n=arr.size();

        // max product ending at the current index
        int currMax=arr[0];

        // min product ending at the current index
        int currMin=arr[0];

        // initialize overall max product
        int maxProd=arr[0];

        // iterate through the array
        for(int i=1;i<n;i++){

            // temporary varible to store the maximum product ending at the current index
            int temp=max({arr[i], arr[i]*currMax, arr[i]*currMin});

            // update the minimum product ending at the current index

            currMin=min({arr[i], arr[i]*currMax, arr[i]*currMin});

            //update the maximum product ending at the current index

            currMax=temp;

            //update the overall maximum product

            maxProd=max(maxProd, currMax);


        }
        return maxProd;
    }
};
```

T.C. O(n), where n is the size of the input array

Auxiliary Space: O(1)

[Expected Approach 2] By traversing in both directions -

O(n) Time & O(1) space

We will follow a simple approach that is to traverse from the start & keep track of the running product & if the running product is greater than the max product then we update the max product. Also, if we encounter '0' then make product of all elements till now equal to 1 because from the next element, we will start a new subarray.

Problem with this approach:-

Problem will occur when our array will contain odd no. of negative elements. In that case, we have to reject one negative element so that we can have even no. of negative elements & their product can be positive. Now, since subarray should be contiguous so we can't simply reject any one negative element. We have to either reject the first or last negative element.

Now, if we traverse from start then only the last negative element can be rejected & if we traverse from the last then the first element can be rejected. So will traverse from both

ends & find the maximum product subarray.

```cpp
#include <bits/stdc++.h>
using namespace std;

// function to find the product of max product subarray.
int maxProduct(vector<int> &arr) {
    int n = arr.size();
    int maxProd = INT_MIN;

    // leftToRight to store product from left to Right
    int leftToRight = 1;

    // rightToLeft to store product from right to left
    int rightToLeft = 1;

    for (int i = 0; i < n; i++) {
        if (leftToRight == 0)
            leftToRight = 1;
        if (rightToLeft == 0)
            rightToLeft = 1;

        // calculate product from index left to right
        leftToRight *= arr[i];

        // calculate product from index right to left
        int j = n - i - 1;
        rightToLeft *= arr[j];
        maxProd = max({leftToRight, rightToLeft, maxProd});
    }
    return maxProd;
}
```

T.C : O(N) where N is the size of array

S.C : O(1)