<https://leetcode.com/problems/maximum-product-subarray/>

Given an integer array nums, find a  subarray that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a **32-bit** integer.

**Example 1:**

Input: nums = [2,3,-2,4]

Output: 6

Explanation: [2,3] has the largest product 6.

**Example 2:**

Input: nums = [-2,0,-1]

Output: 0

Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

**Constraints:**

1 <= nums.length <= 2 \* 10^4

-10 <= nums[i] <= 10

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

**Attempt 1: 2023-03-13**

**Solution 1: Brute Force using three points control 'start' , 'end' and loop respectively as O(N^3) (10 min, TLE)**

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        int maxPro = Integer.MIN\_VALUE;

        for(int i = 0; i < n; i++) {

            for(int j = i; j < n; j++) {

                int curPro = 1;

                for(int k = i; k <= j; k++) {

                    curPro \*= nums[k];

                }

                maxPro = Math.max(maxPro, curPro);

            }

        }

        return maxPro;

    }

}

Time complexity: O(n^3)

Space complexity: O(1)

**Solution 2:  Brute Force with optimization using two points control 'start' , ('end' & loop) respectively as O(N^2) (10 min)**

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        int maxPro = Integer.MIN\_VALUE;

        for(int i = 0; i < n; i++) {

            int curPro = 1;

            for(int j = i; j < n; j++) {

                curPro \*= nums[j];

                maxPro = Math.max(maxPro, curPro);

            }

        }

        return maxPro;

    }

}

Time complexity: O(n^2)

Space complexity: O(1)

**Refer to**

<https://leetcode.com/problems/maximum-product-subarray/solutions/1609493/c-simple-solution-w-explanation-optimization-from-brute-force-to-dp/>

❌ ***Solution - I (Brute-Force)***

We can simply check for each subarray and return the subarray which gives the maximum product

class Solution {

public:

    int maxProduct(vector<int>& A) {

        int ans = INT\_MIN;

        for(int i = 0; i < size(A); i++)

            for(int j = i; j < size(A); j++)

                ans = max(ans, accumulate(begin(A)+i, begin(A)+j+1, 1, multiplies<>()));

        return ans;

    }

};

Or a slightly better version of brute-force -

class Solution {

public:

    int maxProduct(vector<int>& A) {

        int ans = INT\_MIN;

        for(int i = 0; i < size(A); i++) {

            int curProd = 1;

            for(int j = i; j < size(A); j++)

                curProd \*= A[j],

                ans = max(ans, curProd);

        }

        return ans;

    }

};

***Time Complexity :*** O(N^3) / O(N^2) for the 1st version and 2nd version respectively.

***Space Complexity :*** O(1), only constant extra space is used.

**Solution 3: Recursion (360 min, too hard to organize the logic on hard point 'considerPreValue', TLE)**

**Wrong Solution (If only copy the style from L53. Maximum Subarray without modification)**

**Test case:**

Input: nums =[-3,-1,-1]

Output: -1

Expected: 3

**Failure reason:**

**But we must first realize that maximum subarray product starting at index i is not always A[i]\*maxProduct(A[i+1:]) and can also be A[i]\*minProduct(A[i+1:]).**

class Solution {

    public int maxProduct(int[] nums) {

        return helper(nums, 0, false);

    }

    private int helper(int[] nums, int index, boolean considerPreValue) {

        if(index >= nums.length) {

            return considerPreValue ? 1 : (int)-2e5;

        }

        if(considerPreValue) {

            return Math.max(1, nums[index] \* helper(nums, index + 1, true));

        }

        return Math.max(nums[index] \* helper(nums, index + 1, true), helper(nums, index + 1, false));

    }

}

**Correct recursion way but TLE also**

class Solution {

    public int maxProduct(int[] nums) {

        return getMax(nums, 0, false);

    }

    // Finds maximum subarray product starting at i

    private int getMax(int[] nums, int index, boolean considerPreValue) {

        if(index >= nums.length) {

            return considerPreValue ? 1 : (int)-2e5;

        }

        if(considerPreValue) {

            return Math.max(1, nums[index] \* (nums[index] >= 0 ? getMax(nums, index + 1, true) : getMin(nums, index + 1)));

        }

        return Math.max(nums[index] \* (nums[index] >= 0 ? getMax(nums, index + 1, true) : getMin(nums, index + 1)), getMax(nums, index + 1, false));

    }

    // Finds minimum subarray product starting at i

    private int getMin(int[] nums, int index) {

        if(index >= nums.length) {

            return 1;

        }

        return nums[index] \* (nums[index] >= 0 ? getMin(nums, index + 1) : getMax(nums, index + 1, true));

    }

}

Time Complexity : O(N^2)

Space Complexity : O(N), required by implicit recursive stack

**Refer to**

<https://leetcode.com/problems/maximum-product-subarray/solutions/1609493/c-simple-solution-w-explanation-optimization-from-brute-force-to-dp/>

❌ ***Solution - II (Recursive)***

We can recursively calculate the maximum product of subarray as well. Here, we start at index 0 and recursively calculate the maximum subarray product starting at each index and return the one yielding the maximum overall product.

But we must first realize that maximum subarray product starting at index i is not always A[i]\*maxProduct(A[i+1:]) and can also be

A[i]\*minProduct(A[i+1:]). This is because when A[i] >= 0, multiplying it with maximum product after i will yield the largest overall product.

However, when A[i] < 0, we need to multiply it with minimum possible product after i to get the largest overall product (try with some examples if you didn't get this).

So, whenever A[i] >= 0, we call a function which returns maximum product of subarray starting from i+1 (dfsMax in the below code) and whenever A[i] < 0, we call function which returns minimum product of subarray starting from i+1 (dfsMin in below code).

In the below solution, I have also passed another boolean parameter mustPick which denotes if every element from now on must be picked or not. This helps ensure that we pick a subarray and not a subsequence. If A[i] is picked, for every index j > i, we must either pick every element

A[j] after that or end the recursion at any index j by return 1.

class Solution {

public:

    int maxProduct(vector<int>& A) {

        return dfsMax(A, 0, false);

    }

    // finds maximum subarray product starting at i

    long dfsMax(vector<int>& A, int i, bool mustPick) {

        if (i >= size(A)) return mustPick ? 1 : INT\_MIN;    // if mustPick remains false till end, the no element is chosen so return -∞ denoting invalid case

        if (mustPick)

            return max(1l, A[i] \* (A[i] >= 0 ? dfsMax(A, i+1, true) : dfsMin(A, i+1)));                // either end recursion by returning 1 or recurse by picking A[i]. Choose whichever gives max product

        return max(dfsMax(A, i+1, false), A[i] \* (A[i] >= 0 ? dfsMax(A, i+1, true) : dfsMin(A, i+1)));  // either pick A[i] or recurse for next index without picking. Choose whichever gives max product

    }

    // finds minimum subarray product starting at i

    long dfsMin(vector<int>& A, int i) {

        if (i >= size(A)) return 1;

        return A[i] \* (A[i] >= 0 ? dfsMin(A, i+1) : dfsMax(A, i+1, true));

    }

};

***Time Complexity :*** O(N^2)

***Space Complexity :*** O(N), required by implicit recursive stack

**Solution 4: Memoization + Recursion (30 min, based on Solution 3)**

class Solution {

    public int maxProduct(int[] nums) {

        Integer[][] max\_memo = new Integer[nums.length][2];

        Integer[] min\_memo = new Integer[nums.length];

        return getMax(nums, 0, 0, max\_memo, min\_memo);

    }

    // Finds maximum subarray product starting at i

    private int getMax(int[] nums, int index, int considerPreValue, Integer[][] max\_memo, Integer[] min\_memo) {

        if(index >= nums.length) {

            return considerPreValue == 1 ? 1 : (int)-2e5;

        }

        if(max\_memo[index][considerPreValue] != null) {

            return max\_memo[index][considerPreValue];

        }

        if(considerPreValue == 1) {

            return max\_memo[index][considerPreValue] = Math.max(1, nums[index] \* (nums[index] >= 0 ? getMax(nums, index + 1, 1, max\_memo, min\_memo) : getMin(nums, index + 1, max\_memo, min\_memo)));

        }

        return max\_memo[index][considerPreValue] = Math.max(nums[index] \* (nums[index] >= 0 ? getMax(nums, index + 1, 1, max\_memo,min\_memo) : getMin(nums, index + 1, max\_memo, min\_memo)), getMax(nums, index + 1, 0, max\_memo, min\_memo));

    }

    // Finds minimum subarray product starting at i

    private int getMin(int[] nums, int index, Integer[][] max\_memo, Integer[] min\_memo) {

        if(index >= nums.length) {

            return 1;

        }

        if(min\_memo[index] != null) {

            return min\_memo[index];

        }

        return min\_memo[index] = nums[index] \* (nums[index] >= 0 ? getMin(nums, index + 1, max\_memo, min\_memo) : getMax(nums, index + 1, 1, max\_memo, min\_memo));

    }

}

Time Complexity : O(N), we calculate dpMax[i] for each 0 <= i < N only once and return the stored result for every future call.

Space Complexity : O(N), required by dp arrays and recursive stack

**Refer to**

<https://leetcode.com/problems/maximum-product-subarray/solutions/1609493/c-simple-solution-w-explanation-optimization-from-brute-force-to-dp/>

✔️ ***Solution - III (Dynamic Programming - Memoization)***

We can observe a lot of repeated calculations if we draw out the recursive tree for above solution. The maximum product of subarray starting at i will always remain the same and there's no need to calculate it again and again. These redundant calculations can be eliminated if we store the results for a given state and reuse them later whenever required rather than recalculating them over and over again.

Thus, we can use memoization technique here to make our solution more efficient. Here, we use dpMax and dpMin arrays where dpMax[i][mustPick] denotes the maximum product subarray starting from i and mustPick denotes whether the current element must be compulsorily picked or we have an choice between picking and not picking it. Similarly, dpMin[i] denotes the minimum product subarray starting from i. Each element of dpMax and dpMin is initialized to INT\_MIN denoting they haven't been computed yet.

class Solution {

public:

    vector<int> dpMin;

    vector<vector<int>> dpMax;

    int maxProduct(vector<int>& A) {

        dpMin.resize(size(A), INT\_MIN);

        dpMax.resize(size(A), vector<int>(2, INT\_MIN));

        return dfsMax(A, 0, false);

    }

    long dfsMax(vector<int>& A, int i, bool mustPick) {

        if (i >= size(A)) return mustPick ? 1 : INT\_MIN;

        if(dpMax[i][mustPick] != INT\_MIN) return dpMax[i][mustPick];

        if (mustPick)

            return dpMax[i][mustPick] = max(1l, A[i] \* (A[i] >= 0 ? dfsMax(A, i+1, true) : dfsMin(A, i+1)));

        return dpMax[i][mustPick] = max(dfsMax(A, i+1, false), A[i] \* (A[i] >= 0 ? dfsMax(A, i+1, true) : dfsMin(A, i+1)));

    }

    long dfsMin(vector<int>& A, int i) {

        if (i >= size(A)) return 1;

        if(dpMin[i] != INT\_MIN) return dpMin[i];

        return dpMin[i] = A[i] \* (A[i] >= 0 ? dfsMin(A, i+1) : dfsMax(A, i+1, true));

    }

};

***Time Complexity :*** O(N), we calculate dpMax[i] for each 0 <= i < N only once and return the stored result for every future call.

***Space Complexity :*** O(N), required by dp arrays and recursive stack

**Solution 5: Dynamic Programming (60 min)**

**Style 1: 2D DP (Refer to L53.Maximum Subarray)**

class Solution {

    public int maxProduct(int[] nums) {

        int[][] dpMax = new int[2][nums.length];

        int[][] dpMin = new int[2][nums.length];

        // dpMax[1][i] denotes maximum subarray sum ending at i (including nums[i])

        // dpMax[0][i] denotes maximum subarray sum upto i (may or may not include nums[i])

        dpMax[1][0] = nums[0];

        dpMax[0][0] = dpMax[1][0];

        // dpMin[1][i] denotes minimum subarray sum ending at i (including nums[i])

        // dpMin[0][i] denotes minimum subarray sum upto i (may or may not include nums[i])

        dpMin[1][0] = nums[0];

        dpMin[0][0] = dpMin[1][0];

        for(int i = 1; i < nums.length; i++) {

            // At each index, we update dpMin[1][i] as min between either only

            // choosing current element - nums[i] or extending from previous

            // subarray and choosing current element as well - if nums[i] >= 0 then

            // dpMin[1][i-1] \* nums[i], if nums[i] < 0 then dpMax[1][i-1] \* nums[i]

            dpMin[1][i] = Math.min(nums[i], nums[i] \* (nums[i] >= 0 ? dpMin[1][i - 1] : dpMax[1][i - 1]));

            // Similarly, dpMin[0][i] can be updated as min between minimum sum

            // subarray found till last index - dpMin[0][i-1] or min subarray

            // sum found ending at current index dpMin[1][i]

            dpMin[0][i] = Math.min(dpMin[0][i - 1], dpMin[1][i]);

            // At each index, we update dpMax[1][i] as max between either only

            // choosing current element - nums[i] or extending from previous

            // subarray and choosing current element as well - if nums[i] >= 0 then

            // dpMax[1][i-1] \* nums[i], if nums[i] < 0 then dpMin[1][i-1] \* nums[i]

            dpMax[1][i] = Math.max(nums[i], nums[i] \* (nums[i] >= 0 ? dpMax[1][i - 1] : dpMin[1][i - 1]));

            // Similarly, dpMax[0][i] can be updated as max between maximum sum

            // subarray found till last index - dpMax[0][i-1] or max subarray

            // sum found ending at current index dpMax[1][i]

            dpMax[0][i] = Math.max(dpMax[0][i - 1], dpMax[1][i]);

        }

        return dpMax[0][nums.length - 1];

    }

}

**Style 2: 1D DP**

class Solution {

    public int maxProduct(int[] nums) {

        int[] dpMax = new int[nums.length];

        int[] dpMin = new int[nums.length];

        // dpMax[i] denotes maximum subarray sum ending at i (including nums[i])

        dpMax[0] = nums[0];

        // dpMin[i] denotes minimum subarray sum ending at i (including nums[i])

        dpMin[0] = nums[0];

        int max = nums[0];

        for(int i = 1; i < nums.length; i++) {

            dpMin[i] = Math.min(nums[i], nums[i] \* (nums[i] >= 0 ? dpMin[i - 1] : dpMax[i - 1]));

            dpMax[i] = Math.max(nums[i], nums[i] \* (nums[i] >= 0 ? dpMax[i - 1] : dpMin[i - 1]));

            max = Math.max(max, dpMax[i]);

        }

        return max;

    }

}

**Refer to**

<https://leetcode.com/problems/maximum-product-subarray/solutions/1609493/c-simple-solution-w-explanation-optimization-from-brute-force-to-dp/>

✔️ ***Solution - IV (Dynamic Programming - Tabulation)***

We can solve this iteratively as well using DP. Here, we again use dpMax and dpMin arrays where dpMax[i] denotes maximum subarray product ending at i and dpMin[i] denotes minimum subarray product ending at i.

At each index i, we will update dpMin[i] as minimum of A[i] (denotes forming new subarray by choosing current element) and min(A[i] \* dpMin[i-1], A[i] \* dpMax[i-1]) (denotes expanding previous subarray product by including current element). We do it similarly for dpMax[i] as well.

Finally, the maximum product subarray will be the maximum value in dpMax array.

class Solution {

public:

    int maxProduct(vector<int>& A) {

        vector<int> dpMin(A), dpMax(A);

        int ans = A[0];

        for(int i = 1; i < size(A); i++) {

            dpMin[i] = min(A[i], A[i] \* (A[i] >= 0 ? dpMin[i-1] : dpMax[i-1]));

            dpMax[i] = max(A[i], A[i] \* (A[i] >= 0 ? dpMax[i-1] : dpMin[i-1]));

            ans = max(ans, dpMax[i]);

        }

        return ans;

    }

};

***Time Complexity :*** O(N)

***Space Complexity :*** O(N)

**Style 3: 1D DP with O(1) space**

class Solution {

    public int maxProduct(int[] nums) {

        // dpMax denotes maximum subarray sum ending at i (including nums[i])

        int dpMax = nums[0];

        // dpMin denotes minimum subarray sum ending at i (including nums[i])

        int dpMin = nums[0];

        int max = nums[0];

        for(int i = 1; i < nums.length; i++) {

            // Store previous max/min (represents original dpMax[i - 1]/dpMin[i - 1])

            int preDpMax = dpMax;

            int preDpMin = dpMin;

            dpMin = Math.min(nums[i], nums[i] \* (nums[i] >= 0 ? preDpMin : preDpMax));

            dpMax = Math.max(nums[i], nums[i] \* (nums[i] >= 0 ? preDpMax : preDpMin));

            max = Math.max(max, dpMax);

        }

        return max;

    }

}

**Refer to**

<https://leetcode.com/problems/maximum-product-subarray/solutions/1609493/c-simple-solution-w-explanation-optimization-from-brute-force-to-dp/>

✔️ ***Solution - V (Space-Optimized Dynamic Programming)***

In the above solution, dpMax[i] and dpMin[i] only depended on previous index and thus we can optimize space by not storing the entire array but only two variables dpMax and dpMin and update them iteratively based on their previous value in the same way as done above.

class Solution {

public:

    int maxProduct(vector<int>& A) {

        int ans = A[0], dpMin = A[0], dpMax = A[0];

        for(int i = 1; i < size(A); i++) {

            auto prevDpMin = dpMin, prevDpMax = dpMax;

            dpMin = min(A[i], A[i] \* (A[i] >= 0 ? prevDpMin : prevDpMax));

            dpMax = max(A[i], A[i] \* (A[i] >= 0 ? prevDpMax : prevDpMin));

            ans = max(ans, dpMax);

        }

        return ans;

    }

};

***Time Complexity :*** O(N)

***Space Complexity :*** O(1)

**Style 4: 1D DP with O(1) space**

class Solution {

    public int maxProduct(int[] nums) {

        // dpMax denotes maximum subarray sum ending at i (including nums[i])

        int dpMax = nums[0];

        // dpMin denotes minimum subarray sum ending at i (including nums[i])

        int dpMin = nums[0];

        int max = nums[0];

        for(int i = 1; i < nums.length; i++) {

            if(nums[i] < 0) {

                int tmp = dpMax;

                dpMax = dpMin;

                dpMin = tmp;

            }

            dpMin = Math.min(nums[i], nums[i] \* dpMin);

            dpMax = Math.max(nums[i], nums[i] \* dpMax);

            max = Math.max(max, dpMax);

        }

        return max;

    }

}

**Refer to**

<https://leetcode.com/problems/maximum-product-subarray/solutions/1608862/java-3-solutions-detailed-explanation-using-image/>

Just the slight modification of previous approach. As we know that on multiplying with negative number max will become min and min will become max, so why not as soon as we encounter negative element, we swap the max and min already.

class Solution {

    public int maxProduct(int[] nums) {

        int max = nums[0], min = nums[0], ans = nums[0];

        int n = nums.length;

        for (int i = 1; i < n; i++) {

            // Swapping min and max

            if (nums[i] < 0){

                int temp = max;

                max = min;

                min = temp;

            }

            max = Math.max(nums[i], max \* nums[i]);

            min = Math.min(nums[i], min \* nums[i]);

            ans = Math.max(ans, max);

        }

        return ans;

    }

}

**Thought process and useful strategy**

<https://leetcode.com/problems/maximum-product-subarray/solutions/847520/thought-process-and-useful-strategy/>

The following documents my thought process for solving the question, shared to illuminate a possible strategy for approaching problems such as this one.First, come up with a brute-force solution with ~O(N^2) running time:

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        if (n == 0) {

            throw new IllegalArgumentException();

        }

        int ans = Integer.MIN\_VALUE;

        for (int i = 0; i <n; i++) {

            int cur = 1;

            for (int j = i; j < n; j++) {

                cur = cur \* nums[j];

                ans = Math.max(ans, cur);

            }

        }

        return ans;

    }

}

Then, thinking about Kadane's algorithm used to solve [Maximum Subarray](https://leetcode.com/problems/maximum-subarray/), we can come up with the following optimization.

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        if (n == 0) {

            throw new IllegalArgumentException();

        }

        int cur = nums[0];

        int ans = nums[0];

        for (int i = 1; i < n; i++) {

            cur = Math.max(nums[i]\*cur, nums[i]);

            ans = Math.max(ans, cur);

        }

        return ans;

    }

}

**Only, it fails test cases in which two negative numbers produce a maximum sum. An example is [-2,3,-4]. Therefore, it only works for an input with no more than one negative number.**

For me, I was stuck at this point until I saw that the question was tagged with "Dynamic Programming". In the interview setting, the interviewer might give you this hint.

With the hint, we can proceed to keep track of the subproblem solutions. Since we observed that negative values can produce the maximum product, we keep track of both maximum product and the minimum product. The minimum product, when multiplied by another negative number, can produce a possible answer. The following solution passes.

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        if (n == 0) {

            throw new IllegalArgumentException();

        }

        int[] maxVal = new int[n];

        int[] minVal = new int[n];

        maxVal[0] = nums[0];

        minVal[0] = nums[0];

        for (int i = 1; i < n; i++) {

            maxVal[i] = Math.max(maxVal[i-1]\*nums[i], Math.max(minVal[i-1]\*nums[i], nums[i]));

            minVal[i] = Math.min(maxVal[i-1]\*nums[i], Math.min(minVal[i-1]\*nums[i], nums[i]));

        }

        int ans = Integer.MIN\_VALUE;

        for (int num : maxVal) {

            ans = Math.max(ans, num);

        }

        return ans;

    }

}

It is a good idea to check for space optimization for a DP solution. In this case, we observe that we only ever need to keep track of the subproblem solution immediately preceding our current subproblem. Hence we can avoid allocating an array, and improve the space complexity from ~O(N) to ~O(1).

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        if (n == 0) {

            throw new IllegalArgumentException();

        }

        int curMax = nums[0];

        int curMin = nums[0];

        int ans = curMax;

        for (int i = 1; i < n; i++) {

            int tmp = curMax;

            curMax = Math.max(tmp\*nums[i], Math.max(curMin\*nums[i], nums[i]));

            curMin = Math.min(tmp\*nums[i], Math.min(curMin\*nums[i], nums[i]));

            ans = Math.max(ans, curMax);

        }

        return ans;

    }

}

**Refer to**

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)