**Maximum Product of Three Numbers**

**1-Brute Force Approach**

### ****Problem Statement****

Given an integer array nums, find three numbers whose product is maximum and return the product.

### ****Brute Force (Naive) Approach****

**Time Complexity:** O(n³)  
**Space Complexity:** O(1)

#### ****Pseudo Code****

FUNCTION maximumProduct(nums):

n = LENGTH(nums)

max\_product = -INFINITY

FOR i FROM 0 TO n-3:

FOR j FROM i+1 TO n-2:

FOR k FROM j+1 TO n-1:

current\_product = nums[i] \* nums[j] \* nums[k]

IF current\_product > max\_product:

max\_product = current\_product

RETURN max\_product

#### ****Explanation****

**1-Input:**

A list of numbers: nums

n stores the number of elements in the list.

max\_product is initialized to a very small number so that any real product will be larger.

**2-Triple Nested Loop**: Checks all possible triplets (i, j, k) where i < j < k.

**3-Track Maximum Product**:

These loops generate **all unique combinations of three different indices** in the array.

i < j < k ensures that no index is repeated and that the same triplet in different orders is not considered again.

-current\_product = nums[i] \* nums[j] \* nums[k]

--If the current product is greater than the maximum found so far, update max\_product

**4**- **Result** : The largest product after all iterations.  
 --RETURN max\_product.

#### ****Complexity Analysis****

* ****Time**: O(n³) → Evaluates all combinations (slow for large arrays).**
* ****Space**: O(1) → Uses only constant extra space.**

#### ****Example****

For nums = [1, 2, 3, 4]:

* Checks (1,2,3)=6, (1,2,4)=8, (1,3,4)=12, (2,3,4)=24 → Returns **24**.

#### ****Limitation****

Inefficient for large n (e.g., 1000 elements → ~166M operations).

2-Recursive Structure:

**At each recursive step, we can either include or exclude an element. So, for each element, we have two choices (to include or exclude).**

**The recursive process will explore all possible combinations of selecting elements, but we stop when count == 3 (i.e., when 3 elements are selected) or when index == n (i.e., we've considered all elements).**

#### ****Pseudo Code:****

**Function MaxProductRecursive(A, n)**

**{**

**// Initialize variables for maximum product**

**maxProduct ← -∞;**

**// Function to calculate the product recursively**

**Function CalculateProduct(index, count, currentProduct)**

**{**

**// Base case: When 3 elements are selected, calculate the product**

**if count = 3**

**{**

**if currentProduct > maxProduct**

**{**

**maxProduct ← currentProduct;**

**}**

**return;**

**}**

**// Base case: If all elements have been processed, return**

**if index = n**

**{**

**return;**

**}**

**// Case 1: Include the current element in the product**

**CalculateProduct(index + 1, count + 1, currentProduct \* A[index]);**

**// Case 2: Exclude the current element and move to the next**

**CalculateProduct(index + 1, count, currentProduct);**

**}**

**// Start the recursive function with index 0, count 0, and currentProduct 1**

**Call CalculateProduct(0, 0, 1);**

**// Return the maximum product found**

**return maxProduct;**

**}**

**Number of Recursive Calls:**

**The function explores all subsets of elements of size 3 from the array of size n.**

**This is equivalent to generating all possible combinations of 3 elements from the array, which can be done in C(n, 3) ways, where:**

**C(n, 3) = n(n-1)(n-2)/6**

**This results in O(n³) combinations**.

**Time Complexity:**

**Each recursive call involves a constant amount of work (basic arithmetic and comparison).**

**The number of recursive calls is proportional to C(n, 3), which is O(n³).**

**Thus, the time complexity of the algorithm is O(n³).**

**Space Complexity:**

**The space complexity is driven by the recursive call stack. In the worst case, the depth of recursion will be n, as we could potentially visit each element once.**

**Therefore, the space complexity is O(n) due to the call stack.**

**Conclusion:**

**Time complexity: O(n³)**

**Space complexity: O(n)**

**3-Non-Recursive Structure:**

#### ****Pseudo Code:****

**FUNCTION MaxProductOfThree(arr):**

**IF length of arr < 3:**

**RETURN "Not enough numbers to form a product"**

**SORT arr in ascending order**

**# Case 1: Largest three numbers**

**max\_product1 ← arr[-1] \* arr[-2] \* arr[-3]**

**# Case 2: Two smallest numbers (negative) \* Largest number**

**max\_product2 ← arr[0] \* arr[1] \* arr[-1]**

**RETURN max(max\_product1, max\_product2)**

**Iteration Analysis:**

**Selection Process:**

**Instead of exploring all possible subsets of three elements, sorting enables direct selection of numbers that could form the maximum product. Sorting ensures that:**

**1. The largest three numbers are positioned at the end of the array.**

**2. The smallest two numbers (potentially negative) are positioned at the beginning.**

**Thus, only these numbers need to be examined, eliminating unnecessary comparisons.**

**Number of Computations:**

**The function avoids \*exponential growth\* seen in exhaustive subset selection.**

**- Sorting requires O(N log N) operations.**

**- Selecting elements and computing their product requires O(1) operations.**

**- Comparing the two cases also requires O(1) operations.**

**Thus, the total number of computations follows**

**O(N log N) complexity.**

**Time Complexity:**

**- Sorting dominates the execution time at O(N log N).**

**- All further operations (selection, comparison) occur in O(1).**

**-Overall time complexity: O(N log N).**

**Space Complexity:**

**- Sorting may be performed \*in-place, requiring O(1) extra space.**

**- No additional memory allocation beyond basic variable storage.**

**-Overall space complexity: O(1).**

**Conclusion:**

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

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