Maximum Product of Three Numbers: Algorithm Comparison

Method 1: Heap Approach Pseudocode (Non-Recursive Approach)

Pseudocode:

```
Algorithm MaxProductHeap(nums)
{
    largest_three := FindNLargest(nums, 3);
    smallest_two := FindNSmallest(nums, 2);

    product1 := largest_three[0] * largest_three[1] * largest_three[2];
    product2 := smallest_two[0] * smallest_two[1] * largest_three[0];
    Return Maximum(product1, product2);
}

Algorithm FindNLargest(nums, n)
{
    maxHeap := CreateMaxHeap(nums);
    result := [];
    For i := 1 to n step 1 do
    {
        If NOT IsEmpty(maxHeap) then
        {
            result.append(ExtractMax(maxHeap));
        }
    }
    Return result;
}

Algorithm FindNSmallest(nums, n)
{
    minHeap := CreateMinHeap(nums);

    result := [];
    For i := 1 to n step 1 do
    {
        If NOT IsEmpty(minHeap) then
        {
            result.append(ExtractMin(minHeap));
        }
    }
    Return result;
}
```

```
Time Complexity:
O(n): Building the heap takes O(n), and extracting the top elements
takes O(log n) for each extraction.
```

Code:

```python
import heapq

def max_product_heap(nums):

    largest_three = heapq.nlargest(3, nums)
    smallest_two = heapq.nsmallest(2, nums)

    product1 = largest_three[0] * largest_three[1] * largest_three[2]
    product2 = smallest_two[0] * smallest_two[1] * largest_three[0]

    return max(product1, product2)

def main():
    try:
        input_str = input("Enter integers separated by spaces: ")
        nums = list(map(int, input_str.strip().split()))

        if len(nums) < 3:
            print("Error: At least 3 integers are required.")
            return

        result = max_product_heap(nums)
        print(f"Maximum product of three numbers: {result}")

    except ValueError:
        print("Error: Please enter valid integers.")


if __name__ == "__main__":
    main()
```

Approach:

- This method extracts the top 3 largest and 2 smallest values using heaps. The same two product possibilities are considered as in the recursive method.

Time Complexity:

- Building heap: **O(n)**
- Extracting k elements: **O(k log n)**
- Total: **O(n + k log n)** → **O(n)** for fixed small k

Analysis:

Time Complexity of time efficiency : $T(n) \approx C\_op * C(n)$

Input Size : n

Analysis of Find Largest (max, 3):

create Max Heap(nums) → $T(n) = O(n)$ → proportional to size

Extract Max: $T(n) = 3 \times \log n = O(\log n)$

$C\_1 = O(n) + O(\log n) = \max\{O(n), O(\log n)\} = O(n)$

Analysis of Find Smallest (min, 2) is the same as Find Largest

$T(n) = O(n)$ → create min heap(nums) $= O(n)$

→ extract min $= O(\log n)$

Analysis of Max Product Heap (nums): (3 multiplications together, 1 comparison) → $T(n) = O(1)$

C_1: heapq.nlargest(3, nums) → $O(n)$, **C_2:** heapq.nsmallest(2, nums) → $O(n)$

C_3: Constant-time multiplications and comparisons → $O(1)$

$C(n) = C_1 + C_2 + C_3 = O(n) + O(n) + O(1) = O(n)$ → Total C(n)

$T(n) \approx C\_op * C(n) = O(n)$

Method 2: LinearScanRecursive (Recursive Approach)

Pseudocode:

```
Algorithm MaxProductLinearScanRecursive(nums)
{
    Return Recurse(nums, 0, -∞, -∞, -∞, ∞, ∞)
}

Algorithm Recurse(nums, index, max1, max2, max3, min1, min2)
{
    If index = length(nums) then
    {
        product1 := max1 * max2 * max3
        product2 := min1 * min2 * max1
        Return Maximum(product1, product2)
    }

    num := nums[index]
    If num >= max1 then
    {
        max3 := max2
        max2 := max1
        max1 := num
    }
    Else if num >= max2 then
    {
        max3 := max2
        max2 := num
    }
    Else if num >= max3 then
    {
        max3 := num
    }

    If num <= min1 then
    {
        min2 := min1
        min1 := num
    }
    Else if num <= min2 then
    {
        min2 := num
    }
    Return Recurse(nums, index + 1, max1, max2, max3, min1, min2)
}
```

```
Time Complexity: O(n), where n is the length of the input array.
```

Code:

```
def max_product_recursive(nums):
    def recurse(index, max1, max2, max3, min1, min2):
        if index == len(nums):
            product1 = max1 * max2 * max3
            product2 = min1 * min2 * max1
            return max(product1, product2)

        num = nums[index]

        if num >= max1:
            max3, max2, max1 = max2, max1, num
        elif num >= max2:
            max3, max2 = max2, num
        elif num >= max3:
            max3 = num

        if num <= min1:
            min2, min1 = min1, num
        elif num <= min2:
            min2 = num

        return recurse(index + 1, max1, max2, max3, min1, min2)

    return recurse(0, float('-inf'), float('-inf'), float('-inf'), float('inf'),
float('inf'))

def main():
    try:
        input_str = input("Enter integers separated by spaces: ")
        nums = list(map(int, input_str.strip().split()))

        if len(nums) < 3:
            print("Error: At least 3 integers are required.")
            return

        result = max_product_recursive(nums)
        print(f"Maximum product of three numbers: {result}")

    except ValueError:
        print("Error: Please enter valid integers.")
```

```python
if __name__ == "__main__":
    main()
```

Approach:

This method tracks the three largest and two smallest numbers recursively. Two possible products are considered:

- Product of the three largest numbers
- Product of the two smallest numbers (potentially negative) and the largest number

---

Time Complexity:

Recurrence Relation:

- T(n) = T(n-1) + O(1), T(n): represents the time complexity for an array of size n
- T(n-1) is the recursive call with one fewer element
- O(1) is the constant time comparison operations at each step
- Base case: T(0) = O(1)

- Each element is processed exactly once: O(n)

Using iteration method for analysing (Recursive Approach):

Recurrence Relation:

T(n) = T(n-1) + 1

T(n) = T(n-2) + 1 + 1

T(n) = T(n-3) + 1 + 1 + 1

T(n) = T(n-4) + 1 + 1 + 1 + 1

T(n) = T(n-5) + 1 + 1 + 1 + 1 + 1

T(n) = T(n-k) + k , let n-k = 0 , n = k

T(n) = n , T(n) = O(n)

Algorithm Comparison Criteria:

Algorithm	**Linear Scan (Recursive)**	**Heap-Based Approach**
Time Complexity	O(n)	O(n + k log n) ≈ O(n)
Handles Negatives?	Yes	Yes
Code Simplicity	Simple logic, recursive	More abstract, uses heaps
Generalizable to k?	Not easily	Easily (Find top-k or bottom-k)
Best Use Case	Static arrays, small size	Large arrays, frequent top-k ops

---

Key Insights

Linear Scan Approach:

- More straightforward implementation
- Better space efficiency with iterative implementation
- Ideal for one-time processing of smaller arrays

Heap-Based Approach:

- More flexible and generalizable to other "top-k" problems
- Provides a structured data organization
- Better for scenarios where you need to find extremes frequently

Important Edge Cases:

- Arrays with negative numbers (potentially large product from two negatives)
- Arrays with zeros
- Arrays with fewer than 3 elements (special handling needed)