628. Maximum Product of Three Numbers

Easy Array Math Sorting

Problem Description

The given problem provides an integer array called `nums`. The objective is to determine three numbers from this array such that their product is the maximum possible among all combinations of three numbers from the array. The function should then return this maximum product. This is essentially a problem of combination and maximizing an objective function under certain constraints (in this case, the product of three numbers).

Intuition

The intuition behind the solution approach is based on considering the nature of the product operation in mathematics and the distribution of positive and negative values within the integer array:

1. The largest product of three numbers could be the product of the three largest numbers in the array. This is simply because larger numbers generally lead to larger products, especially when they are all positive.
2. However, if the array contains negative numbers, the scenario changes slightly. The product of two negative numbers is positive. Therefore, if the largest number is positive, the product of this number with two large negative numbers (which become positive when multiplied together) could potentially be larger than the product of three positive numbers. This situation arises if there are at least two negative numbers in the array with large absolute values.

Combining these two observations, we realize there are two scenarios to consider for obtaining the maximum product:

- The product of the three largest numbers (in case they are all positive or two of them are negative).
- The product of the largest number and the two smallest numbers (which would be the largest negative numbers if negatives are present).

The Python code implements this approach by first finding the three largest numbers in the array using a built-in function `nlargest(3, nums)`, which returns the three largest numbers in descending order. Then it finds the two smallest numbers by once again using `nlargest(2, nums, key=lambda x: -x)` where the key argument transforms the problem into finding the "largest" numbers when viewed as negative, essentially giving us the smallest numbers of the array.

Finally, it computes the maximum product by considering both scenarios mentioned above and returns the greater of the two.

Solution Approach

To implement the solution, the Python code uses two important functions from the `heapq` module: `nlargest()` and `nsmallest()`. However, the provided solution cleverly only uses `nlargest()` in both cases, once with the default behavior and once with a key function to change the behavior.

The `heapq.nlargest(n, iterable)` function is used to find out the `n` largest numbers from the given `iterable`. It is an efficient way to obtain the largest values without the need to sort the entire array. Sorting would have a time complexity of `O(nlogn)`, but `nlargest()` can perform the same task in `O(nlogk)` time complexity, where `k` is the number of largest elements to find, which is more efficient when `k` is much smaller than `n`.

Here's a breakdown of how the code operates:

- `top3 = nlargest(3, nums)`: This line uses the `nlargest()` function to find the three largest numbers in the `nums` array. The resulting list `top3` contains these numbers in descending order.
- `bottom2 = nlargest(2, nums, key=lambda x: -x)`: This line again uses the `nlargest()` function, but with a key function `lambda x: -x` that negates the numbers in `nums`. By doing this, the function effectively retrieves the two smallest numbers, because negating the numbers turns the task of finding the largest negative numbers into finding the largest numbers post-negation.
- `return max(top3[0] * top3[1] * top3[2], top3[0] * bottom2[0] * bottom2[1])`: This is the final step which compares the product of the top three largest numbers with the product of the largest number and the two smallest numbers (which could be negative). The `max()` function is used here to return the larger of these two products, which is the desired maximum product of three numbers in the array.

By using a combination of heap operations and the `max()` function, the solution arrives at the correct maximum product without having to sort the entire array, thus making the solution more efficient for larger inputs.

Example Walkthrough

Let's consider a small example with the following integer array:

```
nums = [-10, -10, 5, 4]
```

Here is how the solution approach would work with this array:

1. **Find the three largest numbers**: We use the `heapq.nlargest(3, nums)` function, which would return `[5, 4, -10]` because 5 is the largest, followed by 4, and then -10 (which is larger than the other -10 based on how `nlargest` breaks ties). This gives us our first potential set of numbers for the maximum product.
2. **Find the two smallest numbers**: We employ the `heapq.nlargest(2, nums, key=lambda x: -x)` function, which with the negating key finds the "largest" numbers when viewed as negative, effectively returning `[-10, -10]` because these are the smallest numbers when their sign is negated (they become the largest positively).
3. **Calculate the products and find the maximum**: We have two sets of numbers to consider, `[5, 4, -10]` and `[5, -10, -10]`. We calculate the products of these combinations:
   - The product of `5 * 4 * -10` is `-200`.
   - The product of `5 * -10 * -10` is `500`.
4. **Return the maximum product**: Between the two products `-200` and `500`, the larger product is `500`, which is the expected output from our function.

The application of the solution has clearly walked us through a simple example and demonstrated the effectiveness of considering both the largest positive and largest negative numbers in the array to obtain the maximum product of three numbers.

Solution Implementation

Python

```python
from heapq import nlargest  # Import the nlargest function from heapq module

class Solution:
    def maximumProduct(self, nums: List[int]) -> int:
        # Find the three largest numbers from the list using nlargest
        top_3 = nlargest(3, nums)
        # Find the two smallest numbers from the list using nlargest with a key that inverts the values for sorting
        bottom_2 = nlargest(2, nums, key=lambda x: -x)

        # The maximum product can be either the product of the three largest numbers
        # or the product of the two smallest numbers and the largest number
        # (in case of two large negative numbers and one positive number)
        return max(
            top_3[0] * top_3[1] * top_3[2],  # Product of the three largest numbers
            top_3[0] * bottom_2[0] * bottom_2[1]  # Product of the largest number and two smallest (negative) numbers
        )
```

Java

```java
class Solution {
    public int maximumProduct(int[] nums) {
        // Define infinity value to represent very high positive value
        // as Java doesn't include infinity for integers.
        final int infinity = Integer.MAX_VALUE;

        // Initialize variables to represent the smallest and second smallest numbers
        int min1 = infinity;
        int min2 = infinity;

        // Initialize variables to represent the largest, second largest, and third largest numbers
        int max1 = -infinity;
        int max2 = -infinity;
        int max3 = -infinity;

        // Traverse through the array
        for (int num : nums) {

            // Check if current number is smaller than the smallest or second smallest
            if (num <= min1) {
                min2 = min1;  // Smallest number becomes second smallest
                min1 = num;   // Current number is the new smallest
            } else if (num <= min2) {
                min2 = num;   // Current number is the new second smallest
            }

            // Check if current number is larger than the largest, second or third largest
            if (num >= max1) {
                max3 = max2;  // Second largest number becomes third largest
                max2 = max1;  // Largest number becomes second largest
                max1 = num;   // Current number is the new largest
            } else if (num >= max2) {
                max3 = max2;  // Second largest number becomes third largest
                max2 = num;   // Current number is the new second largest
            } else if (num >= max3) {
                max3 = num;   // Current number is the new third largest
            }
        }

        // Compute the maximum product by comparing two possibilities:
        // 1. Product of the three largest numbers.
        // 2. Product of the smallest two numbers and the largest number.
        // This accounts for the case where the two smallest numbers might be negative,
        // and their product with the largest positive number could be maximum.
        return Math.max(min1 * min2 * max1, max1 * max2 * max3);
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Required for std::max

class Solution {
public:
    int maximumProduct(vector<int>& nums) {
        // Initialize constants and variables to keep track of the smallest and largest values.
        const int MAX_INT = INT_MAX;  // Using INT_MAX from climits instead of a hardcoded value.
        const int MIN_INT = INT_MIN;  // Using INT_MIN for clarity when initializing maximums.
        int min1 = MAX_INT, min2 = MAX_INT;  // Smallest and second smallest numbers.
        int max1 = MIN_INT, max2 = MIN_INT, max3 = MIN_INT; // Largest, second, and third largest numbers.

        // Iterate over the numbers to find the top two minimums and top three maximums.
        for (int num : nums) {
            // Check for new smallest or second smallest.
            if (num <= min1) {
                min2 = min1;
                min1 = num;
            } else if (num < min2) {
                min2 = num;
            }

            // Check for new largest, second or third largest.
            if (num > max1) {
                max3 = max2;
                max2 = max1;
                max1 = num;
            } else if (num > max2) {
                max3 = max2;
                max2 = num;
            } else if (num > max3) {
                max3 = num;
            }
        }

        // The maximum product can either be from three largest numbers
        // or from two smallest numbers (which might be negative) and the largest number.
        return std::max(min1 * min2 * max1, max1 * max2 * max3);
    }
};
```

TypeScript

```typescript
function maximumProduct(nums: number[]): number {
    // Initialize the smallest and second smallest values with the maximum safe integer.
    let smallest = Number.MAX_SAFE_INTEGER;
    let secondSmallest = Number.MAX_SAFE_INTEGER;
    // Initialize the largest, second largest, and third largest values with the minimum safe integer.
    let largest = Number.MIN_SAFE_INTEGER;
    let secondLargest = Number.MIN_SAFE_INTEGER;
    let thirdLargest = Number.MIN_SAFE_INTEGER;

    for (const num of nums) {
        // Check if current number is smaller than the smallest or the second smallest.
        if (num < smallest) {
            secondSmallest = smallest;
            smallest = num;
        } else if (num < secondSmallest) {
            secondSmallest = num;
        }

        // Check if current number is larger than the largest, second largest, or third largest.
        if (num > largest) {
            thirdLargest = secondLargest;
            secondLargest = largest;
            largest = num;
        } else if (num > secondLargest) {
            thirdLargest = secondLargest;
            secondLargest = num;
        } else if (num > thirdLargest) {
            thirdLargest = num;
        }
    }

    // Calculate and return the maximum product of three numbers.
    // Need to consider the product of the largest with two smallest (could be negatives making a positive)
    // and the product of the three largest numbers.
    return Math.max(smallest * secondSmallest * largest, largest * secondLargest * thirdLargest);
}
```

```python
from heapq import nlargest  # Import the nlargest function from heapq module

class Solution:
    def maximumProduct(self, nums: List[int]) -> int:
        # Find the three largest numbers from the list using nlargest
        top_3 = nlargest(3, nums)
        # Find the two smallest numbers from the list using nlargest with a key that inverts the values for sorting
        bottom_2 = nlargest(2, nums, key=lambda x: -x)

        # The maximum product can be either the product of the three largest numbers
        # or the product of the two smallest numbers and the largest number
        # (in case of two large negative numbers and one positive number)
        return max(
            top_3[0] * top_3[1] * top_3[2],  # Product of the three largest numbers
            top_3[0] * bottom_2[0] * bottom_2[1]  # Product of the largest number and two smallest (negative) numbers
        )
```

Time and Space Complexity

The provided Python code computes the maximum product of three numbers in a list using two operations: finding the three largest elements and the two smallest (or "bottom") elements.

Time Complexity

The time complexity of the `nlargest(3, nums)` operation is `O(N * log(3))` since it maintains a heap of size 3 during iteration over the list of `N` numbers, and each insertion into the heap takes logarithmic time. Similarly, the `nlargest(2, nums, key=lambda x: -x)` operation has a time complexity of `O(N * log(2))`. However, because the base of the logarithm is constant and small, the complexities can be considered close to `O(N)` for each operation, and because they do not depend on each other and are not nested, they could be summed up. Therefore, the overall time complexity is `O(N + N)`, which simplifies to `O(N)`.

Space Complexity

The space complexity is the additional space required besides the input. Here, it includes the space for storing the largest and smallest elements. Since it stores a constant number of elements (three for the largest and two for the smallest), the space complexity is `O(1)` as it does not scale with the size of the input.