1464. Maximum Product of Two Elements in an Array

Easy | Array | Sorting | Heap (Priority Queue)

Problem Description

In this LeetCode problem, we are given an array of integers called `nums`. Our goal is to select two different indices `i` and `j` within this array. We are then asked to calculate the product of the values at these indices, decreased by 1. Specifically, we need to find `(nums[i] - 1) * (nums[j] - 1)` that results in the maximum possible value. To clarify, since `i` and `j` must be different, the elements `nums[i]` and `nums[j]` must be distinct elements—even if they have the same value.

Intuition

The key intuition here is that in order to maximize the product `(nums[i] - 1) * (nums[j] - 1)`, we need to identify the two largest numbers in the array `nums`. This is because the product of any other pair would be less than or equal to the product of the two largest numbers.

The thought process starts with initializing two variables, which will hold the two largest numbers found while iterating through the array. As we go through each number in the array:

- If we find a number greater than the largest number we've found so far (`a`), then we shift the previous largest number to be the second-largest (`b`) and update the largest number (`a`) with the new number.
- If the current number is not larger than `a` but is larger than `b`, we just update the second-largest number (`b`).

Once we have the two largest numbers, we subtract 1 from each (as per the problem statement) and return their product. By following this approach, we do not need to worry about which indices we chose; we only need the values to compute the desired maximum product.

Solution Approach

The solution uses a straightforward linear scan algorithm to iterate through the array of numbers. We use two variables, `a` and `b`, to keep track of the largest and second-largest numbers in the array, respectively. There's no need for any additional data structure as we only need to maintain these two variables through the iteration.

Here's a step-by-step walkthrough of the implementation:

1. Initialize two variables, `a` and `b`, both set to 0. These will be used to track the largest (`a`) and second largest (`b`) numbers in the array.
2. Loop through each value `v` in the array `nums`.
3. Check if the current value `v` is greater than our current largest value `a`.
   - If `v` is larger than `a`, then we need to update both `a` and `b`. Set `b` to the old value of `a` (because `a` is going to be replaced with a larger value, and thus the previous `a` becomes the second-largest), and `a` to `v`.
4. If `v` is not larger than `a` but is larger than `b`, then just update `b` to be `v`, because we found a new second-largest number.
5. Once the loop is over, we have identified the two largest values in the array. We calculate the result as `(a - 1) * (b - 1)` and return it.
6. Return the computed product as the solution.

In terms of complexity:

- Time Complexity is `O(n)` because we go through the array of numbers exactly once.
- Space Complexity is `O(1)` as we are using a fixed amount of space regardless of the input array size.

The elegance of this approach lies in its simplicity and efficiency, as there is no need for sorting or additional data structures like heaps or trees, which would increase the complexity of the solution.

Example Walkthrough

Consider an example array `nums` given as `[3, 4, 5, 2]`.

Following the solution approach:

1. Initialize two variables `a` and `b` to `0`. So initially, `a = 0` and `b = 0`.
2. Loop through each value `v` in the array `nums`.
3. Start with the first value `3`:
   - Is `3` greater than `a` (which is `0`)? Yes.
   - Update `b` to the current `a`, so now `b = 0`.
   - Update `a` to the current value `v`, now `a = 3`.
4. Move to the second value `4`:
   - Is `4` greater than `a` (which is `3`)? Yes.
   - Update `b` to the current `a`, so now `b = 3`.
   - Update `a` to the current value `v`, now `a = 4`.
5. Now, look at the third value `5`:
   - Is `5` greater than `a` (which is `4`)? Yes.
   - Update `b` to the current `a`, so `b = 4`.
   - Update `a` to `5`, the current value `v`.
6. Finally, consider the last value `2`:
   - Is `2` greater than `a` (which is `5`)? No.
   - Is `2` greater than `b` (which is `4`)? No.
   - So no updates to `a` or `b` occur because `2` is neither the largest nor the second-largest number found so far.
7. After the loop, our two largest values have been found: `a = 5` and `b = 4`.
8. Calculate the result as `(a - 1) * (b - 1)`, which is `(5 - 1) * (4 - 1) = 4 * 3 = 12`.
9. Return the product `12` as the solution.

In this example, the selected values are `5` (at index `2`) and `4` (at index `1`). Subtracting `1` from each and then multiplying, we get the maximum possible product value `12` as the output for this input array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Initialize the two largest numbers as 0
        max_num1 = max_num2 = 0

        # Loop through each number in the list
        for value in nums:
            # If the current value is greater than the first maximum number
            if value > max_num1:
                # Update the first and second maximum numbers
                max_num1, max_num2 = value, max_num1
            # Else if the current value is only greater than the second maximum number
            elif value > max_num2:
                # Update the second maximum number
                max_num2 = value

        # Return the product of the two highest numbers after subtracting 1 from each
        return (max_num1 - 1) * (max_num2 - 1)
```

Java

```java
class Solution {
    public int maxProduct(int[] nums) {
        // Initialize two variables to store the largest and second largest values
        // We start with the smallest possible values for integers
        int maxVal = Integer.MIN_VALUE;
        int secondMaxVal = Integer.MIN_VALUE;

        // Iterate through each value in the nums array
        for (int value : nums) {
            // Check if the current value is greater than the largest value found so far
            if (value > maxVal) {
                // If it is, the current largest becomes the second largest,
                // and the current value becomes the new largest
                secondMaxVal = maxVal;
                maxVal = value;
            } else if (value > secondMaxVal) {
                // If the current value is not larger than the largest but is larger
                // than the second largest, update the second largest
                secondMaxVal = value;
            }
        }

        // Return the product of the largest and second largest values decreased by 1
        // This is because the problem statement likely intended for a pair of values
        // whose product is maximized after each is decreased by 1
        return (maxVal - 1) * (secondMaxVal - 1);
    }
}
```

C++

```cpp
#include <vector> // Include the necessary header for vector

class Solution {
public:
    // Function to calculate the maximum product of (the max number - 1) and (the second max number - 1) in a vector
    int maxProduct(vector<int>& nums) {
        int maxNum = 0; // Initialize the maximum number to 0
        int secondMaxNum = 0; // Initialize the second maximum number to 0

        // Iterate through each number in the vector
        for (int value : nums) {
            // If current value is greater than the maximum number found so far
            if (value > maxNum) {
                secondMaxNum = maxNum; // Assign the old maximum to be the second maximum
                maxNum = value; // Update the maximum number to the current value
            }
            // Else if current value is not greater than maxNum but greater than secondMaxNum
            else if (value > secondMaxNum) {
                secondMaxNum = value; // Update the second maximum number to the current value
            }
        }

        // Calculate and return the product of (maxNum - 1) and (secondMaxNum - 1)
        return (maxNum - 1) * (secondMaxNum - 1);
    }
};
```

TypeScript

```typescript
/**
 * Finds the maximum product of (num1 - 1) * (num2 - 1) where num1 and num2
 * are the two largest numbers in the array.
 * @param nums Array of numbers.
 * @returns The maximum product.
 */
function maxProduct(nums: number[]): number {
    let firstMax = 0; // Holds the largest number found in the array
    let secondMax = 0; // Holds the second largest number found in the array

    // Iterate through each number in the provided array
    for (const num of nums) {
        if (num > firstMax) {
            // If the current number is greater than firstMax, update secondMax to firstMax
            // and then update firstMax to the current number
            secondMax = firstMax;
            firstMax = num;
        } else if (num > secondMax) {
            // If the current number is not greater than firstMax but is greater than secondMax,
            // update secondMax to the current number
            secondMax = num;
        }
    }

    // Return the product of (firstMax - 1) and (secondMax - 1)
    return (firstMax - 1) * (secondMax - 1);
}
```

```python
from typing import List

class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Initialize the two largest numbers as 0
        max_num1 = max_num2 = 0

        # Loop through each number in the list
        for value in nums:
            # If the current value is greater than the first maximum number
            if value > max_num1:
                # Update the first and second maximum numbers
                max_num1, max_num2 = value, max_num1
            # Else if the current value is only greater than the second maximum number
            elif value > max_num2:
                # Update the second maximum number
                max_num2 = value

        # Return the product of the two highest numbers after subtracting 1 from each
        return (max_num1 - 1) * (max_num2 - 1)
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input array `nums`. This is because the code includes a single for-loop that iterates over all elements in the array once to find the two largest elements.

The space complexity of the solution is `O(1)`. This is constant space because the solution only uses a fixed amount of extra space to store the largest (`a`) and the second-largest (`b`) values in the array, regardless of the input size.