2874. Maximum Value of an Ordered Triplet II

Medium Array [Leetcode Link]

Problem Description

In this problem, we are given an array of integers, `nums`, indexed starting from 0. The task is to find the *maximum value* that can be obtained from choosing any three different indices `(i, j, k)` in the array such that `i < j < k`. The value of a triplet `(i, j, k)` is calculated using the formula `(nums[i] - nums[j]) * nums[k]`.

If we consider different combinations of three elements from the array, some might yield negative values, while others might yield positive values or zero. We want to find the maximum positive value, or if all the possible triplets result in a negative value, we return `0`.

To understand the value of a triplet, let's consider that `nums[i]` is greater than `nums[j]`, then `(nums[i] - nums[j])` will be positive. In this case, to maximize the value of the triplet, you would want `nums[k]` to be as large as possible because you are multiplying by `nums[k]`. If `nums[i]` is less than `nums[j]`, the difference becomes negative, and a large `nums[k]` will only lead to a more negative value. In that case, we are not interested in those combinations as our goal is to find the maximum positive value.

Intuition

The intuition behind the solution is to traverse the array while keeping track of two key values: the maximum value found so far (`mx`), and the maximum difference between `mx` and the current value (`mx_diff`). These two variables help in efficiently computing the maximum value of the triplet without having to explicitly check all possible triplet combinations.

The maximum prefix value `mx` represents the largest number we have seen so far as we iterate through the array. This value could potentially be `nums[i]` of our triplet. As we are looking for `i < j < k`, any number we see can be a candidate for `nums[k]`.

The maximum difference `mx_diff` represents the highest value obtained from subtracting any previously encountered number from `mx`. It effectively keeps track of the maximum `(nums[i] - nums[j])` that we have seen so far.

While we iterate through the array, for any current number `num`, we calculate and update `ans` with `max(ans, mx_diff * num)`. This step calculates the maximum triplet value for the current number as `nums[k]` (since `num` is always the right-most element in the potential triplet). We ensure to first update `ans` before updating `mx_diff` because `mx_diff` needs to be the result of the prefix, not including the current number.

By following this approach, we avoid having to compare each possible triplet explicitly, which would result in a higher computational complexity. Instead, we make use of the information that is gathered while traversing through the array to keep updating our potential maximum triplet value, all in linear time which is efficient.

Solution Approach

The approach is to cleverly keep track of the necessary values as we iterate through the array, enabling us to calculate the maximum triplet value on the fly without the need to consider each triplet individually.

When we start iterating through the array `nums`, we initialize two variables `mx` and `mx_diff` with zero. `mx` will keep track of the maximum value we have encountered so far (i.e., the maximum value for `nums[i]`), and `mx_diff` will store the maximum difference computed as `(nums[i] - nums[j])` during the iteration.

Below are the steps that we perform as we traverse the array `nums` from left to right:

1. **Update the Answer**: For the current value `num` in `nums`, we attempt to update the answer, `ans`, with the maximum of either `ans` itself or the product of `mx_diff` and `num`. The reason behind this calculation is that `num` will serve as `nums[k]` (the potential third element in our triplet), and `mx_diff` represents the maximum difference from previous elements `nums[i]` and `nums[j]`. This step ensures that we always have the maximum possible product for the current state of the array.

   `ans = max(ans, mx_diff * num)`

2. **Update Maximum Value**: Next, we update `mx` to be the maximum of itself or the current value `num`, because as we move through the array, we need to keep track of the largest value seen so far that could become `nums[i]` for future triplets.

   `mx = max(mx, num)`

3. **Update Maximum Difference**: Finally, we update `mx_diff`. To maintain the invariant that `mx_diff` is the maximum difference for a valid triplet, we must ensure that it is always calculated uptil the elements before `num` (as `num` is a candidate for `nums[k]`). Hence, it is updated as the maximum of itself or `mx - num`.

   `mx_diff = max(mx_diff, mx - num)`

Since we always update `ans` before `mx_diff`, we can guarantee that the difference applied to the calculation of `ans` doesn't include the current `num` as `mx_diff` gets updated only after `ans`.

This way, as we move forward through the array, we dynamically update and maintain the necessary values to find the maximum triplet product based on the constraints given in the problem. The final answer is stored in `ans`, which by the end of the iteration of the array contains the maximum value for all the valid triplets `(i, j, k)` that we were tasked with finding.

The algorithm leverages two fundamental ideas:

- **Dynamic Updates**: Instead of considering every triplet separately, it keeps track of the potential maximums dynamically.
- **Greedy Choice Property**: By choosing the best options at each step (maximum value and maximum difference), we ensure the end result is optimal.

This concise yet effective approach and the minimal use of extra space (just two variables) contributes to an efficient solution with linear time complexity `O(n)` and constant space complexity `O(1)`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the array `nums = [1, 5, 6, 3, 7]`. We need to find the maximum value of `(nums[i] - nums[j]) * nums[k]` where `i < j < k`.

We start by initializing `mx` and `mx_diff` as `0`. The variable `ans` will be used to keep track of the maximum value we find.

1. We begin with the first element `1`. `mx` is updated to `1` (it's the first element). There are no previous elements to consider, so we move to the next element.
2. At element `5`, we update `mx` to `5` (since `5` is greater than `mx` which was `1`). We cannot update `mx_diff` yet since we do not have a `j` index.
3. Moving to element `6`, `mx` remains at `5`. `mx_diff` is updated to `0` (which was `5 - 5`, the only pair we've considered). We can now consider the difference between the current `mx` and `nums[j]` which is from previous elements. However, we do not update `ans` since we still need a `k` index.
4. At element `3`, `mx` is not updated since `3` is not greater than `5`. `mx_diff` can be updated to `2` (`mx` which is `5` minus current element `3`). The answer `ans` is now the maximum of `0` and `mx_diff * num`, which is `2 * 3 = 6`.
5. Finally, we arrive at element `7`. `mx` is not updated since `7` is not greater than current `mx` which is `5`. We then calculate `ans` as the maximum of `6` and `mx_diff * num`, which is `2 * 7 = 14`. Thus, `ans` is updated to `14`. `mx_diff` is updated to `4` (since `5` - `3` was less than `5` - `1` which is `4`).

After the iteration, `ans` holds the value `14`, which is the maximum value for the expression `(nums[i] - nums[j]) * nums[k]`.

To summarize, the steps were as follows:

- `nums[0] = 1`: Initialize `mx = 1`, `mx_diff = 0`, `ans = 0`.
- `nums[1] = 5`: Update `mx = 5`. `ans` and `mx_diff` cannot be updated as we need a `k`.
- `nums[2] = 6`: `mx_diff` can now be calculated, but remains `0`, `mx` remains `5`.
- `nums[3] = 3`: Update `mx_diff = 2`. Update `ans = max(0, 2 * 3) = 6`.
- `nums[4] = 7`: Do not update `mx` but update `ans = max(6, 2 * 7) = 14`. Update `mx_diff = max(2, 5 - 3) = 4`.

The procedure efficiently finds the maximum value by dynamically updating the `mx`, `mx_diff`, and `ans` variables while traversing the array a single time. The solution did not require examining all possible triplets, thus maintaining a linear time complexity.

Python Solution

```python
# Import typing List to specify the type of the input nums.
from typing import List

class Solution:
    def maximumTripletValue(self, nums: List[int]) -> int:
        # Initialize variables to store the maximum triplet value, the maximum number encountered so far,
        # and the maximum difference between the maximum number and the current number.
        max_triplet_value = 0
        max_number = 0
        max_difference = 0

        # Iterate through each number in the input list.
        for number in nums:
            # Calculate the maximum triplet value by taking the maximum between the current max_triplet_value
            # and the product of max_difference and the current number.
            max_triplet_value = max(max_triplet_value, max_difference * number)

            # Update max_number if the current number is greater than the max_number seen so far.
            max_number = max(max_number, number)

            # Update max_difference which is the maximum difference found between max_number and any number.
            max_difference = max(max_difference, max_number - number)

        # Return the maximum possible triplet value found.
        return max_triplet_value

```

Java Solution

```java
class Solution {

    // Method to calculate the maximum triplet value.
    public long maximumTripletValue(int[] nums) {
        // Initialize three variables to store the current maximum value,
        // the maximum difference found so far, and the answer we will return.
        long currentMax = 0;
        long maximumDiff = 0;
        long answer = 0;

        // Iterate through each number in the array.
        for (int num : nums) {
            // Calculate the tentative answer as the current number times the maximum difference,
            // update the answer if the result is greater than the current answer.
            answer = Math.max(answer, num * maximumDiff);

            // Update currentMax if the current number is greater than currentMax.
            currentMax = Math.max(currentMax, num);

            // Update maximumDiff if the difference between currentMax and current number
            // is greater than maximumDiff.
            maximumDiff = Math.max(maximumDiff, currentMax - num);
        }

        // Return the final answer.
        return answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>  // Include necessary headers for vector and max function

class Solution {
public:
    long long maximumTripletValue(std::vector<int>& nums) {
        long long maxTripletValue = 0;  // This will hold the maximum value of the triplet found so far
        int maxNum = 0;  // This will keep track of the maximum number encountered in the array
        int maxDifference = 0;  // This will keep the maximum difference we've found between maxNum and a smaller number

        // Iterate over the elements of nums
        for (int num : nums) {
            // Update the maximum value of any triplet found so far.
            // The maximum triplet value is defined by maxDifference multiplied by the current number.
            maxTripletValue = std::max(maxTripletValue, 1LL * maxDifference * num);

            // Update maxNum if the current number is greater than the previous maxNum.
            maxNum = std::max(maxNum, num);

            // Update maxDifference if the difference between the current maxNum and num is greater
            // than the previous maxDifference.
            maxDifference = std::max(maxDifference, maxNum - num);
        }

        return maxTripletValue;  // Return the maximum triplet value found.
    }
};

```

Typescript Solution

```typescript
function maximumTripletValue(nums: number[]): number {
    // Initialize variables:
    // ans - to store the maximum product of the triplet.
    // maxNum - to store the maximum number encountered so far.
    // maxDifference - to store the maximum difference encountered so far.
    let [maxProduct, maxNum, maxDifference] = [0, 0, 0];

    // Loop through each number in the array to calculate the maximum triplet value.
    for (const num of nums) {
        // Update the maxProduct with the maximum of current maxProduct and
        // the product of maxDifference and the current num.
        maxProduct = Math.max(maxProduct, maxDifference * num);

        // Update maxNum with the maximum of current maxNum and the current num.
        maxNum = Math.max(maxNum, num);

        // Update maxDifference with the maximum of current maxDifference and
        // the difference between current maxNum and the current num.
        maxDifference = Math.max(maxDifference, maxNum - num);
    }

    // Return the calculated maximum product of a triplet.
    return maxProduct;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)` because there is a single for loop that iterates through the array `nums` once. Each operation inside the loop, such as updating `ans`, `mx`, and `mx_diff`, is done in constant time, independent of the size of the input array. As the loop runs `n` times, where `n` is the length of the array, the overall time complexity is linear.

The space complexity of the code is `O(1)`. The reason for this constant space complexity is that only a fixed number of variables (`ans`, `mx`, and `mx_diff`) are used. These variables do not depend on the size of the input, hence, the amount of allocated memory stays constant regardless of the input size.