2873. Maximum Value of an Ordered Triplet I

Easy Array

Problem Description

You are provided with an array of integers, `nums`, which has a 0-based index. The goal is to find the maximum value of a triplet `(i, j, k)` in this array, following the condition that `i`, `j`, and `k` are distinct indices, with `i` being less than `j`, and `j` being less than `k`. In other words, you need to find the maximum value obtained from the formula `(nums[i] - nums[j]) * nums[k]` across all possible combinations of triplets.

In a case where the values for all possible triplets are negative, the result should be `0`.

The problem asks for the efficient computation of this value without having to compare every possible triplet directly, which would be inefficient for large arrays.

Intuition

To avoid an exhaustive search which is highly inefficient, we observe that we can solve the problem by keeping track of two pieces of information as we iterate through the array from left to right.

- `mx`: This is the maximum value found in `nums` up to the current point in our traversal. We can think of it as the 'best' first element `nums[i]` seen so far for any triplet.
- `mx_diff`: This represents the maximum value of `mx - nums[j]`, which is the first part of our target equation `(nums[i] - nums[j])`. It essentially stores the best-case scenario for the difference between the first and second elements of our triplet encountered so far.

As we traverse the array, at each step, we attempt to update `mx_diff` with the largest possible value by considering the current `mx` and the current number `num` as if it were `nums[j]`. We also update `mx` if the current number is larger than the current `mx`. After updating `mx` and `mx_diff`, we then calculate the potential best-case scenario for the triplet value with the current number as `nums[k]`, and update the answer if it's greater than the current answer.

Solution Approach

The solution implements a single pass approach, traversing the list once, which keeps it very efficient in terms of both time and space complexity. The algorithm does not use any extra data structures, as it simply maintains two variables `mx` and `mx_diff` to track the maximum value found so far and the maximum difference encountered so far, respectively.

The approach makes use of the following steps:

- Initialize `ans`, `mx`, and `mx_diff` as zero. `ans` will hold the answer to be returned, `mx` is used to keep track of the maximum value in `nums` as we iterate, and `mx_diff` keeps track of the maximum value of `mx - nums[j]` found so far.
- Loop through each element `num` in `nums`. For each `num`:
  - Update the `ans` if the current `mx_diff * num` is greater than `ans`. This step checks if the current number as `nums[k]` combined with the best `mx_diff` so far makes for a higher product than we've seen.
  - Update `mx` if `num` is greater than `mx`. This step simply keeps `mx` as the maximum value seen up to the current element in the array, representing the best choice for `nums[i]` up to this point.
  - Update `mx_diff` if `mx - num` is greater than `mx_diff`. By doing this, you are ensuring that `mx_diff` holds the largest possible difference between the best `nums[i]` and any `nums[j]` seen so far.

During each iteration, the algorithm dynamically updates the potential first two elements of the triplet, which allows it to calculate the potential best-case scenario for the triplet value in constant time.

By maintaining the maximum found value and the maximum difference during the iteration, the algorithm eliminates the need to check every possible combination of `i`, `j`, and `k`, which is the key to its efficiency.

Here is the pseudocode that captures the essence of the implementation:

```
def maximumTripletValue(nums):
    ans = mx = mx_diff = 0
    for num in nums:
        ans = max(ans, mx_diff * num)
        mx = max(mx, num)
        mx_diff = max(mx_diff, mx - num)
    return ans
```

This algorithm runs in $O(n)$ time, where n is the length of the input array, making it extremely efficient for large datasets.

Example Walkthrough

Let's consider a small example array `nums` to illustrate the solution approach.

Example `nums` array: `[3, 1, 6, 4]`

We have to find the maximum value of the expression `(nums[i] - nums[j]) * nums[k]` with the constraints `i < j < k`.

We initialize `ans`, `mx`, and `mx_diff` to `0`.

1. We start with the first element `3`:
   - `mx` is updated to `3` because it's the only element we've seen.
   - `mx_diff` remains `0` because we don't have a `j` yet.
   - `ans` remains `0` for the same reason.

   After the first iteration: `ans = 0`, `mx = 3`, `mx_diff = 0`.
2. Moving to the second element `1`:
   - We consider the element `1` as potential `nums[j]`. The difference `mx - num` is `3 - 1 = 2`.
   - `mx_diff` is updated to `2` because `2` is greater than the current `mx_diff` which is `0`.
   - `ans` is still `0` as we have not yet encountered a valid `k`.

   After the second iteration: `ans = 0`, `mx = 3`, `mx_diff = 2`.
3. Next, we process the third element `6`:
   - This element is considered as potential `nums[k]`. We compute `mx_diff * num` which is `2 * 6 = 12`.
   - `ans` is updated to `12` because `12` is greater than the current `ans` which is `0`.
   - Now we update `mx` to `6` because `6` is greater than the current `mx` which is `3`.
   - `mx_diff` does not change because `mx - num` is `-3` which is not greater than `2`.

   After the third iteration: `ans = 12`, `mx = 6`, `mx_diff = 2`.
4. Finally, we look at the fourth element `4`:
   - We calculate `mx_diff * num` which is `2 * 4 = 8`. However, `ans` remains `12` because `8` is not greater than `12`.
   - `mx` does not change because `4` is not greater than `6`.
   - `mx_diff` is updated, as `mx - num` is `6 - 4 = 2`, but since `mx_diff` is already `2`, it remains the same.

   After the fourth and final iteration: `ans = 12`, `mx = 6`, `mx_diff = 2`.

At the end of the iterations, the maximum value found for the expression `(nums[i] - nums[j]) * nums[k]` is `12`, which is the final answer. The triplet that gives us this value is `(3, 1, 6)` where `i = 0`, `j = 1`, and `k = 2`. Therefore, the function `maximumTripletValue` with the array `[3, 1, 6, 4]` as input will return `12`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def maximumTripletValue(self, nums: List[int]) -> int:
        # Initialize variables:
        # max_product - to keep track of the maximum product of max difference and the current number,
        # max_number - to store the maximum value encountered so far,
        # max_difference - to store the maximum difference between max_number and any other number.
        max_product = 0
        max_number = 0
        max_difference = 0

        # Iterate through all numbers in the list.
        for num in nums:
            # Update max product with the maximum product obtained
            # by multiplying max difference with the current num.
            max_product = max(max_product, max_difference * num)

            # Update max number if the current num is greater than the previously stored max_number.
            max_number = max(max_number, num)

            # Update max difference if the difference between the current max_number and num
            # is greater than the previously stored max difference.
            # This difference represents a potential first and second element of a triplet,
            # with num potentially being the third element.
            max_difference = max(max_difference, max_number - num)

        # After iterating through all numbers, max product will hold
        # the maximum product of a triplet's first and third elements.
        return max_product
```

Java

```java
class Solution {
    public long maximumTripletProduct(int[] nums) {
        long maxVal = 0;           // Initialize maximum value found in the array to 0
        long maxDiff = 0;          // Initialize maximum difference between maxVal and any other value to 0
        long answer = 0;           // Initialize the result for the maximum product of the triplet

        // Iterate through all elements in the nums array
        for (int num : nums) {
            // Update the answer with the maximum between the current max product
            // or the product of the current number and maxDiff
            answer = Math.max(answer, num * maxDiff);

            // Update maxVal with the maximum between the current maxVal or the current number
            maxVal = Math.max(maxVal, num);

            // Update maxDiff with the maximum difference found so far
            maxDiff = Math.max(maxDiff, maxVal - num);
        }

        // Return the maximum product of a triplet found in the array
        return answer;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Function to calculate the maximum product of a triplet in the array such that
    // the indices of the triplet (i, j, k) satisfy i < j < k and nums[i] < nums[j] < nums[k].
    long long maximumTripletValue(vector<int>& nums) {
        long long maxProduct = 0;           // To store the maximum product of a triplet
        int maxElement = 0;                 // To store the maximum element seen so far
        int maxDifference = 0;              // To store the maximum difference seen so far

        // Loop through each number in the array 'nums'
        for (int num : nums) {
            // Update maxProduct with the maximum between current maxProduct and the product
            // of maxDifference and the current number 'num'. This accounts for the third element of the triplet.
            maxProduct = max(maxProduct, static_cast<long long>(maxDifference) * num);

            // Update maxElement with the maximum between current maxElement and the current number 'num'
            maxElement = max(maxElement, num);

            // Update maxDifference with the maximum difference between maxElement and the current number 'num'
            maxDifference = max(maxDifference, maxElement - num);
        }

        // Return the maximum product of a triplet found in the array
        return maxProduct;
    }
};
```

TypeScript

```typescript
// Calculates the maximum product of any triplet in the given array
// that can be formed by multiplying any three numbers which indices
// are strictly in increasing order.
function maximumTripletValue(nums: number[]): number {
    let maxProduct = 0; // Variable to store the maximum product found
    let maxNum = 0; // Variable to store the maximum number found so far
    let maxDifference = 0; // Variable to store the maximum difference found so far

    // Iterate through the array of numbers
    for (const num of nums) {
        // Update the maxProduct with the maximum between the current maxProduct and
        // the product of num and maxDifference which represents a potential triplet product
        maxProduct = Math.max(maxProduct, maxDifference * num);
        // Update the maxNum with the greatest number encountered so far
        maxNum = Math.max(maxNum, num);
        // Update the maxDifference with the greatest difference between maxNum and the current num
        maxDifference = Math.max(maxDifference, maxNum - num);
    }

    // Return the maximum product found for the triplet
    return maxProduct;
}
```

```python
from typing import List

class Solution:
    def maximumTripletValue(self, nums: List[int]) -> int:
        # Initialize variables:
        # max_product - to keep track of the maximum product of max difference and the current number,
        # max_number - to store the maximum value encountered so far,
        # max_difference - to store the maximum difference between max_number and any other number.
        max_product = 0
        max_number = 0
        max_difference = 0

        # Iterate through all numbers in the list.
        for num in nums:
            # Update max product with the maximum product obtained
            # by multiplying max difference with the current num.
            max_product = max(max_product, max_difference * num)

            # Update max number if the current num is greater than the previously stored max_number.
            max_number = max(max_number, num)

            # Update max difference if the difference between the current max_number and num
            # is greater than the previously stored max difference.
            # This difference represents a potential first and second element of a triplet,
            # with num potentially being the third element.
            max_difference = max(max_difference, max_number - num)

        # After iterating through all numbers, max product will hold
        # the maximum product of a triplet's first and third elements.
        return max_product
```

Time and Space Complexity

The time complexity of the given code segment is $O(n)$, where n is the length of the array `nums`. This is because there is a single for-loop that iterates over all the elements in the array once.

The space complexity is $O(1)$ since the extra space used does not grow with the input size; only a fixed number of variables `ans`, `mx`, and `mx_diff` are used regardless of the size of the input array.