152. Maximum Product Subarray

Medium Array Dynamic Programming

Problem Description

The given LeetCode problem is about finding the subarray within an integer array `nums` that yields the highest product. A subarray is essentially a contiguous part of an array. The key point here is that we are not just looking for the maximum element but the contiguous sequence of elements that when multiplied together give the maximum product. This problem is challenging because the array may contain negative numbers, and a product of two negatives yields a positive, which could potentially be part of the maximum product subarray. The solution must also take into account that the final product has to fit in a 32-bit integer, which sets a defined range of integer values.

Intuition

The intution behind the solution is to keep track of both the maximum and minimum product at each position in the array as we iterate through it. This is important because a new maximum product can emerge from the multiplication of the current element with the previous minimum product if the current element is a negative number.

To arrive at the solution approach, we need to think about how the sign of an integer (positive or negative) affects the product. The dynamic nature of this problem comes from the fact that including a number in a product might change the maximum or minimum product because of multiplication rules (a positive multiplied by a negative yields a negative, and a negative multiplied by a negative yields a positive).

Therefore, at each step, we have three choices to make for both the maximum and minimum products:

1. Take the current number itself (which might be beneficial if the current maximum/minimum product held before is less/more beneficial than starting afresh from the current number).
2. Multiply the current number by the previous maximum product (as usual, to maintain a running product).
3. Multiply the current number by the previous minimum product (particularly useful when multiplying by a negative number could yield a larger product).

We maintain two variables `f` and `g` to keep track of the running 'maximum' and 'minimum' products, respectively, as we iterate through the array. For each new element x, we calculate the new maximum `f` by choosing the maximum between x, $f * x$, and $g * x$. Analogously, we calculate the new minimum `g` by choosing the minimum between x, $f * x$, and $g * x$. Meanwhile, we also maintain a running maximum `ans` to store the maximum product found so far.

By keeping track of both the maximum and minimum products at each step, we are able to handle the intricacies introduced by negative numbers, ensuring that we can always capture the highest product subarray even when it is formed by turning a negative maximum product positive with another negative number.

Solution Approach

The solution uses a simple but clever approach leveraging dynamic programming. Instead of keeping a single running product, it keeps track of two products at each iteration - the maximum product and the minimum product up to that point in the array. The algorithm uses three variables:

- `ans` to store the global maximum product of any subarray encountered so far. It is initialized with the first element of the array since the maximum product subarray might just be the first element if all other elements are non-positive.
- `f` to store the maximum product of a subarray ending at the current element. It's like a "local maximum" that is updated at each step.
- `g` to store the minimum product of a subarray ending at the current element. This "local minimum" is crucial as it can turn into the "local maximum" if the current element and the `g` value are both negative.

The algorithm then iterates through the array starting at the second element (since the first element is already used to initialize the variables). For each element x in `nums` starting from index 1:

1. It temporarily stores the previous values of `f` and `g` in variables `ff` and `gg` before they get updated. This is because the new `f` and `g` will depend on the previous values of both `f` and `g`, and the calculation of one should not affect the other.
2. The algorithm updates `f` by choosing the maximum value among three candidates:
   - The current element x itself. This is because the maximum product could start fresh from the current position.
   - The product of the current element x and the previous `f`, which is the previous maximum product ending at the last element.
   - The product of the current element x and the previous `g`, because when x is negative and `g` is also negative, their product will be positive and can potentially be larger than $f * x$.
3. It updates `g` in a similar manner but chooses the minimum among the same three candidates mentioned above:
   - The current element x.
   - The product of x and the previous `f`.
   - The product of x and the previous `g`.
4. It then updates `ans` with the maximum value between `ans` and the new `f`. The loop will ensure that by the end, `ans` contains the highest product of any subarray within `nums`.

By maintaining both `f` and `g` at each step, the current element has the flexibility to contribute to either a new maximum or minimum subarray product depending on its value and the current status of `f` and `g`. This dynamic behavior allows the solution to effectively handle sequences with negative numbers and track the maximum subarray product accurately.

The reference solution uses this approach efficiently, leading to a time complexity of O(n) since it goes through the array only once, and a space complexity of O(1) since it only uses a constant amount of extra space.

Example Walkthrough

Let's take a small array to illustrate how the solution approach works. Consider `nums = [2, 3, -2, 4]`.

1. Initialize `ans`, `f`, and `g` with the first element of the array:
   - `ans = 2`
   - `f = 2`
   - `g = 2`
2. Iterate from the second element to the end of the array, updating `f`, `g`, and `ans` at each step.
3. At index 1 (`nums[1] = 3`):
   - Store previous `f` and `g` in `ff` and `gg`:
     - `ff = 2`
     - `gg = 2`
   - Calculate new `f`: max(3, 2 * 3, 2 * 3) = max(3, 6, 6) = 6
   - Calculate new `g`: min(3, 2 * 3, 2 * 3) = min(3, 6, 6) = 3
   - Update `ans`: max(`ans`, `f`) = max(2, 6) = 6
4. At index 2 (`nums[2] = -2`):
   - Store previous `f` and `g` in `ff` and `gg`:
     - `ff = 6`
     - `gg = 3`
   - Calculate new `f`: max(-2, 6 * -2, 3 * -2) = max(-2, -12, -6) = -2
   - Calculate new `g`: min(-2, 6 * -2, 3 * -2) = min(-2, -12, -6) = -12
   - Update `ans`: max(`ans`, `f`) = max(6, -2) = 6
5. At index 3 (`nums[3] = 4`):
   - Store previous `f` and `g` in `ff` and `gg`:
     - `ff = -2`
     - `gg = -12`
   - Calculate new `f`: max(4, -2 * 4, -12 * 4) = max(4, -8, 48) = 48
   - Calculate new `g`: min(4, -2 * 4, -12 * 4) = min(4, -8, 48) = -8
   - Update `ans`: max(`ans`, `f`) = max(6, 48) = 48

After iterating through the entire array, our `ans` is 48, which is the highest product of a subarray in `nums`. This subarray is actually `[2, 3, -2, 4]` itself, and the maximum product is 2 * 3 * -2 * 4 = 48.

By tracking both the maximum (`f`) and minimum (`g`) products at each step and updating the global maximum (`ans`), we arrive at the correct solution efficiently.

Python Solution

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Initialize the maximum product, current maximum, and current minimum.
        max_product = current_max = current_min = nums[0]

        # Iterate over the numbers starting from the second element.
        for num in nums[1:]:
            # Save the previous step's maximum and minimum.
            temp_max, temp_min = current_max, current_min

            # Calculate the current maximum product ending at the current number.
            # It's the maximum of the current number, product of the current number and previous maximum,
            # or the product of the current number and previous minimum.
            current_max = max(num, temp_max * num, temp_min * num)

            # Calculate the current minimum product ending at the current number.
            # It's the minimum for the same reason above but will be used to handle negative numbers.
            current_min = min(num, temp_max * num, temp_min * num)

            # Update the overall maximum product found so far.
            max_product = max(max_product, current_max)

        # Return the maximum product of the array.
        return max_product
```

Java Solution

```java
class Solution {
    public int maxProduct(int[] nums) {
        // Initialize the maximum, minimum, and answer with the first element.
        // The max value `maxProduct` represents the largest product found so far and
        // could be the maximum product of a subarray ending at the current element.
        // The min value `minProduct` represents the smallest product found so far and
        // could be the minimum product of a subarray ending at the current element.
        // This is required because a negative number could turn the smallest value into
        // the largest when multiplied by another negative number.
        int maxProduct = nums[0];
        int minProduct = nums[0];
        int answer = nums[0];

        // Iterate through the array starting from the second element.
        for (int i = 1; i < nums.length; ++i) {
            // Store the current max and min before updating them.
            int currentMax = maxProduct;
            int currentMin = minProduct;

            // Update the maxProduct to be the maximum between the current number,
            // currentMax multiplied by the current number, and currentMin multiplied
            // by the current number. This accounts for both positive and negative numbers.
            maxProduct = Math.max(nums[i], Math.max(currentMax * nums[i], currentMin * nums[i]));

            // Update the minProduct similarly by choosing the minimum value.
            minProduct = Math.min(nums[i], Math.min(currentMax * nums[i], currentMin * nums[i]));

            // Update the answer if the newly found maxProduct is greater than the previous answer.
            answer = Math.max(answer, maxProduct);
        }

        // Return the largest product of any subarray found.
        return answer;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for max() and min() functions

class Solution {
public:
    // Function to calculate the maximum product subarray
    int maxProduct(vector<int>& nums) {
        // Initialize variables: maxEndHere for the current maximum product,
        // minEndHere for the current minimum product, and
        // maxProductOverall for the overall maximum product found.
        int maxEndHere = nums[0], minEndHere = nums[0], maxProductOverall = nums[0];

        // Iterate through the vector starting from the second element
        for (int i = 1; i < nums.size(); ++i) {
            // Store the previous values of maxEndHere and minEndHere
            int tempMaxEndHere = maxEndHere, tempMinEndHere = minEndHere;

            // Calculate the new maxEndHere by considering the current number itself,
            // the product of the current number with the previous maxEndHere, and
            // the product of the current number with the previous minEndHere.
            // This accounts for the possibility of a negative number times
            // a negative minEndHere becoming a maximum.
            maxEndHere = max(nums[i], tempMaxEndHere * nums[i], tempMinEndHere * nums[i]);

            // Similarly, calculate the new minEndHere
            minEndHere = min(nums[i], tempMaxEndHere * nums[i], tempMinEndHere * nums[i]);

            // Update the maxProductOverall with the maximum value found so far
            maxProductOverall = max(maxProductOverall, maxEndHere);
        }
        // Return the maximum product subarray found
        return maxProductOverall;
    }
};
```

Typescript Solution

```typescript
function maxProduct(nums: number[]): number {
    // Initialize the max product, min product, and answer all to the first element
    // as the array since we will compare these to every other element moving forward.
    let maxProd = nums[0];
    let minProd = nums[0];
    let answer = nums[0];

    // Loop through the array starting from the second element
    for (let i = 1; i < nums.length; ++i) {
        // Keep temporary copies of the current max and min products before they get updated.
        const tempMaxProd = maxProd;
        const tempMinProd = minProd;

        // Compute the new maxProd by considering the current element,
        // the product of the current element and the previous maxProd,
        // and the product of the current element and the previous minProd.
        // The maxProd can either include the current element by itself
        // or include a subarray ending at the current element.
        maxProd = Math.max(nums[i], tempMaxProd * nums[i], tempMinProd * nums[i]);

        // Compute the new minProd using similar logic to maxProd.
        // The minProd accounts for negative numbers which could be useful
        // if the current element is negative (a negative times a negative is positive).
        minProd = Math.min(nums[i], tempMaxProd * nums[i], tempMinProd * nums[i]);

        // Update answer with the larger value between the current answer and the new maxProd.
        // This could be the max product of a subarray up to the current index.
        answer = Math.max(answer, maxProd);
    }

    // Return the maximum product of any subarray found in the nums array.
    return answer;
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the number of elements in the input list `nums`. This is because it processes each element exactly once in a single loop.

The space complexity of the code is `O(1)` because it uses a constant amount of additional space. The variables `ans`, `f`, `g`, `ff`, and `gg` do not depend on the size of the input, so the space used does not scale with the size of the input list.