2495. Number of Subarrays Having Even Product

Medium Array Math Dynamic Programming

Leetcode Link

Problem Description

We are tasked with finding the number of subarrays from a given array `nums`, where the subarrays have a product that is an even number. A subarray is defined as a contiguous part of the original array. Since the array is 0-indexed, the first element is at position 0. Remember, a single element is also considered a subarray, so if any element in the array is even, that alone counts as one subarray with an even product. The final output should be the total count of such subarrays.

Intuition

To solve this problem, we can use the fact that the product of integers is even if and only if at least one of the integers is even. With this in mind, we scan the array and keep track of the last position `last` where we encountered an even number. When we find an even number, we know that all subarrays that end at that position and start before or at the last found even number's position will have an even product.

For every element `v` at index `i`, if `v` is odd, the number of subarrays ending at `i` with an even product is the same as the number of subarrays ending at `i-1`. If `v` is even, then all subarrays ending at `i` have an even product. We add 1 to account for the current element, which is a subarray itself if it's even. We keep on adding this quantity to a cumulative answer variable `ans`.

This approach avoids the need to check the product of each subarray individually, which would not be efficient for large arrays, and instead uses the properties of even/odd numbers to count the subarrays in a linear pass through the array.

Solution Approach

The solution is implemented in Python using a single pass through the array. The algorithm can be broken down into the following steps:

1. Initialize two variables, `ans` and `last`. `ans` will hold the final result — the number of subarrays with an even product. `last` will store the index of the most recently found even number in the array. Initially, `last` is set to -1 to indicate that we haven't encountered any even numbers yet.
2. Iterate through `nums` using a loop, where `i` is the current index, and `v` is the value of the current element:
   - Inside the loop, check if the current element `v` is even (using `v % 2 == 0`). If it is,
     - Update `last` to the current index `i` because we now have a new last index where the subarray with an even product can start.
   - Regardless of whether the current element is even or odd, update `ans` by adding `last + 1` to it. The `last + 1` represents the number of subarrays ending at `i` that include an even number and thus have an even product. This works because:
     - If `v` is even, all subarrays ending at `i` will have an even product. The number of such subarrays is `i + 1` (since arrays are 0-indexed).
     - If `v` is odd, the subarrays that have an even product are those which start before or at the last even number found. That's why we increment `ans` by `last + 1` instead of `i + 1`.
3. Return the value of `ans` once the loop is completed.

The algorithm relies on simple arithmetic and control structures and does not use any specific data structure or complex pattern. It merely exploits the mathematical properties of multiplication (an even number times any other number is always even) to count the subarrays efficiently.

This approach does not explicitly construct each subarray; instead, it calculates the number of valid subarrays ending at a certain point based on knowledge of earlier components of the array. This is why the algorithm runs in O(n) time, where n is the length of the input array, because it processes each element of the array only once.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the following input array:

```
nums = [3, 2, 5, 6, 7]
```

Here is a step-by-step explanation of how we would apply the algorithm:

1. We start by initializing `ans` to 0 and `last` to -1 because we haven't encountered any even numbers yet.
2. We begin iterating through `nums`:
   - **Index 0**: The value `3` is odd. We do not update `last`, but we still update `ans`. Since `last` is -1, `ans += (last + 1)` results in `ans = 0`, indicating no subarrays ending here have an even product.
   - **Index 1**: The value `2` is even, so we update `last` to the current index (`last = 1`). For updating `ans`, we add `last + 1` to it, resulting in `ans += (1 + 1)` which gives us `ans = 2`. This accounts for two subarrays `[2]` and `[3, 2]`.
   - **Index 2**: The value `5` is odd, so `last` remains at `1`. We update `ans` by adding `last + 1` to it, resulting in `ans += (1 + 1)` which gives us `ans = 4`. The subarrays `[2, 5]` and `[3, 2, 5]` are the new subarrays added to the count.
   - **Index 3**: The value `6` is even, so we update `last` to the current index (`last = 3`). We now add `last + 1` to `ans`, resulting in `ans += (3 + 1)` which gives us `ans = 8`. The new subarrays considered here are `[6]`, `[5, 6]`, `[2, 5, 6]`, and `[3, 2, 5, 6]`.
   - **Index 4**: The value `7` is odd, so `last` does not change and remains at `3`. We update `ans` once again by adding `last + 1` to it, resulting in `ans += (3 + 1)` which brings us to `ans = 12`. The new subarrays considered are the extensions of previous subarrays ending with `7`.
3. Once the loop is completed, we have our final answer. The value of `ans` is `12`, so there are 12 subarrays with an even product within the array `[3, 2, 5, 6, 7]`.

In this example, each step either adds a new set of subarrays ending with an even number or extends the existing subarrays with an odd number, while keeping track of the total count with the `ans` variable. This linear algorithm is efficient and takes into account the characteristics of even and odd products to dynamically calculate the total amount without the need to enumerate each possible subarray explicitly.

Python Solution

```python
from typing import List

class Solution:
    def evenProduct(self, nums: List[int]) -> int:
        # Initialize sum_even_products to accumulate the sum of the counts
        # and last_even_index to keep track of the last seen even number index
        sum_even_products, last_even_index = 0, -1

        # Iterate over the list of numbers with index and value
        for index, value in enumerate(nums):
            # Check if the current value is even
            if value % 2 == 0:
                # Update the last_even_index with the current index
                last_even_index = index
            # Add to the sum_even_products the count of subarrays ending with an even product
            # This is found by adding 1 to the last_even_index because if last_even_index is -1,
            # it means no even number has been found yet, so we add 0 (-1 + 1) to the sum.
            sum_even_products += last_even_index + 1

        # Return the accumulated sum of counts of subarrays with an even product
        return sum_even_products
```

Java Solution

```java
class Solution {
    public long evenProduct(int[] numbers) {
        long answer = 0; // Initialize the answer as 0 which will hold the final result
        int lastEvenIndex = -1; // Initialize the last even index to -1 indicating no even number encountered yet

        // Iterate over the array
        for (int i = 0; i < numbers.length; ++i) {
            // Check if the current element is even
            if (numbers[i] % 2 == 0) {
                lastEvenIndex = i; // Update the index of the last seen even number
            }
            // Add the index of the last seen even number plus one to the answer
            // (if no even number has been encountered, this adds 0)
            answer += lastEvenIndex + 1;
        }

        return answer; // Return the computed sum
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the sum of indices (plus one) of all even numbers encountered so far in the array.
    long long evenProduct(std::vector<int>& nums) {
        long long total = 0; // Initialize the sum as a long long to handle large numbers.
        int lastIndex = -1; // Keep track of the last encountered even number's index. Start with -1 since we haven't encountered any

        // Loop through each number in the vector.
        for (int i = 0; i < nums.size(); ++i) {
            // Check if the current number is even.
            if (nums[i] % 2 == 0) {
                // Update the lastIndex with the current index i if the number is even.
                lastIndex = i;
            }
            // Add one to the lastIndex since we need to accumulate indices starting from 1.
            // When lastIndex is -1, it implies that no even numbers have been found up to the current index,
            // so it adds 0 (lastIndex + 1) to the total.
            total += lastIndex + 1;
        }
        // Return the accumulated sum.
        return total;
    }
};
```

Typescript Solution

```typescript
// A function to calculate the sum of indices (plus one) of all even numbers encountered so far in the array.
function evenProduct(nums: number[]): number {
    let total: number = 0; // Initialize the sum to handle large numbers.
    let lastEvenIndex: number = -1; // Keep track of the last encountered even number's index. Start with -1 since we haven't encount

    // Loop through each number in the array.
    for (let i = 0; i < nums.length; i++) {
        // Check if the current number is even.
        if (nums[i] % 2 === 0) {
            // Update the lastEvenIndex with the current index i if the number is even.
            lastEvenIndex = i;
        }
        // Add one to the lastEvenIndex. If lastEvenIndex is -1, this adds 0 to the total.
        total += lastEvenIndex + 1;
    }
    // Return the accumulated sum.
    return total;
}

// Example usage:
// let arr = [1, 2, 3, 4];
// let sum = evenProduct(arr);
// console.log(sum); // Output will depend on the array provided
```

Time and Space Complexity

The given Python code defines a method `evenProduct` that calculates the sum of the indices of the last even number seen in a list of integers up to each element in the list.

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. This is because the code contains a single loop that iterates through the list `nums`. During each iteration, the code performs a constant-time check to determine whether an element is even (i.e., `v % 2 == 0`), and it updates two variables, `last` and `ans`, with simple arithmetic operations, which also take constant time.

Space Complexity

The space complexity of the code is `O(1)` because the extra space used by the algorithm is constant and does not depend on the size of the input list. The variables `ans` and `last` are used for storing the running total and the index of the last encountered even number, respectively, and their space usage does not scale with the input size.