2683. Neighboring Bitwise XOR

Medium Bit Manipulation Array

Problem Description

In this problem, you are given an array `derived`, which is said to be created from another binary array `original` by applying a bitwise XOR operation. The array `original` is a binary array, meaning it only contains `0`'s and `1`'s. The elements of the array `derived` are formed as follows:

- Each element at index `i` in `derived` is the result of `original[i]` XOR `original[i + 1]`, except for the last element.
- The last element in the `derived` array is the result of `original[n - 1]` XOR `original[0]`, creating a circular calculation from the end of the array back to the start.

Your task is to determine if there exists any valid `original` binary array that could have been used to obtain the given `derived` array through the described process.

Intuition

To solve this problem, we utilize the property of XOR operation. The fundamental point to notice is that XOR of a number with itself is zero, and the XOR of zero with any number is the number itself. With these properties in mind, let's consider the provided `derived` array and think about what happens if we take XOR of all its elements.

1. If we XOR all elements of the hypothetical `original` array in a circular manner as described, we would end up with zero. This is because each element would be XORed with itself at some point in the process (since `original[i]` XOR `original[i]` is always 0).
2. Conversely, if we XOR all elements of the provided `derived` array, and the result is non-zero, this implies there is no such `original` array that could have produced `derived`, as this would violate the property stated in step 1.
3. Hence, if the cumulative XOR of all the elements of `derived` is zero, a valid `original` array could exist as it indicates that each number has been XORed with itself.
4. The provided solution uses Python's `reduce` function and `xor` operator from the `operator` module to apply the XOR operation cumulatively across all the elements of the `derived` array. It checks whether the final result of the cumulative XOR is equal to zero.

Solution Approach

The solution approach is surprisingly straightforward due to the properties of the XOR operation. The algorithm doesn't require any additional data structures, complex patterns, or multiple iterations over the data; it relies purely on a single pass over the array to reduce it to one value. Here's an explanation of the code:

- The `reduce` function in Python is a tool from the `functools` module that is used to apply a particular function cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value. In this scenario, it is being used to apply the `xor` operation to the elements of the `derived` array.
- The `xor` operation is a bitwise operation that is found in the `operator` module. This operation takes two numbers and returns their bitwise XOR. The XOR of two bits is `1` if the bits are different, and `0` if they are the same.
- The implementation `reduce(xor, derived)` continuously applies the XOR operation across all elements of the `derived` array. As Python processes each element, it calculates the cumulative XOR from the start of the array up to the current element. This process results in a single integer value, which represents the XOR of the entire array.
- Once the cumulative XOR is computed, we compare it with `0`. If it is equal to zero (`reduce(xor, derived) == 0`), it means a valid `original` array can exist based on the properties of XOR discussed earlier. Otherwise, if the cumulative XOR is not zero, no such valid `original` array exists that could produce the `derived` array.
- The decision is made in a single line of code, thanks to the efficiency of the `reduce` function and the `xor` operator. It elegantly verifies the possibility of the existence of a valid `original` array without explicitly reconstructing it, which makes this solution both efficient and clever.

Here is the full implementation in Python:

```
from functools import reduce
from operator import xor

class Solution:
    def doesValidArrayExist(self, derived: List[int]) -> bool:
        return reduce(xor, derived) == 0
```

This implementation utilizes functional programming concepts in Python and showcases how a combination of mathematics and efficient use of built-in functions can lead to optimal and elegant solutions.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have a derived array given as:

```
derived = [1, 0, 1]
```

We want to determine if there is an original binary array that XORs to the given derived array. To do this, we can use the XOR properties to our advantage:

1. We start by XOR-ing all the elements of the derived array:
   - Step 1: XOR 1 and 0 which gives us 1.
   - Step 2: XOR the result from step 1 with the next element, which is 1, so we get 1 XOR 1 = 0.
2. After XOR-ing all the elements of derived array, we end up with 0, which confirms that there might be a valid original binary array, as the cumulative XOR equals zero.

Following the steps of the proposed solution:

- We apply `reduce` with `xor` from the `operator` module to all items of the derived array:

```
from functools import reduce
from operator import xor

result = reduce(xor, [1, 0, 1]) # This will be 0
```

- The result from the reduce function would give us 0 which complies with our intuition that a valid `original` binary array exists.

3. Since the result is equal to 0, our function doesValidArrayExist([1, 0, 1]) returns True, indicating that there is a possibility that an `original` binary array could exist to arrive at this `derived` array.

```
derived = [1, 0, 1]
Solution().doesValidArrayExist(derived) # Returns True
```

This small example demonstrates the effectiveness of the XOR operation for solving this problem and how a cumulative XOR of `derived` being equal to zero serves as the condition to determine the existence of a corresponding `original` array.

Solution Implementation

Python

```
from functools import reduce
from typing import List
from operator import xor

class Solution:
    def doesValidArrayExist(self, derived: List[int]) -> bool:
        # The function checks if there exists a valid array
        # An array is considered valid if the cumulative XOR of all elements is 0
        # 'reduce' applies the 'xor' function cumulatively to the items of 'derived', from left to right
        # Hence, the entire list is reduced to a single value
        # If this final reduced value is 0, it means all pairs are matched (i.e., a valid array exists)

        return reduce(xor, derived) == 0
```

Java

```
class Solution {

    // A method that checks if there's a valid array whose derived xor-sum is zero
    public boolean doesValidArrayExist(int[] derivedArray) {
        int xorSum = 0; // Initialize xorSum to 0 to use it as the initial value

        // Iterate over each element in the derived array
        for (int element : derivedArray) {
            // Perform XOR operation between the xorSum and the current element
            xorSum ^= element;
        }

        // Return true if the xorSum is 0, which means a valid array exists
        // Otherwise, return false
        return xorSum == 0;
    }
}
```

C++

```
#include <vector> // Include vector header for using vectors

// The Solution class definition
class Solution {
public:
    // Function checks if there exists a valid array such that all of its elements XORed equals zero
    bool doesValidArrayExist(std::vector<int>& derivedArray) {
        // Initialize a variable to store the cumulative XOR of array elements
        int cumulativeXOR = 0;

        // Iterate through each element in the derivedArray
        for (int element : derivedArray) {
            // Perform XOR operation and store the result back in cumulativeXOR
            cumulativeXOR ^= element;
        }

        // If the final result of cumulativeXOR is zero, a valid array exists (return true)
        // Otherwise, no such array exists (return false)
        return cumulativeXOR == 0;
    }
};
```

TypeScript

```
/**
 * Determines if a "valid" array exists; an array is considered valid
 * if the XOR of all its elements is zero.
 *
 * @param {number[]} numbers - The array of numbers to be checked.
 * @returns {boolean} - True if the array is valid, False otherwise.
 */
function doesValidArrayExist(numbers: number[]): boolean {
    // Initialize sum as zero to perform XOR operation.
    let xorSum = 0;

    // Iterate over each number in the array.
    for (const number of numbers) {
        // Perform XOR operation with current number and update the xorSum.
        xorSum ^= number;
    }

    // Check if xorSum is zero (all pairs XOR to zero); return true if so.
    return xorSum === 0;
}
```

```
from functools import reduce
from typing import List
from operator import xor

class Solution:
    def doesValidArrayExist(self, derived: List[int]) -> bool:
        # The function checks if there exists a valid array
        # An array is considered valid if the cumulative XOR of all elements is 0
        # 'reduce' applies the 'xor' function cumulatively to the items of 'derived', from left to right
        # Hence, the entire list is reduced to a single value
        # If this final reduced value is 0, it means all pairs are matched (i.e., a valid array exists)

        return reduce(xor, derived) == 0
```

Time and Space Complexity

The provided Python function `doesValidArrayExist` uses the `reduce` function with the `xor` operator from the `functools` and `operator` modules respectively to determine if the XOR of all the numbers in a list is equal to 0. The XOR operation is applied pairwise to the elements of the list until a single result remains.

Time Complexity

The `reduce` function applies the `xor` operation to the list `derived` with a time complexity of O(n), where n is the number of elements in the list. This is because each element in the list must be accessed exactly once to perform the XOR operation with the accumulated result.

The overall time complexity is $O(n)$.

Space Complexity

Since the `reduce` function applies the `xor` operation in-place and accumulates the result without using any additional data structures that grow with input size, the space complexity is constant.

The overall space complexity is $O(1)$.