1486. XOR Operation in an Array

Easy Bit Manipulation Math

Problem Description

You are tasked with creating an array named `nums`, where each element follows a specific pattern based on two given integers: `n` and `start`. The array should be constructed such that the element at index `i` is calculated by the formula `start + 2 * i`. In this formula, `i` is the index in the array, starting from `0` (hence it's 0-indexed), and `n` is the total number of elements that should exist within the array `nums`. Once the array is constructed, your goal is to calculate the result of applying the bitwise XOR operation to all the elements within the array.

Bitwise XOR is an operation that takes two bits and returns `1` if the bits are different and `0` if they are the same. When doing this operation between numbers, it's applied bit by bit from their binary representations. The XOR of a single number with itself is always `0`, and the XOR of a number with `0` is always the number itself.

The challenge is to write a function that will carry out this process and return the single integer that results from the cumulative XOR of all the elements in `nums`.

Intuition

To solve this problem, we can iterate over the range from `0` to `n - 1` and compute the value of `start + 2 * i` for each `i`. After computing each element based on the given pattern, we perform a bitwise XOR operation sequentially on these elements.

We start with an initial value of `0` for the answer (`ans`). Then, for each `i`, we calculate the `i`th number in the sequence with `start + 2 * i` and use the XOR operator (`^`) to combine it with our running total in `ans`. This is equivalent to saying that `ans` becomes `ans XOR (start + 2 * i)` for each iteration. Since XOR is associative, the order in which we combine the numbers doesn't matter, meaning we can sequentially update `ans` as we go through the loop.

After completing the loop, `ans` will contain the cumulative XOR of all the array's elements, which is what we return as our final solution.

The solution is straightforward and relies on the properties of the XOR operation to combine each new element into our running total. This cumulative approach is useful because we don't need to maintain the array `nums` in memory; we only need to remember the current XOR value through each iteration, which is a more memory-efficient solution.

Solution Approach

The implementation of the solution uses a simple iterative approach, leveraging the properties of the XOR operation. The reference solution provided in Python demonstrates the approach clearly.

The primary algorithmic pattern used here is iteration with a single loop. We exploit two key insights: first, we know exactly how many times to iterate since we are given `n`, the size of the array `nums`. Second, the XOR operation allows us to combine elements one by one without needing to store the entire array in memory.

Here is a step-by-step breakdown of the code implementation:

1. Initialize `ans` to `0`. This serves as an accumulation variable to store the cumulative XOR result.
2. Iterate over a range from `0` to `n - 1`. The range function generates a sequence of numbers, allowing us to calculate each array element without an actual array.
    - Within the loop, update `ans` by XORing its current value with `start + 2 * i`, which represents the current element of the array. In code: `ans ^= start + 2 * i`.
3. After completing the loop, all elements have been XORed into `ans`.
4. Finally, return `ans`, which now contains the cumulative XOR of all elements as per the array `nums[i] = start + 2 * i`.

The reason no additional data structures are used here is due to the nature of the XOR operation, which is both associative and commutative. As a result, we don't need to remember past values once they've been incorporated into `ans`.

The code uses bit manipulation (`^` operator in Python) to perform the XOR operation. Bit manipulation is a powerful technique in programming that allows for efficient computation, often with lower memory usage and faster execution, especially suitable for such bitwise arithmetic tasks.

The simplicity of the approach lies in the fact that the problem requires no more than applying the given formula iteratively and combining the results using the XOR operation. No complex data structures or algorithms are needed. The final result is directly returned after the loop's completion.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider `n = 4` and `start = 3`. Based on the problem description, we need to create an array `nums` such that:

- `nums[0] = start + 2 * 0`
- `nums[1] = start + 2 * 1`
- `nums[2] = start + 2 * 2`
- `nums[3] = start + 2 * 3`

Calculating the above values based on the given formula:

- `nums[0] = 3 + 2 * 0 = 3`
- `nums[1] = 3 + 2 * 1 = 5`
- `nums[2] = 3 + 2 * 2 = 7`
- `nums[3] = 3 + 2 * 3 = 9`

Now, we apply the bitwise XOR operation to all elements in this array. The goal is to perform the XOR operation in a cumulative manner.

- Start with an initial value of `ans = 0`.
- Perform `ans ^= nums[0]`, hence `ans = 0 ^ 3 = 3` (since XOR with zero gives the number itself).
- Update `ans` with `ans ^= nums[1]`, so `ans = 3 ^ 5`.
    - The binary representation of 3 is `011`.
    - The binary representation of 5 is `101`.
    - Performing XOR on these yields `110`, which is binary for `6`. Now, `ans = 6`.
- Update `ans` again with `ans ^= nums[2]`, so `ans = 6 ^ 7`.
    - The binary representation of 6 is `110`.
    - The binary representation of 7 is `111`.
    - Performing XOR on these yields `001`, which is binary for `1`. Now, `ans = 1`.
- Finally, update `ans` a last time with `ans ^= nums[3]`, so `ans = 1 ^ 9`.
    - The binary representation of 1 is `001`.
    - The binary representation of 9 is `1001`.
    - Performing XOR on these (after aligning the bits) yields `1000`, which is binary for `8`. Now, `ans = 8`.

The XOR of the entire array `nums` is `8`. Therefore, given `n = 4` and `start = 3`, the function would return `8`. This example demonstrates how the iterative approach can be used to calculate the cumulative XOR without explicitly creating the array `nums`.

Solution Implementation

Python

```python
class Solution:
    def xor_operation(self, n: int, start: int) -> int:
        # Initialize the answer to zero
        answer = 0

        # Loop from 0 to n-1
        for i in range(n):
            # Perform XOR operation between the current answer and the new element in the series
            answer ^= start + 2 * i

        # Return the final answer after performing all XOR operations
        return answer

# The Solution class contains a method xor_operation which takes two arguments:
# n - the number of elements in the array
# start - the first element in the array where the array is defined by the formula start + 2 * i for each element i in the range [0,
# The method applies the XOR operation cumulatively over the entire array to find the single resultant value and returns it.
```

Java

```java
// Class Solution contains a method to perform XOR operations on a sequence of numbers.
class Solution {

    // The xorOperation method takes the number of elements 'n' and the 'start' value of the series as parameters.
    public int xorOperation(int n, int start) {
        // Initialize 'result' that will hold the cumulative XOR value.
        int result = 0;

        // Loop over the series from 0 to 'n-1'.
        for (int i = 0; i < n; ++i) {
            // Perform XOR of 'result' with each element in the sequence.
            // Element value is calculated as 'start + 2*i'.
            result ^= start + 2 * i;
        }

        // Return the final XOR result after processing all elements.
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to compute the XOR of all numbers in the generated array.
    int xorOperation(int n, int start) {
        int xor_result = 0; // Initialize result variable to store the final XOR

        // Loop through the sequence to compute the XOR
        for (int i = 0; i < n; ++i) {

            // Each element in the sequence is the start value plus twice the index
            // XOR it with the current result
            xor_result ^= start + 2 * i;
        }

        // Return the final XOR result of the sequence
        return xor_result;
    }
};
```

TypeScript

```typescript
// Variable to hold the XOR result
let xorResult: number = 0;

// Function to compute the XOR of all numbers in the generated array.
function xorOperation(n: number, start: number): number {
    // Initialize the XOR result to zero for each call of the function
    xorResult = 0;

    // Loop through the numbers from 0 to n-1 to compute the XOR of the generated elements
    for (let i = 0; i < n; i++) {
        // The current element in the sequence is calculated by adding start with twice the index
        // XOR the current element with the accumulated result stored in xorResult
        xorResult ^= start + 2 * i;
    }

    // Return the XOR result of all the elements in the array
    return xorResult;
}
```

```python
class Solution:
    def xor_operation(self, n: int, start: int) -> int:
        # Initialize the answer to zero
        answer = 0

        # Loop from 0 to n-1
        for i in range(n):
            # Perform XOR operation between the current answer and the new element in the series
            answer ^= start + 2 * i

        # Return the final answer after performing all XOR operations
        return answer

# The Solution class contains a method xor_operation which takes two arguments:
# n - the number of elements in the array
# start - the first element in the array where the array is defined by the formula start + 2 * i for each element i in the range [0,
# The method applies the XOR operation cumulatively over the entire array to find the single resultant value and returns it.
```

Time and Space Complexity

- The **time complexity** of the code is `O(n)`, where `n` is the number of elements we are performing the XOR operation on. This is because there is a single loop in the function `xorOperation` that iterates `n` times, and the XOR operation itself is a constant-time operation.
- The **space complexity** of the code is `O(1)`, implying that the space required for computation does not depend on the input size `n`. The function maintains a single integer `ans` that is used to accumulate the result, hence no additional space proportional to `n` is utilized.