2425. Bitwise XOR of All Pairings

Medium Bit Manipulation Brainteaser Array

Leetcode Link

Problem Description

In the given problem, we have two arrays `nums1` and `nums2` made up of non-negative integers. We need to create a virtual array `nums3`, which would contain the results of performing the bitwise XOR operation on every possible pair formed by taking one number from `nums1` and another from `nums2`. However, instead of generating `nums3` explicitly, we are asked to directly calculate the bitwise XOR of all the elements it would contain.

In simpler terms, imagine we have, for example, `nums1 = [1, 2]` and `nums2 = [3, 4]`. The pairs and the bitwise XOR would be as follows:

- 1 XOR 3
- 1 XOR 4
- 2 XOR 3
- 2 XOR 4

We suspect `nums3` would be `[1 XOR 3, 1 XOR 4, 2 XOR 3, 2 XOR 4]`. Our task is to find the cumulative XOR of these results, i.e., `(1 XOR 3) XOR (1 XOR 4) XOR (2 XOR 3) XOR (2 XOR 4)`, without actually generating the intermediate array `nums3`.

Intuition

To understand the solution, let us first consider a property of the XOR operation: XORing the same number twice cancels it out, resulting in zero.

Now, let's analyze when a number in one of the arrays (`nums1` or `nums2`) would be XORed an even or odd number of times:

- If `nums2` has an even number of elements, each element in `nums1` is XORed with the elements in `nums2` an even number of times, which means that their overall effect will be zero (since any number XORed with itself an even number of times is zero).
- If `nums2` has an odd number of elements, each element in `nums1` is XORed with the elements in `nums2` an odd number of times, which means the numbers in `nums1` should be considered once in the final XOR calculation.
- The same logic applies when considering the elements in `nums2` with respect to the even or odd count of elements in `nums1`.

So, if the length of one of the arrays is odd, we need to consider each element in the other array for the final XOR calculation. If the lengths of both arrays are even, none of the elements are considered since the even-odd pairing would cancel out their effect.

With this intuition, we arrive at the solution approach: we initially set an answer variable `ans` to 0. We then check if the length of `nums2` is odd and XOR all elements in `nums1` to `ans`. Then we check if the length of `nums1` is odd and XOR all elements in `nums2` to `ans`. The final value of `ans` is the XOR of all elements that would be in `nums3`.

Solution Approach

The solution uses a simple but clever observation about the XOR operation and its properties:

- **XOR of a number with itself**: `a XOR a = 0`
- **XOR with zero**: `a XOR 0 = a`
- **Commutative property**: `a XOR b = b XOR a`
- **Associative property**: `a XOR (b XOR c) = (a XOR b) XOR c`

These properties mean that when we XOR an even number of the same numbers, the result is `0`, and when we do it an odd number of times, we get the number itself.

Let's go over the implementation details based on the solution provided:

1. We initialize a variable `ans` to `0`. This variable will serve as the accumulator for the XOR operations.
2. We then check the length of `nums2` using the bitwise `&` operator with `1` to determine if it's odd (`len(nums2) & 1`). The `& 1` trick checks the least significant bit of a number, which represents its odd/even status (odd numbers have a least significant bit of `1`, even numbers have a `0`).
3. If `nums2` has an odd length, we iterate over each value in `nums1` and apply the XOR operation to our `ans` variable. Due to the asymmetric nature of the required pairings (every element in `nums1` is paired with every element in `nums2`), the cumulative effect when the count is odd is that the numbers in `nums1` indeed contribute to the final result.
4. We do the same for `nums1` by checking if it has an odd length, iterating over `nums2`, and updating `ans` accordingly.
5. At this stage, `ans` will hold the bitwise XOR of all integers that would be in `nums3`, and it's returned as the solution.

In the implementation, no extra data structures are needed because the solution leverages the XOR operation's properties to avoid constructing the `nums3` array. This approach is notably efficient in both time and space complexity, as it requires iterating over each array only once (O(n) where n is the length of the longer array) and uses only a constant amount of extra space for the `ans` variable.

Another key aspect is the use of bitwise operations (`^` for XOR and `&` for AND), which are low-level operations that are generally very fast in execution compared to higher-level arithmetic or logic operations.

No complex patterns or algorithms are used; the simplicity of the approach comes from a deep understanding of the XOR operation, which makes it a very elegant solution.

Example Walkthrough

Let's take two arrays `nums1 = [5, 9]` and `nums2 = [7, 11, 12]` to illustrate the solution approach.

1. We start by initializing `ans` to `0`.
2. We then check if the length of `nums2` is odd. Since it has 3 elements, which is odd, we proceed to the next step.
3. We iterate over each element in `nums1` and XOR it with `ans`.
   - After XORing `5` with `ans` (initially `0`), `ans` becomes `5` (`5 XOR 0 = 5`).
   - Next, we XOR `9` (the second element in `nums1`) with `ans`, resulting in `ans = 5 XOR 9 = 12`.
4. We don't need to check the length of `nums1` because we already know `nums2` has an odd length, and that's enough to determine that all elements of `nums1` should be part of the final XOR. If `nums1` has an even length, XORing it with an even count of `nums2` would cancel out its elements.

At this point, we have `ans = 12`, which is the cumulative XOR of array `nums1` with each element in `nums2` considering the odd-even pairing effect.

Since the implementation leverages the XOR operation's properties to avoid constructing `nums3`, the calculation is direct and efficient. The odd-length check using `len(nums2) & 1` ensures that we only iterate through `nums1` or `nums2` if necessary.

In this example, we're avoiding the unnecessary and memory-intensive task of generating all possible pairs and their XOR results by directly accumulating the XORs that would result from such pairs. Thus, applying the algorithm's insight provides us with an answer: the bitwise XOR of all the elements that would be in the virtual array `nums3` is `12`.

Python Solution

```python
from typing import List

class Solution:
    def xor_all_nums(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize the result of XOR operation to 0.
        result_xor = 0

        # If the length of nums2 is odd, XOR all elements in nums1 with result_xor.
        if len(nums2) % 2:
            for num in nums1:
                result_xor ^= num

        # If the length of nums1 is odd, XOR all elements in nums2 with result_xor.
        if len(nums1) % 2:
            for num in nums2:
                result_xor ^= num

        # Return the final result after performing all XOR operations.
        return result_xor
```

Java Solution

```java
// A class to find a solution for the XOR problem
class Solution {
    // Method to calculate the XOR of all elements after performing XOR as if each element of nums1 is paired with all elements of n
    public int xorAllNums(int[] nums1, int[] nums2) {
        int result = 0; // Initialize result to zero

        // If the length of nums2 is odd, XOR result with all elements in nums1
        // Because if nums2 has an odd number of elements, each element in nums1 will be represented an odd number of times when tak
        if (nums2.length % 2 == 1) {
            for (int value : nums1) {
                result ^= value;
            }
        }

        // Likewise, if the length of nums1 is odd, XOR result with all elements in nums2
        if (nums1.length % 2 == 1) {
            for (int value : nums2) {
                result ^= value;
            }
        }

        return result; // Return the accumulated XOR result
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the XOR of all elements as if they were all combined
    // Parameter nums1 is the first vector of integers
    // Parameter nums2 is the second vector of integers
    // Returns the XOR of all elements combined
    int xorAllNums(vector<int>& nums1, vector<int>& nums2) {
        int result = 0;

        // If nums2 has an odd number of elements, XOR all elements of nums1
        if (nums2.size() % 2 == 1) {
            for (int num : nums1) {
                result ^= num;
            }
        }

        // If nums1 has an odd number of elements, XOR all elements of nums2
        if (nums1.size() % 2 == 1) {
            for (int num : nums2) {
                result ^= num;
            }
        }

        // Return the final result of the XOR operation
        return result;
    }
};
```

Typescript Solution

```typescript
/**
 * XOR all numbers from two arrays in a specific way.
 *
 * If one array has an odd length, XOR all numbers from the other array.
 * If both arrays have odd lengths, XOR all numbers from both arrays.
 *
 * @param {number[]} nums1 - The first array of numbers.
 * @param {number[]} nums2 - The second array of numbers.
 * @return {number} - The resulting XOR from the above rule.
 */
function xorAllNums(nums1: number[], nums2: number[]): number {
    // Initialize the answer variable to store the final result.
    let result = 0;

    // Check if the length of nums2 is odd.
    // If it is, accumulate the XOR of all elements in nums1 with the result.
    if (nums2.length % 2 !== 0) {
        result ^= nums1.reduce((accumulator, currentValue) => accumulator ^ currentValue, 0);
    }

    // Check if the length of nums1 is odd.
    // If it is, accumulate the XOR of all elements in nums2 with the result.
    if (nums1.length % 2 !== 0) {
        result ^= nums2.reduce((accumulator, currentValue) => accumulator ^ currentValue, 0);
    }

    // Return the final XOR result.
    return result;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code depends on the lengths of the input lists `nums1` and `nums2`.

1. Checking `if len(nums2) & 1`: This is an `O(1)` operation as it involves checking the parity of the length of `nums2`.
2. Iterating over `nums1`: If the length of `nums2` is odd, we iterate over all elements in `nums1`, resulting in `O(n)` complexity where `n` is the length of `nums1`.
3. Checking `if len(nums1) & 1`: This is another `O(1)` operation similar to the first check but for `nums1`.
4. Iterating over `nums2`: If the length of `nums1` is odd, we iterate over all elements in `nums2`, resulting in `O(m)` complexity where `m` is the length of `nums2`.

Combining these operations, in the worst case, both `nums1` and `nums2` lengths are odd, which results in iterating over both lists. Hence, the total time complexity is `O(n + m)`.

Space Complexity

The space complexity is related to the amount of extra space required that is not part of the input. For the given code, we only use an extra variable `ans` to store the intermediate results of the XOR operation, regardless of the input size.

The space complexity is thus `O(1)` because the space used does not scale with the size of the input.