# 1835. Find XOR Sum of All Pairs Bitwise AND

Hard | Bit Manipulation | Array | Math

## Problem Description

In this problem, we are given two arrays, `arr1` and `arr2`, both containing non-negative integers. We are required to compute the XOR sum, which involves several steps. The first step is to find every possible pair of elements where one element comes from `arr1` and the other comes from `arr2`. For each pair `(i, j)`, we calculate the bitwise AND of `arr1[i]` and `arr2[j]`. After we've calculated the AND for all possible pairs, we'll have a new list containing all these values. The next step is to calculate the XOR sum of all the elements in this new list. This will be our answer.

To simplify this process, we can use the properties of the XOR and AND operations. Instead of doing a pairwise AND followed by XOR over all elements, we can look for a pattern or relationship that simplifies the computation.

## Intuition

The smart approach is based on a key mathematical insight that when working with bitwise operations, particularly AND (`&`) and XOR (`^`), we can move the operations around because they are associative. That means, instead of calculating the AND for every pair and then XORing all of those, we can first XOR all elements within each array and then perform the AND over these two results.

This works because XOR is a linear operation in the context of a field with two elements (the numbers 0 and 1). Here, XOR is akin to addition, and AND is like multiplication. So we can rearrange the operations in the same way that you would when working with addition and multiplication (distributive property).

So, let the variable `a` represents the XOR of all elements in `arr1`, and let `b` represents the XOR of all elements in `arr2`. The bitwise AND of `a` and `b` will result in the same number as if we had done all the individual ANDs and then XORed those together. This simplifies the algorithm significantly and makes it much more efficient since we only need to go through each array once for the XOR computations and then perform a single AND operation.

The provided solution encapsulates this approach neatly by first using Python's `reduce` function with the XOR operator to condense each array down to a single value that represents the XOR of all its elements. Then, it simply performs the AND operation on these two values and returns the result, which is the sought-after XOR sum of the list containing all possible ANDs between `arr1` and `arr2`.

## Solution Approach

The solution approach leverages a couple of key programming and mathematical concepts to deliver an efficient solution without the need to manually iterate through every possible `(i, j)` pair across the `arr1` and `arr2` arrays.

**Key Concepts Used**:

- **Bitwise XOR**: In computing, the XOR is a type of bitwise operation that returns 1 only if the two bits are different; otherwise, it returns 0. In terms of arithmetic, it can be seen as addition without carry.
- **Bitwise AND**: The AND operation takes two bits and returns 1 only if both bits are 1.
- **Associative property of XOR**: Allows the rearrangement of XOR operations over a set of values without altering the final result.
- **Distributive property over XOR and AND**: In the problem's context, bitwise XOR and AND exhibit similar properties to addition and multiplication, respectively. It means that we can simplify calculations by "distributing" the operations, much like how we would in algebra.

**Algorithm Explaination**:

- The problem suggests that we first find all the results of `arr1[i] AND arr2[j]` for all `i` and `j`, then calculate the XOR of all these results. But this is computationally expensive.
- The solution leverages the fact that XOR is associative and distributive over AND, so we can reduce the entire operation into much simpler steps:
  1. XOR all elements of `arr1` to get a single number `a`.
  2. XOR all elements of `arr2` to get a single number `b`.
  3. AND the results `a` and `b` to get the final answer.

The provided solution uses Python's `reduce` function from the `functools` library to apply the XOR operation across all elements of each array, effectively collapsing them into a single integer representing the cumulative XOR for that array.

Here is how the solution step-by-step implementation aligns with the algorithm:

1. `reduce(xor, arr1)` - This line uses `reduce` to apply the XOR operation successively over the elements of `arr1`. It starts with the first two elements of `arr1`, applies XOR, takes that result, and then XORs it with the next element, and so on until all elements have been combined into a single value `a`.
2. `reduce(xor, arr2)` - Similarly, this line processes `arr2` to produce a single XOR result `b`.
3. `return a & b` - Finally, the function returns the bitwise AND of the two previously computed values, `a` and `b`.

By the properties of XOR and AND operations, this result is equivalent to XORing all the results of `arr1[i] AND arr2[j]` while being much more performant since it avoids the need to calculate each pair explicitly.

This clever use of bitwise operators and the properties of XOR and AND provides an elegant and efficient solution to the problem.

### Example Walkthrough

Let's consider an example with the two following arrays:

`arr1 = [1, 2] arr2 = [3, 4]`

If we were to follow the original problem instructions directly, we would calculate the AND for every possible pair `(i, j)`:

- `1 AND 3 = 1`
- `1 AND 4 = 0`
- `2 AND 3 = 2`
- `2 AND 4 = 0`

We would then XOR all these results together:

- `1 ^ 0 ^ 2 ^ 0 = 3`

However, the solution approach simplifies this process by first XORing all elements within each array:

- XOR all elements of `arr1`: `1 ^ 2 = 3`
- XOR all elements of `arr2`: `3 ^ 4 = 7`

Now, instead of XORing 4 values as in the initial approach, we only need to perform one AND operation:

- `3 AND 7 = 3`

This single value, `3`, is the result we are looking for and the same as the XOR sum of all possible ANDs between `arr1` and `arr2`. The simplification from the solution approach has saved us time by reducing the number of operations required.

## Solution Implementation

### Python

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

class Solution:
    def getXORSum(self, arr1: List[int], arr2: List[int]) -> int:
        # Calculate the XOR of all elements in arr1
        xor_arr1 = reduce(xor, arr1)
        # Calculate the XOR of all elements in arr2
        xor_arr2 = reduce(xor, arr2)
        # Return the bitwise AND of the two XOR results
        return xor_arr1 & xor_arr2
```

### Java

```java
class Solution {
    public int getXORSum(int[] arr1, int[] arr2) {
        // Initialize xorSum1 to store the XOR of all the elements in arr1
        int xorSum1 = 0;
        // Initialize xorSum2 to store the XOR of all the elements in arr2
        int xorSum2 = 0;

        // Iterate over each element in arr1 and perform XOR
        // This will give the cumulative XOR for arr1
        for (int value : arr1) {
            xorSum1 ^= value;
        }

        // Iterate over each element in arr2 and perform XOR
        // This will give the cumulative XOR for arr2
        for (int value : arr2) {
            xorSum2 ^= value;
        }

        // Return the bitwise AND of the two cumulative XOR results
        return xorSum1 & xorSum2;
    }
}
```

### C++

```cpp
#include <numeric> // Required for std::accumulate
#include <functional> // Required for std::bit_xor
#include <vector>

class Solution {
public:
    int getXORSum(vector<int>& arr1, vector<int>& arr2) {
        // Calculate the XOR of all elements in arr1
        int xorSumArr1 = std::accumulate(arr1.begin(), arr1.end(), 0, std::bit_xor<int>());
        // Calculate the XOR of all elements in arr2
        int xorSumArr2 = std::accumulate(arr2.begin(), arr2.end(), 0, std::bit_xor<int>());

        // Return the bitwise AND of the two XOR sums
        return xorSumArr1 & xorSumArr2;
    }
};
```

### TypeScript

```typescript
// Function to compute the bitwise XOR of all elements in an array
function bitwiseXOROfArray(elements: number[]): number {
    // Reduce the array by applying the XOR operation between elements
    return elements.reduce((accumulated, current) => accumulated ^ current, 0);
}

// Function to calculate the bitwise AND of the XOR sum of two arrays
function getXORSum(arr1: number[], arr2: number[]): number {
    // Calculate the XOR sum of the first array
    const xorSumArr1 = bitwiseXOROfArray(arr1);
    // Calculate the XOR sum of the second array
    const xorSumArr2 = bitwiseXOROfArray(arr2);

    // Return the result of the bitwise AND operation between the two XOR sums
    return xorSumArr1 & xorSumArr2;
}
```


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

class Solution:
    def getXORSum(self, arr1: List[int], arr2: List[int]) -> int:
        # Calculate the XOR of all elements in arr1
        xor_arr1 = reduce(xor, arr1)
        # Calculate the XOR of all elements in arr2
        xor_arr2 = reduce(xor, arr2)
        # Return the bitwise AND of the two XOR results
        return xor_arr1 & xor_arr2
```


## Time and Space Complexity

The given Python code is designed to calculate the bitwise XOR sum of two arrays and then return the bitwise AND of these two sums.

### Time Complexity:

The function uses the `reduce` method along with the `xor` bitwise operation to compute the bitwise XOR sum of all elements in `arr1` and `arr2`. Doing this for one array (either `arr1` or `arr2`) takes `O(n)` time, where `n` is the number of elements in the array.

The function performs this operation once for each array, which means the total time complexity for both operations is `O(n + m)`, where `n` is the length of `arr1` and `m` is the length of `arr2`.

Therefore, the time complexity of the provided code is `O(n + m)`.

### Space Complexity:

The space complexity of the code is based on the additional memory used by the algorithm. Here, the variables `a` and `b` are used to store the result of the XOR operation on `arr1` and `arr2`. No other additional space that grows with the input size is used.

Therefore, the extra space is constant, giving us a space complexity of `O(1)`.