2997. Minimum Number of Operations to Make Array XOR Equal to K

Medium | Bit Manipulation | Array

Problem Description

In this problem, we are given an array `nums` of integers and a separate integer `k`. Each number in the array is represented in binary and we can flip any of the bits in its binary representation (i.e., change a `0` to a `1` or vice versa) as many times as we want. Our goal is to perform the minimum number of such bit flips so that when we compute the bitwise XOR of all the numbers in the array, the result equals `k`. It is also mentioned that we can flip the leading zeros—which are usually omitted in binary notation. Our task is to find out the minimum number of flips needed to achieve this.

Intuition

To approach this problem, we don't really need to flip any bits initially. Instead, we can directly compute the bitwise XOR of all the elements in `nums`. Since XOR is an associative operation, the order in which we perform it doesn't matter, and since XORing a number with itself yields zero, any duplicate numbers effectively cancel each other out. After this computation, we'll have a single number which is the result of the XOR of all the array elements.

The problem now simplifies to finding the minimum number of bits we need to flip in this result to make it turn into `k`. To do this, we can XOR this result with `k`.

Why does that work? XORing two numbers highlights the bits where they are different (since XOR gives us 1 when we compare bits that are different and 0 when bits are the same).

The final step is to count the number of bits that are set to `1` in this XOR result. Since these 1's represent differing bits, they are the exact bits we need to flip to turn our array XOR result into `k`. The number of such bits is the minimum operations required, which is precisely what we need.

Solution Approach

The solution relies on bit manipulation and the properties of the XOR operation. The approach is straightforward with four key steps:

1. **XOR all elements with `k`:** The `reduce` function from Python's `functools` is utilized here with the `xor` operator from the `operator` module. `reduce` applies the `xor` function cumulatively to the items of `nums`, from left to right, so as to reduce the array to a single value. This value represents the XOR of all elements of `nums`. Then, by XORing this result with `k`, we find out which bits differ between the XOR of the array elements and `k`.
2. **Calculate the number of differing bits:** In the final result, a bit set to `1` indicates a position where a bit must be flipped to obtain `k`. The Python method `bit_count()` is used on the resulting number to get the count of bits that are set to `1`.
3. **Minimum operations required:** The count obtained from `bit_count()` is the minimum number of operations required. This is because each bit that is set to `1` can be flipped individually, and no unnecessary bit flips are required.
4. **Return the result:** The final count of set bits is returned as the solution to the problem: the minimum number of bit flips needed.

In terms of data structures, this solution is efficient as it operates with integers directly and does not require any additional data structures. The complexity of the algorithm is primarily dependent on the number of bits required to represent the numbers in the array, which is at most `O(log(max(nums, k)))`.

Here's the consolidated approach in code:

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

class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        return reduce(xor, nums, k).bit_count()
```

In this code:

- `reduce(xor, nums, k)` computes the XOR of all elements in `nums` along with `k`.
- `.bit_count()` then counts the number of set bits (bits that are `1`) of the result from the reduce function, thereby counting the number of operations needed.

This solution leverages the elegance and efficiency of bit operations to solve the problem in a crisp and concise manner.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we have the following array of integers `nums` and the integer `k`:

- `nums = [2, 3, 1]`
- `k = 5`

The binary representations of the numbers are:

- `2` → `10`
- `3` → `11`
- `1` → `01`
- `5` → `101`

Here's how we apply the solution approach step by step:

1. **XOR all elements with `k`**: First, calculate the XOR of all the elements in `nums`:

- `2 XOR 3 XOR 1 = 10 XOR 11 XOR 01 = 00 (binary for 0)`

Now we XOR the result with `k`:

- `0 XOR 5 = 000 XOR 101 = 101`

2. **Calculate the number of differing bits**: We use `bit_count()` to count how many bits are set to `1` in `101`:

- `101` has two bits set to `1`.

3. **Minimum operations required**: The number of set bits is the minimum number of operations required. Thus, we need **2 operations** to transform the array's XOR result into `k`.
4. **Return the result**: The final result is `2`, so our function `minOperations(nums, k)` will return `2`.

This example demonstrates how to apply the proposed algorithm step by step, showing that we can achieve the desired result with a minimum number of bit flips. Each bit that differs between the XOR result of the array and `k` represents a flip we need to make. In this case, flipping the second and third bits (from the right) of the number `0` (the XOR of `nums`) will result in the binary number `101`, which represents the integer `5`. The result aligns with the expected output of the `minOperations` method, confirming the validity of the algorithm.

Solution Implementation

Python

~~~~python
from functools import reduce
from operator import xor

class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        # This method calculates the minimum number of operations
        # using bitwise XOR and counts the number of set bits.

        # Perform bitwise XOR of all numbers in the list with k
        xor_result = reduce(xor, nums, k)

        # Return the count of set bits in the XOR result
        # .bit_count() is a method available from Python 3.10 onwards
        # that counts the number of set bits (1s) in the integer
        return xor_result.bit_count()
```

Note: In the original code, `reduce(xor, nums, k)` applies the `xor` operation consecutively to the elements of `nums` list start

If you are working in an environment that does not support Python 3.10 or newer, you could replace the `bit_count()` method with

```python
        # Counts the number of set bits ('1's) in the binary representation of the integer
        return bin(xor_result).count('1')
~~~~

Java

```java
class Solution {
    /**
     * This method calculates the minimum number of operations to reduce the XOR of all elements in the array to zero.
     * An operation is defined as changing any element of the array to any other value.
     *
     * @param nums The array of integers to be processed.
     * @param k The initial value to be XORed with elements of the array.
     * @return The minimum number of operations required.
     */
    public int minOperations(int[] nums, int k) {
        // Iterate through each element in the array
        for (int num : nums) {
            // XOR the current element with k
            k ^= num;
        }

        // Return the count of bits set to 1 in the result of the XOR operations, as each bit set to 1 requires one operation to
        return Integer.bitCount(k);
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For std::count

class Solution {
public:
    // Function to find the minimum number of operations to convert a vector of integers
    // so that the XOR of all its elements is equal to a given integer k.
    // Each operation can flip a bit in an integer of the vector.
    int minOperations(std::vector<int>& nums, int k) {
        int xorResult = k;
        // Calculate the XOR of all elements in nums with k
        for (int num : nums) {
            xorResult ^= num;
        }

        // __builtin_popcount returns the number of set bits (1-bits) in an integer
        // which represents the number of different bits from the desired XOR result.
        // This will be the minimum number of operations required to achieve the XOR result k.
        return __builtin_popcount(xorResult);
    }
};
```

TypeScript

```typescript
// Calculates the minimum number of operations to reduce the XOR of all elements to zero.
// Each operation consists of incrementing or decrementing an element in the array.
function minOperations(nums: number[], targetXOR: number): number {
    // Calculate the cumulative XOR of all numbers in the array and the initial targetXOR.
    for (const num of nums) {
        targetXOR ^= num; // XOR current number with accumulator.
    }
    // Return the count of set bits in the resulting XOR to determine
    // the minimum number of operations required.
    return countSetBits(targetXOR);
}

// Counts the number of set bits (1s) in the binary representation of the integer.
function countSetBits(num: number): number {
    // Apply bit manipulation techniques to count the set bits.
    num = num - ((num >>> 1) & 0x55555555);           // Group bit pairs and sum them up.
    num = (num & 0x33333333) + ((num >>> 2) & 0x33333333); // Group nibbles and sum them up.
    num = (num + (num >>> 4)) & 0x0f0f0f0f;                // Group bytes and sum them up.
    num += num >>> 8;                                      // Sum up halves.
    num += num >>> 16;                                     // Sum up words.
    // Mask out the irrelevant bits and return the number of set bits which
    // is contained in the least significant 6 bits of the result.
    return num & 0x3f;
}
```

~~~~python
from functools import reduce
from operator import xor

class Solution:
    def minOperations(self, nums: List[int], k: int) -> int:
        # This method calculates the minimum number of operations
        # using bitwise XOR and counts the number of set bits.

        # Perform bitwise XOR of all numbers in the list with k
        xor_result = reduce(xor, nums, k)

        # Return the count of set bits in the XOR result
        # .bit_count() is a method available from Python 3.10 onwards
        # that counts the number of set bits (1s) in the integer
        return xor_result.bit_count()
```

Note: In the original code, `reduce(xor, nums, k)` applies the `xor` operation consecutively to the elements of `nums` list starting

If you are working in an environment that does not support Python 3.10 or newer, you could replace the `bit_count()` method with the

```python
        # Counts the number of set bits ('1's) in the binary representation of the integer
        return bin(xor_result).count('1')
~~~~

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the array `nums`. This is because the `reduce` function applies the `xor` operation to each element of the array exactly once.

The space complexity of the code is `O(1)` since the `xor` operation and `bit_count()` method do not require any additional space that grows with the size of the input array. The operation is performed in constant space, regardless of the length of `nums`.