2317. Maximum XOR After Operations

Medium Bit Manipulation Array Math

Problem Description

In this problem, we are given an array of integers `nums`, and our objective is to maximize the bitwise XOR of all elements of the array after performing a certain operation any number of times. The operation defined allows us to:

1. Choose any non-negative integer `x`.
2. Select any index `i`.
3. Update `nums[i]` to `nums[i] AND (nums[i] XOR x)`.

Here `AND` and `XOR` are bitwise operations. Bitwise `AND` results in a bit being 1 if both bits at the same position are 1, otherwise it's 0. Bitwise `XOR` results in a bit being 1 if the bits at the same position differ, otherwise it's 0.

The main challenge is to figure out what `x` and `i` we should choose to maximize the final XOR of all elements.

Intuition

To solve this problem, we should understand the properties of the `XOR` and `AND` operations. An important fact to consider is the behavior of `AND` when combined with `XOR`: for any integer `n`, `n AND (n XOR x)` equals `n`. Armed with this knowledge, we can simplify the problem significantly.

Since applying the given operation does not change the value of any element in the array (because for any given `i` and `x`, `nums[i] AND (nums[i] XOR x)` will always equal `nums[i]`), the elements of `nums` can be left as they are. Thus, the operation has no effect on the outcome of maximizing the bitwise XOR of all array elements. This leads us to the conclusion that the maximum possible bitwise XOR can be obtained by simply computing the bitwise XOR of all elements in the array without modifying them.

To implement the solution, we use a `reduce` function from Python's `functools` module with `or_`, which is a function that performs the bitwise OR operation (imported from the `operator` module). By performing a bitwise OR between all numbers in `nums`, we effectively combine all the bits set in any number in the array. Since `XOR` of identical bits is `0` and `1` otherwise, the bitwise OR accumulates all the `1` bits across all the numbers, yielding the maximum XOR value achievable.

Solution Approach

The solution employs the `reduce` function, a Python technique that cumulatively applies an operation to all elements of an iterable. The specific operation used here is the bitwise OR, provided by the `or_` function from the `operator` module.

Here's a step-by-step breakdown of the algorithm used in the solution:

1. Import the `reduce` function from the `functools` module and the `or_` function from the `operator` module (this import is not shown in the given code, but is required for `or_` to be used).
2. Initialize the `maximumXOR` method, which takes `nums`, the list of integers.
3. Within the method, call `reduce` with two arguments:
   - The first argument is `or_`, the function that will be applied to the elements of `nums`.
   - The second argument is the `nums` list itself.
4. `reduce` then applies the `or_` function cumulatively to the items of `nums`, from left to right. This means that it starts with the first two elements `a` and `b` of `nums`, computes `a | b`, then applies the `or_` operation to this result and the next element, and so on until all elements have been included.
5. Since the bitwise OR accumulates all bits set in any number in `nums`, the final result represents the maximum possible XOR that can be obtained, as each bit position in the final result is set to `1` if it's set in at least one number in the array.
6. The result of the `reduce` operation is the maximum XOR value, which is returned by the `maximumXOR` method.

In summary, the given solution cleverly sidesteps the need to perform any complex calculations by realizing that the maximum XOR is simply the accumulation of all the unique bits set in the original `nums` list, and it uses `reduce` with bitwise OR to calculate this result efficiently.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following array of integers `nums`:

```
nums = [3, 10, 5]
```

Based on the provided content, our goal is to maximize the bitwise XOR of all elements in the array by performing defined operations. However, since we know that the operations won't change any value (as `n AND (n XOR x)` equals `n`), the process simplifies to finding the XOR of all numbers as they are.

The binary representations of the numbers in `nums` are as follows:

```
3  ->  011
10 -> 1010
5  ->  101
```

Now let's compute the bitwise OR (not XOR, as incorrectly mentioned in the content, but it's intended to be the OR operation based on the explanation) of all elements in `nums`:

- Start with the first two numbers: 3 (011) | 10 (1010) = 11 (1011).
- Now, take the previous OR result and apply it to the next number: 11 (1011) | 5 (0101) = 15 (1111).

The result of 15 (1111) in binary indicates that all bit positions have been set to `1`. This is the maximum value we can obtain from a bitwise XOR of any subsequence of these numbers because all possible `1` bits are included in the result.

Thus, using the solution approach, we would simply call `reduce` with the `or_` function:

```
from functools import reduce
from operator import or_

nums = [3, 10, 5]
max_xor_value = reduce(or_, nums)
print(max_xor_value)  # Output should be 15
```

Indeed, the maximum possible bitwise XOR value for this array is 15, which is the result we obtained by applying the bitwise OR operation to all elements of `nums` using the `reduce` function. The computation of the result agrees perfectly with our manual calculation, illustrating the correctness and efficiency of the solution approach.

Solution Implementation

Python

```
from functools import reduce  # Import the reduce function from the functools module
from operator import or_      # Import the or_ function from the operator module
from typing import List       # Import the List type from the typing module for type hints

class Solution:
    def maximum_xor(self, nums: List[int]) -> int:
        # Compute the maximum XOR value of all elements in the nums list.
        #
        # The reduce function iteratively applies the binary OR operation (or_)
        # to all elements of the nums list. This is because the maximum XOR
        # value can be achieved by performing a bitwise OR on all numbers, since
        # XORing a bit with 0 keeps the original bit, and XOR of all different
        # bits set in the numbers will be reflected in the OR operation.
        #
        # Aras:
        #     nums (List[int]): A list of integers.
        #
        # Returns:
        #     int: The maximum XOR value of all elements in nums.

        return reduce(or_, nums)

# Example usage:
# Instantiate the Solution class.
solution = Solution()
# Call the maximum_xor method with a list of integers.
max_xor = solution.maximum_xor([1, 2, 3])
# The returned value would be 1 | 2 | 3 which is 3 (in binary: 01 | 10 | 11 = 11)
print(max_xor) # Output will be 3
```

Java

```
class Solution {
    // Method to calculate the maximum XOR of all elements in the array
    public int maximumXOR(int[] nums) {
        int maxXor = 0; // Initialize the variable to store the maximum XOR result
        // Loop through each number in the array
        for (int num : nums) {
            // Perform bitwise OR operation with current number and store the result
            // This will set maxXor to the union of bits set in any of the numbers
            maxXor |= num;
        }
        // Return the maximum XOR value obtained
        return maxXor;
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Function to calculate the maximum XOR of all elements in the given vector
    int maximumXOR(vector<int>& nums) {
        int result = 0; // Initialize result to 0, the identity element for XOR

        // Iterate over all the numbers in the vector
        for (int num : nums) {
            // Perform bitwise OR on the result with the current number
            // This ensures that any bit set in any number will be set in the final result
            result |= num;
        }

        // The result now contains the maximum XOR possible with the given numbers
        return result;
    }
};
```

TypeScript

```
// Function to calculate the maximum XOR of an array of numbers
function maximumXOR(nums: number[]): number {
    // Initialize answer with 0
    let answer = 0;

    // Iterate through each number in the array
    for (const number of nums) {
        // Perform bitwise OR between the current answer and the number
        // This accumulates the set bits across all numbers in the array
        answer |= number;
    }

    // Return the final answer which is the maximum XOR value
    return answer;
}
```

```
from functools import reduce  # Import the reduce function from the functools module
from operator import or_      # Import the or_ function from the operator module
from typing import List       # Import the List type from the typing module for type hints

class Solution:
    def maximum_xor(self, nums: List[int]) -> int:
        # Compute the maximum XOR value of all elements in the nums list.
        #
        # The reduce function iteratively applies the binary OR operation (or_)
        # to all elements of the nums list. This is because the maximum XOR
        # value can be achieved by performing a bitwise OR on all numbers, since
        # XORing a bit with 0 keeps the original bit, and XOR of all different
        # bits set in the numbers will be reflected in the OR operation.
        #
        # Aras:
        #     nums (List[int]): A list of integers.
        #
        # Returns:
        #     int: The maximum XOR value of all elements in nums.

        return reduce(or_, nums)

# Example usage:
# Instantiate the Solution class.
solution = Solution()
# Call the maximum_xor method with a list of integers.
max_xor = solution.maximum_xor([1, 2, 3])
# The returned value would be 1 | 2 | 3 which is 3 (in binary: 01 | 10 | 11 = 11)
print(max_xor) # Output will be 3
```

Time and Space Complexity

The time complexity of the code is `O(N)`, where `N` is the number of elements in the list `nums`. This is because the function `reduce` applies the `or_` operator (`|`) sequentially to the elements of the list, and applying the `or_` operator takes constant time `O(1)` for each pair of integers.

The space complexity of the function is `O(1)`, as it reduces all the elements to a single integer without using any additional space that grows with the input size.