1829. Maximum XOR for Each Query

Medium Bit Manipulation Array Prefix Sum Leetcode Link

Problem Description

In the given problem, we have a sorted array called `nums` which contains `n` non-negative integers, and we also have an integer named `maximumBit`. The goal is to perform `n` queries by following two steps:

1. You need to find a non-negative integer `k` that is less than `2^maximumBit`. This integer `k`, when XORed (`^`) with the cumulative XOR of all elements in `nums`, should yield the maximum possible value.
2. After finding `k`, remove the last element from the array `nums`.

We should continue performing these queries until the array `nums` is empty and return an array `answer` that contains the results of each query, where `answer[i]` is the result of the `i-th` query.

Intuition

To understand the solution, we first need to understand the properties of the XOR operation:

- XOR of a number with itself is 0.
- XOR of a number with 0 is the number itself.
- XOR is associative and commutative, which means the order of operands doesn't affect the result.

Given these properties and the fact that we want to maximize the result of XORing the cumulative XOR of the `nums` array with `k`, an efficient way to do this is by considering what makes a number bigger in binary terms. A number is bigger if it has more significant bits set to 1. Since the array `nums` is sorted and we want to maximize the XOR result, we can iteratively calculate the cumulative XOR of the numbers in the reverse order (starting from the end), because, on each query, we are removing the last element.

For each cumulative XOR value, we want to find the best `k` such that `cumulative XOR ^ k` is as large as possible. As `k` should be less than `2^maximumBit`, we can generate a mask that has all bits set up to `maximumBit` (by doing `(1 << maximumBit) - 1`). By XORing the cumulative XOR with this mask, we're essentially flipping all the bits of the cumulative XOR within the range allowed by `maximumBit`. This flipping ensures that we get the biggest number possible since, if a bit in the cumulative XOR is 0, it gets flipped to 1, and since we start with the most significant bit down to the least, we maximize the result. The answer for each query is then this maximized XOR value.

So, the solution code does this in the following steps:

1. It initializes an empty list `ans` to store the answer for each query.
2. It calculates the initial cumulative XOR of all elements in `nums` with `reduce(xor, nums)`.
3. It creates a mask to define the range of valid values for `k` with `(1 << maximumBit) - 1`.
4. Lastly, it iterates through each number in `nums` in reverse, finds the answer for each query using the mask, appends it to `ans`, and updates `xs` by XORing it with the current element being removed - effectively calculating the new cumulative XOR for the next query.

The computed `ans` array is returned as the final result.

Solution Approach

The implementation of the provided solution leverages the cumulative XOR property and bit manipulation to answer each query efficiently. Let's walk through the crucial steps involved:

1. **Accumulate the Global XOR**: The solution first employs the `reduce` function with the `xor` operator from Python's functools to compute the cumulative XOR (`xs`) of all elements in the array `nums`. This step gives us the starting point for performing our queries.

   ```
   xs = reduce(xor, nums)
   ```

2. **Prepare the Mask**: A mask is prepared using bitwise left shift and subtraction:

   ```
   mask = (1 << maximumBit) - 1
   ```

   This creates a number where all bits less than `maximumBit` are set to 1. The purpose of the mask is to invert the bits in the cumulative XOR within the range specified by `maximumBit`.

3. **Iterate in Reverse**: The solution then iterates through `nums` in reverse. The purpose of iterating in reverse is to simulate the step-by-step removal of the last element from the `nums` array after each query, as required by the problem statement.

   ```
   for x in nums[::-1]:
   ```

4. **Find `k` and Update**: In each iteration, the current cumulative XOR (`xs`) is XORed with the mask to find the desired `k`. This `k` is the number that, when XORed with the current cumulative XOR, yields the maximum result under the constraints of `maximumBit`.

   ```
   k = xs ^ mask
   ```

   `k` is then appended to the answer array `ans`. After finding the answer to the current query and before moving on to the next iteration (the next query), `xs` is updated by XORing it with the current number `x`. This effectively removes the last element of the `nums` in the cumulative XOR perspective.

   ```
   ans.append(k)
   xs ^= x
   ```

5. **Return the Result**: After completing all iterations, the list `ans` contains the answers to all `n` queries in the respective order. The list `ans` is then returned.

The approach notably utilizes bitwise operations, a mask to flip relevant bits to maximize XOR results, and iterates in reverse to simulate the query process without manually updating the array. Data structures used include only the input list `nums` and the output list `ans`. The code is efficient as it does not use extra space for a modified array and has a time complexity of O(n) where n is the number of elements in `nums`, as it requires just one iteration over the array elements.

Example Walkthrough

Let's assume `nums` is `[3, 8, 2]` and `maximumBit` is `3`. The goal is to find a non-negative integer `k` for each query, which gives us the maximum possible value when XORed with the cumulative XOR of all elements in `nums`, then remove the last element from `nums`.

We'll start by calculating the cumulative XOR of the entire array `nums`:

```
xs = reduce(xor, nums)  # xs = 3 ^ 8 ^ 2 = 9
```

Next, we'll prepare the mask:

```
mask = (1 << maximumBit) - 1  # mask = (1 << 3) - 1 = 7 (binary 111)
```

Now, let's walk through the steps for each query in our example:

1. **First Query:**

   Starting with the full array `[3, 8, 2]`, our cumulative XOR, `xs` is `9` (binary `1001`), and the mask is `7` (binary `0111`).

   ```
   k = xs ^ mask  # k = 9 ^ 7 = 14 (binary 1110)
   ```

   We append `14` to our answers array, `ans = [14]`, and remove the last element from `nums`.

   Update `xs`:

   ```
   xs ^= nums[-1]  # xs ^= 2 (xs was 9, now it is 11, binary 1011)
   nums.pop()  # Updated nums = [3, 8]
   ```

2. **Second Query:**

   With the updated array `[3, 8]`, our new `xs` is `11` (binary `1011`).

   ```
   k = xs ^ mask  # k = 11 ^ 7 = 12 (binary 1100)
   ```

   We append `12` to our answers array, `ans = [14, 12]`, and remove the last element from `nums`.

   Update `xs`:

   ```
   xs ^= nums[-1]  # xs ^= 8 (xs was 11, now it is 3, binary 0011)
   nums.pop()  # Updated nums = [3]
   ```

3. **Third Query:**

   The updated array now is just `[3]`, `xs` is `3` (binary `0011`).

   ```
   k = xs ^ mask  # k = 3 ^ 7 = 4 (binary 0100)
   ```

   We append `4` to our answers array, `ans = [14, 12, 4]`, and now `nums` becomes empty.

Since the `nums` array is now empty, our process stops here. We return the results array `ans`:

```
return ans  # [14, 12, 4]
```

In summary, our small example outputs the results `[14, 12, 4]` for the respective queries. Each number represents the maximum value achievable for `k` by XORing with the cumulative XOR of the remaining array `nums` at each step.

Python Solution

```python
from functools import reduce # Import the reduce function from the functools module
from operator import xor # Import the xor function from the operator module
from typing import List # Import List type for type hinting

class Solution:
    def getMaximumXor(self, nums: List[int], maximumBit: int) -> List[int]:
        # Intialize an empty list to store the answers
        answers = []

        # Perform XOR on all elements in nums to get the initial xor_sum
        xor_sum = reduce(xor, nums)

        # Create a mask which will have ones for the number of maximumBit bits
        mask = (1 << maximumBit) - 1

        # Iterate over the nums list in reverse order
        for num in reversed(nums):
            # Calculate k as XOR of xor_sum with mask to get the maximum XOR value
            k = xor_sum ^ mask
            # Append the resultant k to the answers list
            answers.append(k)
            # Update the xor_sum by XORing it with the current number
            xor_sum ^= num

        # Return the list of answers
        return answers

```

Java Solution

```java
class Solution {
    public int[] getMaximumXor(int[] nums, int maximumBit) {
        // Initialize 'xorSum' to store the cumulative XOR of all elements in 'nums'.
        int xorSum = 0;

        // Calculate the cumulative XOR for all the elements in 'nums'.
        for (int num : nums) {
            xorSum ^= num;
        }

        // Calculate the mask by considering the maximum number of bits.
        // The mask will be a number with 'maximumBit' number of 1s in binary representation.
        int mask = (1 << maximumBit) - 1;

        // The length of the input array.
        int length = nums.length;

        // Initialize an array 'maximumXors' to hold the maximum XOR for each element in reverse order.
        int[] maximumXors = new int[length];

        // Iterate over the 'nums' array in reverse, computing the maximum XOR for each prefix.
        for (int i = 0; i < length; ++i) {
            // XOR the current cumulative sum with the mask to find the maximum XOR value.
            int maxXor = xorSum ^ mask;

            // Add the maximum XOR value to the result array.
            maximumXors[i] = maxXor;

            // Update the 'xorSum' to remove the contribution of the current element because
            // we are moving from the end of the array towards the start.
            int currentNum = nums[length - i - 1];
            xorSum ^= currentNum;
        }

        // Return the array of maximum XOR values.
        return maximumXors;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<int> getMaximumXor(vector<int>& nums, int maximumBit) {
        // Initialize a variable to store the cumulative XOR of all numbers
        int cumulativeXor = 0;

        // Calculate the cumulative XOR for the whole array
        for (int num : nums) {
            cumulativeXor ^= num;
        }

        // Compute the bitmask with all bits set to 1 up to the maximumBit
        int mask = (1 << maximumBit) - 1;

        // Get the size of the input array
        int n = nums.size();

        // Initialize the answer vector with the same size as the input array
        vector<int> answer(n);

        // Iterate over the array to find the maximum XOR for each element
        for (int i = 0; i < n; ++i) {
            // Compute the XOR of the current xor state with the mask to find the maximum XOR
            int maxXor = cumulativeXor ^ mask;
            answer[i] = maxXor;

            // Update the cumulative XOR by removing the current element (from the end)
            cumulativeXor ^= nums[n - i - 1];
        }

        // Return the vector containing the maximum XOR values
        return answer;
    }
};

```

Typescript Solution

```typescript
function getMaximumXor(nums: number[], maximumBit: number): number[] {
    // Initialize the cumulative XOR variable.
    let cumulativeXor = 0;

    // Compute the cumulative XOR for all numbers in the array.
    for (const num of nums) {
        cumulativeXor ^= num;
    }

    // Calculate the mask to get the maximum XOR by setting maximumBit bits to 1
    const mask = (1 << maximumBit) - 1;

    // Determine the number of elements in the numbers array.
    const length = nums.length;

    // Initialize the answer array with the same length as the input array.
    const answer = new Array(length);

    // Iterate over the numbers to calculate the maximum XOR for each number
    // in reverse order.
    for (let i = 0; i < length; ++i) {
        // Find the current number by indexing from the end of the nums array.
        const currentNum = nums[length - i - 1];

        // Calculate the maximum XOR for the current number as per the problem statement.
        let maxXor = cumulativeXor ^ mask;

        // Store the maximum XOR in the answer array.
        answer[i] = maxXor;

        // Update the cumulative XOR by removing the effect of the current number.
        cumulativeXor ^= currentNum;
    }

    // Return the answer array containing maximum XORs.
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several factors:

1. The `reduce(xor, nums)` operation, which computes the XOR of all elements in the `nums` list. This operation takes `O(n)` time, where `n` is the length of `nums`.
2. The loop that reverses `nums` and computes the maximum XOR for each prefix. The reversal is `O(n)` due to the slicing operation `nums[::-1]`, and the loop runs `n` times. Inside the loop, the XOR computation and the assignment `xs ^= x` both take constant time `O(1)`.

Therefore, the total time complexity is `O(n)` due to the linear scan through all elements of `nums`.

Space Complexity

The space complexity of the code is also influenced by several parts:

1. The `ans` list that stores the maximum XOR values for each prefix, which will contain `n` elements at the end of the execution. This contributes `O(n)` to the space complexity.
2. The constants `xs` and `mask` use `O(1)` space.
3. The reversed nums `nums[::-1]` creates a new list, which also takes `O(n)` space.

However, the space used for input (such as `nums`) is typically not counted in space complexity analysis, as this is considered space that the algorithm needs to read its input rather than working space used by the algorithm. With that convention, the auxiliary space complexity of this algorithm is `O(n)`, owing to the `ans` list. If you do include the space taken by `nums[::-1]`, the space complexity would still be `O(n)`.

In conclusion, the time complexity of the code is `O(n)`, and the space complexity of the code is `O(n)`.