1310. XOR Queries of a Subarray

Medium | Bit Manipulation | Array | Prefix Sum

Leetcode Link

Problem Description

In this problem, we are presented with two arrays: one called `arr`, which holds positive integers, and another called `queries`. Each element in `queries` is a pair `[left_i, right_i]`, where `left_i` and `right_i` are indices into the `arr`. For each query pair, we need to calculate the XOR (exclusive OR) of all elements in `arr` between these two indices, inclusive. The XOR of a sequence of numbers is a binary operation that takes two bits, returning 0 if the bits are the same and 1 if they are different. For a sequence of numbers, the XOR is applied in a pairwise fashion from left to right.

The XOR operation has a unique property where `X XOR X` equals 0 for any number `X`, and `X XOR 0` equals `X`. Another important property is that XOR operations can be performed in any order due to their associativity.

Therefore, the task is to return a new array, `answer`, where each element `answer[i]` is the result of the corresponding XOR operation from the `i`th query.

Intuition

The straightforward approach to solving this problem would involve running through each query, and performing the XOR operation across the range of indices specified for every query. This would result in an algorithm that runs in O(n*m) time, where n is the length of `arr` and m is the number of queries, which can be very slow if both are large.

The solution code uses a clever insight combined with a property of the XOR operation to avoid recomputing the XOR for overlapping ranges and to handle each query in constant time, resulting in a much more efficient algorithm.

The intuition behind the solution lies in precomputing a list `s` that holds the cumulative XOR up to each index in `arr`. The cumulative XOR `s[i]` at index `i` will be the XOR of all elements from `arr[0]` to `arr[i - 1]`. We start the accumulation with an initial value of 0 (since `X XOR 0` equals `X` for any number `X`).

Once we have this precomputed cumulative XOR array, to find the XOR for any range `[left_i, right_i]`, we can use the following observation: The XOR of a range from `left_i` to `right_i` can be calculated by XOR-ing the cumulative XORs up to `right_i` and `left_i - 1`. This is because the cumulative XOR up to `right_i` includes all the elements we want but also includes all the elements before `left_i` that we don't want. By XORing with the cumulative XOR up to `left_i - 1`, we cancel out the unwanted elements due to the property that `X XOR X` equals 0.

Therefore, the answer for each query `i` is `s[right_i + 1] XOR s[left_i]`. The reason `right_i + 1` is used instead of `right_i` is to correctly handle the end index, because the cumulative XOR `s[i]` is calculated up to, but not including, index `i`.

Using this approach, we reduce the time complexity of answering all the queries to O(n + m), which is much more efficient, especially when dealing with a large number of queries.

Solution Approach

The solution for the XOR query problem is built upon the efficient computation of multiple range XOR queries over an immutable array. To achieve this, we use a prefix XOR and a simple array traversal. Here's a step-by-step walk-through of the implementation:

1. **Prefix XOR Computation**: We initialize an array, `s`, to store the prefix XOR values. Prefix XOR at position `i` represents the XOR of all elements from the beginning of the array `arr` up to the `i-1`th position. We use Python's `accumulate` function from the `itertools` module with the bitwise XOR operator, passing an `initial` value of 0 to include the XOR for the element at index 0 as well.

2. **Iterating Over Queries**: We loop through each query in `queries`, which contains pairs of `[l, r]` representing the range of indices `left` to `right`.

3. **Computing Range XOR**: For each query `[l, r]`, we compute the XOR of the range by XOR-ing the prefix XOR values `s[r + 1]` and `s[l]`. The reason we do `s[r + 1]` instead of `s[r]` is because our prefix accumulation starts with a 0, offsetting each index by 1. The value `s[r + 1]` thus contains the XOR of all elements from 0 to `r` inclusive. Since XOR is an associative and commutative operation, we can remove the prefix up to `l - 1` (contained in `s[l]`) from this cumulative value to get the XOR of elements from `l` to `r`.

4. **Appending Results**: The result for the current query is appended to the list `answer`. This list will eventually contain the XOR results for all the queries.

The final solution code thus looks as follows:

```
class Solution:
    def xorQueries(self, arr: List[int], queries: List[List[int]]) -> List[int]:
        # Step 1: Compute the prefix XOR list `s`
        s = list(accumulate(arr, xor, initial=0))

        # Step 2 & 3: Process each query and compute the range XOR
        # Step 4: Collect range XOR results into the final answer list
        return [s[r + 1] ^ s[l] for l, r in queries]
```

By precomputing the cumulative XOR up to each point in the array and cleverly using the XOR properties, this approach answers each query in constant time after an initial preprocessing step, making the solution highly efficient.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose the input array `arr = [4, 8, 2, 10]` and the queries are `[[0,1], [1,3], [2,3] ]`.

1. **Prefix XOR Computation**: We first build the cumulative `s` array that holds the XOR from start up to index `i-1` as follows:

   - `s[0]` is initialized to 0.
   - `s[1] = arr[0] XOR s[0] = 4 XOR 0 = 4`
   - `s[2] = arr[1] XOR s[1] = 8 XOR 4 = 12`
   - `s[3] = arr[2] XOR s[2] = 2 XOR 12 = 14`
   - `s[4] = arr[3] XOR s[3] = 10 XOR 14 = 4`

   So, our cumulative XOR array, `s`, now looks like `[0, 4, 12, 14, 4]`.

2. **Iterating Over Queries**: Now let's process each of the queries.

3. **Computing Range XOR**:

   - For the first query `[0,1]`, we calculate the result as `s[1+1] ^ s[0]` which is `s[2] ^ s[0] = 12 XOR 0 = 12`.
   - For the second query `[1,3]`, the result is `s[3+1] ^ s[1]` which is `s[4] ^ s[1] = 4 XOR 4 = 8`.
   - For the third query `[2,3]`, the result is `s[3+1] ^ s[2]` which is `s[4] ^ s[2] = 4 XOR 12 = 8`.

4. **Appending Results**: We append each result to the `answer` list, which at the end of the iteration contains `[12, 0, 8]`.

The output for the provided example is therefore `[12, 0, 8]`, corresponding to the XOR of elements from:

- Indices `0` to `1`: XOR of `4` and `8` is `12`
- Indices `1` to `3`: XOR of `8`, `2`, `10` is `0`
- Indices `2` to `3`: XOR of `2` and `10` is `8`

By using a precomputed cumulative XOR array and understanding the associativity of the XOR operation, this approach efficiently computes the result for all queries without having to recompute the XOR from scratch for each query range.

Python Solution

```
from itertools import accumulate
from operator import xor

class Solution:
    def xorQueries(self, arr: List[int], queries: List[List[int]]) -> List[int]:
        # Calculate the accumulated XOR values for the entire array.
        # The initial value is 0 because 0 XOR with any number returns that number.
        accumulated_xor = list(accumulate(arr, xor, initial=0))

        # Process each query to get the XOR from arr[l] to arr[r].
        # We utilize the property: XOR from arr[l] to arr[r] is
        # accumulated_xor[r + 1] XOR accumulated_xor[l].
        results = [accumulated_xor[r + 1] ^ accumulated_xor[l] for l, r in queries]

        return results
```

Java Solution

```
class Solution {
    // Function to perform XOR queries on an array
    public int[] xorQueries(int[] arr, int[][] queries) {
        // Length of the original array
        int n = arr.length;

        // Initialize a prefix XOR array with an additional element to handle zero-indexing
        int[] prefixXOR = new int[n + 1];

        // Populate the prefix XOR array where each element is the XOR of all elements before it
        for (int i = 1; i <= n; ++i) {
            prefixXOR[i] = prefixXOR[i - 1] ^ arr[i - 1];
        }

        // Number of queries
        int m = queries.length;

        // Initialize the array to hold the results of the queries
        int[] answer = new int[m];

        // Iterate through each query
        for (int i = 0; i < m; ++i) {
            // Get the left and right indices of the current query
            int left = queries[i][0], right = queries[i][1];

            // Calculate the XOR for the given range by using the prefix XOR array
            // s[r + 1] gives the XOR from arr[0] to arr[r] inclusive
            // s[l] gives the XOR from arr[0] to arr[l - 1] exclusive.
            // XORing these two gives the XOR from arr[l] to arr[r] inclusive, which is the answer for this query
            answer[i] = prefixXOR[right + 1] ^ prefixXOR[left];
        }

        // Return the array containing the result of each query
        return answer;
    }
}
```

C++ Solution

```
#include <vector>
#include <cstring>

using namespace std;

class Solution {
public:
    // Function that returns the result of XOR queries on an array.
    vector<int> xorQueries(vector<int>& arr, vector<vector<int>>& queries) {
        // Find the size of the input array.
        int arraySize = arr.size();

        // Create an array to store the prefix XOR up to each element.
        int prefixXor[arraySize + 1];

        // Initialize the prefixXor array to 0.
        memset(prefixXor, 0, sizeof(prefixXor));

        // Compute the prefix XOR array, where prefixXor[i] stores the XOR of all elements up to index i-1.
        for (int i = 1; i <= arraySize; ++i) {
            prefixXor[i] = prefixXor[i - 1] ^ arr[i - 1];
        }

        // Create a vector to store the answers to the queries.
        vector<int> answers;

        // Iterate over each query.
        for (auto& query : queries) {
            // Extract the left and right indices from the current query.
            int leftIndex = query[0], rightIndex = query[1];

            // Calculate the XOR from leftIndex to rightIndex using the prefix XOR values.
            // The result of XOR from leftIndex to rightIndex is prefixXor[rightIndex + 1] ^ prefixXor[leftIndex]
            // because prefixXor[leftIndex] contains the XOR of all elements up to leftIndex - 1.
            answers.push_back(prefixXor[rightIndex + 1] ^ prefixXor[leftIndex]);
        }

        // Return the answers to the queries.
        return answers;
    }
};
```

Typescript Solution

```
// Function to perform XOR queries on an array.
// For every query, which consists of a pair `left, right`, the function
// calculates the XOR of elements from arr[left] to arr[right].
function xorQueries(arr: number[], queries: number[][]): number[] {
    const arrLength = arr.length; // Get the length of the array.

    // Create an array to store the prefix XORs with an additional 0 at the beginning.
    const prefixXOR: number[] = new Array(arrLength + 1).fill(0);

    // Calculate the prefix XOR values for the array.
    for (let i = 0; i < arrLength; ++i) {
        prefixXOR[i + 1] = prefixXOR[i] ^ arr[i];
    }

    // Initialize an array to hold the result of each query.
    const results: number[] = [];

    // Process each query and calculate the XOR for the given range.
    for (const [left, right] of queries) {
        // XOR between the prefix XORs gives the XOR of the range.
        results.push(prefixXOR[right + 1] ^ prefixXOR[left]);
    }

    // Return the array of results.
    return results;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code consists of two parts: the computation of the prefix xors and the processing of the queries.

1. Computing the prefix xors with `accumulate` is an O(n) operation, where `n` is the number of elements in the array `arr`. This is because it processes each element of the array exactly once with the `xor` operation.

2. The list comprehension iterates through each query and performs a constant-time xor operation. If we have `m` queries, the time to process all queries will be O(m).

Combining both parts, the overall time complexity is O(n + m), where `n` is the length of `arr` and `m` is the number of queries.

Space Complexity

The space complexity of the provided code can be analyzed as follows:

1. The space taken by the prefix xors list `s` is O(n), where `n` is the size of the input array `arr`.

2. The space for the output list is O(m), where `m` is the number of queries.

Therefore, the total space complexity is O(n + m), which accounts for the space occupied by the prefix xors and the output list.