2588. Count the Number of Beautiful Subarrays

Medium | Bit Manipulation | Array | Hash Table | Prefix Sum

Problem Description

Given an array `nums`, the task is to count the number of "beautiful subarrays". A subarray is considered beautiful if you can perform a series of operations that make all its elements equal to zero. In one operation, you can pick any two different indices `i` and `j` and a non-negative integer `k`. The `kth` bit in the binary representation of both `nums[i]` and `nums[j]` should be a `1`. You then subtract `2^k` from both `nums[i]` and `nums[j]`. A subarray, in this context, is defined as a contiguous non-empty sequence of elements within the array `nums`.

Intuition

The intuition behind the solution involves understanding the properties of XOR and bit manipulation. The important observation is that a subarray can be made all zeros if, for each bit position, the total count of 1s is even. This is because the defined operation allows us to simultaneously subtract the value of that bit position from two numbers with a 1 in that same position, effectively reducing the count of 1s by two.

To track the number of 1s at each bit position across subarrays, we use the concept of prefix XOR. When we XOR a number with itself, it results in zero, and XORing a number with zero results in the original number. By applying prefix XOR as we traverse the array, we can determine if there is an equal number of 1s in each bit position between two indices: if the prefix XOR at two different indices is the same, then the subarray between them can be turned into an array of zeros.

We utilize a hash table to store the counts of each prefix XOR encountered. For every element in the array, we calculate the prefix XOR up to that element (`mask`) and check how many times this `mask` has occurred before (using the counter `cnt`). These counts correspond to the number of subarrays ending at the current index that can be converted into beautiful arrays. Each time we visit an element, we update the count for its corresponding `mask`. The sum of all these counts gives us the number of beautiful subarrays in the original array.

Solution Approach

The solution approach leverages a hash table to store the occurrence count of each prefix XOR value encountered while traversing the array. Specifically, we use Python's `Counter` data structure for this purpose. The steps can be broken down into the following algorithm:

1. Initialize a `Counter` named `cnt` with a starting count of `1` for a prefix XOR of `0` to handle the case where a subarray starting from index `0` can be made beautiful.
2. Initialize variables `ans` to store the count of beautiful subarrays and `mask` to keep track of the current prefix XOR value, both set initially to `0`.
3. Iterate through each element `x` in the input array `nums`. a. XOR the current element `x` with `mask` to update `mask` to the new prefix XOR value (this keeps track of the cumulative XOR from the start of the array to the current index). b. Add the count of the current `mask` from the `cnt` to `ans`. This total represents the number of previous subarrays that had the same XOR value, which means the subarray from the end of any of those to the current index can be made beautiful. c. Increase the count of the current `mask` in `cnt` by 1 to account for the new occurrence of this XOR value.
4. Continue this process until the end of the array.
5. After completing the iteration, `ans` contains the total count of beautiful subarrays.

This approach is efficient since it only needs to pass through the array once, with computations at each index being constant time due to the use of XOR and hash table lookups. Thus, the overall time complexity is O(n), where n is the number of elements in the input array `nums`.

The key to this solution is the use of prefix XOR to effectively identify subarrays that can be transformed to zero, based on the property that the XOR of any number with itself is zero and XOR with zero preserves the number. By applying this logic, and by keeping track of the number of occurrences of each XOR value seen so far, we can count all beautiful subarrays without needing to examine each potential subarray explicitly.

Example Walkthrough

Let's assume we have an array `nums = [3, 4, 5, 2, 4]`, and we want to apply the solution approach to count the number of beautiful subarrays. Here's a step-by-step walkthrough:

1. Initialize a `Counter`, `cnt`, with a count of `1` for a prefix XOR of `0` and set `ans = 0` and `mask = 0`.
2. Starting with the first element:
   - `mask` XOR `3` = `3` (binary `011`). We update `ans` by adding the occurrence of `mask` (which is `0`, as `3` has not been encountered before), and then increment the count of `mask = 3` in `cnt`.
3. Moving to the second element:
   - `mask` (which was `3`) XOR `4` = `7` (binary `111`). We update `ans` by adding the occurrence of `mask = 7` (again `0`), and increment the count of `mask` in `cnt`.
4. For the third element:
   - `mask = 7` XOR `5` = `2` (binary `010`). We update `ans` with the count of `mask = 2` (which is `0`) and increment the count of `mask` in `cnt`.
5. For the fourth element:
   - `mask = 2` XOR `2` = `0`. This is interesting because the prefix XOR is now `0`, meaning a subarray from the start can be made beautiful. We update `ans` with the count of `mask` (which is `1`, from the initial `cnt`), and increment the count of `mask = 0` in `cnt`.
6. For the fifth element:
   - `mask = 0` XOR `4` = `4`. We update `ans` by adding the occurrence of `mask = 4` (which is `0`), and increment the count of `mask` in `cnt`.

The counts in the `Counter` at the end of this traversal are `cnt = {0: 2, 3: 1, 7: 1, 2: 1, 4: 1}`, and `ans = 1`, representing the single beautiful subarray `[3, 4, 5, 2]` which can be made all zeros by the series of operations specified.

This small example demonstrates how the solution works. As we traverse `nums`, we cumulatively XOR the numbers and use a hash table to keep track of how many times we've seen each XOR result. When we encounter the same XOR result again, it means we have found a subarray that can be made beautiful. The final count of beautiful subarrays in this example is `1`.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def beautifulSubarrays(self, nums: List[int]) -> int:
        # Initialize a counter to keep track of the frequency of XOR accumulative sums.
        # The counter is initialized with the accumulative sum '0' having a count of 1.
        xor_freq_counter = Counter({0: 1})

        # Initialize 'ans' to count the number of beautiful subarrays.
        # Initialize 'xor_accumulative' to store the accumulative XOR sum.
        ans, xor_accumulative = 0, 0

        # Iterate over the elements in nums.
        for num in nums:
            # Update the accumulative XOR sum with the current number.
            xor_accumulative ^= num

            # If the XOR sum has been seen before, it means there is a subarray
            # with an even number of 1's, so add the previous count to 'ans'.
            ans += xor_freq_counter[xor_accumulative]

            # Update the frequency counter for the current XOR accumulative sum.
            xor_freq_counter[xor_accumulative] += 1

        # Return the total count of beautiful subarrays.
        return ans

# Note: The List type requires importing from the typing module in Python 3.
# from typing import List
```

Java

```java
class Solution {
    public long beautifulSubarrays(int[] nums) {
        // Map to store the count of each prefix XOR value encountered
        Map<Integer, Integer> prefixXorCount = new HashMap<>();

        // Initialize the map with the base case where there is no prefix (XOR value is 0)
        prefixXorCount.put(0, 1);

        // To keep track of the total number of beautiful subarrays
        long totalCount = 0;

        // To store the XOR of all numbers from the start of the array to the current index
        int currentXor = 0;

        for (int num : nums) {
            // Update the currentXor with the XOR of the current number
            currentXor ^= num;

            // Increment the totalCount by the number of times this XOR value has been seen before
            totalCount += prefixXorCount.getOrDefault(currentXor, 0);

            // Increment the count of the currentXor value in our prefixXorCount map
            prefixXorCount.merge(currentXor, 1, Integer::sum);
        }

        // Return the total number of beautiful subarrays
        return totalCount;
    }
}
```

C++

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

class Solution {
public:
    long long beautifulSubarrays(vector<int>& nums) {
        // This unordered_map will store the frequency of each XOR value encountered.
        unordered_map<int, int> countMap{{0, 1}};
        // This variable will store the total number of beautiful subarrays.
        long long totalBeautifulSubarrays = 0;
        // The mask will hold the cumulative XOR value as we iterate through the vector.
        int cumulativeXor = 0;

        // Iterate over the vector to calculate the beautiful subarrays.
        for (int num : nums) {
            // Compute the cumulative XOR up to the current number.
            cumulativeXor ^= num;
            // Add the current count of the cumulativeXor to our answer, as any previous occurrence
            // of the same cumulativeXor indicates a subarray with an even number of each integer.
            totalBeautifulSubarrays += countMap[cumulativeXor];
            // Increase the count of the cumulativeXor in our map.
            ++countMap[cumulativeXor];
        }
        // Return the final count of beautiful subarrays.
        return totalBeautifulSubarrays;
    }
};
```

TypeScript

```typescript
// Function to count the number of beautiful subarrays in an array of numbers.
// A beautiful subarray is defined as the one where the bitwise XOR of all elements is 0.
function beautifulSubarrays(nums: number[]): number {
    // Initialize a Map to store the frequency of XORed results.
    const xorFrequency = new Map<number, number>();
    xorFrequency.set(0, 1); // Set the frequency for 0 as 1 to account for the initial state.

    // Variable to store the total count of beautiful subarrays.
    let totalBeautifulSubarrays = 0;

    // Variable to keep track of the cumulative XOR result as we iterate through the array.
    let cumulativeXor = 0;

    // Loop through each number in the given array.
    for (const num of nums) {
        // Update the cumulative XOR.
        cumulativeXor ^= num;

        // If the current cumulative XOR result has been seen before, add its frequency to the total count.
        totalBeautifulSubarrays += xorFrequency.get(cumulativeXor) || 0;

        // Update the frequency of the current cumulative XOR result in the map.
        // If it doesn't exist, initialize it with 0 and then add 1.
        xorFrequency.set(cumulativeXor, (xorFrequency.get(cumulativeXor) || 0) + 1);
    }

    // Return the total count of beautiful subarrays found.
    return totalBeautifulSubarrays;
}
```


```python
from collections import Counter

class Solution:
    def beautifulSubarrays(self, nums: List[int]) -> int:
        # Initialize a counter to keep track of the frequency of XOR accumulative sums.
        # The counter is initialized with the accumulative sum '0' having a count of 1.
        xor_freq_counter = Counter({0: 1})

        # Initialize 'ans' to count the number of beautiful subarrays.
        # Initialize 'xor_accumulative' to store the accumulative XOR sum.
        ans, xor_accumulative = 0, 0

        # Iterate over the elements in nums.
        for num in nums:
            # Update the accumulative XOR sum with the current number.
            xor_accumulative ^= num

            # If the XOR sum has been seen before, it means there is a subarray
            # with an even number of 1's, so add the previous count to 'ans'.
            ans += xor_freq_counter[xor_accumulative]

            # Update the frequency counter for the current XOR accumulative sum.
            xor_freq_counter[xor_accumulative] += 1

        # Return the total count of beautiful subarrays.
        return ans

# Note: The List type requires importing from the typing module in Python 3.
# from typing import List
```


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 code iterates through the array exactly once with a sequence of `O(1)` operations within the loop, such as XOR operation, dictionary lookup, and dictionary update.

The space complexity of the code is also `O(n)` because the `Counter` object `cnt` potentially stores a unique entry for every prefix XOR result in the `nums` array. In the worst case, this could be as many as `n` unique entries, if every prefix XOR results in a different value.