1863. Sum of All Subset XOR Totals

Problem Description

The problem asks us to calculate the sum of all XOR totals for every subset of a given array `nums`. The XOR total for an array is the result of applying the XOR operation to all elements within it. The XOR operation is a binary operation that takes two bits and returns `0` if they are the same and `1` if they are different.

A subset of an array can be formed by eliminating some or none of the elements in the array. Since subsets with identical elements count as distinct if they originate from different steps in the subset forming process, we need to account for each subset instance separately.

For example, if we start with an array `[2,5,6]`, we need to consider all its subsets, including the empty subset, calculate the XOR total for each, and sum them all.

The challenge thus lies in generating all possible subsets efficiently and computing their XOR totals.

Intuition

The key to solving this problem is recognizing that we can use recursive depth-first search (DFS) to explore all possible subsets of the array. In this approach, at each step, we choose whether to include or exclude the current element, and then recurse accordingly to handle the next element until no more elements are left to be considered.

Our recursive function keeps track of two pieces of information as it progresses: the depth and the current XOR value of the elements chosen so far (`prev`).

- The `depth` tells us how far we've traversed in the array and which element to consider next.
- The `prev` variable holds the accumulated XOR of the currently forming subset.

Since every function call represents a unique path of decisions (inclusion or exclusion of elements), the subsets are generated implicitly as part of the exploration process. The recursive function visits and calculates the XOR totals for each subset, and the sum of these totals is accumulated in a global variable `self.res`.

Hence, we arrive at an elegant solution where the sum total can be computed systematically through a series of recursive calls, without ever having to explicitly list out all subsets, which would be computationally expensive and memory-intensive.

Solution Approach

The solution provided is a recursive, depth-first search algorithm. Let's break down how it is implemented:

1. A helper function named `dfs` is defined within the `subsetXORSum` method of the `Solution` class. This `dfs` function has three parameters: `nums` (the input array), `depth` (the current depth or index within the input array), and `prev` (the XOR total of elements included in the current subset).

2. The main objective of the `dfs` function is to iterate through all possible subsets, calculate their XOR total, and add it to the cumulative sum `self.res`.

3. On each invocation of `dfs`, the current XOR total `prev` is added to `self.res`. It represents the XOR total of the current subset.

4. The `dfs` function then loops through the remaining elements in `nums` starting from the current `depth`. For each subsequent element, it toggles (XORs) the current element with `prev` to include it in the subset XOR total and calls itself recursively with the updated `depth` and `prev`.

5. After the recursive call returns, it toggles (XORs) the element again to backtrack, effectively removing the element from the current subset before proceeding to the next iteration of the loop.

6. The recursion starts off with the initial call `dfs(nums, 0, 0)`, where `depth` is 0 (start of the array) and `prev` is 0 (the initial XOR total).

7. Each recursive call either includes the current element in the subset or moves past it (as the subsequent recursive call is after the XOR operation). This way, all possible subsets, including the empty subset, are implicitly generated and considered.

8. The global variable `self.res` is used to accumulate the sum of all XOR totals. It is initialized to 0 before starting the recursion.

9. After exploring all subsets and calculating the sums of their XOR totals, the `subsetXORSum` method returns the total sum (`self.res`) as the final answer.

This implementation efficiently traverses the "subset-space" of the input array, considering all possible combinations without duplicate effort or storing intermediate subsets, making it both time and space-efficient. The use of recursion elegantly captures the problem's substructure, with the base case naturally handled by the loop termination and the backtrack within the `dfs` function.

Example Walkthrough

Let's illustrate the solution approach using a simple example. Suppose we have the input array `nums = [1, 3]`. We want to find the sum of XOR totals for every subset.

Step 1: Initiating a DFS call stack with `dfs(nums, 0, 0)` which means starting with `depth` 0 and `prev` XOR total as 0.

Step 2: On the first call, `prev` is added to `self.res`. Initially, both are 0.

Step 3: We now consider the element at index 0, which is `1`. We include it by XORing `prev` (0) with `1`, getting a new `prev` which is `1`.

Step 4: We make a recursive call to `dfs(nums, 1, 1)` with the new `prev`. Upon this invocation, `prev` is once again added to `self.res`. At this point, `self.res = 0 + 1 = 1`.

Since we don't have more elements to process (as `depth` equals the length of the num array), this branch of recursion ends here, effectively considering the subset `[1]`.

Step 5: Backtrack and consider not including `1` anymore. We return to the earlier state of the algorithm.

Step 6: Now we consider the next element at index 1, which is `3`. We ignore the first element and make a recursive call `dfs(nums, 1, 0)` since we're moving forward without including the first element. After this invocation, we add `prev` to `self.res` again. Now, `self.res = 1 + 0 = 1`.

Step 7: Include element `3` in the subset by XORing with current `prev` (0 XOR 3). We make another recursive call `dfs(nums, 2, 3)`. Subsequently, `prev` is added to `self.res`. After this, `self.res = 1 + 3 = 4`.

This path of recursion ends here too, now having considered the subset `[3]`.

Step 8: With the current depth reaching the end of the array, we have considered all paths from the root of the recursion and have these results:

- Empty subset `[]` contributes `0` (initial `self.res`).
- Subset `[1]` contributes `1`.
- Subset `[3]` contributes `3`.

Adding them up, we have `self.res = 0 + 1 + 3 = 4`.

Summing Up:

- The process involves considering all subsets by recursively including and not including each element.
- In this example, we are able to consider all subsets: `[]`, `[1]`, `[3]`, `[1,3]`.
- We do not directly calculate the XOR total for the subset `[1,3]` because by the time we've processed all elements, it has already been added incrementally through recursive calls.

Therefore, for the array `[1, 3]`, the final `self.res` would be the sum of XOR of all subset totals:

- `[1]` → 1 (0 XOR 1)
- `[3]` → 3 (0 XOR 3)
- `[1,3]` → 2 (1 XOR 3)

Adding them with the empty subset's XOR total of 0:

`self.res` = 1 + 3 + 2 = `6`

The final answer is 6, and this demonstrates how the DFS algorithm navigates through the input array to find the required sum of XOR totals for every subset.

Python Solution

```python
class Solution:
    def subsetXORSum(self, nums: List[int]) -> int:
        # Helper function that performs a Depth First Search (DFS)
        # to traverse all subsets and calculate their XOR sum.
        # `current_index` is the current starting point in the array for choosing the next element.
        # `current_xor` is the cumulative XOR value of the current subset.
        def dfs(current_index, current_xor):
            # Increment the result with the cumulative XOR of the current subset.
            self.result += current_xor

            # Iterate over the remaining elements to explore further subsets.
            for i in range(current_index, len(nums)):
                # Include the next number in the subset and calculate the new XOR.
                next_xor = current_xor ^ nums[i]
                # Recurse with the updated XOR value and the next starting index.
                dfs(i + 1, next_xor)

        # Initialize the result variable as 0.
        self.result = 0
        # Start the DFS with initial index 0 and initial XOR value 0.
        dfs(0, 0)
        # Return the accumulated result after traversing all subsets.
        return self.result
```

Java Solution

```java
class Solution {
    // To keep track of the accumulated result of all subset XORs.
    private int totalXorSum;

    // Public method to initiate the XOR sum calculation process.
    public int subsetXORSum(int[] nums) {
        // Start the depth-first search (DFS) from the beginning of the array with initial XOR sum as 0.
        dfs(nums, 0, 0);
        return totalXorSum;
    }

    // Helper method to perform depth-first search (DFS) for subsets and calculate XOR sum.
    private void dfs(int[] nums, int index, int currentXor) {
        // Add the current XOR sum of the subset to the total result.
        totalXorSum += currentXor;
        // Proceed to find other subsets and calculate their XOR.
        for (int i = index; i < nums.length; ++i) {
            // Include the next number in the subset and update the current XOR.
            currentXor ^= nums[i];
            // Recurse with the new subset XOR and the next index.
            dfs(nums, i + 1, currentXor);
            // Exclude the last number from the current subset to backtrack for the next iteration.
            currentXor ^= nums[i];
        }
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric> // for std::accumulate

// Function prototype declarations
void dfs(const std::vector<int>& nums, int index, int currentXOR, std::vector<int>& results);
int subsetXORSum(const std::vector<int>& nums);

/**
 * Calculates the sum of all subset XOR totals from the given vector.
 * @param nums - Vector of integers to process.
 * @return - The sum of all subset XOR totals.
 */
int subsetXORSum(const std::vector<int>& nums) {
    std::vector<int> results; // To store XOR of all subsets
    int currentXOR = 0; // Current XOR value at any point of DFS traversal
    dfs(nums, 0, currentXOR, results);
    // Sum up and return all XOR values from the results vector
    return std::accumulate(results.begin(), results.end(), 0);
}

/**
 * Depth-first search to explore all subsets and calculate their XOR.
 * @param nums - The vector of integers to generate subsets from.
 * @param index - The current index in the `nums` vector.
 * @param currentXOR - The current XOR value.
 * @param results - Vector to store XOR of all subsets.
 */
void dfs(const std::vector<int>& nums, int index, int currentXOR, std::vector<int>& results) {
    results.push_back(currentXOR); // Add the current XOR to the results vector

    // Explore further subsets by including elements one by one
    for (int i = index; i < nums.size(); ++i) {
        currentXOR ^= nums[i]; // Include nums[i] in the current subset and update the XOR
        dfs(nums, i + 1, currentXOR, results); // Recur for next elements

        // Backtrack: remove nums[i] from the current subset by XORing again
        currentXOR ^= nums[i]; // This reverts the currentXOR to its value before including nums[i]
    }
}

/**
 * Example usage
 */
int main() {
    std::vector<int> nums = {1, 2, 3};
    int totalSum = subsetXORSum(nums);
    // totalSum should now contain the sum of all subset XOR totals
    return 0;
}
```

Typescript Solution

```typescript
/**
 * Calculates the sum of all subset XOR totals from the given array.
 * @param {number[]} nums - Array of numbers to process.
 * @return {number} - The sum of all subset XOR totals.
 */
const subsetXORSum = (nums: number[]): number => {
    let results: number[] = []; // To store XOR of all subsets
    let currentXOR = 0; // Current XOR value at any point of DFS traversal
    dfs(nums, 0, currentXOR, results);
    // Sum up and return all XOR values from the results array
    return results.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
};

/**
 * Depth-first search to explore all subsets and calculate their XOR.
 * @param {number[]} nums - The array of numbers to generate subsets from.
 * @param {number} index - The current index in the `nums` array.
 * @param {number} currentXOR - The current XOR value.
 * @param {number[]} results - Array to store XOR of all subsets.
 */
function dfs(nums: number[], index: number, currentXOR: number, results: number[]): void {
    results.push(currentXOR); // Add the current XOR to the results array

    // Explore further subsets by including elements one by one
    for (let i = index; i < nums.length; i++) {
        currentXOR ^= nums[i]; // Include nums[i] in the current subset and update the XOR
        dfs(nums, i + 1, currentXOR, results); // Recur for next elements

        // Backtrack: remove nums[i] from the current subset and revert the XOR
        currentXOR ^= nums[i];
    }
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is $O(2^N)$, where N is the length of the `nums` array. This is because the function explores each possible subset of `nums` by using a depth-first search algorithm (DFS). On each call, it branches out to include or not include the current number into the subset, effectively generating all potential subsets and calculating their XOR. Since there are 2^N subsets for a set of size N, and each subset requires a constant amount of time to process, the time complexity is exponential.

Space Complexity

The space complexity of the given code is $O(N)$, which accounts for the call stack used by the DFS. In the worst case, the recursion goes as deep as the number of elements in the `nums` array, resulting in a call stack of depth N. No additional space is used other than the recursion stack and the variable `self.res` which is used to accumulate the result.