1863. Sum of All Subset XOR Totals

Array Math

Backtracking

Problem Description

Bit Manipulation

Easy

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

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

Combinatorics

elements count as distinct if they originate from different steps in the subset forming process, we need to account for each subset instance separately.

total for each, and sum them all. The challenge thus lies in generating all possible subsets efficiently and computing their XOR totals.

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

ntuition

of the array. In this approach, at each step, we choose whether to include or exclude the current element, and then recurse

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 key to solving this problem is recognizing that we can use recursive depth-first search (DFS) to explore all possible subsets

• 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.

accordingly to handle the next element until no more elements are left to be considered.

Hence, we arrive at an elegant solution where the sum total can be computed systematically through a series of recursive calls,

current subset before proceeding to the next iteration of the loop.

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

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

without ever having to explicitly list out all subsets, which would be computationally expensive and memory-intensive.

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).

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.

Example Walkthrough

effectively considering the subset [1].

• Empty subset [] contributes 0 (initial self.res).

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

self.res = 1 + 0 = 1.

• Subset [1] contributes 1.

• Subset [3] contributes 3.

Summing Up:

• [1] → 1 (0 XOR 1)

• [3] → 3 (0 XOR 3)

• $[1,3] \rightarrow 2 (1 \text{ XOR } 3)$

XOR totals for every subset.

Solution Implementation

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

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. After the recursive call returns, it toggles (XORs) the element again to backtrack, effectively removing the element from the

The dfs function then loops through the remaining elements in nums starting from the current depth. For each subsequent

total). 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.

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

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

(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

After exploring all subsets and calculating the sums of their XOR totals, the subsetXORSum method returns the total sum

function.

the problem's substructure, with the base case naturally handled by the loop termination and the backtrack within the dfs

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

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,

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,

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).

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

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:

• 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

The final answer is 6, and this demonstrates how the DFS algorithm navigates through the input array to find the required sum of

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

• In this example, we are able to consider all subsets: [], [1], [3], [1,3].

The process involves considering all subsets by recursively including and not including each element.

Subsequently, prev is added to self. res. After this, self. res = 1 + 3 = 4.

- Adding them with the empty subset's XOR total of 0: self.res = 1 + 3 + 2 = 6
- **Python** class Solution:

to traverse all subsets and calculate their XOR sum.

for i in range(current_index, len(nums)):

next_xor = current_xor ^ nums[i]

Helper function that performs a Depth First Search (DFS)

`current_xor` is the cumulative XOR value of the current subset.

Return the accumulated result after traversing all subsets.

// To keep track of the accumulated result of all subset XORs.

// Public method to initiate the XOR sum calculation process.

def subsetXORSum(self, nums: List[int]) -> int:

def dfs(current_index, current_xor):

self.result += current_xor

dfs(i + 1, next_xor)

return self.result

private int totalXorSum;

public int subsetXORSum(int[] nums) {

Java

C++

/**

/**

*/

#include <vector>

#include <numeric> // for std::accumulate

int subsetXORSum(const std::vector<int>& nums);

* @param nums - Vector of integers to process.

* @return - The sum of all subset XOR totals.

int subsetXORSum(const std::vector<int>& nums) {

* @param currentXOR - The current XOR value.

dfs(nums, 0, currentXOR, results);

* Calculates the sum of all subset XOR totals from the given vector.

int currentXOR = 0; // Current XOR value at any point of DFS traversal

std::vector<int> results; // To store XOR of all subsets

// Sum up and return all XOR values from the results vector

* Depth-first search to explore all subsets and calculate their XOR.

return std::accumulate(results.begin(), results.end(), 0);

* @param nums - The vector of integers to generate subsets from.

// Explore further subsets by including elements one by one

* @param index - The current index in the 'nums' vector.

* @param results - Vector to store XOR of all subsets.

for (int i = index; i < nums.size(); i++) {</pre>

// Function prototype declarations

class Solution {

Initialize the result variable as 0. self.result = 0 # Start the DFS with initial index 0 and initial XOR value 0. dfs(0, 0)

void dfs(const std::vector<int>& nums, int index, int currentXOR, std::vector<int>& results);

void dfs(const std::vector<int>& nums, int index, int currentXOR, std::vector<int>& results) {

currentXOR ^= nums[i]; // Include nums[i] in the current subset and update the XOR

currentXOR ^= nums[i]; // This reverts the currentXOR to its value before including nums[i]

results.push_back(currentXOR); // Add the current XOR to the results vector

dfs(nums, i + 1, currentXOR, results); // Recur for next elements

// Backtrack: remove nums[i] from the current subset by XORing again

Increment the result with the cumulative XOR of the current subset.

Include the next number in the subset and calculate the new XOR.

Recurse with the updated XOR value and the next starting index.

Iterate over the remaining elements to explore further subsets.

`current_index` is the current starting point in the array for choosing the next element.

```
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++) {</pre>
       // 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];
```

// Start depth-first search (DFS) from the beginning of the array with initial XOR sum as 0.

```
/**
```

* 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** /** * 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

currentXOR ^= nums[i]; // Include nums[i] in the current subset and update the XOR

`current_index` is the current starting point in the array for choosing the next element.

```
# 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
```

// Explore further subsets by including elements one by one

Helper function that performs a Depth First Search (DFS)

`current_xor` is the cumulative XOR value of the current subset.

to traverse all subsets and calculate their XOR sum.

dfs(nums, i + 1, currentXOR, results); // Recur for next elements

// Backtrack: remove nums[i] from the current subset and revert the XOR

for (let i = index; i < nums.length; i++) {</pre>

def subsetXORSum(self, nums: List[int]) -> int:

def dfs(current_index, current_xor):

currentXOR ^= nums[i];

Time Complexity The time complexity of the given code is 0(2^N), where N is the length of the nums array. This is because the function explores

Time and Space Complexity

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.

class Solution:

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.