1442. Count Triplets That Can Form Two Arrays of Equal XOR

Math

**Prefix Sum** 

Hash Table

# **Problem Description**

Medium

Intuition

Bit Manipulation Array

The LeetCode problem requires us to find the number of triplets (i, j, k) in an integer array where  $(0 \le i \le j \le k \le k]$ arr.length). For each triplet, we define two values, a and b, where a is the bitwise XOR of arr[i] through arr[j - 1], and b is the bitwise XOR of arr[j] through arr[k]. Our goal is to count how many such triplets yield a == b.

associative and commutative, which implies that the order of elements does not change the result of the XOR. Another insight is that XOR-ing a number with itself yields zero. By taking advantage of this, we can precompute the XOR of all elements up to k for every index k in the array, storing the results in a prefix XOR array pre. This precomputation allows us to find the XOR of any subarray in constant time. For any two indices i and j, the XOR of the subarray from i to j-1 can be obtained by  $pre[j] \land pre[i]$ . This is because pre[j] contains the XOR of all elements up to j-1

To solve this problem, we begin by thinking about the properties of XOR. A key insight is that the XOR operation is both

just the XOR of the subarray. The next step is to check every possible combination of (i, j, k). This requires three nested loops. For each triplet:

and pre[i] contains the XOR of all elements up to i-1. So, when we XOR these two, all the elements before i are nullified, leaving

1. We calculate a as the XOR of the subarray from i to j-1. 2. We calculate b as the XOR of the subarray from j to k. 3. We check if a is equal to b.

If a equals b, we increment our answer count (ans). After considering all possible triplets, ans will contain the total number of triplets for which a equals b.

The solution's time complexity is O(n^3) due to the use of three nested loops, which might not be the most efficient for large

XOR properties and precomputed prefix sums to solve the problem.

Calculate the length of the input array arr and denote it as n.

input arrays. However, for the purpose of understanding the problem, this brute force approach shows the direct application of

**Solution Approach** In the implementation of the solution for counting the triplets that satisfy a == b where a and b are defined through the bitwise XOR operation, we use the <u>prefix sum</u> pattern with a slight tweak - using XOR instead of addition.

The steps of the implementation include: Initialization:

# the array up to the i-1th index.

Precomputation: We calculate the prefix XOR sequence by iterating through the input array and performing the XOR operation for each element. The pre [0]

is set to be 0 as a base case since XOR with 0 gives us the number itself, which starts our sequence.

■ Compute a as pre[j] ^ pre[i] which gives the XOR of the subarray from i to j-1.

■ Compute b as pre[k + 1] ^ pre[j] which gives the XOR of the subarray from j to k.

After iterating through all triplets, the counter ans holds the number of triplets satisfying a == b. Return ans.

This brute-force algorithm uses the concept of prefix sums along with the properties of XOR to solve the problem in a

o Initialize a list pre with a length of n + 1 to store the prefix XOR values. The pre[i] will store the XOR of all elements from the beginning of

**Triplets Counting:** 

• After the precomputation step, we iterate over all potential starting indices i for the array segment a. • For each i, iterate over all potential starting indices j where j > i for the array segment b. Note that j can also be the ending index of segment a.  $\circ$  For each pair (i, j), iterate over all possible ending indices k for the segment b where k >= j.

If a equals b, increment the counter ans. Return the result:

enumeration of triplets.

Following the solution approach:

arr: [ 3, 10, 5, 25, 2, 8 ]

pre: [ 0, 3, 9, 12, 21, 23, 31 ]

**Example Walkthrough** Let's illustrate the solution approach with an example. Suppose we have the following array: arr = [3, 10, 5, 25, 2, 8]

straightforward way. The primary data structure used here is the array for storing prefix XORs. The pattern utilized is a classic

computational geometry approach to handle subarray or subrange queries efficiently by preparation combined with a brute-force

**Initialization:** ◦ The length of the array n is 6.  $\circ$  We initialize a list pre with length n + 1 to store the prefix XOR values. Thus, pre has 7 elements. **Precomputation:** 

### $\circ$ We iterate over all combinations of i, j, and k to find all possible (i, j, k) triplets: ■ For i = 0, j = 1, and k = 2, we have:

a = pre[j] ^ pre[i] = pre[1] ^ pre[0] = 3 ^ 0 = 3  $\blacksquare$  b = pre[k + 1] ^ pre[j] = pre[3] ^ pre[1] = 12 ^ 3 = 9

Since a is not equal to b, we do not increment ans.

Since a is not equal to b, we do not increment ans.

a = pre[j] ^ pre[i] = pre[3] ^ pre[1] = 12 ^ 3 = 9

■ We continue this process for all possible i, j, and k.

• For i = 1, j = 3, and k = 5, we find that:

prefix = [0] \* (array\_length + 1)

# Initialize the count of triplets

for i in range(array\_length - 1):

**if** a == b:

# Compute the prefix XOR values

prefix[i + 1] = prefix[i] ^ arr[i]

for i in range(array\_length):

triplet\_count = 0

return triplet\_count

b = pre[k + 1] ^ pre[j] = pre[4] ^ pre[2] = 21 ^ 9 = 12

• We set pre[0] to 0. We then iterate over the array to fill in the rest of the pre array with prefix XOR values:

```
• For i = 0, j = 2, and k = 3, we have:
   a = pre[j] ^ pre[i] = pre[2] ^ pre[0] = 9 ^ 0 = 9
```

equality.

**Triplets Counting:** 

- b = pre[k + 1] ^ pre[j] = pre[6] ^ pre[3] = 31 ^ 12 = 19 a is not equal to b, so ans remains unchanged. • Finally, upon reaching i = 1, j = 4, and k = 5, we get: a = pre[j] ^ pre[i] = pre[4] ^ pre[1] = 21 ^ 3 = 22 - b = pre[k + 1] ^ pre[j] = pre[6] ^ pre[4] = 31 ^ 21 = 10 Once again, a is not equal to b. ■ This iterative process is performed for all combinations to search for a == b. **Return the result:** 

o In our example, let's say there were no instances where a equaled b. Therefore, the answer ans is 0.

Solution Implementation **Python** from typing import List class Solution: def countTriplets(self, arr: List[int]) -> int: # Length of the array array\_length = len(arr)

# Iterate over each element considering it as the start of the triplet

# Calculate XOR of elements from index j to k

b = prefix[k + 1] ^ prefix[j]

triplet\_count += 1

# Return the total count of triplets found

# Iterate over each element considering it as the middle of the triplet

# Prefix XOR array where prefix[i] represents XOR of all elements from index 0 to i-1

possible triplets and calculate the XOR for the segments defined by i, j, and k.

for j in range(i + 1, array\_length): # Iterate over each element considering it as the end of the triplet for k in range(j, array\_length): # Calculate XOR of elements from index i to j-1 a = prefix[j] ^ prefix[i]

# If XORs are same, increment the count as it satisfies the given condition

After considering all combinations of i, j, k in array arr, we calculated the value of a and b for each triplet and compared them for

In this example, we did not find any triplets such that a == b. However, we followed the solution approach closely to check for all

```
Java
```

public:

int countTriplets(vector<int>& arr) {

for (int i = 0; i < n; ++i) {

int n = arr.size(); // Get the size of the array 'arr'

// XOR of elements from i to j-1

// XOR of elements from j to k

int a = prefixXOR[j] ^ prefixXOR[i];

// Calculate prefix XOR values for the array

prefixXOR[i + 1] = prefixXOR[i] ^ arr[i];

vector<int> prefixX0R(n + 1); // Initialize a vector for prefix XOR

int ans = 0; // Initialize the answer variable to store the count of triplets

for (int i = 0; i < n - 1; ++i) { // 'i' iterates from 0 to second last element

for (int j = i + 1; j < n; ++j) { // 'j' starts from the element next to 'i'

for (int k = j; k < n; ++k) { // 'k' starts from 'j' and covers all elements till the end

// Triple nested loop to compare all possible combinations of i, j, and k

```
class Solution {
    public int countTriplets(int[] arr) {
        int length = arr.length; // The length of the input array.
        int[] prefixXor = new int[length + 1]; // Prefix XOR array, with an extra slot to handle 0 case.
       // Construct the prefix XOR array where prefixXor[i] is XOR of all elements from start upto i-1.
        for (int i = 0; i < length; ++i) {</pre>
            prefixXor[i + 1] = prefixXor[i] ^ arr[i];
        int count = 0; // The result count for triplets.
        // Iterate through all possible starts i of subarray (arr[i] to arr[k]).
        for (int i = 0; i < length - 1; ++i) {
            // Iterate through all possible ends j (where i < j \le k) of subarray starting at arr[i].
            for (int j = i + 1; j < length; ++j) {
                // Iterate for all possible ends k of the second subarray, starting from arr[j].
                for (int k = j; k < length; ++k) {</pre>
                    // XOR of subarray arr[i] to arr[j-1].
                    int xorA = prefixXor[j] ^ prefixXor[i];
                    // XOR of subarray arr[j] to arr[k].
                    int xorB = prefixXor[k + 1] ^ prefixXor[j];
                    if (xorA == xorB) { // If the XOR of both subarrays is equal, it's a valid triplet.
                        count++; // Increment the count of valid triplets.
        return count; // Return the final count of triplets.
C++
class Solution {
```

```
int b = prefixXOR[k + 1] ^ prefixXOR[j];
                   // If the XOR subarray values are the same, increment the answer
                    if (a == b) {
                        ++ans;
       // Return the final triplet count
       return ans;
};
TypeScript
function countTriplets(arr: number[]): number {
    const n = arr.length; // Get the size of the array 'arr'
    let prefixXOR: number[] = new Array(n + 1).fill(0); // Initialize an array for prefix XOR with default values of 0
    // Calculate prefix XOR values for the array
   for (let i = 0; i < n; ++i) {
       prefixXOR[i + 1] = prefixXOR[i] ^ arr[i];
    let answer = 0; // Initialize the answer variable to store the count of triplets
    // Triple nested loop to compare all possible combinations of i, j, and k
    for (let i = 0; i < n - 1; ++i) { // 'i' iterates from 0 to the second last element
        for (let j = i + 1; j < n; ++j) { // 'j' starts from the element next to 'i'
            for (let k = j; k < n; ++k) { // 'k' starts from 'j' and covers all elements till the end
                // XOR of elements from i to j-1
                let a = prefixXOR[j] ^ prefixXOR[i];
                // XOR of elements from j to k
                let b = prefixXOR[k + 1] ^ prefixXOR[j];
                // If the XOR subarray values are the same, increment the answer
               if (a === b) {
                    ++answer;
```

```
return triplet_count
Time and Space Complexity
```

if a == b:

// Return the final triplet count

# Length of the array

triplet\_count = 0

array\_length = len(arr)

def countTriplets(self, arr: List[int]) -> int:

# Compute the prefix XOR values

prefix[i + 1] = prefix[i] ^ arr[i]

for j in range(i + 1, array\_length):

for k in range(j, array\_length):

a = prefix[j] ^ prefix[i]

b = prefix[k + 1] ^ prefix[j]

input array arr. This initial loop has a time complexity of O(n).

triplet\_count += 1

# Return the total count of triplets found

prefix = [0] \* (array\_length + 1)

# Initialize the count of triplets

for i in range(array\_length - 1):

for i in range(array\_length):

return answer;

from typing import List

class Solution:

## After that, there are three nested loops indexed by i, j, and k. Loop i runs from 0 to n-2, loop j runs from i+1 to n-1, and loop k runs from j to n−1.

# Prefix XOR array where prefix[i] represents XOR of all elements from index 0 to i-1

# Iterate over each element considering it as the start of the triplet

# Calculate XOR of elements from index i to j-1

# Calculate XOR of elements from index j to k

# Iterate over each element considering it as the middle of the triplet

# Iterate over each element considering it as the end of the triplet

# If XORs are same, increment the count as it satisfies the given condition

The time complexity of this code can be analyzed through the nested loops within the countTriplets method.

Therefore, in the worst case, the number of times the innermost loop runs can be computed as: Sum(i=0 to n-2) Sum(j=i+1

There is an initial loop responsible for calculating the prefix XOR array pre, which runs n times, where n is the length of the

to n-1) Sum(k=j to n-1) 1 operations which reduces to 0(n^3) because for each outer loop iteration, the innermost loop runs in a decreasing order creating a cubic number of iterations.

Combining these complexities, the total time complexity of the code is dominated by the three nested loops giving us T(n) =  $0(n) + 0(n^3) = 0(n^3).$ 

# **Space Complexity**

**Time Complexity** 

The space complexity can be observed through the use of extra memory in the code, which is mainly due to the prefix XOR array pre. The array pre has a length n + 1, where n is the length of the input array arr. Thus, the space required for the prefix XOR

- array is O(n). Besides the array pre, the variables i, j, k, a, and b use a constant amount of space each. •
- Therefore, considering the extra space used, the total space complexity of the code is S(n) = O(n) because the space used does not grow with respect to the number of loops or operations accomplished, but is directly related to the size of the input n.