1803. Count Pairs With XOR in a Range Bit Manipulation Trie Array Hard

Leetcode Link

Problem Description In this problem, we're given an integer array nums and two integers low and high. Our task is to find the number of 'nice pairs' in the

array. A 'nice pair' is defined as a pair of indices (i, j) such that $0 \ll i < j < nums.length and low <math>\ll (nums[i])$ XOR nums[j] $\ll (nums[i])$ high. The XOR here is the bitwise exclusive OR operation, which compares the bits of two numbers and returns 1 for each position where the corresponding bits of the two numbers are different, and returns 0 where they are the same.

Intuition

This problem is a good candidate for a Trie (prefix tree), especially when dealing with bits and XOR operations. A Trie allows us to efficiently store and retrieve binary representations of numbers. The main intuition is that, for each number x in nums, we can insert its binary representation into the Trie and simultaneously query the Trie to count how many numbers previously inserted into the Trie would form a 'nice pair' with x.

We need to calculate two quantities for each number x in nums: 1. The count of numbers in Trie so far that form a 'nice pair' with x when XORed, such that the result is less than or equal to high. 2. The count of numbers in Trie so far that form a 'nice pair' with x when XORed, such that the result is less than low.

The difference between these two counts gives us the number of 'nice pairs' for that particular value of x. By subtracting the count

The Trie structure is a special tree where each node represents a bit position (from the most significant bit to the least), and each

for low from the count for high + 1, we exclude pairs where the XOR is too small.

of elements that share the same bit prefix up to that node. To ensure that we are considering all 16-bits of the integers, we iterate from the 15th to the 0th bit (since array elements are within the range of 0 to 10⁴, and 10⁴ in binary takes up to 14 bits, we use 16 for a safe bound).

path from the root to a node represents the prefix of the binary representation of the inserted numbers. Every node stores the count

The search method on the Trie tries to maximize the XOR result while keeping it under the given limit (high or low). At every bit, we

decide whether to continue with the same bit or switch to the opposite bit to maximize the XOR based on the limit's current bit. If we can afford to switch the bit (based on the limit), we add the count of nodes under the current bit to the overall count and then move to the opposite bit for higher XOR value. Otherwise, we just follow the current bit path. This search method allows us to efficiently count all the suitable pairs for each entered number.

The solution involves constructing a Trie to handle the binary representations of numbers for quick retrieval and comparison. Here's a step-by-step explanation of how the solution is structured and how it works with the Trie:

• We define a Trie class to handle each bit of the 16-bit integers we are working with. Each node in the Trie has an array children

of size 2 (one for bit 0 and another for bit 1) and a cnt variable to store the number of elements that follow the node's path.

Trie Searching

count.

• The search function calculates how many numbers in the Trie, when XORed with x, would produce a result lower than the given

limit (which will be low or high + 1). It iterates through each bit (from the most significant to the least significant bit) and

• If the current bit of limit is 0, it means we cannot switch to the higher XOR value path, because doing so would lead to an XOR result higher than the limit. Therefore, it follows the path consistent with x's current bit.

- 1. Calls tree.search(x, high + 1) to find the number of elements that give an XOR with x less than or equal to high. 2. Calls tree.search(x, low) to find the number of elements that give an XOR with x less than low.
- After searching, the function then inserts x into the Trie to include it in the subsequent searches for the following numbers in nums.

3. The difference of the two search results is the count of 'nice pairs' for the number x (since we want the XOR to be at least

illustration are 0011, 1010, 0101, and 11001. We first initialize our Trie and prepare to insert each number from the array and simultaneously search for 'nice pairs'.

Suppose we have nums = [3, 10, 5, 25], low = 10, and high = 20. The binary representations of these numbers up to 5 bits for

• Next, consider the number 10 (binary 1010). We search the Trie for numbers that could form 'nice pairs':

For high condition (high + 1 = 21): The binary of 20 (limit) is 10101, and we traverse the Trie. We notice that the first bit in

1010 and 10101 is the same (bit 1) so we follow the same path (no flips). For the second bit, 0 in our number and 0 in high +

1, we proceed similarly. For the third bit, we have a 1 in high + 1 which means we could flip our current 1 to 0 to increase the

XOR, but there is no such path (since we only have 3 in the Trie), so we keep the path. This process continues until we reach

• For low condition: The binary of low is 01010, we traverse the Trie following a similar process but since 3 XOR 10 is lower than

condition.

Python Solution

class TrieNode:

12

13

14

15

16

17

24

25

26

27

28

34

35

36

37

38

39

40

41

42

43

44

45

6

50

51

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

C++ Solution

class Trie {

Trie()

public:

#include <vector>

using namespace std;

: children(2, nullptr)

int ans = 0;

} else {

return ans;

for (int i = 15; $i \ge 0 \&\& node$; --i) {

int limitBit = (limit >> i) & 1;

if (node->children[bit]) {

if (limitBit) { // if current bit in limit is 1

ans += node->children[bit]->count;

int bit = (x >> i) & 1;

class Solution:

result = 0

trie = TrieNode()

return result

for number in nums:

With 5 inserted, our Trie now contains 3, 10, and 5. We have found 2 'nice pairs' so far.

• For the number 5 (binary 0101), we search the Trie for potential 'nice pairs':

 For high condition (high + 1 = 21): During the Trie traversal, we find that 5 XOR 3 is 6, and 5 XOR 10 is 15. Both are within our range [10, 20], thus adding 2 to our 'nice pairs' count.

• For low condition: Since our XOR results (6 and 15) are already greater than 10, we do not subtract any pairs for the low

 No additional 'nice pairs' are found, and 25 is inserted into the Trie. The total number of 'nice pairs' found is 2: (5 XOR 3) and (5 XOR 10).

The Trie and search operations allow us to efficiently manage and calculate the XOR pairings without having to resort to a brute-

For low condition: They also exceed low, so again, they are not subtracted from our 'nice pairs' count.

def insert(self, number): node = self for i in range(15, -1, -1): # Represent the number as a 16 bit integer

bit = (number >> i) & 1 # Extract the i-th bit of number

Search the trie to find the count of pairs with XOR less than limit

if limit_bit: # Checking if the bit is set in the limit.

If so, all numbers with the same bit at this position

Add count of valid pairs between number and numbers already in the trie

result += trie.search(number, high + 1) - trie.search(number, low)

result = 0 18 19 for i in range(15, -1, -1): # Traverse from most significant bit to least if node is None: 20 21 return result 22 bit = (number >> i) & 1 23 limit_bit = (limit >> i) & 1

node.count += 1 # Increment the count of numbers passing through this node

```
1 class Trie {
      private Trie[] children = new Trie[2]; // Trie node children, one for bit 0, one for bit 1
      private int count; // Number of elements that pass through this node
      // Inserts a number into the trie
5
```

Java Solution

```
21
            int answer = 0; // Initialize the answer
22
            for (int i = 15; i \ge 0 && node != null; --i) {
23
                int bit = (x >> i) & 1; // Extract the current bit
24
               if (((limit >> i) & 1) == 1) {
25
                   // If the bit is '1' in the limit, add count of numbers that have the opposite bit
26
                   if (node.children[bit] != null) {
27
                        answer += node.children[bit].count;
28
29
                   node = node.children[bit ^ 1]; // Move to the opposite bit's child node if it exists
30
               } else {
31
                   node = node.children[bit]; // If the bit is '0', just move to the same bit's child node
32
33
34
            return answer;
35
36
37
   class Solution {
       // Counts the number of pairs (i, j) where i < j and XOR of nums[i] and nums[j] is in the range [low, high]
39
        public int countPairs(int[] nums, int low, int high) {
41
            Trie trie = new Trie(); // Initialize the trie
            int answer = 0; // Initialize the number of valid pairs
42
43
            for (int number : nums) {
44
               // Search through the trie and check for valid pairs within bounds, high + 1 is used to include 'high' in range
               answer += trie.search(number, high + 1) - trie.search(number, low);
45
46
               trie.insert(number); // Insert the number into the trie
47
48
            return answer; // Return the final count of valid pairs
49
```

// Initialize with 2 children as nullptrs for binary trie.

51 52 53 54

```
for (int x : nums) {
                 // Count and subtract to get number of pairs with XOR in range.
 55
                 ans += trie->search(x, high + 1) - trie->search(x, low);
 56
                 trie->insert(x); // Insert the number into the Trie.
 57
 58
             return ans; // Return the final answer.
 59
 60 };
 61
Typescript Solution
    type TrieNode = {
         children: (TrieNode | null)[];
         count: number;
  4 };
  5
  6 // Initialize the Trie root node
    const root: TrieNode = { children: [null, null], count: 0 };
     // Insert number x into Trie
    function insert(x: number) {
         let node = root;
 11
 12
         for (let i = 15; i >= 0; --i) {
 13
             const bit = (x \gg i) \& 1;
             if (!node.children[bit]) {
 14
 15
                 node.children[bit] = { children: [null, null], count: 0 };
 16
             node = node.children[bit]!;
 17
 18
             ++node.count;
 19
 20
 21
    // Search and count number of elements less than or equal to limit XOR'd with x
     function search(x: number, limit: number) {
 24
         let node = root;
 25
         let ans = 0;
 26
         for (let i = 15; i >= 0; --i) {
             const bit = (x \gg i) \& 1;
 27
 28
             const limitBit = (limit >> i) & 1;
             if (limitBit) {
 29
                 if (node.children[bit]) {
 30
 31
                     ans += node.children[bit].count;
 32
 33
                 node = node.children[bit ^ 1];
             } else {
 34
 35
                 node = node.children[bit];
 36
             if (!node) break;
```

The time complexity of the insert and search methods in the Trie class is 0(16) for each call which simplifies to 0(1) since the loop runs for a fixed number of iterations (16 in this case), corresponding to the binary representation of the integers within the fixed range [0, 2^16 - 1].

38

39

40

41

45

47

48

then the overall time complexity of the countPairs function would be 0(3n), which simplifies to 0(n). Hence, the total time complexity of the solution is O(n).

The space complexity is determined by the size of the Trie data structure. In the worst case, if all the binary representations of the numbers in the list are unique, the trie could have up to 2^16 nodes (for a 16-bit integer). However, since the Trie is built as the numbers are added, and it only contains paths for numbers in the nums list, the space complexity would be 0(16n) which simplifies to

integer).

Adding all the differences together for each number in the array gives us the total count of 'nice pairs' within the entire array.

Solution Approach

Trie Data Structure

Trie Insertion • The insert function takes an integer x and inserts it into the Trie bit by bit, starting from the most significant bit (15th bit). For each bit in x, if the corresponding children node doesn't exist, it creates a new Trie node. It then updates the current node's

• If the current bit of limit is 1, the function adds the count of the current bit path of x to the answer (because flipping the current bit of x could only increase the XOR result) and then proceeds to check the opposite path for further potential matches.

• The countPairs function from the Solution class is where the algorithm starts applying the Trie structure. For every number x in the array nums, the function performs the following steps:

low and at most high).

Example Walkthrough

The CountPairs Function

compares the bits of x and limit.

The use of Trie to handle the binary representations of numbers and the clever bit manipulation during Trie traversal enables efficient calculation of 'nice pairs', which would be computationally intensive to obtain with a brute-force approach.

• Finally, the countPairs function returns the total number of 'nice pairs' found for the entire array nums.

Let's use a small example to illustrate the solution approach detailed in the content above.

- Starting with the first number 3 (binary 0011), we insert it into the Trie. Since there are no previous numbers, the search for 'nice pairs' will return 0 for both high + 1 and low.
- 10, this pair is not valid either. • The Trie now has 3 and 10. No 'nice pairs' have been found yet.

the end, and in this case, since 3 XOR 10 is 9 which is less than 10 (our low), there are no valid pairs for high so far.

- Finally, for number 25 (binary 11001), we search the Trie: ∘ For high condition (high + 1 = 21): We find that 25 XOR 3 = 26, 25 XOR 10 is 27, and 25 XOR 5 is 28. All these results exceed the high value, so they are not valid.
- def __init__(self): self.children = [None] * 2 # A binary trie, so 2 children, for 0 and 1 self.count = 0 # Maintain a count of numbers that pass through this node
- 29 # Move to the opposite bit to keep the XOR sum less than limit node = node.children[bit ^ 1] 30 31 else: node = node.children[bit] 32 33 return result

if node.children[bit]:

def countPairs(self, nums, low, high):

trie.insert(number)

public void insert(int number) {

Trie node = this; // Start from the root

will have a XOR less than limit.

result += node.children[bit].count

force comparison, which would be less efficient for larger datasets.

if node.children[bit] is None:

node = node.children[bit]

def search(self, number, limit):

node = self

node.children[bit] = TrieNode()

```
for (int i = 15; i \ge 0; --i) { // Iterate over the bits of the number
 8
               int bit = (number >> i) & 1; // Extract the current bit
               if (node.children[bit] == null) {
10
                   node.children[bit] = new Trie(); // Create new node if it doesn't exist
11
12
13
               node = node.children[bit];
14
               ++node.count; // Increment the count because we're adding a number
15
16
17
18
        // Searches for numbers in the trie that have a XOR result with x below the given limit
19
        public int search(int x, int limit) {
20
           Trie node = this; // Start from the root
```

, count(0) {} 9 10 // Insert number x into Trie. void insert(int x) { 11 12 Trie* node = this; 13 for (int i = 15; $i \ge 0$; --i) { // Assuming 16-bit integer here. int bit = (x >> i) & 1;14 if (!node->children[bit]) { 15 node->children[bit] = new Trie(); 16 17 node = node->children[bit]; 18 19 ++node->count; // Increment count at each node traversed. 20 21 22 23 // Search and count number of elements less than or equal to limit XOR'd with x. int search(int x, int limit) { 24 25 Trie* node = this;

node = node->children[bit ^ 1]; // Move to the opposite bit node.

node = node->children[bit]; // Move to the same bit node as current bit of x

41 private: vector<Trie*> children; // Children is a vector of Trie pointers for the binary trie. 43 44 int count; // Count of numbers in the Trie that have traversed through this node. 45 }; 46 class Solution { public: 48 // Counts the number of pairs that have a bitwise XOR in [low, high]. 49 int countPairs(vector<int>& nums, int low, int high) { 50 Trie* trie = new Trie(); // Create a new Trie. int ans = 0;

return ans;

let ans = 0;

for (const x of nums) {

insert(x);

49 return ans; 50 51 Time and Space Complexity **Time Complexity**

// Count the number of pairs that have a bitwise XOR in [low, high]

function countPairs(nums: number[], low: number, high: number) {

ans += search(x, high + 1) - search(x, low);

Space Complexity

The countPairs function calls insert once and search twice for every element in the nums list. If n is the number of elements in nums,

O(n) because each number in nums could theoretically end up contributing to 16 nodes in the Trie (one for each bit of the 16-bit

Therefore, the total space complexity of the solution is O(n).