1803. Count Pairs With XOR in a Range

Hard | Bit Manipulation | Trie | Array   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 <= i < j < nums.length` and `low <= (nums[i] XOR nums[j]) <= 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 for `low` from the count for `high + 1`, we exclude pairs where the XOR is too small.

The Trie structure is a special tree where each node represents a bit position (from the most significant bit to the least), and each path from the root to a node represents the prefix of the binary representation of the inserted numbers. Every node stores the count 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 `10^4`, and `10^4` in binary takes up to 14 bits, we use 16 for a safe bound).

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.

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

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:

Trie Data Structure

- 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 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 count.

Trie Searching

- 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 compares the bits of `x` and `limit`.

- 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.

- 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.

The CountPairs Function

- 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:

    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`.

    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 `low` and at most `high`).

- After searching, the function then inserts `x` into the Trie to include it in the subsequent searches for the following numbers in `nums`.

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

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.

Example Walkthrough

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

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

- 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`.

- 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 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.

    - 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 `10`, this pair is not valid either.

- The Trie now has `3` and `10`. No 'nice pairs' have been found yet.

- 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` condition.

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

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

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

Python Solution

```python
class TrieNode:
    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

    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
            if node.children[bit] is None:
                node.children[bit] = TrieNode()
            node = node.children[bit]
            node.count += 1  # Increment the count of numbers passing through this node

    # Search the trie to find the count of pairs with XOR less than limit
    def search(self, number, limit):
        node = self
        result = 0
        for i in range(15, -1, -1):  # Traverse from most significant bit to least
            if node is None:
                return result
            bit = (number >> i) & 1
            limit_bit = (limit >> i) & 1
            if limit_bit:  # Checking if the bit is set in the limit.
                # If so, all numbers with the same bit at this position
                # will have a XOR less than limit.
                if node.children[bit]:
                    result += node.children[bit].count
                # Move to the opposite bit to keep the XOR sum less than limit
                node = node.children[bit ^ 1]
            else:
                node = node.children[bit]
        return result


class Solution:
    def countPairs(self, nums, low, high):
        result = 0
        trie = TrieNode()
        for number in nums:
            # Add count of valid pairs between number and numbers already in the trie
            result += trie.search(number, high + 1) - trie.search(number, low)
            trie.insert(number)
        return result
```

Java Solution

```java
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
    public void insert(int number) {
        Trie node = this; // Start from the root
        for (int i = 15; i >= 0; --i) { // Iterate over the bits of the number
            int bit = (number >> i) & 1; // Extract the current bit
            if (node.children[bit] == null) {
                node.children[bit] = new Trie(); // Create new node if it doesn't exist
            }
            node = node.children[bit];
            ++node.count; // Increment the count because we're adding a number
        }
    }

    // Searches for numbers in the trie that have a XOR result with x below the given limit
    public int search(int x, int limit) {
        Trie node = this; // Start from the root
        int answer = 0; // Initialize the answer
        for (int i = 15; i >= 0 && node != null; --i) {
            int bit = (x >> i) & 1; // Extract the current bit
            if (((limit >> i) & 1) == 1) {
                // If the bit is '1' in the limit, add count of numbers that have the opposite bit
                if (node.children[bit] != null) {
                    answer += node.children[bit].count;
                }
                node = node.children[bit ^ 1]; // Move to the opposite bit's child node if it exists
            } else {
                node = node.children[bit]; // If the bit is '0', just move to the same bit's child node
            }
        }
        return answer;
    }
}

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]
    public int countPairs(int[] nums, int low, int high) {
        Trie trie = new Trie(); // Initialize the trie
        int answer = 0; // Initialize the number of valid pairs
        for (int number : nums) {
            // 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);
            trie.insert(number); // Insert the number into the trie
        }
        return answer; // Return the final count of valid pairs
    }
}
```

C++ Solution

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

class Trie {
public:
    Trie()
        : children(2, nullptr)     // Initialize with 2 children as nullptrs for binary trie.
        , count(0) {}

    // Insert number x into Trie.
    void insert(int x) {
        Trie* node = this;
        for (int i = 15; i >= 0; --i) { // Assuming 16-bit integer here.
            int bit = (x >> i) & 1;
            if (!node->children[bit]) {
                node->children[bit] = new Trie();
            }
            node = node->children[bit];
            ++node->count;  // Increment count at each node traversed.
        }
    }

    // Search and count number of elements less than or equal to limit XOR'd with x.
    int search(int x, int limit) {
        Trie* node = this;
        int ans = 0;
        for (int i = 15; i >= 0 && node; --i) {
            int bit = (x >> i) & 1;
            int limitBit = (limit >> i) & 1;
            if (limitBit) { // if current bit in limit is 1
                if (node->children[bit]) {
                    ans += node->children[bit]->count;
                }
                node = node->children[bit ^ 1]; // Move to the opposite bit node.
            } else {
                node = node->children[bit]; // Move to the same bit node as current bit of x
            }
        }
        return ans;
    }

private:
    vector<Trie*> children;  // Children is a vector of Trie pointers for the binary trie.
    int count;  // Count of numbers in the Trie that have traversed through this node.
};

class Solution {
public:
    // Counts the number of pairs that have a bitwise XOR in [low, high].
    int countPairs(vector<int>& nums, int low, int high) {
        Trie* trie = new Trie(); // Create a new Trie.
        int ans = 0;
        for (int x : nums) {
            // Count and subtract to get number of pairs with XOR in range.
            ans += trie->search(x, high + 1) - trie->search(x, low);
            trie->insert(x);  // Insert the number into the Trie.
        }
        return ans;  // Return the final answer.
    }
};
```

Typescript Solution

```typescript
type TrieNode = {
    children: (TrieNode | null)[];
    count: number;
};

// 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;
    for (let i = 15; i >= 0; --i) {
        const bit = (x >> i) & 1;
        if (!node.children[bit]) {
            node.children[bit] = { children: [null, null], count: 0 };
        }
        node = node.children[bit]!;
        ++node.count;
    }
}

// Search and count number of elements less than or equal to limit XOR'd with x
function search(x: number, limit: number) {
    let node = root;
    let ans = 0;
    for (let i = 15; i >= 0; --i) {
        const bit = (x >> i) & 1;
        const limitBit = (limit >> i) & 1;
        if (limitBit) {
            if (node.children[bit]) {
                ans += node.children[bit].count;
            }
            node = node.children[bit ^ 1];
        } else {
            node = node.children[bit];
        }
        if (!node) break;
    }
    return ans;
}

// Count the number of pairs that have a bitwise XOR in [low, high]
function countPairs(nums: number[], low: number, high: number) {
    let ans = 0;
    for (const x of nums) {
        ans += search(x, high + 1) - search(x, low);
        insert(x);
    }
    return ans;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the `insert` and `search` methods in the `Trie` class is `O(16)` for each call which simplifies to `O(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]`.

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`, then the overall time complexity of the `countPairs` function would be `O(3n)`, which simplifies to `O(n)`.

Hence, the total time complexity of the solution is `O(n)`.

Space Complexity

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 `O(16n)` which simplifies to `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 integer).

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