421. Maximum XOR of Two Numbers in an Array

Medium | Bit Manipulation | Trie | Array | Hash Table

Problem Description

The given problem involves finding the maximum result of `nums[i] XOR nums[j]` for all pairs of `i` and `j` in an array of integers named `nums`, where `i` is less than or equal to `j`. The XOR (^) is a bitwise operator that returns a number which has all the bits that are set in exactly one of the two operands. For example, the XOR of `5 (101)` and `3 (011)` is `6 (110)`. In simple terms, the task is to pick two numbers from the array such that their XOR is as large as possible, and return the value of this maximum XOR.

Intuition

To solve this problem efficiently, the solution uses a greedy approach with bitwise manipulation. The intuition is that to maximize the XOR value, we want to maximize the bit contributions from the most significant bit (MSB) to the least significant bit (LSB).

We know that if we XOR two equal bits, we get `0`, and if we XOR two different bits, we get `1`. To maximize the XOR value, we want a `1` in the MSB position of the result, which means the bits of numbers in that position should be different.

The code uses a loop that starts from the MSB (in this case, 30th bit assuming the numbers are within the range of 32-bit integers) and goes all the way down to the LSB. In each iteration of the loop, we check if there is a pair of numbers in `nums` which will give us a `1` in the current bit position we are looking at, when XORed together.

Here's how it works:

- We build a `mask` which contains the bits we've considered so far from the MSB to the current bit. At first, the mask is `0`, but in each iteration, we add the current bit to the mask.
- We then create a set `s` that will contain all the unique values of `nums` elements after applying the `mask`. This implicitly removes all less significant bits and focuses only on the portion of the bit sequence we are interested in at each stage of the loop.
- Next, we check if we can create a new `max` value by setting the current bit (going from left to right) to `1`. This is done by `flag = max | current`.
- For each 'prefix' in the set `s`, we check if there's another prefix such that when XORed with the `flag`, we get a result that is also in the set `s`. This would mean we have two numbers that, when the MSB to the current bit is considered, would produce a `1` in the current bit's position if XORed.
- If such pair exists, we update the `max` to be the `flag` which has the current bit set to `1`, and break since we've found the maximum possible XOR for the current bit position.
- We repeat this process for each bit, each time building on the previously established `max` to check whether the next bit can be increased.

By iteratively ensuring that we get the maximum contribution from each bit position, we eventually end up with the maximum possible XOR value of any two numbers from the array.

Solution Approach

The implemented solution is based on a greedy algorithm with bit manipulation. The key concept is to determine, bit by bit from the most significant bit to the least significant, whether we can achieve a larger XOR value with the current set of numbers. To facilitate this process, the code utilizes the following elements:

- **Bitwise Operations**: The use of bitwise operations (`^`, `&`, `|`) helps to isolate specific bits and manipulate them. These operations are essential for comparing bits and constructing the maximum XOR value.
- **Masking**: A mask is used to isolate the bits from the most significant bit to the current bit we are considering. This allows us to ignore the influence of the less significant bits which have not been considered yet.
- **Sets**: Sets are used to store unique prefixes of the numbers when only the masked bits are considered. This data structure facilitates constant time access to check if a certain prefix exists, which is crucial for optimizing the solution.
- **Greedy Choice**: The greedy choice is made by deciding to include the current bit into the maximum XOR value. This decision is based on whether adding a `1` on the current bit would increase the overall maximum XOR value.

To visualize the approach, let's walk through the code:

1. Initialize a variable `max` to 0 to keep track of the maximum XOR we've found so far.
2. Initialize a mask to 0 to progressively include bits from the MSB to LSB into the consideration.
3. Loop from 30 down to 0, representing the bit positions from the MSB to LSB:
   - Update the mask to include the current bit position by XORing it with `1 << i`, which creates a number with only the i-th bit set.
   - Create a new empty set `s`. Iterate through `nums` and add to `s` the result of bitwise `AND` of each number with the mask. This captures the unique prefixes up to the current bit.
   - Calculate `flag` by setting the current bit in `max`. This hypothetically represents a larger XOR if it's possible to achieve.
   - Now, use a nested loop to check if any two prefixes in `s` can achieve this hypothetical XOR:
     - For each prefix in `s`, calculate the result of `prefix ^ flag`. If this result is also in `s`, we know that these two prefixes can achieve `flag` when XORed.
     - If we find such a pair, update `max` to `flag` because we confirmed that setting the current bit results in a bigger XOR.
   - If a pair wasn't found, do nothing, and the current bit in `max` remains `0`.
4. After the loop ends, `max` will contain the maximum XOR value possible with any two numbers in `nums`.
5. Return the `max` value which is the answer.

This approach uses the fact that if `prefix1 ^ max = prefix2` (this implies `prefix1 ^ prefix2 = max`), then it stands to reason that there is a pair of numbers with the prefix1 and prefix2 that would result in `max` when XORed. Thus, it guarantees that the maximum value is found considering each bit position.

Example Walkthrough

Let's consider `nums = [3, 10, 5, 25, 2, 8]` and walk through the solution process using the described algorithm.

Step 1: Initialize `max` to `0`. This will keep track of the maximum XOR found at any step.

Step 2: Initialize `mask` to `0` to include bits to the consideration progressively.

Step 3: Loop through bit positions from 30 down to 0.

For illustration, let's loop from the 4th bit position (since our largest number 25 is `11001` in binary and requires only 5 bits to represent).

i. Bit position 4 (counting from zero, so it's the fifth most significant bit)

- Update `mask` with `mask = mask | (1 << 4)`, so `mask` becomes `10000` in binary.
- Create an empty set `s` and populate it with the masked elements of `nums`.
  - For instance, `25 & mask = 16` and `8 & mask = 0`. So our set `s` will be `{0, 16}`.
- Calculate `flag = max | (1 << 4)` (`flag = 10000` in binary).
- Check if any two prefixes XOR together can equal `flag` (by checking for each prefix if `prefix ^ flag` exists in `s`).
  - In this case, we check `0 ^ 16` and see that `16` is in `s`, and `16 ^ 16` will check if `0` is in `s`.
- Since both `16` and `0` are in `s`, we can update `max = flag`. Now `max` is `10000` in binary, or `16` in decimal.

ii. Bit position 3

- Update `mask` to now include this 3rd bit: `mask = mask | (1 << 3)`, so `mask` is now `11000`.
- Populate the new set `s`, which will be `{0, 8, 16, 24}`.
- Calculate the new `flag = max | (1 << 3)` (`flag = 11000` in binary, or `24` in decimal).
- Check the XOR conditions as before.
  - In this case, `8 ^ 24 = 16` is in `s`, and so is `16 ^ 24 = 8`.
- Update `max` to `flag`. Now `max` is `24`.

iii. Bit position 2

- Repeat the process. However, for this step, let's assume that there are no two prefixes in `s` that can achieve the new `flag` with XOR.
- Therefore, we do not update `max` in this round.

iv. Continue looping in this manner down to bit position 0.

Step 4: After loop ends, `max` holds the maximum XOR that we can find.

Given our array, the maximum XOR is between `5 (101)` and `25 (11001)`, which equals `28 (11100)` in binary notation.

Thus, the final answer returned is `max` which is `28`.

Python Solution

```python
class Solution:
    def findMaximumXOR(self, nums: List[int]) -> int:
        maximum_xor = 0  # Initialize the maximum XOR value.
        mask = 0  # Initialize the mask, which is used to consider bits from the most significant bit down.

        # Start from the highest bit and go down to the least significant bit (31st to 0th bit).
        for i in range(31, -1, -1):
            mask = mask | (1 << i)  # Update the mask to include the next bit.
            prefixes = set()  # Create a set to store prefixes of the current length.

            # Collect all prefixes with bits up to the current bit.
            for num in nums:
                prefixes.add(num & mask)  # Bitwise AND to isolate the prefix.

            # We assume the new bit is '1' and combine it with maximum XOR so far.
            proposed_max = maximum_xor | (1 << i)

            # Check if there's a pair of prefixes which XOR equals our proposed maximum so far.
            for prefix in prefixes:
                if (prefix ^ proposed_max) in prefixes:
                    maximum_xor = proposed_max  # Update the maximum XOR since we found a pair.
                    break  # No need to check other prefixes.

        # After checking all bits, return the maximum XOR we found.
        return maximum_xor

```

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

class Solution {

    public int findMaximumXOR(int[] numbers) {
        int maximumXOR = 0;   // Variable to hold the maximum XOR value found
        int mask = 0;         // Variable to hold the bit mask

        // Start from the highest bit position and check for each bit
        for (int i = 30; i >= 0; i--) {
            // Update the bit mask to include the current bit
            mask = mask | (1 << i);
            Set<Integer> prefixes = new HashSet<>();  // Set to store prefixes of numbers

            // Collect all unique prefixes of "i" bits of all numbers
            for (int number : numbers) {
                // Add the prefix of current number to the set
                prefixes.add(number & mask);
            }

            // Guess that current maximum XOR would have the current bit set
            int guessedMaximumXOR = maximumXOR | (1 << i);

            // Check if there are two prefixes that would result in the guessed maximum XOR
            for (Integer prefix : prefixes) {
                if (prefixes.contains(prefix ^ guessedMaximumXOR)) {
                    // If such two prefixes found, update the maximum XOR
                    maximumXOR = guessedMaximumXOR;
                    break;  // No need to check other prefixes
                }
            }
        }
        // Return the maximum XOR found
        return maximumXOR;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

// Trie class to implement the Trie data structure with binary representation
class Trie {
public:
    std::vector<Trie*> children; // Children for binary digits 0 and 1

    // Constructor initializes the children to hold two possible values (0 or 1)
    Trie() : children(2, nullptr) {}

    // Function to insert a number into the Trie
    void insert(int number) {
        Trie* node = this;
        // Start from the most significant bit and insert the number into the Trie
        for (int i = 30; i >= 0; --i) {
            int bit = (number >> i) & 1; // Extract the i-th bit of the number
            if (!node->children[bit]) { // If the bit node does not exist, create it
                node->children[bit] = new Trie();
            }
            node = node->children[bit]; // Move to the corresponding child
        }
    }

    // Function to search for the maximum XOR of a number with the existing numbers in the Trie
    int search(int number) {
        Trie* node = this;
        int result = 0; // Initialize the result to 0
        for (int i = 30; i >= 0; --i) {
            int bit = (number >> i) & 1; // Extract the i-th bit
            // Try to find the opposite bit in the Trie for maximized XOR
            if (node->children[bit ^ 1]) {
                result = (result << 1) | 1; // If found, update the result with set bit at current position
                node = node->children[bit ^ 1]; // Move to the opposite bit child
            } else {
                result <<= 1; // Else, result is updated just by shifting, bit remains unset
                node = node->children[bit]; // Move to the same bit child
            }
        }
        return result; // Return the maximum XOR value found
    }
};

// Solution class to handle the LeetCode question logic
class Solution {
public:
    // Function to find the maximum XOR of any two numbers in the array
    int findMaximumXOR(std::vector<int>& nums) {
        Trie* trie = new Trie(); // Initialize Trie
        int ans = 0; // Initialize answer to 0
        for (int num : nums) {
            trie->insert(num); // Insert each number into the Trie
            ans = std::max(ans, trie->search(num)); // Update the answer with the maximum XOR value found so far
        }
        return ans; // Return the answer
    }
};

```

Typescript Solution

```typescript
// Node structure for Trie with an optional array that holds children nodes
interface TrieNode {
  children: (TrieNode | undefined)[];
}

// Initializes a Trie node with undefined children
function createTrieNode(): TrieNode {
  return { children: [undefined, undefined] };
}

// Inserts a number into the Trie
function insert(trie: TrieNode, number: number): void {
  let node = trie;
  // Start from the most significant bit and insert the number into the Trie
  for (let i = 30; i >= 0; --i) {
    const bit = (number >> i) & 1; // Extract the i-th bit of the number
    if (!node.children[bit]) { // If the bit node does not exist, create it
      node.children[bit] = createTrieNode();
    }
    node = node.children[bit] as TrieNode; // Move to the corresponding child node
  }
}

// Searches for the maximum XOR of a number with the existing numbers in the Trie
function search(trie: TrieNode, number: number): number {
  let node = trie;
  let result = 0; // Initialize the result to 0
  for (let i = 30; i >= 0; --i) {
    const bit = (number >> i) & 1; // Extract the i-th bit
    // Try to find the opposite bit in the Trie for maximized XOR
    if (node.children[bit ^ 1]) {
      result = (result << 1) | 1; // If found, update the result with set bit at current position
      node = node.children[bit ^ 1] as TrieNode; // Move to the opposite bit child node
    } else {
      result <<= 1; // Else, result is updated just by shifting, bit remains unset
      node = node.children[bit] as TrieNode; // Move to the same bit child node
    }
  }
  return result; // Return the maximum XOR value found
}

// Finds the maximum XOR of any two numbers in an array
function findMaximumXOR(nums: number[]): number {
  const trie = createTrieNode(); // Create the root of the Trie
  let maxResult = 0; // Initialize maxResult to 0

  // Go through each number in the array
  nums.forEach((num) => {
    insert(trie, num); // Insert the number into the Trie
    // Update maxResult with the maximum XOR found so far
    maxResult = Math.max(maxResult, search(trie, num));
  });

  return maxResult; // Return the maximum XOR of two numbers in the array
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by nested loops where the outer loop runs 31 times (as bit-positions are checked from the most significant bit at index 30 down to the least significant bit at index 0) and the inner loops process each element in the list to build a set and subsequently check whether the XOR of each prefix with the candidate maximum value exists in the set.

- The outer loop runs 31 times since we are assuming that the integers are 32-bit and we do not need to check the sign bit for XOR operations.
- For each bit position, we iterate over all `n` numbers to add their left-prefixed bits to the set `s`. This operation takes `O(n)` time.
- We then iterate again through all unique prefixes, which are at most `n` in the worst case. For each prefix, we perform an `O(1)` time check to see if a certain XOR-combination could exist in the set. Therefore, this operation also takes `O(n)` time in the worst case.

Multiplying these together, the time complexity is `O(31n)`, which simplifies to `O(n)` because 31 is a constant factor which does not grow with `n`.

Space Complexity

The space complexity is determined by the additional memory taken up by the set `s` used to store the unique prefixes:

- The set `s` can have at most `n` elements as each number contributes a unique left-prefixed bit representation to the set. Therefore, the space required for the set is `O(n)`.

Considering the space used by `max`, `mask`, `current`, and `flag` are all constant `O(1)` space overheads, the total space complexity is `O(n)`.