421. Maximum XOR of Two Numbers in an Array Medium Bit Manipulation Hash Table Trie Array

Leetcode Link

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

Problem Description

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

• We then create a set s that will contain all the unique values of nums elements after applying the mask. This implicitly removes all

max | current.

each iteration, we add the current bit to the mask.

possible XOR value of any two numbers from the array.

are essential for comparing bits and constructing the maximum XOR value.

1. Initialize a variable max to 0 to keep track of the maximum XOR we've found so far.

- 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 =
- 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

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

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

• 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

• 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:

set.

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

o 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.

 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.

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

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

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

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 ② ^ 16 and see that 16 is in s, and 16 ^ 16 will check if ② is in s.

Update mask with mask = mask | (1 << 4), so mask becomes 10000 in binary.

For instance, 25 & mask = 16 and 8 & mask = 0. So our set s will be {0, 16}.

• Update mask to now include this 3rd bit: mask = mask | (1 << 3), so mask is now 11000.

• Calculate the new flag = max | (1 << 3) (flag = 11000 in binary, or 24 in decimal).

Create an empty set s and populate it with the masked elements of nums.

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.

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

• Calculate flag = max | (1 << 4) (flag = 10000 in binary).

Populate the new set s, which will be {0, 8, 16, 24}.

Therefore, we do not update max in this round.

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

- · Check the XOR conditions as before. In this case, 8 ^ 24 = 16 is in s, and so is 16 ^ 24 = 8.
- 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
- Thus, the final answer returned is max which is 28. **Python Solution**

maximum_xor = 0 # Initialize the maximum XOR value.

if (prefix ^ proposed_max) in prefixes:

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

break # No need to check other prefixes.

int maximumXOR = 0; // Variable to hold the maximum XOR value found

// Start from the highest bit position and check for each bit

// Collect all unique prefixes of "i" bits of all numbers

// Guess that current maximum XOR would have the current bit set

// If such two prefixes found, update the maximum XOR

// Add the prefix of current number to the set

// Update the bit mask to include the current bit

int guessedMaximumXOR = maximumXOR | (1 << i);</pre>

maximumXOR = guessedMaximumXOR;

break; // No need to check other prefixes

// Trie class to implement the Trie data structure with binary representation

std::vector<Trie*> children; // Children for binary digits 0 and 1

// Constructor initializes the children to hold two possible values (0 or 1)

// Start from the most significant bit and insert the number into the Trie

// Variable to hold the bit mask

Set<Integer> prefixes = new HashSet<>(); // Set to store prefixes of numbers

// Check if there are two prefixes that would result in the guessed maximum XOR

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

Java Solution

public int findMaximumXOR(int[] numbers) {

for (int i = 30; i >= 0; i--) {

 $mask = mask \mid (1 \ll i);$

// Return the maximum XOR found

Trie() : children(2, nullptr) {}

void insert(int number) {

Trie* node = this;

// Function to insert a number into the Trie

return ans; // Return the answer

function insert(trie: TrieNode, number: number): void {

node.children[bit] = createTrieNode();

25 function search(trie: TrieNode, number: number): number {

const bit = (number >> i) & 1; // Extract the i-th bit

// Try to find the opposite bit in the Trie for maximized XOR

const trie = createTrieNode(); // Create the root of the Trie

insert(trie, num); // Insert the number into the Trie

// Update maxResult with the maximum XOR found so far

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

maxResult = Math.max(maxResult, search(trie, num));

let result = 0; // Initialize the result to 0

42 // Finds the maximum XOR of any two numbers in an array

let maxResult = 0; // Initialize maxResult to 0

function findMaximumXOR(nums: number[]): number {

// Go through each number in the array

nums.forEach((num) => {

1 // Node structure for Trie with an optional array that holds children nodes

// Start from the most significant bit and insert the number into the Trie

if (!node.children[bit]) { // If the bit node does not exist, create it

24 // Searches for the maximum XOR of a number with the existing numbers in the Trie

node = node.children[bit] as TrieNode; // Move to the corresponding child node

result = (result << 1) | 1; // If found, update the result with set bit at current position

const bit = (number >> i) & 1; // Extract the i-th bit of the number

return maximumXOR;

for (int number : numbers) {

prefixes.add(number & mask);

for num in nums: 12 prefixes.add(num & mask) # Bitwise AND to isolate the prefix. 13 14 # We assume the new bit is '1' and combine it with maximum XOR so far. 15 proposed_max = maximum_xor | (1 << i)</pre> 16 17 18 # Check if there's a pair of prefixes which XOR equals our proposed maximum so far.

maximum_xor = proposed_max # Update the maximum XOR since we found a pair.

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

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).

mask = mask | (1 << i) # Update the mask to include the next bit.

for (Integer prefix : prefixes) { 26 27 if (prefixes.contains(prefix ^ guessedMaximumXOR)) { 28 30

```
17
           for (int i = 30; i >= 0; --i) {
18
               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
19
                   node->children[bit] = new Trie();
20
21
22
               node = node->children[bit]; // Move to the corresponding child
23
24
25
26
       // Function to search for the maximum XOR of a number with the existing numbers in the Trie
       int search(int number) {
27
28
           Trie* node = this;
           int result = 0; // Initialize the result to 0
29
           for (int i = 30; i >= 0; --i) {
30
                int bit = (number >> i) & 1; // Extract the i-th bit
31
32
               // Try to find the opposite bit in the Trie for maximized XOR
33
               if (node->children[bit ^ 1]) {
34
                    result = (result << 1) | 1; // If found, update the result with set bit at current position
35
                   node = node->children[bit ^ 1]; // Move to the opposite bit child
36
               } else {
37
                   result <<= 1; // Else, result is updated just by shifting, bit remains unset
                   node = node->children[bit]; // Move to the same bit child
38
39
40
41
           return result; // Return the maximum XOR value found
42
43 };
44
   // 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) {
49
           Trie* trie = new Trie(); // Initialize Trie
50
51
           int ans = 0; // Initialize answer to 0
52
           for (int num : nums) {
53
               trie->insert(num); // Insert each number into the Trie
54
               ans = std::max(ans, trie->search(num)); // Update the answer with the maximum XOR value found so far
55
```

37 38 40

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.

XOR operations.

grow with n.

• We then iterate again through all unique prefixes, which are at most n in the worst case. For each prefix, we perform an 0(1) time check to see if a certain XOR-combination could exist in the set. Therefore, this operation also takes 0(n) time in the worst case.

Multiplying these together, the time complexity is 0(31n), which simplifies to 0(n) because 31 is a constant factor which does not

• 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

• 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.

- **Space Complexity**
- Therefore, the space required for the set is O(n). Considering the space used by max, mask, current, and flag are all constant 0(1) space overheads, the total space complexity is

• The set s can have at most n elements as each number contributes a unique left-prefixed bit representation to the set.

facilitates constant time access to check if a certain prefix exists, which is crucial for optimizing the solution.

5. Return the max value which is the answer.

value is found considering each bit position.

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

represent).

ii. Bit position 3

XOR.

class Solution:

6

19

20

21

22

23

24

25

26

9

11

13

14

15

16

17

19

20

21

22

24

25

34

35

36

38

37 }

 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.

- Example Walkthrough
- Since both 16 and 0 are in s, we can update max = flag. Now max is 10000 in binary, or 16 in decimal.
- Update max to flag. Now max is 24.
- **Step 4**: After loop ends, max holds the maximum XOR that we can find.
- prefixes = set() # Create a set to store prefixes of the current length. 9 10 11 # Collect all prefixes with bits up to the current bit.

for prefix in prefixes:

return maximum_xor

1 import java.util.HashSet;

int mask = 0;

import java.util.Set;

class Solution {

for i in range(31, -1, -1):

31 32 33

C++ Solution

6 class Trie {

7 public:

8

9

10

11

12

13

14

15

16

1 #include <vector>

2 #include <string>

#include <algorithm>

- 2 interface TrieNode { children: (TrieNode | undefined)[]; 4 } // Initializes a Trie node with undefined children 7 function createTrieNode(): TrieNode { return { children: [undefined, undefined] };

Typescript Solution

11 // Inserts a number into the Trie

for (let i = 30; i >= 0; --i) {

for (let i = 30; i >= 0; --i) {

if (node.children[bit ^ 1]) {

let node = trie;

let node = trie;

56

57

59

9

10

13

14

15

16

17

18

19

20

21

22

23

26

27

28

29

30

31

32

41

45

46

47

48

49

50

51

52

53

54

56

55 }

});

58 };

- node = node.children[bit ^ 1] as TrieNode; // Move to the opposite bit child node 33 34 } else { 35 result <<= 1; // Else, result is updated just by shifting, bit remains unset 36 node = node.children[bit] as TrieNode; // Move to the same bit child node return result; // Return the maximum XOR value found
- 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

Time and Space Complexity

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

0(n).