2429. Minimize XOR Medium <u>Greedy</u> **Bit Manipulation**

Problem Description

(bits which are 1) as in another given positive integer num2. Second, when x is XORed with a given positive integer num1, the result should be as small as possible. This problem guarantees a unique solution for the given integers. To give an example, if we have num1 = 2 (binary 10) and num2 = 3 (binary 11), an x with the same number of set bits as num2

The problem asks us to find a positive integer x that adheres to two conditions. First, x must have the same number of set bits

would be 3 (binary 11), and \times XOR num1 would result in a minimal value, which is 1.

Set bits are important in computing because they often represent boolean flags or other binary indicators within a number's binary representation.

To solve this problem, we need to balance between matching the number of set bits in x and num2, and minimizing the XOR with

Intuition

num1. Creatively manipulating the bits is essential here. Beginning with the count of set bits (bit_count) in both num1 and num2 provides us with the targets we need to hit. If num1 has

more set bits than num2, we need to turn off some bits in num1. We do this by performing the operation num1 &= num1 - 1 which turns off the rightmost set bit in num1.

Conversely, if num1 has fewer set bits than num2, we need to turn on some bits in num1. The most efficient way to do this is to turn on the rightmost unset bit, which can be done using num1 |= num1 + 1.

Use these operations to adjust the number of set bits in num1 to match num2. This process relies on the bitwise nature of

numbers and the properties of XOR to ensure that the value remains minimal. No sorting or array transformation is necessary, making this algorithm efficient and elegant. We are certain that this solution is unique because of the constraints set by the problem and the deterministic nature of the

smallest possible amounts, guaranteeing the minimal possible result for \times XOR num1. Solution Approach

operations used to adjust the set bits. By focusing on changing the rightmost bits first, we affect the value of num1 by the

The implementation of the solution to the given problem follows a simple yet elegant approach, which uses standard bitwise

operations to directly manipulate the bits of the input numbers, thereby maintaining a very efficient time complexity. Here are the steps involved in the implementation using Python:

num2.

Counting Set Bits: We begin by counting the number of set bits (1s) in the binary representation of num1 and num2. Python's bit_count() function gives us this value directly. This is critical because we need to make the number of set bits in x match

cnt1 = num1.bit count() cnt2 = num2.bit count()

```
Reducing Excess Set Bits: If num1 has more set bits than num2 (cnt1 > cnt2), it means we need to reduce the number of set
bits in num1. We do this by turning off set bits from the least significant end. The operation num1 &= num1 - 1 will unset the
rightmost set bit in num1 each time it is executed. We continue this in a loop until the number of set bits in num1 (cnt1) is the
```

same as in num2 (cnt2). while cnt1 > cnt2: num1 &= num1 - 1 cnt1 -= 1

Increasing Insufficient Set Bits: If num1 has fewer set bits than num2 (cnt1 < cnt2), we need to increase the number of set

bits in num1. We target the least significant bits to have minimal impact on the resulting value. The operation num1 |= num1 +

1 effectively turns on the first off bit (from the right), increasing the count by one. We repeat this until the count matches that

Returning the Result: After ensuring that num1 has the same number of set bits as num2, and thus transforming num1 into x,

```
of num2.
while cnt1 < cnt2:
    num1 \mid = num1 + 1
    cnt1 += 1
```

return num1 No additional data structures are needed. The pattern lies in bitwise manipulation, specifically toggling bits based on their state and position, all the while prioritizing changes to bits of lower significance to achieve the minimal XOR value. By repeatedly

applying basic bitwise operations, we incrementally mold num1 to into the desired x that meets all the problem's conditions.

we return x which will satisfy the condition that x XOR num1 is minimized.

```
Let's illustrate the solution approach with a small example. Suppose we have num1 = 8 (binary 1000) and num2 = 5 (binary 101).
We want to find x, a positive integer that matches the number of set bits in num2 and minimizes x XOR num1.
```

• We perform the operation num1 |= num1 + 1. The binary representation of num1 + 1 is 1001. • After using the |= operation, num1 turns into 1001 (which is 9 in decimal), and now cnt1 = 2, matching cnt2.

Example Walkthrough

Step 1: Counting Set Bits

Step 2 & 3: Adjusting Set Bits cnt1 < cnt2 (1 < 2), so we need to increase the number of set bits in num1.

```
Step 4: Returning the Result

    No further actions needed because cnt1 now equals cnt2.
```

bit_count_num2 = num2.bit_count()

bit_count_num1 -= 1

num1 |= number to or

bit_count_num1 += 1

while bit count num1 < bit count num2:</pre>

Decrement the counter for the number of set bits in num1

Add this number to num1 (same as ORing it with num1)

Increment the counter for the number of set bits in num1

Return the modified num1 which has the same number of set bits as num2

// After the adjustments, num1 should have the same number of 1-bits as num2,

// If num1 has more set bits than num2, turn off set bits from the LSB side until they match

// $^{\circ}$ $^{\circ}$ operation with (numl - 1) turns off the rightmost set bit in num1

// Minimize the XOR value of num1 by making the number of 1-bits in num1 similar to num2.

function minimizeXor(num1: number, num2: number): number {

// Count the number of 1-bits in num1 and num2

let count1 = bitCount(num1);

let count2 = bitCount(num2);

// and this is the minimized XOR value we are looking for

// Count the number of 1-bits (set bits) in num1 and num2

• The x we've found is 9, which maintains the set bit requirement.

• For num1 (1000), the number of set bits (cnt1) is 1.

• For num2 (101), the number of set bits (cnt2) is 2.

- The result is minimal because we've only turned on the least significant bit that was originally off in num1, which has the smallest impact on the value when XORed with num1.
- Finally, x XOR num1 equals 9 XOR 8 which is 1 in binary (0001), and indeed this is the smallest possible result for x XOR num1. Thus, x = 9 is the solution.

If num1 has fewer set bits than num2, we need to increase the number of set bits in num1

class Solution: def minimizeXor(self, num1: int, num2: int) -> int: # Count the number of set bits (1s) in num1 and num2 bit count num1 = num1.bit count()

If num1 has more set bits than num2, we need to decrease the number of set bits in num1 while bit count num1 > bit count num2: # Remove the rightmost set bit from num1 using (num1 & num1 - 1) num1 &= num1 - 1

Get the number that is the smallest power of two greater than num1, which doesn't have a set bit in common with num1 number to or = num1 + 1while num1 & number to or: number_to_or = number_to_or << 1</pre>

return num1

return num1;

int minimizeXor(int num1, int num2) {

while (count1 > count2) {

--count1;

num1 &= (num1 - 1);

int count1 = builtin popcount(num1);

int count2 = __builtin_popcount(num2);

// Decrement the set bit count for num1

Java

Solution Implementation

Python

```
class Solution {
   public int minimizeXor(int num1, int num2) {
       // Count the number of 1-bits (set bits) in both num1 and num2
       int count1 = Integer.bitCount(num1);
       int count2 = Integer.bitCount(num2);
       // If count1 is greater than count2, we need to turn off some 1-bits in num1
       while (count1 > count2) {
           // Turn off (unset) the rightmost 1-bit in num1
           num1 \&= (num1 - 1);
           // Decrement count1 as we have reduced the number of 1-bits by one
           --count1;
       // If count1 is less than count2, we need to turn on (set) additional 1-bits in num1
       while (count1 < count2) {</pre>
           // Turn on (set) the rightmost 0-bit in num1
           | num1 | = (num1 + 1);
           // Increment count1 as we have increased the number of 1-bits by one
           ++count1;
```

C++

public:

class Solution {

```
// If num1 has fewer set bits than num2, turn on the unset bits from the LSB side until they match
        while (count1 < count2) {</pre>
            // '|' operation with (num1 + 1) turns on the rightmost unset bit in num1
            num1 |= (num1 + 1);
            // Increment the set bit count for num1
            ++count1;
        // Return the modified num1 after trying to match the set bit count to num2
        return num1;
TypeScript
// Calculate the number of 1-bits in the binary representation of a number
function bitCount(number: number): number {
    // Subtracting a bit pattern from its right-shifted version clears the set least significant bit
    number = number - ((number >>> 1) & 0x555555555);
    // Perform binary partition and sum to count the bits
    number = (number \& 0x33333333) + ((number >>> 2) \& 0x333333333);
    // Combine partitioned sums with a further partition
    number = (number + (number >>> 4)) & 0x0f0f0f0f;
    // Sum all counts using cascaded summing
    number = number + (number >>> 8);
    number = number + (number >>> 16);
    // Mask the result to get the final count
    return number & 0x3f;
```

```
// If num1 has more 1-bits than num2, remove 1-bits from num1
   for (; count1 > count2; --count1) {
       num1 &= num1 - 1; // Remove the lowest set bit from num1
   // If num1 has fewer 1-bits than num2, add 1-bits to num1
   for (; count1 < count2; ++count1) {</pre>
       num1 |= num1 + 1; // Add the lowest non-set bit to num1
   return num1; // Return the modified num1 with the number of 1-bits resembling that of num2
class Solution:
   def minimizeXor(self, num1: int, num2: int) -> int:
       # Count the number of set bits (1s) in num1 and num2
       bit count num1 = num1.bit count()
       bit_count_num2 = num2.bit_count()
       # If num1 has more set bits than num2, we need to decrease the number of set bits in num1
       while bit count num1 > bit count num2:
           # Remove the rightmost set bit from num1 using (num1 \& num1 - 1)
           num1 \&= num1 - 1
           # Decrement the counter for the number of set bits in num1
           bit_count_num1 -= 1
       # If num1 has fewer set bits than num2, we need to increase the number of set bits in num1
       while bit count num1 < bit count num2:</pre>
           # Get the number that is the smallest power of two greater than num1, which doesn't have a set bit in common with num1
           number to or = num1 + 1
           while num1 & number to or:
                number_to_or = number_to_or << 1</pre>
           # Add this number to num1 (same as ORing it with num1)
```

Time Complexity

Time and Space Complexity

return num1

num1 |= number to or

bit_count_num1 += 1

operation involves a loop that repeats as long as cnt1 is greater than cnt2. Inside the loop, it performs num1 &= num1 - 1, which removes a 1 bit from num1 each time. Since this operation depends on the number of 1 bits to be removed, its time

The given code has two main operations that affect the time complexity:

Increment the counter for the number of set bits in num1

Return the modified num1 which has the same number of set bits as num2

complexity is O(K), where K is the difference in the number of 1 bits between num1 and num2. Increasing the number of 1 bits in num1 to match the number of 1 bits in num2 when num1 has fewer 1 bits than num2. This operation involves another loop that repeats while cnt1 is less than cnt2, performing num1 |= num1 + 1 to add 1 bits to num1. Since num1 + 1 includes at least one 1 bit at the lowest-order position (binary form), there's a guarantee that a new 1 bit is added to num1 each time this operation completes. Thus, its time complexity is also 0(K).

Reducing the number of 1 bits in num1 to match the number of 1 bits in num2 when num1 has more 1 bits than num2. This

Hence, the overall time complexity of the code is O(K), where K is the absolute difference in the number of 1 bits between num1 and num2. **Space Complexity**

The space complexity of the given code is 0(1). Apart from the input numbers num1 and num2, the code only uses a fixed number of integer variables (cnt1 and cnt2), and no additional structures or recursive calls that consume memory proportional to the input size.