MISTAKES

Let's go through the important points, mistakes, and complexities based on today's problem:

Problem Recap:

We were discussing a bitwise operation problem where the goal was to find the **complement of a number** in its binary representation, and you were asking about **bitwise XOR operations**, understanding how to use a **mask**, and how to manipulate bits.

Key Points:

1. **Bitwise XOR (^) Operation**:
    - The XOR operation compares each bit of two numbers and returns 1 if the corresponding bits are different, and 0 if they are the same.
    - **Commutative**: a ^ b is the same as b ^ a. So, there is no practical difference between num ^ mask and mask ^ num. Both yield the same result.
    - **Properties**:
        - x ^ x = 0: XOR-ing a number with itself results in 0.
        - x ^ 0 = x: XOR-ing with 0 leaves the number unchanged.
        - x ^ 1 flips the least significant bit (LSB), turning 1 to 0 or 0 to 1.
2. **Complement of a Number**:
    - To find the **binary complement** of a number, you need to flip all the bits (i.e., change 1s to 0s and 0s to 1s).
    - A common method to do this is to use a **mask** and perform XOR with the number. If mask is created with all 1s for the bits in the number's binary representation, num ^ mask will flip all the bits of num.
3. **Masking Technique**:
    - **Creating a Mask**:
        - A mask can be created by shifting 1 left and then performing XOR operations to flip the bits. The mask is often a number with all 1s in the bit positions of interest.
    - **Example**:
    - int num = 5; // Binary: 101
    - int mask = (1 << (bit_length)) - 1; // Create a mask of length bit_length
    - int complement = num ^ mask; // Flip the bits of num
4. **Complexity Considerations**:
    - **Time Complexity**:

- Creating the mask using bit manipulation (e.g., mask = (1 << num_length) - 1) typically runs in **O(log n)**, where n is the value of num because the number of bits in a number grows logarithmically with respect to its value.
- **XOR** operation takes **O(1)** because it's a single bitwise operation on two integers.

- **Space Complexity**:
  - **O(1)** because we are using a constant amount of space for the result and the mask (both are integers).

Mistakes in Code:

Here are a few common mistakes that could occur in this type of problem:

1. **Wrong Mask Size**:
   - If the mask size does not match the number's bit length, you might end up flipping more bits than intended or leaving the higher bits untouched. Always ensure that the mask is created based on the number's bit length.
2. **Not Accounting for Edge Cases**:
   - **Zero**: If num is 0, its complement will be the maximum number representable in the same bit length (e.g., 111...111 in binary).
   - **Single Bit Numbers**: Numbers like 1 or 0 need special attention when using masks to ensure the complement is calculated correctly.

Optimized Approach for Finding the Complement:

Here's the optimized approach:

```
class Solution {
public:
  int findComplement(int num) {
    // Create a mask with the same number of bits as num
    int mask = 1;

    // Shift mask left until it's larger than num
    while (mask <= num) {
      mask <<= 1;  // Shift left
    }

    // Subtract 1 to get a mask where all bits up to the length of num are 1's
    mask -= 1;
```

```
        // XOR num with the mask to flip all bits
        return num ^ mask;
    }
};
```

Explanation:

1. **Create a Mask**:
    - The mask starts with 1. We shift it left (mask <<= 1) until it becomes greater than num.
    - Subtract 1 to get a mask that matches the number of bits in num.
2. **XOR Operation**:
    - XOR the mask with num. This flips all the bits of num within the valid bit range, giving you the complement.

Complexity:

- **Time Complexity**: O(log n) because we shift the mask until it exceeds the number of bits in num.
- **Space Complexity**: O(1) because only a few integers are used (mask, num).

Summary:

- The XOR operation is commutative, so order does not matter (num ^ mask is the same as mask ^ num).
- The complement of a number involves flipping its bits using a mask.
- You need to ensure the mask has the same number of bits as the number to get the correct complement.
- The optimal time complexity for this approach is O(log n), and the space complexity is O(1).

This is a clean and efficient way to solve the problem using bitwise operations!