Number Complement — Complete Documentation

1. **Problem Statement**

   Given a positive integer num, return *its complement*. The complement of a number is formed by flipping all bits in its binary representation, excluding any leading zeros.

   - Example 1:

     Input: 5 → Binary: 101

     Output: 2 → Complement: 010
   - Example 2:

     Input: 1 → Binary: 1

     Output: 0 → Complement: 0

   Constraints:

   - $1 <= num < 2^{31}$

2. **Intuition**

   To find the complement of a number:

   - Convert the number to binary (ignoring leading zeros).
   - Flip all bits (change 1 to 0 and 0 to 1).
   - Convert the flipped binary number back to decimal.

   This can be done efficiently using bit manipulation.

3. **Key Observations**

   - Python's bin() representation includes a 0b prefix. We must ignore this.
   - The number of bits required to represent num can be found using num.bit_length().
   - A bitmask of the same length with all bits set to 1 can be used to flip the bits using XOR.

4. **Approach**

- Find the bit length of num (excluding leading zeros).
- Create a mask with all bits set to 1 of the same length.
- Perform XOR between num and the mask to get the complement.

Example Walkthrough:

- num = 5 → binary: 101 → bit_length = 3
- Mask = 111 (binary) = 7 (decimal)
- $5^7 = 2$ → Complement: 010

5. **Edge Cases**

- num = 1 → Binary: 1 → Complement: 0
- Since the problem ensures num >= 1, we do not have to handle zero or negative inputs.

6. **Complexity Analysis**

 Time Complexity:

- O(1)
  Bit manipulation operations and bit_length() are constant times for 32-bit integers.

 Space Complexity:

- O(1)
  No extra space is used other than variables.

7. Alternative Approaches

🔁 String-Based Approach:

- Convert num to a binary string.
- Flip each bit manually using string operations.
- Convert back to integer using int(string, 2).

```
def findComplement(num):
    binary = bin(num)[2:]  # Remove '0b'
    flipped = ''.join('1' if b == '0' else '0' for b in binary)
    return int(flipped, 2)
```

⛔ Less efficient due to string operations.

8. Test Cases

Input	Binary	Complement (Binary)	Output
5	101	010	2
1	1	0	0
10	1010	0101	5
7	111	000	0
100	1100100	0011011	27

9. Final Thoughts

- This problem is a great example of how bit manipulation can simplify issues that otherwise seem string-oriented.
- The key is to understand how XOR works and how to generate masks efficiently.
- The solution is concise, optimal, and avoids unnecessary conversions.