1009. Complement of Base 10 Integer

Easy Bit Manipulation Leetcode Link

Problem Description

The problem requires us to determine the complement of a given integer `n`. The complement operation flips every bit in the binary representation of the number, which means that each '1' changes to '0' and each '0' to '1'. This is similar to finding the bitwise NOT operation in most programming languages, but with a caveat: we only flip the bits up to the most significant '1' in the original number and ignore all leading '0's. For example, if `n` has the binary representation of '101', it has leading '0's that are not represented like '00000101'. After flipping the bits, the complement would be '010'.

The challenge lies in finding an efficient way to perform this task programmatically without manually converting the number to its binary string representation and back. Also, we have to ensure that we are not including any leading '0's that might appear if the number were represented using a fixed number of bits (like 32 bits in many computer systems).

Intuition

The solution to this problem uses bitwise operations to find the complement without converting the number to a binary string. The intuition comes from observing how binary numbers work and utilizing bit manipulation to find the complement:

- We iterate over the bits of integer `n`, starting from the most significant bit (MSB) to the least significant bit (LSB). In a 32-bit integer, the MSB would be at position 31 (0-indexed).
- We need to find where the first '1' bit from the left (MSB) is located in `n`, skipping all leading '0's.
- Once the first '1' bit is encountered, each subsequent '0' bit in `n` becomes '1' in the answer, and each '1' bit in `n` does not contribute to the answer (remains '0').
- The algorithm creates an initial `ans` variable of 0 and starts flipping the bits of `ans` when the first '1' is found in `n`.
- Bitwise AND `(&)` operation is used to test the bit at each position (`1 << i`). If the bit is '0', then we bitwise OR `(|)` the `ans` with `1 << i` to set the corresponding bit to '1'.

The code effectively loops through the bits of `n` using a `for` loop that checks 31 bits (adjust this according to the size of the integers you're working with). It uses two flags: `find` to mark that we've found the first '1' bit and should start flipping bits, and `b` to hold the result of the bitwise AND operation for the current bit. If `find` is `True`, and `b` is `0`, the `ans` variable is updated using bitwise OR with `1 << i` to flip the current bit to '1'. If `n` is `0`, it's a special case, and the complement would be `1`.

Solution Approach

To implement the solution to find the complement of an integer, the following steps are taken using bit manipulation techniques:

1. First, we handle the edge case where `n` is `0`. Since the binary representation of `0` is just '0' and its complement is '1', we immediately return `1` without further processing.

2. Then, we initialize an answer variable `ans` to `0`. This variable will accumulate the result bit by bit.

3. The variable `find` is initialized to `False`. This boolean flag will indicate when we've encountered the first '1' bit from the left in the binary representation of `n`. We do not want to flip any leading '0's, so we should start flipping bits only after `find` is set to `True`.

4. We then iterate through the potential bit positions of `n` using a for loop with the range `30` down to `0`, which covers up to 31 bits for a non-negative integer in a 32-bit system. The loop index `i` represents the bit position we're checking, starting from the most significant bit.

5. Inside the loop, the bitwise AND operation `b = n & (1 << i)` checks if the bit at position `i` in `n` is a '1'. The expression `1 << i` creates a number with a single '1' bit at the `i`-th position and '0's elsewhere.

6. If `find` is `False` and `b` is `0`, that means we are still encountering leading '0's, so we continue to the next iteration without changing `ans`.

7. Once we find the first '1' (when `b` is not `0`), we set `find` to `True` to indicate that we should start flipping bits.

8. Thereafter, for every position `i` where `b` is `0` (indicating the bit in `n` was '0'), we flip the bit to '1' in `ans` by performing the bitwise OR operation `ans |= 1 << i`.

The use of bitwise operations makes this solution very efficient, as no string conversion or arithmetic operations with potentially large numbers are required. The core algorithm iterates through the bits of the number once, making it run in `O(1)` time with respect to the size of the integer (since the number of bits is constant for a standard integer size) and `O(1)` space because it only uses a fixed number of extra variables.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume our input number `n` is `5`, which has a binary representation of `'101'`.

1. First, we check if `n` is `0`. If it were `0`, we would return `1`. But since `n` is `5`, we move to the next step.

2. We initialize the answer variable `ans` to `0`. This will hold the result of the complemented bits.

3. We set our flag `find` to `False` as we have not yet encountered the first '1' from the left.

4. We then loop from `30` down to `0`. For simplicity in this example, we will only loop from `2` down to `0` because `5` is a small number and only needs 3 bits to represent it in binary.

5. During each iteration, we use the bitwise AND operation `b = n & (1 << i)` to test if the current bit is '1'.
   - On the first iteration (`i = 2`), `b = 5 & (1 << 2)` which is `5 & 4` or `101 & 100` in binary, which equals `100`. Since `b` is not `0`, we set `find` to `True`.

6. Subsequent iterations will now flip bits after the most significant '1' has been found.

7. On the second iteration (`i = 1`), `b = 5 & (1 << 1)` which is `5 & 2` or `101 & 010` in binary, which equals 0. Since `find` is `True` and `b` is `0`, we flip the `i`-th bit of `ans`: `ans |= 1 << 1`. Now, `ans = 0 | 010`, so `ans` is now `2` in decimal.

8. On the last iteration (`i = 0`), `b = 5 & (1 << 0)` which is `5 & 1` or `101 & 001` in binary, which equals `1`. There is no flipping since `b` is not `0`.

Therefore, the binary complement of '101' (the binary representation of `5`) is '010', which in decimal is `2`. The final output of the complement of `5` is `2`.

Python Solution

```python
class Solution:
    def bitwiseComplement(self, N: int) -> int:
        # If the input is 0, we know the bitwise complement is 1
        if N == 0:
            return 1

        # Initialize the answer to 0
        answer = 0
        # This variable is to determine when to start flipping bits
        found_first_one = False

        # Iterate through 31 bits of the integer to include the case of a 32-bit integer
        for i in range(31, -1, -1):
            # Check if the bit at the ith position is set (1)
            bit_is_set = N & (1 << i)

            # Skip leading zeros until we find the first set bit
            if not found_first_one and bit_is_set == 0:
                continue

            # The first set bit is found
            found_first_one = True
            # If the current bit is 0, set the corresponding bit in the answer
            if bit_is_set == 0:
                answer |= 1 << i

        # Return the computed bitwise complement
        return answer

```

Java Solution

```java
class Solution {
    // Method that calculates the bitwise complement of a given integer 'n'
    public int bitwiseComplement(int n) {
        // Check for the base case where n is 0,
        // the bitwise complement of 0 is 1.
        if (n == 0) {
            return 1;
        }
        // Initialize answer to 0
        int answer = 0;
        // A flag to indicate when the first non-zero bit from the left is found
        boolean foundFirstNonZeroBit = false;
        // Iterate from the 30th bit to the 0th bit,
        // because an integer in Java has 31 bits for the integer value
        // and 1 bit for the sign.
        for (int i = 30; i >= 0; --i) {
            // Check if i-th bit in n is set
            int bit = n & (1 << i);
            // Ignore leading zeroes in n,
            // we do not want to take them into the complement calculation.
            if (!foundFirstNonZeroBit && bit == 0) {
                continue;
            }
            // Once the first non-zero bit is found, we set the flag to true
            foundFirstNonZeroBit = true;
            // If the current bit is 0, its complement is 1,
            // and we set the corresponding bit in the answer.
            if (bit == 0) {
                answer |= (1 << i);
            }
            // Note: There is no need to explicitly handle the case when the bit is 1,
            // because its complement is 0 and the answer is already initialized to 0.
        }
        // Return the computed bitwise complement of n
        return answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int bitwiseComplement(int n) {
        if (n == 0) {
            // The complement of 0 is 1, as all bits are flipped.
            return 1;
        }

        int result = 0; // This will hold the result of the complement.
        bool foundOne = false; // Flag to check if a '1' bit has been found.

        // Loop through the bits of the integer, starting from the most significant bit (MSB).
        for (int i = 30; i >= 0; --i) {
            int bit = n & (1 << i); // Get the i-th bit.

            // Skip the leading zeroes to find the first '1'.
            if (!foundOne && bit == 0) continue;

            // Mark that we have found the first '1', so we include the rest of the bits.
            foundOne = true;

            // If the current bit is '0', flip it to '1' in the result.
            if (bit == 0) {
                result |= (1 << i);
            }
            // Otherwise, the bit is '1' and the result remains '0' (implicitly flipped to '0').
        }

        // Return the bitwise complement of the original number.
        return result;
    }
};

```

Typescript Solution

```typescript
function bitwiseComplement(n: number): number {
    if (n === 0) {
        // The complement of 0 is 1, as all bits are flipped.
        return 1;
    }

    let result: number = 0; // This will hold the result of the complement.
    let foundOne: boolean = false; // Flag to check if a '1' bit has been found.

    // Loop through the bits of the integer, starting from the most significant bit (MSB).
    for (let i: number = 30; i >= 0; --i) {
        let bit: number = n & (1 << i); // Get the i-th bit.

        // Skip the leading zeroes to find the first '1'.
        if (!foundOne && bit === 0) continue;

        // Mark that we have found the first '1', so we include the rest of the bits.
        foundOne = true;

        // If the current bit is '0', flip it to '1' in the result.
        if (bit === 0) {
            result |= (1 << i);
        }
        // Otherwise, the bit is '1' and the result remains '0' (implicitly flipped to '0').
    }

    // Return the bitwise complement of the original number.
    return result;
}

```

Time and Space Complexity

The time complexity of the given code is `O(1)` because the loop runs for a maximum of 31 iterations (since the loop is from i = 30 to i = -1), which is a constant time operation irrespective of the input size `n`.

The space complexity of the code is also `O(1)`, as there's a fixed and limited amount of extra space being used (variables `ans`, `find`, `b`, and constant space for `i`). No additional space that scales with input size `n` is being utilized.