476. Number Complement

Easy | Bit Manipulation

Problem Description

In this problem, we are given a positive integer `num`. Our task is to return the complement of this number. The complement is defined as the number obtained by switching all bits of its binary representation: Every `0` bit becomes a `1`, and every `1` becomes a `0`.

For example, if the input is `5`, its binary form is `101`. Its complement will be `010` which, in decimal, is `2`. The challenge is to compute this complement without directly manipulating the binary string.

Intuition

The key to solving this problem lies in understanding bitwise operations. In particular, the XOR operation becomes very useful. XOR, or exclusive or, gives us a `1` bit whenever the bits in the same position of the two numbers are different, and a `0` bit when they are the same.

So, if we XOR the given number with a number that has all bits set to `1`, and is of the same length, we will effectively switch all bits of the original number - `0`s will become `1`s, and `1`s will become `0`s. This is exactly the complement.

To get a number with all bits set to `1` and of the same length as `num`, we can use the fact that a power of two, when decreased by `1`, will give us a number with all bits set to `1` that is one less in size. For instance, `2^3` is `1000` in binary, and `2^3 - 1` is `0111`.

Solution Approach

The solution provided uses bitwise operators to obtain the complement of the given integer. There are no additional data structures used, and the entire procedure relies on built-in operations and an understanding of how numbers are represented in binary.

1. First, we convert the integer to its binary representation as a string using `bin(num)`. `bin()` is a built-in Python function that takes an integer and returns its binary representation, prefixed with `0b`.
2. We then take the substring excluding the first two characters (`'0b'`) by using slice notation `[2:]`. This gives us just the binary digits.
3. We now look for the length of this binary string representation using `len()`. This length represents the number of bits in the original integer's binary representation.
4. We raise `2` to the power of this length, which gives us a number that in binary has a `1` followed by as many `0`s as the length we found: `2 ** len(bin(num)[2:])`.
5. We then subtract `1` from this number. Because of the nature of binary numbers, subtracting `1` from a power of `2` gives us a sequence of `1`s. For example, `2 ** 3` is `1000` in binary, and `2 ** 3 - 1` is `0111`. This sequence of `1`s matches the length of our input number `num`.
6. Finally, we perform an XOR operation between the original number `num` and the number with the sequence of `1`s to flip all the bits. In Python, the XOR operator is `^`.
7. The result of this operation `num ^ (2 ** (len(bin(num)[2:])) - 1)` is the complement of the input integer `num`.

Here is a step-by-step conversion of `5` as an example:

- `bin(5)` is `'0b101'`
- Slicing off `'0b'` gives `'101'`
- `len('101')` is `3`
- `2 ** 3` is `8` which is `1000` in binary
- `2 ** 3 - 1` is `7` which is `0111` in binary
- `5 XOR 7` (or `101 XOR 0111`) gives `010` which is `2` in decimal

No explicit loops or complex data structures are required, making this approach efficient and concise.

Example Walkthrough

Let's illustrate the solution approach using the number `10` as a small example.

1. Convert the integer `10` to its binary representation. Using the `bin()` function in Python, we have `bin(10)`, which yields `'0b1010'`.
2. Remove the `'0b'` prefix by slicing. `'0b1010'[2:]` becomes `'1010'`.
3. Find the length of the binary string `'1010'`: `len('1010')`, which is `4`.
4. Compute `2` raised to the power of this length: `2 ** len('1010')` is `2 ** 4`, which gives us `16`. In binary, `16` is `10000`.
5. Subtract `1` from `16` to get a number with `4` bits set to `1`: `16 - 1` equals `15`, and its binary form is `'1111'`.
6. Perform an XOR operation between the original number `10` (binary `'1010'`) and `15` (binary `'1111'`): `10 ^ 15`. In binary form, this operation looks like `1010 XOR 1111`.
7. The result of the XOR operation is `0101` in binary, which corresponds to `5` in decimal.

The complement of `10` is therefore `5`. Using the aforementioned solution approach, the steps can be applied to any positive integer to find its binary complement without dealing with the binary string directly.

Solution Implementation

Python

```python
class Solution:
    def findComplement(self, num: int) -> int:
        # Calculate the length of the binary representation of the number excluding the '0b' prefix.
        binary_length = len(bin(num)) - 2

        # Create a mask with all 1's that is the same length as the binary representation of the number.
        mask = (2 ** binary_length) - 1

        # XOR the number with the mask to flip all the bits and obtain its complement.
        return num ^ mask
```

Java

```java
public class Solution {
    // This method computes the bitwise complement of a positive integer.
    public int findComplement(int num) {
        int complement = 0; // This will hold the result, the bitwise complement of 'num'.
        boolean significantBitFound = false; // Flag to indicate when the first '1' bit from the left is found.

        // Iterate from the 30th bit to the 0th bit (31st bit is not considered for a positive Integer).
        for (int i = 30; i >= 0; i--) {
            int bit = num & (1 << i); // Isolate the i-th bit of 'num'.

            // Skip leading zeros - we don't want to consider them.
            if (!significantBitFound && bit == 0) {
                continue;
            }

            // Once the first non-zero bit is found, we flip the flag to true.
            significantBitFound = true;

            // If the current bit is 0, set the corresponding bit in the result.
            if (bit == 0) {
                complement |= (1 << i);
            }
            // No need to do anything if the bit is 1, as we want to flip it to 0 (which is the default).
        }

        return complement; // Return the computed complement.
    }
}
```

C++

```cpp
class Solution {
public:
    int findComplement(int num) {
        int complement = 0; // Initialize variable to store the complement
        bool foundNonZeroBit = false; // Flag to check when the first non-zero bit is found

        // Iterate from the 30th bit to the 0th bit (31st bit is not considered for a signed integer)
        for (int i = 30; i >= 0; --i) {
            int currentBit = (num & (1 << i)); // Check if the ith bit is set or not

            // Skip leading zeroes and look for the first 1
            if (!foundNonZeroBit && currentBit == 0) continue;

            // After the first non-zero bit is found, set foundNonZeroBit to true
            foundNonZeroBit = true;

            // If the current bit is 0, set the corresponding bit in complement
            if (currentBit == 0) {
                complement |= (1 << i);
            }
            // If the current bit is 1, it remains 0 in the complement,
            // so no action is needed since complement is initialized to 0
        }
        return complement; // Return the computed complement
    }
};
```

TypeScript

```typescript
function findComplement(num: number): number {
    // Initialize a variable to store the complement of the number
    let complement: number = 0;
    // Boolean flag to check when the first non-zero bit is found
    let foundNonZeroBit: boolean = false;

    // Iterate from the 30th bit to the 0th bit (avoiding the 31st bit for a signed integer)
    for (let i: number = 30; i >= 0; --i) {
        // Isolate the ith bit of 'num' to determine if it is set (1) or not (0)
        let currentBit: number = (num & (1 << i));

        // Skip the leading zeroes until the first set bit is found
        if (!foundNonZeroBit && currentBit === 0) {
            continue;
        }

        // Once the first non-zero bit is found, set the flag to true
        foundNonZeroBit = true;

        // If the current bit is 0, set the corresponding bit in 'complement'
        if (currentBit === 0) {
            complement |= (1 << i);
        }
        // Note: When the current bit is 1, the complement bit will remain 0.
        // This happens by default since 'complement' was initialized to 0.
    }

    // Return the computed complement of the original number
    return complement;
}

// Usage example:
let result = findComplement(5);
console.log(result); // Outputs the complement of 5
```


```python
class Solution:
    def findComplement(self, num: int) -> int:
        # Calculate the length of the binary representation of the number excluding the '0b' prefix.
        binary_length = len(bin(num)) - 2

        # Create a mask with all 1's that is the same length as the binary representation of the number.
        mask = (2 ** binary_length) - 1

        # XOR the number with the mask to flip all the bits and obtain its complement.
        return num ^ mask
```


Time and Space Complexity

Time Complexity

The time complexity of the function is `O(1)`. This is because the operations within the function take constant time. The length of the binary representation of the number (`len(bin(num)[2:])`) is proportional to the number of bits in `num`, which is a fixed size for standard data types (like 32-bit or 64-bit integers). The exponentiation operation `2 ** <bit_length>`, bitwise `XOR` operation, and subtraction are all completed in constant time regardless of the input size.

Space Complexity

The space complexity of the function is also `O(1)`. No additional space that grows with the input size is used by the algorithm. Only a fixed number of variables are used regardless of the size of the input number, which means the amount of memory used does not scale with `num`.