Complement of Base 10 Number (medium)

We'll cover the following

Problem Statement

Every non-negative integer N has a binary representation, for example, 8 can be represented as “1000” in binary and 7 as “0111” in binary.

The complement of a binary representation is the number in binary that we get when we change every 1 to a 0 and every 0 to a 1. For example, the binary complement of “1010” is “0101”.

For a given positive number N in base-10, return the complement of its binary representation as a base-10 integer.

Example 1:

```
Input: 8
Output: 7
Explanation: 8 is 1000 in binary, its complement is 0111 in binary, which is 7 in base-10.
```

Example 2:

```
Input: 10
Output: 5
Explanation: 10 is 1010 in binary, its complement is 0101 in binary, which is 5 in base-10.
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```java
import java.lang.Math;

class CalculateComplement {
    public static int bitwiseComplement(int n) {
      // TODO: Write your code here
      return -1;
    }

    public static void main( String args[] ) {
      System.out.println("Bitwise complement is: " + CalculateComplement.bitwiseComplement(8));
      System.out.println("Bitwise complement is: " + CalculateComplement.bitwiseComplement(10));
    }
}
```

Run | Save | Reset

Solution

Recall the following properties of XOR:

1. It will return 1 if we take XOR of two different bits i.e. `1^0 = 0^1 = 1`.
2. It will return 0 if we take XOR of two same bits i.e. `0^0 = 1^1 = 0`. In other words, XOR of two same numbers is 0.
3. It returns the same number if we XOR with 0.

From the above-mentioned first property, we can conclude that XOR of a number with its complement will result in a number that has all of its bits set to 1. For example, the binary complement of “101” is “010”; and if we take XOR of these two numbers, we will get a number with all bits set to 1, i.e., `101 ^ 010 = 111`

We can write this fact in the following equation:

```
number ^ complement = all_bits_set
```

Let's add 'number' on both sides:

```
number ^ number ^ complement = number ^ all_bits_set
```

From the above-mentioned second property:

```
0 ^ complement = number ^ all_bits_set
```

From the above-mentioned third property:

```
complement = number ^ all_bits_set
```

We can use the above fact to find the complement of any number.

How do we calculate 'all_bits_set'? One way to calculate `all_bits_set` will be to first count the bits required to store the given number. We can then use the fact that for a number which is a complete power of '2' i.e., it can be written as pow(2, n), if we subtract '1' from such a number, we get a number which has 'n' least significant bits set to '1'. For example, '4' which is a complete power of '2', and '3' (which is one less than 4) has a binary representation of '11' i.e., it has '2' least significant bits set to '1'.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```java
import java.lang.Math;

class CalculateComplement {
  public static int bitwiseComplement(int num) {
    // count number of total bits in 'num'
    int bitCount = 0;
    int n = num;
    while (n > 0) {
      bitCount++;
      n = n >> 1;
    }

    // for a number which is a complete power of '2' i.e., it can be written as pow(2, n), if we
    // subtract '1' from such a number, we get a number which has 'n' least significant bits set to '1'.
    // For example, '4' which is a complete power of '2', and '3' (which is one less than 4) has a binary
    // representation of '11' i.e., it has '2' least significant bits set to '1'
    int all_bits_set = (int) Math.pow(2, bitCount) - 1;

    // from the solution description: complement = number ^ all_bits_set
    return num ^ all_bits_set;
  }

  public static void main(String[] args) {
    System.out.println("Bitwise complement is: " + CalculateComplement.bitwiseComplement(8));
    System.out.println("Bitwise complement is: " + CalculateComplement.bitwiseComplement(10));
  }
}
```

Run Save Reset

Time Complexity

Time complexity of this solution is $O(b)$ where 'b' is the number of bits required to store the given number.

Space Complexity

Space complexity of this solution is $O(1)$.