1611. Minimum One Bit Operations to Make Integers Zero

Hard | Bit Manipulation | Memoization | Dynamic Programming | Leetcode Link

Problem Description

Given an integer `n`, the task is to calculate the minimum number of operations required to reduce it to `0`. The operations permitted are as follows:

1. Flip the rightmost (0th) bit in the binary representation of `n`.
2. Flip the `i`th bit if and only if the `(i-1)`th bit is `1`, and all bits from the `(i-2)`th down to the 0th bit are `0`.

It's a binary operation challenge that requires an understanding of bit manipulation and binary representation of numbers.

Intuition

The intuition behind the solution lies in recognizing a pattern that occurs with the binary representation of the numbers while performing the given operations. Upon iteratively flipping bits according to the rules, we can observe that the problem resembles the Gray code to binary conversion.

In Gray code, each subsequent number differs from the preceding one in only one bit. Similarly, while flipping the bits as per the given rules, when we reach zero, we would have effectively constructed a series of numbers in reverse Gray code order starting at `n`.

Therefore, the minimum number of operations required to reduce `n` to zero would be the unique number that we would get if we were to flip bits from `n` to zero following the rules. This number is obtained by performing a bitwise XOR (`^`) between the current answer and the right-shifted (`>>`) version of `n`. We continue this process until `n` becomes zero, at which point our `ans` would contain the minimum number of operations required.

To add further clarification, the bitwise XOR operation here effectively keeps track of flipping the bits, as XORing a bit with `1` changes it, and XORing it with `0` leaves it unchanged. The right shift operation simulates the dependency of flipping a given bit based on the state of its right neighbor, as described by the rules.

Solution Approach

The solution uses a simple while loop and bitwise operations to solve the problem with no need for additional data structures. The pattern used is akin to the iterative bit flipping carried out in a Gray code to binary conversion. Here's a step-by-step breakdown of the solution:

1. Initialize the `ans` variable to `0`. This variable will accumulate the number of operations performed.
2. Enter a while loop that continues as long as `n` is not `0`.
3. Within the loop, perform a bitwise XOR between `ans` and `n`. This operation updates `ans` with the accumulated result of the "flips" so far. The XOR operation is perfectly suited for this as it effectively toggles the bits—flipping a `1` to `0` or a `0` to `1`—based on whether there is a `1` in the corresponding position of `n` (analogous to performing the flip operation).
   - The expression `ans ^= n` means `ans = ans ^ n`, where `^` is the bitwise XOR operator.
4. Perform a right bitwise shift on `n` using `n >>= 1`. This operation prepares the next bit in `n` to be XORed with `ans` in the next iteration of the loop by moving all bits one place to the right. It is effectively checking whether the current bit should be flipped or not based on the rule of flipping bit `i` if bit `(i-1)` is set to `1` and all lesser significant bits are `0`. The right shift moves the bits down such that each "next" bit becomes the "current" bit for the next operation.
5. The loop continues, flipping bits one by one until `n` is reduced to `0`.
6. The final value of `ans` after the loop terminates is the total number of operations required. Return `ans`.

In this implementation, there is no need for complex algorithms or large data structures as the problem's constraints and behavior allow for a more direct and efficient approach using bitwise manipulation only.

The performance of this algorithm is determined by the number of bits in `n`, which is `O(log n)` since each iteration of the while loop shifts `n` one bit to the right until it becomes zero.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we are given the integer `n = 6`.

We need to determine the minimum number of operations required to reduce it to `0` following the rules provided.

First, let's look at the binary representation of `n`: `n = 6` in binary is `110`.

Now we will apply the operations defined in the problem description while keeping track of the number of operations.

1. Initialize `ans` to `0`.
2. Begin the while loop since `n` is not yet `0`.
   a. Perform `ans ^= n`:
      - `ans = 0`
      - `n` in binary is `110`
      - `ans ^= n` gives us `110`, so now `ans` is `110`.
   b. Perform `n >>= 1` (right shift `n`):
      - `n` after the shift is `11` in binary (which is `3` in decimal).
3. The next iteration starts as `n` is still not `0`.
   a. Perform `ans ^= n`:
      - `ans = 110` in binary.
      - `n` is now `11` in binary.
      - `ans ^= n` gives us `101`, so now `ans` is `101`.
   b. Perform `n >>= 1`:
      - `n` after the shift is `1` in binary (which is `1` in decimal).
4. Proceed to the next iteration since `n` is not `0`.
   a. Perform `ans ^= n`:
      - `ans = 101` in binary.
      - `n` is `1` in binary.
      - `ans ^= n` gives us `100`, so now `ans` is `100`.
   b. Perform `n >>= 1`:
      - `n` after the shift is `0` in binary (which is `0` in decimal).

At this point, `n` has become `0`, so the loop terminates. The `ans`, which represents the minimum number of operations required to reduce `n` to `0`, is `100` in binary. This is `4` in decimal.

Therefore, the minimum number of operations required to reduce `6` to `0` is `4`.

The steps taken illustrate the incremental changes in `n` and `ans`. Each loop iteration performs one or two operations complying with the rules: either flipping the rightmost bit or shifting right and preparing for the next bit flip.

The final binary number `100` represents a count of every flip that happened from `6` to `0`. Using this approach, for any given integer `n`, one can find the minimum number of operations required to reduce it to `0`.

Python Solution

```python
class Solution:
    def minimumOneBitOperations(self, n: int) -> int:
        # Initialize the answer to zero.
        result = 0

        # Continue the loop until n becomes zero.
        while n:
            # Apply bitwise XOR to the current result and n.
            # This is part of the process to find the minimum operations.
            result ^= n

            # Right shift the bits of n by one position.
            # This operation reduces the number being processed bit by bit.
            n >>= 1

        # Return the computed result.
        return result
```

Java Solution

```java
class Solution {

    // Function to calculate the minimum number of operations required
    // to transform the integer 'n' into zero by flipping exactly one bit in a binary representation
    public int minimumOneBitOperations(int n) {
        int result = 0; // Initialize the result variable to store the minimum operations

        // Iterate until n becomes zero
        while (n > 0) {
            // Use XOR to keep track of bit operations, exploiting the
            // property that going down to zero with one-bit operations can be seen as toggling bits
            result ^= n;

            // Right shift 'n' by 1 to divide it by 2, moving to the next bit
            n >>= 1;
        }

        // Return the calculated result
        return result;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the minimum number of operations required to
    // reduce the number 'n' to zero, where each operation consists of flipping
    // exactly one bit, and the operation must change the value of the integer.
    int minimumOneBitOperations(int n) {
        int result = 0; // Initialize result to 0, which will hold the final answer.

        // Loop until 'n' is reduced to zero. In each iteration, 'n' is right shifted
        // by one bit (n >>= 1), effectively dividing 'n' by 2 and discarding the least significant bit.
        while (n > 0) {
            // Perform bitwise XOR between the current result and 'n' and store the result back.
            // This progressively calculates the necessary bit-flips.
            result ^= n;

            // Right shift 'n' by one bit to examine and process the next bit in the next iteration.
            n >>= 1;
        }

        // Return the accumulated result, which represents the minimum one bit operations
        // necessary to reduce the binary number to zero.
        return result;
    }
};
```

Typescript Solution

```typescript
// Function to find the minimum number of operations to transform a given number into zero
// by toggling the least significant bit (rightmost bit) of the number.
function minimumOneBitOperations(n: number): number {
    // Initialize the answer to zero.
    let answer = 0;

    // Loop until the number n is greater than zero.
    while (n > 0) {
        // Apply bitwise XOR between the answer and the current number
        // to accumulate the effect of one-bit operations.
        answer ^= n;

        // Right shift the number by one to eliminate the least significant bit
        // and proceed with the next bit in the next iteration.
        n >>= 1;
    }

    // Return the total number of operations needed.
    return answer;
}
```

Time and Space Complexity

The given Python function `minimumOneBitOperations` consists of a while loop that runs as long as the input integer `n` is non-zero. On each iteration, the current value of `n` is XORed with `ans`, and then `n` is right-shifted by 1.

Time Complexity

The time complexity of the algorithm depends on the number of bits in the binary representation of the integer `n`. If `n` has a total of `b` bits, the loop iterates at most `b` times, since with each shift, one bit is removed from `n`. Thus, the time complexity is `O(b)`, which can also be considered `O(log n)` since the number of bits `b` is proportional to the logarithm of the number `n`.

Space Complexity

The space complexity of the algorithm is `O(1)`. This is because the algorithm uses a fixed amount of space: a single integer variable `ans`. The space required does not change with the size of the input `n`, indicating constant space complexity.