190. Reverse Bits

Easy | Bit Manipulation | Divide and Conquer

Problem Description

The goal of this problem is to reverse the bits of a given 32-bit unsigned integer. This means that every bit in the binary representation of the integer is inverted from end to start; for example, the bit at the far left end (the most significant bit) would swap places with the bit at the far right end (the least significant bit), and so on for each pair of bits in the number.

Intuition

To reverse the bits of an integer, the solution takes each bit individually from the least significant bit to the most significant bit, and places that bit into the reversed position in a new number, which would then become the most significant bit of the new number. This is repeatedly done for all 32 bits of the integer.

Starting with `res` as 0, which will hold the reversed bits, we loop 32 times since we're dealing with 32-bit numbers. In each iteration, we:

1. Check the least significant bit of `n` via `(n & 1)` which uses the bitwise AND operator to isolate the bit.
2. Shift this bit to its reversed position with `<< (31 - i)`, where `i` is the current iteration index, thus "moving" the bit to the correct position in the reversed number.
3. Use `|=` (bitwise OR assignment) to add this bit to the result `res`.
4. Right shift `n` by 1 with `n >>= 1` to move to the next bit for the next iteration.

After the loop finishes, all bits have been reversed in `res`, and we return `res` as the answer.

Solution Approach

The implementation of the solution involves a straightforward approach that mostly utilizes bitwise operations. Let's walk through the algorithm step by step:

1. Initialize an integer `res` to `0`. This variable will hold the reversed bits of the input.
2. Loop through a range of 32, because we are dealing with a 32-bit unsigned integer. For each index `i` in this range:

   a. Isolate the least significant bit (LSB) of `n` by using the bitwise AND operation `(n & 1)`. If the LSB of `n` is 1, the result of the operation will be 1, otherwise, it will be 0.

   b. Shift the isolated bit to its reversed position. The reversed position is calculated by subtracting the loop index `i` from 31 `(31 - i)`, as the most significant bit should go to the least significant position and vice versa. The operator `<<` is used for bitwise left shift.

   c. Use the bitwise OR assignment `|=` to add the shifted bit to `res`. This step effectively "builds" the reversed number by adding the bits from `n` in reversed order to `res`.

   d. Right shift the input number `n` by `1` using `n >>= 1` to proceed to the next bit for analysis in the subsequent iteration.
3. Once the loop is completed, all the bits from the original number `n` have been reversed and placed in `res`.
4. Return `res` as it now contains the reversed bits of the original input.

No extra data structures are needed in this approach. The solution relies solely on bitwise operations and integer manipulation. This problem is a typical example of bit manipulation technique, where understanding and applying bitwise operators is crucial to achieve the desired outcome.

Example Walkthrough

Let's consider the 8-bit binary representation of the number `5` to simplify the understanding of our solution. In binary, `5` is represented as `00000101`. To reverse the bits, we would want our output to be `10100000`.

Now, let's walk through the steps of the solution approach using this 8-bit number (note: the actual problem expects a 32-bit number, but this simplified example serves to illustrate the approach):

1. We initialize `res` to `0`, which is `00000000` in binary.
2. We then loop 8 times (in our simplified case), and for each `i` from `0` to `7` (for a 32-bit integer, `i` would range from `0` to `31`), we:

   a. Isolate the LSB of `5`:
   - When `i = 0`, `n & 1` is `1` (`00000101 & 00000001` equals `00000001`).

   b. We shift this isolated bit to its reversed position which is `7 - i` (for an 8-bit number):
   - When `i = 0`, shift `1` by `7` positions to left, we get `10000000`.

   c. We then use bitwise OR to add this to `res`:
   - `res` is now `10000000`.

   d. We right shift `n` by 1:
   - `n` becomes `00000010`.
3. Repeat these steps for the remaining iterations of the loop—each time `n` will be shifted to the right by 1, `res` will have `1` added in the correct reversed position if the isolated bit from `n` is `1`.

   For instance, when `i = 2`, the LSB is `0` (after shifting `n` to the right twice). There is nothing to add to `res` for this bit, as shifting a `0` and OR-ing it would not change `res`.

   Let's look at the case when `i = 5`, which corresponds to the third least significant `1` in our original number `5`:
   - `n & 1` is `1` (since `n` was right-shifted by 5 and now is `000000000`).
   - Shift `1` to the left by `2` (`7 - 5`), we get `00000100`.
   - `res` is now `10000100` after OR-ing.
4. After looping 8 times (32 times for an actual 32-bit integer), `res` would contain the reversed bits. In our example, `res` would be `10100000`.
5. Finally, we return `res`.

In the detailed steps of the 32-bit reversal, we would do this loop 32 times, and the resulting `res` would represent the reversed bits of the original 32-bit integer.

Solution Implementation

Python

```python
class Solution:
    def reverseBits(self, n: int) -> int:
        # Initialize result as 0. This will hold the reversed bits.
        result = 0

        # Loop over the 32 bits of the integer.
        for i in range(32):
            # Extract the least significant bit (LSB) using the bitwise AND operation (&)
            # and shift it to its reversed position. Then, use the bitwise OR operation (|)
            # to set the corresponding bit in the result.
            # (31 - i) gives the bit's reversed position.
            result |= (n & 1) << (31 - i)

            # Shift `n` to the right by 1 bit to process the next bit in the next iteration.
            n >>= 1

        # Return the reversed bit pattern as an integer.
        return result
```

Java

```java
public class Solution {

    /**
     * Reverses the bits of a given integer treating it as an unsigned value.
     *
     * @param number The integer to reverse bits for.
     * @return The reversed bits integer.
     */
    // The method name is kept as-is to maintain API contract
    public int reverseBits(int number) {
        // Initialize result to zero to start with a clean slate of bits
        int result = 0;

        // Loop over all the 32 bits of an integer
        // The loop continues while there are non-zero bits remaining
        for (int i = 0; i < 32 && number != 0; i++) {
            // Using bitwise OR and shift to add the least significant bit of 'number' to the result
            // (1) number & 1 isolates the least significant bit of 'number'
            // (2) << (31 - i) moves the bit to its reversed position
            // (3) |= assigns the bit to the correct position in result
            result |= ((number & 1) << (31 - i));

            // Unsigned right shift the 'number' by one to process the next bit in the next iteration
            number >>>= 1;
        }

        // Return the reversed bits integer
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t reversedNumber = 0; // Initialize the result to represent the reversed number.

        // Loop through all 32 bits of the input number.
        for (int bitPosition = 0; bitPosition < 32; ++bitPosition) {
            // Isolate the least-significant bit (rightmost bit) of 'n' and shift it to the correct position
            // in 'reversedNumber' (which starts from the leftmost bit and goes rightwards with each iteration).
            reversedNumber |= (n & 1) << (31 - bitPosition);
            // Shift input number 'n' right by one bit to process the next bit in the next iteration.
            n >>= 1;
        }

        // Return the reversed number after all 32 bits have been processed.
        return reversedNumber;
    }
};
```

TypeScript

```typescript
/**
 * Reverses bits of a given 32-bits unsigned integer.
 *
 * @param n - A positive integer representing the 32-bits unsigned integer to be reversed.
 * @returns A positive integer representing the reversed bits of the input.
 */
function reverseBits(n: number): number {
    let result: number = 0;

    // Loop through all 32 bits of the integer
    for (let i = 0; i < 32 && n > 0; ++i) {
        // Extract the least significant bit of 'n' and shift it to the correct position,
        // then OR it with the result to put it in its reversed position.
        result |= (n & 1) << (31 - i);

        // Logical shift the bits of 'n' right by 1, to process the next bit in the next iteration.
        n >>>= 1;
    }

    // The >>> 0 ensures the result is an unsigned 32-bit integer.
    return result >>> 0;
}
```


```python
class Solution:
    def reverseBits(self, n: int) -> int:
        # Initialize result as 0. This will hold the reversed bits.
        result = 0

        # Loop over the 32 bits of the integer.
        for i in range(32):
            # Extract the least significant bit (LSB) using the bitwise AND operation (&)
            # and shift it to its reversed position. Then, use the bitwise OR operation (|)
            # to set the corresponding bit in the result.
            # (31 - i) gives the bit's reversed position.
            result |= (n & 1) << (31 - i)

            # Shift `n` to the right by 1 bit to process the next bit in the next iteration.
            n >>= 1

        # Return the reversed bit pattern as an integer.
        return result
```


Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(1)`. This is because the loop runs a fixed number of times (32 iterations), one for each bit in a 32-bit integer. Because this number does not depend on the size of the input, but rather on the fixed size of an integer, the time complexity is constant.

Space Complexity

The space complexity of the code is also `O(1)`. The function uses a fixed amount of space: one variable to accumulate the result (`res`) and a single integer (`n`) whose bits are being reversed. No additional space that grows with the input size is used, so the space complexity is constant.