Problem: Reverse Bits

Objective:

- Given a 32-bit unsigned integer n, reverse its binary representation and return the resulting integer.

Key Points:

- You are tasked with reversing the binary bits of a 32-bit unsigned integer.
- The input number is always a 32-bit integer (meaning it has exactly 32 binary digits).
- The function should return a new integer whose binary representation is the reverse of the input integer's binary representation.

Input:

- **n:** A 32-bit unsigned integer (in decimal) with binary representation.
- **Example:** 00000010100101000001111010011100 (binary for the decimal number 43261596).

Output:

- **result:** The decimal value corresponding to the reversed 32-bit binary string of the input.
- **Example:** After reversing the input bits, the output should be the decimal value of the reversed binary string.

Example 1:

- **Input:** n = 00000010100101000001111010011100
- **Binary Representation:** 43261596 in decimal is represented as 00000010100101000001111010011100 in binary.
- **Output:** 964176192
- **Explanation:** Reversing the bits of 00000010100101000001111010011100 yields 00111001011110000010100101000000, which is 964176192 in decimal.

Example 2:

- **Input:** n = 11111111111111111111111111111101
- **Binary Representation:** 4294967293 in decimal is represented as 11111111111111111111111111111101 in binary.
- **Output:** 3221225471
- **Explanation:** Reversing the bits of 11111111111111111111111111111101 yields 10111111111111111111111111111111, which is 3221225471 in decimal.

Constraints:

- The input must be a 32-bit unsigned integer.
- The function should handle edge cases where the input consists entirely of 1's or 0's.

Binary Representation:

In this problem, understanding the binary representation of numbers is crucial:

- A 32-bit unsigned integer means the number is stored using 32 bits (binary digits), which can represent values from 0 to 4294967295.
- Each bit represents a power of 2, from the least significant bit (rightmost) representing (2^0) to the most significant bit (leftmost) representing ($2^{\{31\}}$).

Strategy to Solve:

To reverse the bits of a 32-bit unsigned integer, follow these steps:

1. **Initialization:** Start with an integer result = 0, which will store the reversed bits.
2. **Process the Bits:** Loop over each of the 32 bits of n. For each iteration:
    - Shift the result to the left by 1 to make room for the next bit.
    - Extract the least significant bit (rightmost bit) of n and add it to result.
    - Shift the input n to the right by 1 to process the next bit in the next iteration.
3. **Return the Result:** After 32 iterations, the result will contain the reversed binary number, which can be returned as the final answer.

Handling Signed vs. Unsigned Integers:

- In many languages like Java, there is no specific type for unsigned integers, and integers are represented in 2's complement format. However, this should not affect your implementation, as the internal binary representation remains the same regardless of whether the integer is signed or unsigned.
- The constraints ensure that the input and output should be treated as unsigned 32-bit integers, and you do not need to worry about handling negative numbers in Python.

Optimizations:

- If the function is called many times, you can consider optimizing the process using a lookup table for 8-bit chunks. Since there are only 256 possible 8-bit values, you can precompute the reversed values of all possible 8-bit integers. For any 32-bit input, you can break it into four 8-bit chunks, look up their reversed values, and combine them to get the final reversed result.

Edge Cases:

1. **All Zeros Input:** If the input is all zeros (00000000000000000000000000000000), the output should also be all zeros.
2. **All Ones Input:** If the input is all ones (11111111111111111111111111111111), the output will be the same, as reversing does not change the arrangement of all ones.
3. **Single One Bit:** If the input has a single one bit at the least significant position (e.g., 00000000000000000000000000000001), the output will have that bit at the most significant position (10000000000000000000000000000000).

Time Complexity:

- The time complexity is O(1) since the algorithm performs a constant number of operations (32 iterations) for every input, regardless of its value.

Space Complexity:

- The space complexity is O(1) because the algorithm uses a fixed amount of space (a few integer variables) regardless of the input size.

Follow-Up:

- If this function is called many times, an optimization strategy could involve precomputing the reversed values of smaller chunks of bits (e.g., 8 bits) and combining those precomputed results to form the reversed 32-bit integer. This would reduce the computation time at the cost of additional memory.

This approach would be especially beneficial in scenarios where many 32-bit integers need to be reversed frequently.