1680. Concatenation of Consecutive Binary Numbers

Medium Bit Manipulation Math Simulation Leetcode Link

Problem Description

This problem asks us to construct a large binary number that represents the concatenation of the binary representations of all the integers from `1` to `n`, and then return the decimal value of this large binary number modulo `10^9 + 7`.

Imagine writing down all the numbers from `1` to `n` in decimal, then converting each of those to binary and stringing all the binary numbers together into one long binary sequence. This problem is essentially about finding the decimal value of that long sequence, but since the number could be very large, we take its modulus with `10^9 + 7` to keep the result within manageable bounds.

For instance, if `n = 3`, the binary representations are `1` for `1`, `10` for `2`, and `11` for `3`. Concatenating these binaries we would get `11011`. The decimal equivalent of `11011` is `27`.

Intuition

The solution relies on understanding how binary numbers work and realizing that to append a binary number `b` to another binary number `a`, you can shift `a` to the left by the number of bits in `b` and then use bitwise OR to add `b` into `a`.

Here's our thought process to arrive at the solution:

- We want to add each binary number from `1` to `n` to our concatenation. To do this, we iterate over this range.
- Before adding a new number, we must make space for it by shifting the bits of the current concatenated binary number to the left. The number of shifts equals the number of bits in the next number to add.
- We can find out when the number of bits increases by checking if the number is a power of two because that's when an additional bit is required (`2`, `4`, `8`, etc.). A neat trick to check if a number is a power of two is to use the expression `(i & (i - 1)) == 0`.
- Once we know how much to shift, we shift the current answer to the left by that amount and use bitwise OR to append the new number.
- We want our final result to be within the bounds of `10^9 + 7`, so we take the modulus after each concatenation to prevent overflow.
- Keep appending the numbers in the described fashion, taking the modulus as needed, until you reach `n`.
- The final result after appending all numbers and taking the modulus is our answer.

Using this approach ensures that we get the result without actually constructing an impossibly large binary number, which would be impractical and inefficient.

Solution Approach

The implementation of the solution follows a bitwise manipulation approach. Here's a step-by-step walkthrough of the algorithm referring to the given Python code:

1. Define a `mod` variable representing the modulo value `10**9 + 7`. This ensures all operations are bound within this range to avoid integer overflow.

2. Initialize `ans` and `shift` to `0`. `ans` will hold the final result, and `shift` represents the number of positions we need to shift `ans` to the left to make space for the next binary number.

3. Loop through each integer `i` from `1` to `n` (inclusive), performing the following steps:

    - Check if `i` is a power of two by verifying whether `(i & (i - 1)) == 0`. This works because a power of two in binary representation has a single `1` followed by zeroes, and subtracting `1` from it flips all the bits up to that `1` (e.g., `1000` becomes `0111`). The bitwise AND of `i` and `i - 1` will be zero if `i` is a power of two.

    - If `i` is a power of two, increment the `shift` variable by one to account for the additional bit in the binary representation of `i`.

    - Concatenate `i` to `ans` by left-shifting `ans` by `shift` positions (using the `<<` operator) and then performing bitwise OR with `i` (using the `|` operator). This concatenates the binary representation of `i` to the right of `ans`.

    - Apply modulus operation to `ans` after the concatenation to ensure the result never exceeds `10**9 + 7`.

4. After the loop, `ans` holds the decimal value of the concatenated binary string modulo `10**9 + 7`, and we return `ans`.

The Python code uses no additional data structures, taking advantage of bitwise operations for efficient manipulation of the numbers. The iterative method keeps memory usage low, as we only work with integers and update our answer bit by bit. There's a direct correlation between each number and its binary representation, making this approach highly suitable for this problem.

Example Walkthrough

Let's use the example where `n = 5` to illustrate the solution approach. Our goal is to calculate the decimal value of the binary number formed by concatenating the binary representations of the numbers from `1` to `n`.

Here's how the algorithm would proceed, step by step:

1. Initialize constants and variables:
    - `mod = 10**9 + 7`
    - `ans = 0` (to hold the final result)
    - `shift = 0` (to indicate the number of bit positions `ans` must be shifted)
2. Start looping from `i = 1` to `i = 5`:
    - When `i = 1`:
        - Binary representation is `1`.
        - `shift` does not increase because `1 & (1 - 1)` is not equal to `0`.
        - `ans = (ans << shift) | i` becomes `(0 << 0) | 1` which is `1`.
        - Apply modulus: `ans = 1`.
    - When `i = 2`:
        - Binary representation is `10`.
        - Since `2 & (2 - 1)` equals `0`, `shift` increments by `1` (now `shift = 1`).
        - `ans = (ans << shift) | i` becomes `(1 << 1) | 2` which is `4 | 2` or `110` in binary, which is `6` in decimal.
        - Apply modulus: `ans = 6`.
    - When `i = 3`:
        - Binary representation is `11`.
        - `shift` does not increase because `3 & (3 - 1)` is not equal to `0`.
        - `ans = (ans << shift) | i` becomes `(6 << 1) | 3` which is `12 | 3` or `1111` in binary, which is `15` in decimal.
        - Apply modulus: `ans = 15`.
    - When `i = 4`:
        - Binary representation is `100`.
        - Since `4 & (4 - 1)` equals `0`, `shift` increments by `1` (now `shift = 2`).
        - `ans = (ans << shift) | i` becomes `(15 << 2) | 4` which is `60 | 4` or `111100` in binary, which is `60` in decimal.
        - Apply modulus: `ans = 60`.
    - When `i = 5`:
        - Binary representation is `101`.
        - `shift` does not increase because `5 & (5 - 1)` is not equal to `0`.
        - `ans = (ans << shift) | i` becomes `(60 << 2) | 5` which is `240 | 5` or `11110101` in binary, which is `245` in decimal.
        - Apply modulus: `ans = 245`.
3. After the loop, we have `ans = 245`, which is the decimal value of the concatenated binary number `11110101` (formed by concatenating `1`, `10`, `11`, `100`, and `101`). As the final result is smaller than `10**9 + 7`, it is also the return value.

By following these steps, we've converted the sequence of binary numbers to their concatenated decimal equivalent using bitwise manipulation and kept the value below `10**9 + 7` as required.

Python Solution

```python
class Solution:
    def concatenatedBinary(self, n: int) -> int:
        # Define the modulus to prevent integer overflow
        modulus = 10**9 + 7

        # Initialize the result and the number of bits to shift
        result = shift = 0

        # Iterate through each number from 1 to n
        for num in range(1, n + 1):
            # Check if the number is a power of 2 (has only one '1' in binary)
            # If so, increase the shift counter as the binary length increases by 1
            if (num & (num - 1)) == 0:
                shift += 1

            # Left shift the result by the number of bits required, then OR it with
            # the current number, and take modulo to maintain the result within limits
            result = (result << shift | num) % modulus

        # Return the concatenated binary number modulo 10^9 + 7
        return result

```

Java Solution

```java
class Solution {
    public int concatenatedBinary(int n) {
        // Constant for the modulus value to ensure the result stays within integer bounds
        final int MOD = 1_000_000_007;

        // Initialize answer as a long to avoid integer overflow
        long answer = 0;

        // Initialize the number of bits required for binary shift
        int shiftCount = 0;

        // Iterate over each number from 1 to n
        for (int i = 1; i <= n; ++i) {
            // Check if the current number is a power of two by using bitwise AND
            // A number is a power of two if it has a single 1-bit and all other bits are 0
            if ((i & (i - 1)) == 0) {
                // If the current number is a power of two, increment the shift count
                ++shiftCount;
            }

            // Concatenate the current number in binary to the answer
            // Shift the current answer to the left by shiftCount bits
            // OR with the current number to append it
            // And take the result modulo MOD to keep it within bounds
            answer = ((answer << shiftCount) | i) % MOD;
        }

        // Cast the answer back to an integer before returning
        return (int) answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to concatenate the binary representation of numbers from 1 to n
    int concatenatedBinary(int n) {
        const int MOD = 1e9 + 7; // Define the modulo to prevent integer overflow
        long result = 0; // Initialize the result variable to store the concatenated binary number
        int bitShift = 0; // Initialize bit shift count to determine how many positions to shift

        // Loop through all numbers from 1 to n and build the concatenated binary representation
        for (int i = 1; i <= n; ++i) {
            // Check if 'i' is a power of 2 by using bitwise AND on 'i' and 'i-1'
            // If i is a power of 2, the number of bits needed increases by 1
            if ((i & (i - 1)) == 0) {
                ++bitShift;
            }
            // Left shift the current result by the current bitShift value to make space for the new number 'i'
            // OR with 'i' to append 'i' in its binary form to the result
            // Apply modulo operation to keep the result within the MOD range
            result = ((result << bitShift) | i) % MOD;
        }
        return result; // Return the final result after processing all numbers from 1 to n
    }
};

```

Typescript Solution

```typescript
function concatenatedBinary(n: number): number {
    // Define the modulo constant for the final answer to avoid integer overflow.
    const MOD = BigInt(10 ** 9 + 7);

    // Initialize 'answer' to store our running binary concatenation result.
    let answer = 0n;

    // Initialize 'shiftCount' to keep track of the number of binary digits to shift.
    let shiftCount = 0n;

    // Iterate from 1 to n in BigInt to handle binary operations.
    for (let i = 1n; i <= BigInt(n); ++i) {
        // Check if 'i' is a power of 2, if it is, increment 'shiftCount'.
        // This is done by checking if 'i' is a power of 2 by using bitwise AND on (i & (i - 1n)).
        // If 'i' is a power of 2, (i & (i - 1n)) will be 0.
        if ((i & (i - 1n)) == 0n) {
            ++shiftCount;
        }

        // Perform the concatenation in binary by shifting 'answer' to the left by 'shiftCount' bits,
        // then OR with 'i' to append 'i' to the binary representation.
        // Apply modulo operation to keep the number within the MOD limit.
        answer = ((answer << shiftCount) | i) % MOD;
    }

    // Return the final answer as a number (the answer is currently a BigInt).
    return Number(answer);
}

```

Time and Space Complexity

Time Complexity

The main operation of the code involves a for loop that iterates `n` times, where `n` is the input integer. Inside the loop, the code checks whether an integer is a power of two, which is performed in O(1) by using bitwise AND operation `i & (i - 1)`. Then, the solution shifts the `ans` variable to the left by `shift` positions and performs a bitwise OR with the current number `i`, also in constant time. Finally, it takes the modulus of the result, again an O(1) operation. Therefore, overall, the time complexity of this loop is O(n).

Space Complexity

The space complexity is O(1) because the algorithm uses a fixed number of integer variables (`mod`, `ans`, `shift`, and `i`) that do not depend on the size of the input number `n`.