1404. Number of Steps to Reduce a Number in Binary Representation to One

Medium Bit Manipulation String

Problem Description

The problem presents a scenario where you are given the binary representation of an integer as a string `s`. Your task is to determine how many steps it would take to reduce this number to `1` by following two rules:

1. If the current number is even, divide it by `2`.
2. If the current number is odd, add `1` to it.

The constraint mentioned in the problem is that it's guaranteed that following these rules will always lead you to reach the number `1`, regardless of the starting number.

Intuition

To solve this problem, we need to understand how binary numbers work and how we can apply the given rules to get to '1'. The last digit of a binary number determines if it's even (0) or odd (1). Therefore:

1. If the last digit is '0', we know the number is even, and we need to divide the number by '2'. In binary terms, dividing by '2' is equivalent to simply removing the last digit (which is a '0' in this case).
2. If the last digit is '1', then the number is odd, and we need to add '1' to it. Adding '1' to an odd binary number will flip the last digit to '0' and may cause a carry to propagate towards the most significant bit.

The key challenge in this problem is handling the carry when incrementing an odd binary number. We must traverse the binary string from least significant bit (rightmost) to the most significant bit (leftmost), flipping the bits until we either reach a '0', which does not require a carry, or we reach the beginning of the string, at which point an additional final carry would complete the task.

We iterate over the binary string in reverse, keeping track of carries. For each character:

- If there is no carry and the character is '0' (even), no action is needed, just an additional step is counted.
- If there is no carry and the character is '1' (odd), we must flip it to '0' and count a step for the flip and another step for adding '1' to make it even (which enables the division by '2').
- If there is a carry and the character is '0', we flip it to '1', eliminate the carry, count a step for the flip, and move on.
- If there is a carry and the character is '1', we keep it as '1', maintain the carry, but still count a step.

Solution Approach

The solution provided in Python follows these steps:

1. Initialize a variable `carry` to `False`. This will act as a flag to determine whether we need to carry a '1' when we flip a '0' to a '1' or vice versa.
2. Initialize a counter `ans` to `0`, which will count the number of steps taken to get to '1'.
3. Loop through the binary string's characters in reverse, except for the first character, using `for c in s[:0:-1]:`. This is because the first bit (the leftmost) does not directly affect the counting, as the problem assumes that the number will eventually be reduced to '1'.
4. For each iteration, check if there is a carry. If so, and the current bit is a '0', flip it to a '1' and reset the carry to `False`. If the current bit is a '1', flip it to a '0' and keep the carry as `True`.
5. If the current bit is '1', increment the steps by one because '1' represents an odd number which needs to add '1' to become even. Then, set the carry to `True`.
6. Regardless of the conditions above, increment the steps by one for the division by '2' operation, except when we flip the last '1' to '0' and there is no more carry.
7. Finally, outside of the loop, if there is still a carry `if carry:` after all the flips, which means all bits were '1's, and by adding '1' we overflow into an additional bit (e.g., from '111' to '1000'), increment the steps by one more (`ans += 1`), since this represents an extra step needed to reach '1'.
8. Return the counter `ans` as the number of steps taken.

The algorithm makes use of a simple simulation of the manual process one might use to add binary numbers on paper while taking into account how addition and division by '2' affect binary digits. The solution does not use any additional data structures and takes advantage of the property of binary numbers and arithmetic to achieve the intended result efficiently.

Example Walkthrough

Let's take the binary representation of the integer `13` as an example, which is `1101` in binary.

1. We initialize the `carry` as `False` and the step counter `ans` as `0`.
2. We examine the binary string from right to left, starting with the second-to-last digit because the last one determines if we need to add one to make it even (if it's a '1') or if we can divide immediately (if it's a '0').

Starting from the second-to-last bit (from the right), our steps will be:

- The next-to-last character is '0'. Since there is no carry and it's even, we can divide by '2'. This gives us one step (`ans += 1`), and `carry` remains `False`.

3. The next character (moving right to left) is a '1'.

- Since it's '1', it's odd; we add '1' to it, flip it to '0', and add two steps (`ans += 2` – one for adding '1' and one for the division by '2').
- Now, because we added '1' to a '1', it means we have a carry over.

4. We move to the first character, which is also a '1'.

- There is still a carry from before. This '1' becomes a '0' due to the carry, but we still have a carry because adding '1' to '1' gives '10' in binary.
- We flip the '1' to '0', count the step for the flip (`ans += 1`), but do not add another step for division because that will happen next.

5. After completing the string traversal, we check for carry. There is still a carry which means we had a string of all '1's before, so we need an extra step to account for the overflow (e.g., from '111' to '1000').

- We increment `ans` by one more step (`ans += 1`).

As a summary of the steps:

- Start: `1101`
- Divide by 2 (remove last '1'): `110`, steps = 1
- Add 1 (to the odd number): `111`, steps = 3
- Divide by 2 (remove last '1'): `11`, steps = 4
- Add 1 (to handle carry from '111'): `1000`, steps = 5
- Divide by 2 three times (remove ending '0's): `1`, steps = 8

Thus, the number of steps taken to reduce the binary string `1101` to `1` is `8`.

Solution Implementation

Python

```python
class Solution:
    def numSteps(self, binary_str: str) -> int:
        # Initialize a boolean to track if there is a carry in binary addition
        has_carry = False

        # Initialize a counter to track the number of steps
        steps_count = 0

        # Iterate over the string in reverse order, excluding the most significant bit
        for bit in binary_str[:0:-1]:
            # If there is a carry, add it to the current bit
            if has_carry:
                # If the current bit is '0', it becomes '1' after adding carry
                if bit == '0':
                    has_carry = False  # Carry is consumed
                else:
                    # If the current bit is '1', it becomes '0' after adding carry,
                    # and we keep the carry for the next bit
                    bit = '0'

            # If the current bit is '1' and there is no carry, or if we just had a carry,
            # increment the step count since we would need an additional step to reduce it
            if bit == '1':
                steps_count += 1  # One step to make it '0'
                has_carry = True  # A carry is generated for the next bit

            # One step is always needed for each bit, either to add the carry or to divide by 2
            steps_count += 1

        # If there is a carry after processing all bits, add an extra step to handle it
        if has_carry:
            steps_count += 1

        # Return the total number of steps needed to reduce the binary string to '1'
        return steps_count
```

Java

```java
class Solution {
    public int numSteps(String s) {
        boolean carry = false; // Initialize carry to keep track of addition carry
        int steps = 0; // Initialize steps to count the number of operations

        // Loop backwards through the binary string (ignoring the MSB at index 0 initially)
        for (int i = s.length() - 1; i > 0; --i) {
            char c = s.charAt(i); // Get the current bit

            // If there is a carry from the previous operation
            if (carry) {
                // Determine the actions based on the current bit
                if (c == '0') {
                    // If it's 0, carry turns it to 1 without generating a new carry
                    carry = false;
                } else {
                    // If it's 1, it remains 1 and we also have a carry
                    c = '0';
                }
            }

            // If current bit is 1, we need to flip it which results in a carry
            if (c == '1') {
                carry = true;
                steps++; // Increment the steps for flipping the bit
            }
            steps++; // Increment the steps for the potential addition (or bit move)
        }

        // If a carry is left after processing all bits, an additional step is needed
        if (carry) {
            steps++;
        }

        return steps; // Return the total number of steps to convert the binary number to 1
    }
}
```

C++

```cpp
class Solution {
public:
    int numSteps(string s) {
        // Initialize the number of steps to 0
        int steps = 0;
        // The carry variable to handle the addition carry over
        bool carry = false;

        // Traverse the binary string from the least significant bit to the most
        // Ignoring the most significant bit as we stop when we reach the beginning of the string
        for (int i = s.size() - 1; i > 0; --i) {
            // Get the current bit
            char bit = s[i];

            // If there's a carry from the previous operation
            if (carry) {
                // Flip the current bit due to the carry
                if (bit == '0') {
                    bit = '1';
                    // The carry has been used, reset it to false
                    carry = false;
                } else // If the bit is '1', turning into '0' keeps carry true
                    bit = '0';
            }

            // If the bit is '1', we will need a step to make it '0' and create a carry
            if (bit == '1') {
                // Increment steps for the flip to '0'
                ++steps;
                // Set the carry for the next iteration
                carry = true;
            }
            // Regardless of the bit, we do a right shift operation
            ++steps;
        }

        // If there's a carry at the end, we need to add one more step to add it to the MSB
        if (carry) ++steps;

        // Return the total number of steps
        return steps;
    }
};
```

TypeScript

```typescript
function numSteps(s: string): number {
  // Initialize the number of steps to 0
  let steps: number = 0;
  // The carry variable to handle the addition carry over
  let carry: boolean = false;

  // Traverse the binary string from the least significant bit to the most
  // Ignoring the most significant bit as we stop when we reach the start of the string
  for (let i: number = s.length - 1; i > 0; --i) {
    // Get the current bit
    let bit: string = s[i];

    // If there's a carry from the previous operation
    if (carry) {
      // Flip the current bit due to the carry
      if (bit === '0') {
        bit = '1';
        // The carry has been absorbed, so we reset it to false
        carry = false;
      } else {
        // If the bit is '1', it will turn into '0' and we keep the carry as true
        bit = '0';
      }
    }

    // If the bit is '1', we will need a step to make it '0' and generate a carry
    if (bit === '1') {
      // Increment steps for the flip to '0'
      steps++;
      // Set the carry for the next iteration
      carry = true;
    }
    // Regardless of the bit, we perform a right shift operation
    steps++;
  }

  // If there's a carry at the end, we need one more step to add it to the MSB
  if (carry) steps++;

  // Return the total number of steps
  return steps;
}
```

```python
class Solution:
    def numSteps(self, binary_str: str) -> int:
        # Initialize a boolean to track if there is a carry in binary addition
        has_carry = False

        # Initialize a counter to track the number of steps
        steps_count = 0

        # Iterate over the string in reverse order, excluding the most significant bit
        for bit in binary_str[:0:-1]:
            # If there is a carry, add it to the current bit
            if has_carry:
                # If the current bit is '0', it becomes '1' after adding carry
                if bit == '0':
                    has_carry = False  # Carry is consumed
                else:
                    # If the current bit is '1', it becomes '0' after adding carry,
                    # and we keep the carry for the next bit
                    bit = '0'

            # If the current bit is '1' and there is no carry, or if we just had a carry,
            # increment the step count since we would need an additional step to reduce it
            if bit == '1':
                steps_count += 1  # One step to make it '0'
                has_carry = True  # A carry is generated for the next bit

            # One step is always needed for each bit, either to add the carry or to divide by 2
            steps_count += 1

        # If there is a carry after processing all bits, add an extra step to handle it
        if has_carry:
            steps_count += 1

        # Return the total number of steps needed to reduce the binary string to '1'
        return steps_count
```

Time and Space Complexity

The provided code implements a function to transform a binary number given as a string into the number 1, by applying certain operations and counts the number of steps taken.

Time Complexity:

To analyze the time complexity, let's look at the loop in the code. It iterates over the string `s` from the end (excluding the first character) towards the beginning. For each character `c`, it performs a constant number of operations, checking the value of `c`, possibly incrementing `ans`, and modifying `carry`. The loop runs for `n - 1` iterations, where `n` is the length of the input string.

Since each iteration takes a constant time, the time complexity of the loop is `O(n)`. The final `if` block is also executed at most once and takes constant time, so the overall time complexity of the code is `O(n)`.

Space Complexity:

The space complexity is determined by the amount of extra space used by the algorithm besides the input itself. Here, we have a few variables (`carry`, `ans`, and `c`) which only require a constant amount of space.

Hence, the space complexity of the provided code is `O(1)`, which means it uses a constant amount of additional space regardless of the size of the input.