67. Add Binary

Easy Bit Manipulation Math String Simulation

Problem Description

Given two strings `a` and `b` that represent binary numbers (0 or 1), the goal is to find the sum of these two binary strings and return the result also as a binary string. Binary addition is similar to decimal addition, but it carries over a `1` when the sum of two bits is `2` (since `1 + 1` in binary is `10`).

In a simpler form, you are required to add two binary numbers without using built-in binary to decimal conversion functions, and then represent the result as a binary number in string format.

Intuition

The intuition for solving this problem aligns with how we generally perform addition by hand in decimal numbers, but instead we apply binary rules. We begin by adding the least significant bits (rightmost bits) of the input binary strings, `a` and `b`. We work our way to the most significant bit (left side) considering any carry that arises from the addition of two bits.

Each bit can only be `0` or `1`. If the sum of two bits plus any carry from the previous bit is `2` or `3`, a carry of `1` is passed to the next left bit. The binary sum (ignoring the carry) for that position will be `0` if the sum is `2` (since `10` in binary represents `2`), or `1` if the sum is `3` (since `11` in binary represents `3`).

For positions where one of the strings may have run out of bits (because one string can be shorter than the other), we treat the missing bit as `0`. We also need to consider the possibility of a carry remaining after we've finished processing both strings.

The process can be summarized as follows:

1. Initialize an answer array to build the result string.
2. Use two pointers starting from the end of both strings and a variable to hold the carry value (initially `0`).
3. Iterate while at least one pointer is valid or there is a carry remaining.
4. Compute the sum for the current position by adding bits `a[i]` and `b[j]` along with the carry.
5. Use the `divmod` function to obtain the result for the current position and the new carry.
6. Append the result to the answer array.
7. Once the iteration is complete, reverse the answer array to represent the binary sum in the proper order.
8. Join the answer array elements into a string and return it.

The solution uses `divmod` for a compact and readable way to handle both the carry and the current bit representation at the same time.

Solution Approach

The implementation employs several important concepts for working with binary numbers:

- **Pointer Iteration:** We use two pointers `i` and `j` that iterate through both strings `a` and `b`, respectively, starting from the end (least significant bit) moving towards the start (most significant bit). This ensures that addition occurs like manual hand addition, from right to left.
- **Carry Handling:** We initialize a variable `carry` to `0`, which will be used to handle the carry-over that occurs when the sum of bits exceeds `1`.
- **Loop Control:** The loop continues as long as there is a bit to process (either `i` or `j` is greater than or equal to `0`) or there is a carry to apply. The `or` logic ensures we process all bits and handle the final carry.
- **Bit Addition and Carry Update:** Inside the loop, the `carry` is updated by adding the values of the current bits `a[i]` and `b[j]`. We use a conditional expression (`0 if i < 0 else int(a[i])`) and similarly for `b[j]` to account for cases where one binary string is shorter than the other; in such cases, the nonexistent bit is considered as `0`.
- **Result and Carry Calculation:** The `carry, v = divmod(carry, 2)` line cleverly updates both the carry for the next iteration and the result for the current position. The `divmod` function is a built-in Python function that takes two numbers and returns a tuple containing the quotient and remainder. Since we are working with binary, dividing by `2` covers both scenarios: if `carry` is `3` or `2`, the quotient would be the next carry, and the remainder would be the bit to append for the current position.
- **Building the Result:** The bit for the current position is appended to the `ans` list. After the loop, we reverse the `ans` list to obtain the actual binary representation, because we've been adding bits in reverse order, starting from the least significant bit.
- **Result Conversion:** Finally, we convert the list of bits into a string using the `join()` method and return it as the final binary sum.

Here's how the algorithm and its components come into play in the code:

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        ans = []  # List to store the binary result bits
        i, j, carry = len(a) - 1, len(b) - 1, 0  # Initialize pointers and carry

        # Loop while there are bits to process or a carry
        while i >= 0 or j >= 0 or carry:
            # Perform bit addition for current position and update carry
            carry += (0 if i < 0 else int(a[i])) + (0 if j < 0 else int(b[j]))
            # divmod returns a tuple (carry, value) and the new carry
            carry, v = divmod(carry, 2)
            # Append the result bit to the ans list
            ans.append(str(v))
            # Move to the previous bits
            i, j = i - 1, j - 1

        # Reverse the ans list to get the correct bit order and join to form a binary string
        return "".join(ans[::-1])
```

This clean and efficient approach leverages the mechanics of binary addition and takes full advantage of Python's powerful features for readability and concise code.

Example Walkthrough

Let's use a simple example with two binary strings `a = "101"` and `b = "110"` to illustrate the solution approach step by step:

1. **Initialization**
   We initialize `i` to `2` (index of the last character of `a`), `j` to `2` (index of the last character of `b`), and `carry` to `0`. Our `ans` array starts empty.
2. **Iteration 1**
   We begin from the end of both strings:
   - `i` (index `2` of `a`) corresponds to `1`.
   - `j` (index `2` of `b`) corresponds to `0`.

   Adding these with `carry = 0`, we get `1 + 0 + 0 = 1`.
   `carry, v = divmod(1, 2)` makes `carry = 0` and `v = 1`.
   `ans` array becomes `['1']`.
   Decrement `i` and `j` by `1`. Now `i = 1, j = 1`.
3. **Iteration 2**
   - `i` now points to `0` (`a[1]`)
   - `j` now points to `1` (`b[1]`)

   Adding these with `carry = 0`, we get `0 + 1 + 0 = 1`.
   `carry, v = divmod(1, 2)` makes `carry = 0` and `v = 1`.
   `ans` array becomes `['1', '1']`.
   Decrement `i` and `j` by `1`. Now `i = 0, j = 0`.
4. **Iteration 3**
   - `i` now points to `1` (`a[0]`)
   - `j` now points to `1` (`b[0]`)

   Adding these with `carry = 0`, we get `1 + 1 + 0 = 2`.
   `carry, v = divmod(2, 2)` makes `carry = 1` and `v = 0`.
   `ans` array becomes `['1', '1', '0']`.
   Decrement `i` and `j` by `1`. Now `i = -1, j = -1`.
5. **Iteration 4**
   Both pointers `i` and `j` are less than `0`, but we still have `carry = 1` to process.
   Since there are no more bits to add (`a[-1]` and `b[-1]` don't exist), we only add the carry.
   `carry, v = divmod(1, 2)` makes `carry = 0` and `v = 1`.
   `ans` array becomes `['1', '1', '0', '1']`.
   Since `carry` is now `0`, we will exit the loop after this step.
6. **Finalizing the Result**
   `ans` is currently `['1', '1', '0', '1']`, which is the reverse of our desired binary sum.
   We reverse `ans` to get `['1', '0', '1', '1']`.
   Joining `ans` we get the binary string `"1011"` which is the sum of `a` and `b`.

Thus, calling `Solution().addBinary("101", "110")` would return `"1011"`.

Solution Implementation

Python

```
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        # Initialize result list, pointers for a and b, and carry variable.
        result = []
        pointer_a, pointer_b, carry = len(a) - 1, len(b) - 1, 0

        # Loop until both pointers are out of range and there is no carry left.
        while pointer_a >= 0 or pointer_b >= 0 or carry:
            # Add carry to the sum of the current digits from a and b.
            # Use ternary operator to handle index out of range scenarios.
            carry += (0 if pointer_a < 0 else int(a[pointer_a])) + \
                     (0 if pointer_b < 0 else int(b[pointer_b]))

            # Perform division by 2 to get carry and the value to store.
            carry, value = divmod(carry, 2)

            # Append the value to the result list.
            result.append(str(value))

            # Move the pointers one step to the left.
            pointer_a, pointer_b = pointer_a - 1, pointer_b - 1

        # Since the append operation adds the least significant bits first,
        # the result string should be reversed to represent the correct binary number.
        return "".join(result[::-1])

# Example usage:
solution = Solution()
print(solution.addBinary("1010", "1011"))  # Output: "10101"
```

Java

```
public class Solution {
    public String addBinary(String a, String b) {
        // StringBuilder to store the result of the binary sum
        StringBuilder result = new StringBuilder();
        // Initialize pointers to the end of the strings from the end to the start
        int indexA = a.length() - 1;
        int indexB = b.length() - 1;
        // Carry will be used for the addition if the sum of two bits is greater than 1
        int carry = 0;
        // Loop until all characters are processed or there is no carry left
        while (indexA >= 0 || indexB >= 0 || carry > 0) {
            // If still within the bounds of string a, add the numeric value of the bit to carry
            if (indexA >= 0) {
                carry += a.charAt(indexA) - '0';
                indexA--; // Decrement index for string a
            }
            // If still within the bounds of string b, add the numeric value of the bit to carry
            if (indexB >= 0) {
                carry += b.charAt(indexB) - '0';
                indexB--; // Decrement index for string b
            }
            // Append the remainder of dividing carry by 2 (either 0 or 1) to the result
            result.append(carry % 2);
            // Carry is updated to the quotient of dividing carry by 2 (either 0 or 1)
            carry /= 2;
        }
        // Since the bits were added from right to left, the result needs to be reversed to match the correct order
        return result.reverse().toString();
    }
}
```

C++

```
#include <algorithm> // include algorithm for using the reverse function
#include <string>  // include string library to use the string class

class Solution {
public:
    // Function to add two binary strings and return the sum as a binary string
    string addBinary(string a, string b) {
        string result; // Resultant string to store the binary sum

        // Indices to iterate through strings a and b from the end
        int indexA = a.size() - 1;
        int indexB = b.size() - 1;

        // Iterate over the strings from the end while there is a digit or a carry
        for (int carry = 0; indexA >= 0 || indexB >= 0 || carry; --indexA, --indexB) {

            // Add carry and the corresponding bits from a and b. If index is negative, add 0
            carry += (indexA >= 0 ? a[indexA] - '0' : 0) + (indexB >= 0 ? b[indexB] - '0' : 0);

            // Append the least significant bit of the carry (either 0 or 1) to the result
            result.push_back(carry % 2 + '0');

            // Divide carry by 2 to get the carry for the next iteration
            carry /= 2;
        }

        // Since the bits were added from right to left, reverse the result to get the correct order
        reverse(result.begin(), result.end());

        // Return the resulting binary sum string
        return result;
    }
};
```

TypeScript

```
function addBinary(a: string, b: string): string {
    // Initialize indices for the last characters of strings `a` and `b`
    let indexA = a.length - 1;
    let indexB = b.length - 1;
    // Array to store the result in reverse order
    let result: number[] = [];
    let carry = 0; // This will hold the carry-over for binary addition

    // Loop until both strings are traversed or carry is non-zero
    while (indexA >= 0 || indexB >= 0 || carry > 0) {
        // If index is valid, add corresponding digit from `a` to carry
        if (indexA >= 0) {
            carry += a.charCodeAt(indexA) - '0'.charCodeAt(0);
            indexA--;
        }
        // If indexB is valid, add corresponding digit from `b` to carry
        if (indexB >= 0) {
            carry += b.charCodeAt(indexB) - '0'.charCodeAt(0);
            indexB--;
        }

        // The binary digit is carry % 2, add to result
        result.push(carry % 2);
        // Update carry for next iteration: divide by 2 and floor
        carry = Math.floor(carry / 2);
    }

    // Since we stored the result in reverse, reverse it back to get the actual result
    return result.reverse().join('');
}


class Solution {
    def addBinary(self, a: str, b: str) -> str:
        # Initialize result list, pointers for a and b, and carry variable.
        result = []
        pointer_a, pointer_b, carry = len(a) - 1, len(b) - 1, 0

        # Loop until both pointers are out of range and there is no carry left.
        while pointer_a >= 0 or pointer_b >= 0 or carry:
            # Add carry to the sum of the current digits from a and b.
            # Use ternary operator to handle index out of range scenarios.
            carry += (0 if pointer_a < 0 else int(a[pointer_a])) + \
                     (0 if pointer_b < 0 else int(b[pointer_b]))

            # Perform division by 2 to get carry and the value to store.
            carry, value = divmod(carry, 2)

            # Append the value to the result list.
            result.append(str(value))

            # Move the pointers one step to the left.
            pointer_a, pointer_b = pointer_a - 1, pointer_b - 1

        # Since the append operation adds the least significant bits first,
        # the result string should be reversed to represent the correct binary number.
        return "".join(result[::-1])

# Example usage:
solution = Solution()
print(solution.addBinary("1010", "1011"))  # Output: "10101"
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the while loop, which continues until the end of the longer of the two binary strings `a` and `b` is reached. The loop runs once for each digit in the longer string, plus potentially one additional iteration if there is a carry out from the final addition. If `n` is the length of the longer string, then at each iteration at most a constant amount of work is done (addition, modulo operation, and index decrease), making the time complexity $O(n)$.

Space Complexity

The space complexity of the code is primarily due to the `ans` list that accumulates the results of the binary addition. In the worst case, this list will have a length that is one element longer than the length of the longer input string (in the case where there is a carry out from the last addition). This gives us a space complexity of $O(n)$, where `n` is the length of the longer string.