415. Add Strings

Easy Math String Simulation

Leetcode Link

Problem Description

The problem requires us to write a function that takes two non-negative integer numbers as strings, `num1` and `num2`, and returns their sum, also as a string. We must achieve this without using any built-in library functions for handling large numbers, such as `BigInteger`, and we are not allowed to directly convert the input strings into integers. This is a common requirement when the numbers involved can exceed the typical integer limits of the programming language, which might lead to overflow issues.

Intuition

To solve this problem, we need to mimic the way we perform addition by hand. We usually start adding the digits from the rightmost side (the least significant digit) and carry over any overflow to the next digit to the left. This process is repeated until all digits are processed.

For the implementation:

1. We initialize pointers for both input strings at their respective ends (rightmost digits).
2. We also initialize a variable `c` to keep track of the carryover during addition.
3. We iterate through both strings from right to left, digit by digit, adding corresponding digits along with any carryover from the previous step.
4. If one string is shorter and we run out of digits, we simply treat the missing digits as 0.
5. As we add digits, we calculate both the digit that should be in the current position (`v`) and the new carryover (`c`) for the next position to the left.
6. We append the result of the current single-digit addition to an answer list.
7. After processing all digits (including the final carryover if it exists), we have our answer in reverse. We reverse it and join the list into a string.
8. That string is then returned as the result.

With this approach, we can handle the addition of numbers of any size, as long as they fit into the computer's memory as strings.

Solution Approach

The implementation of the solution involves a few crucial steps and uses basic data structures like lists and strings. Here's a breakdown of the approach:

1. **Initialize Indices**: We start by initializing two indices, `i` and `j`, to point to the last characters of `num1` and `num2` respectively. These indices will be used to traverse the strings from right to left.

2. **Result List**: An empty list `ans` is created to store the digits of the result as we calculate them.

3. **Carry Variable**: A variable `c` is initialized to `0`. This variable will be responsible for holding the carry that may come from the addition of two digits that sum more than 9.

4. **Iterating Backwards**: Using a `while` loop, we iterate over the digits of the numbers from right to left. This loop continues until both indices `i` and `j` have traversed all the digits in their respective strings and there is no carry leftover.

5. **Adding Digits**: Inside the loop, we add corresponding digits from `num1` and `num2`. If one of the strings has been fully traversed (i.e., the index is less than `0`), we treat the missing digit as `0`.

6. **Handling Carry and Value**: We calculate both the carry and the value of the current digit we are processing using `divmod`. `divmod(a + b + c, 10)` gives us the carry for the next addition, and the digit to append to our result in this position.

7. **Appending the Result**: We append the value `v` as a string to our `ans` list.

8. **Updates**: We decrement both indices `i` and `j` by 1 to move to the next digits on the left.

9. **Finalizing the Result**: After processing all digits and the carry, we join the reversed `ans` list into a string and return it.

By iterating in reverse and using a simple carry system, this algorithm efficiently mimics manual addition, avoiding any potential overflow issues associated with large integer values.

Example Walkthrough

Let's walk through a simple example. Consider the input strings `num1 = "123"` and `num2 = "456"`. We want to find their sum.

1. **Initialize Indices**: Initialize `i = 2` (the index of the last character '3' in `num1`) and `j = 2` (the index of the last character '6' in `num2`).

2. **Result List**: Create an empty list `ans` to store the result digits.

3. **Carry Variable**: Initialize the carry variable `c` to `0`.

4. **Iterating Backwards**: Since `i` and `j` are both not less than `0`, and `c` is `0`, we enter the while loop.

5. **Adding Digits**: Add the digits at `i` and `j` indices from `num1` and `num2`. For the first iteration, these are `3 + 6 = 9`. No carry here, so `c` remains `0`.

6. **Appending the Result**: Append `'9'` to `ans`, making it `['9']`.

7. **Updates**: Decrement `i` and `j` to `1`.

8. **Iterating Backwards - Next Step**: `i` and `j` are still not less than `0`, we continue the loop.

9. **Adding Digits - Next Step**: Now `i = 1` and `j = 1`, so we add `'2'` and `'5'` plus the carry (`0`). The sum is `7`. We append `'7'` to `ans`, making it `['9', '7']`.

10. **Updates - Next Step**: Decrement `i` and `j` to `0`.

11. **Iterating Backwards - Next Step**: `i` and `j` are both `0`, we continue the loop.

12. **Adding Digits - Last Digit**: Now `i = 0` and `j = 0`, we add `'1'` and `'4'`. The sum is `5`. We append `'5'` to `ans`, making it `['9', '7', '5']`.

13. **Updates - Final Update**: Decrement `i` and `j` to `-1`.

14. **Finalizing the Result**: We exit the loop since `i` and `j` are both less than `0` and `c` is `0`, so we reverse `ans` to get `['5', '7', '9']`, and join it into a string to return `'579'`.

Thus, the string sum of `num1` and `num2` is `'579'`.

Python Solution

```python
class Solution:
    def addStrings(self, num1: str, num2: str) -> str:
        """Add two non-negative numbers represented as strings."""
        index1, index2 = len(num1) - 1, len(num2) - 1  # Start from the end of both strings
        result = []  # Result list to store the addition results
        carry = 0  # Initialize carry to 0 for addition

        # Loop until both strings have been processed or there is a carry
        while index1 >= 0 or index2 >= 0 or carry:
            digit1 = 0 if index1 < 0 else int(num1[index1])  # Get current digit or 0 if index is out of range
            digit2 = 0 if index2 < 0 else int(num2[index2])  # Same for second number
            carry, value = divmod(digit1 + digit2 + carry, 10)  # Add digits and carry, then divide by 10 for new carry and digit
            result.append(str(value))  # Append the computed digit to result
            index1, index2 = index1 - 1, index2 - 1  # Move to next digits

        return "".join(reversed(result))  # Reverse the result list and convert it to a string

    def subStrings(self, num1: str, num2: str) -> str:
        """Subtract two non-negative numbers represented as strings."""
        len1, len2 = len(num1), len(num2)
        negative_result = len1 < len2 or (len1 == len2 and num1 < num2)  # Determine if result should be negative
        # Swap numbers if the result is going to be negative
        if negative_result:
            num1, num2 = num2, num1

        index1, index2 = len(num1) - 1, len(num2) - 1  # Start from the end of both strings
        result = []  # Result list for storing subtraction results
        borrow = 0  # Initialize borrow to 0 for subtraction

        while index1 >= 0:
            temp = int(num1[index1]) - borrow - (0 if index2 < 0 else int(num2[index2]))
            digit = (temp + 10) % 10  # Normalize digit and possibly take a borrow
            result.append(str(digit))  # Append current digit to the result list
            borrow = 1 if temp < 0 else 0  # Update borrow
            index1, index2 = index1 - 1, index2 - 1  # Move to the next digits

        # Remove leading zeros from the result
        while len(result) > 1 and result[-1] == "0":
            result.pop()

        # Append '-' if the result is negative
        if negative_result:
            result.append("-")

        return "".join(reversed(result))  # Reverse and join the result list to form the final answer

```

Java Solution

```java
class Solution {
    // Method to add two numeric strings
    public String addStrings(String num1, String num2) {
        // Pointers to the end of each string
        int i = num1.length() - 1;
        int j = num2.length() - 1;
        StringBuilder answer = new StringBuilder();
        // Initialize carry to 0
        int carry = 0;

        // Process both strings from the end till both strings are processed or there is no carry left
        while(i >= 0 || j >= 0 || carry > 0) {
            // Get digit from string num1 if available, else use 0
            int digit1 = i < 0 ? 0 : num1.charAt(i) - '0';
            // Get digit from string num2 if available, else use 0
            int digit2 = j < 0 ? 0 : num2.charAt(j) - '0';
            // Calculate sum of digits and carry
            carry += digit1 + digit2;
            // Append the unit digit of sum to the answer
            answer.append(carry % 10);
            // Calculate new carry
            carry /= 10;
            // Move to the next digits in each string
            --i;
            --j;
        }

        // The answer is in reverse order, so reverse it to get the correct result
        answer.reverse();
        // Convert StringBuilder to String and return
        return answer.toString();
    }

    // Method to subtract two numeric strings
    public String subStrings(String num1, String num2) {
        int length1 = num1.length(), length2 = num2.length();
        // Check if the result will be negative
        boolean isNegative = length1 < length2 || (length1 == length2 && num1.compareTo(num2) < 0);
        // Swap numbers if the result is negative
        if (isNegative) {
            String temp = num1;
            num1 = num2;
            num2 = temp;
        }
        // Pointers to the end of each string
        int i = num1.length() - 1, j = num2.length() - 1;
        StringBuilder answer = new StringBuilder();
        // Initialize borrow to 0
        int borrow = 0;

        // Process the larger number from the end
        while(i >= 0) {
            // Subtract borrow and digit from num2 if available, else use 0, from the digit from num1
            borrow = (num1.charAt(i) - '0') - borrow - (j < 0 ? 0 : num2.charAt(j) - '0');
            // Handle negative results and prepare the next borrow if necessary
            answer.append((borrow + 10) % 10);
            borrow = borrow < 0 ? 1 : 0;
            // Move to the next digits
            --i;
            --j;
        }

        // Remove leading zeros from the answer
        while(answer.length() > 1 && answer.charAt(answer.length() - 1) == '0') {
            answer.deleteCharAt(answer.length() - 1);
        }

        // Append the negative sign if the result is negative
        if (isNegative) {
            answer.append('-');
        }

        // The answer is in reverse order, so reverse it to get the correct result
        answer.reverse();

        // Convert StringBuilder to String and return
        return answer.toString();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Adds two non-negative numbers represented as strings.
    string addStrings(string num1, string num2) {
        int i = num1.size() - 1;
        int j = num2.size() - 1;
        string result;
        int carry = 0; // Initialize the carry for addition to 0.

        // Loop until all digits are processed or there is a carry.
        while (i >= 0 || j >= 0 || carry > 0) {
            // Get the value of current digits and add to carry.
            int digit1 = i >= 0 ? num1[i] - '0' : 0;
            int digit2 = j >= 0 ? num2[j] - '0' : 0;
            carry += digit1 + digit2;

            // Append the current digit to the result string.
            result += to_string(carry % 10);
            carry /= 10; // Calculate carry for the next iteration.

            // Move to the next digits.
            --i;
            --j;
        }

        // Since we have added digits in reverse, reverse the string to get the final result.
        reverse(result.begin(), result.end());

        return result;
    }

    // Subtracts the smaller number from the larger number represented as strings.
    string subStrings(string num1, string num2) {
        // Determine if the result will be negative.
        bool isNegative = num1.size() < num2.size() || (num1.size() == num2.size() && num1 < num2);
        if (isNegative) {
            // Ensure num1 is always greater than num2 for direct subtraction.
            swap(num1, num2);
        }

        int i = num1.size() - 1;
        int j = num2.size() - 1;
        string result;
        int borrow = 0; // Initialize the borrow for subtraction to 0.

        // Loop until all digits of num1 are processed.
        while (i >= 0) {
            // Calculate current digits and subtract borrow.
            int diff = (num1[i] - '0') - borrow - (j < 0 ? 0 : num2[j] - '0');
            if (diff < 0) {
                diff += 10; // If difference is negative, handle the borrow.
                borrow = 1; // Set borrow for the next iteration.
            } else {
                borrow = 0; // No borrow if the difference is positive.
            }

            // Append the current digit to the result string.
            result += to_string(diff % 10);

            // Move to the next digits.
            --i;
            --j;
        }

        // Remove any leading zeros from the result string.
        while (result.length() > 1 && result.back() == '0') {
            result.pop_back();
        }

        // If the result is negative, append the negative sign.
        if (isNegative) {
            result += '-';
        }

        // Since we have subtracted digits in reverse, reverse the string to get the final result.
        reverse(result.begin(), result.end());

        return result;
    }
};

```

Typescript Solution

```typescript
function addStrings(num1: string, num2: string): string {
    const result = []; // Array to store the result of addition
    let index1 = num1.length - 1; // Start from the end of num1
    let index2 = num2.length - 1; // Start from the end of num2
    let carryOver = false; // Flag for the carry over value

    // Loop until both string indices go below zero or carry over is true
    while (index1 >= 0 || index2 >= 0 || carryOver) {
        // Get the current digit from num1 or 0 if index is out of bounds
        const digit1 = index1 >= 0 ? Number(num1[index1--]) : 0;
        // Get the current digit from num2 or 0 if index is out of bounds
        const digit2 = index2 >= 0 ? Number(num2[index2--]) : 0;
        // Calculate the sum of the two digits and the carry over, if any
        const sum = digit1 + digit2 + (carryOver ? 1 : 0);
        // If sum is greater or equal to 10, we have a new carry over
        carryOver = sum >= 10;
        // Push the last digit of the sum into the result array
        result.push(sum % 10);
    }
    // Reverse the result array and join it to form the final result string
    return result.reverse().join('');
}

```

Time and Space Complexity

The above Python code implements two functions, `addStrings` and `subStrings`, that operate on string representations of non-negative numbers.

`addStrings` Time Complexity

The function iterates over each digit of the longer input string (num1 or num2) at most once, and arithmetic computations of constant time complexity are carried out for the digits. No nested loops are present. Therefore, if `m` and `n` are the lengths of num1 and num2 respectively, the time complexity is `O(max(m, n))`.

`addStrings` Space Complexity

The space complexity is dominated by the space required for the `ans` array, which holds the result of the addition. The length of this array will be at most `max(m, n) + 1`. Therefore, the space complexity is `O(max(m, n))`.

`subStrings` Time Complexity

Similar to `addStrings`, the `subStrings` function iterates over the digits of the inputs in a single pass. Comparisons and subtractions of constant time complexity are carried out. Thus, if `m` is the length of num1 and `n` is the length of num2, the time complexity is again `O(max(m, n))`.

`subStrings` Space Complexity

The space complexity is determined by the `ans` array here as well. In the worst case, this array's length is equal to the length of the longer input string since at most one additional digit (for a possible `'-1'` carried over or a leading '–' for a negative result) can be added. Hence, the space complexity is `O(max(m, n))`.

Overall, both functions exhibit linear time and space complexities relative to the sizes of the input strings.