#### 415. Add Strings String Simulation Math Easy

#### **Problem Description**

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.

The problem requires us to write a function that takes two non-negative integer numbers as strings, num1 and num2, and returns their

### Intuition

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:

To solve this problem, we need to mimic the way we perform addition by hand. We usually start adding the digits from the rightmost

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.

1. We initialize pointers for both input strings at their respective ends (rightmost digits).

- 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.
- into a string.
- 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
- 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.
- The implementation of the solution involves a few crucial steps and uses basic data structures like lists and strings. Here's a

1. Initialize Indices: We start by initializing two indices, i and j, to point to the last characters of num1 and num2 respectively. These

#### breakdown of the approach:

**Solution Approach** 

8. That string is then returned as the result.

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

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.

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

both indices i and j have traversed all the digits in their respective strings and there is no carry leftover.

- 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.
- overflow issues associated with large integer values. Example Walkthrough

By iterating in reverse and using a simple carry system, this algorithm efficiently mimics manual addition, avoiding any potential

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).

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

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

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

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

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

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

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.

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,

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',

making it ['9', '7']. 10. **Updates - Next Step**: Decrement **i** and **j** to **0**.

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

'5'].

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

Python Solution

1 class Solution:

6

8

9

10

16

17

18

19

20

21

22

23

28

29

30

31

32

33

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

8

9

10

11

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

46

47

48

49

50

51

C++ Solution

public:

1 class Solution {

answer.reverse();

if (isNegative) {

int borrow = 0;

 $while(i >= 0) {$ 

--j;

if (isNegative)

answer.reverse();

answer.append('-');

return answer.toString();

int i = num1.size() - 1;

int j = num2.size() - 1;

string result;

--i;

--j;

return result;

if (isNegative) {

while (i >= 0) {

**if** (diff < 0) {

swap(num1, num2);

num1 = num2;

num2 = temp;

return answer.toString();

String temp = num1;

// Initialize borrow to 0

// Method to subtract two numeric strings

// Convert StringBuilder to String and return

public String subStrings(String num1, String num2) {

// Check if the result will be negative

// Pointers to the end of each string

// Swap numbers if the result is negative

int length1 = num1.length(), length2 = num2.length();

int i = num1.length() - 1, j = num2.length() - 1;

StringBuilder answer = new StringBuilder();

// Process the larger number from the end

answer.append((borrow + 10) % 10);

borrow = borrow < 0 ? 1 : 0;

// Remove leading zeros from the answer

answer.deleteCharAt(answer.length() - 1);

// Convert StringBuilder to String and return

// Adds two non-negative numbers represented as strings.

int digit1 =  $i \ge 0$  ? num1[i] - '0' : 0;

int digit2 = j >= 0 ? num2[j] - '0' : 0;

int carry = 0; // Initialize the carry for addition to 0.

// Loop until all digits are processed or there is a carry.

// Get the value of current digits and add to carry.

// Append the current digit to the result string.

carry /= 10; // Calculate carry for the next iteration.

// Subtracts the smaller number from the larger number represented as strings.

// Ensure num1 is always greater than num2 for direct subtraction.

int diff = (num1[i] - '0') - borrow - (j < 0 ? 0 : num2[j] - '0');

diff += 10; // If difference is negative, handle the borrow.

// Since we have added digits in reverse, reverse the string to get the final result.

bool isNegative = num1.size() < num2.size() || (num1.size() == num2.size() && num1 < num2);</pre>

string addStrings(string num1, string num2) {

while (i >= 0 || j >= 0 || carry > 0) {

result += to\_string(carry % 10);

carry += digit1 + digit2;

// Move to the next digits.

reverse(result.begin(), result.end());

string subStrings(string num1, string num2) {

// Determine if the result will be negative.

// Loop until all digits of num1 are processed.

// Calculate current digits and subtract borrow.

// Append the negative sign if the result is negative

// Move to the next digits

boolean isNegative = length1 < length2 || (length1 == length2 && num1.compareTo(num2) < 0);</pre>

// 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

while(answer.length() > 1 && answer.charAt(answer.length() - 1) == '0') {

// The answer is in reverse order, so reverse it to get the correct result

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'.
- """Add two non-negative numbers represented as strings.""" 3 index1, index2 = len(num1) - 1, len(num2) - 1 # Start from the end of both strings result = [] # Result list to store the addition results

def addStrings(self, num1: str, num2: str) -> str:

while index1 >= 0 or index2 >= 0 or carry:

def subStrings(self, num1: str, num2: str) -> str:

len1, len2 = len(num1), len(num2)

carry = 0 # Initialize carry to 0 for addition

# Loop until both strings have been processed or there is a carry

"""Subtract two non-negative numbers represented as strings."""

# Swap numbers if the result is going to be negative

borrow = 0 # Initialize borrow to 0 for subtraction

11 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 12 13 result.append(str(value)) # Append the computed digit to result index1, index2 = index1 - 1, index2 - 1 # Move to next digits 14 15

digit1 = 0 if index1 < 0 else int(num1[index1]) # Get current digit or 0 if index is out of range

negative\_result = len1 < len2 or (len1 == len2 and num1 < num2) # Determine if result should be negative

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

temp = int(num1[index1]) - borrow - (0 if index2 < 0 else int(num2[index2]))</pre>

digit = (temp + 10) % 10 # Normalize digit and possibly take a borrow

result.append(str(digit)) # Append current digit to the result list

24 num1, num2 = num2, num125 index1, index2 = len(num1) - 1, len(num2) - 1 # Start from the end of both strings 26 27 result = [] # Result list for storing subtraction results

while index1 >= 0:

if negative\_result:

```
borrow = 1 if temp < 0 else 0 # Update borrow
 34
 35
                 index1, index2 = index1 - 1, index2 - 1 # Move to the next digits
 36
 37
             # Remove leading zeros from the result
 38
             while len(result) > 1 and result[-1] == "0":
                 result.pop()
 39
 40
 41
             # Append '-' if the result is negative
 42
             if negative_result:
                 result.append("-")
 43
 44
 45
             return "".join(reversed(result)) # Reverse and join the result list to form the final answer
 46
Java Solution
  1 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;
  5
             int j = num2.length() - 1;
  6
             StringBuilder answer = new StringBuilder();
  8
             // Initialize carry to 0
             int carry = 0;
  9
 10
             // Process both strings from the end till both strings are processed or there is no carry left
 11
             while(i >= 0 || j >= 0 || carry > 0) {
 12
 13
                 // Get digit from string num1 if available, else use 0
                 int digit1 = i < 0 ? 0 : num1.charAt(i) - '0';</pre>
 14
 15
                 // Get digit from string num2 if available, else use 0
                 int digit2 = j < 0 ? 0 : num2.charAt(j) - '0';</pre>
 16
                 // Calculate sum of digits and carry
                 carry += digit1 + digit2;
 10
 19
                 // Append the unit digit of sum to the answer
 20
                 answer.append(carry % 10);
 21
                 // Calculate new carry
 22
                 carry /= 10;
 23
                 // Move to the next digits in each string
 24
                 --i;
 25
                 --j;
 26
 27
 28
             // The answer is in reverse order, so reverse it to get the correct result
```

```
40
41
            int i = num1.size() - 1;
           int j = num2.size() - 1;
42
43
            string result;
44
            int borrow = 0; // Initialize the borrow for subtraction to 0.
45
```

```
52
                     borrow = 1; // Set borrow for the next iteration.
 53
                 } else {
                     borrow = 0; // No borrow if the difference is positive.
 54
 55
 56
 57
                 // Append the current digit to the result string.
 58
                 result += to_string(diff % 10);
 59
 60
                 // Move to the next digits.
 61
 62
                 --j;
 63
 64
 65
             // Remove any leading zeros from the result string.
             while (result.length() > 1 && result.back() == '0') {
 66
 67
                 result.pop_back();
 68
 69
             // If the result is negative, append the negative sign.
 70
 71
             if (isNegative) {
 72
                 result += '-';
 73
 74
 75
             // Since we have subtracted digits in reverse, reverse the string to get the final result.
 76
             reverse(result.begin(), result.end());
 77
 78
             return result;
 79
 80
    };
 81
Typescript Solution
   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 || carry0ver) {
           // Get the current digit from num1 or 0 if index is out of bounds
           const digit1 = index1 >= 0 ? Number(num1[index1--]) : 0;
10
           // 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 + (carry0ver ? 1 : 0);
14
           // If sum is greater or equal to 10, we have a new carry over
15
16
           carryOver = sum >= 10;
           // Push the last digit of the sum into the result array
           result.push(sum % 10);
18
19
       // Reverse the result array and join it to form the final result string
20
       return result.reverse().join('');
21
22 }
23
Time and Space Complexity
```

# addStrings Time Complexity

**subStrings** Time Complexity

0(max(m, n)).

negative numbers.

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 0(max(m, n)).

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

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

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

The function iterates over each digit of the longer input string (num1 or num2) at most once, and arithmetic computations of

## subStrings Space Complexity

num2 respectively, the time complexity is  $0(\max(m, n))$ .

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.