2. Add Two Numbers

Medium Recursion Linked List Math Leetcode Link

Problem Description

Imagine you have two numbers, but instead of writing them down in the usual way, you write each digit down separately, in reverse order, and then link all these digits together into a chain where each link is a single digit. These chains are what we call linked lists in computer science, and each digit lives in its own node, a little container with the number and a pointer to the next digit in the list.

Now, let's say someone gives you two of these chains, both representing non-negative integers, and asks you to add these numbers together just like you would on a piece of paper. But here's the twist: the result should be presented in the same reversed chain format.

The problem resembles simple addition, starting from the least significant digit (which is at the head of the chain because of the reverse order) and moving up to the most significant one, carrying over any overflow.

And there's one more thing - if our numbers were zeroes, they wouldn't have any leading digits, except for a single zero node to represent the number itself.

The challenge here is to simulate this familiar arithmetic operation using the rules and structures of linked lists.

Intuition

Adding two numbers is something we learn early in school, and the process is almost second nature - start with the rightmost digits, add them, carry over if needed, and move left. Doing this with linked lists means mimicking this step-by-step addition. However, linked lists don't allow us to access an element directly by position; we traverse from the start node to the end node.

We start the simulation by simultaneously walking down both linked lists - these are our two numbers in reverse order. At each step, we sum the current digit of each number along with any carry left over from the previous step.

We keep track of the carry-over because, during addition, if we add two digits and the sum is 10 or higher, we need to carry over the '1' to the next set of digits. In coding terms, think of 'carry' as a temporary storage space where we keep that extra digit.

To hold our resulting number, we create a new linked list (we'll call it the 'sum list'). For each pair of digits we add, we'll create a new node in the 'sum list' that contains the result of the addition modulo 10 (which is the remainder after division by 10 – basically, the digit without the carry part). The carry (if any) is computed as the floor division of the sum by 10.

We continue traversing both input lists, adding corresponding digits and carry as we go. If one list is longer, we simply carry on with the digits that remain. After we've exhausted both lists, if there's still a carry, it means we need one more node with the value of the carry.

When we're done adding, we simply return the head of our 'sum list,' and voilà, that's our total, neatly reversed just as we started.

Solution Approach

As outlined in the reference solution approach, we simulate the addition process using a simple iterative method. In computational terms, this is relatively straightforward.

Firstly, we need a placeholder for the sum of the two numbers, which in this case will be a new linked list. We create a `dummy` node that acts as the head of our sum list. This `dummy` node is very handy because it allows us to easily return the summed list at the end, avoiding the complexities of handling where the list begins in the case of an overflow on the most significant digit.

We initialize two variables:

- The `carry` variable (starting at `0`), to keep track our carryover in each iteration.
- The `curr` variable, which points to the current node in the sum list; initially, this is set to the `dummy` node.

We enter a loop that traverses both input linked lists. The loop continues as long as at least one of the following conditions holds true: there is at least one more node in either `l1` or `l2`, or there is a non-zero value in `carry`. Within the loop:

- We sum the current digit from each list (`l1.val` and `l2.val`) with the current carry. If we've reached the end of a list, we treat the missing digits as `0`.
- The sum produced can be broken down into two parts: the digit at the current position, and the carryover for the next position. This is computed using `divmod(s, 10)` which gives us the quotient representing `carry` and the remainder representing `val` – the current digit to add to our sum list.
- We create a new node for `val` and find its place at the end of the sum list indicated by `curr.next`.
- We update `curr` to point to this newly added node.
- We update `l1` and `l2` to point to their respective next nodes – moving to the next digit or setting to `None` if we've reached the end of the list.

The loop exits once there are no more digits to add and no carry. Since `dummy` was only a placeholder, the actual resultant list starts from `dummy.next`.

Lastly, we return `dummy.next`, which points to the head of the linked list representing the sum of our two numbers. The way our loop is structured ensures that this process carries out the addition operation correctly for linked lists of unequal lengths as well, without any additional tweaks or condition checks.

Example Walkthrough

To illustrate the solution approach, consider two linked lists representing the numbers 342 and 465. The linked lists would look like this:

- `l1`: 2 -> 4 -> 3 (Representing 342 written in reverse as a linked list)
- `l2`: 5 -> 6 -> 4 (Representing 465 written in reverse as a linked list)

According to the solution:

1. Initialize a `dummy` node to serve as the head of the new linked list that will store our result.
2. Set a `carry` variable to `0` and `curr` to point to `dummy`.
3. Iterate over `l1` and `l2` as long as there is a node in either list or `carry` is not zero.

   For the first iteration:

   - `l1.val` is `2`, and `l2.val` is `5`. The sum is `2 + 5 + carry (0) = 7`.
   - The digit for the new node is `7 % 10 = 7`, and the new carry is `7 // 10 = 0`.
   - Create a new node with a value of `7` and link it to `curr`.

The resulting list is now `7`, the `dummy` node points to `7`, and `curr` also points to `7`.

Continuing to the second digit:

- `l1.val` is `4`, `l2.val` is `6`. The sum is `4 + 6 + carry (0) = 10`.
- The digit for the new node is `10 % 10 = 0`, and the new carry is `10 // 10 = 1`.
- Create a new node with a value of `0` and link it to `curr`.

Now, the resulting list is `7 -> 0`, `dummy` points to `7`, and `curr` points to `0`.

For the third digit:

- `l1.val` is `3`, `l2.val` is `4`. The sum is `3 + 4 + carry (1) = 8`.
- The digit for the new node is `8 % 10 = 8`, and the new carry is `8 // 10 = 0`.
- Create a new node with a value of `8` and link it to `curr`.

Our final list becomes `7 -> 0 -> 8`, which is the reverse of `807`, the sum of `342` and `465`.

After the end of the loop, we check to see if `carry` is non-zero. In this case, it's zero, so we do not add another node.

Lastly, since the `dummy` was just a placeholder, we return `dummy.next`, which gives us `7 -> 0 -> 8`, the final linked list representing our summed number in reverse order.

Python Solution

```python
class ListNode:
    def __init__(self, value=0, next_node=None):
        self.value = value
        self.next_node = next_node

class Solution:
    def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
        # Initialize a dummy head to build the result list
        dummy_head = ListNode()
        # Initialize the current node to the dummy head and a carry variable
        carry, current = 0, dummy_head

        # Loop until both lists are exhausted and there is no carry left
        while l1 or l2 or carry:
            # Calculate the sum using the values of the current nodes and the carry
            sum_ = (l1.value if l1 else 0) + (l2.value if l2 else 0) + carry
            # Update carry for next iteration (carry, if any, would be 1)
            carry, value = divmod(sum_, 10)
            # Create the next node with the sum value mod 10
            current.next_node = ListNode(value)
            # Move to the next node in the result list
            current = current.next_node
            # Move to the next nodes on the input lists, if available
            l1 = l1.next_node if l1 else None
            l2 = l2.next_node if l2 else None

        # Return the result list, which starts from the dummy head's next node
        return dummy_head.next_node
```

Java Solution

```java
// Definition for singly-linked list.
class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
    public ListNode addTwoNumbers(ListNode firstList, ListNode secondList) {
        // Create a dummy node which will be the starting point of the result list.
        ListNode dummyNode = new ListNode(0);

        // This variable will keep track of the carry-over.
        int carry = 0;

        // This will be used to iterate over the new list.
        ListNode current = dummyNode;

        // Iterate as long as there is a node left in either list or there is a carry-over.
        while (firstList != null || secondList != null || carry != 0) {
            // Sum the values of the two nodes if they are not null, else add 0.
            int sum = (firstList == null ? 0 : firstList.val) + (secondList == null ? 0 : secondList.val) + carry;

            // Update carry for the next iteration.
            carry = sum / 10;

            // Create a new node with the digit value of the sum.
            current.next = new ListNode(sum % 10);

            // Move to the next node in the result list.
            current = current.next;

            // Proceed in each input list.
            firstList = firstList == null ? null : firstList.next;
            secondList = secondList == null ? null : secondList.next;
        }

        // The first node was a dummy node, so the real list starts at dummyNode.next.
        return dummyNode.next;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    // Function to add two numbers represented by two linked lists
    ListNode* addTwoNumbers(ListNode *l1, ListNode* l2) {
        // Initialize a dummy head to build the result list
        ListNode* dummyHead = new ListNode();
        // Variable to keep track of the carry
        int carry = 0;
        // Pointer to 'current' to add new nodes to the result list
        ListNode* current = dummyHead;
        // Continue looping until both lists are traversed completely and there is no carry
        while (l1 || l2 || carry) {
            // Calculate the sum of the current digits along with the carry
            int sum = (l1 ? l1->val : 0) + (l2 ? l2->val : 0) + carry;
            // Update the carry for the next iteration
            carry = sum / 10;
            // Create a new node with the digit part of the sum and append to the result list
            current->next = new ListNode(sum % 10);
            // Move the 'current' pointer to the new node
            current = current->next;
            // Move the list pointers l1 and l2 to the next nodes if they exist
            l1 = l1 ? l1->next : nullptr;
            l2 = l2 ? l2->next : nullptr;
        }
        // The result list starts after the dummy head's next pointer
        return dummyHead->next;
    }
};
```

Typescript Solution

```typescript
// ListNode class definition for a singly-linked list.
class ListNode {
    val: number
    next: ListNode | null;
    constructor(val?: number, next?: ListNode | null) {
        this.val = val === undefined ? 0 : val;
        this.next = next === undefined ? null : next;
    }
}

// Adds two numbers represented by two singly linked lists (l1 and l2) and returns the sum as a linked list.
function addTwoNumbers(l1: ListNode | null, l2: ListNode | null): ListNode | null {
    // A dummy head node for the resulting linked list, used to simplify appending new nodes.
    const dummyHead = new ListNode();
    // The current node in the resulting linked list as we are building it.
    let currentNode = dummyHead;
    // The sum variable carries value when we need to 'carry' a digit to the next decimal place.
    let carry = 0;

    // Iterate while there is something to add, or we have a carry from the last digits.
    while (carry !== 0 || l1 !== null || l2 !== null) {
        // Add the values from l1 and l2 to the carry if they are available.
        if (l1 !== null) {
            carry += l1.val;
            l1 = l1.next; // Move to the next node in l1.
        }
        if (l2 !== null) {
            carry += l2.val;
            l2 = l2.next; // Move to the next node in l2.
        }
        // The new digit is the remainder of sum when divided by 10.
        currentNode.next = new ListNode(carry % 10);
        // Move to the newly created node.
        currentNode = currentNode.next;
        // Calculate the new carry, which is the floor division of sum by 10.
        carry = Math.floor(carry / 10);
    }
    // Return the next node of dummyHead to skip the dummy node at the beginning.
    return dummyHead.next;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(max(m, n))`, where `m` and `n` are the lengths of the input linked lists `l1` and `l2`, respectively. This is because we iterate through both lists in parallel, and at each step, we add the corresponding nodes' values along with any carry from the previous step, which takes constant time `O(1)`. The iteration continues until both lists have been fully traversed and there is no carry left to add.

Space Complexity

The space complexity of the code is `O(1)`, ignoring the space consumption of the output list. The variables `carry`, `curr`, and the nodes we iterate through (`l1` and `l2`) only use a constant amount of space. However, if we take into account the space required for the result list, the space complexity would be `O(max(m, n))`, since in the worst case, the resultant list could be as long as the longer of the two input lists, plus one extra node for an additional carry.