2. Add Two Numbers

Medium Recursion Linked List Math

Problem Description

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

Imagine you have two numbers, but instead of writing them down in the usual way, you write each digit down separately, in

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

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.

Adding two numbers is something we learn early in school, and the process is almost second nature - start with the rightmost

Intuition

reverse order) and moving up to the most significant one, carrying over any overflow.

digits, add them, carry over if needed, and move left. Doing this with linked lists means mimicking this step-by-step addition.

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

However, linked lists don't allow us to access an element directly by position; we traverse from the start node to the end node.

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 <u>linked list</u> (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.

Firstly, we need a placeholder for the sum of the two numbers, which in this case will be a new <u>linked list</u>. We create a <u>dummy</u>

the end, avoiding the complexities of handling where the list begins in the case of an overflow on the most significant digit.

As outlined in the reference solution approach, we simulate the addition process using a simple iterative method. In

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

We initialize two variables:

as 0.

starts from dummy_next.

Solution Approach

computational terms, this is relatively straightforward.

• 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

• We sum the current digit from each list (11.val and 12.val) with the current carry. If we've reached the end of a list, we treat the missing digits

• 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

true: there is at least one more node in either 11 or 12, or there is a non-zero value in carry. Within the loop:

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 11 and 12 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

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

Initialize a dummy node to serve as the head of the new linked list that will store our result.

Iterate over 11 and 12 as long as there is a node in either list or carry is not zero.

Set a carry variable to 0 and curr to point to dummy.

• l1.val is 4, l2.val is 6. The sum is 4 + 6 + carry (0) = 10.

• l1.val is 3, l2.val is 4. The sum is 3 + 4 + carry (1) = 8.

Create a new node with a value of 0 and link it to curr.

Create a new node with a value of 8 and link it to curr.

representing our summed number in reverse order.

def init (self, value=0, next_node=None):

Initialize a dummy head to build the result list

Initialize the current node to the dummy head and a carry variable

Update carry for next iteration (carry, if any, would be 1)

Return the result list, which starts from the dummy head's next node

ListNode(int val, ListNode next) { this.val = val; this.next = next; }

public ListNode addTwoNumbers(ListNode firstList, ListNode secondList) {

// Create a dummy node which will be the starting point of the result list.

// Iterate as long as there is a node left in either list or there is a carry-over.

Calculate the sum using the values of the current nodes and the carry

sum = (list1.value if list1 else 0) + (list2.value if list2 else 0) + carry

Loop until both lists are exhausted and there is no carry left

Create the next node with the sum value mod 10

Solution Implementation

dummy head = ListNode()

carry, current = 0, dummy_head

while list1 or list2 or carry:

return dummy_head.next_node

ListNode(int val) { this.val = val; }

ListNode current = dummyNode;

current = current->next;

return dummyHead->next;

const dummyHead = new ListNode();

let currentNode = dummyHead;

if (list1 !== null) {

dummy head = ListNode()

carry, current = 0, dummy_head

while list1 or list2 or carry:

carry, value = divmod(sum , 10)

current = current.next node

fully traversed and there is no carry left to add.

return dummy_head.next_node

Time and Space Complexity

current.next node = ListNode(value)

};

TypeScript

class ListNode {

val: number;

let carry = 0;

next: ListNode | null;

l1 = l1 ? l1->next : nullptr;

l2 = l2 ? l2->next : nullptr;

// ListNode class definition for a singly-linked list.

this.val = val === undefined ? 0 : val;

constructor(val?: number, next?: ListNode | null) {

this.next = next === undefined ? null : next;

while (carry !== 0 || list1 !== null || list2 !== null) {

Initialize a dummy head to build the result list

Initialize the current node to the dummy head and a carry variable

Update carry for next iteration (carry, if any, would be 1)

Move to the next nodes on the input lists, if available

longer of the two input lists, plus one extra node for an additional carry.

Return the result list, which starts from the dummy head's next node

Calculate the sum using the values of the current nodes and the carry

sum = (list1.value if list1 else 0) + (list2.value if list2 else 0) + carry

Loop until both lists are exhausted and there is no carry left

Create the next node with the sum value mod 10

Move to the next node on the result list

list1 = list1.next node if list1 else None

list2 = list2.next_node if list2 else None

// Move the list pointers l1 and l2 to the next nodes if they exist

// Adds two numbers represented by two singly linked lists (l1 and l2) and returns the sum as a linked list.

def addTwoNumbers(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:

function addTwoNumbers(list1: ListNode | null. list2: ListNode | null): ListNode | null {

// Iterate while there is something to add, or we have a carry from the last digits.

// Add the values from list1 and list2 to the carry if they are available.

// The current node in the resulting linked list as we are building it.

// A dummy head node for the resulting linked list, used to simplify appending new nodes.

// The sum variable carries value when we need to 'carry' a digit to the next decimal place.

// The result list starts after the dummy head's next pointer

ListNode dummyNode = new ListNode(0);

// This variable will keep track of the carry-over.

// This will be used to iterate over the new list.

carry, value = divmod(sum , 10)

current.next node = ListNode(value)

Python

class ListNode:

• 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.

11: 2 → 4 → 3 (Representing 342 written in reverse as a linked list)
12: 5 → 6 → 4 (Representing 465 written in reverse as a linked list)

For the first iteration:

3.

According to the solution:

Continuing to the second digit:

like this:

• 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.

For the third digit:

Now, the resulting list is $7 \rightarrow 0$, dummy points to 7, and curr points to 0.

• The digit for the new node is 8 % 10 = 8, and the new carry is 8 // 10 = 0.

The digit for the new node is 10 % 10 = 0, and the new carry is 10 // 10 = 1.

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.

Our final list becomes $7 \rightarrow 0 \rightarrow 8$, which is the reverse of 807, the sum of 342 and 465.

self.value = value
self.next_node = next_node
class Solution:

def addTwoNumbers(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:

Move to the next node on the result list
current = current.next node
Move to the next nodes on the input lists, if available
list1 = list1.next node if list1 else None
list2 = list2.next_node if list2 else None

Lastly, since the dummy was just a placeholder, we return dummy next, which gives us 7 -> 0 -> 8, the final linked list

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

class Solution {

int carry = 0;

Java

```
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++
/**
 * 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;
        // We use '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
```

```
carrv += list1.val;
            list1 = list1.next; // Move to the next node in list1.
        if (list2 !== null) {
            carry += list2.val;
            list2 = list2.next; // Move to the next node in list2.
       // 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;
class ListNode:
    def init (self. value=0, next_node=None):
       self.value = value
       self.next_node = next_node
class Solution:
```

The time complexity of the given code is $0(\max(m, n))$, where m and n are the lengths of the input linked lists 11 and 12, respectively. This is because we iterate through both lists in parallel, and at each step, we add the corresponding nodes' values

Time Complexity

Space Complexity

The space complexity of the code is 0(1), ignoring the space consumption of the output list. The variables carry, curr, and the nodes we iterate through (11 and 12) 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 0(max(m, n)), since in the worst case, the resultant list could be as long as the

along with any carry from the previous step, which takes constant time 0(1). The iteration continues until both lists have been