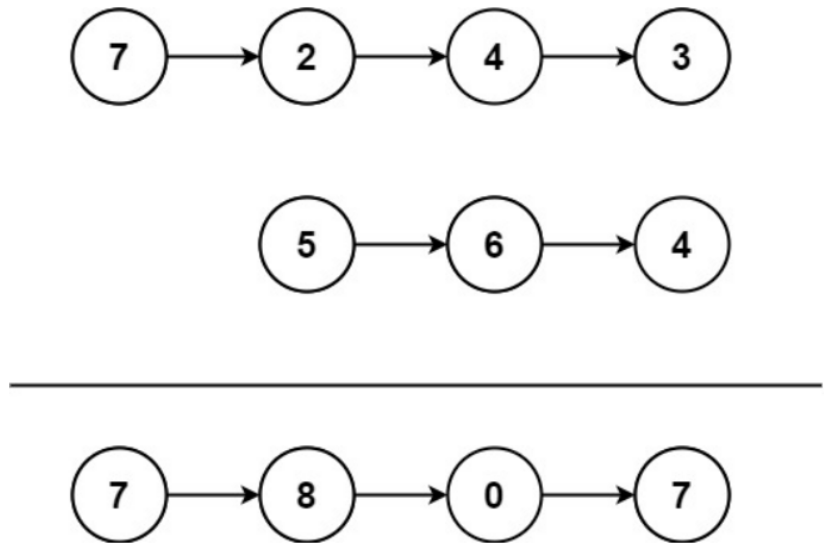
<https://leetcode.com/problems/add-two-numbers-ii/>

You are given two **non-empty** linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example 1:**



Input: l1 = [7,2,4,3], l2 = [5,6,4]

Output: [7,8,0,7]

**Example 2:**

Input: l1 = [2,4,3], l2 = [5,6,4]

Output: [8,0,7]

**Example 3:**

Input: l1 = [0], l2 = [0]

Output: [0]

**Constraints:**

* The number of nodes in each linked list is in the range [1, 100].
* 0 <= Node.val <= 9
* It is guaranteed that the list represents a number that does not have leading zeros.

**Follow up:** Could you solve it without reversing the input lists?

**Attempt 1: 2023-02-14**

**Solution 1:  Recursive Solution by making up 0 ahead (30 min)**

/\*\*

\* Definition for singly-linked list.

\* public 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 {

// Setup 'carry' as global variable to make sure go through

// each recursion (digit by digit sum up from right to left)

// without losing previous status

int carry = 0;

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

// Adding zeros to the start of the smaller list

ListNode iter1 = l1;

ListNode iter2 = l2;

while(iter1 != null || iter2 != null) {

if(iter1 == null) {

ListNode newNode = new ListNode(0);

newNode.next = l1;

l1 = newNode;

iter2 = iter2.next;

} else if(iter2 == null) {

ListNode newNode = new ListNode(0);

newNode.next = l2;

l2 = newNode;

iter1 = iter1.next;

} else {

iter1 = iter1.next;

iter2 = iter2.next;

}

}

//int carry = 0;

ListNode dummy = new ListNode();

//dummy.next = helper(l1, l2, carry);

dummy.next = helper(l1, l2);

// If the right most digit sum > 10 (carry > 0), insert 'carry'

// as a node between dummy node and original head 'dummy.next'

if(carry > 0) {

ListNode newNode = new ListNode(carry);

newNode.next = dummy.next;

dummy.next = newNode;

}

return dummy.next;

}

// Note: The 'carry' cannot be local variable in this style

//private ListNode helper(ListNode l1, ListNode l2, int carry) {

// if(l1 == null && l2 == null) {

// return null;

// }

// ListNode tmp = new ListNode(0);

// tmp.next = helper(l1.next, l2.next, carry);

// tmp.val = (l1.val + l2.val + carry) % 10;

// carry = (l1.val + l2.val + carry) / 10;

// return tmp;

//}

// The natural nature of recursion conduct add operation start from

// right end backwards to left end

private ListNode helper(ListNode l1, ListNode l2) {

if(l1 == null && l2 == null) {

return null;

}

ListNode tmp = new ListNode(0);

// Why 'tmp.next' ?

// Think about example:

// 7->2->4->3->null

// 0->5->6->4->null

// The 1st recursion will start from null after right most digit, because

// the recursion termination condition is (l1 == null && l2 == null) and

// return null, this happen when reach both null on right end of 3 and 4.

// The 2nd recursion will happen on right most digit (3 and 4), since in

// 1st recursion it already return null, what's the relation between sum

// up of 3 and 4 on right most digit and previous returned null ?

// It has to be: 3 + 4 + 0 = 7(tmp) -> null, represent as "tmp.next = null"

// and since 'null' comes return result from 1st recursion, so full statement

// is "tmp.next = helper(l1.next, l2.next)".

// The 3rd recursion will be similar to 2nd one, 4 + 6 + 0 = 10(tmp) -> 7 -> null

// ...etc. Then comes to logic for handling '10' and 'carry', especially for

// 'carry' must use global variable in current style which only return node

tmp.next = helper(l1.next, l2.next);

tmp.val = (l1.val + l2.val + carry) % 10;

carry = (l1.val + l2.val + carry) / 10;

return tmp;

}

}

**Refer to**

<https://leetcode.com/problems/add-two-numbers-ii/solutions/798754/easy-c-solution-without-using-stack-or-reversing-the-input-output-list/>

The idea is to add zeros to the start of the smaller list (which is **allowed** since it doesn't tamper with the original structure of the list) such that both the lists become of equal size and then use recursion to perform digit by digit addition (starting from the last digits, obviously).

For example consider the following lists,

l1: 7->2->4->3

l2: 5->6->4

After adding zeros to l2 (the smaller list), the lists become,

l1: 7->2->4->3

l2: 0->5->6->4

We now use recursion to dive to the end of both the lists and start addition from the end. After each recursion ends, l1 and l2 will be waiting at the previous nodes, so an indirect reverse traversal is obtained without the use of a doubly linked list. The key is being able to pass carry from current recursive function to the previous recursive function, for which we can pass the reference variable carry to function calls so that the changes made to carry reflect through all the recursive calls made.

If the concept is still not clear, dry running the code using pen and paper will surely help.

(P.S - This program might seem lengthy, and although most of it stems from the "adding zero" process, I did not want to reduce the length and compromise readability.)

class Solution {

public:

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

//Adding zeros to the start of the smaller list:

ListNode \*ptr1 = l1, \*ptr2 = l2;

while(ptr1 != NULL || ptr2 != NULL)

{

if(ptr1 == NULL)

{

ListNode \*newNode = new ListNode(0);

newNode->next = l1;

l1 = newNode;

ptr2 = ptr2->next;

}

else if(ptr2 == NULL)

{

ListNode \*newNode = new ListNode(0);

newNode->next = l2;

l2 = newNode;

ptr1 = ptr1->next;

}

else

{

ptr1 = ptr1->next;

ptr2 = ptr2->next;

}

}

//Main operation:

int carry = 0;

ListNode \*dummy = new ListNode(-1);

dummy->next = addTwoDigit(l1, l2, carry);

if(carry != 0)

{

ListNode \*newNode = new ListNode(carry);

newNode->next = dummy->next;

dummy->next = newNode;

}

return dummy->next;

}

ListNode\* addTwoDigit(ListNode\* l1, ListNode\* l2, int &carry)

{

if(l1 == NULL && l2 == NULL)

return NULL;

ListNode \*newNode = new ListNode(-1);

newNode->next = addTwoDigit(l1->next, l2->next, carry);

newNode->val = (l1->val + l2->val + carry) % 10;

carry = (l1->val + l2->val + carry) / 10;

return newNode;

}

};

**Solution 2:  Iterative Solution with reverse linked list (10 min)**

/\*\*

\* Definition for singly-linked list.

\* public 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 l1, ListNode l2) {

ListNode r1 = reverse(l1);

ListNode r2 = reverse(l2);

ListNode dummy = new ListNode();

ListNode iter = dummy;

int sum = 0;

while(r1 != null || r2 != null) {

// Get 'carry' for current digit based on previous digit sum

sum /= 10;

if(r1 != null) {

sum += r1.val;

r1 = r1.next;

}

if(r2 != null) {

sum += r2.val;

r2 = r2.next;

}

iter.next = new ListNode(sum % 10);

iter = iter.next;

}

if(sum / 10 == 1) {

iter.next = new ListNode(1);

}

// 'dummy.next' represent the first digit of reversed two lists

// e.g

// 7 -> 2 -> 4 -> 3 => reverse to 3 -> 4 -> 2 -> 7

// 5 -> 6 -> 4 => reverse to 4 -> 6 -> 5

// 'dummy.next' point to first node as (3 + 4 = 7)

return reverse(dummy.next);

}

private ListNode reverse(ListNode head) {

ListNode prev = null;

ListNode cur = head;

while(cur != null) {

ListNode next = cur.next;

cur.next = prev;

prev = cur;

cur = next;

}

return prev;

}

}

**Solution 3:  Iterative Solution with Stack (10 min)**

/\*\*

\* Definition for singly-linked list.

\* public 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 l1, ListNode l2) {

Stack<Integer> s1 = new Stack<Integer>();

Stack<Integer> s2 = new Stack<Integer>();

Stack<Integer> s = new Stack<Integer>();

while(l1 != null) {

s1.push(l1.val);

l1 = l1.next;

}

while(l2 != null) {

s2.push(l2.val);

l2 = l2.next;

}

ListNode dummy = new ListNode();

ListNode iter = dummy;

int sum = 0;

while(!s1.empty() || !s2.empty()) {

sum /= 10;

if(!s1.empty()) {

sum += s1.pop();

}

if(!s2.empty()) {

sum += s2.pop();

}

s.push(sum % 10);

}

if(sum / 10 == 1) {

s.push(1);

}

while(!s.empty()) {

iter.next = new ListNode(s.pop());

iter = iter.next;

}

return dummy.next;

}

}