Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

Solution :

import java.util.\*;

class Solution {

    public int[] twoSum(int[] nums, int target) {

        int length = nums.length;

        for(int i = 0 ; i< length-1; i++)

        {

            for(int j = i+1; j < length; j++)

            {

                if(nums[i] + nums[j] == target)

                {

                    return new int[]{i,j};

                }

            }

        }

            return new int[] {-1,-1};

    }

}

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, 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:**

O imagine care conține diagramă

Descriere generată automat

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

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

**Example 2:**

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

**Output:** [0]

**Example 3:**

**Input:** l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

**Output:** [8,9,9,9,0,0,0,1]

**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.

Solution :

/\*\*

 \* 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 {

    // Add Two Numbers (Java improved)

    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

        ListNode dummyHead = new ListNode(0);

        ListNode curr = dummyHead;

        int carry = 0;

        while (l1 != null || l2 != null || carry != 0) {

            int x = (l1 != null) ? l1.val : 0;

            int y = (l2 != null) ? l2.val : 0;

            int sum = carry + x + y;

            carry = sum / 10;

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

            curr = curr.next;

            if (l1 != null)

                l1 = l1.next;

            if (l2 != null)

                l2 = l2.next;

        }

        return dummyHead.next;

    }

}

9. Palindrome Number

Given an integer x, return true*if*x*is a*

***palindrome***

*, and*false*otherwise*.

**Example 1:**

**Input:** x = 121

**Output:** true

**Explanation:** 121 reads as 121 from left to right and from right to left.

**Example 2:**

**Input:** x = -121

**Output:** false

**Explanation:** From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

**Example 3:**

**Input:** x = 10

**Output:** false

**Explanation:** Reads 01 from right to left. Therefore it is not a palindrome.

class Solution {

    public boolean isPalindrome(int x) {

        int aux = 0;

        int y = x;

        if(x < 0 )

         return false;

        while(y != 0)

        {

            aux = aux\*10 + y%10;

            y = y/10;

        }

        if(aux == x)

           return true;

           else

             return false;

    }

}

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

**Input:** s = "III"

**Output:** 3

**Explanation:** III = 3.

**Example 2:**

**Input:** s = "LVIII"

**Output:** 58

**Explanation:** L = 50, V= 5, III = 3.

**Example 3:**

**Input:** s = "MCMXCIV"

**Output:** 1994

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

class Solution {

    public int romanToInt(String s) {

        int n = s.length();

        int result = 0;

        Map<Character, Integer> romanToDecimal = new HashMap<>();

        romanToDecimal.put('I',1);

        romanToDecimal.put('V',5);

        romanToDecimal.put('X',10);

        romanToDecimal.put('L',50);

        romanToDecimal.put('C',100);

        romanToDecimal.put('D',500);

        romanToDecimal.put('M',1000);

        for(int i = 0 ; i < n ; i++)

        {

            char currentChar = s.charAt(i);

            int currentValue = romanToDecimal.get(currentChar);

            if(i + 1 < n)

            {

                char nextChar = s.charAt(i+1);

                int nextValue = romanToDecimal.get(nextChar);

                if(currentValue < nextValue)

                {

                    result = result - currentValue;

                }

                else if(currentValue > nextValue)

                {

                    result = result + currentValue;

                }

                else

                  result = result + currentValue;

            }

             else

                  result = result + currentValue;

        }

        return result;

    }

}