Q1class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

class Solution {

public ListNode detectAndRemoveLoop(ListNode head) {

ListNode slow = head;

ListNode fast = head;

boolean hasLoop = false;

// Find the meeting point of slow and fast pointers

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) {

hasLoop = true;

break;

}

}

// If there is no loop, return the original list

if (!hasLoop) {

return head;

}

// Move the slow pointer to the head and start moving both pointers at the same pace

slow = head;

while (slow.next != fast.next) {

slow = slow.next;

fast = fast.next;

}

// Unlink the last node which is forming the loop

fast.next = null;

return head;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

ListNode head = new ListNode(1);

ListNode node2 = new ListNode(3);

ListNode node3 = new ListNode(4);

head.next = node2;

node2.next = node3;

node3.next = node2; // Creating a loop

// Remove the loop from the linked list

head = solution.detectAndRemoveLoop(head);

// Print the resulting linked list

ListNode current = head;

while (current != null) {

System.out.print(current.val + " ");

current = current.next;

}

}

}

Q2.class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

class Solution {

public ListNode addOne(ListNode head) {

// Reverse the linked list

head = reverseLinkedList(head);

// Add 1 to the reversed number

int carry = 1;

ListNode current = head;

while (current != null) {

int sum = current.val + carry;

current.val = sum % 10;

carry = sum / 10;

if (carry == 0) {

break;

}

current = current.next;

}

// If there is still a carry, create a new node

// and append it to the end of the linked list

if (carry > 0) {

ListNode newNode = new ListNode(carry);

current.next = newNode;

}

// Reverse the linked list again to get the final result

head = reverseLinkedList(head);

return head;

}

private ListNode reverseLinkedList(ListNode head) {

ListNode prev = null;

ListNode current = head;

while (current != null) {

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

ListNode head = new ListNode(4);

ListNode node2 = new ListNode(5);

ListNode node3 = new ListNode(6);

head.next = node2;

node2.next = node3;

// Add 1 to the linked list

head = solution.addOne(head);

// Print the resulting linked list

ListNode current = head;

while (current != null) {

System.out.print(current.val);

current = current.next;

}

}

}

Q3.class Node {

int data;

Node next;

Node bottom;

Node(int data) {

this.data = data;

this.next = null;

this.bottom = null;

}

}

class Solution {

public Node flatten(Node head) {

if (head == null || head.next == null) {

return head;

}

// Merge the first two lists

head = mergeLists(head, head.next);

// Flatten the remaining lists

head = flatten(head);

return head;

}

private Node mergeLists(Node a, Node b) {

if (a == null) {

return b;

}

if (b == null) {

return a;

}

Node result;

if (a.data < b.data) {

result = a;

result.bottom = mergeLists(a.bottom, b);

} else {

result = b;

result.bottom = mergeLists(a, b.bottom);

}

result.next = null;

return result;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

Node head = new Node(5);

Node node2 = new Node(10);

Node node3 = new Node(19);

Node node4 = new Node(28);

head.next = node2;

node2.next = node3;

node3.next = node4;

head.bottom = new Node(7);

head.bottom.bottom = new Node(8);

head.bottom.bottom.bottom = new Node(30);

node2.bottom = new Node(20);

node3.bottom = new Node(22);

node3.bottom.bottom = new Node(50);

node4.bottom = new Node(35);

node4.bottom.bottom = new Node(40);

node4.bottom.bottom.bottom = new Node(45);

// Flatten the linked list

head = solution.flatten(head);

// Print the flattened linked list

Node current = head;

while (current != null) {

System.out.print(current.data + " -> ");

current = current.bottom;

}

System.out.println("null");

}

}

Q4.class Node {

int data;

Node next, arb;

Node(int data) {

this.data = data;

next = arb = null;

}

}

class Solution {

public Node copyRandomList(Node head) {

if (head == null) {

return null;

}

Map<Node, Node> map = new HashMap<>();

// Create a copy of each node and store it in the map

Node current = head;

while (current != null) {

map.put(current, new Node(current.data));

current = current.next;

}

// Adjust the next and arb pointers of the copied nodes

current = head;

while (current != null) {

Node copyNode = map.get(current);

copyNode.next = map.get(current.next);

copyNode.arb = map.get(current.arb);

current = current.next;

}

return map.get(head);

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

Node head = new Node(1);

Node node2 = new Node(2);

Node node3 = new Node(3);

Node node4 = new Node(4);

head.next = node2;

node2.next = node3;

node3.next = node4;

head.arb = node2;

node2.arb = node4;

// Copy the linked list

Node copiedHead = solution.copyRandomList(head);

// Print the copied linked list

Node current = copiedHead;

while (current != null) {

System.out.print(current.data + " ");

current = current.next;

}

System.out.println();

}

}

Q5.class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

class Solution {

public ListNode oddEvenList(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode oddHead = head;

ListNode evenHead = head.next;

ListNode odd = oddHead;

ListNode even = evenHead;

while (even != null && even.next != null) {

odd.next = even.next;

odd = odd.next;

even.next = odd.next;

even = even.next;

}

odd.next = evenHead;

return oddHead;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

head.next.next.next = new ListNode(4);

head.next.next.next.next = new ListNode(5);

// Reorder the linked list

ListNode reorderedList = solution.oddEvenList(head);

// Print the reordered linked list

ListNode current = reorderedList;

while (current != null) {

System.out.print(current.val + " ");

current = current.next;

}

System.out.println();

}

}

Q6.class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

class Solution {

public ListNode leftShift(ListNode head, int k) {

if (head == null || head.next == null || k == 0) {

return head;

}

int length = 0;

ListNode current = head;

while (current.next != null) {

current = current.next;

length++;

}

length++;

k %= length;

if (k == 0) {

return head;

}

int shiftIndex = length - k - 1;

ListNode kthNode = head;

while (shiftIndex > 0) {

kthNode = kthNode.next;

shiftIndex--;

}

ListNode newHead = kthNode.next;

kthNode.next = null;

current.next = head;

return newHead;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

ListNode head = new ListNode(2);

head.next = new ListNode(4);

head.next.next = new ListNode(7);

head.next.next.next = new ListNode(8);

head.next.next.next.next = new ListNode(9);

int k = 3;

// Left-shift the linked list

ListNode shiftedList = solution.leftShift(head, k);

// Print the shifted linked list

ListNode current = shiftedList;

while (current != null) {

System.out.print(current.val + " ");

current = current.next;

}

System.out.println();

}

}

Q7import java.util.\*;

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

class Solution {

public int[] nextLargerNodes(ListNode head) {

if (head == null) {

return new int[0];

}

List<Integer> result = new ArrayList<>();

Deque<Integer> stack = new ArrayDeque<>();

// Traverse the linked list in reverse order

ListNode current = reverse(head);

while (current != null) {

// Pop nodes with smaller values from the stack

while (!stack.isEmpty() && stack.peek() <= current.val) {

stack.pop();

}

// Set the next greater value for the current node

result.add(stack.isEmpty() ? 0 : stack.peek());

// Push the current node into the stack

stack.push(current.val);

current = current.next;

}

// Reverse the resulting array

Collections.reverse(result);

// Convert the list to an array

int[] answer = new int[result.size()];

for (int i = 0; i < result.size(); i++) {

answer[i] = result.get(i);

}

return answer;

}

// Helper function to reverse a linked list

private ListNode reverse(ListNode head) {

ListNode prev = null;

ListNode current = head;

while (current != null) {

ListNode next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

ListNode head = new ListNode(1);

head.next = new ListNode(7);

head.next.next = new ListNode(5);

head.next.next.next = new ListNode(3);

head.next.next.next.next = new ListNode(6);

head.next.next.next.next.next = new ListNode(4);

// Find the next greater nodes

int[] answer = solution.nextLargerNodes(head);

// Print the result

for (int i = 0; i < answer.length; i++) {

System.out.print(answer[i] + " ");

}

System.out.println();

}

}

Q8import java.util.\*;

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

class Solution {

public ListNode removeZeroSumSublists(ListNode head) {

// Create a dummy node

ListNode dummy = new ListNode(0);

dummy.next = head;

// Traverse the linked list and maintain a running sum

ListNode current = dummy;

int sum = 0;

Map<Integer, ListNode> map = new HashMap<>();

while (current != null) {

sum += current.val;

if (map.containsKey(sum)) {

// Remove consecutive nodes that sum to 0

ListNode start = map.get(sum).next;

int tempSum = sum + start.val;

while (tempSum != sum) {

map.remove(tempSum);

start = start.next;

tempSum += start.val;

}

map.get(sum).next = start.next;

} else {

map.put(sum, current);

}

current = current.next;

}

return dummy.next;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Create the linked list

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(-3);

head.next.next.next = new ListNode(3);

head.next.next.next.next = new ListNode(1);

// Remove zero sum subsequences

ListNode result = solution.removeZeroSumSublists(head);

// Print the result

while (result != null) {

System.out.print(result.val + " ");

result = result.next;

}

System.out.println();

}

}