**ASSIGNMENT-XIV**

**Question 1**

Given a linked list of **N** nodes such that it may contain a loop.

A loop here means that the last node of the link list is connected to the node at position X(1-based index). If the link list does not have any loop, X=0.

Remove the loop from the linked list, if it is present, i.e. unlink the last node which is forming the loop.

**Example 1:**

Input:  
N = 3  
value[] = {1,3,4}  
X = 2  
Output:1  
Explanation:The link list looks like  
1 -> 3 -> 4  
 ^ |  
 |\_\_\_\_|  
A loop is present. If you remove it  
successfully, the answer will be 1.  
**Example 2:**

Input:  
N = 4  
value[] = {1,8,3,4}  
X = 0  
Output:1  
Explanation:The Linked list does not  
contains any loop.  
**Example 3:**

**I**nput:

N = 4

value[] = {1,2,3,4}

X = 1

Output:1

Explanation:The link list looks like

1 -> 2 -> 3 -> 4

^ |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

A loop is present.

If you remove it successfully,

the answer will be 1.

**Ans:** class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

def removeLoop(head):

if head is None or head.next is None:

return 0

slowPtr = head

fastPtr = head

loopExists = False

while fastPtr.next and fastPtr.next.next:

slowPtr = slowPtr.next

fastPtr = fastPtr.next.next

if slowPtr == fastPtr:

loopExists = True

break

if not loopExists:

return 0

slowPtr = head

while slowPtr != fastPtr:

slowPtr = slowPtr.next

fastPtr = fastPtr.next

while fastPtr.next != slowPtr:

fastPtr = fastPtr.next

fastPtr.next = None

return 1

**Question 2**

A number **N** is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

**Example 1:**

Input:  
LinkedList: 4->5->6  
Output:457  
  
**Example 2:**

Input:

LinkedList: 1->2->3

Output:124

**Ans:** Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

def addOne(head):

if head is None:

return head

prev = None

curr = head

while curr:

nextNode = curr.next

curr.next = prev

prev = curr

curr = nextNode

head = prev

carry = 1

curr = head

prev = None

while curr:

sum = curr.data + carry

carry = sum // 10

curr.data = sum % 10

prev = curr

curr = curr.next

class

if carry == 1:

prev.next = Node(1)

prev = None

curr = head

while curr:

nextNode = curr.next

curr.next = prev

prev = curr

curr = nextNode

head = prev

return head

**Question 3**

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:(i) a **next** pointer to the next node,(ii) a **bottom** pointer to a linked list where this node is head.Each of the sub-linked-list is in sorted order.Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order. **Note:** The flattened list will be printed using the bottom pointer instead of next pointer.

**Example 1:**

Input:  
5 -> 10 -> 19 -> 28  
| | | |  
7 20 22 35  
| | |  
8 50 40  
| |  
30 45  
Output: 5-> 7-> 8- > 10 -> 19-> 20->  
22-> 28-> 30-> 35-> 40-> 45-> 50.  
Explanation:  
The resultant linked lists has every  
node in a single level.(Note:| represents the bottom pointer.)  
  
**Example 2:**

Input:

5 -> 10 -> 19 -> 28

| |

7 22

| |

8 50

|

30

Output: 5->7->8->10->19->22->28->30->50

Explanation:

The resultant linked lists has every

node in a single level.

(Note:| represents the bottom pointer.)

**Ans:** class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

self.bottom = None

def merge(a, b):

if a is None:

return b

if b is None:

return a

result = None

if a.data <= b.data:

result = a

result.bottom = merge(a.bottom, b)

else:

result = b

result.bottom = merge(a, b.bottom)

result.next = None

return result

def flatten(head):

if head is None or head.next is None:

return head

head.next = flatten(head.next)

head = merge(head, head.next)

return head

def printList(head):

curr = head

while curr:

print(curr.data, end=" ")

curr = curr.bottom

print()

# Example usage:

head = Node(5)

head.next = Node(10)

head.next.next = Node(19)

head.next.next.next = Node(28)

head.bottom = Node(7)

head.bottom.bottom = Node(8)

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

head.next.bottom = Node(20)

head.next.next.bottom = Node(22)

head.next.next.next.bottom = Node(35)

head.next.next.next.bottom.bottom = Node(50)

head.next.next.next.next = Node(40)

head.next.next.next.next.bottom = Node(45)

head = flatten(head)

printList(head)

**Question 4**

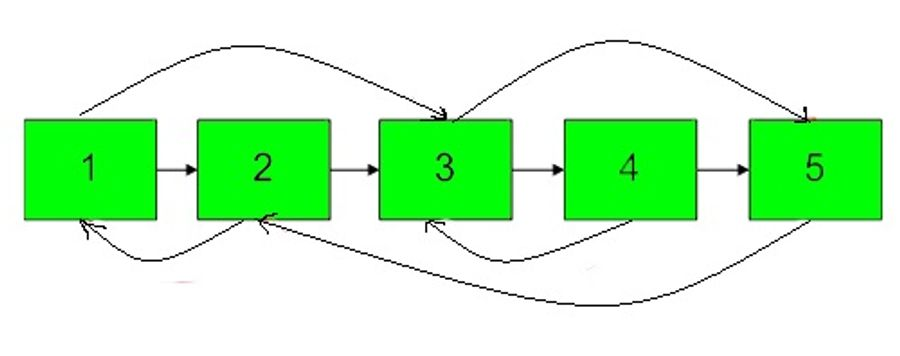
You are given a special linked list with **N** nodes where each node has a next pointer pointing to its next node. You are also given **M** random pointers, where you will be given **M** number of pairs denoting two nodes **a** and **b** **i.e. a->arb = b** (arb is pointer to random node)**.**

Construct a copy of the given list. The copy should consist of exactly **N** new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes **X** and **Y** in the original list, where **X.arb** **-->** **Y**, then for the corresponding two nodes **x** and **y** in the copied list, **x.arb --> y.**

Return the head of the copied linked list.

**Note** :- The diagram isn't part of any example, it just depicts an example of how the linked list may look like.



**Example 1:**

Input:  
N = 4, M = 2  
value = {1,2,3,4}  
pairs = {{1,2},{2,4}}  
Output:1  
Explanation:In this test case, there  
are 4 nodes in linked list. Among these  
4 nodes, 2 nodes have arbitrary pointer  
set, rest two nodes have arbitrary pointer  
as NULL. Second line tells us the value  
of four nodes. The third line gives the  
information about arbitrary pointers.  
The first node arbitrary pointer is set to  
node 2. The second node arbitrary pointer  
is set to node 4.

**Example 2:**

Input:

N = 4, M = 2

value[] = {1,3,5,9}

pairs[] = {{1,1},{3,4}}

Output:1

Explanation:In the given testcase ,

applying the method as stated in the

above example, the output will be 1.

**Ans:** class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

self.random = None

def copyRandomList(head):

if head is None:

return None

mapping = {}

curr = head

while curr:

mapping[curr] = Node(curr.data)

curr = curr.next

curr = head

while curr:

new\_node = mapping[curr]

new\_node.next = mapping.get(curr.next)

new\_node.random = mapping.get(curr.random)

curr = curr.next

return mapping[head]

head = Node(1)

head.next = Node(2)

head.next.next = Node(3)

head.next.next.next = Node(4)

head.random = head.next.next

head.next.random = head.next.next.next

copied\_head = copyRandomList(head)

curr = copied\_head

while curr:

print("Node:", curr.data)

if curr.random:

print("Random Pointer:", curr.random.data)

else:

print("Random Pointer: None")

print("---")

curr = curr.next

**Question 5**

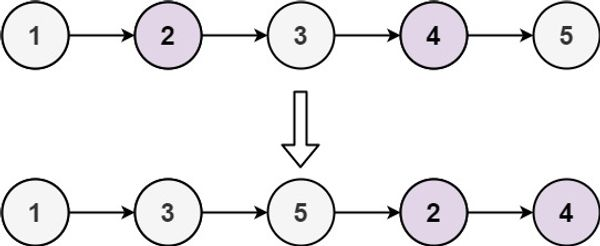
Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in O(1) extra space complexity and O(n) time complexity.

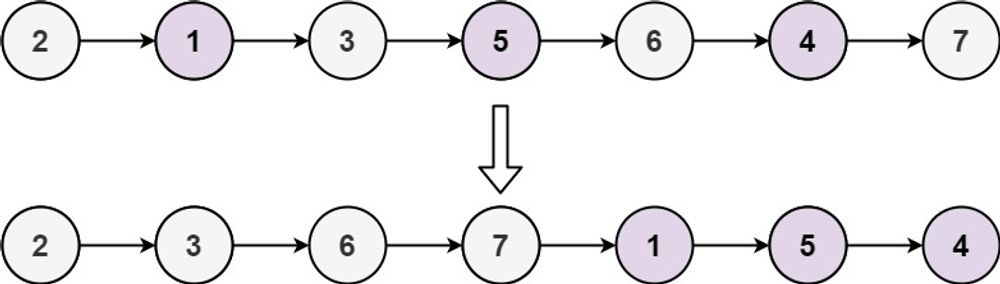
**Example 1:**



Input: head = [1,2,3,4,5]

Output: [1,3,5,2,4]

**Example 2:**



Input: head = [2,1,3,5,6,4,7]

Output: [2,3,6,7,1,5,4]

**Ans:** class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def oddEvenList(head):

if head is None or head.next is None:

return head

oddHead = odd = head

evenHead = even = head.next

while even and even.next:

odd.next = even.next

odd = odd.next

even.next = odd.next

even = even.next

odd.next = evenHead

return oddHead

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

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

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

reordered\_head = oddEvenList(head)

curr = reordered\_head

while curr:

print(curr.val, end=" ")

curr = curr.next

**Question 6**

Given a singly linked list of size **N**. The task is to **left-shift** the linked list by **k** nodes, where **k** is a given positive integer smaller than or equal to length of the linked list.

**Example 1:**

Input:  
N = 5  
value[] = {2, 4, 7, 8, 9}  
k = 3  
Output:8 9 2 4 7  
Explanation:Rotate 1:4 -> 7 -> 8 -> 9 -> 2  
Rotate 2: 7 -> 8 -> 9 -> 2 -> 4  
Rotate 3: 8 -> 9 -> 2 -> 4 -> 7  
  
**Example 2:**

Input:

N = 8

value[] = {1, 2, 3, 4, 5, 6, 7, 8}

k = 4

Output:5 6 7 8 1 2 3 4

**Ans:** class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def rotateLeft(head, k):

if head is None or head.next is None or k == 0:

return head

length = 0

curr = head

while curr:

length += 1

curr = curr.next

k = k % length

if k == 0:

return head

slow = fast = head

for \_ in range(k):

fast = fast.next

while fast.next:

slow = slow.next

fast = fast.next

newHead = slow.next

slow.next = None

fast.next = head

return newHead

head = ListNode(2)

head.next = ListNode(4)

head.next.next = ListNode(7)

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

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

k = 3

rotated\_head = rotateLeft(head, k)

curr = rotated\_head

while curr:

print(curr.val, end=" ")

curr = curr.next

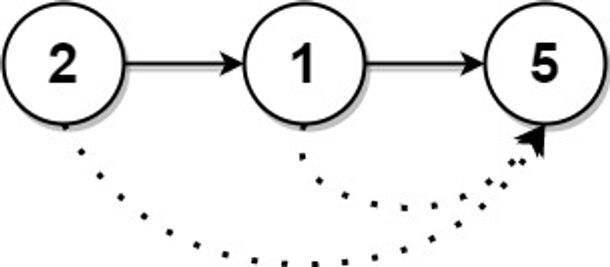
**Question 7**

You are given the head of a linked list with n nodes.

For each node in the list, find the value of the **next greater node**. That is, for each node, find the value of the first node that is next to it and has a **strictly larger** value than it.

Return an integer array answer where answer[i] is the value of the next greater node of the ith node (**1-indexed**). If the ith node does not have a next greater node, set answer[i] = 0.

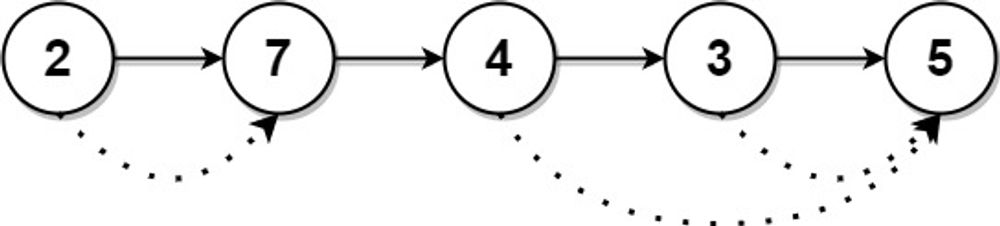
**Example 1:**



Input: head = [2,1,5]

Output: [5,5,0]

**Example 2:**



Input: head = [2,7,4,3,5]

Output: [7,0,5,5,0]

**Ans:** class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def nextLargerNodes(head):

# Convert the linked list to an array

arr = []

curr = head

while curr:

arr.append(curr.val)

curr = curr.next

n = len(arr)

stack = []

ans = [0] \* n

for i in range(n - 1, -1, -1):

while stack and arr[i] >= arr[stack[-1]]:

stack.pop()

if stack:

ans[i] = arr[stack[-1]]

stack.append(i)

return ans

head = ListNode(2)

head.next = ListNode(7)

head.next.next = ListNode(4)

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

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

result = nextLargerNodes(head)

print(result)

**Question 8**

**Example:8**

Given the head of a linked list, we repeatedly delete consecutive sequences of nodes that sum to 0 until there are no such sequences.

After doing so, return the head of the final linked list. You may return any such answer.

(Note that in the examples below, all sequences are serializations of ListNode objects.)

**Example 1:**

Input: head = [1,2,-3,3,1]  
Output: [3,1]  
Note: The answer [1,2,1] would also be accepted.  
  
**Example 2:**

Input: head = [1,2,3,-3,4]  
Output: [1,2,4]  
  
**Example 3:**

Input: head = [1,2,3,-3,-2]

Output: [1]

**Ans:** class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def removeZeroSumSublists(head):

cum\_sum = 0

sum\_dict = {}

curr = head

while curr:

cum\_sum += curr.val

if cum\_sum == 0:

# If the cumulative sum is zero, ignore the current node

curr = curr.next

continue

if cum\_sum in sum\_dict:

sum\_dict[cum\_sum].next = curr.next

else:

sum\_dict[cum\_sum] = curr

curr = curr.next

stack = []

curr = head

while curr:

if curr.val in sum\_dict:

curr = curr.next

continue

stack.append(curr)

curr = curr.next

if not stack:

return None

new\_head = stack[0]

curr = new\_head

for i in range(1, len(stack)):

curr.next = stack[i]

curr = curr.next

curr.next = None

return new\_head

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

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

head.next.next.next.next = ListNode(-2)

result = removeZeroSumSublists(head)

curr = result

while curr:

print(curr.val, end=" ")

curr = curr.next