**Question 1: Loop**

class LinkedList:

# Function to initialize head

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

self.head = None

def detectAndRemoveLoop(self):

slow\_p = fast\_p = self.head

while(slow\_p and fast\_p and fast\_p.next):

slow\_p = slow\_p.next

fast\_p = fast\_p.next.next

# If slow\_p and fast\_p meet at some point then

# there is a loop

if slow\_p == fast\_p:

self.removeLoop(slow\_p)

# Return 1 to indicate that loop is found

return 1

# Return 0 to indicate that there is no loop

return 0

# Function to remove loop

# loop\_node --> pointer to one of the loop nodes

# head --> Pointer to the start node of the linked list

def removeLoop(self, loop\_node):

ptr1 = loop\_node

ptr2 = loop\_node

# Count the number of nodes in loop

k = 1

while(ptr1.next != ptr2):

ptr1 = ptr1.next

k += 1

# Fix one pointer to head

ptr1 = self.head

# And the other pointer to k nodes after head

ptr2 = self.head

for i in range(k):

ptr2 = ptr2.next

# Move both pointers at the same place

# they will meet at loop starting node

while(ptr2 != ptr1):

ptr1 = ptr1.next

ptr2 = ptr2.next

# Get pointer to the last node

while(ptr2.next != ptr1):

ptr2 = ptr2.next

# Set the next node of the loop ending node

# to fix the loop

ptr2.next = None

# Function to insert a new node at the beginning

def push(self, new\_data):

new\_node = Node(new\_data)

new\_node.next = self.head

self.head = new\_node

# Utility function to print the LinkedList

def printList(self):

temp = self.head

while(temp):

print(temp.data, end = ' ')

temp = temp.next

Q2. 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.

class Solution:

def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:

m=d=ListNode(0)

nums1=''

nums2=''

while l1:

nums1+=str(l1.val)

l1=l1.next

while l2:

nums2+=str(l2.val)

l2=l2.next

res=str(int(nums1[::-1])+int(nums2[::-1]))[::-1]

for i in res:

d.next=ListNode(i)

d=d.next

return m.next

Q3.

class LinkedList():

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

# head of list

self.head = None

# Utility function to insert a node at beginning of the

# linked list

def push(self, head\_ref, data):

# 1 & 2: Allocate the Node &

# Put in the data

new\_node = Node(data)

# Make next of new Node as head

new\_node.down = head\_ref

# 4. Move the head to point to new Node

head\_ref = new\_node

# 5. return to link it back

return head\_ref

def printList(self):

temp = self.head

while(temp != None):

print(temp.data, end=" ")

temp = temp.down

print()

# An utility function to merge two sorted linked lists

def merge(self, a, b):

# if first linked list is empty then second

# is the answer

if(a == None):

return b

# if second linked list is empty then first

# is the result

if(b == None):

return a

# compare the data members of the two linked lists

# and put the larger one in the result

result = None

if (a.data < b.data):

result = a

result.down = self.merge(a.down, b)

else:

result = b

result.down = self.merge(a, b.down)

result.right = None

return result

def flatten(self, root):

# Base Case

if(root == None or root.right == None):

return root

# recur for list on right

root.right = self.flatten(root.right)

# now merge

root = self.merge(root, root.right)

# return the root

# it will be in turn merged with its left

return root

Q4 copy to next

class Solution:

def copyRandomList(self, head: 'Optional[Node]') -> 'Optional[Node]':

oldtocopy = {None: None}

curr = head

while curr:

copy = Node(curr.val)

oldtocopy[curr] = copy

curr = curr.next

curr = head

while curr:

nxt = curr.next

oldtocopy[curr].next = oldtocopy[nxt]

ran = curr.random

oldtocopy[curr].random = oldtocopy[ran]

curr = curr.next

return oldtocopy[head]

Q5. Odd and even

class Solution:

def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:

if head == None or head.next == None or head.next.next == None:

return head

odd,even = head, head.next

pointer1,pointer2 = odd,even

prev = None

while(pointer1 != None and pointer2 != None):

pointer1.next = pointer2.next

prev = pointer1

pointer1 = pointer1.next

if pointer1 == None:

pointer2.next = None

else:

pointer2.next = pointer1.next

pointer2 = pointer2.next

if pointer1 == None:

prev.next = even

else:

pointer1.next = even

return odd

Q6 rotate by k

def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:

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

return head

# Obtener la longitud de la lista enlazada

length = 0

itr = head

while itr:

length += 1

itr = itr.next

# Obtener la cantidad de pasos necesarios para rotar

k = k % length

# Si k es igual a 0, no es necesario rotar

if k == 0:

return head

# Obtener el nodo en la posición (length - k)

itr = head

for i in range(length - k - 1):

itr = itr.next

# Hacer la rotación

new\_head = itr.next

itr.next = None

itr = new\_head

while itr.next:

itr = itr.next

itr.next = head

return new\_head

Q7 Next bigger

def nextLargerNodes(self, head: Optional[ListNode]) -> List[int]:

ans = []

stack = []

i = 0

curr = head

while(curr):

# just for the length of the linked list.

ans.append(0)

curr = curr.next

while(head):

while(stack and stack[-1][1] < head.val):

index, \_ = stack.pop()

ans[index] = head.val

stack.append([i, head.val])

i += 1

head = head.next

Q8

class Solution:

def removeZeroSumSublists(self, head: Optional[ListNode]) -> Optional[ListNode]:

dummy = ListNode(0,head)

pre = 0

dic = {0: dummy}

while head:

pre+=head.val

dic[pre] = head

head = head.next

head = dummy

pre = 0

while head:

pre+=head.val

head.next = dic[pre].next

head = head.next

return dummy.next