Program for n’th node from the end of a Linked List

# Simple Python3 program to find

# n'th node from end

class Node:

    def \_\_init\_\_(self, new\_data):

        self.data = new\_data

        self.next = None

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    # createNode and and make linked list

    def push(self, new\_data):

        new\_node = Node(new\_data)

        new\_node.next = self.head

        self.head = new\_node

    # Function to get the nth node from

    # the last of a linked list

    def printNthFromLast(self, n):

        temp = self.head # used temp variable

        length = 0

        while temp is not None:

            temp = temp.next

            length += 1

        # print count

        if n > length: # if entered location is greater

                       # than length of linked list

            print('Location is greater than the' +

                         ' length of LinkedList')

            return

        temp = self.head

        for i in range(0, length - n):

            temp = temp.next

        print(temp.data)

# Driver Code

llist = LinkedList()

llist.push(20)

llist.push(4)

llist.push(15)

llist.push(35)

llist.printNthFromLast(4)

# Detect loop in a linked list

Given a linked list, check if the linked list has loop or not. Below diagram shows a linked list with a loop.

[](https://www.geeksforgeeks.org/wp-content/uploads/2009/04/Linked-List-Loop.gif)

# Python program to detect loop

# in the linked list

# Node class

class Node:

    # Constructor to initialize

    # the node object

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    # Function to initialize head

    def \_\_init\_\_(self):

        self.head = 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 prit

    # the linked LinkedList

    def printList(self):

        temp = self.head

        while(temp):

            print (temp.data, end =" ")

            temp = temp.next

    def detectLoop(self):

         s = set()

         temp = self.head

         while (temp):

             # If we have already has

             # this node in hashmap it

             # means their is a cycle

             # (Because you we encountering

             # the node second time).

            if (temp in s):

                return True

            # If we are seeing the node for

            # the first time, insert it in hash

            s.add(temp)

            temp = temp.next

         return False

# Driver program for testing

llist = LinkedList()

llist.push(20)

llist.push(4)

llist.push(15)

llist.push(10)

# Create a loop for testing

llist.head.next.next.next.next = llist.head;

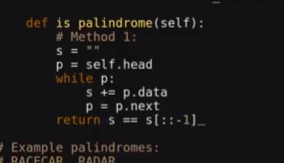
if( llist.detectLoop()):

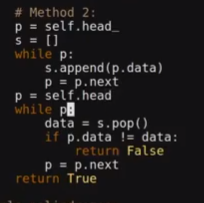
    print ("Loop found")

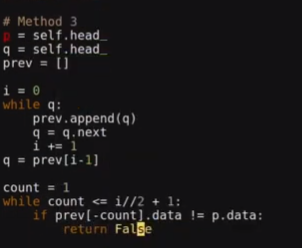
else :

    print ("No Loop ")

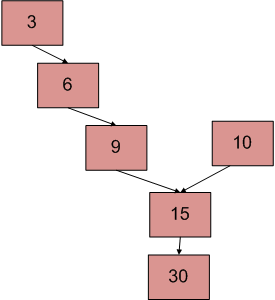
# Function to check if a singly linked list is palindrome







# Write a function to get the intersection point of two Linked Lists





def FindMergeNode(headA, headB):

curA = headA

curB = headB

while not curA == curB:

if curA.next is None:

curA = headB

else:

curA = curA.next

if curB.next is None:

curB = headA

else:

curB = curB.next

return curA.data

# Python Program to Find Intersection & Union of 2 Linked Lists

**class** Node:

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

self.data = data

self.next = None

**class** LinkedList:

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

self.head = None

**def** get\_prev\_node(self, ref\_node):

current = self.head

**while** (current **and** current.next != ref\_node):

current = current.next

**return** current

**def** duplicate(self):

copy = LinkedList()

current = self.head

**while** current:

node = Node(current.data)

copy.insert\_at\_end(node)

current = current.next

**return** copy

**def** insert\_at\_end(self, new\_node):

**if** self.head **is** None:

self.head = new\_node

**else**:

current = self.head

**while** current.next **is** **not** None:

current = current.next

current.next = new\_node

**def** remove(self, node):

prev\_node = self.get\_prev\_node(node)

**if** prev\_node **is** None:

self.head = self.head.next

**else**:

prev\_node.next = node.next

**def** display(self):

current = self.head

**while** current:

**print**(current.data, end = ' ')

current = current.next

**def** remove\_duplicates(llist):

current1 = llist.head

**while** current1:

current2 = current1.next

data = current1.data

**while** current2:

temp = current2

current2 = current2.next

**if** temp.data == data:

llist.remove(temp)

current1 = current1.next

**def** find\_union(llist1, llist2):

**if** llist1.head **is** None:

union = llist2.duplicate()

remove\_duplicates(union)

**return** union

**if** llist2.head **is** None:

union = llist1.duplicate()

remove\_duplicates(union)

**return** union

union = llist1.duplicate()

last\_node = union.head

**while** last\_node.next **is** **not** None:

last\_node = last\_node.next

llist2\_copy = llist2.duplicate()

last\_node.next = llist2\_copy.head

remove\_duplicates(union)

**return** union

**def** find\_intersection(llist1, llist2):

**if** (llist1.head **is** None **or** llist2.head **is** None):

**return** LinkedList()

intersection = LinkedList()

current1 = llist1.head

**while** current1:

current2 = llist2.head

data = current1.data

**while** current2:

**if** current2.data == data:

node = Node(data)

intersection.insert\_at\_end(node)

**break**

current2 = current2.next

current1 = current1.next

remove\_duplicates(intersection)

**return** intersection

a\_llist1 = LinkedList()

a\_llist2 = LinkedList()

data\_list = input('Please enter the elements in the first linked list: ').split()

**for** data **in** data\_list:

node = Node(int(data))

a\_llist1.insert\_at\_end(node)

data\_list = input('Please enter the elements in the second linked list: ').split()

**for** data **in** data\_list:

node = Node(int(data))

a\_llist2.insert\_at\_end(node)

union = find\_union(a\_llist1, a\_llist2)

intersection = find\_intersection(a\_llist1, a\_llist2)

**print**('Their union: ')

union.display()

**print**()

**print**('Their intersection: ')

intersection.display()

**print**()

### PYTHON: ADD TWO NUMBERS – LINKED LIST

Example:  
789 + 478 = 1267

Input: (9 -> 8 -> 7) + (8 -> 7 -> 4)  
Output: 7 -> 6 -> 2 -> 1

**class** Node:

**def** \_\_init\_\_(self, x, nextNode = None):

self.val = x

self.next = nextNode

**def** printList(l):

value = []

**while**(l):

value.append(l.val)

l = l.next

**print**(' -> '.join(map(str, value)))

**def** addTwoNumbers(l1, l2):

"""

:type l1: Node

:type l2: Node

:rtype: Node

"""

sum = l1.val + l2.val

carry = int(sum / 10)

l3 = Node(sum%10)

p1 = l1.next

p2 = l2.next

p3 = l3

**while**(p1 != None **or** p2 != None):

sum = carry + ( p1.val **if** p1 **else** 0) + ( p2.val **if** p2 **else** 0)

carry = int(sum/10)

p3.next = Node(sum % 10)

p3 = p3.next

p1 = p1.next **if** p1 **else** None

p2 = p2.next **if** p2 **else** None

**if**(carry > 0):

p3.next = Node(carry)

**return** l3

*#789*

l1 = Node(9, Node(8, Node(7)))

printList(l1)

*#478*

l2 = Node(8, Node(7, Node(4)))

printList(l2)

l3 = addTwoNumbers(l1, l2)

printList(l3)

**print**()

*#342*

l1 = Node(2, Node(4, Node(3)))

printList(l1)

*#465*

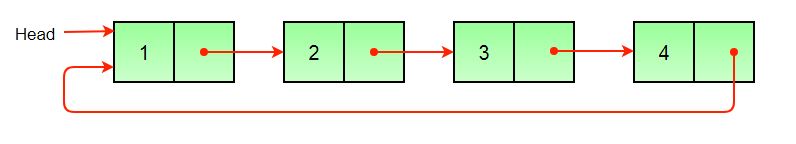
l2 = Node(5, Node(6, Node(4)))

printList(l2)

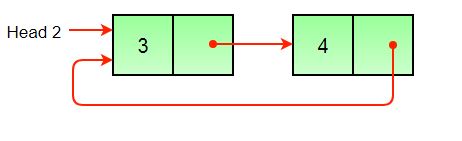
l3 = addTwoNumbers(l1, l2)

printList(l3)

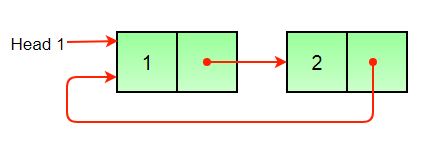
# Split a Circular Linked List into two halves



Original Linked List



Result Linked List 1



# Python program to split circular linked list into two halves

# A node structure

class Node:

    # Constructor to create a new node

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

# Class to create a new  Circular Linked list

class CircularLinkedList:

    # Constructor to create a empty circular linked list

    def \_\_init\_\_(self):

        self.head = None

    # Function to insert a node at the beginning of a

    # circular linked list

    def push(self, data):

        ptr1 = Node(data)

        temp = self.head

        ptr1.next = self.head

        # If linked list is not None then set the next of

        # last node

        if self.head is not None:

            while(temp.next != self.head):

                temp = temp.next

            temp.next = ptr1

        else:

            ptr1.next = ptr1 # For the first node

        self.head = ptr1

    # Function to print nodes in a given circular linked list

    def printList(self):

        temp = self.head

        if self.head is not None:

            while(True):

                print "%d" %(temp.data),

                temp = temp.next

                if (temp == self.head):

                    break

    # Function to split a list (starting with head) into

    # two lists. head1 and head2 are the head nodes of the

    # two resultant linked lists

    def splitList(self, head1, head2):

        slow\_ptr = self.head

        fast\_ptr = self.head

        if self.head is None:

            return

        # If htere are odd nodes in the circular list then

        # fast\_ptr->next becomes head and for even nodes

        # fast\_ptr->next->next becomes head

        while(fast\_ptr.next != self.head and

            fast\_ptr.next.next != self.head ):

            fast\_ptr = fast\_ptr.next.next

            slow\_ptr = slow\_ptr.next

        # If there are event elements in list then

        # move fast\_ptr

        if fast\_ptr.next.next == self.head:

            fast\_ptr = fast\_ptr.next

        # Set the head pointer of first half

        head1.head = self.head

        # Set the head pointer of second half

        if self.head.next != self.head:

            head2.head = slow\_ptr.next

        # Make second half circular

        fast\_ptr.next = slow\_ptr.next

        # Make first half circular

        slow\_ptr.next = self.head

# Driver program to test above functions

# Initialize lists as empty

head = CircularLinkedList()

head1 = CircularLinkedList()

head2 = CircularLinkedList()

head.push(12)

head.push(56)

head.push(2)

head.push(11)

print "Original Circular Linked List"

head.printList()

# Split the list

head.splitList(head1 , head2)

print "\nFirst Circular Linked List"

head1.printList()

print "\nSecond Circular Linked List"

head2.printList()

Merge Two Sorted linked list:

def merge\_lists(head1, head2):

if head1 is None:

return head2

if head2 is None:

return head1

# create dummy node to avoid additional checks in loop

s = t = node()

while not (head1 is None or head2 is None):

if head1.value < head2.value:

# remember current low-node

c = head1

# follow ->next

head1 = head1.next

else:

# remember current low-node

c = head2

# follow ->next

head2 = head2.next

# only mutate the node AFTER we have followed ->next

t.next = c

# and make sure we also advance the temp

t = t.next

t.next = head1 or head2

# return tail of dummy node

return s.next

Recursive algorithm:-

def merge\_lists(h1, h2):

if h1 is None:

return h2

if h2 is None:

return h1

if (h1.value < h2.value):

h1.next = merge\_lists(h1.next, h2)

return h1

else:

h2.next = merge\_lists(h2.next, h1)

return h2

Flattening a linked List:-

Add two numbers:-

Clone List with Random Pointer

XOR Linked List - A Memory Efficient Doubly Linked List

# Sorted insert for circular linked list

1) *Linked List is empty:*

a) since new\_node is the only node in CLL, make a self loop.

new\_node->next = new\_node;

b) change the head pointer to point to new node.

\*head\_ref = new\_node;

2) *New node is to be inserted just before the head node:*

(a) Find out the last node using a loop.

while(current->next != \*head\_ref)

current = current->next;

(b) Change the next of last node.

current->next = new\_node;

(c) Change next of new node to point to head.

new\_node->next = \*head\_ref;

(d) change the head pointer to point to new node.

\*head\_ref = new\_node;

3) *New node is to be inserted somewhere after the head:*

(a) Locate the node after which new node is to be inserted.

while ( current->next!= \*head\_ref &&

current->next->data data)

{ current = current->next; }

(b) Make next of new\_node as next of the located pointer

new\_node->next = current->next;

(c) Change the next of the located pointer

current->next = new\_node;

# Node class

class Node:

    # Constructor to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    # Function to initialize head

    def \_\_init\_\_(self):

        self.head = 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 linked LinkedList

    def printList(self):

        temp = self.head

        print temp.data,

        temp = temp.next

        while(temp != self.head):

            print temp.data,

            temp = temp.next

    """ function to insert a new\_node in a list in sorted way.

       Note that this function expects a pointer to head node

       as this can modify the head of the input linked list """

    def sortedInsert(self, new\_node):

        current = self.head

        # Case 1 of the above algo

        if current is None:

            new\_node.next = new\_node

            self.head = new\_node

        # Case 2 of the above algo

        elif (current.data >= new\_node.data):

            # If value is smaller than head's value then we

            # need to change next of last node

            while current.next != self.head :

                current = current.next

            current.next = new\_node

            new\_node.next = self.head

            self.head = new\_node

        # Case 3 of the above algo

        else:

            # Locate the node before the point of insertion

            while (current.next != self.head  and

                   current.next.data < new\_node.data):

                current = current.next

            new\_node.next = current.next

            current.next = new\_node

# Driver program to test the above function

#llist = LinkedList()

arr = [12, 56, 2, 11, 1, 90]

list\_size = len(arr)

# start with empty linked list

start = LinkedList()

# Create linked list from the array arr[]

# Created linked list will be 1->2->11->12->56->90

for i in range(list\_size):

    temp = Node(arr[i])

    start.sortedInsert(temp)

start.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Odd Even Linked Lists**

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

**Example 1:**

**Input:** 1->2->3->4->5->NULL

**Output:** 1->3->5->2->4->NULL

**Example 2:**

**Input:** 2->1->3->5->6->4->7->NULL

**Output:** 2->3->6->7->1->5->4->NULL

def oddEvenList(self, head: ListNode) -> ListNode:

dicts = collections.defaultdict(list)

i = 0

while head:

if i%2==0:

dicts['odd'].append(head)

else:

dicts['even'].append(head)

i+=1

head = head.next

dummy = ListNode()

cur = dummy

for key in ['odd', 'even']:

for node in dicts[key]:

cur.next = node

cur = cur.next

cur.next = None

return dummy.next\*\*

**Middle of the Linked List**

Given a non-empty, singly linked list with head node head, return a middle node of linked list.

If there are two middle nodes, return the second middle node.

**Example 1:**

**Input:** [1,2,3,4,5]

**Output:** Node 3 from this list (Serialization: [3,4,5])

The returned node has value 3. (The judge's serialization of this node is [3,4,5]).

Note that we returned a ListNode object ans, such that:

ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, and ans.next.next.next = NULL.

**Example 2:**

**Input:** [1,2,3,4,5,6]

**Output:** Node 4 from this list (Serialization: [4,5,6])

Since the list has two middle nodes with values 3 and 4, we return the second one.

counter = 0

curr = head

while curr:

curr = curr.next

counter += 1

mid = counter // 2

counter = 0

while head:

if counter == mid:

return head

head = head.next

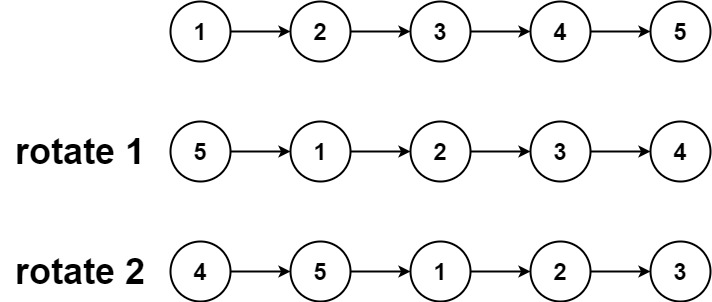
counter += 1

return None

**Rotate List**

Given the head of a linked list, rotate the list to the right by k places.

**Example 1:**



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

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

# Node class

class Node:

    # Constructor to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    # Function to initialize head

    def \_\_init\_\_(self):

        self.head = None

    # Function to insert a new node at the beginning

    def push(self, new\_data):

        # allocate node and put the data

        new\_node = Node(new\_data)

        # Make next of new node as head

        new\_node.next = self.head

        # move the head to point to the new Node

        self.head = new\_node

    # Utility function to print it the linked LinkedList

    def printList(self):

        temp = self.head

        while(temp):

            print temp.data,

            temp = temp.next

    # This function rotates a linked list counter-clockwise and

    # updates the head. The function assumes that k is smaller

    # than size of linked list. It doesn't modify the list if

    # k is greater than of equal to size

    def rotate(self, k):

        if k == 0:

            return

        # Let us understand the below code for example k = 4

        # and list = 10->20->30->40->50->60

        current = self.head

        # current will either point to kth or NULL after

        # this loop

        # current will point to node 40 in the above example

        count = 1

        while(count <k and current is not None):

            current = current.next

            count += 1

        # If current is None, k is greater than or equal

        # to count of nodes in linked list. Don't change

        # the list in this case

        if current is None:

            return

        # current points to kth node. Store it in a variable

        # kth node points to node 40 in the above example

        kthNode = current

        # current will point to lsat node after this loop

        # current will point to node 60 in above example

        while(current.next is not None):

            current = current.next

        # Change next of last node to previous head

        # Next of 60 is now changed to node 10

        current.next = self.head

        # Change head to (k + 1)th node

        # head is not changed to node 50

        self.head = kthNode.next

        # change next of kth node to NULL

        # next of 40 is not NULL

        kthNode.next = None

# Driver program to test above function

llist = LinkedList()

# Create a list 10->20->30->40->50->60

for i in range(60, 0, -10):

    llist.push(i)

print "Given linked list"

llist.printList()

llist.rotate(4)

print "\nRotated Linked list"

llist.printList()

**Next greater element in the Linked List**

Given a [linked list](http://www.geeksforgeeks.org/data-structures/linked-list/) **L** of integers, the task is to return a linked list of integers such that it contains [next greater element](http://www.geeksforgeeks.org/next-greater-element/) for each element in the given linked list. If there doesn’t any greater element for any element then insert **0** for it.

**Examples:**

***Input:****2->1->3->0->5****Output:****3->3->5->5->0*

***Input:****1->2->3****Output:****2->3->0*

# Definition for singly-linked list.

# class ListNode:

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

# self.val = val

# self.next = next

class Solution:

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

arr = []

while head != None:

arr.append(head.val)

head = head.next

ret = [0 for b in range(len(arr))]

stack = []

for i in range(len(arr)):

if len(stack) > 0:

while len(stack) > 0 and arr[stack[-1]] < arr[i]:

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

stack.pop()

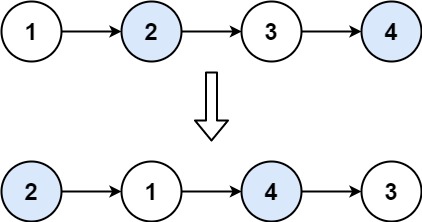
stack.append(i)

return ret

**Swap Nodes in Pairs**

Given a linked list, swap every two adjacent nodes and return its head.

**Example 1:**



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

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

class Solution:

def swapPairs(self, head: ListNode) -> ListNode:

p = ListNode(0,head)

prev = p

while head and head.next:

prev.next, head.next = head.next, head.next.next

prev.next.next = head

prev = head

head = head.next

return p.next

**flatten a linked list**

**Merge Two Sorted Lists**

Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

**Example 1:**



**Input:** l1 = [1,2,4], l2 = [1,3,4]

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

def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:

ptr = ListNode(0)

head = ptr

while l1 and l2:

if l1.val <= l2.val:

ptr.next = ListNode(l1.val)

l1 = l1.next

ptr = ptr.next

else:

ptr.next = ListNode(l2.val)

l2 = l2.next

ptr = ptr.next

while l1:

ptr.next = ListNode(l1.val)

l1 = l1.next

ptr = ptr.next

while l2:

ptr.next = ListNode(l2.val)

l2 = l2.next

ptr = ptr.next

return head.next

**Reverse Nodes in k-Group**

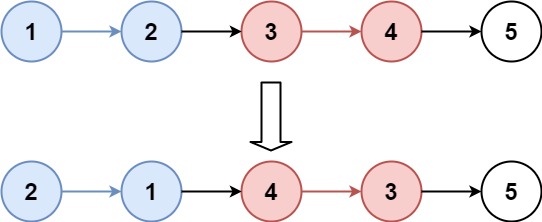
Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

*k* is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should remain as it is.

**Follow up:**

* Could you solve the problem in O(1) extra memory space?
* You may not alter the values in the list's nodes, only nodes itself may be changed.

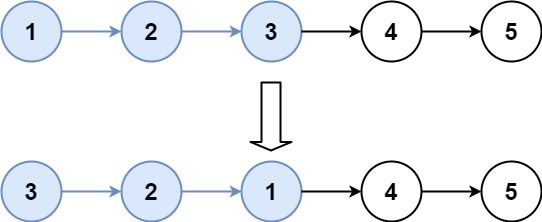
**Example 1:**



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

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

**Example 2:**



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

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

ime: O(n) - beats 97%  
Space: O(1) - beats 38%

Had to be verbose to keep this solution clear.  
The dummy variable enables us to avoid if clauses in the loop to position the head.

Pointers **prev**, **cur** and **nxt** are used to reverse the intervals.  
We need to rig the last node of the last reversed interval to the following node (whether it will be reversed or not). We use the pointer **first\_in\_interval** for that. Note that the first node in the interval will be the last after reversion.  
After that, if no more reversions are needed, we can return the list.

However, if another reversion takes place, the last rig we did will be invalidated (because the first node of the following interval will be the last after the reversion). Hence, we maintain a pointer **one\_before\_interval** that points to the node in the list located immediately before the last reversed interval. We use it to rig the last node from the previous reversed interval with the first node of the last reversed interval. We update this pointer with **first\_in\_interval**, right before **first\_in\_interval** moves to the next interval.

def reverseKGroup(self, head: ListNode, k: int) -> ListNode:

if head is None or k == 1:

return head

# count number of elements

cur = head

n\_elem = 0

while cur is not None:

n\_elem += 1

cur = cur.next

if n\_elem == 1:

return head

# determine number of swaps

n\_swaps = n\_elem // k

dummy = ListNode(val = 0, next = head)

prev = dummy

cur = head

one\_before\_interval = dummy

for i in range(n\_swaps):

first\_in\_interval = cur

for j in range(k):

nxt = cur.next

cur.next = prev

prev = cur

cur = nxt

# rig the first node from the original interval

# (i.e the last node in the current reversed interval)

# to the first following node (in interval or not)

# this gets overturned if more intervals will be reversed

first\_in\_interval.next = cur

# rig the last node from the previous interval to the

# NEW first node from the current reversed interval

one\_before\_interval.next = prev

one\_before\_interval = first\_in\_interval

return dummy.next

class Solution:

def reverseKGroup(self, head: ListNode, k: int) -> ListNode:

nodes = [head]

for node in nodes:

if node.next:

nodes.append(node.next)

if k == 1:

return head

for i in range(len(nodes) // k):

#left, right = k\*i, k\*(i+1)

Left = i

I = I + k

Right = i

nodes[left:right] = nodes[left:right][::-1]

for a, b in zip(nodes, nodes[1:]):

a.next = b

nodes[-1].next = None

return nodes[0]