Q.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.

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.

Input:

N = 4

value[] = {1,8,3,4}

X = 0

Output:1

Explanation:The Linked list does not

contains any loop.

Input:

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.

Q. 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.

Input:

LinkedList: 4->5->6

Output:457

Input:

LinkedList: 1->2->3

Output:124

Q. 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.

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.)

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.)

Q. 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.

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

Input:

N = 8

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

k = 4

Output:5 6 7 8 1 2 3 4

Q. 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.)

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

Output: [3,1]

Note: The answer [1,2,1] would also be accepted.

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

Output: [1,2,4]

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

Output: [1]

Ans: \*\*Solution 1: Removing Loop from Linked List\*\*

To remove a loop from a linked list, we can use Floyd's cycle-finding algorithm. This algorithm uses two pointers, a slow pointer and a fast pointer, to detect the presence of a loop in the linked list.

Here's the step-by-step approach to remove the loop:

1. Initialize the slow pointer and fast pointer to the head of the linked list.

2. Move the slow pointer by one step and the fast pointer by two steps until they meet at the same node or the fast pointer reaches the end of the list.

3. If the fast pointer reaches the end of the list, it means there is no loop in the linked list. Return the original linked list.

4. If the slow and fast pointers meet at the same node, it indicates the presence of a loop. Move the slow pointer back to the head of the list.

5. Move both the slow and fast pointers one step at a time until they meet again. The meeting point will be the node where the loop starts.

6. To remove the loop, set the next pointer of the node where the slow pointer is currently pointing to NULL.

Here's the implementation in Python:

```python

class Node:

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

self.data = data

self.next = None

def remove\_loop(head):

slow = head

fast = head

# Find the meeting point of slow and fast pointers

while fast and fast.next:

slow = slow.next

fast = fast.next.next

if slow == fast:

break

# If there is no loop, return the original linked list

if not fast or not fast.next:

return head

# Move slow pointer to the head and find the loop start point

slow = head

while slow.next != fast.next:

slow = slow.next

fast = fast.next

# Remove the loop

fast.next = None

return head

```

\*\*Complexity Analysis:\*\*

The time complexity of this solution is O(N), where N is the number of nodes in the linked list. The algorithm visits each node at most twice.

---

\*\*Solution 2: Adding 1 to a Number Represented as a Linked List\*\*

To add 1 to a number represented as a linked list, we can follow a simple approach. Starting from the rightmost digit, we add 1 to the digit. If the sum is less than 10, we update the digit and return the modified linked list. Otherwise, we set the digit to 0 and move to the next digit.

If we reach the leftmost digit and the sum is still greater than or equal to 10, we insert a new node at the beginning with a value of 1.

Here's the step-by-step approach to add 1 to the linked list:

1. Reverse the linked list to make it easier to add the digits.

2. Initialize a carry variable to 1 to add 1 to the rightmost digit.

3. Traverse the linked list and add the carry to each digit.

- If the sum is less than 10, update the digit and set the carry to 0.

- If the sum is 10 or greater, set the digit to (sum % 10) and carry to 1.

4. Reverse the modified linked list to restore the original order.

5. If the carry is still 1 after traversing the entire list, insert a new node at the beginning with a value of 1.

Here's the implementation in Python:

```python

class Node:

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

self.data = data

self.next = None

def add\_one(head):

# Reverse the linked list

prev = None

curr = head

while curr:

next\_node = curr.next

curr.next = prev

prev = curr

curr = next\_node

head = prev

# Add 1 to the linked list

carry = 1

curr = head

while curr:

sum = curr.data + carry

curr.data = sum % 10

carry = sum // 10

if carry == 0:

break

curr = curr.next

# Reverse the linked list again

prev = None

curr = head

while curr:

next\_node = curr.next

curr.next = prev

prev = curr

curr = next\_node

head = prev

# If carry is still 1, insert a new node at the beginning

if carry == 1:

new\_node = Node(1)

new\_node.next = head

head = new\_node

return head

```

\*\*Complexity Analysis:\*\*

The time complexity of this solution is O(N), where N is the number of digits in the linked list. We reverse the list twice, perform addition on each digit, and reverse the list again. Each operation takes linear time.

---

\*\*Solution 3: Flatten a Linked List\*\*

To flatten a linked list with sub-linked lists, we can use a modified merge sort algorithm. The idea is to merge two sorted linked lists at a time, starting from the first two lists, then merging the result with the next list, and so on.

Here's the step-by-step approach to flatten the linked list:

1. Define a merge function that merges two sorted linked lists and returns the merged list.

2. Initialize the result variable with the first sub-linked list.

3. Iterate over the remaining sub-linked lists and merge each list with the result using the merge function.

4. Update the result with the merged list.

5. Return the result.

Here's the implementation in Python:

```python

class Node:

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

self.data = data

self.next = None

self.bottom = None

def merge(a, b):

if not a:

return b

if not b:

return a

if a.data <= b.data:

result = a

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

else:

result = b

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

return result

def flatten(root):

if not root or not root.next:

return root

root.next = flatten(root.next)

root = merge(root, root.next)

return root

def print\_list(head):

curr = head

while curr:

print(curr.data, end=" ")

curr = curr.bottom

# 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.next.bottom = Node(20)

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

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

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

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

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

head = flatten(head)

print\_list(head)

```

\*\*Complexity Analysis:\*\*

The time complexity of this solution is O(N log M), where N is the total number of nodes in all the sub-linked lists, and M is the number of sub-linked lists. The algorithm merges two linked lists at a time, and the number of merges

depends on the height of the flattened tree. The height of the tree is log M, where M is the number of sub-linked lists.

---

\*\*Solution 4: Left Shift a Linked List\*\*

To left-shift a linked list by k nodes, we can follow a simple approach. Since k is a positive integer smaller than or equal to the length of the linked list, we can find the (k+1)th node from the end of the list and make it the new head of the list.

Here's the step-by-step approach to left-shift the linked list:

1. Calculate the length of the linked list.

2. If k is greater than or equal to the length of the linked list, set k to k % length to ensure k is within the valid range.

3. If k is 0, return the original linked list.

4. Initialize two pointers, slow and fast, to the head of the list.

5. Move the fast pointer k positions ahead of the slow pointer.

6. Move both the slow and fast pointers simultaneously until the fast pointer reaches the last node of the list.

7. Set the next pointer of the last node to the head of the list.

8. Set the head of the list to the next node of the slow pointer.

9. Set the next pointer of the slow pointer to None.

10. Return the modified linked list.

Here's the implementation in Python:

```python

class Node:

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

self.data = data

self.next = None

def left\_shift(head, k):

if not head or not head.next or k == 0:

return head

# Calculate the length of the linked list

length = 0

curr = head

while curr:

length += 1

curr = curr.next

# Adjust k if it's greater than or equal to the length

k = k % length

if k == 0:

return head

# Move fast pointer k positions ahead of slow pointer

slow = head

fast = head

for \_ in range(k):

fast = fast.next

# Move both pointers until fast reaches the last node

while fast.next:

slow = slow.next

fast = fast.next

# Update pointers to left-shift the list

fast.next = head

head = slow.next

slow.next = None

return head

```

\*\*Complexity Analysis:\*\*

The time complexity of this solution is O(N), where N is the number of nodes in the linked list. We need to traverse the list to calculate its length and then traverse it again to adjust the pointers and perform the left shift. The number of traversals is constant, so the overall complexity is linear in terms of the number of nodes.