Question 1

Given two linked list of the same size, the task is to create a new linked list using those linked lists. The condition is that the greater node among both linked list will be added to the new linked list.

Soln:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def merge_lists(l1, l2):
    dummy = ListNode()  # Dummy node to hold the result
    current = dummy  # Pointer to the current node in the result list

    while l1 and l2:
        if l1.val >= l2.val:
            current.next = ListNode(l1.val)
            l1 = l1.next
        else:
            current.next = ListNode(l2.val)
            l2 = l2.next
        current = current.next

    # Append the remaining nodes from the first list
    while l1:
        current.next = ListNode(l1.val)
        l1 = l1.next
        current = current.next

    # Append the remaining nodes from the second list
    while l2:
        current.next = ListNode(l2.val)
        l2 = l2.next
        current = current.next

    return dummy.next  # Return the head of the merged list
```

Question 2

Write a function that takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60

Soln:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def remove_duplicates(head):
    if not head:
        return head

    current = head

    while current.next:
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            current = current.next

    return head
```

Question 3:
Given a linked list of size N. The task is to reverse every k nodes (where k is an input to the function) in the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should be considered as a group and must be reversed
Soln:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverse_k_nodes(head, k):
    if not head or not head.next or k == 1:
        return head

    # Create a dummy node to serve as the previous node of the first group
    dummy = ListNode(0)
    dummy.next = head

    # Initialize pointers for the current group
    prev = dummy
    curr = head

    # Find the length of the linked list
    length = 0
    node = head
    while node:
        length += 1
        node = node.next

    # Reverse each group of k nodes
    while length >= k:
        for _ in range(k - 1):
            # Extract the next node to be reversed
            next_node = curr.next

            # Move the next node to the beginning of the group
            curr.next = next_node.next
            next_node.next = prev.next
            prev.next = next_node

        # Move the pointers for the next group
        prev = curr
        curr = prev.next
        length -= k

    return dummy.next
```

Question 4

Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

Soln:

```
class ListNode:

    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverse_alternate_k_nodes(head, k):
    if not head or not head.next or k == 1:
        return head

    # dummy node to serve as the previous node of the first group
    dummy = ListNode(0)
    dummy.next = head

    # Initialize pointers for the current group
    prev = dummy
    curr = head

    # Reverse every alternate group of k nodes
    reverse = True  # Flag to indicate if the group should be reversed
    while curr:
        count = 0
        group_start = curr
        prev_group_end = prev

        # Traverse k nodes or until the end of the list
        while curr and count < k:
            prev = curr

            curr = curr.next

            count += 1


        # Reverse the group if the flag is True

        if reverse:

            prev_group_end.next = prev

            while group_start != curr:

                next_node = group_start.next

                group_start.next = prev.next

                prev.next = group_start

                group_start = next_node

                prev = prev.next


        # Move the pointers for the next group

        prev = prev_group_end

        reverse = not reverse


    return dummy.next
```

Question 5

Given a linked list and a key to be deleted. Delete last occurrence of key from linked. The list may have duplicates.

Soln:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def delete_last_occurrence(head, key):
    if not head:
        return None

    # Find the last occurrence of the key
    prev = None
    last_occurrence = None
    current = head

    while current:
        if current.val == key:
            last_occurrence = current
        current = current.next

    # If the last occurrence is the head node, update the head
    if last_occurrence and last_occurrence == head:
        head = head.next
    else:
        # Traverse the list again to delete the last occurrence
        current = head
        while current:
            if current == last_occurrence:
                prev.next = current.next
                break
            prev = current
            current = current.next

    return head
```

Question 6

Given two sorted linked lists consisting of **N** and **M** nodes respectively. The task is to merge both of the lists (in place) and return the head of the merged list.

Examples:

Input: a: 5->10->15, b: 2->3->20

Output: 2->3->5->10->15->20

Input: a: 1->1, b: 2->4

Output: 1->1->2->4

Soln:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

```
def merge_sorted_lists(a, b):
    # Create a dummy node as the head of the merged list
    dummy = ListNode(0)
    current = dummy

    # Compare the values of the nodes and merge them
    while a and b:
        if a.val <= b.val:
            current.next = a
            a = a.next
        else:
            current.next = b
            b = b.next
        current = current.next

    # Append the remaining nodes from the list that still has elements
    if a:
        current.next = a
    if b:
        current.next = b

    return dummy.next
```

Question 7:

Given a Doubly Linked List, the task is to reverse the given Doubly Linked List.

Soln:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None


def reverse_doubly_linked_list(head):
    # Check if the list is empty or contains only one node
    if not head or not head.next:
        return head


    current = head
    while current:
        # Swap the prev and next pointers of the current node
        current.prev, current.next = current.next, current.prev
        # Move to the next node
        current = current.prev


    # The last node of the original list is now the head of the reversed list
    return head.prev
```

Question 8

Given a doubly linked list and a position. The task is to delete a node from given position in a doubly linked list.

Soln:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.prev = None
        self.next = None


def delete_node_at_position(head, position):
    # Check if the list is empty
    if not head:
        return None

    # Special case: Delete the head node
    if position == 0:
        head = head.next
        if head:
            head.prev = None
        return head

    current = head
```

```
count = 0
# Traverse to the node at the given position
while current and count < position:
    current = current.next
    count += 1

# Check if the position is out of range
if not current:
    return head

# Update the prev and next pointers of adjacent nodes
current.prev.next = current.next
if current.next:
    current.next.prev = current.prev

return head
```