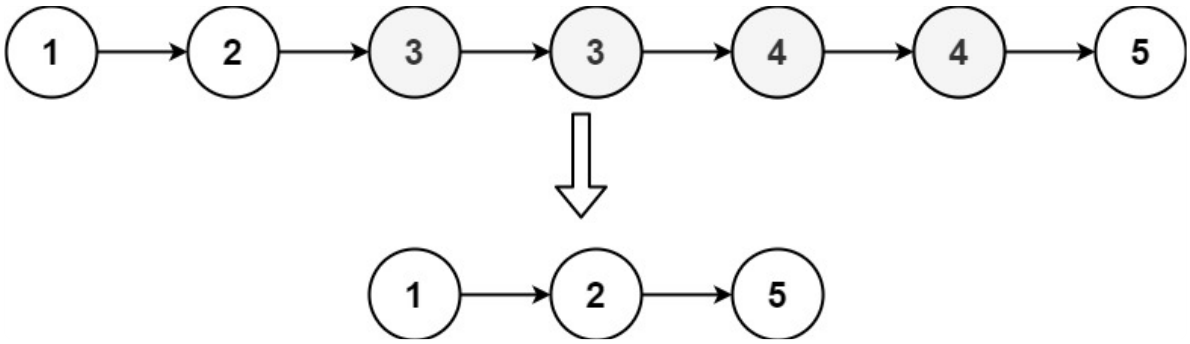
## Assignment Solutions | Linkedlist - 3 | Week 15

1. Given the `head` of a sorted linked list, *delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list*. Return *the linked list* ***sorted*** *as well*. [Leetcode 82]

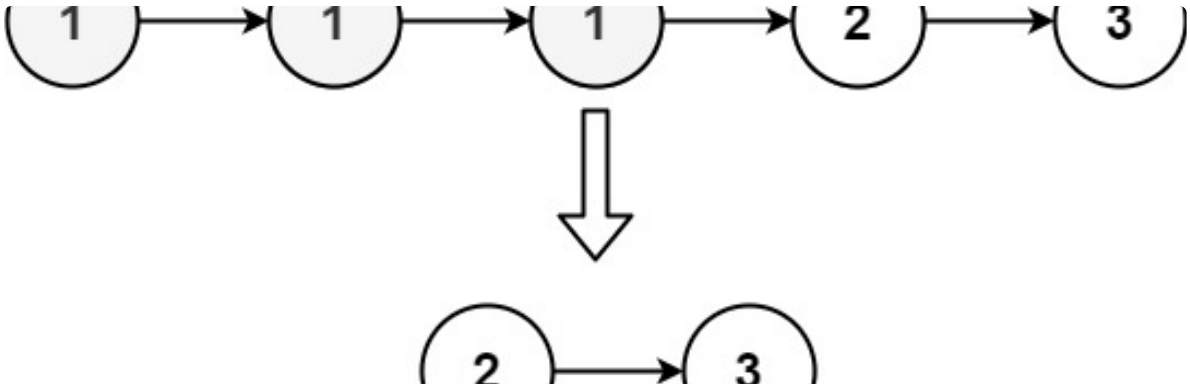
   **Example 1:**




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

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

   **Example 2:**




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

   **Output:** [2,3]

   **Constraints:**

   - The number of nodes in the list is in the range `[0, 300]`.

- `-100 <= Node.val <= 100`
- The list is guaranteed to be **sorted** in ascending order.

Solution:

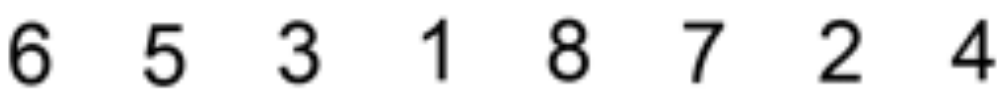
```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode *temp = dummy;
        while(head != NULL){
            if(head->next and head->val == head->next->val){
                while(head->next and head->val == head->next->val){
                    head = head->next;
                }
                temp->next = head->next;
            }
            else temp = temp->next;
                head = head->next;
        }
        return dummy->next;
    }
};
```

2. Given the `head` of a singly linked list, sort the list using **insertion sort**, and return *the sorted list's head*. [Leetcode 147]

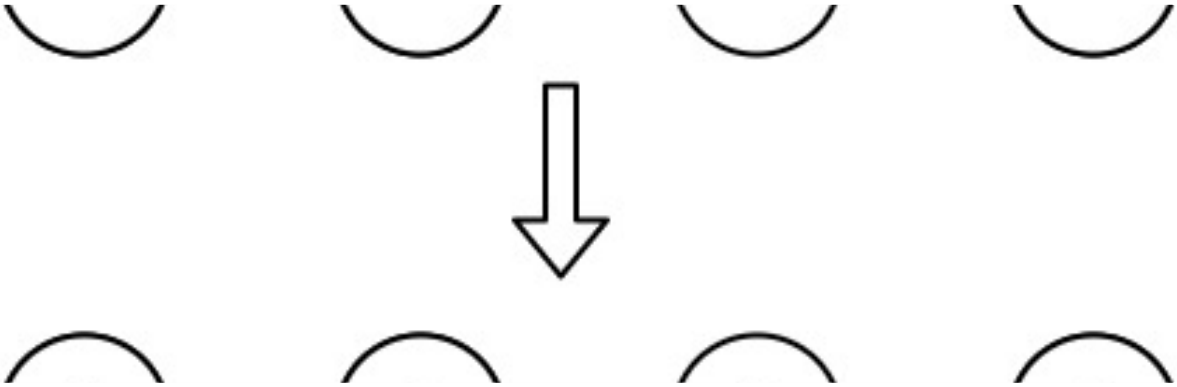
   The steps of the **insertion sort** algorithm:

   1. Insertion sort iterates, consuming one input element each repetition and growing a sorted output list.
   2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list and inserts it there.
   3. It repeats until no input elements remain.

   The following is a graphical example of the insertion sort algorithm. The partially sorted list (black) initially contains only the first element in the list. One element (red) is removed from the input data and inserted in-place into the sorted list with each iteration.



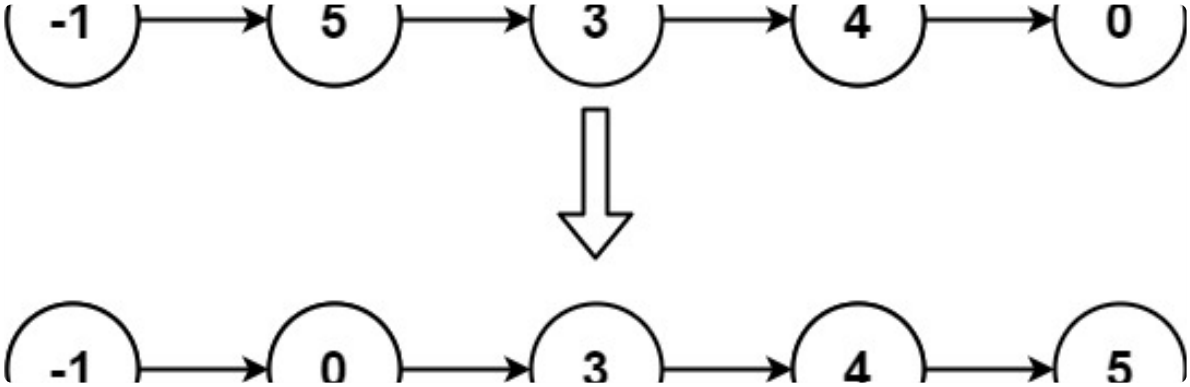
**Example 1:**



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

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

**Example 2:**




**Input:** head = [-1,5,3,4,0]

**Output:** [-1,0,3,4,5]

**Constraints:**

- The number of nodes in the list is in the range `[1, 5000]`.
- `-5000 <= Node.val <= 5000`

Solution:

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy -> next = head;
        ListNode *pre = dummy, *cur = head;
        while (cur) {
            if ((cur -> next) && (cur -> next -> val < cur -> val)) {
                while ((pre -> next) && (pre -> next -> val < cur -> next -> val)) {
                    pre = pre -> next;
                }
                ListNode* temp = pre -> next;
                pre -> next = cur -> next;
                cur -> next = cur -> next -> next;
                pre -> next -> next = temp;
                pre = dummy;
            }
            else {
                cur = cur -> next;
            }
        }
        return dummy -> next;
    }
```

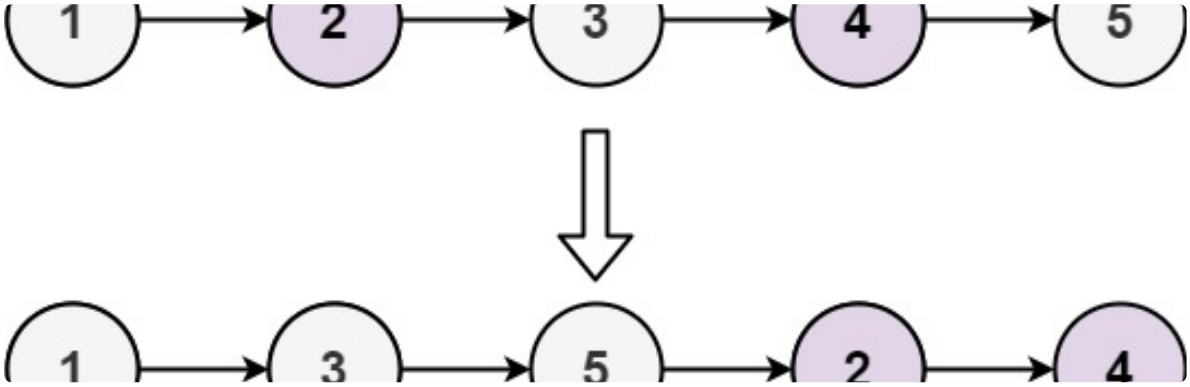
3. Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*. [Leetcode 328]

   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.

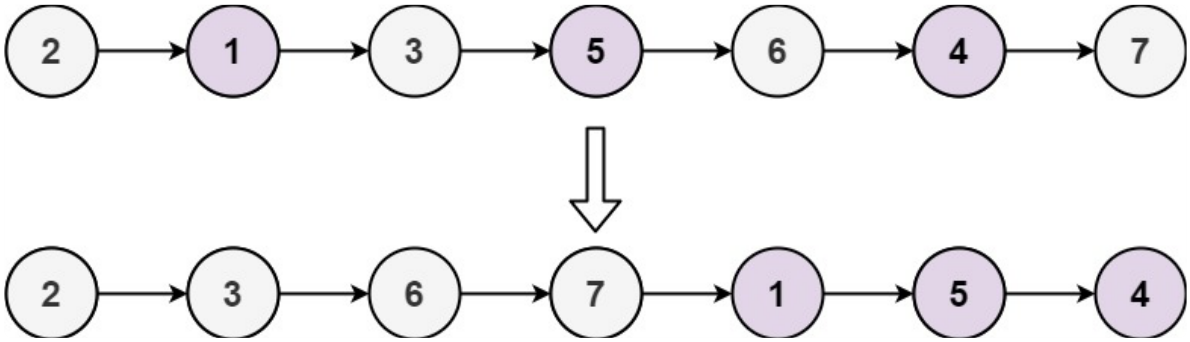
   **Example 1:**




   **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]

**Constraints:**

- The number of nodes in the linked list is in the range `[0, 104]`.
- `-1e6 <= Node.val <= 1e6`

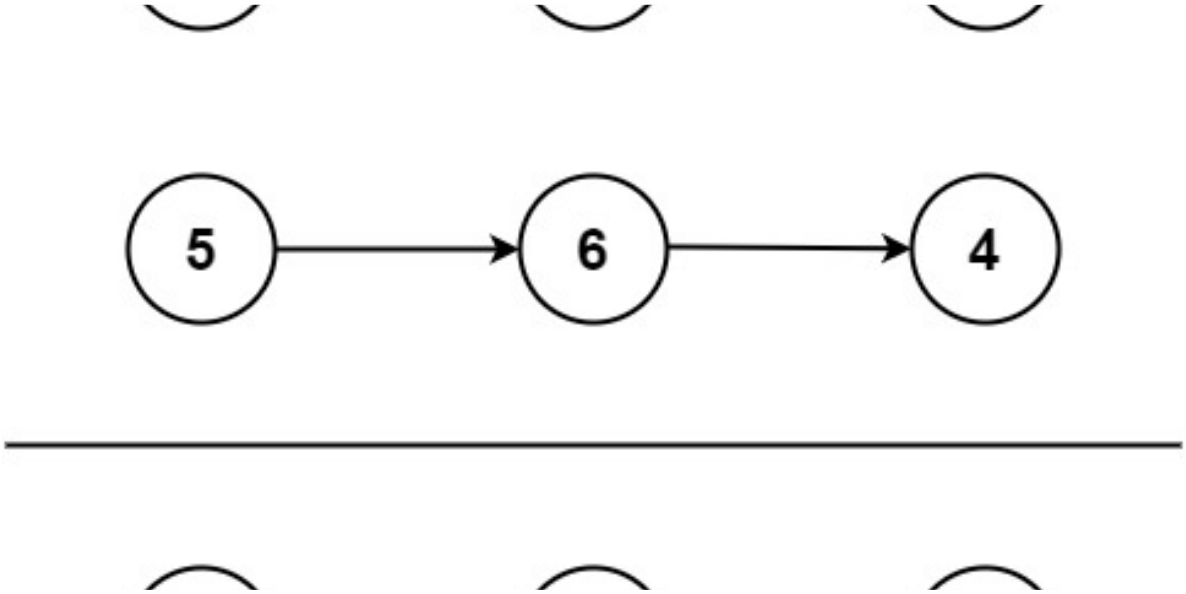
Solution :

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(!head or !head->next)return head;

        ListNode *odd=head;
        ListNode *eve=head->next;
        ListNode *newhead=eve;
        while(eve and eve->next){
            odd->next=eve->next;
            odd=odd->next;
            eve->next=odd->next;
            eve=eve->next;
        }
        odd->next=newhead;
        return head;
    }
};
```

4. You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. [Leetcode 2]

   You may assume the two numbers do not contain any leading zero, except the number 0 itself.

   **Example 1:**




**Input:** l1 = [2,4,3], l2 = [5,6,4]

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

**Example 2:**

**Input:** l1 = [0], l2 = [0]

**Output:** [0]

**Example 3:**

**Input:** l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

**Output:** [8,9,9,9,0,0,0,1]

**Constraints:**

- The number of nodes in each linked list is in the range `[1, 100]`.
- `0 <= Node.val <= 9`
- It is guaranteed that the list represents a number that does not have leading zeros.

Solution :

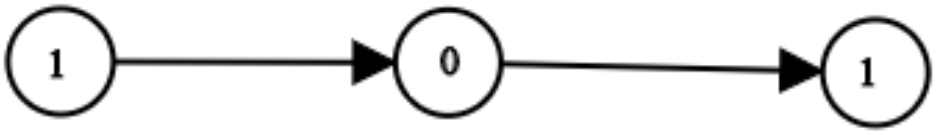
```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
       int carry = 0;
        if(!l1)return l2;
        if(!l2)return l1;

        ListNode *a = l1;
        ListNode *b = l2;
        ListNode *c = new ListNode (0);
        ListNode *head = c;
        while(a or b){
            int x = a ? a->val : 0;
            int y = b ? b->val : 0;
            int sum = x + y + carry;
            head->next = new ListNode(sum%10);
            carry = sum/10;
            head = head->next;
            if(a)a = a->next;
            if(b)b = b->next;
        }
        if(carry)head->next = new ListNode(carry);
        return c->next;
```

5. Given `head` which is a reference node to a singly-linked list. The value of each node in the linked list is either `0` or `1`. The linked list holds the binary representation of a number.

   Return the *decimal value* of the number in the linked list. [Leetcode 1290]

   The **most significant bit** is at the head of the linked list.

   **Example 1:**




   **Input:** head = [1,0,1]

   **Output:** 5

   **Explanation:** (101) in base 2 = (5) in base 10

   **Example 2:**

   **Input:** head = [0]

   **Output:** 0

   **Constraints:**

   - The Linked List is not empty.

- Number of nodes will not exceed `30`.
- Each node's value is either `0` or `1`.

Solution :

```
class Solution {
public:
    ListNode* reverse(ListNode *&head){
        ListNode *nex;
        ListNode *curr = head , *prev = NULL;
        while(curr){
           nex = curr->next;
           curr->next = prev;
           prev = curr;
            curr = nex;
        }
        head = prev;
        return head;
    }
    int getDecimalValue(ListNode* head) {
        head = reverse(head);
        int val = 1;
        int ans = 0;
        ListNode *tmp = head;
        while(tmp){
            ans+=val*tmp->val;
            tmp=tmp->next;
            val<<=1;
```

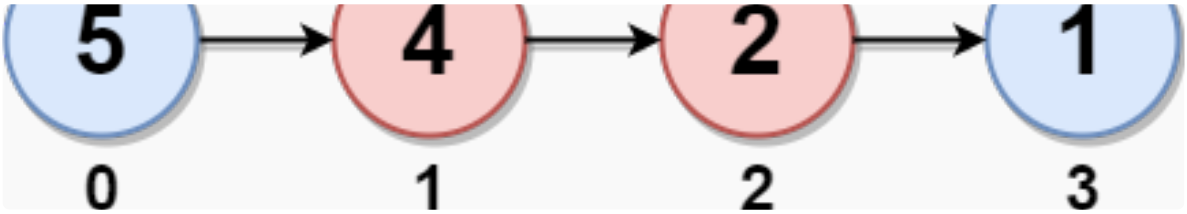
6. In a linked list of size `n`, where `n` is **even**, the `ith` node (**0-indexed**) of the linked list is known as the **twin** of the `(n-1-i)th` node, if `0 <= i <= (n / 2) - 1`. [Leetcode 2130]

- For example, if `n = 4`, then node `0` is the twin of node `3`, and node `1` is the twin of node `2`. These are the only nodes with twins for `n = 4`.

  The **twin sum** is defined as the sum of a node and its twin.

  Given the `head` of a linked list with even length, return *the **maximum twin sum** of the linked list*.

  **Example 1:**



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

**Output:** 6

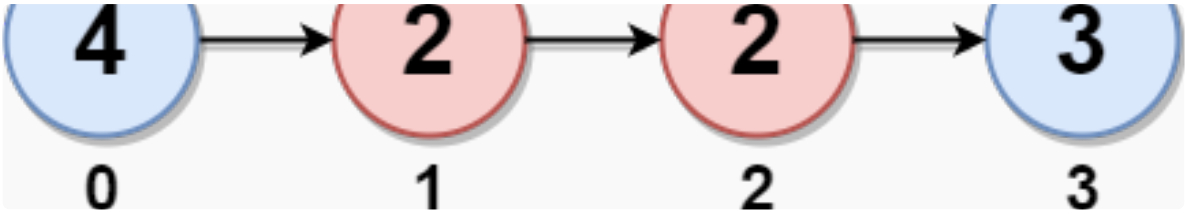
**Explanation:**

Nodes 0 and 1 are the twins of nodes 3 and 2, respectively. All have twin sum = 6.

There are no other nodes with twins in the linked list.

Thus, the maximum twin sum of the linked list is 6.

**Example 2:**




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

**Output:** 7

**Explanation:**

The nodes with twins present in this linked list are:

- Node 0 is the twin of node 3 having a twin sum of 4 + 3 = 7.
- Node 1 is the twin of node 2 having a twin sum of 2 + 2 = 4.

Thus, the maximum twin sum of the linked list is max(7, 4) = 7.

**Example 3:**


**Input:** head = [1,100000]

**Output:** 100001

**Explanation:**

There is only one node with a twin in the linked list having twin sum of 1 + 100000 = 100001.

**Constraints:**

- The number of nodes in the list is an **even** integer in the range `[2, 1e5]`.

- `1 <= Node.val <= 1e5`

Solution :

```
class Solution {
public:
ListNode *middle(ListNode *head){
    ListNode *fast = head->next;
    ListNode *slow = head;
    while(fast and fast->next){
    fast = fast->next->next;
    slow = slow->next;
    }
    return slow;
}
ListNode* reverse(ListNode *mid){
    ListNode *curr = mid;
    ListNode *prev = NULL;
    while(curr){
    ListNode *plus = curr->next;
    curr->next = prev;
    prev = curr;
    curr = plus;
    }
    return prev;
}
int pairSum(ListNode* head) {
    ListNode *mid = middle(head);
    int sum = 0;
    int maxi = 0;
    ListNode *p = reverse(mid);
    mid->next = NULL;
    ListNode *temp = head;
    while(temp and p){
    sum = p->val + temp->val;
    maxi = max(maxi , sum);
    p = p->next;
    temp=temp->next;
```

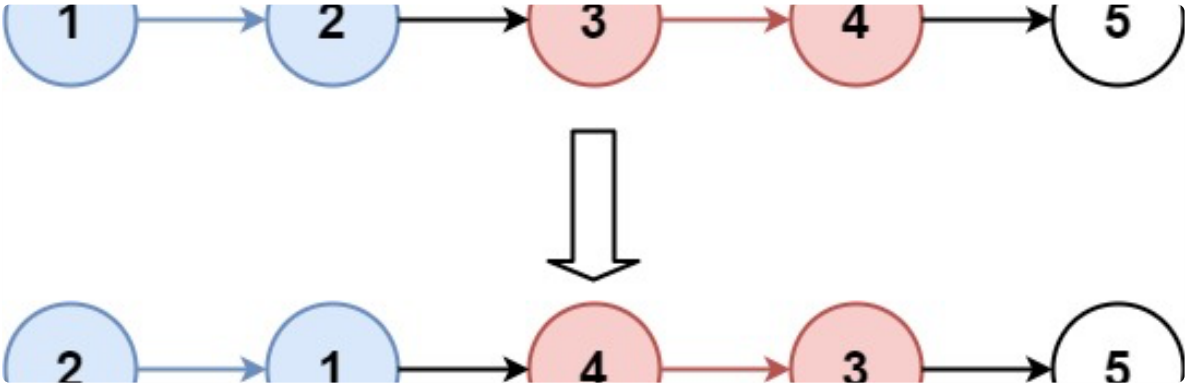
```
    }
    return maxi;
}
};
```

7. Given the `head` of a linked list, reverse the nodes of the list `k` at a time, and return *the modified list*. [Leetcode 25]

   `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.

   You may not alter the values in the list's nodes, only nodes themselves may be changed.

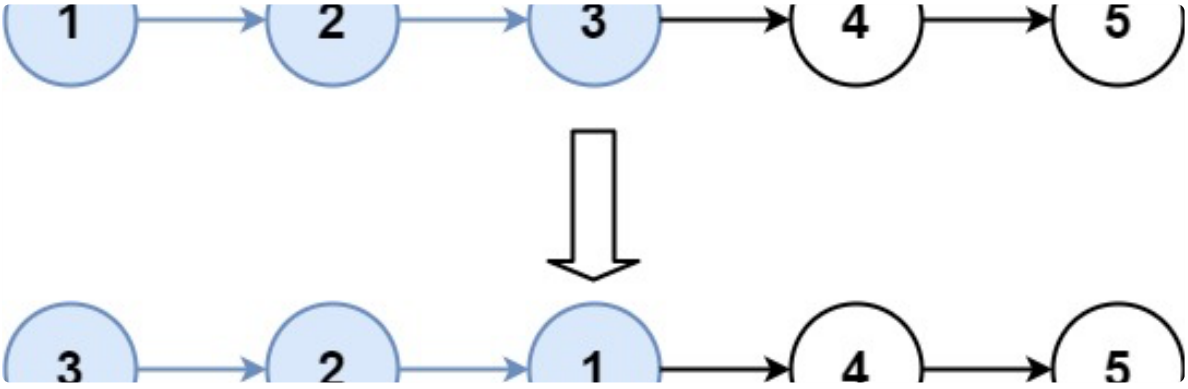
   **Example 1:**




   **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]

   **Constraints:**

   - The number of nodes in the list is `n`.
   - `1 <= k <= n <= 5000`

- `0 <= Node.val <= 1000`

Solution :

```
class Solution {
public:
int length(ListNode *head){
    int len = 0;
    while(head){
        head=head->next;
        len++;
    }
    return len;
}
ListNode* reverseKGroup(ListNode* head, int k) {
        int len = length(head);
        if(!head or len<k)return head;
        ListNode *dummy = new ListNode(0);
        dummy->next = head;
        ListNode* curr = dummy;
        ListNode* prev = dummy;
        ListNode* nex = dummy;
    while(len>=k){
        curr = prev->next;
        nex = curr->next;
        for(int i=1;i<k;i++){
            curr->next = nex->next;
            nex->next = prev->next;
            prev->next = nex;
            nex = curr->next;
        }
        prev = curr;
        len-=k;
    }
    return dummy->next;
}
};
```