__/__/____

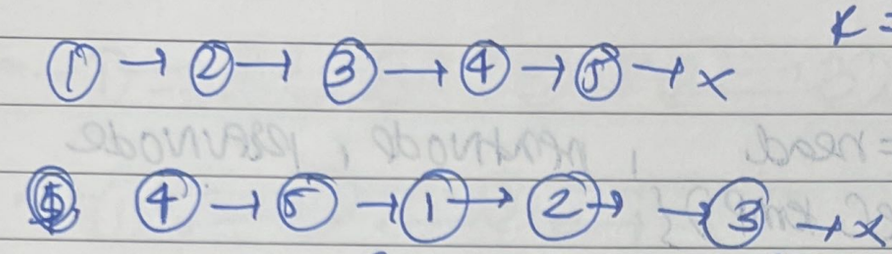
(12) { Reverse Nodes in K Group Size }

```
temp = head , nextNode , prevNode
while( temp ){

    kthNode = giveKthNode(temp, k);
    if(kthNode == NULL){ prevNode -> next = temp
        break;           if(prevNode)
    }
    nextNode = ~~temp~~ kthNode -> next
    kthNode -> next = NULL;
               reverse(temp);
    if(temp == head)
        head = kthNode
    else
        prevNode -> next = kthNode;

    prevNode = temp
    temp = nextNode;
}
```

TC → O(2N)
SC → O(1).

__/__/____

(17) { Rotate LL }

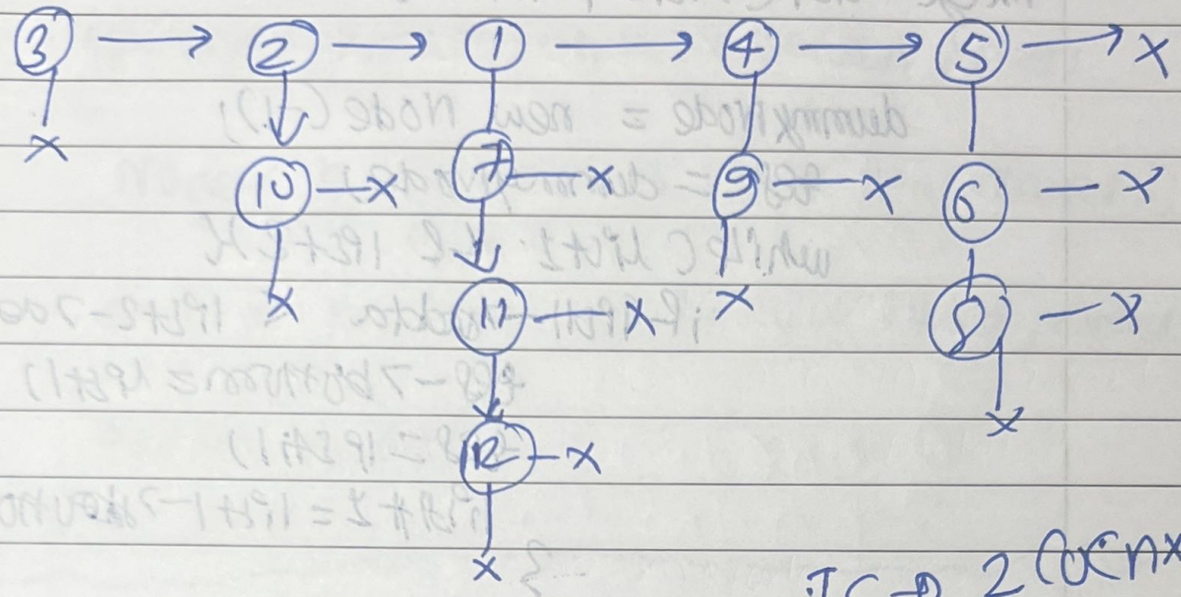



```
Node* rotateLL(Node* head, int k){
    if(k==0) return head;
    int cnt=0;
    Node* temp=head;
    while(temp){
        cnt++;
        temp=temp->next;
    }
    k%=cnt;
    if(k==0) return head;
    cnt-=k;
    temp=head;
    Node* newHead=head;
    while(temp){
        cnt--;
        if(cnt==0){
            newHead=temp->next;
            temp->next=NULL;
            Node* temp2=newHead;
            while(temp2->next) temp2=temp2->next;
            break;
        }
        temp=temp->next;
    }
    return newHead;
}
```

TC → O(2N)
SC → O(1)

__/__/____

(18) Flattening a LL




```
Node* Brute(Node** head){
    vd<int>vec;
    temp = head;
    while(temp){
        temp2 = temp;
        while(temp2){
            vec.add(temp2->data);
            temp2 = temp2->bottom;
        }
        temp = temp->next;
    }
    sort(vec());
    *newHead = new Node(vec[0]);
    temp = newHead;
    for(i, 1, len){
        temp->bottom = new Node(vec[i]);
        temp = temp->bottom;
    }
    return newHead;
```

TC → 2(n×m) + n×m log(n×m)

SC → O(n×m)

O(n×m)

O(n×m log(n×m))

O(n×m)

__/__/____

① Optimal Approach →

```
merge2List(List1, List2){

    dummyNode = new Node(-1);
    res = dummyNode;
    while(List1 && List2){
        if(List1->data < List2->data){
            res->bottom = List1;
            res = List1;
            List1 = List1->bottom;
        }
        else{
            res->bottom = List2;
            res = List2;
            List2 = List2->bottom;
        }
        res->next = NULL;
    }
    if(List1) res->bottom = List1;
    else res->bottom = List2;

    return dummyNode->bottom;
}
```

List1 List2

TC → O(N1+N2)

N / M

__/__/__

```
Recursion(Node* head){

    if(head->next==NULL) return head;

    Node* head2 = recursion(head->next);

    Node* newHead = merge2List(head, head2);

    return newHead;
}
```

TC → N × O(2M) ≈ O(2NM)

SC → O(N) Rec. stack space

(19) Clone a LL with next & random pointers

```
Brute(Node* head){
    temp=head;
    map<Node*, Node*> mapp;
    while(temp){
        Node* deepCopy = new Node(temp->data);
        mapp[temp]= deepCopy;
        temp = temp->next;
    }
    temp=head;
    while(temp){
        Node* copyNode = mapp[temp];
```

__/__/____

```
copyNode->next = map[temp->next];

copyNode->random = map[temp->random];
}

return map[head];
}
```

T(n) → O(2N)

S(n) → O(N) + O(N)
- map
- node creation.

Optimal

```
temp=head                                  | T(n) → O(3N)
while(temp){                               |
    copyNode = newNode(temp->val);         | S(n) → O(N).
    copyNode->next = temp->next
    temp->next = copyNode
    temp = temp.next.next
}
temp=head
    copyNode = temp.next
    copyNode->random = temp.random.next.
    temp = temp.next.next;
}
    dummyNode = new Node(-1); res = dummyNode
    while (temp){
        res.next = temp.next
        temp.next = temp.next.next
        res = res.next
        temp = temp.next;
```

return dummyNode.next