DSA [Linked List]

Q1: For finding the middle of the linked list in one go, go with a **two pointer approach fast and slow**. Fast move two times and slow move 1 step each time. So when fast reach end of linked list, slow reach half of it , ie. middle of linked list. **Time complexity: O(n)**

```
class Solution:
    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        s=head
        f=head
        while(f!=None and f.next!=None):
            s=s.next
            f=f.next.next
        return s
    #for even no of nodes, we take f.next!=None
```

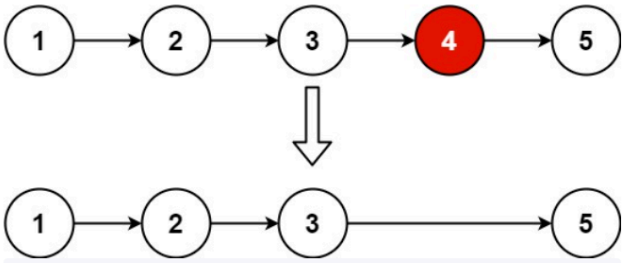
Q2: For removing the kth node from end OR the ques swapping kth node from start and end , we use a two pointer approach.

19. Remove Nth Node From End of List

Medium 15630 656 Add to List Share

Given the `head` of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:




```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

```python
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        fast = head
        slow = head
        # advance fast to nth position
        for i in range(n):
            fast = fast.next

        if fast==None:
            return head.next
        # then advance both fast and slow now they are nth postions apart
        # when fast gets to None, slow will be just before the item to be deleted
        while fast.next:
            slow = slow.next
            fast = fast.next
        # delete the node
        slow.next = slow.next.next
        return head
```

Note: In above solution, fast==None means fast reach to end which indicates that we have to delete the 1st node from beginning or nth node from end.so return head.next

Time complexity: O(n)

Q3: Swapping nodes in the linked list : approach is the same as above.

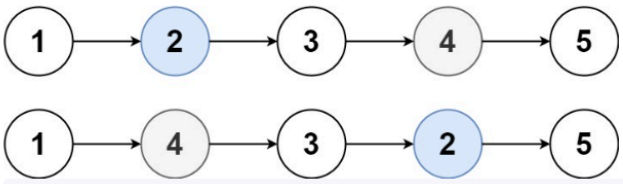
1721. Swapping Nodes in a Linked List

Medium 4613 155 Add to List Share

You are given the `head` of a linked list, and an integer `k`.

Return *the head of the linked list after **swapping** the values of the* k^{th} *node from the beginning and the* k^{th} *node from the end (the list is **1-indexed**).*

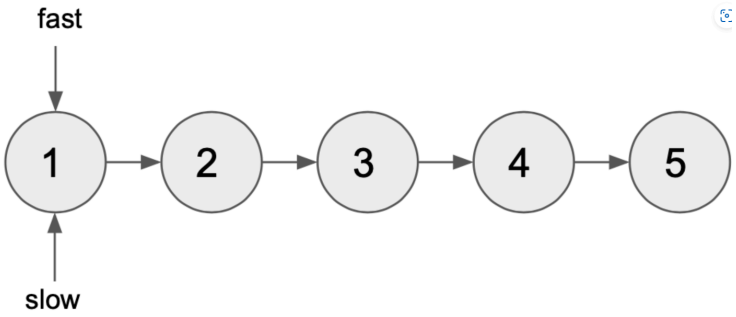
Example 1:




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

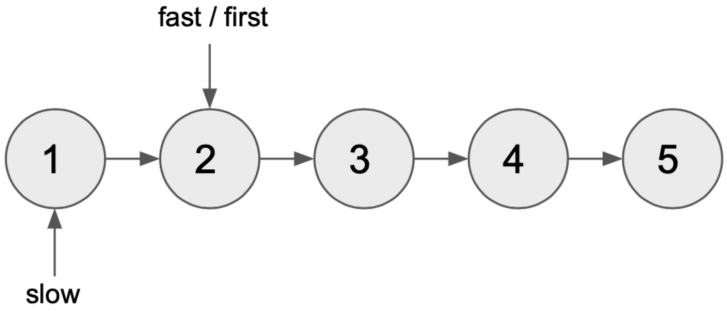
Go with two pointer approach as discussed below:

There are 2 pointers named fast and slow pointing to the first node of the Linked List.



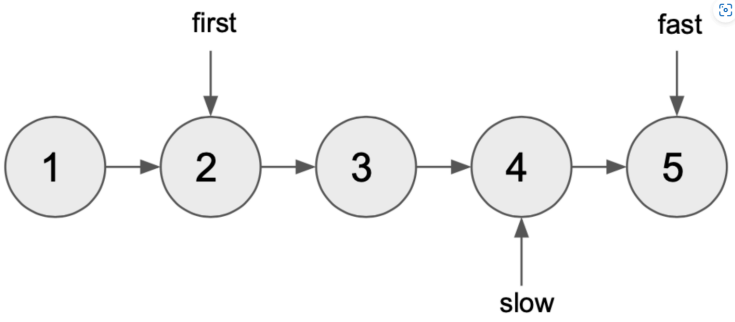
Phase 1

Move **fast** k-1 times. Now **fast** points to the kth node from the beginning. Marked this node **first**.



Phase 2

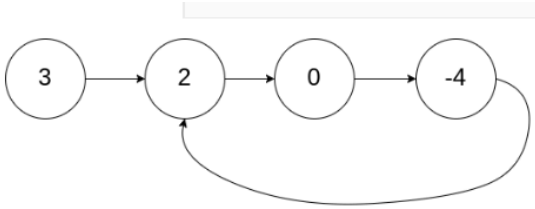
Move **fast** and **slow** together until **fast** points to the last node. Now **slow** points to the kth node from the end.



```python
class Solution:
    def swapNodes(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        fast=head
        slow=head
        n=1
        while(n!=k):
            fast=fast.next
            n=n+1
        first=fast
        while(fast.next!=None):
            fast=fast.next
            slow=slow.next
        first.val,slow.val=slow.val,first.val
        return head
```

Time complexity: O(n)

Q4: Detecting cycle in linked list



Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

It can also be possible that any in-between nodes points to the previous node. That is also a cycle.
Approach Optimized: here also we use two pointers fast and slow, if both pointers meet then there exists a cycle.

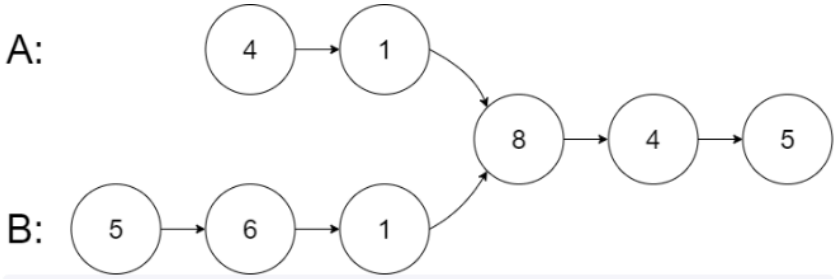
```python
class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        fast=head
        slow=head
        while(fast!=None and fast.next!=None):
            fast=fast.next.next
            slow=slow.next
            if(fast==slow):
                return True
        return False
```

Note: We first check fast!=None and then check fast.next!=None as fast pointer get the None value early as fast.next.next

Time complexity: O(n)

Space complexity: O(1)

Q5: Intersection of link list:



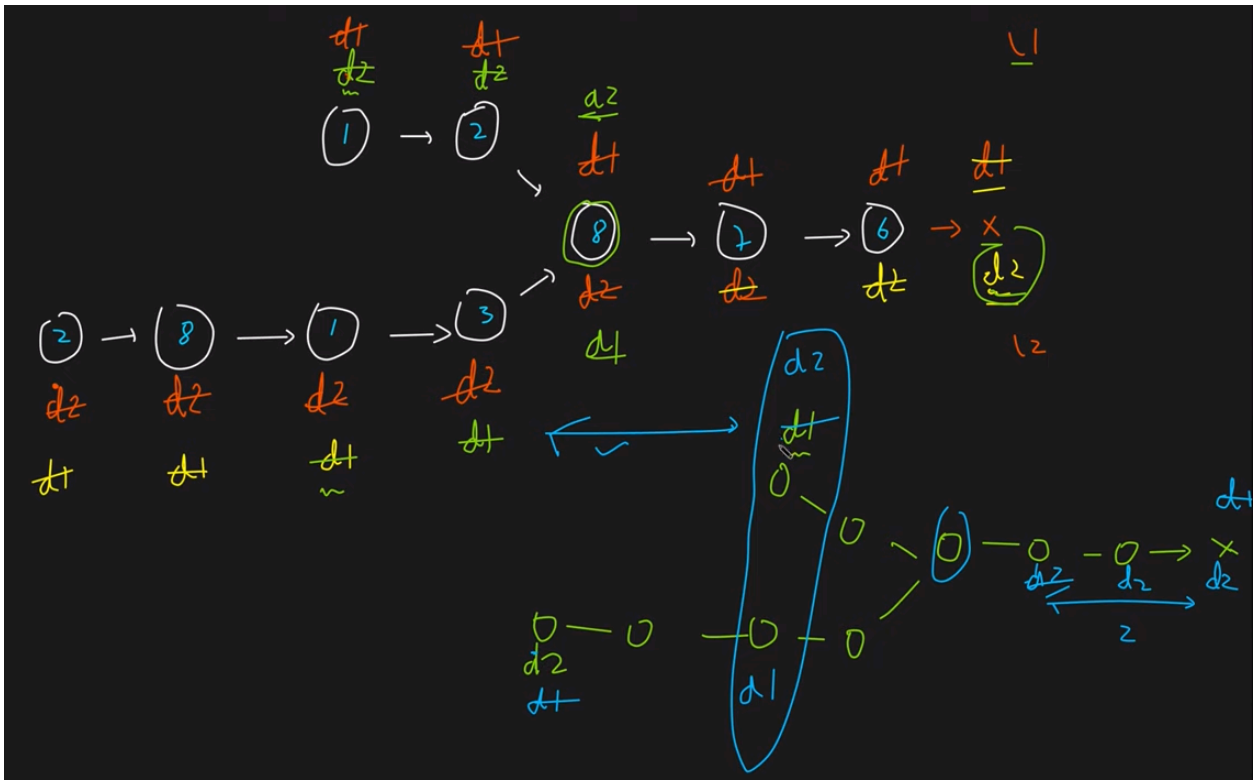

```
Input: intersectVal = 8, listA = [4,1,8,4,5], listB =
[5,6,1,8,4,5], skipA = 2, skipB = 3
Output: Intersected at '8'
```

Brute force approach: which takes extra space.

```python
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> Optional[ListNode]:
        l=[]
        while(headA!=None):
            l.append(headA)
            headA=headA.next
        while(headB!=None):
            if(headB in l):
                return headB
            headB=headB.next
        return None
```

Time complexity: O(m+n)
Space complexity: O(n)
Second approach in which we count the length of both linked list and find the difference . and then start with the link list having greater length with difference.

Optimized approach:



Intuition: move two heads pointer simultaneously, whichever pointer becomes Null, then pointes it to head of other unless both the pointers meets.
Time Complexity: O(2*max(length of list1,length of list2))
Space: O(1)

```python
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> Optional[ListNode]:
        d1=headA
        d2=headB
        while(d1!=d2):
            if(d1==None):
                d1=headB
            elif(d2==None):
                d2=headA
            else:
                d1=d1.next
                d2=d2.next
        return d1
```

Q6: Reverse the link list:

```python
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        curr=head
        prev=None
        nexti=None
        while(curr!=None):
            nexti=curr.next
            curr.next=prev
            prev=curr
            curr=nexti
        return prev
```

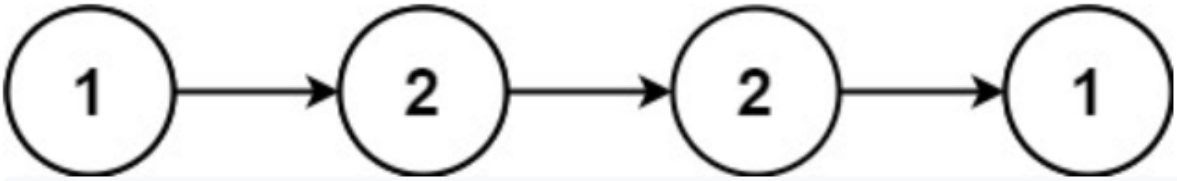
Intuition, optimized: Always make 3 nodes and then put the prev on the first node , curr on the middle and next on the third node, you will get the intuition behind it.

Time: O(n) for reversing

Space: O(1)

Q7: Palindrome link list:

Example 1:




```
Input: head = [1,2,2,1]
Output: true
```

Bruteforce: you can also take one list and store the val of each node , and then reverse it, if both the list are same then palindrome. Time: O(n) for traversing+ O(n) for reversing + O(n) for checking each val of both the list. Space: O(n) for storing one list+O(n) for storing other list.
Another: Reverse the linked list and then check whether the original and reversed list are same or not.Time: O(n) for reversing whole link list. Space: O(n) for creating reverse link list.

Optimized approach: Time Complexity: O(N/2)+O(N/2)+O(N/2)

Reason: O(N/2) for finding the middle element, reversing the list from the middle element, and traversing again to find palindrome respectively.

Space Complexity: O(1)

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        fast=head
        slow=head
        while(fast!=None and fast.next!=None):
            fast=fast.next.next
            slow=slow.next
        curr=slow
        prev=None
        while(curr!=None):
            nexti=curr.next
            curr.next=prev
            prev=curr
            curr=nexti
        temp=prev
        h=head
        while(temp!=None):
            if(temp.val!= h.val):

                return False
            temp=temp.next
            h=h.next
        return True
```

Another way of doing it is by making another reverse function, in this when we call the function we call by self.function_name.

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        fast=head
        slow=head
        while(fast!=None and fast.next!=None):
            fast=fast.next.next
            slow=slow.next
        temp=self.lreverse(slow)
        h=head
        while(temp!=None):
            if(temp.val!= h.val):

                return False
            temp=temp.next
            h=h.next
        return True

    def lreverse(self, head: Optional[ListNode]) -> Optional[ListNode]:
        curr=head
        prev=None
        nexti=None
        while(curr!=None):
            nexti=curr.next
            curr.next=prev
            prev=curr
            curr=nexti
        return prev
```

Q7: Reverse nodes in group of k: Recursion based solution

https://www.youtube.com/watch?v=fi2vh0nQLi0

Watch this video for recursion based solution

```
class Solution:
    def reverseKGroup(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        if(head==None):
            return head
        h=head
        for i in range(k):
            if(h==None):          #to check whether k nodes present or not
                return head
            h=h.next
        prev=None                 #logic to reverse
        curr=head
        for i in range(k):
            nxt = curr.next
            curr.next=prev
            prev=curr
            curr=nxt
        head.next= self.reverseKGroup(curr,k)
        return prev
```

Try this ques by making a diagram of linked list and solve for one then recursion solve for rest.

TC: O(n)

SC: O(n/k) as k nodes are reversed in one go so taking stack of k and we have to do it n/k times as n=no of nodes.

Q8: Merge the two sorted linked list:

TC: O(M+N)

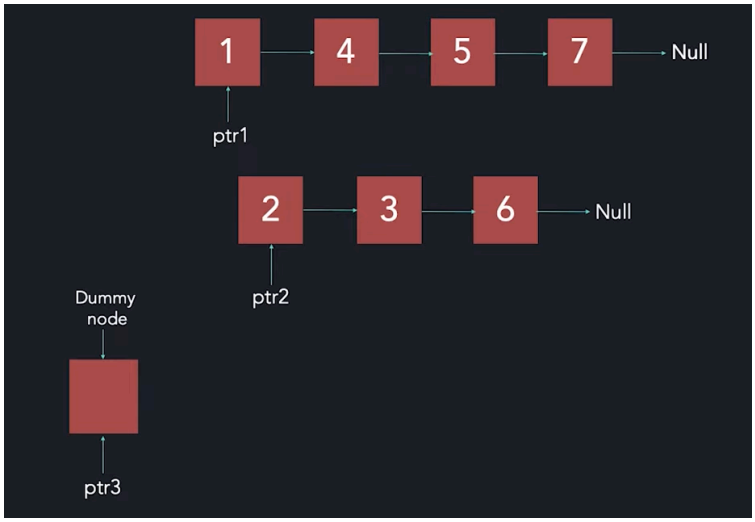
SC:O(1)

Intuition Optimized: Make a dummy node and use three pointers to do it like we merge in an array. only change the links

```
#         self.next = next
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        res=ListNode(0)
        temp= res
        while(list1 and list2):
            if list1.val<list2.val:
                temp.next=list1
                temp=temp.next
                list1=list1.next
            else:
                temp.next=list2
                temp=temp.next
                list2=list2.next
        if(list1):
            temp.next=list1

        if(list2):
            temp.next=list2
        return res.next
```

Above is striver solution and it is good.

```
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) ->
Optional[ListNode]:
        ptr1=list1
        ptr2=list2
        dummy=ListNode(0)
        ptr3=dummy
        while(ptr1!=None and ptr2!=None):
            if(ptr1.val<=ptr2.val):
                ptr3.next=ptr1
                ptr1=ptr1.next
            else:
                ptr3.next=ptr2
                ptr2=ptr2.next
            ptr3=ptr3.next
        if(ptr1!=None):
            ptr3.next=ptr1
        if(ptr2!=None):
            ptr3.next=ptr2
        return dummy.next
```



Reference: https://www.youtube.com/watch?v=n5_9DMCX0Yk

Q9: Add two number :
Optimized approach: TC: O(max(m,n))
SC: O(max(m,n))

Your input	[9,9,9] [9,9,9,9,9,9,9]	Your input	[2,4,3] [5,6,4]
Output	[8,9,9,0,0,0,0,1]	Output	[7,0,8]

```
class Solution:
    def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListNode]) ->
Optional[ListNode]:
        dummy=ListNode()
        temp=dummy
        carry=0
        sm=0
        while(l1!= None or l2!=None or carry!=0):
            if(l1!=None):
                sm+=l1.val
                l1=l1.next
            if(l2!=None):
                sm+=l2.val
                l2=l2.next
            sm+=carry
            t=ListNode(sm%10)
            temp.next=t
            temp=temp.next
            carry=int(sm/10)
            sm=0
        return dummy.next
```

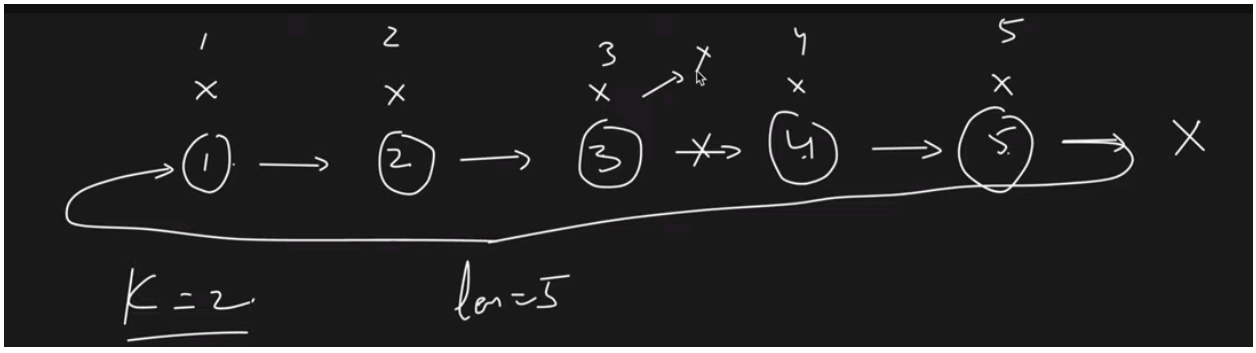
Pratik :

```
class Solution:
    def addTwoNumbers(self, l1: Optional[ListNode], l2: Optional[ListN
Optional[ListNode]:
        dummy = ListNode()
        temp = dummy
        carry=0

        while l1 or l2 or carry:
            sum=0
            if l1:
                sum+=l1.val
                l1=l1.next
            if l2:
                sum+=l2.val
                l2=l2.next
            sum+=carry
            carry = sum//10
            temp.next = ListNode(sum%10)

            temp = temp.next

        return dummy.next
```

Q10: Rotate the link list by k:

Approach: there are two cases:

- When length of link list> k
- When length of link list<k



```
def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
    if(head==None or k==0 or head.next==None):
        return head
    temp=head
    count=1
    while(temp.next!=None):
        temp=temp.next
        count=count+1
    k=k%count
    n=count-k
    print(n)
    if(n==0):
        return head
    temp.next=head
    temp2=head
    for i in range(n-1):
        temp2=temp2.next
    head=temp2.next
    temp2.next=None
    return head
```

Q: Flattening the link list:

Intuition: Concept of merging the two sorted link list. We recursively merge last two linklists and then return it.

```
def flatten(root):
    def mergeTwoLists(list1,list2):
        res=Node(0)
        temp= res
        while(list1 and list2):
            if list1.data<list2.data:
                temp.bottom=list1
                temp=temp.bottom
                list1=list1.bottom
            else:
                temp.bottom=list2
                temp=temp.bottom
                list2=list2.bottom
        if(list1):
            temp.bottom=list1

        if(list2):
            temp.bottom=list2
        return res.bottom

    if root==None or root.next==None:
        return root
    root.next= flatten(root.next)
    root=mergeTwoLists(root,root.next)
    return root
```

Q:CLONE A LINKLIST:

```
class Solution(object):
    def copyRandomList(self, head):
        d = {None:None}
        temp=head
        while(temp):
            t=Node(temp.val)
            d[temp]=t
            temp=temp.next
        curr=head

        while(curr):
            copy= d[curr]
            copy.next=d[curr.next]
            copy.random=d[curr.random]
            curr=curr.next
        return d[head]
```