PW SKILLS

Lesson Plan

Linked List-3

PW SKILLS

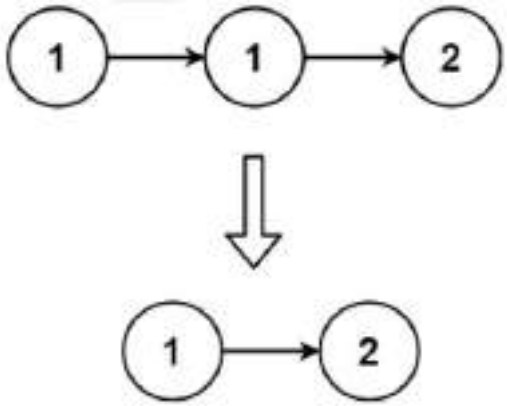
Today's Checklist

1. Remove Duplicates from the Sorted List (Leetcode-83)
2. Rotate List (Leetcode-61)
3. Spiral Matrix IV (Leetcode-2326)
4. Merge 2 sorted lists (Leetcode-21)
5. Merge k sorted lists (Leetcode-23)
6. Sort List (Leetcode-148)
7. Partition List (Leetcode-86)
8. Reverse Linked List (Leetcode-206)
9. Palindrome Linked List (Leetcode-234)
10. Reverse Linked List II (Leetcode-92)
11. Reorder List (Leetcode-143)

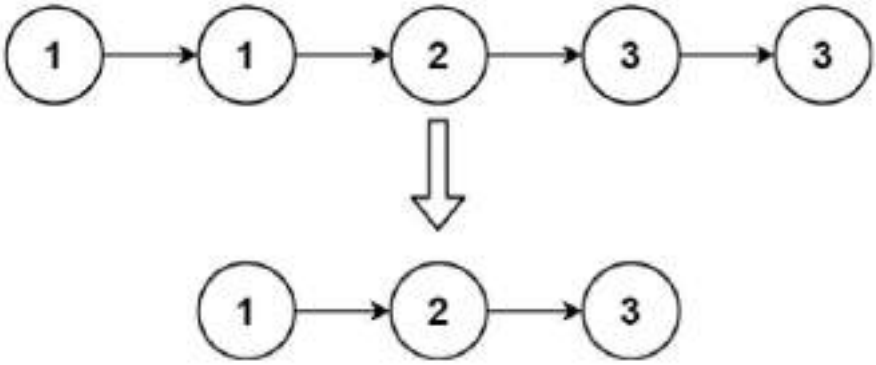
Remove Duplicates from the Sorted List (Leetcode-83)

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

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



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



Code:

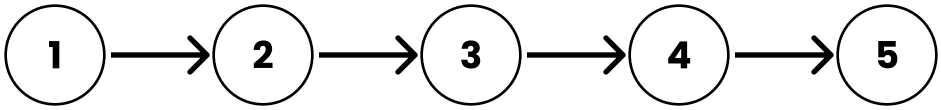
```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return head;
        }

        ListNode* slow = head;
        ListNode* fast = head->next;

        while (fast != nullptr) {
            if (slow->val == fast->val) {
                slow->next = fast->next;
            } else {
                slow = slow->next;
            }
            fast = fast->next;
        }

        return head;
    }
};
```

Time complexity: O(n) - where n is the number of nodes in the linked list.
Space complexity: O(1) - constant space is used, as no additional data structures are employed in the algorithm.

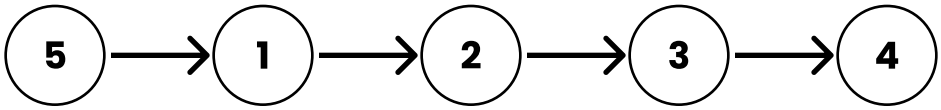
Rotate List (Leetcode-61)

Given the head of a linked list, rotate the list to the right by k places.

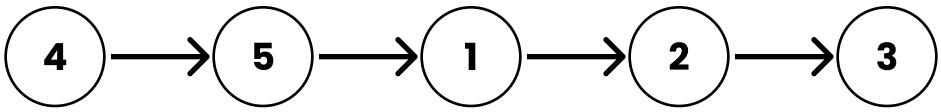
Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]



Rotate 1



Rotate 2



Code:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        ListNode* i = head;
        int n = 0;

        // Calculate Size of LinkedList
        while (i ≠ nullptr) {
            n++;
            i = i→next;
        }

        if (n == 0)
            return head;

        k = k % n;

        if (n == 0 || n == 1 || k == 0 || k == n)
            return head;

        // Breaking at point n-k
        n = n - k - 1;
        i = head;

        while (n > 0) {
            i = i→next;
            n--;
        }

        ListNode* j = i→next;
        ListNode* o = i→next;
        i→next = nullptr;
        k--;

        // Joining lists
        while (k > 0 && j→next ≠ nullptr) {
            j = j→next;
            k--;
        }

        j→next = head;

        return o;
    }
};
```

Time complexity: O(n) - where n is the number of nodes in the linked list. The algorithm iterates through the list twice: once to calculate the length and once to find the new head.
Space complexity: O(1) - constant space is used, as only a constant number of pointers are used regardless of the size of the input linked list.

Spiral Matrix IV (Leetcode-2326)

You are given two integers m and n, which represent the dimensions of a matrix.

You are also given the head of a linked list of integers.

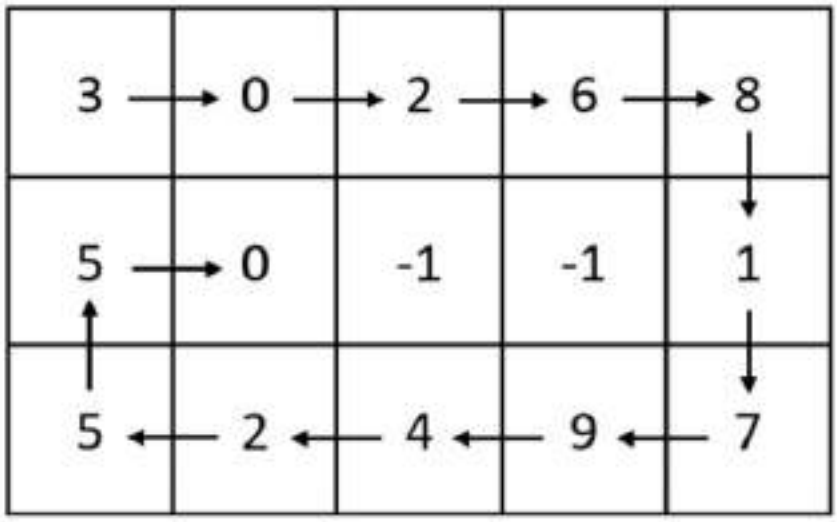
Generate an m x n matrix that contains the integers in the linked list presented in spiral order (clockwise), starting from the top-left of the matrix. If there are remaining empty spaces, fill them with -1.

Return the generated matrix.

Time complexity: m = 3, n = 5, head = [3,0,2,6,8,1,7,9,4,2,5,5,0]
Space complexity: [[3,0,2,6,8],[5,0,-1,-1,1],[5,2,4,9,7]]

Explanation: The diagram above shows how the values are printed in the matrix.

Note: That the remaining spaces in the matrix are filled with -1.



Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val;
this.next = next; }
 * }
 */
class Solution {
    public int[][] spiralMatrix(int m, int n, ListNode head) {
```

```
        int row=0;
        int nrow=m-1;
        int col=0;
        int ncol=n-1;
        int size=n*m;
        int c=0;

        int[][] arr=new int[m][n];

        for(int i=0;i<m;i++)
        {
            for(int j=0;j<n;j++)
                arr[i][j]=-1;
        }

        while(head != null){

            for(int i=row;head != null && i<=ncol;i++)
            {
                arr[row][i]=head.val;
                c+=1;
                head=head.next;
            }
            row+=1;

            for(int i=row;head != null && i<=nrow;i++)
            {
                arr[i][ncol]=head.val;
                c+=1;
                head=head.next;
            }
            ncol-=1;

            for(int i=ncol;head != null && i>=col;i--){
                arr[nrow][i]=head.val;
                c+=1;
                head=head.next;
            }
            nrow-=1;

            for(int i=nrow;head != null && i>=row;i--){
                arr[i][col]=head.val;
                c+=1;
                head=head.next;
            }
            col+=1;

        }

        return arr;
    }
}
```

Time complexity: Time complexity: O(m * n) - where m is the number of rows and n is the number of columns in the resulting matrix. The algorithm iterates through each cell in the matrix to fill it with values from the linked list.
Space complexity: O(m * n) - the space used by the result matrix. The space complexity is determined by the size of the output matrix, which is m x n.

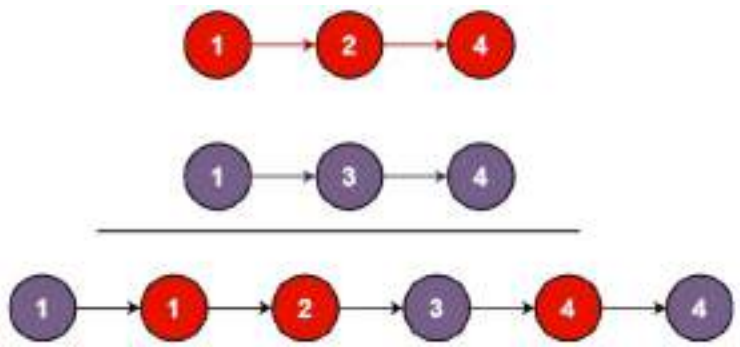
Merge 2 sorted lists (Leetcode-21)

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]



Code:

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2)
{

        if(list1≠null && list2≠null){
        if(list1.val<list2.val){
            list1.next=mergeTwoLists(list1.next,list2);
            return list1;
            }
            else{
                list2.next=mergeTwoLists(list1,list2.next);
                return list2;
        }
        }
        if(list1==null)
            return list2;
        return list1;
    }
}
```

Time complexity: O(m + n) where m and n are the lengths of the two linked lists, l1 and l2. The function recursively traverses through each node in the linked lists.
Space complexity: O(m + n) due to the recursive stack. In the worst case, the maximum depth of the recursion would be the length of the longer linked list between l1 and l2.

Merge k sorted lists (Leetcode-23)

You are given an array of k linked-lists lists, each linked list is sorted in ascending order.
Merge all the linked lists into one sorted linked list and return it.

Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]

Explanation: The linked lists are:

```
[
  1→4→5,
  1→3→4,
  2→6
]
```

merging them into one sorted list: 1->1->2->3->4->4->5->6

Code:

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }
        return mergeKListsHelper(lists, 0, lists.length - 1);
    }

    private ListNode mergeKListsHelper(ListNode[] lists, int
start, int end) {
        if (start == end) {
            return lists[start];
        }
        if (start + 1 == end) {
            return merge(lists[start], lists[end]);
        }
        int mid = start + (end - start) / 2;
        ListNode left = mergeKListsHelper(lists, start, mid);
        ListNode right = mergeKListsHelper(lists, mid + 1, end);
        return merge(left, right);
    }
```

```
    private ListNode merge(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode curr = dummy;

        while (l1 ≠ null && l2 ≠ null) {
            if (l1.val < l2.val) {
                curr.next = l1;
                l1 = l1.next;
            } else {
                curr.next = l2;
                l2 = l2.next;
            }
            curr = curr.next;
        }

        curr.next = (l1 ≠ null) ? l1 : l2;

        return dummy.next;
    }
}
```

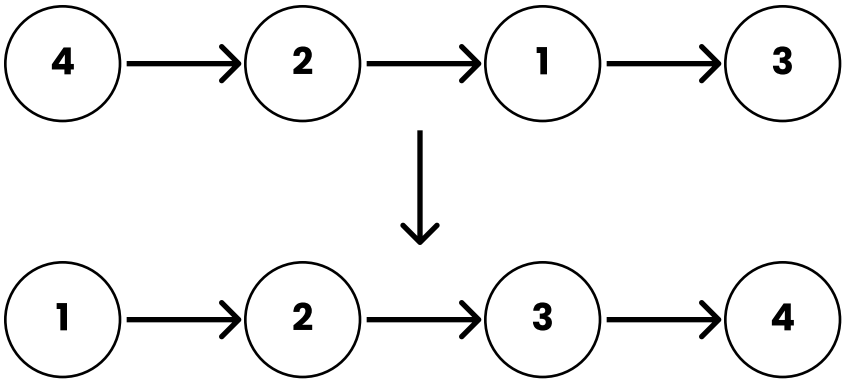
Time complexity: O(N log k), where N is the total number of nodes in all linked lists, and k is the number of linked lists. The mergeKLists function uses a divide-and-conquer strategy to merge the lists, and at each level of the recursion, it performs a linear-time merge operation.

Space complexity: O(log k), where k is the number of linked lists. This is the space used by the recursive call stack. In the worst case, the maximum depth of the recursion would be log k.

Sort List (Leetcode-148)

Given the head of a linked list, return the list after sorting it in ascending order.

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



Code:

```
public class Solution {

  public ListNode sortList(ListNode head) {
    if (head == null || head.next == null)
      return head;

    // step 1. cut the list to two halves
    ListNode prev = null, slow = head, fast = head;

    while (fast != null && fast.next != null) {
      prev = slow;
      slow = slow.next;
      fast = fast.next.next;
    }

    prev.next = null;

    // step 2. sort each half
    ListNode l1 = sortList(head);
    ListNode l2 = sortList(slow);

    // step 3. merge l1 and l2
    return merge(l1, l2);
  }

  ListNode merge(ListNode l1, ListNode l2) {
    ListNode l = new ListNode(0), p = l;

    while (l1 != null && l2 != null) {
      if (l1.val < l2.val) {
        p.next = l1;
        l1 = l1.next;
      } else {
        p.next = l2;
        l2 = l2.next;
      }
      p = p.next;
    }

    if (l1 != null)
      p.next = l1;

    if (l2 != null)
      p.next = l2;

    return l.next;
  }

}
```

Time complexity: O(n log n), where n is the number of nodes in the linked list. This is because the sortList function recursively divides the list into halves, and the merge function performs a linear-time merge operation at each level of the recursion.

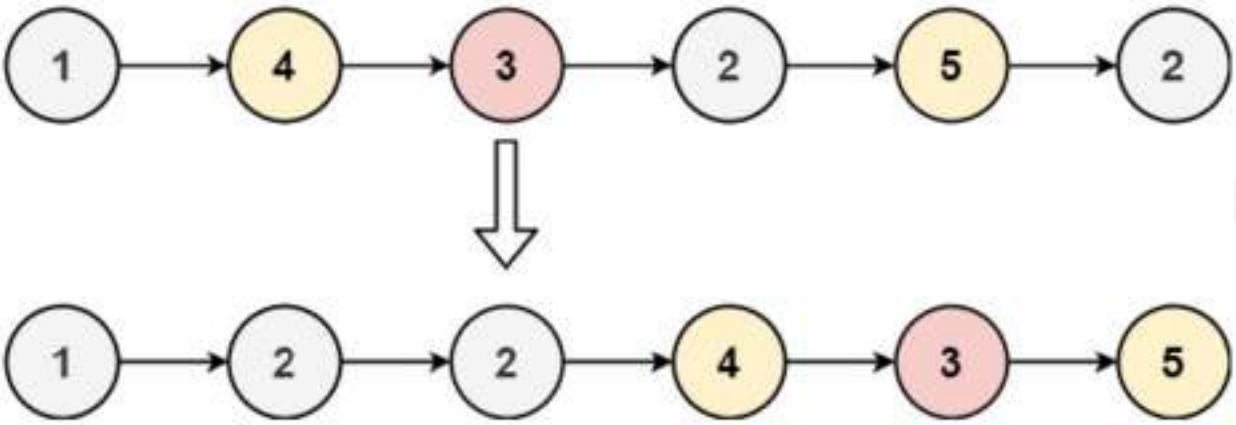
Space complexity: O(log n), where n is the number of nodes in the linked list. This is the space used by the recursive call stack. In the worst case, the maximum depth of the recursion would be log n.

Partition List (Leetcode-86)

Given the head of a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

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



Code:

```
class Solution {
    public ListNode partition(ListNode head, int x) {
        if(head == null)return head;
        ListNode dummyHeadSml = new ListNode();
        ListNode curSmlNode = dummyHeadSml;

        ListNode dummyHeadEg = new ListNode();
        ListNode curEgNode = dummyHeadEg;

        for(ListNode curr = head; curr != null;){
            ListNode next = curr.next;
            curr.next = null;
            if(curr.val < x){
                curSmlNode.next = curr;
                curSmlNode = curr;
            }else{
```

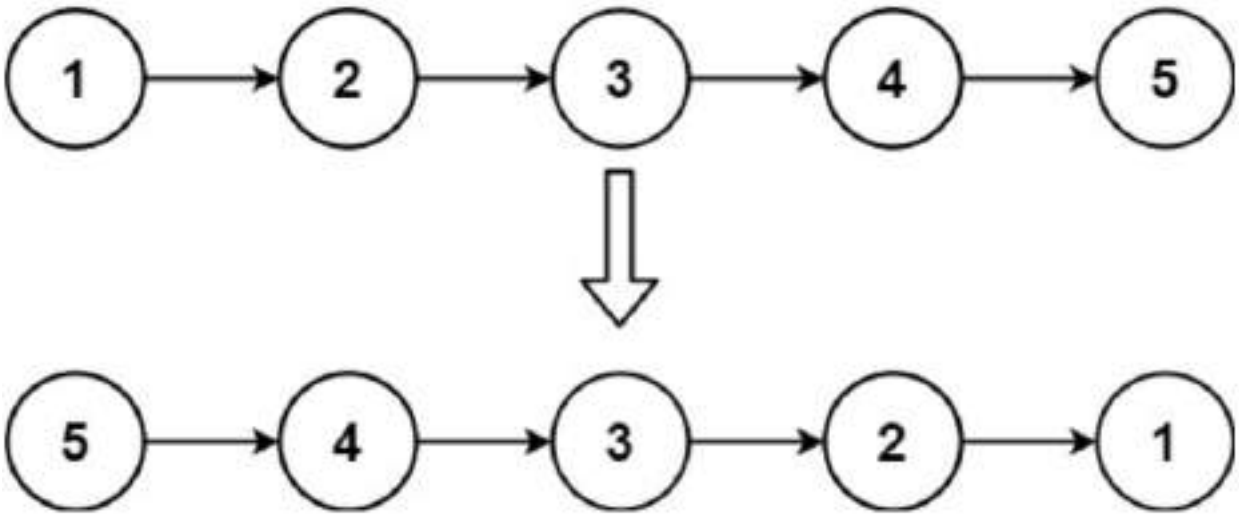
```
                curEgNode.next = curr;
                curEgNode = curr;
            }
            curr = next;
        }

        curSmlNode.next = dummyHeadEg.next;
        return dummyHeadSml.next;
    }
}
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through each node once and performs constant-time operations.

Space complexity: O(1). The algorithm uses a constant amount of extra space to store the left and right partitions, regardless of the size of the input linked list.

Reverse Linked List (Leetcode-206)

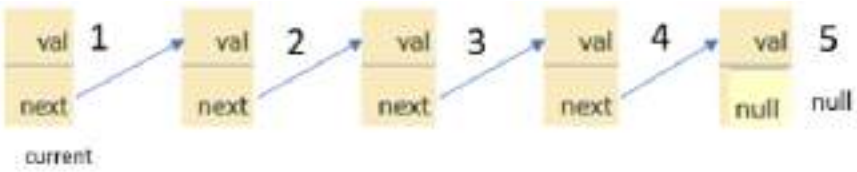
Given the head of a singly linked list, reverse the list, and return the reversed list.



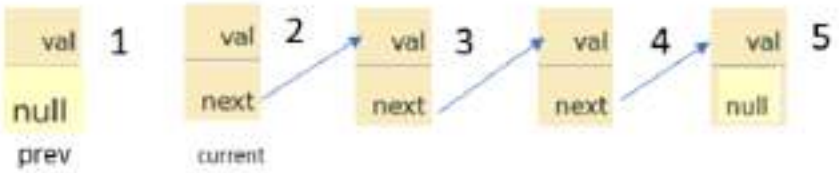
Code:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;


        while(current != null) {
            ListNode next = current.next;
            current.next = prev;
```

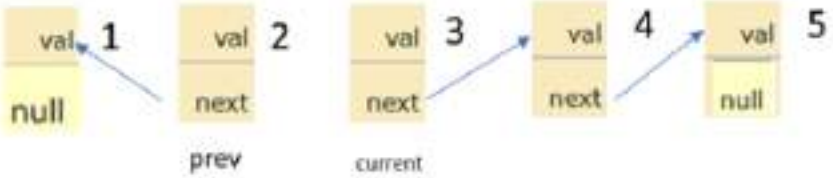
```
            prev = current;
            current = next;
        }
        return prev;
    }
}
```



initially



After 1st iteration



After 2nd iteration

After 3rd iteration

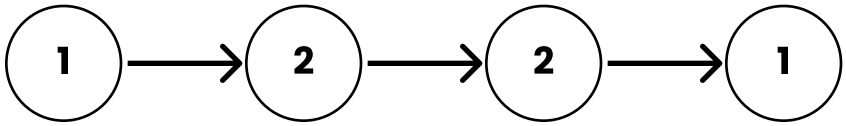
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through each node once, performing constant-time operations in each iteration.

Space complexity: O(1). The algorithm uses a constant amount of extra space for the three-pointers (prev, current, next), regardless of the size of the input linked list.

Palindrome Linked List (Leetcode-234)

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

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



Code:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val;
this.next = next; }
 * }
 */
class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode reverse = reverse(copy(head));

        while(head≠null && reverse≠null){
            if(head.val ≠ reverse.val){
                return false;
            }
            head = head.next;
            reverse = reverse.next;
        }

        return true;
    }

    public ListNode reverse(ListNode node){
        ListNode prev = null;
        ListNode current = node;

        while(current≠null){
            ListNode nextNode = current.next;
            current.next = prev;
            prev = current;
            current = nextNode;
        }

        return prev;
    }

    public ListNode copy(ListNode node){
        ListNode newHead = new ListNode(node.val);
        ListNode current = newHead;
        node = node.next;
```

Code:

```
        while(node != null){
            current.next = new ListNode(node.val);
            node = node.next;
            current = current.next;
        }

        return newHead;
    }
}
```

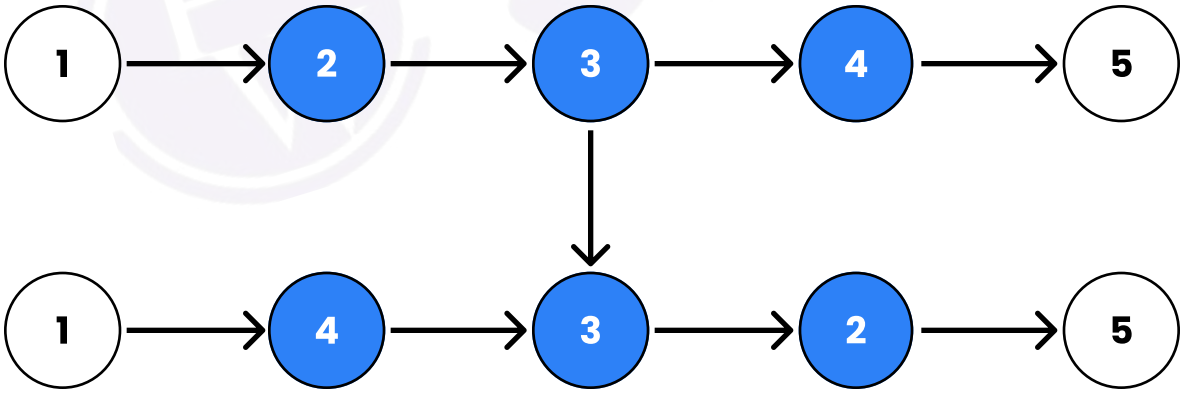
Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list twice—once to find the middle and reverse the second half, and once to compare the reversed second half with the first half.

Space complexity: O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

Reverse Linked List II (Leetcode-92)

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

Input: head = [1,2,3,4,5], left = 2, right = 4
Output: [1,4,3,2,5]



Code:

```
class Solution {
    public ListNode reverseBetween(ListNode head, int left, int
right) {
        if(head.next==null || right==left) return head;
        boolean check = false;
        ListNode Pleft = null;
```

```
        ListNode Pright = null;
        ListNode Tleft = null;
        ListNode Tright = null;
        ListNode curr = head;
        int count = 0;
        while(curr≠null){
            count++;
            if(count==left-1)   Pleft = curr;
            if(count==right+1)  Pright = curr;
            if(count==left)     Tleft = curr;
            if(count==right)    Tright = curr;
            curr = curr.next;
        }
        if(Tleft==head)  check = true;
        if(Pleft≠null && Pleft.next≠null)  Pleft.next = null;
        if(Tright.next≠null) Tright.next = null;
        curr = Tleft;
        ListNode prev = null;
        ListNode next = null;
        while(curr≠null){
            next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        if(Pleft≠null )  Pleft.next = Tright;
        Tleft.next = Pright;
        if(head.next==null || check)  return Tright;
        return head;
    }
}
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm iterates through the linked list once, reversing the specified portion of the list.

Space complexity: O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

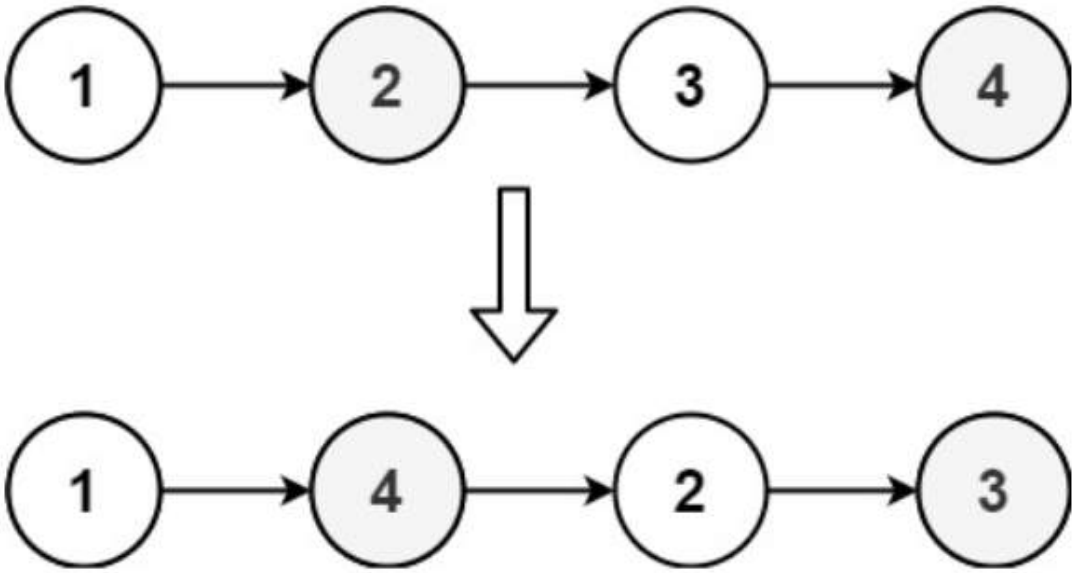
Reorder List (Leetcode-143)

You are given the head of a singly linked-list. The list can be represented as:

L0 → L1 → … → Ln - 1 → Ln
Reorder the list to be on the following form:

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

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



Code:

```
class Solution {
    public void reorderList(ListNode head) {
        ListNode mid = getMid(head);

        ListNode list1 = head;
        ListNode list2 = reverseList(mid.next);
        mid.next = null;

        while(list2 ≠ null) {
            ListNode temp1 = list1.next, temp2 = list2.next;
            list1.next = list2;
            list2.next = temp1;

            list1 = temp1;
            list2 = temp2;
        }
    }

    private ListNode getMid(ListNode head) {
        ListNode slow = head, fast = head;

        while(fast ≠ null && fast.next ≠ null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        return slow;
    }
```

```
    private ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) return head;

        ListNode prev = head;
        ListNode curr = head.next;

        while(curr != null) {
            prev.next = curr.next;
            curr.next = head;

            head = curr;
            curr = prev.next;
        }

        return head;
    }
}
```

Time complexity: O(n), where n is the number of nodes in the linked list. The algorithm has three main steps: cutting the list into two halves, reversing the second half, and merging the two halves. Each step involves iterating through a portion of the linked list, resulting in linear time complexity.

Space complexity: O(1). The algorithm uses a constant amount of extra space for pointers and temporary variables, regardless of the size of the input linked list.

PW SKILLS

THANK YOU !

PW SKILLS