Advance CC-2

STS4006(Advanced Competitive Coding-II)

Course Objectives:

- To enable students to be clear with Advanced Java language
- To have a clear understanding of subject related concepts for advanced competitive coding
- To develop computational ability in Java Programing language

Course Outcome:

- Clear knowledge about problem solving skills in advanced JAVA concepts under competitive coding.

Note: This is for reference purposes only when you are practicing for placements or doing revision for exams.

Loop Detection in a Linked List

Intuition:-

Loop Detection algorithm in a **singly linked list** using Floyd's Tortoise and Hare algorithm. This is a well-known algorithm for detecting cycles in a linked list, and it works by using two pointers, one moving slowly (one step at a time) and the other moving quickly (two steps at a time). If there is a cycle in the linked list, the fast pointer will eventually meet the slow pointer.

Key Points:

- **Node Class**:
    - A Node class is defined with two fields:
    - **data:** The integer data stored in the node.
    - **next:** A reference to the next node in the linked list.

    - The constructor of the Node class initializes the data and sets next to null.
- **Linked List Construction:**
    - The insert(int n) method adds a new node with data n at the end of the linked list. If the list is empty (i.e., head is null), it initializes the list with the new node as the head.
    - The method traverses the list to find the last node and appends the new node to it.
- **Creating a Loop in the Linked List:**
    - The createLoop(int a, int b) method creates a loop in the linked list. Here:
        - a is the data value of the node where the loop will point to.
        - b is the number of steps from the head node at which the loop will be created.
    - It first locates the node with data a, then traverses b steps from the head node and sets the next pointer of the node at position b to point to the node with data a, creating a loop.

Loop Detection Using Floyd's Tortoise and Hare Algorithm:

Code:

```
import java.util.*;
class Node{
   int data;
   Node next;
   Node(int data){
      this.data=data;
      this.next=null;
   }
}
class Main{
   static Node head=null;
   static void insert(int n){
      Node newNode = new Node(n);
      if(head==null){
```

```
        head=newNode;
    }
    else{
        Node curr = head;
        while( curr.next != null ){
            curr = curr.next;
        }
        curr.next = newNode;
    }
}
static boolean createLoop(int a,int b){
    int count = 0;
    Node n1 = head;
    Node n2 = head;
    while( n1 != null && n1.data != a){
        n1 = n1.next;
    }
    if(n1 == null)
        return false;
    while(count < b && n2!=null){
        n2 = n2.next;
        count++;
    }
    n2.next = n1;
    return true;
}
static boolean detetctloop(Node head){
    Node slow = head;
    Node fast = head;

    while(fast!=null && fast.next!=null){
        slow=slow.next;
        fast=fast.next.next;
        if(slow == fast){
            return true;
        }
    }
    return false;
```

```
    }
    public static void main (String[] args) {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        for(int i=0;i<n;i++){
            insert(sc.nextInt());
        }
        int a=sc.nextInt();
        int b= n-1;
        createLoop(a,b);
        boolean d = detetctloop(head);
        if(d){
            System.out.println("Detected");
        }
        else{
            System.out.println("Not Detected");
        }
    }
}
```

Time Complexity : O(N)
Space Complexity : O(constant)

Online Platform Reference Links:

1. Find the first node of the Loop in Linked List
2. Count Loop Length in Linked List
3. Code Force - Floyd's Loop Detection
4. Hacker Earth - Detect and Remove Loop in Linked List

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

1. Detect Loop
2. Find the starting node of the loop
3. Find the Length of the Loop in Linked List
4. Linked List - CodeChef
5. Insert and Delete in Doubly Linked List

Sort the Bitonic Double Linked List

Intuition:-

Given a biotonic doubly linked list. The task is to sort the given linked list. A biotonic doubly linked list is a doubly linked list that is **first increasing and then decreasing**. A **strictly increasing** or a **strictly decreasing** list is also a biotonic doubly linked list.

Key Points:

- **Two Pointers:**
    - first: Start at the head of the list (beginning of the increasing part).
    - last: Starts at the tail of the list (end of the decreasing part).
- **Merging Logic:**
    - Compare first.data and last.data:
    - Append the smaller value to the result list.
    - Move the respective pointer (first forward or last backward).
- **In-Place Reorganization:**
    - Adjust the next and prev pointers to build the sorted list.
    - No extra space is required.
- **Handles All Cases:**
    - Works for fully increasing, fully decreasing, and edge cases like single-node or empty lists.

Code:

```
import java.util.Scanner;
class Main{
    static node head=null;
    static class node{
        int data;
        node next;
        node prev;
        node(int n){
            data=n;
            next=null;
            prev=null;
        }
    }
    static void insert(int n){
        node newnode = new node(n);
        if(head==null) head=newnode;
        else{
            node cur=head;
            while(cur.next!=null) cur=cur.next;
            cur.next=newnode;
            newnode.prev=cur;
        }
    }
    static void display(){
        node cur=head;
        while(cur!=null){
            System.out.print(cur.data+" ");
            cur=cur.next;
        }
        System.out.println();
    }
    static void bit(){
        node first=head;
        node last=head;
        node res=null;
        node resend=null;
        while(last.next!=null) last=last.next;
        while(first!=last){
```

```
        if(first.data<=last.data){
            if(res==null){
                res=resend=first;
                first=first.next;
            }
            else{
                node cur=first.next;
                resend.next=first;
                first.prev=resend;
                resend=resend.next;
                first=cur;
                first.prev=null;
            }
        }
        else{
            if(res==null){
                res=resend=last;
                last=last.prev;
            }
            else{
                node cur=last.prev;
                resend.next=last;
                last.prev=resend;
                resend=resend.next;
                last=cur;
                last.next=null;
            }
        }
    }
    resend.next=first;
    first.prev=resend;
    first.next=null;
    head=res;
}
public static void main(String ar[]){
    Scanner sw = new Scanner(System.in);
    int n=sw.nextInt();
        for(int i=0;i<n;i++)
            insert(sw.nextInt());
```

```
            bit();
        display();
    }
}
```

Time Complexity : O(N)
Space Complexity : O(constant)

Code: (Merge-Based Approach)

```
import java.util.*;
class Node
{
    int data;
    Node next, prev;
    Node(int data)
    {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}

public class Main
{
    public static Node reverse(Node head)
    {
        Node temp = null;
        Node current = head;
        while (current != null)
        {
            temp = current.prev;
            current.prev = current.next;
            current.next = temp;
            current = current.prev;
        }
        return temp != null ? temp.prev : head;
    }
```

```
    public static Node mergeSortedLists(Node first, Node second)
    {
        if (first == null)
return second;
        if (second == null)
return first;
        Node dummy = new Node(0);
        Node current = dummy;
        while (first != null && second != null)
        {
            if (first.data <= second.data)
            {
                current.next = first;
                first.prev = current;
                first = first.next;
            }
            else
            {
                current.next = second;
                second.prev = current;
                second = second.next;
            }
            current = current.next;
        }
        if (first != null)
        {
            current.next = first;
            first.prev = current;
        }
        else if (second != null)
        {
            current.next = second;
            second.prev = current;
        }
        return dummy.next;
    }
    public static Node sortBitonicDoublyLinkedList(Node head)
    {
```

```
        if (head == null || head.next == null)
            return head;
        Node increasing = head;
        Node decreasing = head.next;
            while (decreasing != null && decreasing.next != null && decreasing.data <=
decreasing.next.data)
        {
            decreasing = decreasing.next;
        }
        if (decreasing != null)
        {
            decreasing.prev.next = null;
            decreasing.prev = null;
        }
        decreasing = reverse(decreasing);
        return mergeSortedLists(increasing, decreasing);
    }
    public static void printList(Node head)
    {
        Node temp = head;
        while (temp != null)
        {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        Node head = null, tail = null;
        for (int i = 0; i < n; i++)
        {
            int value = sc.nextInt();
            Node newNode = new Node(value);
            if (head == null)
            {
                head = newNode;
```

```
            tail = newNode;
        }
        else
        {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
        }
    }
    head = sortBitonicDoublyLinkedList(head);
    printList(head);
  }
}
```

Online Platform Reference Links:

1. Sort the Bitonic Double Linked List

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Segregate Even and Odd nodes in a Linked List

Intuition:-

Given a Linked List of integers, The task is to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, preserve the order of even and odd numbers.The task is to segregate the nodes of a singly linked list into two groups:

- **Even nodes:** Nodes with even values.
- **Odd nodes:** Nodes with odd values.

The resulting list will first contain all even nodes, followed by all odd nodes, while preserving the original relative order within each group.

Key Points:

- **Two Separate Sub-Lists:**
    - Create two sub-lists:
        - One for even nodes (es and ee: even start and even end).
        - One for odd nodes (os and oe: odd start and odd end).
    - Traverse the original list, appending nodes to the appropriate sub-list based on their values.
- **Combine the Two Sub-Lists:**
    - If the even list is non-empty:
        - Make the even list the head of the resulting list.
        - Link the last node of the even list (ee) to the head of the odd list (os).
    - If the even list is empty:
        - Make the odd list the head.
- **Preserve Original Order:**
    - Nodes are appended to the sub-lists in the order they appear in the original list, ensuring that the relative order within the even and odd groups is maintained.
- **Handles Edge Cases:**
    - Fully even or fully odd lists.
    - Empty list (no nodes).

Code:

```
import java.util.Scanner;
class Main{
   static node head=null;
   static class node{
   int data;
   node next;
   node(int n){
      data=n;
      next=null;
   }
}
static void insert(int n){
   node newnode = new node(n);
   if(head==null) head=newnode;
```

```
    else{
        node cur=head;
        while(cur.next!=null) cur=cur.next;
        cur.next=newnode;
    }
}
static void seg(){
    node es=null;
    node ee=null;
    node os=null;
    node oe=null;
    node cur=head;
    while(cur!=null){
        if(cur.data%2==0){
            if(es==null) es=ee=cur;
            else ee=ee.next=cur;
        }
        else{
            if(os==null) os=oe=cur;
            else oe=oe.next=cur;
        }
        cur=cur.next;
    }
    if(es==null) head=os;
    else{
        head=es;
        ee.next=os;
        oe.next=null;
    }
}

static void display(){
    node cur=head;
    while(cur!=null){
        System.out.print(cur.data+"-->");
        cur=cur.next;
    }
    System.out.print("null");
}
```

```
    public static void main(String ar[]){
        Scanner sw = new Scanner(System.in);
        int n=sw.nextInt();
        for(int i=0;i<n;i++){
            insert(sw.nextInt());
        }
        display();
        seg();
        System.out.println();
        display();

    }
}
```

Time Complexity : O(N)
Space Complexity : O(constant)

Online Platform Reference Links:

1. Segregate Even and Odd nodes in a Linked List - Code Chef
2. Swap nodes in Pairs - Code Chef
3. Reverse a Linked List - Code Chef
4. Reversed Linked List - Hacker Earth

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

1. Segregate Even and Odd Nodes in a Linked List

Merge Sort for Double Linked List

Intuition:-

Given a doubly linked list, The task is to sort the doubly linked list in non-decreasing order using merge sort.. Also, preserve the order of even and odd numbers.The goal is to sort a **doubly linked list** (DLL) using the merge sort algorithm. Merge sort is a divide-and-conquer algorithm that splits the list into smaller parts, sorts them individually, and merges them back to create a fully sorted list. The doubly linked list's ability to traverse both forward and backward simplifies the merging process.

Key Points:

- **Divide and Conquer:**
  - Split the linked list into two halves using the **fast and slow pointer approach**.
  - Recursively sort each half.
- **Merge Two Sorted Lists:**
  - Combine two sorted sublists into a single sorted list.
  - Use the prev and next pointers to maintain the doubly linked structure.
- **Edge Cases:**
  - Empty list or single-node list: Return the list as it is already sorted.
  - Handle merging of sublists where one list is empty.
- **Preserve Node Links:**
  - Ensure the prev and next pointers are correctly updated during the merge process to maintain the integrity of the doubly linked list.

Code:

```
import java.util.Scanner;
```

```
class Main{
   static node head=null;
   static class node{
   int data;
   node next;
   node prev;
   node(int n){
      data=n;
      next=null;
      prev=null;
   }
}
static void insert(int n){
   node newnode = new node(n);
   if(head==null) head=newnode;
   else{
      node cur=head;
      while(cur.next!=null) cur=cur.next;
      cur.next=newnode;
      newnode.prev=cur;
   }
}
static node sort(node first){
   if(first==null||first.next==null) return first;
   node second=split(first);
   first=sort(first);
   second=sort(second);
   return merge(first,second);
}
static node split(node first){
   node fast=first;
   node slow=first;
   while(fast.next!=null&&fast.next.next!=null){
      fast=fast.next.next;
      slow=slow.next;
   }
   node temp=slow.next;
   slow.next=null;
   return temp;
```

```
}
static node merge(node first,node second){
   if(first==null) return second;
   if(second==null) return first;
   if(first.data<=second.data){
      first.next=merge(first.next,second);
      first.next.prev=first;
      first.prev=null;
      return first;
   }
   else{
     second.next=merge(first,second.next);
     second.next.prev=second;
     second.prev=null;
     return second;
   }
}
static void display(){
   node cur=head;
   while(cur!=null){
      System.out.print(cur.data+"-->");
      cur=cur.next;
   }
   System.out.print("null");
}
   public static void main(String ar[]){
      Scanner sw = new Scanner(System.in);
      int n=sw.nextInt();
      for(int i=0;i<n;i++){
         insert(sw.nextInt());
      }
      head=sort(head);
      display();

   }
}
```

Time Complexity : O(N * Log N)
Space Complexity : O(constant)

Online Platform Reference Links:

Merge Sort using DLL

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Minimum Stack

Intuition:-

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.
1. **Push(num):** Push the given number in the stack.
2. **Pop:** Remove and return the top element from the stack if present, else return -1.
3. **Top:** return the top element of the stack if present, else return -1.
4. **getMin:** Returns minimum element of the stack if present, else return -1.

Key Points:

- **Push with Condition:**
    - When a new element is pushed, check if it is smaller than the current minimum.
    - If it is, encode the current state of the minimum in the stack to allow retrieval later.

- **Pop with Condition:**
    - When popping an element, if the popped element was the encoded value for the minimum, update the minimum using the encoded logic.
- **Efficiency:**
    - All operations (push, pop, top, getMin) should execute in O(1) time.
    - Minimize space overhead by using the stack itself to store auxiliary information.
- **Edge Cases:**
    - Empty stack operations should return a safe value, e.g., 0.
    - Handle cases where a single element is repeatedly pushed and popped.
- **Preservation:**
    - Ensure that the stack order and retrieval of the correct minimum are maintained after multiple push and pop operations.

Code:

```
import java.util.*;

public class Main {
   static Stack<Integer> st=new Stack<>();
   static int min=Integer.MAX_VALUE;
   static void push(int data)
   {
      if(st.isEmpty())
      {
         st.push(data);
         min=data;
      }
      else {
         if(data<min)
         {
            st.push(2*data-min);
            min=data;
```

```
        }
        else
        {
            st.push(data);
        }
    }
}
static int pop() {
    int p,q;
    if(st.isEmpty())
    {
        return -1;
    }
    else
    {
        int n=st.peek();
        p=st.pop();
        q=min;
        if(n<min) {
        min=2*min-n;
        return q;
    }
        return p;
}
}
static int top()
{
    if(st.isEmpty())
    {
        return -1;
    }
    int n=st.peek();
    if(min<n) { return n;}
    return min;
}
static int getMin()
{
    if(st.isEmpty())
    {
```

```
            return -1;
        }
        return min;
    }
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);

        /*

        push(12);
        push(15);
        push(10);
        System.out.println(getMin());
        System.out.println("deleted element : "+pop());
        System.out.println("top of the stack:"+top());
        //push(10);
        System.out.println(getMin());

         */

        int t=sc.nextInt();
        while(t>0){
        int n=sc.nextInt();
        for(int i=0;i<n;i++)
        {
            int data=sc.nextInt();
            push(data);
        }
        System.out.println(getMin());
            t--;
        }

    }

}
```

Time Complexity : O(1)
Space Complexity : O(N)

Online Platform Reference Links:

1.Minimum Stack - CodeForce
2.Minimum Stack - Coding Ninja
3.Simple Stack - Hacker Earth
4.Stack Operation - Hacker Earth

1. Stacks&Queues - CodeChef
2. Stacks&Queues - HackerEarth
3. Stack - Application(Roller Coaster)
4. Stack&Queues - (Age_Limit)
5. Stack_Basic -HackerEarth

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

1. Stacks and Queues

The Celebrity Problem

Intuition:-

> Given a square matrix mat[][] of size n x n, such that mat[i][j] = 1 means i th person knows j th person, the task is to find the celebrity. A celebrity is a person who is known to all but does not know anyone. Return the index of the celebrity, if there is no celebrity return no celebrity.

The **celebrity problem** involves identifying a celebrity at a party. A celebrity is defined as a person who:

1. Knows no one else at the party.
2. Is known by everyone else at the party.

Given an n×n matrix mat:

- mat[i][j] = 1 means person i knows person j.
- mat[i][j] = 0 means person i does not know person j.

The goal is to determine whether there is a celebrity and, if so, find their identity.

Key Points:

Stack-Based Approach:

- **Initialization:**
  - Push all n people into a stack.
- **Pairwise Comparison:**
  - Pop two people (a and b) from the stack.
  - Use the matrix to check if aaa knows b or vice versa.
  - Push the potential celebrity back into the stack.
    - If mat[a][b]=1, a knows b, so a cannot be the celebrity. Push b.
    - If mat[a][b]=0, b cannot be the celebrity. Push a.
- **Single Candidate Check:**
  - After processing, the stack contains at most one person.
  - Verify the remaining person by checking:
    - They are known by everyone.
    - They know no one else.
- **Result:**
  - If the candidate passes the verification, they are the celebrity.
  - Otherwise, there is no celebrity.
- **Edge Cases:**
  - Everyone knows everyone.
  - Nobody knows anyone.
  - n=1: A single person is trivially the celebrity.

Code (Using Stack):

```
import java.util.*;
class Main{
    public static void main (String[] args) {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        int mat[][] = new int[n][n];
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                mat[i][j]=sc.nextInt();
            }
        }
        FindCelebrity(n,mat);
    }
    static void FindCelebrity(int n,int mat[][]){
        Stack<Integer> sts = new Stack<>();
        for(int i=0;i<n;i++){
            sts.push(i);
        }
        while(sts.size()>1){
            int a=sts.pop();
            int b=sts.pop();
            if(mat[a][b]==1){
                sts.push(b);
            }
            else{
                sts.push(a);
            }
        }
        int cel = sts.pop();
        boolean temp = true;
        for(int i=0;i<n;i++){
            if(i!=cel){
                if(mat[i][cel]==0 || mat[cel][i] == 1){
                    temp = false;
                    break;
                }
```

```
            }
        }
        if(temp){
            System.out.println("Celebrity is " + cel);
            return;
        }
        else{
            System.out.println("No Celebrity found");
        }
    }
}
```

Time Complexity : O(N) using Stack
Space Complexity : O(N) using Stack

Code (Two Pointer Approach):

```
import java.util.*;
public class Main {
    public static int findCelebrity(int[][] matrix, int n)
    {
        int candidate = 0;
        for (int i = 1; i < n; i++)
      {
            if (matrix[candidate][i] == 1)
            {
                candidate = i;
            }
        }
        for (int i = 0; i < n; i++) {
            if (i != candidate && (matrix[candidate][i] == 1 || matrix[i][candidate] == 0))
            {
                return -1;
            }
        }
        return candidate;
    }
    public static void main(String[] args)
    {
```

```
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[][] matrix = new int[n][n];
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                matrix[i][j] = sc.nextInt();
            }
        }
        int celebrity = findCelebrity(matrix, n);
        if (celebrity == -1) {
            System.out.println("No celebrity found.");
        } else {
            System.out.println(celebrity);
        }
    }
}
```

Time Complexity : O(N)
Space Complexity : O(1)

Online Platform Reference Links:
Celebrity Problem

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

1. Stacks and Queues

Tower of Hanoi

Intuition:-

Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the

entire stack to another rod (here considered C), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Example:
Input: 2
Output: The Disk 1 is moved from S to A
The Disk 2 is moved from S to D
The Disk 1 is moved from A to D

Input: 3
Output: The Disk 1 is moved from S to D
The Disk 2 is moved from S to A
The Disk 1 is moved from D to A
The Disk 3 is moved from S to D
The Disk 1 is moved from A to S
The Disk 2 is moved from A to D
The Disk 1 is moved from S to D

Key Points:

Stack-Based Approach:

1. **Initialization:**
    - Use three stacks to represent the source rod (sr), auxiliary rod (ax), and destination rod (ds).
    - Push all disks (represented as integers) onto the source stack in decreasing order of size.
2. **Disk Movement:**
    - Simulate the movement of disks between rods using the following rules:
        - Compare the top disks of two stacks.

        - Move the smaller disk to the top of the other stack.
        - If a stack is empty, move the disk from the other stack.
    - Print the details of each move (e.g., "Disk X moved from S to D").
3. **Checking n % 2 and Swapping Auxiliary and Destination Rods:**
    - If the number of disks (n) is even, the roles of the auxiliary (A) and destination (D) rods are swapped.
    - This ensures that the sequence of moves aligns correctly with the rules of the Tower of Hanoi for even and odd numbers of disks.
    - For example:
        - For odd n, the first move is from the source (S) to the destination (D).
        - For even n, the first move is from the source (S) to the auxiliary (A).
4. **Iteration Over Moves:**
    - Determine the total number of moves required as 2^n-1, where n is the number of disks.
    - Alternate moves between the source, auxiliary, and destination stacks using modulo operations:
    - Move between source and destination (S → D).
    - Move between source and auxiliary (S → A).
    - Move between auxiliary and destination (A → D).
5. **Result:**
    - By the end of all moves, all disks are transferred from the source rod to the destination rod in the correct order.

Code (Using Stack):

```
import java.util.Scanner;
import java.util.Stack;
class Main{
   static Stack<Integer> sr = new Stack<>();
   static Stack<Integer> ax = new Stack<>();
   static Stack<Integer> ds = new Stack<>();
```

```
static void change(Stack<Integer> s1,Stack<Integer> s2,char a,char b){
    int v1,v2;
    if(s1.isEmpty()) v1=Integer.MIN_VALUE;
    else v1=s1.pop();
    if(s2.isEmpty()) v2=Integer.MIN_VALUE;
    else v2=s2.pop();
    if(v1==Integer.MIN_VALUE){
        s1.push(v2);
        System.out.println("The Disk "+v2+" is moved from "+b+" to "+a);
    }
    else if(v2==Integer.MIN_VALUE){
        s2.push(v1);
        System.out.println("The Disk "+v1+" is moved from "+a+" to "+b);
    }
    else if(v1<v2){
        s2.push(v2);
        s2.push(v1);
        System.out.println("The Disk "+v1+" is moved from "+a+" to "+b);
    }
    else{
        s1.push(v1);
        s1.push(v2);
        System.out.println("The Disk "+v2+" is moved from "+b+" to "+a);
    }
}
public static void main(String ar[]){
    Scanner sw = new Scanner(System.in);
    int n = sw.nextInt();
    for(int i=n;i>0;i--) sr.push(i);
    char s='S',a='A',d='D';
    if(n%2==0){
        char temp=a;
        a=d;
        d=temp;
    }
    int moves = (int)(Math.pow(2,n)-1);
    for(int i=1;i<=moves;i++){
        if(i%3==1) change(sr,ds,s,d);
        if(i%3==2) change(sr,ax,s,a);
```

```
            if(i%3==0) change(ax,ds,a,d);
        }
    }
}
```

Time Complexity : O(2^N)
Space Complexity : O(N)

Code (Using Recursion):

```
import java.util.*;
public class Main
{
    public static void solveHanoi(int n, char source, char target, char auxiliary)
    {
        if (n == 1)
        {
            System.out.println("Move disk 1 from " + source + " to " + target);
            return;
        }
        solveHanoi(n - 1, source, auxiliary, target);
        System.out.println("Move disk " + n + " from " + source + " to " + target);
        solveHanoi(n - 1, auxiliary, target, source);
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        solveHanoi(n, 'S', 'D', 'A');
    }
}
```

Time Complexity : O(2^N)
Space Complexity : O(N)

Online Platform Reference Links:

1. Again Multi-Peg Tower of Hanoi - Code Chef
2. Multi-Peg Tower of Hanoi - Code Chef
3. Tower of Hanoi - Hacker Earth
4. Disk Tower-Hacker Earth
5. Hanoi - CodeForces
6. Tower of Hanoi- CodeForces

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

1. Stacks and Queues

Stock Span Problem

Intuition:-

The stock span problem is a financial problem where we have a series of daily price quotes for a stock denoted by an array arr[] and the task is to calculate the span of the stock's price for all days. The span arr[i] of the stock's price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the given day is less than or equal to its price on the current day.

Examples:
Input: 7
100 80 60 70 60 75 85
Output: 1 1 1 2 1 4 6
Explanation:
Traversing the given input span 100 is greater than equal to 100 and there are no more elements behind it so the span is 1, 80 is greater than equal to 80 and smaller than 100 so the span is 1, 60 is greater than equal to 60 and smaller than 80 so the span is 1, 70 is greater

than equal to 60,70 and smaller than 80 so the span is 2 and so on. Hence the output will be 1 1 1 2 1 4 6.

Input: 6
10 4 5 90 120 80
Output: 1 1 2 4 5 1
Explanation:

Traversing the given input span 10 is greater than equal to 10 and there are no more elements behind it so the span is 1, 4 is greater than equal to 4 and smaller than 10 so the span is 1, 5 is greater than equal to 4,5 and smaller than 10 so the span is 2, and so on. Hence the output will be 1 1 2 4 5 1.

Key Points:

Stack-Based Approach:

1. **Initialization:**
   - Create an empty stack and initialize the span of the first day to 1. Push the index of the first day onto the stack.
2. **Iterate Over Prices:**
   - For each day, check if the current price is greater than or equal to the price represented by the index at the top of the stack. If true, pop elements from the stack until the condition is false or the stack becomes empty.
   - Calculate the span:
     - If the stack is empty, the span is (i + 1).
     - Otherwise, the span is (i - st.peek()).
   - Push the current day's index onto the stack.
   - Repeat this process for all days.
3. **Result:**
   - The array s[] contains the spans of stock prices for each day.
4. **Edge Cases:**

- All stock prices are the same.
- Stock prices are strictly increasing.
- Stock prices are strictly decreasing.
- Single day (n = 1): The span is always 1.

Code (Using Stack):

```
import java.util.*;
public class Main {
   static void span(int p[],int n,int s[]){
      Stack<Integer> st = new Stack<>();
      st.push(0);
      s[0]=1;
      for(int i=0;i<n;i++){
         while(!st.isEmpty()&&p[st.peek()]<=p[i]){
            st.pop();
         }
         s[i]=(st.isEmpty()?(i+1):(i-st.peek()));
         st.push(i);
      }
   }
         public static void main(String[] args)
         {
            Scanner sw= new Scanner(System.in);
            int n=sw.nextInt();
                  int p[] = new int[n];
                  for(int i=0;i<n;i++) p[i]=sw.nextInt();
                  int s[] = new int[n];
                  span(p, n, s);
                  for(int i=0;i<n;i++)
                  System.out.print(s[i]+" ");
         }
}
```

Time Complexity : O(N)
Why?
Outer Loop (Iterating over Prices):
The for loop runs exactly n times, where n is the number of days (or elements in the p[] array).

Inner While Loop (Stack Operations):

- The while loop pops elements from the stack while the condition p[st.peek()] <= p[i] is true.
- Each element is pushed onto the stack exactly once and popped at most once.
- Therefore, the total number of stack operations (push and pop) across the entire execution of the program is at most 2n.

Overall Time Complexity:

- The outer loop runs n times, and the combined push and pop operations also account for O(n).
- Thus, the total time complexity of the algorithm is O(n).

Space Complexity : O(N)
Online Platform Reference Links:

1. Stock Span Problem - HackerEarth

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

1. Stacks and Queues

Priority Queue using Double Linked List

Intuition:-

Given Nodes with their priority, implement a priority queue using a doubly linked list.
Prerequisite : Priority Queue

- offer() / add(): This function is used to insert new data into the queue.
- poll(): This function removes the element with the lowest priority value from the queue.

- peek() / top(): This function is used to get the lowest priority element in the queue without removing it from the queue.

Approach :

1. Create a doubly linked list having fields info(hold the information of the Node), priority(hold the priority of the Node), prev(point to previous Node), next(point to next Node).
2. Insert the element and priority in the Node.
3. Arrange the Nodes in the increasing order of priority.

Key Points:

1. **Node Structure:**
    - Each node in the doubly linked list stores:
        - data: The value of the element.
        - pr: The priority of the element.
        - next: Pointer to the next node.
        - prev: Pointer to the previous node.
2. **Insertion Logic:**
    - If the list is empty, the new node becomes both the front and rear.
    - If the priority of the new node is smaller than the priority of the front node, it becomes the new front.
    - Otherwise, traverse the list to find the correct position based on priority, and insert the node in sorted order.
    - Handle edge cases like inserting at the end of the list.
3. **Display Logic:**
    - Traverse the list from front to rear to display the elements and their priorities.
4. **Edge Cases:**
    - Empty queue operations should be handled gracefully.
    - Handling duplicate priorities correctly.
    - Support insertion for single or multiple elements dynamically.

Code :

```
import java.util.*;
class node{
        int data;
```

```
        int pr;
        node next;
        node prev;
        node(int n,int pri){
        this.data=n;
        this.pr=pri;
        this.next=null;
        this.prev=null;
        }
}
class Main{
        static node front =null;
        static node rear=null;
        static void insert(int n,int prio){
        node newnode = new node(n,prio);
        if(front ==null){
        front=newnode;
        rear=newnode;
        }
        else if(prio<front.pr){
        newnode.next=front;
        front.prev=newnode;
        front=newnode;
        }
        else{
        node temp=front;
        while(temp.next!=null&&temp.next.pr<=prio){
        temp=temp.next;
        }
        if(temp.next==null){
        temp.next=newnode;
        newnode.prev=temp;
        rear=newnode;
        }
        else{
        newnode.next=temp.next;
        newnode.prev=temp;
        temp.next.prev=newnode;
        temp.next=newnode;
```

```
        }
        }
        }
        static void display(){
        node cur=front;
        while(cur!=null){
        System.out.println(cur.data+" "+cur.pr);
        cur=cur.next;
        }
        }
        public static void main(String ar[]){
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        for(int i=0;i<n;i++){
        int c=sw.nextInt();
        int d=sw.nextInt();
        insert(c,d);
        }
        display();
        }
}
```

Time Complexity : O(N)
Space Complexity : O(N)
Online Platform Reference Links:

1. Priority Queue - Practice - Code-Chef

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Sort Without Extra Space

Intuition:-

Given a queue with random elements, we need to sort it. We are not allowed to use extra space.Sorting a queue without using extra space requires a process where elements are rearranged within the queue itself. The algorithm leverages the queue's existing structure and repeated rotations to achieve a sorted order. This approach relies on identifying the smallest element within a portion of the queue and then placing it at the correct position.

Key Points:

1. **Find Minimum Element:**
    - Identify the minimum element in the current range of the queue using a loop.
    - The index of the minimum element is tracked to preserve its position.
2. **Insert at End:**
    - Rotate the queue to move the minimum element to the end of the queue.
    - All other elements are added back to the queue in their original order.
3. **Iterative Sorting:**
    - For each iteration, the range of unsorted elements in the queue is reduced.
    - The minimum element is identified and moved to its correct position in the queue.
4. **Efficiency:**
    - No additional data structures are used, achieving "in-place" sorting.
    - The algorithm leverages the queue's nature of FIFO (First-In-First-Out) and cyclically rearranges elements.
5. **Edge Cases:**

- Handle queues with duplicate elements, ensuring proper sorting.
- Empty queues should result in no operation.
- Single-element queues should return the same queue.

Code :

```
import java.util.*;
class Main{
   static int min(Queue<Integer> q,int limit){
      int minval=Integer.MAX_VALUE;
      int min_index=-1;
      int n=q.size();
      for(int i=0;i<n;i++){
         int cur=q.poll();
         if(cur<=minval&&i<limit){
            min_index=i;
            minval=cur;
         }
         q.add(cur);
      }
      return min_index;
   }
   static void insertatend(Queue<Integer> q,int min_index){
      int minval=0;
      int n=q.size();
      for(int i=0;i<n;i++){
         int cur=q.poll();
         if(i!=min_index) q.add(cur);
         else minval=cur;
      }
      q.add(minval);
   }
   static void sort (Queue<Integer> q){
      for(int i=0;i<q.size();i++){
         int min_index=min(q,q.size()-i);
         insertatend(q,min_index);
      }
   }
```

```
    public static void main(String ar[]){
        Scanner sw = new Scanner(System.in);
        int n = sw.nextInt();
        Queue<Integer> q = new LinkedList<>();
        for(int i=0;i<n;i++) q.add(sw.nextInt());
        sort(q);
        while(!q.isEmpty()){
            System.out.print(q.poll()+" ");
        }
    }
}
```

Time Complexity : O(N^2)
Space Complexity : O(N)
Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference]() & [Placement Reference]()

1. [Sort without extra space - Youtube Reference]()

Stack Permutations

Intuition:-

A stack permutation is a permutation of objects in the given input queue which is done by transferring elements from the input queue to the output queue with the help of a stack and the built-in push and pop functions.

The rules are:

- Only dequeue from the input queue.
- Use inbuilt push, and pop functions in the single stack.

- The stack and input queue must be empty at the end.
- Only enqueue to the output queue.

There are a huge number of permutations possible using a stack for a single input queue.
Given two arrays, both of unique elements. One represents the input queue and the other represents the output queue. Our task is to check if the given output is possible through stack permutation.

Examples:
Input: 3
q1 = 1 2 3
q2 = 2 1 3
Output: YES
Explanation:
push 1 from input to stack
push 2 from input to stack
pop 2 from stack to output
pop 1 from stack to output
push 3 from input to stack
pop 3 from stack to output

Input: 3
q1 = 1 2 3
q2 = 3 1 2
Output: Not Possible

Key Points:

1. **Push and Pop Logic:**
   - Push elements from the input queue into the stack.
   - Pop elements from the stack if the top of the stack matches the front of the output queue.
2. **Intermediate State:**
   - Use a stack to hold intermediate elements that may need to be popped later in the desired order.
3. **Matching Output Sequence:**

- Continuously check if the top of the stack matches the front of the output queue.
- If it matches, pop from the stack and dequeue from the output queue.
- If it doesn't, continue processing elements from the input queue.

4. **Validation:**
   - If all elements from the input queue are processed, and the stack becomes empty, the output sequence is valid.
   - If the stack contains elements or there is a mismatch between the stack and the output queue, the permutation is invalid.
5. **Edge Cases:**
   - Input and output queues of different lengths (invalid case).
   - Duplicate elements (ensure correct sequence matching).
   - Empty input or output queue (trivial valid case).

Code :

```
import java.util.*;
class Main{
   public static void main(String ar[]){
      Scanner sw = new Scanner(System.in);
      int n=sw.nextInt();
      Queue<Integer> q1 = new LinkedList<>();
      Queue<Integer> q2 = new LinkedList<>();
      for(int i=0;i<n;i++) q1.add(sw.nextInt());
      for(int i=0;i<n;i++) q2.add(sw.nextInt());
      Stack<Integer> st = new Stack<>();
      while(!q1.isEmpty()){
         int ele=q1.poll();
         if(ele==q2.peek()){
            q2.poll();
            while(!st.isEmpty()){
               if(st.peek()==q2.peek()){
                  st.pop();
                  q2.poll();
               }
               else{
                  break;
```

```
            }
          }
        }
        else st.push(ele);
      }
      if(q1.isEmpty()&&st.isEmpty()){
        System.out.print("Yes");
      }
      else System.out.print("No");
    }
}
```

Time Complexity : O(N)
Space Complexity : O(N)
Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Maximum Sliding Window

Intuition:-

Given an array of integers arr, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

Key Points:

1. **Efficient Retrieval of Maximum:**
    - A **deque (double-ended queue)** is used to store indices of array elements in decreasing order.

    - The front of the deque always holds the index of the maximum element in the current window.
2. **Maintaining the Window:**
    - Remove elements that are **out of the current window** (i.e., deque.peek() < i - k + 1).
    - Maintain elements in **decreasing order** in the deque.
    - Remove all **smaller elements** from the back (pollLast()) before inserting the current element.
3. **Efficiency:**
    - Each element is **inserted and removed at most once**, making the solution run in **O(n) time**.
    - The space complexity is **O(k)** due to storing indices in the deque.
4. **Edge Cases:**
    - **Array of size 1** (return the only element).
    - **Window size** k **equal to array size** (return a single max element).
    - **All elements are identical** (ensure deque maintains correct order).
    - **Decreasing or increasing order arrays** (ensure deque operations work as expected).

Code :

```
import java.util.*;
public class Main {
   public static int[] maxSlidingWindow(int[] nums, int k) {
      if (nums == null || nums.length == 0)
         return new int[0];
      int n = nums.length;
      int[] result = new int[n - k + 1];
      Deque<Integer> deque = new LinkedList<>();
      for (int i = 0; i < n; i++) {
         while (!deque.isEmpty() && deque.peek() < i - k + 1) {
            deque.poll();
         }
```

```
            while (!deque.isEmpty() && nums[i] >= nums[deque.peekLast()]) {
                deque.pollLast();
            }
            deque.offer(i);
            if (i >= k - 1) {
                result[i - k + 1] = nums[deque.peek()];
            }
        }
        return result;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        int[] nums = new int[n];
        for (int i = 0; i < n; i++) {
            nums[i] = scanner.nextInt();
        }
        int k = scanner.nextInt();
        int[] result = maxSlidingWindow(nums, k);
        for (int num : result) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity : O(N)
Space Complexity : O(N + k -1)

Online Platform Reference Links:

1. Sliding Window Maximum-CodeChef
2. Maximum Sum of K Elements-CodeChef
3. Maximum Common Elements-CodeChef
4. Hacker-Earth

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Recover the BST (Binary Search Tree)

Intuition:-

Given a Binary Search Tree (BST) with Two nodes swapped, the task is to recover the original BST.

A Binary Search Tree (BST) should have the property that for any node, its left subtree contains smaller values, and its right subtree contains larger values. If two nodes in a BST are swapped by mistake, the BST property is violated.

The goal is to **identify and recover** the two swapped nodes.

Key Observations

- Inorder Traversal of a BST results in a sorted sequence.
- If two nodes are swapped, the sorted order is disturbed at one or two places.
- By traversing the tree in inorder, we can detect misplaced nodes.

1. **Approach:**
   - Perform an inorder traversal to detect swapped nodes.
   - Use a pointer prev to keep track of the last visited node.
   - Identify two misplaced nodes:
     - **Case 1 (Non-Adjacent Swap):** Two nodes (first and last) are misplaced far apart.
     - **Case 2 (Adjacent Swap):** Two nodes (first and middle) are adjacent.
   - Swap the identified nodes to restore BST property.
2. **Edge Cases:**
   - Tree already satisfies BST properties.
   - Only two adjacent nodes are swapped.
   - Tree has only one node.

- Multiple misplaced nodes (not handled in this approach).

Code :

```
import java.util.*;
class node {
   int data;
   node left, right;

   node(int d) {
      data = d;
      left = right = null;
   }
}
class BT {
   node root = null;
   node first = null;
   node last = null;
   node middle = null;
   node prev = null;
   Scanner sc = new Scanner(System.in);

   node insert() {
      int d = sc.nextInt();
      if (d == -1) {
         return null;
      }
      node tnode = new node(d);
      Queue<node> q = new LinkedList<>();
      q.add(tnode);
      root = tnode;

      while (!q.isEmpty()) {
         node curr = q.poll();
         int l = sc.nextInt();
```

```
        if (l != -1) {
            node lnode = new node(l);
            curr.left = lnode;
            q.add(lnode);
        }
        int r = sc.nextInt();
        if (r != -1) {
            node rnode = new node(r);
            curr.right = rnode;
            q.add(rnode);
        }
    }

    return root;
}

private void inorder(node root) {
    if (root == null) {
        return;
    }
    inorder(root.left);
    if (prev != null && root.data < prev.data) {
        if (first == null) {
            first = prev;
            middle = root;
        } else {
            last = root;
        }
    }
    prev = root;
    inorder(root.right);
}

public void lnorder(node root) {
    if (root == null) {
        return;
    }
    lnorder(root.left);
    System.out.print(root.data + " ");
```

```
        Inorder(root.right);
    }

    void recoverTree(node root) {
        prev = new node(Integer.MIN_VALUE);
        inorder(root);
        if (first != null && last != null) {
            int temp = first.data;
            first.data = last.data;
            last.data = temp;
        } else if (first != null && middle != null) {
            int temp = first.data;
            first.data = middle.data;
            middle.data = temp;
        }
    }
}
class Main {
    public static void main(String ar[]) {
        BT b = new BT();
        b.root = b.insert();
        b.Inorder(b.root);
        b.recoverTree(b.root);
        b.Inorder(b.root);
    }
}
```

Time Complexity : O(N)
Space Complexity : O(1)

Online Platform Reference Links:

1. Binary Search Tree - Hacker Earth
2. Codeforces - Recover the BST

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Views of a Tree

In Data Structures and Algorithms (DSA), the view of a binary tree refers to how the tree appears when observed from a particular angle. Views are significant in solving problems related to visual representation, hierarchy-based traversals, and graphical rendering.

The concept mainly revolves around level-order traversal (BFS) and vertical order traversal using lists,queues, recursion, or HashMaps.

Types of Views:

1.Left View
2.Right View
3.Top View
4.Bottom View
5.Front View
6.Back View

1. Left View

Intuition:-

Given a Binary Tree, the task is to print the left view of the Binary Tree. The left view of a Binary Search Tree (BST) consists of the nodes visible when the tree is viewed from the left side. The first node encountered at each level is included in the left view.

Key Observations

1. **Recursive Approach:**
    - Traverse the tree level by level.

- Maintain the current level and a list to store the first node encountered at each level.
- If the level is equal to the list size, add the node to the list.
- Recursively call for the left subtree first, then the right subtree.

2. **Iterative Approach (alternative method using queue):**
   - Use a queue to perform level order traversal.
   - Store the first node encountered at each level.
3. **Edge Cases:**
   - Empty tree should return an empty list.
   - Trees with only left or right subtrees.
   - Trees with unbalanced structures.

Code :

```
import java.io.*;
import java.util.*;

class Node {
   int data;
   Node left, right;

   Node(int data) {
      this.data = data;
      this.left = null;
      this.right = null;
   }
}

public class Main {
   static void leftview(Node root, List<Integer> l, int level) {
      if (root == null) return;

      if (level == l.size()) l.add(root.data);

      leftview(root.left, l, level + 1);
      leftview(root.right, l, level + 1);
   }
```

```
static Node build(String s[]) {
    if (s.length == 0 || s[0].equals("-1")) return null;

    Node root = new Node(Integer.parseInt(s[0]));
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    int i = 1;

    while (!q.isEmpty() && i < s.length) {
        Node cur = q.poll();
        String cval = s[i];

        if (!cval.equals("-1")) {
            cur.left = new Node(Integer.parseInt(cval));
            q.add(cur.left);
        }
        i++;

        if (i >= s.length) break;
        cval = s[i];

        if (!cval.equals("-1")) {
            cur.right = new Node(Integer.parseInt(cval));
            q.add(cur.right);
        }
        i++;
    }
    return root;
}

public static void main(String[] args) {
    Scanner sw = new Scanner(System.in);
    String s[] = sw.nextLine().split("");
    Node root = build(s);
    ArrayList<Integer> l = new ArrayList<>();
    leftview(root, l, 0);
    for (int i : l) System.out.print(i + " ");
}
```

}

Time Complexity : O(N)
Space Complexity : O(H) where H is the height of the tree (O(N) in the worst case for a skewed tree, O(log N) for a balanced tree).

Online Platform Reference Links:

1. Hacker Earth

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

2. Right View

Intuition:-

Given a Binary Tree, the task is to print the Right view of it. The right view of a Binary Tree is a set of rightmost nodes for every level. The right view of a BST consists of the rightmost nodes visible when the tree is viewed from the right side. This is achieved using level-order traversal while ensuring that the first node encountered at each level is stored.

Key Observations

1. **Recursive Approach (DFS with level tracking):**
    - Start from the root and traverse the right subtree first, then the left subtree.
    - Keep track of the current level and store the first node encountered at that level.
    - Continue the traversal until all nodes are processed.
2. **Iterative Approach (Using a Queue – BFS):**
    - Perform a level-order traversal using a queue.
    - At each level, add the last node encountered to the result list.

Code :

```
import java.io.*;
```

```
import java.util.*;
class node{
    int data;
    node left;
    node right;
    node(int data){
        this.data=data;
        this.left=null;
        this.right=null;
    }
}
public class Solution {
    static void rightview(node root,List<Integer> l, int level){
        if(root==null) return;
        if(level==l.size()) l.add(root.data);
        if(root.right!=null) rightview(root.right,l,level+1);
        if(root.left!=null) rightview(root.left,l,level+1);
    }
    static node built(String s[]){
         if (s.length == 0 || s[0].equals("-1")) return null;
        node root = new node(Integer.parseInt(s[0]));
        Queue<node> q = new LinkedList<>();
        q.add(root);
        int i=1;
        while(!q.isEmpty()&&i<s.length){
            node cur=q.poll();
            String cval=s[i];
            if(!cval.equals("-1")){
                cur.left=new node(Integer.parseInt(cval));
                q.add(cur.left);
            }
            i++;
            if(i>=s.length) break;
            cval=s[i];
            if(!cval.equals("-1")){
                cur.right=new node(Integer.parseInt(cval));
                q.add(cur.right);
            }
            i++;
```

```
        }
        return root;
    }
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        String s[]=sw.nextLine().split(" ");
        node root=built(s);
        ArrayList<Integer> l = new ArrayList<>();
        rightview(root,l,0);
        for(int i:l) System.out.print(i+" ");
    }
}
```

Time Complexity : O(N)
Space Complexity : O(H) where H is the height of the tree (O(N) in the worst case for a skewed tree, O(log N) for a balanced tree).

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

3. Top View

Intuition:-

The **top view** of a Binary Tree refers to the set of nodes visible when the tree is viewed from the top. It consists of the leftmost and rightmost nodes at every horizontal distance. Nodes that are covered by other nodes when viewed from the top are not included.

Key Observations

1. **BFS (Level Order Traversal with Horizontal Distance Tracking)**

- Traverse the tree level by level.
- Use a Queue to store nodes along with their horizontal distances (HD).
- Maintain a Map to store the first node encountered at each HD.
- The left child decreases the HD, while the right child increases it.

2. **Edge Cases:**
   - An empty tree should return an empty list.
   - Trees with only left or right children.
   - Unbalanced trees.

Code :

```
import java.util.*;
class Node {
   int data;
   Node left, right;
   Node(int n) {
      data = n;
      left = right = null;
   }
}
class Pair {
   Node node;
   int hd;
   Pair(Node node, int hd) {
      this.node = node;
      this.hd = hd;
   }
}
public class Main {
   static Node buildTree(String s[]) {
      if (s.length == 0 || s[0].equals("N")) return null;
      Node root = new Node(Integer.parseInt(s[0]));
      Queue<Node> q = new LinkedList<>();
      q.add(root);
      int i = 1;
      while (!q.isEmpty() && i < s.length) {
         Node cur = q.poll();
```

```
            String cval = s[i];
            if (!cval.equals("N")) {
                cur.left = new Node(Integer.parseInt(cval));
                q.add(cur.left);
            }
            i++;
            if (i >= s.length) break;
            cval = s[i];
            if (!cval.equals("N")) {
                cur.right = new Node(Integer.parseInt(cval));
                q.add(cur.right);
            }
            i++;
        }
        return root;
    }
    static void topView(Node root) {
        if (root == null) return;
        TreeMap<Integer, Integer> map = new TreeMap<>();
        Queue<Pair> queue = new LinkedList<>();
        queue.add(new Pair(root, 0));
        while (!queue.isEmpty()) {
            Pair temp = queue.poll();
            Node cur = temp.node;
            int hd = temp.hd;
            if (!map.containsKey(hd)) {
                map.put(hd, cur.data);
            }
            if (cur.left != null) queue.add(new Pair(cur.left, hd - 1));
            if (cur.right != null) queue.add(new Pair(cur.right, hd + 1));
        }
        for (int val : map.values()) {
            System.out.print(val + " ");
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s[] = sc.nextLine().split(" ");
        Node root = buildTree(s);
```

```
        topView(root);
    }
}
```

4. Bottom View

Intuition:-

The **bottom view** of a Binary Tree refers to the set of nodes visible when the tree is viewed from the bottom. It consists of the last nodes encountered at each **horizontal distance** during traversal.

Key Observations

1. **BFS (Level Order Traversal with Horizontal Distance Tracking)**
    - Use a Queue to store nodes along with their horizontal distance (HD).
    - Maintain a Map to store the last node encountered at each HD.
    - The left child decreases the HD, and the right child increases it.

Code :

```
import java.util.*;
class Node {
    int data;
    Node left, right;
    Node(int n) {
        data = n;
        left = right = null;
    }
}
class Pair {
    Node first;
    int second;
```

```
    Pair(Node first, int second) {
      this.first = first;
      this.second = second;
    }
}
public class Main {
    static Node build(String s[]) {
      if (s.length == 0 || s[0].equals("N")) return null;
      Node root = new Node(Integer.parseInt(s[0]));
      Queue<Node> q = new LinkedList<>();
      q.add(root);
      int i = 1;
      while (!q.isEmpty() && i < s.length) {
        Node cur = q.poll();
        String cval = s[i];
        if (!cval.equals("N")) {
          cur.left = new Node(Integer.parseInt(cval));
          q.add(cur.left);
        }
        i++;
        if (i >= s.length) break;
        cval = s[i];
        if (!cval.equals("N")) {
          cur.right = new Node(Integer.parseInt(cval));
          q.add(cur.right);
        }
        i++;
      }
      return root;
    }
    static void bottomView(Node root) {
      if (root == null) return;
      TreeMap<Integer, Integer> m = new TreeMap<>();
      Queue<Pair> q = new LinkedList<>();
      q.add(new Pair(root, 0));
      while (!q.isEmpty()) {
        Pair temp = q.poll();
        Node cur = temp.first;
```

```
            int hd = temp.second;
            m.put(hd, cur.data);
            if (cur.left != null) q.add(new Pair(cur.left, hd - 1));
            if (cur.right != null) q.add(new Pair(cur.right, hd + 1));
        }
        for (int val : m.values()) {
            System.out.print(val + " ");
        }
    }
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        String s[] = sw.nextLine().split(" ");
        Node root = build(s);
        bottomView(root);
    }
}
```

5. Front View

Intuition:-

- The **Front View** of a Binary Tree is essentially the **entire tree** when viewed from the front.
- This means that every node at every level is visible.
- The simplest way to achieve this is through a **Level Order Traversal (BFS).**

Key Observations:

- **BFS (Queue-Based Level Order Traversal)**
    - Traverse the tree level by level.
    - Print every node in the traversal, as all nodes are visible from the front.

Edge Cases:

- Empty tree → Return an empty list.
- Single-node tree → Return that single node.

- Unbalanced trees → Still print all nodes level by level.

Code :

```
import java.util.*;
class Node {
    int data;
    Node left, right;
    Node(int n) {
        data = n;
        left = right = null;
    }
}
public class Main {
    static Node build(String s[]) {
        if (s.length == 0 || s[0].equals("N")) return null;
        Node root = new Node(Integer.parseInt(s[0]));
        Queue<Node> q = new LinkedList<>();
        q.add(root);
        int i = 1;
        while (!q.isEmpty() && i < s.length) {
            Node cur = q.poll();
            String cval = s[i];
            if (!cval.equals("N")) {
                cur.left = new Node(Integer.parseInt(cval));
                q.add(cur.left);
            }
            i++;
            if (i >= s.length) break;
            cval = s[i];
            if (!cval.equals("N")) {
                cur.right = new Node(Integer.parseInt(cval));
                q.add(cur.right);
            }
            i++;
        }
        return root;
```

```
    }
    static void levelOrder(Node root) {
        if (root == null) return;
        Queue<Node> q = new LinkedList<>();
        q.add(root);
        while (!q.isEmpty()) {
            Node cur = q.poll();
            System.out.print(cur.data + " ");
            if (cur.left != null) q.add(cur.left);
            if (cur.right != null) q.add(cur.right);
        }
    }
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        String s[] = sw.nextLine().split(" ");
        Node root = build(s);
        levelOrder(root);
    }
}
```

6. Back View

Since the Back View of a binary tree is identical to the Front View, the same Level Order Traversal (BFS) can be used.
Thus, the Back View is the same as Front View.

Code:

```
import java.util.*;
class Node {
    int data;
    Node left, right;
    Node(int n) {
        data = n;
        left = right = null;
    }
}
public class Main {
    static Node build(String s[]) {
```

```
    if (s.length == 0 || s[0].equals("N")) return null;
    Node root = new Node(Integer.parseInt(s[0]));
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    int i = 1;
    while (!q.isEmpty() && i < s.length) {
        Node cur = q.poll();
        String cval = s[i];
        if (!cval.equals("N")) {
            cur.left = new Node(Integer.parseInt(cval));
            q.add(cur.left);
        }
        i++;
        if (i >= s.length) break;
        cval = s[i];
        if (!cval.equals("N")) {
            cur.right = new Node(Integer.parseInt(cval));
            q.add(cur.right);
        }
        i++;
    }
    return root;
}
static void reverseLevelOrder(Node root) {
    if (root == null) return;
    Queue<Node> q = new LinkedList<>();
    List<List<Integer>> levels = new ArrayList<>();
    q.add(root);

    while (!q.isEmpty()) {
        int size = q.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            Node cur = q.poll();
            level.add(cur.data);
            if (cur.right != null) q.add(cur.right);
            if (cur.left != null) q.add(cur.left);
        }
        levels.add(level);
```

```
        }
        for (List<Integer> level : levels) {
            for (int val : level) {
                System.out.print(val + " ");
            }
        }
    }
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        String s[] = sw.nextLine().split(" ");
        Node root = build(s);
        reverseLevelOrder(root);
    }
}
```

Vertical Order Traversal

Intuition:-

Given a Binary Tree, the task is to find its vertical traversal starting from the leftmost level to the rightmost level. If multiple nodes pass through a vertical line, they should be printed as they appear in the level order traversal of the tree.

The goal is to perform a vertical order traversal of a BST, meaning we traverse the tree column-wise from left to right. Each node is assigned a **horizontal distance (HD)**, which is:

- Decreased by 1 when moving left.
- Increased by 1 when moving right.

We use a **HashMap** to store nodes at each horizontal distance, and a **Queue** for level-order traversal.

Key Observations

1. **Use BFS with a Queue:**

- Each node is stored in the queue with its horizontal distance.
- Nodes are processed from left to right in level order.

2. **Maintain Horizontal Distance Mapping:**
   - A HashMap<Integer, ArrayList<Integer>> stores nodes at each horizontal distance.
3. **Edge Cases:**
   - An empty tree should return an empty result.
   - A single node tree should return only one value.
   - Ensure nodes are printed column-wise from left to right.

Code :

```
import java.util.*;
class node {
   int data;
   node left, right;

   node(int n) {
      data = n;
      left = right = null;
   }
}

class pair {
   node first;
   int second;

   pair(node first, int second) {
      this.first = first;
      this.second = second;
   }
}

public class Main {
   static node built(String s[]) {
      if (s.length == 0 || s[0].equals("N")) return null;
```

```
    node root = new node(Integer.parseInt(s[0]));
    Queue<node> q = new LinkedList<>();
    q.add(root);

    int i = 1;
    while (!q.isEmpty() && i < s.length) {
        node cur = q.poll();
        String cval = s[i];

        if (!cval.equals("N")) {
            cur.left = new node(Integer.parseInt(cval));
            q.add(cur.left);
        }
        i++;

        if (i >= s.length) break;
        cval = s[i];

        if (!cval.equals("N")) {
            cur.right = new node(Integer.parseInt(cval));
            q.add(cur.right);
        }
        i++;
    }
    return root;
}

static void pv(node root) {
    if (root == null) return;
    HashMap<Integer, ArrayList<Integer>> m = new HashMap<>();
    Queue<pair> q = new LinkedList<>();
    int hd = 0, mn = 0, mx = 0;
    q.add(new pair(root, hd));
    while (!q.isEmpty()) {
        pair temp = q.poll();
        node cur = temp.first;
        hd = temp.second;
        m.putIfAbsent(hd, new ArrayList<>());
```

```
            m.get(hd).add(cur.data);
            if (cur.left != null) q.add(new pair(cur.left, hd - 1));
            if (cur.right != null) q.add(new pair(cur.right, hd + 1));
            mn = Math.min(mn, hd);
            mx = Math.max(mx, hd);
        }
        for (int i = mn; i <= mx; i++) {
            for (int val : m.get(i)) {
                System.out.print(val + " ");
            }
        }
    }

    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        String s[] = sw.nextLine().split(" ");
        node root = built(s);
        pv(root);
    }
}
```

Time Complexity : O(N)
Space Complexity : O(N)

Online Platform Reference Links:

1. Binary Tree

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Boundary Traversal

Intuition:-

Given a binary tree, the task is to find the boundary nodes of the binary tree Anti-Clockwise starting from the root.
The boundary includes:

- left boundary (nodes on left excluding leaf nodes)
- leaves (consist of only the leaf nodes)
- right boundary (nodes on right excluding leaf nodes)

The left boundary is defined as the path from the root to the left-most leaf node (excluding leaf node itself).
The right boundary is defined as the path from the root to the right-most leaf node (excluding leaf node itself).
Note: If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.

Key Observations

1. **Traversal Order:**
    - Start from the root.
    - Traverse the left boundary (excluding leaf nodes).
    - Traverse all leaf nodes from left to right.
    - Traverse the right boundary (excluding leaf nodes, printed in reverse order).
2. **Left Boundary:**
    - The left boundary is the path from the root to the left-most leaf node.
    - Exclude leaf nodes from the left boundary.
    - If there is no left subtree, the root itself is the left boundary.
3. **Leaf Nodes:**
    - Leaf nodes are nodes with no left or right children.
    - Include all leaf nodes from left to right (both left and right subtrees).
4. **Right Boundary:**
    - The right boundary is the path from the root to the right-most leaf node.
    - Exclude leaf nodes from the right boundary.
    - If there is no right subtree, the root itself is the right boundary.

- Print the right boundary in reverse order to maintain the anti-clockwise direction.

5. **Edge Cases:**
   - Empty Tree: No output.
   - Single Node Tree: Root itself is the left, leaf, and right boundary.
   - No Left Subtree: The root becomes part of the left boundary.
   - No Right Subtree: The root becomes part of the right boundary.

Code :

```
import java.util.*;
class node{
   int data;
   node left;
   node right;
   node(int n){
      data=n;
      left=null;
      right=null;
   }
}
public class Solution
{
        static Stack<Integer> st=new Stack<>();
   static node built(String s[]){
      if(s.length==0||s[0]=="-1") return null;
      node root = new node(Integer.parseInt(s[0]));
      Queue<node> q = new LinkedList<>();
      q.add(root);
      int i=1;
      while(!q.isEmpty()&&i<s.length){
         node cur=q.poll();//1
         String cval = s[i];//2
         if(!cval.equals("-1")){
            cur.left = new node(Integer.parseInt(cval));
            q.add(cur.left);
         }
```

```
        i++;//2
        if(i>=s.length) break;
        cval=s[i];//3
        if(!cval.equals("-1")){
            cur.right = new node(Integer.parseInt(cval));
            q.add(cur.right);
        }
        i++;
    }
    return root;
}
static void pb(node root){
    if(root==null) return;
    System.out.print(root.data+" ");
    lb(root.left);
    leaf(root.left);
    leaf(root.right);
    rb(root.right);
          for(int i=0;i<=st.size();i++){
                System.out.print(st.pop()+" ");
    }
}
static void lb(node root){
    if(root==null) return;
    if(root.left!=null){
        System.out.print(root.data+" ");
        lb(root.left);
    }
    else if(root.right!=null){
        System.out.print(root.data+" ");
        lb(root.right);
    }
}
static void rb(node root){
    if(root==null) return;
    if(root.right!=null){
          st.push(root.data);
    rb(root.right);
    }
```

```
        else if(root.left!=null){
                st.push(root.data);
        rb(root.left);
        }
    }
    static void leaf(node root){
        if(root==null) return;
        leaf(root.left);
        if(root.left==null&&root.right==null){
            System.out.print(root.data+" ");
        }
        leaf(root.right);
    }
    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        String s[]=sw.nextLine().split(" ");
        node root = built(s);
        pb(root);
    }
}
```

Time Complexity : O(N)
Space Complexity : O(H) for recursion (where H is the height of the tree).

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference]() & [Placement Reference]()

BFS [Graph]

Intuition:-

Given an undirected graph represented by an adjacency list adj, where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Traversal (BFS) starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.

BFS (Breadth-First Search) is a traversal algorithm used in graphs . It explores all neighbors of a node before moving to their neighbors. It uses a **queue** to maintain the order of exploration.

Operations:

1. **Initialize:** A queue is used to store nodes to be visited.
2. **Visit Nodes:** Dequeue a node, process it, and enqueue all its unvisited adjacent nodes.
3. **Mark Visited:** Keep track of visited nodes to avoid revisiting and prevent infinite loops in cyclic graphs.
4. **Repeat Until Empty:** Continue dequeuing and visiting until all nodes are processed.

Key Observations

1. **FIFO Principle:**
    - Ensures level-wise traversal in trees and shortest path discovery in unweighted graphs.
2. **Visited Tracking:**
    - An auxiliary array (visited[]) prevents redundant processing.
3. **Edge Cases:**
    - If the graph is disconnected, we may need to run BFS multiple times.
    - Single-node graphs should not cause errors.
    - Large graphs should be handled efficiently with optimized data structures.

Code :

```
import java.util.*;

class Graph {
    LinkedList<Integer> adj[];

    Graph(int v) {
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<>();
        }
    }

    void insertEdge(int s, int d) {
        adj[s].add(d);
        adj[d].add(s); // Assuming an undirected graph
    }

    void bfs(int source) {
        boolean vis[] = new boolean[adj.length];
        Queue<Integer> q = new LinkedList<>();
        q.add(source);
        vis[source] = true;

        while (!q.isEmpty()) {
            int n = q.poll();
            System.out.print(n + " ");
            for (int i : adj[n]) {
                if (!vis[i]) {
                    vis[i] = true;
                    q.add(i);
                }
            }
        }
    }
}

class Main {
    public static void main(String ar[]) {
```

```
        Scanner sc = new Scanner(System.in);
        int v = sc.nextInt();
        int e = sc.nextInt();
        Graph g = new Graph(v);

        for (int i = 0; i < e; i++) {
            int s = sc.nextInt();
            int d = sc.nextInt();
            g.insertEdge(s, d);
        }

        int source = sc.nextInt();
        g.bfs(source);
        sc.close();
    }
}
```

Time Complexity : O(V+E)
Space Complexity : O(V)

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

DFS [Graph]

Intuition:-

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees,

graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

Depth-First Search (DFS) is an algorithm used to traverse or search graph data structures. It explores as far as possible along each branch before backtracking. It can be implemented using recursion or an explicit stack.

Operations:

1. **Graph Representation:**
    - The graph is represented using an adjacency list.
    - Each vertex has a list of adjacent vertices.
2. **DFS Traversal:**
    - A stack is used for iterative DFS.
    - A visited array ensures that each node is processed only once.
    - The traversal starts from a given source vertex and visits nodes in a depth-first manner.

Key Observations

1. **Push with Condition:**
    - The stack ensures nodes are explored in a last-in, first-out (LIFO) order.
    - A node is pushed into the stack if it has not been visited.
2. **Pop with Condition:**
    - The stack is popped when a node is processed, ensuring DFS behavior.
    - Backtracking occurs when no further adjacent unvisited nodes exist.
3. **Edge Cases:**
    - The graph could be disconnected, requiring multiple DFS calls.
    - If the graph has cycles, proper marking of visited nodes is essential.
    - If an isolated vertex (with no edges) is selected as the source, only that vertex is printed.
4. **Preservation:**
    - DFS ensures all reachable nodes from the source are visited.
    - The traversal order depends on the insertion sequence in the adjacency list.

Code :

```
import java.util.*;
class graph{
        LinkedList<Integer> adj[];
        graph(int v){
            adj=new LinkedList[v];
            for(int i=0;i<v;i++){
                adj[i]=new LinkedList<Integer>();
            }
        }
    void Insertedge(int s,int d){
        adj[s].add(d);
        adj[d].add(s);
    }
    void dfs(int source){
        boolean vis[]=new boolean[adj.length];
        Stack<Integer> st = new Stack<>();
        st.add(source);
        vis[source]=true;
        while(!st.isEmpty()){
            int n=st.pop();
            System.out.print(n+" ");
            for(int i:adj[n]){
                if(vis[i]!=true){
                    vis[i]=true;
                    st.add(i);
                }
            }
        }
    }
}
class Main{
    public static void main(String ar[]){
        Scanner sw = new Scanner(System.in);
        int v=sw.nextInt();
        int e=sw.nextInt();
        graph g = new graph(v);
        for(int i=0;i<e;i++){
            int s=sw.nextInt();
            int d=sw.nextInt();
```

```
            g.Insertedge(s,d);
        }
        int source=sw.nextInt();
        g.dfs(source);
    }
}
```

Time Complexity : O(V+E)
Space Complexity : O(V)

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Dial's Algorithm

Intuition:-

Dial's Algorithm is an optimized version of Dijkstra's Algorithm for graphs with small edge weights. It uses a bucket-based priority queue to efficiently find the shortest path from a source node to all other nodes.

Key Observations

1. **Uses Buckets Instead of Priority Queue**
    - Unlike Dijkstra's algorithm (which uses a min-heap), Dial's Algorithm utilizes an array of queues (buckets) to store vertices based on their tentative distance.
2. **Conditions for Efficiency**
    - Works best when edge weights are bounded by a small integer value (maxWeight).
    - Runs in O(V + E + W), where W is the maximum edge weight.
    - More efficient than Dijkstra's algorithm (O((V+E) log V)) for graphs with small edge weights.

3. **Algorithm Steps**
    - **Initialize distances:** Set all distances to ∞, except the source (0).
    - **Bucket Structure:** Maintain an array of queues (buckets) from 0 to maxWeight * V.
    - **Processing Nodes:**
        - Extract nodes from the smallest non-empty bucket.
        - Relax their edges and place updated distances in appropriate buckets.
4. **Edge Cases Handled**
    - If all edges have weight 1, Dial's Algorithm behaves similarly to Breadth-First Search (BFS).
    - If the graph contains an unreachable node, its distance remains ∞.
    - Handles graphs with multiple connected components.

Code :

```
import java.util.*;
class Main {
    static class Node {
        int vertex, weight;
        Node(int v, int w) {
            this.vertex = v;
            this.weight = w;
        }
    }
    public static int[] dialsAlgorithm(List<List<Node>> graph, int src, int maxWeight) {
        int V = graph.size();
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;
        List<Queue<Integer>> buckets = new ArrayList<>();
        for (int i = 0; i <= maxWeight * V; i++) {
            buckets.add(new LinkedList<>());
        }
        buckets.get(0).add(src);
        int index = 0;
        while (index < buckets.size()) {
```

```
            while (!buckets.get(index).isEmpty()) {
                int u = buckets.get(index).poll();
                for (Node neighbor : graph.get(u)) {
                    int v = neighbor.vertex;
                    int weight = neighbor.weight;
                    int newDist = dist[u] + weight;
                    if (newDist < dist[v]) {
                        dist[v] = newDist;
                        buckets.get(newDist).add(v);
                    }
                }
            }
            while (index < buckets.size() && buckets.get(index).isEmpty()) {
                index++;
            }
        }
        return dist;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int V = scanner.nextInt();
        int E = scanner.nextInt();
        int maxWeight = scanner.nextInt();
        List<List<Node>> graph = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            graph.add(new ArrayList<>());
        }
        for (int i = 0; i < E; i++) {
            int u = scanner.nextInt();
            int v = scanner.nextInt();
            int w = scanner.nextInt();
            graph.get(u).add(new Node(v, w));
            graph.get(v).add(new Node(u, w));
        }
        int source = scanner.nextInt();
        scanner.close();
        int[] distances = dialsAlgorithm(graph, source, maxWeight);
        for (int i = 0; i < distances.length; i++) {
                System.out.println("To " + i + " -> " + (distances[i] == Integer.MAX_VALUE ? "INF" :
distances[i]));
        }
    }
}
```

Time Complexity : O(E+WV) (when weights(W) are uniformly small).
Space Complexity : O(V + maxWeight) (for buckets and distance array).

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Bellman Ford Algorithm

Intuition:-

The Bellman-Ford algorithm is used to find the shortest path from a single source to all other vertices in a weighted graph. It is particularly useful for graphs with negative weight edges and can detect negative weight cycles.

Operations

1. **Relaxation Step:**
    - For each vertex, relax all edges (i.e., update the shortest known distance to each vertex) up to (V - 1) times.
    - If dist[u] + weight < dist[v], update dist[v].
2. **Negative Cycle Detection:**
    - If an additional relaxation step still reduces a distance, then a negative weight cycle exists in the graph.

Key Observations

- Works for both directed and undirected graphs, but negative cycles must be carefully handled.
- Unlike Dijkstra's algorithm, it works with negative weights.
- Can detect if a negative weight cycle exists by checking if a relaxation step further reduces a distance after (V - 1) iterations..

1. **Edge Cases Handled:**
    - **Graph with a negative weight cycle:** If a shorter path is found after (V - 1) iterations, a negative weight cycle exists.
    - **Disconnected Graph:** Nodes unreachable from the source should return -1.
    - **Single Vertex Graph:** Should return 0 for the single node.
    - **Multiple paths with different weights:** Should correctly pick the shortest path.

Code :

```
import java.util.*;
class Graph {
    static class Edge {
        int src, dest, weight;

        Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    }
    int V, E;
    List<Edge> edges;
    Graph(int v, int e) {
        V = v;
        E = e;
        edges = new ArrayList<>();
    }
    void BellmanFord(int src) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;
```

```
        for (int i = 0; i < V - 1; i++) {
            for (Edge edge : edges) {
                        if (dist[edge.src] != Integer.MAX_VALUE && dist[edge.src] + edge.weight <
dist[edge.dest]) {
                    dist[edge.dest] = dist[edge.src] + edge.weight;
                }
            }
        }
        for (Edge edge : edges) {
                        if (dist[edge.src] != Integer.MAX_VALUE && dist[edge.src] + edge.weight <
dist[edge.dest]) {
                System.out.println("-1");
                return;
            }
        }
        for (int i = 0; i < V; i++) {
            System.out.print((dist[i] == Integer.MAX_VALUE ? "-1" : dist[i]) + " ");
        }
        System.out.println();
    }
}
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        int m = scanner.nextInt();
        Graph graph = new Graph(n, m);
        for (int i = 0; i < m; i++) {
            int src = scanner.nextInt() ;
            int dest = scanner.nextInt();
            int weight = scanner.nextInt();
            graph.edges.add(new Graph.Edge(src, dest, weight));
        }
        int source = scanner.nextInt();
        graph.BellmanFord(source);
        scanner.close();
    }
}
```

Time Complexity : O(V * E) (since it iterates over all edges (V - 1) times).
Space Complexity : O(V) (to store distances and edges)

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Topological Sort(BFS)

Intuition:-

Topological sorting of a Directed Acyclic Graph (DAG) is a linear ordering of its vertices such that for every directed edge u→v, vertex u appears before v in the ordering. Kahn's Algorithm (BFS-based approach) efficiently computes this ordering.

Steps:

1. **Compute In-degree:**
   - The in-degree of a node is the number of incoming edges.
   - Traverse the adjacency list and compute in-degrees for all nodes.
2. **Initialize Queue:**
   - Enqueue all nodes with an in-degree of 0 (i.e., nodes with no dependencies).
3. **Process Queue:**
   - Dequeue a node, add it to the topological order, and reduce the in-degree of its adjacent nodes.
   - If any adjacent node's in-degree becomes 0, enqueue it.
4. **Detect Cycle (DAG Check):**
   - If the total number of nodes processed is less than NNN (total nodes), a cycle exists.

Key Observations

1. **Edge Cases:**
    - If no valid topological order exists (i.e., the graph contains a cycle).
    - Handling disconnected components in the graph.
2. **Preservation:**
    - Ensures correct topological order is maintained.
    - Cycle detection is achieved by checking if the number of processed nodes is less than N.

Code :

```
import java.util.*;

class Solution {
   public boolean isCyclic(int N, ArrayList<ArrayList<Integer>> adj) {
      int topo[] = new int[N];
      int indegree[] = new int[N];
      for (int i = 0; i < N; i++) {
         for (Integer it : adj.get(i)) {
            indegree[it]++;
         }
      }
      Queue<Integer> q = new LinkedList<Integer>();
      for (int i = 0; i < N; i++) {
         if (indegree[i] == 0) {
            q.add(i);
         }
      }

      int cnt = 0;
      int ind = 0;

      while (!q.isEmpty()) {
         Integer node = q.poll();
         topo[ind++] = node;
         cnt++;
```

```
            for (Integer it : adj.get(node)) {
                indegree[it]--;
                if (indegree[it] == 0) {
                    q.add(it);
                }
            }
        }
        for (int i = 0; i < topo.length; i++) {
            System.out.print(topo[i] + " ");
        }
        System.out.println();
        return cnt != N;
    }
}

public class Main {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
        int E = sc.nextInt();
        ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            adj.add(new ArrayList<>());
        }
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            adj.get(u).add(v);
        }
        boolean hasCycle = new Solution().isCyclic(N, adj);
        System.out.println("Cycle Present: " + hasCycle);
    }
}
```

Time Complexity : O(V + E)
Space Complexity : O(V + E)

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Heap Sort[Max-Heap]

Intuition:-

Heap Sort is a **comparison-based sorting algorithm** that uses the properties of a **Binary Heap** (Max-Heap) to efficiently sort an array. It works in two major steps:

1. **Building a Max-Heap:** Convert the array into a Max-Heap (a complete binary tree where the parent node is always greater than its children).
2. **Extract Elements & Heapify:** Repeatedly swap the root (largest element) with the last element in the heap and reduce the heap size, then **heapify** to maintain the heap structure.

Steps:

1. **Heapify Process:**
    - Start from the last non-leaf node and apply the heapify function (bottom-up approach).
    - Ensures that each subtree satisfies the Max-Heap property (i.e., the parent node is greater than its children).
2. **Heap Sort Process:**
    - Swap the largest element (root of Max-Heap) with the last element.
    - Reduce heap size and reapply heapify.
    - Repeat until all elements are sorted.

Key Observations

1. **Edge Cases:**

- **Already sorted input:** Heap Sort still runs in O(N log N).
- **Reverse sorted input:** Heap Sort efficiently sorts it with the same complexity.
- **Duplicate elements:** Handles them naturally.
- **Single element array:** Returns the same array.
- **Empty array:** No operations performed.

3. **Preservation:**
   - **Heap Sort is not stable** (relative order of equal elements may change).
   - **In-place sorting algorithm** (does not require extra memory like Merge Sort).

Code :

```
import java.util.*;
class Main {
    static void heapify(int arr[], int n, int i) {
        int largest = i;
        int l = 2 * i + 1;
        int r = 2 * i + 2;
        if (l < n && arr[l] > arr[largest]) {
            largest = l;
        }
        if (r < n && arr[r] > arr[largest]) {
            largest = r;
        }
        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            heapify(arr, n, largest);
        }
    }
    static void heapSort(int arr[]) {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }
        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0];
```

```
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
    }
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int arr[] = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        heapSort(arr);
       for (int num : arr) {
            System.out.print(num + " ");
        }
    }}
```

Time Complexity : Building the Heap: O(N)
Heapify (called N times): O(log N)
Total Complexity: O(N log N) (Better than $O(N^2)$ sorting algorithms like Bubble Sort and Insertion Sort)
Space Complexity : O(1) **(In-place sorting)** since it does not require extra memory.

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Heap Sort[Min-Heap]

Intuition:-

Heap Sort using **Min-Heap** is a **comparison-based sorting algorithm** that arranges elements in ascending order. Unlike Max-Heap, where the root contains the largest element, **Min-Heap ensures that the root node is always the smallest element**. The sorting process involves:

1. **Building a Min-Heap** (each parent node is smaller than its children).
2. **Extracting the minimum element** (root) repeatedly and placing it at the end of the array.

Steps:

1. **Heapify Process (Min-Heap):**
    - Convert the array into a Min-Heap by maintaining the heap property.
    - The smallest element is always at the root.
2. **Heap Sort Process:**
    - Swap the smallest element (root of Min-Heap) with the last element.
    - Reduce heap size and reapply heapify to maintain Min-Heap property.
    - Repeat until all elements are sorted in ascending order.

Key Observations

1. **Edge Cases:**
    - **Already sorted input:** Heap Sort still runs in O(N log N).
    - **Reverse sorted input:** Heap Sort efficiently sorts it with the same complexity.
    - **Duplicate elements:** Handles them naturally.
    - **Single element array:** Returns the same array.
    - **Empty array:** No operations performed.
2. **Preservation:**
    - **Heap Sort is not stable** (relative order of equal elements may change).

- **In-place sorting algorithm** (does not require extra memory like Merge Sort).

Code :

```
import java.util.*;

class Main {
    static void heapify(int arr[], int n, int i) {
        int smallest = i;
        int l = 2 * i + 1;
        int r = 2 * i + 2;
        if (l < n && arr[l] < arr[smallest]) {
            smallest = l;
        }
        if (r < n && arr[r] < arr[smallest]) {
            smallest = r;
        }

        if (smallest != i) {
            int temp = arr[i];
            arr[i] = arr[smallest];
            arr[smallest] = temp;
            heapify(arr, n, smallest);
        }
    }
    static void heapSort(int arr[]) {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }
        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
    }
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int arr[] = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        heapSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity : Building the Heap: O(N)
Heapify (called N times): O(log N)
Total Complexity: O(N log N) (Better than $O(N^2)$ sorting algorithms like Bubble Sort and Insertion Sort)
Space Complexity : O(1) **(In-place sorting)** since it does not require extra memory.

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

K-ary Heap

Intuition:-

A **K-ary Heap** is a generalization of a **binary heap,** where each node has at most **K children** instead of just two. This is particularly useful when working with data structures that require efficient priority queue operations with a different branching factor.

Steps:

1. **Parent-Child Relationship**
    - For a 0-based index, the parent of node i is at (i - 1) / K.
    - The children of node i are at K * i + 1, K * i + 2, ..., K * i + K.
2. **Heap Property**
    - **Max-Heap:** The parent node is greater than or equal to all its children.
    - **Min-Heap:** The parent node is smaller than or equal to all its children.
3. **Operations:**
    - **Build Heap:** Convert an array into a valid K-ary heap.
    - **Insert Element:** Add a new element while maintaining the heap property.
    - **Extract Max:** Remove and return the maximum element while maintaining the heap property.

Key Observations

1. **Edge Cases:**
    - If **K = 2**, the K-ary heap behaves like a standard binary heap.
    - If the heap is empty, operations like extract-max should handle it gracefully.
    - For large values of **K**, the height of the heap decreases, leading to fewer levels and potentially faster insertions and deletions.
2. **Preservation:**
    - Priority Queues (e.g., Dijkstra's Algorithm).
    - Multi-way merging (e.g., External Sorting).
    - Scheduling Processes in Operating Systems.

Code :

```
import java.util.*;

public class KaryHeap {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int k = scanner.nextInt();
        int n = scanner.nextInt();
        int[] arr = new int[100];
```

```
    for (int i = 0; i < n; i++) {
        arr[i] = scanner.nextInt();
    }

    buildHeap(arr, n, k);
    printHeap(arr, n);

    int element = scanner.nextInt();
    insert(arr, n, k, element);
    n++;
    printHeap(arr, n);

    System.out.println(extractMax(arr, n, k));
    n--;
    printHeap(arr, n);

    scanner.close();
}

public static void buildHeap(int[] arr, int n, int k) {
    for (int i = n / k - 1; i >= 0; i--) {
        restoreDown(arr, n, i, k);
    }
}

public static void insert(int[] arr, int n, int k, int elem) {
    arr[n] = elem;
    restoreUp(arr, n, k);
}

public static int extractMax(int[] arr, int n, int k) {
    int max = arr[0];
    arr[0] = arr[n - 1];
    restoreDown(arr, n - 1, 0, k);
    return max;
}

public static void restoreDown(int[] arr, int len, int index, int k) {
    int[] child = new int[k + 1];
    while (true) {
        for (int i = 1; i <= k; i++) {
            child[i] = (k * index + i) < len ? (k * index + i) : -1;
        }
        int maxChild = -1, maxChildIndex = 0;
```

```
            for (int i = 1; i <= k; i++) {
                if (child[i] != -1 && arr[child[i]] > maxChild) {
                    maxChildIndex = child[i];
                    maxChild = arr[child[i]];
                }
            }
            if (maxChild == -1) {
                break;
            }
            if (arr[index] < arr[maxChildIndex]) {
                swap(arr, index, maxChildIndex);
            }
            index = maxChildIndex;
        }
    }

    public static void restoreUp(int[] arr, int index, int k) {
        int parent = (index - 1) / k;
        while (parent >= 0) {
            if (arr[index] > arr[parent]) {
                swap(arr, index, parent);
                index = parent;
                parent = (index - 1) / k;
            } else {
                break;
            }
        }
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static void printHeap(int[] arr, int n) {
        for (int i = 0; i < n; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

Time Complexity :

Operation	Complexity
Build Heap	O(N)
Insert	O(log_K N)
Extract Max	O(log_K N)

- **Build Heap** is **O(N)** since we restore down from the bottom half of the heap.
- **Insert and Extract Max are O(log_K N)** since the heap height reduces as K increases.

Space Complexity : O(N)for storing the heap.

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Binomial Heap

Intuition:-

A **Binomial Heap** is a collection of Binomial Trees that follow certain properties:

- Min-Heap Property: The key of each node is smaller than or equal to the keys of its children.
- Unique Order of Trees: A binomial heap contains at most one binomial tree of each order.
- Sorted by Rank: The binomial trees are arranged in increasing order of their degrees.

Finding the Number of Binomial Trees in a Binomial Heap

Given n nodes in a binomial heap, the number of binomial trees present can be determined using:

- Formula: $\text{floor}(\log_2(n)) + 1$
- The binomial heap structure is based on binary representation.
- The binomial trees in the heap correspond to the set bits (1s) in the binary representation of n.

Identifying Binomial Trees in a Binomial Heap

To determine which binomial trees exist, convert n to binary. Each bit position where there is a 1 corresponds to a binomial tree of that order.

Example:

- If n = 13, its binary representation is 1101.
- The positions of 1s indicate the presence of binomial trees of order 0, 2, and 3.
- So, the heap consists of B_0, B_2, and B_3 binomial trees.

Advantages of Binomial Heap

- Efficient merge operation with a time complexity of O(log n), significantly better than binary heaps.
- Supports insertion, deletion, and extraction of the minimum element with efficient operations.
- Useful in priority queue applications where quick merging is required.

Operations in Binomial Heap

Merge Operation

- The merge operation is a straightforward process where two binomial heaps are combined by simply linking their trees in non-decreasing order of degrees.
- This is similar to merging two sorted linked lists.
- Merge does not resolve duplicate tree degrees; it only combines the heaps into one structure.

Union Operation

- The union operation first merges two heaps and then ensures that the binomial heap properties are maintained.
- Trees of the same degree are linked together, maintaining at most one binomial tree per order.
- This requires adjustments to balance the heap properly.

Example:

Consider two binomial heaps:
Heap 1: B_0, B_2
Heap 2: B_1, B_2

- Merge results in: B_0, B_1, B_2, B_2
- Union then links the duplicate B_2 trees, resulting in: B_0, B_1, B_3 (after combining two B_2 trees)

1. **Insert Operation (O(1) Amortized Time Complexity)**
   - A new key is inserted as a singleton tree (binomial tree of order 0).
   - The new tree is merged with the existing binomial heap.
2. **Find Minimum (O(log n) Time Complexity)**
   - The minimum key in a binomial heap is always located at the root of one of the binomial trees.
   - Traverse through the root list to find the minimum key.
3. **Extract Minimum (O(log n) Time Complexity)**
   - Find and remove the minimum node.
   - Break its children into separate binomial trees and merge them back into the heap.
4. **Union/Merge (O(log n) Time Complexity)**
   - Merging two binomial heaps follows a similar approach to merging sorted linked lists.
   - Trees of the same degree are combined to maintain binomial heap properties.

5. **Decrease Key (O(log n) Time Complexity)**
   - Decrease the key of a node.
   - If heap property is violated, perform a bubble-up operation by swapping with the parent node.
6. **Delete Node (O(log n) Time Complexity)**
   - Decrease the node's value to negative infinity and perform an exact minimum.

Code :

```
import java.util.*;
class Node {
    int value, degree;
    Node parent, sibling;
    List<Node> children;

    public Node(int value) {
        this.value = value;
        this.degree = 0;
        this.parent = null;
        this.sibling = null;
        this.children = new ArrayList<>();
    }
}

class BinomialHeap {
    Node head;
    Node minNode;

    public BinomialHeap() {
        this.head = null;
        this.minNode = null;
    }

    private void linkTrees(Node root1, Node root2) {
        root1.parent = root2;
        root2.children.add(root1);
        root2.degree += 1;
        root1.sibling = null;
    }

    private Node mergeHeaps(Node root1, Node root2) {
        if (root1 == null) return root2;
```

```
    if (root2 == null) return root1;

    Node head, tail;
    if (root1.degree <= root2.degree) {
        head = root1;
        root1 = root1.sibling;
    } else {
        head = root2;
        root2 = root2.sibling;
    }
    tail = head;

    while (root1 != null && root2 != null) {
        if (root1.degree <= root2.degree) {
            tail.sibling = root1;
            root1 = root1.sibling;
        } else {
            tail.sibling = root2;
            root2 = root2.sibling;
        }
        tail = tail.sibling;
    }

    if (root1 != null) tail.sibling = root1;
    else tail.sibling = root2;

    return head;
}

public void insert(int value) {
    Node newNode = new Node(value);
    BinomialHeap tempHeap = new BinomialHeap();
    tempHeap.head = newNode;
    unionHeap(tempHeap);
}

public void unionHeap(BinomialHeap heap) {
    this.head = mergeHeaps(this.head, heap.head);
    if (this.head == null) return;

    Node prev = null, curr = this.head, next = curr.sibling;

    while (next != null) {
        if ((curr.degree != next.degree) ||
```

```
            (next.sibling != null && next.sibling.degree == curr.degree)) {
            prev = curr;
            curr = next;
        } else if (curr.value <= next.value) {
            curr.sibling = next.sibling;
            linkTrees(next, curr);
        } else {
            if (prev == null) this.head = next;
            else prev.sibling = next;
            linkTrees(curr, next);
            curr = next;
        }
        next = curr.sibling;
    }

    this.minNode = this.head;
    Node temp = this.head;
    while (temp != null) {
        if (temp.value < this.minNode.value) this.minNode = temp;
        temp = temp.sibling;
    }
}

public int extractMin() {
    if (this.minNode == null) {
        System.out.println("Heap is empty.");
        return -1;
    }

    Node minNode = this.minNode;
    Node prev = null, curr = this.head;

    while (curr != minNode) {
        prev = curr;
        curr = curr.sibling;
    }

    if (prev == null) this.head = curr.sibling;
    else prev.sibling = curr.sibling;

    List<Node> reversedChildren = new ArrayList<>();
    for (Node child : minNode.children) {
        child.parent = null;
        reversedChildren.add(0, child);
```

```
        }

        BinomialHeap tempHeap = new BinomialHeap();
        if (!reversedChildren.isEmpty()) {
            for (int i = 0; i < reversedChildren.size() - 1; i++) {
                reversedChildren.get(i).sibling = reversedChildren.get(i + 1);
            }
            tempHeap.head = reversedChildren.get(0);
        }

        this.unionHeap(tempHeap);
        return minNode.value;
    }

    public void printTree(Node root, String indent) {
        if (root == null) return;
        System.out.println(indent + root.value);
        for (Node child : root.children) {
            printTree(child, indent + "  ");
        }
    }

    public void printHeap() {
        if (this.head == null) {
            System.out.println("Heap is empty.");
            return;
        }
        Node curr = this.head;
        while (curr != null) {
            System.out.println("Binomial Tree B" + curr.degree + ":");
            printTree(curr, "");
            System.out.println();
            curr = curr.sibling;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        BinomialHeap heap = new BinomialHeap();

        while (true) {
            System.out.println("\nMenu:");
```

```
        System.out.println("1. Insert Key");
        System.out.println("2. Find Minimum");
        System.out.println("3. Extract Minimum");
        System.out.println("4. Print Heap");
        System.out.println("5. Exit");
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                System.out.print("Enter the key to insert: ");
                int key = scanner.nextInt();
                heap.insert(key);
                break;
            case 2:
                                if (heap.minNode != null) System.out.println("Minimum Key: " +
heap.minNode.value);
                else System.out.println("Heap is empty.");
                break;
            case 3:
                int minKey = heap.extractMin();
                if (minKey != -1) System.out.println("Extracted Minimum: " + minKey);
                break;
            case 4:
                heap.printHeap();
                break;
            case 5:
                System.out.println("Exiting...");
                scanner.close();
                return;
            default:
                System.out.println("Invalid choice! Try again.");
        }
    }
  }
}
```

Time Complexity :

Space Complexity :

The space complexity of a binomial heap is **O(n)** since each node is stored in memory. During operations like **extracting the minimum**, the additional space required is at most **O(logn)** for storing child trees temporarily.

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference

Winner Tree

Intuition:

A **Winner Tree** is a **complete binary tree** used for tournament-style comparisons, particularly for selecting the minimum (or maximum) element efficiently. It is commonly used in **multiway merging** (e.g., **k-way merge sort**).

- Each **leaf node** represents an input element.
- Each **internal node** stores the winner (minimum or maximum) of its two children.
- The **root node** contains the overall winner (minimum/maximum of the entire array).

Steps for Building a Winner Tree:

1. **Initialize Tree:**
    - Create an array of size (2k−1), where k is the number of input elements.
    - Store the input values in the leaf nodes (last k positions in the array).
2. **Build the Tree:**
    - Start from the second-last level (i.e., k−2) and move up.
    - Compare **left** and **right child nodes** and store the winner (minimum/maximum) in the parent node.
3. **Output the Winner:**
    - The **root node** contains the final winner.

Key Points:

1. **Efficiency:**
    - Tree Construction: O(k)

- Winner Retrieval: O(1)
- Update Operations (if an element is modified): O(logk)

2. **Space Complexity:**
   - Uses an array representation of the binary tree, requiring O(k) space.
3. **Edge Cases:**
   - If all elements are the same, the root will still contain that repeated value.
   - Handles scenarios where an element is repeatedly modified and requires an update in the tree.
4. **Preservation:**
   - Ensures the correct winner is maintained after each comparison.
   - If elements are updated, only the affected part of the tree is modified, keeping updates efficient.

Code 1 :

```
import java.util.*;
class Main{
   public static void main(String args[])
   {
      Scanner sc=new Scanner(System.in);
      int k=sc.nextInt();
      int tree[]=new int[(2*k)-1];
      for(int i=k-1;i<(2*k)-1;i++)
      {
         tree[i]=sc.nextInt();
      }
      for(int i=k-2;i>=0;i--){
      if(tree[(2*i)+1]<tree[(2*i)+2])
      {
         tree[i]=tree[(2*i)+1];
      }
      else{
         tree[i]=tree[(2*i)+2];
      }
   }
   System.out.println(tree[0]);
}
}
```

Time Complexity : Building the tree: O(k)
Querying the winner: O(1)
Updating a node: O(logk)
Space Complexity : **O(k)** (array-based storage).

Code 2 :

```
import java.util.*;

public class Main {
    static int K;
    static List<Integer> heap = new ArrayList<>();

    private static int parent(int i) {
        return (i - 1) / K;
    }

    private static int child(int i, int k) {
        return K * i + k;
    }

    public static void insert(int value) {
        heap.add(value);
        int index = heap.size() - 1;
        while (index > 0 && heap.get(parent(index)) < heap.get(index)) {
            Collections.swap(heap, index, parent(index));
            index = parent(index);
        }
    }

    private static void heapify(int i) {
        int largest = i;
        for (int j = 1; j <= K; j++) {
            int childIndex = child(i, j);
            if (childIndex < heap.size() && heap.get(childIndex) > heap.get(largest)) {
                largest = childIndex;
            }
        }
        if (largest != i) {
            Collections.swap(heap, i, largest);
            heapify(largest);
```

```
        }
    }

    public static int extractMax() {
        if (heap.isEmpty()) throw new NoSuchElementException("Heap is empty");
        int max = heap.get(0);
        heap.set(0, heap.remove(heap.size() - 1));
        heapify(0);
        return max;
    }

    public static void main(String[] args) {
        Scanner sw = new Scanner(System.in);
        K = sw.nextInt();
        int n = sw.nextInt();
        for (int i = 0; i < n; i++) insert(sw.nextInt());
        System.out.println(extractMax());
    }
}
```

Time Complexity: O(log n base k)
Space Complexity: O(1) (modifies in place)

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference]() & [Placement Reference]()

Hash Map to Tree Map

Intuition:

A **HashMap** is a data structure that stores key-value pairs without maintaining any order of keys. On the other hand, a **TreeMap** is a Red-Black Tree-based implementation of the **Map** interface that automatically sorts the keys in natural order or by a custom comparator.

Converting from a **HashMap** to a **TreeMap** is useful when:

- You need the data to be retrieved in sorted key order.
- You want to perform range-based operations (like submaps).
- You prefer log(n) time complexity for ordered retrieval.

Steps for Conversion:

1. **Input Initialization:**
    - Read key-value pairs from the user and store them in a HashMap.
2. **Conversion Approaches:**

I. Using TreeMap Constructor:

- Pass the entire HashMap into the TreeMap constructor.

II. Using putAll() Method:

- Create an empty TreeMap and copy all elements using **putAll().**

III. Using For-Each Loop:

- Iterate over the HashMap and insert each entry into the TreeMap manually.

Key Points:

- TreeMap stores keys in sorted order (Comparable or custom Comparator).
- All three methods preserve values and result in sorted key-value pairs.
- This is especially helpful in scenarios where data needs to be displayed in sorted form.

Code 1 :

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        HashMap<String, Integer> map = new HashMap<>();
        int n = sc.nextInt();
        sc.nextLine();
        for (int i = 0; i < n; i++) {
            String key = sc.nextLine();
            int value = sc.nextInt();
            sc.nextLine();
            map.put(key, value);
        }
        TreeMap<String, Integer> sortedMap = new TreeMap<>(map);
        System.out.println(sortedMap);
    }
}
```

Code 2 :

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        HashMap<String, Integer> map = new HashMap<>();
        int n = sc.nextInt();
        sc.nextLine();
        for (int i = 0; i < n; i++) {
            String key = sc.nextLine();
            int value = sc.nextInt();
            sc.nextLine();
            map.put(key, value);
        }
        TreeMap<String, Integer> sortedMap = new TreeMap<>();
        sortedMap.putAll(map);
        System.out.println(sortedMap);
    }
}
```

Code 3 :

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        HashMap<String, Integer> map = new HashMap<>();
        int n = sc.nextInt();
        sc.nextLine();
        for (int i = 0; i < n; i++) {
            String key = sc.nextLine();
            int value = sc.nextInt();
            sc.nextLine();
            map.put(key, value);
        }
        TreeMap<String, Integer> sortedMap = new TreeMap<>();
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            sortedMap.put(entry.getKey(), entry.getValue());
        }
        System.out.println(sortedMap);
    }
}
```

Time Complexity : Insertion into HashMap: O(n)
Conversion to TreeMap:
Constructor / putAll(): O(n log n)
For-each insertion: O(n log n)

Space Complexity : **O(n) —** to store both maps

Online Platform Reference Links:

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference .

Types of Sets

Intuition:-

In Java, Sets are used to store **unique elements** only. Java provides three commonly used implementations of the **Set** interface:

- **HashSet:** Does **not maintain any order** of elements.
- **LinkedHashSet:** Maintains **insertion order** of elements.
- **TreeSet:** Maintains elements in **sorted (natural) order**.

Each type of Set offers different performance characteristics and internal data structure behavior. By applying the same input to all sets, we can clearly observe how they differ.

Steps to Demonstrate Types of Sets:

1. **Input Collection**
    - Accept n strings from the user via Scanner.
    - Store them in a base HashSet.
2. **Create Different Sets**
    - Use the same input to initialize all 3 sets:
        - HashSet
        - LinkedHashSet
        - TreeSet
3. **Perform Set Operations**
    - Add elements
    - Remove elements
    - Check for existence using contains()
    - Check size and emptiness
    - Traverse each set
    - Perform:
        - Union
        - Intersection
        - Difference
    - Clear a set and print the result.

Key Observations

- HashSet uses hashing – fastest for add/remove/contains but no ordering.
- LinkedHashSet maintains insertion order.
- TreeSet uses a Red-Black Tree for sorted order and allows efficient navigation.
- All sets store only unique elements (no duplicates allowed).

Efficiency:

Operation	**HashSet**	**LinkedHashSet**	**TreeSet**
add(), remove()	O(1) avg	O(1) avg	O(log n)
contains()	O(1) avg	O(1) avg	O(log n)
iteration order	unordered	insertion	sorted
space	O(n)	O(n)	O(n)

Edge Cases:

- Duplicate elements are automatically filtered.
- Removing non-existing elements has no effect.
- **clear()** empties the set entirely.
- Union/Intersection with empty sets behave as per set theory.

Code :

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        Set<String> inputData = new HashSet<>();
        for (int i = 0; i < n; i++) {
            inputData.add(sc.next());
        }
        Set<String> hashSet = new HashSet<>(inputData);
        Set<String> linkedHashSet = new LinkedHashSet<>(inputData);
        Set<String> treeSet = new TreeSet<>(inputData);

        System.out.println("HashSet: " + hashSet);
        System.out.println("LinkedHashSet: " + linkedHashSet);
        System.out.println("TreeSet: " + treeSet);

        hashSet.add("Orange");
        linkedHashSet.add("Orange");
        treeSet.add("Orange");

        hashSet.remove("Mango");
        linkedHashSet.remove("Mango");
        treeSet.remove("Mango");

        System.out.println("HashSet contains 'Apple': " + hashSet.contains("Apple"));
        System.out.println("LinkedHashSet contains 'Apple': " + linkedHashSet.contains("Apple"));
        System.out.println("TreeSet contains 'Apple': " + treeSet.contains("Apple"));

        System.out.println("HashSet size: " + hashSet.size());
        System.out.println("LinkedHashSet size: " + linkedHashSet.size());
        System.out.println("TreeSet size: " + treeSet.size());

        System.out.println("HashSet is empty: " + hashSet.isEmpty());
        System.out.println("LinkedHashSet is empty: " + linkedHashSet.isEmpty());
        System.out.println("TreeSet is empty: " + treeSet.isEmpty());

        System.out.println("HashSet elements:");
        for (String s : hashSet) System.out.println(s);
```

```
        System.out.println("LinkedHashSet elements:");
        for (String s : linkedHashSet) System.out.println(s);

        System.out.println("TreeSet elements:");
        for (String s : treeSet) System.out.println(s);

        Set<String> union = new HashSet<>(hashSet);
        union.addAll(linkedHashSet);
        System.out.println("Union of HashSet and LinkedHashSet: " + union);

        Set<String> intersection = new HashSet<>(hashSet);
        intersection.retainAll(treeSet);
        System.out.println("Intersection of HashSet and TreeSet: " + intersection);

        Set<String> difference = new HashSet<>(hashSet);
        difference.removeAll(treeSet);
        System.out.println("Difference of HashSet - TreeSet: " + difference);

        hashSet.clear();
        System.out.println("HashSet after clear(): " + hashSet);
    }
}
```

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference] & [Placement Reference]

Distributing items when a person cannot take more than two items of the same type.

Intuition:-

When distributing items among k people, each person cannot receive more than two items of the same type.
This means that no item type can appear more than 2 * k times in the distribution.
To validate this:

- Count the frequency of each item using a HashMap.
- Check if any frequency exceeds 2 * k. If it does, the distribution is invalid.

Steps to Demonstrate Types of Sets:

1. Read the number of people k and total items n.
2. Read array of n integers (item types).
3. Use HashMap to count occurrences of each item.
4. If any item count exceeds 2 * k, return NOT VALID.
5. Else, return VALID.

Key Observations

- Uses HashMap for frequency counting.
- Simple logic to check maximum allowed items of each type.
- Handles all integer types for items.

Edge Cases:

- All items are the same → invalid if count > 2*k.
- Array is empty → always valid.
- Some items exactly appear 2*k times → still valid.
- Multiple item types, mix of valid and invalid frequencies.

Code :

```
import java.util.*;
public class Main {
    public static boolean checkValidDistribution(int[] arr, int n, int k) {
        HashMap<Integer, Integer> hash = new HashMap<>();
        for (int i = 0; i < n; i++) {
            hash.put(arr[i], hash.getOrDefault(arr[i], 0) + 1);
        }
        for (Map.Entry<Integer, Integer> entry : hash.entrySet()) {
            if (entry.getValue() > 2 * k) {
                return false;
            }
```

```
        }
        return true;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int k = sc.nextInt();
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        if (checkValidDistribution(arr, n, k)) {
            System.out.println("Distribution is VALID.");
        } else {
            System.out.println("Distribution is NOT VALID. Some items occur more than 2*k times.");
        }
    }
}
```

Operation	**Complexity**
Frequency Counting	O(n)
Validation Check	O(m), m = unique item types
Overall Time	O(n)
Space (HashMap)	O(m)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference] & [Placement Reference]

Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique used to solve complex problems by breaking them down into simpler subproblems. It is particularly effective for problems exhibiting overlapping subproblems and **optimal substructure** properties. In DP, each subproblem is solved once, and its result is stored for future reference, eliminating the need for redundant computations.

Why Use Dynamic Programming?

Recursion is a straightforward approach to solving problems by breaking them down into smaller subproblems. However, it can lead to redundant computations and increased time complexity, especially when the same subproblems are solved multiple times. Dynamic Programming addresses this inefficiency by storing the results of subproblems (a method known as memoization), ensuring each subproblem is solved only once. This optimization significantly reduces the time complexity compared to naive recursive approaches.

Recursion Drawbacks:

- Redundant calls for the same subproblems.
- Exponential time complexity.
- Stack overflow risk for large n.

DP Advantages:

- Avoids recomputation by storing subproblem results.
- Reduces time complexity to linear.
- Optimized further with tabulation and space optimization.

Types of Dynamic Programming Approaches:

1. **Top-Down Approach (Memoization):**
    - This approach starts solving the main problem and breaks it down into subproblems. If the solution to a subproblem is already known (stored in memory), it is reused; otherwise, the subproblem is solved, and its result is stored for future use. This method works from the original problem towards the base cases.

2. **Bottom-Up Approach (Tabulation):**
    - In this approach, the problem is solved by first addressing the simplest subproblems and combining their solutions to solve more complex subproblems. It starts from the base cases and works its way up to the solution of the main problem, typically using an iterative process.

Key Point:

- In the **Top-Down Approach**, the solution progresses from the main problem towards the base condition.
- In the **Bottom-Up Approach**, the solution builds up from the base conditions to the main problem.

Fibonacci Series

Intuition:-

The Fibonacci sequence is a series where each number is the sum of the two preceding ones, typically starting with 0 and 1. It appears in various natural phenomena and algorithmic problems.

To generate the full series rather than just the nth Fibonacci number, we calculate and store each value from the beginning up to n.

- **Recursive Approach**: Simple but inefficient due to repeated calls.
- **Memoization (Top-Down DP)**: Avoids redundant computation by storing results.
- **Tabulation (Bottom-Up DP)**: Builds the result from base cases iteratively.
- **Space Optimization**: Uses only two variables instead of arrays.

Steps:

1. **Recursive:**
    - Call the function fibonacci(n) recursively to get the nth number.
2. **Memoization:**
    - Use an array to store already computed Fibonacci numbers and avoid recomputation.

3. **Tabulation:**
    - Initialize base cases fib[0] = 0, fib[1] = 1 and fill the array from 2 to n.
4. **Space Optimization:**
    - Track only the last two computed values and update them iteratively.

Key Observations

1. **Edge Cases:**
    - **n = 0:** Should output nothing.
    - **n = 1:** Outputs **0.**
    - Duplicate results are avoided in memoization & tabulation.
    - Recursion gives exponential time, DP reduces it to linear.
2. **Preservation:**
    - All approaches generate the same output.
    - Memoization, Tabulation, and Space Optimization drastically improve time and/or space usage.

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
   public static int fibonacci(int n) {
      if (n <= 1) return n;
      return fibonacci(n - 1) + fibonacci(n - 2);
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      for (int i = 0; i < n; i++) {
         System.out.print(fibonacci(i) + " ");
      }
   }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;
public class Main {
   static int[] memo;
```

```
    public static int fibonacci(int n) {
        if (n <= 1) return n;
        if (memo[n] != -1) return memo[n];
        memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
        return memo[n];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        memo = new int[n + 1];
        Arrays.fill(memo, -1);
        for (int i = 0; i < n; i++) {
            System.out.print(fibonacci(i) + " ");
        }
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if (n == 0) return;
        int[] fib = new int[n];
        fib[0] = 0;
        if (n > 1) fib[1] = 1;
        for (int i = 2; i < n; i++) {
            fib[i] = fib[i - 1] + fib[i - 2];
        }
        for (int i = 0; i < n; i++) {
            System.out.print(fib[i] + " ");
        }
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if (n > 0) System.out.print(0 + " ");
        if (n > 1) System.out.print(1 + " ");
        int a = 0, b = 1;
        for (int i = 2; i < n; i++) {
            int c = a + b;
            System.out.print(c + " ");
            a = b;
            b = c;
        }
    }
}
```

Time & Space Complexities:

Approach	**Time Complexity**	**Space Complexity**
Recursion	O(2^n)	O(n) (rec. stack)
Memoization (Top)	O(n)	O(n)
Tabulation (Bottom)	O(n)	O(n)
Space Optimized	O(n)	O(1)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference] & [Placement Reference]

[Dynamic Programming Series]

Longest Common Subsequence

Intuition:-

The **Longest Common Subsequence (LCS)** problem involves finding the length of the longest subsequence common to two given strings. A subsequence is a sequence that appears in the same relative order but not necessarily contiguous. LCS has applications in diff tools, bioinformatics (DNA sequencing), and version control systems.

Steps:

1. **Recursive:**
    - Compare characters from the end of both strings. If they match, add 1 to the result and move both indices back. If not, move one index at a time and take the maximum.
2. **Memoization:**
    - Store the results of subproblems in a 2D array to avoid redundant calculations.
3. **Tabulation:**
    - Iteratively build a 2D DP table from the base cases up to the desired indices.
4. **Space Optimization:**
    - Since only the previous row is needed at any step, use two 1D arrays to save space.

Key Observations

1. **Edge Cases:**
    - If either string is empty, the LCS length is 0.
    - If both strings are identical, the LCS is the length of either string.
2. **Preservation:**
    - All approaches yield the same result. Memoization and Tabulation improve time efficiency, while Space Optimization reduces space usage.

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
    public static int lcs(String s1, String s2, int i, int j) {
        if (i == 0 || j == 0) return 0;
        if (s1.charAt(i - 1) == s2.charAt(j - 1))
            return 1 + lcs(s1, s2, i - 1, j - 1);
        else
            return Math.max(lcs(s1, s2, i - 1, j), lcs(s1, s2, i, j - 1));
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s1 = sc.nextLine();
        String s2 = sc.nextLine();
        int n = s1.length();
        int m = s2.length();
        System.out.println(lcs(s1, s2, n, m));
    }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;
public class Main {
    static int[][] dp;
    public static int lcs(String s1, String s2, int i, int j) {
        if (i == 0 || j == 0) return 0;
        if (dp[i][j] != -1) return dp[i][j];
        if (s1.charAt(i - 1) == s2.charAt(j - 1))
            return dp[i][j] = 1 + lcs(s1, s2, i - 1, j - 1);
        else
            return dp[i][j] = Math.max(lcs(s1, s2, i - 1, j), lcs(s1, s2, i, j - 1));
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s1 = sc.nextLine();
        String s2 = sc.nextLine();
        int n = s1.length();
        int m = s2.length();
        dp = new int[n + 1][m + 1];
```

```
        for (int[] row : dp)
            Arrays.fill(row, -1);
        System.out.println(lcs(s1, s2, n, m));
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s1 = sc.nextLine();
        String s2 = sc.nextLine();
        int n = s1.length();
        int m = s2.length();
        int[][] dp = new int[n + 1][m + 1];
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1))
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
        System.out.println(dp[n][m]);
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s1 = sc.nextLine();
        String s2 = sc.nextLine();
        int n = s1.length();
        int m = s2.length();
        int[] prev = new int[m + 1];
```

```
        int[] curr = new int[m + 1];
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1))
                    curr[j] = 1 + prev[j - 1];
                else
                    curr[j] = Math.max(prev[j], curr[j - 1]);
            }
            int[] temp = prev;
            prev = curr;
            curr = temp;
        }
        System.out.println(prev[m]);
    }
}
```

Time & Space Complexities:

Approach	**Time Complexity**	**Space Complexity**
Recursive	O(2^(n+m))	O(n + m)
Memoization	O(n * m)	O(n * m)
Tabulation	O(n * m)	O(n * m)
Space Optimization	O(n * m)	O(m)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference Dynamic Programming Series

Longest Palindromic Subsequence

Intuition:-

The **Longest Palindromic Subsequence (LPS)** problem seeks the longest subsequence of a string that reads the same forward and backward. It's different from the **Longest Palindromic Substring**, which requires contiguity.

A key insight is:

> LPS of a string `s` is the **Longest Common Subsequence (LCS)** of `s` and its **reverse**.

Steps:

1. **Recursive:**
   - Compare characters at both ends. If they match, include them and move inward. Else, try excluding one character at a time and take the maximum result.
2. **Memoization:**
   - Store overlapping subproblems in a 2D array to eliminate recomputation.
3. **Tabulation:**
   - Build a 2D table iteratively using the LCS principle on the original and reversed string.
4. **Space Optimization:**
   - Use only two 1D arrays to reduce space complexity while following the LCS concept.

Key Observations

1. **Edge Cases:**
   - Empty string: LPS = 0
   - All characters same: LPS = string length
   - No palindromic subsequence beyond single letters: LPS = 1
2. **Preservation:**
   - All approaches produce the same result.
   - DP methods optimize for time/space.

- Space Optimization is best when memory is critical.

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
    public static int lps(String s, int i, int j) {
        if (i > j) return 0;
        if (i == j) return 1;
        if (s.charAt(i) == s.charAt(j))
            return 2 + lps(s, i + 1, j - 1);
        else
            return Math.max(lps(s, i + 1, j), lps(s, i, j - 1));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        int n = s.length();
        System.out.println(lps(s, 0, n - 1));
    }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;

public class Main {
    static int[][] dp;
    public static int lps(String s, int i, int j) {
        if (i > j) return 0;
        if (i == j) return 1;
        if (dp[i][j] != -1) return dp[i][j];
        if (s.charAt(i) == s.charAt(j))
            return dp[i][j] = 2 + lps(s, i + 1, j - 1);
        else
            return dp[i][j] = Math.max(lps(s, i + 1, j), lps(s, i, j - 1));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
        String s = sc.nextLine();
        int n = s.length();
        dp = new int[n][n];
        for (int[] row : dp)
            Arrays.fill(row, -1);
        System.out.println(lps(s, 0, n - 1));
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        int n = s.length();
        int[][] dp = new int[n][n];

        for (int i = 0; i < n; i++)
            dp[i][i] = 1;

        for (int i = n - 1; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                if (s.charAt(i) == s.charAt(j))
                    dp[i][j] = 2 + dp[i + 1][j - 1];
                else
                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
        System.out.println(dp[0][n - 1]);
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
```

```
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        int n = s.length();
        int[] prev = new int[n];
        int[] curr = new int[n];

        for (int i = n - 1; i >= 0; i--) {
            curr[i] = 1;
            for (int j = i + 1; j < n; j++) {
                if (s.charAt(i) == s.charAt(j))
                    curr[j] = 2 + prev[j - 1];
                else
                    curr[j] = Math.max(prev[j], curr[j - 1]);
            }
            int[] temp = prev;
            prev = curr;
            curr = temp;
        }
        System.out.println(prev[n - 1]);
    }
}
```

Time & Space Complexities:

Approach	**Time Complexity**	**Space Complexity**
Recursive	O(2^n)	O(n)
Memoization	O(n²)	O(n²)
Tabulation	O(n²)	O(n²)
Space Optimization	O(n²)	O(n)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference] & [Placement Reference]
[Dynamic Programming Series]

Longest Increasing Subsequence

Intuition:-

The **Longest Increasing Subsequence** problem requires finding the length of the longest subsequence such that all elements in the subsequence are in strictly increasing order.

The LIS problem helps us understand how to make optimal decisions over multiple options using Dynamic Programming and is foundational in areas like version control, stock analysis, and sequence alignment.

Steps:

1. **Recursive:**
    - Try picking or not picking each number based on whether it maintains the increasing order.
2. **Memoization:**
    - Store subproblem results using **index** and **prevIndex** to avoid recomputation.
3. **Tabulation:**
    - Use a bottom-up approach and iteratively fill the DP table for each index.
4. **Space Optimization:**
    - Use only 1D arrays to keep track of LIS lengths instead of 2D DP matrix.

Key Observations

1. **Edge Cases:**
    - Empty array: LIS = 0
    - All equal elements: LIS = 1
    - Strictly increasing array: LIS = length of array
2. **Preservation:**
    - All approaches return the same LIS length. Tabulation and Space Optimization drastically improve performance over recursion.

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
    public static int lis(int[] arr, int i, int prev) {
        if (i == arr.length) return 0;
        int notTake = lis(arr, i + 1, prev);
        int take = 0;
        if (prev == -1 || arr[i] > arr[prev])
            take = 1 + lis(arr, i + 1, i);
        return Math.max(take, notTake);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        System.out.println(lis(arr, 0, -1));
    }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;
public class Main {
    static int[][] dp;
    public static int lis(int[] arr, int i, int prev) {
        if (i == arr.length) return 0;
        if (dp[i][prev + 1] != -1) return dp[i][prev + 1];
        int notTake = lis(arr, i + 1, prev);
        int take = 0;
        if (prev == -1 || arr[i] > arr[prev])
            take = 1 + lis(arr, i + 1, i);
        return dp[i][prev + 1] = Math.max(take, notTake);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
```

```
        dp = new int[n][n + 1];
        for (int[] row : dp) Arrays.fill(row, -1);
        System.out.println(lis(arr, 0, -1));
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int[][] dp = new int[n + 1][n + 1];
        for (int i = n - 1; i >= 0; i--) {
            for (int prev = i - 1; prev >= -1; prev--) {
                int notTake = dp[i + 1][prev + 1];
                int take = 0;
                if (prev == -1 || arr[i] > arr[prev])
                    take = 1 + dp[i + 1][i + 1];
                dp[i][prev + 1] = Math.max(take, notTake);
            }
        }
        System.out.println(dp[0][0]);
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int[] next = new int[n + 1];
        int[] curr = new int[n + 1];
        for (int i = n - 1; i >= 0; i--) {
            for (int prev = i - 1; prev >= -1; prev--) {
                int notTake = next[prev + 1];
                int take = 0;
                if (prev == -1 || arr[i] > arr[prev])
                    take = 1 + next[i + 1];
                curr[prev + 1] = Math.max(take, notTake);
            }
            next = curr.clone();
        }
        System.out.println(next[0]);
    }
}
```

Time & Space Complexities:

Approach	**Time Complexity**	**Space Complexity**
Recursive	O(2^n)	O(n)
Memoization	$O(n^2)$	$O(n^2)$
Tabulation	$O(n^2)$	$O(n^2)$
Space Optimization	$O(n^2)$	O(n)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference](#) & [Placement Reference Dynamic Programming Series](#)

Rod Cutting

Intuition:-

Given a rod of length **W** and an array of prices that contains prices of all pieces of size smaller than or equal to **W**, determine the **maximum value obtainable** by cutting up the rod and selling the pieces.
This is essentially a variation of the **Unbounded Knapsack Problem**, where:

- You can take the same piece multiple times.
- You're optimizing value with size constraints.

Steps:

1. **Recursive:**
    - Try all cuts: either take a piece (stay at the same index) or skip to the next index.
2. **Memoization:**
    - Store the results of subproblems using a 2D array with dimensions **(index, W)** to avoid recomputation.
3. **Tabulation:**
    - Fill up the DP table bottom-up based on available rod sizes and cut options.
4. **Space Optimization:**
    - Reduce 2D DP array to 1D by reusing previously computed values.

Key Observations

1. **Edge Cases:**
    - **W = 0:** Profit is 0.
    - All prices = 0: Profit is 0.
    - All weights > W: No cuts can be made.
2. **Duplication Handling:**
    - You can reuse a cut, so we do not move to the next index when a cut is taken (unbounded knapsack style).

3. **Efficiency:**
    - Recursion is exponential.
    - DP solutions are **linear to quadratic** in complexity and handle large **W.**

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
    public static int cutRod(int[] weight, int[] price, int i, int W) {
        if (i == 0) return (W / weight[0]) * price[0];
        int notTake = cutRod(weight, price, i - 1, W);
        int take = Integer.MIN_VALUE;
        if (weight[i] <= W)
            take = price[i] + cutRod(weight, price, i, W - weight[i]);

        return Math.max(take, notTake);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n], price = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) price[i] = sc.nextInt();
        int W = sc.nextInt();
        System.out.println(cutRod(weight, price, n - 1, W));
    }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;
public class Main {
    static int[][] dp;
    public static int cutRod(int[] weight, int[] price, int i, int W) {
        if (i == 0) return (W / weight[0]) * price[0];
        if (dp[i][W] != -1) return dp[i][W];
        int notTake = cutRod(weight, price, i - 1, W);
        int take = Integer.MIN_VALUE;
```

```
        if (weight[i] <= W)
            take = price[i] + cutRod(weight, price, i, W - weight[i]);
        return dp[i][W] = Math.max(take, notTake);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n], price = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) price[i] = sc.nextInt();
        int W = sc.nextInt();
        dp = new int[n][W + 1];
        for (int[] row : dp) Arrays.fill(row, -1);
        System.out.println(cutRod(weight, price, n - 1, W));
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;
public class Main {
    public static int cutRod(int[] weight, int[] price, int n, int W) {
        int[][] dp = new int[n][W + 1];
        for (int w = 0; w <= W; w++) {
            dp[0][w] = (w / weight[0]) * price[0];
        }
        for (int i = 1; i < n; i++) {
            for (int w = 0; w <= W; w++) {
                int notTake = dp[i - 1][w];
                int take = Integer.MIN_VALUE;
                if (weight[i] <= w)
                    take = price[i] + dp[i][w - weight[i]];
                dp[i][w] = Math.max(take, notTake);
            }
        }
        return dp[n - 1][W];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n], price = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) price[i] = sc.nextInt();
```

```
        int W = sc.nextInt();
        System.out.println(cutRod(weight, price, n, W));
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;
public class Main {
    public static int cutRod(int[] weight, int[] price, int n, int W) {
        int[] prev = new int[W + 1];
        for (int w = 0; w <= W; w++) {
            prev[w] = (w / weight[0]) * price[0];
        }
        for (int i = 1; i < n; i++) {
            for (int w = 0; w <= W; w++) {
                int notTake = prev[w];
                int take = Integer.MIN_VALUE;
                if (weight[i] <= w)
                    take = price[i] + prev[w - weight[i]];
                prev[w] = Math.max(take, notTake);
            }
        }
        return prev[W];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n], price = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) price[i] = sc.nextInt();
        int W = sc.nextInt();
        System.out.println(cutRod(weight, price, n, W));
    }
}
```

Time & Space Complexities:

Approach	Time Complexity	Space Complexity
Recursive	O(2^n)	O(W)
Memoization	O(n × W)	O(n × W)
Tabulation	O(n × W)	O(n × W)
Space Optimization	O(n × W)	O(W)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference] & [Placement Reference]
[Dynamic Programming Series]

0/1 Knapsack

Intuition:-

Given weights and values of **n** items, put these items in a knapsack of capacity **W** to get the **maximum total value** in the knapsack.
You can **either pick or not pick** an item — i.e., **you cannot take an item more than once.**

This is the classical **0/1 Knapsack** where:

- Either include the item (1) or exclude it (0).
- Use dynamic programming because of overlapping subproblems and optimal substructure.

Steps:

1. **Recursive:**
    - At every item, you have 2 choices:
        - **Pick** it (if it fits in the current capacity).

        - **Don’t pick** it and move to the next item.
    - Base Case: when **i == 0** (first item), handle the value accordingly.
2. **Memoization:**
    - Cache the results using **dp[i][W]** to avoid recomputation.
3. **Tabulation:**
    - Build a **dp** table of size **n x W+1** from bottom up.
4. **Space Optimization:**
    - Reduce space from 2D to 1D array using previous state reuse.

Key Observations

1. **Edge Cases:**
    - **W == 0:** Return 0
    - All weights > **W:** No item fits → return 0
    - All values = 0 → return 0
2. **Duplication Handling:**
    - Since we can't take an item more than once, move to next index when included.

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
   public static int knapsack(int[] weight, int[] value, int i, int W) {
      if (i == 0) {
         if (weight[0] <= W) return value[0];
         return 0;
      }
      int notTake = knapsack(weight, value, i - 1, W);
      int take = Integer.MIN_VALUE;
      if (weight[i] <= W)
         take = value[i] + knapsack(weight, value, i - 1, W - weight[i]);
      return Math.max(take, notTake);
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      int[] weight = new int[n];
      int[] value = new int[n];
      for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
      for (int i = 0; i < n; i++) value[i] = sc.nextInt();
```

```
        int W = sc.nextInt();
        System.out.println(knapsack(weight, value, n - 1, W));
    }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;
public class Main {
    static int[][] dp;
    public static int knapsack(int[] weight, int[] value, int i, int W) {
        if (i == 0) {
            if (weight[0] <= W) return value[0];
            return 0;
        }
        if (dp[i][W] != -1) return dp[i][W];
        int notTake = knapsack(weight, value, i - 1, W);
        int take = Integer.MIN_VALUE;
        if (weight[i] <= W)
            take = value[i] + knapsack(weight, value, i - 1, W - weight[i]);
        return dp[i][W] = Math.max(take, notTake);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n];
        int[] value = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) value[i] = sc.nextInt();
        int W = sc.nextInt();
        dp = new int[n][W + 1];
        for (int[] row : dp) Arrays.fill(row, -1);
        System.out.println(knapsack(weight, value, n - 1, W));
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;
public class Main {
    public static int knapsack(int[] weight, int[] value, int n, int W) {
        int[][] dp = new int[n][W + 1];
```

```
        for (int w = 0; w <= W; w++) {
            if (weight[0] <= w) dp[0][w] = value[0];
        }
        for (int i = 1; i < n; i++) {
            for (int w = 0; w <= W; w++) {
                int notTake = dp[i - 1][w];
                int take = Integer.MIN_VALUE;
                if (weight[i] <= w)
                    take = value[i] + dp[i - 1][w - weight[i]];
                dp[i][w] = Math.max(take, notTake);
            }
        }
        return dp[n - 1][W];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n];
        int[] value = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) value[i] = sc.nextInt();
        int W = sc.nextInt();
        System.out.println(knapsack(weight, value, n, W));
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;

public class Main {
    public static int knapsack(int[] weight, int[] value, int n, int W) {
        int[] prev = new int[W + 1];
        for (int w = 0; w <= W; w++) {
            if (weight[0] <= w) prev[w] = value[0];
        }
        for (int i = 1; i < n; i++) {
            for (int w = W; w >= 0; w--) {
                int notTake = prev[w];
                int take = Integer.MIN_VALUE;
                if (weight[i] <= w)
                    take = value[i] + prev[w - weight[i]];
                prev[w] = Math.max(take, notTake);
```

```
            }
        }
        return prev[W];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] weight = new int[n];
        int[] value = new int[n];
        for (int i = 0; i < n; i++) weight[i] = sc.nextInt();
        for (int i = 0; i < n; i++) value[i] = sc.nextInt();
        int W = sc.nextInt();
        System.out.println(knapsack(weight, value, n, W));
    }
}
```

Time & Space Complexities:

Approach	Time Complexity	Space Complexity
Recursive	O(2^n)	O(W) (recursion stack)
Memoization	O(n × W)	O(n × W)
Tabulation	O(n × W)	O(n × W)
Space Optimization	O(n × W)	O(W)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - [DSA - Linked List Reference](#) & [Placement Reference](#)
[Dynamic Programming Series](#)

SubSet Sum Problem

Intuition:-

Given an array of **n** integers and a target **sum,** determine if there exists a subset whose sum is exactly equal to the target.
This is a **variation of the 0/1 Knapsack** where:

- We don't care about maximizing value.
- We only need to **check the existence** of a subset that matches the sum.

Steps:

1. **Recursive:**
   - Explore both choices: include the current element or skip.
2. **Memoization:**
   - Cache **(i, target)** results in a 2D DP array to avoid recomputation.
3. **Tabulation:**
   - Fill a DP table iteratively from smaller to larger subproblems.
4. **Space Optimization:**
   - Reduce space from 2D to 1D using only the previous state row.

Key Observations

1. **Edge Cases:**
   - If **target == 0:** always true (empty subset).
   - If the **array is empty** and **target != 0:** false.
2. **Duplication Handling:**
   - Each element can be used at most once (like 0/1 Knapsack).

Code 1: **Recursive Approach**

```
import java.util.*;
public class Main {
    public static boolean subsetSum(int[] arr, int i, int target) {
        if (target == 0) return true;
        if (i == 0) return arr[0] == target;
```

```
        boolean notTake = subsetSum(arr, i - 1, target);
        boolean take = false;
        if (arr[i] <= target) take = subsetSum(arr, i - 1, target - arr[i]);

        return take || notTake;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int target = sc.nextInt();
        System.out.println(subsetSum(arr, n - 1, target));
    }
}
```

Code 2: **Memoization (Top-Down DP)**

```
import java.util.*;
public class Main {
    static int[][] dp;
 public static boolean subsetSum(int[] arr, int i, int target) {
        if (target == 0) return true;
        if (i == 0) return arr[0] == target;
        if (dp[i][target] != -1) return dp[i][target] == 1;
        boolean notTake = subsetSum(arr, i - 1, target);
        boolean take = false;
        if (arr[i] <= target) take = subsetSum(arr, i - 1, target - arr[i]);
        dp[i][target] = (take || notTake) ? 1 : 0;
        return take || notTake;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int target = sc.nextInt();
        dp = new int[n][target + 1];
        for (int[] row : dp) Arrays.fill(row, -1);
        System.out.println(subsetSum(arr, n - 1, target));
    }
}
```

Code 3: **Tabulation (Bottom-Up DP)**

```
import java.util.*;
public class Main {
    public static boolean subsetSum(int[] arr, int n, int target) {
        boolean[][] dp = new boolean[n][target + 1];
        for (int i = 0; i < n; i++) dp[i][0] = true;
        if (arr[0] <= target) dp[0][arr[0]] = true;
        for (int i = 1; i < n; i++) {
            for (int t = 1; t <= target; t++) {
                boolean notTake = dp[i - 1][t];
                boolean take = false;
                if (arr[i] <= t) take = dp[i - 1][t - arr[i]];
                dp[i][t] = take || notTake;
            }
        }
        return dp[n - 1][target];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int target = sc.nextInt();
        System.out.println(subsetSum(arr, n, target));
    }
}
```

Code 4: **Space Optimization**

```
import java.util.*;
public class Main {
    public static boolean subsetSum(int[] arr, int n, int target) {
        boolean[] prev = new boolean[target + 1];
        prev[0] = true;
        if (arr[0] <= target) prev[arr[0]] = true;
        for (int i = 1; i < n; i++) {
            boolean[] curr = new boolean[target + 1];
            curr[0] = true;
            for (int t = 1; t <= target; t++) {
                boolean notTake = prev[t];
                boolean take = false;
                if (arr[i] <= t) take = prev[t - arr[i]];
```

```
                curr[t] = take || notTake;
            }
            prev = curr;
        }
        return prev[target];
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextInt();
        int target = sc.nextInt();
        System.out.println(subsetSum(arr, n, target));
    }
}
```

Time & Space Complexities:

Approach	Time Complexity	Space Complexity
Recursive	O(2^n)	O(n)
Memoization	O(n × target)	O(n × target)
Tabulation	O(n × target)	O(n × target)
Space Optimized	O(n × target)	O(target)

***If you are doing a few more problem statements, that is appreciable.**

Youtube Reference - DSA - Linked List Reference & Placement Reference Dynamic Programming Series