Ex.No:1 **Date:**

Aim:

Write a program to implement the following operations on Binary Search Tree

a) Insert b) Delete c) Search d) Display

Description:

- Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree. There must be no duplicate nodes.
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The following operations are performed on a binary search tree...

1. Insertion
2. Search
3. Deletion

Procedure:

1. Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows...

Step 1 - Create a newNode with given value and set its left and right to NULL.

Step 2 - Check whether tree is Empty.

Step 3 - If the tree is Empty, then set root to newNode.

Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.

Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

Step 7-After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6- If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity.

Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

Step 1 - Find the node to be deleted using search operation
Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

Step 1 - Find the node to be deleted using search operation
Step 2 - If it has only one child then create a link between its parent node and child node.
Step 3 - Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

Step 1 - Find the node to be deleted using search operation
Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
Step 3 - Swap both deleting node and node which is found in the above step.
Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
Step 5 - If it comes to case 1, then delete using case 1 logic.
Step 6- If it comes to case 2, then delete using case 2 logic.
Step 7 - Repeat the same process until the node is deleted from the tree.

```
Public class BinarySearchTree
{
    //Represent a node of binary tree
      public static class Node{
          int data;
          Node left;
          Node right;

          public Node(int data){
              //Assign data to the new node, set left and right children to null
              this.data = data;
              this.left = null;
              this.right = null;
           }
         }

         //Represent the root of binary tree
         public Node root;

         public BinarySearchTree(){
            root = null;
         }

         //insert() will add new node to the binary search tree
         public void insert(int data) {
```

```
//Create a new node
Node newNode = new Node(data);


//Check whether tree is empty
if(root == null){
   root = newNode;
   return;
  }
else {
   //current node point to root of the tree
   Node current = root, parent = null;


   while(true) {
      //parent keep track of the parent node of current node.
      parent = current;


     //If data is less than current's data, node will be inserted to the left of tree
      if(data < current.data) {
         current = current.left;
         if(current == null) {
            parent.left = newNode;
            return;
         }
      }
            //If data is greater than current's data, node will be inserted to the right of tree
```

```
            else {
                current = current.right;
                if(current == null) {
                    parent.right = newNode;
                  return;
                }
            }
        }
    }
}

     //minNode() will find out the minimum node
     public Node minNode(Node root) {
    if (root.left != null)
        return minNode(root.left);
    else
        return root;
      }

//deleteNode() will delete the given node from the binary search tree
public Node deleteNode(Node node, int value) {
    if(node == null){
        return null;
     }
    else {
```

```
//value is less than node's data then, search the value in left subtree
if(value < node.data)
    node.left = deleteNode(node.left, value);

//value is greater than node's data then, search the value in right subtree
else if(value > node.data)
    node.right = deleteNode(node.right, value);

//If value is equal to node's data that is, we have found the node to be deleted
else {
    //If node to be deleted has no child then, set the node to null
    if(node.left == null && node.right == null)
        node = null;

    //If node to be deleted has only one right child
    else if(node.left == null) {
        node = node.right;
    }

  //If node to be deleted has only one left child
    else if(node.right == null) {
        node = node.left;
    }

    //If node to be deleted has two children node
```

```
            else {
                //then find the minimum node from right subtree
                Node temp = minNode(node.right);
                //Exchange the data between node and temp
                node.data = temp.data;
                //Delete the node duplicate node from right subtree
                node.right = deleteNode(node.right, temp.data);
            }
        }
        return node;
    }
}

//inorder() will perform inorder traversal on binary search tree
public void inorderTraversal(Node node) {

    //Check whether tree is empty
    if(root == null){
       System.out.println("Tree is empty");
        return;
    }
    else {

        if(node.left!= null)
            inorderTraversal(node.left);
```

```
            System.out.print(node.data + " ");
            if(node.right!= null)
                inorderTraversal(node.right);

        }
    }

    public static void main(String[] args) {

        BinarySearchTree bt = new BinarySearchTree();
       //Add nodes to the binary tree
        bt.insert(50);
        bt.insert(30);
        bt.insert(70);
        bt.insert(60);
        bt.insert(10);
        bt.insert(90);

        System.out.println("Binary search tree after insertion:");
        //Displays the binary tree
        bt.inorderTraversal(bt.root);

       Node deletedNode = null;
        //Deletes node 90 which has no child
        deletedNode = bt.deleteNode(bt.root, 90);
```

```
        System.out.println("\nBinary search tree after deleting node 90:");
        bt.inorderTraversal(bt.root);

        //Deletes node 30 which has one child
        deletedNode = bt.deleteNode(bt.root, 30);
        System.out.println("\nBinary search tree after deleting node 30:");
        bt.inorderTraversal(bt.root);

        //Deletes node 50 which has two children
        deletedNode = bt.deleteNode(bt.root, 50);
        System.out.println("\nBinary search tree after deleting node 50:");
        bt.inorderTraversal(bt.root);
    }
}
```

Ex.No: 2 **Date:**

Aim:

Write a program to perform a Binary Search for a given set of integer values

Description:

Binary Search: A binary search is an algorithm to find a particular element in the list.

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

In the binary search algorithm, we can find the element position using the following methods.

a) Recursive Method

b) Iterative Method

The divide and conquer approach technique is followed by the recursive method. In this method, a function is called itself again and again until it found an element in the list.

A set of statements is repeated multiple times to find an element's index position in the iterative method. The **while** loop is used for accomplish this task.

Procedure:

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

USING RECURSIVE:

```
class BST
{
	private static class Node
	{
		int data;
		Node left;
		Node right;
		public Node (int data)
		{
			this.data=data;
		}
	}
	public static Node insert(Node root,intval)
	{
		if(root==null)
		{
			root=new Node(val);
			return root;
		}
		if(root.data>val)
		{
			root.left=insert(root.left,val);
		}
		else
		{
			root.right=insert(root.right,val);
		}
```

```
                return root;
        }
        public static void inorder (Node root)
        {
                if(root==null)
                {
                        return;
                }
                inorder(root.left);
                System.out.println(root.data+" ");
                inorder(root.right);
        }

        public static void main(String args[])
        {
                int val[]={3,4,6,7,9,5,8};
                Node root=null;
                for(int i=0;i<val.length;i++)
                {
                        root=insert(root,val[i]);
                }
                inorder(root);
        }
}
```

2. Iterative Method

```
class BST
{
	private static class Node
	{
		int data;
		Node left;
		Node right;
		public Node (int data)
		{
			this.data=data;
		}
	}
	public static Node insert(Node root,intval)
	{
		if(root==null)
		{
			root=new Node(val);
			return root;
		}
		if(root.data>val)
		{
			root.left=insert(root.left,val);
		}
		else
		{
			root.right=insert(root.right,val);
		}
```

```
                return root;
        }
        public static void inorder (Node root)
        {
                if(root==null)
                {
                        return;
                }
                inorder(root.left);
                System.out.println(root.data+" ");
                inorder(root.right);
        }

        public static void main(String args[])
        {
                int val[]={3,4,6,7,9,5,8};
                Node root=null;
                for(int i=0;i<val.length;i++)
                {
                        root=insert(root,val[i]);
                }
                inorder(root);
        }
}
```

Ex.No: 3 **Date:**

Aim:

Write a program to implement Splay trees.

Description:

- Splay tree is self-balancing BST.
- The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again.
- The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with AVL and Red-Black Trees.
- All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

```
class  SPLAY
{
        static class node
        {
                int key;
                node left, right;
        };
        static node newNode(int key)
        {
                node Node = new node();
                Node.key = key;
                Node.left = Node.right = null;
                return (Node);
        }
        static node rightRotate(node x)
        {
                node y = x.left;
                x.left = y.right;
                y.right = x;
                return y;
        }
        static node leftRotate(node x)
        {
                node y = x.right;
                x.right = y.left;
                y.left = x;
```

```
        return y;
}
static node splay(node root, int key)
{
if (root == null || root.key == key)
        return root;
if (root.key > key)
{
        if (root.left == null)
        return root;
        if (root.left.key > key)
        {
                root.left.left = splay(root.left.left, key);
                root = rightRotate(root);
        }
        else if (root.left.key < key)
        {
                root.left.right = splay(root.left.right, key);
                if (root.left.right != null)
                root.left = leftRotate(root.left);
        }
        return (root.left == null) ? root : rightRotate(root);
}
else
{
        if (root.right == null)
        return root;
        if (root.right.key > key)
```

```
                {
                        root.right.left = splay(root.right.left,key);
                        if (root.right.left != null)
                        root.right = rightRotate(root.right);
                }
                else if (root.right.key < key)
                {
                        root.right.right = splay(root.right.right, key);
                        root = leftRotate(root);
                }
                return (root.right == null) ? root : leftRotate(root);
        }
}
static node search(node root, int key)
{
        return splay(root, key);
}
static void preOrder(node root)
{
        if (root != null)
        {
                System.out.print(root.key + " ");
                preOrder(root.left);
                preOrder(root.right);
        }
}
public static void main(String[] args)
{
```

```
        node root = newNode(100);
        root.left = newNode(50);
        root.right = newNode(200);
        root.left.left = newNode(40);
        root.left.left.left = newNode(30);
        root.left.left.left.left = newNode(20);


        root = search(root, 20);
        System.out.print("Preorder traversal of the" + " modified Splay tree is \n");
        preOrder(root);
}
}
```

Ex.No: 4 **Date:**

Aim:

Write a program to implement Merge sort for the given list of integer values.

Description:

Merge Sort

- Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
- **The merge() function** is used for merging two halves.
- The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Three main components of the divide-and-conquer approach to algorithm design:

1. **Divide: c**ontinuously break down the larger problem into smaller parts.
2. **Conquer: s**olve each of the smaller parts by utilising a function that's called recursively.
3. **Combine: m**erge all solutions for all smaller parts into a single unified solution, which becomes the solution to the starting problem.

Procedure:

MergeSort(arr[], l, r)
If r > l

1. Find the middle point to divide the array into two halves:
   middle m = l+ (r-l)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

```
class MergeSort
{
        void merge(int arr[], int l, int m, int r)
        {

                int n1 = m - l + 1;
                int n2 = r - m;
                int L[] = new int[n1];
                int R[] = new int[n2];
                for (int i = 0; i < n1; ++i)
                        L[i] = arr[l + i];
                for (int j = 0; j < n2; ++j)
                        R[j] = arr[m + 1 + j];
                int i = 0, j = 0;
                int k = l;
                while (i < n1 && j < n2)
                {
                        if (L[i] <= R[j])
                    {
                                arr[k] = L[i];
                                i++;
                        }
                        else
                        {
                                arr[k] = R[j];
                                j++;
                        }
```

```
                k++;
            }
            while (i < n1)
            {
                arr[k] = L[i];
                i++;
                k++;
            }
            while (j < n2)
            {
                arr[k] = R[j];
                j++;
                k++;
            }
    }
    void sort(int arr[], int l, int r)
    {
            if (l < r)
             {
                int m = l + (r - l) / 2;
                sort(arr, l, m);
                sort(arr, m + 1, r);
                merge(arr, l, m, r);
            }
    }
    static void printArray(int arr[])
    {
            int n = arr.length;
```

```
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        System.out.println("Given Array");
        printArray(arr);
        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length - 1);
        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

Ex.No: 5 **Date:**

Aim:

Write a program to implement Quick sort for the given list of integer values.

Description:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

```
import java.util.*;

class Main
 {
   //partitions the array around pivot=> last element
   static int partition(int numArray[], int low, int high)
  {
      int pivot = numArray[high];
      // smaller element index
      int i = (low - 1);
      for (int j = low; j <= high - 1; j++) {
         // check if current element is less than or equal to pivot
         if (numArray[j] <= pivot) {
            i++;
            // swap the elements
            int temp = numArray[i];
            numArray[i] = numArray[j];
            numArray[j] = temp;
         }
      }
      // swap numArray[i+1] and numArray[high] (or pivot)
      int temp = numArray[i + 1];
      numArray[i + 1] = numArray[high];
      numArray[high] = temp;
      return i + 1;
   }


//sort the array using quickSort
   static void quickSort(int numArray[], int low, int high)
   {
      //auxillary stack
      int[] intStack = new int[high - low + 1];

      // top of stack initialized to -1
      int top = -1;

      // push initial values of low and high to stack
      intStack[++top] = low;
      intStack[++top] = high;

      // Keep popping from stack while is not empty
      while (top >= 0) {
         // Pop h and l
         high = intStack[top--];
         low = intStack[top--];

         // Set pivot element at its correct position
         // in sorted array
         int pivot = partition(numArray, low, high);

         // If there are elements on left side of pivot,
```

```
        // then push left side to stack
        if (pivot - 1 > low)
{
           intStack[++top] = low;
           intStack[++top] = pivot - 1;
        }

        // If there are elements on right side of pivot,
        // then push right side to stack
        if (pivot + 1 < high) {
           intStack[++top] = pivot + 1;
           intStack[++top] = high;
        }
     }
  }

public static void main(String args[])
  {
     //define array to be sorted
     int numArray[] = { 3,2,6,-1,9,1,-6,10,5 };
     int n = 8;
     System.out.println("Original Array:" + Arrays.toString(numArray));
     // call quickSort routine to sort the array
     quickSort(numArray, 0, n - 1);
     //print the sorted array
     System.out.println("\nSorted Array:" + Arrays.toString(numArray));
  }
}
```

Output:
Original Array:[3, 2, 6, -1, 9, 1, -6, 10, 5]
Sorted Array:[-6, -1, 1, 2, 3, 6, 9, 10, 5]

Ex.No: 6 **Date:**

Aim:

Write a program to find the solution for the knapsack problem using the greedy method.

Description:

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Input:
Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;

Output:
Maximum possible value = 240
by taking items of weight 10 and 20 kg and 2/3 fraction
of 30 kg. Hence total price will be 60+100+(2/3)(120) = 240

```
import java.util.Scanner;
class Knapsack
{
public static void main(String[] args)
{
Scanner sc=new Scanner(System.in);
int object,m;
System.out.println("Enter the Total Objects");
object=sc.nextInt();
int weight[]=new int[object];
int profit[]=new int[object];
for(int i=0;i<object;i++)
{
System.out.println("Enter the Profit");
profit[i]=sc.nextInt();
System.out.println("Enter the weight");
weight[i]=sc.nextInt();
}
System.out.println("Enter the Knapsack capacity");
m=sc.nextInt();
double p_w[]=new double[object];
for(int i=0;i<object;i++)
{
p_w[i]=(double)profit[i]/(double)weight[i];
}
System.out.println("");
System.out.println("--------------------");
```

```
System.out.println("-----Data-Set------");
System.out.print("--------------------");
System.out.println("");
System.out.print("Objects");
for(int i=1;i<=object;i++)
{
System.out.print(i+" ");
}
System.out.println();
System.out.print("Profit ");
for(int i=0;i<object;i++)
{
System.out.print(profit[i]+" ");
}
System.out.println();
System.out.print("Weight ");
for(int i=0;i<object;i++)
{
System.out.print(weight[i]+" ");
}
System.out.println();
System.out.print("P/W ");
for(int i=0;i<object;i++)
{
System.out.print(p_w[i]+" ");
}
for(int i=0;i<object-1;i++)
{
```

```
for(int j=i+1;j<object;j++)
{
if(p_w[i]<p_w[j])
{
double temp=p_w[j];
p_w[j]=p_w[i];
p_w[i]=temp;
int temp1=profit[j];
profit[j]=profit[i];
profit[i]=temp1;
int temp2=weight[j];
weight[j]=weight[i];
weight[i]=temp2;
}
}
}
System.out.println("");
System.out.println("--------------------");
System.out.println("--After Arranging--");
System.out.print("--------------------");
System.out.println("");
System.out.print("Objects");
for(int i=1;i<=object;i++)
{
System.out.print(i+" ");
}
System.out.println();
System.out.print("Profit ");
```

```
for(int i=0;i<object;i++)
{
System.out.print(profit[i]+" ");
}
System.out.println();
System.out.print("Weight ");
for(int i=0;i<object;i++)
{
System.out.print(weight[i]+" ");
}
System.out.println();
System.out.print("P/W ");
for(int i=0;i<object;i++)
{
System.out.print(p_w[i]+" ");
}
int k=0;
double sum=0;
System.out.println();
while(m>0)
{
if(weight[k]<m)
{
sum+=1*profit[k];
m=m-weight[k];
}
else
{
```

```
double x4=m*profit[k];

double x5=weight[k];

double x6=x4/x5;

sum=sum+x6;

m=0;

}

k++;

}

System.out.println("Final Profit is="+sum);

}

}
```

Ex.No:7 **Date:**

Aim:

Write a program to find minimum cost spanning tree using Prim's algorithm.

Description:

A **Spanning Tree (ST)** of a connected undirected weighted graph **G** is a subgraph of **G** that is a **tree** and **connects (spans) all vertices of G**. A graph **G** can have multiple STs, each with different total weight (the sum of edge weights in the ST).

A **Min(imum) Spanning Tree (MST)** of **G** is an ST of **G** that has the **smallest total weight** amongthe various STs.

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edgeto make it a *Minimum* Spanning Tree.

Algorithm :

Prims minimum spanning tree (Graph G, Souce_Node S)

1. Create a dictionary (to be used as a priority queue) PQ to hold pairs of (node, cost).
2. Push [S, 0] (node, cost) in the dictionary PQ i.e Cost of reaching vertex S from source node Sis zero.
3. While PQ contains (V, C) pairs :
4. Get the adjacent node V (key) with the smallest edge cost (value) from the dictionary PQ.
5. Cost C = PQ [V]
6. Delete the key-value pair (V, C) from the dictionary PQ.
7. If the adjacent node V is not added to the spanning tree.
8. Add node V to the spanning tree.
9. Cost of the spanning tree += Cost C
10. For all vertices adjacent to vertex V not added to spanning tree.
11. Push pair of (adjacent node, cost) into the dictionary PQ.

```
// A Java program for Prim's Minimum Spanning Tree (MST)
// algorithm. The program is for adjacency matrix
// representation of the graph

import java.io.*;
import java.lang.*;
import java.util.*;

class MST {

    // Number of vertices in the graph
    private static final int V = 5;

    // A utility function to find the vertex with minimum
    // key value, from the set of vertices not yet included
    // in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index = -1;

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min) {
                min = key[v];
                min_index = v;
            }

        return min_index;
```

```
}

// A utility function to print the constructed MST
// stored in parent[]
void printMST(int parent[], int graph[][])
{
        System.out.println("Edge \tWeight");
        for (int i = 1; i < V; i++)
                System.out.println(parent[i] + " - " + i + "\t"
                                                + graph[i][parent[i]]);
}

// Function to construct and print MST for a graph
// represented using adjacency matrix representation
void primMST(int graph[][])
{
        // Array to store constructed MST
        int parent[] = new int[V];

        // Key values used to pick minimum weight edge in
        // cut
        int key[] = new int[V];

        // To represent set of vertices included in MST
        Boolean mstSet[] = new Boolean[V];

        // Initialize all keys as INFINITE
        for (int i = 0; i < V; i++) {
```

```
            key[i] = Integer.MAX_VALUE;
            mstSet[i] = false;
    }

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is
    // picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
            // Pick thd minimum key vertex from the set of
            // vertices not yet included in MST
            int u = minKey(key, mstSet);

            // Add the picked vertex to the MST Set
            mstSet[u] = true;

            // Update key value and parent index of the
            // adjacent vertices of the picked vertex.
            // Consider only those vertices which are not
            // yet included in MST
            for (int v = 0; v < V; v++)

                    // graph[u][v] is non zero only for adjacent
                    // vertices of m mstSet[v] is false for
                    // vertices not yet included in MST Update
                    // the key only if graph[u][v] is smaller
```

```
                        // than key[v]
                        if (graph[u][v] != 0 && mstSet[v] == false
                                && graph[u][v] < key[v]) {
                                parent[v] = u;
                                key[v] = graph[u][v];
                        }
                }

                // print the constructed MST
                printMST(parent, graph);
        }

        public static void main(String[] args)
        {
                /* Let us create the following graph
                2 3
                (0)--(1)--(2)
                | / \ |
                6| 8/ \5 |7
                | /       \ |
                (3)-------(4)
                        9                */
                MST t = new MST();
                int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
                                             { 2, 0, 3, 8, 5 },
                                             { 0, 3, 0, 0, 7 },
                                             { 6, 8, 0, 0, 9 },
                                             { 0, 5, 7, 9, 0 } };

                // Print the solution
```

```
                t.primMST(graph);

        }

}
```

Ex.No:8 **Date:**

Aim:

Write a program to find minimum cost spanning tree using Kruskal's algorithm.

Description:

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects toanother only and only if, it has the least cost among all available options and does not violate MSTproperties.

Algorithm :

Kruskal's minimum spanning tree (Graph G)

0. Create an empty minimum spanning tree M i.e M = ∅ (zero edges)
1. Sort the edge-list of the graph G in ascending order of weights.
2. For each edge (A, B) in the sorted edge-list.
3. If Find_Set_Of_A != Find_Set_Of_B, then
4. Add edge A-B to the minimum spanning tree M i.e M = M + edge (A - B)
5. Make a set / union of Find_Set_Of_A + Find_Set_Of_B.

```
// Java program for Kruskal's algorithm to

// find Minimum Spanning Tree of a given

// connected, undirected and weighted graph



import java.io.*;

import java.lang.*;

import java.util.*;



public class Graph {



        // A class to represent a graph edge

        class Edge implements Comparable<Edge> {

                int src, dest, weight;



                // Comparator function used for

                // sorting edgesbased on their weight

                public int compareTo(Edge compareEdge)

                {
```

```
                return this.weight - compareEdge.weight;
        }
};

// A class to represent a subset for
// union-find
class subset {
        int parent, rank;
};
int V, E; // V-> no. of vertices & E->no.of edges
Edge edge[]; // collection of all edges
Graph(int v, int e)
{
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i = 0; i < e; ++i)
                edge[i] = new Edge();
}
```

```
// A utility function to find set of an

// element i (uses path compression technique)

int find(subset subsets[], int i)

{

        // find root and make root as parent of i

        // (path compression)

        if (subsets[i].parent != i)

                subsets[i].parent

                        = find(subsets, subsets[i].parent);



        return subsets[i].parent;

}



// A function that does union of two sets

// of x and y (uses union by rank)

void Union(subset subsets[], int x, int y)

{

        int xroot = find(subsets, x);
```

```
        int yroot = find(subsets, y);



        // Attach smaller rank tree under root

        // of high rank tree (Union by Rank)

        if (subsets[xroot].rank < subsets[yroot].rank)

                subsets[xroot].parent = yroot;

        else if (subsets[xroot].rank > subsets[yroot].rank)

                subsets[yroot].parent = xroot;



        // If ranks are same, then make one as

        // root and increment its rank by one

        else {

                subsets[yroot].parent = xroot;

                subsets[xroot].rank++;

        }

}



// The main function to construct MST using Kruskal's

// algorithm
```

```
void KruskalMST()

{

        // This will store the resultant MST

        Edge result[] = new Edge[V];



        // An index variable, used for result[]

        int e = 0;



        // An index variable, used for sorted edges

        int i = 0;

        for (i = 0; i < V; ++i)

                result[i] = new Edge();



        // Step 1: Sort all the edges in non-decreasing

        // order of their weight. If we are not allowed to

        // change the given graph, we can create a copy of

        // array of edges

        Arrays.sort(edge);
```

```
// Allocate memory for creating V subsets

subset subsets[] = new subset[V];

for (i = 0; i < V; ++i)

        subsets[i] = new subset();



// Create V subsets with single elements

for (int v = 0; v < V; ++v) {

        subsets[v].parent = v;

        subsets[v].rank = 0;

}



i = 0; // Index used to pick next edge



// Number of edges to be taken is equal to V-1

while (e < V - 1) {

        // Step 2: Pick the smallest edge. And increment

        // the index for next iteration

        Edge next_edge = edge[i++];
```

```
            int x = find(subsets, next_edge.src);

            int y = find(subsets, next_edge.dest);



            // If including this edge doesn't cause cycle,

            // include it in result and increment the index

            // of result for next edge

            if (x != y) {

                  result[e++] = next_edge;

                  Union(subsets, x, y);

            }

            // Else discard the next_edge

      }



      // print the contents of result[] to display

      // the built MST

      System.out.println("Following are the edges in "

                                    + "the constructed MST");

      int minimumCost = 0;

      for (i = 0; i < e; ++i) {
```

```
                System.out.println(result[i].src + " -- "
                                                    + result[i].dest
                                                    + " == " + result[i].weight);
                minimumCost += result[i].weight;
            }
            System.out.println("Minimum Cost Spanning Tree "
                                                + minimumCost);
    }

    // Driver's Code
    public static void main(String[] args)
    {
            int V = 4; // Number of vertices in graph
            int E = 5; // Number of edges in graph
            Graph graph = new Graph(V, E);

            // add edge 0-1
            graph.edge[0].src = 0;
            graph.edge[0].dest = 1;
```

```
graph.edge[0].weight = 10;

// add edge 0-2

graph.edge[1].src = 0;

graph.edge[1].dest = 2;

graph.edge[1].weight = 6;

// add edge 0-3

graph.edge[2].src = 0;

graph.edge[2].dest = 3;

graph.edge[2].weight = 5;

// add edge 1-3

graph.edge[3].src = 1;

graph.edge[3].dest = 3;

graph.edge[3].weight = 15;

// add edge 2-3

graph.edge[4].src = 2;

graph.edge[4].dest = 3;

graph.edge[4].weight = 4;

// Function call

graph.KruskalMST();
```

```
        }

}
```

Ex.No:9 **Date:**

Aim:

Write a program to find a single source shortest path for a given graph.

Description:

Bellman-Ford single source shortest path algorithm is a below :

- Bellman-Ford algorithm finds the shortest path (in terms of distance / cost) from a single source in a directed, weighted graph containing positive and negative edge weights.
- Bellman-Ford algorithm performs edge relaxation of all the edges for every node. Bellman-Ford Edge Relaxation

  If (distance-from-source-to [node_v] > distance-from-source-to [node_u] +weight-of-path-from-node-u-to-v)

  then

  distance-from-source-to [node_v] = distance-from-source-to [node_u] +weight-of-path-from-node-u-to-v)
- With negative edge weights in a graph Bellman-Ford algorithm is preferred over Dijkstra's algorithm as Dijkstra's algorithm cannot handle negative edge weights in a graph.

Below data structures are used for storing the graph before running Bellman-Ford algorithm

- EdgeList : List of all the edges in the graph.
- EdgeWeight : Map of edges and their corresponding weights.

Algorithm :

Bellman-Ford Single Source Shortest Path (EdgeList, EdgeWeight)

1. Initialize the distance from the source node S to all other nodes as infinite (999999999) and to itself as 0.
   Distance [AllNodes] = 999999999, Distance [S] = 0.
2. For every node in the graph
3. For every edge E in the EdgeList
4. Node_u = E.first, Node_v = E.second
5. Weight_u_v = EdgeWeight (Node_u, Node_v)
6. If (Distance [v] > Distance [u] + Weight_u_v)
7. Distance [v] = Distance [u] + Weight_u_v

```
// A Java program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph
import java.io.*;
import java.lang.*;
import java.util.*;

class ShortestPath {
	// A utility function to find the vertex with minimum
	// distance value, from the set of vertices not yet
	// included in shortest path tree
	static final int V = 9;
	int minDistance(int dist[], Boolean sptSet[])
	{
		// Initialize min value
		int min = Integer.MAX_VALUE, min_index = -1;

		for (int v = 0; v < V; v++)
			if (sptSet[v] == false && dist[v] <= min) {
				min = dist[v];
				min_index = v;
			}

		return min_index;
	}

	// A utility function to print the constructed distance
	// array
```

```
void printSolution(int dist[])
{
        System.out.println(
                "Vertex \t\t Distance from Source");
        for (int i = 0; i < V; i++)
                System.out.println(i + " \t\t " + dist[i]);
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[][], int src)
{
        int dist[] = new int[V]; // The output array.
                                                        // dist[i] will hold
        // the shortest distance from src to i

        // sptSet[i] will true if vertex i is included in
        // shortest path tree or shortest distance from src
        // to i is finalized
        Boolean sptSet[] = new Boolean[V];

        // Initialize all distances as INFINITE and stpSet[]
        // as false
        for (int i = 0; i < V; i++) {
                dist[i] = Integer.MAX_VALUE;
                sptSet[i] = false;
        }
```

```
// Distance of source vertex from itself is always 0
dist[src] = 0;

// Find shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set
        // of vertices not yet processed. u is always
        // equal to src in first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of
        // the picked vertex.
        for (int v = 0; v < V; v++)

                // Update dist[v] only if is not in sptSet,
                // there is an edge from u to v, and total
                // weight of path from src to v through u is
                // smaller than current value of dist[v]
                if (!sptSet[v] && graph[u][v] != 0
                        && dist[u] != Integer.MAX_VALUE
                        && dist[u] + graph[u][v] < dist[v])
                        dist[v] = dist[u] + graph[u][v];
}
```

```
        // print the constructed distance array
        printSolution(dist);
    }

    // Driver's code
    public static void main(String[] args)
    {
        /* Let us create the example graph discussed above
        */
        int graph[][]
            = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                            { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                            { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                            { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                            { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                            { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                            { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                            { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                            { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
        ShortestPath t = new ShortestPath();

        // Function call
        t.dijkstra(graph, 0);
    }
}
```

Ex.No:10 **Date:**

Aim:

Write a program to find the solution for job sequencing with deadlines problems.

Description:

Job sequencing is the set of jobs, associated with the job i where deadline di >= 0 and profit pi > 0. For any job i the profit is earned if and only if the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing the jobs.

Algorithm for job sequencing

Input: A is the array of jobs with deadline and profit S array will be the output.

1. Begin
2. Sort all the jobs based on profit Pi so
3. P1 > P2 > P3 ..... >=Pn
4. d = maximum deadline of job in A
5. Create array S[1,... .....,d]
6. For i=1 to n do
7. Find the largest job x
8. For j=i to 1
9. If ((S[j] = 0) and (x deadline<= d))
10. Then
11. S[x] = i;
12. Break;
13. End if
14. End for
15. End for
16. End

Procedure:

Steps for performing job sequencing with deadline using greedy approach is as follows:

1. Sort all the jobs based on the profit in an increasing order.
2. Let α be the maximum deadline that will define the size of array.
3. Create a solution array S with d slots.

4. Initialize the content of array S with zero.
5. Check for all jobs.
    1. If scheduling is possible a lot i^{th} slot of array s to job i.
    2. Otherwise look for location (i-1), (i-2)…1.
    3. Schedule the job if possible else reject.
6. Return array S as the answer.
7. End.

```
import java.util.*;
class Job
{
	char id;
	int deadline, profit;
	// Constructors
	public Job() {}

	public Job(char id, int deadline, int profit)
	{
		this.id = id;
		this.deadline = deadline;
		this.profit = profit;
	}
	// Function to schedule the jobs take 2 arguments
	// arraylist and no of jobs to schedule
	void printJobScheduling(ArrayList<Job> arr, int t)
	{
		// Length of array
		int n = arr.size();
		// Sort all jobs according to decreasing order of
		// profit
		Collections.sort(arr,
                                                (a, b) -> b.profit - a.profit);

		// To keep track of free time slots
```

```
        boolean result[] = new boolean[t];
        // To store result (Sequence of jobs)
        char job[] = new char[t];
        // Iterate through all given jobs
        for (int i = 0; i < n; i++) {
            // Find a free slot for this job (Note that we
            // start from the last possible slot)
            for (int j
                = Math.min(t - 1, arr.get(i).deadline - 1);
                j >= 0; j--) {
                // Free slot found
                if (result[j] == false) {
                    result[j] = true;
                    job[j] = arr.get(i).id;
                    break;
                }
            }
        }

        // Print the sequence
        for (char jb : job)
            System.out.print(jb + " ");
        System.out.println();
    }

    // Driver's code
    public static void main(String args[])
    {
```

```
        ArrayList<Job> arr = new ArrayList<Job>();
        arr.add(new Job('a', 2, 100));
        arr.add(new Job('b', 1, 19));
        arr.add(new Job('c', 2, 27));
        arr.add(new Job('d', 1, 25));
        arr.add(new Job('e', 3, 15));
        System.out.println("Following is maximum profit sequence of jobs");
        Job job = new Job();
        // Function call
        job.printJobScheduling(arr, 3);
    }
}
```

Ex.No:11 **Date:**

Aim:

Write a program to find the solution for a 0-1 knapsack problem using dynamic programming.

Description:

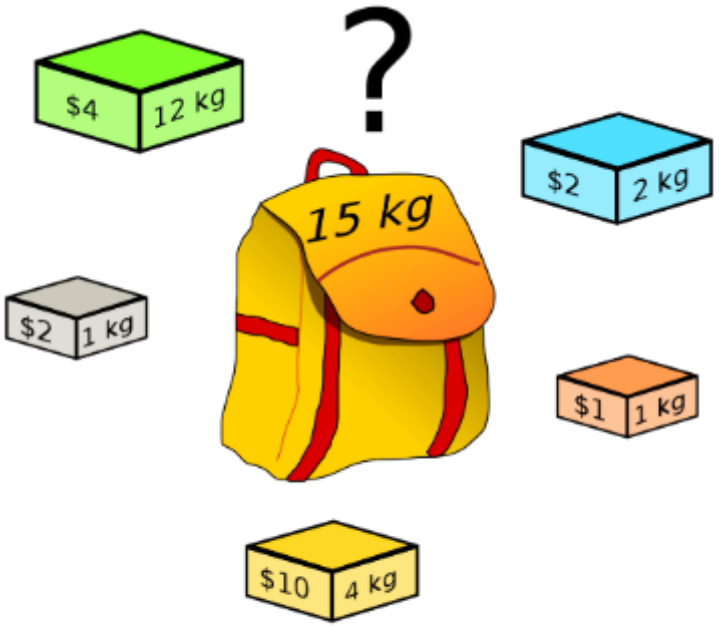
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states:

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem

Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

0/1 Knapsack Problem-

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. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Procedure:

Recursion by Brute-Force algorithm OR Exhaustive Search.

Approach: A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

***Optimal Sub-structure*:** To consider all subsets of items, there can be two cases for every item.

1. Case 1: The item is included in the optimal subset.
2. Case 2: The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from 'n' items is the max of the following twovalues.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).

2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of thenth item (including nth item).

If the weight of 'nth' item is greater than 'W', then the nth item cannot be included and Case 1 isthe only possibility.

Below is the implementation of the above approach:

```
class Knapsack {

	// A utility function that returns
	// maximum of two integers
	static int max(int a, int b) { return (a > b) ? a : b; }

	// Returns the maximum value that
	// can be put in a knapsack of
	// capacity W
	static int knapSack(int W, int wt[], int val[], int n)
	{
		// Base Case
		if (n == 0 || W == 0)
			return 0;

		// If weight of the nth item is
		// more than Knapsack capacity W,
		// then this item cannot be included
		// in the optimal solution
		if (wt[n - 1] > W)
			return knapSack(W, wt, val, n - 1);

		// Return the maximum of two cases:
		// (1) nth item included
		// (2) not included
		else
			return max(val[n - 1]+ knapSack(W - wt[n - 1], wt,val, n - 1),
			knapSack(W, wt, val, n - 1));
	}

	public static void main(String args[])
	{
		int val[] = new int[] { 60, 100, 120 };
		int wt[] = new int[] { 10, 20, 30 };
		int W = 50;
		int n = val.length;
		System.out.println(knapSack(W, wt, val, n));
	}
}
```

Ex.No:12 **Date:**

Aim:

Write a program to solve the Sum of subsets problem for a given set of distinct numbers using backtracking.

Description:

In this article, we will solve Subset Sum problem using a backtracking approach which will take O(2^N) time complexity but is significantly faster than the recursive approach which take exponential time as well.

Subset sum problem is the problem of finding a subset such that the sum of elements equals a given number. The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem.

A subset A of n positive integers and a value **sum** is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.

Algorithm:

subsetSum(set, subset, n, subSize, total, node, sum)

Input − The given set and subset, size of set and subset, a total of the subset, number of elements in the subset and the given sum.

Output − All possible subsets whose sum is the same as the given sum.

```
Begin
  if total = sum, then display the subset
    //go for finding next subset
    subsetSum(set, subset, , subSize-1, total-
    set[node], node+1, sum)return
  else
    for all element i in the set, dosubset[subSize] := set[i]
      subSetSum(set, subset, n, subSize+1,
    total+set[i], i+1, sum)done
End
```

```
import java.io.*;
class GFG {

	// Returns true if there is a subset
	// of set[] with sum equal to given sum
	static boolean isSubsetSum(int set[], int n, int sum)
	{
		// Base Cases
		if (sum == 0)
			return true;
		if (n == 0)
			return false;

		// If last element is greater than
		// sum, then ignore it
		if (set[n - 1] > sum)
			return isSubsetSum(set, n - 1, sum);

		/* else, check if sum can be obtained
		by any of the following
			(a) including the last element
			(b) excluding the last element */
		return isSubsetSum(set, n - 1, sum)
			|| isSubsetSum(set, n - 1, sum - set[n - 1]);
	}
```

```
        /* Driver code */
        public static void main(String args[])

{
                int set[] = { 3, 34, 4, 12, 5, 2 };
                int sum = 9;
                int n = set.length;
                if (isSubsetSum(set, n, sum) == true)
                        System.out.println("Found a subset"
                                                + " with given sum");
                else
                        System.out.println("No subset with"
                                                + " given sum");
        }
}
```

Ex.No:13 **Date:**

Aim:

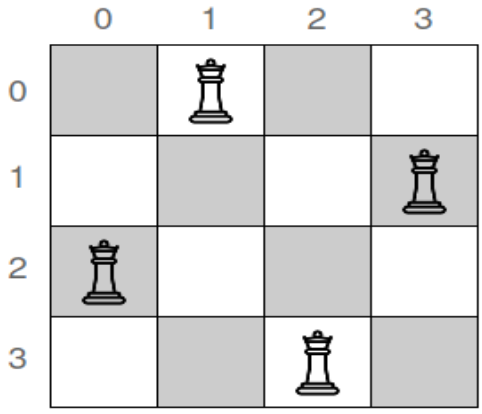
Implement N Queens problem using Backtracking.

Description:

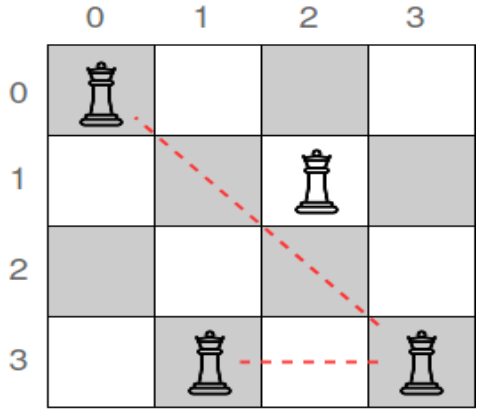
N Queens problem :

Place N queens on a chessboard of dimension N x N i.e N rows x N columns, such that no twoqueens can attack each other.

Consider below chessboards of size 4, the board on the left side is valid in which no two queens can attack each other; whereas the board on the right is invalid.



Valid queen positions



Invalid queen positions

Algorithm : N Queens

bool IsBoardOk (chessboard, row R, column C) {

If there is a queen 'Q' positioned to the left of column C in

row R, thenreturn False;

If there is queen 'Q' positioned on the upper left

diagonal, thenreturn false;

If there is queen 'Q' positioned on the lower left

diagonal, then

return false;

return true;

```
/* Java program to solve N Queen Problem using
backtracking */
public class NQueenProblem {
	final int N = 4;

	/* A utility function to print solution */
	void printSolution(int board[][])
	{
		for (int i = 0; i < N; i++) {
			for (int j = 0; j < N; j++)
				System.out.print(" " + board[i][j]
												+ " ");
			System.out.println();
		}
	}

	/* A utility function to check if a queen can
	be placed on board[row][col]. Note that this
	function is called when "col" queens are already
	placed in columns from 0 to col -1. So we need
	to check only left side for attacking queens */
	boolean isSafe(int board[][], int row, int col)
	{
		int i, j;

		/* Check this row on left side */
		for (i = 0; i < col; i++)
			if (board[row][i] == 1)
				return false;

		/* Check upper diagonal on left side */
		for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
			if (board[i][j] == 1)
				return false;

		/* Check lower diagonal on left side */
		for (i = row, j = col; j >= 0 && i < N; i++, j--)
			if (board[i][j] == 1)
				return false;

		return true;
	}

	/* A recursive utility function to solve N
	Queen problem */
	boolean solveNQUtil(int board[][], int col)
	{
		/* base case: If all queens are placed
		then return true */
		if (col >= N)
			return true;
```

```
                for (int i = 0; i < N; i++) {
                        /* Check if the queen can be placed on
                        board[i][col] */
                        if (isSafe(board, i, col)) {
                                /* Place this queen in board[i][col] */
                                board[i][col] = 1;

                                /* recur to place rest of the queens */
                                if (solveNQUtil(board, col + 1) == true)
                                        return true;

                                /* If placing queen in board[i][col]
                                doesn't lead to a solution then
                                remove queen from board[i][col] */
                                board[i][col] = 0; // BACKTRACK
                        }
                }


        }
        boolean solveNQ()
        {
                int board[][] = { { 0, 0, 0, 0 },
                                                { 0, 0, 0, 0 },
                                                { 0, 0, 0, 0 },
                                                { 0, 0, 0, 0 } };

                if (solveNQUtil(board, 0) == false) {
                        System.out.print("Solution does not exist");
                        return false;
                }

                printSolution(board);
                return true;
        }

        // driver program to test above function
        public static void main(String args[])
        {
                NQueenProblem Queen = new NQueenProblem();
                Queen.solveNQ();
        }
}
// This code is contributed by Abhishek Shankhadhar
```