SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
DESIGN AND ANALYSIS OF ALGORITHMS LAB

LIST OF PROGRAMS

1. Write a programs to implement the following:
   a) Prim's algorithm.
   b) Kruskal's algorithm.
2. Implementation of quicksort using divide and conquer methodology
3. Implementation of mergesort using divide and conquer methodlogy
4. Implementation of binary serach using divide and conquer methodlogy
5. Write a program to find optimal ordering of matrix multiplication. (Note: Use Dynamic programming method).
6. Write a program that implements backtracking algorithm to solve the problem i.e. place eight non-attacking queens on the board.
7. Write a program to find the strongly connected components in a digraph.
8. Write a program to implement file compression (and un-compression) using Huffman's algorithm.
9. Write a program to implement dynamic programming algorithm to solve all pairs shortest path problem.
10. Write a program to solve 0/1 knapsack problem using Greedy algorithm.
11. Write a program to solve 0/1 knapsack problem using Dynamic programming algorithm.
12. Write a program to solve 0/1 knapsack problem using Backtracking algorithm.
13. Write a program to solve 0/1 knapsack problem using Dynamic Programming
14. Write a program that uses dynamic programming algorithm to solve the optimal binary search tree problem.
15. Write a program for solving traveling sales person's problem using Dynamic programming algorithm.

1. Write a programs to implement the following:

a) Prim’s algorithm.

b) Kruskal’s algorithm.

```
import java.util.InputMismatchException;
import java.util.Scanner;
 public class Prims
{
    private boolean unsettled[];
    private boolean settled[];
    private int numberofvertices;
    private int adjacencyMatrix[][];
    private int key[];
    public static final int INFINITE = 999;
    private int parent[];

    public Prims(int numberofvertices)
    {
        this.numberofvertices = numberofvertices;
        unsettled = new boolean[numberofvertices + 1];
        settled = new boolean[numberofvertices + 1];
        adjacencyMatrix = new int[numberofvertices + 1][numberofvertices +
1];
        key = new int[numberofvertices + 1];
        parent = new int[numberofvertices + 1];
    }

    public int getUnsettledCount(boolean unsettled[])
    {
        int count = 0;
        for (int index = 0; index < unsettled.length; index++)
        {
            if (unsettled[index])
            {
                count++;
            }
        }
        return count;
    }

    public void primsAlgorithm(int adjacencyMatrix[][])
    {
        int evaluationVertex;
        for (int source = 1; source <= numberofvertices; source++)
        {
            for (int destination = 1; destination <= numberofvertices; destinati
on++)
            {
                this.adjacencyMatrix[source][destination] = adjacencyMatrix[sou
rce][destination];
            }
        }

        for (int index = 1; index <= numberofvertices; index++)
        {
            key[index] = INFINITE;
        }
        key[1] = 0;
        unsettled[1] = true;
        parent[1] = 1;

        while (getUnsettledCount(unsettled) != 0)
        {
            evaluationVertex = getMimumKeyVertexFromUnsettled(unsettled);
            unsettled[evaluationVertex] = false;
            settled[evaluationVertex] = true;
            evaluateNeighbours(evaluationVertex);
        }
    }

    private int getMimumKeyVertexFromUnsettled(boolean[] unsettled2)
    {
        int min = Integer.MAX_VALUE;
        int node = 0;
        for (int vertex = 1; vertex <= numberofvertices; vertex++)
        {
            if (unsettled[vertex] == true && key[vertex] < min)
            {
                node = vertex;
                min = key[vertex];
            }
        }
        return node;
    }

    public void evaluateNeighbours(int evaluationVertex)
    {

        for (int destinationvertex = 1; destinationvertex <= numberofvertices;
destinationvertex++)
        {
            if (settled[destinationvertex] == false)
            {
                if (adjacencyMatrix[evaluationVertex][destinationvertex] != INFIN
ITE)
                {
                    if (adjacencyMatrix[evaluationVertex][destinationvertex] < key
[destinationvertex])
                    {
                        key[destinationvertex] = adjacencyMatrix[evaluationVertex]
[destinationvertex];
                        parent[destinationvertex] = evaluationVertex;
                    }
                    unsettled[destinationvertex] = true;
                }
            }
        }
    }

    public void printMST()
    {
        System.out.println("SOURCE  : DESTINATION = WEIGHT");
        for (int vertex = 2; vertex <= numberofvertices; vertex++)
        {
            System.out.println(parent[vertex] + "\t:\t" + vertex +"\t=\t"+ adjac
encyMatrix[parent[vertex]][vertex]);
        }
    }

    public static void main(String... arg)
    {
        int adjacency_matrix[][];
        int number_of_vertices;
        Scanner scan = new Scanner(System.in);

        try
        {
            System.out.println("Enter the number of vertices");
            number_of_vertices = scan.nextInt();
            adjacency_matrix = new int[number_of_vertices + 1][number_of_ve
rtices + 1];
```

```
            System.out.println("Enter the Weighted Matrix for the graph");
            for (int i = 1; i <= number_of_vertices; i++)
            {
                for (int j = 1; j <= number_of_vertices; j++)
                {
                    adjacency_matrix[i][j] = scan.nextInt();
                    if (i == j)
                    {
                        adjacency_matrix[i][j] = 0;
                        continue;
                    }
                    if (adjacency_matrix[i][j] == 0)
                    {
                        adjacency_matrix[i][j] = INFINITE;
                    }
                }
            }

            Prims prims = new Prims(number_of_vertices);
            prims.primsAlgorithm(adjacency_matrix);
            prims.printMST();

        } catch (InputMismatchException inputMismatch)
        {
            System.out.println("Wrong Input Format");
        }
        scan.close();
    }
}
```

```
Enter the number of vertices
5
```

```
Enter the Weighted Matrix for the graph
0 4 0 0 5
4 0 3 6 1
0 3 0 6 2
0 6 6 0 7
5 1 2 7 0
SOURCE  : DESTINATION   =    WEIGHT
1       :     2         =       4
5       :     3         =       2
2       :     4         =       6
2       :     5         =       1
```

KRUSKALS ALGORITHM

```
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;

public class KruskalAlgorithm
{
    private List<Edge> edges;
    private int numberOfVertices;
    public static final int MAX_VALUE = 999;
    private int visited[];
    private int spanning_tree[][];

    public KruskalAlgorithm(int numberOfVertices)
    {
        this.numberOfVertices = numberOfVertices;
        edges = new LinkedList<Edge>();
        visited = new int[this.numberOfVertices + 1];
        spanning_tree = new int[numberOfVertices + 1][numberOfVertices + 1
];
    }

    public void kruskalAlgorithm(int adjacencyMatrix[][])
    {
        boolean finished = false;
        for (int source = 1; source <= numberOfVertices; source++)
        {
            for (int destination = 1; destination <= numberOfVertices; destinati
on++)
            {
                if (adjacencyMatrix[source][destination] != MAX_VALUE && sou
rce != destination)
                {
                    Edge edge = new Edge();
                    edge.sourcevertex = source;
                    edge.destinationvertex = destination;
                    edge.weight = adjacencyMatrix[source][destination];
                    adjacencyMatrix[destination][source] = MAX_VALUE;
                    edges.add(edge);
                }
            }
        }
        Collections.sort(edges, new EdgeComparator());
        CheckCycle checkCycle = new CheckCycle();
        for (Edge edge : edges)
        {
            spanning_tree[edge.sourcevertex][edge.destinationvertex] = edge.w
eight;
            spanning_tree[edge.destinationvertex][edge.sourcevertex] = edge.w
eight;
```

```
            if (checkCycle.checkCycle(spanning_tree, edge.sourcevertex))
            {
                spanning_tree[edge.sourcevertex][edge.destinationvertex] = 0;
                spanning_tree[edge.destinationvertex][edge.sourcevertex] = 0;
                edge.weight = -1;
                continue;
            }
            visited[edge.sourcevertex] = 1;
            visited[edge.destinationvertex] = 1;
            for (int i = 0; i < visited.length; i++)
            {
                if (visited[i] == 0)
                {
                    finished = false;
                    break;
                } else
                {
                    finished = true;
                }
            }
            if (finished)
                break;
        }
        System.out.println("The spanning tree is ");
        for (int i = 1; i <= numberOfVertices; i++)
            System.out.print("\t" + i);
        System.out.println();
        for (int source = 1; source <= numberOfVertices; source++)
        {
            System.out.print(source + "\t");
            for (int destination = 1; destination <= numberOfVertices; destinati
on++)
            {
                System.out.print(spanning_tree[source][destination] + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String... arg)
    {
        int adjacency_matrix[][];
        int number_of_vertices;

        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the number of vertices");
        number_of_vertices = scan.nextInt();
        adjacency_matrix = new int[number_of_vertices + 1][number_of_vertic
es + 1];

        System.out.println("Enter the Weighted Matrix for the graph");
        for (int i = 1; i <= number_of_vertices; i++)
        {
            for (int j = 1; j <= number_of_vertices; j++)
            {
                adjacency_matrix[i][j] = scan.nextInt();
                if (i == j)
                {
                    adjacency_matrix[i][j] = 0;
                    continue;
                }
                if (adjacency_matrix[i][j] == 0)
                {
                    adjacency_matrix[i][j] = MAX_VALUE;
                }
            }
        }
        KruskalAlgorithm kruskalAlgorithm = new KruskalAlgorithm(number
_of_vertices);
        kruskalAlgorithm.kruskalAlgorithm(adjacency_matrix);
        scan.close();
    }
}

class Edge
{
    int sourcevertex;
    int destinationvertex;
    int weight;
}

class EdgeComparator implements Comparator<Edge>
{
    @Override
    public int compare(Edge edge1, Edge edge2)
    {
        if (edge1.weight < edge2.weight)
            return -1;
        if (edge1.weight > edge2.weight)
            return 1;
        return 0;
    }
}

class CheckCycle
{
    private Stack<Integer> stack;
    private int adjacencyMatrix[][];
    public CheckCycle()
    {
        stack = new Stack<Integer>();
    }

    public boolean checkCycle(int adjacency_matrix[][], int source)
    {
        boolean cyclepresent = false;
        int number_of_nodes = adjacency_matrix[source].length - 1;

        adjacencyMatrix = new int[number_of_nodes + 1][number_of_nodes +
1];
        for (int sourcevertex = 1; sourcevertex <= number_of_nodes; sourceve
rtex++)
        {
            for (int destinationvertex = 1; destinationvertex <= number_of_nod
es; destinationvertex++)
            {
                adjacencyMatrix[sourcevertex][destinationvertex] = adjacency_m
atrix[sourcevertex[destinationvertex];
            }
        }

         int visited[] = new int[number_of_nodes + 1];
         int element = source;
         int i = source;
         visited[source] = 1;
         stack.push(source);

         while (!stack.isEmpty())
         {
             element = stack.peek();
             i = element;
            while (i <= number_of_nodes)
            {
                if (adjacencyMatrix[element][i] >= 1 && visited[i] == 1)
                {
                    if (stack.contains(i))
                    {
                        cyclepresent = true;
                        return cyclepresent;
                    }
                }
                if (adjacencyMatrix[element][i] >= 1 && visited[i] == 0)
                {
                    stack.push(i);
                    visited[i] = 1;
                    adjacencyMatrix[element][i] = 0;// mark as labelled;
                    adjacencyMatrix[i][element] = 0;
                    element = i;
                    i = 1;
                    continue;
                 }
                 i++;
            }
            stack.pop();
        }
        return cyclepresent;
    }
}
```

```
Enter the number of vertices
6
Enter the Weighted Matrix for the graph
```

```
0 6 8 6 0 0
6 0 0 5 10 0
8 0 0 7 5 3
6 5 7 0 0 0
0 10 5 0 0 3
0 0 3 0 3 0
The spanning tree is
        1       2       3       4       5       6
1       0       6       0       0       0       0
2       6       0       0       5       0       0
3       0       0       0       7       0       3
4       0       5       7       0       0       0
5       0       0       0       0       0       3
6       0       0       3       0       3       0
```

2. Implementation of quicksort using divide and conquer methodlogy

```
#include<stdio.h>
int a[5],mid;
main()
{
 int i;
 printf("enter the array elements:");
 for(i=0;i<5;i++)
 scanf("%d",&a[i]);

 quicksort(0,4);
 printf("the sorted array is ");
 for(i=0;i<5;i++)
 printf("%d\t",a[i]);
 getch();
}
```

```
quicksort(int first,int last)
{
 if(first<last)
 {
  mid=partition(first,last);
  if((mid-1)>first)
  quicksort(first,mid-1);
  if((mid+1)<last)
  quicksort((mid+1),last);
 }
 return;
}
int partition(int first,int last)
{
 int temp,pivot,up,down;
 pivot=a[first];
 up=last;
 down=first;
 while(down<up)
 {
  while((a[down]<=pivot) && (down<last))
  down++;
  while(a[up]>pivot)
  up--;
  if(down<up)
  {
   temp=a[down];
   a[down]=a[up];
   a[up]=temp;
  }
 }
 a[first]=a[up];
 a[up]=pivot;
 pivot=up;
 return pivot;
}
_
```

3. Implementation of mergesort using divide and conquer methodology

```
void mergesort(int,int);
void merge(int,int,int);
int a[6],b[6];
main()
{
 int i;
 printf("enter 5 elements ");
 for(i=1;i<6;i++)
 scanf("%d",&a[i]);
 mergesort(1,5);
 printf("the sorted order is");
 for(i=1;i<6;i++)
 printf("%d\t",a[i]);
 getch();
}
void mergesort(int low,int high)
{
 int mid;
 if(low<high)
 {
  mid=(low+high)/2;
  mergesort(low,mid);
  mergesort(mid+1,high);
  merge(low,mid,high);
 }
}
void merge(int low,int mid,int high)
```

```
{
 int h,i,j,k;
 h=low;
 i=low;
 j=mid+1;
 while((h<=high)&&(j<=high))
 {
  if(a[h]<=a[j])
  {
   b[i]=a[h];
   h=h+1;
  }
  else
  {
   b[i]=a[j];
   j=j+1;
  }
  i=i+1;
 }
 if(h>mid)
 {
  for(k=j;k<=high;k++)
  {
   b[i]=a[k];
   i++;
  }
 }
 else
 {
  for(k=h;k<=mid;k++)
  {
   b[i]=a[k];
   i=i+1;
  }
 }
for(k=low;k<=high;k++)
a[k]=b[k];
}_
```

4. Implementation of binary search using divide and conquer Methodology

```
#include<stdio.h>
int binarysearch(int a[],int key,int l,int h)
{
 int mid,val;
 if(l>h)
 return(-1);
 else
 {
  mid=(l+h)/2;
  if(key==a[mid])
  return (mid+1);
  else
  {
   if(key>a[mid])
   {
    l=mid+1;
    val=binarysearch(a,key,l,h);
   }
   else
   {
    if(key<a[mid])
    {
     h=mid-1;
     val=binarysearch(a,key,l,h);
    }
```

```
      }
      return val;
     }
    }
   }
   main()
   {
    int len,i=0,key,pos,a[10];
    printf("enter the length of array: ");
    scanf("%d",&len);
    printf("enter array values:");
    for(i=0;i<len;i++)
    scanf("%d",&a[i]);
    printf("enter the key value:");
    scanf("%d",&key);
    pos=binarysearch(a,key,0,len-1);
    if(pos==-1)
    printf("element is not in array");
    else
    printf("element is in position %d",pos);
    getch();
   }_
```

5. Write a program to find optimal ordering of matrix multiplication. (Note: Use Dynamic programming method).

```
class MatrixChainMultiplication {
    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
    static int MatrixChainOrder(int p[], int i, int j)
    {
        if (i == j)
            return 0;
        int min = Integer.MAX_VALUE;

        for (int k = i; k < j; k++) {
            int count = MatrixChainOrder(p, i, k) +
                        MatrixChainOrder(p, k + 1, j) +
                        p[i - 1] * p[k] * p[j];

            if (count < min)
                min = count;
        }

        // Return minimum count
        return min;
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        int arr[] = new int[] { 1, 2, 3, 4, 3 };
        int n = arr.length;

        System.out.println("Minimum number of multiplications is "
                           + MatrixChainOrder(arr, 1, n - 1));
    }
}


class GfG
{

static int N = 4;
static int k = 1;
```

```
/* A utility function to print solution */
static void printSolution(int board[][])
{
    System.out.printf("%d-\n", k++);
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            System.out.printf(" %d ", board[i][j]);
        System.out.printf("\n");
    }
    System.out.printf("\n");
}

/* 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 */
static 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;
}
```

6. Write a program that implements backtracking algorithm to solve the problem i.e. place eight non-attacking queens on the board.

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

    /* Consider this column and try placing       this queen in all rows one
by one */
    boolean res = false;
    for (int i = 0; i < N; i++)
    {
        /* Check if queen can be placed on
        board[i][col] */
        if ( isSafe(board, i, col) )
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            // Make result true if any placement
            // is possible
            res = solveNQUtil(board, col + 1) || res;

            /* 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
        }
    }

     /* If queen can not be place in any row in            this column col then
return false */
    return res;
}

/* This function solves the N Queen problem using  Backtracking. It mainly
uses solveNQUtil() to  solve the problem. It returns false if queens
cannot be placed, otherwise return true and   prints placement of queens
in the form of 1s.   Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.*/
static void solveNQ()
{
    int board[][] = new int[N][N];

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

// Driver code
public static void main(String[] args)
{
    solveNQ();
}
}
```

```
1-
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0


2-
 0  1  0  0
 0  0  0  1
 1  0  0  0
 0  0  1  0
```

7. Write a program to find the strongly connected components in a digraph.

```
import java.util.HashMap;
import java.util.InputMismatchException;
import java.util.Map;
import java.util.Scanner;
import java.util.Stack;
```

```
public class StrongConnectedComponents
{
    private int leader = 0;
    private int[] leader_node;
    private int explore[];
    private int finishing_time_of_node[];
    private int finishing_time = 1;
    private int number_of_nodes;
    private Stack<Integer> stack;
    private Map<Integer, Integer> finishing_time_map;

    public StrongConnectedComponents(int number_of_nodes)
    {
        this.number_of_nodes = number_of_nodes;
        leader_node = new int[number_of_nodes + 1];
        explore = new int[number_of_nodes + 1];
        finishing_time_of_node = new int[number_of_nodes + 1];
        stack = new Stack<Integer>();
        finishing_time_map = new HashMap<Integer, Integer>();
    }

    public void strongConnectedComponent(int adjacency_matrix[][])
    {
        for (int i = number_of_nodes; i > 0; i--)
        {
            if (explore[i] == 0)
            {
                dfs_1(adjacency_matrix, i);
            }
        }
        int rev_matrix[][] = new int[number_of_nodes + 1][number_of_nodes +
1];
```

```
        for (int i = 1; i <= number_of_nodes; i++)
        {
            for (int j = 1; j <= number_of_nodes; j++)
            {
                if (adjacency_matrix[i][j] == 1)
                    rev_matrix[finishing_time_of_node[j]][finishing_time_of_node[i
]] = adjacency_matrix[i][j];
            }
        }

        for (int i = 1; i <= number_of_nodes; i++)
        {
            explore[i] = 0;
            leader_node[i] = 0;
        }

        for (int i = number_of_nodes; i > 0; i--)
        {
            if (explore[i] == 0)
            {
                leader = i;
                dfs_2(rev_matrix, i);
            }
        }
    }

    public void dfs_1(int adjacency_matrix[][], int source)
    {
        explore[source] = 1;
        stack.push(source);
        int i = 1;
        int element = source;
        while (!stack.isEmpty())
        {
            element = stack.peek();
            i = 1;
            while (i <= number_of_nodes)
            {
                if (adjacency_matrix[element][i] == 1 && explore[i] == 0)
                {
                    stack.push(i);
                    explore[i] = 1;
                    element = i;
                    i = 1;
                    continue;
                }
                i++;
            }
            int poped = stack.pop();
            int time = finishing_time++;
            finishing_time_of_node[poped] = time;
            finishing_time_map.put(time, poped);
        }
    }

    public void dfs_2(int rev_matrix[][], int source)
    {
        explore[source] = 1;
        leader_node[finishing_time_map.get(source)] = leader;
        stack.push(source);
        int i = 1;
        int element = source;
        while (!stack.isEmpty())
        {
            element = stack.peek();
            i = 1;
            while (i <= number_of_nodes)
            {
                if (rev_matrix[element][i] == 1 && explore[i] == 0)
                {
                    if (leader_node[finishing_time_map.get(i)] == 0)
                        leader_node[finishing_time_map.get(i)] = leader;
                    stack.push(i);
                    explore[i] = 1;
                    element = i;
                    i = 1;
                    continue;
                }
                i++;
            }
            stack.pop();
        }
    }

    public static void main(String... arg)
    {
        int number_of_nodes;
        Scanner scanner = null;
        try
        {
            System.out.println("Enter the number of nodes in the graph");
            scanner = new Scanner(System.in);
            number_of_nodes = scanner.nextInt();

            int adjacency_matrix[][] = new int[number_of_nodes + 1][number_o
f_nodes + 1];
            System.out.println("Enter the adjacency matrix");
            for (int i = 1; i <= number_of_nodes; i++)
```

```
                for (int j = 1; j <= number_of_nodes; j++)
                    adjacency_matrix[i][j] = scanner.nextInt();

            StrongConnectedComponents strong = new StrongConnectedCom
ponents(number_of_nodes);
            strong.strongConnectedComponent(adjacency_matrix);

            System.out.println("The Strong Connected Components are");
            for (int i = 1; i < strong.leader_node.length; i++)
            {
                System.out.println( "Node " + i+ "belongs to SCC"
                    + strong.finishing_time_map.get(strong.leader_node[i]));
            }
        } catch (InputMismatchException inputMismatch)
        {
            System.out.println("Wrong Input Format");
        }
    }
}
```

```
Enter the number of nodes in the graph
8
Enter the adjacency matrix
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 1 1 0 0 0
0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 1
1 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 1 0 0 0 0
The Strong Connected Components are
```

```
Node 1 belongs to SCC 2
Node 2 belongs to SCC 2
Node 3 belongs to SCC 8
Node 4 belongs to SCC 4
Node 5 belongs to SCC 8
Node 6 belongs to SCC 2
Node 7 belongs to SCC 2
Node 8 belongs to SCC 8
```

8. Write a program to implement file compression (and un-compression) using Huffman's algorithm.

```
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Comparator;

// node class is the basic structure
// of each node present in the Huffman - tree.
class HuffmanNode {

    int data;
    char c;

    HuffmanNode left;
    HuffmanNode right;
}

// comparator class helps to compare the node
// on the basis of one of its attribute.
// Here we will be compared
// on the basis of data values of the nodes.
class MyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y)
    {

        return x.data - y.data;
    }
}

public class Huffman {

    // recursive function to print the
    // huffman-code through the tree traversal.
    // Here s is the huffman - code generated.
    public static void printCode(HuffmanNode root, String s)
    {

        // base case; if the left and right are null
        // then its a leaf node and we print
        // the code s generated by traversing the tree.
        if (root.left
                == null
            && root.right
                   == null
            && Character.isLetter(root.c)) {

            // c is the character in the node
            System.out.println(root.c + ":" + s);

            return;
        }
    // if we go to left then add "0" to the code.
    // if we go to the right add"1" to the code.

    // recursive calls for left and
    // right sub-tree of the generated tree.
    printCode(root.left, s + "0");
    printCode(root.right, s + "1");
}

// main function
public static void main(String[] args)
{

    Scanner s = new Scanner(System.in);

    // number of characters.
    int n = 6;
    char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int[] charfreq = { 5, 9, 12, 13, 16, 45 };

    // creating a priority queue q.
    // makes a min-priority queue(min-heap).
    PriorityQueue<HuffmanNode> q
        = new PriorityQueue<HuffmanNode>(n, new MyComparator());

    for (int i = 0; i < n; i++) {

        // creating a Huffman node object
        // and add it to the priority queue.
        HuffmanNode hn = new HuffmanNode();

        hn.c = charArray[i];
        hn.data = charfreq[i];
    hn.left = null;
    hn.right = null;

    // add functions adds
    // the huffman node to the queue.
    q.add(hn);
}

// create a root node
HuffmanNode root = null;

// Here we will extract the two minimum value
// from the heap each time until
// its size reduces to 1, extract until
// all the nodes are extracted.
while (q.size() > 1) {

    // first min extract.
    HuffmanNode x = q.peek();
    q.poll();

    // second min extarct.
    HuffmanNode y = q.peek();
    q.poll();

    // new node f which is equal
    HuffmanNode f = new HuffmanNode();

    // to the sum of the frequency of the two nodes
    // assigning values to the f node.
    f.data = x.data + y.data;
    f.c = '-';
```

```
            // first extracted node as left child.
            f.left = x;

            // second extracted node as the right child.
            f.right = y;

            // marking the f node as the root node.
            root = f;

            // add this node to the priority-queue.
            q.add(f);
        }

        // print the codes by traversing the tree
        printCode(root, "");
    }
}
```

```
 f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111
```

9. Write a program to implement dynamic programming algorithm to solve all pairs shortest path problem.

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



class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
           Or we can say the initial values of shortest distances
           are based on shortest paths considering no intermediate
           vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
           vertices.
          ---> Before start of an iteration, we have shortest
               distances between all pairs of vertices such that
               the shortest distances consider only the vertices in
               set {0, 1, 2, .. k-1} as intermediate vertices.
          ----> After the end of an iteration, vertex no. k is added
                to the set of intermediate vertices and the set
                becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

void printSolution(int dist[][])
{
    System.out.println("The following matrix shows the shortest "+
                    "distances between every pair of vertices");
    for (int i=0; i<V; ++i)
    {
        for (int j=0; j<V; ++j)
        {
            if (dist[i][j]==INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j]+"   ");
        }
        System.out.println();
    }
}
```

```
    // Driver program to test above function
    public static void main (String[] args)
    {
        /* Let us create the following weighted graph
           10
        (0)------->(3)
        |         /|\
        5 |         |
        |         | 1
        \|/         |
        (1)------->(2)
           3           */
        int graph[][] = { {0,   5,  INF, 10},
                          {INF, 0,   3, INF},
                          {INF, INF, 0,   1},
                          {INF, INF, INF, 0}
                        };
        AllPairShortestPath a = new AllPairShortestPath();

        // Print the solution
        a.floydWarshall(graph);
    }
}
```

```
Following matrix shows the shortest distances between every pair of
vertices
      0      5      8      9
   INF      0      3      4
   INF    INF      0      1
   INF    INF    INF      0
```

10. Write a program to solve 0/1 knapsack problem using Greedy algorithm.

```
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));
    }

    // Driver program to test above function
    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));
    }
}
```

Output:

```
220
```

11. Write a program to solve 0/1 knapsack problem using Dynamic programming algorithm.

```
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)
    {
        int i, w;
        int K[][] = new int[n + 1][W + 1];

        // Build table K[][] in bottom up manner
        for (i = 0; i<= n; i++) {
            for (w = 0; w<= W; w++) {
                if (i == 0 || w == 0)
                    K[i][w] = 0;
                else if (wt[i - 1]<= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}

// Driver program to test above function
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));
}
}
```

Output:

```
220
```

12. Write a program to solve 0/1 knapsack problem using Backtracking algorithm.

```
public class JohnnyJanuan {

    static int valueAux = 0;    // added
    static int weightAux = 0;   // added

    public static void main(String[] args) {
        int[] weights = {10, 20, 30};
        int[] values = {60, 100, 120};
        int[] sol = new int[weights.length];
        int[] finalSol = new int[weights.length];
        int max = 50;



        knapsack(weights, values, max, 0, sol, finalSol);    // removed the two
parameters
        for (int i = 0; i < finalSol.length; i++) {
            System.out.println(finalSol[i] * weights[i]);
        }

    }

    public static void knapsack(int[] weights, int[] values, int max, int index,
int[] sol, int[] finalSol) {  // removed the parameters
        sol[index] = -1;
        int n = weights.length;
        while (sol[index] < 1) {
            sol[index] = sol[index] + 1;
            if (sum(index, sol, weights) <= max && index == n - 1) {
                System.out.println("Sol: " + Arrays.toString(sol));
                System.out.println("weight = " + sum(index, sol, weights));
                update(weights, values, max, index, sol, finalSol);
                System.out.println("******************************");
            } else if (index < n - 1) {    // changed
                knapsack(weights, values, max, index + 1, sol, finalSol);

            }
//            sol[index]=-1;   // removed this line

        }

    }

    private static int sum(int index, int[] weights, int[] sol) {
        int res = 0;
//        for (int i = 0; i < index; i++) {   // thrown out
        for (int i = 0; i < sol.length; i++) {
              if (sol[i] < 0 ) System.out.println("in sum: i = " + i + "   sol[i] = " +
sol[i]);
            res += sol[i] * weights[i];
        }
        return res;
    }

    private static void update(int[] weights, int[] values, int max, int index,
int[] sol, int[] finalSol) {  //removed the two parameters

        int totalValue = 0;
        int totalWeight = 0;

        for (int i = 0; i < weights.length; i++) {
            if (sol[i] == 1) {
                totalValue += values[i];
                totalWeight += weights[i];
            }
        }

        if (totalValue > valueAux) {
            for (int i = 0; i < weights.length; i++) {
                finalSol[i] = sol[i];
            }
            valueAux = totalValue;
            weightAux = totalWeight;
            System.out.println("new valueAux: " + valueAux);   // changed
        }

    }
```

```
}
```

13. Write a program to solve 0/1 knapsack problem using Branch and bound algorithm.

```
#include <bits/stdc++.h>
using namespace std;

// Stucture for Item which store weight and corresponding
// value of Item
struct Item
{
    float weight;
    int value;
};

// Node structure to store information of decision
// tree
struct Node
{
    // level  --> Level of node in decision tree (or index
    //             in arr[]
    // profit --> Profit of nodes on path from root to this
    //            node (including this node)
    // bound ---> Upper bound of maximum profit in subtree
    //            of this node/
    int level, profit, bound;
    float weight;
};

// Comparison function to sort Item according to
// val/weight ratio
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Returns bound of profit in subtree rooted with u.
// This function mainly uses Greedy solution to find
// an upper bound on maximum profit.
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
        return 0;

    // initialize bound on profit by current profit
    int profit_bound = u.profit;

    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
    int totweight = u.weight;

    // checking index condition and knapsack capacity
    // condition
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight    += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }

    // If k is not n, include last item partially for
    // upper bound on profit
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /
                                          arr[j].weight;

    return profit_bound;
}

// Returns maximum profit we can get with capacity W
int knapsack(int W, Item arr[], int n)
{
    // sorting Item on basis of value per unit
    // weight.
    sort(arr, arr + n, cmp);

    // make a queue for traversing the node
    queue<Node> Q;
    Node u, v;

    // dummy node at starting
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    // One by one extract an item from decision tree
    // compute profit of all children of extracted item
    // and keep saving maxProfit
    int maxProfit = 0;
    while (!Q.empty())
    {
        // Dequeue a node
        u = Q.front();
        Q.pop();
```

```
// If it is starting node, assign level 0
if (u.level == -1)
    v.level = 0;

// If there is nothing on next level
if (u.level == n-1)
    continue;

// Else if not last node, then increment level,
// and compute profit of children nodes.
v.level = u.level + 1;

// Taking current level's item add current
// level's weight and value to node u's
// weight and value
v.weight = u.weight + arr[v.level].weight;
v.profit = u.profit + arr[v.level].value;

// If cumulated weight is less than W and
// profit is greater than previous profit,
// update maxprofit
if (v.weight <= W && v.profit > maxProfit)
    maxProfit = v.profit;

// Get the upper bound on profit to decide
// whether to add v to Q or not.
v.bound = bound(v, n, W, arr);

// If bound value is greater than profit,
// then only push into queue for further
// consideration
if (v.bound > maxProfit)
```

```
            Q.push(v);

        // Do the same thing,  but Without taking
        // the item in knapsack
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }

    return maxProfit;
}

// driver program to test above function
int main()
{
    int W = 10;   // Weight of knapsack
    Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                  {5, 95}, {3, 30}};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum possible profit = "
         << knapsack(W, arr, n);

    return 0;
}
```

Output :

```
Maximum possible profit = 235
```

14. Write a program that uses dynamic programming algorithm to solve the optimal binary search tree problem.

```
public class GFG
{
    // A recursive function to calculate cost of
        // optimal binary search tree
    static int optCost(int freq[], int i, int j)
    {
       // Base cases
       if (j < i)      // no elements in this subarray
         return 0;
       if (j == i)     // one element in this subarray
         return freq[i];

       // Get sum of freq[i], freq[i+1], ... freq[j]
       int fsum = sum(freq, i, j);

       // Initialize minimum value
       int min = Integer.MAX_VALUE;

       // One by one consider all elements as root and
           // recursively find cost of the BST, compare the
           // cost with min and update min if needed
       for (int r = i; r <= j; ++r)
       {
           int cost = optCost(freq, i, r-1) +
                          optCost(freq, r+1, j);
           if (cost < min)
              min = cost;
       }

       // Return minimum value
       return min + fsum;
    }

    // The main function that calculates minimum cost of
        // a Binary Search Tree. It mainly uses optCost() to
        // find the optimal cost.
    static int optimalSearchTree(int keys[], int freq[], int n)
    {
         // Here array keys[] is assumed to be sorted in
             // increasing order. If keys[] is not sorted, then
             // add code to sort keys, and rearrange freq[]
             // accordingly.
         return optCost(freq, 0, n-1);
    }

    // A utility function to get sum of array elements
        // freq[i] to freq[j]
    static int sum(int freq[], int i, int j)
    {
        int s = 0;
        for (int k = i; k <=j; k++)
           s += freq[k];
        return s;
    }

    // Driver code
    public static void main(String[] args) {
        int keys[] = {10, 12, 20};
        int freq[] = {34, 8, 50};
        int n = keys.length;
        System.out.println("Cost of Optimal BST is " +
                         optimalSearchTree(keys, freq, n));
    }
}
```

Output:

```
Cost of Optimal BST is 142
```