DESIGN AND ANALYSIS OF ALGORITHMS LAB

LAB MANUAL

Subject Code: CS505PC

Regulations : R16-JNTUH

Class : III Year B.Tech. CSE and IT I Semester

Prepared By

Mrs. CHANNABASAMMA, Asst.Prof, CSE
Mr. P SRINIVAS RAO. Asst.Prof,CSE
Mr.K. NAGAHARIBABU, Asst.Prof,CSE

Department of Computer Science & Engineering

and

Information Technology

BHARAT INSTITUTE OF ENGINEERING AND TECHNOLOGY

Ibrahimpatnam - 501 510, Hyderabad

VISION AND MISSION OF THE INSTITUTION

Vision

To achieve the autonomous and university status and spread universal education by inculcating discipline, character and knowledge into the young minds and mould them into enlightened citizens**.**

Mission

BIET's mission is to impart high quality education to students in the field of Engineering and Technology and to conduct advanced research programs by fostering close partnership with R&D institutions and Industry.

VISION AND MISSION OF CSE DEPARTMENT

Vision

Serving the high quality educational needs of local and rural students within the core areas of Computer Science and Engineering and Information Technology through a rigorous curriculum of theory, research and collaboration with other disciplines that is distinguished by its impact on academia, industry and society.

Mission

The Mission of the department of Computer Science and Engineering is

- To work closely with industry and research organizations to provide high quality computer education in both the theoretical and applications of Computer Science and Engineering.
- The department encourages original thinking, fosters research and development, evolve innovative applications of technology.

COMPUTER SCIENCE AND ENGINEERING

&

INFORMATION TECHNOLOGY

Program Educational Objectives (PEOs):

Program Educational Objective 1: (PEO1)
The graduates of Computer Science and Engineering will have successful career in technology or managerial functions.
Program Educational Objective 2: (PEO2)
The graduates of the program will have solid technical and professional foundation to continue higher studies.
Program Educational Objective 3: (PEO3)
The graduates of the program will have skills to develop products, offer services and create new knowledge.
Program Educational Objective 4: (PEO4)
The graduates of the program will have fundamental awareness of Industry processes, tools and technologies.

Program Outcomes (POs):

PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. .
PO2	**Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences
PO3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations
PO4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess Societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs):

PSO1	**Software Development and Research Ability:** Ability to understand the structure and development methodologies of software systems. Possess professional skills and knowledge of software design process. Familiarity and practical competence with a broad range of programming language and open source platforms. Use knowledge in various domains to identify research gaps and hence to provide solution to new ideas and innovations.
PSO2	**Foundation of mathematical concepts:** Ability to apply the acquired knowledge of basic skills, principles of computing, mathematical foundations, algorithmic principles, modeling and design of computer- based systems in solving real world engineering Problems.
PSO3	**Successful Career:** Ability to update knowledge continuously in the tools like Rational Rose, MATLAB, Argo UML, R Language and technologies like Storage, Computing, Communication to meet the industry requirements in creating innovative career paths for immediate employment and for higher studies.

ATTAINMENT OF PROGRAM OUTCOMES & PROGRAM SPECIFIC OUTCOMES

S. NO	NAME OF EXPERIMENT	Program Outcomes(POs) Attained	Program Specific Outcomes(PSOs) Attained
1	Write a java program to implement Bubble sort algorithm for sorting a list of integers in ascending order*****	PO1, PO2,PO4,PO5	**PSO1,PSO2**
2	Write a java program to implement Quick sort algorithm for sorting a list of integers in ascending order	PO1, PO2,PO4,PO5	**PSO1,PSO2**
3	Write a java program to implement Merge sort algorithm for sorting a list of integers in ascending order.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
4	Write a java program to implement the dfs algorithm for a graph.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
5	Write a java program to implement the bfs algorithm for a graph.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
6	Write a java programs to implement backtracking algorithm for the N-queens problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
7	Write a java program to implement the backtracking algorithm for the sum of subsets problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
8	Write a java program to implement the backtracking algorithm for the Hamiltonian Circuits problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
9	Write a java program to implement greedy algorithm for job sequencing with deadlines.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
10	Write a java program to implement Dijkstra's algorithm for the Single source shortest path problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
11	Write a java program that implements Prim's algorithm to generate minimum cost spanning tree.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
12	Write a java program that implements Kruskal's algorithm to generate minimum cost spanning tree	PO1, PO2,PO4,PO5	**PSO1,PSO2**
13	Write a java program to implement Floyd's algorithm for the all pairs shortest path problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
14	Write a java program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
15	Write a java program to implement Dynamic Programming algorithm for the Optimal Binary Search Tree Problem.	PO1, PO2,PO4,PO5	**PSO1,PSO2**
16	Write a java program to implement Greedy algorithm for the 0/1 Knapsack problem**.***	PO1, PO2,PO4,PO5	**PSO1,PSO2**

*Experiments out of syllabus.

System Requirements

1. Intel based desktop PC with minimum of 2.6GHz or faster processor with at least 1GB RAM and 40 GB free disk space and LAN connected.
2. Operating System: Ubuntu.
3. Software: JAVA Compiler

Lab Objectives and Outcomes

Course Objectives:

- To write programs in java to solve problems using divide and conquer strategy.
- To write programs in java to solve problems using backtracking strategy.
- To write programs in java to solve problems using greedy and dynamic programming techniques.

Course Outcomes:

Ability to write programs in java to solve problems using algorithm design techniques such as Divide and Conquer, Greedy, Dynamic programming, and Backtracking.

CO-PO Mapping:

Course outcomes	Program Outcomes (PO's)														
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PS02	PS03
CO1	3	3		2	2	-	-	-	-	-	-	-	2	2	-
Average	3	3		2	2	-	-	-	-	-	-	-	2	2	-

EXPERIMENT 1 - BUBBLE SORT

AIM:

Write a java program to implement Bubble sort algorithm for sorting a list of integers in ascending order

DESCRIPTION:

Bubble sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. Bubble sort can be practical if the input is in mostly sorted order with some out-of-order elements nearly in position.
Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted.

PROGRAM:

```
import java.util.Scanner;

class BubbleSort {
  public static void main(String []args) {
    int n, c, d, swap;
    Scanner in = new Scanner(System.in);

    System.out.println("Input number of integers to sort");
    n = in.nextInt();

    int array[] = new int[n];

    System.out.println("Enter " + n + " integers");

    for (c = 0; c < n; c++)
      array[c] = in.nextInt();

    for (c = 0; c < ( n - 1 ); c++) {
      for (d = 0; d < n - c - 1; d++) {
        if (array[d] > array[d+1]) /* For descending order use < */
        {
          swap       = array[d];
          array[d]   = array[d+1];
          array[d+1] = swap;
        }
      }
    }

    System.out.println("Sorted list of numbers");
```

```
    for (c = 0; c < n; c++)
      System.out.println(array[c]);
  }
}
```

OUTPUT:

```
Input number of integers to sort
5
Enter integers
9 6 7 3 2
Sorted list of numbers
2 3 6 7 9
```

EXPERIMENT 2 - QUICK SORT

AIM:

Write a java program to implement Quick sort algorithm for sorting a list of integers in ascending order

DESCRIPTION:

Quick sort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

PROGRAM:

```
import java.util.Random;
import java.util.Scanner;
public class quicksort {
static int max=2000;
int partition (int[] a, int low,int high)
{
int p,i,j,temp;
p=a[low];
i=low+1;
j=high;
while(low<high)
{
while(a[i]<=p&&i<high)
i++;
while(a[j]>p)
j--;
if(i<j)
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
else
{
temp=a[low];
a[low]=a[j];
a[j]=temp;
return j;
}
```

```
}
return j;
}
void sort(int[] a,int low,int high)
{
if(low<high)
{
int s=partition(a,low,high);
sort(a,low,s-1);
sort(a,s+1,high);
}
}
public static void main(String[] args) {
// TODO Auto-generated method stub
int[] a;
int i;
System.out.println("Enter the array size");
Scanner sc =new Scanner(System.in);
int n=sc.nextInt();
a= new int[max];
Random generator=new Random();
for( i=0;i<n;i++)
a[i]=generator.nextInt(20);
System.out.println("Array before sorting");
for( i=0;i<n;i++)
System.out.println(a[i]+" ");
long startTime=System.nanoTime();
quicksort m=new quicksort();
m.sort(a,0,n-1);
long stopTime=System.nanoTime();
long elapseTime=(stopTime-startTime);
System.out.println("Time taken to sort array is:"+elapseTime+"nano
seconds");
System.out.println("Sorted array is");
for(i=0;i<n;i++)
System.out.println(a[i]);
}
}
```

OUTPUT:

```
Enter the array size
10
Array before sorting
17
```

17
12
2
10
3
18
15
15
17
Time taken to sort array is:16980 nano seconds
Sorted array is
23
10
12
15
15
17
17
17
18

EXPERIMENT 3 - MERGE SORT

AIM:

Write a java program to implement Merge sort algorithm for sorting a list of integers in ascending order

DESCRIPTION:

Merge sort is a divide and conquer algorithm . Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

PROGRAM:

```
import java.util.Random;
import java.util.Scanner;
public class mergesort {
static int max=10000;
void merge( int[] array,int low, int mid,int high)
{
int i=low;
int j=mid+1;
int k=low;
int[]resarray;
resarray=new int[max];
while(i<=mid&&j<=high)
{
if(array[i]<array[j])
{
resarray[k]=array[i];
i++;
k++;
}
else
{
resarray[k]=array[j];
j++;
k++;
}
}
while(i<=mid)
resarray[k++]=array[i++];
while(j<=high)
resarray[k++]=array[j++];
for(int m=low;m<=high;m++)
```

```
array[m]=resarray[m];
}
void sort( int[] array,int low,int high)
{
if(low<high)
{
int mid=(low+high)/2;
sort(array,low,mid);
sort(array,mid+1,high);
merge(array,low,mid,high);
}
}
public static void main(String[] args) {
int[] array;
int i;
System.out.println("Enter the array size");
Scanner sc =new Scanner(System.in);
int n=sc.nextInt();
array= new int[max];
Random generator=new Random();
for( i=0;i<n;i++)
array[i]=generator.nextInt(20);
System.out.println("Array before sorting");
for( i=0;i<n;i++)
System.out.println(array[i]+" ");
long startTime=System.nanoTime();
mergesort m=new mergesort();
m.sort(array,0,n-1);
long stopTime=System.nanoTime();
long elapseTime=(stopTime-startTime);
System.out.println("Time taken to sort array is:"+elapseTime+"nano
seconds");
System.out.println("Sorted array is");
for(i=0;i<n;i++)
System.out.println(array[i]);
}
}
```

OUTPUT:

```
Enter the array size
10
Array before sorting
13
9
13
```

16
13
3
0
6
4
5
Time taken to sort array is:171277nano seconds
Sorted array is
0
3
4
5
6
9
13
13
13
16

EXPERIMENT 4 - DFS

AIM:

Write a java program to implement the dfs algorithm for a graph.

DESCRIPTION:

Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

PROGRAM:

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

public class DFS
{
   private Stack<Integer> stack;

   public DFS()
   {
      stack = new Stack<Integer>();
   }

   public void dfs(int adjacency_matrix[][], int source)
   {
      int number_of_nodes = adjacency_matrix[source].length - 1;

      int visited[] = new int[number_of_nodes + 1];
      int element = source;
      int i = source;
      System.out.print(element + "\t");
      visited[source] = 1;
      stack.push(source);

      while (!stack.isEmpty())
      {
         element = stack.peek();
         i = element;
       while (i <= number_of_nodes)
       {
          if (adjacency_matrix[element][i] == 1 && visited[i] == 0)
          {
                stack.push(i);
                visited[i] = 1;
```

```
                element = i;
                i = 1;
                System.out.print(element + "\t");
              continue;
             }
             i++;
        }
            stack.pop();
        }
    }

    public static void main(String...arg)
    {
       int number_of_nodes, source;
       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_of_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();

        System.out.println("Enter the source for the graph");
          source = scanner.nextInt();

          System.out.println("The DFS Traversal for the graph is given by ");
          DFS dfs = new DFS();
          dfs.dfs(adjacency_matrix, source);
       }catch(InputMismatchException inputMismatch)
       {
          System.out.println("Wrong Input format");
       }
       scanner.close();
    }
}
```

OUTPUT:

```
Enter the number of nodes in the graph
4
Enter the adjacency matrix
```

0 1 0 1
0 0 1 0
0 1 0 1
0 0 0 1
Enter the **source** for the graph
1
The DFS Traversal for the graph is given by
1 2 3 4

EXPERIMENT 5 - BFS

AIM:

Write a. java program to implement the bfs algorithm for a graph.

DESCRIPTION:

Breadth-first search (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

PROGRAM:

Write a java program to implement the BFS algorithm for a graph.

```
import java.util.InputMismatchException;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;
 public class BFS
{
   private Queue<Integer> queue;
    public BFS()
   {
      queue = new LinkedList<Integer>();
   }
    public void bfs(int adjacency_matrix[][], int source)
   {
      int number_of_nodes = adjacency_matrix[source].length - 1;

      int[] visited = new int[number_of_nodes + 1];
      int i, element;
       visited[source] = 1;
      queue.add(source);

      while (!queue.isEmpty())
      {
         element = queue.remove();
         i = element;
         System.out.print(i + "\t");
         while (i <= number_of_nodes)
         {
            if (adjacency_matrix[element][i] == 1 && visited[i] == 0)
            {
               queue.add(i);
               visited[i] = 1;
            }
            i++;
```

```
            }
        }
    }
    public static void main(String... arg)
    {
        int number_no_nodes, source;
        Scanner scanner = null;
        try
        {
            System.out.println("Enter the number of nodes in the graph");
            scanner = new Scanner(System.in);
            number_no_nodes = scanner.nextInt();
            int adjacency_matrix[][] = new int[number_no_nodes + 1][number_no_nodes + 1];
            System.out.println("Enter the adjacency matrix");
            for (int i = 1; i <= number_no_nodes; i++)
                for (int j = 1; j <= number_no_nodes; j++)
                    adjacency_matrix[i][j] = scanner.nextInt();
            System.out.println("Enter the source for the graph");
            source = scanner.nextInt();
            System.out.println("The BFS traversal of the graph is ");
            BFS bfs = new BFS();
            bfs.bfs(adjacency_matrix, source);

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

OUTPUT:

```
Enter the number of nodes in the graph
4
Enter the adjacency matrix
0 1 0 1
0 0 1 0
0 1 0 1
0 0 0 1
Enter the source for the graph
1
The BFS traversal of the graph is
1       2       4       3
```

EXPERIMENT 6 - N-QUEENS PROBLEM

AIM:

Write a java programs to implement backtracking algorithm for the N-queens problem.

DESCRIPTION:

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

PROGRAM:

```
import java.io.*;
class operation
{
  int x[]=new int[20];
int count=0;
public boolean place(int row,int column)
{
int i;
for(i=1;i<=row-1;i++)
{ //checking for column and diagonal conflicts
if(x[i] == column)
return false;
else
if(Math.abs(x[i] – column) == Math.abs(i – row))
return false;
}
return true;
}
public void Queen(int row,int n)
{
int column;
for(column=1;column<=n;column++)
{
if(place(row,column))
{
x[row] = column;
if(row==n)
print_board(n);//printing the board configuration
else //try next queen with next position
Queen(row+1,n);
}
}
}
```

```
public void print_board(int n)
{
int i;
System.out.println("\n\nSolution :"+(++count));
for(i=1;i<=n;i++)
{
System.out.print(" "+i);
}
for(i=1;i<=n;i++)
{
System.out.print("\n\n"+i);
for(int j=1;j<=n;j++)// for nXn board
{
if(x[i]==j)
System.out.print(" Q");
else
System.out.print(" -");
} }
} }
class BacktrackDemo
{
public static void main (String args[] )throws IOException
{
DataInputStream in=new DataInputStream(System.in);
System.out.println("Enter no Of Queens");
int n=Integer.parseInt(in.readLine());
operation op=new operation();
op.Queen(1,n);
}
}
```

OUTPUT:

```
Enter no Of Queens
4
Solution :1
1 2 3 4
1 – Q – –
2 – – – Q
3 Q – – –
4 – – Q –
Solution :2
1 2 3 4
1 – – Q –
2 Q – – –
3 – – – Q
4 – Q – –
```

EXPERIMENT 7 - SUM OF SUBSETS PROBLEM

AIM:

Write a java program to implement the backtracking algorithm for the sum of subsets problem.

DESCRIPTION:

The **subset sum problem** is an important problem in complexity theory and cryptography. The problem is this: given a set (or multiset) of integers, is there a non-empty subset whose sum is zero? For example, given the set {−7, −3, −2, 5, 8}, the answer is *yes* because the subset {−3, −2, 5} sums to zero. The problem is NP-complete, meaning roughly that while it is easy to confirm whether a proposed solution is valid, it may inherently be prohibitively difficult to determine in the first place whether any solution exists.

PROGRAM:

```
// A recursive solution for subset sum problem
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)
        {
                if (sum == 0)
                        return true;
                if (n == 0 && sum != 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]);
        }
        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");
        }
}
```

OUTPUT:

```
Found a subset with given sum
```

EXPERIMENT 8 - HAMILTONIAN CIRCUITS

AIM:

Write a java program to implement the backtracking algorithm for the Hamiltonian Circuits problem.

DESCRIPTION:

Hamiltonian cycle problem are problems of determining whether a Hamiltonian path (a path in an undirected or directed graph that visits each vertex exactly once) or a Hamiltonian cycle exists in a given graph (whether directed or undirected). Both problems are NP-complete.

PROGRAM:

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

/** Class HamiltonianCycle **/
public class HamiltonianCycle
{
   private int V, pathCount;
   private int[] path;
   private int[][] graph;

   /** Function to find cycle **/
   public void findHamiltonianCycle(int[][] g)
   {
      V = g.length;
      path = new int[V];

      Arrays.fill(path, -1);
      graph = g;
      try
      {
         path[0] = 0;
         pathCount = 1;
         solve(0);
         System.out.println("No solution");
      }
      catch (Exception e)
      {
         System.out.println(e.getMessage());
         display();
      }
   }
   /** function to find paths recursively **/
   public void solve(int vertex) throws Exception
```

```
{
   /** solution **/
   if (graph[vertex][0] == 1 && pathCount == V)
      throw new Exception("Solution found");
   /** all vertices selected but last vertex not linked to 0 **/
   if (pathCount == V)
      return;

   for (int v = 0; v < V; v++)
   {
      /** if connected **/
      if (graph[vertex][v] == 1 )
      {
         /** add to path **/
         path[pathCount++] = v;
         /** remove connection **/
         graph[vertex][v] = 0;
         graph[v][vertex] = 0;

         /** if vertex not already selected  solve recursively **/
         if (!isPresent(v))
            solve(v);

         /** restore connection **/
         graph[vertex][v] = 1;
         graph[v][vertex] = 1;
         /** remove path **/
         path[--pathCount] = -1;
      }
   }
}
/** function to check if path is already selected **/
public boolean isPresent(int v)
{
   for (int i = 0; i < pathCount - 1; i++)
      if (path[i] == v)
         return true;
   return false;
}
/** display solution **/
public void display()
{
   System.out.print("\nPath : ");
   for (int i = 0; i <= V; i++)
      System.out.print(path[i % V] +" ");
```

```
        System.out.println();
    }
        /** Main function **/
        public static void main (String[] args)
        {
            Scanner scan = new Scanner(System.in);
            System.out.println("HamiltonianCycle Algorithm Test\n");
            /** Make an object of HamiltonianCycle class **/
            HamiltonianCycle hc = new HamiltonianCycle();

            /** Accept number of vertices **/
            System.out.println("Enter number of vertices\n");
            int V = scan.nextInt();

            /** get graph **/
            System.out.println("\nEnter matrix\n");
            int[][] graph = new int[V][V];
            for (int i = 0; i < V; i++)
                for (int j = 0; j < V; j++)
                    graph[i][j] = scan.nextInt();

            hc.findHamiltonianCycle(graph);
        }
    }
```

OUTPUT:

```
HamiltonianCycle Algorithm Test
Enter number of vertices

8

Enter matrix

0 1 0 1 1 0 0 0
1 0 1 0 0 1 0 0
0 1 0 1 0 0 1 0
1 0 1 0 0 0 0 1
1 0 0 0 0 1 0 1
0 1 0 0 1 0 1 0
0 0 1 0 0 1 0 1
0 0 0 1 1 0 1 0
Solution found

Path : 0 1 2 3 7 6 5 4 0
```

EXPERIMENT 9 - JOB SEQUENCING WITH DEADLINES

AIM:

Write a java program to implement greedy algorithm for job sequencing with deadlines.

DESCRIPTION:

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1.

PROGRAM:

```
import java.util.*;
class job
{
int p;    //.............for profit of a job
int d;    //.............for deadline of a job
int v;    //.............for checking if that job has been selected
job()
{
p=0;
d=0;
v=0;
}
job(int x,int y,int z) // parameterised constructor
{
p=x;
d=y;
v=z;
} }
class js
{
static int n;
static int out(job jb[],int x)
{
for(int i=0;i<n;++i)
if(jb[i].p==x)
return i;
return 0;
}
public static void main(String args[])
{
Scanner scr=new Scanner(System.in);
System.out.println("Enter the number of jobs");
```

```
n=scr.nextInt();
int max=0;   // this is to find the maximum deadline
job jb[]=new job[n];
/************************Accepting job from user************************/
for(int i=0;i<n;++i)
{
System.out.println("Enter profit and deadline(p d)");
int p=scr.nextInt();
int d=scr.nextInt();

if(max<d)
max=d;     // assign maximum value of deadline to "max" variable
jb[i]=new job(p,d,0);  //zero as third parameter to mark that initially it is unvisited
}
//accepted jobs from user
/****************Sorting in increasing order of deadlines*********************/
for(int i=0;i<=n-2;++i)
{
 for(int j=i;j<=n-1;++j)
 {
   if(jb[i].d>jb[j].d)
   {
     job temp=jb[i];
     jb[i]=jb[j];
     jb[j]=temp;
   }
 }
}
// sorting process ends
/**************************Displaying the jobs to the user**********************/
System.out.println("The jobs are as follows ");
for(int i=0;i<n;++i)
System.out.println("Job "+i+" Profit = "+jb[i].p+" Deadline = "+jb[i].d);
// jobs displayed to the user
int count;
int hold[]=new int[max];
for(int i=0;i<max;++i)
hold[i]=0;
/************************Process of job sequencing begins**************************/
for(int i=0;i<n;++i)
{
count=0;
  for(int j=0;j<n;++j)
  {
```

```
 if(count<jb[j].d && jb[j].v==0 && count<max && jb[j].p>hold[count])
 {
  int ch=0;

 if(hold[count]!=0)
  {
 ch=out(jb,hold[count]);
  jb[ch].v=0;
 }
  hold[count]=jb[j].p;
 jb[j].v=1;
  ++count;
  } // end of if
  } //end of inner for
}// end of outer for
/**********************job sequencing process ends**************************/
/************************calculating max profit*********************************/
int profit=0;
for(int i=0;i<max;++i)
profit+=hold[i];
System.out.println("The maximum profit is "+profit);
}//end main method
}//end class
```

OUTPUT:

```
Enter the number of jobs
4
Enter profit and deadline(p d)
70 2
Enter profit and deadline(p d)
12 1
Enter profit and deadline(p d)
18 2
Enter profit and deadline(p d)
35 1
The jobs are as follows
Job 0 Profit = 12 Deadline = 1
Job 1 Profit = 35 Deadline = 1
Job 2 Profit = 18 Deadline = 2
Job 3 Profit = 70 Deadline = 2

The maximum profit is 105
```

EXPERIMENT 10 - SINGLE SOURCE SHORTEST PATH PROBLEM

AIM:

Write a java program to implement Dijkstra's algorithm for the Single source shortest path problem.

DESCRIPTION:

Single-Source Shortest Paths – **Dijkstra's Algorithm**. Given a source vertex s from set of vertices V in a **weighted** graph where all its edge weights w(u, v) are non-**negative**, find the shortest-path weights d(s, v) from given source s for all vertices v present in the graph.

PROGRAM:

```
import java.util.HashSet;
import java.util.InputMismatchException;
import java.util.Iterator;
import java.util.Scanner;
import java.util.Set;

public class DijkstraAlgorithmSet
{
   private int distances[];
   private Set<Integer> settled;
   private Set<Integer> unsettled;
   private int number_of_nodes;
   private int adjacencyMatrix[][];

   public DijkstraAlgorithmSet(int number_of_nodes)
   {
      this.number_of_nodes = number_of_nodes;
      distances = new int[number_of_nodes + 1];
      settled = new HashSet<Integer>();
      unsettled = new HashSet<Integer>();
      adjacencyMatrix = new int[number_of_nodes + 1][number_of_nodes + 1];
   }

   public void dijkstra_algorithm(int adjacency_matrix[][], int source)
   {
      int evaluationNode;
      for (int i = 1; i <= number_of_nodes; i++)
         for (int j = 1; j <= number_of_nodes; j++)
            adjacencyMatrix[i][j] = adjacency_matrix[i][j];

      for (int i = 1; i <= number_of_nodes; i++)
      {
         distances[i] = Integer.MAX_VALUE;
      }
```

```
            unsettled.add(source);
            distances[source] = 0;
            while (!unsettled.isEmpty())
            {
               evaluationNode = getNodeWithMinimumDistanceFromUnsettled();

      unsettled.remove(evaluationNode);
      settled.add(evaluationNode);
      evaluateNeighbours(evaluationNode);
   }
}

private int getNodeWithMinimumDistanceFromUnsettled()
{
   int min ;
   int node = 0;

   Iterator<Integer> iterator = unsettled.iterator();
   node = iterator.next();
   min = distances[node];
   for (int i = 1; i <= distances.length; i++)
   {
      if (unsettled.contains(i))
      {
         if (distances[i] <= min)
         {
            min = distances[i];
            node = i;
         }
      }
   }
   return node;
}

private void evaluateNeighbours(int evaluationNode)
{
   int edgeDistance = -1;
   int newDistance = -1;

   for (int destinationNode = 1; destinationNode <= number_of_nodes; destinationNode++)
   {
      if (!settled.contains(destinationNode))
      {
         if (adjacencyMatrix[evaluationNode][destinationNode] != Integer.MAX_VALUE)
```

```
            {
                edgeDistance = adjacencyMatrix[evaluationNode][destinationNode];
                newDistance = distances[evaluationNode] + edgeDistance;
                if (newDistance < distances[destinationNode])
                {
                    distances[destinationNode] = newDistance;
                }
                unsettled.add(destinationNode);
            }
        }
    }
}

public static void main(String... arg)
{
    int adjacency_matrix[][];
    int number_of_vertices;
    int source = 0;
    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_vertices + 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] =  Integer.MAX_VALUE;
                }
            }
        }

        System.out.println("Enter the source ");
        source = scan.nextInt();
```

```
        DijkstraAlgorithmSet dijkstrasAlgorithm = new DijkstraAlgorithmSet(number_of_vertices);
        dijkstrasAlgorithm.dijkstra_algorithm(adjacency_matrix, source);

        System.out.println("The Shorted Path to all nodes are ");
        for (int i = 1; i <= dijkstrasAlgorithm.distances.length - 1; i++)
        {
           System.out.println(source + " to " + i + " is "+ dijkstrasAlgorithm.distances[i]);
             }
          } catch (InputMismatchException inputMismatch)
          {
             System.out.println("Wrong Input Format");
          }
          scan.close();
       }
    }
```

OUTPUT:

```
$ javac DijkstraAlgorithmSet.java
$ java DijkstraAlgorithmSet
  Enter the number of vertices
5
Enter the Weighted Matrix for the graph
0 9 6 5 3
0 0 0 0 0
0 2 0 4 0
0 0 0 0 0
0 0 0 0 0
Enter the source
1
The Shorted Path to all nodes are
1 to 1 is 0
1 to 2 is 8
1 to 3 is 6
1 to 4 is 5
1 to 5 is 3
```

EXPERIMENT 11 - PRIM'S ALGORITHM

AIM:

Write a java program that implements Prim's algorithm to generate minimum cost spanning tree.

DESCRIPTION:

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

PROGRAM:

```
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; destination++)
     {
        this.adjacencyMatrix[source][destination] = adjacencyMatrix[source][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] != INFINITE)
                {
                    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"+
adjacencyMatrix[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_vertices + 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();
    }
}
```

OUTPUT:

```
$javac Prims.java
$java Prims

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
```

EXPERIMENT 12 - KRUSKAL'S ALGORITHM

AIM:

Write a java program that implements Kruskal's algorithm to generate minimum cost spanning tree.

DESCRIPTION:

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.[1] It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.[1] This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

PROGRAM:

```
// Java program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

        // Comparator function used for sorting edges
        // based 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

    // Creates a graph with V vertices and E 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()
```

```
{
   Edge result[] = new Edge[V];  // Tnis will store the resultant MST
   int e = 0;  // An index variable, used for result[]
   int i = 0;  // An index variable, used for sorted edges
   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 ssubsets
   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 = new Edge();
      next_edge = edge[i++];

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

      // If including this edge does'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");
    for (i = 0; i < e; ++i)
        System.out.println(result[i].src+" -- " +
            result[i].dest+" == " + result[i].weight);
}

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

    /* Let us create following weighted graph
            10
        0--------1
        | \     |
       6|   5\   |15
        |      \ |
        2--------3
            4       */
    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;

    graph.KruskalMST();
  }
   }
```

OUTPUT:

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

EXPERIMENT 13 - FLOYD'S ALGORITHM

AIM:

Write a java program to implement Floyd's algorithm for the all pairs shortest path problem.

DESCRIPTION:

Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).[1][2] A single execution of the algorithm will find the lengths (summed weights) of shortest paths between *all* pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions

of the algorithm can also be used for finding the transitive closure of a relation , or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

PROGRAM:

```
import java.util.PriorityQueue;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

class Vertex implements Comparable<Vertex>
{
public final String name;
public Edge[] adjacencies;
public double minDistance = Double.POSITIVE_INFINITY;
public Vertex previous;
public Vertex(String argName) { name = argName; }
public String toString() { return name; }
public int compareTo(Vertex other)
{
return Double.compare(minDistance, other.minDistance);
}

}


class Edge
{
public final Vertex target;
public final double weight;
public Edge(Vertex argTarget, double argWeight)
{ target = argTarget; weight = argWeight; }
}

public class Dijkstra
{
```

```
public static void computePaths(Vertex source)
{
source.minDistance = 0.;
PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();
vertexQueue.add(source);

while (!vertexQueue.isEmpty()) {
Vertex u = vertexQueue.poll();

// Visit each edge exiting u
for (Edge e : u.adjacencies)
{
        Vertex v = e.target;
        double weight = e.weight;
        double distanceThroughU = u.minDistance + weight;
if (distanceThroughU < v.minDistance) {
vertexQueue.remove(v);

v.minDistance = distanceThroughU ;
v.previous = u;
vertexQueue.add(v);
}
}
}
}

public static List<Vertex> getShortestPathTo(Vertex target)
{
List<Vertex> path = new ArrayList<Vertex>();
for (Vertex vertex = target; vertex != null; vertex = vertex.previous)
path.add(vertex);

Collections.reverse(path);
return path;
}

public static void main(String[] args)
{
// mark all the vertices
Vertex A = new Vertex("A");
Vertex B = new Vertex("B");
Vertex D = new Vertex("D");
Vertex F = new Vertex("F");
Vertex K = new Vertex("K");
Vertex J = new Vertex("J");
```

```
Vertex M = new Vertex("M");
Vertex O = new Vertex("O");
Vertex P = new Vertex("P");
Vertex R = new Vertex("R");
Vertex Z = new Vertex("Z");

// set the edges and weight
A.adjacencies = new Edge[]{ new Edge(M, 8) };
B.adjacencies = new Edge[]{ new Edge(D, 11) };
D.adjacencies = new Edge[]{ new Edge(B, 11) };
F.adjacencies = new Edge[]{ new Edge(K, 23) };
K.adjacencies = new Edge[]{ new Edge(O, 40) };
J.adjacencies = new Edge[]{ new Edge(K, 25) };
M.adjacencies = new Edge[]{ new Edge(R, 8) };
O.adjacencies = new Edge[]{ new Edge(K, 40) };
P.adjacencies = new Edge[]{ new Edge(Z, 18) };
R.adjacencies = new Edge[]{ new Edge(P, 15) };
Z.adjacencies = new Edge[]{ new Edge(P, 18) };


computePaths(A); // run Dijkstra
System.out.println("Distance to " + Z + ": " + Z.minDistance);
List<Vertex> path = getShortestPathTo(Z);
System.out.println("Path: " + path);
}
}
```

OUTPUT:

```
Distance to Z: 49.0
Path: [A, M, R, P, Z]
```

EXPERIMENT 14 - 0/1 KNAPSACK PROBLEM

AIM:

Write a java program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem.

DESCRIPTION:

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).

PROGRAM:

```
//This is the java program to implement the knapsack problem using Dynamic Programming
import java.util.Scanner;

public class Knapsack_DP
{
   static int max(int a, int b)
   {
      return (a > b)? a : b;
   }
   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];
   }

   public static void main(String args[])
   {
```

```
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of items: ");
        int n = sc.nextInt();
        System.out.println("Enter the items weights: ");
        int []wt = new int[n];
        for(int i=0; i<n; i++)
           wt[i] = sc.nextInt();

        System.out.println("Enter the items values: ");
        int []val = new int[n];
        for(int i=0; i<n; i++)
           val[i] = sc.nextInt();

        System.out.println("Enter the maximum capacity: ");
        int W = sc.nextInt();

        System.out.println("The maximum value that can be put in a knapsack of capacity W is: " +
    knapSack(W, wt, val, n));
        sc.close();
    }
}
```

OUTPUT:

```
$ javac Knapsack_DP.java
$ java Knapsack_DP

Enter the number of items:
5
Enter the items weights:
01 56 42 78 12
Enter the items values:
50 30 20 10 50
Enter the maximum capacity:
150
The maximum value that can be put in a knapsack of capacity W is: 150
```

EXPERIMENT 15 - OPTIMAL BINARY SEARCH TREE PROBLEM

AIM:

Write a java program to implement Dynamic Programming algorithm for the Optimal Binary Search Tree Problem.

DESCRIPTION:

optimal binary search tree (Optimal BST), sometimes called a **weight-balanced binary tree**,[1] is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities). Optimal BSTs are generally divided into two types: static and dynamic.

In the **static optimality** problem, the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities. Various algorithms exist to construct or approximate the statically optimal tree given the information on the access probabilities of the elements.

PROGRAM:

```
import java.io.*;
import java.util.*;
class Optimal
{
 public int p[];
 public int q[];
 public int a[];
 public int w[][];
 public int c[][];
 public int r[][];
 public int n;
 int front,rear,queue[];
 public Optimal(int SIZE)
 {
  p=new int[SIZE];
  q= new int[SIZE];
  a=new int[SIZE];
  w=new int[SIZE][SIZE];
  c=new int[SIZE][SIZE];
  r=new int[SIZE][SIZE];
  queue=new int[SIZE];
  front=rear=-1;
 }
/* This function returns a value in the range r[i][j-1] to r[i+1][j] SO that the cost c[i][k-1] + c[k][j] is
minimum  */

public int Min_Value(int i, int j)
{
  int m,k=0;
```

```
  int minimum = 32000;
  for (m=r[i][j-1] ; m<=r[i+1][j] ; m++)
  {
  if ((c[i][m-1]+c[m][j]) < minimum)
  {
  minimum = c[i][m-1] + c[m][j];
  k = m;
  }
  }
 return k;
}
/*  This function builds the table from all the given probabilities It basically computes C,r,W values */

public void OBST()
{
 int i, j, k, l, m;
 for (i=0 ; i<n ; i++)
 {
 //  Initialize
 w[i][i] = q[i];
 r[i][i] = c[i][i] = 0;
 //  Optimal trees with one node
 w[i][i+1] = q[i] + q[i+1] + p[i+1];
 r[i][i+1] = i+1;
 c[i][i+1] = q[i] + q[i+1] + p[i+1];
 }
 w[n][n] = q[n];
 r[n][n] = c[n][n] = 0;
 //  Find optimal trees with m nodes
  for (m=2 ; m<=n ; m++)
  {
  for (i=0 ; i<=n-m ; i++)
  {
   j = i+m;
   w[i][j] = w[i][j-1] + p[j] + q[j];
   k = Min_Value(i,j);
   c[i][j] = w[i][j] + c[i][k-1] + c[k][j];
   r[i][j] = k;
  }
  }
}

/*This function builds the tree from the tables made by the OBST function */

public void build_tree()
```

```
{
 int i, j, k;
 System.out.print("The Optimal Binary Search Tree For The Given Nodes Is ....\n");
 System.out.print("\n The Root of this OBST is :: "+r[0][n]);
 System.out.print("\n The Cost Of this OBST is :: "+c[0][n]);
 System.out.print("\n\n\tNODE\tLEFT CHILD\tRIGHT CHILD");
 System.out.println("\n ----------------------------------------------------");
 queue[++rear] = 0;
 queue[++rear] = n;
 while(front != rear)
 {
 i = queue[++front];
 j = queue[++front];
 k = r[i][j];
 System.out.print("\n         "+k);
 if (r[i][k-1] != 0)
 {
  System.out.print("              "+r[i][k-1]);
  queue[++rear] = i;
  queue[++rear] = k-1;
 }
  else
   System.out.print("              -");
  if(r[k][j] != 0)
  {
   System.out.print("              "+r[k][j]);
   queue[++rear] = k;
   queue[++rear] = j;
  }
  else
   System.out.print("              -");
  }
  System.out.println("\n");
}
}
/* This is the main function */
class OBSTDemo
{
 public static void main (String[] args )throws IOException,NullPointerException
{
  Optimal obj=new Optimal(10);
 int i;
 System.out.print("\n Optimal Binary Search Tree \n");
 System.out.print("\n Enter the number of nodes   ");
 obj.n=getInt();
```

```
 System.out.print("\n Enter the data as  ....\n");
  for (i=1;i<=obj.n;i++)
 {
  System.out.print("\n a["+i+"]");
  obj.a[i]=getInt();
 }
 for (i=1 ; i<=obj.n ; i++)
 {
  System.out.println("p["+i+"]");
  obj.p[i]=getInt();
  }
  for (i=0 ; i<=obj.n ; i++)
  {
   System.out.print("q["+i+"]");
   obj.q[i]=getInt();
  }
   obj.OBST();
   obj.build_tree();
}

public static String getString() throws IOException
 {
     InputStreamReader input = new InputStreamReader(System.in);
     BufferedReader b = new BufferedReader(input);
     String str = b.readLine();//reading the string from console
     return str;
 }

public static char getChar() throws IOException
 {
   String str = getString();
   return str.charAt(0);//reading first char of console string
 }
public static int getInt() throws IOException
 {
    String str = getString();
    return Integer.parseInt(str);//converting console string to numeric value
 }
}
```

OUTPUT:

Optimal Binary Search Tree
Enter the number of nodes 4
Enter the data as ....

a[1] 1
a[2] 2
a[3] 3
a[4] 4

p[1] 3
p[2] 3
p[3] 1
p[4] 1

q[0] 2
q[1] 3
q[2] 1
q[3] 1
q[4] 1

The Optimal Binary Search Tree For The Given Nodes Is ....

The Root of this OBST is :: 2
The Cost Of this OBST is :: 32

NODE	LEFT CHILD	RIGHT CHILD
2	1	3
1	-	-
3	-	4
4	-	-

EXPERIMENT 16 - 0/1 KNAPSACK PROBLEM USING GREEDY METHOD

AIM:

Implement in Java, the **0/1 Knapsack** problem using Greedy method.

DESCRIPTION:

The **continuous knapsack problem** (also known as the **fractional knapsack problem**) is an algorithmic problem in combinatorial optimization in which the goal is to fill a container (the "knapsack") with fractional amounts of different materials chosen to maximize the value of the selected materials.[1][2] It resembles the classic knapsack problem, in which the items to be placed in the container are indivisible; however, the continuous knapsack problem may be solved in polynomial time whereas the classic knapsack problem is NP-hard.[1] It is a classic example of how a seemingly small change in the formulation of a problem can have a large impact on its computational complexity.

PROGRAM:

```
import java.util.Scanner;
public class knapsacgreedy {
/**
* @param args
*/
public static void main(String[] args) {
int i,j=0,max_qty,m,n;
float sum=0,max;
Scanner sc = new Scanner(System.in);
int array[][]=new int[2][20];
System.out.println("Enter no of items");
n=sc.nextInt();
System.out.println("Enter the weights of each
items");
for(i=0;i<n;i++)
array[0][i]=sc.nextInt();
System.out.println("Enter the values of each
items");
for(i=0;i<n;i++)
array[1][i]=sc.nextInt();
System.out.println("Enter maximum volume of
knapsack :");
max_qty=sc.nextInt();
m=max_qty;
while(m>=0)
{
max=0;
for(i=0;i<n;i++)
{
if(((float)array[1][i])/((float)array[0][i])>max)
```

```
{
max=((float)array[1][i])/((float)array[0][i]);
j=i;
}
}
if(array[0][j]>m)
{
System.out.println("Quantity of item number: "
+ (j+1) + " added is " +m);
sum+=m*max;
m=-1;
}
else
{
System.out.println("Quantity of item
number: " + (j+1) + " added is " + array[0][j]);
m-=array[0][j];
sum+=(float)array[1][j];
array[1][j]=0;
}
```



```
}
System.out.println("The total profit is " + sum);
sc.close();
}
}
```

OUTPUT:

```
Enter no of items
4
Enter the weights of each items
2132
Enter the values of each items
12
10
20
15
Enter maximum volume of knapsack :
5
Quantity of item number: 2 added is 1
Quantity of item number: 4 added is 2
Quantity of item number: 3 added is 2
The total profit is 38.333332
```