DESIGN AND ANALYSIS OF ALGORITHMS LAB MANUAL

MALLA REDDY UNIVERSITY

Maisammaguda, Dulapally, Hyderabad, Telangana-500043

SCHOOL OF ENGINEERING
DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
DESIGN AND ANALYSIS OF ALGORITHMS
LAB MANUAL USING JAVA

PROGRAM OUTCOMES (POs)

B.Tech – Graduate should possess the following Program Outcomes.

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.

7. **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.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **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.

11. **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.

12. **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.

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to the starting time), those who come after 5 minutes will not be allowed into the lab.

2. Plan your task properly much before to the commencement, come prepared to the lab with the synopsis / program / experiment details.

3. Student should enter into the laboratory with:
a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
b. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
c. Proper Dress code and Identity card.

4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.

5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.

6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.

7. Computer labs are established with sophisticated and high-end branded systems, which should be utilized properly.

8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviors with the staff and systems etc., will attract severe punishment.

9. Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.

10. Students should LOG OFF/ SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

(MR22-1CS0409) DESIGN AND ANALYSIS OF ALGORITHMS LAB

COURSE OBJECTIVES:

This course will enable students to
1. Design and implement various algorithms in JAVA.
2. Employ various design strategies for problem solving.
3. Measure and compare the performance of different algorithms.

COURSE OUTCOMES:

1. Design algorithms using appropriate design Techniques (Greedy, Dynamic programming, etc.)
2. Implement a variety of algorithms such as sorting, graph related, etc., in a high-level language.
3. Analyze and compare the performance of algorithms using language features.
4. Apply and implement learned algorithm design techniques and datastructures to solve real-world problems.
5. Analyze algorithms to deduce their time complexities.
6. Choose appropriate algorithms techniques to solve computational problems.

INDEX

Exp. No:1(WEEK-1)	Implement n-Queen's problem using backtracking method in Java.

Algorithm:

Step 1: Start
Step 2: Read the no. of queens
Step 3: Find out solution for N-queens
Step 4: Print solution
Step 5: Stop

Program:

```
public class NQueenProblem {
    final int N = 4;

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

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

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

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

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

```
     return true;
}

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

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

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

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

    /* If the queen can not be placed in any row in
       this column col, then return false */
    return false;
}

/* 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.*/
boolean solveNQ()
{
    int board[][] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };
```

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


        printSolution(board);
        return true;
    }

    // driver program to test above function
    public static void main(String args[])
    {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }
}
```

Output:

```
0  0  1  0

1  0  0  0

0  0  0  1

0  1  0  0
```

Exp. No:2(WEEK-2)	Implement Sum of subsets problem using backtracking method in Java.

Algorithm:

Step 1: Start
Step 2: Read the set along with M the keyboard
Step 3: Find out sequence of sub set
Step 4: Print solution
Step 5: Stop

Program:

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

public class SubsetSum {

   // Flag to check if there exists a subset with the given
   // sum
   static boolean flag = false;

   // Print all subsets if there is at least one subset of
   // set[] with the sum equal to the given sum
   static void printSubsetSum(int i, int n, int[] set,
                              int targetSum,
                              List<Integer> subset)
   {
      // If targetSum is zero, then there exists a subset.
      if (targetSum == 0) {
         // Prints a valid subset
         flag = true;
         System.out.print("[ ");
         for (int j = 0; j < subset.size(); j++) {
            System.out.print(subset.get(j) + " ");
         }
         System.out.print("]");
         return;
      }

      if (i == n) {
         // Return if we have reached the end of the
         // array
         return;
      }

      // Not considering the current element
      printSubsetSum(i + 1, n, set, targetSum, subset);

      // Consider the current element if it is less than
```

```
        // or equal to the targetSum
        if (set[i] <= targetSum) {


            // Push the current element in the subset
            subset.add(set[i]);

            // Recursive call for considering the current
            // element
            printSubsetSum(i + 1, n, set,
                           targetSum - set[i], subset);

            // Pop-back element after the recursive call to
            // restore the subset's original configuration
            subset.remove(subset.size() - 1);
        }
    }

    // Driver code
    public static void main(String[] args)
    {
        // Test case 1
        int[] set1 = { 1, 2, 1 };
        int sum1 = 3;
        int n1 = set1.length;
        List<Integer> subset1 = new ArrayList<>();
        System.out.println("Output 1:");
        printSubsetSum(0, n1, set1, sum1, subset1);
        System.out.println();
        flag = false;

        // Test case 2
        int[] set2 = { 3, 34, 4, 12, 5, 2 };
        int sum2 = 30;
        int n2 = set2.length;
        List<Integer> subset2 = new ArrayList<>();
        System.out.println("Output 2:");
        printSubsetSum(0, n2, set2, sum2, subset2);
        if (!flag) {
            System.out.println("There is no such subset");
        }
    }
}
```

Output 1:

```
[ 2 1 ][ 1 2 ]
```

Output 2:

```
There is no such subset
```

Exp. No:3(WEEK-3)	Implement Graph Coloring problem using backtracking method in Java.

Algorithm:

Step 1: Start
Step 2: Read the no. of vertices, edges from the keyboard
Step 3: Assign colors to each vertex
Step 4: Print coloring of the vertex
Step 5: Stop

Program:

```
public class mColoringProblem {
    final int V = 4;
    int color[];

    /* A utility function to check
       if the current color assignment
       is safe for vertex v */
    boolean isSafe(int v, int graph[][], int color[], int c)
    {
        for (int i = 0; i < V; i++)
            if (graph[v][i] == 1 && c == color[i])
                return false;
        return true;
    }

    /* A recursive utility function
       to solve m coloring  problem */
    boolean graphColoringUtil(int graph[][], int m,
                              int color[], int v)
    {
        /* base case: If all vertices are
           assigned a color then return true */
        if (v == V)
            return true;

        /* Consider this vertex v and try
           different colors */
        for (int c = 1; c <= m; c++) {
            /* Check if assignment of color c to v
               is fine*/
            if (isSafe(v, graph, color, c)) {
                color[v] = c;

                /* recur to assign colors to rest
                   of the vertices */
                if (graphColoringUtil(graph, m, color,
                                      v + 1))
                    return true;

                /* If assigning color c doesn't lead
                   to a solution then remove it */
                color[v] = 0;
            }
        }

        /* If no color can be assigned to
           this vertex then return false */
        return false;
    }

    /* This function solves the m Coloring problem using
       Backtracking. It mainly uses graphColoringUtil()
       to solve the problem. It returns false if the m
       colors cannot be assigned, otherwise return true
       and  prints assignments of colors to all vertices.
       Please note that there  may be more than one
       solutions, this function prints one of the
       feasible solutions.*/
    boolean graphColoring(int graph[][], int m)
    {
        // Initialize all color values as 0. This
        // initialization is needed correct
        // functioning of isSafe()
        color = new int[V];
        for (int i = 0; i < V; i++)
            color[i] = 0;

        // Call graphColoringUtil() for vertex 0
        if (!graphColoringUtil(graph, m, color, 0)) {
            System.out.println("Solution does not exist");
            return false;
        }

        // Print the solution
        printSolution(color);
        return true;
    }

        /* A utility function to print solution */
    void printSolution(int color[])
    {
        System.out.println("Solution Exists: Following"
                           + " are the assigned colors");
        for (int i = 0; i < V; i++)
            System.out.print(" " + color[i] + " ");
        System.out.println();
    }
```

```
    // Driver code
    public static void main(String args[])
    {
        mColoringProblem Coloring = new mColoringProblem();


   /* Create following graph and
         test whether it is
         3 colorable

        (3)---(2)
          |   / |
          |  /  |
          | /   |
         (0)---(1)
        */
        int graph[][] = {
            { 0, 1, 1, 1 },
            { 1, 0, 1, 0 },
            { 1, 1, 0, 1 },
            { 1, 0, 1, 0 },
        };
        int m = 3; // Number of colors

        // Function call
        Coloring.graphColoring(graph, m);
    }
}
```

Output:

```
Solution Exists: Following are the assigned colors
 1  2  3  2
```

Exp. No:4(WEEK-4)	Implement Hamiltonian Cycle Problem using backtracking method in Java.

Algorithm:

Step 1: Start
Step 2: Select a starting point
Step 3: Move to the nearest unvisited vertex
Step 4: Repeat until the circuit is complete
Step 5: Stop

Program:

```
class HamiltonianCycle
{
    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
       added at index 'pos'in the Hamiltonian Cycle
       constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][], int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
           the previously added vertex. */
        if (graph[path[pos - 1]][v] == 0)
            return false;

        /* Check if the vertex has already been included.
           This step can be optimized by creating an array
           of size V */
        for (int i = 0; i < pos; i++)
            if (path[i] == v)
                return false;

        return true;
    }

    /* A recursive utility function to solve hamiltonian
       cycle problem */
    boolean hamCycleUtil(int graph[][], int path[], int pos)
    {
        /* base case: If all vertices are included in
           Hamiltonian Cycle */
        if (pos == V)
        {
            // And if there is an edge from the last included
            // vertex to the first vertex
            if (graph[path[pos - 1]][path[0]] == 1)
                return true;
            else
                return false;
        }

        // Try different vertices as a next candidate in
        // Hamiltonian Cycle. We don't try for 0 as we
        // included 0 as starting point in hamCycle()
        for (int v = 1; v < V; v++)
        {

/* Check if this vertex can be added to Hamiltonian
              Cycle */
            if (isSafe(v, graph, path, pos))
            {
                path[pos] = v;

                /* recur to construct rest of the path */
                if (hamCycleUtil(graph, path, pos + 1) == true)
                    return true;

                /* If adding vertex v doesn't lead to a solution,
                   then remove it */
                path[pos] = -1;
            }
        }

        /* If no vertex can be added to Hamiltonian Cycle
           constructed so far, then return false */
        return false;
    }

    /* This function solves the Hamiltonian Cycle problem using
       Backtracking. It mainly uses hamCycleUtil() to solve the
       problem. It returns false if there is no Hamiltonian Cycle
       possible, otherwise return true and prints the path.
       Please note that there may be more than one solutions,
       this function prints one of the feasible solutions. */
    int hamCycle(int graph[][])
    {
        path = new int[V];
        for (int i = 0; i < V; i++)
            path[i] = -1;

        /* Let us put vertex 0 as the first vertex in the path.
           If there is a Hamiltonian Cycle, then the path can be
           started from any point of the cycle as the graph is
           undirected */
        path[0] = 0;
        if (hamCycleUtil(graph, path, 1) == false)
        {
            System.out.println("\nSolution does not exist");
            return 0;
        }

        printSolution(path);
        return 1;
    }
     /* A utility function to print solution */
    void printSolution(int path[])
    {
        System.out.println("Solution Exists: Following" +
                           " is one Hamiltonian Cycle");
        for (int i = 0; i < V; i++)

System.out.print(" " + path[i] + " ");

        // Let us print the first vertex again to show the
        // complete cycle
        System.out.println(" " + path[0] + " ");
    }

    // driver program to test above function
    public static void main(String args[])
    {
        HamiltonianCycle hamiltonian =
                                new HamiltonianCycle();
        /* Let us create the following graph
           (0)--(1)--(2)
            |   / \   |
            |  /   \  |
            | /     \ |
           (3)-------(4)    */
        int graph1[][] = {{0, 1, 0, 1, 0},
            {1, 0, 1, 1, 1},
            {0, 1, 0, 0, 1},
            {1, 1, 0, 0, 1},
            {0, 1, 1, 1, 0},
        };

        // Print the solution
        hamiltonian.hamCycle(graph1);

        /* Let us create the following graph
           (0)--(1)--(2)
            |   / \   |
            |  /   \  |
            | /     \ |
           (3)       (4)    */
        int graph2[][] = {{0, 1, 0, 1, 0},
            {1, 0, 1, 1, 1},
            {0, 1, 0, 0, 1},
            {1, 1, 0, 0, 0},
            {0, 1, 1, 0, 0},
        };
         // Print the solution
        hamiltonian.hamCycle(graph2);
    }
 }
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0

Solution does not exist
```

Exp. No:5(WEEK-5)	Implement Job sequencing with deadlines problem using Greedy Method.

Algorithm:

```
Step 1: Start
Step 2: Read the no. of jobs, profits, weights from the keyboard
Step 3: Find out sequence of jobs according to its deadline
Step 4: Print maximum profit
Step 5: Stop
```

Program:

```
import java.util.*;

class Job {

   // Each job has a unique-id,profit and deadline
   char id;
   int deadline, profit;

   // Constructors
   public Job() {}

   public Job(char id, int deadline, int profit)
   {
      this.id = id;
      this.deadline = deadline;
      this.profit = profit;
   }

   // Function to schedule the jobs take 2 arguments
   // arraylist and no of jobs to schedule
   void printJobScheduling(ArrayList<Job> arr, int t)
   {
      // Length of array
      int n = arr.size();

      // Sort all jobs according to decreasing order of
      // profit
      Collections.sort(arr,
                   (a, b) -> b.profit - a.profit);

      // To keep track of free time slots
      boolean result[] = new boolean[t];

      // To store result (Sequence of jobs)
      char job[] = new char[t];

      // Iterate through all given jobs
```

```
        for (int i = 0; i < n; i++) {
            // Find a free slot for this job (Note that we
            // start from the last possible slot)

    for (int j
                = Math.min(t - 1, arr.get(i).deadline - 1);
                j >= 0; j--) {
                // Free slot found
                if (result[j] == false) {
                    result[j] = true;
                    job[j] = arr.get(i).id;
                    break;
                }
            }
        }

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

    // Driver's code
    public static void main(String args[])
    {
        ArrayList<Job> arr = new ArrayList<Job>();
        arr.add(new Job('a', 2, 100));
        arr.add(new Job('b', 1, 19));
        arr.add(new Job('c', 2, 27));
        arr.add(new Job('d', 1, 25));
        arr.add(new Job('e', 3, 15));

        System.out.println(
            "Following is maximum profit sequence of jobs");

        Job job = new Job();

        // Function call
        job.printJobScheduling(arr, 3);
    }
}
```

Output:

```
Following is maximum profit sequence of jobs

c a e
```

Exp. No:6(WEEK-6)	Implement Knapsack problem using greedy method.

Algorithm:

Step 1: Start
Step 2: Read the profit and weight values from keyboard
Step 3: Calculate max p_i/w_i values
Step 4: Print the maximum profit
Step 5: Stop

Program:

```
import java.util.Scanner;
class FKnapsack
{
public static void main(String[] args)
{
Scanner sc=new Scanner(System.in);
int object,m;
System.out.println("Enter the Total Objects");
object=sc.nextInt();
int weight[]=new int[object];
int profit[]=new int[object];
for(int i=0;i<object;i++)
{
System.out.println("Enter the Profit");
profit[i]=sc.nextInt();
System.out.println("Enter the weight");
weight[i]=sc.nextInt();
}
System.out.println("Enter the Knapsack capacity");
m=sc.nextInt();
double p_w[]=new double[object];
for(int i=0;i<object;i++)
{
p_w[i]=(double)profit[i]/(double)weight[i];
}
System.out.println("");
System.out.println(" ------------------");
System.out.println("-----Data-Set------ ");
System.out.print("--------------------");
System.out.println("");
System.out.print("Objects");
for(int i=1;i<=object;i++)
{
System.out.print(i+" ");
}
System.out.println();
System.out.print("Profit ");
```

```
for(int i=0;i<object;i++)
{
System.out.print(profit[i]+" ");

}
System.out.println();
System.out.print("Weight ");
for(int i=0;i<object;i++)
{
System.out.print(weight[i]+"  ");
}
System.out.println();
System.out.print("P/W ");
for(int i=0;i<object;i++)
{
System.out.print(p_w[i]+" ");
}
for(int i=0;i<object-1;i++)
{
for(int j=i+1;j<object;j++)
{
if(p_w[i]<p_w[j])
{
double temp=p_w[j];
p_w[j]=p_w[i];
p_w[i]=temp;
int temp1=profit[j];
profit[j]=profit[i];
profit[i]=temp1;
int temp2=weight[j];
weight[j]=weight[i];
weight[i]=temp2;
}
}
}
System.out.println("");
System.out.println(" ------------------");
System.out.println("--After Arranging--");
System.out.print("-------------------");
System.out.println("");
System.out.print("Objects");
for(int i=1;i<=object;i++)
{
System.out.print(i+" ");
}
System.out.println();
System.out.print("Profit ");
for(int i=0;i<object;i++)
{
System.out.print(profit[i]+"  ");
}
```

```
System.out.println();
System.out.print("Weight ");
for(int i=0;i<object;i++)
{
System.out.print(weight[i]+"  ");
}
System.out.println();

System.out.print("P/W ");
for(int i=0;i<object;i++)
{
System.out.print(p_w[i]+" ");
}
int k=0;
double sum=0;
System.out.println();
while(m>0)
{
if(weight[k]<m)
{
sum+=1*profit[k];
m=m-weight[k];
}
else
{
double x4=m*profit[k];
double x5=weight[k];
double x6=x4/x5;
sum=sum+x6;
m=0;
}
k++;
}
System.out.println("Final Profit is="+sum);
}
}
```

Output:

```
Enter the Total Objects
7
Enter the Profit
10
Enter the weight
2
Enter the Profit
5
Enter the weight
3
Enter the Profit
15
```

```
Enter the weight
5
Enter the Profit
7
Enter the weight
7
Enter the Profit
6
Enter the weight
1


Enter the Profit
18
Enter the weight
4
Enter the Profit
3
Enter the weight
1
Enter the Knapsack capacity
15

----------------------
------Data-Set-------
----------------------
Objects1 2 3 4 5 6 7
Profit 10 5 15 7 6 18 3
Weight 2 3 5 7 1 4 1
P/W 5.0 1.6666666666666667 3.0 1.0 6.0 4.5 3.0
----------------------
--After Arranging--
----------------------
Objects1 2 3 4 5 6 7
Profit 6 10 18 15 3 5 7
Weight 1 2 4 5 1 3 7
P/W 6.0 5.0 4.5 3.0 3.0 1.6666666666666667 1.0
Final Profit is=55.333333333333336
```

Exp. No:7(WEEK-7)	Implement Prim’s minimum cost spanning tree problem using Greedy Method.

Algorithm:

```
Step 1: Start
Step 2: Read the no. of vertices, edges along with cost from the keyboard
Step 3: Find out minimum cost spanning tree using prims algorithm
Step 4: Print minimum cost
Step 5: Stop
```

Program:

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

class MST {

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

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

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

        return min_index;
    }

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

  {
        // Array to store constructed MST
        int parent[] = new int[V];

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

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

        // Initialize all keys as INFINITE
        for (int i = 0; i < V; i++) {
            key[i] = Integer.MAX_VALUE;
            mstSet[i] = false;
        }

        // Always include first 1st vertex in MST.
        // Make key 0 so that this vertex is
        // picked as first vertex
        key[0] = 0;

        // First node is always root of MST
        parent[0] = -1;

        // The MST will have V vertices
        for (int count = 0; count < V - 1; count++) {

            // Pick the minimum key vertex from the set of
            // vertices not yet included in MST
            int u = minKey(key, mstSet);

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

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

                // graph[u][v] is non zero only for adjacent
                // vertices of m mstSet[v] is false for
                // vertices not yet included in MST Update
                // the key only if graph[u][v] is smaller
                // than key[v]
                if (graph[u][v] != 0 && mstSet[v] == false
                    && graph[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = graph[u][v];
                }
            }



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

    public static void main(String[] args)
    {
        MST t = new MST();
        int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
                                      { 2, 0, 3, 8, 5 },
                                      { 0, 3, 0, 0, 7 },
                                      { 6, 8, 0, 0, 9 },
                                      { 0, 5, 7, 9, 0 } };

        // Print the solution
        t.primMST(graph);
    }
}
```

Output:

```
Edge    Weight

0 - 1    2

1 - 2    3

0 - 3    6

1 - 4    5
```

Exp. No:8(WEEK-8)	Implement Kruskal’s minimum cost spanning tree problem using Greedy Method.

Algorithm:

Step 1: Start
Step 2: Read the no. of vertices, edges along with cost from the keyboard
Step 3: Find out minimum cost spanning tree using Kruskal’s algorithm
Step 4: Print minimum cost
Step 5: Stop

Program:

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class KruskalsMST {

   // Defines edge structure
   static class Edge {
      int src, dest, weight;

      public Edge(int src, int dest, int weight)
      {
         this.src = src;
         this.dest = dest;
         this.weight = weight;
      }
   }

   // Defines subset element structure
   static class Subset {
      int parent, rank;

      public Subset(int parent, int rank)
      {
         this.parent = parent;
         this.rank = rank;
      }
   }

   // Starting point of program execution
   public static void main(String[] args)
   {
      int V = 4;
      List<Edge> graphEdges = new ArrayList<Edge>(
         List.of(new Edge(0, 1, 10), new Edge(0, 2, 6),
               new Edge(0, 3, 5), new Edge(1, 3, 15),
            new Edge(2, 3, 4)));

        // Sort the edges in non-decreasing order
        // (increasing with repetition allowed)
        graphEdges.sort(new Comparator<Edge>() {
            @Override public int compare(Edge o1, Edge o2)
            {
                return o1.weight - o2.weight;
            }
        });

        kruskals(V, graphEdges);
    }

    // Function to find the MST
    private static void kruskals(int V, List<Edge> edges)
    {
        int j = 0;
        int noOfEdges = 0;

        // Allocate memory for creating V subsets
        Subset subsets[] = new Subset[V];

        // Allocate memory for results
        Edge results[] = new Edge[V];

        // Create V subsets with single elements
        for (int i = 0; i < V; i++) {
            subsets[i] = new Subset(i, 0);
        }

        // Number of edges to be taken is equal to V-1
        while (noOfEdges < V - 1) {

            // Pick the smallest edge. And increment
            // the index for next iteration
            Edge nextEdge = edges.get(j);
            int x = findRoot(subsets, nextEdge.src);
            int y = findRoot(subsets, nextEdge.dest);

            // If including this edge doesn't cause cycle,
            // include it in result and increment the index
            // of result for next edge
            if (x != y) {
                results[noOfEdges] = nextEdge;
                union(subsets, x, y);
                noOfEdges++;
            }

            j++;
        }
```

```
          // Print the contents of result[] to display the
        // built MST
        System.out.println(
            "Following are the edges of the constructed MST:");

    int minCost = 0;
        for (int i = 0; i < noOfEdges; i++) {
            System.out.println(results[i].src + " -- "
                               + results[i].dest + " == "
                               + results[i].weight);
            minCost += results[i].weight;
        }
        System.out.println("Total cost of MST: " + minCost);
    }

    // Function to unite two disjoint sets
    private static void union(Subset[] subsets, int x,
                              int y)
    {
        int rootX = findRoot(subsets, x);
        int rootY = findRoot(subsets, y);

        if (subsets[rootY].rank < subsets[rootX].rank) {
            subsets[rootY].parent = rootX;
        }
        else if (subsets[rootX].rank
                 < subsets[rootY].rank) {
            subsets[rootX].parent = rootY;
        }
        else {
            subsets[rootY].parent = rootX;
            subsets[rootX].rank++;
        }
    }

    // Function to find parent of a set
    private static int findRoot(Subset[] subsets, int i)
    {
        if (subsets[i].parent == i)
            return subsets[i].parent;

        subsets[i].parent
            = findRoot(subsets, subsets[i].parent);
        return subsets[i].parent;
    }
}
```

Output:

```
Following are the edges in the constructed MST
Edge    Weight
2 - 3      4
0 - 3      5
0 - 1     10
Total cost of MST: 19
```

Exp. No:9(WEEK-9)	Implement Single source shortest path problem using Greedy Method.

Algorithm:

```
Step 1: Start
Step 2: Read the no. of vertices, edges along with cost from the keyboard
Step 3: Find out shortest distance from source to destination
Step 4: Print minimum cost
Step 5: Stop
```

Program:

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

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

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

        return min_index;
    }

    // A utility function to print the constructed distance
    // array
    void printSolution(int dist[])
    {
        System.out.println(
            "Vertex \t\t Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i + " \t\t " + dist[i]);
    }

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

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

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

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

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

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

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

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

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

    // Driver's code
    public static void main(String[] args)
    {
        /* Let us create the example graph discussed above
         */
        int graph[][]
           = new int[][] { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                          { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                          { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                          { 0, 0, 7, 0, 9, 14, 0, 0, 0 },

                          { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                          { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                          { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                          { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                          { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };
        ShortestPath t = new ShortestPath();

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

Output:

```
Vertex      Distance from Source

0             0

1             4

2             12

3             19

4             21

5             11

6             9

7             8

8             14
```

Exp. No:10(WEEK-10)	Implement 0/1 Knapsack problem using Dynamic Programming.

Algorithm:

Step 1: Start
Step 2: Read the no. of items along with their profit and weight the keyboard
Step 3: Find out the total profit
Step 4: Print maximum profit
Step 5: Stop

Program:

```
import java.util.Scanner;
public class Knapsack6a
{
static final int MAX = 20; // max. no. of objects
static int w[]; // weights 0 to n-1
static int p[]; // profits 0 to n-1
static int n;
// no. of objects
static int M;
// capacity of Knapsack
static int V[][];
// DP solution process -table
static int Keep[][]; // to get objects in optimal solution
public static void main(String args[])
{
w = new int[MAX];
p = new int[MAX];
V = new int [MAX][MAX];
Keep = new int[MAX][MAX];
int optsoln;
ReadObjects();
for (int i = 0; i <= M; i++)
V[0][i] = 0;
for (int i = 0; i <= n; i++)
V[i][0] = 0;
optsoln = Knapsack();
System.out.println("Optimal solution = " + optsoln);
}
static int Knapsack()
{
int r; // remaining Knapsack capacity
for (int i = 1; i <= n; i++)
for (int j = 0; j <= M; j++)
if ((w[i] <= j) && (p[i] + V[i -1][j -w[i]] > V[i -1][j]))
{
V[i][j] = p[i] + V[i -1][j -w[i]];
Keep[i][j] = 1;
}
else
{

V[i][j] = V[i -1][j];
Keep[i][j] = 0;
}
// Find the objects included in the Knapsack
r = M;
System.out.println("Items = ");
for (int i = n; i > 0; i--) // start from Keep[n,M]
if (Keep[i][r] == 1)
{
System.out.println(i + " ");
r = r -w[i];
}
System.out.println();
return V[n][M];
}
static void ReadObjects()
{
Scanner scanner = new Scanner(System.in);
System.out.println("Knapsack Problem -Dynamic Programming Solution: ");
System.out.println("Enter the max capacity of knapsack: ");
M = scanner.nextInt();
System.out.println("Enter number of objects: ");
n = scanner.nextInt();
System.out.println("Enter Weights: ");
for (int i = 1; i <= n; i++)
w[i] = scanner.nextInt();
System.out.println("Enter Profits: ");
for (int i = 1; i <= n; i++)
p[i] = scanner.nextInt();
scanner.close();
}
}
```

Output:

```
Knapsack Problem -Dynamic Programming Solution:
Enter the max capacity of knapsack:
15

Enter number of objects:
7

Enter Weights:
2 3 5 7 1 4 1

Enter Profits:
10 5 15 7 6 18 3

Items =
6
5
3
2
1

Optimal solution = 54
```

Exp. No:11(WEEK-11)	Implement All-Pairs Shortest Paths problem using Floyd's algorithm.

Algorithm:

Step 1: Start
Step 2: Read the no. of vertices, edges along with cost from the keyboard
Step 3: Find out minimum cost among all existing vertices
Step 4: Print minimum cost among all existing vertices
Step 5: Stop

Program:

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

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

    void floydWarshall(int dist[][])
    {

        int i, j, k;

        /* 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's code
    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();

        // Function call
        a.floydWarshall(graph);
    }
}
```

Output:

The 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
```

Exp. No:12(WEEK-12)	Implement Travelling sales person problem using Dynamic Programming.

Algorithm:

Step 1: Start
Step 2: Read the no. of city's along with cost from the keyboard
Step 3: Visit all city’s exactly once and return back to initial city
Step 4: Print total tour cost
Step 5: Stop

Program:

```
import java.util.Scanner;
public class Tsp
{
public static void main(String[] args)
{
int c[][]=new int[10][10], tour[]=new int[10];
Scanner in = new Scanner(System.in);
int i, j,cost;
System.out.println("**** TSP DYNAMIC PROGRAMMING *******");
System.out.println("Enter the number of cities: ");
int n = in.nextInt();
if(n==1)
{
System.out.println("Path is not possible");
System.exit(0);
}
System.out.println("Enter the cost matrix");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
c[i][j] = in.nextInt();
System.out.println("The entered cost matrix is");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
System.out.print(c[i][j]+"\t");
}
System.out.println();
}
for(i=1;i<=n;i++)
tour[i]=i;
cost = tspdp(c, tour, 1, n);
System.out.println("The accurate path is");
for(i=1;i<=n;i++)
System.out.print(tour[i]+"->");
System.out.println("1");
System.out.print("The accurate mincost is "+cost);
System.out.println("******* *****************************");
}

static int tspdp(int c[][], int tour[], int start, int n)
{
int mintour[]=new int[10], temp[]=new int[10], mincost=999,
ccost, i, j, k;
if(start == n-1)
{
return (c[tour[n-1]][tour[n]] + c[tour[n]][1]);
}
for(i=start+1; i<=n; i++)
{
for(j=1; j<=n; j++)
temp[j] = tour[j];
temp[start+1] = tour[i];
temp[i] = tour[start+1];
if((c[tour[start]][tour[i]]+(ccost=tspdp(c,temp,start+1,n)))<mincost)
{
mincost = c[tour[start]][tour[i]] + ccost;
for(k=1; k<=n; k++)
mintour[k] = temp[k];
}
}
for(i=1; i<=n; i++)
tour[i] = mintour[i];
return mincost;
}
}
```

Output:

```
**** TSP DYNAMIC PROGRAMMING *******
Enter the number of cities:
4
Enter the cost matrix
0 1 3 6
1 0 2 3
3 2 0 1
6 3 1 0
The entered cost matrix is
0	1	3	6
1	0	2	3
3	2	0	1
6	3	1	0
The accurate path is
1->2->4->3->1
The accurate mincost is 8******* *****************************
```