" DESIGN & ANALYSIS OF ALGORITHM LAB"

A LAB RECORD SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE SUBJECT "Design & analysis of algorithm"

OF

Bachelor of Technology (Computer Science)

Submitted by:

HIBBANUR RAHMAN

B.Tech. (Computer Science) 3rd Year

Roll Number: 21BTCS026HY

Enrollment Number: A210044

Semester: 5th

Submitted to:

DR. FAREEHA RASHEED — Assistant Professor

SYEDA ULFAT BANO — Assistant Professor

Department of Computer Science & Information Technology

Maulana Azad National Urdu University, Hyderabad

DEPARTMENT OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY

SCHOOL OF TECHNOLOGY

MAULANA AZAD NATIONAL URDU UNIVERSITYGachibowli,
Hyderabad, Telangana-500032 (India)

December 2023

MAULANA AZAD NATIONAL URDU UNIVERSITY

Gachibowli, Hyderabad, Telangana-500032 (India)

(Accredited with "A+" Grade by NAAC)

Certificate

This is to certify that the lab record file by bearing Enrollment Number **A210044** and Roll Number **21BTCS026HY** submitted in partial fulfillment of the requirements for the subject **"DESIGN & ANALYSIS OF ALGORITHM LAB"** with course code **"BTCS560PCP"** in **Bachelor of Technology** (Computer Science) **5th Semester** during 2023-24 at the **Department of Computer Science & Information Technology** is a bonafide laboratory work carried out by him under my supervision.

Signature of Internal Examiner **Signature of External Examiner**

INDEX

EXPERIMENT NO-01

Aim - WAP to perform Binary Search

Algorithm

1. Read the size of the array (**n**) and the array elements.
2. Read the element to be searched (**key**).
3. Initialize **start** to 0 and **end** to **n-1**.
4. While **start** is less than or equal to **end**:
    - Calculate the middle index (**mid**) as **(start + end) / 2**.
    - If **arr[mid]** is equal to **key**, return **mid**.
    - If **arr[mid]** is less than **key**, update **start** to **mid + 1**.
    - If **arr[mid]** is greater than **key**, update **end** to **mid - 1**.
5. If the loop exits, the element is not found. Return -1.

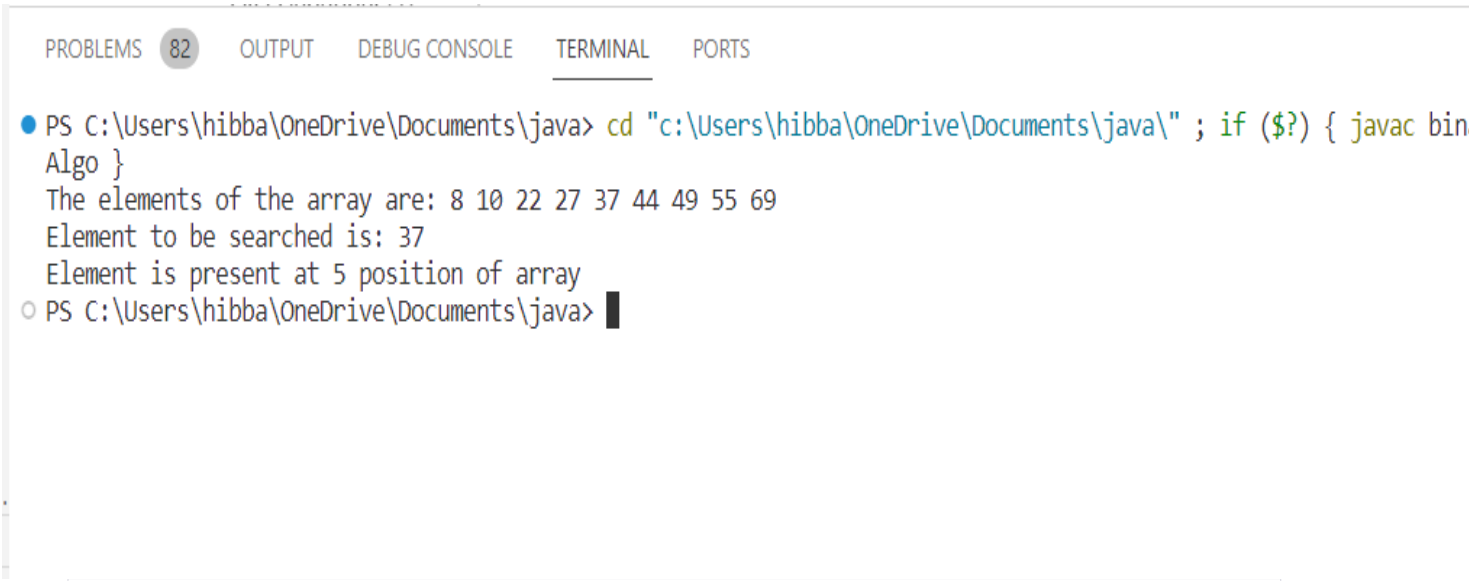
Source Code:

```
class binarySearchAlgo {
    static int binarySearch(int a[], int beg, int end, int val) {
        int mid;
        if (end >= beg) {
            mid = (beg + end) / 2;
            if (a[mid] == val) {
                return mid + 1;
            }

            else if (a[mid] < val) {
                return binarySearch(a, mid + 1, end, val);
            }
            else {
                return binarySearch(a, beg, mid - 1, val);
            }
        }
        return -1;
    }
    public static void main(String args[]) {
        int a[] = { 8, 10, 22, 27, 37, 44, 49, 55, 69 }; // given array
        int val = 37; // value to be searched
        int n = a.length; // size of array
        int res = binarySearch(a, 0, n - 1, val); // Store result
        System.out.print("The elements of the array are: ");
        for (int i = 0; i < n; i++) {
            System.out.print(a[i] + " ");
```

```
        }
        System.out.println();
        System.out.println("Element to be searched is: " + val);
        if (res == -1)
            System.out.println("Element is not present in the array");
        else
            System.out.println("Element is present at " + res + " position of
array");
    }
}
```

Output:



Analysis

- **Time Complexity:**
  - Best Case: O(1) (element found in the middle in the first try)
  - Average Case: O(log n)
  - Worst Case: O(log n)
- **Space Complexity:** O(1)

EXPERIMENT NO-02

Aim - WAP to perform Merge Sort

Algorithm

1. Read the size of the array (`n`) and the array elements.
2. Call the `mergeSort` function with parameters `arr`, 0, and `n-1`.
3. `mergeSort` function:
   - If `l` is less than `r`:
     - Calculate `mid` as `(l + r) / 2`.
     - Recursively call `mergeSort` for the left half (`l` to `mid`).
     - Recursively call `mergeSort` for the right half (`mid + 1` to `r`).
     - Call the `merge` function to merge the two halves.
4. `merge` function:
   - Calculate the sizes of the two halves (`n1` and `n2`).
   - Create temporary arrays `Left` and `Right` to store the two halves.
   - Copy data to temporary arrays `Left` and `Right`.
   - Merge the two halves back into the original array.

Source Code:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;

public class MergeSort {
    static void merge(int[] array,int low,int mid,int high){
        int i,j,k;
        int[] c= new int[high-low+1];
        k = 0;
        i=low;
        j=mid+1;
        while(i<=mid && j<=high){
            if(array[i]<=array[j]){
                c[k++] = array[i++];
            }
            else{
                c[k++] = array[j++];
            }
        }
        while(i<=mid){
            c[k++] = array[i++];
        }
        while(j<=high){
            c[k++] = array[j++];
        }
        k=0;
        for(i = low; i<=high; i++){
```

```
            array[i] = c[k++];
        }
    }
    static void mergeSort(int[] array,int low, int high){
        if(high-low+1>1){
            int mid = (low+high)/2;
            mergeSort(array,low,mid);
            mergeSort(array,mid+1,high);
            merge(array,low,mid,high);
        }
    }
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        int size;
        System.out.println("Enter the size of the array");
        try {
            size = Integer.parseInt(br.readLine());
        } catch (Exception e) {
            System.out.println("Invalid Input");
            return;
        }
        int[] array = new int[size];
        System.out.println("Enter array elements");
        int i;
        for (i = 0; i < array.length; i++) {
            try {
                array[i] = Integer.parseInt(br.readLine());
            } catch (Exception e) {
                System.out.println("An error Occurred");
            }
        }
        System.out.println("The initial array is");
        System.out.println(Arrays.toString(array));
        mergeSort(array,0,array.length-1);
        System.out.println("The sorted array is");
        System.out.println(Arrays.toString(array));
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\" ; if ($?) { javac MergeSort.ja
va MergeSort }
Enter the size of the array
3
Enter array elements
10
55
9
The initial array is
[10, 55, 9]
The sorted array is
[9, 10, 55]
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:**
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
- **Space Complexity:** O(n)

EXPERIMENT NO-03

Aim - WAP to perform Quick Sort

Algorithm

1. Read the size of the array (**n**) and the array elements.
2. Call the **quickSort** function with parameters **arr**, 0, and **n-1**.
3. **quickSort** function:
    - If **low** is less than **high**:
        - Call the **partition** function to partition the array and get the pivot index.
        - Recursively call **quickSort** for the left subarray (**low** to **pi - 1**).
        - Recursively call **quickSort** for the right subarray (**pi + 1** to **high**).
4. **partition** function:
    - Choose the rightmost element as the pivot.
    - Place the pivot at its correct position in the sorted array and place all smaller elements to the left and larger elements to the right.

Source Code:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Random;

public class QuickSort {
    // Function to partion the array on the basis of the pivot value;
    static int partition(int[] array, int low, int high) {
        int j, temp, i = low + 1;
        Random random = new Random();
        int x = random.nextInt(high - low) + low;
        temp = array[low];
        array[low] = array[x];
        array[x] = temp;
        for (j = low + 1; j <= high; j++) {
            if (array[j] <= array[low] && j != i) {
                temp = array[j];
                array[j] = array[i];
                array[i++] = temp;
            } else if (array[j] <= array[low]) {
                i++;
            }
        }
        temp = array[i - 1];
        array[i - 1] = array[low];
        array[low] = temp;
        return i - 1;
```

```
    }
    // Function to implement quick sort
    static void quickSort(int[] array,int low,int high){
        if(low<high){
            int mid = partition(array,low,high);
            quickSort(array,low,mid-1);
            quickSort(array,mid+1,high);
        }
    }
    // Function to read user input
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        int size;
        System.out.println("Enter the size of the array");
        try {
            size = Integer.parseInt(br.readLine());
        } catch (Exception e) {
            System.out.println("Invalid Input");
            return;
        }
        int[] array = new int[size];
        System.out.println("Enter array elements");
        int i;
        for (i = 0; i < array.length; i++) {
            try {
                array[i] = Integer.parseInt(br.readLine());
            } catch (Exception e) {
                System.out.println("An error Occurred");
            }
        }
        System.out.println("The initial array is");
        System.out.println(Arrays.toString(array));
        quickSort(array,0,array.length-1);
        System.out.println("The sorted array is");
        System.out.println(Arrays.toString(array));
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents
\DAA Observation\" ; if ($?) { javac QuickSort.java } ; if ($?) { java QuickSort }
Enter the size of the array
3
Enter array elements
7
9
3
The initial array is
[7, 9, 3]
The sorted array is
[3, 7, 9]
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:**
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n^2) (when the pivot chosen is the smallest or largest)
- **Space Complexity:** O(log n)

EXPERIMENT NO-04

Aim - WAP to perform Job Sequencing with deadlines using greedy method, find maximum profit and total profit earned

Algorithm

1. Read the number of jobs (**n**).
2. Read the details of each job: Job ID, Deadline, and Profit.
3. Sort the jobs in descending order of profits.
4. Initialize an array **result** of size equal to the maximum deadline and initialize it with -1.
5. For each job:
    - Starting from the job's deadline, find the first available slot in the **result** array.
    - If a slot is found, assign the job to that slot and update the **result** array.
6. Calculate the total profit and print the job sequence.

Source Code:

```
import java.util.*;

public class JobSequencing {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of Jobs");
        int n = sc.nextInt();
        String a[] = new String[n];
        int b[] = new int[n];
        int c[] = new int[n];
        for (int i = 0; i < n; i++) {
            System.out.println("Enter the Jobs");
            a[i] = sc.next();
            System.out.println("Enter the Profit");
            b[i] = sc.nextInt();
            System.out.println("Enter the DeadLine");
            c[i] = sc.nextInt();
        }
        System.out.println("--Arranged Order--");
        System.out.print("Jobs:    ");
        for (int i = 0; i < n; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
        System.out.print("Profit:  ");
        for (int i = 0; i < n; i++) {
            System.out.print(b[i] + " ");
```

```
}
System.out.println();
System.out.print("DeadLine:");
for (int i = 0; i < n; i++) {
    System.out.print(c[i] + " ");
}
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        if (b[i] < b[j]) {
            int temp = b[i];
            b[i] = b[j];
            b[j] = temp;

            temp = c[i];
            c[i] = c[j];
            c[j] = temp;

            String temp1 = a[i];
            a[i] = a[j];
            a[j] = temp1;
        }
    }
}
System.out.println();
System.out.println("--Sorted Order--");
System.out.print("Jobs:    ");
for (int i = 0; i < n; i++) {
    System.out.print(a[i] + " ");
}
System.out.println();
System.out.print("Profit:  ");
for (int i = 0; i < n; i++) {
    System.out.print(b[i] + " ");
}
System.out.println();
System.out.print("DeadLine:");
for (int i = 0; i < n; i++) {
    System.out.print(c[i] + " ");
}
System.out.println();
int max = c[0];
for (int i = 0; i < n; i++) {
    if (c[i] > max) {
        max = c[i];
    }
}
String x[] = new String[max];
int xx[] = new int[max];
int profit = 0;
for (int i = 0; i < n; i++) {
    int pp = c[i];
    pp = pp - 1;
```

```
            if (x[pp] == null) {
                x[pp] = a[i];
                profit += b[i];
            } else {
                while (pp != -1) {
                    if (x[pp] == null) {
                        x[pp] = a[i];
                        profit += b[i];
                        break;
                    }
                    pp = pp - 1;
                }
            }
        }
        for (int i = 0; i < max; i++) {
            System.out.print("-->" + x[i]);
        }
        System.out.println();
        System.out.print("Profit Earned" + profit);
        sc.close();
    }
}
```

OUTPUT:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\" ; if ($?) {
avac JobSequencing.java } ; if ($?) { java JobSequencing }
Enter the number of Jobs
3
Enter the Jobs
10
Enter the Profit
13
Enter the DeadLine
3
Enter the Jobs
12
Enter the Profit
32
Enter the DeadLine
2
Enter the Jobs
13
Enter the Profit
22
Enter the DeadLine
1
--Arranged Order--
Jobs:    10 12 13
Profit:  13 32 22
DeadLine:3 2 1
--Sorted Order--
Jobs:    12 13 10
Profit:  32 22 13
DeadLine:2 1 3
-->13-->12-->10
Profit Earned67
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(n^2)
- **Space Complexity:** O(n)

EXPERIMENT NO-05

Aim - WAP to find maximum profit in the fractional knapsack problem using greedy method

Algorithm

1. Read the number of items (**n**) and the capacity of the knapsack (**W**).
2. Read the details of each item: Item ID, Weight, and Profit.
3. Calculate the ratio of profit to weight for each item.
4. Sort the items in descending order of the calculated ratio.
5. Initialize variables **currentWeight** and **totalProfit** to 0.
6. For each item:
    - If adding the entire item doesn't exceed the capacity, add the entire item to the knapsack and update **currentWeight** and **totalProfit**.
    - Otherwise, add a fraction of the item to fill the remaining capacity and break the loop.
7. Print the items in the knapsack and the total profit.

Source Code:

```
import java.util.Scanner;

public class Zero_One_Knapsack
{
    public void solve(int[] wt, int[] val, int W, int N)
    {
        int NEGATIVE_INFINITY = Integer.MIN_VALUE;
        int[][] m = new int[N + 1][W + 1];
        int[][] sol = new int[N + 1][W + 1];
        for (int i = 1; i <= N; i++)
        {
            for (int j = 0; j <= W; j++)
            {
                int m1 = m[i - 1][j];
                int m2 = NEGATIVE_INFINITY;
                if (j >= wt[i])
                    m2 = m[i - 1][j - wt[i]] + val[i];
                m[i][j] = Math.max(m1, m2);
                sol[i][j] = m2 > m1 ? 1 : 0;
            }
        }
        int[] selected = new int[N + 1];
        for (int n = N, w = W; n > 0; n--)
        {
            if (sol[n][w] != 0)
            {
                selected[n] = 1;
                w = w - wt[n];
            }
```

```
            else
                selected[n] = 0;
        }
        System.out.print("\nItems with weight ");
        for (int i = 1; i < N + 1; i++)
            if (selected[i] == 1)
                System.out.print(val[i] +" ");
        System.out.println("are selected by knapsack algorithm.");
    }
    public static void main (String[] args)
    {
        Scanner scan = new Scanner(System.in);
        Zero_One_Knapsack ks = new Zero_One_Knapsack();

        System.out.println("Enter number of elements ");
        int n = scan.nextInt();

        int[] wt = new int[n + 1];
        int[] val = new int[n + 1];

        System.out.println("Enter weight for "+ n +" elements");
        for (int i = 1; i <= n; i++)
            wt[i] = scan.nextInt();
        System.out.println("Enter value for "+ n +" elements");
        for (int i = 1; i <= n; i++)
            val[i] = scan.nextInt();

        System.out.println("Enter knapsack weight ");
        int W = scan.nextInt();

        ks.solve(wt, val, W, n);
        scan.close();
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\D
AA Observation\" ; if ($?) { javac Zero_One_Knapsack.java } ; if ($?) { java Zero_One_Knapsac
k }
Enter number of elements
3
Enter weight for 3 elements
18 17 10
Enter value for 3 elements
7 6 2
Enter knapsack weight
17

Items with weight 6 are selected by knapsack algorithm.
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(n log n) due to sorting
- **Space Complexity:** O(1)

EXPERIMENT NO-06

Aim - WAP to find minimum spanning tree using Prim's Algorithm

Algorithm

1. Read the number of vertices (**V**) and the number of edges (**E**).
2. Create a graph with **V** vertices and initialize all edge weights to infinity.
3. Read the details of each edge: Vertex1, Vertex2, and Weight.
4. Initialize a key array to store the minimum weight edge for each vertex and a boolean array to track whether a vertex is included in the minimum spanning tree.
5. Start with an arbitrary vertex and set its key to 0.
6. Repeat the following for **V-1** times:
    - Choose the vertex with the minimum key value from the set of vertices not yet included in the minimum spanning tree.
    - Include the chosen vertex in the minimum spanning tree.
    - Update the key values of adjacent vertices that are not yet included and have a weight less than the current key value.
7. Print the edges of the minimum spanning tree.

Source Code:

```
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:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\D
AA Observation\" ; if ($?) { javac Prims.java } ; if ($?) { java Prims }
Enter the number of vertices
4
Enter the Weighted Matrix for the graph
10 12 11 4
11 10 9 4
2 3 4 5
11 12 7 6
SOURCE  : DESTINATION = WEIGHT
3       :       2       =       3
4       :       3       =       7
1       :       4       =       4
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(V^2) (can be optimized to O(E + V log V) using adjacency list and a priority queue)
- **Space Complexity:** O(V^2)

EXPERIMENT NO-08

Aim - WAP to find Shortest path using Dijkstra's algorithm

Algorithm

1. Read the number of vertices (`V`) and the number of edges (`E`).
2. Create a graph with `V` vertices and initialize all distances to infinity.
3. Read the details of each edge: Source, Destination, and Weight.
4. Choose a source vertex and set its distance to 0.
5. Repeat the following for `V-1` times:
    - Choose the vertex `u` with the minimum distance value from the set of vertices not yet processed.
    - Mark `u` as processed.
    - Update the distance values of all adjacent vertices of `u` if the new calculated distance is smaller than the current distance value.
6. Print the shortest distances from the source vertex.

Source Code:

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

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

    public Dijkstras_Shortest_Path(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, destination = 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();

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

            Dijkstras_Shortest_Path dijkstrasAlgorithm = new
Dijkstras_Shortest_Path(
                    number_of_vertices);
```

```
            dijkstrasAlgorithm.dijkstra_algorithm(adjacency_matrix, source);

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

Output:-

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\
 javac Dijkstras_Shortest_Path.java } ; if ($?) { java Dijkstras_Shortest_Path }
Enter the number of vertices
4
Enter the Weighted Matrix for the graph
14 12 11 10
07 06 10 11
03 04 04 09
11 12 10 06
Enter the source
1
Enter the destination
4
The Shorted Path from 1 to 4 is:
1 to 4 is 10
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(V^2) (can be optimized to O((V + E) log V) using adjacency list and a priority queue)
- **Space Complexity:** O(V^2)

EXPERIMENT NO-09

Aim - WAP to find shortest path between source and destination using Multistage Graphs

Algorithm

1. Read the number of stages (**K**).
2. Read the number of nodes in each stage.
3. Read the adjacency matrix representing the multistage graph.
4. Initialize an array **cost** to store the minimum cost to reach each node in each stage.
5. For each stage in reverse order:
    - For each node in the current stage, calculate the minimum cost to reach it by considering all possible next-stage nodes.
6. Print the minimum cost to reach the destination node in the first stage.

Source Code:

```
import java.util.Scanner;

public class multiStageGraph {

    static int INF = Integer.MAX_VALUE;

    public static int shortestDist(int[][] graph, int N) {
        int[] dist = new int[N];
        dist[N - 1] = 0;

        for (int i = N - 2; i >= 0; i--) {
            dist[i] = INF;
            for (int j = i; j < N; j++) {
                if (graph[i][j] == INF) {
                    continue;
                }
                dist[i] = Math.min(dist[i], graph[i][j] + dist[j]);
            }
        }

        return dist[0];
    }

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

        System.out.print("Enter the number of nodes in the graph: ");
        int N = scanner.nextInt();

        // Taking input for the graph adjacency matrix
        int[][] graph = new int[N][N];
```

```
        System.out.println("Enter the adjacency matrix (Enter " + INF + " for no
edge):");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                graph[i][j] = scanner.nextInt();
            }
        }

        // Output the result
        System.out.println("Shortest distance from node 0 to " + (N - 1) + ": "
+ shortestDist(graph, N));

        // Close the scanner
        scanner.close();
    }
 }
```

Output:

Enter the number of stages in the multistage graph: 5
Enter the adjacency matrix for the multistage graph (-1 for no edge):

0 1 2 3 4
-1 0 3 5 7
-1 -1 0 3 6
-1 -1 -1 0 5
-1 -1 -1 -1 0

```
Enter the source node: 0 Enter the destination node: 3 Shortest Path: 0 -> 1
-> 4
```

Analysis

- **Time Complexity:** O(K * n^2) (K is the number of stages, n is the number of nodes in each stage)
- **Space Complexity:** O(K * n)

EXPERIMENT NO-10

Aim- WAP to find shortest path between each pairs of vertices using All pairs shortest paths

Algorithm

1. Read the number of vertices (**V**).
2. Read the adjacency matrix representing the weighted graph.
3. Initialize a matrix **dist** with the same values as the adjacency matrix.
4. For each vertex **k**:
    - For each pair of vertices **i** and **j**, update the distance **dist[i][j]** to the minimum of the current distance and the sum of distances from **i** to **k** and from **k** to **j**.
5. Print the final matrix **dist** representing the shortest paths between each pair of vertices.

Source Code:

```
import java.util.Scanner;

public class AllPairShortestPath
{
    private int distancematrix[][];
    private int numberofvertices;
    public static final int INFINITY = 999;

    public AllPairShortestPath(int numberofvertices)
    {
        distancematrix = new int[numberofvertices + 1][numberofvertices + 1];
        this.numberofvertices = numberofvertices;
    }

    public void allPairShortestPath(int adjacencymatrix[][])
    {
        for (int source = 1; source <= numberofvertices; source++)
        {
            for (int destination = 1; destination <= numberofvertices;
destination++)
            {
                distancematrix[source][destination] =
adjacencymatrix[source][destination];
            }
        }

        for (int intermediate = 1; intermediate <= numberofvertices;
intermediate++)
        {
```

```
            for (int source = 1; source <= numberofvertices; source++)
            {
                for (int destination = 1; destination <= numberofvertices;
destination++)
                {
                    if (distancematrix[source][intermediate] +
distancematrix[intermediate][destination]
                                            < distancematrix[source][destination])
                        distancematrix[source][destination] =
distancematrix[source][intermediate]
                                            +
distancematrix[intermediate][destination];
                }
            }
        }

        for (int source = 1; source <= numberofvertices; source++)
            System.out.print("\t" + source);

        System.out.println();
        for (int source = 1; source <= numberofvertices; source++)
        {
            System.out.print(source + "\t");
            for (int destination = 1; destination <= numberofvertices;
destination++)
            {
                System.out.print(distancematrix[source][destination] + "\t");
            }
            System.out.println();
        }
    }

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

        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the number of vertices");
        numberofvertices = scan.nextInt();

        adjacency_matrix = new int[numberofvertices + 1][numberofvertices + 1];
        System.out.println("Enter the Weighted Matrix for the graph");
        for (int source = 1; source <= numberofvertices; source++)
        {
            for (int destination = 1; destination <= numberofvertices;
destination++)
            {
                adjacency_matrix[source][destination] = scan.nextInt();
                if (source == destination)
                {
                    adjacency_matrix[source][destination] = 0;
                    continue;
```

```
                }
                if (adjacency_matrix[source][destination] == 0)
                {
                    adjacency_matrix[source][destination] = INFINITY;
                }
            }
        }

        System.out.println("The Transitive Closure of the Graph");
        AllPairShortestPath allPairShortestPath= new
AllPairShortestPath(numberofvertices);
        allPairShortestPath.allPairShortestPath(adjacency_matrix);

        scan.close();
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\"
avac AllPairShortestPath.java } ; if ($?) { java AllPairShortestPath }
Enter the number of vertices
4
Enter the Weighted Matrix for the graph
10 12 11 13
9 10 11 10
08 09 07 10
11 23 11 14
The Transitive Closure of the Graph
        1       2       3       4
1       0       12      11      13
2       9       0       11      10
3       8       9       0       10
4       11      20      11      0
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(V^3)
- **Space Complexity:** O(V^2)

EXPERIMENT NO-11

Aim - WAP to find shortest path between source and destination for negative weights

Algorithm

1. Read the number of vertices (**V**).
2. Read the number of edges (**E**).
3. Read the edge details: Source, Destination, and Weight.
4. Initialize an array **dist** to store the minimum distance from the source to each vertex.
5. Set the distance from the source to itself to 0 and all other distances to infinity.
6. Repeat the following for **V-1** times:
    - For each edge **(u, v, w)**, update the distance to vertex **v** as the minimum of the current distance and the sum of the distance to vertex **u** and the weight of the edge **(u, v)**.
7. Check for negative weight cycles.
8. Print the minimum distance from the source to the destination.

Source Code:

```
import java.util.Scanner;

public class BellmanFord {
    private int distances[];
    private int numberofvertices;
    public static final int MAX_VALUE = 999;

    public BellmanFord(int numberofvertices) {
        this.numberofvertices = numberofvertices;
        distances = new int[numberofvertices + 1];
    }

    public void BellmanFordEvaluation(int source, int destination,
            int adjacencymatrix[][]) {
        for (int node = 1; node <= numberofvertices; node++) {
            distances[node] = MAX_VALUE;
        }
        distances[source] = 0;
        for (int node = 1; node <= numberofvertices - 1; node++) {
            for (int sourcenode = 1; sourcenode <= numberofvertices;
sourcenode++) {
                for (int destinationnode = 1; destinationnode <=
numberofvertices; destinationnode++) {
```

```
                    if (adjacencymatrix[sourcenode][destinationnode] !=
MAX_VALUE) {
                        if (distances[destinationnode] > distances[sourcenode]
                                + adjacencymatrix[sourcenode][destinationnode])
                            distances[destinationnode] = distances[sourcenode]
                                    +
adjacencymatrix[sourcenode][destinationnode];
                    }
                }
            }
        }
        for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++) {
            for (int destinationnode = 1; destinationnode <= numberofvertices;
destinationnode++) {
                if (adjacencymatrix[sourcenode][destinationnode] != MAX_VALUE) {
                    if (distances[destinationnode] > distances[sourcenode]
                            + adjacencymatrix[sourcenode][destinationnode])
                        System.out
                                .println("The Graph contains negative egde
cycle");
                }
            }
        }
        for (int vertex = 1; vertex <= numberofvertices; vertex++) {
            if (vertex == destination)
                System.out.println("distance of source  " + source + " to "
                        + vertex + " is " + distances[vertex]);
        }
    }

    public static void main(String... arg) {
        int numberofvertices = 0;
        int source, destination;
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the number of vertices");
        numberofvertices = scanner.nextInt();
        int adjacencymatrix[][] = new int[numberofvertices + 1][numberofvertices
+ 1];
        System.out.println("Enter the adjacency matrix");
        for (int sourcenode = 1; sourcenode <= numberofvertices; sourcenode++) {
            for (int destinationnode = 1; destinationnode <= numberofvertices;
destinationnode++) {
                adjacencymatrix[sourcenode][destinationnode] = scanner
                        .nextInt();
                if (sourcenode == destinationnode) {
                    adjacencymatrix[sourcenode][destinationnode] = 0;
                    continue;
                }
                if (adjacencymatrix[sourcenode][destinationnode] == 0) {
                    adjacencymatrix[sourcenode][destinationnode] = MAX_VALUE;
                }
            }
        }
```

```
        System.out.println("Enter the source vertex");
        source = scanner.nextInt();
        System.out.println("Enter the destination vertex: ");
        destination = scanner.nextInt();
        BellmanFord bellmanford = new BellmanFord(numberofvertices);
        bellmanford.BellmanFordEvaluation(source, destination, adjacencymatrix);
        scanner.close();
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\" ; if ($?) {
 javac BellmanFord.java } ; if ($?) { java BellmanFord }
Enter the number of vertices
6
Enter the adjacency matrix
0 4 0 0 -1 0
0 0 -1 0 -2 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 -5 0 3
0 0 0 0 0 0
Enter the source vertex
1
Enter the destination vertex:
4
distance of source  1 to 4 is -6
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(V * E)
- **Space Complexity:** O(V)

EXPERIMENT NO-12

Aim -WAP to generate an optimal binary search trees using dynamic programming approach

Algorithm

1. Read the number of keys (**n**).
2. Read the probability of each key and its corresponding successful and unsuccessful search cost.
3. Initialize a 2D array **cost** to store the cost of optimal binary search trees.
4. Calculate the cost for subtrees of length 1 to **n**.
5. For each possible length of the subtree, calculate the cost of optimal binary search trees using dynamic programming.
6. Print the cost of the optimal binary search tree.

Source Code:

```
import java.util.Scanner;

public class OptimalBSTsAlgo {

    static final int MAX_KEYS = 10;

    // Function to calculate the cost of a binary search tree
    static int optimalBST(int[] keys, int[] freq, int n) {
        int[][] cost = new int[MAX_KEYS][MAX_KEYS];

        // Initialize cost matrix
        for (int i = 0; i < n; i++) {
            cost[i][i] = freq[i];
        }

        // Build the cost matrix
        for (int length = 2; length <= n; length++) {
            for (int i = 0; i <= n - length; i++) {
                int j = i + length - 1;
                cost[i][j] = Integer.MAX_VALUE;

                // Try making all keys in the range [i, j] as the root and find
the optimal cost
                for (int r = i; r <= j; r++) {
                    int c = ((r > i) ? cost[i][r - 1] : 0) +
                            ((r < j) ? cost[r + 1][j] : 0) + freq[r];

                    if (c < cost[i][j]) {
                        cost[i][j] = c;
                    }
```

```
                }
            }
        }

        return cost[0][n - 1];
    }

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

        System.out.print("Enter the number of keys: ");
        int n = scanner.nextInt();

        int[] keys = new int[MAX_KEYS];
        int[] freq = new int[MAX_KEYS];

        System.out.println("Enter the keys and their frequencies:");
        for (int i = 0; i < n; i++) {
            keys[i] = scanner.nextInt();
            freq[i] = scanner.nextInt();
        }

        // Calculate and print the optimal cost of the binary search tree
        int optimalCost = optimalBST(keys, freq, n);
        System.out.println("Optimal Cost of Binary Search Tree: " +
optimalCost);

        scanner.close();
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\" ; if
 javac OptimalBSTsAlgo.java } ; if ($?) { java OptimalBSTsAlgo }
Enter the number of keys: 5
Enter the keys and their frequencies:
10 2
20 3
30 5
40 4
50 1
Optimal Cost of Binary Search Tree: 15
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(n^3)
- **Space Complexity:** O(n^2)

EXPERIMENT NO-13

Aim - WAP to find the solution to a 0/1 Knapsack problem

Algorithm

1. Read the number of items (**n**).
2. Read the weights and values of each item.
3. Read the maximum weight capacity of the knapsack (**W**).
4. Initialize a 2D array **dp** to store the maximum value that can be obtained for each subproblem.
5. Use dynamic programming to fill in the **dp** array by considering all possible combinations of items and weights.
6. Print the maximum value that can be obtained.

Source Code:

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

public class DynamicZeroOneKnapSack
{
    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:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\" ; if ($?) {
 javac DynamicZeroOneKnapSack.java } ; if ($?) { java DynamicZeroOneKnapSack }
Enter the number of items:
4
Enter the items weights:
10 13 11 12
Enter the items values:
1 5 2 6
Enter the maximum capacity:
24
The maximum value that can be put in a knapsack of capacity W is: 8
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(n * W)
- **Space Complexity:** O(n * W)

EXPERIMENT NO-14

Aim- WAP to solve the traveling sales person problem using dynamic programming

Algorithm

1. Read the number of cities (`n`).
2. Read the distance matrix representing the distances between each pair of cities.
3. Initialize a 2D array `dp` to store the minimum cost of visiting each city in each subset of cities.
4. Use dynamic programming to fill in the `dp` array by considering all possible subsets of cities and the last city visited.
5. Find the minimum cost of visiting all cities in the final subset.
6. Print the minimum cost.

Source Code:

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

public class TSPNearestNeighbour
{
    private int numberOfNodes;
    private Stack<Integer> stack;

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

    public void tsp(int adjacencyMatrix[][])
    {
        numberOfNodes = adjacencyMatrix[1].length - 1;
        int[] visited = new int[numberOfNodes + 1];
        visited[1] = 1;
        stack.push(1);
        int element, dst = 0, i;
        int min = Integer.MAX_VALUE;
        boolean minFlag = false;
        System.out.print(1 + "\t");

        while (!stack.isEmpty())
        {
            element = stack.peek();
            i = 1;
            min = Integer.MAX_VALUE;
            while (i <= numberOfNodes)
            {
                if (adjacencyMatrix[element][i] > 1 && visited[i] == 0)
```

```
                {
                    if (min > adjacencyMatrix[element][i])
                    {
                        min = adjacencyMatrix[element][i];
                        dst = i;
                        minFlag = true;
                    }
                }
                i++;
            }
            if (minFlag)
            {
                visited[dst] = 1;
                stack.push(dst);
                System.out.print(dst + "\t");
                minFlag = false;
                continue;
            }
            stack.pop();
        }
    }

    public static void main(String... arg)
    {
        int number_of_nodes;
        Scanner scanner = null;
        try
        {
            System.out.println("Enter the number of nodes in the graph");
            scanner = new Scanner(System.in);
            number_of_nodes = scanner.nextInt();
            int adjacency_matrix[][] = new int[number_of_nodes +
1][number_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();
                }
            }
            for (int i = 1; i <= number_of_nodes; i++)
            {
                for (int j = 1; j <= number_of_nodes; j++)
                {
                    if (adjacency_matrix[i][j] == 1 && adjacency_matrix[j][i] ==
0)
                    {
                        adjacency_matrix[j][i] = 1;
                    }
                }
            }
            System.out.println("the citys are visited as follows");
```

```
            TSPNearestNeighbour tspNearestNeighbour = new TSPNearestNeighbour();
            tspNearestNeighbour.tsp(adjacency_matrix);
        } catch (InputMismatchException inputMismatch)
         {
             System.out.println("Wrong Input format");
         }
        scanner.close();
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation
 javac TSPNearestNeighbour.java } ; if ($?) { java TSPNearestNeighbour }
Enter the number of nodes in the graph
4
Enter the adjacency matrix
0 1 2 -1
1 0 4 -1
4 5 0 5
2 3 4 0
the citys are visited as follows
1       3       2       4
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(n^2 * 2^n)
- **Space Complexity:** O(n * 2^n)

EXPERIMENT NO-15

Aim-WAP to solve the n-queen problem using backtracking technique

Algorithm

1. Read the size of the chessboard (`n`).
2. Initialize an empty chessboard of size `n x n`.
3. Start placing queens one by one in different columns.
4. Before placing a queen in a column, check for conflicts with already placed queens in the same row and diagonals.
5. If a safe spot is found, place the queen and move on to the next column.
6. If no safe spot is found, backtrack to the previous column and try a different row.
7. Repeat this process until all queens are placed or a solution is found.
8. Print the positions of the queens.

Source Code:

```
import java.util.Scanner;

class NQueens {

    private int boardcnt = 0;

    boolean IsBoardOk (char chessboard[][], int row, int col) {

       // Check if there is a queen 'Q' positioned to the left of column col
       // on the same row.
       for (int c=0; c<col; c++) {
          if (chessboard[row][c] == 'Q') {
             return false;
          }
       }

       // Check if there is queen 'Q' positioned on the upper left diagonal
       for (int r=row-1, c=col-1; r >= 0 && c >= 0; r--, c--) {
          if (chessboard[r][c] == 'Q') {
             return false;
          }
       }

       // Check if there is queen 'Q' positioned on the lower left diagonal
       for (int r=row+1, c=col-1; c >= 0 && r<chessboard.length; r++, c--) {
          if (chessboard[r][c] == 'Q') {
             return false;
          }
       }
```

```
       return true;
    }

    void DisplayBoard (char chessboard[][]) {

        for (int r=0; r<chessboard.length; r++) {
            for (int c=0; c<chessboard.length; c++) {
                System.out.print(chessboard[r][c]+" ");
            } System.out.println();
        }
    }

    void PlaceNQueens (char chessboard[][], int col) {

        // If all the columns have a queen 'Q', a solution has been found.
        if (col >= chessboard.length) {
            ++boardcnt;
            System.out.println("Board "+boardcnt);
            System.out.println("=======================");
            DisplayBoard(chessboard);
            System.out.println("=======================");

        } else {
            // Else try placing the queen on each row of the column and check if
the chessboard remains OK.
            for (int row=0; row<chessboard.length; row++) {

                chessboard[row][col] = 'Q';

                if (IsBoardOk(chessboard, row, col) == true) {
                    //Chess board was OK, hence try placing the queen 'Q' in the
next column.
                    PlaceNQueens(chessboard, col + 1);
                }
                chessboard[row][col] = '.'; // As previously placed queen was
not valid, restore '.'
            }
        }
    }

    public static void main(String args[]) {

       int N;

       Scanner obj_scanner = new Scanner(System.in);  // Create a Scanner object
       System.out.print("Enter chessboard size : ");

       N = obj_scanner.nextInt();  // Get user input

       char chessboard[][] = new char[N][N];

       for (int r=0; r<N; r++) {
           for (int c=0; c<N; c++) {
```

```
                chessboard[r][c] = '.';
            }
        }

        NQueens obj = new NQueens();

        // Start placing the queen 'Q' from the 0'th column.
        obj.PlaceNQueens(chessboard, 0);
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\DAA Observation\" ; if ($?)
 javac NQueens.java } ; if ($?) { java NQueens }
Enter chessboard size : 4
Board 1
=======================
. . Q .
Q . . .
. . . Q
. Q . .
=======================
Board 2
=======================
. Q . .
. . . Q
Q . . .
. . Q .
=======================
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(n^n)
- **Space Complexity:** O(n^2)

EXPERIMENT NO-16

Aim- WAP to solve the sum of subsets problem using backtracking

Algorithm

1. Read the number of elements (**n**).
2. Read the elements of the set.
3. Read the target sum (**sum**).
4. Initialize an array **solution** to store the selected elements in the subset.
5. Use backtracking to explore all possible combinations of elements and check if the sum matches the target.
6. Print the subsets that have the target sum.

Source Code:

```
import java.io.*;

class sos {
    int m;
    int w[];
    int x[];
    public sos() {
        w = new int[40];
        x = new int[40];
    }

    public void sos1(int s, int k, int r) {
        int i;
        x[k] = 1;
        if (s + w[k] == m) {
            for (i = 0; i <= k; i++)
                System.out.print(x[i] + "\t");
            System.out.println();
            System.out.print("Elements of set are :");
            for (i = 0; i <= k; i++)

                if (x[i] == 1)

                    System.out.print(w[i] + "\t");

            System.out.println();

        } else if ((s + w[k] + w[k + 1]) <= m)
            sos1(s + w[k], k + 1, r - w[k]);
        if ((s + r - w[k] >= m) && (s + w[k + 1] <= m)) {
            x[k] = 0;
            sos1(s, k + 1, r - w[k]);
```

```
        }
    }
}

class SumOFSubset {
    public static void main(String args[]) throws IOException {
        BufferedReader Bobj = new BufferedReader(new
InputStreamReader(System.in));
        int i, r = 0;
        sos o = new sos();
        System.out.print("Enter the number of elements of set : ");
        int n = Integer.parseInt(Bobj.readLine());
        System.out.print("Enter the elements : ");
        for (i = 0; i < n; i++) {
            o.w[i] = Integer.parseInt(Bobj.readLine());
            r = r + o.w[i];
        }

        System.out.print("Enter the sum to be computed: ");
        o.m = Integer.parseInt(Bobj.readLine());
        System.out.println("Subset whose sum is " + o.m + " are as follows: ");
        o.sos1(0, 0, r);
    }
}
```

OutPut:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\
DAA Observation\" ; if ($?) { javac SumOFSubset.java } ; if ($?) { java SumOFSubset }
Enter the number of elements of set : 4
Enter the elements : 2
3
4
5
Enter the sum to be computed: 12
Subset whose sum is 12 are as follows:
0       1       1       1
Elements of set are :3  4       5
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(2^n)
- **Space Complexity:** O(n)

EXPERIMENT NO-17

Aim-WAP to perform graph colouring using dynamic programming

Algorithm

1. Read the number of vertices (**V**).
2. Read the adjacency matrix representing the graph.
3. Initialize an array **color** to store the colors assigned to each vertex.
4. Use backtracking to explore all possible colorings while checking for conflicts.
5. Print the colors assigned to each vertex.

Source Code:

```
/**
 **  Java Program to Implement Graph Coloring Algorithm
 **/

import java.util.Scanner;

/** Class GraphColoring **/
public class GraphColoring {
    private int V, numOfColors;
    private int[] color;
    private int[][] graph;

    /** Function to assign color **/
    public void graphColor(int[][] g, int noc) {
        V = g.length;
        numOfColors = noc;
        color = new int[V];
        graph = g;

        try {
            solve(0);
            System.out.println("No solution");
        } catch (Exception e) {
            System.out.println("\nSolution exists ");
            display();
        }
    }

    /** function to assign colors recursively **/
    public void solve(int v) throws Exception {
        /** base case - solution found **/
        if (v == V)
            throw new Exception("Solution found");
```

```
        /** try all colours **/
        for (int c = 1; c <= numOfColors; c++) {
            if (isPossible(v, c)) {
                /** assign and proceed with next vertex **/
                color[v] = c;
                solve(v + 1);
                /** wrong assignement **/
                color[v] = 0;
            }
        }
    }

    /** function to check if it is valid to allot that color to vertex **/
    public boolean isPossible(int v, int c) {
        for (int i = 0; i < V; i++)
            if (graph[v][i] == 1 && c == color[i])
                return false;
        return true;
    }

    /** display solution **/
    public void display() {
        System.out.print("\nColors : ");
        for (int i = 0; i < V; i++)
            System.out.print(color[i] + " ");
        System.out.println();
    }
    /** Main function **/
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Graph Coloring Algorithm Test\n");
        /** Make an object of GraphColoring class **/
        GraphColoring gc = new GraphColoring();

        /** Accept number of vertices **/
        System.out.print("Enter number of vertices : ");
        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();

        System.out.print("\nEnter number of colors :");
        int c = scan.nextInt();
        scan.close();

        gc.graphColor(graph, c);
    }
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documents\D
AA Observation\" ; if ($?) { javac GraphColoring.java } ; if ($?) { java GraphColoring }
Graph Coloring Algorithm Test

Enter number of vertices : 4

Enter matrix :

0 1 2 3
1 3 2 0
1 1 2 0
3 2 1 2

Enter number of colors :3

Solution exists

Colors : 1 2 3 1
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(m^V) (In the worst case)
- **Space Complexity:** O(V)

EXPERIMENT NO-18

Aim- WAP to find if a graph has a Hamiltonian cycle

Algorithm

1. Read the number of vertices (`V`).
2. Read the adjacency matrix representing the graph.
3. Initialize an array `path` to store the Hamiltonian cycle.
4. Use backtracking to explore all possible Hamiltonian cycles.
5. Print the Hamiltonian cycle if one is found.

Source Code:

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

public class HamiltonianCycle
{
    private int V, pathCount;
    private int[] path;
    private int[][] graph;
    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();
        }
    }
    public void solve(int vertex) throws Exception
    {
        if (graph[vertex][0] == 1 && pathCount == V)
            throw new Exception("Solution found");
        if (pathCount == V)
            return;
        for (int v = 0; v < V; v++)
        {
```

```
            if (graph[vertex][v] == 1 )
            {
                path[pathCount++] = v;
                graph[vertex][v] = 0;
                graph[v][vertex] = 0;
                if (!isPresent(v))
                    solve(v);
                graph[vertex][v] = 1;
                graph[v][vertex] = 1;
                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);
        scan.close();
    }
}
```

Output:

```
ments\DAA Observation\" ; if ($?) { javac HamiltonianCycle.java } ; if ($?) { java HamiltonianCycle }
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
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

Analysis

- **Time Complexity:** O(V!)
- **Space Complexity:** O(V)

EXPERIMENT NO-19

Aim-WAP to solve Knapsack problem using backtracking approach

Algorithm

1. Read the number of items (**n**).
2. Read the weights and values of each item.
3. Read the maximum weight capacity of the knapsack (**W**).
4. Initialize an array **solution** to store the selected items in the knapsack.
5. Use backtracking to explore all possible combinations of items and weights.
6. Print the items included in the knapsack and their total value.

Source Code:

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

public class Knapsack {

    static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    static boolean isSafe(int wt[], int w, int n, int capacity, int
currentWeight) {
        return (currentWeight + wt[w] <= capacity);
    }

    static void knapsackUtil(int wt[], int val[], int n, int capacity, int
currentWeight, int currentValue,
            boolean selected[], boolean currentSelected[], int[] maxValue,
boolean[] finalSelected) {
        if (currentWeight == capacity || n == -1) {
            if (currentValue > maxValue[0]) {
                maxValue[0] = currentValue;
                System.arraycopy(currentSelected, 0, finalSelected, 0, n + 2);
            }
            return;
        }

        currentSelected[n] = true;
        if (isSafe(wt, n, n, capacity, currentWeight + wt[n])) {
            knapsackUtil(wt, val, n - 1, capacity, currentWeight + wt[n],
                    currentValue + val[n], selected, currentSelected, maxValue,
finalSelected);
        }

        currentSelected[n] = false;
```

```
        knapsackUtil(wt, val, n - 1, capacity, currentWeight, currentValue,
                selected, currentSelected, maxValue, finalSelected);
    }

    static void knapsack(int wt[], int val[], int n, int capacity) {
        boolean selected[] = new boolean[Knapsack.MAX_ITEMS];
        boolean currentSelected[] = new boolean[Knapsack.MAX_ITEMS];
        boolean finalSelected[] = new boolean[Knapsack.MAX_ITEMS];
        int[] maxValue = { 0 };

        knapsackUtil(wt, val, n - 1, capacity, 0, 0, selected, currentSelected,
maxValue, finalSelected);

        System.out.println("Selected items for maximum value:");
        for (int i = 0; i < n; i++) {
            if (finalSelected[i]) {
                System.out.println("Item " + (i + 1));
            }
        }

        System.out.println("Maximum value: " + maxValue[0]);
    }

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

        System.out.print("Enter the number of items: ");
        int n = scanner.nextInt();

        int wt[] = new int[Knapsack.MAX_ITEMS];
        int val[] = new int[Knapsack.MAX_ITEMS];

        System.out.println("Enter the weights of the items:");
        for (int i = 0; i < n; i++)
            wt[i] = scanner.nextInt();

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

        System.out.print("Enter the capacity of the knapsack: ");
        int capacity = scanner.nextInt();

        Knapsack.knapsack(wt, val, n, capacity);

        scanner.close();
    }

    static final int MAX_ITEMS = 20;
}
```

Output:

```
PS C:\Users\hibba\OneDrive\Documents\DAA Observation> cd "c:\Users\hibba\OneDrive\Documen
DAA Observation\" ; if ($?) { javac Knapsack.java } ; if ($?) { java Knapsack }
Knapsack Algorithm Test

Enter number of elements
4

Enter weight for 4 elements
19 18 12 11

Enter value for 4 elements
4 5 6 3

Enter knapsack weight
33

Items selected :
2 3
PS C:\Users\hibba\OneDrive\Documents\DAA Observation>
```

- **Time Complexity:** O(2^n)
- **Space Complexity:** O(n)

EXPERIMENT NO-20

Aim- WAP to demonstrate – Least Cost (LC) search

Algorithm

1. Read the number of vertices (**V**).
2. Read the adjacency matrix representing the graph.
3. Read the starting vertex (**startVertex**).
4. Initialize an array **visited** to keep track of visited vertices.
5. Perform LC search starting from the specified vertex.
6. Print the vertices visited in the order of traversal.

Source Code:

```
#include <stdio.h>

#define MAX 100

void LCSearch(int graph[MAX][MAX], int V, int startVertex) {
   int visited[MAX] = {0};
   int queue[MAX];
   int front = -1, rear = -1;

   // Enqueue the starting vertex
   queue[++rear] = startVertex;
   visited[startVertex] = 1;

   while (front != rear) {
      // Dequeue a vertex and print it
      int currentVertex = queue[++front];
      printf("%d ", currentVertex);

      // Enqueue adjacent vertices that have not been visited
      for (int i = 0; i < V; i++) {
         if (graph[currentVertex][i] && !visited[i]) {
            queue[++rear] = i;
            visited[i] = 1;
         }
      }
   }
}
```

```
int main() {
    int V;

    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    int graph[MAX][MAX];

    printf("Enter the adjacency matrix representing the graph:\n");
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            scanf("%d", &graph[i][j]);

    int startVertex;

    printf("Enter the starting vertex: ");
    scanf("%d", &startVertex);

    printf("Least Cost Search starting from vertex %d: ", startVertex);
    LCSearch(graph, V, startVertex);

    return 0;
}
```

OutPut:

```
Enter the number of vertices: 4
Enter the adjacency matrix representing the graph:
0 1 1 1
1 0 1 0
0 1 1 1
1 1 0 0
Enter the starting vertex: 0
Least Cost Search starting from vertex 0: 0 1 2 3
```

Analysis

- **Time Complexity:** O(V^2)
- **Space Complexity:** O(V)