DAYANANDA SAGAR COLLEGE OF ENGINEERING

An Autonomous Institution
Affiliated to VTU
Approved by AICTE & UGC
Accredited by NBA
Accredited by NAAC with 'A' grade

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY 22CGL45

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

DAYANANDA SAGAR COLLEGE OF ENGINEERING

(AN AUTONOMOUS INSTITUTE AFFILIATED TO VTU, BELAGAVI)

Shavige Malleshwara Hills, Kumaraswamy Layout, Bangalore-560078

Course Name and Course Code	**: Design and Analysis of Algorithms Laboratory(22CGL45)**
Year and Semester	**: II year, IV semester**
Name of the Faculty	**: Dr. Shobha N**

VISION AND MISSION OF THE INSTITUTION

INSTITUTION VISION

To impact quality technical education with a focus on Research and Innovation emphasizing on Development of Sustainable and Inclusive Technology for the benefit of society.

INSTITUTION MISSION

- To provide an environment that enhances creativity and Innovation in pursuit of Excellence.
- To nurture teamwork in order to transform individuals as responsible leaders and entrepreneurs.
- To train the students to the changing technical scenario and make them to understand the importance of Sustainable and Inclusive technologies.

VISION AND MISSION OF CSE DEPARTMENT

DEPARTMENT VISION

Computer Science and Design Engineering Department shall architect the most innovative programs to deliver competitive and sustainable solutions using cutting edge technologies and implementations, for betterment of society and research.

DEPARTMENT MISSION

- To adopt the latest industry trends in teaching learning process in order to make students competitive in the job market
- To encourage forums that enable students to develop skills in multidisciplinary areas and emerging technologies
- To encourage research and innovation among students by creating an environment of learning through active participation and presentations
- To collaborate with industry and professional bodies for the students to gauge the market trends and train accordingly.
- To create an environment which fosters ethics and human values to make students responsible citizens.

COURSE OUTCOMES (CO)

COURSEOUTCOMES:ATTHE ENDOFTHECOURSE,STUDENT WILL BE ABLE TO	
CO1	Develop a solid understanding of algorithmic problem-solving techniques and their efficiency analysis. Gain proficiency in analyzing the time and space complexity of algorithms using mathematical analysis and asymptotic notations. Apply these skills to evaluate and compare the performance of brute force algorithms such as selection sort, bubble sort, sequential search, and brute-force string matching.
CO2	Demonstrate proficiency in applying decrease and conquer and divide and conquer techniques to solve algorithmic problems. Implement and analyze algorithms such as insertion sort, depth-first search (DFS), breadth-first search (BFS), topological sorting, merge sort, quicksort, multiplication of long integers, and Strassen's matrix multiplication.
CO3	Apply transform-and-conquer techniques, including pre-sorting, heapsort, and Horner's rule. Understand the space and time trade-offs in algorithms, and apply techniques such as sorting by counting, naive string matching, Horspool's algorithm, and the Boyer-Moore algorithm.
CO4	Develop a strong understanding of dynamic programming and greedy techniques in algorithm design. Apply dynamic programming to solve problems such as the binomial coefficient, the Knapsack problem, and the algorithms of Warshall and Floyd. Apply greedy techniques to solve problems such as Prim's Algorithm, Kruskal's Algorithm, and Dijkstra's Algorithm. Gain proficiency in analysing problem characteristics and selecting appropriate algorithmic approaches for efficient problem-solving.
CO5	Apply advanced algorithmic problem-solving techniques such as backtracking and branch-and-bound to solve complex problems like the n-Queens problem, the Subset-Sum problem, the Traveling Salesperson problem, and the 0/1 Knapsack problem. Understand the concepts of NP and NP-complete problems, including basic concepts, nondeterministic algorithms, and the classes P, NP, NP-complete, and NP-hard.

Design and Analysis of Algorithms Laboratory			
Course code:	**22CGL45**	**Credits:**	**1**
L:T:P:	**0:0:2**	**CIE Marks:**	**50**
Exam Hours:	**03**	**SEE Marks:**	**50**
Total Hours:	**12**		

Experiment No.	Contents of the experiment	Hours	COs
1.	Sort a given set of elements using Selection Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.	2	CO1
2.	Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.	2	CO2
3.	Sort a given set of N integer elements using Quick Sort technique and compute its time taken.	2	CO2
4.	Write a program to implement Horspool's algorithm for String Matching.	2	CO3
5.	Design and implement Program to solve 0/1 Knapsack problem using Dynamic Programming method.	2	CO4
6.	a. Implement Warshall's algorithm using dynamic programming. b. Implement All Pair Shortest paths problem using Floyd's algorithm.	2	CO4
7.	Implement Single source shortest path using Dijkstra's algorithm.	2	CO4
8.	Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.	2	CO4
9.	Find Minimum Cost Spanning Tree of a given undirected graph using Kruskals algorithm.	2	CO4
10.	Design and implement Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.	2	CO4

11.	Implement “Sum of Subsets” using Backtracking.: Find a subset of a given set S = {s1, s2,……,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S = {1,2,5,6,8} and d = 9 there are two solutions {1,2,6} and {1,8}. A suitable message is to be displayed if the given problem instance doesn’t have a solution.	2	CO5
12.	Implement “N-Queens Problem” using Backtracking.	2	CO5

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	-	2	-	-	-	-	-	-	-	-	1	-	-
CO2	3	2	-	-	-	2	2	-	-	-	-	-	1	-	-
CO3	-	3	-	-	-	2	2	-	-	-	-	1	2	1	-
CO4	-	3		-	-	2	-	2	-	-	-	1	1	1	-
CO5	3	2	2	-	-	-	-	-	-	-	-	1	1	1	-

INTRODUCTION

An algorithm is a description of a procedure which terminates with a result. It is a step by step procedure designed to perform an operation, and which will lead to the sought result if followed correctly. Algorithms have a definite beginning and a definite end, and a finite number of steps. An algorithm produces the same output information given the same input information, and several short algorithms can be combined to perform complex tasks.

PROPERTIES OF ALGORITHMS

1. Finiteness – an algorithm must terminate after a finite number of steps.
2. Definiteness – each step must be precisely defined.
3. Effective – all operations can be carried out exactly in finite time.
4. Input – an algorithm has one or more outputs.

MODELS OF COMPUTATION

There are many ways to compute the algorithm's complexity; they are all equivalent in some sense.

1. Turing machines.
2. Random access machines (RAM).
3. λ Calculus.
4. Recursive functions.
5. Parallel RAM (PRAM).

For the analysis of algorithms, the RAM model is most often employed. Note time does not appear to be reusable, but space often is.

MEASURING AN ALGORITHM'S COMLEXITY

General consideration of time and space is:

- One time unit per operation.
- One space unit per value (all values fit in fixed size register).

There are other considerations:

- On real machines, different operations typically take different times to execute. This will generally be ignored, but sometimes we may wish to count different types of operations, e.g., Swaps and compares, or additions and multiplications.
- Some operations may be "noise" not truly affecting the running time; we typically only count "major" operations example, for loop index arithmetic and boundary checking is "noise." operations inside for loops are usually "major."
- Logarithmic cost model: it takes $\lfloor lgn \rfloor + 1$ bits to represent natural number *n* in binary notation. Thus the uniform model of time and space may not bias results for large integers (data).

An algorithm solves an instance of a problem. There is, in general, one parameter, the input size, denoted by *n*, which is used to characterize the problem instance. The input size *n* is the number of registers needed to hold input (data segment size).

Given n, we'd like to find:

1. The time complexity, denoted by T(n), which is the count of operations the algorithm performs on the given input.
2. The space complexity, denoted by S(n), which is the number of memory registers used by the algorithm (stack/heap size, registers).

Note that T(n) and S(n) are relations rather than functions. That is, for different input of the same size n, T(n) and S(n) may provide different answers.

WORST AVERAGE AND BEST COMPLEXITY

Complexities usually not measured exactly: big- O, Ω, and Θ notation is used.

WORST CASE:

This is the longest time (or most space) that the algorithm will use over all instances of size n. Often this can be represented by a function f(n) such as **$f(n)=n^2$** or **f(n)=n lg n.** We write T(n)= O(f(n)) for the worst case time complexity. Roughly, this means the algorithm will take mo more than f(n) operations.

BEST CASE:

This is the shortest time that the algorithm will use over all instances of size n. often this can be represented by a function f(n) such as **$f(n)=n^2$** or **f(n)=n lg n**. We write **T(n)= Ω (f(n))** for the best case. Roughly, this means the algorithm will take no less than f(n) operations. The best case is seldom interesting.

When the worst and best case performance of an algorithm are the same we can write **T(n)= Θ (f(n))**. Roughly, this says the algorithm always uses f(n) operations on all instances of size n.

AVERAGE CASE:

This is the average time that the algorithm will use over all instances if size n. it depends on the probability distribution of instances of the problem.

AMORTIZED COST:

This is used when a sequence of operations occur, e.g., inserts and deletes in a tress, where the costs vary depending on the operations and their order. For example, some may take a few steps, some many.

TYPES OF ALGORITHMS

1. Off-line algorithms: all input in memory before time starts, want final result.
2. On-line: input arrives at discrete time steps, intermediate result furnished before next input.
3. Real-time: elapsed time between two inputs (outputs) is a constant O(1).

COMPLEXITY CLASSES

Collection of problems that required roughly the same amount of resources from complexity classes. Here is a list of the most important:

1. The class P of problems that can be solved in a polynomial number of operations of the input size on a deterministic Turing machine.
2. The class NP of problems that can be solved in a polynomial number of operations of the input size on a non-deterministic Turing machine.
3. The class of problems that can be solved in a constant amount of space.

4. The class L that can be solved in a logarithmic amount of space based on the input size.
5. The class PSPACE of problems that can be solved in a polynomial amount of space based on the input size.
6. The class NC of problems that can be solved in poly-logarithmic time on a polynomial number of processors.

ALGORITHM PARADIGMS

Often there are large collections of problems that can be solved using the same general techniques or paradigms. A few of the most common are described below:

Brute Force:

A straightforward approach to solving a problem based on the problem statement and concepts involved. Brute force algorithms are rarely efficient. Example algorithms include:

- Bubble sort.
- Computing the sum of n numbers by direct addition.
- Standard matrix multiplication.
- Linear search.

Divide and Conquer:

Perhaps the most famous algorithm paradigm, divide and conquer is based on partitioning the problem into two or more smaller sub-problems, solving them and combining the sub-problem solutions into a solution for the original problem. Example algorithms include:

- Merge sort and quick sort.
- The Fast Fourier Transform (FFT).
- Stassen's matrix multiplication.

Greedy Algorithms:

Greedy algorithms always make the choice that seems best at the moment. This is locally optimal choice is made with the hope that it leads to a globally optimal solution. Some greedy algorithms may not be guaranteed to always produce an optimal solution.

Greedy algorithms are often applied to combinatorial optimization problems.

- Given an instance 1 of the problem.
- There is a set of candidates or feasible solutions that satisfy the constraints of the problem.
- For each feasible solution there is a value determined by an objective function.
- An optimal solution minimizes (or maximizes) the value of objective function.

Example algorithms include:

- Kruskal's and Prim's minimal spanning tree algorithms.
- Dijkstra's single source shortest path algorithm.
- Huffman coding.

Dynamic Programming:

A nutshell definition of dynamic programming is difficult, but to summarize, problems which lend themselves to a dynamic programming attack have the following characteristics:

- We have to search over a large space for an optimal solution.
- The optimal solution can be expressed in terms of optimal solution to sub-problem.
- The number of sub-problems that must be solved is small.

Dynamic programming algorithms have the following features:

- A recurrence that is implemented iteratively.
- A table, built to support the iteration.
- Tracing through the table to find the optimal solution.

Example algorithms include:

- Binomial Coefficient, Knapsack problem
- The Floyd's, Warshall's algorithm.

1. Sort a given set of elements using Selection Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

```
import java.util.*;
public class Main {
   public static int [] randomTester(int max, int min, int min_count, int
max_count){
      Scanner s = new Scanner(System.in);
    System.out.println("Enter the size of the array ");
      int len = s.nextInt();
      int [] arr = new int[len];
      int i=0;
      while(i<len){
         int value=(int)(Math.random() * max) + min;
         arr[i]=value;
         System.out.print(arr[i] + " ");
         i++;
      }
      System.out.println();

      return arr;
   }

   static void selectionSort(int[] arr) {
   for (int i = 0; i < arr.length - 1; i++) {
      int minIdx = i;
      for (int j = i + 1; j < arr.length; j++) {
         if (arr[j] < arr[minIdx])
            minIdx = j;
      }
      if (minIdx != i) {
         int temp = arr[i];
         arr[i] = arr[minIdx];
         arr[minIdx] = temp;
      }
   }
}

   public static void arrPrint(int arr[]){
      for(int i=0; i<arr.length; i++)
         System.out.print(arr[i]+"\t");
         System.out.println();
   }
   public static void main(String[] args) {
      int[] arr = randomTester(1000, 1, 10, 100);

      long startTime = System.nanoTime();
      selectionSort(arr);
      long time_taken = System.nanoTime() - startTime;
```

```
        arrPrint(arr);

        System.out.println("SelectionSort - Time taken: "+ time_taken +" ns |
Length - "+arr.length);
    }
}
```

OUT PUT

```
 enter the number of elements
10

 Array elements are
383     886     777     915     793     335     386     492     649     421
 sorted elements are
 335
 383
 386
 421
 492
 649
 777
 793
 886
 915
 time taken is:10.000000
```

2. Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

```
import java.util.Random;

public class MergeSort {

   static int[] a = new int[20];

   public static void simpleMerge(int low, int mid, int high) {
      int i, j, k;
      int[] c = new int[20];
      k = low;
      i = low;
      j = mid + 1;
      while (i <= mid && j <= high) {
         if (a[i] < a[j])
            c[k++] = a[i++];
         else
            c[k++] = a[j++];
      }
      while (i <= mid)
         c[k++] = a[i++];
      while (j <= high)
         c[k++] = a[j++];

try {
            Thread.sleep(1000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      for (i = low; i <= k - 1; i++)
         a[i] = c[i];
   }

   public static void mergeSort(int low, int high) {
      int mid;

      if (low < high) {
         mid = (low + high) / 2;
         mergeSort(low, mid);
         mergeSort(mid + 1, high);
         simpleMerge(low, mid, high);
      }
   }

   public static void main(String[] args) {
      int n, i, val;
      long start, end;
```

```
        System.out.println("Enter value of n:");
        n = Integer.parseInt(System.console().readLine());

        System.out.println("\nArray values are (randomly generated):");
        Random rand = new Random();
        for (i = 0; i < n; i++) {
           val = rand.nextInt(100);
           a[i] = val;
           System.out.println(a[i] + ", Thread id=" +
    Thread.currentThread().getId());
        }

        start = System.currentTimeMillis();
        mergeSort(0, n - 1);
        System.out.println("Sorted array is:");
        for (i = 0; i < n; i++)
           System.out.println(a[i]);

        end = System.currentTimeMillis();

        System.out.println("\nTime taken : " + (end - start) + " milliseconds");
     }
  }
```

OUTPUT

```
Enter the number of elements 9
Enter the Array elements
83
93
35
86
92
15
77
49
21
Sorted Array
15
21
35
49
77
83
86
92
93
 Time taken  : 9.00000
```

3. Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

```
import java.util.Random;

public class QuickSort {

   static int[] a = new int[20];
   static int n, i, j, temp;

   public static void quickSort(int low, int high) {
      int j;
      if (low <= high) {
         j = partition(low, high);
         quickSort(low, j - 1);
         quickSort(j + 1, high);
      }
   }

   public static int partition(int low, int high) {
      int i, j, key;
      key = a[low];
      i = low + 1;
      j = high;

      while (true) {
         while (i < high && a[i] < key)
            i++;
         while (key < a[j])
            j--;
         if (i < j) {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
         } else {
            temp = a[low];
            a[low] = a[j];
            a[j] = temp;
             return j;
         }
      }
   }

   public static void main(String[] args) {
      long start, end;
      int val;

      System.out.println("\nEnter the number of elements:");
      n = Integer.parseInt(System.console().readLine());
      System.out.println("\nArray elements are:");
```

```
        Random rand = new Random();
        for (i = 0; i < n; i++) {
            val = rand.nextInt(100);
            a[i] = val;
            System.out.print(a[i] + "\t");
        }
        start = System.currentTimeMillis();
        quickSort(0, n - 1);
        System.out.println("\nSorted elements are:");
        for (i = 0; i < n; i++)
            System.out.println(a[i]);
        end = System.currentTimeMillis();

        System.out.println("\nTime taken is: " + (end - start) + " milliseconds");
    }
}
```

OUT PUT

```
Enter the number of elements:
20

Array elements are:
65      20      63      98      65      68      81      14      4       31      52
        74      98      26      67      19      13      71      84      50
Sorted elements are:
4
13
14
19
20
26
31
50
52
63
65
65
67
68
71
74
81
84
98
98

Time taken is: 1 milliseconds
```

4. Write a program to implement Horspool's algorithm for String Matching.

```
public class HorspoolStringMatching {
    private static final int ALPHABET_SIZE = 256;

    public static void main(String[] args) {
        String text = "BESS_KNEW_ABOUT_BAOBABS";
        String pattern = "BAOBAB";

        int index = search(text, pattern);
        if (index != -1) {
            System.out.println("Pattern found at index " + index);
        } else {
            System.out.println("Pattern not found in the text.");
        }
    }

    public static int[] preprocessPattern(String pattern) {
        int[] table = new int[ALPHABET_SIZE];
        int patternLength = pattern.length();

        for (int i = 0; i < ALPHABET_SIZE; i++) {
            table[i] = patternLength;
        }

        for (int i = 0; i < patternLength - 1; i++) {
            char c = pattern.charAt(i);
            table[c] = patternLength - 1 - i;
        }

        return table;
    }

    public static int search(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();

        int[] shiftTable = preprocessPattern(pattern);
        int i = patternLength - 1;

        while (i < textLength) {
            int j = patternLength - 1;

            while (j >= 0 && text.charAt(i) == pattern.charAt(j)) {
                i--;
                j--;
            }

            if (j == -1) {
                return i + 1;           } else {
```

```
                i += Math.max(1, patternLength - 1 - j + shiftTable[text.charAt(i)]);
            }
        }

        return -1;
    }
}
```

OUTPUT

Pattern found at index 16

5. Design and implement Program to solve 0/1 Knapsack problem using Dynamic Programming method.

```
import java.util.Scanner;

public class Knapsack {

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

   public static void knapsack(int n, int[] w, int m, int[][] v, int[] p) {
      int i, j;

      for (i = 0; i <= n; i++) {
         for (j = 0; j <= m; j++) {
            if (i == 0 || j == 0)
               v[i][j] = 0;
            else if (j < w[i])
               v[i][j] = v[i - 1][j];
            else
               v[i][j] = max(v[i - 1][j], v[i - 1][j - w[i]] + p[i]);
         }
      }
   }

   public static void main(String[] args) {
      int m, i, j, n;
      int[] p = new int[10];
      int[] w = new int[10];
      int[][] v = new int[10][10];

      Scanner scanner = new Scanner(System.in);

      System.out.println("Enter the number of objects:");
      n = scanner.nextInt();

      System.out.println("Enter the weights of n objects:");
      for (i = 1; i <= n; i++)
         w[i] = scanner.nextInt();

      System.out.println("Enter the profits of n objects:");
      for (i = 1; i <= n; i++)
         p[i] = scanner.nextInt();

      System.out.println("Enter the capacity of Knapsack:");
      m = scanner.nextInt();

      knapsack(n, w, m, v, p);
```

```
        System.out.println("The output is:");
        for (i = 0; i <= n; i++) {
            for (j = 0; j <= m; j++)
                System.out.print(v[i][j] + " ");
            System.out.println();
        }

        scanner.close();
    }
}
```

OUTPUT

```
Enter the number of objects:
4
Enter the weights of n objects:
2
1
3
2
Enter the profits of n objects:
12
20
15
10
Enter the capacity of Knapsack:
5
The output is:
0 0 0 0 0 0
0 0 12 12 12 12
0 20 20 32 32 32
0 20 20 32 35 35
0 20 20 32 35 42
```

6. a. Implement Warshall's algorithm using dynamic programming.

```
import java.util.Scanner;

public class TransitiveClosure {
   public static void main(String[] args) {
      int i, j, k, n;
      int[][] cost = new int[20][20];

      Scanner scanner = new Scanner(System.in);

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

      System.out.println("\nEnter the adjacency matrix:");
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            cost[i][j] = scanner.nextInt();

      for (k = 1; k <= n; k++) {
         for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
               cost[i][j] = cost[i][j] | (cost[i][k] & cost[k][j]);
            }
         }
      }

      System.out.println("\nThe transitive closure is:");
      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            System.out.print(cost[i][j] + "\t");
         }
         System.out.println();
      }

      scanner.close();
   }
}
```

Output

```
Enter the number of nodes: 4
Enter the adjacency matrix:
0 1 0 1
1 0 0 0
1 0 1 0
0 1 0 1
The transitive closure is:
1    1    0    1
1    1    0    1
1    1    1    1
1    1    0    1
```

b. **Implement All Pair Shortest paths problem using Floyd's algorithm.**

```
import java.util.Scanner;

public class FloydWarshall {
   static int[][] a = new int[10][10];
   static int n;

   public static void floyd() {
      int i, j, k;

      for (k = 1; k <= n; k++) {
         for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
               if ((a[i][k] + a[k][j]) < a[i][j])
                  a[i][j] = a[i][k] + a[k][j];
            }
         }
      }
   }

   public static void main(String[] args) {
      int i, j;

      Scanner scanner = new Scanner(System.in);

      System.out.print("\nEnter the number of nodes: ");
      n = scanner.nextInt();

      System.out.println("\nEnter the cost adjacency matrix:");
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            a[i][j] = scanner.nextInt();

      floyd();

      System.out.println("\nThe shortest path matrix is:");
      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            System.out.print(a[i][j] + "\t");
         }
         System.out.println();
      }

      scanner.close();
   }
}
```

OUTPUT

```
Enter the number of nodes:
4

Enter the cost adjacency matrix:
0 999 3 999
2 0 999 999
999 7 0 1
6 999 999 0

The shortest path matrix is:
0       10      3       4
2       0       5       6
7       7       0       1
6       16      9       0
```

7. Implement Single source shortest path using Dijkstra's algorithm.

```
import java.util.Scanner;

public class Dijkstra {

   public static void dijkstra(int n, int v, int[][] cost, int[] dist) {
      int i, u, count, w;
      int[] flag = new int[10];
      int min;

      for (i = 1; i <= n; i++) {
         flag[i] = 0;
         dist[i] = cost[v][i];
      }
      flag[v] = 1;
      dist[v] = 0;
      count = 2;

      while (count <= n) {
         min = 999;
         u = 0;
         for (w = 1; w <= n; w++) {
            if (dist[w] < min && flag[w] == 0) {
               min = dist[w];
               u = w;
            }
         }
         flag[u] = 1;
         count++;

         for (w = 1; w <= n; w++) {
            if ((dist[u] + cost[u][w] < dist[w]) && flag[w] == 0) {
               dist[w] = dist[u] + cost[u][w];
            }
         }
      }
   }


   public static void main(String[] args) {
      int n, v, i, j;
      int[][] cost = new int[10][10];
      int[] dist = new int[10];

      Scanner scanner = new Scanner(System.in);

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

```
        System.out.println("\nEnter the cost matrix:");
        for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
                cost[i][j] = scanner.nextInt();
                if (cost[i][j] == 0) {
                    cost[i][j] = 999;
                }
            }
        }

        System.out.print("\nEnter the source vertex: ");
        v = scanner.nextInt();

        dijkstra(n, v, cost, dist);

        System.out.println("Shortest paths from vertex " + v + ":");
        for (j = 1; j <= n; j++) {
            if (j != v) {
                System.out.println(v + "->" + j + ":::::" + dist[j]);
            }
        }

        scanner.close();
    }
}
```

OUTPUT

```
Enter the number of nodes:
5

Enter the cost matrix:
0 3 999 7 999
3 0 4 2 999
999 4 0 5 6
7 2 5 0 4
999 999 6 4 0

Enter the source vertex: 1
Shortest paths from vertex 1:
1->2:::::3
1->3:::::7
1->4:::::5
1->5:::::9
```

8. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

```
import java.util.Scanner;

public class PrimsAlgorithm {
   static int a, b, u, v, n, i, j, ne = 1;
   static int min, mincost = 0;
   static int[] visited = new int[10];
   static int[][] cost = new int[10][10];

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

      System.out.print("\nEnter the number of nodes: ");
      n = scanner.nextInt();

      System.out.println("\nEnter the adjacency matrix:");
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++) {
            cost[i][j] = scanner.nextInt();
            if (cost[i][j] == 0)
               cost[i][j] = 999;
         }

      visited[1] = 1;
      System.out.println();

      while (ne < n) {
         for (i = 1, min = 999; i <= n; i++)
            for (j = 1; j <= n; j++)
               if (cost[i][j] < min)
                  if (visited[i] != 0) {
                     min = cost[i][j];
                     a = u = i;
                     b = v = j;
                  }
         if (visited[u] == 0 || visited[v] == 0) {
            System.out.printf("\nEdge %d: (%d %d) cost: %d", ne++, a, b, min);
            mincost += min;
            visited[b] = 1;
         }
         cost[a][b] = cost[b][a] = 999;
      }

      System.out.println("\nMinimum cost = " + mincost);
      scanner.close();
   }
}
```

OUTPUT

```
Enter the number of nodes: 6

Enter the adjacency matrix:
0 3 999 999 6 5
3 0 1 999 999 4
999 1 0 6 999 4
999 999 6 0 8 5
6 999 999 8 0 2
5 4 4 5 2 0


Edge 1: (1 2) cost: 3
Edge 2: (2 3) cost: 1
Edge 3: (2 6) cost: 4
Edge 4: (6 5) cost: 2
Edge 5: (6 4) cost: 5
Minimum cost = 15
```

9. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskals algorithm.

```
import java.util.Scanner;

public class Kruskal {
   static int i, j, k, a, b, u, v, n, ne = 1;
   static int min, mincost = 0;
   static int[][] cost = new int[9][9];
   static int[] parent = new int[9];

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

      System.out.println("\n\n\tImplementation of Kruskal's algorithm\n\n");
      System.out.println("Enter the number of vertices:");
      n = scanner.nextInt();
      System.out.println("\nEnter the cost adjacency matrix:");

      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            cost[i][j] = scanner.nextInt();
            if (cost[i][j] == 0) {
               cost[i][j] = 999;
            }
         }
      }

      System.out.println("\nThe edges of Minimum Cost Spanning Tree are\n");
      while (ne < n) {
         for (i = 1, min = 999; i <= n; i++) {
            for (j = 1; j <= n; j++) {
               if (cost[i][j] < min) {
                  min = cost[i][j];
                  a = u = i;
                  b = v = j;
               }
            }
         }
         u = find(u);
         v = find(v);
         if (uni(u, v) == 1) {
            System.out.println(ne++ + " edge (" + a + "," + b + ") =" + min);
            mincost += min;
         }
         cost[a][b] = cost[b][a] = 999;
      }

      System.out.println("\nMinimum cost = " + mincost);
```

```
        scanner.close();
    }

    public static int find(int i) {
        while (parent[i] != 0)
            i = parent[i];
        return i;
    }

    public static int uni(int i, int j) {
        if (i != j) {
            parent[j] = i;
            return 1;
        }
        return 0;
    }
}
```

OUTPUT

```
Enter the number of vertices:
6

Enter the cost adjacency matrix:
0 3 999 999 6 5
3 0 1 999 999 4
999 1 0 6 999 4
999 999 6 0 8 5
6 999 999 8 0 2
5 4 4 5 2 0

The edges of Minimum Cost Spanning Tree are

1 edge (2,3) =1
2 edge (5,6) =2
3 edge (1,2) =3
4 edge (2,6) =4
5 edge (4,6) =5

Minimum cost = 15
```

10. Design and implement Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

```
import java.util.Scanner;

public class KnapsackProblem {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      float[] weight = new float[50];
      float[] profit = new float[50];
      float[] ratio = new float[50];
      float totalValue = 0, temp, capacity;

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

      for (int i = 0; i < n; i++) {
         System.out.printf("Enter Weight and Profit for item[%d]:\n", i);
         weight[i] = scanner.nextFloat();
         profit[i] = scanner.nextFloat();
      }

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

      for (int i = 0; i < n; i++)
         ratio[i] = profit[i] / weight[i];

      for (int i = 0; i < n; i++) {
         for (int j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
               temp = ratio[j];
               ratio[j] = ratio[i];
               ratio[i] = temp;

               temp = weight[j];
               weight[j] = weight[i];
               weight[i] = temp;

               temp = profit[j];
               profit[j] = profit[i];
               profit[i] = temp;
            }
         }
      }

      System.out.println("Knapsack problems using Greedy Algorithm:");

      int i;
      for (i = 0; i < n; i++) {
         if (weight[i] > capacity)
```

```
                break;
            else {
                totalValue += profit[i];
                capacity -= weight[i];
            }
        }
        if (i < n)
            totalValue += ratio[i] * capacity;

        System.out.println("\nThe maximum value is: " + totalValue);

        scanner.close();
    }
}
```

OUTPUT

```
Enter the number of items: 4
Enter Weight and Profit for item[0]:
2
12
Enter Weight and Profit for item[1]:
3
10
Enter Weight and Profit for item[2]:
1
15
Enter Weight and Profit for item[3]:
2
20
Enter the capacity of knapsack:
5
Knapsack problems using Greedy Algorithm:

The maximum value is: 47.0
```

11. Implement "Sum of Subsets" using Backtracking.: Find a subset of a given set S = {s1, s2,……,sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S = {1,2,5,6,8} and d = 9 there are two solutions {1,2,6} and {1,8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

```
import java.util.Scanner;

public class SubsetSum {
   public static void subset(int n, int d, int[] w) {
      int[] x = new int[10];
      int i, k, s;

      for (i = 1; i < n; i++)
         x[i] = 0;

      s = 0;
      k = 1;
      x[k] = 1;

      while (true) {
         if (k <= n && x[k] == 1) {
            if (s + w[k] == d) {
               System.out.println("\nSolution is:");
               for (i = 1; i <= n; i++) {
                  if (x[i] == 1)
                     System.out.print(w[i] + " ");
               }
               System.out.println();
               x[k] = 0;
            } else if (s + w[k] < d)
               s += w[k];
            else
               x[k] = 0;
         } else {
            k--;
            while (k > 0 && x[k] == 0)
               k--;
            if (k == 0)
               break;
            s = s - w[k];
            x[k] = 0;
         }
         k = k + 1;
         x[k] = 1;
      }
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      int n, d, i;
```

```
        int[] w = new int[20];

        System.out.println("\nEnter the value of n:");
        n = scanner.nextInt();

        System.out.println("Enter the set values in increasing order:");
        for (i = 1; i <= n; i++)
            w[i] = scanner.nextInt();

        System.out.println("\nEnter the maximum value:");
        d = scanner.nextInt();

        subset(n, d, w);

        scanner.close();
    }
}
```

OUTPUT

```
Enter the value of n:
5
Enter the set values in increasing order:
1
2
3
4
5

Enter the maximum value:
9

Solution is:
1 3 5

Solution is:
2 3 4
```

12. Implement "N-Queens Problem" using Backtracking.

```
import java.util.Scanner;

public class NQueens {
   static int[][] s = new int[100][100];

   static void display(int[] m, int n) {
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            s[i][j] = 0;
         }
      }

      for (int i = 0; i < n; i++) {
         s[i][m[i]] = 1;
      }
      System.out.println();

      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            System.out.print(s[i][j] + " ");
         }
         System.out.println();
      }
   }

   static int place(int[] m, int k) {
      for (int i = 0; i < k; i++)
         if (m[i] == m[k] || (Math.abs(m[i] - m[k]) == Math.abs(i - k)))
            return 0;
      return 1;
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      int[] m = new int[25];
      int n, k;

      System.out.println("\nENTER THE NO OF QUEENS:");
      n = scanner.nextInt();

      System.out.println("\nTHE SOLUTION TO QUEENS PROBLEM IS\n");
      n--;

      for (m[0] = 0, k = 0; k >= 0; m[k] = m[k] + 1) {
         while (m[k] <= n && !(place(m, k) == 1))
            m[k] = m[k] + 1;

         if (m[k] <= n)
            if (k == n)
```

```
                display(m, n + 1);
              else {
                k++;
                m[k] = -1;
              }
            else
              k--;
          }
        }
      }
```

OUTPUT

```
ENTER THE NO OF QUEENS:
4

THE SOLUTION TO QUEENS PROBLEM IS


0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

Viva Questions:

1. What is an algorithm? What is the need to study Algorithms?
2. Explain Euclid's Algorithm to find GCD of two integers with an e.g.
3. Explain Consecutive Integer Checking algorithm to find GCD of two numbers with an e.g.
4. Middle School Algorithm with an e.g.
5. Explain the Algorithm design and analysis process with a neat diagram.
6. Define: a) Time Efficiency b) Space Efficiency.
7. What are the important types of problems that encounter in the area of computing?
8. What is a data structure? How are data structures classified?
9. Briefly explain linear and non-linear data structures.
10. What is a set? How does it differ from a list?
11. What are the different operations that can be performed on a set?
12. What are the different ways of defining a set?
13. How can sets be implemented in computer application?
14. What are different ways of measuring the running time of an algorithm?
15. What is Order of Growth?
16. Define Worst case, Average case and Best case efficiencies.
17. Explain the Linear Search algorithm.
18. Define O, Ω, Θ notations.
19. Give the general plan for analyzing the efficiency of non-recursive algorithms with an e.g.
20. Give an algorithm to find the smallest element in a list of n numbers and analyze the efficiency.
21. Give an algorithm to check whether all the elements in a list are unique or not and analyze the efficiency
22. Give an algorithm to multiply two matrices of order N*N and analyze the efficiency.
23. Give the general plan for analyzing the efficiency of Recursive algorithms with an e.g.
24. Give an algorithm to compute the Factorial of a positive integer n and analyze the efficiency.
25. Give an algorithm to solve the Tower of Hanoi puzzle and analyze the efficiency.
26. Define an explicit formula for the nth Fibonacci number.
27. Define a recursive algorithm to compute the nth Fibonacci number and analyze its efficiency.
28. What is Exhaustive Search?
29. What is Traveling Salesmen Problem (TSP)? Explain with e.g.
30. Give a Brute Force solution to the TSP. What is the efficiency of the algorithm?
31. What is an Assignment Problem? Explain with an e.g.
32. Give a Brute Force solution to the Assignment Problem. What is the efficiency of the algorithm?
33. Explain Divide and Conquer technique and give the general divide and conquer recurrence.

34. Define: a)Eventually non-decreasing function b)Smooth function c)Smoothness rule d)Masters theorem
35. Explain the Merge Sort algorithm with an e.g. and also draw the tree structure of the recursive calls made.
36. Analyze the efficiency of Merge sort algorithm.
37. Explain the Quick Sort algorithm with an example and also draw the tree structure of the recursive calls made.
38. Analyze the efficiency of Quick sort algorithm.
39. Give the Binary Search algorithm and analyze the efficiency.
40. Give an algorithm to find the height of a Binary tree and analyze the efficiency.
41. Give an algorithm each to traverse the Binary tree in Inorder, Preorder and Postorder.
42. Explain how do you multiply two large integers and analyze the efficiency of the algorithm. Give an e.g.
43. Explain the Stassen's Matrix multiplication with an e.g. and analyze the efficiency.
44. Explain the concept of Decrease and Conquer technique and explain its three major variations.
45. Give the Insertion Sort algorithm and analyze the efficiency.
46. Explain DFS and BFS with an e.g. and analyze the efficiency.
47. Give two solutions to sort the vertices of a directed graph topologically.
48. Discuss the different methods of generating Permutations.
49. Discuss the different methods of generating Subsets.
50. What is Heap? What are the different types of heaps?
51. Explain how do you construct heap?
52. Explain the concept of Dynamic programming with an e.g.
53. What is Transitive closure? Explain how do you find out the Transitive closure with an e.g.
54. Give the Warshall's algorithm and analyze the efficiency.
55. Explain how do you solve the All-Pairs-Shortest-Paths problem with an e.g.
56. Give the Floyd's algorithm and analyze the efficiency.
57. What is Knapsack problem? Give the solution to solve it using dynamic programming technique.
58. What are Memory functions? What are the advantages of using memory functions?
59. Give an algorithm to solve the knapsack problem.
60. Explain the concept of Greedy technique.
61. Explain Prim's algorithm with e.g.
62. Prove that Prim's algorithm always yields a minimum spanning tree.
63. Explain Kruskal's algorithm with an e.g.
64. Explain Dijkstra's algorithm with an e.g.
65. What are Huffman trees? Explain how to construct a Huffman trees with an e.g.
66. Explain the concept of Backtracking with an e.g.
67. What is state space tree? Explain how to construct a state space tree?
68. What is n-Queen's problem? Generate the state space tree for n=4.
69. Explain the subset sum problem with an e.g.
70. What are Decision Trees? Explain.

71. Define P, NP, and NP-Complete problems.
72. Explain the Branch and Bound technique with an e.g.
73. What are the steps involved in quick sort?
74. What is the principle used in the quick sort?
75. What are the advantages and disadvantages of quick sort?
76. What are the steps involved in merge sort?
77. What is divide, conquer, combine?
78. xplain the concept of topological ordering?
79. What are the other ordering techniques that you know?
80. How topological ordering is different from other ordering techniques?
81. What is transitive closure?
82. Time complexity of warshall's algorithm?
83. Define knapsack problem?
84. What is dynamic programming?
85. hat is single-source shortest path problem?
86. What is the time complexity of dijkstra's algorithm?
87. What is the purpose of kruskal's algorithm?
88. How is kruskal's algorithm different from prims?
89. What is BACK TRACKING?
90. What is branch and bound?
91. What is the main idea behind solving the TSP?
92. Do you know any other methodology for implementing a solution to this problem?
93. What does the term optimal solution of a given problem mean?
94. What is a spanning tree?
95. What is a minimum spanning tree?
96. Applications of spanning tree?