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 USING JAVA LABORATORY MANUAL

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 using Java Laboratory(21CG42)**

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.

DAYANANDA SAGAR COLLEGE OF ENGINEERING
(An Autonomous Institution Affiliated to VTU, Belagavi)
DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

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.

Experiment No.	Contents of the experiment	Hours	COs
1.	Implement Binary search and Linear search and determine the time required to search an element. Repeat the experiment for different values of N and plot a graph of the time taken versus N.	2	CO1
2.	Sort a given set of integer 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
3.	Sort a given set of N integer elements using Insertion Sort technique and compute its time taken.	2	CO2
4.	Sort a given set of N integer elements using Quick Sort technique and compute its time taken.	2	CO2
5.	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
6.	Write program to a. Print all the nodes reachable from a given starting node in a digraph using BFS method. **b.** Print all the nodes reachable from a given starting node in a digraph using DFS method.	2	CO2
7.	Write a program to implement Horspool's algorithm for String Matching.	2	CO3
8.	Sort a given set of N integer elements using Heap Sort technique and compute its time taken.	2	CO3
9.	Implement in Java, the 0/1 Knapsack problem using Dynamic Programming method.	2	CO4

10.	From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.	2	CO4
11.	Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal'salgorithm. Use Union-Find algorithms in your program.	2	CO4
12.	Find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.	2	CO4
13.	Write Java programs to Implement All-Pairs Shortest Paths problem using Floyd's algorithm. Warshalls Algorithm.	2	CO4
14.	Design and implement a program to find a subset of a given set S = $\{S_1, S_2,.....,S_n\}$ 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}. Display a suitable message, if the given problem instance doesn't have a solution.	2	CO5
15.	Implement “N-Queens Problem” using Backtracking	2	CO5

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. **Implement Binary search and Linear search and determine the time required to search an element. Repeat the experiment for different values of N and plot a graph of the time taken versus N.**

```
import java.util.*;
class Search{

   public static int [] randomTester(int max, int min, int min_count, int max_count){
      int len = (int)(Math.random() * max_count) + min_count;
      int [] arr = new int[len];
      int i=0;
      while(i<len){
         int value=(int)(Math.random() * max) + min;
         arr[i]=value;
         i++;
      }
      return arr;
   }

   public static int linear(int [] arr, int key) {
      for (int i=0; i<arr.length; i++) {
         if (arr[i]==key)
            return i;

      }
      return -1;
   }

   public static int binary(int [] arr, int key) {
      int low=0;
      int high=arr.length-1;

      while(low<=high){
         int mid=(low+high)/2;

         if(arr[mid]==key)
            return mid;
         else if(arr[mid]>key)
            high=mid-1;
         else if(arr[mid]<key)
            low=mid+1;
      }

      return -1;
   }

   public static void main(String[] args) {
      int[] arr = randomTester(1000, 1, 10, 100);
```

```
        Arrays.sort(arr);

        for(int i=0; i<5; i++){
            int index = (int)(Math.random() * arr.length);

            long startTime = System.nanoTime();
            int result = linear(arr, arr[index]);
            long time_taken = System.nanoTime() - startTime;
            System.out.println("Linear Search - Result: "+ result +" | Time taken: "+
time_taken +" ns");

            startTime = System.nanoTime();
            result = binary(arr, arr[index]);
            time_taken = System.nanoTime() - startTime;
            System.out.println("Binary Search - Result: "+ result +" | Time taken: "+
time_taken +" ns");
        }
    }
}
```

Output:

```
Linear Search - Result: 1 | Time taken: 3105 ns
Binary Search - Result: 1 | Time taken: 3591 ns
```

2. Sort a given set of integer 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.

```
public class SelectionSort {
   public static int [] randomTester(int max, int min, int min_count, int max_count){
      int len = (int)(Math.random() * max_count) + min_count;
      int [] arr = new int[len];
      int i=0;
      while(i<len){
         int value=(int)(Math.random() * max) + min;
         arr[i]=value;
         i++;
      }
      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");
   }
   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);
   }
}
```

Output:
SelectionSort - Time taken: 49768 ns | Length – 60

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

```
public class InsertionSortExample {
   public static void insertionSort(int array[]) {
      int n = array.length;
      for (int j = 1; j < n; j++) {
         int key = array[j];
         int i = j-1;
         while ( (i > -1) && ( array [i] > key ) ) {
            array [i+1] = array [i];
            i--;
         }
         array[i+1] = key;
      }
   }

   public static void main(String a[]){
      int[] arr1 = {9,14,3,2,43,11,58,22};
      System.out.println("Before Insertion Sort");
      for(int i:arr1){
         System.out.print(i+" ");
      }
      System.out.println();

         long startTime = System.nanoTime();
       insertionSort(arr1);//sorting array using insertion sort
long time_taken = System.nanoTime() - startTime;

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

```
        System.out.println("After Insertion Sort");
        for(int i:arr1){
            System.out.print(i+" ");
        }
    }
}
```

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

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

public class QuickSortComplexity1 {
  static final int MAX = 200000;
  static int[] a = new int[MAX];
  public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        System.out.print("Enter Max array size: ");
        int n = input.nextInt();
        Random random = new Random();
        System.out.println("Enter the array elements: ");
        for (int i = 0; i < n; i++)
              // a[i] = input.nextInt();   // for keyboard entry
              a[i] = random.nextInt(10000); // generate
    // random numbers ñ uniform distribution

        // a = Arrays.copyOf(a, n); // keep only non zero elements
        // Arrays.sort(a); // for worst-case time complexity

        System.out.println("Input Array:");
        for (int i = 0; i < n; i++)
              System.out.print(a[i] + " ");
        // set start time
        long startTime = System.nanoTime();
        QuickSortAlgorithm(0, n - 1);
        long stopTime = System.nanoTime();
        long elapsedTime = stopTime - startTime;
        System.out.println("\nSorted Array:");
        for (int i = 0; i < n; i++)
              System.out.print(a[i] + " ");
        System.out.println();
        System.out.println("Time Complexity in ms for
            n=" + n + " is: " + (double) elapsedTime / 1000000);
  }

  public static void QuickSortAlgorithm(int p, int r) {
        int i, j, temp, pivot;
        if (p < r) {
              i = p;
              j = r + 1;
              pivot = a[p]; // mark first element as pivot
              while (true) {
```

```
                        i++;
                        while (a[i] < pivot && i < r)
                                i++;
                        j--;
                        while (a[j] > pivot)
                                j--;
                        if (i < j) {
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        } else
                                break; // partition is over
                }
                a[p] = a[j];
                a[j] = pivot;
                QuickSortAlgorithm(p, j - 1);
                QuickSortAlgorithm(j + 1, r);
        }
  }
}
```

Output

```
Enter Max array size: 20
Enter the array elements:
Input Array:
326 719 983 701 490 230 595 474 341 75 916 173 324 852 728 434 758 445 303 566
Sorted Array:
75 173 230 303 324 326 341 434 445 474 490 566 595 701 719 728 758 852 916 983
Time Complexity in ms for n=20 is: 0.023225

Enter Max array size: 20000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=20000 is: 4.953809

Enter Max array size: 30000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=30000 is: 7.141865

Enter Max array size: 40000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=40000 is: 8.698231

Enter Max array size: 50000
```

Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=50000 is: 9.103897

Enter Max array size: 60000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=60000 is: 12.380137

Enter Max array size: 70000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=70000 is: 24.719828

Enter Max array size: 80000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=80000 is: 21.150887

Enter Max array size: 90000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=90000 is: 35.894418

Enter Max array size: 100000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=100000 is: 31.430762

Enter Max array size: 200000
Enter the array elements:
Input Array:
Sorted Array:
Time Complexity in ms for n=200000 is: 47.498161

Plot Graph: time taken versus n on graph sheet

Time Complexity Analysis:

Quick Sort Algorithm
Average performance O(n log n)

5. 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;
import java.util.Scanner;

public class MergeSort{
        static final int MAX = 200000;
        static int[] a = new int[MAX];

        public static void main(String[] args) {
                Scanner input = new Scanner(System.in);
                System.out.print("Enter Max array size: ");
                int n = input.nextInt();
                Random random = new Random();
                System.out.println("Enter the array elements: ");
                for (int i = 0; i < n; i++)
                {
                //        a[i] = input.nextInt(); // for keyboard entry
                 a[i] = random.nextInt(100000);
              System.out.print(a[i] + " ");
                }
                long startTime = System.nanoTime();
                MergeSortAlgorithm(0, n - 1);
                long stopTime = System.nanoTime();
                long elapsedTime = stopTime - startTime;
                System.out.println("Time Complexity (ms) for n = " +
n + " is : " + (double) elapsedTime / 1000000);
                System.out.println("Sorted Array (Merge Sort):");
                for (int i = 0; i < n; i++)
                        System.out.print(a[i] + " ");
                input.close();
        }

        public static void MergeSortAlgorithm(int low, int high) {
                int mid;
                if (low < high) {
                        mid = (low + high) / 2;
                        MergeSortAlgorithm(low, mid);
                        MergeSortAlgorithm(mid + 1, high);
                        Merge(low, mid, high);
                }
        }

        public static void Merge(int low, int mid, int high) {
                int[] b = new int[MAX];
                int i, h, j, k;
                h = i = low;
```

```
                j = mid + 1;
                while ((h <= mid) && (j <= high))
                        if (a[h] < a[j])
                                b[i++] = a[h++];
                        else
                                b[i++] = a[j++];

                if (h > mid)
                        for (k = j; k <= high; k++)
                                b[i++] = a[k];
                else
                        for (k = h; k <= mid; k++)
                                b[i++] = a[k];

                for (k = low; k <= high; k++)
                        a[k] = b[k];
        }
}
```

Output

```
Enter Max array size: 5
Enter the array elements:
856 604 528 287 321 Time Complexity (ms) for n = 5 is : 0.090071
Sorted Array (Merge Sort):
287 321 528 604 856

Enter Max array size: 10000
Enter the array elements:
Time Complexity (ms) for n = 10000 is : 1194.135767
Sorted Array (Merge Sort):

Enter Max array size: 20000
Enter the array elements:
Time Complexity (ms) for n = 20000 is : 2040.96632
Sorted Array (Merge Sort):

Enter Max array size: 30000
Enter the array elements:
Time Complexity (ms) for n = 30000 is : 3098.642188
Sorted Array (Merge Sort):

Enter Max array size: 40000
Enter the array elements:
Time Complexity (ms) for n = 40000 is : 3914.650313
Sorted Array (Merge Sort):

Enter Max array size: 50000
Enter the array elements:
Time Complexity (ms) for n = 50000 is : 4700.729745
```

Sorted Array (Merge Sort):

Enter Max array size: 60000
Enter the array elements:
Time Complexity (ms) for n = 60000 is : 5457.318457
Sorted Array (Merge Sort):

Enter Max array size: 70000
Enter the array elements:
Time Complexity (ms) for n = 70000 is : 6630.648568
Sorted Array (Merge Sort):

Enter Max array size: 80000
Enter the array elements:
Time Complexity (ms) for n = 80000 is : 7419.150889
Sorted Array (Merge Sort):

Enter Max array size: 90000
Enter the array elements:
Time Complexity (ms) for n = 90000 is : 8119.913672
Sorted Array (Merge Sort):

Enter Max array size: 100000
Enter the array elements:
Time Complexity (ms) for n = 100000 is : 8865.6302
Sorted Array (Merge Sort):

Plot Graph: time taken versus n on graph sheet

Time Complexity Analysis:

Merge Sort Algorithm

Average performance O(n log n)

6. Write program to do the following:

a. Print all the nodes reachable from a given starting node in a digraph using BFS method.

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

class Graph {

private int V;

private LinkedList<Integer> adj[];

Graph(int v)
{
V = v;
adj = new LinkedList[v];
for (int i = 0; i < v; ++i)
adj[i] = new LinkedList();
}

// Function to add an edge into the graph
void addEdge(int v, int w) { adj[v].add(w); }

void BFS(int s)
{
boolean visited[] = new boolean[V];

// Create a queue for BFS
LinkedList<Integer> queue
= new LinkedList<Integer>();

// Mark the current node as visited and enqueue it
visited[s] = true;
queue.add(s);

while (queue.size() != 0) {

// Dequeue a vertex from queue and print it
s = queue.poll();
System.out.print(s + " ");

Iterator<Integer> i = adj[s].listIterator();
while (i.hasNext()) {
int n = i.next();
if (!visited[n]) {
visited[n] = true;
queue.add(n);
}
```

```
}
}
}


public static void main(String args[])
{
Graph g = new Graph(7);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(0, 3);
g.addEdge(1, 4);
g.addEdge(2, 5);
g.addEdge(3, 6);

System.out.println("Following is Breadth First Traversal "+ "(starting from vertex
0)");

g.BFS(0);}
}
```

OUTPUT:
Breadth First Traversal for the graph is: 0 1 2 3 4 5 6

b. Print all the nodes reachable from a given starting node in a digraph using DFS method.

```
import java.util.*;

class Graph {
  private LinkedList<Integer> adjLists[];
  private boolean visited[];


  Graph(int vertices) {
    adjLists = new LinkedList[vertices];
    visited = new boolean[vertices];

    for (int i = 0; i < vertices; i++)
      adjLists[i] = new LinkedList<Integer>();
  }


  void addEdge(int src, int dest) {
    adjLists[src].add(dest);
  }

  // DFS algorithm
  void DFS(int vertex) {
    visited[vertex] = true;
    System.out.print(vertex + " ");

    Iterator<Integer> ite = adjLists[vertex].listIterator();
    while (ite.hasNext()) {
      int adj = ite.next();
      if (!visited[adj])
        DFS(adj);
    }
  }

  public static void main(String args[]) {
   Graph g = new Graph(7);
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(0, 3);
g.addEdge(1, 4);
g.addEdge(2, 5);
g.addEdge(3, 6);

    System.out.println("Following is Depth First Traversal");

    g.DFS(0);
```

```
  }
}
```

OUTPUT:
Following is Depth First Traversal 0 1 4 2 5 3 6

7. 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; // Pattern found at index i + 1
            } else {
                i += Math.max(1, patternLength - 1 - j + shiftTable[text.charAt(i)]);
            }
        }

        return -1; // Pattern not found in the text
    }
}
```

Pattern found at index 16

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

```
import java.util.Scanner;

public class HeapSort{
     private static int N;
   public static void sort(int arr[]){
              heapMethod(arr);
      for (int i = N; i > 0; i--){
         swap(arr,0, i);
         N = N-1;
         heap(arr, 0);
      }
   }
   public static void heapMethod(int arr[]){
      N = arr.length-1;
      for (int i = N/2; i >= 0; i--)
         heap(arr, i);
   }
   public static void heap(int arr[], int i){
      int left = 2*i ;
      int right = 2*i + 1;
      int max = i;
      if (left <= N && arr[left] > arr[i])
         max = left;
              if (right <= N && arr[right] > arr[max])
         max = right;
      if (max != i){
         swap(arr, i, max);
         heap(arr, max);
      }
   }
   public static void swap(int arr[], int i, int j){
      int tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
   }
   public static void main(String[] args) {
      Scanner in = new Scanner( System.in );
      int n;
      System.out.println("Enter the number of elements to be sorted:");
      n = in.nextInt();
      int arr[] = new int[ n ];
      System.out.println("Enter "+ n +" integer elements");
      for (int i = 0; i < n; i++)
         arr[i] = in.nextInt();
      sort(arr);
      System.out.println("After sorting ");
```

```
        for (int i = 0; i < n; i++)
            System.out.println(arr[i]+" ");
        System.out.println();
    }
}
```

Output is:

```
Enter the number of elements to be sorted:
6
Enter 6 integer elements
99
54
67
32
1
78
After sorting
1
32
54
67
78
99
```

9. Implement in Java, the 0/1 Knapsack problem using Dynamic Programming method

```
import java.util.Scanner;

public class KnapsackDP {
	static final int MAX = 20; // max. no. of objects
	static int w[];  // weights 0 to n-1
	static int p[];  // profits 0 to n-1
	static int n;		// no. of objects
	static int M;		// capacity of Knapsack
	static int V[][];	// DP solution process - table
	static int Keep[][]; // to get objects in optimal solution

	public static void main(String args[]) {
		w = new int[MAX];
		p = new int[MAX];
		V = new int [MAX][MAX];
		Keep = new int[MAX][MAX];
		int optsoln;
		ReadObjects();
		for (int i = 0; i <= M; i++)
			V[0][i] = 0;
		for (int i = 0; i <= n; i++)
			V[i][0] = 0;
		optsoln = Knapsack();
		System.out.println("Optimal solution = " + optsoln);
	}

	static int Knapsack() {
		int r; // remaining Knapsack capacity
		for (int i = 1; i <= n; i++)
			for (int j = 0; j <= M; j++)
				if ((w[i] <= j) && (p[i] + V[i - 1][j - w[i]] > V[i - 1][j]))
				{
					V[i][j] = p[i] + V[i - 1][j - w[i]];
					Keep[i][j] = 1;
				} else {
					V[i][j] = V[i - 1][j];
					Keep[i][j] = 0;
				}

		// Find the objects included in the Knapsack
		r = M;
		System.out.println("Items = ");
		for (int i = n; i > 0; i--) // start from Keep[n,M]
			if (Keep[i][r] == 1) {
```

```
                                System.out.println(i + " ");
                                r = r - w[i];
                        }
                System.out.println();
                return V[n][M];
        }

        static void ReadObjects() {
                Scanner scanner = new Scanner(System.in);
                System.out.println("Knapsack Problem - Dynamic Programming
Solution: ");
                System.out.println("Enter the max capacity of knapsack: ");
                M = scanner.nextInt();
                System.out.println("Enter number of objects:  ");
                n = scanner.nextInt();
                System.out.println("Enter Weights: ");
                for (int i = 1; i <= n; i++)
                        w[i] = scanner.nextInt();
                System.out.println("Enter Profits: ");
                for (int i = 1; i <= n; i++)
                        p[i] = scanner.nextInt();
                scanner.close();
        }
}
```

Output

```
Knapsack Problem - Dynamic Programming Solution:
Enter the max capacity of knapsack:
5
Enter number of objects:
4
Enter Weights:
1
2
2
1
Enter Profits:
15
20
10
12

Optimal solution = 47
```

10. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm. Write the program in Java.

```
import java.util.*;

public class DijkstrasClass {

        final static int MAX = 20;
        final static int infinity = 9999;
        static int n;               // No. of vertices of G
        static int a[][]; // Cost matrix
        static Scanner scan = new Scanner(System.in);

        public static void main(String[] args) {
                ReadMatrix();
                int s = 0;                  // starting vertex
                System.out.println("Enter starting vertex: ");
                s = scan.nextInt();
                Dijkstras(s);    // find shortest path
        }

        static void ReadMatrix() {
                a = new int[MAX][MAX];
                System.out.println("Enter the number of vertices:");
                n = scan.nextInt();
                System.out.println("Enter the cost adjacency matrix:");
                for (int i = 1; i <= n; i++)
                        for (int j = 1; j <= n; j++)
                                a[i][j] = scan.nextInt();
        }

        static void Dijkstras(int s) {
                int S[] = new int[MAX];
                int d[] = new int[MAX];
                int u, v;
                int i;
                for (i = 1; i <= n; i++) {
                        S[i] = 0;
                        d[i] = a[s][i];
                }
                S[s] = 1;
                d[s] = 1;
                i = 2;
                while (i <= n) {
                        u = Extract_Min(S, d);
                        S[u] = 1;
                        i++;
                        for (v = 1; v <= n; v++) {
```

```
                                if (((d[u] + a[u][v] < d[v]) && (S[v] == 0)))
                                        d[v] = d[u] + a[u][v];
                        }
                }
                for (i = 1; i <= n; i++)
                        if (i != s)
                                System.out.println(i + ":" + d[i]);
        }

        static int Extract_Min(int S[], int d[]) {
                int i, j = 1, min;
                min = infinity;
                for (i = 1; i <= n; i++) {
                        if ((d[i] < min) && (S[i] == 0)) {
                                min = d[i];
                                j = i;
                        }
                }
                return (j);
        }
}
```

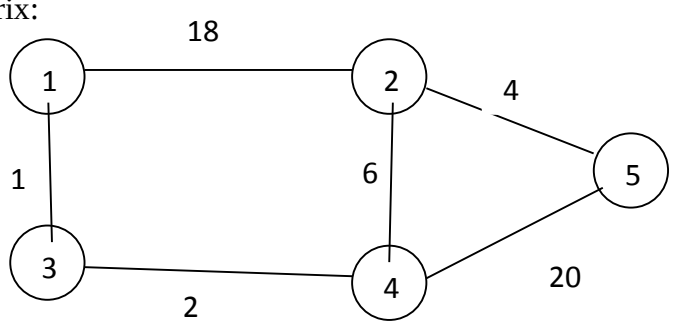
Output

Enter the number of vertices:

5

Enter the cost adjacency matrix:

0 18 1 9999 9999
18 0 9999 6 4
1 9999 0 2 9999
9999 6 2 0 20
9999 4 9999 20 0



Enter starting vertex:

1

2:9

3:1

4:3

5:13

11. Find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal'salgorithm. Use Union-Find algorithms in your program.

```
import java.util.Scanner;

public class KruskalsClass {

        final static int MAX = 20;
        static int n; // No. of vertices of G
        static int cost[][]; // Cost matrix
        static Scanner scan = new Scanner(System.in);

        public static void main(String[] args) {
                ReadMatrix();
                Kruskals();
        }

        static void ReadMatrix() {

                int i, j;
                cost = new int[MAX][MAX];

                System.out.println("Implementation of Kruskal's algorithm");
                System.out.println("Enter the no. of vertices");
                n = scan.nextInt();

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

        static void Kruskals() {

                int a = 0, b = 0, u = 0, v = 0, i, j, ne = 1, min, mincost = 0;
```

```
                System.out.println("The edges of Minimum Cost Spanning Tree are");
                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 (u != v)
                        {
                                uni(u, v);
                        System.out.println(ne++ + "edge (" + a + "," + b + ") =" +
min);
                                mincost += min;
                        }
                        cost[a][b] = cost[b][a] = 999;
                }
                System.out.println("Minimum cost :" + mincost);
        }

        static int find(int i) {
                int parent[] = new int[9];
                while (parent[i] == 1)
                        i = parent[i];
                return i;
        }

        static void uni(int i, int j) {
                int parent[] = new int[9];
                parent[j] = i;
        }
}
```

Output
Enter the no. of vertices
4
Enter the cost adjacency matrix
0 5 999 20
5 0 10 999
999 10 0 15
20 999 15 0
The edges of Minimum Cost Spanning Tree are
1edge (1,2) =5
2edge (2,3) =10
3edge (3,4) =15
Minimum cost :30

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

```
import java.util.Scanner;

public class PrimsClass {

	final static int MAX = 20;
	static int n; // No. of vertices of G
	static int cost[][]; // Cost matrix
	static Scanner scan = new Scanner(System.in);

	public static void main(String[] args) {
		ReadMatrix();
		Prims();
	}

	static void ReadMatrix() {
		int i, j;
		cost = new int[MAX][MAX];

		System.out.println("\n Enter the number of nodes:");
		n = scan.nextInt();
		System.out.println("\n Enter the cost matrix:\n");
		for (i = 1; i <= n; i++)
			for (j = 1; j <= n; j++) {
				cost[i][j] = scan.nextInt();
				if (cost[i][j] == 0)
					cost[i][j] = 999;
			}
	}

	static void Prims() {

		int visited[] = new int[10];
		int ne = 1, i, j, min, a = 0, b = 0, u = 0, v = 0;
		int mincost = 0;

		visited[1] = 1;
```

```
                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.println("Edge" + ne++ + ":(" + a + "," + b + ")" + "cost:" + min);
                                        mincost += min;
                                        visited[b] = 1;
                                }
                                cost[a][b] = cost[b][a] = 999;
                        }
                        System.out.println("\n Minimun cost" + mincost);
                }

        }
```

Output

```
Enter the number of nodes:

4

 Enter the cost matrix:

0 5 999 20
5 0 10 999
999 10 0 15
20 999 15 0

Edge1:(1,2)cost:5
Edge2:(2,3)cost:10
Edge3:(3,4)cost:15

 Minimun cost 30
```

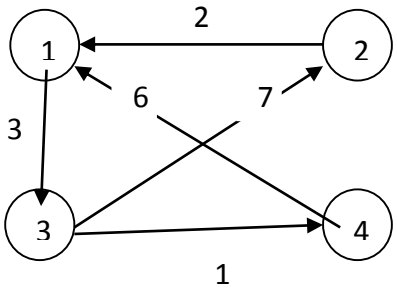
13. Write Java programs to Implement All-Pairs Shortest Paths problem using
a. Floyd's algorithm.

```
import java.util.Scanner;
public class FloydsClass {
static final int MAX = 20;      // max. size of cost matrix
static int a[][];               // cost matrix
static int n;                           // actual matrix size

public static void main(String args[]) {
        a = new int[MAX][MAX];
        ReadMatrix();
        Floyds();                               // find all pairs shortest path
        PrintMatrix();
}

static void ReadMatrix() {
        System.out.println("Enter the number of vertices\n");
        Scanner scanner = new Scanner(System.in);
        n = scanner.nextInt();
        System.out.println("Enter the Cost Matrix (999 for infinity) \n");
        for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                        a[i][j] = scanner.nextInt();
                }
        }
        scanner.close();
}

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

static void PrintMatrix() {
        System.out.println("The All Pair Shortest Path Matrix is:\n");
        for(int i=1; i<=n; i++)
        {
                for(int j=1; j<=n; j++)
                        System.out.print(a[i][j] + "\t");
                System.out.println("\n");
        }
        }
}
```

Output

Enter the number of vertices: 4
Enter the Cost Matrix (999 for infinity)

0	999	3	999
2	0	999	999
999	7	0	1
6	999	999	0



The All Pair Shortest Path Matrix is:

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

b. Warshalls Algorithm.

```
import java.util.Scanner;

public class Warshall
{
   private int V;
   private boolean[][] tc;

   public void getTC(int[][] graph)
   {
      this.V = graph.length;
      tc = new boolean[V][V];
      for (int i = 0; i < V; i++)
      {
         for (int j = 0; j < V; j++)
            if (graph[i][j] != 0)
               tc[i][j] = true;
         tc[i][i] = true;
      }
      for (int i = 0; i < V; i++)
      {
         for (int j = 0; j < V; j++)
         {
            if (tc[j][i])
               for (int k = 0; k < V; k++)
                  if (tc[j][i] && tc[i][k])
                     tc[j][k] = true;
         }
      }
   }
   public void displayTC()
   {
      System.out.println("\nTransitive closure :\n");
      System.out.print(" ");
      for (int v = 0; v < V; v++)
        System.out.print("   " + v );
      System.out.println();
      for (int v = 0; v < V; v++)
      {
         System.out.print(v +" ");
         for (int w = 0; w < V; w++)
         {
            if (tc[v][w])
               System.out.print(" 1 ");
            else
               System.out.print(" 0 ");
         }
         System.out.println();
```

```
    }
  }

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

    Warshall w = new Warshall();


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

    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();

    w.getTC(graph);
    w.displayTC();
  }
}
```

Enter number of vertices

4

Enter matrix

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

Transitive closure :

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

14. Design and implement in Java to find a subset of a given set S = {S_1, S_2,.....,S_n} 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}. Display a suitable message, if the given problem instance doesn't have a solution.

```
import java.util.Scanner;
public class P11 {
static int d,flag=0;
static int []S = new int [10];
static int []x = new int [10];
static void sumofsub(int s,int k,int r)
{
int i;
x[k]=1;
if((s+S[k]) == d)
{
flag=1;
for(i=1;i<=k;i++)
if(x[i]==1)
System.out.print(S[i]+"\t");
System.out.println();
}
else
if(s+S[k]+S[k+1]<=d)
sumofsub(s+S[k],k+1,r-S[k]);
if((s+r-S[k]>=d) && (s+S[k+1]<=d))
{
x[k]=0;
sumofsub(s,k+1,r-S[k]);
}
}
public static void main(String [] args)
{
int i,n,sum=0;
Scanner read= new Scanner(System.in);
System.out.println("enter the no. of elements in the set");
n=read.nextInt();
System.out.println("enter the set in increasing order");
for(i=1;i<=n;i++)
S[i]=read.nextInt();
System.out.println("enter the max subset value");
d=read.nextInt();
for(i=1;i<=n;i++)
sum=sum+S[i];
if(sum<d||S[1]>d)
System.out.println("no subset possible");
else
```

```
{
System.out.println("The possible subsets are");
sumofsub(0,1,sum);
if(flag==0)
System.out.println("no subset possible ");
}
}
}
```

OUTPUT:

Output

```
Enter number of elements:
5
Enter the set in increasing order:
2
3
4
5
6
Enter the max. subset value(d): 9

2  3  4
3  6
4  5
```

15. Implement "N-Queens Problem" using Backtracking

```
public class Main {
   static final int N = 4;

  // print the final solution matrix
   static void printSolution(int board[][])
   {
      for (int i = 0; i < N; i++) {
         for (int j = 0; j < N; j++)
            System.out.print(" " + board[i][j]
                  + " ");
         System.out.println();
      }
   }

   // function to check whether the position is safe or not
   static boolean isSafe(int board[][], int row, int col)
   {
      int i, j;
      for (i = 0; i < col; i++)
         if (board[row][i] == 1)
            return false;


      for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
         if (board[i][j] == 1)
            return false;

      for (i = row, j = col; j >= 0 && i < N; i++, j--)
         if (board[i][j] == 1)
            return false;

      return true;
   }

   // The function that solves the problem using backtracking
   public static boolean solveNQueen(int board[][], int col)
   {
      if (col >= N)
         return true;

      for (int i = 0; i < N; i++) {
         //if it is safe to place the queen at position i,col -> place it
         if (isSafe(board, i, col)) {
            board[i][col] = 1;

            if (solveNQueen(board, col + 1))
```

```
                return true;

            //backtrack if the above condition is false
            board[i][col] = 0;
         }
      }
      return false;
   }

   public static void main(String args[])
   {
      int board[][] = { { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 },
            { 0, 0, 0, 0 } };

      if (!solveNQueen(board, 0)) {
         System.out.print("Solution does not exist");
         return;
      }

      printSolution(board);

   }
}
```

Output :

```
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?