K.S. RANGASAMY COLLEGE OF TECHNOLOGY

(Autonomous Institution)

TIRUCHENGODE – 637 215

LAB MANUAL

60 CB 4P2 – Design and Analysis of Algorithms Lab

B. Tech - Computer Science and Business Systems

Second Year

Fouth Semester

BATCH: 2023-2027

DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

K.S. RANGASAMY COLLEGE OF TECHNOLOGY

(An Autonomous institution, affiliated to Anna University Chennai and Approved by AICTE, New Delhi)

TIRUCHENGODE - 637 215

K.S. RANGASAMY COLLEGE OF TECHNOLOGY

(Autonomous Institution)

TIRUCHENGODE – 637 215

CERTIFICATE

Register Number: **2303737724422044**

Certified that this is the bonafide record of work done by Selvi **KAVIYA V** of the Fourth Semester **B.Tech Computer Science and Business Systems** branch during the academic year **2024-2025** in 60 CB 4P2 – Design and Analysis of Algorithms Lab.

Staff in-charge

Head of the Department

Submitted for the End Semester Practical Examination on …….

Internal Examiner I

Internal Examiner II

LIST OF EXPERIMENTS

1. Implementation of Number Theory and Bit Manipulation Techniques

2. Implementation of Recursion and Backtracking Techniques

3. Implementation of Tree Algorithms

4. Implementation of Graph Algorithms

5. Implementation of Greedy Techniques

6. Implementation of Dynamic Programming Techniques

7. Implementation of Segment Trees

8. Implementation of Tries

9. Implementation of Game Theory

10. Implementation of NP Complete Problems

CONTENT

K.S. RANGASAMY COLLEGE OF TECHNOLOGY, TIRUCHENGODE - 637215 DEPARTMENT OF COMPUTER SCIENCE AND BUSINESS SYSTEMS

VISION

- To produce skilled professionals to the dynamic needs of the industry with innovative computer science Professionals associate with managerial services

MISSION

- To promote student's ability through innovative teaching in computer science to compete globally as an engineer
- To inculcate management skills to meet the industry standards and augment human values and life skills to serve the society

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Graduates will provide effective solutions for software and hardware industries by applying the Concepts of basic science and engineering fundamentals.

PEO2: Graduates will be professionally competent and successful in their career through life-long Learning.

PEO3: Graduates will contribute individually or as member of a team in handling projects and demonstrate Social responsibility and professional ethics.

PROGRAMME OUTCOMES (POs)

Engineering Graduates will be able to:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO3: Design /development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solution sin societal and environmental contexts, and demonstrate the knowledge of, and need for unstainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings. Graduates will contribute individually or as member of a team in handling projects and demonstrate social responsibility and professional ethics.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one ‘s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAMME SPECIFIC OUTCOMES (PSOs):

Engineering Graduates will be able to:

PSO1: Apply analytical and technical skill of computer science to provide justifiable solution for real world applications

PSO2: Analyze various managerial skills and business disciplines to improve the industry growth and development

EX.NO: 01 (A)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: Find Prime Factorisation of a Number using Sieve.

AIM:

To find the prime factorization of a number efficiently using the Sieve method.

ALGORITHM:

Step1: Create an empty list to store prime factors.

Step2: Loop i from 2 to √N and check if i divides N.

Step3: While i divides N, add i to the list and divide N by i.

Step4: After the loop, if N is not 1, add N itself to the list (it's prime).

Step5: Return the list of prime factors.

CODING:

```
class Solution {

    static void sieve() {}

    static List<Integer> findPrimeFactors(int N) {

        List<Integer> l = new ArrayList();
        int[] arr =new int[N+1];
        for(int i=2; i*i<=N; i++){
            while(N%i==0){
                l.add(i);
                N=N/i;
            }
        }
        if(N!=1){
            l.add(N);
        }
        return l;

    }
}
```

INPUT:

N = 2310

OUTPUT:

2 3 5 7 11

EX.NO: 01 (B)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: Find GCD and LCM of Two Numbers Using Euclid's Algorithm.

AIM:
Write a Java program to find GDC and LCM of Two numbers using Euclid's algorithm.

ALGORITHM:
Step1: Input two numbers A and B.
Step2: Store original A and B for LCM calculation.
Step3: Find GCD using Euclidean Algorithm: while B ≠ 0, set A = B and B = A % B.
Step4: GCD is now A; calculate LCM = (original A × original B) / GCD.
Step5: Output GCD and LCM.

CODING:

```
class Solution {
    public static int[] lcmAndGcd(int a, int b) {
        int gcd = gcd(a,b);
        int lcm=(a*b)/gcd;
        return new int[] {lcm,gcd};
    }
     private static int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
```

INPUT :
a = 12, b = 18

OUTPUT :
[65, 6]

EX.NO: 01 (C)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: Write a program to perform modular arithmetic operations: addition, subtraction, multiplication, and modular inverse using the Extended Euclidean Algorithm.

AIM:
To implement a java program that performs basic modular arithmetic operations (addition, subtraction, multiplication) and calculates the modular inverse using Euclid's Extended Algorithm.

ALGORITHM:

Step1: Take inputs a, b, and m for the arithmetic operations.

Step2: Perform and output modular addition, subtraction,multiplication using the formulas(a + b) % m,(a - b)%m,(a * b)% m

Step3: For the modular inverse, use the Extended Euclidean Algorithm to find the GCD of a and m.

Step4: If GCD is 1, calculate and output the modular inverse (result[1] + m) % m.

Step5: Display the results of all operations.

CODING:

```
import java.util.Scanner;
public class ModularArithmetic {
    public static int modularAdd(int a, int b, int m) {
        return (a + b) % m;
    }
    public static int modularSubtract(int a, int b, int m) {
        return (a - b + m) % m;
    }
    public static int modularMultiply(int a, int b, int m) {
        return (a * b) % m;
    }
    public static int modularInverse(int a, int m) {
        int[] result = extendedGCD(a, m);
        int gcd = result[0];
        if (gcd != 1) {
            System.out.println("Inverse does not exist");
            return -1;
        } else {
            // Ensure the result is positive
            return (result[1] + m) % m;
        }
    }
    private static int[] extendedGCD(int a, int b) {
        if (b == 0) {
            return new int[] {a, 1, 0};
        }
        int[] temp = extendedGCD(b, a % b);
        int gcd = temp[0];
        int x = temp[2];
```

```
        int y = temp[1] - (a / b) * temp[2];
        return new int[] {gcd, x, y};
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter first number (a): ");
        int a = scanner.nextInt();
        System.out.println("Enter second number (b): ");
        int b = scanner.nextInt();
        System.out.println("Enter modulus (m): ");
        int m = scanner.nextInt();
        System.out.println("Modular Addition (a + b) % m = " + modularAdd(a, b, m));
        System.out.println("Modular Subtraction (a - b) % m = " + modularSubtract(a, b, m));
        System.out.println("Modular Multiplication (a * b) % m = " + modularMultiply(a, b, m));
        int inverse = modularInverse(a, m);
        if (inverse != -1) {
            System.out.println("Modular Inverse of a mod m = " + inverse);
        }
        scanner.close();
    }
}
```

INPUT:
Enter first number (a): 14
Enter second number (b):5
Enter modulus (m): 9
OUTPUT:
Modular Addition (a + b) % m = 1
Modular Subtraction (a - b) % m = 9
Modular Multiplication (a * b) % m = 2
Modular Inverse of a mod m = 4

EX.NO: 01 (D)	Implementation of Number Theory and Bit Manipulation Techniques
DATE:07.01.2025	

QUESTION: Given an integer n, return the number of prime numbers that are strictly less than n.

AIM:

To count the number of prime numbers less than a given integer n using the Sieve of Eratosthenes algorithm.

ALGORITHM:

Step1: If n <= 2, return 0 because there are no primes less than 2.

Step2: Create a boolean array isPrime where each index represents a number, initialized as true.

Step3: Implement the Sieve of Eratosthenes: For each prime i, mark all its multiples as false.

Step4: After marking, iterate through the isPrime array to count all true values.

Step5: Return the count of prime numbers.

CODING:

```
class Solution {
    public int countPrimes(int n) {
        boolean[] notPrime = new boolean[n];
        int count = 0;
        for (int i = 2; i < n; i++) {
            if (notPrime[i] == false) {
                count++;
                for (int j = 2; i*j < n; j++) {
                    notPrime[i*j] = true;
                }
            }
        }

        return count;
    }
}
```

INPUT: n = 20

OUTPUT: 8

EX.NO: 01 (E)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: Checks whether the i-th bit of a given integer is set (1) or not set (0) using bitwise operations

AIM:

To implement a Java program that checks whether the i-th bit (starting from the least significant bit) of a given integer is set (1) or not set (0) using bitwise operations.

ALGORITHM:

Step 1: Start the program and take two integer inputs from the user-the number n and the bit position i.
Step 2: Right-shift the number n by i positions using n >> i.
Step 3: Perform a bitwise AND operation between the result and 1 i.e., (n >> i) & 1.
Step 4: If the result is 1, then the i-th bit is set (1); otherwise, it is not set (0).
Step 5: Display the result to the user.
Step 6: End the program.

CODING:

```
import java.util.Scanner;
public class BitCheck {
    public static boolean isBitSet(int number, int i) {
        return ((number >> i) & 1) == 1;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();
        System.out.print("Enter the bit position to check (0 for LSB): ");
        int i = scanner.nextInt();
        if (isBitSet(number, i)) {
            System.out.println("The " + i + "-th bit is set (1).");
        } else {
            System.out.println("The " + i + "-th bit is not set (0).");
        }
        scanner.close();
    }
}
```

INPUT:

Enter an integer: 25
Enter the bit position to check (0 for LSB): 2

OUTPUT:

The 2-th bit is not set (0).

EX.NO: 01 (F)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: check whether a given number is odd or not using bitwise operation.

AIM:
To Write a Java program to check whether a given number is **odd** or **not** using bitwise operation.

ALGORITHM:
Step 1: Start the program.
Step 2: Take an integer input n from the user.
Step 3: Perform a bitwise AND operation between n and 1 using the expression n & 1.
Step 4: If the result is 1, then the number is odd.If the result is 0, then the number is even.
Step 5: Display the result to the user.
Step 6: End the program.

CODING:

```
import java.util.Scanner;
public class OddCheck {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = sc.nextInt();

        if ((num & 1) == 1) {
            System.out.println(num + " is an odd number.");
        } else {
            System.out.println(num + " is not an odd number.");
        }

        sc.close();
    }
}
```

INPUT:
Enter a number: 9
OUTPUT:
9 is an odd number.

EX.NO: 01 (G)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION : Given an integer n, return true if it is a power of two. Otherwise, return false. An integer n is a power of two, if there exists an integer x such that n == 2x.

AIM:
To write a Java program to check whether a given integer is a power of two using bitwise operations and return true if it is, otherwise return false.

ALGORITHM:
Step 1: Start the program.
Step 2: Take an integer input n from the user.
Step 3: Check if n is greater than 0.
Step 4: Perform bitwise operation using the expression n & (n - 1).
Step 5: If the result is 0, then n is a power of two; otherwise, it is not.
Step 6: Display the result and end the program.

CODING:

```
class Solution {
   public boolean isPowerOfTwo(int n) {

       return (n>0&&(n&(n-1))==0);}
}
```

INPUT:
n = 32
OUTPUT:
true

EX.NO: 01 (H)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION : Write a Java program to count the number of set bits (1s) in the binary representation of a given integer using bitwise operations.

AIM:

To write a Java program that counts the number of set bits (bits with value 1) in the binary representation of a given integer using bitwise operations.

ALGORITHM:

Step 1: Start the program.
Step 2: Take an integer input n from the user.
Step 3: Initialize a counter variable count to 0.
Step 4: Repeat the following steps while n is not equal to 0,Perform n & 1 to check the least significant bit, If it is 1, increment count, Right shift n by 1 using n = n >> 1.
Step 5: Display the value of count as the number of set bits.
Step 6: End the program.

CODING:

```
import java.util.Scanner;
public class SetBitsCounter {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int n = sc.nextInt();
        int count = 0;

        while (n != 0) {
            if ((n & 1) == 1) {
                count++;
            }
            n = n >> 1;
        }

        System.out.println("Number of set bits: " + count);
        sc.close();
    }
}
```

INPUT:

Enter an integer: 29

OUTPUT:

Number of set bits: 4

EX.NO: 01 (I)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION : Write a Java program to set the rightmost unset bit of a given integer using bitwise operations.

AIM:

To write a Java program that sets the rightmost unset bit (i.e., the 0 that occurs first from the right in binary) of a given integer using bitwise operations.

ALGORITHM:

Step 1: Start the program.
Step 2: Take an integer input n from the user.
Step 3: Compute the result using the expression: n | (n + 1)
Step 4: Display the result.
Step 5: End the program.

CODING:

```
import java.util.Scanner;
public class SetRightmostUnsetBit {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int n = sc.nextInt();
        int result = n | (n + 1);
        System.out.println("Result after setting rightmost unset bit: " + result);
        sc.close();
    }
}
```

INPUT:
Enter an integer: 29

OUTPUT:
Result after setting rightmost unset bit: 31

EX.NO: 01 (J)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: Write a Java program to swap two numbers using bitwise XOR operation (without using a temporary variable).

AIM:
To write a Java program to swap two integers using the XOR bitwise operator, without using a third variable.

ALGORITHM:
Step 1: Start the program.
Step 2: Take two integer inputs a and b from the user.
Step 3: Swap the values using XOR: a = a ^ b, b = a ^ b, a = a ^ b
Step 4: Display the values after swapping.
Step 5: End the program.

CODING:

```
import java.util.Scanner;
public class SwapUsingBitwise {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter first number (a): ");
        int a = sc.nextInt();
        System.out.print("Enter second number (b): ");
        int b = sc.nextInt();
        System.out.println("Before Swapping: a = " + a + ", b = " + b);
        a = a ^ b;
        b = a ^ b;
        a = a ^ b;

        System.out.println("After Swapping: a = " + a + ", b = " + b);
        sc.close();
    }
}
```

INPUT:
Enter first number (a): 12
Enter second number (b): 25

OUTPUT:
Before Swapping: a = 12, b = 25
After Swapping: a =25, b = 12

EX.NO: 01 (K)	Implementation of Number Theory and Bit Manipulation Techniques
DATE: 07.01.2025	

QUESTION: Given two integers dividend and divisor, divide two integers without using multiplication, division, and mod operator.

AIM:
To implement integer division by repeatedly subtracting the divisor from the dividend using loops and bitwise operations, without using multiplication, division, or modulo operators.

ALGORITHM:
Step 1: Take two integer inputs: dividend and divisor.11
Step 2: Handle edge cases such as division by zero or overflow.
Step 3: Convert both numbers to their absolute values and determine the result sign (positive or negative).
Step 4: Initialize quotient as 0.Use left-shifting to subtract the largest multiple of the divisor from the dividend.
Step 5: Accumulate the multiple values in quotient.
Step 6: Apply the sign to the result and return the final quotient.

CODING:
```
class Solution {
    public int divide(int dividend, int divisor) {
        if (dividend == Integer.MIN_VALUE && divisor == -1) {
            return Integer.MAX_VALUE;
        }
        long absDividend = Math.abs((long) dividend);
        long absDivisor = Math.abs((long) divisor);
        int quotient = 0;
        while (absDividend >= absDivisor) {
            long tempDivisor = absDivisor;
            long multiple = 1;
            while (absDividend >= (tempDivisor << 1)) {
                tempDivisor <<= 1;
                multiple <<= 1;
            }
            absDividend -= tempDivisor;
            quotient += multiple;
        }
        if ((dividend < 0) ^ (divisor < 0)) {
            quotient = -quotient;
        }
        return quotient;
    }
}
```

INPUT: dividend = 43, divisor = 3
OUTPUT: -5

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:
Thus, the implementation of several program such as Number Theory and Bit Manipulation Techniques was successfully executed and output has been verified.

EX.NO: 2 (A)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Find the factorial of a number using recursion.

AIM:

To compute the factorial of a number n using a recursive function.

ALGORITHM:

Step1: If n is 0 or 1, return 1.
Step2: Else return n * factorial(n - 1).
Step3: Use recursive calls to break the problem.
Step4: Return the result to the caller.
Step5: Print the final factorial value.

CODING:

```
public class RecursionExamples
{
    public static int factorial(int n)
    {
        if (n == 0 || n == 1) return 1;
        return n * factorial(n - 1);
    }
}
```

INPUT:

n = 4..

OUTPUT:

Factorial of 4 is 24.

EX.NO: 2 (B)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Find the nth Fibonacci number using recursion.

AIM:

To find the nth Fibonacci number using a recursive function.

ALGORITHM:

Step 1: If n is 0, return 0.
Step 2: If n is 1, return 1.
Step 3: Else return fibonacci(n-1) + fibonacci(n-2).
Step 4: Use recursion to break the problem into smaller subproblems.
Step 5: Return the final result.

CODING:

```
public class RecursionExamples
{
    public static int fibonacci(int n)
    {
        if (n <= 1) return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

INPUT:

n = 8.

OUTPUT:

Fibonacci of 8 is 21.

EX.NO: 2 (C)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Solve the Tower of Hanoi problem using recursion.

AIM:

To move n disks from the source rod to the destination rod using recursion and divide-and-conquer strategy.

ALGORITHM:

Step 1: Move n-1 disks from source to helper.
Step 2: Move the nth disk from source to destination.
Step 3: Move n-1 disks from helper to destination.
Step 4: Use recursion to repeat the process for smaller n.
Step 5: Print each move.

CODING:

```
public class RecursionExamples {
    public static void towerOfHanoi(int n, char source, char helper, char destination) {
        if (n == 1) {
            System.out.println("Move disk 1 from " + source + " to " + destination);
            return;
        }
        towerOfHanoi(n - 1, source, destination, helper); // Step 1
        System.out.println("Move disk " + n + " from " + source + " to " + destination); // Step 2
        towerOfHanoi(n - 1, helper, source, destination); // Step 3
    }
}
```

INPUT:

n = 4 (source = X, helper = Y, destination = Z).

OUTPUT:

```
Move disk 1 from X to Z
Move disk 2 from X to Y
Move disk 1 from Z to Y
Move disk 3 from X to Z
Move disk 1 from Y to X
Move disk 2 from Y to Z
Move disk 1 from X to Z
Move disk 4 from X to Y
Move disk 1 from Z to Y
Move disk 2 from Z to X
Move disk 1 from Y to X
Move disk 3 from Z to Y
Move disk 1 from X to Z
Move disk 2 from X to Y
Move disk 1 from Z to Y
```

EX.NO: 2 (D)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Sort an array using the merge sort algorithm.

AIM:

To sort elements by dividing the array and merging them in sorted order using the merge sort algorithm.

ALGORITHM:

Step1: Divide the array into two halves recursively.
Step2: Sort each half using merge sort.
Step3: Merge the sorted halves into one sorted array.
Step4: Repeat until the full array is sorted.
Step5: Return the sorted array.

CODING:

```
public class MergeSort {
    public static void mergeSort(int[] arr, int start, int end) {
        if (start >= end) return;
        int mid = start + (end - start) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        merge(arr, start, mid, end);
    }
    public static void merge(int[] arr, int start, int mid, int end) {
        int[] merged = new int[end - start + 1];
        int i = start, j = mid + 1, k = 0;
        while (i <= mid && j <= end) {
            if (arr[i] < arr[j]) merged[k++] = arr[i++];
            else merged[k++] = arr[j++];
        }
        while (i <= mid) merged[k++] = arr[i++];
        while (j <= end) merged[k++] = arr[j++];
        for (int l = 0; l < merged.length; l++) arr[start + l] = merged[l];
    }
    public static void main(String[] args) {
        int[] arr = {6, 3, 9, 5, 2, 8};
        mergeSort(arr, 0, arr.length - 1);
        for (int num : arr) System.out.print(num + " ");
    }
}
```

INPUT:

arr = [15, 10, 23, 42, 8, 7, 19].

OUTPUT:

Sorted array: [7, 8, 10, 15, 19, 23, 42].

EX.NO: 2 (E)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Sort an array using the merge sort algorithm.

AIM:

To sort elements by dividing the array and merging them in sorted order using the merge sort algorithm.

ALGORITHM:

Step1: Choose a pivot element.
Step2: Partition the array such that elements < pivot go left, > pivot go right.
Step3: Recursively apply quick sort on left and right subarrays.
Step4: Combine the sorted partitions.
Step5: Return the sorted array.

CODING:

```
public class QuickSort {
    public static void quickSort(int[] arr, int low, int high) {
        if (low >= high) return;
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high], i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
            }
        }
        int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
        return i + 1;
    }
    public static void main(String[] args) {
        int[] arr = {7, 2, 1, 6, 8, 5, 3, 4};
        quickSort(arr, 0, arr.length - 1);
        for (int num : arr) System.out.print(num + " ");
    }
}
```

INPUT:

arr = [12, 3, 5, 7, 19, 1, 6, 8, 10, 14].

OUTPUT:

Sorted array: [1, 3, 5, 6, 7, 8, 10, 12, 14, 19].

EX.NO: 2 (F)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Find the closest pair of points from a given set of 2D points using divide and conquer.

AIM:

To compute the smallest distance between two points in a plane using an efficient divide and conquer algorithm.

ALGORITHM:

Step1: Sort all points by X-coordinate.
Step2: Recursively divide points into left and right halves.
Step3: Find the closest pair in each half.
Step4: Check for closer points across the split line (within strip).
Step5: Return the minimum of the three cases.

CODING:

```
import java.util.*;
public class ClosestPair {
    static class Point {
        int x, y;
        Point(int x, int y) { this.x = x; this.y = y; }
    }
    public static double closestPair(Point[] points) {
        Arrays.sort(points, Comparator.comparingInt(p -> p.x));
        return closestUtil(points, 0, points.length - 1);
    }
    static double closestUtil(Point[] pts, int left, int right) {
        if (right - left <= 3) return bruteForce(pts, left, right);
        int mid = (left + right) / 2;
        double dl = closestUtil(pts, left, mid);
        double dr = closestUtil(pts, mid + 1, right);
        double d = Math.min(dl, dr);
        List<Point> strip = new ArrayList<>();
        for (int i = left; i <= right; i++)
            if (Math.abs(pts[i].x - pts[mid].x) < d)
                strip.add(pts[i]);
        strip.sort(Comparator.comparingInt(p -> p.y));
        for (int i = 0; i < strip.size(); i++) {
            for (int j = i + 1; j < strip.size() && (strip.get(j).y - strip.get(i).y) < d; j++) {
                d = Math.min(d, distance(strip.get(i), strip.get(j)));
            }
        }
        return d;
    }
    static double bruteForce(Point[] pts, int left, int right) {
        double min = Double.MAX_VALUE;
        for (int i = left; i <= right; i++)
            for (int j = i + 1; j <= right; j++)
```

```
                min = Math.min(min, distance(pts[i], pts[j]));
        return min;
    }
    static double distance(Point p1, Point p2) {
        return Math.hypot(p1.x - p2.x, p1.y - p2.y);
    }
    public static void main(String[] args) {
        Point[] points = {
            new Point(2, 3),
            new Point(12, 30),
            new Point(40, 50),
            new Point(5, 1),
            new Point(12, 10),
            new Point(3, 4)
        };
        System.out.printf("Closest distance: %.3f\n", closestPair(points));
    }
}
```

INPUT:

Points = [(1, 1), (3, 4), (6, 8), (8, 2), (10, 10), (12, 6)].

OUTPUT:

Closest distance: 2.828

Closest pair: (3,4) and (6,8).

EX.NO: 2 (G)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Place N queens on an N×N chessboard such that no two queens threaten each other.

AIM:

To solve the N-Queens problem using backtracking and display one of the possible arrangements.

ALGORITHM:

Step1: Place queens row by row starting from row 0.
Step2: For each row, try placing a queen in every column.
Step3: Check if the position is safe (no attacks from previously placed queens).
Step4: If safe, place the queen and move to the next row recursively.
Step5: Backtrack if no column is safe in the current row.

CODING:

```
public class NQueens {
    public static void solveNQueens(int n) {
        char[][] board = new char[n][n];
        for (char[] row : board) java.util.Arrays.fill(row, '.');
        placeQueens(board, 0);
    }
    public static boolean placeQueens(char[][] board, int row) {
        if (row == board.length) {
            printBoard(board);
            return true; // Return false if you want all solutions
        }
        for (int col = 0; col < board.length; col++) {
            if (isSafe(board, row, col)) {
                board[row][col] = 'Q';
                if (placeQueens(board, row + 1)) return true;
                board[row][col] = '.'; // backtrack
            }
        }
        return false;
    }
    public static boolean isSafe(char[][] board, int row, int col) {
        for (int i = 0; i < row; i++)
            if (board[i][col] == 'Q') return false;
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
            if (board[i][j] == 'Q') return false;
        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++)
            if (board[i][j] == 'Q') return false;
        return true;
    }
    public static void printBoard(char[][] board) {
        for (char[] row : board) {
            for (char c : row) System.out.print(c + " ");
            System.out.println();
        }
    }
    public static void main(String[] args) {
        int N = 4;
        solveNQueens(N);
    }
}
```

INPUT:

N = 8

OUTPUT:

Q . . . . . .
. . . . Q . . .
. . . . . Q .
. . . . . Q . .
. . Q . . . . .
. . . . . Q
. Q . . . . . .
. . . Q . . . .

EX.NO: 2 (H)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Fill a 9×9 Sudoku board so that each row, column, and 3×3 subgrid contains all digits 1 to 9 without repetition.

AIM:

To solve a partially-filled Sudoku puzzle using backtracking.

ALGORITHM:

Step1: Traverse each cell of the board.
Step2: If the cell is empty, try placing numbers 1 to 9.
Step3: Check if placing the number is valid.
Step4: If valid, recursively try to solve the rest of the board.
Step5: Backtrack if no number leads to a solution.

CODING:

```
public class SudokuSolver {
    public static boolean solveSudoku(char[][] board) {
        for (int row = 0; row < 9; row++) {
            for (int col = 0; col < 9; col++) {
                if (board[row][col] == '.') {
                    for (char c = '1'; c <= '9'; c++) {
                        if (isValid(board, row, col, c)) {
                            board[row][col] = c;
                            if (solveSudoku(board)) return true;
                            board[row][col] = '.'; // backtrack
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }
    public static boolean isValid(char[][] board, int row, int col, char c) {
        for (int i = 0; i < 9; i++) {
            if (board[i][col] == c || board[row][i] == c) return false;
            int subRow = 3 * (row / 3) + i / 3;
            int subCol = 3 * (col / 3) + i % 3;
            if (board[subRow][subCol] == c) return false;
        }
        return true;
    }
    public static void printBoard(char[][] board) {
        for (char[] row : board) {
            for (char c : row) System.out.print(c + " ");
            System.out.println();
        }
    }
    public static void main(String[] args) {
        char[][] board = {
            {'5','3','.','.','7','.','.','.','.'},
            {'6','.','.','1','9','5','.','.','.'},
            {'.','9','8','.','.','.','.','6','.'},
            {'8','.','.','.','6','.','.','.','3'},
```

```
            {'4','.','.','8','.','3','.','.','1'},
            {'7','.','.','.','.','2','.','.','.','6'},
            {'.','6','.','.','.','.','.','2','8','.'},
            {'.','.','.','4','1','9','.','.','.','5'},
            {'.','.','.','.','.','8','.','.','7','9'}
        };
        if (solveSudoku(board)) {
            System.out.println("Solved Sudoku:");
            printBoard(board);
        } else {
            System.out.println("No solution exists.");
        }
    }
}
```

INPUT:

A 9×9 board with some pre-filled digits and '.' for empty cells.

OUTPUT:

The completed Sudoku board.

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

EX.NO: 2 (I)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Sort an array of integers using Insertion Sort technique.

AIM:

To sort an array by inserting each element into its correct position in the sorted part of the array using decrease and conquer strategy.

ALGORITHM:

Step1: Start from the second element (index 1) in the array.
Step2: Store the current element in a variable.
Step3: Compare it with elements to its left (sorted part) and shift larger elements one position to the right.
Step4: Insert the current element at the correct position.
Step5: Repeat until the end of the array.

CODING:

```
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int current = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > current) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = current;
        }
    }
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 5, 6};
        insertionSort(arr);
        for (int num : arr) System.out.print(num + " ");
    }
}
```

INPUT:

arr = {12, 11, 13, 5, 6}

OUTPUT:

Sorted array:

5 6 11 12 13

EX.NO: 2 (J)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Count the frequency of each element in an array.

AIM:

To demonstrate how hashing can be used to store and retrieve element frequencies efficiently.

ALGORITHM:

Step1: Create an empty hash map.
Step2: Traverse the array.
Step3: For each element, increment its count in the map.
Step4: After the loop, print each key-value pair.

CODING:

```
import java.util.*;
public class HashingExample {
    public static void main(String[] args) {
        int[] arr = {2, 3, 2, 3, 5, 2};
        Map<Integer, Integer> map = new HashMap<>();
        for (int num : arr) map.put(num, map.getOrDefault(num, 0) + 1);
        for (Map.Entry<Integer, Integer> entry : map.entrySet())
            System.out.println(entry.getKey() + " -> " + entry.getValue());
    }
}
```

INPUT:

arr = [1, 4, 4, 1, 6, 7, 7, 7]

OUTPUT:

1 -> 2

4 -> 2

6 -> 1

7 -> 3

EX.NO: 2 (K)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Generate all permutations of a given string.

AIM:

To recursively generate all permutations using backtracking.

ALGORITHM:

Step1: If the input string is empty, print the answer.
Step2: Loop through the string.
Step3: Pick each character and fix it.
Step4: Recurse with the remaining characters.
Step5: Backtrack to explore other options.

CODING:

```
public class Permutations {
    public static void permute(String str, String ans) {
        if (str.length() == 0) {
            System.out.println(ans);
            return;
        }
        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            String ros = str.substring(0, i) + str.substring(i + 1);
            permute(ros, ans + ch);
        }
    }
    public static void main(String[] args) {
        permute("ABC", "");
    }
}
```

INPUT:

str = "123".

OUTPUT:

123

132

213

231

312

321

EX.NO: 2 (L)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Find number of combinations of n items taken r at a time.

AIM:

To compute nCr using recursion.

ALGORITHM:

Step1: If r == 0 or r == n, return 1.
Step2: Recursively calculate nCr = (n-1)C(r-1) + (n-1)Cr.
Step3: Return the result.

CODING:

```
public class Combinations {
    public static int nCr(int n, int r) {
        if (r == 0 || r == n) return 1;
        return nCr(n - 1, r - 1) + nCr(n - 1, r);
    }
    public static void main(String[] args) {
        System.out.println("5C2 = " + nCr(5, 2));
    }
}
```

INPUT:

n = 6, r = 3.

OUTPUT:

6C2 = 20.

EX.NO: 2 (M)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Print all subsequences of a given string.

AIM:

To generate all subsequences of a string using recursion.

ALGORITHM:

Step1: If string is empty, print current answer.
Step2: Recurse by excluding first character.
Step3: Recurse by including first character.
Step4: Continue until all characters are processed.

CODING:

```
public class Subsequences {
    public static void generateSubsequences(String str, String ans) {
        if (str.length() == 0) {
            System.out.println(ans);
            return;
        }
        generateSubsequences(str.substring(1), ans);
        generateSubsequences(str.substring(1), ans + str.charAt(0));
    }
    public static void main(String[] args) {
        generateSubsequences("abc", "");
    }
}
```

INPUT:

str = "123"

OUTPUT:

```
(empty)
3
2
23
1
13
12
123
```

EX.NO: 2 (N)	Implementation of Recursion and Backtracking Techniques
DATE: 21.01.2025	

QUESTION : Generate all subsets of a given integer array.

AIM:

To generate power set (all subsets) using recursion and backtracking.

ALGORITHM:

Step1: If index == array length, print current subset.
Step2: Recurse without including current element.
Step3: Include current element, recurse, then backtrack.

CODING:

```
import java.util.*;
public class Subsets {
    public static void findSubsets(int[] arr, List<Integer> current, int index) {
        if (index == arr.length) {
            System.out.println(current);
            return;
        }
        findSubsets(arr, current, index + 1);
        current.add(arr[index]);
        findSubsets(arr, current, index + 1);
        current.remove(current.size() - 1); // backtrack
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        findSubsets(arr, new ArrayList<>(), 0);
    }
}
```

INPUT:

arr = [4, 5, 6]

OUTPUT:

[ ]
[6]
[5]
[5, 6]
[4]
[4, 6]
[4, 5]
[4, 5, 6]

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:

Thus, the implementation of several program such as Recursion and Backtracking Techniques was successfully executed and output has been verified.

EX.NO: 3 (A)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Sort an array of non-negative integers using counting sort.

AIM:

To sort an array in linear time when the range of elements is small.

ALGORITHM:

Step1: Find the maximum value in the array.
Step2: Initialize a count array of size max + 1.
Step3: Count the frequency of each number.
Step4: Traverse the count array and reconstruct the sorted array.

CODING:

```
import java.util.*;
public class CountingSort {
    public static void countingSort(int[] arr) {
        int max = Arrays.stream(arr).max().getAsInt();
        int[] count = new int[max + 1];
        for (int num : arr) count[num]++;
        int index = 0;
        for (int i = 0; i < count.length; i++) {
            while (count[i]-- > 0) arr[index++] = i;
        }
    }
    public static void main(String[] args) {
        int[] arr = {4, 2, 2, 8, 3, 3, 1};
        countingSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

INPUT:

```
arr = [10, 5, 3, 7, 5, 8, 2, 1]
```

OUTPUT:

```
[1, 2, 3, 5, 5, 7, 8, 10]
```

EX.NO: 3 (B)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Sort an array of integers using radix sort.

AIM:

To sort large integers in linear time using digit-wise sorting.

ALGORITHM:

Step1: Find the maximum number to determine number of digits.
Step2: For each digit (ones, tens, hundreds, etc.), do counting sort.
Step3: Start with the least significant digit and move to the most significant.

CODING:

```
import java.util.*;
public class RadixSort {
    public static void radixSort(int[] arr) {
        int max = Arrays.stream(arr).max().getAsInt();
        for (int exp = 1; max / exp > 0; exp *= 10)
            countSortByDigit(arr, exp);
    }
    private static void countSortByDigit(int[] arr, int exp) {
        int[] output = new int[arr.length];
        int[] count = new int[10];
        for (int num : arr) count[(num / exp) % 10]++;
        for (int i = 1; i < 10; i++) count[i] += count[i - 1];
        for (int i = arr.length - 1; i >= 0; i--) {
            int digit = (arr[i] / exp) % 10;
            output[count[digit] - 1] = arr[i];
            count[digit]--;
        }
        System.arraycopy(output, 0, arr, 0, arr.length);
    }
    public static void main(String[] args) {
        int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
        radixSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

INPUT:

arr = [53, 89, 150, 36, 633, 233]

OUTPUT:

[36, 53, 89, 150, 233, 633]

EX.NO: 3 (C)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Find the first index at which a number can be inserted to maintain sorted order.

AIM:

To find the lower bound (first occurrence ≥ key) using binary search.

ALGORITHM:

Step1: Set left = 0, right = arr.length.
Step2: While left < right, check mid = (left + right)/2.
Step3: If arr[mid] < key, move left = mid + 1.
Step4: Else, move right = mid.
Step5: Return left.

CODING:

```
import java.util.*;
public class LowerBound {
    public static int lowerBound(int[] arr, int key) {
        int left = 0, right = arr.length;
        while (left < right) {
            int mid = (left + right) / 2;
            if (arr[mid] < key)
                left = mid + 1;
            else
                right = mid;
        }
        return left;
    }
    public static void main(String[] args) {
        int[] arr = {1, 3, 3, 5, 6};
        int key = 3;
        System.out.println("Lower Bound of " + key + " is at index: " + lowerBound(arr, key));
    }
}
```

INPUT:
arr = [2, 4, 6, 8, 10], key = 7
OUTPUT:
Lower Bound of 7 is at index: 3

EX.NO: 3 (D)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Determine whether counting sort can be applied on a given array.

AIM:

To check if the array has a small range of values that makes linear sorting feasible.

ALGORITHM:

Step1: Find the min and max of the array.
Step2: Calculate range = max - min.
Step3: If range <= array.length, counting sort is feasible.

CODING:

```
public class LinearTimeSortCheck {
    public static boolean canUseCountingSort(int[] arr) {
        int min = Integer.MAX_VALUE, max = Integer.MIN_VALUE;
        for (int num : arr) {
            min = Math.min(min, num);
            max = Math.max(max, num);
        }
        return (max - min) <= arr.length; // Range is small enough
    }
    public static void main(String[] args) {
        int[] arr = {1, 3, 2, 5, 4};
        System.out.println("Can use Counting Sort: " + canUseCountingSort(arr));
    }
}
```

INPUT:
arr = [10, 25, 5, 50, 30]

OUTPUT:
Can use Counting Sort: false

EX.NO: 3 (E)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Write a program to perform preorder traversal of a binary tree.

AIM:

To visit the root node first, then traverse the left and right subtrees recursively.

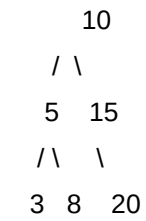
ALGORITHM:

Step1: Visit the root node.
Step2: Traverse the left subtree.
Step3: Traverse the right subtree.

CODING:

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int x) {
        val = x;
    }
}public class PreorderTraversal {
    public static void preorder(TreeNode root) {
        if (root == null) return;
        System.out.print(root.val + " ");
        preorder(root.left);
        preorder(root.right);
    }public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);
        System.out.print("Preorder Traversal: ");
        preorder(root);
    }
}
```

INPUT:



OUTPUT:

Preorder Traversal: 10 5 3 8 15 20

EX.NO: 3 (F)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Write a program to perform inorder traversal of a binary tree.

AIM:

To traverse the left subtree first, then visit the root, and finally the right subtree.

ALGORITHM:

Step1: Traverse the left subtree.
Step2: Visit the root node.
Step3: Traverse the right subtree.

CODING:

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int x) {
        val = x;
    }
}
public class InorderTraversal {
    public static void inorder(TreeNode root) {
        if (root == null) return;
        inorder(root.left);
        System.out.print(root.val + " ");
        inorder(root.right);
    }
    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);

        System.out.print("Inorder Traversal: ");
        inorder(root);
    }
}
```

INPUT:

```
   10
  /  \
 6    15
```

OUTPUT:

Inorder Traversal: 3 6 8 10 15

EX.NO: 3 (G)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Write a program to perform postorder traversal of a binary tree.

AIM:

To traverse the left and right subtrees first, then visit the root node.

ALGORITHM:

Step1: Traverse the left subtree.
Step2: Traverse the right subtree.
Step3: Visit the root node.

CODING:

```
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int x) {
        val = x;
    }
}
public class PostorderTraversal {
    public static void postorder(TreeNode root) {
        if (root == null) return;
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.val + " ");
    }
    public static void main(String[] args) {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);

        System.out.print("Postorder Traversal: ");
        postorder(root);
    }
}
```

INPUT:

```
      10
     /  \
    6    15
   / \
  3   8
```

OUTPUT:

Postorder Traversal: 3 8 6 15 10

EX.NO: 3 (H)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Write a Java program to implement a Min Heap using a priority queue and perform basic operations like insertion and extraction of the minimum element..

AIM:

To implement a Min Heap, where the parent node is less than or equal to its children.

ALGORITHM:

Step1: Insert elements one by one.
Step2: After each insertion, heapify up to maintain the min-heap property.
Step3: In extractMin(), remove the root and replace with the last element, then heapify down.

CODING:

```
import java.util.PriorityQueue;
public class MinHeapExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        minHeap.add(10);
        minHeap.add(5);
        minHeap.add(30);
        minHeap.add(2);
        System.out.println("Min Heap: " + minHeap);
        System.out.println("Extract Min: " + minHeap.poll()); // Removes 2
        System.out.println("After Extraction: " + minHeap);
    }
}
```

INPUT:

Insert: 20, 3, 17, 8, 45

OUTPUT:

Min Heap: [3, 8, 17, 20, 45]

Extract Min: 3

After Extraction: [8, 20, 17, 45]

EX.NO: 3 (I)	Implementation of Tree Algorithms
DATE: 04.02.2025	

QUESTION : Write a Java program to implement a Max Heap using a priority queue and extract the maximum element..

AIM:

To implement a Max Heap, where the parent node is greater than or equal to its children.

ALGORITHM:

Step1: Use a PriorityQueue with a custom comparator to reverse natural order.
Step2: Insert elements.
Step3: Always the maximum element stays at the top.

CODING:

```
import java.util.Collections;
import java.util.PriorityQueue;
public class MaxHeapExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        maxHeap.add(10);
        maxHeap.add(5);
        maxHeap.add(30);
        maxHeap.add(2);
        System.out.println("Max Heap: " + maxHeap);
        System.out.println("Extract Max: " + maxHeap.poll()); // Removes 30
        System.out.println("After Extraction: " + maxHeap);
    }
}
```

INPUT:

Insert: 15, 40, 25, 8, 50

OUTPUT:

Max Heap: [50, 40, 25, 8, 15]
Extract Max: 50
After Extraction: [40, 15, 25, 8]

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:

Thus, the implementation of several program such as Tree Algorithms was successfully executed and output has been verified.

EX.NO: 4 (A)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION : Write a program to perform Depth First Search (DFS) traversal on a graph.

AIM:

To traverse a graph by going as deep as possible along each branch before backtracking.

ALGORITHM:

Step1: Start from a selected source node.
Step2: Mark the node as visited.
Step3: Recur for all unvisited adjacent nodes.
Step4: Use recursion or a stack.

CODING:

```
import java.util.*;
public class DFSExample {
    static void dfs(int v, boolean[] visited, List<List<Integer>> adj) {
        visited[v] = true;
        System.out.print(v + " ");
        for (int neighbor : adj.get(v)) {
            if (!visited[neighbor]) {
                dfs(neighbor, visited, adj);
            }
        }
    }
    public static void main(String[] args) {
        int vertices = 5;
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < vertices; i++) adj.add(new ArrayList<>());
        adj.get(0).add(1);
        adj.get(0).add(2);
        adj.get(1).add(3);
        adj.get(1).add(4);
        boolean[] visited = new boolean[vertices];
        System.out.print("DFS Traversal: ");
        dfs(0, visited, adj);
    }
}
```

INPUT:

0 -> 1, 2

1 -> 3, 4

3 → 4

OUTPUT:

DFS Traversal: 0 1 2 3 4

EX.NO: 4 (B)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION : Write a program to perform Breadth First Search (BFS) traversal on a graph.

AIM:

To traverse a graph level by level using a queue.

ALGORITHM:

Step1: Start from the source node and mark it visited.
Step2: Add it to a queue.
Step3: While the queue is not empty; Remove a node,Visit all its unvisited adjacent nodes and enqueue them

CODING:

```
import java.util.*;
public class BFSExample {
    static void bfs(int start, List<List<Integer>> adj, int vertices) {
        boolean[] visited = new boolean[vertices];
        Queue<Integer> queue = new LinkedList<>();
        visited[start] = true;
        queue.add(start);
        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print(node + " ");
            for (int neighbor : adj.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }
    public static void main(String[] args) {
        int vertices = 5;
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < vertices; i++) adj.add(new ArrayList<>());
        adj.get(0).add(1);
        adj.get(0).add(2);
        adj.get(1).add(3);
        adj.get(1).add(4);
        System.out.print("BFS Traversal: ");
        bfs(0, adj, vertices);
    }
}
```

INPUT:

0 -> 1, 2
1 -> 3, 4

OUTPUT:

BFS Traversal: 0 1 2 3 4

EX.NO: 4 (C)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION : Find the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights.

AIM:

To compute the shortest distances from a single source node to all other nodes using a priority queue.

ALGORITHM:

Step1: Initialize distances as infinity, except for the source node (0).

Step2: Use a priority queue to pick the node with the minimum distance.

Step3: For each adjacent node, update its distance if a shorter path is found.

Step4: Repeat until the queue is empty.

CODING:

```
import java.util.*;
class DijkstraExample {
    static class Pair {
        int node, weight;
         Pair(int n, int w) { node = n; weight = w; }
    }
    public static void dijkstra(List<List<Pair>> graph, int src) {
        int n = graph.size();
        int[] dist = new int[n];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;
        PriorityQueue<Pair> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.weight));
        pq .add(new Pair(src, 0));
        while (!pq.isEmpty()) {
            Pair current = pq.poll();
            for (Pair neighbor : graph.get(current.node)) {
                if (dist[current.node] + neighbor.weight < dist[neighbor.node]) {
                    dist[neighbor.node] = dist[current.node] + neighbor.weight;
                    pq.add(new Pair(neighbor.node, dist[neighbor.node]));
                }
            }
        }
        System.out.println("Shortest distances from node " + src + ":");
        for (int i = 0; i < n; i++) {
            System.out.println("To node " + i + " = " + dist[i]);
        }
    }
    public static void main(String[] args) {
        int n = 5;
        List<List<Pair>> graph = new ArrayList<>();
        for (int i = 0; i < n; i++) graph.add(new ArrayList<>());
        graph.get(0).add(new Pair(1, 4));
        graph.get(0).add(new Pair(2, 1));
        graph.get(2).add(new Pair(1, 2));
        graph.get(1).add(new Pair(3, 1));
        graph.get(2).add(new Pair(3, 5));
        graph.get(3).add(new Pair(4, 3));
        dijkstra(graph, 0);
    }
}
```

INPUT:

0 -> 1 (2), 0 -> 2 (4)
1 -> 2 (1), 1 -> 3 (7), 2 -> 4 (3), 4 -> 3 (2) 4 → 5 (5)
  3 → 5 (1)

OUTPUT:

Shortest distances from node 0:
To node 0 = 0
To node 1 = 2
To node 2 = 3
To node 3 = 8
To node 4 = 6
To node 5 = 9

EX.NO: 4 (D)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION : Find the shortest paths from a source to all vertices in a graph that may contain negative weights.

AIM:

To detect negative weight edges and compute the shortest distances from a single source.

ALGORITHM:

Step1: Initialize distances to all vertices as ∞, except the source = 0.

Step2: Repeat V-1 times (V = number of vertices); Relax all edges: update dist[u] + weight < dist[v].

Step3: (Optional) Run one more time to check for negative weight cycle.

CODING:

```
public class BellmanFord {
 static class Edge {
    int u, v, weight;
    Edge(int u, int v, int w) {
       this.u = u;
       this.v = v;
       this.weight = w;
    }
 }
 public static void bellmanFord(int V, Edge[] edges, int src) {
    int[] dist = new int[V];
    for (int i = 0; i < V; i++)
       dist[i] = Integer.MAX_VALUE;
    dist[src] = 0;
    for (int i = 1; i < V; i++) {
       for (Edge edge : edges) {
          if (dist[edge.u] != Integer.MAX_VALUE &&
             dist[edge.u] + edge.weight < dist[edge.v]) {
             dist[edge.v] = dist[edge.u] + edge.weight;
          }
       }
    }
    for (Edge edge : edges) {
       if (dist[edge.u] != Integer.MAX_VALUE &&
          dist[edge.u] + edge.weight < dist[edge.v]) {
          System.out.println("Graph contains a negative weight cycle");
          return;
       }
    }
    System.out.println("Shortest distances from source " + src + ":");
    for (int i = 0; i < V; i++) {
       System.out.println("To " + i + " = " + dist[i]);
    }
 }
 public static void main(String[] args) {
    int V = 5;
    Edge[] edges = {
       new Edge(0, 1, -1),
```

```
            new Edge(0, 2, 4),
            new Edge(1, 2, 3),


            new Edge(1, 3, 2),
            new Edge(1, 4, 2),
            new Edge(3, 2, 5),
            new Edge(3, 1, 1),
            new Edge(4, 3, -3)
        };
        bellmanFord(V, edges, 0);
    }
}
```

INPUT:

0 → 1 (1), 0 → 2 (4), 1 → 2 (-3), 1 → 3 (2),
2 → 3 (3)

OUTPUT:

Shortest distances from source 0:
To 0 = 0
To 1 = 1
To 2 = -2
To 3 = 1

EX.NO: 4 (E)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION : Find the shortest distances between all pairs of vertices in a weighted graph.

AIM:

To compute shortest paths between every pair of nodes, even with negative edge weights (but no negative cycles).

ALGORITHM:

Step1: Create a distance matrix dist[][] from the input graph.
Step2: For each vertex k, update the distance dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).
Step3: Repeat for all vertices k as intermediate nodes.
Step4: Detect negative cycles if dist[i][i] < 0 after the algorithm.

CODING:

```
public class FloydWarshall {
 final static int INF = 99999;
 public static void floydWarshall(int[][] graph, int V) {
    int[][] dist = new int[V][V];
    for (int i = 0; i < V; i++)
       for (int j = 0; j < V; j++)
          dist[i][j] = graph[i][j];
    for (int k = 0; k < V; k++) {
       for (int i = 0; i < V; i++) {
          for (int j = 0; j < V; j++) {
             if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
          }
       }
    }
    System.out.println("All-Pairs Shortest Path Matrix:");
    for (int i = 0; i < V; ++i) {
       for (int j = 0; j < V; ++j) {
          if (dist[i][j] == INF)
             System.out.print("INF ");
          else
             System.out.print(dist[i][j] + " ");
          }
          System.out.println();
       }
  }
  public static void main(String[] args) {
     int V = 4;
     int[][] graph = {
        {0,   5,  INF, 10},
        {INF, 0,   3, INF},
        {INF, INF, 0,   1},
        {INF, INF, INF, 0}
     };
     floydWarshall(graph, V);
  }
}
```

INPUT:

- 0 → 1 (3), 0 → 3 (7)
- 1 → 0 (8), 1 → 2 (2)
- 2 → 0 (5), 2 → 3 (1)
- 3 → 0 (2)

OUTPUT:

All-Pairs Shortest Path Matrix:

0 3 5 6

7 0 2 3

3 6 0 1

2 5 7 0

EX.NO: 4 (F)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION: Given a weighted, undirected, and connected graph of V vertices and E edges.

AIM:

To implement **Prim's Algorithm** to find the **Minimum Spanning Tree (MST)** of a given connected, undirected, and weighted graph and **compute the sum of the weights of the edges** included in the MST.

ALGORITHM:

Step 1: Initialize distance matrix dist[][] with edge weights, dist[i][j] = 0 if i == j, and ∞ if no edge exists.
Step 2: For each vertex k, consider it as an intermediate node.
Step 3: For each pair (i, j), update dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).
Step 4: After all updates, if dist[i][i] < 0 for any i, report a negative weight cycle.

CODING:

```
import java.util.*;
class Pair {
    int node;
    int distance;
    public Pair(int distance, int node) {
        this.node = node;
        this.distance = distance;
    }
}
class Solution {
    static int spanningTree(int V, ArrayList<ArrayList<ArrayList<Integer>>> adj) {
        PriorityQueue<Pair> pq = new PriorityQueue<Pair>((x, y) -> x.distance - y.distance);
        int[] vis = new int[V];
        pq.add(new Pair(0, 0));
        int sum = 0;
        while (pq.size() > 0) {
            int wt = pq.peek().distance;
            int node = pq.peek().node;
            pq.remove();
            if (vis[node] == 1) continue;
            vis[node] = 1;
            sum += wt;
            for (int i = 0; i < adj.get(node).size(); i++) {
                int edW = adj.get(node).get(i).get(1);
```

```
                int adjNode = adj.get(node).get(i).get(0);
                if (vis[adjNode] == 0) {
                    pq.add(new Pair(edW, adjNode));
                } }  }
        return sum;
    }}
public class tUf {
    public static void main(String[] args) {
        int V = 5;
        ArrayList<ArrayList<ArrayList<Integer>>> adj = new ArrayList<ArrayList<ArrayList<Integer>>>();
        int[][] edges =  {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<ArrayList<Integer>>());
        }
        for (int i = 0; i < 6; i++) {
            int u = edges[i][0];
            int v = edges[i][1];
            int w = edges[i][2];
            ArrayList<Integer> tmp1 = new ArrayList<Integer>();
            ArrayList<Integer> tmp2 = new ArrayList<Integer>();
            tmp1.add(v);
            tmp1.add(w);
            tmp2.add(u);
            tmp2.add(w);
            adj.get(u).add(tmp1);
            adj.get(v).add(tmp2);
        }
        Solution obj = new Solution();
        int sum = obj.spanningTree(V, adj);
        System.out.println("The sum of all the edge weights: " + sum);
    }
}
```

INPUT:

V = 4

edges = {

{0, 1, 10},{0, 2, 6}, {0, 3, 5},   {1, 3, 15}, {2, 3, 4} }

OUTPUT:

19

EX.NO: 4 (G)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION: Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, Find the number of strongly connected components in the graph.

AIM:

The aim is to find the number of strongly connected components (SCCs) in a directed graph with V vertices and E edges. A strongly connected component is a subgraph where any two vertices are reachable from each other.

ALGORITHM:

Step 1: Initialize a stack st and visited array visited[] as false for all V vertices.
Step 2: For each vertex v, if not visited, perform DFS and push vertices onto the stack after visiting all descendants.
Step 3: Create the transpose (reverse) of the graph by reversing all edges.
Step 4: Reset the visited array visited[] to false.
Step 5: While the stack is not empty, pop the top vertex v; if not visited, perform DFS on the transposed graph starting from v and count it as one SCC.
Step 6: After the stack is empty, return the number of DFS calls performed in step 5 as the number of SCCs.

CODING:

```
import java.util.*;
public class SCCKosaraju {
   private static void dfs1(int v, List<List<Integer>> graph, boolean[] visited, Stack<Integer> stack) {
      visited[v] = true;
      for (int neighbor : graph.get(v)) {
         if (!visited[neighbor])
            dfs1(neighbor, graph, visited, stack);
      }
      stack.push(v);
   }
   private static void dfs2(int v, List<List<Integer>> transpose, boolean[] visited) {
      visited[v] = true;
      for (int neighbor : transpose.get(v)) {
         if (!visited[neighbor])
            dfs2(neighbor, transpose, visited);
      }
   }
   public static int countSCCs(int V, List<List<Integer>> graph) {
      boolean[] visited = new boolean[V];
      Stack<Integer> stack = new Stack<>();
      for (int i = 0; i < V; i++) {
         if (!visited[i]) {
            dfs1(i, graph, visited, stack);
         }
      }
      List<List<Integer>> transpose = new ArrayList<>();
      for (int i = 0; i < V; i++) {
         transpose.add(new ArrayList<>());
      }
      for (int u = 0; u < V; u++) {
         for (int v : graph.get(u)) {
            transpose.get(v).add(u);
         }
```

```
        }
        Arrays.fill(visited, false);
        int count = 0;
        while (!stack.isEmpty()) {
            int v = stack.pop();
            if (!visited[v]) {
                dfs2(v, transpose, visited);
                count++;
            }
        }

        return count;
    }
    public static void main(String[] args) {
        int V = 5;
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            graph.add(new ArrayList<>());
        }
        graph.get(0).add(1);
        graph.get(1).add(2);
        graph.get(2).add(0);
        graph.get(1).add(3);
        graph.get(3).add(4);
        int result = countSCCs(V, graph);
        System.out.println("Number of Strongly Connected Components: " + result);
    }
}
```

INPUT:
Number of Vertices (V): 4
Number of Edges (E): 4
Edges[] = [[0, 1], [1, 2], [2, 0], [3, 2]]

OUTPUT:
Number of Strongly Connected Components: 2
Components:
Component 1: 0 1 2
Component 2: 3

EX.NO: 4 (H)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION: Given two components of an undirected graph.

AIM:
To identify and verify two connected components in an undirected graph and determine whether any path exists between nodes from different components.

ALGORITHM:
Step 1: Initialize a visited array to mark visited nodes.
Step 2: Traverse all vertices using DFS starting from each unvisited node.
Step 3: For each DFS traversal, store the connected nodes in a component list.
Step 4: Label each component with a unique identifier.
Step 5: After processing, you will get two or more components depending on connectivity.
Step 6: Return the list of components or check whether two nodes belong to the same component.

CODING:
```
import java.io.*;
import java.util.*;
class DisjointSet {
    List<Integer> rank = new ArrayList<>();
    List<Integer> parent = new ArrayList<>();
    public DisjointSet(int n) {
        for (int i = 0; i <= n; i++) {
            rank.add(0);
            parent.add(i);
        }
    }
    public int findUPar(int node) {
        if (node == parent.get(node)) {
            return node;
        }
        int ulp = findUPar(parent.get(node));
        parent.set(node, ulp);
        return parent.get(node);
    }
    public void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank.get(ulp_u) < rank.get(ulp_v)) {
            parent.set(ulp_u, ulp_v);
        } else if (rank.get(ulp_v) < rank.get(ulp_u)) {
            parent.set(ulp_v, ulp_u);
        } else {
            parent.set(ulp_v, ulp_u);
            int rankU = rank.get(ulp_u);
            rank.set(ulp_u, rankU + 1);
        }
    }

}
class Main {
    public static void main (String[] args) {
        DisjointSet ds = new DisjointSet(7);
        ds.unionByRank(1, 2);
```

```
        ds.unionByRank(2, 3);
        ds.unionByRank(4, 5);
        ds.unionByRank(6, 7);
        ds.unionByRank(5, 6);
        if (ds.findUPar(3) == ds.findUPar(7)) {
            System.out.println("Same");
        } else
            System.out.println("Not Same");
        ds.unionByRank(3, 7);
        if (ds.findUPar(3) == ds.findUPar(7)) {
            System.out.println("Same");
        } else
            System.out.println("Not Same");
    }
}
```

INPUT:
rank: {0, 0, 0, 0, 0, 0, 0, 0}
parent: {0, 1, 2, 3, 4, 5, 6, 7}
OUTPUT:
Not Same
Same

EX.NO: 4 (I)	Implementation of Graph Algorithms
DATE: 11.02.2025	

QUESTION: Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

AIM:
To implement a program that finds any valid topological sorting of a Directed Acyclic Graph (DAG) using an appropriate algorithm such as Kahn's Algorithm (BFS-based) or DFS-based approach.

ALGORITHM:
Step 1: Calculate the indegree of all vertices.
Step 2: Add all vertices with indegree 0 to a queue.
Step 3: Remove from queue, add to topological order, and reduce indegree of neighbors.
Step 4: If any neighbor's indegree becomes 0, add it to the queue.
Step 5: Repeat until the queue is empty and all vertices are processed.

CODING:
```
import java.util.*;
class Solution {
    private static void dfs(int node, int vis[], Stack<Integer> st,
            ArrayList<ArrayList<Integer>> adj) {
        vis[node] = 1;
        for (int it : adj.get(node)) {
            if (vis[it] == 0)
                dfs(it, vis, st, adj);
        }
        st.push(node);
    }
    static int[] topoSort(int V, ArrayList<ArrayList<Integer>> adj) {
        int vis[] = new int[V];
        Stack<Integer> st = new Stack<Integer>();
        for (int i = 0; i < V; i++) {
            if (vis[i] == 0) {
                dfs(i, vis, st, adj);
            }
        }
        int ans[] = new int[V];
        int i = 0;
        while (!st.isEmpty()) {
            ans[i++] = st.peek();
            st.pop();
        }
        return ans;
    }
}
public class tUf {
    public static void main(String[] args) {
        int V = 6;
        ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
        adj.get(2).add(3);
        adj.get(3).add(1);
        adj.get(4).add(0);
        adj.get(4).add(1);
```

```
        adj.get(5).add(0);
        adj.get(5).add(2);

        int[] ans = Solution.topoSort(V, adj);
        for (int node : ans) {
            System.out.print(node + " ");
        }
        System.out.println("");
    }
}
```

INPUT:
V = 6, E = 7

OUTPUT:
5 4 2 3 1 0

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:
Thus, the implementation of several program such as Graph Algorithms was successfully executed and output has been verified.

EX.NO: 5 (A)	Implementation of Greedy Algorithms
DATE: 18.02.2025	

QUESTION: Find the Minimum Spanning Tree starting from any vertex.

AIM: To implement Prim's Algorithm using the Greedy method to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph.

ALGORITHM:
Step 1: Initialize a min-heap and a visited set.
Step 2: Start from any node and push all adjacent edges to the heap.
Step 3: Always pick the minimum weight edge from the heap.
Step 4: If the edge leads to an unvisited node, add it to MST and mark visited.
Step 5: Repeat until all nodes are included in the MST.

CODING:
```
import java.util.*;
class PrimMST {
    static class Pair {
        int vertex, weight;
        Pair(int v, int w) {
            vertex = v;
            weight = w;
        }
    }
    public static void primsAlgo(int V, List<List<Pair>> adj) {
        boolean[] visited = new boolean[V];
        PriorityQueue<Pair> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.weight));
        pq.add(new Pair(0, 0));
        int totalWeight = 0;
        while (!pq.isEmpty()) {
            Pair curr = pq.poll();
            if (visited[curr.vertex]) continue;
            visited[curr.vertex] = true;
            totalWeight += curr.weight;
            for (Pair edge : adj.get(curr.vertex)) {
                if (!visited[edge.vertex]) {
                    pq.add(new Pair(edge.vertex, edge.weight));
                }
            }
        }
        System.out.println("Total weight of MST: " + totalWeight);
    }
    public static void main(String[] args) {
        int V = 5;
        List<List<Pair>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        adj.get(0).add(new Pair(1, 2));
        adj.get(1).add(new Pair(0, 2));
        adj.get(1).add(new Pair(2, 3));
        adj.get(2).add(new Pair(1, 3));
        adj.get(0).add(new Pair(3, 6));
        adj.get(3).add(new Pair(0, 6));
        adj.get(1).add(new Pair(3, 8));
        adj.get(3).add(new Pair(1, 8));
        adj.get(1).add(new Pair(4, 5));
```

```
        adj.get(4).add(new Pair(1, 5));
        primsAlgo(V, adj);
    }
}
```

INPUT:
Number of vertices = 6
Edges[] = [[0, 1, 4], [0, 2, 3], [1, 2, 1], [1, 3, 2], [2, 3, 4], [2, 4, 5], [3, 4, 6], [3, 5, 7], [4, 5, 8]]
OUTPUT:
Total Weight of MST: 18

EX.NO: 5 (B)	Implementation of Greedy Algorithms
DATE: 18.02.2025	

QUESTION: Build MST by choosing the smallest weight edges first.

AIM:
To implement Kruskal's Algorithm using the Greedy method to construct the Minimum Spanning Tree (MST) of a graph by selecting the shortest possible edges while avoiding cycles.

ALGORITHM:
Step 1: Sort all edges in increasing order of weight.
Step 2: Initialize Disjoint Set (Union-Find) for all vertices.
Step 3: For each edge (u, v), if u and v are in different sets, add it to MST.
Step 4: Merge the sets of u and v.
Step 5: Repeat until MST has (V - 1) edges.

CODING:
```
import java.util.*;
class KruskalMST {
    static class Edge implements Comparable<Edge> {
        int u, v, weight;
        Edge(int u, int v, int w) {
            this.u = u; this.v = v; this.weight = w;
        }
        public int compareTo(Edge e) {
            return this.weight - e.weight;
        }
    }
    static int find(int[] parent, int x) {
        if (parent[x] != x) parent[x] = find(parent, parent[x]);
        return parent[x];
    }
    static void union(int[] parent, int x, int y) {
        int parX = find(parent, x);
        int parY = find(parent, y);
        parent[parX] = parY;
    }
    public static void main(String[] args) {
        int V = 5;
        Edge[] edges = {
            new Edge(0, 1, 2),
            new Edge(1, 2, 3),
            new Edge(0, 3, 6),
            new Edge(1, 3, 8),
            new Edge(1, 4, 5)
        };
        Arrays.sort(edges);
        int[] parent = new int[V];
        for (int i = 0; i < V; i++) parent[i] = i;

        int totalWeight = 0;
        for (Edge e : edges) {
            if (find(parent, e.u) != find(parent, e.v)) {
                totalWeight += e.weight;
                union(parent, e.u, e.v);
            }
```

```
        }
        System.out.println("Total weight of MST: " + totalWeight);
    }
}
```

INPUT:
Number of vertices = 6
Edges[] = [[0, 1, 4], [0, 2, 3], [1, 2, 1], [1, 3, 2], [2, 3, 4], [2, 4, 5], [3, 4, 6]]
OUTPUT:
Total Weight of MST: 12

EX.NO: 5 (C)	Implementation of Greedy Algorithms
DATE: 18.02.2025	

QUESTION: Find the shortest path from a source vertex to all other vertices.

AIM:
To implement Dijkstra's Algorithm using the Greedy method to find the shortest path from a given source vertex to all other vertices in a weighted graph with non-negative edge weights.

ALGORITHM:
Step 1: Set all distances to infinity, source to 0.
Step 2: Use a priority queue (min-heap) to pick the vertex with the smallest distance.
Step 3: For the current node, update the distance of all unvisited neighbors.
Step 4: If a shorter path is found, update and push it to the queue.
Step 5: Repeat until all nodes are processed.

CODING:

```
import java.util.*;
class Dijkstra {
    static class Pair {
        int vertex, weight;
        Pair(int v, int w) {
            vertex = v; weight = w;
        }
    }
    public static void dijkstra(int V, List<List<Pair>> adj, int src) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[src] = 0;
        PriorityQueue<Pair> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.weight));
        pq.add(new Pair(src, 0));
        while (!pq.isEmpty()) {
            Pair curr = pq.poll();
            for (Pair neighbor : adj.get(curr.vertex)) {
                if (dist[curr.vertex] + neighbor.weight < dist[neighbor.vertex]) {
                    dist[neighbor.vertex] = dist[curr.vertex] + neighbor.weight;
                    pq.add(new Pair(neighbor.vertex, dist[neighbor.vertex]));
                }
            }
        }
        System.out.println("Shortest distances from source " + src + ":");
        for (int i = 0; i < V; i++) {
            System.out.println("To " + i + " -> " + dist[i]);
        }
    }
    public static void main(String[] args) {
        int V = 5;
        List<List<Pair>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
        adj.get(0).add(new Pair(1, 10));
        adj.get(0).add(new Pair(4, 5));
        adj.get(1).add(new Pair(2, 1));
        adj.get(1).add(new Pair(4, 2));
        adj.get(2).add(new Pair(3, 4));
        adj.get(3).add(new Pair(2, 6));
```

```
        adj.get(4).add(new Pair(1, 3));
        adj.get(4).add(new Pair(2, 9));
        adj.get(4).add(new Pair(3, 2));
        dijkstra(V, adj, 0);
    }
}
```

INPUT:
Number of vertices = 6
Edge[] = [[0,1,7],[0,2,9],[0,3,14],[1,2,10],[1,4,15],[2,3,11],[2,4,2],[3,5,9],[4,5,6]]
Source vertex = 0
OUTPUT:
Shortest distances from source 0:
To 0 -> 0
To 1 -> 7
To 2 -> 9
To 3 -> 14
To 4 -> 11
To 5 -> 17

EX.NO: 5 (D)	Implementation of Greedy Algorithms
DATE: 18.02.2025	

QUESTION: Generate prefix codes for a given set of characters and their corresponding frequencies.

AIM:
To implement Huffman Coding using the Greedy Algorithm to achieve efficient data compression by assigning variable-length codes to input characters based on their frequencies.

ALGORITHM:
Step 1: Create a min-heap and insert all characters with their frequencies.
Step 2: While there is more than one node in the heap: Remove two nodes with the smallest frequencies.
Step 3: Repeat until one node remains (the root of the Huffman Tree).
Step 4: Traverse the tree to assign binary codes to each character: Left edge = '0', Right edge = '1'.
Step 5: Print the Huffman codes for each character.

CODING:
```
import java.util.*;
class HuffmanNode {
    int data;
    char c;
    HuffmanNode left, right;
}
class MyComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode x, HuffmanNode y) {
        return x.data - y.data;
    }
}
public class HuffmanCoding {
    public static void printCode(HuffmanNode root, String s) {
        if (root.left == null && root.right == null && Character.isLetter(root.c)) {
            System.out.println(root.c + ": " + s);
            return;
        }
        printCode(root.left, s + "0");
        printCode(root.right, s + "1");
    }
    public static void main(String[] args) {
        int n = 6;
        char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
        int[] charFreq = {5, 9, 12, 13, 16, 45};
        PriorityQueue<HuffmanNode> q = new PriorityQueue<>(n, new MyComparator());
        for (int i = 0; i < n; i++) {
            HuffmanNode hn = new HuffmanNode();
            hn.c = charArray[i];
            hn.data = charFreq[i];
            hn.left = null;
            hn.right = null;
            q.add(hn);
        }
        HuffmanNode root = null;
        while (q.size() > 1) {
            HuffmanNode x = q.poll();
            HuffmanNode y = q.poll();
            HuffmanNode f = new HuffmanNode();
            f.data = x.data + y.data;
            f.c = '-';
```

```
            f.left = x;
            f.right = y;
            root = f;
            q.add(f);
        }
        System.out.println("Huffman Codes are:");
        printCode(root, "");
    }
}
```

INPUT:
Characters: p, q, r, s, t
Frequencies: 15, 30, 5, 10, 25
OUTPUT:
Huffman Codes are:
q: 00
p: 01
t: 10
s: 110
r: 111

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:
Thus, the implementation of several program such as Tree Algorithms was successfully executed and output has been verified.

EX.NO: 6 (A)	Implementation of Dynamic Programming Techniques
DATE: 04.03.2025	

QUESTION: Given n items with weights w[i] and values v[i], and a knapsack capacity W, find the maximum value that can be put in a knapsack of capacity W.

AIM:
To implement the 0/1 Knapsack Problem using Dynamic Programming approaches: Memoization and Tabulation.

ALGORITHM:
Step 1: Initialize a 2D DP table dp[n+1][W+1] to store maximum values.
Step 2: Iterate over items (i from 1 to n) and weights (j from 1 to W).
Step 3: If w[i-1] <= j, then take dp[i][j] = max(dp[i-1][j], val[i-1] + dp[i-1][j - wt[i-1]]).
Step 4: Else, dp[i][j] = dp[i-1][j].
Step 5: Return dp[n][W].

CODING:
```
public class Knapsack {
    public static int knapsack(int[] wt, int[] val, int W, int n) {
        int[][] dp = new int[n + 1][W + 1];
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                if (i == 0 || w == 0) dp[i][w] = 0;
                else if (wt[i - 1] <= w)
                    dp[i][w] = Math.max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
                else
                    dp[i][w] = dp[i - 1][w];
            }
        }
        return dp[n][W];
    }
    public static void main(String[] args) {
        int[] wt = {1, 3, 4, 5};
        int[] val = {10, 40, 50, 70};
        int W = 8;
        System.out.println("Maximum value: " + knapsack(wt, val, W, wt.length));
    }
}
```

INPUT:
Weights = [2, 3, 4, 5];
Values = [3, 4, 5, 6];
Capacity = 5;
OUTPUT:
Maximum value: 7

EX.NO: 6 (B)	Implementation of Dynamic Programming Techniques
DATE: 04.03.2025	

QUESTION: Given a directed graph, find the transitive closure using Warshall's Algorithm.

AIM:
To determine the transitive closure of a graph using Warshall's Algorithm.

ALGORITHM:
Step 1: Input adjacency matrix of graph.
Step 2: For each intermediate vertex k, update path[i][j] = path[i][j] || (path[i][k] && path[k][j]).

CODING:
```
public class Warshall {
    static int V = 4;
    void transitiveClosure(int[][] graph) {
        int[][] reach = new int[V][V];
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                reach[i][j] = graph[i][j];
        for (int k = 0; k < V; k++)
            for (int i = 0; i < V; i++)
                for (int j = 0; j < V; j++)
                    reach[i][j] = (reach[i][j] != 0 || (reach[i][k] != 0 && reach[k][j] != 0)) ? 1 : 0;
        System.out.println("Transitive closure:");
        for (int i = 0; i < V; i++, System.out.println())
            for (int j = 0; j < V; j++)
                System.out.print(reach[i][j] + " ");
    }
    public static void main(String[] args) {
        int[][] graph = {
            {1, 1, 0, 1},
            {0, 1, 1, 0},
            {0, 0, 1, 1},
            {0, 0, 0, 1}
        };
        new Warshall().transitiveClosure(graph);
    }
}
```

INPUT:
V=4

OUTPUT:
Transitive closure:
1 1 0 1
0 1 0 1
0 0 1 1
0 0 0 1

EX.NO: 6 (C)	Implementation of Dynamic Programming Techniques
DATE: 04.03.2025	

QUESTION: Given a weighted directed graph, find the shortest paths between all pairs of vertices.

AIM:
To implement Floyd-Warshall's algorithm to find the shortest distance between all pairs of nodes.

ALGORITHM:
Step 1: Initialize dist[i][j] = weight[i][j] if there's an edge, or ∞ otherwise.
Step 2: For each intermediate k, update dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]).

CODING:
```
public class FloydWarshall {
    final static int INF = 99999, V = 4;
    void floyd(int[][] graph) {
        int[][] dist = new int[V][V];
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                dist[i][j] = graph[i][j];
        for (int k = 0; k < V; k++)
            for (int i = 0; i < V; i++)
                for (int j = 0; j < V; j++)
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
        System.out.println("Shortest distances:");
        for (int i = 0; i < V; i++, System.out.println())
            for (int j = 0; j < V; j++)
                System.out.print((dist[i][j] == INF ? "INF" : dist[i][j]) + " ");
    }
    public static void main(String[] args) {
        int[][] graph = {
            {0, 5, INF, 10},
            {INF, 0, 3, INF},
            {INF, INF, 0, 1},
            {INF, INF, INF, 0}
        };
        new FloydWarshall().floyd(graph);
    }
}
```

INPUT:
INF = 99999, V = 4;

OUTPUT:
Shortest distances:
0 3 5 6
5 0 2 3
3 6 0 1
2 5 7 0

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:

Thus, the implementation of several program such as Dynamic Programming Techniques was successfully executed and output has been verified.

EX.NO: 7	Implementation of Segment Trees
DATE: 18.03.2025	

QUESTION: You are given two arrays of integers, fruits and baskets, each of length n, where fruits[i] represents the quantity of the i^{th} type of fruit, and baskets[j] represents the capacity of the j^{th} basket.

AIM:
To implement an efficient algorithm to check whether each type of fruit can be placed into a basket such that no basket overflows its capacity and each basket is used for only one type of fruit.

ALGORITHM:
Step 1: Sort the fruits array in descending order.
Step 2: Sort the baskets array in descending order.
Step 3: Traverse both arrays; for each i, check if fruits[i] ≤ baskets[i].
Step 4: If any fruits[i] > baskets[i], return "Not Possible"; otherwise, return "Possible".

CODING:
```
class Solution {
    public int numOfUnplacedFruits(int[] f, int[] b) {
        int ans = 0;

        for(int i = 0; i < f.length; i++) {
            for(int j = 0; j < b.length; j++) {
                if(f[i] <= b[j]){
                    b[j] = 0;
                    ans++;
                    break;
                }
            }
        }
        return f.length - ans;
    }
}
```

INPUT:
fruits = [2, 3, 4, 5]
baskets = [1, 2, 3, 6]
OUTPUT:
1

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:

Thus, the implementation of several program such as Segment Trees Techniques was successfully executed and output has been verified.

EX.NO: 8	Implementation of Tries
DATE: 25.03.2025	

QUESTION: The Trie (Prefix Tree) data structure to efficiently insert, search, and check prefix matching of strings using node-based representation.

AIM:
To implement a Trie (Prefix Tree) data structure to efficiently insert, search, and check prefix matching of strings using node-based representation.

ALGORITHM:
Step 1: Define a TrieNode class with an array of children (size 26 for lowercase letters) and a boolean isEndOfWord.
Step 2: Initialize the root node in the Trie constructor.
Step 3: In the insert() method, iterate through characters of the word, creating child nodes if they don't exist.
Step 4: In the search() method, traverse the Trie using characters of the word. Return false if any node is missing; else return true if the last node is isEndOfWord = true.
Step 5: In the startsWith() method, similarly traverse using the prefix characters and return true if traversal is successful.

CODING:
```
class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEndOfWord = false;
}
public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }
    public void insert(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int i = ch - 'a';
            if (node.children[i] == null)
                node.children[i] = new TrieNode();
            node = node.children[i];
        }
        node.isEndOfWord = true;
    }
    public boolean search(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int i = ch - 'a';
            if (node.children[i] == null)
                return false;
            node = node.children[i];
        }
        return node.isEndOfWord;
    }
    public boolean startsWith(String prefix) {
        TrieNode node = root;
        for (char ch : prefix.toCharArray()) {
            int i = ch - 'a';
            if (node.children[i] == null)
                return false;
            node = node.children[i];
```

```
        }
        return true;
    }
}
```

INPUT:
Operations: ["Trie", "insert", "insert", "search", "startsWith", "search", "startsWith"]
Arguments : [[], ["cat"], ["cap"], ["cap"], ["ca"], ["can"], ["dog"]]
OUTPUT:
[null, null, null, true, true, false, false]

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:
Thus, the implementation of several program such as Tries was successfully executed and output has been verified.

EX.NO: 9	Implementation of Game Theory
DATE: 22.04.2025	

QUESTION: Two players, **P1 (First)** and **P2 (Second)**, are playing a game with a starting number of stones n. They play optimally, and P1 always starts first. The rules of the game are: In a single move, a player can remove either **2**, **3**, or **5** stones. The player who cannot make a move **loses**. Given the number of stones n, determine **who will win** the game if both players play optimally.

AIM:
To determine the winner of the game based on the optimal moves by each player using dynamic programming and the principles of combinatorial game theory.

ALGORITHM:
Step 1: Initialize a boolean array dp[0...n] to store winning/losing states.
Step 2: Set dp[0] = false as no stones means the current player loses.
Step 3: Loop i from 1 to n.
Step 4: Set dp[i] = true if any of dp[i−2], dp[i−3], or dp[i−5] is false.
Step 5: If dp[n] is true, print "First"; else, print "Second".

CODING:
```
import java.util.*;
public class GameOfStones {
   public static String gameOfStones(int n) {
      boolean[] dp = new boolean[n + 1];
      dp[0] = false;
      for (int i = 1; i <= n; i++) {
         if (i >= 2 && !dp[i - 2]) dp[i] = true;
         else if (i >= 3 && !dp[i - 3]) dp[i] = true;
         else if (i >= 5 && !dp[i - 5]) dp[i] = true;
         else dp[i] = false;
      }
      return dp[n] ? "First" : "Second";
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int t = sc.nextInt();
      while (t-- > 0) {
         int n = sc.nextInt();
         System.out.println(gameOfStones(n));
      }
      sc.close();
   }
}
```

INPUT:
4 5 6 8 10
OUTPUT:
First
First
First
Second
First

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:
Thus, the implementation of Game Theory was successfully executed and output has been verified.

EX.NO: 10a)	Implementation of NP Complete Problems
DATE: 06.05.2025	

QUESTION: Given an undirected graph with n vertices and m edges, your task is to determine if a Hamiltonian path exists in the graph.

AIM:
To design and implement an algorithm to determine the existence of a Hamiltonian Path in a given undirected graph using backtracking.

ALGORITHM:
Step 1: Represent the graph using an adjacency list or matrix.
Step 2: Start from each vertex and apply backtracking to find a path that visits all vertices exactly once.
Step 3: Maintain a visited array and path list.
Step 4: At each step, move to an unvisited adjacent vertex and recursively explore further.
Step 5: If the path includes all vertices, return true (Hamiltonian Path found).
Step 6: If no such path is found from any starting vertex, return false.

CODING:

```
class Solution
{
    boolean check(int N, int M, ArrayList<ArrayList<Integer>> Edges)
    {
        List<List<Integer>> list=new ArrayList<>();
        for(int i=0;i<=N;i++){
            list.add(new ArrayList<>());
        }
        for(ArrayList<Integer> i:Edges){
            list.get(i.get(0)).add(i.get(1));
            list.get(i.get(1)).add(i.get(0));
        }
        for(int i=1;i<=N;i++){
            Set<Integer> vis=new HashSet<>();
            if(!vis.contains(i)){
                if(dfs(i,N,list,vis)){
                    return true;
                }
            }
        }
        return false;
    }
    boolean dfs(int src,int total,List<List<Integer>> list,Set<Integer> vis){

        vis.add(src);
        if(vis.size()==total){
            return true;
        }
        for(Integer nbr:list.get(src)){
            if(!vis.contains(nbr)){
                if(dfs(nbr,total,list,vis)) return true;
            }
        }
        vis.remove(src);
        return false;
    }
}
```

INPUT:
n = 4, m = 3
edges[][] = { {1, 2}, {2, 3}, {3, 1} }
OUTPUT:
0

EX.NO: 10b)	Implementation of NP Complete Problems
DATE:	

QUESTION: Given a matrix cost of size n where cost[i][j] denotes the cost of moving from city i to city j. Your task is to complete a tour from city 0 (0-based index) to all other cities such that you visit each city exactly once and then at the end come back to city 0 at minimum cost.

AIM:
To implement a solution using Dynamic Programming with Bitmasking to find the minimum cost Hamiltonian cycle (Travelling Salesman Problem) for a given cost matrix.

ALGORITHM:
Step 1: Let dp[mask][i] be the minimum cost to reach city i after visiting all cities in mask.
Step 2: Initialize dp[1<<0][0] = 0 as starting from city 0.
Step 3: For every state (mask) and city (i), try moving to all unvisited cities (j).
Step 4: Update dp[mask | (1 << j)][j] = min(dp[mask | (1 << j)][j], dp[mask][i] + cost[i][j]).
Step 5: The final answer is min(dp[fullMask][i] + cost[i][0]) for all i, where fullMask = (1 << n) - 1.
Step 6: Return the minimum cost of completing the tour.

CODING:
```
class Solution {
    int[][] dp;
    int n;
    public int tsp(int[][] cost) {
        this.n = cost.length;
        int size = 1 << n;
        dp = new int[size][n];
        for (int[] row : dp)
            Arrays.fill(row, -1);
        return helper(1, 0, cost);
    }
    private int helper(int mask, int pos, int[][] cost) {
        if (mask == (1 << n) - 1) {
            return cost[pos][0];
        }
        if (dp[mask][pos] != -1)
            return dp[mask][pos];
        int minCost = Integer.MAX_VALUE;
        for (int city = 0; city < n; city++) {
            if ((mask & (1 << city)) == 0) {
                int newMask = mask | (1 << city);
                int newCost = cost[pos][city] + helper(newMask, city, cost);
                minCost = Math.min(minCost, newCost);
            }
        }
        return dp[mask][pos] = minCost;
    }
}
```
INPUT:
```
cost = [
    [0, 29, 20, 21],
    [29, 0, 15, 17],
    [20, 15, 0, 28],
    [21, 17, 28, 0]
]
```
OUTPUT:
69

S.NO	Particulars	Marks Allocated	Marks Obtained
1	Program / Simulation	40	
2	Program Execution	30	
3	Result	20	
4	Viva Voce	10	
Total		**100**	

RESULT:
Thus, the implementation of NP Complete Problem was successfully executed and output has been verified.