**MODULE 1**

**1. Apply brute force technique to recursively implement the following**

* 1. **Linear Search**
  2. **To find the maximum and minimum from a given list of n elements.**

**a. Linear Search**

import java.util.Scanner;

class Reclinear

{

 public static int search(int arr[], int N, int key, int i)

 {

 if (i >= N)

 return -1;

 if (arr[i] == key)

 return i;

 return search(arr, N, key, i + 1);

 }

 public static void main(String args[])

 {

 Scanner sc = new Scanner(System.in);

 System.out.print("Enter the size of the array: ");

 int N = sc.nextInt();

 int[] arr = new int[N];

 System.out.println("Enter the elements of the array: ");

 for (int i = 0; i < N; i++)

 arr[i] = sc.nextInt();

 System.out.print("Enter the element to search: ");

 int key = sc.nextInt();

 int result = search(arr, N, key, 0);

 if (result == -1)

 System.out.println("Element is not present in the array");

 else

 System.out.println("Element is present at index " + result);

 sc.close();

 }

}

**b. To find the maximum and minimum from a given list of n elements import**

java.util.Scanner;

public class Findminmax

{

public static void main(String[] args)

 {

int n;

int a[]=new int[20];

 System.out.println("Enter the number of elements");

 Scanner sc = new Scanner(System.in);

 n = sc.nextInt();

 System.out.println("Enter the array elements");

 for(int i=0; i<n; i++)

 a[i]=sc.nextInt();

 int max = recMax(a,n);

 int min=recMin(a,n);

 System.out.println("Maximum element: " + max);

 System.out.println("Minimum element: " + min);

}

static int rMax(int[] a, int n)

{

 if (n == 1)

 return a[0];

 int max=recMax(a, n-1);

 return (max>a[n - 1]? max: a[n - 1]);

}

static int recMin(int[] a, int n)

{

 if (n == 1)

 return a[0];

 int min=recMin(a, n-1);

 return (min< a[n - 1] ?min: a[n - 1]);

}

}

**2. There are 5 books in the shelf, find the number of ways to select 3 books from 5 books on the shelf using the NCR with recursion.**

public class NCR

{

        public static int fact(int n)

        {

          if (n == 0 || n == 1)

             return 1;

         return n \* fact(n - 1);

    }

    public static int com(int n, int r)

   {

        if (r > n)

            return 0;

        return fact(n) / (fact(r) \* fact(n - r));

    }

    public static void main(String[] args)

   {

        int n = 5;

        int r = 3;

        int ways = com(n, r);

**3.** **Find the next three terms of the sequence 15, 23, 38, 61, … Fibonacci series of the given number using recursion.**

import java.util.Scanner;

public class FIB

{

public static int fib(int n)

{

if (n == 1)

return 15;

if (n == 2)

return 23;

return fib(n - 1) + fib(n - 2);

}

public static void main(String[] args)

{

Scanner s= new Scanner(System.in);

System.out.print("Enter the number of terms to generate: ");

int n = s.nextInt();

System.out.println("Fibonacci Sequence:");

for (int i = 1; i <= n; i++)

System.out.print(fib(i) + " ");

s.close();

}

}

**MODULE 2**

**1. Implement the Selection sort algorithm.**

import java.util.Scanner;

public class SelectionSortInteractive

{

    public static void main(String[] args)

    {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");

        int n = scanner.nextInt();

        int[] A = new int[n];

        System.out.println("Enter the elements:");

        for (int i = 0; i < n; i++)

            A[i] = scanner.nextInt();

        for (int i = 0; i <= n - 2; i++)

       {

            int min = i, temp;

            for (int j = i + 1; j <= n - 1; j++)

                if (A[j] < A[min])

                       min = j;

            temp = A[i];

            A[i] = A[min];

            A[min] = temp;

        }

        System.out.println("Sorted array:");

        for (int num : A)

            System.out.print(num + " ");

        scanner.close();

    }

}

**2. Write a program to search a key in a given set of elements using Binary search method and find the time complexity required to find the key.**

import java.util.Scanner;

public class BinarySearch {

public static int binarySearch(int[] arr, int left, int right, int target) {

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target)

return mid;

if (arr[mid] > target)

right = mid - 1;

else // Search right half

left = mid + 1;

}

return -1; // Element not found

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of elements: ");

int n = scanner.nextInt();

int[] arr = new int[n];

System.out.println("Enter the sorted elements:");

for (int i = 0; i < n; i++) {

arr[i] = scanner.nextInt();

}

System.out.print("Enter the element to search: ");

int target = scanner.nextInt();

int result = binarySearch(arr, 0, n - 1, target);

if (result != -1)

System.out.println("Element found at index: " + result);

else

System.out.println("Element not found.");

scanner.close();

}

}

**3. Sort a given set of elements using Merge Sort method and determine the time required sort the elements. Plot a graph of number of elements versus time taken. Specify the time efficiency class of this algorithm.**

import java.util.Scanner;

public class MergeSortProgram

{

    public static void mergeSort(int[] arr, int low, int high)

   {

        if (low >= high)

            return;

        int mid = (low + high) / 2;

        mergeSort(arr, low, mid);

        mergeSort(arr, mid + 1, high);

        merge(arr, low, mid, high);

    public static void merge(int[] arr, int low, int mid, int high)

   {

        int n1 = mid - low + 1;

        int n2 = high - mid

        int[] L = new int[n1];

        int[] R = new int[n2]

        for (int i = 0; i < n1; i++)

            L[i] = arr[low + i];

        for (int j = 0; j < n2; j++)

            R[j] = arr[mid + 1 + j]

        int i = 0, j = 0, k = low;

        while (i < n1 && j < n2)

       {

            if (L[i] <= R[j])

           {

                arr[k] = L[i];

                i++;

            }

            else

           {

                arr[k] = R[j];

                j++;

            }

            k++;

        while (i < n1)

       {

            arr[k] = L[i];

            i++;

            k++;

        while (j < n2)

       {

            arr[k] = R[j];

            j++;

            k++;

        }

    public static void main(String[] args)

   {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements: ");

        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements:");

        for (int i = 0; i < n; i++)

            arr[i] = scanner.nextInt();

        long startTime = System.nanoTime();

        mergeSort(arr, 0, n - 1);

        long endTime = System.nanoTime();

        System.out.println("Sorted Array:");

        for (int num : arr)

            System.out.print(num + " ");

        long duration = (endTime - startTime) / 1000000; // Convert to milliseconds

        System.out.println("\nTime taken: " + duration + " ms");

        scanner.close();

    }

}

**4. Sort a given set of elements using Quick Sort method and determine the time required sort the elements. Plot a graph of number of elements versus time taken. Specify the time efficiency class of this algorithm.**

import java.util.Scanner;

import java.util.Random;

public class QuickSortProgram {

    static final int MAX = 50005;

    static int[] a = new int[MAX];

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.print("Enter Max array size: ");

        int n = input.nextInt();

        Random random = new Random();

        System.out.println("Generating random array elements: ");

        for (int i = 0; i < n; i++)

            a[i] = random.nextInt(1000);

        System.out.println("Input Array:");

        for (int i = 0; i < n; i++)

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

        System.out.println();

        long startTime = System.nanoTime();

        QuickSortAlgorithm(0, n - 1);

        long stopTime = System.nanoTime();

        long elapsedTime = stopTime - startTime;

        System.out.println("\nSorted Array:");

        for (int i = 0; i < n; i++)

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

        System.out.println();

        System.out.println("Time Complexity in ms for n=" + n + " is: " + (double) elapsedTime / 1000000);

        input.close();

    }

    public static void QuickSortAlgorithm(int L, int H)

  {

        if (L < H)

       {

            int i = L, j = H + 1, pivot = a[L];

            while (true)

           {

                do

              {

                    i++;

                } while (i < H && a[i] <= pivot);

                do

               {

                    j--;

                } while (j > L && a[j] > pivot);

                if (i < j)

               {

                    int temp = a[i];

                    a[i] = a[j];

                    a[j] = temp;

                }

                else

                {

                    break;

                }

            }

            a[L] = a[j];

            a[j] = pivot;

            QuickSortAlgorithm(L, j - 1);

            QuickSortAlgorithm(j + 1, H);

        }

    }

}

**5. Implement Topological sort using source removal method find the time required to sort the elements.**

import java.util.Scanner;

public class TopologicalSort {

static int[] temp = new int[10];

static int k = 0;

static void topo(int n, int[] indegree, int[][] a) {

for (int i = 1; i <= n; i++) {

if (indegree[i] == 0) {

indegree[i] = -1; // Mark as visited

temp[++k] = i; // Store in topological order

for (int j = 1; j <= n; j++) {

if (a[i][j] == 1)

indegree[j]--; // Reduce in-degree

}

i = 0; // Restart loop to check again

}

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int[][] a = new int[10][10]; // Adjacency matrix

int[] indegree = new int[10]; // Array to store in-degrees

System.out.print("Enter the number of vertices: ");

int n = sc.nextInt();

// Initialize indegree array

for (int i = 1; i <= n; i++)

indegree[i] = 0;

System.out.println("\nEnter the adjacency matrix (1-based indexing):");

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++) {

a[i][j] = sc.nextInt();

if (a[i][j] == 1)

indegree[j]++; // Increment in-degree for vertex j

}

}

long startTime = System.nanoTime();

topo(n, indegree, a);

long stopTime = System.nanoTime();

long elapsedTime = stopTime - startTime;

if (k != n)

System.out.println("Topological ordering is not possible.");

else {

System.out.println("\nTopological ordering is:");

for (int i = 1; i <= k; i++)

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

}

System.out.println("Time Complexity in ms for n=" + n + " is: " + (double) elapsedTime / 1000000);

sc.close();

}

}

**MODULE 3**

**1. Write a program to find maximum profit using Knapsack technique**

import java.util.\*;

public class Knapsack {

public static void main(String[] args) {

Scanner s1 = new Scanner(System.in);

System.out.print("Enter the number of objects: ");

int n = s1.nextInt();

double w[] = new double[n]; // Weights

double p[] = new double[n]; // Profits

double r[] = new double[n]; // Profit-to-weight ratios

System.out.println("Enter the object's weights:");

for (int i = 0; i < n; ++i) {

w[i] = s1.nextDouble();

}

System.out.println("Enter the object's profits:");

for (int i = 0; i < n; ++i) {

p[i] = s1.nextDouble();

r[i] = p[i] / w[i]; // Calculating profit-to-weight ratio

}

System.out.print("Enter the Capacity of the knapsack: ");

double m = s1.nextDouble();

s1.close(); // Closing scanner to prevent resource leak

// Sorting items based on profit-to-weight ratio in descending order

for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

if (r[i] < r[j]) {

// Swapping ratios

double temp = r[i];

r[i] = r[j];

r[j] = temp;

// Swapping profits

double tempProfit = p[i];

p[i] = p[j];

p[j] = tempProfit;

// Swapping weights

double tempWeight = w[i];

w[i] = w[j];

w[j] = tempWeight;

}

}

}

knap(n, m, p, w);

}

static void knap(int n, double m, double p[], double w[]) {

double profit = 0.0;

double x[] = new double[n]; // Fraction of items taken

double rc = m; // Remaining capacity

for (int i = 0; i < n; i++) {

if (rc == 0) {

break;

}

if (rc >= w[i]) {

x[i] = 1; // Take the whole item

} else {

x[i] = rc / w[i]; // Take fraction of the item

}

profit += x[i] \* p[i];

rc -= x[i] \* w[i];

}

System.out.println("Total profit is: " + profit);

}

}

**2.** **Implement Kruskal’s algorithm and Find Minimum Cost Spanning Tree of a given connected undirected graph.**

import java.util.\*;

public class Kruskal{

public static void main(String[] args){

int vertices=6;

int [][]edges={{0,1,4},{0,2,4},{1,2,2},{1,3,6},{2,3,8},{2,4,9},{3,4,7},{3,5,10},{4,5,5}};

Arrays.sort(edges,Comparator.comparingInt(e->e[2]));

int []parent=new int[vertices];

for(int i=0;i<vertices;i++) parent[i]=i;

int mstweight=0;

System.out.println("Edges in Maximum spanning tree: ");

for(int []edge:edges){

int u=edge[0],v=edge[1],weight=edge[2];

if(find(parent,u)!=find(parent,v)){

union(parent,u,v);

mstweight+=weight;

System.out.println(u+"-"+v+": "+weight);

}

}

System.out.println("Total weight of MST: "+mstweight);

}

static int find(int []parent,int v){

if(parent[v]!=v) parent[v]=find(parent,parent[v]);

return parent[v];

}

static void union(int []parent,int u,int v){

int rootu=find(parent,u);

int rootv=find(parent,v);

parent[rootu]=rootv;

}

}

**OR**

public class Kruskals {

final static int MAX = 20;

static int n;

static int cost[][];

static Scanner scan = new Scanner(System.in);

public static void main(String[] args) {

ReadMatrix();

Kruskals();

}

static void ReadMatrix() {

int i, j;

cost = new int[MAX][MAX];

System.out.println("Implementation of Kruskal's algorithm:");

System.out.print("Enter the number of vertices: ");

n = scan.nextInt();

System.out.println("Enter the cost adjacency matrix:");

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

cost[i][j] = scan.nextInt();

if (cost[i][j] == 0) {

cost[i][j] = 999; // Assigning a high value to represent no direct edge

}

}

}

}

static void Kruskals() {

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

int parent[] = new int[MAX];

System.out.println("The edges of Minimum Cost Spanning Tree are:");

while (ne < n) {

min = 999;

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (cost[i][j] < min) {

min = cost[i][j];

a = u = i;

b = v = j;

}

}

}

while (parent[u] != 0) {

u = parent[u];

}

while (parent[v] != 0) {

v = parent[v];

}

if (u != v) {

System.out.println(ne++ + " edge (" + a + ", " + b + ") = " + min);

mincost += min;

parent[v] = u;

}

cost[a][b] = cost[b][a] = 999; // Marking edge as used

}

System.out.println("Minimum cost: " + mincost);

}

}