Q.1) Write a program to sort a list of n numbers in ascending order using selection sort and determine the time required to sort the elements .

import java.util.Scanner;

public class SelectionSort {

// Method to perform selection sort

public static void selectionSort(int[] arr) {

int n = arr.length;

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

int minIndex = i;

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

if (arr[j] < arr[minIndex]) {

minIndex = j; // Find the minimum element

}

}

// Swap the minimum element with the first unsorted element

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input size of the array

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

int n = scanner.nextInt();

int[] arr = new int[n];

// Input elements

System.out.println("Enter " + n + " numbers:");

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

arr[i] = scanner.nextInt();

}

// Measure sorting time

long startTime = System.nanoTime();

selectionSort(arr);

long endTime = System.nanoTime();

long duration = endTime - startTime;

// Display sorted array

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

for (int num : arr) {

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

}

// Display execution time

System.out.println("\nTime required for sorting (in nanoseconds): " + duration);

scanner.close();

}

}

Q.2) Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from a file or can be generated using the random number generator.

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

import java.util.Random;

public class QuickSortExperiment {

// Quick Sort algorithm

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

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

// Partition function for Quick Sort

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

// Swap elements

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

// Generate random array

public static int[] generateRandomArray(int n) {

Random rand = new Random();

int[] arr = new int[n];

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

arr[i] = rand.nextInt(1000); // Random numbers from 0-999

}

return arr;

}

// Read elements from a file

public static int[] readFromFile(String fileName, int n) throws FileNotFoundException {

int[] arr = new int[n];

Scanner scanner = new Scanner(new File(fileName));

for (int i = 0; i < n && scanner.hasNextInt(); i++) {

arr[i] = scanner.nextInt();

}

scanner.close();

return arr;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int n = scanner.nextInt();

System.out.print("Choose data source - 1: File | 2: Random Generator: ");

int choice = scanner.nextInt();

int[] arr = new int[n];

try {

if (choice == 1) {

System.out.print("Enter file name: ");

String fileName = scanner.next();

arr = readFromFile(fileName, n);

} else {

arr = generateRandomArray(n);

}

} catch (FileNotFoundException e) {

System.out.println("File not found. Using random generator instead.");

arr = generateRandomArray(n);

}

long startTime = System.nanoTime();

quickSort(arr, 0, n - 1);

long endTime = System.nanoTime();

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

for (int num : arr) {

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

}

System.out.println("\nTime required for sorting (nanoseconds): " + (endTime - startTime));

scanner.close();

}

}

Q.1) Write a program to sort n randomly generated elements using Heapsort method.

import java.util.Random;

import java.util.Scanner;

public class HeapSort {

// Heap Sort function

public static void heapSort(int[] arr) {

int n = arr.length;

// Build max heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// Extract elements one by one from heap

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

// Swap root (largest) with last element

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Heapify reduced heap

heapify(arr, i, 0);

}

}

// Heapify function

private static void heapify(int[] arr, int n, int i) {

int largest = i; // Root

int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

// Recursively heapify affected subtree

heapify(arr, n, largest);

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int n = scanner.nextInt();

int[] arr = new int[n];

// Generate random numbers

Random rand = new Random();

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

arr[i] = rand.nextInt(1000); // Random numbers between 0-999

}

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

for (int num : arr) {

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

}

// Measure sorting time

long startTime = System.nanoTime();

heapSort(arr);

long endTime = System.nanoTime();

long duration = endTime - startTime;

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

for (int num : arr) {

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

}

System.out.println("\nTime required for sorting (nanoseconds): " + duration);

scanner.close();

}

}

2 **Write a program to implement Strassen’s Matrix multiplication**

import java.util.Scanner;

public class StrassenMatrixMultiplication {

// Method to multiply matrices using Strassen's Algorithm

public static int[][] strassenMultiply(int[][] A, int[][] B) {

int n = A.length;

int[][] result = new int[n][n];

// Base case: when matrix is 1x1

if (n == 1) {

result[0][0] = A[0][0] \* B[0][0];

return result;

}

int newSize = n / 2;

int[][] A11 = new int[newSize][newSize];

int[][] A12 = new int[newSize][newSize];

int[][] A21 = new int[newSize][newSize];

int[][] A22 = new int[newSize][newSize];

int[][] B11 = new int[newSize][newSize];

int[][] B12 = new int[newSize][newSize];

int[][] B21 = new int[newSize][newSize];

int[][] B22 = new int[newSize][newSize];

// Splitting matrices into submatrices

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

for (int j = 0; j < newSize; j++) {

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + newSize];

A21[i][j] = A[i + newSize][j];

A22[i][j] = A[i + newSize][j + newSize];

B11[i][j] = B[i][j];

B12[i][j] = B[i][j + newSize];

B21[i][j] = B[i + newSize][j];

B22[i][j] = B[i + newSize][j + newSize];

}

}

// Computing Strassen's multiplication intermediate values

int[][] M1 = strassenMultiply(addMatrices(A11, A22), addMatrices(B11, B22));

int[][] M2 = strassenMultiply(addMatrices(A21, A22), B11);

int[][] M3 = strassenMultiply(A11, subtractMatrices(B12, B22));

int[][] M4 = strassenMultiply(A22, subtractMatrices(B21, B11));

int[][] M5 = strassenMultiply(addMatrices(A11, A12), B22);

int[][] M6 = strassenMultiply(subtractMatrices(A21, A11), addMatrices(B11, B12));

int[][] M7 = strassenMultiply(subtractMatrices(A12, A22), addMatrices(B21, B22));

// Computing result matrix using Strassen's formula

int[][] C11 = addMatrices(subtractMatrices(addMatrices(M1, M4), M5), M7);

int[][] C12 = addMatrices(M3, M5);

int[][] C21 = addMatrices(M2, M4);

int[][] C22 = addMatrices(subtractMatrices(addMatrices(M1, M3), M2), M6);

// Combining submatrices into final result matrix

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

for (int j = 0; j < newSize; j++) {

result[i][j] = C11[i][j];

result[i][j + newSize] = C12[i][j];

result[i + newSize][j] = C21[i][j];

result[i + newSize][j + newSize] = C22[i][j];

}

}

return result;

}

// Method to add two matrices

private static int[][] addMatrices(int[][] A, int[][] B) {

int n = A.length;

int[][] result = new int[n][n];

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

for (int j = 0; j < n; j++)

result[i][j] = A[i][j] + B[i][j];

return result;

}

// Method to subtract two matrices

private static int[][] subtractMatrices(int[][] A, int[][] B) {

int n = A.length;

int[][] result = new int[n][n];

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

for (int j = 0; j < n; j++)

result[i][j] = A[i][j] - B[i][j];

return result;

}

// Main method for execution

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter matrix size (n x n, must be power of 2): ");

int n = scanner.nextInt();

int[][] A = new int[n][n];

int[][] B = new int[n][n];

// Read matrix A

System.out.println("Enter matrix A:");

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

for (int j = 0; j < n; j++)

A[i][j] = scanner.nextInt();

// Read matrix B

System.out.println("Enter matrix B:");

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

for (int j = 0; j < n; j++)

B[i][j] = scanner.nextInt();

// Measuring execution time

long startTime = System.nanoTime();

int[][] result = strassenMultiply(A, B);

long endTime = System.nanoTime();

// Displaying result

System.out.println("Resultant Matrix:");

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

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

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

}

System.out.println();

}

// Displaying execution time

System.out.println("Time required (nanoseconds): " + (endTime - startTime));

scanner.close();

}

}

1. **Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements**

import java.util.Scanner;

public class QuickSort {

// Quick Sort function

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

if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

// Partition function for Quick Sort

private static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

// Swap function

private static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input size of array

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

int n = scanner.nextInt();

int[] arr = new int[n];

// Input elements

System.out.println("Enter " + n + " numbers:");

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

arr[i] = scanner.nextInt();

}

// Measure sorting time

long startTime = System.nanoTime();

quickSort(arr, 0, n - 1);

long endTime = System.nanoTime();

long duration = endTime - startTime;

// Display sorted array

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

for (int num : arr) {

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

}

// Display execution time

System.out.println("\nTime required for sorting (nanoseconds): " + duration);

scanner.close();

}

}

**Q.2) Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prims algorithm**

import java.util.\*;

class PrimsAlgorithm {

// Function to find MST using Prim's Algorithm

public static void primMST(int[][] graph, int vertices) {

int[] parent = new int[vertices]; // Store MST

int[] key = new int[vertices]; // Minimum edge weight

boolean[] mstSet = new boolean[vertices]; // Track processed vertices

// Initialize keys with maximum value

Arrays.fill(key, Integer.MAX\_VALUE);

key[0] = 0; // Start from the first vertex

parent[0] = -1;

for (int count = 0; count < vertices - 1; count++) {

int u = minKey(key, mstSet, vertices);

mstSet[u] = true;

// Update key values for adjacent vertices

for (int v = 0; v < vertices; v++) {

if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

// Print the MST

printMST(parent, graph, vertices);

}

// Function to find the vertex with minimum key value

private static int minKey(int[] key, boolean[] mstSet, int vertices) {

int min = Integer.MAX\_VALUE, minIndex = -1;

for (int v = 0; v < vertices; v++) {

if (!mstSet[v] && key[v] < min) {

min = key[v];

minIndex = v;

}

}

return minIndex;

}

// Function to print the MST

private static void printMST(int[] parent, int[][] graph, int vertices) {

System.out.println("Edge \tWeight");

int totalWeight = 0;

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

System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);

totalWeight += graph[i][parent[i]];

}

System.out.println("Total Minimum Cost: " + totalWeight);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Input number of vertices

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

int vertices = scanner.nextInt();

int[][] graph = new int[vertices][vertices];

// Input adjacency matrix representation of graph

System.out.println("Enter adjacency matrix (0 for no edge):");

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

for (int j = 0; j < vertices; j++) {

graph[i][j] = scanner.nextInt();

}

}

// Measure execution time

long startTime = System.nanoTime();

primMST(graph, vertices);

long endTime = System.nanoTime();

System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));

scanner.close();

}

}

**Q.1) Write a program to implement a Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements**

**import java.util.Scanner;**

**public class MergeSort {**

**// Merge Sort function**

**public static void mergeSort(int[] arr, int left, int right) {**

**if (left < right) {**

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

**// Recursively sort left and right halves**

**mergeSort(arr, left, mid);**

**mergeSort(arr, mid + 1, right);**

**// Merge sorted halves**

**merge(arr, left, mid, right);**

**}**

**}**

**// Merge function to combine sorted subarrays**

**public static void merge(int[] arr, int left, int mid, int right) {**

**int n1 = mid - left + 1;**

**int n2 = right - mid;**

**int[] leftArr = new int[n1];**

**int[] rightArr = new int[n2];**

**for (int i = 0; i < n1; i++) {**

**leftArr[i] = arr[left + i];**

**}**

**for (int j = 0; j < n2; j++) {**

**rightArr[j] = arr[mid + 1 + j];**

**}**

**int i = 0, j = 0, k = left;**

**while (i < n1 && j < n2) {**

**if (leftArr[i] <= rightArr[j]) {**

**arr[k] = leftArr[i];**

**i++;**

**} else {**

**arr[k] = rightArr[j];**

**j++;**

**}**

**k++;**

**}**

**// Copy remaining elements, if any**

**while (i < n1) {**

**arr[k] = leftArr[i];**

**i++;**

**k++;**

**}**

**while (j < n2) {**

**arr[k] = rightArr[j];**

**j++;**

**k++;**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input size of array**

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

**int n = scanner.nextInt();**

**int[] arr = new int[n];**

**// Input elements**

**System.out.println("Enter " + n + " numbers:");**

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

**arr[i] = scanner.nextInt();**

**}**

**// Measure sorting time**

**long startTime = System.nanoTime();**

**mergeSort(arr, 0, n - 1);**

**long endTime = System.nanoTime();**

**long duration = endTime - startTime;**

**// Display sorted array**

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

**for (int num : arr) {**

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

**}**

**// Display execution time**

**System.out.println("\nTime required for sorting (nanoseconds): " + duration);**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to implement Knapsack problems using Greedy method**

**import java.util.Arrays;**

**import java.util.Scanner;**

**class Item {**

**int weight, value;**

**double ratio; // Value-to-weight ratio**

**Item(int value, int weight) {**

**this.value = value;**

**this.weight = weight;**

**this.ratio = (double) value / weight;**

**}**

**}**

**public class GreedyKnapsack {**

**public static double fractionalKnapsack(Item[] items, int capacity) {**

**// Sort items in descending order of value-to-weight ratio**

**Arrays.sort(items, (a, b) -> Double.compare(b.ratio, a.ratio));**

**double maxValue = 0.0; // Maximum value achieved**

**int currentWeight = 0; // Current knapsack weight**

**for (Item item : items) {**

**if (currentWeight + item.weight <= capacity) {**

**// Take the full item**

**currentWeight += item.weight;**

**maxValue += item.value;**

**} else {**

**// Take a fraction of the item**

**int remainingWeight = capacity - currentWeight;**

**maxValue += item.ratio \* remainingWeight;**

**break; // Knapsack is full**

**}**

**}**

**return maxValue;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// User input**

**System.out.print("Enter number of items: ");**

**int n = scanner.nextInt();**

**Item[] items = new Item[n];**

**System.out.println("Enter value and weight of each item:");**

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

**int value = scanner.nextInt();**

**int weight = scanner.nextInt();**

**items[i] = new Item(value, weight);**

**}**

**System.out.print("Enter knapsack capacity: ");**

**int capacity = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**double maxValue = fractionalKnapsack(items, capacity);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Maximum value obtained: " + maxValue);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program for the Implementation of Kruskal’s algorithm to find minimum cost spanning tree.**

**import java.util.\*;**

**class Edge implements Comparable<Edge> {**

**int src, dest, weight;**

**public Edge(int src, int dest, int weight) {**

**this.src = src;**

**this.dest = dest;**

**this.weight = weight;**

**}**

**// Sorting edges based on weight**

**public int compareTo(Edge other) {**

**return this.weight - other.weight;**

**}**

**}**

**class DisjointSet {**

**int[] parent, rank;**

**public DisjointSet(int vertices) {**

**parent = new int[vertices];**

**rank = new int[vertices];**

**// Initialize each vertex as its own parent**

**for (int i = 0; i < vertices; i++) {**

**parent[i] = i;**

**rank[i] = 0;**

**}**

**}**

**// Find representative of a set**

**public int find(int vertex) {**

**if (parent[vertex] != vertex) {**

**parent[vertex] = find(parent[vertex]); // Path compression**

**}**

**return parent[vertex];**

**}**

**// Union of two sets by rank**

**public void union(int root1, int root2) {**

**if (rank[root1] > rank[root2]) {**

**parent[root2] = root1;**

**} else if (rank[root1] < rank[root2]) {**

**parent[root1] = root2;**

**} else {**

**parent[root2] = root1;**

**rank[root1]++;**

**}**

**}**

**}**

**public class KruskalsAlgorithm {**

**public static void kruskalMST(List<Edge> edges, int vertices) {**

**Collections.sort(edges); // Sort edges by weight**

**DisjointSet ds = new DisjointSet(vertices);**

**List<Edge> mst = new ArrayList<>();**

**int totalCost = 0;**

**for (Edge edge : edges) {**

**int root1 = ds.find(edge.src);**

**int root2 = ds.find(edge.dest);**

**// If they belong to different sets, include edge in MST**

**if (root1 != root2) {**

**mst.add(edge);**

**totalCost += edge.weight;**

**ds.union(root1, root2);**

**}**

**}**

**// Printing MST**

**System.out.println("Edges in Minimum Cost Spanning Tree:");**

**for (Edge edge : mst) {**

**System.out.println(edge.src + " - " + edge.dest + " (Weight: " + edge.weight + ")");**

**}**

**System.out.println("Total Cost of MST: " + totalCost);**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

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

**int vertices = scanner.nextInt();**

**System.out.print("Enter number of edges: ");**

**int edgesCount = scanner.nextInt();**

**List<Edge> edges = new ArrayList<>();**

**System.out.println("Enter edges (src dest weight):");**

**for (int i = 0; i < edgesCount; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**int weight = scanner.nextInt();**

**edges.add(new Edge(src, dest, weight));**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**kruskalMST(edges, vertices);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to implement Huffman Code using greedy methods and also calculate the best case and worst-case complexity.**

**import java.util.PriorityQueue;**

**import java.util.Scanner;**

**// Node class representing the Huffman tree**

**class HuffmanNode implements Comparable<HuffmanNode> {**

**char data;**

**int frequency;**

**HuffmanNode left, right;**

**HuffmanNode(char data, int frequency) {**

**this.data = data;**

**this.frequency = frequency;**

**}**

**@Override**

**public int compareTo(HuffmanNode other) {**

**return this.frequency - other.frequency;**

**}**

**}**

**public class HuffmanCoding {**

**// Function to build Huffman Tree using Greedy Algorithm**

**public static HuffmanNode buildHuffmanTree(char[] chars, int[] freq) {**

**PriorityQueue<HuffmanNode> minHeap = new PriorityQueue<>();**

**// Create leaf nodes and add them to the priority queue**

**for (int i = 0; i < chars.length; i++) {**

**minHeap.add(new HuffmanNode(chars[i], freq[i]));**

**}**

**while (minHeap.size() > 1) {**

**// Extract two minimum frequency nodes**

**HuffmanNode left = minHeap.poll();**

**HuffmanNode right = minHeap.poll();**

**// Create a new node with combined frequency**

**HuffmanNode newNode = new HuffmanNode('-', left.frequency + right.frequency);**

**newNode.left = left;**

**newNode.right = right;**

**// Add new node to minHeap**

**minHeap.add(newNode);**

**}**

**return minHeap.poll(); // Root of Huffman Tree**

**}**

**// Function to generate Huffman Codes using DFS**

**public static void generateHuffmanCodes(HuffmanNode root, String code) {**

**if (root == null) {**

**return;**

**}**

**// If leaf node, print character and its code**

**if (root.data != '-') {**

**System.out.println(root.data + ": " + code);**

**}**

**// Recursive calls for left and right subtrees**

**generateHuffmanCodes(root.left, code + "0");**

**generateHuffmanCodes(root.right, code + "1");**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of characters**

**System.out.print("Enter number of characters: ");**

**int n = scanner.nextInt();**

**char[] chars = new char[n];**

**int[] freq = new int[n];**

**System.out.println("Enter characters and their frequencies:");**

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

**chars[i] = scanner.next().charAt(0);**

**freq[i] = scanner.nextInt();**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**HuffmanNode root = buildHuffmanTree(chars, freq);**

**long endTime = System.nanoTime();**

**System.out.println("Huffman Codes:");**

**generateHuffmanCodes(root, "");**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q-1) Write a program for the Implementation of Prim’s algorithm to find minimum cost spanning tree.**

**import java.util.Scanner;**

**public class PrimsAlgorithm {**

**private static final int INF = Integer.MAX\_VALUE; // Represents infinite weight**

**// Function to find the Minimum Cost Spanning Tree using Prim's Algorithm**

**public static void primMST(int[][] graph, int vertices) {**

**int[] key = new int[vertices]; // Stores minimum edge weight**

**int[] parent = new int[vertices]; // Stores MST structure**

**boolean[] mstSet = new boolean[vertices]; // Tracks processed vertices**

**// Initialize all keys as infinite**

**for (int i = 0; i < vertices; i++) {**

**key[i] = INF;**

**mstSet[i] = false;**

**}**

**// Start from the first vertex**

**key[0] = 0;**

**parent[0] = -1; // First node is always root of MST**

**for (int count = 0; count < vertices - 1; count++) {**

**int u = minKey(key, mstSet, vertices);**

**mstSet[u] = true;**

**// Update key values for adjacent vertices**

**for (int v = 0; v < vertices; v++) {**

**if (graph[u][v] != 0 && !mstSet[v] && graph[u][v] < key[v]) {**

**parent[v] = u;**

**key[v] = graph[u][v];**

**}**

**}**

**}**

**// Print the Minimum Cost Spanning Tree**

**printMST(parent, graph, vertices);**

**}**

**// Function to find the vertex with the minimum key value**

**private static int minKey(int[] key, boolean[] mstSet, int vertices) {**

**int min = INF, minIndex = -1;**

**for (int v = 0; v < vertices; v++) {**

**if (!mstSet[v] && key[v] < min) {**

**min = key[v];**

**minIndex = v;**

**}**

**}**

**return minIndex;**

**}**

**// Function to print the Minimum Cost Spanning Tree**

**private static void printMST(int[] parent, int[][] graph, int vertices) {**

**System.out.println("Edges in Minimum Cost Spanning Tree:");**

**int totalCost = 0;**

**for (int i = 1; i < vertices; i++) {**

**System.out.println(parent[i] + " - " + i + " (Weight: " + graph[i][parent[i]] + ")");**

**totalCost += graph[i][parent[i]];**

**}**

**System.out.println("Total Minimum Cost: " + totalCost);**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

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

**int vertices = scanner.nextInt();**

**int[][] graph = new int[vertices][vertices];**

**System.out.println("Enter adjacency matrix (0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**primMST(graph, vertices);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a Program to find only length of Longest Common Subsequence.**

**import java.util.Scanner;**

**public class LCSLength {**

**// Function to find the length of the Longest Common Subsequence**

**public static int findLCSLength(String s1, String s2) {**

**int m = s1.length();**

**int n = s2.length();**

**int[][] dp = new int[m + 1][n + 1];**

**// Filling the DP table**

**for (int i = 1; i <= m; i++) {**

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

**if (s1.charAt(i - 1) == s2.charAt(j - 1)) {**

**dp[i][j] = 1 + dp[i - 1][j - 1];**

**} else {**

**dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);**

**}**

**}**

**}**

**// Return the length of LCS**

**return dp[m][n];**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input strings**

**System.out.print("Enter first string: ");**

**String s1 = scanner.next();**

**System.out.print("Enter second string: ");**

**String s2 = scanner.next();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int lcsLength = findLCSLength(s1, s2);**

**long endTime = System.nanoTime();**

**// Display result**

**System.out.println("Length of Longest Common Subsequence: " + lcsLength);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q-1) Write a program for the Implementation of Dijkstra’s algorithm to find shortest path to other vertices**

**import java.util.\*;**

**class DijkstraAlgorithm {**

**public static void dijkstra(int[][] graph, int source) {**

**int vertices = graph.length;**

**int[] distance = new int[vertices]; // Stores shortest distances from source**

**boolean[] visited = new boolean[vertices]; // Tracks processed vertices**

**// Initialize distances to infinity, except source**

**Arrays.fill(distance, Integer.MAX\_VALUE);**

**distance[source] = 0;**

**// Priority queue to select vertex with the smallest known distance**

**PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.comparingInt(v -> distance[v]));**

**pq.add(source);**

**while (!pq.isEmpty()) {**

**int u = pq.poll(); // Extract the vertex with the smallest distance**

**visited[u] = true;**

**// Update distances for adjacent vertices**

**for (int v = 0; v < vertices; v++) {**

**if (graph[u][v] != 0 && !visited[v] && distance[u] + graph[u][v] < distance[v]) {**

**distance[v] = distance[u] + graph[u][v];**

**pq.add(v);**

**}**

**}**

**}**

**// Print shortest distances**

**System.out.println("Vertex \t Distance from Source " + source);**

**for (int i = 0; i < vertices; i++) {**

**System.out.println(i + " \t " + distance[i]);**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**int[][] graph = new int[vertices][vertices];**

**// Input adjacency matrix representation of graph**

**System.out.println("Enter adjacency matrix (0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Input source vertex**

**System.out.print("Enter source vertex: ");**

**int source = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**dijkstra(graph, source);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.3) Write a program for finding Topological sorting for Directed Acyclic Graph (DAG)**

**import java.util.\*;**

**public class TopologicalSort {**

**// Function to perform Topological Sorting using Kahn's Algorithm (BFS)**

**public static List<Integer> topologicalSort(int vertices, List<List<Integer>> adjacencyList) {**

**int[] inDegree = new int[vertices]; // Stores in-degree of each vertex**

**// Calculate in-degree of each vertex**

**for (List<Integer> neighbors : adjacencyList) {**

**for (int neighbor : neighbors) {**

**inDegree[neighbor]++;**

**}**

**}**

**// Queue to store vertices with in-degree 0**

**Queue<Integer> queue = new LinkedList<>();**

**for (int i = 0; i < vertices; i++) {**

**if (inDegree[i] == 0) {**

**queue.add(i);**

**}**

**}**

**List<Integer> topOrder = new ArrayList<>();**

**while (!queue.isEmpty()) {**

**int node = queue.poll();**

**topOrder.add(node);**

**// Reduce in-degree of adjacent vertices**

**for (int neighbor : adjacencyList.get(node)) {**

**inDegree[neighbor]--;**

**if (inDegree[neighbor] == 0) {**

**queue.add(neighbor);**

**}**

**}**

**}**

**// If the number of processed nodes isn't equal to total vertices, the graph isn't a DAG**

**if (topOrder.size() != vertices) {**

**throw new IllegalArgumentException("Graph contains cycles, topological sorting is not possible.");**

**}**

**return topOrder;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**List<List<Integer>> adjacencyList = new ArrayList<>();**

**for (int i = 0; i < vertices; i++) {**

**adjacencyList.add(new ArrayList<>());**

**}**

**// Input edges**

**System.out.print("Enter number of edges: ");**

**int edges = scanner.nextInt();**

**System.out.println("Enter edges (source destination):");**

**for (int i = 0; i < edges; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**adjacencyList.get(src).add(dest);**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**List<Integer> sortedOrder = topologicalSort(vertices, adjacencyList);**

**long endTime = System.nanoTime();**

**// Output result**

**System.out.println("Topological Sorting Order: " + sortedOrder);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement Fractional Knapsack problems using Greedy Method**

**import java.util.Arrays;**

**import java.util.Scanner;**

**class Item {**

**int weight, value;**

**double ratio; // Value-to-weight ratio**

**Item(int value, int weight) {**

**this.value = value;**

**this.weight = weight;**

**this.ratio = (double) value / weight;**

**}**

**}**

**public class FractionalKnapsack {**

**public static double fractionalKnapsack(Item[] items, int capacity) {**

**// Sort items in descending order of value-to-weight ratio**

**Arrays.sort(items, (a, b) -> Double.compare(b.ratio, a.ratio));**

**double maxValue = 0.0; // Maximum value achieved**

**int currentWeight = 0; // Current knapsack weight**

**for (Item item : items) {**

**if (currentWeight + item.weight <= capacity) {**

**// Take the full item**

**currentWeight += item.weight;**

**maxValue += item.value;**

**} else {**

**// Take a fraction of the item**

**int remainingWeight = capacity - currentWeight;**

**maxValue += item.ratio \* remainingWeight;**

**break; // Knapsack is full**

**}**

**}**

**return maxValue;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// User input**

**System.out.print("Enter number of items: ");**

**int n = scanner.nextInt();**

**Item[] items = new Item[n];**

**System.out.println("Enter value and weight of each item:");**

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

**int value = scanner.nextInt();**

**int weight = scanner.nextInt();**

**items[i] = new Item(value, weight);**

**}**

**System.out.print("Enter knapsack capacity: ");**

**int capacity = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**double maxValue = fractionalKnapsack(items, capacity);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Maximum value obtained: " + maxValue);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write Program to implement Traveling Salesman Problem using nearest neighbor algorithm**

**import java.util.Scanner;**

**public class TravelingSalesmanNN {**

**private static final int INF = Integer.MAX\_VALUE;**

**// Function to find the shortest path using Nearest Neighbor algorithm**

**public static void nearestNeighborTSP(int[][] graph, int startVertex) {**

**int vertices = graph.length;**

**boolean[] visited = new boolean[vertices];**

**int[] path = new int[vertices + 1];**

**int totalCost = 0;**

**int currentVertex = startVertex;**

**visited[currentVertex] = true;**

**path[0] = currentVertex;**

**for (int i = 1; i < vertices; i++) {**

**int nearestNeighbor = -1;**

**int minDistance = INF;**

**for (int v = 0; v < vertices; v++) {**

**if (!visited[v] && graph[currentVertex][v] < minDistance && graph[currentVertex][v] > 0) {**

**minDistance = graph[currentVertex][v];**

**nearestNeighbor = v;**

**}**

**}**

**if (nearestNeighbor == -1) {**

**System.out.println("No valid path found.");**

**return;**

**}**

**visited[nearestNeighbor] = true;**

**path[i] = nearestNeighbor;**

**totalCost += minDistance;**

**currentVertex = nearestNeighbor;**

**}**

**// Return to the start vertex to complete the cycle**

**totalCost += graph[currentVertex][startVertex];**

**path[vertices] = startVertex;**

**// Display the path and cost**

**System.out.println("Optimal Path using Nearest Neighbor Algorithm:");**

**for (int i = 0; i <= vertices; i++) {**

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

**}**

**System.out.println("\nTotal Cost: " + totalCost);**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

**System.out.print("Enter number of cities: ");**

**int vertices = scanner.nextInt();**

**int[][] graph = new int[vertices][vertices];**

**// Input adjacency matrix representation of graph**

**System.out.println("Enter adjacency matrix (0 for no direct path, INF for unreachable cities):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graph[i][j] = scanner.nextInt();**

**}**

**}**

**System.out.print("Enter starting city index: ");**

**int startVertex = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**nearestNeighborTSP(graph, startVertex);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement optimal binary search tree and also calculate the best-case complexity.**

**import java.util.Scanner;**

**public class OptimalBST {**

**// Function to compute the optimal cost of Binary Search Tree**

**public static int optimalBST(int[] keys, int[] freq, int n) {**

**int[][] cost = new int[n][n];**

**// Initialize the cost of one-key subtrees**

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

**cost[i][i] = freq[i];**

**}**

**// Compute cost for subtrees of increasing sizes**

**for (int L = 2; L <= n; L++) {**

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

**int j = i + L - 1;**

**cost[i][j] = Integer.MAX\_VALUE;**

**// Compute sum of frequencies from i to j**

**int sumFreq = sum(freq, i, j);**

**// Try each key as root and find minimum cost**

**for (int r = i; r <= j; r++) {**

**int c = sumFreq + (r > i ? cost[i][r - 1] : 0) + (r < j ? cost[r + 1][j] : 0);**

**if (c < cost[i][j]) {**

**cost[i][j] = c;**

**}**

**}**

**}**

**}**

**return cost[0][n - 1]; // Minimum cost of optimal BST**

**}**

**// Function to compute sum of frequencies from i to j**

**private static int sum(int[] freq, int i, int j) {**

**int sum = 0;**

**for (int k = i; k <= j; k++) {**

**sum += freq[k];**

**}**

**return sum;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of keys**

**System.out.print("Enter number of keys: ");**

**int n = scanner.nextInt();**

**int[] keys = new int[n];**

**int[] freq = new int[n];**

**// Input keys and their frequencies**

**System.out.println("Enter keys and their corresponding frequencies:");**

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

**keys[i] = scanner.nextInt();**

**freq[i] = scanner.nextInt();**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int minCost = optimalBST(keys, freq, n);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Minimum cost of Optimal Binary Search Tree: " + minCost);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to implement Sum of Subset by Backtracking**

**import java.util.Scanner;**

**public class SumOfSubsets {**

**static int[] subset;**

**static int[] weights;**

**static int targetSum;**

**static int n;**

**// Function to find subsets with sum equal to target using backtracking**

**public static void findSubsets(int index, int currentSum) {**

**if (currentSum == targetSum) {**

**// Print the valid subset**

**System.out.print("Subset found: ");**

**for (int i = 0; i < index; i++) {**

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

**}**

**System.out.println();**

**return;**

**}**

**if (index >= n || currentSum > targetSum) {**

**return;**

**}**

**// Include current element in the subset**

**subset[index] = weights[index];**

**findSubsets(index + 1, currentSum + weights[index]);**

**// Exclude current element from subset (Backtracking step)**

**findSubsets(index + 1, currentSum);**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of elements**

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

**n = scanner.nextInt();**

**weights = new int[n];**

**subset = new int[n];**

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

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

**weights[i] = scanner.nextInt();**

**}**

**// Input target sum**

**System.out.print("Enter target sum: ");**

**targetSum = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**findSubsets(0, 0);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement Huffman Code using greedy methods**

**import java.util.PriorityQueue;**

**import java.util.Scanner;**

**// Node class representing the Huffman tree**

**class HuffmanNode implements Comparable<HuffmanNode> {**

**char data;**

**int frequency;**

**HuffmanNode left, right;**

**HuffmanNode(char data, int frequency) {**

**this.data = data;**

**this.frequency = frequency;**

**}**

**@Override**

**public int compareTo(HuffmanNode other) {**

**return this.frequency - other.frequency;**

**}**

**}**

**public class HuffmanCoding {**

**// Function to build Huffman Tree using Greedy Algorithm**

**public static HuffmanNode buildHuffmanTree(char[] chars, int[] freq) {**

**PriorityQueue<HuffmanNode> minHeap = new PriorityQueue<>();**

**// Create leaf nodes and add them to the priority queue**

**for (int i = 0; i < chars.length; i++) {**

**minHeap.add(new HuffmanNode(chars[i], freq[i]));**

**}**

**while (minHeap.size() > 1) {**

**// Extract two minimum frequency nodes**

**HuffmanNode left = minHeap.poll();**

**HuffmanNode right = minHeap.poll();**

**// Create a new node with combined frequency**

**HuffmanNode newNode = new HuffmanNode('-', left.frequency + right.frequency);**

**newNode.left = left;**

**newNode.right = right;**

**// Add new node to minHeap**

**minHeap.add(newNode);**

**}**

**return minHeap.poll(); // Root of Huffman Tree**

**}**

**// Function to generate Huffman Codes using DFS**

**public static void generateHuffmanCodes(HuffmanNode root, String code) {**

**if (root == null) {**

**return;**

**}**

**// If leaf node, print character and its code**

**if (root.data != '-') {**

**System.out.println(root.data + ": " + code);**

**}**

**// Recursive calls for left and right subtrees**

**generateHuffmanCodes(root.left, code + "0");**

**generateHuffmanCodes(root.right, code + "1");**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of characters**

**System.out.print("Enter number of characters: ");**

**int n = scanner.nextInt();**

**char[] chars = new char[n];**

**int[] freq = new int[n];**

**System.out.println("Enter characters and their frequencies:");**

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

**chars[i] = scanner.next().charAt(0);**

**freq[i] = scanner.nextInt();**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**HuffmanNode root = buildHuffmanTree(chars, freq);**

**long endTime = System.nanoTime();**

**System.out.println("Huffman Codes:");**

**generateHuffmanCodes(root, "");**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q-2) Write a program to solve 4 Queens Problem using Backtracking**

**public class FourQueens {**

**static final int N = 4;**

**// Function to print the chessboard**

**public static void printSolution(int board[][]) {**

**System.out.println("Solution:");**

**for (int i = 0; i < N; i++) {**

**for (int j = 0; j < N; j++) {**

**System.out.print(board[i][j] + " ");**

**}**

**System.out.println();**

**}**

**}**

**// Function to check if a queen can be placed at board[row][col]**

**public static boolean isSafe(int board[][], int row, int col) {**

**// Check the same column**

**for (int i = 0; i < row; i++) {**

**if (board[i][col] == 1) return false;**

**}**

**// Check upper-left diagonal**

**for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {**

**if (board[i][j] == 1) return false;**

**}**

**// Check upper-right diagonal**

**for (int i = row, j = col; i >= 0 && j < N; i--, j++) {**

**if (board[i][j] == 1) return false;**

**}**

**return true;**

**}**

**// Function to solve the N Queens problem using backtracking**

**public static boolean solveNQueens(int board[][], int row) {**

**if (row >= N) {**

**printSolution(board);**

**return true;**

**}**

**boolean foundSolution = false;**

**for (int col = 0; col < N; col++) {**

**if (isSafe(board, row, col)) {**

**board[row][col] = 1;**

**foundSolution = solveNQueens(board, row + 1) || foundSolution;**

**board[row][col] = 0; // Backtrack**

**}**

**}**

**return foundSolution;**

**}**

**public static void main(String[] args) {**

**int board[][] = new int[N][N];**

**if (!solveNQueens(board, 0)) {**

**System.out.println("No solution exists.");**

**}**

**}**

**}**

**Q.1) Write a programs to implement DFS (Depth First Search) and determine the time complexity for the same.**

**import java.util.\*;**

**public class DFSAlgorithm {**

**private Map<Integer, List<Integer>> adjacencyList;**

**public DFSAlgorithm(int vertices) {**

**adjacencyList = new HashMap<>();**

**for (int i = 0; i < vertices; i++) {**

**adjacencyList.put(i, new ArrayList<>());**

**}**

**}**

**public void addEdge(int src, int dest) {**

**adjacencyList.get(src).add(dest);**

**}**

**public void dfs(int startVertex, boolean[] visited) {**

**visited[startVertex] = true;**

**System.out.print(startVertex + " ");**

**for (int neighbor : adjacencyList.get(startVertex)) {**

**if (!visited[neighbor]) {**

**dfs(neighbor, visited);**

**}**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**DFSAlgorithm graph = new DFSAlgorithm(vertices);**

**// Input number of edges**

**System.out.print("Enter number of edges: ");**

**int edges = scanner.nextInt();**

**System.out.println("Enter edges (source destination):");**

**for (int i = 0; i < edges; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**graph.addEdge(src, dest);**

**}**

**// Input starting vertex**

**System.out.print("Enter starting vertex: ");**

**int startVertex = scanner.nextInt();**

**// Measure execution time**

**boolean[] visited = new boolean[vertices];**

**long startTime = System.nanoTime();**

**System.out.println("DFS Traversal:");**

**graph.dfs(startVertex, visited);**

**long endTime = System.nanoTime();**

**System.out.println("\nExecution Time (nanoseconds): " + (endTime - startTime));**

**// Time Complexity Analysis**

**System.out.println("\nTime Complexity:");**

**System.out.println("DFS runs in \*\*O(V + E)\*\*, where V = number of vertices and E = number of edges.");**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to find shortest paths from a given vertex in a weighted connected graph, to other vertices using Dijkstra’s algorithm.**

**import java.util.\*;**

**class DijkstraAlgorithm {**

**public static void dijkstra(int[][] graph, int source) {**

**int vertices = graph.length;**

**int[] distance = new int[vertices]; // Stores shortest distances from source**

**boolean[] visited = new boolean[vertices]; // Tracks processed vertices**

**// Initialize distances to infinity, except source**

**Arrays.fill(distance, Integer.MAX\_VALUE);**

**distance[source] = 0;**

**// Priority queue to select vertex with the smallest known distance**

**PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.comparingInt(v -> distance[v]));**

**pq.add(source);**

**while (!pq.isEmpty()) {**

**int u = pq.poll(); // Extract the vertex with the smallest distance**

**visited[u] = true;**

**// Update distances for adjacent vertices**

**for (int v = 0; v < vertices; v++) {**

**if (graph[u][v] != 0 && !visited[v] && distance[u] + graph[u][v] < distance[v]) {**

**distance[v] = distance[u] + graph[u][v];**

**pq.add(v);**

**}**

**}**

**}**

**// Print shortest distances**

**System.out.println("Vertex \t Distance from Source " + source);**

**for (int i = 0; i < vertices; i++) {**

**System.out.println(i + " \t " + distance[i]);**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**int[][] graph = new int[vertices][vertices];**

**// Input adjacency matrix representation of graph**

**System.out.println("Enter adjacency matrix (0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Input source vertex**

**System.out.print("Enter source vertex: ");**

**int source = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**dijkstra(graph, source);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement BFS (Breadth First Search) and determine the time complexity for the same.**

**import java.util.\*;**

**public class BFSAlgorithm {**

**private Map<Integer, List<Integer>> adjacencyList;**

**public BFSAlgorithm(int vertices) {**

**adjacencyList = new HashMap<>();**

**for (int i = 0; i < vertices; i++) {**

**adjacencyList.put(i, new ArrayList<>());**

**}**

**}**

**public void addEdge(int src, int dest) {**

**adjacencyList.get(src).add(dest);**

**}**

**public void bfs(int startVertex) {**

**Queue<Integer> queue = new LinkedList<>();**

**boolean[] visited = new boolean[adjacencyList.size()];**

**queue.add(startVertex);**

**visited[startVertex] = true;**

**while (!queue.isEmpty()) {**

**int vertex = queue.poll();**

**System.out.print(vertex + " ");**

**for (int neighbor : adjacencyList.get(vertex)) {**

**if (!visited[neighbor]) {**

**visited[neighbor] = true;**

**queue.add(neighbor);**

**}**

**}**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**BFSAlgorithm graph = new BFSAlgorithm(vertices);**

**// Input number of edges**

**System.out.print("Enter number of edges: ");**

**int edges = scanner.nextInt();**

**System.out.println("Enter edges (source destination):");**

**for (int i = 0; i < edges; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**graph.addEdge(src, dest);**

**}**

**// Input starting vertex**

**System.out.print("Enter starting vertex: ");**

**int startVertex = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**System.out.println("BFS Traversal:");**

**graph.bfs(startVertex);**

**long endTime = System.nanoTime();**

**System.out.println("\nExecution Time (nanoseconds): " + (endTime - startTime));**

**// Time Complexity Analysis**

**System.out.println("\nTime Complexity:");**

**System.out.println("BFS runs in \*\*O(V + E)\*\*, where V = number of vertices and E = number of edges.");**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to sort a given set of elements using the Selection sort method and determine the time required to sort the elements.**

**import java.util.Scanner;**

**public class SelectionSort {**

**// Method to perform selection sort**

**public static void selectionSort(int[] arr) {**

**int n = arr.length;**

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

**int minIndex = i;**

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

**if (arr[j] < arr[minIndex]) {**

**minIndex = j; // Find the minimum element**

**}**

**}**

**// Swap the minimum element with the first unsorted element**

**int temp = arr[minIndex];**

**arr[minIndex] = arr[i];**

**arr[i] = temp;**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input size of the array**

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

**int n = scanner.nextInt();**

**int[] arr = new int[n];**

**// Input elements**

**System.out.println("Enter " + n + " numbers:");**

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

**arr[i] = scanner.nextInt();**

**}**

**// Measure sorting time**

**long startTime = System.nanoTime();**

**selectionSort(arr);**

**long endTime = System.nanoTime();**

**long duration = endTime - startTime;**

**// Display sorted array**

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

**for (int num : arr) {**

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

**}**

**// Display execution time**

**System.out.println("\nTime required for sorting (nanoseconds): " + duration);**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to find minimum number of multiplications in Matrix Chain Multiplication.**

**import java.util.Scanner;**

**public class MatrixChainMultiplication {**

**// Function to compute minimum multiplication cost**

**public static int matrixChainOrder(int[] dimensions) {**

**int n = dimensions.length;**

**int[][] dp = new int[n][n];**

**// Initialize table: single matrix cost is zero**

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

**dp[i][i] = 0;**

**}**

**// Compute cost for chain lengths from 2 to n-1**

**for (int chainLength = 2; chainLength < n; chainLength++) {**

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

**int j = i + chainLength - 1;**

**dp[i][j] = Integer.MAX\_VALUE;**

**// Try different partitions**

**for (int k = i; k < j; k++) {**

**int cost = dp[i][k] + dp[k + 1][j] + dimensions[i - 1] \* dimensions[k] \* dimensions[j];**

**if (cost < dp[i][j]) {**

**dp[i][j] = cost;**

**}**

**}**

**}**

**}**

**return dp[1][n - 1];**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of matrices**

**System.out.print("Enter number of matrices: ");**

**int n = scanner.nextInt();**

**int[] dimensions = new int[n + 1];**

**System.out.println("Enter dimensions (e.g., P1 P2 P3 for A1: P1xP2, A2: P2xP3):");**

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

**dimensions[i] = scanner.nextInt();**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int minCost = matrixChainOrder(dimensions);**

**long endTime = System.nanoTime();**

**// Output result**

**System.out.println("Minimum number of multiplications required: " + minCost);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to implement an optimal binary search tree and also calculate the best case and worst-case complexity.**

**import java.util.Scanner;**

**public class OptimalBST {**

**// Function to compute the optimal cost of Binary Search Tree**

**public static int optimalBST(int[] keys, int[] freq, int n) {**

**int[][] cost = new int[n][n];**

**// Initialize the cost of one-key subtrees**

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

**cost[i][i] = freq[i];**

**}**

**// Compute cost for subtrees of increasing sizes**

**for (int L = 2; L <= n; L++) {**

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

**int j = i + L - 1;**

**cost[i][j] = Integer.MAX\_VALUE;**

**// Compute sum of frequencies from i to j**

**int sumFreq = sum(freq, i, j);**

**// Try each key as root and find minimum cost**

**for (int r = i; r <= j; r++) {**

**int c = sumFreq + (r > i ? cost[i][r - 1] : 0) + (r < j ? cost[r + 1][j] : 0);**

**if (c < cost[i][j]) {**

**cost[i][j] = c;**

**}**

**}**

**}**

**}**

**return cost[0][n - 1]; // Minimum cost of optimal BST**

**}**

**// Function to compute sum of frequencies from i to j**

**private static int sum(int[] freq, int i, int j) {**

**int sum = 0;**

**for (int k = i; k <= j; k++) {**

**sum += freq[k];**

**}**

**return sum;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of keys**

**System.out.print("Enter number of keys: ");**

**int n = scanner.nextInt();**

**int[] keys = new int[n];**

**int[] freq = new int[n];**

**// Input keys and their frequencies**

**System.out.println("Enter keys and their corresponding frequencies:");**

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

**keys[i] = scanner.nextInt();**

**freq[i] = scanner.nextInt();**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int minCost = optimalBST(keys, freq, n);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Minimum cost of Optimal Binary Search Tree: " + minCost);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to sort a list of n numbers in ascending order using Insertion sort and determine the time required to sort the elements.**

**import java.util.Scanner;**

**public class InsertionSort {**

**// Function to perform Insertion Sort**

**public static void insertionSort(int[] arr) {**

**int n = arr.length;**

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

**int key = arr[i];**

**int j = i - 1;**

**// Move elements that are greater than key one position ahead**

**while (j >= 0 && arr[j] > key) {**

**arr[j + 1] = arr[j];**

**j--;**

**}**

**arr[j + 1] = key;**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input size of array**

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

**int n = scanner.nextInt();**

**int[] arr = new int[n];**

**// Input elements**

**System.out.println("Enter " + n + " numbers:");**

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

**arr[i] = scanner.nextInt();**

**}**

**// Measure sorting time**

**long startTime = System.nanoTime();**

**insertionSort(arr);**

**long endTime = System.nanoTime();**

**long duration = endTime - startTime;**

**// Display sorted array**

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

**for (int num : arr) {**

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

**}**

**// Display execution time**

**System.out.println("\nTime required for sorting (nanoseconds): " + duration);**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to implement DFS and BFS. Compare the time complexity**

**import java.util.\*;**

**public class GraphTraversal {**

**private Map<Integer, List<Integer>> adjacencyList;**

**public GraphTraversal(int vertices) {**

**adjacencyList = new HashMap<>();**

**for (int i = 0; i < vertices; i++) {**

**adjacencyList.put(i, new ArrayList<>());**

**}**

**}**

**public void addEdge(int src, int dest) {**

**adjacencyList.get(src).add(dest);**

**adjacencyList.get(dest).add(src); // Assuming an undirected graph**

**}**

**// DFS implementation using recursion**

**public void dfs(int startVertex, boolean[] visited) {**

**visited[startVertex] = true;**

**System.out.print(startVertex + " ");**

**for (int neighbor : adjacencyList.get(startVertex)) {**

**if (!visited[neighbor]) {**

**dfs(neighbor, visited);**

**}**

**}**

**}**

**// BFS implementation using a queue**

**public void bfs(int startVertex) {**

**Queue<Integer> queue = new LinkedList<>();**

**boolean[] visited = new boolean[adjacencyList.size()];**

**queue.add(startVertex);**

**visited[startVertex] = true;**

**while (!queue.isEmpty()) {**

**int vertex = queue.poll();**

**System.out.print(vertex + " ");**

**for (int neighbor : adjacencyList.get(vertex)) {**

**if (!visited[neighbor]) {**

**visited[neighbor] = true;**

**queue.add(neighbor);**

**}**

**}**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**GraphTraversal graph = new GraphTraversal(vertices);**

**// Input number of edges**

**System.out.print("Enter number of edges: ");**

**int edges = scanner.nextInt();**

**System.out.println("Enter edges (source destination):");**

**for (int i = 0; i < edges; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**graph.addEdge(src, dest);**

**}**

**// Input starting vertex**

**System.out.print("Enter starting vertex: ");**

**int startVertex = scanner.nextInt();**

**// Measure execution time for DFS**

**boolean[] visitedDFS = new boolean[vertices];**

**long startTimeDFS = System.nanoTime();**

**System.out.println("\nDFS Traversal:");**

**graph.dfs(startVertex, visitedDFS);**

**long endTimeDFS = System.nanoTime();**

**System.out.println("\nExecution Time (nanoseconds) for DFS: " + (endTimeDFS - startTimeDFS));**

**// Measure execution time for BFS**

**long startTimeBFS = System.nanoTime();**

**System.out.println("\nBFS Traversal:");**

**graph.bfs(startVertex);**

**long endTimeBFS = System.nanoTime();**

**System.out.println("\nExecution Time (nanoseconds) for BFS: " + (endTimeBFS - startTimeBFS));**

**// Time Complexity Analysis**

**System.out.println("\nTime Complexity Comparison:");**

**System.out.println("DFS runs in \*\*O(V + E)\*\*, where V = number of vertices and E = number of edges.");**

**System.out.println("BFS also runs in \*\*O(V + E)\*\*, since every vertex and edge is processed once.");**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using LCBB (Least Cost Branch and Bound).**

**import java.util.\*;**

**class KnapsackNode {**

**int level, profit, weight;**

**double bound;**

**public KnapsackNode(int level, int profit, int weight, double bound) {**

**this.level = level;**

**this.profit = profit;**

**this.weight = weight;**

**this.bound = bound;**

**}**

**}**

**public class LCBBKnapsack {**

**// Function to compute upper bound (max possible profit)**

**private static double computeBound(KnapsackNode node, int[] weights, int[] profits, int capacity, int n) {**

**if (node.weight >= capacity) return 0; // If weight exceeds capacity, bound is 0**

**double bound = node.profit;**

**int totalWeight = node.weight;**

**int i = node.level + 1;**

**// Include items while within capacity**

**while (i < n && totalWeight + weights[i] <= capacity) {**

**totalWeight += weights[i];**

**bound += profits[i];**

**i++;**

**}**

**// Include fractional part of next item if available**

**if (i < n) {**

**bound += (double) (profits[i] \* (capacity - totalWeight)) / weights[i];**

**}**

**return bound;**

**}**

**public static int knapsackLCBB(int[] weights, int[] profits, int capacity) {**

**int n = weights.length;**

**Arrays.sort(weights);**

**Arrays.sort(profits);**

**// Priority queue sorted by bound value (max heap)**

**PriorityQueue<KnapsackNode> pq = new PriorityQueue<>(Comparator.comparingDouble(node -> -node.bound));**

**// Start from root node**

**KnapsackNode root = new KnapsackNode(-1, 0, 0, computeBound(new KnapsackNode(-1, 0, 0, 0), weights, profits, capacity, n));**

**pq.add(root);**

**int maxProfit = 0;**

**while (!pq.isEmpty()) {**

**KnapsackNode currentNode = pq.poll();**

**if (currentNode.level == n - 1) continue;**

**// Left child (including the next item)**

**KnapsackNode leftChild = new KnapsackNode(currentNode.level + 1, currentNode.profit + profits[currentNode.level + 1],**

**currentNode.weight + weights[currentNode.level + 1], 0);**

**if (leftChild.weight <= capacity && leftChild.profit > maxProfit) {**

**maxProfit = leftChild.profit;**

**}**

**leftChild.bound = computeBound(leftChild, weights, profits, capacity, n);**

**if (leftChild.bound > maxProfit) {**

**pq.add(leftChild);**

**}**

**// Right child (excluding the next item)**

**KnapsackNode rightChild = new KnapsackNode(currentNode.level + 1, currentNode.profit, currentNode.weight, 0);**

**rightChild.bound = computeBound(rightChild, weights, profits, capacity, n);**

**if (rightChild.bound > maxProfit) {**

**pq.add(rightChild);**

**}**

**}**

**return maxProfit;**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of items**

**System.out.print("Enter number of items: ");**

**int n = scanner.nextInt();**

**int[] weights = new int[n];**

**int[] profits = new int[n];**

**System.out.println("Enter weights and profits of each item:");**

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

**weights[i] = scanner.nextInt();**

**profits[i] = scanner.nextInt();**

**}**

**System.out.print("Enter knapsack capacity: ");**

**int capacity = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int maxProfit = knapsackLCBB(weights, profits, capacity);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Maximum profit obtained: " + maxProfit);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to implement Graph Coloring Algorithm**

**import java.util.Scanner;**

**public class GraphColoring {**

**private int vertices;**

**private int[][] graph;**

**private int[] colors;**

**// Constructor to initialize graph and colors**

**public GraphColoring(int vertices) {**

**this.vertices = vertices;**

**graph = new int[vertices][vertices];**

**colors = new int[vertices];**

**}**

**// Function to check if assigning color is valid**

**private boolean isSafe(int vertex, int color) {**

**for (int i = 0; i < vertices; i++) {**

**if (graph[vertex][i] == 1 && colors[i] == color) {**

**return false;**

**}**

**}**

**return true;**

**}**

**// Recursive function for Graph Coloring**

**private boolean graphColoringUtil(int vertex, int numColors) {**

**if (vertex == vertices) {**

**return true; // All vertices are colored**

**}**

**for (int color = 1; color <= numColors; color++) {**

**if (isSafe(vertex, color)) {**

**colors[vertex] = color;**

**if (graphColoringUtil(vertex + 1, numColors)) {**

**return true;**

**}**

**colors[vertex] = 0; // Backtracking**

**}**

**}**

**return false;**

**}**

**// Function to solve the Graph Coloring Problem**

**public void solveGraphColoring(int numColors) {**

**if (!graphColoringUtil(0, numColors)) {**

**System.out.println("Solution does not exist.");**

**return;**

**}**

**System.out.println("Color Assignments:");**

**for (int i = 0; i < vertices; i++) {**

**System.out.println("Vertex " + i + " ---> Color " + colors[i]);**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**GraphColoring graphColoring = new GraphColoring(vertices);**

**// Input adjacency matrix**

**System.out.println("Enter adjacency matrix (1 for edge, 0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graphColoring.graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Input number of colors**

**System.out.print("Enter number of colors: ");**

**int numColors = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**graphColoring.solveGraphColoring(numColors);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using dynamic programming.**

**import java.util.Scanner;**

**public class KnapsackDP {**

**// Function to compute the maximum value that can be stored in the knapsack**

**public static int knapsack(int[] weights, int[] values, int capacity, int n) {**

**int[][] dp = new int[n + 1][capacity + 1];**

**// Build the DP table**

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

**for (int w = 0; w <= capacity; w++) {**

**if (weights[i - 1] <= w) {**

**dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);**

**} else {**

**dp[i][w] = dp[i - 1][w];**

**}**

**}**

**}**

**return dp[n][capacity];**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of items**

**System.out.print("Enter number of items: ");**

**int n = scanner.nextInt();**

**int[] weights = new int[n];**

**int[] values = new int[n];**

**System.out.println("Enter weights and values of each item:");**

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

**weights[i] = scanner.nextInt();**

**values[i] = scanner.nextInt();**

**}**

**System.out.print("Enter knapsack capacity: ");**

**int capacity = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int maxValue = knapsack(weights, values, capacity, n);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Maximum value obtained: " + maxValue);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to determine if a given graph is a Hamiltonian cycle or not.**

**import java.util.Scanner;**

**public class HamiltonianCycle {**

**private int[][] graph;**

**private int[] path;**

**private int vertices;**

**// Constructor to initialize the graph**

**public HamiltonianCycle(int vertices) {**

**this.vertices = vertices;**

**graph = new int[vertices][vertices];**

**path = new int[vertices];**

**}**

**// Function to check if the vertex can be added to the Hamiltonian cycle**

**private boolean isSafe(int vertex, int position) {**

**if (graph[path[position - 1]][vertex] == 0) {**

**return false;**

**}**

**for (int i = 0; i < position; i++) {**

**if (path[i] == vertex) {**

**return false;**

**}**

**}**

**return true;**

**}**

**// Recursive function to find the Hamiltonian cycle**

**private boolean hamiltonianCycleUtil(int position) {**

**if (position == vertices) {**

**return graph[path[position - 1]][path[0]] == 1; // Check if last vertex connects to the start vertex**

**}**

**for (int vertex = 1; vertex < vertices; vertex++) {**

**if (isSafe(vertex, position)) {**

**path[position] = vertex;**

**if (hamiltonianCycleUtil(position + 1)) {**

**return true;**

**}**

**path[position] = -1; // Backtracking step**

**}**

**}**

**return false;**

**}**

**// Function to check for a Hamiltonian cycle**

**public void checkHamiltonianCycle() {**

**for (int i = 0; i < vertices; i++) {**

**path[i] = -1;**

**}**

**path[0] = 0; // Start from vertex 0**

**if (!hamiltonianCycleUtil(1)) {**

**System.out.println("No Hamiltonian Cycle exists.");**

**} else {**

**System.out.print("Hamiltonian Cycle found: ");**

**for (int i = 0; i < vertices; i++) {**

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

**}**

**System.out.println(path[0]); // Print the cycle**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**HamiltonianCycle graph = new HamiltonianCycle(vertices);**

**// Input adjacency matrix**

**System.out.println("Enter adjacency matrix (1 for edge, 0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graph.graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**graph.checkHamiltonianCycle();**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement solve ‘N’ Queens Problem using Backtracking.**

**import java.util.Scanner;**

**public class NQueens {**

**private int N;**

**private int[][] board;**

**// Constructor to initialize the chessboard**

**public NQueens(int N) {**

**this.N = N;**

**board = new int[N][N];**

**}**

**// Function to check if a queen can be placed at board[row][col]**

**private boolean isSafe(int row, int col) {**

**// Check column for attacking queen**

**for (int i = 0; i < row; i++) {**

**if (board[i][col] == 1) return false;**

**}**

**// Check upper-left diagonal**

**for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {**

**if (board[i][j] == 1) return false;**

**}**

**// Check upper-right diagonal**

**for (int i = row, j = col; i >= 0 && j < N; i--, j++) {**

**if (board[i][j] == 1) return false;**

**}**

**return true;**

**}**

**// Recursive function to solve the N-Queens problem**

**private boolean solveNQueensUtil(int row) {**

**if (row >= N) {**

**printSolution();**

**return true;**

**}**

**boolean foundSolution = false;**

**for (int col = 0; col < N; col++) {**

**if (isSafe(row, col)) {**

**board[row][col] = 1; // Place the queen**

**foundSolution = solveNQueensUtil(row + 1) || foundSolution;**

**board[row][col] = 0; // Backtrack**

**}**

**}**

**return foundSolution;**

**}**

**// Function to print the chessboard configuration**

**private void printSolution() {**

**System.out.println("Solution:");**

**for (int i = 0; i < N; i++) {**

**for (int j = 0; j < N; j++) {**

**System.out.print(board[i][j] + " ");**

**}**

**System.out.println();**

**}**

**System.out.println();**

**}**

**// Function to initiate solving process**

**public void solveNQueens() {**

**if (!solveNQueensUtil(0)) {**

**System.out.println("No solution exists for " + N + " Queens.");**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**System.out.print("Enter the number of Queens (N): ");**

**int N = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**NQueens nQueens = new NQueens(N);**

**nQueens.solveNQueens();**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to find out solution for 0/1 knapsack problem.**

**import java.util.Scanner;**

**public class Knapsack01DP {**

**// Function to compute the maximum value that can be stored in the knapsack**

**public static int knapsack(int[] weights, int[] values, int capacity, int n) {**

**int[][] dp = new int[n + 1][capacity + 1];**

**// Build the DP table**

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

**for (int w = 0; w <= capacity; w++) {**

**if (weights[i - 1] <= w) {**

**dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);**

**} else {**

**dp[i][w] = dp[i - 1][w];**

**}**

**}**

**}**

**return dp[n][capacity];**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of items**

**System.out.print("Enter number of items: ");**

**int n = scanner.nextInt();**

**int[] weights = new int[n];**

**int[] values = new int[n];**

**System.out.println("Enter weights and values of each item:");**

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

**weights[i] = scanner.nextInt();**

**values[i] = scanner.nextInt();**

**}**

**System.out.print("Enter knapsack capacity: ");**

**int capacity = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**int maxValue = knapsack(weights, values, capacity, n);**

**long endTime = System.nanoTime();**

**// Output results**

**System.out.println("Maximum value obtained: " + maxValue);**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to implement Graph Coloring Algorithm.**

**import java.util.Scanner;**

**public class GraphColoring {**

**private int vertices;**

**private int[][] graph;**

**private int[] colors;**

**// Constructor to initialize graph and colors**

**public GraphColoring(int vertices) {**

**this.vertices = vertices;**

**graph = new int[vertices][vertices];**

**colors = new int[vertices];**

**}**

**// Function to check if assigning color is valid**

**private boolean isSafe(int vertex, int color) {**

**for (int i = 0; i < vertices; i++) {**

**if (graph[vertex][i] == 1 && colors[i] == color) {**

**return false;**

**}**

**}**

**return true;**

**}**

**// Recursive function for Graph Coloring**

**private boolean graphColoringUtil(int vertex, int numColors) {**

**if (vertex == vertices) {**

**return true; // All vertices are colored**

**}**

**for (int color = 1; color <= numColors; color++) {**

**if (isSafe(vertex, color)) {**

**colors[vertex] = color;**

**if (graphColoringUtil(vertex + 1, numColors)) {**

**return true;**

**}**

**colors[vertex] = 0; // Backtracking**

**}**

**}**

**return false;**

**}**

**// Function to solve the Graph Coloring Problem**

**public void solveGraphColoring(int numColors) {**

**if (!graphColoringUtil(0, numColors)) {**

**System.out.println("Solution does not exist.");**

**return;**

**}**

**System.out.println("Color Assignments:");**

**for (int i = 0; i < vertices; i++) {**

**System.out.println("Vertex " + i + " ---> Color " + colors[i]);**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**GraphColoring graphColoring = new GraphColoring(vertices);**

**// Input adjacency matrix**

**System.out.println("Enter adjacency matrix (1 for edge, 0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graphColoring.graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Input number of colors**

**System.out.print("Enter number of colors: ");**

**int numColors = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**graphColoring.solveGraphColoring(numColors);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to find out live node, E node and dead node from a given graph.**

**import java.util.\*;**

**class Node {**

**int id;**

**boolean isLive;**

**boolean isExpanding;**

**boolean isDead;**

**public Node(int id) {**

**this.id = id;**

**this.isLive = true; // Initially all nodes are live**

**this.isExpanding = false;**

**this.isDead = false;**

**}**

**}**

**public class GraphNodeClassification {**

**private Map<Integer, Node> nodes = new HashMap<>();**

**private Map<Integer, List<Integer>> adjacencyList = new HashMap<>();**

**public void addNode(int id) {**

**nodes.put(id, new Node(id));**

**adjacencyList.put(id, new ArrayList<>());**

**}**

**public void addEdge(int src, int dest) {**

**adjacencyList.get(src).add(dest);**

**}**

**public void classifyNodes(int startNode) {**

**Queue<Integer> queue = new LinkedList<>();**

**queue.add(startNode);**

**nodes.get(startNode).isExpanding = true; // Start node as E-node**

**while (!queue.isEmpty()) {**

**int current = queue.poll();**

**Node currentNode = nodes.get(current);**

**currentNode.isLive = false;**

**currentNode.isExpanding = true;**

**for (int neighbor : adjacencyList.get(current)) {**

**if (nodes.get(neighbor).isLive) {**

**queue.add(neighbor);**

**nodes.get(neighbor).isExpanding = true;**

**}**

**}**

**// If no further expansion, mark as dead node**

**if (adjacencyList.get(current).isEmpty()) {**

**currentNode.isExpanding = false;**

**currentNode.isDead = true;**

**}**

**}**

**}**

**public void displayNodeTypes() {**

**System.out.println("\nNode Classification:");**

**for (Node node : nodes.values()) {**

**System.out.println("Node " + node.id + " -> " +**

**(node.isExpanding ? "E-Node" : node.isDead ? "Dead Node" : "Live Node"));**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**GraphNodeClassification graph = new GraphNodeClassification();**

**// Input number of nodes**

**System.out.print("Enter number of nodes: ");**

**int numNodes = scanner.nextInt();**

**// Add nodes to graph**

**for (int i = 0; i < numNodes; i++) {**

**graph.addNode(i);**

**}**

**// Input edges**

**System.out.print("Enter number of edges: ");**

**int numEdges = scanner.nextInt();**

**System.out.println("Enter edges (source destination):");**

**for (int i = 0; i < numEdges; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**graph.addEdge(src, dest);**

**}**

**// Input starting node**

**System.out.print("Enter starting node: ");**

**int startNode = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**graph.classifyNodes(startNode);**

**long endTime = System.nanoTime();**

**// Output results**

**graph.displayNodeTypes();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.1) Write a program to determine if a given graph is a Hamiltonian cycle or Not.**

**import java.util.Scanner;**

**public class HamiltonianCycle {**

**private int[][] graph;**

**private int[] path;**

**private int vertices;**

**// Constructor to initialize the graph**

**public HamiltonianCycle(int vertices) {**

**this.vertices = vertices;**

**graph = new int[vertices][vertices];**

**path = new int[vertices];**

**}**

**// Function to check if the vertex can be added to the Hamiltonian cycle**

**private boolean isSafe(int vertex, int position) {**

**if (graph[path[position - 1]][vertex] == 0) {**

**return false;**

**}**

**for (int i = 0; i < position; i++) {**

**if (path[i] == vertex) {**

**return false;**

**}**

**}**

**return true;**

**}**

**// Recursive function to find the Hamiltonian cycle**

**private boolean hamiltonianCycleUtil(int position) {**

**if (position == vertices) {**

**return graph[path[position - 1]][path[0]] == 1; // Check if last vertex connects to the start vertex**

**}**

**for (int vertex = 1; vertex < vertices; vertex++) {**

**if (isSafe(vertex, position)) {**

**path[position] = vertex;**

**if (hamiltonianCycleUtil(position + 1)) {**

**return true;**

**}**

**path[position] = -1; // Backtracking step**

**}**

**}**

**return false;**

**}**

**// Function to check for a Hamiltonian cycle**

**public void checkHamiltonianCycle() {**

**for (int i = 0; i < vertices; i++) {**

**path[i] = -1;**

**}**

**path[0] = 0; // Start from vertex 0**

**if (!hamiltonianCycleUtil(1)) {**

**System.out.println("No Hamiltonian Cycle exists.");**

**} else {**

**System.out.print("Hamiltonian Cycle found: ");**

**for (int i = 0; i < vertices; i++) {**

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

**}**

**System.out.println(path[0]); // Print the cycle**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**HamiltonianCycle graph = new HamiltonianCycle(vertices);**

**// Input adjacency matrix**

**System.out.println("Enter adjacency matrix (1 for edge, 0 for no edge):");**

**for (int i = 0; i < vertices; i++) {**

**for (int j = 0; j < vertices; j++) {**

**graph.graph[i][j] = scanner.nextInt();**

**}**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**graph.checkHamiltonianCycle();**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**

**Q.2) Write a program to show board configuration of 4 queens’ problem.**

**public class FourQueens {**

**static final int N = 4;**

**static int[][] board = new int[N][N];**

**// Function to print the chessboard configuration**

**public static void printSolution() {**

**System.out.println("Board Configuration:");**

**for (int i = 0; i < N; i++) {**

**for (int j = 0; j < N; j++) {**

**System.out.print(board[i][j] + " ");**

**}**

**System.out.println();**

**}**

**System.out.println();**

**}**

**// Function to check if a queen can be placed at board[row][col]**

**public static boolean isSafe(int row, int col) {**

**for (int i = 0; i < row; i++) {**

**if (board[i][col] == 1) return false;**

**}**

**for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {**

**if (board[i][j] == 1) return false;**

**}**

**for (int i = row, j = col; i >= 0 && j < N; i--, j++) {**

**if (board[i][j] == 1) return false;**

**}**

**return true;**

**}**

**// Recursive function to solve the 4 Queens problem**

**public static boolean solve4Queens(int row) {**

**if (row >= N) {**

**printSolution();**

**return true;**

**}**

**boolean foundSolution = false;**

**for (int col = 0; col < N; col++) {**

**if (isSafe(row, col)) {**

**board[row][col] = 1;**

**foundSolution = solve4Queens(row + 1) || foundSolution;**

**board[row][col] = 0; // Backtracking step**

**}**

**}**

**return foundSolution;**

**}**

**public static void main(String[] args) {**

**System.out.println("Solving 4 Queens Problem...");**

**long startTime = System.nanoTime();**

**solve4Queens(0);**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**}**

**}**

**Q.1) Write a program to implement for finding Topological sorting and determine the time complexity for the same.**

**import java.util.\*;**

**public class TopologicalSorting {**

**private int vertices;**

**private List<Integer>[] adjacencyList;**

**// Constructor to initialize the graph**

**public TopologicalSorting(int vertices) {**

**this.vertices = vertices;**

**adjacencyList = new ArrayList[vertices];**

**for (int i = 0; i < vertices; i++) {**

**adjacencyList[i] = new ArrayList<>();**

**}**

**}**

**// Add directed edge to the graph**

**public void addEdge(int src, int dest) {**

**adjacencyList[src].add(dest);**

**}**

**// Function to perform topological sorting using Kahn's algorithm (BFS-based approach)**

**public void topologicalSort() {**

**int[] inDegree = new int[vertices]; // Array to store incoming edges count**

**for (int i = 0; i < vertices; i++) {**

**for (int neighbor : adjacencyList[i]) {**

**inDegree[neighbor]++;**

**}**

**}**

**Queue<Integer> queue = new LinkedList<>();**

**for (int i = 0; i < vertices; i++) {**

**if (inDegree[i] == 0) {**

**queue.add(i);**

**}**

**}**

**List<Integer> topologicalOrder = new ArrayList<>();**

**while (!queue.isEmpty()) {**

**int current = queue.poll();**

**topologicalOrder.add(current);**

**for (int neighbor : adjacencyList[current]) {**

**inDegree[neighbor]--;**

**if (inDegree[neighbor] == 0) {**

**queue.add(neighbor);**

**}**

**}**

**}**

**if (topologicalOrder.size() != vertices) {**

**System.out.println("Cycle detected! Topological sorting not possible.");**

**return;**

**}**

**// Print topological order**

**System.out.println("Topological Order:");**

**for (int node : topologicalOrder) {**

**System.out.print(node + " ");**

**}**

**System.out.println();**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**// Input number of vertices**

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

**int vertices = scanner.nextInt();**

**TopologicalSorting graph = new TopologicalSorting(vertices);**

**// Input number of edges**

**System.out.print("Enter number of edges: ");**

**int edges = scanner.nextInt();**

**System.out.println("Enter edges (source destination):");**

**for (int i = 0; i < edges; i++) {**

**int src = scanner.nextInt();**

**int dest = scanner.nextInt();**

**graph.addEdge(src, dest);**

**}**

**// Measure execution time**

**long startTime = System.nanoTime();**

**graph.topologicalSort();**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**// Time Complexity Analysis**

**System.out.println("\nTime Complexity:");**

**System.out.println("Topological sorting using Kahn's algorithm runs in \*\*O(V + E)\*\*,");**

**System.out.println("where V = number of vertices and E = number of edges.");**

**scanner.close();**

**}**

**}**

1. **Write a program to solve N Queens Problem using Backtracking.**

**import java.util.Scanner;**

**public class NQueens {**

**private int N;**

**private int[][] board;**

**// Constructor to initialize the chessboard**

**public NQueens(int N) {**

**this.N = N;**

**board = new int[N][N];**

**}**

**// Function to check if a queen can be placed at board[row][col]**

**private boolean isSafe(int row, int col) {**

**// Check column for attacking queen**

**for (int i = 0; i < row; i++) {**

**if (board[i][col] == 1) return false;**

**}**

**// Check upper-left diagonal**

**for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {**

**if (board[i][j] == 1) return false;**

**}**

**// Check upper-right diagonal**

**for (int i = row, j = col; i >= 0 && j < N; i--, j++) {**

**if (board[i][j] == 1) return false;**

**}**

**return true;**

**}**

**// Recursive function to solve the N-Queens problem**

**private boolean solveNQueensUtil(int row) {**

**if (row >= N) {**

**printSolution();**

**return true;**

**}**

**boolean foundSolution = false;**

**for (int col = 0; col < N; col++) {**

**if (isSafe(row, col)) {**

**board[row][col] = 1; // Place the queen**

**foundSolution = solveNQueensUtil(row + 1) || foundSolution;**

**board[row][col] = 0; // Backtrack**

**}**

**}**

**return foundSolution;**

**}**

**// Function to print the chessboard configuration**

**private void printSolution() {**

**System.out.println("Solution:");**

**for (int i = 0; i < N; i++) {**

**for (int j = 0; j < N; j++) {**

**System.out.print(board[i][j] + " ");**

**}**

**System.out.println();**

**}**

**System.out.println();**

**}**

**// Function to initiate solving process**

**public void solveNQueens() {**

**if (!solveNQueensUtil(0)) {**

**System.out.println("No solution exists for " + N + " Queens.");**

**}**

**}**

**public static void main(String[] args) {**

**Scanner scanner = new Scanner(System.in);**

**System.out.print("Enter the number of Queens (N): ");**

**int N = scanner.nextInt();**

**// Measure execution time**

**long startTime = System.nanoTime();**

**NQueens nQueens = new NQueens(N);**

**nQueens.solveNQueens();**

**long endTime = System.nanoTime();**

**System.out.println("Execution Time (nanoseconds): " + (endTime - startTime));**

**scanner.close();**

**}**

**}**