**Slip 1 – 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.**

java

public class SelectionSortDemo {

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;

}

}

// Swap arr[i] with arr[minIndex]

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

public static void main(String[] args) {

int[] arr = {64, 25, 12, 22, 11};

long startTime = System.nanoTime();

selectionSort(arr);

long endTime = System.nanoTime();

System.out.println("Selection Sort - Sorted array:");

for (int x : arr) {

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

}

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 1 – 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. (Experiment with different values of n.)**

java

public class QuickSortDemo {

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);

}

}

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

int pivot = arr[high]; // Using last element as pivot

int i = low - 1;

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

if (arr[j] < pivot) {

i++;

// swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Place pivot in the correct position

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

public static void main(String[] args) {

// You may experiment with larger arrays or generate random numbers.

int[] arr = {10, 7, 8, 9, 1, 5};

long startTime = System.nanoTime();

quickSort(arr, 0, arr.length - 1);

long endTime = System.nanoTime();

System.out.println("Quick Sort - Sorted array:");

for (int x : arr) {

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

}

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 2 – Q.1) Write a program to sort n randomly generated elements using Heapsort method.**

java

import java.util.Random;

public class HeapSortDemo {

public static void heapSort(int[] arr) {

int n = arr.length;

// Build heap (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--) {

// Move current root to end

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

public static void heapify(int[] arr, int heapSize, int i) {

int largest = i; // initialize largest as root

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

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

if (left < heapSize && arr[left] > arr[largest])

largest = left;

if (right < heapSize && arr[right] > arr[largest])

largest = right;

// If largest is not root, swap and continue heapifying.

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

heapify(arr, heapSize, largest);

}

}

public static void main(String[] args) {

// Generate random array of n elements

int n = 10; // change n as desired

int[] arr = new int[n];

Random rand = new Random();

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

arr[i] = rand.nextInt(100); // random numbers from 0 to 99

}

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

for (int x : arr) {

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

}

System.out.println();

long startTime = System.nanoTime();

heapSort(arr);

long endTime = System.nanoTime();

System.out.print("HeapSort - Sorted array: ");

for (int x : arr) {

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

}

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 2 – Q.2) Write a program to implement Strassen’s Matrix multiplication.**

java

public class StrassenMatrixMultiplication {

// Strassen algorithm works best for square matrices with dimensions as power of 2.

// For simplicity, assume matrix dimensions are 2^n x 2^n.

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

int n = A.length;

int[][] C = new int[n][n];

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

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

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

return C;

}

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

int n = A.length;

int[][] C = new int[n][n];

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

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

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

return C;

}

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

int n = A.length;

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

if (n == 1) {

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

} else {

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];

// dividing matrices

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];

}

}

// Calculating p1 to p7:

int[][] p1 = strassenMultiply(a11, subtract(b12, b22));

int[][] p2 = strassenMultiply(add(a11, a12), b22);

int[][] p3 = strassenMultiply(add(a21, a22), b11);

int[][] p4 = strassenMultiply(a22, subtract(b21, b11));

int[][] p5 = strassenMultiply(add(a11, a22), add(b11, b22));

int[][] p6 = strassenMultiply(subtract(a12, a22), add(b21, b22));

int[][] p7 = strassenMultiply(subtract(a11, a21), add(b11, b12));

// Compute sub-matrices of result matrix

int[][] c11 = add(subtract(add(p5, p4), p2), p6);

int[][] c12 = add(p1, p2);

int[][] c21 = add(p3, p4);

int[][] c22 = subtract(subtract(add(p5, p1), p3), p7);

// Combine sub-matrices into one 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;

}

public static void printMatrix(int[][] M) {

for (int[] row : M) {

for (int val : row) {

System.out.print(val + "\t");

}

System.out.println();

}

}

public static void main(String[] args) {

// Example with 4x4 matrices (2^2 x 2^2)

int[][] A = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10,11,12},

{13,14,15,16}

};

int[][] B = {

{16,15,14,13},

{12,11,10,9},

{8,7,6,5},

{4,3,2,1}

};

long startTime = System.nanoTime();

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

long endTime = System.nanoTime();

System.out.println("Strassen Matrix Multiplication Result:");

printMatrix(C);

System.out.println("Time taken (ns): " + (endTime - startTime));

}

}

**Slip 3 – Q.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.**

java

public class QuickSortDemoSlip3 {

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);

}

}

public 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++;

int swap = arr[i];

arr[i] = arr[j];

arr[j] = swap;

}

}

int swap = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = swap;

return i + 1;

}

public static void main(String[] args) {

int[] arr = {45, 23, 89, 12, 67, 34};

long startTime = System.nanoTime();

quickSort(arr, 0, arr.length - 1);

long endTime = System.nanoTime();

System.out.println("Quick Sort (Slip 3) - Sorted array:");

for (int x : arr)

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

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 3 – Q.2) Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm.**

java

import java.util.\*;

public class PrimMST {

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

// Function to construct MST using Prim's Algorithm.

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

int V = graph.length;

int[] key = new int[V];

int[] parent = new int[V];

boolean[] mstSet = new boolean[V];

Arrays.fill(key, INF);

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

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

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

int u = minKey(key, mstSet);

mstSet[u] = true;

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

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

parent[v] = u;

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

}

}

}

// Print the constructed MST

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

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

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

}

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

int min = INF, minIndex = -1;

for (int v = 0; v < key.length; v++) {

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

min = key[v];

minIndex = v;

}

}

return minIndex;

}

public static void main(String[] args) {

// Example graph (Adjacency matrix)

int[][] graph = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0},

};

primMST(graph);

}

}

**Slip 4 – 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.**

java

public class MergeSortDemo {

public static void mergeSort(int[] arr, int l, int r) {

if (l < r) {

int m = (l + r) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

public static void merge(int[] arr, int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int[] L = new int[n1];

int[] R = new int[n2];

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

L[i] = arr[l + i];

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

R[j] = arr[m + 1 + j];

int i = 0, j = 0;

int k = l;

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

if (L[i] <= R[j]) {

arr[k++] = L[i++];

} else {

arr[k++] = R[j++];

}

}

while (i < n1) {

arr[k++] = L[i++];

}

while (j < n2) {

arr[k++] = R[j++];

}

}

public static void main(String[] args) {

int[] arr = {38, 27, 43, 3, 9, 82, 10};

long startTime = System.nanoTime();

mergeSort(arr, 0, arr.length - 1);

long endTime = System.nanoTime();

System.out.println("Merge Sort - Sorted array:");

for (int x : arr)

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

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 4 – Q.2) Write a program to implement Knapsack problems using Greedy method.**

java

import java.util.Arrays;

import java.util.Comparator;

public class FractionalKnapsackGreedy {

static class Item {

int weight, value;

Item(int weight, int value) {

this.weight = weight;

this.value = value;

}

}

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

// Sort items based on value/weight ratio in descending order.

Arrays.sort(items, new Comparator<Item>() {

public int compare(Item a, Item b) {

double r1 = (double)a.value / a.weight;

double r2 = (double)b.value / b.weight;

return Double.compare(r2, r1);

}

});

int currentWeight = 0;

double totalValue = 0;

for (Item item : items) {

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

currentWeight += item.weight;

totalValue += item.value;

} else {

int remain = capacity - currentWeight;

totalValue += item.value \* ((double)remain / item.weight);

break;

}

}

return totalValue;

}

public static void main(String[] args) {

Item[] items = {new Item(10, 60), new Item(20, 100), new Item(30, 120)};

int capacity = 50;

double maxValue = fractionalKnapsack(items, capacity);

System.out.println("Maximum value for Fractional Knapsack: " + maxValue);

}

}

**Slip 5 – Q.1) Write a program for the Implementation of Kruskal’s algorithm to find minimum cost spanning tree.**

java

import java.util.Arrays;

public class KruskalMST {

static class Edge implements Comparable<Edge> {

int src, dest, weight;

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

this.src = src;

this.dest = dest;

this.weight = weight;

}

public int compareTo(Edge compareEdge) {

return this.weight - compareEdge.weight;

}

}

static class Graph {

int V, E;

Edge[] edges;

Graph(int v, int e) {

V = v;

E = e;

edges = new Edge[E];

}

}

static class Subset {

int parent, rank;

}

// Find set of an element i (uses path compression)

static int find(Subset[] subsets, int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// Do union of two sets x and y (uses union by rank)

static void union(Subset[] subsets, int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

public static void kruskalMST(Graph graph) {

Edge[] result = new Edge[graph.V];

int e = 0;

int i = 0;

for (i = 0; i < graph.V; ++i)

result[i] = null;

Arrays.sort(graph.edges);

Subset[] subsets = new Subset[graph.V];

for (i = 0; i < graph.V; ++i) {

subsets[i] = new Subset();

subsets[i].parent = i;

subsets[i].rank = 0;

}

i = 0;

while (e < graph.V - 1 && i < graph.E) {

Edge nextEdge = graph.edges[i++];

int x = find(subsets, nextEdge.src);

int y = find(subsets, nextEdge.dest);

if (x != y) {

result[e++] = nextEdge;

union(subsets, x, y);

}

}

System.out.println("Edges in the constructed MST:");

for (i = 0; i < e; ++i)

System.out.println(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);

}

public static void main(String[] args) {

int V = 4; // number of vertices

int E = 5; // number of edges

Graph graph = new Graph(V, E);

// add edge 0-1

graph.edges[0] = new Edge(0, 1, 10);

// add edge 0-2

graph.edges[1] = new Edge(0, 2, 6);

// add edge 0-3

graph.edges[2] = new Edge(0, 3, 5);

// add edge 1-3

graph.edges[3] = new Edge(1, 3, 15);

// add edge 2-3

graph.edges[4] = new Edge(2, 3, 4);

kruskalMST(graph);

}

}

**Slip 5 – Q.2) Write a program to implement Huffman Code using greedy methods and also calculate the best case and worst-case complexity.**

java

import java.util.PriorityQueue;

public class HuffmanCoding {

static class Node implements Comparable<Node> {

char ch;

int freq;

Node left, right;

Node(char ch, int freq) {

this.ch = ch;

this.freq = freq;

}

Node(int freq, Node left, Node right) {

this.ch = '-';

this.freq = freq;

this.left = left;

this.right = right;

}

public int compareTo(Node other) {

return this.freq - other.freq;

}

}

public static Node buildHuffmanTree(char[] charArray, int[] freqArray) {

PriorityQueue<Node> pq = new PriorityQueue<>();

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

pq.add(new Node(charArray[i], freqArray[i]));

}

while (pq.size() > 1) {

Node left = pq.poll();

Node right = pq.poll();

Node merged = new Node(left.freq + right.freq, left, right);

pq.add(merged);

}

return pq.peek();

}

public static void printCodes(Node root, String code) {

if (root.left == null && root.right == null) {

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

return;

}

if (root.left != null)

printCodes(root.left, code + "0");

if (root.right != null)

printCodes(root.right, code + "1");

}

public static void main(String[] args) {

char[] charArray = {'a', 'b', 'c', 'd', 'e'};

int[] freqArray = {5, 9, 12, 13, 16};

Node root = buildHuffmanTree(charArray, freqArray);

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

printCodes(root, "");

System.out.println("Time Complexity: O(n log n) in general (best and worst cases depend on frequency distribution).");

}

}

**Slip 6 – Q.1) Write a program for the Implementation of Prim’s algorithm to find minimum cost spanning tree.**

java

// (A repeat of Prim's MST from Slip 3 can be used; here is a self-contained version)

import java.util.\*;

public class PrimMST\_Slip6 {

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

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

int V = graph.length;

int[] key = new int[V];

int[] parent = new int[V];

boolean[] mstSet = new boolean[V];

Arrays.fill(key, INF);

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, mstSet);

mstSet[u] = true;

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

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

parent[v] = u;

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

}

}

}

System.out.println("Prim's MST:");

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

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

}

}

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

int min = INF, minIndex = -1;

for (int i = 0; i < key.length; i++)

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

min = key[i];

minIndex = i;

}

return minIndex;

}

public static void main(String[] args) {

int[][] graph = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0},

};

primMST(graph);

}

}

**Slip 6 – Q.2) Write a Program to find only length of Longest Common Subsequence.**

java

public class LCSLength {

public static int lcsLength(String s1, String s2) {

int m = s1.length(), n = s2.length();

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

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

else

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

}

}

return dp[m][n];

}

public static void main(String[] args) {

String s1 = "ABCBDAB";

String s2 = "BDCABA";

int length = lcsLength(s1, s2);

System.out.println("Length of LCS: " + length);

}

}

**Slip 7 – Q.1) Write a program for the Implementation of Dijkstra’s algorithm to find shortest path to other vertices.**

java

import java.util.\*;

public class DijkstraAlgorithm {

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

int V = graph.length;

int[] dist = new int[V];

boolean[] sptSet = new boolean[V];

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

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] != 0 &&

dist[u] != Integer.MAX\_VALUE &&

dist[u] + graph[u][v] < dist[v])

{

dist[v] = dist[u] + graph[u][v];

}

}

}

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

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

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

}

private static int minDistance(int[] dist, boolean[] sptSet) {

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

for (int v = 0; v < dist.length; v++)

if (!sptSet[v] && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

return min\_index;

}

public static void main(String[] args) {

int[][] graph = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

dijkstra(graph, 0);

}

}

**Slip 7 – Q.2) Write a program for finding Topological sorting for Directed Acyclic Graph (DAG).**

java

import java.util.\*;

public class TopologicalSort {

public static void topologicalSortUtil(int v, boolean[] visited, Stack<Integer> stack, List<List<Integer>> adj) {

visited[v] = true;

for (int neighbor : adj.get(v)) {

if (!visited[neighbor])

topologicalSortUtil(neighbor, visited, stack, adj);

}

stack.push(v);

}

public static void topologicalSort(List<List<Integer>> adj, int V) {

Stack<Integer> stack = new Stack<>();

boolean[] visited = new boolean[V];

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

if (!visited[i])

topologicalSortUtil(i, visited, stack, adj);

}

System.out.println("Topological Order:");

while (!stack.isEmpty())

System.out.print(stack.pop() + " ");

}

public static void main(String[] args) {

int V = 6;

List<List<Integer>> adj = new ArrayList<>();

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

adj.add(new ArrayList<>());

}

// Example DAG

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

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

adj.get(4).add(0);

adj.get(4).add(1);

adj.get(2).add(3);

adj.get(3).add(1);

topologicalSort(adj, V);

}

}

**Slip 8 – Q.1) Write a program to implement Fractional Knapsack problems using Greedy Method.**

java

// This is essentially the same as Fractional Knapsack from Slip 4 Q.2.

public class FractionalKnapsack\_Slip8 {

static class Item {

int weight, value;

Item(int weight, int value) {

this.weight = weight;

this.value = value;

}

}

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

// Sort items based on value/weight ratio in descending order.

java.util.Arrays.sort(items, (a, b) -> {

double r1 = (double)a.value / a.weight;

double r2 = (double)b.value / b.weight;

return Double.compare(r2, r1);

});

int currentWeight = 0;

double totalValue = 0;

for (Item item : items) {

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

currentWeight += item.weight;

totalValue += item.value;

} else {

int remain = capacity - currentWeight;

totalValue += item.value \* ((double)remain / item.weight);

break;

}

}

return totalValue;

}

public static void main(String[] args) {

Item[] items = {new Item(10, 60), new Item(20, 100), new Item(30, 120)};

int capacity = 50;

double maxValue = fractionalKnapsack(items, capacity);

System.out.println("Fractional Knapsack Maximum Value: " + maxValue);

}

}

**Slip 8 – Q.2) Write Program to implement Traveling Salesman Problem using nearest neighbor algorithm.**

java

public class TSPNearestNeighbor {

public static int tsp(int[][] graph, int start) {

int n = graph.length;

boolean[] visited = new boolean[n];

int count = 1;

int current = start;

int cost = 0;

visited[start] = true;

while (count < n) {

int next = -1;

int min = Integer.MAX\_VALUE;

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

if (!visited[j] && graph[current][j] < min) {

min = graph[current][j];

next = j;

}

}

if (next == -1) break;

visited[next] = true;

cost += min;

current = next;

count++;

}

// return to start

cost += graph[current][start];

return cost;

}

public static void main(String[] args) {

int[][] graph = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

int cost = tsp(graph, 0);

System.out.println("TSP cost using Nearest Neighbor: " + cost);

}

}

**Slip 9 – Q.1) Write a program to implement optimal binary search tree and also calculate the best-case complexity.**

java

public class OptimalBST {

// Given keys and frequency arrays, compute the minimum cost of a binary search tree.

public static int optimalCost(int[] freq) {

int n = freq.length;

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

// cost for single key tree

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

dp[i][i] = freq[i];

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

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

int j = i + L - 1;

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

int sum = 0;

for (int k = i; k <= j; k++) sum += freq[k];

// Try making all keys in interval keys[i..j] as root.

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

int cost = ((r > i) ? dp[i][r - 1] : 0)

+ ((r < j) ? dp[r + 1][j] : 0);

dp[i][j] = Math.min(dp[i][j], cost);

}

dp[i][j] += sum;

}

}

return dp[0][n-1];

}

public static void main(String[] args) {

// Example keys frequencies

int[] freq = {34, 8, 50};

int cost = optimalCost(freq);

System.out.println("Optimal BST Cost: " + cost);

System.out.println("Best case complexity depends on frequency distribution, " +

"but DP runs in O(n^3) in general.");

}

}

**Slip 9 – Q.2) Write a program to implement Sum of Subset by Backtracking.**

java

import java.util.ArrayList;

import java.util.List;

public class SubsetSumBacktracking {

public static void subsetSum(int[] arr, int target) {

List<Integer> current = new ArrayList<>();

subsetSumUtil(arr, target, 0, current);

}

public static void subsetSumUtil(int[] arr, int target, int index, List<Integer> current) {

if (target == 0) {

System.out.println("Subset found: " + current);

return;

}

if (target < 0 || index >= arr.length)

return;

// choose the element

current.add(arr[index]);

subsetSumUtil(arr, target - arr[index], index + 1, current);

current.remove(current.size() - 1);

// skip the element

subsetSumUtil(arr, target, index + 1, current);

}

public static void main(String[] args) {

int[] arr = {3, 34, 4, 12, 5, 2};

int target = 9;

subsetSum(arr, target);

}

}

**Slip 10 – Q.1) Write a program to implement Huffman Code using greedy methods.**

java

// This is similar to the Huffman coding implementation in Slip 5 Q.2.

public class HuffmanCoding\_Slip10 {

static class Node implements Comparable<Node> {

char ch;

int freq;

Node left, right;

Node(char ch, int freq) {

this.ch = ch;

this.freq = freq;

}

Node(int freq, Node left, Node right) {

this.ch = '-';

this.freq = freq;

this.left = left;

this.right = right;

}

public int compareTo(Node other) {

return this.freq - other.freq;

}

}

public static Node buildHuffmanTree(char[] charArray, int[] freqArray) {

java.util.PriorityQueue<Node> pq = new java.util.PriorityQueue<>();

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

pq.add(new Node(charArray[i], freqArray[i]));

}

while (pq.size() > 1) {

Node left = pq.poll();

Node right = pq.poll();

Node merged = new Node(left.freq + right.freq, left, right);

pq.add(merged);

}

return pq.peek();

}

public static void printCodes(Node root, String code) {

if (root.left == null && root.right == null) {

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

return;

}

if (root.left != null)

printCodes(root.left, code + "0");

if (root.right != null)

printCodes(root.right, code + "1");

}

public static void main(String[] args) {

char[] charArray = {'a', 'b', 'c', 'd'};

int[] freqArray = {5, 9, 12, 13};

Node root = buildHuffmanTree(charArray, freqArray);

System.out.println("Huffman Codes (Slip 10):");

printCodes(root, "");

}

}

**Slip 10 – Q.2) Write a program to solve 4 Queens Problem using Backtracking.**

java

public class FourQueens {

final int N = 4;

void printSolution(int board[][]) {

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

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

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

}

System.out.println();

}

}

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

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

if (board[row][i] == 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; j >= 0 && i < N; i++, j--)

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

return false;

return true;

}

boolean solveNQUtil(int board[][], int col) {

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0; // backtracking

}

}

return false;

}

public void solve() {

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

if (!solveNQUtil(board, 0)) {

System.out.println("Solution does not exist");

return;

}

printSolution(board);

}

public static void main(String[] args) {

new FourQueens().solve();

}

}

**Slip 11 – Q.1) Write a programs to implement DFS (Depth First Search) and determine the time complexity for the same.**

java

import java.util.\*;

public class DFSGraph {

private int V;

private List<Integer>[] adj;

public DFSGraph(int V) {

this.V = V;

adj = (List<Integer>[]) new ArrayList[V];

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

adj[i] = new ArrayList<>();

}

public void addEdge(int v, int w) {

adj[v].add(w);

}

public void DFSUtil(int v, boolean[] visited) {

visited[v] = true;

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

for (int n : adj[v])

if (!visited[n])

DFSUtil(n, visited);

}

public void DFS(int v) {

boolean[] visited = new boolean[V];

DFSUtil(v, visited);

}

public static void main(String[] args) {

DFSGraph g = new DFSGraph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Depth First Traversal starting from vertex 2:");

long startTime = System.nanoTime();

g.DFS(2);

long endTime = System.nanoTime();

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 11 – 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.**

*(This is similar to Slip 7 – Q.1; you may refer to that implementation.)*

**Slip 12 – Q.1) Write a program to implement BFS (Breadth First Search) and determine the time complexity for the same.**

java

import java.util.\*;

public class BFSGraph {

private int V;

private List<Integer>[] adj;

public BFSGraph(int V) {

this.V = V;

adj = (List<Integer>[]) new ArrayList[V];

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

adj[i] = new ArrayList<>();

}

public void addEdge(int v, int w) {

adj[v].add(w);

}

public void BFS(int s) {

boolean[] visited = new boolean[V];

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

visited[s] = true;

queue.add(s);

while (!queue.isEmpty()) {

int v = queue.poll();

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

for (int n : adj[v]) {

if (!visited[n]) {

visited[n] = true;

queue.add(n);

}

}

}

}

public static void main(String[] args) {

BFSGraph g = new BFSGraph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Breadth First Traversal starting from vertex 2:");

long startTime = System.nanoTime();

g.BFS(2);

long endTime = System.nanoTime();

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 12 – 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.**

java

public class SelectionSort\_Slip12 {

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;

}

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

public static void main(String[] args) {

int[] arr = {29, 10, 14, 37, 13};

long startTime = System.nanoTime();

selectionSort(arr);

long endTime = System.nanoTime();

System.out.println("Selection Sort (Slip 12) - Sorted array:");

for (int x : arr)

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

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 13 – Q.1) Write a program to find minimum number of multiplications in Matrix Chain Multiplication.**

java

public class MatrixChainMultiplication {

// Returns the minimum number of multiplications needed for matrix chain multiplication

public static int matrixChainOrder(int[] p) {

int n = p.length - 1;

int[][] m = new int[n][n];

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

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

int j = i + L - 1;

m[i][j] = Integer.MAX\_VALUE;

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

int cost = m[i][k] + m[k+1][j] + p[i] \* p[k+1] \* p[j+1];

m[i][j] = Math.min(m[i][j], cost);

}

}

}

return m[0][n-1];

}

public static void main(String[] args) {

int[] dimensions = {10, 30, 5, 60};

int minMultiplications = matrixChainOrder(dimensions);

System.out.println("Minimum number of multiplications: " + minMultiplications);

}

}

**Slip 13 – Q.2) Write a program to implement an optimal binary search tree and also calculate the best case and worst-case complexity.**

java

public class OptimalBST\_Slip13 {

public static int optimalBSTCost(int[] freq) {

int n = freq.length;

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

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

dp[i][i] = freq[i];

}

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

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

int j = i + L - 1;

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

int sum = 0;

for (int k = i; k <= j; k++) sum += freq[k];

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

int cost = ((r > i) ? dp[i][r - 1] : 0) +

((r < j) ? dp[r + 1][j] : 0);

dp[i][j] = Math.min(dp[i][j], cost);

}

dp[i][j] += sum;

}

}

return dp[0][n-1];

}

public static void main(String[] args) {

int[] freq = {34, 8, 50};

int cost = optimalBSTCost(freq);

System.out.println("Optimal BST Cost: " + cost);

System.out.println("Note: Complexity of the DP solution is O(n^3)");

}

}

**Slip 14 – 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.**

java

public class InsertionSortDemo {

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;

while (j >= 0 && arr[j] > key) {

arr[j+1] = arr[j];

j--;

}

arr[j+1] = key;

}

}

public static void main(String[] args) {

int[] arr = {12, 11, 13, 5, 6};

long startTime = System.nanoTime();

insertionSort(arr);

long endTime = System.nanoTime();

System.out.println("Insertion Sort - Sorted array:");

for (int x : arr)

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

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 14 – Q.2) Write a program to implement DFS and BFS. Compare the time complexity.**

java

import java.util.\*;

public class DFSandBFS {

private int V;

private List<Integer>[] adj;

public DFSandBFS(int V) {

this.V = V;

adj = (List<Integer>[]) new ArrayList[V];

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

adj[i] = new ArrayList<>();

}

public void addEdge(int v, int w) {

adj[v].add(w);

}

// DFS using recursion

public void DFS(int v) {

boolean[] visited = new boolean[V];

DFSUtil(v, visited);

}

private void DFSUtil(int v, boolean[] visited) {

visited[v] = true;

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

for (int n : adj[v]) {

if (!visited[n])

DFSUtil(n, visited);

}

}

// BFS using queue

public void BFS(int s) {

boolean[] visited = new boolean[V];

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

visited[s] = true;

queue.add(s);

while (!queue.isEmpty()) {

int v = queue.poll();

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

for (int n : adj[v]) {

if (!visited[n]) {

visited[n] = true;

queue.add(n);

}

}

}

}

public static void main(String[] args) {

DFSandBFS graph = new DFSandBFS(4);

graph.addEdge(0, 1);

graph.addEdge(0, 2);

graph.addEdge(1, 2);

graph.addEdge(2, 0);

graph.addEdge(2, 3);

graph.addEdge(3, 3);

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

long startDFS = System.nanoTime();

graph.DFS(2);

long endDFS = System.nanoTime();

System.out.println("\nTime taken by DFS (ns): " + (endDFS - startDFS));

System.out.println("\nBFS:");

long startBFS = System.nanoTime();

graph.BFS(2);

long endBFS = System.nanoTime();

System.out.println("\nTime taken by BFS (ns): " + (endBFS - startBFS));

System.out.println("\nNote: Both DFS and BFS have time complexity O(V+E).");

}

}

**Slip 15 – Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using LCBB (Least Cost Branch and Bound).**

java

import java.util.\*;

public class KnapsackBranchBound {

static class Node {

int level, profit, weight;

double bound;

}

public static double bound(Node u, int n, int W, int[] weights, int[] profits) {

if (u.weight >= W)

return 0;

double profitBound = u.profit;

int j = u.level + 1;

int totWeight = u.weight;

while (j < n && totWeight + weights[j] <= W) {

totWeight += weights[j];

profitBound += profits[j];

j++;

}

if (j < n)

profitBound += (W - totWeight) \* ((double)profits[j] / weights[j]);

return profitBound;

}

public static int knapSack(int W, int[] weights, int[] profits, int n) {

Queue<Node> Q = new LinkedList<>();

Node u = new Node();

Node v = new Node();

u.level = -1;

u.profit = 0;

u.weight = 0;

u.bound = bound(u, n, W, weights, profits);

Q.add(u);

int maxProfit = 0;

while (!Q.isEmpty()) {

u = Q.poll();

if (u.bound > maxProfit) {

v.level = u.level + 1;

v.weight = u.weight + weights[v.level];

v.profit = u.profit + profits[v.level];

if (v.weight <= W && v.profit > maxProfit)

maxProfit = v.profit;

v.bound = bound(v, n, W, weights, profits);

if (v.bound > maxProfit)

Q.add(new Node(){{

level = v.level;

weight = v.weight;

profit = v.profit;

bound = v.bound;

}});

Node v2 = new Node();

v2.level = u.level + 1;

v2.weight = u.weight;

v2.profit = u.profit;

v2.bound = bound(v2, n, W, weights, profits);

if (v2.bound > maxProfit)

Q.add(v2);

}

}

return maxProfit;

}

public static void main(String[] args) {

int[] profits = {60, 100, 120};

int[] weights = {10, 20, 30};

int W = 50;

int n = profits.length;

int maxProfit = knapSack(W, weights, profits, n);

System.out.println("Maximum profit by Branch and Bound (LCBB): " + maxProfit);

}

}

**Slip 15 – Q.2) Write a program to implement Graph Coloring Algorithm.**

java

public class GraphColoring {

private int V;

private int[] result;

public GraphColoring(int V) {

this.V = V;

result = new int[V];

}

public boolean isSafe(int v, int[][] graph, int color) {

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

if (graph[v][i] == 1 && color == result[i])

return false;

}

return true;

}

public boolean graphColoringUtil(int[][] graph, int m, int v) {

if (v == V)

return true;

for (int color = 1; color <= m; color++) {

if (isSafe(v, graph, color)) {

result[v] = color;

if (graphColoringUtil(graph, m, v + 1))

return true;

result[v] = 0;

}

}

return false;

}

public void solve(int[][] graph, int m) {

if (!graphColoringUtil(graph, m, 0))

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

else {

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

System.out.println("Vertex " + i + " ---> Color " + result[i]);

}

}

public static void main(String[] args) {

int[][] graph = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3; // number of colors

GraphColoring gc = new GraphColoring(4);

gc.solve(graph, m);

}

}

**Slip 16 – Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using dynamic programming.**

java

public class KnapsackDP {

public static int knapSack(int W, int[] wt, int[] val, int n) {

int[][] K = new int[n+1][W+1];

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

for (int w = 0; w <= W; w++) {

if (i == 0 || w == 0)

K[i][w] = 0;

else if (wt[i-1] <= w)

K[i][w] = Math.max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

else

K[i][w] = K[i-1][w];

}

}

return K[n][W];

}

public static void main(String[] args) {

int[] val = {60, 100, 120};

int[] wt = {10, 20, 30};

int W = 50;

int n = val.length;

System.out.println("Maximum profit using DP: " + knapSack(W, wt, val, n));

}

}

**Slip 16 – Q.2) Write a program to determine if a given graph is a Hamiltonian cycle or not.**

java

import java.util.\*;

public class HamiltonianCycle {

private int V;

private int[] path;

public HamiltonianCycle(int V) {

this.V = V;

path = new int[V];

Arrays.fill(path, -1);

}

public boolean isSafe(int v, int[][] graph, int pos) {

if (graph[path[pos-1]][v] == 0)

return false;

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

if (path[i] == v)

return false;

return true;

}

public boolean hamCycleUtil(int[][] graph, int pos) {

if (pos == V) {

return graph[path[pos-1]][path[0]] == 1;

}

for (int v = 1; v < V; v++) {

if (isSafe(v, graph, pos)) {

path[pos] = v;

if (hamCycleUtil(graph, pos + 1))

return true;

path[pos] = -1;

}

}

return false;

}

public void hamCycle(int[][] graph) {

path[0] = 0;

if (!hamCycleUtil(graph, 1)) {

System.out.println("No Hamiltonian Cycle exists");

return;

}

System.out.println("Hamiltonian Cycle found:");

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

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

System.out.println(path[0]);

}

public static void main(String[] args) {

int[][] graph = {

{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0},

};

new HamiltonianCycle(5).hamCycle(graph);

}

}

**Slip 17 – Q.1) Write a program to implement solve ‘N’ Queens Problem using Backtracking.**

java

public class NQueens {

int N;

public NQueens(int N) {

this.N = N;

}

public void solve() {

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

if (!solveUtil(board, 0)) {

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

return;

}

printBoard(board);

}

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

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

if (board[row][i] == 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; j >= 0 && i < N; i++, j--)

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

return false;

return true;

}

public boolean solveUtil(int[][] board, int col) {

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

public void printBoard(int[][] board) {

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

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

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

System.out.println();

}

}

public static void main(String[] args) {

int N = 8; // Change N for different sizes

new NQueens(N).solve();

}

}

**Slip 17 – Q.2) Write a program to find out solution for 0/1 knapsack problem.**

*(This is the same as the DP solution already provided in Slip 16 Q.1.)*

**Slip 18 – Q.1) Write a program to implement Graph Coloring Algorithm.**

java

// (Same as previous Graph Coloring implementation)

public class GraphColoring\_Slip18 {

private int V;

private int[] result;

public GraphColoring\_Slip18(int V) {

this.V = V;

result = new int[V];

}

public boolean isSafe(int v, int[][] graph, int color) {

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

if (graph[v][i] == 1 && result[i] == color)

return false;

}

return true;

}

public boolean graphColoringUtil(int[][] graph, int m, int v) {

if (v == V)

return true;

for (int color = 1; color <= m; color++) {

if (isSafe(v, graph, color)) {

result[v] = color;

if (graphColoringUtil(graph, m, v + 1))

return true;

result[v] = 0;

}

}

return false;

}

public void solve(int[][] graph, int m) {

if (!graphColoringUtil(graph, m, 0))

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

else {

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

System.out.println("Vertex " + i + " ---> Color " + result[i]);

}

}

public static void main(String[] args) {

int[][] graph = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3;

new GraphColoring\_Slip18(4).solve(graph, m);

}

}

**Slip 18 – Q.2) Write a program to find out live node, E node and dead node from a given graph.**

java

// Here we provide a simple example:

// For demonstration, we assume:

// - A "live" node is one with degree >= 2,

// - An "E" node is one with even degree,

// - A "dead" node is isolated (degree 0).

import java.util.\*;

public class NodeClassification {

public static void classifyNodes(int[][] graph) {

int V = graph.length;

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

int degree = 0;

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

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

degree++;

}

String type;

if (degree == 0)

type = "Dead";

else if (degree % 2 == 0)

type = "E node (Even Degree)";

else if (degree >= 2)

type = "Live node";

else

type = "Undefined";

System.out.println("Node " + i + " (degree " + degree + ") is: " + type);

}

}

public static void main(String[] args) {

// Example graph as adjacency matrix

int[][] graph = {

{0, 1, 0, 0},

{1, 0, 1, 1},

{0, 1, 0, 0},

{0, 1, 0, 0}

};

classifyNodes(graph);

}

}

**Slip 19 – Q.1) Write a program to determine if a given graph is a Hamiltonian cycle or Not.**

java

// Using the HamiltonianCycle class from Slip 16 can be reused.

public class HamiltonianCycle\_Slip19 {

private int V;

private int[] path;

public HamiltonianCycle\_Slip19(int V) {

this.V = V;

path = new int[V];

Arrays.fill(path, -1);

}

public boolean isSafe(int v, int[][] graph, int pos) {

if (graph[path[pos-1]][v] == 0)

return false;

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

if (path[i] == v)

return false;

return true;

}

public boolean hamCycleUtil(int[][] graph, int pos) {

if (pos == V) {

return (graph[path[pos-1]][path[0]] == 1);

}

for (int v = 1; v < V; v++) {

if (isSafe(v, graph, pos)) {

path[pos] = v;

if (hamCycleUtil(graph, pos+1))

return true;

path[pos] = -1;

}

}

return false;

}

public void hamCycle(int[][] graph) {

path[0] = 0;

if (!hamCycleUtil(graph, 1)) {

System.out.println("No Hamiltonian Cycle exists");

return;

}

System.out.println("Hamiltonian Cycle is:");

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

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

System.out.println(path[0]);

}

public static void main(String[] args) {

int[][] graph = {

{0, 1, 1, 0},

{1, 0, 1, 1},

{1, 1, 0, 1},

{0, 1, 1, 0}

};

new HamiltonianCycle\_Slip19(4).hamCycle(graph);

}

}

**Slip 19 – Q.2) Write a program to show board configuration of 4 queens’ problem.**

java

// This is similar to the FourQueens code provided in Slip 10 Q.2.

public class FourQueensDisplay {

final int N = 4;

void printBoard(int board[][]) {

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

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

System.out.print((board[i][j] == 1 ? "Q " : ". "));

}

System.out.println();

}

}

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

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

if (board[row][i] == 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; j >= 0 && i < N; i++, j--)

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

return false;

return true;

}

boolean solveQueens(int board[][], int col) {

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveQueens(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

public void solve() {

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

if (!solveQueens(board, 0)) {

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

return;

}

printBoard(board);

}

public static void main(String[] args) {

new FourQueensDisplay().solve();

}

}

**Slip 19 – Q.3) Viva**

*(No code required.)*

**Slip 20 – Q.1) Write a program to implement for finding Topological sorting and determine the time complexity for the same.**

java

// This is similar to the TopologicalSort code provided in Slip 7 Q.2.

import java.util.\*;

public class TopologicalSort {

public static void topologicalSortUtil(int v, boolean[] visited, Stack<Integer> stack, List<List<Integer>> adj) {

visited[v] = true;

for (int neighbor : adj.get(v)) {

if (!visited[neighbor])

topologicalSortUtil(neighbor, visited, stack, adj);

}

stack.push(v);

}

public static void topologicalSort(List<List<Integer>> adj, int V) {

Stack<Integer> stack = new Stack<>();

boolean[] visited = new boolean[V];

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

if (!visited[i])

topologicalSortUtil(i, visited, stack, adj);

}

System.out.println("Topological Order:");

while (!stack.isEmpty())

System.out.print(stack.pop() + " ");

}

public static void main(String[] args) {

int V = 6;

List<List<Integer>> adj = new ArrayList<>();

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

adj.add(new ArrayList<>());

}

// Example DAG

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

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

adj.get(4).add(0);

adj.get(4).add(1);

adj.get(2).add(3);

adj.get(3).add(1);

long startTime = System.nanoTime();

topologicalSort(adj, V);

long endTime = System.nanoTime();

System.out.println("\nTime taken (ns): " + (endTime - startTime));

}

}

**Slip 20 – Q.2) Write a program to solve N Queens Problem using Backtracking.**

// This is similar to the NQueens code provided in Slip 17 Q.1.

public class NQueens {

int N;

public NQueens (int N) {

this.N = N;

}

public void solve() {

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

if (!solveUtil(board, 0)) {

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

return;

}

printBoard(board);

}

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

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

if (board[row][i] == 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; j >= 0 && i < N; i++, j--)

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

return false;

return true;

}

public boolean solveUtil(int[][] board, int col) {

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

public void printBoard(int[][] board) {

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

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

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

System.out.println();

}

}

public static void main(String[] args) {

int N = 8;

new NQueens (N).solve();

}

}