**Nilesh Shrestha**

B.Sc. (Hons.) Computing, Softwarica College of IT and E-Commerce, Coventry University

ST5008CEM Programming for Developers

Hikmat Saud

February 20, 2024

**Contents**

[Question 1a. 3](#_Toc159341318)

[Question 1b 7](#_Toc159341319)

[Question 2a 10](#_Toc159341320)

[Question 2b 13](#_Toc159341321)

[Question 3a 17](#_Toc159341322)

[Question 3b 22](#_Toc159341323)

[Question 4a 28](#_Toc159341324)

[Question 4b 35](#_Toc159341325)

[Question 5a 40](#_Toc159341326)

[Question 5b 50](#_Toc159341327)

[Question 6 57](#_Toc159341328)

[Question 7 79](#_Toc159341329)

[Introduction 79](#_Toc159341330)

[Challenges Faced 79](#_Toc159341331)

[Adaptation 79](#_Toc159341332)

[Features of the Modified Application: 80](#_Toc159341333)

[Suggestions for Future Improvements 81](#_Toc159341334)

[Advanced Graph Algorithms: 81](#_Toc159341335)

[Known Bugs or Issues that may occur 81](#_Toc159341336)

[Conclusion 82](#_Toc159341337)

[Appendix 83](#_Toc159341338)

# Question 1a.

public class Question1a {

public int calcuateDecorationCost(int cost[][]) {

int n = cost.length; // assuming that someone does not send a null value we can say that n is the

// lengt of rows of venue

// same way k is themes and n x k is cost for applying a theme on a venue

// so assuming this we get that k = cost[0].length assumiing one venue can be

// appied with multiple themes and has different cost

return calculateMinCost(cost, n - 1, -1); // calling the calculate min cost function

// passing n-1 because n is the lenght and array's index starts with 0 so if the

// length is 3 then array is something like [0,1,2] showing this to visualize

// how array is indexed not to be mistaken with value inside the array

// passing -1 as theme because at the start we assume that no theme is selected

// for venue

}

public int calculateMinCost(int[][] cost, int venue, int theme) {

// making sure that venue does not go below 0

// you can also add a validation for checking if costs sent is null or themes

// are less than 0

if (venue < 0) { // (venu<0 || cost.length== null || cost[0].length==0) but i'm assuming data

// cannot be null

return 0;

}

int minCost = Integer.MAX\_VALUE; // setting maximum possible value for integer that is possible

for (int i = 0; i < cost[venue].length; i++) {

// running a loop here

// (theme == -1 || theme != i) this will assume that theme ==-1 and theme !=i

// are the same so loop will run no matter what

if (theme == -1 || theme != i) {

int venueCost = cost[venue][i];

// will only give the the last row of the array or arrays, that is [3,1,5] and

// this will serve as the base case to going through recursion after the first

// base case being -1 as no theme was selected

int recursiveCost = calculateMinCost(cost, venue - 1, i);// using recursion and passing the venue as 2

minCost = Math.min(minCost, venueCost + recursiveCost);

// then 1 then 0 and i as 0, 1,2 to calculate

// the min cost of [3,1,5] i.e. 1 and so on untill the recursion ends when the

// counter hits 0

// System.out.println(minCost);

}

}

return minCost;// returning minimum cost

}

public static void main(String[] args) {

Question1a q = new Question1a();

int[][] costs = {

{ 1, 3, 2 },

{ 4, 6, 8 },

{ 3, 1, 5 },

// { 9, 8, 5 } // adding this should give me a min cost of 12

};

System.out.println(q.calcuateDecorationCost(costs)); // printing min cost

}

}

# Question 1b

class MinimumTimeToBuildEngines {

// This method calculates the minimum time required to build engines with given parameters.

public static int minTimeToBuildEngines(int[] engines, int splitCost) {

int n = engines.length; // Number of engines

// dp[i][j] represents the minimum time to build the first i engines with j engineers

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

// Initialize dp array with infinity

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

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

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

}

}

// Base case: building 0 engines with 0 engineers takes 0 time

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

dp[0][j] = 0;

}

// Nested loops to fill in the dynamic programming table

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

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

// Option 1: Splitting engineers

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

// Update dp[i][j] with the minimum of the current value and the split cost

dp[i][j] = Math.min(dp[i][j], Math.max(dp[k - 1][j - 1], sum(engines, k, i)) + splitCost);

}

}

}

// Return the minimum time to build all engines with n engineers

return dp[n][n];

}

// Helper method to calculate the sum of elements in the array from start to end

private static int sum(int[] engines, int start, int end) {

int result = 0;

for (int i = start - 1; i < end; i++) {

result += engines[i];

}

return result;

}

// Main method to test the functionality

public static void main(String[] args) {

int[] engines = {1, 2, 3}; // Array representing time required to build each engine

int splitCost = 1; // Cost of splitting engineers

// Call the method to calculate and print the minimum time needed to build all engines

int result = minTimeToBuildEngines(engines, splitCost);

System.out.println("Minimum time needed to build all engines: " + result);

}

}

# Question 2a

class MinMovesToEqualizeDresses {

public static int minMovesToEqualizeDresses(int[] dresses) {

int n = dresses.length;

// Calculate the total number of dresses

int totalDresses = 0;

for (int dress : dresses) {

totalDresses += dress;

}

// Check if it's possible to equalize the dresses

if (totalDresses % n != 0) {

return -1;

}

// Calculate the target number of dresses for each machine

int targetDresses = totalDresses / n;

int moves = 0;

// Iterate through the machines and calculate moves

for (int dress : dresses) {

// Calculate the difference between the current machine's dresses and the target

int diff = dress - targetDresses;

if (diff > 0) {

moves += diff;

}

// Update the moves with the absolute value of the current sum

}

return moves;

}

public static void main(String[] args) {

// Example usage:

int[] inputDresses = { 1, 0, 5 };

int outputMoves = minMovesToEqualizeDresses(inputDresses);

System.out.println(outputMoves);

}

}

# Question 2b

import java.util.HashSet;

import java.util.Set;

public class Question2b {

/\*

\* according to question lets say 0 can share secret with only 2 and first

\* person is 0 that has a secret

\* then he needs to share the secret with everyone according to the question but

\* how does he do it ?

\* so we create a set for it so that the first person does not share the secret

\* with same person again and again

\* according to question {{0,2},{1,3},{2,4}} so first person shares his secret

\* first time with 2 in the first interval that is

\* first interval is intervals[0] not to confuse it with interval in the for

\* each loop

\* ;

\*/

public Set<Integer> shareSecretWithIndividuals(int n, int firstPerson, int intervals[][]) {

Set<Integer> result = new HashSet<>(); // to prevent the repition of people who know the secret

result.add(firstPerson);

for (int[] interval : intervals) {

int start = interval[0];

int end = interval[1];

// System.out.println(start); // this gives something like [0,2][1,3] and [2,4]

// System.out.println(end); // if you print the start and end together instead of doing it in different lines like i did

// again going into loop to get the persons who the first person shares the secret in the first interval

//since we know that start is 0 then and the start is interval[0] is also 0 so we know that the first person 0 shared secret with is 2

// on the second iteration 2 has to share secret with another person [1,3] which should be something lik e[0,2][2,4],[2,1][1,3]

//since 2 shares secret multiple times to fulfill the condition of output [0, 1, 2, 3, 4] ([2,1] is an assumption)

//or else 1 would not be able to know the secret and 3 will not be able to know the secret simultaenously

//so 1 gets added since condition is its either person is greater than start or less than end from first iteration i.e (0,2) then

//on second iteration its (1,3) then you have to get 1 to fulfill the above condition again loop runs and 4

for (int person : interval) {

if (start <= person && person <= end) {

result.add(person);

// System.out.println(result);

}

}

}

return result;

}

public static void main(String[] args) {

int n = 5;

int[][] intervals = { { 0, 2 }, { 1, 3 }, { 2, 4 } };

int firstPerson = 0;

Question2b q = new Question2b();

Set<Integer> output = q.shareSecretWithIndividuals(n, firstPerson, intervals);

System.out.println("output: " + output);

}

}

# Question 3a

import java.util.Collections;

import java.util.PriorityQueue;

class ScoreTracker {

private PriorityQueue<Double> minHeap;

private PriorityQueue<Double> maxHeap;

public ScoreTracker() {

minHeap = new PriorityQueue<>();

maxHeap = new PriorityQueue<>(Collections.reverseOrder()); // maxheap: [85.5, 77.8] becomes something like this

// with collection.reverseOrder()

}

void addScore(double score) {

if (maxHeap.isEmpty() || score < maxHeap.peek()) {

maxHeap.add(score); // the firs value 85.5 will always be added to the max heap

// then it will always compare the other values to 85.5 and if its less it will

// add to max heap else it will go to min heap i.e. value > 85.5 =>[88.7, 92.3,

// 90.1]

// System.out.println("maxheap: " + maxHeap);

} else {

minHeap.add(score);

// System.out.println("minheap: " + minHeap);

// values that will be in min heap: [88.7, 92.3, 90.1]

}

balanceHeap();

// System.out.println("after balance");

// System.out.println("maxheap: " + maxHeap);

// System.out.println("minheap: " + minHeap);

}

public double getMedianScore() {

if (maxHeap.size() == minHeap.size()) {

return (maxHeap.peek() + minHeap.peek()) / 2; // same as comment in line 66-67

} else {

return minHeap.peek();// middle value if it is odd number data set

}

}

public void balanceHeap() {

// max heap should always be bigger or equal in size to min heap

// as median means middle number in a set of data i.e [1,2,3,4,5] the middle

// number is 3

// or median is the average of two middle numbers if the data is set of odd

// numbers i.e. [1,2,3,4,5,6] median == 3+4/2 => 3.5

if (maxHeap.size() > minHeap.size()) {

minHeap.add(maxHeap.poll());

// so that median is always the lower half of max heap that is max

} else if (minHeap.size() > maxHeap.size() + 1) {

maxHeap.add(minHeap.poll());

// to satisfy the condition of median same as when doing the if block just the

// opposite case

}

}

public static void main(String[] args) {

ScoreTracker st = new ScoreTracker();

st.addScore(85.5);

st.addScore(92.3);

st.addScore(77.8);

double median1 = st.getMedianScore();

System.out.println("Median 1: " + median1);

st.addScore(90.1);

st.addScore(81.2);

st.addScore(88.7);

double median2 = st.getMedianScore();

System.out.println("Median 2: " + median2);

}

}

# Question 3b

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

// Define a class to represent an edge in the graph

class Edge implements Comparable<Edge> {

int source, destination, weight;

// Constructor to initialize the edge with source, destination, and weight

public Edge(int source, int destination, int weight) {

this.source = source;

this.destination = destination;

this.weight = weight;

}

// Implement the Comparable interface to compare edges based on their weights

@Override

public int compareTo(Edge e) {

return Integer.compare(this.weight, e.weight);

}

}

// Define the KruskalAlgorithm class

class KruskalAlgorithm {

// Find operation in Union-Find data structure

public static int find(int[] parent, int x) {

// If the current element is not the root, recursively find the root

if (parent[x] != x) {

parent[x] = find(parent, parent[x]);

}

return parent[x];

}

// Union operation in Union-Find data structure

private static void union(int[] parent, int[] rank, int x, int y) {

int rootX = find(parent, x);

int rootY = find(parent, y);

// Merge two sets based on their ranks

if (rootX != rootY) {

if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else {

parent[rootX] = rootY;

rank[rootY]++;

}

}

}

// Kruskal's algorithm to find the Minimum Spanning Tree

public static List<Edge> Kruskal(List<Edge> edges, int vertices) {

// List to store the edges in the Minimum Spanning Tree

List<Edge> minimumSpanningTree = new ArrayList<>();

// Sort the edges in ascending order based on weights

Collections.sort(edges);

// Initialize the Union-Find data structure

int parent[] = new int[vertices];

int rank[] = new int[vertices];

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

parent[i] = i; // Each vertex is initially its own parent

rank[i] = 0; // Initialize the rank of each set to 0

}

// Iterate through the sorted edges

for (Edge edge : edges) {

int rootSrc = find(parent, edge.source);

int rootDest = find(parent, edge.destination);

// Check if adding the edge would create a cycle

if (rootSrc != rootDest) {

minimumSpanningTree.add(edge); // Add the edge to the Minimum Spanning Tree

union(parent, rank, rootSrc, rootDest); // Union the sets of source and destination vertices

}

}

// Return the Minimum Spanning Tree

return minimumSpanningTree;

}

// Main method to test the Kruskal's algorithm

public static void main(String[] args) {

// Create a sample graph with edges and vertices

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

edges.add(new Edge(0, 1, 4));

edges.add(new Edge(0, 2, 3));

edges.add(new Edge(1, 2, 2));

edges.add(new Edge(1, 3, 1));

edges.add(new Edge(2, 3, 5));

int vertices = 5; // Number of vertices in the graph

// Call the Kruskal algorithm to find the Minimum Spanning Tree

List<Edge> minimumSpanningTree = Kruskal(edges, vertices);

// Print the edges in the Minimum Spanning Tree

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

for (Edge edge : minimumSpanningTree) {

System.out.println(edge.source + " - " + edge.destination + " : " + edge.weight);

}

}

}

# Question 4a

import java.util.ArrayDeque;

import java.util.HashSet;

import java.util.Queue;

import java.util.Set;

class MazeSolver {

// Function to find the minimum number of moves required to collect keys 'a' to

// 'f'

public static int minMovesToCollectKeys(char[][] grid) {

int m = grid.length; // Number of rows in the grid

int n = grid[0].length; // Number of columns in the grid

int keysCount = 0; // Count of total keys in the maze

Set<Character> targetKeys = new HashSet<>(); // Set to store target keys ('a' to 'f')

for (char[] row : grid) {

for (char cell : row) {

if ('a' <= cell && cell <= 'f') {

keysCount++; // Increment the key count for each lowercase letter 'a' to 'f'

targetKeys.add(cell); // Add the lowercase letter to the set of target keys

}

}

}

int[][] directions = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } }; // Possible movement directions

Set<String> visited = new HashSet<>(); // Set to track visited states during BFS

Queue<State> queue = new ArrayDeque<>(); // Queue for BFS

// Find the starting position 'S' and initialize the BFS queue

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

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

if (grid[i][j] == 'S') {

queue.add(new State(i, j, 0, new HashSet<>())); // Add the starting state to the queue

visited.add(i + "-" + j + "-"); // Mark the starting state as visited

break;

}

}

}

// BFS loop

while (!queue.isEmpty()) {

State current = queue.poll(); // Dequeue the current state

// Check if all keys 'a' to 'f' are collected

if (current.collectedKeys.size() == keysCount) {

return current.steps; // Return the minimum number of steps if all keys are collected

}

// Explore possible next states in all directions

for (int[] dir : directions) {

int nx = current.x + dir[0];

int ny = current.y + dir[1];

// Check if the next position is valid and not a wall

if (isValid(nx, ny, m, n) && grid[nx][ny] != 'W') {

char cell = grid[nx][ny];

// If the cell contains a new key 'a' to 'f', add it to the set of collected

// keys

if ('a' <= cell && cell <= 'f' && !current.collectedKeys.contains(cell)) {

Set<Character> newCollectedKeys = new HashSet<>(current.collectedKeys);

newCollectedKeys.add(cell);

String newState = nx + "-" + ny + "-" + newCollectedKeys;

if (!visited.contains(newState)) {

queue.add(new State(nx, ny, current.steps + 1, newCollectedKeys));

visited.add(newState);

}

}

// If the cell contains a locked door 'A' to 'F' and the corresponding key is

// collected, proceed

else if ('A' <= cell && cell <= 'F'

&& current.collectedKeys.contains(Character.toLowerCase(cell))) {

String newState = nx + "-" + ny + "-" + current.collectedKeys;

if (!visited.contains(newState)) {

queue.add(new State(nx, ny, current.steps + 1, current.collectedKeys));

visited.add(newState);

}

}

// If the cell is a path or already collected key 'a' to 'f', proceed

else if (cell == 'P' || ('a' <= cell && cell <= 'f' && current.collectedKeys.contains(cell))) {

String newState = nx + "-" + ny + "-" + current.collectedKeys;

if (!visited.contains(newState)) {

queue.add(new State(nx, ny, current.steps + 1, current.collectedKeys));

visited.add(newState);

}

}

}

}

}

return -1; // Return -1 if it's impossible to collect all keys 'a' to 'f' and reach the

// exit

}

// Function to check if a position is valid within the grid

private static boolean isValid(int x, int y, int m, int n) {

return x >= 0 && x < m && y >= 0 && y < n;

}

// Class representing the state of the game

static class State {

int x, y, steps;

Set<Character> collectedKeys;

// Constructor for creating a new state

public State(int x, int y, int steps, Set<Character> collectedKeys) {

this.x = x;

this.y = y;

this.steps = steps;

this.collectedKeys = collectedKeys;

}

}

// Main function for testing the maze solver

public static void main(String[] args) {

char[][] grid = {

{ 'S', 'P', 'a', 'P', 'P' },

{ 'W', 'W', 'W', 'P', 'W' },

{ 'b', 'P', 'B', 'P', 'C' }

};

int result = minMovesToCollectKeys(grid);

System.out.println(result); // Output: 8

}

}

# Question 4b

import java.util.LinkedList;

import java.util.List;

// Definition of a binary tree and methods to create a BST and find closest values

class Tree {

// Definition of a Node in the binary tree

public static class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

this.left = this.right = null;

}

}

// Method to create a Binary Search Tree (BST)

Node createBST(Node root, int data) {

// If the tree is empty, create a new node with the given data

if (root == null)

return new Node(data);

// If the data is less than the root's data, go to the left subtree

if (data < root.data) {

root.left = createBST(root.left, data);

}

// If the data is greater than the root's data, go to the right subtree

else if (data > root.data) {

root.right = createBST(root.right, data);

}

// If the data is a duplicate, print a message

else {

System.out.println("Duplicate entry of " + data);

}

return root;

}

// Inorder traversal to find the closest values to a target value

private void findClosestValues(Node root, double target, int k, LinkedList<Integer> closest) {

// Base case: If the current node is null, return

if (root == null)

return;

// Recursively traverse the left subtree

findClosestValues(root.left, target, k, closest);

// If the list has reached its capacity of k elements, check if the farthest element should be removed

if (closest.size() == k) {

if (Math.abs(target - closest.peekFirst()) > Math.abs(target - root.data)) {

closest.removeFirst();

} else {

// If the current element is not closer than the farthest in the list, stop the process

return;

}

}

// Add the current node's data to the list

closest.add(root.data);

// Recursively traverse the right subtree

findClosestValues(root.right, target, k, closest);

}

// Public method to initiate the closest value search

public List<Integer> findClosest(Node root, double target, int k) {

// Create a linked list to store the closest values

LinkedList<Integer> closest = new LinkedList<>();

// Call the private helper method to find closest values

findClosestValues(root, target, k, closest);

return closest;

}

// Main method to demonstrate the functionality

public static void main(String[] args) {

// Example usage:

Tree tree = new Tree();

Node root = null;

int[] values = { 4, 2, 5, 1, 3 };

// Create a BST with the given values

for (int value : values) {

root = tree.createBST(root, value);

}

// Specify the target value and the number of closest values to find

double target = 3.8;

int k = 2;

// Find and print the closest values to the target value

List<Integer> closestValues = tree.findClosest(root, target, k);

System.out.println(closestValues);

}

}

# Question 5a

// a) Implement ant colony algorithm solving travelling a salesman problem

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.Random;

// Class representing an Ant Colony for solving the Traveling Salesman Problem

class AntColony {

private double[][] pheromones; // Matrix to store pheromone levels on edges

private double[][] distances; // Matrix representing distances between nodes

private int nAnts; // Number of ants in the colony

private double decay; // Rate at which pheromones decay

private double alpha; // Weight of pheromones in decision-making

private double beta; // Weight of distances in decision-making

// Constructor to initialize the AntColony with distances, number of ants, decay rate, alpha, and beta values

public AntColony(double[][] distances, int nAnts, double decay, double alpha, double beta) {

this.distances = distances;

this.nAnts = nAnts;

this.decay = decay;

this.alpha = alpha;

this.beta = beta;

int nNodes = distances.length;

this.pheromones = new double[nNodes][nNodes];

// Initialize pheromones to 1.0

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

Arrays.fill(pheromones[i], 1.0);

}

}

// Method to find the optimal tour by running the ant colony optimization algorithm

public List<Integer> findOptimalTour() {

int nNodes = distances.length;

List<Integer> bestTour = null;

double bestTourLength = Double.POSITIVE\_INFINITY;

// Run the optimization algorithm for a fixed number of iterations

for (int iteration = 0; iteration < 100; iteration++) {

List<List<Integer>> antTours = generateAntTours();

updatePheromones(antTours);

// Iterate over each ant's tour and update the best tour if needed

for (List<Integer> tour : antTours) {

double tourLength = calculateTourLength(tour);

if (tourLength < bestTourLength) {

bestTourLength = tourLength;

bestTour = new ArrayList<>(tour);

}

}

// Decay pheromones after each iteration

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

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

pheromones[i][j] \*= decay;

}

}

}

return bestTour;

}

// Method to generate tours for each ant in the colony

private List<List<Integer>> generateAntTours() {

int nNodes = distances.length;

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

// Iterate over each ant in the colony

for (int ant = 0; ant < nAnts; ant++) {

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

boolean[] visited = new boolean[nNodes];

int startNode = new Random().nextInt(nNodes);

tour.add(startNode);

visited[startNode] = true;

// Build the tour for each ant

for (int step = 1; step < nNodes; step++) {

int nextNode = selectNextNode(tour, visited);

tour.add(nextNode);

visited[nextNode] = true;

}

antTours.add(tour);

}

return antTours;

}

// Method to select the next node for an ant based on pheromone and distance information

private int selectNextNode(List<Integer> tour, boolean[] visited) {

int currentNode = tour.get(tour.size() - 1);

int nNodes = distances.length;

double[] probabilities = new double[nNodes];

double sum = 0;

// Calculate probabilities for unvisited nodes

for (int nextNode = 0; nextNode < nNodes; nextNode++) {

if (!visited[nextNode]) {

double pheromone = Math.pow(pheromones[currentNode][nextNode], alpha);

double distance = Math.pow(1.0 / distances[currentNode][nextNode], beta);

probabilities[nextNode] = pheromone \* distance;

sum += probabilities[nextNode];

}

}

// Roulette wheel selection

double rouletteWheel = new Random().nextDouble() \* sum;

double cumulativeProbability = 0;

for (int nextNode = 0; nextNode < nNodes; nextNode++) {

if (!visited[nextNode]) {

cumulativeProbability += probabilities[nextNode];

if (cumulativeProbability >= rouletteWheel) {

return nextNode;

}

}

}

// In case of rounding errors

return -1;

}

// Method to update pheromones based on ant tours

private void updatePheromones(List<List<Integer>> antTours) {

int nNodes = distances.length;

// Decay existing pheromones

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

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

pheromones[i][j] \*= (1 - decay);

}

}

// Update pheromones based on ant tours

for (List<Integer> tour : antTours) {

double tourLength = calculateTourLength(tour);

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

int fromNode = tour.get(i);

int toNode = tour.get(i + 1);

pheromones[fromNode][toNode] += 1.0 / tourLength;

pheromones[toNode][fromNode] += 1.0 / tourLength;

}

}

}

// Method to calculate the length of a tour based on distances

private double calculateTourLength(List<Integer> tour) {

double length = 0;

for (int i = 0; i < tour.size() - 1; i++) {

int fromNode = tour.get(i);

int toNode = tour.get(i + 1);

length += distances[fromNode][toNode];

}

return length;

}

// Main method to demonstrate the functionality

public static void main(String[] args) {

// Example usage:

double[][] distances = {

{0, 2, 3, 4},

{2, 0, 5, 6},

{3, 5, 0, 7},

{4, 6, 7, 0}

};

int nAnts = 5;

double decay = 0.1;

double alpha = 1;

double beta = 2;

AntColony antColony = new AntColony(distances, nAnts, decay, alpha, beta);

List<Integer> optimalTour = antColony.findOptimalTour();

System.out.println("Optimal Tour: " + optimalTour);

}

}

# Question 5b

import java.util.\*;

// Class Q5B for finding impacted devices in a network

class Q5B {

int[] disc, low; // Arrays to store discovery time and low time during DFS

int time = 1; // Variable to track time during DFS

List<List<Integer>> ans = new ArrayList<>(); // List to store connections causing impacts

Map<Integer, List<Integer>> edgeMap = new HashMap<>(); // Map to represent the network connections

// Method to find impacted devices given the network parameters and a target device

public List<Integer> findImpactedDevices(int n, List<List<Integer>> connections, int targetDevice) {

disc = new int[n];

low = new int[n];

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

edgeMap.put(i, new ArrayList<Integer>());

// Populate edgeMap based on the given connections

for (List<Integer> conn : connections) {

edgeMap.get(conn.get(0)).add(conn.get(1));

edgeMap.get(conn.get(1)).add(conn.get(0));

}

// Perform DFS to identify connections causing impacts

dfs(targetDevice, -1);

// Check if the target device is a source node in any connection

boolean isSourceNode = false;

for (List<Integer> conn : connections) {

if (conn.get(0) == targetDevice) {

isSourceNode = true;

break;

}

}

// If the target device is not a source node, return an empty list

if (!isSourceNode) {

return new ArrayList<>();

}

// Set to store impacted devices

Set<Integer> impactedDevicesSet = new HashSet<>();

// Identify impacted devices based on the connections causing impacts

for (List<Integer> connection : ans) {

int u = connection.get(0);

int v = connection.get(1);

if (u == targetDevice) {

impactedDevicesSet.add(v);

} else if (v == targetDevice) {

impactedDevicesSet.add(u);

}

}

// Set to store additional affected devices

Set<Integer> additionalAffectedDevices = new HashSet<>();

// Identify additional affected devices based on neighbors of impacted devices

for (int affectedDevice : impactedDevicesSet) {

for (int neighbor : edgeMap.get(affectedDevice)) {

if (!impactedDevicesSet.contains(neighbor)) {

additionalAffectedDevices.add(neighbor);

}

}

}

// Combine impacted devices and additional affected devices

impactedDevicesSet.addAll(additionalAffectedDevices);

impactedDevicesSet.remove(targetDevice); // Remove the target device from the result

return new ArrayList<>(impactedDevicesSet);

}

// Depth First Search (DFS) to find connections causing impacts

public void dfs(int curr, int prev) {

disc[curr] = low[curr] = time++;

for (int next : edgeMap.get(curr)) {

if (next == prev)

continue;

if (disc[next] == 0) {

dfs(next, curr);

low[curr] = Math.min(low[curr], low[next]);

if (low[next] > disc[curr])

ans.add(Arrays.asList(curr, next));

} else {

low[curr] = Math.min(low[curr], disc[next]);

}

}

}

// Main method to demonstrate the functionality

public static void main(String[] args) {

Q5B q5B = new Q5B();

int n = 8;

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

connections.add(Arrays.asList(0, 1));

connections.add(Arrays.asList(0, 2));

connections.add(Arrays.asList(1, 3));

connections.add(Arrays.asList(1, 6));

connections.add(Arrays.asList(2, 4));

connections.add(Arrays.asList(4, 6));

connections.add(Arrays.asList(4, 5));

connections.add(Arrays.asList(5, 7));

int targetDevice = 4;

List<Integer> impactedDevices = q5B.findImpactedDevices(n, connections, targetDevice);

System.out.println("Impacted Devices (other than target device " + targetDevice + "): " + impactedDevices);

}

}

# Question 6

import javax.swing.\*;

import java.awt.\*;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.HttpURLConnection;

import java.net.MalformedURLException;

import java.net.URL;

import java.nio.file.\*;

import java.text.MessageFormat;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.atomic.AtomicBoolean;

// ExtendedSwingFrame class extending JFrame

class ExtendedSwingFrame extends JFrame {

// Components for the GUI

private JTextField textField;

private JButton addUrlButton;

private JButton downloadButton;

private JButton clearUrlButton;

private JButton cancelButton;

private JPanel progressBarPanel;

// ExecutorService for managing threads

private ExecutorService executorService;

// Lists to store URLs, progress bars, and download workers

private List<String> urlList;

private List<JProgressBar> progressBars;

private List<DownloadWorker> workers;

// AtomicBoolean for managing download cancellation

private AtomicBoolean isCanceled;

// Constructor for the ExtendedSwingFrame

public ExtendedSwingFrame() {

// Set frame properties

setTitle("Extended Swing Frame");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

// Create a panel for components

JPanel panel = new JPanel();

// Initialize GUI components

textField = new JTextField(20);

addUrlButton = new JButton("Add URL");

downloadButton = new JButton("Download Images");

clearUrlButton = new JButton("Clear URLs");

cancelButton = new JButton("Cancel All Downloads");

progressBarPanel = new JPanel(new GridLayout(0, 1));

// Initialize lists and AtomicBoolean

urlList = new ArrayList<>();

progressBars = new ArrayList<>();

workers = new ArrayList<>();

isCanceled = new AtomicBoolean(false);

// Add ActionListener for "Add URL" button

addUrlButton.addActionListener(e -> {

// Get URL from text field, add to list, and add progress bar

String imageUrl = textField.getText();

if (!imageUrl.isEmpty()) {

urlList.add(imageUrl);

textField.setText("");

addProgressBar(urlList.size());

}

});

// Add ActionListener for "Download Images" button

downloadButton.addActionListener(e -> {

// Download images for each URL in the list

if (!urlList.isEmpty()) {

for (int i = 0; i < urlList.size(); i++) {

downloadImage(urlList.get(i), progressBars.get(i), i + 1);

}

} else {

// Show message if no URLs to download

JOptionPane.showMessageDialog(this,

"No URLs to download.",

"Info", JOptionPane.INFORMATION\_MESSAGE);

}

});

// Add ActionListener for "Clear URLs" button

clearUrlButton.addActionListener(e -> {

// Clear URL list and progress bars

urlList.clear();

progressBars.clear();

progressBarPanel.removeAll();

progressBarPanel.revalidate();

progressBarPanel.repaint();

});

// Add ActionListener for "Cancel All Downloads" button

cancelButton.addActionListener(e -> {

// Set cancellation flag and cancel all downloads

isCanceled.set(true);

cancelAllDownloads();

});

// Add components to the panel

panel.add(new JLabel("Image URL:"));

panel.add(textField);

panel.add(addUrlButton);

panel.add(downloadButton);

panel.add(clearUrlButton);

panel.add(cancelButton);

panel.add(progressBarPanel);

// Add panel to the content pane

getContentPane().add(panel);

// Set frame size and make it visible

setSize(400, 300);

setVisible(true);

// Initialize ExecutorService with a fixed thread pool

executorService = Executors.newFixedThreadPool(10);

}

// Method to add a progress bar to the GUI

private void addProgressBar(int imageNumber) {

// Create label, progress bar, and pause button for each image

JLabel label = new JLabel("Image " + imageNumber + ": ");

JProgressBar progressBar = new JProgressBar(0, 100);

progressBar.setStringPainted(true);

JButton pauseButton = new JButton("Pause");

pauseButton.addActionListener(e -> {

// Pause download when the pause button is clicked

int workerIndex = findWorkerIndexByProgressBar(progressBar);

if (workerIndex != -1) {

workers.get(workerIndex).pauseDownload();

}

});

// Create button panel and progress bar panel

JPanel buttonPanel = new JPanel();

buttonPanel.add(pauseButton);

JPanel progressBarPanel = new JPanel(new BorderLayout());

progressBarPanel.add(label, BorderLayout.WEST);

progressBarPanel.add(progressBar, BorderLayout.CENTER);

progressBarPanel.add(buttonPanel, BorderLayout.EAST);

// Add progress bar panel to the main panel and progress bar to the list

this.progressBarPanel.add(progressBarPanel);

progressBars.add(progressBar);

// Update GUI

revalidate();

repaint();

// Introduce a slight delay before initiating download

try {

Thread.sleep(50);

} catch (InterruptedException e) {

e.printStackTrace();

}

// Initiate image download

downloadImage(urlList.get(imageNumber - 1), progressBar, imageNumber);

}

// Method to cancel all ongoing downloads

private void cancelAllDownloads() {

if (workers != null && !workers.isEmpty()) {

for (DownloadWorker worker : workers) {

worker.cancel(true);

}

}

}

// Method to find the index of a DownloadWorker based on its progress bar

private int findWorkerIndexByProgressBar(JProgressBar progressBar) {

for (int i = 0; i < progressBars.size(); i++) {

if (progressBars.get(i) == progressBar) {

return i;

}

}

return -1;

}

// Method to initiate the download of an image

private void downloadImage(String imageUrl, JProgressBar progressBar, int imageNumber) {

DownloadWorker worker = new DownloadWorker(imageUrl, progressBar, imageNumber);

executorService.execute(worker);

workers.add(worker);

}

// Inner class representing the DownloadWorker

private class DownloadWorker extends SwingWorker<Void, Integer> {

private final String imageUrl;

private final JProgressBar progressBar;

private final int imageNumber;

private final AtomicBoolean isPaused;

// Constructor for the DownloadWorker

public DownloadWorker(String imageUrl, JProgressBar progressBar, int imageNumber) {

this.imageUrl = imageUrl;

this.progressBar = progressBar;

this.imageNumber = imageNumber;

this.isPaused = new AtomicBoolean(false);

}

// Method to pause the download

public void pauseDownload() {

isPaused.set(!isPaused.get());

if (isPaused.get()) {

JOptionPane.showMessageDialog(ExtendedSwingFrame.this,

"Download paused for Image " + imageNumber,

"Paused", JOptionPane.INFORMATION\_MESSAGE);

}

}

// Override doInBackground method for background task

@Override

protected Void doInBackground() {

try {

// Check if the URL is valid

if (!isValidUrl(imageUrl)) {

throw new MalformedURLException("Invalid URL: " + imageUrl);

}

URL url = new URL(imageUrl);

// Open a connection to the URL

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

int responseCode = connection.getResponseCode();

// Check for a successful HTTP response

if (responseCode != HttpURLConnection.HTTP\_OK) {

throw new IOException("HTTP error code: " + responseCode);

}

String contentType = connection.getContentType();

// Move the disconnection after obtaining the content length

int contentLength = connection.getContentLength();

connection.disconnect();

// Check if the content type is an image

if (contentType == null || !contentType.startsWith("image")) {

throw new IOException("URL does not point to an image: " + imageUrl);

}

// Extract the file name from the URL

String fileName = url.getFile();

fileName = fileName.substring(fileName.lastIndexOf('/') + 1);

// Handle query parameters in the file name

fileName = fileName.split("\\?")[0];

// Generate a dynamic file name to avoid conflicts

String newFileName = getDynamicFileName(System.getProperty("user.home") + "/Desktop/", fileName);

Path outputPath = Paths.get(System.getProperty("user.home") + "/Desktop/", newFileName);

// Create necessary directories

Files.createDirectories(outputPath.getParent());

// Initialize variables for download progress

int totalBytesRead = 0;

// Open input stream from the URL and output stream to the local file

try (InputStream in = url.openStream();

OutputStream out = Files.newOutputStream(outputPath)) {

// Buffer for reading data

byte[] buffer = new byte[1024];

int bytesRead;

// Loop to read and write data

while ((bytesRead = in.read(buffer)) != -1) {

// Check for pause and wait until resumed

if (isPaused.get()) {

while (isPaused.get()) {

if (isCancelled()) {

return null;

}

try {

Thread.sleep(50);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

return null;

}

}

}

// Write data to the output stream

out.write(buffer, 0, bytesRead);

totalBytesRead += bytesRead;

// Calculate and publish download progress

int progress = (int) ((double) totalBytesRead / contentLength \* 100);

publish(progress);

// Introduce a delay to slow down the download progress

try {

Thread.sleep(100); // Adjust the sleep duration as needed

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

return null;

}

}

}

// Publish 100% progress when download is complete

publish(100);

// Move the downloaded file to the final location

Files.move(Paths.get(System.getProperty("user.home") + "/Desktop/", fileName),

outputPath, StandardCopyOption.REPLACE\_EXISTING);

} catch (MalformedURLException e) {

// Handle MalformedURLException and show error message

e.printStackTrace();

showError("Error downloading image: Invalid URL", imageUrl);

} catch (IOException e) {

// Handle IOException and show error message

e.printStackTrace();

showError("Error downloading image: " + e.getMessage(), imageUrl);

}

return null;

}

// Override process method to update the progress bar

@Override

protected void process(List<Integer> chunks) {

for (int progress : chunks) {

progressBar.setValue(progress);

}

}

// Override done method for cleanup after background task completion

@Override

protected void done() {

// Check if all download workers are done or canceled

boolean allDone = true;

for (DownloadWorker worker : workers) {

if (!worker.isDone() || worker.isCancelled()) {

allDone = false;

break;

}

}

// Show success message if all downloads are complete

if (allDone) {

urlList.clear();

JOptionPane.showMessageDialog(ExtendedSwingFrame.this,

"All images downloaded successfully!",

"Success", JOptionPane.INFORMATION\_MESSAGE);

}

}

}

// Method to check if a given string is a valid URL

private boolean isValidUrl(String urlString) {

try {

new URL(urlString).toURI();

return true;

} catch (Exception e) {

return false;

}

}

// Method to generate a dynamic file name to avoid conflicts

private String getDynamicFileName(String directory, String fileName) {

String baseName = fileName.substring(0, Math.min(fileName.lastIndexOf('.'), 255));

String extension = fileName.substring(fileName.lastIndexOf('.'));

Path filePath = Paths.get(directory, fileName);

int count = 1;

// Check for existing file and generate a new name if needed

while (Files.exists(filePath)) {

String newFileName = MessageFormat.format("{0}\_{1}{2}", baseName, count++, extension);

filePath = Paths.get(directory, newFileName);

}

return filePath.getFileName().toString();

}

// Method to show an error message dialog

private void showError(String message, String imageUrl) {

SwingUtilities.invokeLater(() -> {

JOptionPane.showMessageDialog(this,

message + "\nURL: " + imageUrl,

"Error", JOptionPane.ERROR\_MESSAGE);

});

}

// Main method to launch the Swing application

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> new ExtendedSwingFrame());

}

}

# Question 7

### Introduction

The original assignment tasked the development of a graph-based recommendation system, a task that encountered obstacles owing to limited familiarity with data structures and algorithms, temporal constraints, and a knowledge gap. This report outlines the adaptation of the project to create a GUI application for social network visualization, integrating a notepad file as a static database.

### Challenges Faced

**Limited Proficiency in Data Structures and Algorithms:**

The intricate nature of the original requirements presented challenges due to the existing academic exposure to data structures and algorithms.

**Temporal Constraints:**

The expansive scope of the initial assignment, combined with ongoing academic commitments, constrained the available time for thorough research and implementation.

**Knowledge Gap:**

The demanding prerequisites for an in-depth understanding of graph traversal, analysis techniques, and optimization strategies surpassed the current academic knowledge level.

### Adaptation

In response to the challenges, the modified application centers on the visualization of a social network, incorporating a notepad file as a static database for predefined relationships. This adjustment allows for a more attainable project that harnesses skills in GUI development.

### Features of the Modified Application:

**Graph Visualization:**

The application provides users with a graphical representation of the social network, where nodes symbolize users and edges denote connections.

**Interactivity:**

Users can interact with the graph, exploring connections and gaining insights into the network structure.

**User Influence Representation:**

The application visually represents user influence within the network, emphasizing relationships and interactions.

**Notepad as a Static Database:**

To depict relationships, the application utilizes a notepad file as a simple static database containing predefined user connections and interactions.

**Implementation Details:**

**Reading Data from Notepad:**

The application reads static data from the notepad file during initialization, populating the social network graph with predefined relationships.

**Graph Manipulation:**

Users can visualize and explore the social network graph, gaining insights into the relationships and interactions stored in the notepad file.

**User-Friendly Interface:**

The GUI provides an intuitive interface, allowing users to navigate the graph easily and understand the network structure.

### Suggestions for Future Improvements

**Dynamic Data Integration:**

Enhance the application to dynamically read and update data from external sources, providing real-time representation of the social network.

### Advanced Graph Algorithms:

As the academic understanding of graph algorithms improves, consider implementing more advanced algorithms for enhanced graph analysis.

**User Feedback Mechanism:**

Integrate a mechanism for users to provide feedback on the visualized social network, enhancing the user experience and providing valuable insights for future improvements.

### Known Bugs or Issues that may occur

**Graph Rendering Performance:**

Rendering performance may degrade with a large number of nodes and edges. Further optimization is needed for scalability.

### Conclusion

The modified social network visualization application, employing a notepad file as a static database, serves as a pragmatic response to the challenges posed by the initial assignment. This adaptation allows for a project that aligns with the current academic skill set while providing a valuable tool for understanding social network relationships. This academic endeavor underscores the need for continual learning to bridge gaps in knowledge and tackle increasingly complex projects successfully.

# Appendix

**GitHub Link:**

<https://github.com/nileshshrs/Final-Data-Structure---Algorithm-Assignment>