**Practical No 1 BigInteger**

**import java.math.BigInteger;**

import java.util.Scanner;

public class LargeIntegerSquare {

// Method to compute square using divide and conquer (Karatsuba's Algorithm)

**public static BigInteger karatsuba(BigInteger x, BigInte**ger y) {

//1 step equal number both;

**int n = Math.max(x.toString().length(), y.toString().length());**

// Base case when the numbers are small enough to multiply directly

//1.1 base condition

**if (n <= 2) {**

**return x.multiply(y);**

}

//2 half and pow

**// Split the number into two halves**

**int half = n / 2;**

**BigInteger power = BigInteger.TEN.pow(half);**

**BigInteger xh = x.divide(power); // High part of x**

**BigInteger xl = x.mod(power); // Low part of x**

**BigInteger yh = y.divide(power); // High part of y**

**BigInteger yl = y.mod(power); // Low part of y**

// Step 1: Compute high parts product

BigInteger s1 = karatsuba(xh, yh);

// Step 2: Compute low parts product

BigInteger s2 = karatsuba(xl, yl);

// Step 3: Compute cross parts product

BigInteger s3 = karatsuba(xh.add(xl), yh.add(yl));

// Step 4: Combine the results

**BigInteger result = s1.multiply(power.pow(2)).add(s3.subtract(s1).subtract(s2).multiply(power)).add(s2);**

return result;

}

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

// Create a scanner for user input

Scanner scanner = new Scanner(System.in);

// Ask the user for a large number

System.out.print("Enter a large integer: ");

String input = scanner.nextLine();

**// Convert input string to BigInteger**

**BigInteger x = new BigInteger(input);**

// Compute the square using Karatsuba's Algorithm

BigInteger result = karatsuba(x, x);

// Output the result

System.out.println("The square of the number is: " + result);

// Close the scanner

scanner.close();

**}**

}

**Practical No 2 Job scheduling**

**Consider the scheduling problem. n tasks to be scheduled on a single processor. Let d1, ....dn be deadline and pl,....pn be the profit of each task to execute on a single processor is known. The tasks can be executed in any order but one task at a time and each task take 1 unit of time to execute. Design a greedy algorithm for this problem and find a schedule or sequence of jobs that gives maximum profit.**

import java.io.\*;

import java.util.\*;

class Job {

int id, profit, deadline;

// Constructor to initialize the job

Job(int id, int deadline, int profit) {

this.id = id;

this.deadline = deadline;

this.profit = profit;

}

}

class JobSchedulingSolver {

// Method to schedule jobs to maximize profit and return the number of jobs scheduled and total profit

public int[] jobScheduling(Job arr[], int n) {

// Sort jobs by descending order of profit

**Arrays.sort(arr, (a, b) -> b.profit - a.profit);**

// Find the maximum deadline to determine the number of time slots

int maxDeadline = 0;

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

**if (arr[i].deadline > maxDeadline) {**

**maxDeadline = arr[i].deadline;**

**}**

**}**

// Array to store result (time slots) initialized to -1 (empty)

**int[] result = new int[maxDeadline + 1];**

**Arrays.fill(result, -1); //or for loop also**

// Variables to store the number of jobs and total profit

int countJobs = 0, totalProfit = 0;

// Iterate over each job

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

**// Find a free slot for this job by checking slots from the last possible deadline**

**for (int j = arr[i].deadline; j > 0; j--) {**

**if (result[j] == -1) { // Slot is free**

**result[j] = i; // Assign job to this slot**

**countJobs++; // Increment job count**

**totalProfit += arr[i].profit; // Add profit**

**break; // Move to the next job**

**}**

**}**

**}**

// Return the number of jobs and total profit

**return new int[]{countJobs, totalProfit};**

}

}

public class Main {

public static void main(String[] args) throws IOException {

Scanner sc = new Scanner(System.in);

// Get the number of jobs from the user

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

int n = sc.nextInt();

// Create array of jobs

**Job jobs[] = new Job[n];**

// Input job details from the user

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

System.out.println("Enter job ID (numeric), deadline, and profit for job " + (i + 1) + ":");

**int id = sc.nextInt();**

**int deadline = sc.nextInt();**

**int profit = sc.nextInt();**

jobs[i] = new Job(id, deadline, profit);

}

// Create an object of the solver class

JobSchedulingSolver solver = new JobSchedulingSolver();

// Call the jobScheduling method and get the result

int[] result = solver.jobScheduling(jobs, n);

// Output the result

**System.out.println("Number of jobs scheduled: " + result[0]);**

**System.out.println("Total profit: " + result[1]);**

sc.close();

}

}

Enter the number of jobs:

4

Enter job ID (numeric), deadline, and profit for job 1:

3 2 40

Enter job ID (numeric), deadline, and profit for job 2:

4 2 30

Enter job ID (numeric), deadline, and profit for job 3:

1 4 20

Enter job ID (numeric), deadline, and profit for job 4:

2 1 10

Number of jobs scheduled: 3

Total profit: 90

**Practical No 3 Floyd-Warshall algorithm.**

You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by Floyd-Warshall algorithm.

import java.util.\*;

public class MinimumPhoneLineCost {

private static final int INF = Integer.MAX\_VALUE / 2; // To avoid overflow

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Take input for the number of cities (offices)

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

int numCities = scanner.nextInt();

// Create a cost matrix for the graph

int[][] costMatrix = new int[numCities][numCities];

// Take input for the cost matrix

System.out.println("Enter the cost matrix (Enter 0 for no cost, INF for unreachable offices):");

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

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

if (i == j) {

costMatrix[i][j] = 0; // No cost to connect an office to itself

} else {

System.out.print("Cost to connect office " + (i + 1) + " to office " + (j + 1) + ": ");

int cost = scanner.nextInt();

costMatrix[i][j] = (cost == -1) ? INF : cost; // Use INF if the cost is -1 (no connection)

}

}

}

floydWarshall(costMatrix);

scanner.close();

}

private static void floydWarshall(int[][] graph) {

int numCities = graph.length;

int[][] dist = new int[numCities][numCities];

// Initialize the distance matrix with the input graph

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

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

dist[i][j] = graph[i][j];

}

}

// Floyd-Warshall algorithm to find shortest paths

for (int k = 0; k < numCities; k++) {

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

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

if (dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

// Print the shortest distance matrix

System.out.println("The shortest distances between every pair of cities:");

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

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

if (dist[i][j] == INF) {

System.out.print("INF ");

} else {

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

}

}

System.out.println();

}

// Calculate the minimum spanning tree cost

int mstCost = calculateMSTCost(dist);

System.out.println("The minimum cost to connect all offices: " + mstCost);

}

private static int calculateMSTCost(int[][] dist) {

int numCities = dist.length;

boolean[] visited = new boolean[numCities];

int[] minEdge = new int[numCities];

Arrays.fill(minEdge, INF);

minEdge[0] = 0;

int totalCost = 0;

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

int u = -1;

// Find the unvisited city with the smallest edge

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

if (!visited[j] && (u == -1 || minEdge[j] < minEdge[u])) {

u = j;

}

}

// If no such city exists, we've processed all cities

if (minEdge[u] == INF) {

System.out.println("The graph is not connected.");

return -1;

}

visited[u] = true;

totalCost += minEdge[u];

// Update the minimum edges for neighboring cities

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

if (!visited[v] && dist[u][v] < minEdge[v]) {

minEdge[v] = dist[u][v];

}

}

}

return totalCost;

}

}

Enter the number of offices (cities): 3

Enter the cost matrix (Enter 0 for no cost, INF for unreachable offices):

Cost to connect office 1 to office 2: 1

Cost to connect office 1 to office 3: 3

Cost to connect office 2 to office 1: 1

Cost to connect office 2 to office 3: 3

Cost to connect office 3 to office 1: 4

Cost to connect office 3 to office 2: 2

The shortest distances between every pair of cities:

0 1 3

1 0 3

3 2 0

The minimum cost to connect all offices: 4

**Practical No 4 Dijkstra's algorithm.**

You have been given a network of 'N' nodes from 1 to 'N' and 'M' edges. For each edge, you are given three values (ui, vi, wi) where "ui" and "vi" denote the nodes and "wi" denotes an integer value which represents the time taken by a signal to travel from "ui" to "vi". Now, you are supposed to find the time which a signal takes to travel from a given node 'K' to all nodes. If it is impossible for all nodes to receive the signal then print -1. Implement the given Network Delay Time using Dijkstra's algorithm.

import java.util.\*;

class Graph {

private int V; // Number of vertices

private List<List<Node>> adj; // Adjacency list to store the graph

// Constructor to initialize the graph

Graph(int V) {

this.V = V;

adj = new ArrayList<>();

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

adj.add(new ArrayList<>()); // Create an empty list for each vertex

}

}

// Method to add an edge between two vertices with a weight

void addEdge(int u, int v, int w) {

adj.get(u).add(new Node(v, w)); // Add edge u -> v with weight w

adj.get(v).add(new Node(u, w)); // For undirected graph, add edge v -> u

}

// Dijkstra's algorithm to find the shortest path from a source vertex

void shortestPath(int src) {

PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node -> node.weight));

int[] dist = new int[V]; // Distance array to store shortest distances

Arrays.fill(dist, Integer.MAX\_VALUE); // Initialize all distances to infinity

pq.add(new Node(src, 0)); // Add the source vertex with distance 0

dist[src] = 0; // Distance to the source is always 0

while (!pq.isEmpty()) {

Node current = pq.poll(); // Get the node with the smallest distance

int u = current.vertex;

// Process all neighbors of the current node

for (Node neighbor : adj.get(u)) {

int v = neighbor.vertex; // Neighboring vertex

int weight = neighbor.weight; // Edge weight

// Relaxation: Update the distance if a shorter path is found

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight; // Update the shortest distance

pq.add(new Node(v, dist[v])); // Add the neighbor to the priority queue

}

}

}

// Print the shortest distances from the source

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

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

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

}

}

// Helper class to represent a node in the graph

static class Node {

int vertex, weight; // Vertex number and edge weight

Node(int vertex, int weight) {

this.vertex = vertex;

this.weight = weight;

}

}

}

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Take user input for the number of vertices

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

int V = scanner.nextInt();

// Create the graph with V vertices

Graph graph = new Graph(V);

// Take user input for the number of edges

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

int E = scanner.nextInt();

// Take user input for each edge (u, v, weight)

System.out.println("Enter each edge in the format: u v weight");

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

int u = scanner.nextInt();

int v = scanner.nextInt();

int weight = scanner.nextInt();

graph.addEdge(u, v, weight); // Add the edge to the graph

}

// Take user input for the source vertex for Dijkstra's algorithm

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

int src = scanner.nextInt();

// Find and print shortest paths from the source vertex

graph.shortestPath(src);

scanner.close(); // Close the scanner

}

}

Enter the number of vertices: 5

Enter the number of edges: 6

Enter each edge in the format: u v weight

0 1 2

0 2 4

1 2 1

1 3 7

2 4 3

3 4 1

Enter the source vertex: 0

Vertex Distance from Source

0 0

1 2

2 3

3 7

4 6

**Pratical No 5 Knight's tour Problem.**

**A classic problem that can be solved by backtracking is called the Knight's tour Problem. It is a problem in which we are provided with a NxN chessboard and a knight. For a person who is not familiar with chess, the knight moves two squares horizontally and one square vertically, or two squares vertically and one square horizontally. In this problem, there is an empty chess board, and a knight starting from any location in the board, our task is to check whether the knight can visit all of the squares in the board or not. When It can visit all of the squares, then place the number of jumps needed to reach that location from the starting point.**

import java.util.\*;

class Main {

static int N = 8; // Size of the chessboard (8x8)

// Possible moves of a knight

static int[] dx = {-2, -1, 1, 2, 2, 1, -1, -2};

static int[] dy = {1, 2, 2, 1, -1, -2, -2, -1};

// Method to check if a position is valid

static boolean isSafe(int x, int y, int[][] board) {

return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);

}

// Method to print the board

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

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

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

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

}

System.out.println();

}

}

// Backtracking method to solve the Knight's Tour problem

static boolean solveKT(int x, int y, int moveCount, int[][] board) {

// Base case: If all squares are visited

if (moveCount == N \* N) {

return true;

}

// Try all next moves for the knight

for (int i = 0; i < 8; i++) {

int newX = x + dx[i];

int newY = y + dy[i];

// If the move is safe, make the move

if (isSafe(newX, newY, board)) {

board[newX][newY] = moveCount;

// Recur to make the next move

if (solveKT(newX, newY, moveCount + 1, board)) {

return true;

}

// Backtrack: undo the move if it leads to no solution

board[newX][newY] = -1;

}

}

return false; // Return false if no move is possible

}

// Method to initiate the solution

static void knightTour(int startX, int startY) {

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

// Initialize the board with -1 (indicating unvisited squares)

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

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

board[i][j] = -1;

}

}

// Set the starting position

board[startX][startY] = 0;

// Start solving the Knight's Tour problem

if (solveKT(startX, startY, 1, board)) {

printBoard(board); // Print the board if the tour is successful

} else {

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

}

}

public static void main(String[] args) {

int startX = 0, startY = 0; // Starting position of the knight

knightTour(startX, startY); // Try to solve the Knight's Tour

}

}

**Practical No 6 Assignment Problem**

**Let there be N students and N clubs. Any student can be assigned to any club, incurring some cost that may vary depending on the student club assignment. It is required to allocate all clubs by assigning exactly one student to each club and exactly one club to each agent in such a way that the total cost of the assignment is minimized. Implement**

**club assignment problem using Branch and bound**

import java.util.\*;

public class Main {

// Function to assign clubs with the minimum cost using backtracking

public static int assignClubs(int[][] cost) {

int N = cost.length;

int[] assignedClubs = new int[N];

Arrays.fill(assignedClubs, -1);

int[] minCost = {Integer.MAX\_VALUE};

int[] bestAssignment = new int[N];

solve(cost, 0, assignedClubs, 0, minCost, bestAssignment);

System.out.println("Optimal assignment of students to clubs:");

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

System.out.println("Student " + (i + 1) + " -> Club " + (bestAssignment[i] + 1));

}

return minCost[0];

}

// Backtracking function to explore all possible assignments

public static void solve(int[][] cost, int studentIndex, int[] assignedClubs, int currentCost, int[] minCost, int[] bestAssignment) {

int N = cost.length;

if (studentIndex == N) {

if (currentCost < minCost[0]) {

minCost[0] = currentCost;

System.arraycopy(assignedClubs, 0, bestAssignment, 0, N);

}

return;

}

// Try all possible clubs for the current student

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

// Skip if the club is already assigned

if (isClubAssigned(assignedClubs, clubIndex)) {

continue;

}

assignedClubs[studentIndex] = clubIndex;

solve(cost, studentIndex + 1, assignedClubs, currentCost + cost[studentIndex][clubIndex], minCost, bestAssignment);

assignedClubs[studentIndex] = -1; // Backtrack

}

}

// Helper function to check if a club is already assigned

public static boolean isClubAssigned(int[] assignedClubs, int clubIndex) {

for (int i : assignedClubs) {

if (i == clubIndex) {

return true;

}

}

return false;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Taking input for the number of students and clubs (same number)

System.out.print("Enter the number of students (and clubs): ");

int N = scanner.nextInt();

// Initializing the cost matrix

int[][] cost = new int[N][N];

// Taking input for the cost of assigning each student to each club

System.out.println("Enter the cost matrix (rows represent students and columns represent clubs):");

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

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

System.out.print("Cost of assigning Student " + (i + 1) + " to Club " + (j + 1) + ": ");

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

}

}

// Calculate the minimum cost and display the optimal assignment

int minCost = assignClubs(cost);

System.out.println("The minimum cost of assigning students to clubs is: " + minCost);

scanner.close();

}

}

Enter the number of students (and clubs): 2

Enter the cost matrix (rows represent students and columns represent clubs):

Cost of assigning Student 1 to Club 1: 1

Cost of assigning Student 1 to Club 2: 4

Cost of assigning Student 2 to Club 1: 42

Cost of assigning Student 2 to Club 2: 4

Optimal assignment of students to clubs:

Student 1 -> Club 1

Student 2 -> Club 2

The minimum cost of assigning students to clubs is: 5