import java.util.Comparator;

import java.util.HashSet;

import java.util.InputMismatchException;

import java.util.PriorityQueue;

import java.util.Scanner;

import java.util.Set;

public class DijkstraPriorityQueue

{

private int distances[];

private Set<Integer> settled;

private PriorityQueue<Node> priorityQueue;

private int number\_of\_nodes;

private int adjacencyMatrix[][];

public DijkstraPriorityQueue(int number\_of\_nodes)

{

this.number\_of\_nodes = number\_of\_nodes;

distances = new int[number\_of\_nodes + 1];

settled = new HashSet<Integer>();

priorityQueue = new PriorityQueue<Node>(number\_of\_nodes,new Node());

adjacencyMatrix = new int[number\_of\_nodes + 1][number\_of\_nodes + 1];

}

public void dijkstra\_algorithm(int adjacency\_matrix[][], int source)

{

int evaluationNode;

for (int i = 1; i <= number\_of\_nodes; i++)

for (int j = 1; j <= number\_of\_nodes; j++)

adjacencyMatrix[i][j] = adjacency\_matrix[i][j];

for (int i = 1; i <= number\_of\_nodes; i++)

{

distances[i] = Integer.MAX\_VALUE;

}

priorityQueue.add(new Node(source, 0));

distances[source] = 0;

while (!priorityQueue.isEmpty())

{

evaluationNode = getNodeWithMinimumDistanceFromPriorityQueue();

settled.add(evaluationNode);

evaluateNeighbours(evaluationNode);

}

}

private int getNodeWithMinimumDistanceFromPriorityQueue()

{

Node node = priorityQueue.remove();

return node.node;

}

private void evaluateNeighbours(int evaluationNode)

{

int edgeDistance = -1;

int newDistance = -1;

for (int destinationNode = 1; destinationNode <= number\_of\_nodes; destinationNode++)

{

if (!settled.contains(destinationNode))

{

if (adjacencyMatrix[evaluationNode][destinationNode] != Integer.MAX\_VALUE)

{

edgeDistance = adjacencyMatrix[evaluationNode][destinationNode];

newDistance = distances[evaluationNode] + edgeDistance;

if (newDistance < distances[destinationNode])

{

distances[destinationNode] = newDistance;

}

priorityQueue.add(new Node(destinationNode,distances[destinationNode]));

}

}

}

}

public static void main(String... arg)

{

int adjacency\_matrix[][];

int number\_of\_vertices;

int source = 0;

Scanner scan = new Scanner(System.in);

try

{

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

number\_of\_vertices = scan.nextInt();

adjacency\_matrix = new int[number\_of\_vertices + 1][number\_of\_vertices + 1];

System.out.println("Enter the Weighted Matrix for the graph");

for (int i = 1; i <= number\_of\_vertices; i++)

{

for (int j = 1; j <= number\_of\_vertices; j++)

{

adjacency\_matrix[i][j] = scan.nextInt();

if (i == j)

{

adjacency\_matrix[i][j] = 0;

continue;

}

if (adjacency\_matrix[i][j] == 0)

{

adjacency\_matrix[i][j] = Integer.MAX\_VALUE;

}

}

}

System.out.println("Enter the source ");

source = scan.nextInt();

DijkstraPriorityQueue dijkstrasPriorityQueue = new DijkstraPriorityQueue(number\_of\_vertices);

dijkstrasPriorityQueue.dijkstra\_algorithm(adjacency\_matrix, source);

System.out.println("The Shorted Path to all nodes are ");

for (int i = 1; i <= dijkstrasPriorityQueue.distances.length - 1; i++)

{

System.out.println(source + " to " + i + " is " + dijkstrasPriorityQueue.distances[i]);

}

} catch (InputMismatchException inputMismatch)

{

System.out.println("Wrong Input Format");

}

scan.close();

}

}

class Node implements Comparator<Node>

{

public int node;

public int cost;

public Node()

{

}

public Node(int node, int cost)

{

this.node = node;

this.cost = cost;

}

@Override

public int compare(Node node1, Node node2)

{

if (node1.cost < node2.cost)

return -1;

if (node1.cost > node2.cost)

return 1;

return 0;

}

}

Output:

Enter the number of vertices

5

Enter the Weighted Matrix for the graph

0 9 6 5 3

0 0 0 0 0

0 2 0 4 0

0 0 0 0 0

0 0 0 0 0

Enter the source

1

The Shorted Path to all nodes are

1 to 1 is 0

1 to 2 is 8

1 to 3 is 6

1 to 4 is 5

1 to 5 is 3