```java
/*
 * Ada Trabalho 2 - Legionellosis
 *
 * @author Joana Soares Faria n55754
 * @author Goncalo Martins Lourenco n55780
 */


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

public class Main {

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

        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        String[] inputLine = input.readLine().split(" ");
        int numLocations = Integer.parseInt(inputLine[0]);
        int numConnections = Integer.parseInt(inputLine[1]);

        Legionellosis problem = new Legionellosis(numLocations);

        for (int i = 0; i < numConnections; i++) {
            String[] connection = input.readLine().split(" ");
            int l1 = Integer.parseInt(connection[0]);
            int l2 = Integer.parseInt(connection[1]);
            problem.addConnection(l1, l2);
        }

        int numSick = Integer.parseInt(input.readLine());
        for (int i = 0; i < numSick; i++) {
            String[] sick = input.readLine().split(" ");
            int home = Integer.parseInt(sick[0]);
            int distance = Integer.parseInt(sick[1]);
            problem.addSick(home, distance, numSick);
        }

        List<Integer> perilousLoc = problem.perilousLocations();


        if (perilousLoc.size() == 0) {
            System.out.println(0);
        } else {
            int initial = perilousLoc.remove(0);
            System.out.printf("%d", initial);
            for (int a : perilousLoc) {
                System.out.printf(" %d", a);
            }
            System.out.println();
        }
    }
}

```

```java
/*
 * Ada Trabalho 2 - Legionellosis
 *
 * @author Joana Soares Faria n55754
 * @author Goncalo Martins Lourenco n55780
 */

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class Legionellosis {

    /**
     * number of location in the problem
     */
    private final int numLocations;
    /**
     * number of sick person that passed in which location
     */
    private final int[] locationsCheck;
    /**
     * list of all perilous locations identified
     */
    private final List<Integer> perilousLocations;
    /**
     * connections between locations - adjacency's linked list of successors in the graph
     */
    private List<Integer>[] connections;


    public Legionellosis(int numLocations) {
        this.numLocations = numLocations;
        locationsCheck = new int[numLocations + 1];
        initConnections();
        perilousLocations = new LinkedList<>();
    }

    /**
     * Initiates the array of connections in the graph (adjacency's linked list of successors)
     */
    @SuppressWarnings("unchecked")
    private void initConnections() {
        connections = new List[numLocations + 1];
        for (int i = 0; i <= numLocations; i++) {
            connections[i] = new LinkedList<>();
        }
    }

    /**
     * Adds a connection between to locations
     *
     * @param l1 first location
     * @param l2 second location
     */
    public void addConnection(int l1, int l2) {
        connections[l1].add(l2);
        connections[l2].add(l1);
    }

    /**
     * Computes the locations where a sick person might have been, that information is recorded
     * in the locationsCheck array
     *
```

```java
     * @param home     location of the of the home of the sick person
     * @param distance maximum distance from home the patient has been on the days before the
     *                 first symptoms
     * @param numSick  total number of sick people
     */
    public void addSick(int home, int distance, int numSick) {
        boolean[] found = new boolean[numLocations + 1];
        //explore lever by level (each level represents a distance)
        //current level being explored
        Queue<Integer> currentLevel = new LinkedList<>();
        //next level to explore
        Queue<Integer> nextLevel = new LinkedList<>();
        //current level
        int level = 0;

        //start the exploration from the sick person's home
        currentLevel.add(home);

        int loc;

        //the level cannot exceed the given distance
        while (!currentLevel.isEmpty() && level <= distance) {
            while (!currentLevel.isEmpty()) {

                loc = currentLevel.remove();
                found[loc] = true;

                //increases the number of sick persons that passed by the location loc
                locationsCheck[loc]++;
                //check if all sick persons passed by the location
                if (locationsCheck[loc] == numSick) {
                    //if all the sick persons passed by the location then the location is perilous
                    perilousLocations.add(loc);
                }

                //add the node descendants to the next level to explore
                for (int l : connections[loc]) {
                    if (!found[l]) {
                        nextLevel.add(l);
                        found[l] = true;
                    }
                }
            }

            //level finished, go to next level
            level++;
            Queue<Integer> temp = currentLevel;
            currentLevel = nextLevel;
            nextLevel = temp;
        }
    }

    /**
     * Returns the list with all the perilous locations identified, ordered
     *
     * @return an ordered list with all the identified perilous locations
     */
    public List<Integer> perilousLocations() {
        Collections.sort(perilousLocations);
        return perilousLocations;
    }
}

```