Lab 03: Resolve the Connectivity Problem

Note: ***Students must demonstrate their lab in the following week of October 19th, 2020 to get the grade. This lab is worth 20marks. A Missed demonstration will result in 25% grade deduction. Late submission and late demonstration to the lab is welcomed but with a loss of marks. That is, you do not earn full marks for late submission. 10% penalty for every day late up to 50%. Work will not be accepted after 10 days.***
You must talk to your lab professor in case of any unprecedented situation which may result in late submission.
You are required to submit the screenshot of your terminal showing successful execution of the program (including relevant display messages or output statements) along with properly commented (as per assignment submission standards doc on BrightSpace) text file of your code.

Setup:

Create a directory **`Lastname03`**. You are going to develop your lab here

Connectivity Problem

Suppose that we are given a sequence of pairs of integers, where each integer represents an object of some type and we are to interpret the pair **`p-q`** as meaning **`p is connected to q.`** We assume the relation **`is connected to`** to be transitive: If **`p`** is connected to **`q`**, and **`q`** is connected to **`r`**, then **`p`** is connected to **`r`**.

The problem is to devise a program that can remember sufficient information about the pairs it has seen to be able to decide whether a new pair of objects is connected or not. Informally, we refer to the task of designing such a method as the **connectivity problem**. This problem arises in several important applications.

For example, the integers might represent computers in a large network, and the pairs might represent connections in the network. Then, the program might be used to determine whether we need to establish a new direct connection for **`p`** and **`q`** to be able to communicate, or whether we could use **existing** connections to set up a communications path. In this kind of application, we might need to process millions of points and billions of connections, or more.

Similarly, the integers might represent contact points in an electrical network, and the pairs might represent wires connecting the points. In this case, we could use the program to find a way to connect all the points without any extraneous connections, if that is possible.

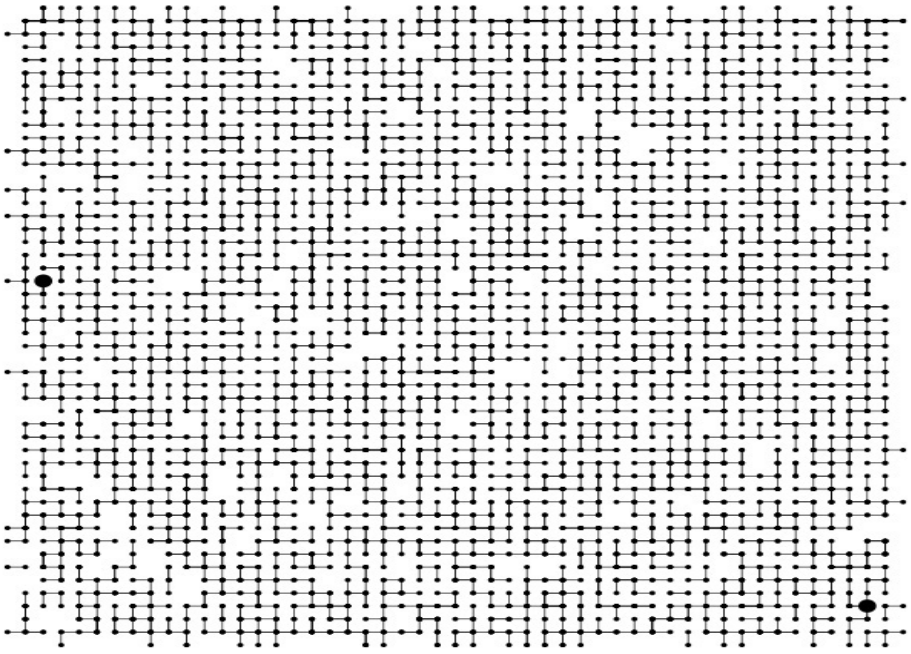

Figure: The objects in a connectivity problem might represent connection points, and the pairs might be connections between them, as indicated in this idealized example that might represent wires connecting buildings in a city or components on a computer chip. This graphical representation makes it possible for a human to spot nodes that are not connected, but the algorithm must work with only the pairs of integers that it is given. Are the two nodes marked with the large black dots connected?

The goal is to write a program to filter out extraneous pairs from the set: When the user inputs a pair **`p-q`**, it should output the pair only if the pairs it has seen to that point do not imply that **`p`** is connected to **`q`**. If the previous pairs do imply that **`p`** is connected to **`q`**, then the program should ignore **`p-q`** and should proceed to input the next pair.

*Write a quick-find solution to the connectivity problem. Your program should read a sequence of pairs of nonnegative integers less than **`N`** from standard input (interpreting the pair **`p q`** to mean "**`connect object p to object q`**") and prints out pairs representing objects that are not yet connected. The program maintains an array **`id`** of size **`N`** that has an entry for each **`object`**, with the property that **`id[p]`** and **`id[q]`** are equal if and only if p and q are connected.*

The basis of the algorithm is an array of integers with the property that **p** and **q** are connected if and only if the **p**th and **q**th array entries are equal. We initialize the **i**th array entry to **i** for **0 ≤ i < N**. To implement the **connection** for **p** and **q**, we go through the array, changing all the entries with the same name as **p** to have the same name as **q**.

p	q	0	1	2	3	4	5	6	7	8	9
3	4	0	1	2	**4**	4	5	6	7	8	9
4	9	0	1	2	**9**	**9**	5	6	7	8	9
8	0	0	1	2	9	9	5	6	7	**0**	9
2	3	0	1	**9**	9	9	5	6	7	0	9
5	6	0	1	9	9	9	**6**	6	7	0	9
2	9	0	1	9	9	9	6	6	7	0	9
6	9	0	1	9	9	9	**9**	**9**	7	0	9
7	3	0	1	9	9	9	9	9	**9**	0	9
4	8	0	1	**0**	**0**	**0**	**0**	**0**	**0**	0	0
5	6	0	1	0	0	0	0	0	0	0	0
6	1	**1**	1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

This sequence depicts the contents of the id array after each of the pairs at left is processed by the program. Shaded entries are those that change. When we process the pair **p q**, we change all entries with the value **id[p]** to have the value **id[q]**.

Program:

Write a small C program **connect.c** that:

1. Initializes an array **id** of **N** elements with the value of the index of the array.
2. Reads from the keyboard or the command line a set of two integer numbers (p and **q**) until it encounters **EOF** or **CTL-D**
3. Given the two numbers, your program should **connect** them by going through the array and changing all the entries with the same name as **p** to have the same name as **q**.
4. Once the EOF has been reached, your program should print the array with a max of 10 positions per line.
5. For testing purposes, define **N** as **10** at the beginning, but by sure of running data with at least 100 elements.