04/08/2010

Topic : Data Structure – Graphs

572 – Oil Deposits

Oil Deposits

The GeoSurvComp geologic survey company is responsible for detecting underground oil deposits. GeoSurvComp works with one large rectangular region of land at a time, and creates a grid that divides the land into numerous square plots. It then analyzes each plot separately, using sensing equipment to determine whether or not the plot contains oil.

A plot containing oil is called a pocket. If two pockets are adjacent, then they are part of the same oil deposit. Oil deposits can be quite large and may contain numerous pockets. Your job is to determine how many different oil deposits are contained in a grid.

Input

The input file contains one or more grids. Each grid begins with a line containing *m* and *n*, the number of rows and columns in the grid, separated by a single space. If *m* = 0 it signals the end of the input; otherwise 1<= m <= 100 and 1<= n <= 100. Following this are *m* lines of *n* characters each (not counting the end-of-line characters). Each character corresponds to one plot, and is either `*', representing the absence of oil, or `@', representing an oil pocket.

Output

For each grid, output the number of distinct oil deposits. Two different pockets are part of the same oil deposit if they are adjacent horizontally, vertically, or diagonally. An oil deposit will not contain more than 100 pockets.

Sample Input

```
1 1
*
3 5
*@*@*
**@**
*@*@*
1 8
@@****@*
5 5
****@
*@@*@
*@**@
@@@*@
@@**@
0 0
```

Sample Output

```
0
1
2
2
```

---

439 - Knight Moves

Knight Moves

A friend of you is doing research on the *Traveling Knight Problem (TKP)* where you are to find the shortest closed tour of knight moves that visits each square of a given set of *n* squares on a chessboard exactly once. He thinks that the most difficult part of the problem is determining the smallest number of knight moves between two given squares and that, once you have accomplished this, finding the tour would be easy.

Of course you know that it is vice versa. So you offer him to write a program that solves the "difficult" part.

Your job is to write a program that takes two squares *a* and *b* as input and then determines the number of knight moves on a shortest route from *a* to *b*.

Input Specification

The input file will contain one or more test cases. Each test case consists of one line containing two squares separated by one space. A square is a string consisting of a letter (`a-h`) representing the column and a digit (`1-8`) representing the row on the chessboard.

Output Specification

For each test case, print one line saying "`To get from xx to yy takes n knight moves.`".

Sample Input

```
e2 e4
a1 b2
b2 c3
a1 h8
a1 h7
h8 a1
b1 c3
f6 f6
```

Sample Output

```
To get from e2 to e4 takes 2 knight moves.
To get from a1 to b2 takes 4 knight moves.
To get from b2 to c3 takes 2 knight moves.
To get from a1 to h8 takes 6 knight moves.
To get from a1 to h7 takes 5 knight moves.
To get from h8 to a1 takes 6 knight moves.
To get from b1 to c3 takes 1 knight moves.
To get from f6 to f6 takes 0 knight moves.
```

10305 - Ordering Tasks

Ordering Tasks

Input: standard input Output: standard output
Time Limit: 1 second Memory Limit: 32 MB

John has n tasks to do. Unfortunately, the tasks are not independent and the execution of one task is only possible if other tasks have already been executed.

Input

The input will consist of several instances of the problem. Each instance begins with a line containing two integers, **1 <= n <= 100** and **m**. **n** is the number of tasks (numbered from **1** to **n**) and **m** is the number of direct precedence relations between tasks. After this, there will be **m** lines with two integers **i** and **j**, representing the fact that task **i** must be executed before task **j**. An instance with **n = m = 0** will finish the input.

Output

For each instance, print a line with n integers representing the tasks in a possible order of execution.

Sample Input

```
5 4
1 2
2 3
1 3
1 5
```

0 0

Sample Output

```
1 4 2 5 3
```

10004 - Bicoloring

Bicoloring

In 1976 the ``Four Color Map Theorem'' was proven with the assistance of a computer. This theorem states that every map can be colored using only four colors, in such a way that no region is colored using the same color as a neighbor region.

Here you are asked to solve a simpler similar problem. You have to decide whether a given arbitrary connected graph can be bicolored. That is, if one can assign colors (from a palette of two) to the nodes in such a way that no two adjacent nodes have the same color. To simplify the problem you can assume:

- no node will have an edge to itself.
- the graph is nondirected. That is, if a node *a* is said to be connected to a node *b*, then you must assume that *b* is connected to *a*.
- the graph will be strongly connected. That is, there will be at least one path from any node to any other node.

Input

The input consists of several test cases. Each test case starts with a line containing the number n ($1 < n < 200$) of different nodes. The second line contains the number of edges l. After this, l lines will follow, each containing two numbers that specify an edge between the two nodes that they represent. A node in the graph will be labeled using a number a (0<=a<n).

An input with $n = 0$ will mark the end of the input and is not to be processed.

Output

You have to decide whether the input graph can be bicolored or not, and print it as shown below.

Sample Input

```
3
3
0 1
1 2
2 0
9
8
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0
```

Sample Output

```
NOT BICOLORABLE.
BICOLORABLE.
```