ADA - Análisis y Diseño de Algoritmos, 2025-1
Tarea 4: Semanas 10 y 11

Para entregar el domingo 6 de abril de 2024

Problemas conceptuales a las 23:59 por BrightSpace

Problemas prácticos a las 23:59 en la arena de programación

Tanto los ejercicios como los problemas deben ser resueltos, pero únicamente las soluciones de los problemas deben ser entregadas. La intención de los ejercicios es entrenarlo para que domine el material del curso; a pesar de que no debe entregar soluciones a los ejercicios, usted es responsable del material cubierto en ellos.

Instrucciones para la entrega

Para esta tarea y todas las tareas futuras, la entrega de soluciones es *individual*. Por favor escriba claramente su nombre, código de estudiante y sección en cada hoja impresa entregada o en cada archivo de código (a modo de comentario). Adicionalmente, agregue la información de fecha y nombres de compañeros con los que colaboró; igualmente cite cualquier fuente de información que utilizó.

¿Cómo describir un algoritmo?

En algunos ejercicios y problemas se pide "dar un algoritmo" para resolver un problema. Una solución debe tomar la forma de un pequeño ensayo (es decir, un par de párrafos). En particular, una solución debe resumir en un párrafo el problema y cuáles son los resultados de la solución. Además, se deben incluir párrafos con la siguiente información:

- una descripción del algoritmo en castellano y, si es útil, pseudo-código;
- por lo menos un diagrama o ejemplo que muestre cómo funciona el algoritmo;
- una demostración de la corrección del algoritmo; y
- un análisis de la complejidad temporal del algoritmo.

Recuerde que su objetivo es comunicar claramente un algoritmo. Las soluciones algorítmicas correctas y descritas *claramente* recibirán alta calificación; soluciones complejas, obtusas o mal presentadas recibirán baja calificación.

Problemas conceptuales

1. Ejercicio 2.5 (a,b): *Super and subsequences* (Erickson, página 95).

Problemas prácticos

Hay cinco problemas prácticos cuyos enunciados aparecen a partir de la siguiente página.

A - The Settlers of Catan

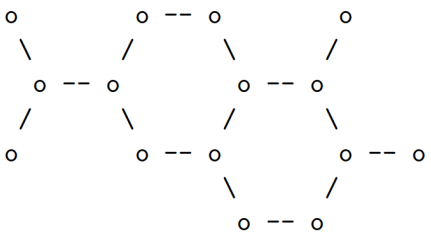
Source file name: `catan.py`
Time limit: 1 second

Within *Settlers of Catan*, the 1995 German game of the year, players attempt to dominate an island by building roads, settlements and cities across its uncharted wilderness. You are employed by a software company that just has decided to develop a computer version of this game, and you are chosen to implement one of the game's special rules: When the game ends, the player who built the longest road gains two extra victory points.

The problem here is that the players usually build complex road networks and not just one linear path. Therefore, determining the longest road is not trivial (although human players usually see it immediately).

Compared to the original game, we will solve a simplified problem here: You are given a set of nodes (cities) and a set of edges (road segments) of length 1 connecting the nodes. The longest road is defined as the longest path within the network that doesn't use an edge twice. Nodes may be visited more than once, though.

For example, the following network contains a road of length 12:



Input

The input will contain one or more test cases. The first line of each test case contains two integers: the number of nodes n ($2 \leq n \leq 25$) and the number of edges m ($1 \leq m \leq 25$). The next m lines describe the m edges. Each edge is given by the numbers of the two nodes connected by it. Nodes are numbered from 0 to $n - 1$. Edges are undirected. Nodes have degrees of three or less. The network is not necessarily connected. Input will be terminated by two values of 0 for n and m.

The input must be read from standard input.

Output

For each test case, print the length of the longest road on a single line.

The output must be written to standard output.

Sample Input	**Sample Output**
3 2 0 1 1 2 15 16 0 2 1 2 2 3 3 4 3 5 4 6 5 7 6 8 7 8 7 9 8 10 9 11 10 12 11 12 10 13 12 14 0 0	2 12

B - Graph Coloring

Source file name: `coloring.py`
Time limit: 5 seconds

You are to write a program that tries to find an optimal coloring for a given graph. Colors are applied to the nodes of the graph and the only available colors are black and white. The coloring of the graph is called optimal if a maximum of nodes is black. The coloring is restricted by the rule that no two connected nodes may be black.

Input

The graph is given as a set of nodes denoted by numbers $1, \ldots n$, $n \leq 100$, and a set of undirected edges denoted by pairs of node numbers $(n_1, n_2), n1 \neq n2$. The input contains m graphs. The number m is given on the first line. The first line of each graph contains n and k, the number of nodes and the number of edges, respectively. The following k lines contain the edges given by a pair of node numbers, which are separated by a space.

The input must be read from standard input.

Output

The output should consists of $2m$ lines, two lines for each graph found in the input file. The first line of should contain the maximum number of nodes that can be colored black in the graph. The second line should contain one possible optimal coloring. It is given by the list of black nodes, separated by a blank. If there exists more than one optimal solution, the lexicographically smallest optimal solution must be printed

The output must be written to standard output.

Sample Input	**Sample Output**
1 6 8 1 2 1 3 2 4 2 5 3 4 3 6 4 6 5 6	3 1 4 5

C - The Dominoes Solitaire

Source file name: `dominoes.py`
Time limit: 1 second

A man used to play dominoes with some friends in a small town. Some families have left the town and at present the only residents in the town are the man and his wife. This man would like to continue playing dominoes, but his wife does not like plaing. He has invented a game to play alone. Two pieces are picked out and are put at the two extremes of a row with a number (n) of spaces. After that, other pieces (m) are picked out to fill the spaces. The number of pieces is greater than or equal to the number of spaces ($m \geq n$), and the number of pieces is less than or equal to 14 ($m \leq 14$). The spaces are filled by putting one piece in each space, accoding to the rules of dominoes: the number of adjacent dots on two different dominoes must coincide. Pieces with repeated values are placed in the same way as the other pieces, and not at right angels.

The problem consists in the design of a program which, given a number of spaces (n), a number of pieces (m), the two initial pieces (i_1, i_2) and (d_1, d_2), and the m pieces $(p_1, q_1), (p_2, q_2), \ldots, (p_m, q_m)$, decides if it is possible to fill the n spaces between the two initial pieces using the m pieces and with the rules of dominoes. For example, with $n = 3$, $m = 4$, initial pieces $(0, 1)$ and $(3, 4)$, and pieces $(2, 1), (5, 6), (2, 2)$ and $(3, 2)$, the answer is '`YES`', because there is a solution: $(0, 1), (1, 2), (2, 2), (2, 3), (3, 4)$. With $n = 2$, $m = 4$, pieces in the extremes $(0, 1)$ and $(3, 4)$, and $(1, 4), (4, 4), (3, 2)$ and $(5, 6)$, the answer is '`NO`'.

Input

The input will consist of a series of problems, with each problem described in a series of lines: in the first line the number of spaces (n) is indicated, in the second line the number of pieces (m) used to fill the spaces, in the next line the piece to be placed on the left, with the two values in the piece separated by a space and in the same way that the numbers appear in the row, in the following lines the piece to be placed on the right, with the two values in the piece separated by a space and in the same way that the numbers appear in the row, and the other m pieces appear in consecutive lines, one in each line, with the two values separated by a space. Different problems appear in the input successively without separation, and the input finishes when '`0`' appears as the number of spaces.

The input must be read from standard input.

Output

For each problem in the input a line is written, with '`YES`' if the problem has a solution, and '`NO`' if it has no solution.

The output must be written to standard output.

Sample Input	**Sample Output**
3 4 0 1 3 4 2 1 5 6 2 2 3 2 2 4 0 1 3 4 1 4 4 4 3 2 5 6 0	YES NO

D - Rooks

Source file name: `rooks.py`
Time limit: 2 seconds

You have unexpectedly become the owner of a large chessboard, having fifteen squares to each side. Because you do not know how to play chess on such a large board, you find an alternative way to make use of it.

In chess, a rook attacks all squares that are in the same row or column of the chessboard as it is. For the purposes of this problem, we define a rook as also attacking the square on which it is already standing.

Given a set of chessboard squares, how many rooks are needed to attack all of them?

Input

Input consists of a number of test cases. Each test case consists of fifteen lines each containing fifteen characters depicting the chess board. Each character is either a period (`.`) or a hash (`#`). Every chessboard square depicted by a hash must be attacked by a rook. After all the test cases, one more line of input appears. This line contains the word '`END`'.

The input must be read from standard input.

Output

Output consists of exactly one line for each test case. The line contains a single integer, the minimum number of rooks that must be placed on the chess board so that every square marked with a hash is attacked.

The output must be written to standard output.

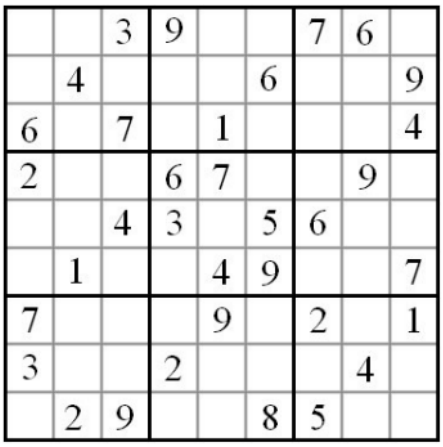
Sample Input	**Sample Output**

`...............`
`...............`
`...............`
`...............`
`...............`
`...............`
`...............`
`.......#.......`
`...............`
`...............`
`...............`
`...............`
`...............`
`...............`
`...............`
`END`
	`1`

E - So Doku Checker

Source file name: `sodoku.py`
Time limit: 5 seconds

The best logical puzzles often are puzzles that are based on a simple idea. So Doku is one such type of puzzle. Although So Dokus have been around for some twenty years, in the last few years they conquered the world exponentially. Hundreds of newspapers and websites are now publishing them on a daily basis. For those of you unfamiliar with these puzzles, let me give a brief introduction.



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

The picture above contains an example of a Su Doku puzzle. As you can see, we have a 9×9 grid filled with single digits from 1 to 9 and empty places. The grid is further divided into nine 3×3 sub-grids, indicated by the thick lines. To solve the puzzle you have to fill the empty places with digits according to the following rules:

- Every row should contain the digits 1 to 9 exactly once;
- Every column should contain the digits 1 to 9 exactly once;
- Every 3×3 sub-grid should contain the digits 1 to 9 exactly once.

A well formed Su Doku can be solved with paper and pencil using logical deduction only. To be well formed it should be legal (no row, column or sub-grid contains a digit more than once), solvable (the empty places can all be filled while respecting the rules) and unique (there is only one solution). This is what your program is going to check.

Input

The input contains several (partially) filled grids, each representing a Su Doku puzzle. For every puzzle there are 9 lines with 9 digits giving the puzzle in row major order. Empty places in the puzzle are represented by the digit '`0`' (zero). Digits on a line are separated by one space. The grids are separated by one empty line.

The first grid in the sample input represents the puzzle given in the picture.

The input must be read from standard input.

Output

For every grid in the input, determine one of the following four verdicts:

- '`Illegal`' if the puzzle violates one of the three rules;
- '`Unique`' if only one solution exists;
- '`Ambiguous`' if more than one solution exists;
- '`Impossible`' if no solution exists;

Print one line per grid, in the format: '`Case` *N*`:` *V*`.`', where *N* is the case number, starting from 1, and *C* is one of the four words in the list. See the sample output for the exact format.

Note: an '`Illegal`' puzzle is also '`Impossible`', of course, but your program should print '`Illegal`' in that case. Only print '`Impossible`' if the input doesn't violate one of the three rules, but the puzzle still can't be solved.

The output must be written to standard output.

Sample Input	**Sample Output**

`0 0 3 9 0 0 7 6 0`
`0 4 0 0 0 6 0 0 9`
`6 0 7 0 1 0 0 0 4`
`2 0 0 6 7 0 0 9 0`
`0 0 4 3 0 5 6 0 0`
`0 1 0 0 4 9 0 0 7`
`7 0 0 0 9 0 2 0 1`
`3 0 0 2 0 0 0 4 0`
`0 2 9 0 0 8 5 0 0`
 	
`0 0 3 9 0 0 7 6 0`
`0 4 0 0 0 6 0 0 9`
`6 0 0 0 1 0 0 0 4`
`0 0 0 6 7 0 0 9 0`
`0 0 4 0 0 5 6 0 0`
`0 1 0 0 4 9 0 0 0`
`7 0 0 0 9 0 2 0 1`
`3 0 0 2 0 0 0 4 0`
`0 2 0 0 0 8 5 0 0`
 	
`0 0 3 9 0 0 7 6 0`
`0 4 0 0 0 6 0 0 9`
`6 0 7 0 1 0 0 0 4`
`2 0 0 6 7 0 0 9 0`
`0 0 4 3 0 5 6 0 0`
`0 1 0 0 4 9 0 0 7`
`7 2 0 0 9 0 2 0 1`
`3 0 0 2 0 0 0 4 0`
`0 2 9 0 0 8 5 0 0`
 	
`0 0 3 9 0 0 7 6 0`
`0 4 0 0 0 6 0 0 9`
`6 0 7 0 1 0 0 0 4`
`2 0 0 6 7 0 0 9 0`
`0 0 4 3 0 5 6 0 0`
`0 1 0 0 4 9 0 0 7`
`7 5 0 0 9 0 2 0 1`
`3 0 0 2 0 0 0 4 0`
`0 2 9 0 0 8 5 0 0`
	`Case 1: Unique.`
	`Case 2: Ambiguous.`
	`Case 3: Illegal.`
	`Case 4: Impossible.`