AGRA - Árboles y Grafos, 2018-2
Tarea 3: Semanas 6 y 7

Para entregar el viernes 07 de septiembre/domingo 09 de septiembre de 2018

Problemas conceptuales a las 16:00 (07 de septiembre) en el Departamento de Electrónica y Ciencias de la Computación

Problemas prácticos a las 23:59 (09 de septiembre) 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.

Ejercicios

22.4-1, 22.4-2, 22.4-4, 22.4-5 (páginas 614 y 615), 22.5-1, 22.5-3, 22.5-4, 22.5-5, 22.5-7 (páginas 620 y 621).

Problemas conceptuales

1. Ejercicio 22-4: *Reachability* (Cormen et al. página 623)

Problemas prácticos

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

A - Lighting Away

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

Ronju is a night-guard at the "Lavish office buildings Ltd." headquarters. The office has a large grass field in front of the building. So every day, when Ronju comes to duty at the evening, it is his duty to turn on all the lights in the field. However, given the large size of the field and the large number of lights, it is very tiring for him to walk to each and every individual light to turn it on.

So he has devised an ingenious plan – he will swap the switches for light-sensitive triggers. A local electronic store nearby sells these funny trigger switches at a very cheap price. Once installed at a light-post, it will automatically turn that light on whenever it can sense some other light lighting up nearby. So from now on, Ronju can just manually flip a few switches, and the light from those will trigger nearby sensors, which will in turn light up some more lights nearby, and so on, gradually lighting up the whole field.

Now Ronju wonders: how many switches does he have to flip manually for this?

Input

The input starts with an integer T, the number of test cases to follow. Each test case will start with two integers N ($1 \leq N \leq 10\,000$) and M ($0 \leq M \leq 100\,000$), where N is the number of lights in the field, and M more lines of input follows in this input case. Each of these extra M lines will have two integers a and b separated by a space, where $1 \leq a, b \leq N$, indicating that if the light a lights up, it will trigger the light b to turn on as well (according to their distance, brightness, sensor sensitivity, orientation and other complicated factors).

Finally, every test case in the input will be followed by a blank line.

The input must be read from standard input.

Output

For each input test case, the output must be a single line of the format 'Case k: c' where k is the case number starting with 1, and c is the minimum number of lights that Ronju must turn on manually before all the lights in the whole field gets lit up.

The output must be written to standard output.

Sample Input	Sample Output

```
2
5 4
1 2
1 3
3 4
5 3
4 4
1 2
1 3
4 2
4 3
```
	```
	Case 1: 2
	Case 2: 2
	```

B - Doves and Bombs

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

It is the year 95 ACM (After the Crash of Microsoft). After many years of peace, a war has broken out. Your nation, the island of Evergreen Macros And Confusing Shortcuts (EMACS), is defending itself against the railway empire ruled by Visually Impaired Machinists (VIM).

In the days leading up to the outbreak of war, your government devoted a great deal of resources toward gathering intelligence on VIM. It discovered the following:

- The empire has a large network of railway stations connected by bidirectional tracks.
- Before the outbreak of the war, each railway station was directly connected to at most 10 other stations.
- Information is constantly being exchanged in VIM, but, due to a design flaw, the only way it can be exchanged is to send the messages by train. Before the outbreak of the war, it was possible to send a message by train from any station to any other station.
- As a last resort, the empire's central command can send messages by carrier pigeon, but it tries to avoid this at all costs, as the only pigeons suitable for the job must be imported, at great expense, from the far-away land of Pigeons In Courier Outfits (PICO).
- Once a pigeon has delivered a message to a railway station, it must rest, and thus cannot be used again. If a pigeon has delivered a broadcast message to a particular station, the message is passed on, if possible, by train.

Based on this information, the government of EMACS has come up with a plan to disrupt the activities of the evil empire. They will send bomber planes to bomb the railway stations, thus hampering communications in the empire. This will necessitate to acquire many carrier pigeons by the empire, distracting it from its deadly wartime activities.

Unfortunately, your government spent so much money on gathering intelligence that it has a very limited amount left to build bombs. As a result, it can bomb only one target. You have been charged with the task of determining the best candidate railway stations in the empire to bomb, based on their "pigeon value".

The "pigeon value" of a station is the minimum number of pigeons that after bombing this station, will be required to broadcast a message from the empire central command to all non-bombed stations. The location of the empire central command is unknown but we know that it is not located at a railway station. This implies, that when the central command needs to send a message to some non-bombed station they have to use at least one pigeon and then the message can be further transmitted by the railway.

Input

The input file contains several test cases. The data for each case begins with a line containing the following two integers:

- n the number of railway stations in the empire ($3 \leq n \leq 10\,000$). The stations will be numbered starting from 0, up to $n - 1$
- m the number of stations to be identified as candidate bombing targets ($1 \leq m \leq n$).

Next few lines consists of pairs of integers. Each pair (x, y) indicates the presence of a bidirectional railway line connecting railway stations x and y. This sequence is terminated by a line containing two minus 1 as shown in the sample input.

Input is terminated by a case where the value of $n = m = 0$. This case should not be processed.

The input must be read from standard input.

Output

For each case of input the output should give m most desirable railway stations to bomb. There should be exactly m lines, each with two integers separated by a single space. The first integer on each line will be the number of a railway station, and the second will be the "pigeon value" of the station. This list should be sorted, first by "pigeon value", in descending order, and within the same "pigeon value" by railway station numbers, in ascending order. Print a blank line after the output for each set of input.

The output must be written to standard output.

Sample Input

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

Sample Output

```
2 3
3 3
4 2
0 1

2 3
3 3
4 2
0 1
```

C - Don’t Care

Source file name: `care.py`
Time limit: 3 seconds

In mathematical logic and computer science, an *abstract rewriting system* (ARS) is a convenient framework for describing important properties of artifacts such as logical inference, software systems, social interaction, etc. In its simplest form, an ARS is a set *objects* together with a *binary relation* on the objects expressing how these can transition.

Some ARS are of very special interest because even if an object can make a transition in more than one way, these transitions will eventually yield the same result. Such an ARS guarantees that if some irreducible object is reached by successive transition steps, then this is also the case independently of the choice of the first transition taken. In other words, an ARS with this property converts nondeterminism from “don’t know” into “don’t care”, thus avoiding the need for backtracking. Let us agree to call an ARS with this property a *Don’t Care ARS*.

Consider the usual rules of elementary arithmetic: they form an abstract rewriting system. For example, the expression $(11 + 9) \times (2 + 4)$ can be evaluated starting either at the left or at the right parentheses; however, in both cases the same *irreducible* result is obtained eventually. This suggests that the arithmetic reduction system is don’t care.

$$\begin{tikzcd}
 & (11+9)\times(2+4) \arrow[dl, "\text{eval left}"'] \arrow[dr, "\text{eval right}"] & \\
20\times(2+4) \arrow[dr, "\text{eval right}"'] & & (11+9)\times 6 \arrow[dl, "\text{eval left}"] \\
 & 20\times 6 \arrow[d] & \\
 & 120 &
\end{tikzcd}$$

Borrowed from Wikipedia.org

There is no general purpose algorithm that can *always* determine if a given ARS is don’t care, i.e., checking the don’t care property is in general undecidable for arbitrary ARS. However, because of its importance, this property has been thoroughly studied and decision procedures have been obtained for restricted versions of the problem. In particular, the don’t care property can always be checked mechanically for systems having a finite number of objects.

Let $\mathcal{A} = (A, \to)$ denote an ARS with a set of objects A and a transition relation $\to$ on A (that is, $\to \subseteq A^2$). Then, $\mathcal{A}$ is don’t care if and only if the following two conditions are satisfied:

1. the relation $\to$ is well-founded, i.e., no object can be reduced with $\to$ indefinitely, and
2. for any $a, b, c \in A$, if there are direct $\to$-transitions from a to b and from a to c, then there is $d \in A$ such that d is reachable via $\to$-transitions both from b and c.

You have been assigned the task of implementing an efficient algorithm for deciding the don’t care property for finite ARS.

Input

The input consists of several test cases. The first line of each test case contains two numbers n and m ($1 \leq n \leq 1000$ and $0 \leq m \leq \min(n^2, 25000)$) defining, respectively, the number of objects and the number of transitions in the ARS. Each of the next m lines contains a pair of blank-separated non-negative integers a, b ($0 \leq a, b < n$) indicating that there is a transition in the ARS from object a to object b.

The end of the input is given by $n = m = 0$.

The input must be read from standard input.

Output

For each ARS output exactly one line. If the input ARS is don’t care, then output

```
1
```

Otherwise, output

```
0
```

The output must be written to standard output.

Sample Input	**Sample Output**

```
3 2
0 1
1 2
2 2
0 1
0 1
2 2
0 1
1 0
3 2
0 1
0 2
0 0
```
	```
	1
	1
	0
	0
	```

D - Dominos

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

Dominos are lots of fun. Children like to stand the tiles on their side in long lines. When one domino falls, it knocks down the next one, which knocks down the one after that, all the way down the line. However, sometimes a domino fails to knock the next one down. In that case, we have to knock it down by hand to get the dominos falling again.

Your task is to determine, given the layout of some domino tiles, the minimum number of dominos that must be knocked down by hand in order for all of the dominos to fall.

Input

The first line of input contains one integer specifying the number of test cases to follow. Each test case begins with a line containing two integers, each no larger than 100 000. The first integer n is the number of domino tiles and the second integer m is the number of lines to follow in the test case. The domino tiles are numbered from 1 to n. Each of the following lines contains two integers x and y indicating that if domino number x falls, it will cause domino number y to fall as well.

The input must be read from standard input.

Output

For each test case, output a line containing one integer, the minimum number of dominos that must be knocked over by hand in order for all the dominos to fall.

The output must be written to standard output.

Sample Input	**Sample Output**

`1`
`3 2`
`1 2`
`2 3`
	`1`