AGRA - Árboles y Grafos, 2023-2
Tarea 2: Semanas 3 y 4

Para entregar el miércoles 23 de agosto de 2023

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.

Ejercicios

La siguiente colección de ejercicios, tomados del libro de Cormen et al. es para repasar y afianzar conceptos, pero no deben ser entregados como parte de la tarea:

20.1-1(página 552), 20.1-2 , 20.1-3, 20.2-1 (página 562), 20.2-2, 20.2-4 , 20.3-1 (página 571), 20.3-2, 22.3-3, 22.3-7, 22.3-12.

Problemas conceptuales

1. Ejercicio 3: *Extended Topological Sort* (Kleinberg & Tardos página 107).
2. Ejercicio 7: *Wireless Networks* (Kleinberg & Tardos página 108).

Problemas prácticos

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

A - Monitoring the Amazon

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

A network of autonomous, battery-powered, data acquisition stations has been installed to monitor the climate in the region of Amazon. An order-dispatch station can initiate transmission of instructions to the control stations so that they change their current parameters. To avoid overloading the battery, each station (including the order-dispatch station) can only transmit to two other stations. The destinataries of a station are the two closest stations. In case of draw, the first criterion is to chose the westernmost (leftmost on the map), and the second criterion is to chose the southernmost (lowest on the map).

You are commissioned by Amazon State Government to write a program that decides if, given the localization of each station, messages can reach all stations.

Input

The input consists of an integer N, followed by N pairs of integers X_i, Y_i, indicating the localization coordinates of each station. The first pair of coordinates determines the position of the order-dispatch station, while the remaining $N - 1$ pairs are the coordinates of the other stations. The following constraints are imposed: $-20 \leq X_i, Y_i \leq 20$, and $1 \leq N \leq 1000$. The input is terminated with $N = 0$.

The input must be read from standard input.

Output

For each given expression, the output will echo a line with the indicating if all stations can be reached or not (see sample output for the exact format).

The output must be written to standard output.

Sample Input	Sample Output

4
1 0 0 1 -1 0 0 -1
8
1 0 1 1 0 1 -1 1 -1 0 -1 -1 0 -1 1 -1
6
0 3 0 4 1 3 -1 3 -1 -4 -2 -5
0
	All stations are reachable.
	All stations are reachable.
	There are stations that are unreachable.

B - Beverages

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

Dilbert has just finished college and decided to go out with friends. Dilbert has some strange habits and thus he decided to celebrate this important moment of his life drinking a lot. He will start drinking beverages with low alcohol content, like beer, and then will move to a beverage that contains more alcohol, like wine, until there are no more available beverages. Once Dilbert starts to drink wine he will not drink beer again, so the alcohol content of the beverages never decreases with the time.

You should help Dilbert by indicating an order in which he can drink the beverages in the way he wishes.

Input

Each test case starts with $1 \leq N \leq 100$, the number of available beverages. Then will follow N lines, informing the name of each beverage, a name has less than 51 characters and has no white spaces. Then there is another line with an integer $0 \leq M \leq 200$ and M lines in the form B_1 B_2 will follow, indicating that beverage B_2 has more alcohol that beverage B_1, so Dilbert should drink beverage B_1 before he starts drinking beverage B_2. Be sure that this relation is transitive, so if there is also a relation B_2 B_3 you should drink B_1 before you can drink B_3. There is a blank line after each test case. In the case there is no relation between two beverages Dilbert should start drinking the one that appears first in the input. The input is terminated by end of file (EOF).

The input must be read from standard input.

Output

For each test case you must print the message:

'`Case #`C`: Dilbert should drink beverages in this order:` B_1 B_2 ... B_N`.`'

where C is the number of the test case, starting from 1, and B_1 B_2 ... B_N is a list of the beverages such that the alcohol content of beverage B_{i+1} is not less than the alcohol content of beverage B_{i-1}. After each test case you must print a blank line.

The output must be written to standard output.

Sample Input

```
3
vodka
wine
beer
2
wine vodka
beer wine

5
wine
beer
rum
apple-juice
cachaca
6
beer cachaca
apple-juice beer
apple-juice rum
beer rum
beer wine
wine cachaca

10
cachaca
rum
apple-juice
tequila
whiskey
wine
vodka
beer
martini
gin
11
beer whiskey
apple-juice gin
rum cachaca
vodka tequila
apple-juice martini
rum gin
wine whiskey
apple-juice beer
beer rum
wine vodka
beer tequila
```

Sample Output

```
Case #1: Dilbert should drink beverages in this order: beer wine vodka.

Case #2: Dilbert should drink beverages in this order: apple-juice beer wine rum cachaca.

Case #3: Dilbert should drink beverages in this order: apple-juice wine vodka beer rum cachaca tequila whiskey martini gin.
```

C - Ocean Currents

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

For a boat on a large body of water, strong currents can be dangerous, but with careful planning, they can be harnessed to help the boat reach its destination. Your job is to help in that planning.

At each location, the current flows in some direction. The captain can choose to either go with the flow of the current, using no energy, or to move one square in any other direction, at the cost of one energy unit. The boat always moves in one of the following eight directions: north, south, east, west, northeast, northwest, southeast, southwest. The boat cannot leave the boundary of the lake.

You are to help him devise a strategy to reach the destination with the minimum energy consumption.

Input

The lake is represented as a rectangular grid. The first line of input contains two integers r and c, the number of rows and columns in the grid. The grid has no more than 1000 rows and no more than 1000 columns. Each of the following r lines contains exactly c characters, each a digit from 0 to 7 inclusive. The character `'0'` means the current flows north (i.e. up in the grid, in the direction of decreasing row number), `'1'` means it flows northeast, `'2'` means east (i.e. in the direction of increasing column number), `'3'` means southeast, and so on in a clockwise manner:

```
7 0 1
 \|/
6-*-2
 /|\
5 4 3
```

The line after the grid contains a single integer n, the number of trips to be made, which is at most 50. Each of the following n lines describes a trip using four integers r_s, c_s, r_d, c_d, giving the row and column of the starting point and the destination of the trip. Rows and columns are numbered starting with 1.

The input must be read from standard input.

Output

For each trip, output a line containing a single integer, the minimum number of energy units needed to get from the starting point to the destination.

The output must be written to standard output.

Sample Input	**Sample Output**

```
5 5
04125
03355
64734
72377
02062
3
4 2 4 2
4 5 1 4
5 3 3 4
```
	```
	0
	2
	1
	```

D - Getting Gold

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

We're building an old-school back-to-basics computer game. It's a very simple text based adventure game where you walk around and try to find treasure, avoiding falling into traps. The game is played on a rectangular grid and the player gets very limited information about her surroundings.

The game will consist of the player moving around on the grid for as long as she likes (or until she falls into a trap). The player can move up, down, left and right (but not diagonally). She will pick up gold if she walks into the same square as the gold is. If the player stands next to (i.e., immediately up, down, left, or right of) one or more traps, she will "sense a draft" but will not know from what direction the draft comes, or how many traps she's near. If she tries to walk into a square containing a wall, she will notice that there is a wall in that direction and remain in the position where she was.

For scoring purposes, we want to show the player how much gold she could have gotten safely. That is, how much gold can a player get playing with an optimal strategy and always being sure that the square she walked into was safe. The player does not have access to the map and the maps are randomly generated for each game so she has no previous knowledge of the game.

Input

The input file contains several test cases, each of them as described below.

The first line of input contains two positive integers W and H, neither of them smaller than 3 or larger than 50, giving the width and the height of the map, respectively. The next H lines contain W characters each, giving the map. The symbols that may occur in a map are as follows:

`P` — the player's starting position

`G` — a piece of gold

`T`— a trap

`#` — a wall

`.` — normal floor

There will be exactly one `'P'` in the map, and the border of the map will always contain walls.

The input must be read from standard input.

Output

For each test case, write to the output the number of pieces of gold the player can get without risking falling into a trap, on a line by itself.

The output must be written to standard output.

Sample Input	**Sample Output**

```
7 4
#######
#P.GTG#
#..TGG#
#######
8 6
########
#...GTG#
#..PG.G#
#...G#G#
#..TG.G#
########
```
	```
	1
	4
	```

E - Exchange Rates

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

Using money to pay for goods and services usually makes life easier, but sometimes people prefer to trade items directly without any money changing hands. In order to ensure a consistent "price", traders set an exchange rate between items. The exchange rate between two items *A* and *B* is expressed as two positive integers *m* and *n*, and says that *m* of item *A* is worth n of item *B*. For example, 2 stoves might be worth 3 refrigerators. (Mathematically, 1 stove is worth 1.5 refrigerators, but since it's hard to find half a refrigerator, exchange rates are always expressed using integers.)

Your job is to write a program that, given a list of exchange rates, calculates the exchange rate between any two items.

Input

The input file contains one or more commands, followed by a line beginning with a period that signals the end of the file. Each command is on a line by itself and is either an assertion or a query. An assertion begins with an exclamation point and has the format

! *m itema* = *n itemb*

where *itema* and *itemb* are distinct item names and *m* and *n* are both positive integers less than 100. This command says that *m* of *itema* are worth *n* of *itemb*. A query begins with a question mark, is of the form

? *itema* = *itemb*

and asks for the exchange rate between *itema* and *itemb*, where *itema* and *itemb* are distinct items that have both appeared in previous assertions (although not necessarily the same assertion).

The input must be read from standard input.

Output

For each query, output the exchange rate between *itema* and *itemb* based on all the assertions made up to that point. Exchange rates must be in integers and must be reduced to lowest terms. If no exchange rate can be determined at that point, use question marks instead of integers. Format all output exactly as shown in the example.

Note:

- Item names will have length at most 20 and will contain only lowercase letters.
- Only the singular form of an item name will be used (no plurals).
- There will be at most 60 distinct items.
- There will be at most one assertion for any pair of distinct items.
- There will be no contradictory assertions. For example, "2 pig = 1 cow", "2 cow = 1 horse", and "2 horse = 3 pig" are contradictory.
- Assertions are not necessarily in lowest terms, but output must be.
- Although assertions use numbers less than 100, queries may result in larger numbers that will not exceed 10000 when reduced to lowest terms.

The output must be written to standard output.

Sample Input

```
! 6 shirt = 15 sock
! 47 underwear = 9 pant
? sock = shirt
? shirt = pant
! 2 sock = 1 underwear
? pant = shirt
```

Sample Output

```
5 sock = 2 shirt
? shirt = ? pant
45 pant = 188 shirt
```