A - Arbitrage II

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

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 US Dollar buys 0.5 British pound, 1 British pound buys 10.0 French francs, and 1 French franc buys 0.21 US dollar. Then, by converting currencies, a clever trader can start with 1 US dollar and buy $0.5 \cdot 10.0 \cdot 0.21 = 1.05$ US dollars, making a profit of 5 percent. Your job is to write a program that takes a list of currency exchange rates as input and then determines whether arbitrage is possible or not.

Input

The input will contain one or more test cases. On the first line of each test case there is an integer n ($1 \leq n \leq 100$), representing the number of different currencies. The next n lines each contain the name of one currency. Within a name no spaces will appear. The next line contains one integer m, representing the length of the table to follow. The last m lines each contain the name c_i of a source currency, a real number r_{ij} which represents the exchange rate from c_i to c_j, and a name c_j of the destination currency. Exchanges which do not appear in the table are impossible. Test cases are separated from each other by a blank line. Input is terminated by a value of zero (0) for n.

The input must be read from standard input.

Output

For each test case, print one line telling whether arbitrage is possible or not in the format "`Case #: Yes`", respectively "`Case #: No`", where "`#`" represents the case number (starting from 1).

The output must be written to standard output.

Sample Input	**Sample Output**

`3`
`USDollar`
`BritishPound`
`FrenchFranc`
`3`
`USDollar 0.5 BritishPound`
`BritishPound 10.0 FrenchFranc`
`FrenchFranc 0.21 USDollar`
 	
`3`
`USDollar`
`BritishPound`
`FrenchFranc`
`6`
`USDollar 0.5 BritishPound`
`USDollar 4.9 FrenchFranc`
`BritishPound 10.0 FrenchFranc`
`BritishPound 1.99 USDollar`
`FrenchFranc 0.09 BritishPound`
`FrenchFranc 0.19 USDollar`
 	
`0`
	`Case 1: Yes`
	`Case 2: No`

B - Number Maze

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

Consider a number maze represented as a two dimensional array of numbers comprehended between 0 and 999, as exemplified below.

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

The maze can be traversed following any orthogonal direction (i.e., north, south, east, and west). Considering that each cell represents a cost, then finding the minimum cost to travel the maze from one entry point to an exit point may pose you a reasonable challenge.

Your task is to find the minimum cost value to go from the top- left corner to the bottom-right corner of a given number maze of size $N \times M$, where $1 \leq N, M \leq 999$. Note that the solution for the given example is 24.

Input

The input contains several mazes. The first input line contains a positive integer defining the number of mazes that follow. Each maze is defined by: one line with the number of rows N, one line with the number of columns M, and N lines, one per each row of the maze, containing the maze numbers separated by spaces.

The input must be read from standard input.

Output

For each maze, output one line with the required minimum value.

The output must be written to standard output.

Sample Input	**Sample Output**
2 4 5 0 3 1 2 9 7 3 4 9 9 1 7 5 5 3 2 3 4 2 5 1 6 0 1 2 3 4 5	24 15

C - Numbering Paths

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

Problems that process input and generate a simple "yes" or "no" answer are called decision problems. One class of decision problems, the *NP-complete problems*, are not amenable to general efficient solutions. Other problems may be simple as decision problems, but enumerating all possible "yes" answers may be very difficult (or at least time-consuming).

This problem involves determining the number of routes available to an emergency vehicle operating in a city of one-way streets. Given the intersections connected by one-way streets in a city, you are to write a program that determines the number of different routes between each intersection. A route is a sequence of one-way streets connecting two intersections.

Intersections are identified by non-negative integers. A one-way street is specified by a pair of intersections. For example, j k indicates that there is a one-way street from intersection j to intersection k. Note that two-way streets can be modeled by specifying two one-way streets: j k and k j.

For example, consider a city of four intersections connected by the following one-way streets:

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

There is one route from intersection 0 to 1, two routes from 0 to 2 (the routes are $0 \to 1 \to 2$ and $0 \to 2$), two routes from 0 to 3, one route from 1 to 2, one route from 1 to 3, one route from 2 to 3, and no other routes.

It is possible for an infinite number of different routes to exist. For example, if the intersections above are augmented by the street 3 2, there is still only one route from 0 to 1, but there are infinitely many different routes from 0 to 2. This is because the street from 2 to 3 and back to 2 can be repeated yielding a different sequence of streets and hence a different route. Thus the route $0 \to 2 \to 3 \to 2 \to 3 \to 2$ is a different route than $0 \to 2 \to 3 \to 2$.

Input

The input is a sequence of city specifications. Each specification begins with the number of one-way streets in the city followed by that many one-way streets given as pairs of intersections. Each pair j k represents a one-way street from intersection j to intersection k. In all cities, intersections are numbered sequentially from 0 to the "largest" intersection. All integers in the input are separated by whitespace. The input is terminated by end-of-input. There will never be a one-way street from an intersection to itself. No city will have more than 10 intersections.

The input must be read from standard input.

Output

For each city specification, a square matrix of the number of different routes from intersection j to intersection k is printed. If the matrix is denoted M, then $M[j][k]$ is the number of different routes from intersection j to intersection k. The matrix M should be printed in row-major order, one row per line. Each matrix should be preceded by the string "`matrix for city k`" (with k appropriately instantiated, beginning with 0).

If there are an infinite number of different paths between two intersections a -1 should be printed. All entries in a row should be separated by exactly one whitespace.

The output must be written to standard output.

Sample Input	**Sample Output**
7 0 1 0 2 0 4 2 4 2 3 3 1 4 3 5 0 2 0 1 1 5 2 5 2 1 9 0 1 0 2 0 3 0 4 1 4 2 1 2 0 3 0 3 1	matrix for city 0 0 4 1 3 2 0 0 0 0 0 0 2 0 2 1 0 1 0 0 0 0 1 0 1 0 matrix for city 1 0 2 1 0 0 3 0 0 0 0 0 1 0 1 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 matrix for city 2 -1 -1 -1 -1 -1 0 0 0 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0 0 0 0

D - Wormholes

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

In the year 2163, wormholes were discovered. A wormhole is a subspace tunnel through space and time connecting two star systems. Wormholes have a few peculiar properties:

- Wormholes are *one-way* only.
- The time it takes to travel through a wormhole is negligible.
- A wormhole has two end points, each situated in a star system.
- A star system may have more than one wormhole end point within its boundaries.
- For some unknown reason, starting from our solar system, it is always possible to end up in any star system by following a sequence of wormholes (maybe Earth is the centre of the universe).
- Between any pair of star systems, there is at most one wormhole in either direction.
- There are no wormholes with both end points in the same star system.

All wormholes have a constant time difference between their end points. For example, a specific wormhole may cause the person travelling through it to end up 15 years in the future. Another wormhole may cause the person to end up 42 years in the past.

A brilliant physicist, living on earth, wants to use wormholes to study the Big Bang. Since warp drive has not been invented yet, it is not possible for her to travel from one star system to another one directly. This *can* be done using wormholes, of course.

The scientist wants to reach a cycle of wormholes somewhere in the universe that causes her to end up in the past. By travelling along this cycle a lot of times, the scientist is able to go back as far in time as necessary to reach the beginning of the universe and see the Big Bang with her own eyes. Write a program to find out whether such a cycle exists.

Input

The input file starts with a line containing the number of cases c to be analyzed. Each case starts with a line with two blank-separated numbers n and m indicating the number of star systems ($1 \leq n \leq 1000$) and the number of wormholes ($0 \leq m \leq 2000$). The star systems are numbered from 0 (our solar system) through $n - 1$. Then m lines follow each with three integer numbers x, y, and t indicating that there is a wormhole allowing anyone to travel from the star system numbered x to the star system numbered y, thereby ending up t ($-1000 \leq t \leq 1000$) years in the future.

The input must be read from standard input.

Output

The output consists of c lines, one line for each case, containing the word

```
possible
```

if it is indeed possible to go back in time indefinitely, or

```
not possible
```

if this is not possible with the given set of star systems and wormholes.

The output must be written to standard output.

Sample Input	**Sample Output**
2 3 3 0 1 1000 1 2 15 2 1 -42 4 4 0 1 10 1 2 20 2 3 30 3 0 -60	possible not possible