Problem A: Find the Winning Move

Source file: `win.{c, cpp, java, pas}`
Input file: `win.in`
Output file: `win.out`

4x4 tic-tac-toe is played on a board with four rows (numbered 0 to 3 from top to bottom) and four columns (numbered 0 to 3 from left to right). There are two players, *x* and *o*, who move alternately with *x* always going first. The game is won by the first player to get four of his or her pieces on the same row, column, or diagonal. If the board is full and neither player has won then the game is a draw.

Assuming that it is *x*'s turn to move, *x* is said to have a ***forced win*** if *x* can make a move such that no matter what moves *o* makes for the rest of the game, *x* can win. This does not necessarily mean that *x* will win on the very next move, although that is a possibility. It means that *x* has a winning strategy that will guarantee an eventual victory regardless of what *o* does.

Your job is to write a program that, given a partially-completed game with *x* to move next, will determine whether *x* has a forced win. You can assume that each player has made at least two moves, that the game has not already been won by either player, and that the board is not full.

The input file contains one or more test cases, followed by a line beginning with a dollar sign that signals the end of the file. Each test case begins with a line containing a question mark and is followed by four lines representing the board; formatting is exactly as shown in the example. The characters used in a board description are the period (representing an empty space), lowercase *x*, and lowercase *o*. For each test case, output a line containing the (*row*, *column*) position of the *first* forced win for *x*, or '#####' if there is no forced win. Format the output exactly as shown in the example.

For this problem, the *first* forced win is determined by board position, not the number of moves required for victory. Search for a forced win by examining positions (0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), ..., (3, 2), (3, 3), in that order, and output the first forced win you find. In the second test case below, note that *x* could win immediately by playing at (0, 3) or (2, 0), but playing at (0, 1) will still ensure victory (although it unnecessarily *delays* it), and position (0, 1) comes first.

Example input:

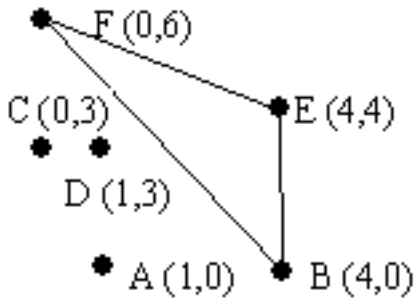
```
?
....
.xo.
.ox.
....
?
o...
.ox.
.xxx
xooo
$
```

Example output:

```
#####
(0,1)
```

Problem B: Myacm Triangles

Source file: `triangle.{c, cpp, java, pas}`
Input file: `triangle.in`
Output file: `triangle.out`



There has been considerable archeological work on the ancient Myacm culture. Many artifacts have been found in what have been called power fields: a fairly small area, less than 100 meters square where there are from four to fifteen tall monuments with crystals on top. Such an area is mapped out above. Most of the artifacts discovered have come from inside a triangular area between just three of the monuments, now called the power triangle. After considerable analysis archeologists agree how this triangle is selected from all the triangles with three monuments as vertices: it is the triangle with the largest possible area that does not contain any other monuments inside the triangle or on an edge of the triangle. Each field contains only one such triangle.

Archeological teams are continuing to find more power fields. They would like to automate the task of locating the power triangles in power fields. Write a program that takes the positions of the monuments in any number of power fields as input and determines the power triangle for each power field.

A useful formula: the area of a triangle with vertices (x_1, y_1), (x_2, y_2), and (x_3, y_3) is the absolute value of

$$0.5 \times [(y_3 - y_1)(x_2 - x_1) - (y_2 - y_1)(x_3 - x_1)].$$

For each power field there are several lines of data. The first line is the number of monuments: at least 4, and at most 15. For each monument there is a data line that starts with a one character label for the monument and is followed by the coordinates of the monument, which are nonnegative integers less than 100. The first label is A, and the next is B, and so on.

There is at least one such power field described. The end of input is indicated by a 0 for the number of monuments. The first sample data below corresponds to the diagram in the problem.

For each power field there is one line of output. It contains the three labels of the vertices of the power triangle, listed in increasing alphabetical order, with no spaces.

Example input:

```
6
```

```
A 1 0
B 4 0
C 0 3
D 1 3
E 4 4
F 0 6
4
A 0 0
B 1 0
C 99 0
D 99 99
0
```

Example output:

```
BEF
BCD
```

Problem C: Exchange Rates

Source file: `exchange.{c, cpp, java, pas}`
Input file: `exchange.in`
Output file: `exchange.out`

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.

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). 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*.

Example input:

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

Example output:

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

Problem E: Automatic Editing

Source file: `autoedit.{c, cpp, java, pas}`
Input file: `autoedit.in`
Output file: `autoedit.out`

Text-processing tools like *awk* and *sed* allow you to automatically perform a sequence of editing operations based on a script. For this problem we consider the specific case in which we want to perform a series of string replacements, within a single line of text, based on a fixed set of rules. Each rule specifies the string to find, and the string to replace it with, as shown below.

Rule	Find	Replace-by
1.	`ban`	`bab`
2.	`baba`	`be`
3.	`ana`	`any`
4.	`ba b`	`hind the g`

To perform the edits for a given line of text, start with the first rule. Replace the first occurrence of the *find* string within the text by the *replace-by* string, then try to perform the same replacement *again* on the new text. Continue until the *find* string no longer occurs within the text, and then move on to the next rule. Continue until all the rules have been considered. Note that (1) when searching for a *find* string, you always start searching at the beginning of the text, (2) once you have finished using a rule (because the *find* string no longer occurs) you never use that rule again, and (3) case is significant.

For example, suppose we start with the line

```
banana boat
```

and apply these rules. The sequence of transformations is shown below, where occurrences of a *find* string are underlined and replacements are boldfaced. Note that rule 1 was used twice, then rule 2 was used once, then rule 3 was used zero times, and then rule 4 was used once.

Before	After
`ban`ana boat	**bab**ana boat
ba`ban`a boat	ba**bab**a boat
`baba`ba boat	**be**ba boat
be`ba b`oat	be**hind the g**oat

The input contains one or more test cases, followed by a line containing only 0 (zero) that signals the end of the file. Each test case begins with a line containing the number of rules, which will be between 1 and 10. Each rule is specified by a pair of lines, where the first line is the *find* string and the second line is the *replace-by* string. Following all the rules is a line containing the text to edit. For each test case, output a line containing the final edited text.

Both *find* and *replace-by* strings will be at most 80 characters long. *Find* strings will contain at least one character, but *replace-by* strings may be empty (indicated in the input file by an empty line). During the

edit process the text may grow as large as 255 characters, but the final output text will be less than 80 characters long.

The first test case in the sample input below corresponds to the example shown above.

Example input:

```
4
ban
bab
baba
be
ana
any
ba b
hind the g
banana boat
1
t
sh
toe or top
0
```

Example output:

```
behind the goat
shoe or shop
```