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:

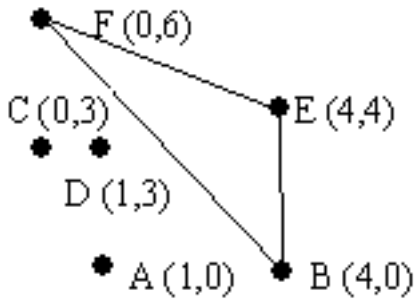
```
#####
(0,1)
```

```
?
....
.xo.
.ox.
....
?
o...
.ox.
.xxx
xooo
?
xoox
..o.
..xx
..o.
?
xoo.
..x.
..x.
xoo.
?
oxxo
xoox
xoox
x..x
?
xoxo
o..x
x..o
oxox
?
xo.x
o..x
ox.x
oo..
?
x..x
.xx.
.oo.
o..o
?
oox.
oo..
x..x
..x.
?
.oo.
.xx.
.oo.
.xx.
$
```

```
#####
(0,1)
(1,0)
(0,3)
#####
#####
(3,3)
#####
(2,2)
#####
```

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
```

```
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
6
A 10 3
B 10 6
C 25 3
D 40 3
E 42 20
F 60 1
4
A 0 1
B 10 1
C 0 10
D 15 0
4
A 0 0
B 1 0
C 2 5
D 3 0
15
A 0 30
B 45 32
C 90 30
D 60 1
E 25 0
F 24 15
G 25 15
H 28 20
I 24 16
J 24 17
K 61 16
L 61 17
M 61 18
N 62 16
O 62 17
0
```

BEF
BCD
BDE
ABC
BCD
BDE

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
```

```
! 63 mango = 14 papaya
? papaya = mango
! 5 passionfruit = 31 guava
? guava = mango
? papaya = passionfruit
! 1 thermonuclearwarhead = 3 secretrockethovercar
! 1 fork = 1 spoon
! 2 thermonuclearwarhead = 4 armoredcommandotruck
? armoredcommandotruck = secretrockethovercar
! 1 p = 1 q
! 3 w = 3 x
! 4 x = 4 y
! 1 s = 1 t
! 1 q = 1 r
! 1 r = 1 s
! 3 j = 12 k
! 1 k = 1 l
! 1 l = 1 m
! 3 b = 4 c
! 1 c = 5 d
! 3 d = 2 e
! 1 e = 7 f
! 4 f = 2 g
! 1 m = 1 n
! 1 t = 1 u
! 1 u = 1 v
! 1 a = 2 b
! 1 n = 1 o
! 1 o = 1 p
! 2 v = 2 w
! 1 g = 10 h
! 7 h = 2 i
! 32 i = 16 j
! 5 y = 1 z
? a = z
! 9 cat = 1 dog
! 1 cow = 12 chicken
? chicken = dog
! 6 sheep = 3 mule
! 4 dog = 2 mule
! 2 chicken = 1 sheep
? cow = cat
? chicken = dog
! 4 television = 83 knife
! 91 knife = 2 dvdplayer
! 1 microwave = 37 fork
! 7 toaster = 2 microwave
! 2 spoon = 1 knife
? dvdplayer = toaster
? fork = cow
? toaster = spoon
! 47 car = 13 sportutilityvehicle
! 32 truck = 53 car
? sportutilityvehicle = truck
! 6 shirt = 15 sock
! 47 underwear = 9 pant
? sock = shirt
? shirt = pant
! 2 sock = 1 underwear
? pant = shirt
.
```

```
2 papaya = 9 mango
? guava = ? mango
? papaya = ? passionfruit
2 armoredcommandotruck = 3 secretrockethovercar
9 a = 320 z
? chicken = ? dog
1 cow = 54 cat
2 chicken = 1 dog
74 dvdplayer = 637 toaster
? fork = ? cow
7 toaster = 74 spoon
689 sportutilityvehicle = 1504 truck
5 sock = 2 shirt
? shirt = ? pant
45 pant = 188 shirt
```

Problem D: Loansome Car Buyer

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

Kara Van and Lee Sabre are lonesome. A few months ago they took out a loan to buy a new car, but now they're stuck at home on Saturday night without wheels and without money. You see, there was a wreck and the car was totaled. Their insurance paid $10,000, the current value of the car. The only problem is that they owed the bank $15,000, and the bank wanted payment immediately, since there was no longer a car for collateral. In just a few moments, this unfortunate couple not only lost their car, but lost an additional $5,000 in cash too.

What Kara and Lee failed to account for was depreciation, the loss in value as the car ages. Each month the buyer's loan payment reduces the amount owed on the car. However, each month, the car also depreciates as it gets older. Your task is to write a program that calculates the first time, measured in months, that a car buyer owes less money than a car is worth. For this problem, depreciation is specified as a percentage of the previous month's value.

Input consists of information for several loans. Each loan consists of one line containing the duration in months of the loan, the down payment, the amount of the loan, and the number of depreciation records that follow. All values are nonnegative, with loans being at most 100 months long and car values at most $75,000. Since depreciation is not constant, the varying rates are specified in a series of depreciation records. Each depreciation record consists of one line with a month number and depreciation percentage, which is more than 0 and less than 1. These are in strictly increasing order by month, starting at month 0. Month 0 is the depreciation that applies immediately after driving the car off the lot and is always present in the data. All the other percentages are the amount of depreciation at the end of the corresponding month. Not all months may be listed in the data. If a month is not listed, then the previous depreciation percentage applies. The end of the input is signalled by a negative loan duration - the other three values will be present but indeterminate.

For simplicity, we will assume a 0% interest loan, thus the car's initial value will be the loan amount plus the down payment. It is possible for a car's value and amount owed to be positive numbers less than $1.00. Do *not* round values to a whole number of cents ($7,347.635 should not be rounded to $7,347.64).

Consider the first example below of borrowing $15,000 for 30 months. As the buyer drives off the lot, he still owes $15,000, but the car has dropped in value by 10% to $13,950. After 4 months, the buyer has made 4 payments, each of $500, and the car has further depreciated 3% in months 1 and 2 and 0.2% in months 3 and 4. At this time, the car is worth $13,073.10528 and the borrower only owes $13,000.

For each loan, the output is the number of complete months before the borrower owes less than the car is worth. Note that English requires plurals (5 months) on all values other than one (1 month).

Example input:

```
30 500.0 15000.0 3
```

```
0 .10
1 .03
3 .002
12 500.0 9999.99 2
0 .05
2 .1
60 2400.0 30000.0 3
0 .2
1 .05
12 .025
-99 0 17000 1
```

Example output:

```
4 months
1 month
49 months
```

```
30 500.0 15000.0 3
0 .10
1 .03
3 .002
12 500 9999.99 2
0 .05
2 .1
60 2400.0 30000.0 3
0 .2
1 .05
12 .025
100 0 60000 101
0 .1
1 .1
2 .01
3 .01
4 .1
5 .1
6 .01
7 .01
8 .01
9 .01
10 .1
11 .01
12 .01
13 .01
14 .01
15 .01
16 .01
17 .01
18 .01
19 .01
20 .1
21 .01
22 .01
23 .01
24 .01
25 .01
26 .01
27 .01
28 .01
29 .01
30 .1
31 .01
32 .01
33 .01
34 .01
35 .01
36 .01
37 .01
38 .01
39 .01
40 .1
41 .01
42 .01
43 .01
44 .01
45 .01
46 .01
47 .01
48 .01
```

```
49 .01
50 .1
51 .01
52 .01
53 .01
54 .01
55 .01
56 .01
57 .01
58 .01
59 .01
60 .1
61 .01
62 .01
63 .01
64 .01
65 .01
66 .01
67 .01
68 .01
69 .01
70 .1
71 .1
72 .1
73 .1
74 .1
75 .01
76 .01
77 .01
78 .01
79 .01
80 .1
81 .1
82 .1
83 .1
84 .1
85 .1
86 .1
87 .1
88 .1
89 .1
90 .1
91 .1
92 .1
93 .01
94 .01
95 .01
96 .1
97 .1
98 .1
99 .1
100 .1
100 0 75000 2
0 .10
99 .5
100 74999 1 1
0 .20
36 2000 10000 4
0 .2
1 .1
3 .01
```

```
10 .005
36 0 12000 3
0 .25
1 .1
10 .005
23 3000 5000 2
0 .15
20 .005
42 4000 0 1
0 .79
-10 0 0 4
```

```
4 months
1 month
49 months
99 months
100 months
0 months
11 months
27 months
0 months
0 months
```

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:

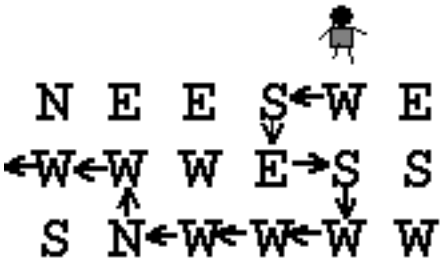
```
behind the goat
shoe or shop
```

```
4
ban
bab
baba
be
ana
any
ba b
hind the g
banana boat
1
t
sh
toe or top
1
x
o
xxx
4
A
?
T

tat
tt
T
?
TtTTaTTTtTTTTaTTTTTt
2
on
off
off
on
Upon this only
10
a
bxb
b
cxc
c
dxd
d
exe
e
fxf
f
gxg
g
hxh
h

xxxxxxxxxxxxxxxxxxxxx
22x
22
b
a
0
```

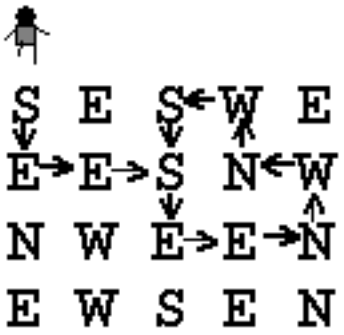
```
behind the goat
shoe or shop
ooo
ttt
Upon this only
bbbbbbx
```

Problem F: Robot Motion

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



Grid 1



Grid 2

A robot has been programmed to follow the instructions in its path. Instructions for the next direction the robot is to move are laid down in a grid. The possible instructions are

`N` north (up the page)
`S` south (down the page)
`E` east (to the right on the page)
`W` west (to the left on the page)

For example, suppose the robot starts on the north (top) side of Grid 1 and starts south (down). The path the robot follows is shown. The robot goes through 10 instructions in the grid before leaving the grid.

Compare what happens in Grid 2: the robot goes through 3 instructions only once, and then starts a loop through 8 instructions, and never exits.

You are to write a program that determines how long it takes a robot to get out of the grid or how the robot loops around.

There will be one or more grids for robots to navigate. The data for each is in the following form. On the first line are three integers separated by blanks: the number of rows in the grid, the number of columns in the grid, and the number of the column in which the robot enters from the north. The possible entry columns are numbered starting with one at the left. Then come the rows of the direction instructions. Each grid will have at least one and at most 10 rows and columns of instructions. The lines of instructions contain only the characters `N`, `S`, `E`, or `W` with no blanks. The end of input is indicated by a row containing `0 0 0`.

For each grid in the input there is one line of output. Either the robot follows a certain number of instructions and exits the grid on any one the four sides or else the robot follows the instructions on a certain number of locations once, and then the instructions on some number of locations repeatedly. The sample input below corresponds to the two grids above and illustrates the two forms of output. The word

"step" is always immediately followed by "(s)" whether or not the number before it is 1.

Example input:

```
3 6 5
NEESWE
WWWESS
SNWWWW
4 5 1
SESWE
EESNW
NWEEN
EWSEN
0 0 0
```

Example output:

```
10 step(s) to exit
3 step(s) before a loop of 8 step(s)
```

```
3 6 5
NEESWE
WWWESS
SNWWWW
4 5 1
SESWE
EESNW
NWEEN
EWSEN
1 6 1
EEEEWN
3 3 2
ENW
SSS
EWE
1 1 1
W
1 3 3
SWE
1 4 2
ESWW
2 1 1
S
S
3 4 2
ESWW
WWEE
NNNN
3 4 2
ESWW
WEEE
NNNN
3 4 2
ESWW
WESE
NSWN
3 4 2
ESNW
WSNE
NENN
4 4 3
SWWE
SWWW
SESN
ENEN
10 10 10
SWWWWWWWWW
EEEEEEEESN
SWWWWWWWWN
EEEEEEEESN
SWWWWWWWWN
EEEEEEEESN
SWWWWWWWWN
EEEEEEEESN
SWWWWWWWWN
EEEEEEEEEN
10 10 10
SWWWWWWWWW
EEEEEEEEES
SWWWWWWWWW
EEEEEEEEES
```

```
SWWWWWWWW
EEEEEEEES
SWWWWWWWW
EEEEEEEES
SWWWWWWWW
EEEEEEEES
0 0 0
```

```
10 step(s) to exit
3 step(s) before a loop of 8 step(s)
3 step(s) before a loop of 2 step(s)
1 step(s) to exit
1 step(s) to exit
1 step(s) to exit
1 step(s) to exit
2 step(s) to exit
3 step(s) to exit
4 step(s) to exit
5 step(s) to exit
6 step(s) to exit
3 step(s) before a loop of 12 step(s)
0 step(s) before a loop of 100 step(s)
100 step(s) to exit
```

Notes to Judges

Read the *Error Messages* and *Notes to Teams* right now. ... Done? OK.

The judges' diskette contains one subdirectory for each problem, which includes the description in HTML, the source file, the input and output files, and either an MS-DOS executable or Java bytecode files. It contains a subdirectory called `NOTES` that includes the HTML source for these notes. It also contains a subdirectory called `JUDGE` that includes enhanced versions of the judging utilities used in the last two contests. If you do not already have a tried-and-true method for judging, you might want to take a look at them. Instructions are provided in *Using the Judging Utilities*, and are available in text form in the `read.me` file.

Regardless of what judging method you use, remember the following (the included utilities take care of these details for you):

- If a program is correct, the team's output file will match the correct output file *exactly*. If the match is not exact, you will have to do a visual inspection to tell whether the problem is a wrong answer or a presentation error.
- Remove a team's diskette or write-protect it before judging to ensure that nothing is written to it.
- Always copy a fresh set of correct input and output files before judging a run, because teams' programs have been known to trash files.

As in last year's contest,

- problem solutions are unique and must be formatted exactly, so output can be judged using a file comparison utility,
- all problems are judged with one test file (which of course will include multiple test cases), and
- all input files have sentinels that signal the end of the input, so it is not necessary for teams to detect end-of-file. (End-of-file handling differs between languages and sometimes between different compilers for the same language. It can cause problems for teams using tools that they're not used to.)

We think the easiest problems are *Loansome Borrower* and *Automatic Editing*, followed by *Robot Motion* and *Myacm Triangles*, and then *Find the Winning Move* and *Exchange Rates*. We don't think any of the problems are exceptionally hard. We expect strong teams to solve all six problems, and most teams to solve one or two.

Andy (the toolsmith) wrote *Automatic Editing*, *Robot Motion*, and *Myacm Triangles*. Eric (the webmaster) wrote *Find the Winning Move* and *Exchange Rates*. I wrote *Loansome Borrower*.

Eric will be your regional contact during the contest. If you have any questions or corrections you can send him email at `ericshade@mail.smsu.edu`, or if it's an emergency you can phone him at 417-836-4944. Also check the Updates section of the web site periodically; if any files need to be changed, corrections will be posted there.

John Cigas
Regional Chief Judge: Editor

Rockhurst University