Problems Index Mon Oct 13 07:26:33 EDT 2014

BOSPRE 2014 PROBLEMS
------ ---- --------

The problems are in approximate order of difficulty,
easiest first.

problems/binshuffle
Decks for the virtually inclined.
Boston Preliminary 2014

problems/shopsign
Communication helper.
Boston Preliminary 2014

problems/simplefsm
Recognizing when you are synchronized.
Boston Preliminary 2014

problems/gamedraw
Without a picture, its nothing!
Boston Preliminary 2014

problems/thue
What a strange little language.
Boston Preliminary 2014

problems/grid
Where, oh where, are my whatevers.
Boston Preliminary 2014

problems/unitcalc
What a scientific calculator should be.
Boston Preliminary 2014

problems/markov
What is the future of my state?
Boston Preliminary 2014

Binary Shuffle
------ -------

You've been hired by 'Giants Gaming Emporium' to help shuffle card decks, but, understand, 'Giant's' is a virtual gaming emporium. So its all in a computer.

The process is simple: you have a (virtual) card deck and you go to a (virtual) black box and press a (virtual) button and out pops a binary number, which instructs you on how to shuffle the deck.

Specifically, you read the bits from left to right, and if a bit is 0 you take the next card from the BACK of the deck, but if it is 1 you take the next card from the FRONT of the deck.

So you can read the binary number '1001100' as

front back back front front back back

and applied to the deck of 7 Tarot cards:

21 - World
20 - Angle
19 - Sun
18 - Moon
17 - Star
16 - Tower
15 - Devil

gives the shuffled deck:

21 - World
15 - Devil
16 - Tower
20 - Angle
19 - Sun
17 - Star
18 - Moon

Special Rules
------- -----

When you run out of cards, stop, even if there are more binary digits.

If you run out of binary digits, go back to the BEGINNING of the binary number. Its as if you used an unbounded number of copies of the binary number concatenated to each other. E.g., '101' is equivalent to '101101101...'.

Input

For each of several test cases, a line containing just a test case name, followed by a line containing the binary number, followed by up to 100 lines each containing the name of one card, followed by a line containing just '.'

Input ends with an end of file.

No line is longer than 80 characters.

Output

For each test case, a copy of the input except that the order of the cards is their order AFTER the deck has been shuffled.

```
Sample Input
------ -----

-- SAMPLE 1 --
1001100
21 - World
20 - Angle
19 - Sun
18 - Moon
17 - Star
16 - Tower
15 - Devil
.

[IMPORTANTLY there are more samples in sample.in]

[BE SURE your program does ALL the samples correctly
 before you submit.]

Sample Output
------ ------

-- SAMPLE 1 --
1001100
21 - World
15 - Devil
16 - Tower
20 - Angle
19 - Sun
17 - Star
18 - Moon
.

[The output for sample.in is in sample.test]
```

```
File:      binshuffle.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Wed Oct 15 07:23:59 EDT 2014

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2014/10/15 11:25:07 $
    $RCSfile: binshuffle.txt,v $
    $Revision: 1.6 $
```

Shop Sign

Sparky, the proprietor of Sparky's Fat and Fun Eatery (she first named it 'Slim and Slick', but that did not work out), has trouble making up signs for the front of her establishment. She has a box of magnetic characters that she sticks on a sign board out in front. But she often runs out of some character.

So to help her you are to write a program that given the contents of a box of characters and several proposed messages to put on the sign, tells her which messages will succeed and which will run out of characters.

For example, given the box contents

```
AAABCCCDDDEEFGHIILLMMNNOOPQRSSSSTTTTUUVWXYY,,!!
```

and the following proposed messages:

```
HOT DOGS AND BUNS!
HOT DOGS AND SANDWICHES!
```

you propose to output:

```
HOT DOGS AND BUNS! - succeed
HOT DOGS AND SANDWICHES! - too few H's
```

The maker of magnetic characters only makes letters, digits, and the following special characters:

```
!@#$%&*():;",./?
```

So, for example, characters such as [and] CANNOT be in the box or in a proposed message.

Whitespace, of course, is free: there are no spaces in the box and none are needed.

Sparky does not plan to put more than one message at a time on the board, so each message is evaluated independently of the other messages.

Input

For each of several test cases, a line containing just the test case name, followed by a line containing the characters in the box (usually there are duplicates), followed by any number of lines each containing a proposed message, followed by a line containing just '.'.

Input ends with an end of file.

No line is longer than 80 characters.

Output

For each test case, a copy of the input, with additions to the proposed message lines of the forms

' - succeed'	If the message would succeed.
' - too few X's'	If the message would fail because there are too few X's in the box, where X is some character.

Be sure you include the single spaces SURROUNDING the '-' in the additions.

If a message would run out of several different characters, output any one.

```
Sample Input
------ -----

-- SAMPLE 1 --
AAABCCCDDDEEFGHIILLMMNNOOPQRSSSSTTTTUUVWXYY,,!!
HOT DOGS AND BUNS!
HOT DOGS AND SANDWICHES!
ON VACATION
ON VACATION TILL JULY
ARE YOU HUNGRY?
ALL YOU CAN EAT!
.
-- SAMPLE 2 --
aaBbCcDdeeFfGHhgiiMmNnooPpQqRrSSssTuVvWwYy$.&00123456789
Good Buns $1.00
Hot Dogs $5.00
Buns $1.00 Dogs $2.95
Dogs w Buns $1.45
Dogs and Buns $1.45
Dogs & Buns $1.45
Dogs & Buns $2.99
.
-- SAMPLE 3 --
!@#$%&*():;",./?
!*#
!*#*
($)/@.%
.
```

```
Sample Output
------ ------

-- SAMPLE 1 --
AAABCCCDDDEEFGHIILLMMNNOOPQRSSSSTTTTUUVWXYY,,!!
HOT DOGS AND BUNS! - succeed
HOT DOGS AND SANDWICHES! - too few H's
ON VACATION - succeed
ON VACATION TILL JULY - too few J's
ARE YOU HUNGRY? - too few R's
ALL YOU CAN EAT! - succeed
.
-- SAMPLE 2 --
aaBbCcDdeeFfGHhgiiMmNnooPpQqRrSSssTuVvWwYy$.&00123456789
Good Buns $1.00 - succeed
Hot Dogs $5.00 - too few t's
Buns $1.00 Dogs $2.95 - too few $'s
Dogs w Buns $1.45 - succeed
Dogs and Buns $1.45 - too few n's
Dogs & Buns $1.45 - succeed
Dogs & Buns $2.99 - too few 9's
.
-- SAMPLE 3 --
!@#$%&*():;",./?
!*# - succeed
!*#* - too few *'s
($)/@.% - succeed
.
```

Note:

The following would also be correct because the proposed message is short more than one character:

ON VACATION TILL JULY - too few L’s
ARE YOU HUNGRY? - too few ?’s
Buns $1.00 Dogs $2.95 - too few .’s

File: shopsign.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Mon Oct 13 06:46:29 EDT 2014

RCS Info (may not be true date or author):

$Author: walton $
$Date: 2014/10/13 10:48:29 $
$RCSfile: shopsign.txt,v $
$Revision: 1.10 $

Simple FSM
------ ---

Your company wants to implement a very simple Finite State Machine (FSM) to recognize certain patterns in an input bit stream. For example, they want to know when the last 7 bits are ‘1111110’. This bit sequence is used in a ‘flag’ that separates data blocks, where the data has been modified so that the flag bit sequence can never occur within a data block.

You have been asked to write a basic FSM simulator. Input is a string of bits. States are labeled with upper case letters or the special characters ‘$’ and ‘*’.

A simple example FSM description is:

```
$ $ A
A * A
* $ A
```

which describes an FSM that is in state * when the last 2 bits read are ‘10’.

The 3 lines of this FSM description say:

when in state $, on reading a 0 go to state $,
but on reading a 1 go to state A

when in state A, on reading a 0 go to state *,
but on reading a 1 go to state A

when in state *, on reading a 0 go to state $,
but on reading a 1 go to state A

For each state you are given two successor states, one to go to if the next bit input is ‘0’, and one to go to if the next bit input is ‘1’. The FSM starts in state ‘$’, and stops when there are no more bits to read.

If the FSM does what it is supposed to, it will be in state ‘*’ if and only if the last several bits read are the pattern sought.

To see how this FSM executes when inputting the binary string ‘0100110011100111110’, write the sequence of states that the machine is in so that each state is underneath the next bit to be read:

```
        0100110011100111100
        $$A*$AA*$AAA*$AAAA*$
```

This means that the FSM:

starts in state ‘$’
goes to state ‘$’ upon reading the first ‘0’
goes to state ‘A’ upon reading the first ‘1’
goes to state ‘*’ upon reading the second ‘0’
goes to state ‘$’ upon reading the third ‘0’
. . . . . .
goes to state ‘A’ upon reading the last ‘1’
goes to state ‘*’ upon reading the next to last ‘0’
goes to state ‘$’ upon reading the last ‘0’
stops when there is no binary digit left to read

Similarly the FSM:

```
$ $ A
A $ B
B $ C
C * C
* $ A
```

which is intended to recognize the pattern ‘1110’ executes as follows on the same input string:

```
        0100110011100111100
        $$A$$AB$$ABC*$ABCC*$
```

Input

For each of several test cases, the following in order:

    a line containing just the test case name
    lines containing the FSM description
    a line containing just '.'
    one or more input lines each containing
        a binary string
    a line containing just '.'

No line is longer than 80 characters.

Input ends with an end of file.

An FSM description consists of 'state description lines' each of the form:

        s z n

which says:

    when in state s, on reading a 0 go to state z,
                 but on reading a 1 go to state n

There are 28 FSM states ($, A, B, ..., X, Y, Z, *), but unused states have no state description line.

Each binary string contains just '0's and '1's and is processed independently of the other binary strings.

Output

For each test case, first an exact copy of the test case name line, and then for each binary string input line two lines:

    (1) an exact copy of the binary string input line
    (2) the state sequence of the FSM execution for
        the input binary string, as described above;
        here the state that the FSM is in just before
        reading a binary digit is placed directly under
        the binary digit

Note there is no whitespace in either of these two lines.

```
Sample Input
------ -----

-- RECOGNIZE ‘10’ --
$ $ A
A * A
* $ A
.
0000
1111
0100
010010100
0100110011100111100
.
-- RECOGNIZE ‘1110’ --
$ $ A
A $ B
B $ C
C * C
* $ A
.
1111
1100
1110
1011
101010
010010100
0100110011100111100
0101101110111100
.
```

```
Sample Output
------ ------

-- RECOGNIZE ‘10’ --
0000
$$$$$
1111
$AAAA
0100
$$A*$
010010100
$$A*$A*A*$
0100110011100111100
$$A*$AA*$AAA*$AAAA*$
-- RECOGNIZE ‘1110’ --
1111
$ABCC
1100
$AB$$
1110
$ABC*
1011
$A$AB
101010
$A$A$A$
010010100
$$A$$A$A$$
0100110011100111100
$$A$$AB$$ABC*$ABCC*$
0101101110111100
$$A$AB$ABC*ABCC*$
```

Extra Notes (not relevant to solving problem):
----- ----- --- -------- -- ------- -------

The synchronous High Level Data Link Control (HDLC) protocol for communications via synchronized bit streams uses ‘01111110’ as a ‘flag’. Successive flags are used to synchronize clocks and bracket data blocks, which are altered so they cannot contain flags. This is done simply by inserting a 0 bit whenever 5 successive 1 bits have occurred in the data, and removing the 0 bit when ‘111110’ is received in data. The flag contains 6 successive 1’s; 7 successive 1’s is considered to be a transmission error.

All FSM’s have the same structure except for input/output. As an example of a slightly more complicated input/output structure, let the FSM output a character when making a transition from one state to another. So the description line ‘s z n’ becomes ‘s z/Z n/N’ where Z and N are characters output when the next state becomes z or n, and the output is optional, so the ‘/Z’ and ‘/N’ are optional. Using this you can describe an FSM that will insert or remove the 0 bit in HDLC data.

A more substantial modification replaces the input string with a tape that can move backward or forward one position, so instead of outputting a character the machine writes a new digit at the current position and then moves the tape either backward or forward one position. If we also replace the set {0,1} of two digits by an arbitrary finite set of ‘symbols’, we have what is called a ‘Turing Machine’.

File: simplefsm.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Tue Oct 7 00:10:16 EDT 2014

RCS Info (may not be true date or author):

$Author: walton $
$Date: 2014/10/07 04:10:44 $
$RCSfile: simplefsm.txt,v $
$Revision: 1.11 $

Game Draw
---- ----

You and your friends are making your first computer game and its fallen to you to draw the pictures, which consist of simple polygons which are to be filled in with some color. You are restricting things so the polygons have only horizontal and vertical edges, the pixels that must be set have integer coordinates, and the polygon vertices also have integer coordinates.

Also to make debugging easier you are, for the moment, representing the picture as a grid of characters on the screen, where '.' denotes a black pixel, and various other characters denote colors.

Suppose we start with a 7x14 grid and then draw a polygon with vertices which we label 1, 2, 3, 4, 5, 6, 7, 8, after which we fill in the polygon with the color 'X'. We get:

```
    ..2.3....6.7..      ..XXX....XXX..
    ..............      ..XXX....XXX..
    ....4....5....      ..XXXXXXXXX..
    ..............      ..XXXXXXXXX..
    ..............      ..XXXXXXXXX..
    ..1........8..      ..XXXXXXXXX..
    ..............      ..............
```

Then if we add two polygons, the first with vertices labeled 1, 2, 3, 4 and the second with vertices labelled 5, 6, 7, 8, and both with color '+', we get

```
    ..XXX....XXX..      ..XXX....XXX..
    ..XXX....XXX..      ..XXX....XXX..
    ..XXXXXXXXX..      ..XXXXXXXXX..
    ..XX12XX67XX..      ..XX++XX++XX..
    ..XX43XX58XX..      ..XX++XX++XX..
    ..XXXXXXXXX..      ..XXXXXXXXX..
    ..............      ..............
```

where these last two polygon's overlay the first polygon.

In general the grid is a rectangle of N x M characters, N lines of M characters each, with (0,0) in the LOWER LEFT and (M-1,N-1) in the UPPER RIGHT. So the vertices of the first polygon are

(2,1) (2,6) (4,6) (4,4) (9,4) (9,6) (11,6) (11,1)

and of the second two polygons are

(4,3) (5,3) (5,2) (4,2)
and
(8,2) (8,3) (9,3) (9,2)

Input

For each of several test cases, first a line containing just a test case name, followed by a line containing:

N M K

where the grid has N lines of M characters and there are K polygons, followed by K polygon description lines each containing:

C V x1 y1 x2 y2 ... xV yV

where C is the character representing the polygon color, V is the number of polygon vertices, and the vertices are (x1,y1), (x2,y2), ..., (xV,yV) IN CLOCKWISE ORDER.

No two edges of the same polygon intersect except at common vertices. All vertices are inside the grid. No edge is 0-length. No two consecutive edges are parallel. No two OVERLAPPING polygons have the same color.

No line is longer than 80 characters, so there can be no more than 20 vertices in a polygon.

Input ends with an end of file.

2 <= N <= 60 2 <= M <= 80 1 <= K <= 40

Output

For each test case, first an exact copy of the test case name line, and then the test case grid consisting of N lines each with exactly M characters, none of which are whitespace.

The grid is initialized to all '.''s before any polygons are drawn. Then the polygons are drawn IN ORDER, so each may overlay previously drawn polygons. After all polygons are drawn the grid is output.

Sample Input
------ -----

```
-- BOX ---
4 14 1
X 4 2 1 2 2 11 2 11 1
-- BOX WITH EARS ---
6 14 1
X 8 2 1 2 4 3 4 3 2 10 2 10 4 11 4 11 1
-- BOX WITH EYES ---
6 14 3
X 4 2 1 2 4 11 4 11 1
+ 4 4 2 4 3 5 3 5 2
+ 4 8 2 8 3 9 3 9 2
-- BOX WITH EARS, EYES, AND NOSE ---
9 14 3
X 8 2 1 2 7 4 7 4 5 9 5 9 7 11 7 11 1
+ 4 4 3 4 4 9 4 9 3
* 4 6 0 6 4 7 4 7 0
```

[IMPORTANTLY there are more samples in sample.in]

[BE SURE your program does ALL the samples correctly before you submit.]

```
Sample Output
------ ------

-- BOX ---
..............
..XXXXXXXXX..
..XXXXXXXXX..
..............
-- BOX WITH EARS ---
..............
..XX......XX..
..XX......XX..
..XXXXXXXXX..
..XXXXXXXXX..
..............
-- BOX WITH EYES ---
..............
..XXXXXXXXX..
..XX++XX++XX..
..XX++XX++XX..
..XXXXXXXXX..
..............
-- BOX WITH EARS, EYES, AND NOSE ---
..............
..XXX....XXX..
..XXX....XXX..
..XXXXXXXXX..
..XX++**++XX..
..XX++**++XX..
..XXXX**XXXX..
..XXXX**XXXX..
......**......

[Output for sample.in is in sample.test]
```

```
File:      gamedraw.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Tue Oct  7 01:41:22 EDT 2014


The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2014/10/13 11:15:29 $
    $RCSfile: gamedraw.txt,v $
    $Revision: 1.18 $
```

Thue

THUE is an esoteric programming language invented by John Colagioia in 2000. It is based on a non-deterministic rewriting system named after the Norwegian mathematician Axel Thue, and has been described as one of the simplest possible ways to construe constraint-based programming. It is capable of simulating a Turing machine.

You have been asked to implement THUE.

Syntax

```
    symbol ::= graphic ASCII character
    string ::= symbol*
    rule ::= left-hand-side single-space
             ‘::=’ single-space right-hand-side
             end-of-line
    left-hand-side ::= non-empty string other than ‘::=’
    right-hand-side ::= string
    input-rule ::= left-hand-side single-space
                   ‘::=’ single-space ‘:::’ end-of-line
    output-rule ::= left-hand-side single-space
                    ‘::=’ single-space ‘~’ string
                    end-of-line
    memory ::= string
    input ::= string other than ‘!!!’
    input-line ::= input end-of-line
    program ::=
        rule*
        ‘::=’ end-of-line
        memory end-of-line
        input-line*
        ‘!!!’ end-of-line
```

Notes:

```
    ‘Graphic’ means non-white-space, non-control.
    The only non-graphic characters in any line are
        the two single-spaces in rules.
    ‘::=’ denotes a 3-character string as opposed to the
          syntax equation meta-symbol ::=
```

Execution

Memory is searched and the sequence of rules is searched until a memory substring and rule are found such that the memory substring matches exactly the rule left-hand-side. Then the memory substring is replaced by the rule right-hand-side. The searches are non-deterministic (search order does not matter). Execution stops when no match can be found.

Two kinds of rules are special.

For an input-rule, the right-hand-side ::: is replaced by the input string from the next input-line. Unless the last line input was the program ending !!! line, in which case that line is RE-READ.

For an output-rule, the right-hand-side without its initial ~ is output and then replaced in the rule by the empty string (so the left-hand-side is deleted from memory). If what would be output is the empty string (the right-hand-side was ~ by itself), an end-of-line is output; otherwise no end-of-line is output.

Input

For each of several test cases, first a line containing just the test case name, then a program.

Input ends with an end of file.

The maximum line length is 80 characters, and the maximum number of rules in a program is 100.

Output

For each test case, first a line containing an exact copy of the test case name input line, then the test case program output, and LASTLY A SINGLE BLANK LINE.

Input will be such that no output line is longer than 80 characters, memory will never be longer than 100 characters (symbols), output will be unique, output will end with an end-of-line, and the program terminating !!! line will be read. Assuming, of course, that you execute the program correctly.

Sample Input
------ -----

```
-- HELLO WORLD --
a ::= ~Hello_World!
** ::= ~
$$ ::= :::
::=
$*a*$
!!!
-- INCREMENT BINARY NUMBERS --
INPUT ::= :::
1_ ::= 1++
0_ ::= *1
01++ ::= *10
11++ ::= 1++0
_1++ ::= _*10
0* ::= *0
1* ::= *1
_*0 ::= _Z*
Z ::= ~0
_*1 ::= _ONE*
ONE ::= ~1
_*! ::= $EOL$
EOL ::= ~
$$ ::= _INPUT_!
::=
_INPUT_!
0
1
00
01
10
11
1001111
!!!
```

```
Sample Output
------ ------

-- HELLO WORLD --
Hello_World!

-- INCREMENT BINARY NUMBERS --
1
10
01
10
11
100
1010000

[Output ends with a single blank line.]


File:      thue.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Wed Oct 15 07:26:53 EDT 2014

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2014/10/15 11:27:40 $
    $RCSfile: thue.txt,v $
    $Revision: 1.5 $
```

Grid

A ‘grid’ (in the sense described here) is a geometric data base that stores a set of ‘points’ in a fashion that makes it easy to look up which points are near a query point. It is possible to represent geometric objects as points and use a grid to look up which objects are near a query point.

The grids we are concerned with are binary trees in which each node is associated with a rectilinear region in (cx, cy, cz, r) 4-dimensional space. Here (cx, cy, cz) is the center of a 3-dimensional sphere of radius r, so the 4-dimensional point represents a 3-sphere.

Each child is associated with a subregion of its parent’s region, so that the regions of the two children of a node are disjoint and their union is the parent region. The child regions are made by intersecting the parent region with the halfspaces d <= v and d > v, for some real number v and coordinate d. Here d may be cx, cy, cz, or r.

Leaves of the grid contain sets of 4-dimensional points. In this problem we shall not be concerned with these sets.

You are given a grid and a set of queries. Each query consists of a point p and a distance R, and the answer is the subtree of nodes whose regions contain a 4D point (cx,cy,cz,r) representing a sphere that is at most distance R from p, i.e., ||p-(cx,cy,cz)|| <= r+R.

If you think about it you will find that the nodes of the desired subtree are those whose 4D regions intersect a 4D cone whose axis is parallel to the r-direction and whose r = C slice is a 3-sphere of radius R+C and center p.

Input

For each of several test cases, a line containing just the test case name, followed a line containing

```
cxmin cxmax cymin cymax czmin czmax rmin rmax
```

followed by lines containing a representation of the grid, followed by one or more query lines, followed by a line containing just a ‘*’.

The region of the grid root is

```
cxmin < cx <= cxmax
cymin < cy <= cymax
czmin < cz <= czmax
 rmin <  r <=  rmax
```

The grid representation has the syntax:

```
<grid> ::= # | d/v(<grid>,<grid>)
d ::= cx | cy | cz | r
v ::= a real number
```

Here <grid> represents a tree node, # represents a leaf, the first <grid> in d/v(<grid>,<grid>) represents the left child, and the second <grid> represents the right child. The left child region is made by intersecting the parent region with the halfspace d <= v, and the right child region is made by intersecting with d > v. Whitespace, including line feeds, is allowed only after ‘(’ and ‘,’.

A query is represented by a single line containing

```
px py pz R
```

where (px, py, pz) is the query point and R the query distance.

No grid will have more than 127 nodes. There will be no more than 1000 queries among all the test cases taken together. For each test case and query:

```
cxmin < cxmax
cymin < cymax
czmin < czmax
0 <= rmin < rmax
0 <= R
```

All input numbers are in the range [-1000, +1000] and have at most 3 decimal places.

Input ends with an end of file.

Output

For each test case, first a line containing an exact copy of the test case name line, and then for each query p, R, exactly one line (possibly very long) containing a representation of the subtree of grid nodes whose regions contain points that represent spheres within distance R of point p. This subtree is represented by the syntax:

```
<subtree> ::= # | * | (<subtree>,<subtree>)
```

and is a pruned version of the grid made by omitting the ‘d/v’s and representing omitted children by ‘*’. Also there may not be ANY whitespace within the query output line.

The input will be such that the output is unambiguous.

```
Sample Input
------ -----

-- SAMPLE 1 --
-10 10 -10 10 -10 10 0 2
cx/0(cx/-5(#,#),cx/5(#,#))
-10 0 0 0
-5 0 0 0
-2.5 0 0 0
-1 0 0 0
+1 0 0 0
+2.5 0 0 0
+5 0 0 0
+10 0 0 0
*
-- SAMPLE 2 --
-10 10 -10 10 -10 10 0 2
cx/5(cy/5(cz/5(r/1(#,#),
               r/1(#,#)),
          cz/5(r/1(#,#),
               r/1(#,#))),
     cy/5(cz/5(r/1(#,#),
               r/1(#,#)),
          cz/5(r/1(#,#),
               r/1(#,#))))
-13 -5 5 0.1
-12 -5 5 0.1
1 -5 5 0.1
1 -5 3 0.1
1 1 1 3.1
1 1 1 2.1
*
```

```
Sample Output
------ ------

-- SAMPLE 1 --
((#,*),*)
((#,#),*)
((*,#),*)
((*,#),(#,*))
((*,#),(#,*))
(*,(#,*))
(*,(#,#))
(*,(*,#))
-- SAMPLE 2 --
*
((((*,#),(*,#)),*),*)
((((#,#),(#,#)),*),*)
((((#,#),(*,#)),*),*)
((((#,#),(#,#)),((#,#),*)),(((#,#),*),*))
((((#,#),(*,#)),((*,#),*)),(((*,#),*),*))

Remarks
-------

In real applications, the spheres bound more complex
objects which are inspected further once it is determin-
ed that their bounding spheres are close enough to the
observation point.  Also, p and part of R may themselves
represent a sphere bounding a complex object.

Lastly, in a real application, the algorithm to select
grid nodes in the subtree may be simplified to be faster
at the cost of including some nodes that do not belong.
But you must NOT simplify for this problem, which
requires precise computation of intersections.
```

```
Reference
---------

Algorithms & Data Structures, with applications to
graphics and geometry, Jurg Nievergelt and Klaus H.
Hinrichs, 23.5.


File:      grid.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Sun Oct  5 05:53:46 EDT 2014

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2014/10/05 09:56:22 $
    $RCSfile: grid.txt,v $
    $Revision: 1.9 $
```

Unit Calculator

You have been asked to write a program that will perform context dependent unit conversions. For example, if you are selling apples and oranges and want to know how many oranges you must sell to generate as much revenue as selling 1 apple, you could give the program:

```
        12 apples = 4.95 dollars
        12 oranges = 3.25 dollars
        1 apples = ? oranges
```

to which the answer is:

```
        1 apples = 1.523 oranges
```

Similarly if you want to know how many kilowatts of energy it takes to heat 1 gallon of water 60 degrees Fahrenheit in 1 minute, you could give the program:

```
        1 BTUs = 1 degrees-F pounds
        1 BTUs = 1055 joules
        1 joules = 1 watts seconds
        1000 watts = 1 kilowatts
        60 seconds = 1 minutes
        1 gallons = 8.3454 pounds
        60 degrees-F gallons = ? kilowatts minutes
```

to which the answer is

```
        60 degrees-F gallons = 8.804 kilowatts minutes
```

Notice the first problem makes apples and oranges equivalent to dollars, which only works in a particular context. The second problem makes 1 gallon equal to 8.3454 pounds, and 1 BTU equal to 1 degrees-F per pound, and these equations work only for water. So our problems are context sensitive.

To simplify things we always use the plural for unit names, thus writing such syntactic barbarisms as ‘1 apples’. The lexemes are

```
    <number> ::= any number readable as floating point
    <unit-name> ::= sequence of graphic characters
                    that begins with a letter
    <special> ::= / | = | ? | .
```

A ‘graphic’ character is a character that prints a mark. Lexemes are always separated from each other by white-space.

There is one statement per input line. The statement syntax is:

```
    <units> ::= <unit-name>* <divisor>*
    <divisor> ::= / <unit-name>
    <quantity> ::= <number> <units>
    <equivalence> ::= <quantity> = <quantity>
    <query> ::= <quantity> = ? <units>
    <end> ::= .
    <statement> ::= <equivalence> | <query> | <end>
```

Input

For each of several test cases, a line containing just a test case name, followed by lines containing equivalences used in the calculation, followed by lines containing queries, followed by a line containing just ‘.’. Note that there can be more than one query, and lines and unit names can be long.

All input is lexically and syntactically legal. All numbers input are > 0.

There are at most 100 distinct <unit-name>s in a test case and at most 100 <equivalence>s. In all test cases there will be at most 1,000 <query>s.

Input ends with an end of file.

Output

For each test case, first a line containing an exact copy of the test case name line, and for each input query line an exact copy of that line but with the ? replaced by a number with exactly 3 decimal places. Input will be such that every query will have a unique answer.

Sample Input
------ -----

```
-- APPLES AND ORANGES --
12 apples = 4.95 dollars
12 oranges = 3.25 dollars
1 apples = ? oranges
.
-- HEATING WATER --
1 BTUs = 1 degrees-F pounds
1 BTUs = 1055 joules
1 joules = 1 watts seconds
1000 watts = 1 kilowatts
60 seconds = 1 minutes
1 gallons = 8.3454 pounds
1 BTUs = ? kilowatts minutes
60 degrees-F gallons = ? BTUs
60 degrees-F gallons = ? kilowatts minutes
.
-- STRANGE --
5 / X = 7
3 X = ?
.
```

```
Sample Output
------ ------

-- APPLES AND ORANGES --
1 apples = 1.523 oranges
-- HEATING WATER --
1 BTUs = 0.018 kilowatts minutes
60 degrees-F gallons = 500.724 BTUs
60 degrees-F gallons = 8.804 kilowatts minutes
-- STRANGE --
3 X = 2.143


File:      unitcalc.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Wed Oct 15 07:28:20 EDT 2014

The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2014/10/15 11:29:46 $
    $RCSfile: unitcalc.txt,v $
    $Revision: 1.10 $
```

Markov Recurrence

A finite Markov Chain is a finite set of N states S1, S2, ..., SN and a matrix p[i,j] of probabilities, 1 <= i,j <= N, such that if the 'system' is in state Si, the next state of the system will be Sj with probability p[i,j]. It is required that

$$0 \le p[i,j] \le 1$$
$$\sum_{1 \le j \le N} p[i,j] = 1$$

Thus the 'transition' of the system from Si to Sj has probability p[i,j].

If a state sequence starts from Si it may or may not return to Si; such a return is called a 'recurrence'. Let f[i,t] be the probability that the sequence returns to Si for the FIRST time after the t'th transition (so f[i,1] == p[i,i]). Here t => 1 is thought of as 'time'. Then

$$f[i] = \sum_{1 \le t} f[i,t]$$

is the probability that the system returns to Si at some future time (the probability that Si ever recurs).

A state Si is said to be persistent if f[i] == 1, and to be transient if f[i] < 1.

A state Si is said to be no-return if f[i] == 0.

A state Si is said to be periodic if it is NOT no-return and there is an integer s > 1 such that f[i,t] == 0 if t mod s != 0. The largest such s is the period of Si.

A state Si that is NEITHER no-return or periodic is said to be aperiodic.

You have been asked to compute f[i] for the states Si of a Markov Chain and determine if Si is transient or persistent and if Si is no-return, periodic, or aperiodic. In the periodic case you are to determine the period.

Input

For each of several test cases, first a line containing just the test case name, then a line containing just N, and then N lines containing the probabilities in the layout

```
p[1,1] p[1,2] ... p[1,N]
p[2,1] p[2,2] ... p[2,N]
. . . . . .
p[N,1] p[N,2] ... p[N,N]
```

1 <= N <= 100, 0 <= p[i,j] <= 1, and for each i, sum p[i,j] over all j = 1.

Input probabilities may have many decimal places and the lines containing them may be long. Double precision floating point will suffice for input and computation.

Input ends with an end of file.

Output

For each test case, the first a line containing an exact copy of the test case name input line, then N lines each with the format:

```
f[#] = #.### X Y
```

where # denotes a digit, X is either ‘persistent’ or ‘transient’, Y is either ‘no-return’, ‘period #’ or ‘aperiodic’.

The only whitespace in the output are 4 single spaces, 2 surrounding the =, 1 before X, and 1 before Y. The N lines are in order of increasing f[#] index, i.e., f[1], f[2], ..., f[N].

The input will be such that a state i will be persistent if and only if f[i] >= 0.9995, and there will be no ambiguous cases.

WARNING: f[i] < 0.0005 does NOT mean the state is no-return. You must use a calculation not involving f[i] to determine whether a state is no-return (use your period calculation).

Sample Input

```
-- SAMPLE 1 --
2
0 1
1 0
-- SAMPLE 2 --
2
0.5 0.5
0 1
-- SAMPLE 3 --
2
0 1
0 1
```

Sample Output

```
-- SAMPLE 1 --
f[1] = 1.000 persistent period 2
f[2] = 1.000 persistent period 2
-- SAMPLE 2 --
f[1] = 0.500 transient aperiodic
f[2] = 1.000 persistent aperiodic
-- SAMPLE 3 --
f[1] = 0.000 transient no-return
f[2] = 1.000 persistent aperiodic
```

Note

For finite markov chains, persistent aperiodic states are called ‘ergodic’. For infinite markov chains, persistent aperiodic states with finite expected time of return are called ‘ergodic’, but there are also persistent aperiodic states with infinite expected time of return which called ‘null states’.

```
File:      markov.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Wed Oct 15 07:30:03 EDT 2014


The authors have placed this file in the public domain;
they make no warranty and accept no liability for this
file.

RCS Info (may not be true date or author):

    $Author: walton $
    $Date: 2014/10/15 11:30:37 $
    $RCSfile: markov.txt,v $
    $Revision: 1.9 $
```