Problems Index Tue Oct 12 08:15:19 PM EDT 2010

BOSPRE 2010 PROBLEMS
------ ---- --------

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

problems/escape
You need to get out more!
Boston Preliminary 2010

problems/lexemes
Words, words, but more than words.
Boston Preliminary 2010

problems/bezier
Curvy is grovy.
Boston Preliminary 2010

problems/congruent
Being similar is sometimes a winner.
Boston Preliminary 2010

problems/escape2
Its getting harder to get out.
Boston Preliminary 2010

problems/plans
Searching for satisfaction.
Boston Preliminary 2010

problems/penrosetiling
Pretty but different.
Boston Preliminary 2010

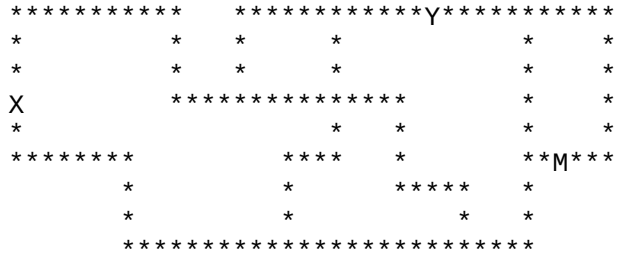
Escape From The Maze
------ ---- --- ----

You are stuck in a maze and need to escape. The maze is constructed from corridors laid out on a grid of squares. Each corridor is a straight line of squares. That is, corridors are 1 square wide and are straight between their ends. All corridors run either horizontally(East/West) or vertically(North/South).

All corridor ends are also an end or middle of another another corridor. In other words, there are no ‘dead end’s in the maze.

There is a single monster in the maze pursuing you. If you and the monster end up on the same square, the monster will eat you.

Here is an example maze:



Here X is the exit, Y is you, M is the monster.

You CANNOT see the whole maze. In each of the four directions you see one of:

W	A wall of the corridor you are on. The wall is a boundary of the square you are on.
E	The end of a corridor you are on, which is just a wall that is the boundary of a square you are NOT on.
X	The exit.
M	The monster.

You can move one square along any corridor you are in. After you move one square the monster moves one square, and then you can move again. When you move to the exit square you escape and the adventure is over. If you and the monster end up on the same square the monster eats you and the game is over. The monster never exits the maze.

The monster is a bit skittish, and as a result, if you suddenly appear in front of it, having come around a corner, it will back away from you in its next move, and not eat you. Because of this you have ‘always escape’ strategies available to you. Otherwise the monster is fairly purposeful, and will pursue you if it can see you.

You are required to implement an ‘always escapes’ strategy in a program called ‘escape’.

The Maze Program

Your program is named ‘escape’ and runs as a subprocess of the ‘maze’ program which is provided to you. The command to run this program is

```
maze escape
```

which invokes the maze program with the ‘escape’ program as its argument. Additional arguments added at the end of this command will be passed to the ‘escape’ program and may be of use for debugging.

As an introduction to this problem, you are given a sample ‘escape’ program named ‘sample_escape’, which is described in some detail below. In this program you just move randomly, so the monster eventually eats you. To see what is going on, type

```
maze sample_escape
> d1
> m1234
> f30
> f
> f
> f
> f1 1000
> b1 1
> b10
> f
> f10
```

(where the line beginning ‘> ’ is a prompt that you do NOT type). The ‘f’ command runs time forward and the ‘b’ command runs time backward: see below for details.

Escape Program Input

The ‘maze’ program writes lines to the ‘escape’ program and reads lines written by the ‘escape’ program. The lines written by ‘maze’ and read by ‘escape’ each have one of the following forms:

-...	A line beginning with ‘-’ indicates the start of a new test case.
nesw	n, e, s, and w each denote one of the characters W, E, X, M described above which tell you what you see in the North, East, South, and West directions respectively. This line has exactly 4 characters. E.g., ‘WEMX’ means there are corridors to your East, South, and West but not North, and in the corridor to the South you see the monster while in the corridor to the West you see the exit.
p...	A line beginning with the letter ‘p’ contains parameters for your ‘escape’ program. This is ONLY for debugging.

‘escape’ program input ends with an end of file, at which point the ‘escape’ program must terminate.

Escape Program Output

For each ‘nesw’ line input you must choose a direction in which to move and output it as the first character on a single line. The possibilities are ‘N’ for North, ‘E’ for East, ‘S’ for South, and ‘W’ for West. The rest of the characters on the line are ignored but may be used for debugging information.

If you make an illegal move (either a line that does not begin with ‘N’, ‘E’, ‘S’, or ‘W’ or you try to move through a wall), the programs will crash.

For debugging you may output lines beginning with ‘i’ before you output the direction line above. These lines will be printed by the ‘maze’ ‘f’ command (see below), and are otherwise ignored.

If you output a line longer than 80 characters, only the first 80 characters will be kept.

WARNING: If you are programming in C you must execute

```
fflush (stdio);
```

after writing each line to the standard output, or your output will be trapped in a buffer and never get to the maze program. In C++ the ‘endl’ IO manipulator flushes the buffer and in JAVA ‘println’ flushes the buffer, so nothing unusual needs to be done for these languages.

When you ran ‘maze sample_escape’ as indicated above, just before the maze a line appeared which has the form

```
escape-input-line >> escape-output-line
```

giving the last input line to the ‘escape’ program and the output line that program produced to cause the last move.

Your ‘escape’ program must be smart enough to escape the maze within 20,000 moves. Otherwise you will be ‘OUT OF TIME’.

Maze Program Input

A sequence of command lines each of which contain one of the following:

`-...`	A line beginning with ‘-’ indicates the start of a new test case. The line can be used to hold the name of the test case.
`m S`	This line creates a maze and starts the action. S is an unsigned integer that is the seed for a pseudo-random number generator that is used to generate the maze and set the initial positions of you, the exit, and the monster. The pseudo-random number generator is also used to make choices for the monster during game action. S must be unsigned and should not have more than 9 digits. Different values of S produce different mazes, but because its a PSEUDO-random number generator, repetitions of this command with the SAME value of S always produce the same maze. If not in debugging mode, this command runs the game action to completion and then prints a single line containing just your fate: ‘ESCAPED’, ‘EATEN’, or ‘OUT OF TIME’.

If in debugging mode the command prints the maze. Then the ‘f’ and ‘b’ commands below may be used to run the action forward and backward.

p... A line beginning with the letter ‘p’ is copied to your ‘escape’ program input, and can be used to pass parameters to the ‘escape’ program. This is for debugging only.

d1 Turns debugging mode on.
d0 Turns debugging mode off.

f N M D Used when the ‘m’ command was given in debugging mode. Runs forward N*M steps, where a step is a one square move on your part followed by a one square move of the monster. After every M steps, prints output, and then pauses D seconds, if D > 0. Running stops if you escape or are eaten or run out of time.

Unlike all other numbers input, D is floating point. It defaults to the previous value given in any ‘f’ or ‘b’ command, or to 0.25 if there was no previous value. If M is also omitted it defaults to the previous value given in any ‘f’ or ‘b’ command, or to 1 if there was no previous value. If N is omitted, it defaults to 1.

For each group of M steps the following are printed. First some blank lines to separate things from previous output. Then for every step any ‘i’ lines output by the ‘escape’ program are printed followed by a line of the format:

input >> output

giving the input line to and output line from the ‘escape’ program for the step. Next the maze is printed, followed by a status line that is often blank, followed by a line that is blank or that begins with a prompt for your next command.

This command can be terminated prematurely by typing control-C. Typing control-C at any other time terminates the ‘maze’ program.

b N M D Just like the ‘f’ command but moves backward in time instead of forward. However, does not print ‘i’ lines or ‘input >> output’ lines. Also, if going forward after going backward, these lines are not printed and the ‘escape’ program is not involved until all previously run steps have been passed.

When input is from a file, each input line is also output. When input is from a terminal, a ‘> ’ prompt is output at the beginning of each input line.

No input line may be longer than 80 characters. Input ends with an end of file (if you are typing input you can produce an end of file by typing control-D).

```
Maze Program Output
---- ------- ------

If not run from a terminal, ‘maze’ outputs a copy of its
input.  It also always outputs the information described
under the ‘m’, ‘f’, and ‘b’ commands above.


Sample Escape Program
------ ------ -------

You are given a program, ‘sample_escape’, that can be
used to see what ‘maze’ does.  ‘sample_escape’ just
executes the following:

    loop:
        read an input line
        on end of file exit program
        if line is a ‘-’ line ignore it
        if the line is an nesw line:
            pick a direction at random in which there is
                 no wall (any non-W direction)
            output a line indicating the picked
                   direction
        if a line is a ‘p’ line, echo the line inside an
           ‘i’ line, but otherwise ignore the ‘p’ line

Suggestions for running the ‘sample_escape’ program are
given above.


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

-- SAMPLE 1 --
m1234
```

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

-- SAMPLE 1 --
m1234
ESCAPED



Notes
-----

If the monster is on the same square as the exit, you
will see the monster and not the exit.  Debugging
displays will also show ‘M’ for the square and not ‘X’.

If you want to write your own version of ‘sample_escape’
for fun or profit, you may find the following pseudo-
random number generators useful:

C:      #include <stdlib.h>
        . . .
        int i = random();

C++:    #include <cstdlib>
        . . .
        int i = random();

Java:   import java.util.*;
        . . .
        static Random r = new Random (123);
        . . .
        int i = r.nextInt();
```

```
File:      escape.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Thu Oct 14 04:38:06 EDT 2010

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: 2010/10/14 08:38:42 $
    $RCSfile: escape.txt,v $
    $Revision: 1.18 $
```

Lexemes

You have been asked to scan lines of input text into lexemes.

For example, given the input line

```
    x = 5*y + "hello world";
```

you are to output

```
    |x| |=| |5|*|y| |+| |"hello world"|;|
     s w o w n o s w o w qqqqqqqqqqqq p
```

The definitions are

```
    <lexeme>     ::= <symbol>
                   | <whitespace>
                   | <operator>
                   | <number>
                   | <quoted-string>
                   | <punctuation>
                   | <illegal>
    <symbol>     ::= <letter><letter-or-digit>*
    <whitespace> ::= <single-space-character>+
    <operator>   ::= '+' | '-' | '*' | '/' | '=' | '.'
    <number>     ::= <digit>+ <fraction-option>
    <fraction-option> ::= <empty> | '.' <digit>+
    <quoted-string>   ::=
        '"' <character-representative>* '"'
    <character-representative> ::=
        <character-except-"-or-\> | '\"' | '\\'
    <punctuation> ::= ',' | '(' | ')' | ';'

    <illegal> ::=
        <any-character-that-starts-no-other-lexeme>
```

Here <x>* means zero or more <x>'s, <x>+ means one or more <x>'s, and <empty> means the empty character string.

Given a position in the input, the next lexeme is the LONGEST lexeme that can be found starting at that position. E.g., '8.1' scans as one number lexeme and does NOT include a '.' operator.

If no other lexeme can be found, the next character is a 1-character 'illegal lexeme'. Note that this produces some idiosyncratic results. For example, if you forget the closing " in a quoted string, there is no quoted string lexeme, and the " starting the string becomes a 1-character illegal lexeme. Similarly if you put an illegal character representative, such as \h, in a quoted string. To be sure you implement the above rules precisely, you should carefully check that your solution gets the Sample Output below when given the Sample Input below.

Input

For each of several test cases two lines. The first line is the test case name. The second line is the line you are to scan into lexemes.

There are NO tab characters in the input, so the only space characters in the input are single space characters and line ending line feeds. No line is longer than 80 characters (not counting line feeds).

Input ends with an end of file.

```
Output
------

For each test case, first an exact copy of the test case
name line.  Then two lines.  The first is a copy of the
input line to be scanned with '|' marks inserted at the
beginning and end and in between scanned lexemes.  The
next line has under each lexeme character a letter
giving the lexeme type.  This letter is simply the first
letter of the lexeme type name (i.e., 's' for symbol,
'o' for operator, 'i' for illegal lexeme, etc.).

Remember to test your program on the Sample Input and
be sure its output EXACTLY matches the Sample Output.
Note that numbers and symbols can be arbitrarily long,
and numbers CANNOT begin or end with '.'.  Also illegal
quoted strings are NOT recognized as quoted strings, and
their initial " is treated as an illegal lexeme.
Lastly, illegal lexemes are all 1-character lexemes,
like punctuation and operators.


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

-- SAMPLE 1 --
x  =  5y21 + 3*x+foo("hi\\n",7.8);
-- SAMPLE 2 --
7.8 7. 8 7 .8 01234567890123456789.x!!
-- SAMPLE 3 --
?"He said: \"Ha\"?\\n" + "He He\n" + "Ho"
```

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

-- SAMPLE 1 --
|x|  |=|  |5|y21| |+| |3|*|x|+|foo|(|"hi\\n"|,|7.8|)|;|
 s ww o ww n sss w o w n o s o sss p qqqqqqq p nnn p p
-- SAMPLE 2 --
|7.8| |7|.| |8| |7| |.|8| |01234567890123456789|.|x|!|!|
 nnn w n o w n w n w o n w nnnnnnnnnnnnnnnnnnn o s i i
-- SAMPLE 3 --
|?|"He said: \"Ha\"?\\n"| |+| |"|He| |He|\|n|" + "|Ho|"|
 i qqqqqqqqqqqqqqqqqqqq w o w i ss w ss i s qqqqq ss i



File:      lexemes.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Tue Oct 12 18:39:49 EDT 2010

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: 2010/10/12 22:41:13 $
    $RCSfile: lexemes.txt,v $
    $Revision: 1.8 $
```

Bezier Curve

A Bezier Curve is a smooth approximation to a segmented line through the points P1, P2, ..., Pn. The segmented line goes straight from P1 to P2, then straight on to P3, etc., and so is composed of a sequence of straight line segments. The Bezier Curve, however, is very smooth, although it starts at P1, ends at Pn, and very roughly approximates the segmented line.

The points on the Bezier Curve are designated

```
B(t, P1, P2, P3, ..., Pn)
```

where t varies from 0 through 1.

If there are only two points, the Bezier Curve is simply the straight line between them given by

```
B(t, P1, P2) = (1-t)*P1 + t*P2
```

If there are n>2 points the Bezier Curve for n points is computed from two Bezier Curves for n-1 points as follows

```
B(t, P1, P2, P3, ..., Pn) =
    (1-t) * B(t, P1, P2, P3, ..., P(n-1))
    +
    t * B(t, P2, P3, P4, ..., Pn)
```

The points P1, P2, ..., Pn are called ‘control points’. The Bezier Curve computed from them starts at P1 and ends at Pn, and is tangent at its start to the straight line P1-P2 and at its end to the straight line P(n-1)-Pn. However, the Bezier Curve often does not get close to non-end control points.

You have been asked to compute and plot Bezier Curves and plot their control points.

Input

For each of several test cases, a single line containing the test case name, followed by one or more lines containing the numbers:

```
m X Y n P1x P1y P2x P2y P3x P3y ... Pnx Pny
```

These numbers may be on a single line or spread out among several lines, and may be aligned in any columns of the input lines.

m is the number of points of the Bezier Curve to be computed, the graph has X columns and Y rows, n is the number of control points, and (P1x,P1y), (P2x,P2y), ... are the control points. 1 <= m; 1 <= X <= 80; 1 <= Y <= 40; 2 <= n <= 50; 0 <= Pjx <= X-1; 0 <= Pjy <= Y-1. m, X, Y, and n are integers while Pjx and Pjy may be floating point.

No input line is longer than 80 characters. The input ends with an end of file.

Output

For each test case one line containing an exact copy of the test case name input line, followed by ceiling(m/5) lines containing m Bezier Curve coordinate pairs, followed by Y lines containing a graph of the Bezier Curve.

The lines containing the Bezier Curve coordinates each have the format:

```
 xx.x yy.y xx.x yy.y xx.x yy.y xx.x yy.y xx.x yy.y
```

where xx.x denotes an x-coordinate value and yy.y denotes a y-coordinate value, each printed in 5 columns with exactly one decimal place (all correct numbers will be >= 0 and < 100). There are 5 (x,y) pairs per line (maybe less on the last line), and these are in order the B(t,...) values for t = 0, 1/m, 2/m, 3/m, ..., (m-1)/m (there is NO value for t = 1).

The graph is Y lines each with X columns. For each B(t,...) point (x,y), an asterisk * is placed in

```
        row    = Y - floor ( y + 0.5 )
        column = 1 + floor ( x + 0.5 )
```

where rows are numbered 1, 2, 3, ... from the top and columns are numbered 1, 2, 3, ... from the left. For each of the initial points (Pix,Piy), a plus sign + is then placed in

```
        row    = Y - floor ( Piy + 0.5 )
        column = 1 + floor ( Pix + 0.5 )
```

overlaying any * that is there.

Thus the lower left corner of the graph corresponds to (x,y) = (0,0), and to compute the graph location of (x,y) the values of x and y are both rounded to the nearest integer.

Note: you MUST do the rounding correctly or your graph will have misplaced ‘*’s or ‘+’s and your output will be scored as INCORRECT. The judge’s input is chosen so there will be no graphed x or y values extremely near the midpoint between two integers.

Sample Input

```
------ -----

-- SAMPLE 1 --
6 21 7
   4 0 0 4 2 8 4 12 6
-- SAMPLE 2 --
40 41 16 16
  20  0
   0  5  0 10
  10 15 14 15 15 15
  20  6 20  5 20  5 20  6
  25 15 26 15 30 15
  40 10 40  5
  20  0
```

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

-- SAMPLE 1 --
  0.0  0.0  2.0  1.0  4.0  2.0  6.0  3.0  8.0  4.0
 10.0  5.0
           +
          *
        +
       *
     +
   *
+
-- SAMPLE 2 --
 20.0  0.0 13.7  1.9  9.7  3.7  7.3  5.5  6.2  7.1
  6.1  8.6  6.6  9.8  7.5 10.8  8.6 11.4  9.8 11.8
 11.1 12.0 12.4 11.9 13.5 11.6 14.6 11.1 15.6 10.6
 16.5 10.0 17.3  9.5 18.1  9.0 18.7  8.6 19.4  8.4
 20.0  8.3 20.6  8.4 21.3  8.6 21.9  9.0 22.7  9.5
 23.5 10.0 24.4 10.6 25.4 11.1 26.5 11.6 27.6 11.9
 28.9 12.0 30.2 11.8 31.4 11.4 32.5 10.8 33.4  9.8
 33.9  8.6 33.8  7.1 32.7  5.5 30.3  3.7 26.3  1.9
          +   ++         ++   +


          *** *           * ***
       * *     **       **     * *
+      *         *     *         *      +
      *          *** ***          *
                   ***
      *                           *

                    +
+      *            +            *      +
          *                   *

               *           *

                    +
```

```
File:      bezier.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Thu Oct  7 02:54:26 EDT 2010

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: 2010/10/08 13:03:33 $
    $RCSfile: bezier.txt,v $
    $Revision: 1.8 $
```

Congruent Polygons
--------- --------

You have been asked to determine whether or not two polygons are congruent, and if so, how to transform the first polygon in order to make it match precisely the second polygon. The permitted transformation consists of an optional reflection about the X-axis followed by a rotation counter-clockwise about the origin followed by a translation.

Input

For each of several test cases, a line containing just the test case name, followed by lines describing the two polygons. Each polygon description is N+1 numbers, where N is the number of vertices in the polygon. The first number is N itself. The rest of the numbers are x,y pairs for each vertex. Thus the format of a polygon description is

```
N v1x v1y v2x v2y ... vNx vNy
```

except the numbers may be distributed in any fashion across one or more lines.

The polygon vertices are always given in clockwise order. The angle between successive polygon sides is always different from 180 degree, so no vertices are superfluous. The two polygons have the SAME number of vertices, N. 3 <= N <= 100.

To make things easier, the first vertex of the first polygon is always (0,0), the origin.

Lastly, all XY-coordinates have exactly 6 decimal places.

No input line is longer than 80 characters. Input ends with an end of file.

Output

For each test case, two lines. First, an exact copy of the test case name line. Then either a line containing just

```
not congruent
```

or a line of the format

```
r angle x y
```

which defines a transformation that carries the first polygon onto the second polygon. Here

r = I	meaning do NOT reflect the first polygon
= R	meaning DO reflect the first polygon about the X-axis; the reflection of (x,y) is (x,-y).
angle	is the angle measured in DEGREES to rotate the first polygon counter-clockwise about the origin (the first vertex of the polygon)
x y	are the amounts to add to the x and y coordinates of the vertices to translate the first polygon; thus the first vertex (0,0) is translated to (0+x,0+y) = (x,y)

The transformation defined consists of first an optional reflection about the X-axis, then a counter-clockwise rotation about the origin, and lastly the translation.

The angle, x, and y may be printed to any number of decimal places, or may be printed in C++ scientific notation. However, if these values are rounded to too few decimal places, the translation they define may become unacceptable (see discussion below), so we suggest these values be output to 6 decimal places.

There is a technical problem related to the fact that vertex input coordinates are only accurate to + or - 0.0000005, given that they have only 6 decimal places. We deal with this as follows.

For any transformation we define the error of the transformation to be the maximum absolute value of the difference between any input second polynomial vertex coordinate and the coordinate value of the transformed corresponding first polynomial vertex. We define a transformation to be acceptable if its error is <= 0.001 (which is 2,000 times the input error). We define two polygons to be congruent if there is an acceptable transformation. Then for judging safety, the judge's input data are carefully chosen so that if the polygons are congruent there are obvious transforms with errors well below 0.001, and if the polygons are not congruent all possible transforms have errors well above 0.001. In other words, there are no 'close calls' in the judging input.

If the polygons are congruent you must output an acceptable transform. If there is more than one, output only one.

Sample Input

```
------ -----

-- SAMPLE 1 --
5
    0.000000  0.000000  0.000000  8.000000
    5.000000  6.000000  8.000000  8.000000
    8.000000
        0.000000
5
    4.000000  5.000000 -4.000000  5.000000
   -2.000000 10.000000 -4.000000 13.000000
    4.000000 13.000000
-- SAMPLE 2 --
5
    0.000000  0.000000  0.000000  8.000000
    5.000000  6.000000  8.000000  8.000000
    8.000000  0.000000
5
   -4.000000 13.000000  4.000000 13.000000
    2.000000 10.000000  4.000000  5.000000
   -4.000000  5.000000
-- SAMPLE 3 --
5
    0.000000  0.000000  0.000000  8.000000
    5.000000  6.000000  8.000000  8.000000
    8.000000  0.000000
5
    4.000000  5.000000 -4.000000  5.000000
   -6.000000 10.000000 -4.000000 13.000000
    4.000000 13.000000
```

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

-- SAMPLE 1 --
I 90.000000 4.000000 5.000000
-- SAMPLE 2 --
R 90.000000 -4.000000 5.000000
-- SAMPLE 3 --
not congruent



File:      congruent.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Tue Oct 12 20:28:22 EDT 2010

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: 2010/10/13 00:39:48 $
    $RCSfile: congruent.txt,v $
    $Revision: 1.11 $
```

Escape From The Maze: Part II
------ ---- --- ----- ---- --

The maze from the ‘escape’ problem has been upgraded to be more difficult.

The big new difficulty is that more than one monster is now present, and you can be trapped. Note that two monsters cannot occupy the same square (they will not fit).

However, there is some good news. First, you now can hurl immobilizing juice at a monster to stop it from doing anything for a while. Exact rules for this are below.

And in addition to seeing whether there is a wall, corridor end, exit, or monster in a particular direction, you can see how far away these things are.

You have a flask which can hold any amount of immobil-izing juice. You can remove N units of this juice from the flask and hurl it at any monster you can see. This will immobilize the monster for N moves. You can occupy the same square as in immobilized monster WITHOUT being eaten.

Jugs of immobilizing juice are on certain squares of the maze, and when you occupy such a square any juice in the square’s jug will be automatically transferred to your flask. Furthermore, the jugs refill at the rate of one juice unit per R moves you make (1 <= R <= 5), so if you come back to a jug you will get more juice. You need not hurl the entire contents of your flask at a monster.

Each jug has a maximum capacity of from 1 to 9 juice units. Jugs are represented in pictures of the maze by the digits 0 through 9 which tell how much juice is currently in each jug. A jug cannot occupy the exit square.

To make things more interesting, the sizes of the the mazes have been increased for this problem. A maze can be at most 40 lines tall and 80 columns wide. You should set your terminal window to 48 lines tall times 80 columns wide to use debugging mode for maze2. Then you can type

```
maze2 sample_escape2
> d1
> m5 5 0 2 1234
> f30
> f
> f
> f1 1000
> b1 1
> b10
> f
> f10
```

to get a feel for the problem.

There is no infallible solution to this problem. In order to make it tractable, a new j command is added to the maze program: see below.

The Maze Program
--- ---- -------

In the same fashion as the ‘escape’ program, you test your program by running

```
maze2 escape2
```

The ‘maze2’ program has the same commands as the ‘maze’ program except for one command that is changed and two new commands. The changed and new commands are:

m M J F R S
: Four parameters, M, J, F, and R, are added to the maze creation command. M is the number of monsters, J the number of jugs, F the number of potion units initially in your flask, and R the number of moves you make before one unit of potion is added to a jug you have emptied. S is the random number generator seed as before.

  In addition, if you give NO parameters, but just input ‘m’, the parameters are set from the FAILED_MAZE file written by the ‘j K’ command below.

j
: Your program is judged, using the same mazes (i.e., ‘m’ command parameters) as the judge will use to score your program. If your program escapes a sufficient number of these mazes in a sufficiently small number of total moves, this command will print ‘SUCCESS!’. If not, it will print ‘FAILURE!’. The command will also print the number of mazes you escaped, your total number of moves, and the numbers required for success.

  A submission for which the ‘j’ command produces ‘SUCCESS!’ will be scored CORRECT, and a submission for which ‘j’ produces ‘FAILURE!’ will be scored INCORRECT.

  Note that debugging is effectively off during the ‘j’ command (so there is no way of seeing all the mazes you succeeded on).

j K
: Same as j but the command stops on the K’th maze you fail to escape from and writes the ‘m’ command for this maze in the FAILED_MAZE file, which can be used by the ‘m’ command above to permit you to debug your program on the first failed maze.

Also, the mazes output by the ‘maze2’ program have three features the ‘maze’ program output does not have. First, a square with a potion jug has a digit, ‘0’ though ‘9’, indicating how full the jug is (the capacity of the jug is not indicated, but the jugs will be at capacity initially and will stop filling when they reach capacity). Second, an immobilized monster displays as an ‘I’ instead of an ‘M’. And third, several things can be on the same square, and in that case the priority of display from lowest to highest is ‘Y’, ‘X’ or jug, ‘I’ or ‘M’. Thus if you and an immobilized monster occupy the same square, you will see an ‘I’ in that square.

The number of mazes tested by the ‘j’ command is large to prevent you from using ‘j’ to extract all the judge’s maze parameters within the time constraints of the contest. The judge’s solution has modest intelligence of a straight forward kind. BUT, to succeed in this problem you should have a plan for increasing the intelligence of your escape program as much as necessary.

Note also that there is no limit on the number of moves you may make on any given maze, and no OUT OF TIME result for a single maze, but there is instead a limit on the number of moves you may make for all mazes tested by the ‘j’ command.

Escape2 Program Input
------- ------- -----

The ‘-...’ and ‘p...’ input lines are as for ‘escape’. A new ‘m’ input line has been added, and the ‘nesw’ line has been changed. The new or changed lines are:

m F R — F is the number of units of potion initially in your flask, and R is the number of moves you must make in order for one unit of potion to be put in any jug that is not full.

Note: you are NOT told M, the number of monsters in the maze, or J, the number of jugs in the maze.

neswx — n, e, s, and w each denote one of the following STRING of characters:

W	
#E	#J#E
#X	#J#X
#M	#J#M

Here the #’s stand for unsigned decimal integers. #E, #X, #M means there are # squares until the the corridor ends or there is an exit or there is a monster. # == 0 means the exit or monster is in the next square; 0E is never used (W is used instead). You cannot see beyond an exit or monster.

#J means there are # squares until there is a jug. You can see corridor ends, exits, and monsters beyond a jug, but you cannot see another jug beyond a jug.

You are NOT told whether or not the monster is mobile.

#M does not tell you about any immobilized monster in the same square as you, and #J does NOT tell you about any jug in the same square as you (but see #F below). If a square holds a monster and something else you see only the monster.

x is one of the following strings of characters:

"" (empty string)
#F

#F means there is a jug on your square from which # units of potion have just been transferred to your flask.

There are no space characters in a neswx line.

Escape2 Program Output
------- ------- ------

For each ‘neswx’ line input you must output a line with one of the following formats:

```
        d               #dd
```

Here ‘d’ is one of the direction characters, N, E, S, or W, as for ‘escape’. The last ‘d’ is the direction in which you want to move. A preceding ‘#d’ means you want to hurl # units of potion in direction d. Thus on input

```
        W4MW10E
```

you might respond

```
        2EW
```

to hurl 2 units of potion at the monster to your East and then move yourself one square West.

You may put debugging information on these lines after the character indicating the direction in which you move. You must not put any space characters before this character. If you output a line with more than 80 characters, only the first 80 will be kept.

If you hurl potion in a direction in which there is no monster, the potion will have no effect. If you try to hurl more units of potion than you have, all the potion that you have will be hurled. In particular, if your flask is empty, hurling potion has no effect.

File: escape2.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Thu Oct 14 07:52:50 EDT 2010

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

```
    $Author: walton $
    $Date: 2010/10/14 11:58:14 $
    $RCSfile: escape2.txt,v $
    $Revision: 1.15 $
```

Making Plans
------ -----

You are being asked to make minimal plans.

These plans concern a world that is described by propositions that are either true or false. There are 26 possible propositions, one for each lower case letter of the alphabet.

For each proposition ‘p’ there are two ‘literals’ that can be used to describe the world: ‘+p’ is a literal that means proposition p is true, and ‘-p’ is a literal that means p is false. The negation of a literal is the same literal with the sign switched, i.e., -p is the negation of +p and +p is the negation of -p.

Plans consist of actions. An action has the form

name:{pre-conditions}=>{post-conditions}

where ‘name’ is a natural number, pre-conditions is a list of literals that must be true in order for the action to be taken, and post-conditions is a list of literals that will be true after the action has been taken. An action is described on a single line which is written without any spaces; for example:

7:+p-q=>-p+q+r+t

A set of literals is conflicting if for some p both +p and -p are in the set. The pre-conditions of an action must be non-conflicting, and similarly the post-conditions of an action must be non-conflicting.

A plan is:

(1) A set ACT of actions.
(2) A strict partial order << on the set of actions ACT. Intuitively m<<n means m comes before n.
(3) A set of causal links. A causal link is denoted by
m=>n:c
where m and n are names of actions in ACT, m is an action to be taken to satisfy the pre-condition c of n, and c is a literal in the post-conditions of m and the pre-conditions of n. m=>n:c necessarily implies m<<n.

A strict partial order is a binary relation that is transitive and anti-symmetric. Antisymmetry means that m<<n implies NOT n<<m.

An action b is said to be a ‘threat’ to causal link m=>n:c if the negation of c is a post-condition of b, and b does not equal m or n. If b is a threat to m=>n:c, then either b<<m or n<<b must be true or the plan is inconsistent.

In order to define the initial state of the world and the final or goal state that our plan seeks we introduce two special actions, the initial action 0 and goal action 1, which have the forms:

0:=>{initial-conditions}
1:{goal-conditions}=>

0, the initial action, has empty pre-conditions and its post-conditions represent the initial state of the world. 1, the goal action, has empty post-conditions, and its pre-conditions represent the desired or goal state of the world. Every plan must contain the 0 and 1 actions. 0<<1 is required, and 0<<m<<1 is required for all actions m in the plan other than 0 or 1.

A plan is consistent if its strict partial order is really a strict partial order and if for every causal link m=>n:c in the plan and every action b in the plan that is a threat to that link, either b<<m or n<<b.

A plan is complete if it includes 0 and 1, 0<<1 and for every other action m in the plan 0<<m<<1, and for every action n in the plan and every literal c in the pre-condition of n, there is EXACTLY ONE causal link m=>n:c in the plan.

Note that an action can be used at most once in a plan; we do NOT permit actions to be replicated. Also note that there can be two causal links in a plan of the forms m=>n:c1 and m=>n:c2; that is, m=>n can appear more than once in a plan but with DIFFERENT literals c1 and c2.

You are being asked to find minimal complete, consistent plans.

Input

For each of several test cases, a line containing just the test case name, followed by any number of lines each describing one action, followed by a line containing just '.'.

There is one action named '0' and one named '1' in each test case. All other action names are unique and are natural numbers. The smallest natural number N such that all action names are in the range 0 .. N-1 is implicitly input and is used to define the output (see below).

3 <= N <= 200.

There is no whitespace in any input line other than the test case name line. Input lines are at most 80 characters long. Input ends with an end of file.

Output

For each test case, first an exact copy of the test case name line, followed by a description of a minimal consistent complete plan for the test case, or a single line containing EXACTLY 'no plan', meaning that no consistent complete plan is possible for the test case.

The description of a minimal plan begins with N lines each containing N characters that describe the strict partial order, followed by lines each describing one clausal link in the plan, followed by a line containing just '.' See 'Input' above for a definition of N.

In the N lines describing the strict partial order, column n of line m is

'1' if m and n are in the plan and m<<n

'0' if m and n are in the plan and NOT m<<n

'0' if either m or n is NOT in the plan

Here lines are numbered 0, 1, 2, ... from top to bottom, and columns are numbered 0, 1, 2, ... from left to right. If no action named m is in the input, m is treated as an action not in the plan for the purposes of this paragraph (you can treat m as an action with no pre- or post-conditions).

The lines describing the causal links each have the form 'm=>n:c'.

There are no spaces in any output lines except test case name lines and ‘no plan’ lines.

The plan output must be minimal in the following senses. First, it must contain a minimum number of actions. Second, among all plans with the minimum number of actions, it must have the minimum number of clausal links. Third, among all plans with the minimum number of actions and minimum number of clausal links, it must have the minimum number of << relations (literally, the minimum number of ‘1’s in the strict partial order description).

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

```
-- SAMPLE 1 --
2:+b=>-b+c
3:+c=>-c+d
4:+c=>+f
0:=>+b
1:+d=>
.
-- SAMPLE 2 --
0:=>+c-d
1:-f+c-g=>
2:-h=>+k-a
3:+b=>-m-c
4:+c=>+b-c
5:-d=>+d-f
6:-d+c=>+f+k
7:+f=>+g+b
8:-k+h=>-h+m
9:+g+c=>-f+m
10:+g=>-g-b
.
```

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

```
-- SAMPLE 1 --
01110
00000
01010
01000
00000
3=>1:+d
2=>3:+c
0=>2:+b
.
-- SAMPLE 2 --
01000111001
00000000000
00000000000
00000000000
00000000000
01000000000
01000101001
01000100001
00000000000
00000000000
01000000000
0=>1:+c
5=>1:-f
0=>5:-d
10=>1:-g
7=>10:+g
6=>7:+f
0=>6:+c
0=>6:-d
.
```

```
File:      plans.txt
Author:    Bob Walton <walton@seas.harvard.edu>
Date:      Thu Oct 14 09:30:13 EDT 2010

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: 2010/10/14 13:30:40 $
    $RCSfile: plans.txt,v $
    $Revision: 1.9 $
```

Penrose Tiling

Sir Roger Penrose investigated aperiodic tilings of the plane in the 1970's. These tilings are generated from a small number of finite shapes by following a set of rules, but no translation of a tiling is identical to the tiling, hence the designation 'aperiodic'.

Penrose rhombus tilings are generated from a pair of rhombi called 't', the 'thin' rhombus, and 'T', the 'thick' rhombus. All sides of these are unit length. The angles of t are 36 and 144 degrees, and those of T are 72 and 108.

The sides of the rhombi also need to be labeled, so we give the following algorithms for drawing them using a pen:

```
  for t, thin rhombus:
    draw a straight line of unit length labeled +1
    turn left 1*36 = 36 degrees
    draw a straight line of unit length labeled -1
    turn left 4*36 = 144 degrees
    draw a straight line of unit length labeled +2
    turn left 1*36 = 36 degrees
    draw a straight line of unit length labeled -2
    turn left 4*36 = 144 degrees
    you are now back in your starting position

  for T, thick rhombus:
    draw a straight line of unit length labeled +1
    turn left 2*36 = 72 degrees
    draw a straight line of unit length labeled +2
    turn left 3*36 = 108 degrees
    draw a straight line of unit length labeled -2
    turn left 2*36 = 72 degrees
    draw a straight line of unit length labeled -1
    turn left 3*36 = 108 degrees
    you are now back in your starting position
```

The rhombi must be fit together so:

1. The rhombi are rotated and/or translated but NOT flipped over.

2. Two rhombi may not intersect. This means that their intersection as sets, including boundaries, must not contain any points EXCEPT for those in shared vertices and shared edges.

3. When an edge is shared between two rhombi, the sum of the two labels of the edge must be 0. E.g., a +2 edge from one rhombus may be shared with a -2 edge from another rhombus, but NOT with a +2 or -1 or +1 edge.

4. There are no holes in the tiling.

In this problem you are given a proposed finite Penrose rhombic tiling and you are asked to determine whether it follows all the above rules.

We need a way to describe a finite Penrose rhombic tiling. We do this by placing the tiles down on the xy-plane so that each tile but the first shares an edge with one of the tiles laid down so far.

The first tile is always a T-tile with its +1 edge directed from (0,0) to (1,0) and its +2 edge directed from (1,0) to (x,y) with x>1, y>0. This is referred to as the 'standard position' for the first tile, which is also tile 1 in a our tile labeling scheme that numbers the n tiles laid down so far from 1 through n.

The position of the n+1'st tile is given by the line

```
k j e
```

where

- k is the kind of tile, either 't' or 'T'
- j is the number of a previous tile that is to share an edge with the new tile; 1 <= j <= n
- e is the label (+1, -1, +2, or -2) of the edge of tile j that is to be shared with the new tile, respecting the rule about the sum of shared edge labels being zero

Thus the line 't 7 -2' says to lay a t-tile so that its +2 edge is shared with the -2 edge of the 7'th tile laid.

Input

The input consists of test cases. Each test case begins with a line containing the name of the test case. This is followed by any number of lines each containing a description 'k j e' of another tile to be laid to make a tiling pattern. The first tile of the pattern is in standard position, and the i'th line of the form 'k j e' describes how to lay the i+1'st tile. After these lines there is a line containing just '.', which is the last line of the test case.

maximum number of tiles <= 10,000

for each tile vertex (x,y):
-100 <= x <= +100
-100 <= y <= +100

Output

For each test case, first output an exact copy of the test case name line, and then output just one line in one of the following formats:

```
tile # intersects tile #
tile # edge # is shared with tile # edge #
there are # holes
tiling OK
```

Here the #'s are integers that are tile labels, edge labels, or counts. The first line is output if two tiles intersect; the second if two share edges have labels not summing to 0. If the tiling violates the rule against intersection AND the rule against edge labels not summing to 0, then either of the first two lines may be output -- only one violation is to be reported.

However, reporting holes must ONLY be done if there are NO intersection or edge label sum violations.

Printing Input

As a debugging aid, the command

```
print_penrosetiling foo.in
```

will print a picture of the tiling described in foo.in. The file sample_input.ps contains the result for the sample input.

The labels in the picture are represented by single arrows (+1, -1) or double arrows (+2, -2) going around the rhombus boundary in the counter-clockwise (+1, +2) or clockwise (-1, -2) directions. They are offset so that usually if a shared edge has labels not summing to zero this will be visible in the picture. But their are perverse cases; consider:

```
        -- PERVERSE CASE --
        t 1 +1
        T 2 -1
        .
```

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

This is available in the file sample_input.in.

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

```
-- PENROSE TILING SAMPLE 1 --
tiling OK
-- PENROSE TILING SAMPLE 2 --
tile 1 and tile 7 intersect
```

File: penrosetiling.txt
Author: Bob Walton <walton@seas.harvard.edu>
Date: Thu Oct 14 09:48:57 EDT 2010

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

```
    $Author: walton $
    $Date: 2010/10/14 13:51:26 $
    $RCSfile: penrosetiling.txt,v $
    $Revision: 1.13 $
```