Problems Index Sat Oct 14 12:05:58 AM EDT 2006

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

```
    problems/jaybot
        Jumping around keeps things lively.
        Boston Preliminary 2006

    problems/handonwall
        How to explore the unknown.
        Boston Preliminary 2006

    problems/blowfish
        Scramble faster.
        Boston Preliminary 2006

    problems/endtoend
        When was what said?
        Boston Preliminary 2006

    problems/serializable
        No conflicted dipping into the food line.
        Boston Preliminary 2006

    problems/render
        The eyes get it.
        Boston Preliminary 2006

    problems/temporal
        And you thought you understood time.
        Boston Preliminary 2006
```

Jumping Robot

A Jumping Robot, or J-Bot, or Jaybot, is a robot that jumps instead of rolling or walking. The robot has a current position (x,y) and a jump vector (dx,dy). The robot moves by jumping from (x,y) to position (x+dx,y+dy).

The possible commands to a jaybot whose position is (x,y) and whose jump vector is (dx,dy) are:

Command	New Position	New Jump Vector
jump	(x+dx,y+dy)	(dx,dy)
turn left	(x,y)	L(dx,dy)
turn right	(x,y)	R(dx,dy)
vector b c	(x,y)	(b,c)

Here R rotates a vector 90 degrees clockwise and L rotates a vector 90 degrees counterclockwise.

In this problem the jaybot lives on a board of MxN squares, each square with integer coordinates. All numbers are integers. The square in the lower left corner of the board has coordinates (0,0), and the square in the upper right corner has coordinates (M-1,N-1).

You are asked to plot the path of the jaybot by putting a letter on each square the jaybot visits. The first visited square (the jaybot's initial position) gets 'A', the second visited square gets 'B', etc. Unvisited squares are represented by the character '.'. If a square is visited several times, it only remembers the last letter it was given.

Input

Each of several test cases. Each case consists of a test case name line followed by the line:

```
M N x y dx dy
```

with 6 integers. The board is MxN. (x,y) is the initial position of the jaybot, and (dx,dy) the initial jump vector. 2 <= M,N <= 40; 0 <= x < M; 0 <= y < N. The x-axis is horizontal (M columns) and the y-axis is vertical (N rows).

After the first two lines there are any number of command lines, each containing just one command as specified by the above table. Thus the 'vector' command is a line with the word 'vector' followed by two integers, b and c, all separated by whitespace.

The commands are followed by a line containing nothing but "end".

The jaybot is guaranteed not to jump off the edge of the board.

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

Output

For each test case, one empty line, followed by an exact copy of the test case name line, followed by a printout of the board with the path of the jaybot marked as indicated above. Note that the first line output is an empty line, and there are no space characters in any output line except perhaps in the test case name line.

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

-- SAMPLE 1 --
30 8 5 1 1 1
jump
turn right
jump
vector 2 2
jump
turn right
jump
vector 3 3
jump
turn right
jump
vector 4 4
jump
turn right
jump
end
-- SAMPLE 2 --
10 5 0 0 1 1
jump
jump
turn left
jump
turn right
jump
vector 4 0
jump
turn right
jump
turn right
jump
turn right
jump
end
```

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

[The output below begins with an empty line.]


-- SAMPLE 1 --
..............................
..............................
.....................H........
..............F...............
.........D....................
......B.......................
.....A.C...E.....G.......I....
..............................

-- SAMPLE 2 --
..I...F...
.D........
..C.......
.B........
A.H...G...
```

```
File:       jaybot.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Wed Oct 11 10:32:51 EDT 2006
```

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: 2006/10/11 14:32:58 $
    $RCSfile: jaybot.txt,v $
    $Revision: 1.8 $
```

Right Hand on the Wall

Karel the Robot lives on a 10x10 board of squares. Each boundary line of a square may or may not have a wall. At any given time, Karel is on one particular square, and is facing in one of four directions: up, left, down, or right.

Karel can move forward one square if there is no wall in front of him. Alternatively, Karel can turn 90 degrees clockwise or 90 degrees counter-clockwise. These are the only moves Karel can make. Karel can sense whether there is a wall in front of him.

In this problem Karel repeats the 'Right Hand on the Wall' algorithm, which is:

(1) If there is a wall in front of Karel, Karel turns counterclockwise 90 degrees.

(2) Otherwise if the square in front of Karel is the original square on which Karel started, Karel stops (WITHOUT moving forward to the original square).

(3) Otherwise Karel moves forward one square and turns clockwise 90 degrees;

Starting from an initial position facing a wall, Karel repeats this algorithm until he stops.

The boundary lines on the edge of the board all have walls, so Karel can never fall off the edge of the board, and will always eventually stop.

You are asked to make Karel move according to this algorithm, and display the results.

Board Display

The 10x10 board is displayed in a 21x21 character matrix that can be printed in 21 lines of 21 columns each.

There is one character position for each square, for each boundary line of a square, and for each corner of a square.

The corner character positions hold the ‘+’ character.

The boundary line character positions hold the space character if there is no wall at the boundary, or ‘-’ for a horizontal boundary wall, or ‘|’ for a vertical boundary wall.

A square character position holds the space character if Karel has never visited the position. Otherwise it holds a character showing the direction Karel LAST faced when he was at that square. ‘<’ and ‘>’ are used for ‘facing left’ and ‘facing right’, respectively. ‘^’ and ‘v’ are used for ‘facing up’ and ‘facing down’, respectively. Here ‘^’ is the circumflex and ‘v’ is the lower case letter.

Input

Each of several test cases. Each case consists of an empty line followed by a board display. On the board display, Karel is shown as being at one position and facing in one direction (there is only one ‘<’, ‘>’, ‘^’, or ‘v’ on the board). The start position is always such that Karel is facing a wall, and all the board edges have walls.

Input ends with an end of file.

Output

For each test case, a copy of the input for the test case, with some board squares changed to hold characters showing that Karel has been at the square and was facing in a particular direction when he was last at the square.

The board should show Karel’s movement using the Right Hand on the Wall algorithm exactly as described above, starting with the initial situation defined by the input board.

The empty lines beginning each input test case are copied to the output, so the first line output is an empty line. The output should be an exact copy of the input except that some square positions are changed to ‘>’, ‘^’, ‘<’, or ‘v’.

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

[There is an empty line before each board.]

```

+-+-+-+-+-+-+-+-+-+-+
|   |               |
+ + + + + + + + + + +
|   |               |
+-+-+ + + + + + + + +
|                   |
+ + + + + + + + + + +
|                   |
+ + + + + + + + + + +
|                   |
+-+-+ + + + + +-+-+-+
|                   |
+-+-+ + + + + + + + +
|                   |
+ + + + + + + +-+-+-+
|             |     |
+ + + + + + + +-+-+-+
|                   |
+ + + + + + + + + + +
|v                  |
+-+-+-+-+-+-+-+-+-+-+
```

```
+-+-+-+-+-+-+-+-+-+-+
|       |           |
+ + + + +-+-+-+ + + +
|             |     |
+ + + + + + + +-+-+-+
|                   |
+ + + + + + + +-+-+-+
|             |     |
+-+-+-+-+-+-+ + + + +
|             |     |
+ + + + +-+-+-+-+-+-+
|                   |
+ + + + + + + +-+-+-+
|             |     |
+ + + + + +-+-+ + + +
|        >|         |
+-+-+ + + + + + + + +
|   |               |
+ + +-+-+ + + + + + +
|       |           |
+-+-+-+-+-+-+-+-+-+-+
```

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

[There is an empty line before each board.]

```
+-+-+-+-+-+-+-+-+-+-+
|   |v < < < < < < <|
+ + + + + + + + + + +
|   |v             ^|
+-+-+ + + + + + + + +
|v < <             ^|
+ + + + + + + + + + +
|v                 ^|
+ + + + + + + + + + +
|> > v       > > > ^|
+-+-+ + + + + +-+-+-+
|> > v       ^ < < <|
+-+-+ + + + + + + + +
|v < <       > > > ^|
+ + + + + + + +-+-+-+
|v           ^|     |
+ + + + + + + +-+-+-+
|v           ^ < < <|
+ + + + + + + + + + +
|> > > > > > > > > ^|
+-+-+-+-+-+-+-+-+-+-+
```

```
+-+-+-+-+-+-+-+-+-+-+
|v < < <|           |
+ + + + +-+-+-+ + + +
|v     ^ < < <|     |
+ + + + + + + +-+-+-+
|v           ^ < < <|
+ + + + + + + +-+-+-+
|> > > > > v ^|     |
+-+-+-+-+-+ + + + + +
|v < < < < < ^|     |
+ + + + +-+-+-+-+-+-+
|v     ^ < < < < < <|
+ + + + + + + +-+-+-+
|v       > > ^|v < <|
+ + + + + +-+-+ + + +
|> > v   ^|v < <   ^|
+-+-+ + + + + + + + +
|   |> > ^ <       ^|
+ + +-+-+ + + + + + +
|       |> > > > > ^|
+-+-+-+-+-+-+-+-+-+-+
```

File: handonwall.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Tue Oct 10 02:10:25 EDT 2006

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: 2006/10/10 06:22:07 $
    $RCSfile: handonwall.txt,v $
    $Revision: 1.6 $
```

Mini-Blowfish
---- --------

The Blowfish algorithm has become a popular encryption algorithm for data streams and large files, as it can be efficiently implemented in software. In this problem you are asked to code and test a miniature version of this algorithm, which we call Mini-Blowfish, or MB for short.

Description of MB:
----------- -- --

MB uses an 18+256 byte vector of ‘subkeys’. The first 18 of these are referred to as P[1] through P[18]. The next 256 are referred to as S[0] through S[255]. The 18+256 subkeys are collectively referred to as K[1] through K[18+256], so K[1] == P[1], K[18] == P[18], K[19] == S[0], K[18+256] == S[255].

The S values define a ‘substitution-box’, or S-box, that takes a byte B as input and returns the byte S[B] as output, where bytes are viewed as unsigned integers from 0 through 255. E.g., if B == 5 the S-box returns S[5].

The data encryption algorithm inputs and outputs 16 bit blocks. These are divided into a high order byte, HB, and a low order byte LB, so block B == 256 * HB + LB.

The encryption algorithm is:

```
    Input B = 256 * HB + LB.
    For round R = 1 though 16:
        HB = HB xor P[R].
        LB = LB xor S[HB].
        swap HB and LB.
    Finishing:
        swap HB and LB (undo the round 16 swap).
        LB = LB xor P[17];
        HB = HB xor P[18];
    Output B = 256 * HB + LB.
```

Note that P[1], ..., P[18] are accessed in order by the encryption algorithm. Decryption uses the same algorithm except that P[1], ..., P[18] are used in the reverse order (P[18] is used in round 1 and P[1] is xor’ed at the end into HB).

The main idea in Blowfish is the method of computing the subkeys. In fact, the idea is to have a lot of subkeys (full Blowfish as 1042 32-bit subkeys). Computing the subkeys takes a long time, so changing the key in MB or Blowfish is slow, and has been made so in order to have a secure algorithm in which encrypting the data given the subkeys is fast.

To initialize the subkey vector K[1], ..., K[18+256] you need as input a password, which is any string of characters. Let the bytes of the password be W[1], W[2], ..., W[N] where N is the length of the password. The MB subkey computation algorithm is then:

```
        Input W[1], ..., W[N].
        For i from 1 through 18+256:
            K[i] = 7 ** i mod 256;
        For i from 1 through N:
            K[i] = K[i] xor W[i];
        Set B = 0, a 16 bit value.
        For round Q from 1 through (18 + 256)/2:
            Encrypt B to obtain Encrypted-B
            Set B = Encrypted-B
            Let B = 256 * HB + LB as above.
            Set K[2*Q-1] = HB and K[2*Q] = LB.
        Output K[1], ..., K[18+256].
```

Note that the output B of the encryption in round Q becomes the input B to the encryption in round Q+1. Also the subkeys at the end of round Q are the subkeys used in the encryption in round Q+1. Thus B and the subkeys keep changing as Q advances. The subkeys at the end of round Q = (18+256)/2 are the final output of the subkey computation algorithm.

Input

Lines each of which contains a password and some integers to be encrypted using the password, all followed by the integer -1 (which is NOT to be encrypted). These are separated from each other by whitespace. No line is longer than 80 characters.

The password is a string of one or more letters and digits, each interpreted as a byte equal to the ASCII code of the letter or digit (ASCII codes are the codes used to represent characters as integers in modern computers, and all ASCII codes are between 0 and 127). The integers to be encrypted are all in the range from 0 through 65535 (= 2**16 - 1), and each integer represents a 16 bit block.

Input ends with an end of file.

Output

For each input line, one output line, in the same format as the input line, except that each integer to be encrypted is replaced by the result of encrypting it.

Sample Input

```
abcdefg 0 1 2 3 4 5 -1
2hotfudge 28647 64826 42873 60872 53872 7648 29640 -1
```

Sample Output

```
abcdefg 61669 41297 34644 22212 18368 679 -1
2hotfudge 37515 44577 40580 64732 42141 33306 62416 -1
```

```
Further Information
------- -----------

Blowfish was invented by Bruce Schneier, and is describ-
ed at
        www.schneier.com/paper-blowfish-fse.html

Blowfish is one of a large number of ‘Feistel ciphers’.
The full algorithm uses 32 bit subkeys, 64 bit blocks,
and 4 S-boxes that are applied to the 4 bytes of a
32-bit half-block to get 4 32-bit half-blocks that are
combined using addition and exclusive-or to make one
32-bit half-block.  The subkeys are initially set to
the fractional digits of PI, which are assumed to be
random.  Short passwords are extended by cycling through
their bytes.  However, like MB full Blowfish also has 16
rounds, 18 P values, and the same control flow as MB.


File:      blowfish.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Tue Oct 10 02:26:24 EDT 2006

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: 2006/10/10 06:26:37 $
    $RCSfile: blowfish.txt,v $
    $Revision: 1.6 $
```

End To End

Communications over an unreliable link can be made reliable by a simple end-to-end protocol. Each end sends messages which it numbers, 1, 2, 3, 4, etc. What is actually sent for each message is the packet:

```
        checksum acknowledgment number message
```

The acknowledgment ACK is such that all messages with numbers less than or equal to ACK have been correctly received. The number is the number of the message being sent (and is not zero). The checksum is the checksum of the acknowledgment, the number, and the message, and is used to test whether the packet has been correctly received.

In addition there are also ‘null packets’ that contain no message, and have the form:

```
        checksum acknowledgment 0
```

The receiver maintains three items of data: ACKOUT, the number of the last message received correctly, ACKIN, the largest acknowledgment received, and Q, a queue of messages that are to be or have been sent but have not yet been acknowledged as having been correctly received. ACKOUT and ACKIN are initialized to 0, and Q is initially empty.

When the receiver receives a packet, the receiver uses the checksum to see if the packet is correct.

Whenever the receiver receives a correct packet containing an acknowledgment ACK, if ACK > ACKIN, the receiver discards ACK - ACKIN messages from the front of Q, as these have been received correctly, and then sets ACKIN = ACK.

Whenever the receive receives a correct packet whose number equals ACKOUT + 1, the receive adds 1 to ACKOUT and processes the message. Processing a message adds output messages to the end of Q.

Whenever the receiver receives a packet, correct or not, and has done the above processing, the receiver sends one or more packets each containing the value of ACKOUT after the above processing in the packet acknowledgment field. If Q is empty, one null packet is sent. If Q is not empty, one packet containing each message in Q is sent, in the order the messages appear in Q, with the first packet having message number ACKIN+1, the next message number ACKIN+2, etc. However, Q itself is NOT changed by this process: no messages are removed from Q just because they have been sent.

As a special exception, if a correct null packet is received with acknowledgment matching ACKIN and Q empty, then no reply packet is sent. Otherwise the two ends would be sending null packets back and forth indefinitely.

This protocol can handle two kinds of link errors. The first is a message that is corrupted, so it has a bad checksum, or is just completely lost, so it never arrives. This may mean that a subsequent message will arrive before it can be used, and will have to be discarded. The second kind of link error is when a copy of an old message that was successfully received and processed arrives long after the original copy arrived.

You have been asked to write the code of one end of the link. For your end, message processing is as follows. Each message is string of text consisting of just space characters and letters. In this a maximal string of letters is called a ‘word’. You are to find the words of each message in order, and output for each word one message that contains just the word, with NO space characters.

NOTES: You need NOT check for protocol errors due to bugs in the code at other end of the link. An example of such an error would be acknowledging a message that has not yet been sent. A more subtle example is sending two different messages with the same message number. In this last case the above algorithm will process only the first message whose message number equals ACKOUT + 1 at the time its packet is received.

Input

Each packet is represented by a line with the format:

```
        checksum                An unsigned integer.

        1 space character

        acknowledgment          An unsigned integer.

        1 space character

        number                  An unsigned integer.

        1 space character

        message                 0 or more space char-
                                acters and letters.

        line feed               This is NOT part of
                                the message.
```

A correct checksum is just the sum of the acknowledgment, the number, and the ASCII codes of all the characters in the message. No packet contains more than 80 characters. No checksum, acknowledgment, or number has a sign or high order zeros.

The input ends with an end of file.

Output

For each input packet, echo the packet on a line with the preface ‘* ’, that is, a * character followed by a single space character. Then do the processing above and output zero or more packets as required by the algorithm. The output packets have the same format as the input packets (and do NOT have any ‘* ’ prefix).

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

```
1101 0 1 hello there
749 2 2 goodbye
1814 3 3 this is a duplicate
1814 3 3 this is a duplicate
7 7 0
2141 7 4 this has a bad checksum
1279 7 4 this does not
8 8 0
9 9 0
10 10 0
10 10 0
```

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

```
* 1101 0 1 hello there
534 1 1 hello
539 1 2 there
* 749 2 2 goodbye
750 2 3 goodbye
* 1814 3 3 this is a duplicate
447 3 4 this
228 3 5 is
106 3 6 a
965 3 7 duplicate
* 1814 3 3 this is a duplicate
447 3 4 this
228 3 5 is
106 3 6 a
965 3 7 duplicate
* 7 7 0
3 3 0
* 2141 7 4 this has a bad checksum
3 3 0
* 1279 7 4 this does not
452 4 8 this
440 4 9 does
351 4 10 not
* 8 8 0
440 4 9 does
351 4 10 not
* 9 9 0
351 4 10 not
* 10 10 0
4 4 0
* 10 10 0
```

Further Information
------- -----------

Real algorithms also use a clock and send packets when they have not received any packet in a sufficiently long time and Q is non-empty. Sometimes they send packets periodically even when Q is empty in order to detect when the link has failed.

Real algorithms only send the low order bits of the acknowledgment and message number.

Real algorithms use more sophisticated checksums.

If you think carefully about our algorithm, you will see that there is an efficiency problem with it if both ends of the link use our algorithm (with different message processing, of course), because we can send a batch of more than one packet in response to a single packet. This can be fixed by batching more than one message in one packet, or by other means.

There is an amusing theorem of computer science that says that if you have two unreliable links connected, as in

```
        X  <---------->  Y  <---------->  Z
```

you CANNOT make communications from X to Z reliable by making each of the two links reliable. The problem is that if Y fails and then restarts, and X has sent a message to Y just before Y fails, X cannot tell whether the message was forwarded to Z or not, and therefore the message may be lost if X does not resend it or duplicated if X does resend it.

The solution is for X and Z to run the end-to-end algor-
ithm themselves, without Y participating in the algor-
ithm (Y merely forwards packets). This does not mean
that making the two links reliable will not improve
efficiency or maintainability; just that it will not
guarantee reliability of XZ communications.

File: endtoend.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Mon Oct 16 10:46:36 EDT 2006

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: 2006/10/16 14:45:08 $
    $RCSfile: endtoend.txt,v $
    $Revision: 1.8 $

```
Serializable
------------

A transaction is a computation that can read and write
objects.  Suppose we have several transactions and a set
of objects whose names are just single lower case
letters.  A transaction is just a list of read and write
operations; for example:

    Rx Ry Rz Wx Wz

where Rx means read object x, Wx means write object x,
and so forth.  These operations must be executed in the
order given.  We are abstracting and so neglect to men-
tion the computations that derive the value written into
x from the values read from x, y, and z.  To simplify we
will assume that every object has a unique name that is
just a single lower case letter.

Suppose we have a set of transactions and give each a
number: 1, 2, 3, ....  For example,

    1: Rx Ry Rz Wx Wz
    2: Ry Rx Wz
    3: Ry Wy

A schedule is a list of transaction numbers that tells
the order in which operations of the transaction are
executed.  For example:

    Schedule A:
        1  2  3  1  2  3  2  1  1  1
        Rx Ry Ry Ry Rx Wy Wz Rz Wx Wz

where we have written the operations executed under each
schedule transaction number.  In other words, for this
schedule transaction 1 executes its first operation Rx,
then transaction 2 executes its first operation Ry, then
transaction 3 executes its first operation Ry, then
transaction 1 executes it next (second) operation Ry,
and so forth.

A schedule is SERIAL if all the operations of each
transaction are consecutive.  Thus

    Schedule B:

        2  2  2  1  1  1  1  1  3  3
        Ry Rx Wz Rx Ry Rz Wx Wz Ry Wy

is a serial schedule.

Two schedules have the same effect of one can be made
from the other by switching the order of NON-CONFLICTING
operations.  Two operations conflict if both are on the
same object and at least one is a write.  Thus Rx and Wx
conflict, as do Wx and Wx, but Rx and Rx are non-con-
flicting, as are Rx and Wy and Wx and Wy.

A schedule is SERIALIZABLE if it has the same effect as
a serial schedule.

You are asked in this problem to determine whether a
schedule is serializable.
```

```
Input
-----

For each of several test cases:

    one line with the format:

                T S

    one or more schedule lines with a total of
        S transaction numbers

    T transaction description lines, each of at most
      80 characters

T is the number of transactions; S is the length of the
schedule; 1 <= T <= 20 and 1 <= S <= 100.  Transactions
are numbered 1 through T, with transaction number 1
being described first.  A transaction description line
consists of one or more operations separated by white-
space, where an operation is a pair of letters Rx or Wx,
and x is any lower case letter, that is, any object
name.  The schedule consists of S whitespace separated
integers, each in the range from 1 through T, and the
schedule may be broken across several lines.

Input ends with an end of file.


Output
------

For each test case, one line containing just the word

        SERIALIZABLE

or just the word

        NON-SERIALIZABLE



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

3 10
1  2  3  1  2  3  2  1  1  1
Rx Ry Rz Wx Wz
Ry Rx Wz
Ry Wy
3 10
1  2  3  1  2  3  1  1  2  1
Rx Ry Rz Wx Wz
Ry Rx Wz
Ry Wy


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

SERIALIZABLE
NON-SERIALIZABLE


Remarks
-------

Programmers expect transactions on a data base to be
executed using a serializable schedule, for if conflic-
ting operations from different transactions are inter-
leaved, it becomes nearly impossible to understand what
is happening.
```

A typical data base system contains a scheduler that employs one of several methods of rendering its schedules serializable. One is strict two-phase locking, in which a transaction locks objects before accessing them, and does not release any locks until the end of the transaction. Such a scheduler may get into deadlock situations, in which transaction M is waiting for a lock held by transaction N, and transaction N is at the same time waiting for a lock head by transaction M. The scheduler must then abort one of the transactions; i.e., the transaction must be stopped, any writes it has done must be undone (which is easy if it never writes until after it has all its locks), and all its locks must be released (allowing other transactions to proceed). Then the aborted transaction must be restarted.

Reference: Concurrency Control and Recovery in Database Systems by Bernstein, Hadzilacos, and Goodman.

```
File:      serializable.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Tue Oct 10 09:00:01 EDT 2006
```

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: 2006/10/10 13:04:01 $
    $RCSfile: serializable.txt,v $
    $Revision: 1.4 $
```

Rendering Triangles

In order to produce a picture, computer algorithms often divide surfaces up into triangles each with its own color. The triangles are then inserted into the image, starting with triangles representing surface parts that are farthest from the viewer, so that if two triangles overlap, the nearer covers the overlapping part of the farther triangle.

This problem asks you to do this for a simplified case. The image consists of MxN pixels that are assigned integer coordinates from (0,0) through (M-1,N-1). The triangle vertices have integer coordinates, with vertex (x,y) corresponding to pixel (x,y). The triangle colors are simply upper case letters. The pixels correspond to characters in a set of N lines each of M characters, so the image can be easily printed. Each pixel prints as its color. The period '.' is used to denote white, so the position of a white pixel can be easily ascertained.

A pixel is painted with the color of a triangle if the pixel is inside OR ON THE BOUNDARY of the triangle.

To simplify the problem, no two triangles will have the same color, and colors earlier in the alphabet are for triangles farther from the observer. That is, if a pixel receives several colors, it remembers only the one that is latest in the alphabet.

Remark: This is the classical rendering problem solved by graphics hardware. However, the hardware has to deal with many more pixels and triangles than we do here. So the hardware needs more efficient algorithms than you need here.

Input

For each of several test cases, first a line with three integers, M, N, and T in that order, and then T lines each describing a triangle. Here 0 < M,N <= 50; 1 <= T <= 26. Each of the triangle description lines has the format

```
        C X1 Y1 X2 Y2 X3 Y3
```

where C is an upper case letter giving the color of the triangle, and (X1,Y1), (X2,Y2), (X3,Y3) are the vertices of the triangle. All vertex coordinates are integers, but they may be outside the ranges 0..M-1 or 0..N-1, i.e., the vertices may be outside the image.

Input numbers and color characters are separated by whitespace. No two triangles have the same color.

Input ends with an end of file.

Output

For each test case, an empty line, followed by N lines each of M characters.

Pixel (x,y) corresponds to the x+1'st character of the y+1'st line, where 0 <= x < M and 0 <= y < N. If the pixel has a color, the character is the color letter. If the pixel has no color, the character is the period '.'.

The first output line is empty. There are no whitespace characters in any output line.

Note that (0,0) is the UPPER left pixel and (M-1,N-1) is the LOWER right pixel.

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

```
40 6 8
A 0 0 5 5 10 0
B 5 0 10 5 15 0
C 10 0 15 5 20 0
D 15 0 20 5 25 0
E 20 0 25 5 30 0
F 25 0 30 5 35 0
G 30 0 35 5 40 0
H 35 0 40 5 45 0
6 6 4
B -8 -10 12 10 12 -10
C -15 20 22 -20 22 -18
A -12 -10 8 10 -12 10
D 4 6 7 3 7 6
```

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

[The first output line is empty.]

```

AAAAABBBBBCCCCCDDDDDEEEEEFFFFFGGGGGHHHHH
.AAAAABBBBBCCCCCDDDDDEEEEEFFFFFGGGGGHHHH
..AAAAABBBBBCCCCCDDDDDEEEEEFFFFFGGGGGHHH
...AAAAABBBBBCCCCCDDDDDEEEEEFFFFFGGGGGHH
....AAA..BBB..CCC..DDD..EEE..FFF..GGG..H
.....A....B....C....D....E....F....G....

..BBCB
...CBB
A.C.BB
AC...B
CAA...
AAAA.D
```

File: render.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Tue Oct 10 08:54:58 EDT 2006

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: 2006/10/10 12:56:35 $
    $RCSfile: render.txt,v $
    $Revision: 1.6 $

Temporal Logic

PF (Past/Future) Temporal Logic is like propositional calculus, except that instead of a proposition p being either TRUE or FALSE, p has a possibly different value of TRUE or FALSE at each of several times t. Thus a proposition p is a function from times in some set T to the 2-element set {TRUE, FALSE}. Alternatively, we may view p as a subset of T, namely the set of all times for which p is TRUE.

There is an order relation < on times, but there are no fixed rules about how this relation behaves. That is, given times t1 and t2, one would expect that one of t1 < t2, or t1 = t2, or t1 > t2, but we do NOT require this; several or none of these may be true. We even permit a time t to be in its own future, i.e., t < t, which means its also in its own past.

Pp means that proposition p was true at some time in the past. Fp means that p will be true at some time in the future. p(t) is the value, TRUE or FALSE, of proposition p at time t. Therefore

```
(Pp)(t) = there exists an s < t such that p(s)

(Fp)(t) = there exists an s > t such that p(s)
```

We define the logical expressions e of PF Temporal Logic as

```
e ::= p | q | r         [propositions]
    | (~e)              [negation]
    | (e&e)             [logical conjunction]
    | (e|e)             [logical disjunction]
    | (Pe)              [was true operation]
    | (Fe)              [will be true operation]
    | constant
```

The classical logical operators ~, &, and | are defined 'pointwise' for temporal logic:

```
(~p)(t)  = ~(p(t))
(p&q)(t) = (p(t)&q(t))
(p|q)(t) = (p(t)|q(t))
```

For standard propositional logic the constants are TRUE and FALSE. But in temporal logic, constants are functions mapping the set of times T to the values TRUE or FALSE, and may be alternatively represented as the subset of T on which the function takes the value TRUE.

For this problem we will take T to be the set of the single digit numbers, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. We will write a subset of T by listing the digits in the subset in parentheses. Thus (345) denotes the set of times {3, 4, 5}, and () denotes the empty set of times. These are our logical constants.

With this notation we find that

```
(~(23478)) = (01569)

((12345)&(456789)) = (45)

((123)|(345)) = (12345)
```

If in addition we assume that < is the standard order on digits, then

```
(P(35)) = (456789)

(F(235)) = (01234)
```

In general, < will be an arbitrary relation which we define with a 10x10 matrix of characters. Each of the 10 rows corresponds to a row digit, R, going from 0 at the top to 9 at the bottom. Each of the 10 columns corresponds to a column digit, C, going from 0 at the left to 9 at the right. The matrix position for R and C is 'X' if R<C is true and '.' if R<C is false. Thus if the matrix is given with one row per line, the matrix

```
        .XXXXXXXXX
        ..XXXXXXXX
        ...XXXXXXX
        ....XXXXXX
        .....XXXXX
        ......XXXX
        .......XXX
        ........XX
        .........X
        ..........
```

corresponds to the standard ordering of the digits.

Input

For each of several test cases:

```
        one line containing the name of the test case
        10 lines each of 10 characters containing the
           matrix representation of the < relation,
           as described above
        any number of non-empty lines each containing
            a logical expression
        an empty line (with no characters)
```

Input ends with an end of file.

There are no whitespace characters in any input line except perhaps the test case name line.

The logical expressions either have no propositional variables, or have 'p' and/or 'q' as their only propositional variables.

Output

For each test case:

```
        one line copying the name of the test case
        for each input logical expression, one line
            beginning with an exact copy of the logical
            expression and followed by one of:

                = <value of expression>
                is valid
                is satisfiable
                is unsatisfiable

        one empty line
```

If there is no proposition letter in the logical expression, output ' = <value of expression>' after the logical expression, where the value of the expression is a constant with the form of zero or more digits in parentheses; i.e., '(357)'. The digits in the parentheses MUST BE IN ASCENDING ORDER.

If the logical expression contains propositions p or q, then output ' is valid' after the logical expression if the logical expression is TRUE at ALL times for ALL values of p and q. Such a formula is an axiom or theorem of PF Temporal Logic for the given structure of time (T,<).

Otherwise if the logical expression is TRUE at ALL times for SOME p value and SOME q value, then output ‘ is satisfiable’ after the logical expression.

Otherwise output ‘ is unsatisfiable’ after the logical expression. This means that for EVERY value of p and EVERY value of q the expression is FALSE at SOME time.

For example, ‘((Fp)|(˜(F(Fp))))’ is an axiom that is true for all times and values of p if and only if < is a transitive relation on times. So only for transitive < is this formula ‘valid’.

The ONLY whitespaces in any output line are those surrounding ‘=’ and ‘is’, and those copied in the test case name line. The last line output is empty.

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

SAMPLE 1
.XXXXXXXXX
..XXXXXXXX
...XXXXXXX
....XXXXXX
.....XXXXX
......XXXX
.......XXX
........XX
.........X
..........
(˜(23478))
((12345)&(456789))
((123)|(345))
(P(35))
(F(235))
((Fp)|(˜(F(Fp))))
((˜p)&p)
(p&q)

SAMPLE 2
.XXXXXXXX.
..XXXXXXXX
...XXXXXXX
....XXXXXX
.....XXXXX
......XXXX
.......XXX
........XX
.........X
..........
((Fp)|(˜(F(Fp))))


[The last input line is an empty line.]
```

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

SAMPLE 1
(~(23478)) = (01569)
((12345)&(456789)) = (45)
((123)|(345)) = (12345)
(P(35)) = (456789)
(F(235)) = (01234)
((Fp)|(~(F(Fp)))) is valid
((~p)&p) is unsatisfiable
(p&q) is satisfiable

SAMPLE 2
((Fp)|(~(F(Fp)))) is satisfiable


[The last output line is an empty line.]
```


```
File:       temporal.txt
Author:     Bob Walton <walton@deas.harvard.edu>
Date:       Wed Oct 11 10:34:05 EDT 2006
```



```
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: 2006/10/11 14:35:04 $
    $RCSfile: temporal.txt,v $
    $Revision: 1.6 $
```