Problems Index Sun Oct 19 08:53:11 EDT 2003

Problems for BOSPRE 2003.

Easier Problems are First.

problems/puredie
Making dice live up to expectations.

problems/changeview
A computational view of the world.

problems/convoy
Traffic accordions.

problems/monotonic
Solving some equations is easy.

problems/coloring
Turning adult’s work to child’s play.

problems/whichcoin
Best strategy to solve a conundrum.

problems/features
Do we think with labeled graphs?

Pure Dice

Olvk is very concerned by bias on dice, and wants to make an unbiased 6-sided dice. He does this by taking an N sided dice and throwing it 3 times. If the 3 values thrown are different, they can come out in one of 6 orderings, and each ordering has the same probability. By assigning a number from 1 through 6 to each of the 6 orderings, Olvk effectively has an unbiased 6-sided dice.

For example, if S represents the smallest of three different values, M the middle value, and L the largest value, then the 6 orderings might be assigned values by

SML	1
SLM	2
MSL	3
MLS	4
LSM	5
LMS	6

One problem is that the three values may not all be different: if N were 6, one might throw 115 or even 111. In this case Olvk choses to ignore the three values he just threw, and throw 3 more times. If we call a set of 3 throws a ‘round’, Olvk keeps throwing rounds until he gets 3 different values. Olvk must throw M rounds to get an 1 unbiased throw if the first M-1 rounds each have 2 or 3 equal values and the M’th round does not.

You are asked to compute the probability that Olvk will need exactly M rounds to get 1 unbiased value, given N and the probabilities that each of the N faces of the biased dice will be thrown.

Input

For each case, 1 line containing

```
M N p1 p2 ... pN
```

where M and N are as above and p1, p2, ..., pN are the N probabilities that each of the N faces of the biased die will be thrown. 1 <= M <= 100. 4 <= N <= 20. 0.0 <= pI <= 1.0, for I from 1 through N. The sum of the pI’s is 1.0.

Input ends with an end of file.

Output

One line per case containing the probability that it will take Olvk exactly M rounds to get his first unbiased value. This probability must have exactly 6 decimal places.

Example Input

```
1 4 0.2 0.2 0.2 0.4
3 4 0.2 0.2 0.2 0.4
```

Example Output

```
0.336000
0.148141
```

```
File:      puredie.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sat Oct 18 22:07:29 EDT 2003

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: hc3-judge $
    $Date: 2003/10/26 11:54:47 $
    $RCSfile: bospre-2003-problems.ps,v $
    $Revision: 1.2 $
```

Changing Point of View

TeffalHead FatBody has stayed out too late on the planet BadTrash and is in danger of being consumed by a Larger BageGarLectorCol. To get to safety TeffalHead must get to base A or base B or the ZoomTube that connects them. He knows his own position, C, and the ZoomTube is a perfectly straight line between A and B (woe betide a zoomer in a curved ZoomTube). TeffalHead needs to know immediately which he is closest to, A, B, or some point on the ZoomTube between A and B.

TeffalHead knows the xy-coordinates of points A, B, and C. Like any good robotminded soul, he expects to translate and rotate the xy-coordinate system to make a new x'y'-coordinate system in which A has x'y'-coordinates (0,0) and B has x'y'-coordinates (L,0), where L is the distance from A to B. Then the answer can be easily read from the x' coordinate of C.

Unfortunately, living up to his first name, which means 'forgetful in emergencies', TeffalHead has forgotten the program that finds the x'y'-coordinate system. He as put out a call for help, and as the only emergency proggrammer within range, you must send him a program tout de suite.

Note you are permitted to translate and rotate the xy-coordinates, but NOT to reflect across a coordinate axis. Unnecessary reflections are a terrible breech of robot etiquette. Thus the y' coordinate of C is unambiguous.

Input

For each of several cases, one line, containing

```
Ax Ay Bx By Cx Cy
```

where the xy-coordinates of points A, B, and C are respectively (Ax,Ay), (Bx,By), and (Cx,Cy). Input ends with an end of file.

Output

For each case one line containing:

```
(Cx',Cy') L ANS
```

where (Cx',Cy') are the x'y'-coordinates of C, L is the length of AB, and ANS is one of the following:

A	If TeffalHead is closest to A.
B	If TeffalHead is closest to B.
ZoomTube	If TeffalHead is closest to a point on the ZoomTube between A and B.

The x'y'-coordinates and L must be accurate to plus or minus 0.001.

Example Input

```
0 0 1 0 0.5 -6
5.0 3.0 5.5 2.5 5.0 4.0
5.0 3.0 5.5 2.5 5.0 1.0
```

```
Example Output
------- ------

(0.500,-6.000) 1.000 ZoomTube
(-0.707,0.707) 0.707 A
(1.414,-1.414) 0.707 B



File:      changeview.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sat Oct 18 08:24:10 EDT 2003

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: hc3-judge $
    $Date: 2003/10/26 11:54:47 $
    $RCSfile: bospre-2003-problems.ps,v $
    $Revision: 1.2 $
```

Convoy

The Safe and Speedy Driver Company makes robot drivers. They have been asked to provide drivers for convoys of trucks, but are unsure if their basic traveling algorithm will work. You have been asked to simulate it, so see under what circumstances it will cause crashes.

The simulation is of N trucks traveling from right to left. In the beginning, all trucks are separated by exactly L0 feet and are traveling a velocity V0 ft/s. The simulation lasts for some number of seconds.

Each driver decides at the beginning of each second whether to accelerate during the second, decelerate (brake) during the second, or maintain velocity during the second. All acceleration is by A0 ft/s/s. All deceleration is by -A0 ft/s/s unless the current truck velocity V ft/s is less than A0, in which case the deceleration is by -V ft/s/s so the truck velocity will be 0 at the end of the second. Maintaining velocity, of course, involves an acceleration of 0 ft/s/s during the second.

During each second, the acceleration in ft/s/s of each truck is constant, the velocity of the truck is a linear function of time, and the distance traveled by the truck is a quadratic function of time.

The algorithm each driver of a non-lead truck follows to determine the truck's acceleration at the beginning of a second is:

> if the truck is approaching the truck it is following at a relative velocity of at least dV ft/s, then decelerate
>
> else if the truck is receding from the truck it is following at a relative velocity of at least dV ft/s, then accelerate
>
> else if the truck is at least L0+dL ft from the truck it is following, accelerate
>
> else if the truck is at at most L0-dL ft from the truck it is following, decelerate
>
> else maintain speed

The lead truck receives instructions that tell its driver what to do for each second.

Input

For each of several cases,

One line containing the numbers

N L0 dL V0 dV A0

One line containing the instructions for the lead driver.

The instructions are a sequence of +, 0, and - characters, one character per second. Each is interpreted as an instruction for the lead driver to

+ accelerate
0 maintain speed
- decelerate

for one second. The leftmost character is for the first second, the rightmost for the last second, and the number of instruction characters is the number of seconds in the simulation. There are no spaces in the instructions line.

Simulations are limited to at most 100 seconds and at most 11 trucks. Input ends with an end of file.

Output

For each case

One line containing just ‘Case #’, where # is the number of the case, 1, 2, 3, etc, and there is exactly one space in the line.

Lines containing the distances between the non-lead trucks and the truck they are following. Each line contains N-1 distances, each in exactly 8 columns with exactly 3 decimal places. Each distance is the distance between a truck and the truck it is following. The distances are for the trucks from left to right: the first is for the truck after the lead truck.

One line with the initial distances is printed, followed by one line for each second of simulation with the distances at the end of the second.

If any output line has a negative distance, the simulation terminates, and a next line containing just ‘CRASH’ is output.

Example Input
------- -----

```
3 88 11 44 11 44
-----
3 88 11 44 11 44
+++++0000---------------
5 88 11 44 11 44
+-----
```

```
Example Output
------- ------

Case 1
  88.000  88.000
  66.000  88.000
  44.000  66.000
  44.000  44.000
  44.000  44.000
  44.000  44.000
Case 2
  88.000  88.000
 110.000  88.000
 154.000 110.000
 198.000 154.000
 242.000 198.000
 286.000 242.000
 308.000 286.000
 286.000 330.000
 264.000 330.000
 242.000 330.000
 198.000 330.000
 154.000 286.000
 110.000 242.000
  66.000 198.000
  22.000 154.000
 -22.000 110.000
CRASH
Case 3
  88.000  88.000  88.000  88.000
 110.000  88.000  88.000  88.000
 110.000 110.000  88.000  88.000
  66.000 110.000 110.000  88.000
  44.000  66.000 110.000 110.000
  44.000  44.000  66.000 110.000
  44.000  44.000  44.000  66.000
```

```
Postscript
----------

Driving safety experts recommend drivers maintain a
at least 3 seconds separation between themselves and
the car in front of them in good weather.



File:      convoy.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sat Oct 18 08:56:00 EDT 2003

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: hc3-judge $
    $Date: 2003/10/26 11:54:47 $
    $RCSfile: bospre-2003-problems.ps,v $
    $Revision: 1.2 $
```

Solving Monotonic Functions

The Department of Monotonic Functions (DMF) will, among other things, solve equations of the form F(x) = 0, where F is a strictly monotonic function. 'F(x) is strictly monotonic' means that whenever x < y, F(x) < F(y).

You are asked to write a program to perform this task for fairly arbitrary F. F is input in polish notation. That is, F(x) is calculated by a stack machine as follows.

The stack machine has a stack of double precision floating point numbers. Initially the stack is empty.

The function is a sequence of symbols and numbers. A number means: push the number into the top of the stack. All numbers begin with a digit: there are no signed numbers. The following are the possible symbols and their meanings:

x	push x into the stack
+	pop the top 2 members of the stack, and push their sum onto the stack
-	pop the top 2 members of the stack, and push the second value popped minus the first value popped onto the stack
*	pop the top 2 members of the stack, and push their product onto the stack
/	pop the top 2 members of the stack, and push onto the stack the second value popped divided by the first value popped

Thus '2.4 x * 1.2 x / -' represents the monotonic function

$$F(x) = 2.4 * x - 1.2 / x$$

Input:

For each of several cases, one line containing

L H TOKEN ... ;

where [L,H] is the range of x values over which F(x) is monotonic and in which the solution is to be found, and 'TOKEN ...' is the sequence of symbols and numbers representing the function F. A TOKEN is just one symbol or number. L, H, TOKENs, and ; are all separated by spaces or tabs. There are at most 80 TOKENs.

It is guaranteed that F(L) < 0 < F(H) and that no value of x in the range [L,H] will cause the computation of F to overflow double precision floating point arithmetic.

Input ends with an end of file.

Output:

For each case, a single line containing nothing but the value of x such that F(x) = 0. The value of x must be accurate to within 10**-9.

```
Example Input:
------- -----

1E-9 1E+9 2.4 x * 1.2 x / - ;
1 1E+9 3 x x x * * * 6 x x * * - 4 x * + 10 - ;



Example Output:
------- ------

0.707106781
2.114827162


File:      monotonic.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sun Oct 19 08:17:38 EDT 2003

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: hc3-judge $
    $Date: 2003/10/26 11:54:47 $
    $RCSfile: bospre-2003-problems.ps,v $
    $Revision: 1.2 $
```

Graph Coloring
----- --------

The classical graph coloring problem is just the following: color the nodes of an undirected graph so that neighbors do not have the same color, and a minimum number of colors are used. This problem has many applications, among which are allocating variables to registers by a compiler. The compiler considers two variables to be neighbors if they are needed in the same block of code.

The graph coloring problem is very hard to solve (its NP complete). But there is a problem that is easy to solve that can lead to good enough, but not optimal, graph coloring solutions. It is this: make a list of the nodes of the graph so that each node has no more than M-1 neighbors after it in the list. Then the graph can be colored with M colors, which can be assigned from the end of the list to the beginning. For each node, at most M-1 neighbors have been assigned colors before the node is, so with M colors the node can always be assigned a color different from any of its neighbors.

You are asked to find the minimum M such that the list can be made.

Input

For each of several cases:

A line containing the number N of nodes.
1 <= N <= 80.

N lines each containing N binary digits
('0's and '1's).

Nodes are identified by integers i, 1 <= i <= N. Lines of digits are numbered 1, 2, 3, from the first line to the last line. Digits in a line are numbered 1, 2, 3, from left to right.

For 1 <= i,j <= N, digit j of line i is '1' if node i is a neighbor of node j, and '0' otherwise. Digit j of line i equals digit i of line j, and digit i of line i is '0' (a node is NOT a neighbor of itself).

No lines contain any spaces. The input terminates with an end of file.

Output

For each case, the single line containing M, the smallest integer for which the nodes of the graph can be put in a list such that at most M-1 neighbors of any node appear after the node in the list.

```
Example Input
------- -----

4
0111
1011
1101
1110
4
0101
1010
0101
1010
5
01111
10000
10000
10000
10000



Example Output
------- ------

4
3
2
```

```
File:      coloring.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sat Oct 18 08:27:15 EDT 2003

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: hc3-judge $
    $Date: 2003/10/26 11:54:47 $
    $RCSfile: bospre-2003-problems.ps,v $
    $Revision: 1.2 $
```

Which Coin is False
----- ---- -- -----

Consider the following problem. You are given N coins, exactly 1 of which is false, in that its weight differs from the others, though you do not know whether the false coin is lighter or heavier than it should be. You are also given 1 additional coin known to be true and a scale. You are asked to make a series of weighings of equal numbers of the coins and at the end tell which coin is false. You are asked to minimize the number of weighings required.

Professor Toowit Toowoo (TT to his friends) has come up with the following strategy for solving this problem. There are two cases: first, where you do not know whether the false coin is lighter than or heavier than a true coin, and second where you do know.

If you do know whether the false coin is lighter than or heavier than a true coin, divide the N coins into three almost equal groups, and weigh two of these that have equal numbers of coins against each other. The result will tell you which of the three groups the false coin is in.

If on the other hand you do NOT know whether the false coin is lighter than or heavier than a true coin, divide the N coins into three groups two of which have the same number J of coins, and weigh these two groups against each other. If the scale balances, the false coin is in the group not weighed.

Otherwise, move some number K of the coins originally on the left scale to the right scale, replacing them by coins not originally weighed, and remove K coins originally on the right scale, so they are not weighed. This second weighing has three outcomes. If the scales now balance, the false coin was removed from the right scale. If the balance (which side is heavier) changes, the false coin was moved from the left to the right scale. If the balance remains the same, the false coin was not moved from either scale. In the first two cases, you now know whether the false coin is lighter than or heavier than a true coin. In the last case, you still do not know.

If N is 2, then you can use the 1 known true coin to solve the problem with a single weighing, as a special case.

Notice that the number of weighings necessary may depend upon the actual results of the weighings. For example, if N=5, the first step might be to weigh 2 coins against 2 coins, and in the best case the scale balances and the false coin is known immediately to be the coin you did not weigh. But in the worst case, where the scale is unbalanced, you have to move 1 coin from one scale to the other, and if the scale remains unbalanced, then you have only narrowed it down to 2 coins and need a third weighing.

We want to use TT’s strategy to minimize the number of weighings required if every result turns out to be worst case. Thus you are asked to find the minimum worst case number of weighings needed to solve the problem using TT’s strategy, for a given N, and to find the number J to be used in the first weighing, and the number K to be used in the second weighing should the first weighing not balance.

Input

One line for each test case. This line just contains N, with 3 <= N <= 500. The input ends with an end of file.

Output

One line for each case. This line contains

W J K

where W is the minimum worst case number of weighings required using TT's strategy to solve the problem for N coins, when you do NOT know whether the false coin is lighter or heavier than a true coin; J is the number of coins on each scale in the first weighing; and K is the number of coins moved in the second weighing if the first weighing has an imbalance. If several values of J give the same W, output only the minimum such J. If for this J several values of K give the same W, output only the minimum such K.

Example Input
------- -----

```
3
4
5
6
7
8
0
```

Example Output
------- ------

```
2 1 1
2 1 1
3 1 1
3 1 1
3 2 1
3 2 1
```

Note: This problem is derived from a problem stated in 'Engaging Students with Theory', by Shilov and Yi, Communications of the ACM, Sept 2002, Vol. 45 No. 9, p 98.

File: whichcoin.txt
Author: Bob Walton <walton@deas.harvard.edu>
Date: Sun Oct 19 08:27:39 EDT 2003

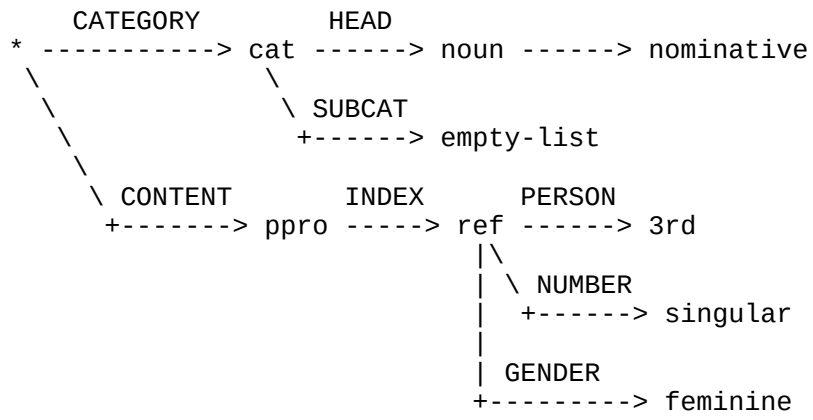
RCS Info (may not be true date or author):

```
$Author: hc3-judge $
$Date: 2003/10/26 11:54:47 $
$RCSfile: bospre-2003-problems.ps,v $
$Revision: 1.2 $
```

Feature Structures
------- ----------

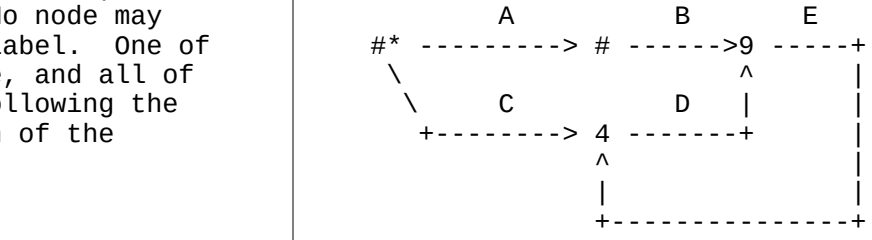
Information can be encoded in ‘feature structures’, which are rooted labeled graphs. To explain, a feature structure is a set of graph nodes (points), a set of directed edges between pairs of nodes (arrows), and an assignment of labels (symbols of some kind) to SOME of the nodes and ALL of the arrows. No node may be the source of two arrows with the same label. One of the nodes is distinguished as the root node, and all of the nodes are reachable from the root by following the directed edges (going only in the direction of the arrows).

An example feature structure is



This is part of the dictionary entry for the word ‘she’ for a typical computer English parser. Feature structures seem to be the most natural way to describe linguistic information and grammar rules, and because of this, it is conceivable that the human mind uses some analog of feature structures to parse sentences.

From now on we will use single upper case characters as arrow labels, and single decimal digits as node labels. Unlabeled nodes will be represented by ‘#’, and the root node will be marked by ‘*’. With this in mind, consider the feature structure



A path from the root to a node N in feature structure is a list of edges starting at the root and going to N, where the target of one edge is the source of the next edge. A path can be named by giving the edge labels in sequence, which we do separated by dots, with an initial dot added to make it easy to distinguish the empty path that names the root. A path is said to name the target of its last edge. Thus in the above example, ‘.’ names the root node, ‘.A’ names the other unlabeled node, and ‘.A.B’ and ‘.C.D’ name the node with value ‘9’. Actually this node has an infinite number of names, including ‘.A.B.E.D’ and ‘.C.D.E.D.E.D’.

A feature structure can be described by a set of equations. Let P, P1, P2 be a path names, and V be a node value. Then the equations have one of three forms:

```
P
P:V
P1=P2
```

The equation P means a node named P exists. The equation P:V means a node named P exists and is labeled with the value V (we will say the node 'has value V'). The equation P1=P2 means there exists a node that has both the name P1 and the name P2.

Our example feature structure can be described by the equations

```
.A.B:9
.C:4
.C.D=.A.B
.C.D.E=.C
```

These are not the only equations true for the this feature structure, but the example feature structure is the smallest feature structure that satisfies these equations, in a sense we will now make precise.

As feature structures store information, we might expect them to have a notion of one feature structure having more information than another. If F1 and F2 are feature structures, we say 'F1 <= F2' (here <= means 'is greater than', or 'has more information than') if and only if every equation true of F1 is also true of F2. Thus F1 is more general than F2, and F2 is more specific than F1. The technical term for this is that 'F1 subsumes F2', where 'subsumes' just means 'is more general than'.

What does 'F1 <= F2' mean in terms of rooted directed labeled graphs. It means (1) for every node in F1 named by some path P, there is a node in F2 with name P; (2) for every node in F1 named by some path P that has value V, the node in F2 named by P also has value V; and (3) if some node in F1 has both names P1 and P2, then the node named P1 in F2 also has name P2. As a consequence of all this there is a map of nodes of F1 to nodes of F2 such that (1) a node named P in F1 is mapped to a node named P in F2, and (2) if a node in F1 has value V, the node it is mapped to in F2 also has value V.

The property of feature structures that makes them extremely useful in computing is that certain computations of feature structures have a minimum answer: that is, a feature structure can be computed that is smaller than any other suitable feature structure.

One instance of this is the following: given a set of equations, either there is no feature structure satisfying all the equations, or there is exactly one minimum feature structure satisfying all the equations. If there is no feature structure satisfying the equations, then the equations are said to be incompatible. For example, the equations

```
.A:1
.A.B=.A
.A.B:2
```

are incompatible, because they imply there is a node with both the names .A and .A.B, which is fine, but also this node has both values 1 and 2, which is not allowed: no node may have more than one label.

Two feature structures F1 and F2 are said to be compatible if and only if there is some feature structure F such that F1 <= F and F2 <= F. Then there is a minimum F, which is just the minimum F satisfying all the equations of F1 and all the equations of F2. This is very useful, because if a computer knows that F1 and F2 are true, it does not have to keep both F1 and F2 around: instead it can compute the minimum F and keep that for future computations. This greatly improves the efficiency of the computation, and is the reason that feature structures are a good way for a computer to represent information.

To compute the minimum feature structure satisfying a set of equations, proceed to add one equation at a time. Start with the feature structure F equal to '*#', which is just an unlabeled root node. Given a new equation P, just add nodes and arrows as necessary to F until a node named P exists. Any nodes added have no value. Given a new equation P:V, make a node named P if none exists, and then give that node the value V. If a node named P already exists and already has a value DIFFERENT from V, the equations are incompatible. Given a new equation P1=P2, make nodes named P1 and P2 if necessary, and then merge them to make a single node. Merging nodes means gluing them together so that they are the same node. If you must merge nodes that have DIFFERENT values, the equations are incompatible. Computationally N1 and N2 can be merged by storing in N1 a forwarding pointer to N2; this forwarding pointer behaves like a forwarding address in a mail system.

When you merge nodes N1 and N2, you keep all the arrows from BOTH nodes, BUT if

```
          X             X
    if N1--->N1' and N2--->N2'
```

you must merge N1' and N2'.

That is, if two nodes being merged are BOTH sources for an arrow labeled X, you keep just one of the arrows but you merge the destinations of the arrows. So merging is a recursive operation.

In this problem you are given sets of equations and are asked to compute for each set the minimum feature structure described and answer questions about it.

Input

For each of several cases,

- A line containing only 'EQUATIONS'.
- Zero or more lines each containing an equation. All these equations together describe a minimum feature structure.
- A line containing only 'QUESTIONS'.
- Zero or more lines each containing an equation to be tested against the minimum feature structure.
- A line containing only 'DONE'.

There are no spaces in any input line. The equations are as described above. No equation is more than 80 characters long. Input ends with an end of file.

```
Output
------

For each case,

    A line containing 'Case #' where # is 1, 2, 3, ...,
    the case number.

    If the EQUATIONS are incompatible, a single line
    containing 'INCOMPATIBLE'.

    Otherwise, for each QUESTION in order, a single line
    containing either 'TRUE' or 'FALSE', that tells
    whether the QUESTION is true of the minimal feature
    structure that satisfies the EQUATIONS.
```

```
Example Input
------- -----

EQUATIONS
.A
.B
.B:6
QUESTIONS
.A
.B
.C
.A:3
.B:6
DONE
EQUATIONS
.A.B:9
.C:4
.C.D=.A.B
.C.D.E=.C
QUESTIONS
.A:3
.A
.C.D:8
.C.D.E.D:9
.A.B.E.D=.C
.A.B.E=.C
.A.B.E:4
.A.B.E:9
.A.E
DONE
EQUATIONS
.A:1
.A.B:2
.A.B=.A
QUESTIONS
DONE
```

```
Example Output
------- ------

Case 1:
TRUE
TRUE
FALSE
FALSE
TRUE
Case 2:
FALSE
TRUE
FALSE
TRUE
FALSE
TRUE
TRUE
FALSE
FALSE
Case 3:
INCOMPATIBLE



Reference
---------

Bob Carpenter, The Logic of Typed Feature Structures,
Cambridge University Press, 1992.
```

```
File:      features.txt
Author:    Bob Walton <walton@deas.harvard.edu>
Date:      Sun Oct 19 08:16:33 EDT 2003

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: hc3-judge $
    $Date: 2003/10/26 11:54:47 $
    $RCSfile: bospre-2003-problems.ps,v $
    $Revision: 1.2 $
```