Problem A: Mutant Flatworld Explorers (novice only)

Background

Robotics, robot motion planning, and machine learning are areas that cross the boundaries of many of the subdisciplines that comprise Computer Science: artificial intelligence, algorithms and complexity, electrical and mechanical engineering to name a few. In addition, robots as "turtles" (inspired by work by Papert, Abelson, and diSessa) and as "beeper-pickers" (inspired by work by Pattis) have been studied and used by students as an introduction to programming for many years.

This problem involves determining the position of a robot exploring a pre-Columbian flat world.

The Problem

Given the dimensions of a rectangular grid and a sequence of robot positions and instructions, you are to write a program that determines for each sequence of robot positions and instructions the final position of the robot.

A robot *position* consists of a grid coordinate (a pair of integers: x-coordinate followed by y-coordinate) and an orientation (N,S,E,W for north, south, east, and west). A robot *instruction* is a string of the letters '*L*', '*R*', and '*F*' which represent, respectively, the instructions:

- *Left*: the robot turns left 90 degrees and remains on the current grid point.
- *Right*: the robot turns right 90 degrees and remains on the current grid point.
- *Forward*: the robot moves forward one grid point in the direction of the current orientation and mantains the same orientation.

The direction *North* corresponds to the direction from grid point (x, y) to grid point $(x, y+1)$.

Since the grid is rectangular and bounded, a robot that moves "off" an edge of the grid is lost forever. However, lost robots leave a robot "scent" that prohibits future robots from dropping off the world at the same grid point. The scent is left at the last grid position the robot occupied before disappearing over the edge. An instruction to move "off" the world from a grid point from which a robot has been previously lost is simply ignored by the current robot.

The Input

The first line of input is the upper-right coordinates of the rectangular world, the lower-left coordinates are assumed to be 0,0.

The remaining input consists of a sequence of robot positions and instructions (two lines per robot). A position consists of two integers specifying the initial coordinates of the robot and an orientation (N,S,E,W), all separated by white space on one line. A robot instruction is a string of the letters 'L', 'R', and 'F' on one line.

Each robot is processed sequentially, i.e., finishes executing the robot instructions before the next robot begins execution.

Input is terminated by end-of-file.

You may assume that all initial robot positions are within the bounds of the specified grid. The maximum value for any coordinate is 50. All instruction strings will be less than 100 characters in length.

The Output

For each robot position/instruction in the input, the output should indicate the final grid position and orientation of the robot. If a robot falls off the edge of the grid the word "LOST" should be printed after the position and orientation. One space should appear between data items on a line.

Sample Input

```
5 3
1 1 E
RFRFRFRF
3 2 N
FRRFLLFFRRFLL
0 3 W
LLFFFLFLFL
```

Sample Output

```
1 1 E
3 3 N LOST
2 3 S
```

Problem B: Greedy Gift Givers (novice only)

Background

The Problem

This problem involves determining, for a group of gift-giving friends, how much more each person gives than they receive (and vice versa for those that view gift-giving with cynicism).

In this problem each person sets aside some money for gift-giving and divides this money evenly among all those to whom gifts are given.

However, in any group of friends, some people are more giving than others (or at least may have more acquaintances) and some people have more money than others.

Given a group of friends, the money each person in the group spends on gifts, and a (sub)list of friends to whom each person gives gifts; you are to write a program that determines how much more (or less) each person in the group gives than they receive.

The Input

The input is a sequence of gift-giving groups. A group consists of several lines:

- the number of people in the group,
- a list of the names of each person in the group,
- a line for each person in the group consisting of the name of the person, the amount of money spent on gifts, the number of people to whom gifts are given, and the names of those to whom gifts are given.

All names are lower-case letters, there are no more than 10 people in a group, and no name is more than 12 characters in length. Money is a non-negative integer less than or equal to 2000.

The input consists of one or more groups and is terminated by end-of-file.

The Output

For each group of gift-givers, the name of each person in the group should be printed on a line followed by a space followed by the net gain (or loss) received (or spent) by the person. Names in a group should be printed in the same order in which they first appear in the input.

The output for each group should be separated from other groups by a blank line. All gifts are integers. Each person gives the same integer amount of money to each friend to whom any money is given, and gives as much as possible. Any money not given is kept and is part of a person's "net worth" printed in the output.

Sample Input

```
5
dave laura owen vick amr
dave 200 3 laura owen vick
owen 500 1 dave
amr 150 2 vick owen
laura 0 2 amr vick
vick 0 0
3
liz steve dave
liz 30 1 steve
steve 55 2 liz dave
dave 0 2 steve liz
```

Sample Output

```
dave 302
laura 66
owen -359
vick 141
amr -150

liz -3
steve -24
dave 27
```

Problem C: Stacks of Flapjacks

Background

Stacks and Queues are often considered the bread and butter of data structures and find use in architecture, parsing, operating systems, and discrete event simulation. Stacks are also important in the theory of formal languages.

This problem involves both butter and sustenance in the form of pancakes rather than bread in addition to a finicky server who flips pancakes according to a unique, but complete set of rules.

The Problem

Given a stack of pancakes, you are to write a program that indicates how the stack can be sorted so that the largest pancake is on the bottom and the smallest pancake is on the top. The size of a pancake is given by the pancake's diameter. All pancakes in a stack have different diameters.

Sorting a stack is done by a sequence of pancake "flips". A flip consists of inserting a spatula between two pancakes in a stack and flipping (reversing) <u>all</u> the pancakes on the spatula (reversing the sub-stack). A flip is specified by giving the position of the pancake on the bottom of the sub-stack to be flipped (relative to the whole stack). The pancake on the bottom of the whole stack has position 1 and the pancake on the top of a stack of n pancakes has position n.

A stack is specified by giving the diameter of each pancake in the stack in the order in which the pancakes appear.

For example, consider the three stacks of pancakes below (in which pancake 8 is the top-most pancake of the left stack):

```
8        7        2
4        6        5
6        4        8
7        8        4
5        5        6
2        2        7
```

The stack on the left can be transformed to the stack in the middle via *flip(3)*. The middle stack can be transformed into the right stack via the command *flip(1)*.

The Input

The input consists of a sequence of stacks of pancakes. Each stack will consist of between 1 and 30 pancakes and each pancake will have an integer diameter between 1 and 100. The input is terminated by end-of-file. Each stack is given as a single line of input with the top pancake on a stack appearing first on a line, the bottom pancake appearing last, and all pancakes separated by a space.

The Output

For each stack of pancakes, the output should list on one line the sequence of flips that results in the stack of pancakes being sorted so that the largest diameter pancake is on the bottom and the smallest on top. For each stack the sequence of flips should be terminated by a 0 (indicating no more flips necessary). Once a stack is sorted, no more flips should be made.

Sample Input

```
1 2 3 4 5
5 4 3 2 1
5 1 2 3 4
```

Sample Output

```
0
1 0
1 2 0
```

Problem D: Pipe Fitters

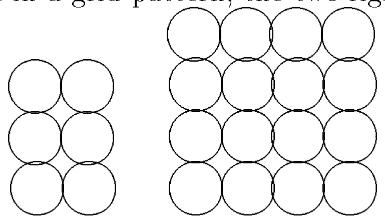
Background

Filters, or programs that pass "processed" data through in some changed form, are an important class of programs in the UNIX operating system. A pipe is an operating system concept that permits data to "flow" between processes (and allows filters to be chained together easily.)

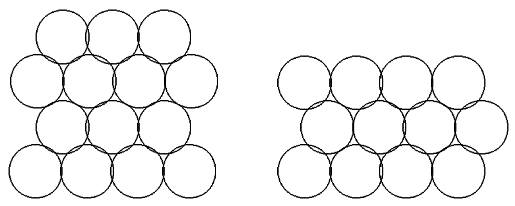
This problem involves maximizing the number of pipes that can be fit into a storage container (but it's a pipe fitting problem, not a bin packing problem).

The Problem

A company manufactures pipes of uniform diameter. All pipes are stored in rectangular storage containers, but the containers come in several different sizes. Pipes are stored in rows within a container so that there is no space between pipes in any row (there may be some space at the end of a row), i.e., all pipes in a row are tangent, or touch. Within a rectangular cross-section, pipes are stored in either a *grid* pattern or a *skew* pattern as shown below: the two left-most cross-sections are in a grid pattern, the two right-most cross-sections are in a skew pattern.



grid patterns



skew patterns

Note that although it may not be apparent from the diagram, there is no space between adjacent pipes in any row. The pipes in any row are tangent to (touch) the pipes in the row below (or rest on the bottom of the container). When pipes are packed into a container, there may be "left-over" space in which a pipe cannot be packed. Such left-over space is packed with padding so that the pipes cannot settle during shipping.

The Input

The input is a sequence of cross-section dimensions of storage containers. Each cross-section is given as two real values on one line separated by white space. The dimensions are expressed in units of pipe diameters. All dimensions will be less than 2^7. Note that a cross section with dimensions $a \times b$ can also be viewed as a cross section with dimensions $b \times a$.

The Output

For each cross-section in the input, your program should print the maximum number of pipes that can be packed into that cross section. The number of pipes is an integer — no fractional pipes can be packed. The maximum number is followed by a space and then the word "grid" if a grid pattern results in the maximal number of pipes or the word "skew" if a skew pattern results in the maximal number of pipes. If the pattern doesn't matter, that is the same number of pipes can be packed with either a grid or skew pattern, then the word "grid" should be printed.

Sample Input

```
3 3
2.9 10
2.9 10.5
11 11
```

Sample Output

```
9 grid
29 skew
30 skew
126 skew
```

Problem E: Trees on the level

Background

Trees are fundamental in many branches of computer science[1]. Current state-of-the art parallel computers such as Thinking Machines' CM-5 are based on *fat trees.* Quad- and octal-trees are fundamental to many algorithms in computer graphics.

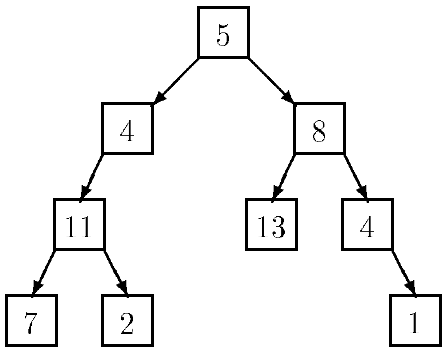
This problem involves building and traversing binary trees.

The Problem

Given a sequence of binary trees, you are to write a program that prints a level-order traversal of each tree. In this problem each node of a binary tree contains a positive integer and all binary trees have have fewer than 256 nodes.

In a *level-order* traversal of a tree, the data in all nodes at a given level are printed in left-to-right order and all nodes at level k are printed before all nodes at level $k + 1$.

For example, a level order traversal of the tree



is: 5, 4, 8, 11, 13, 4, 7, 2, 1.

In this problem a binary tree is specified by a sequence of pairs (n,s) where n is the value at the node whose path from the root is given by the string s. A path is given be a sequence of L's and R's where L indicates a left branch and R indicates a right branch. In the tree diagrammed above, the node containing 13 is specified by (13,RL), and the node containing 2 is specified by (2,LLR). The root node is specified by (5,) where the empty string indicates the path from the root to itself. A binary tree is considered to be *completely specified* if every node on all root-to-node paths in the tree is given a value exactly once.

The Input

The input is a sequence of binary trees specified as described above. Each tree in a sequence consists of several pairs (n, s) as described above separated by whitespace. The last entry in each tree is (). No whitespace appears between left and right parentheses.

All nodes contain a positive integer. Every tree in the input will consist of at least one node and no more than 256 nodes. Input is terminated by end-of-file.

[1]Pun definitely intended.

The Output

For each completely specified binary tree in the input file, the level order traversal of that tree should be printed on a single line. One space should appear between data items on a line. If a tree is not completely specified, i.e., some node in the tree is NOT given a value or a node is given a value more than once, then the string "not complete" should be printed.

Sample Input

```
(11,LL) (7,LLL) (8,R)
(5,) (4,L) (13,RL) (2,LLR) (1,RRR) (4,RR) ()
(3,L) (4,R) ()
```

Sample Output

```
5 4 8 11 13 4 7 2 1
not complete
```

Problem F: Searching Quickly

Background

Searching and sorting are part of the theory and practice of computer science. For example, binary search provides a good example of an easy-to-understand algorithm with sub-linear complexity. Quicksort is an efficient $O(n \log n)$ [average case] comparison based sort.

KWIC-indexing is an indexing method that permits efficient "human search" of, for example, a list of titles.

The Problem

Given a list of titles and a list of "words to ignore", you are to write a program that generates a KWIC (Key Word In Context) index of the titles. In a KWIC-index, a title is listed once for each keyword that occurs in the title. The KWIC-index is alphabetized by keyword.

Any word that is not one of the "words to ignore" is a potential keyword.

For example, if words to ignore are "`the, of, and, as, a`" and the list of titles is:

```
Descent of Man
The Ascent of Man
The Old Man and The Sea
A Portrait of The Artist As a Young Man
```

A KWIC-index of these titles might be given by:

```
                      a portrait of the ARTIST as a young man
                                    the ASCENT of man
                                        DESCENT of man
                             descent of MAN
                          the ascent of MAN
                                the old MAN and the sea
    a portrait of the artist as a young MAN
                                    the OLD man and the sea
                                      a PORTRAIT of the artist as a young man
                    the old man and the SEA
          a portrait of the artist as a YOUNG man
```

The Input

The input is a sequence of lines, the string `::` is used to separate the list of words to ignore from the list of titles. Each of the words to ignore appears in lower-case letters on a line by itself and is no more than 10 characters in length. Each title appears on a line by itself and may consist of mixed-case (upper and lower) letters. Words in a title are separated by whitespace. No title contains more than 15 words.

There will be no more than 50 words to ignore, no more than than 200 titles, and no more than 10,000 characters in the titles and words to ignore combined. No characters other than 'a'–'z', 'A'–'Z', and white space will appear in the input.

The Output

The output should be a KWIC-index of the titles, with each title appearing once for each keyword in the title, and with the KWIC-index alphabetized by keyword. If a word appears more than once in a title, each instance is a potential keyword.

The keyword should appear in all upper-case letters. All other words in a title should be in lower-case letters. Titles in the KWIC-index with the same keyword should appear in the same order as they appeared in the input file. In the case where multiple instances of a word are keywords in the same title, the keywords should be capitalized in left-to-right order.

Case (upper or lower) is irrelevant when determining if a word is to be ignored.

All titles must be listed left-justified, with one space between words in a title.

Sample Input

```
is
the
of
and
as
a
but
::
Descent of Man
The Ascent of Man
The Old Man and The Sea
A Portrait of The Artist As a Young Man
A Man is a Man but Bubblesort IS A DOG
```

Sample Output

```
a portrait of the ARTIST as a young man
the ASCENT of man
a man is a man but BUBBLESORT is a dog
DESCENT of man
a man is a man but bubblesort is a DOG
descent of MAN
the ascent of MAN
the old MAN and the sea
a portrait of the artist as a young MAN
a MAN is a man but bubblesort is a dog
a man is a MAN but bubblesort is a dog
the OLD man and the sea
a PORTRAIT of the artist as a young man
the old man and the SEA
a portrait of the artist as a YOUNG man
```

Problem G: Following Orders (experts only)

Background

Order is an important concept in mathematics and in computer science. For example, Zorn's Lemma states: "a partially ordered set in which every chain has an upper bound contains a maximal element." Order is also important in reasoning about the fix-point semantics of programs.

This problem involves neither Zorn's Lemma nor fix-point semantics, but does involve order.

The Problem

Given a list of variable constraints of the form `x < y`, you are to write a program that prints all orderings of the variables that are consistent with the constraints.

For example, given the constraints `x < y` and `x < z` there are two orderings of the variables x, y, and z that are consistent with these constraints: `x y z` and `x z y`.

The Input

The input consists of a sequence of constraint specifications. A specification consists of two lines: a list of variables on one line followed by a list of contraints on the next line. A constraint is given by a pair of variables, where `x y` indicates that `x < y`.

All variables are single character, lower-case letters. There will be at least two variables, and no more than 20 variables in a specification. There will be at least one constraint, and no more than 50 constraints in a specification. There will be at least one, and no more than 300 orderings consistent with the contraints in a specification.

Input is terminated by end-of-file.

The Output

For each constraint specification, all orderings consistent with the constraints should be printed. Orderings are printed in lexicographical (alphabetical) order, one per line.

Output for different constraint specifications is separated by a blank line.

Sample Input

```
a b f g
a b b f
v w x y z
v y x v z v w v
```

Sample Output

```
abfg
abgf
agbf
gabf

wxzvy
wzxvy
xwzvy
xzwvy
zwxvy
zxwvy
```

Problem H: Numbering Paths (experts only)

Background

Problems that process input and generate a simple "yes" or "no" answer are called decision problems. One class of decision problems, the NP-complete problems, are not amenable to general efficient solutions. Other problems may be simple as decision problems, but enumerating all possible "yes" answers may be very difficult (or at least time-consuming).

This problem involves determining the number of routes available to an emergency vehicle operating in a city of one-way streets.

The Problem

Given the intersections connected by one-way streets in a city, you are to write a program that determines the number of different routes between each intersection. A route is a sequence of one-way streets connecting two intersections.

Intersections are identified by non-negative integers. A one-way street is specified by a pair of intersections. For example, j k indicates that there is a one-way street from intersection j to intersection k. Note that two-way streets can be modeled by specifying two one-way streets: j k and k j.

Consider a city of four intersections connected by the following one-way streets:

```
0 1
0 2
1 2
2 3
```

There is one route from intersection 0 to 1, two routes from 0 to 2 (the routes are $0 \rightarrow 1 \rightarrow 2$ and $0 \rightarrow 2$), one route from 2 to 3, and no other routes.

It is possible for an infinite number of different routes to exist. For example if the intersections above are augmented by the street 3 2, there is still only one route from 0 to 1, but there are infinitely many different routes from 0 to 2. This is because the street from 2 to 3 and back to 2 can be repeated yielding a different sequence of streets and hence a different route. Thus the route $0 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 2$ is a different route than $0 \rightarrow 2 \rightarrow 3 \rightarrow 2$.

The Input

The input is a sequence of city specifications. Each specification begins with the number of one-way streets in the city followed by that many one-way streets given as pairs of intersections. Each pair j k represents a one-way street from intersection j to intersection k. In all cities, intersections are numbered sequentially from 0 to the "largest" intersection. All integers in the input are separated by whitespace. The input is terminated by end-of-file.

There will never be a one-way street from an intersection to itself. No city will have more than 30 intersections.

The Output

For each city specification, a square matrix of the number of different routes from intersection j to intersection k is printed. If the matrix is denoted M, then $M[j][k]$ is the number of different routes from intersection j to intersection k. The matrix M should be printed in row-major order, one row per line. Each matrix should be preceded by the string "`matrix for city` *k*" (with *k* appropriately instantiated, beginning with 0).

If there are an infinite number of different paths between two intersections a -1 should be printed. All entries in a row should be separated by s single space.

Sample Input

```
7 0 1 0 2 0 4 2 4 2 3 3 1 4 3
5
0 2
0 1 1 5 2 5 2 1
9
0 1 0 2 0 3
0 4 1 4 2 1
2 0
3 0
3 1
```

Sample Output

```
matrix for city 0
0 4 1 3 2
0 0 0 0 0
0 2 0 2 1
0 1 0 0 0
0 1 0 1 0
matrix for city 1
0 2 1 0 0 3
0 0 0 0 0 1
0 1 0 0 0 2
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
matrix for city 2
-1 -1 -1 -1 -1
0 0 0 0 1
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
0 0 0 0 0
```